Kontinuerligt Kollisionsdetektering

Jack Nørskov Jørgensen*
14-06-82

3. juli 2006

Resumé

Vi præsenterer i denne rapport en metode til at foretage kontinuerligt kollisionsdetektering. Vi behandler kort de forskellige metoder som er blevet publiceret tidligere, men som beskrevet i vores synopsis vil denne rapport benytte metoden beskrevet i [10].

Vi benytter imellem-bevægelser og interval aritmetik på OBB-træer i OpenTissue [8] og opnår en stabil løsning til kontinuerligt kollisionsdetektering. Metoderne vi bruger vil alle blive gennemgået i denne rapport.

Til sidst forsøger vi at optimere vores metode.

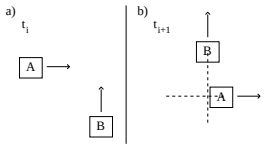
*jackj@diku.dk

Indhold

1 Problemformulering

1.1 Introduktion

I traditionel diskret kollisionsdetektering er det ikke altid muligt at afgøre om to objekter rammer hinanden. Antag f.eks. at vi er givet to objekter A og B der rammer hinanden som vist i Figur 1. Objekterne A og B i billede a bevæger



Figur 1: A og B har passeret hinanden ved tid t_{i+1}.

sig hurtigt nok til at de i billede b har passeret hinanden. Testes der kun for kollision ved tid t_i og t_{i+1} fanges kollision ikke.

Det er muligt at fange en kollision som den i Figur 1 ved at teste i mindre tidsintervaller, men der er flere ulemper ved dette;

- Det er ikke altid muligt at vide hvor f.eks. A befinder sig til et vilkårligt tidspunkt. Er A f.eks. styret af en bruger ved hjælp af en mus opdateres A's position diskret.
- A's position kan være beregningsmæssige tung at udregne. Et eksempel er at A er et objekt i en fysik-simulator hvor mange forskellige elementer kan have en indvirkning på A og dennes position. Her er det derfor ønskværdigt ikke at skulle udregne A's position alt for mange gange.
- Der skal foretages flere test for kollision når der opdeles i flere tidsintervaller, altså skal der bruges mere tid på teste for kollisioner.

De sidste to punkter er selvfølgelig kun et problem hvis der stilles krav til køretiden af det pågældende program. Denne rapport omhandler **kontinuerligt kollisionsdetektering** (der f.eks. ville opdage kollisionen i Figur 1). I første omgang vil vi undersøge metoden beskrevet i [10].

1.2 Afgrænsninger

- Vores metode skal ikke kunne garantere at den finder alle kollisioner.

  Man kan kun garantere dette hvis det er muligt at vide hvor et objekt befinder sig på alle tænkelige tidspunkter. Vi ønsker at vores algoritme også skal kunne bruges i situationer hvor dette ikke er tilfældet.

- Vores metode skal ikke kunne finde kollisioner imellem to elementer hvor det ene element er fuldstændigt omslugt af det andet og hvor der ingen

kontakt er mellem objekterne. Dette er en rimelig begrænsning idet objekterne jo ikke rører hinanden og derfor ikke kan siges at kollidere. Bemærk at hvis objekterne til at starte med er adskilt hvorefter det ene sidenhen bevæger sig ind i det andet, så vil kollisionen kunne blive fundet af vores metode.

1.3 Begrundelse

Kollisionsdetektering benyttes i dag i så godt som alle programmer der skal efterligne elementer i vores verden, eksempler er computerspil, flysimulatorer og virtual reality.

Afhængigt af programmet kan både præcisionen (f.eks. til fysik-simulator) og hastigheden (f.eks. til computerspil) af den benyttede metode til test af kollision være vigtige faktorer. Det er derfor et vigtigt område at udforske for at finde nye forbedrede detekteringsmetoder, enten tids- eller præcisionsmæssigt.

1.4 Fremgangsmåde

Vores fremgangsmåde er som følger

1. Undersøg hvilke løsninger der er beskrevet i litteraturen.
2. Beskriv overordnet metoden i [10].
3. Beskriv de forskellige værktøjer der bruges i metoden.
4. Implementer metoden i OpenTissue.
5. Optimer metoden.
6. Afprøv metoden.

2 Tidligere arbejde

Den valgte metode vi skal beskrive i denne rapport er beskrevet i [10]. Artiklen er relativ ny og er, synes vi, lidt mangelfuld på nogle områder. To artikler er blevet udgivet som benytter samme teknik som i [10], disse er [9] og [11]. I [9] beskrives metoden for stive objekter. En mængde optimeringer beskrives, blandt andet hvordan man under gennemløbning af træer af omringende volumener kan optimere radikalt, mere om dette i Kapitel 9. I [11] beskrives metoden for leddelte objekter. Artiklen beskriver hvordan man ved hjælp af blandt andet "sweeping-volume culling", "CULLIDE" og "Dynamic BVH Culling" kan fjerne mange kollisionstjek. Den benyttede test for kollision imellem geometri er en ny metode kaldet "Improved Newton Interval Method", metoden beskrives kort i Kapitel 9.

Det har ikke været nemt at finde materiale der omhandler metoden i [10]. Faktisk har vi kun fundet de tre artikler ovenfor.

Eftersom vi hele tiden har vidst hvilken metode vi har ønsket at bruge har vi ikke brugt meget tid på at finde andre mulige metoder. Vi har vha. af de tre artikler ovenfor og deres gennemgang af tidligere arbejde kunne finde nogle artikler der omhandler kontinuert kollisionsdetektering. De andre metoder for kontinuert kollisionsdetektering vi har kunnet finde på denne måde har været radikalt anderledes end vores. De har derfor været af begrænset hjælp.

3 Den valgte metode

Vi beskriver her overordnet hvordan metoden [10] virker. I de næste kapitler gennemgås de forskellige begreber introduceret her mere dybdegående.

- I vores udgangspunkt, det diskrete tilfælde, er et objekts position ikke kendt til alle tænkelige tider. For at kunne foretage kontinuert kollisionsdetektering bruger vi **imellem-bevægelser** (Kapitel 4) til at beskrive et objekts bane imellem kendte positioner. Den benyttede imellem-bevægelse har til formål at tjekke for kollision - den bruges ikke til visualisering.
- Vi bruger **interval aritmetik** (Kapitel 5) til at afgrænse den valgte imellem-bevægelse for et objekt og opnår herved et minimum og maksimum for objektets position imellem to kendte positioner.
- Dette maksimum og minimum på objektets position giver os en afgrænsning af den valgte omringende volume type og vi kan afgøre om de omringende volumener for to objekter rammer hinanden i det givne tidsrum (Kapitel 6.1, 6.2 og 6.3).
- I tilfælde af kolliderende omringende volumener undersøges geometrien i de to kolliderende par af volumener for kollisioner (Kapitel 7).

I et senere afsnit vil metoden blive optimeret.

4 Imellem-bevægelser

4.1 Introduktion

Som beskrevet i Kapitel 1 er den faktiske bevægelse af et givet objekt ikke altid kendt. For at kunne tjekke for kollision kontinuert er vi nødt til at finde bevægelser imellem de kendte positioner af et objekt. Vi kalder disse bevægelser for **imellem-bevægelser**. Til en givet imellem-bevægelse m stiller vi følgende krav

- Bevægelsen skal være kontinuert, dvs. $m \in \mathcal{C}^0$. Man kan stille krav om højere kontinuitetsgrad hvis nødvendigt.
- Det er klart at bevægelsen skal interpolere imellem de to positioner. Den valgte interpolering kan være hvad som helst, f.eks. lineær, polynomiel eller en spline.

- Bevægelsen skal bevare formen af objektet. Vi kan f.eks. ikke have at vores bevægelse bevæger nogle objekt-punkter vha. en lige linje hvis objektet bliver roteret.

Udover disse tre krav kan et program selvfølgelig stille sine egne, afhængigt af de objekter der skal behandles.

Vi viser i Figur 2 et eksempel på en imellem-bevægelse. En firkløvers position

Figur 2: Eksempel på imellem-bevægelse. Røde (hullede) firkløvere er interpolerede positioner, sorte (udfyldte) er kendte positioner.

er kendt til seks tider $t_0, \ldots, t_5$, disse positioner $p_0, \ldots, p_5$ er markeret ved at firkløveren er tegnet i sort. Positionerne kan f.eks. være et resultat af brugerens interaktion med en mus. Vi ved intet om objektets bevægelse imellem disse seks punkter. For at kunne foretage kollisionsdetektering interpoleres positioner nu imellem de kendte (det er vigtigt at bemærke at disse nye positioner ikke skal visualiseres - de skal kun bruges til kollisionsdetektering).

De nye positioner interpoleres vha. en imellem-bevægelse, i figuren tegnes fire interpolerede positioner imellem hvert par af de kendte, disse er tegnet i rødt. Den benyttede imellem-bevægelse er en **konstant translation og rotation imellem-bevægelse**, den er beskrevet nedenfor i Kapitel 4.2. Vi ser at den overordnede bevægelse bevares ved brug af en imellem-bevægelse, dette skyldes at bevægelsen skal interpolere mellem kendte positioner og ikke må ændre på objektets form.

I [10] angives to bevægelser. Den første er **bevægelse med konstant rotation og translation**. Den anden er **bevægelse med konstant rotation og translation hvor rotation og translation deler samme akse**, dvs. hvor objekter bevæger sig langs akse u og roteres omkring samme akse u.

Det er klart at den sidstnævnte er et specieltilfælde af den første, i [10] introduceres den udelukkende fordi den hurtigere kan evalueres og herved giver bedre køretider. Vi vælger i vores rapport kun at implementere den generelle bevægelse, dette gøres for overskuelighedens skyld og fordi vi som sagt er i stand til at opnå de samme bevægelser med blot den førstnævnte. I vores implementation af metoden beskrevet i denne rapport vil vi forsøge at benytte redskaber i C++ til at gøre det nemt senere at tilføje nye bevægelser, se Kapitel 8.

Man kan overveje om det overhovedet er realistisk at antage at disse imellem-bevægelser vil interpolere i nærheden af den "rigtige" bane imellem to kendte

positioner. Det er korrekt at vi faktisk ikke kan sige noget om hvor godt vores valgte imellem-bevægelse kan gøre dette.

Under antagelse af at objekterne bevæger sig "pænt" (dvs. at de bevæger sig i en bane der kan beskrives vha. simple funktioner) er det dog i de fleste tilfælde muligt at finde en imellem-bevægelse der tilnærmer sig den rigtige bane tilstrækkeligt meget.

4.2 Imellem-bevægelse med konstant rotation og translation

Lad os antage at positioner er givet ved homogene 4x4 matricer på formen

$$\begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0} & 1 \end{bmatrix},$$

hvor $\mathbf{R}$ er en orienteringsmatrice (en 3x3 matrice) og $\mathbf{T}$ er en translationsmatrice (en tre-dimensionel vektor), $\mathbf{0}$ skal læses som tre vertikale nuller.

Vi er givet to tidspunkter t_0 og t_1. Til disse tidspunkter er tilknyttet en position $p_0 = \begin{bmatrix} \mathbf{R}_0 & \mathbf{T}_0 \\ \mathbf{0} & 1 \end{bmatrix}$ hhv. $p_1 = \begin{bmatrix} \mathbf{R}_1 & \mathbf{T}_1 \\ \mathbf{0} & 1 \end{bmatrix}$.

Den bevægelse vi beskriver her har konstant rotation og konstant translation imellem de to tider t_0 og t_1. Den skal altså bruges når det antages at objektet bevæger sig med konstant rotation og translation imellem kendte positioner.

Lad $a = T_1 - T_0$ være den *totale translation* i tiden $[t_0, t_1]$, så vil vores objekt være blevet translateret med

$$T(t) = T_0 + ta, \tag{4.2.a}$$

til tiden $t \in [t_0, t_1]$.

Lad ω være den *totale rotation* i tiden $[t_0, t_1]$. Lad $\mathbf{u} = (u_x, u_y, u_z)$ være den tilhørende rotationsakse. ω og $\mathbf{u}$ er givet ved matricen $\mathbf{R_1} \left(\mathbf{R}_0\right)^T$.

Orienteringen til tiden $t \in [t_0, t_1]$ er [10]

$$R(t) = \cos(\omega t)\mathbf{A} + \sin(\omega t)\mathbf{B} + \mathbf{C}, \tag{4.2.b}$$

De tre matricer $\mathbf{A}$,$\mathbf{B}$ og $\mathbf{C}$ er konstante og kan udregnes første gang imellem-bevægelsen benyttes. De tre matricer er

$$\mathbf{A} = \mathbf{R}_0 - \mathbf{u}\mathbf{u}^T\mathbf{R}_0, \tag{4.2.c}$$

$$\mathbf{B} = \mathbf{u}^*\mathbf{R}_1, \tag{4.2.d}$$

$$\mathbf{C} = \mathbf{u}\mathbf{u}^T\mathbf{R}_0, \tag{4.2.e}$$

$\mathbf{u}^*$ er matricen med egenskaben $\mathbf{u}^* x = ux$ for alle tre-dimensionelle vektorer x, den er givet ved

$$\mathbf{u}^* = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix}.$$

Disse matricer introduceres i [10], men hvorfor de er defineret som de ses ovenfor er ikke beskrevet i artiklen.

5 Interval aritmetik

5.1 Introduktion

Et redskab vi skal benytte flittigt er intervaller. Intervallerne kan være delmængder af f.eks. $\mathbb{N}, \mathbb{R}$ eller $\mathbb{C}$, eneste krav til mængden er at den er ordnet (dvs. at relationen $<$ giver mening på mængden).

Vi definerer intervallet $[a, b]$ til at være mængden

$$[a, b] = \{x \in X | a \leq x \leq b\}, \tag{5.1.a}$$

dette er selvfølgelig velkendt.

For to intervaller $a = [a_1, a_2]$,$b = [b_1, b_2]$ defineres nu følgende operatorer, samlet set udgør disse vores **interval aritmetik**

$$\begin{aligned}
[a_1, a_2] + [b_1, b_2] &= [a_1 + b_1, a_2 + b_2], && (5.1.b)\\
[a_1, a_2] - [b_1, b_2] &= [a_1 - b_2, a_2 - b_1], && (5.1.c)\\
[a_1, a_2] \times [b_1, b_2] &= [\min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}, && (5.1.d)\\
&\quad \max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}],\\
1/[a_1, a_2] &= [1/a_2, 1/a_1], \quad \text{hvis } a_1 > 0 \text{ eller } a_2 < 0, && (5.1.e)\\
[a_1, a_2]/[b_1, b_2] &= [a_1, a_2] \times (1/[b_1, b_2]), && (5.1.f)\\
&\quad \text{hvis } b_1 > 0 \text{ eller } b_2 < 0, && (5.1.g)
\end{aligned}$$

bemærk at kravet ved division er at intervallet ikke indeholder nul. Mængden af alle intervaller med ovenstående operatorer tilknyttet kalder vi $\mathbb{IR}$. Vi ser at vi kan indlejre $\mathbb{R}$ i $\mathbb{IR}$ ved at identificere $x \in \mathbb{R}$ med intervallet $[x, x]$. Vi kan desuden lade en n-dimensionel vektor v indeholde interval elementer

$$v = \begin{bmatrix} [i_{1_1}, i_{1_2}] \\ [i_{2_1}, i_{2_2}] \\ \vdots \\ [i_{n_1}, i_{n_2}] \end{bmatrix},$$

og vi kalder vektorrummet indeholdende disse vektorer for $\mathbb{IR}^n$.

5.2 Brug af interval aritmetik til afgrænsning

Vi kan benytte interval aritmetik til at afgrænse en funktion på et interval, dette skal vi udnytte senere i denne rapport. Med afgrænse menes at vi, givet en funktion $u : X \mapsto Y$ og et interval $x \subseteq X$, kan finde et interval $y \in Y$ så $u(x) \subseteq y$. Vi viser her et eksempel for at klargører hvad der menes med dette.

Antag at vi ønsker at afgrænse $u(t) = \sqrt{2}\sin(t) + \cos(t)$ for $t \in [0, \pi/2]$. Hvis vi kan afgrænse de to funktioner sin og cos, vil resten følge af regnereglerne i (5.1.b)-(5.1.f). Det er velkendt at $\sin(t) \in [0, 1]$ og $\cos(t) \in [0, 1]$ for $t \in [0, \pi/2]$, tallet $\sqrt{2}$ kan vi identificerer med intervallet $\left[\sqrt{2}, \sqrt{2}\right]$ og vi har

$$u(t) \in \left[\sqrt{2}, \sqrt{2}\right] \times [0, 1] + [0, 1].$$

Nu bruger vi først multiplikation af intervaller (5.1.d) og får

$$u(t) \in \left[0, \sqrt{2}\right] + [0, 1],$$

og nu addition af intervaller (5.1.b)

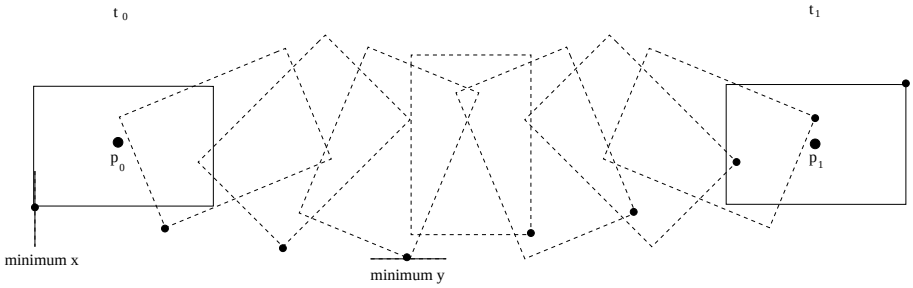
$$u(t) \in \left[0, 1 + \sqrt{2}\right].$$

Vi har nu afgrænset u ved $u(t) = \sqrt{2}\sin(t) + \cos(t) \in \left[0, 1 + \sqrt{2}\right]$ for $t \in [0, \pi/2]$. Bemærk at afgrænsningen ikke er "tæt", der gælder nemlig $u([0, \pi/2]) = [1, \sqrt{2}] \subset \left[0, 1 + \sqrt{2}\right]$. Årsagen er at $\sin(t)$ vokser på $t \in [0, \pi/2]$, mens $\cos(t)$ aftager.

Eftersom vi skal benytte interval aritmetik til at afgrænse vores bevægelse beskrevet i Kapitel 4, skal vi være i stand til at afgrænse analytiske funktioner, som f.eks. sin og cos, resten opnås ved brug af de regneregler vi har defineret i det forrige afsnit.

I de følgende kapitler vil vi gøre flittigt brug af at vi kan afgrænse omringende volumener Lad os illustrere hvordan dette kan gøres. Dette eksempel benytter positioner af samme form som beskrevet i Kapitel 4.2, dvs. 4x4 homogene matricer.

Antag at vi er givet et tidsrum $[t_0, t_1]$ og en bevægelse m der roterer en boks 180 grader om $z-$aksen og translaterer 10 enheder langs $x-$aksen. Se Figur 3 for en illustration af situationen. For at kunne afgrænse vores boks afgrænser vi



Figur 3: En boks roteres 180 grader om $z-$aksen og translateres samtidigt ti enheder langs $x-$aksen.

først vores bevægelse m.

Antag at vi har fundet intervallet $I_1 = [l_1, u_1] \in \mathbb{IR}$ vha. interval aritmetik der afgrænser translationen (4.2.a), dvs. at der gælder

$$l_1 \leq T(t) = T_0 + t(T_1 - T_0) \leq u_1,$$

hvor uligheden skal læses koordinatvis; l_1 er en tre-dimensionel vektor og uligheden siger f.eks. at den mindste $x-$værdi translationen påtager sig under tidsrummet $[t_0, t_1]$ er større end eller lig med $x-$elementet af l_1. Det samme gør sig gældende for y og z koordinaterne.

Antag på samme måde at $I_2 = [l_2, u_2]$ er et interval der afgrænser rotationen (4.2.b), dvs. at der gælder

$$l_2 \leq R(t) = \cos(\omega t)\mathbf{A} + \sin(\omega t)\mathbf{B} + \mathbf{C} \leq u_2.$$

Dette interval findes ved at begrænse sin og cos samt ved at bruge vores regneregler for interval aritmetik.

Bemærk her at l_2 og u_2 er 3x3 matricer; en indgang i matricen l_2 indeholder en værdi der er mindre end eller lig med tilsvarende indgang i rotationen givet af m i hele tidsintervallet $[t_0, t_1]$.

Vi indser nu følgende; hvis et punkt i vores boks roteres med l_2 og translateres med l_1, så er den resulterende vektor den *mindste* (koordinatvis) punktet påtager sig under hele tidsintervallet. På samme måde ser vi at hvis et punkt i vores boks roteres med u_2 og translateres med u_1, så er den resulterende vektor den *største* (koordinatvis) punktet påtager sig under hele tidsintervallet.

På denne måde kan vi nu afgrænse et vilkårligt punkt i vores boks indenfor et tidsrum, hvordan dette gøres præcist i vores program vil blive klargjort i Kapitel 8. På Figur 3 er de mindste værdier for punktet i nederste venstre hjørne angivet.

6 Kollisionsdetektering imellem omringende volumener

I dette kapitel præsenterer vi tre omringende volumener og hvordan de benyttes i diskret og i kontinuert kollisionsdetektering.

6.1 Omringende kugler

En omringende kugler er som bekendt en kugle K, givet ved et centrum c og en radius r, så der for punkterne V i det omringede objekt gælder

$$\forall v \in V, ||c - v|| \leq r,$$

hvor $||a - b||$ er den euklidiske afstand imellem punkterne a og b. Lad os bruge notationen $K(c, r)$ til at angive en kugle K med centrum c og radius r.

I det diskrete tilfælde foretages kollisionsdetekteringen imellem to kugler $K_1(c_1, r_1)$ og $K_2(c_2, r_2)$ ved testen

$$||c_1 - c_2|| > r_1 + r_2,$$

hvis denne ulighed er sand, er kuglerne adskilt.

Lad os nu udvide denne test til det kontinuerte tilfælde. Antag igen vi er givet to kugler $K_1(c_1, r_1)$ og $K_2(c_2, r_2)$ som omringer objekt O_1 hhv. objekt O_2. Antag desuden at positionen af O_1 er givet ved imellem-bevægelsen m_1 og positionen af O_2 er givet ved imellem-bevægelsen m_2. Vores tidsrum er $[t_0, t_1]$.

Vi bemærker først at begge kuglers radius er konstant imellem de to kendte positioner, dette ved vi af de tre krav[1] til vores imellem-bevægelse (at imellem-bevægelsen bevarer objektets form).

Lad os nu kigge på kuglernes centrum. Deres positioner til tiden $t \in [t_0, t_1]$ vil vi kalde $c_1'(t)$ hhv. $c_2'(t)$. $c_1'(t)$ og $c_2'(t)$ er funktioner af $m_1(t)$ hhv. $m_2(t)$.

Antag at vi har afgrænset de to kuglers centrum i tidsrummet $[t_0, t_1]$ vha. interval aritmetik, dvs. vi er givet to intervaller $I_1 = [a, b]$ og $I_2 = [c, d]$ hvorom der gælder

$$\begin{matrix} c_1'(t) \in I_1 \\ c_2'(t) \in I_2 \end{matrix} , \quad \text{for alle } t \in [t_0, t_1].$$

Bemærk at dette betyder at $c_1'(t) \geq a$ og $c_2'(t) \leq d$ i vores tidsrum.

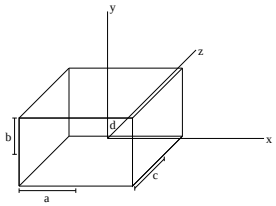
Vi bruger nu interval aritmetik til at udregne intervallet $I = [e, f] = I_1 - I_2 = [a, b] - [c, d] = [a - d, b - c]$, for denne mængde har vi $c_1'(t) - c_2'(t) \in I$ for alle $t \in [t_0, t_1]$. Idet den nedre grænse af I er den mindste afstand imellem de to kugler *i hele tidsrummet* $[t_0, t_1]$ er K_1 og K_2 adskilt i hele tidsintervallet hvis

$$e > r_1 + r_2,$$

hvis denne ulighed er sand kan K_1 og K_2 ikke kolliderer, hvis uligheden derimod er falsk er der potientielle for en kollision (vi kan ikke vide dette med sikkerhed hvis vores afgrænsninger ikke er tætte).

6.2 Omringende akse-orienterede bokse

Akse-orienterede bokse (engelsk forkortet AABB) er en boks givet ved et centrum d og boksens længde a, b, c langs de tre akser. Boksens længde er normalt angivet i afstanden fra boksens centrum til én af dens kanter i den givne akses retning (i modsætning til afstanden fra kant til kant i aksens retning). Boksen skal være orienteret langs de tre akser x,y og z, et eksempel kan ses i Figur 4.



Figur 4: En akse-orienteret boks. d er boksens centrum, a, b og c boksens længde langs de tre akser x, y og z.

Lad os benytte notationen $A(d, a, b, c)$ til at angive en boks A med centrum d samt akse-afstande (i rækkefølgen x, y, z) a, b og c.

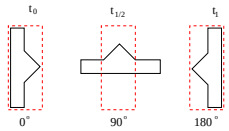
I det traditionelle diskrete tilfælde foretages en "adskillende-akse-test" til at afgøre om to AABBer $A_1(d_1, a_1, b_1, c_1, d_1)$ og $A_2(d_2, a_2, b_2, c_2, d_2)$ rammer

[1] Se Kapitel 4.

hinanden [2] [5]. Vi vælger ikke at gennemgå denne test her, da testen er den samme som den der vil blive beskrevet i næste kapitel i forbindelse med OBBer. Testen for AABBer er svagere end for OBBer idet man udnytter at boksen er orienteret langs de tre akser x,y og z. Testen for AABBer er på grund af dette hurtigere end den for OBBer og det er derfor AABBer overhovedet overvejes som omringende volume.

Antag nu at vi er givet et objekt O med positionen P_0 til tiden t_0 og positionen P_1 til tiden t_1. Antag at vi er givet en boks A som omringer objektet O ved de to positioner P_1 og P_0. Afhængigt af objektets størrelse kan det tage lang tid at finde en omringende boks A, derfor vil A normalt være blevet udregnet på et tidligere stadie og gemt i en fil.

Vi ser at vi ikke kan vide om A vil omringe objektet O i hele tidsintervallet $[t_0, t_1]$ hvis den valgte imellem-bevægelse m roterer O, dette er vist i Figur 5. Vi kan vælge også at roterer A efterhånden som vi bevæger os fra t_0 til t_1, men



Figur 5: Den omringende akse-orienterede boks A (tegnet med stiplede røde linjer) omringer objektet O til tiderne t_0 og t_1. Idet vores imellem-bevægelse m roterer O passer boksen dog ikke til tiden $t_{1/2}$.

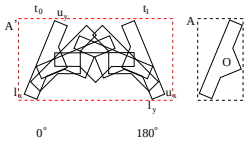
gør vi dette er boksen A ikke længere orienteret med de tre akser x, y og z; A bliver en OBB.

En anden mulighed er at definere en boks A' så den omringer O *under hele tidsintervallet* $[t_0, t_1]$. Dette kan gøres på følgende måde:

Det er klart at hvis vi bevæger A med samme bevægelse som O, så vil A til ethvert tidspunkt omringe O, idet m bevarer form af O og A. Hvis vi benytter interval aritmetik til at afgrænse m kan vi afgrænse A, idet A jo blev flyttet vha. m. At vi har afgrænset A vil sige at vi har fundet de mindste og maksimale $x-$, $y-$ og $z-$ værdier alle hjørner af A påtager sig under tidsintervallet $[t_0, t_1]$. Lad l_x, l_y og l_z være den mindste $x-$, $y-$ og $z-$værdi hjørnerne af A påtager sig under tidsrummet. u_x,u_y og u_z defineres på samme måde til de maksimale værdier der antages. Se Figur 6 for et eksempel. I figuren er objektet O og dets omringende AABB A tegnet til højre. l_z og u_z er undladt da eksemplet foregår i x/y planet. Når først A' er fundet kan denne omringende volume benyttes til at teste for kollision i hele tidsintervallet $[t_0, t_1]$: hvis A' ikke kolliderer med en anden AABB så kan den underliggende geometri heller ikke kollidere.

Det er klart at den nye boks A' på ingen måde er en optimal (dvs. mindste) omringede boks for O[2]. Er tidsintervallet tilstrækkeligt lille vil det dog, afhæn-

[2] A' kan selvfølgelig også dannes ved at afgrænse O og derved finde minimum og maksimum for punkter i O i tidsintervallet $[t_0, t_1]$. Dette kan dog tage væsentligt længere tid end at gøre det tilsvarende på A idet O kan indeholde vilkårligt mange punkter.

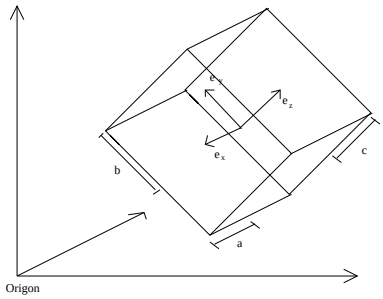


Figur 6: Objektet O er blevet afgrænset i tidsintervallet $[t_0, t_1]$ og A' er herefter defineret som den mindste boks der indeholder O i hele tidsintervallet $[t_0, t_1]$. A' er tegnet i rødt med stiplede linjer.

gigt af objektet og bevægelsen, være muligt at A' passer tilstrækkeligt godt nok til at AABBer kan bruges effektivt i det kontinuerte tilfælde.

6.3 Orienterede omringende bokse

Vi beskriver nu orienterede omringende bokse (engelsk forkortet OBB). OBBer er defineret på samme måde som AABBer, blot behøver OBBer ikke at være orienteret langs $x-$, $y-$ og z-aksen. Lad os bruge notationen $O(d, \mathbf{e}, a, b, c)$ til at beskrive boksen O med centrum d og orientering $\mathbf{e} = (e_1, e_2, e_3) \in \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3$. a, b og c er boksens halve længde langs akserne e_1, e_2 og e_3, se Figur 7 for et eksempel.



Figur 7: Orienteret omringende boks O.

I [4] beskrives den "adskillende-akse-test" til at teste om to OBBer

$$O_1(d_1, \mathbf{e}, a_1, a_2, a_3)$$
$$O_2(d_2, \mathbf{f}, b_1, b_2, b_3)$$

rammer hinanden, fremgangsmåden er som følger.

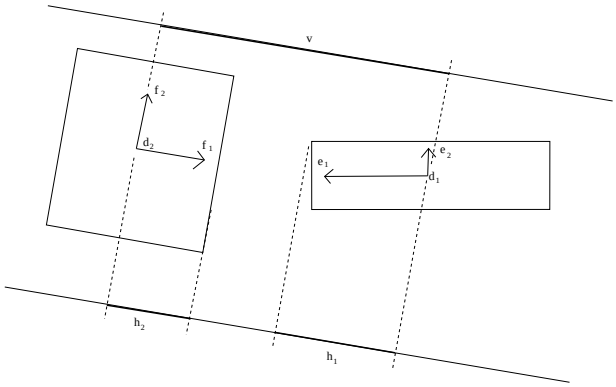
For alle akser h i mængden

$$\{e_i, f_j, e_i \% f_j \, 1 \leq i \leq 3, 1 \leq j \leq 3\},$$

hvor % er krydsproduktet og $\cdot$ prik-produktet, undersøges uligheden

$$|h \cdot (d_1 - d_2)| > \sum_{i=1}^{3} a_i |a \cdot e_i| + \sum_{i=1}^{3} b_i |a \cdot f_i|. \tag{6.3.a}$$

Vi ser at venstre side af (6.3.a) er afstanden imellem de to OBBers centrum, projekteret ned på aksen h. Højre side er den projekterede afstand af boksenes sider (den halve længde af siderne) ned på samme akse h, et eksempel i x/y-planet kan ses i Figur 8. I figuren er v venstre side af (6.3.a), dvs. afstanden



Figur 8: "Adskillende-akse test" imellem OBBerne O_1 og O_2.

$|h \cdot (d_1 - d_2)|$. De to variable h_1 og h_2 er afstandene $\sum_{i=1}^{3} a_i |a \cdot e_i|$ og $\sum_{i=1}^{3} b_i |a \cdot f_i|$. Aksen der testes imod er f_1, den adskiller de to OBBer.

Der gælder følgende [4]; O_1 og O_2 rammer hinanden hvis samtlige 15 akser ikke adskiller O_1 og O_2, altså når uligheden (6.3.a) er falsk i alle 15 tilfælde. Hvis minimum én ulighed viser sig at være sand når de 15 akser undersøges, så kolliderer O_1 og O_2 derimod ikke. Det er altså ikke nødvendigt at tjekke flere akser hvis først en akse er fundet hvor uligheden er sand. Lad os nu udvide denne test til det kontinuerte tilfælde.

Vi kan vha. interval aritmetik afgrænse alle indgående variable i tidsrummet $[t_0, t_1]$: Først ved at afgrænse de givne imellem-bevægelser m_1 og m_2 og hernæst de to OBBer O_1 og O_2. Når først positionerne af O_1 og O_2 er afgrænset er det trivielt at afgrænse OBBernes parameter (orienteringen af dem osv., dette beskrives mere udførligt i Kapitel 8). Vha. regnereglerne i Kapitel 5 kan vi, for en given akse $h \in \{e_i, f_j, e_i \% f_j \, 1 \leq i \leq 3, 1 \leq j \leq 3\}$, nu omskrive (6.3.a) til at være to intervaller I_1 og I_2 hvor I_1 er afgrænsningen af venstre side og I_2 afgrænsningen af højre side af (6.3.a).

Vi har nu uligheden

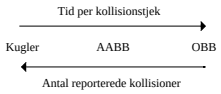
$$I_1 > I_2,$$

det er klart at hvis den nedre grænse af I_1 er større end den øvre grænse af I_2, så adskiller h de to OBBer i hele tidsrummet $[t_0, t_1]$, idet uligheden altid

må være sand. Hvis dette er tilfældet kan vi konkluderer at de to OBBer ikke kolliderer i tidsrummet $[t_0, t_1]$.

6.4 Valg af omringende volume i vores implementation

Generelt gælder følgende for de tre gennemgåede omringede volumener; kugler reporterer flest kollisioner idet de har den største volume i forhold til objektet, samtidigt er kugler dog de hurtigste at teste for kollisioner. For OBBer er det omvendt; OBBer passer generelt bedst omkring objektet men testen for kollision imellem OBBer er samtidigt den mest komplekse, situationen er afbildet i Figur 9. Som vi har set er AABBer ikke praktiske i det kontinuerte tilfælde; hvis be-



Figur 9: De tre omringende volumener vi har gennemgået.

vægelsen der skal interpolere imellem kendte positioner benytter sig af rotation, så vil de tilhørende AABBer ikke altid være orienteret langs koordinatsystemets tre akser.

Vi så at den kontinuerte test for kugler er meget lig den velkendte diskrete, denne test er let at implementere grundet dens simplicitet.

Testen for OBBer er også meget lig den tilsvarende diskrete test; forskellen er blot at de indgående variable er funktioner af tid og afgrænses over det givne tidsinterval hvorved intervaller opnås.

Vi vælger i vores implementation at bruge OBBer. Vi skal i Kapitel 9 også benytte os af kugler.

6.5 Træer af omringende volumener

Som beskrevet i [5] er træstrukturer en effektiv metode til at teste for kollision.

Et træ af OBB-volumer opbygges ved at roden af træet indeholder en omringede volume der omringer hele det givne objekt. Afhængigt af antallet n af børn af roden dannes n omringende volumener der tilsammen omringer hele objektet. De n volumener vil normalt (det afhænger af den valgte opslitningsmetode, se nedenfor) hver have et volume ca. lig N/n hvor N er volumen af det omringede volume i roden af træet. For hver af rodens børn foretages denne opdeling igen, de nye omringede volumener har et volume ca. lig N/n^2 hver (igen afhængigt af metoden der bruges til at splitte det givne volume). Samlet set omringer volumener den geometri deres far omringer. Opbygningen af træet stopper når de nederste volumener alle indeholder et prædefineret antal af geometri-elementer, f.eks. hvis alle blade hver indeholder maks to trekanter.

Lad os antage at vi er givet to objekter og tilhørende træer af omringende volumener. Hvis de to rødder viser sig ikke at kollidere så kan de to underliggende objekter heller ikke kollidere og testen er færdig.

Hvis de to rødder kolliderer vælges et træ ud og børnene af dette træs rod testes nu for kollision imod det andet træs rod. Udvælgelsen af et træ kan f.eks. ske ved at vælge at opsplitte træet hvor roden har den største volume.

Opdelingen fortsættes indtil der ikke er flere kolliderende volumener Hvis to blade viser sig at ramme hinanden under testen, så tjekkes den underliggende geometri i de to volumener mod hinanden, evt. kollisioner lagres og kan senere behandles. Vi viser nedenfor pseudokode for denne test

```
Q er en kø
A<-rod af træ1
B<-rod af træ2

if (A og B kolliderer) {
   if (volume af A > volume af B) {
        Lad n_0,...,n_k være alle børn af A
        Tilføj tupler (n_i, B) til Q for 0<=i<=k
   } else {
        Lad n_0,...,n_k være alle børn af B
        Tilføj tupler (A, n_i) til Q for 0<=i<=k
   }

   mens (Q ikke tom) {
        (A,B)<-forreste element af Q
        Fjern forreste element af Q

        if (A og B ikke kolliderer) continue

        if (A og B er blade) {
            Tjek geometri i A og B
            Reporter evt. kollisioner
        } else {
            if (A ikke blad og volume af A > volume af B) {
                Lad n_0,...,n_k være alle børn af A
                Tilføj tupler (n_i, B) til Q for 0<=i<=k
            } else {
                Lad n_0,...,n_k være alle børn af B
                Tilføj tupler (A, n_i) til Q for 0<=i<=k
            }
        }
   }
}
```

Algoritmen kan uden ændring overføres til det kontinuerte tilfælde. Vi tilføjer dog en tid t til alle reporterede kollisioner. Når træet er gennemløbet slettes alle kollisioner der har større kollisionstid end den tidligste fundet: vi er kun interreseret i hvornår objekterne første gang rammer hinanden.

Vi diskuterer i Kapitel 9 hvordan vi kan optimere algoritmen i det kontinuerte tilfælde.

7 Kollisionsdetektering imellem geometrier

I dette afsnit beskriver vi hvordan vi kan tjekke for kollision imellem geometrier kontinuert. Der er seks (af symmetriske grunde) måder to geometrier kan kollidere på [7], disse navngives ofte via. forkortelserne V for vertex (punkt), F for face (flade) og E for edge (kant). De seks typer er

$$(V,V),(V,E),(V,F),(E,E),(E,F),(F,F),$$

hvor et par (i,j) skal læses som en kollision imellem et element af typen i og et element af typen j.

Lad nu et tidsinterval $[t_0,t_1]$ være givet. Lad desuden to objekter O_1 og O_2 være givet. Hvis vi til at starte med antager at objekterne er fuldstændigt fraskilt til tiden t_0 og evt. senere kolliderer (grundet deres bevægelser) kan vi indskrænke disse seks tilfælde til to

$$(E,E),(V,F). \tag{7.0.a}$$

Lad os argumenterer for hvorfor dette er tilfældet ved at forklare hvordan de resterende fire ikke kan forekomme uden en af de to i (7.0.a) enten samtidigt forekommer, eller er forekommet tidligere.

- (V,V) er to punkter der rammer hinanden. Idet disse to punkter har tilhørende kanter i hver sit objekt vil minimum to kanter også ramme hinanden, dvs. at denne kollisionstype er inkluderet i typen (E,E).

- (V,E) Knuden V der rammer kanten E har som ovenfor tilhørende kanter og derfor er dette tilfælde inkluderet i typen (E,E).

- (E,F) Hvis kanten E i objektet O_1 rammer fladen F af objektet O_2 må en af følgende være sandt: a) Ét punkt af E er udenfor O_2, det andet er indenfor O_2. b) Ét eller begge punkter af E ligger på fladen F.

  I tilfælde a) argumenterer vi som følger. Idet vi ovenfor antog at objekterne til at starte med var adskilt må det punkt der nu ligger indenfor O_2 havde passeret igennem fladen F, derfor er der tidligere forekommet en kollision af typen (V,F). Idet vi tester for denne kollisionstype vil kollisionen derfor blive fanget før kollisionstypen (E,F) overhovedet kan forekomme.

  I tilfælde b) argumenterer vi som følger. Hvis én eller begge punkter af E rammer fladen F, så er der også kollisioner af typen (V,F) og kollisionen fanges herved.

- (F,F) Begge flader er givet ved deres hjørnepunkter og deres kanter. Hvis de to flader rammer må der enten være en kollision af typen (E,E) eller en kollision af typen (V,F).

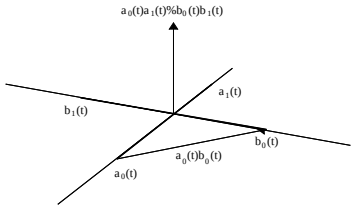
Bemærk at vi skal tjekke tre ting når vi tjekker for kollision imellem de to objekter O_1 og O_2; de to kollisionstyper i (7.0.a) giver nemlig følgende tre muligheder

- En kant i O_1 rammer en kant i O_2 (dette er (E, E)).
- Et punkt i O_1 rammer en flade af O_2 (dette er (V, F)).
- En flade af O_1 rammer et punkt i O_2 (dette er også (V, F)).

Vi antog til at starte med at objekterne var fuldstændigt adskilt til at starte med. For at fange kollisioner hvor objekterne kolliderer allerede til tiden t_0 kan vi til denne tid foretage en test hvor vi tjekker for alle seks mulige kollisionstyper, dvs. en traditionel diskret test. Lad os nu vise hvordan vi kan tjekke for kollisionstyperne i (7.0.a), vi starter med (E, E).

7.1 Test for kollision imellem kanter

Antag at vi er givet to kanter a og b. Antag at a er kanten fra a_0 til a_1 og b er kanten fra b_0 til b_1. Idet vi skal foretage en kontinuert test er vi nødt til at teste for kollision over et tidsrum, lad derfor alle disse fire hjørnepunkter være en funktion af tiden (deres position er en funktion af tiden og er givet ved den valgte imellem-bevægelse). Lad os skrive linen a, som funktion af tiden $t \in [t_0, t_1]$, ved $a_0(t)a_1(t)$, lad os på samme måde skrive b ved $b_0(t)b_1(t)$. Vi indser at hvis a og b kolliderer så kolliderer de to linjer indeholdene linjestykkerne a hhv. b også, dette er illustreret i Figur 10. Lad os derfor starte med at afgøre



Figur 10: De to kanter $a_0(t)a_1(t)$ og $b_0(t)b_1(t)$ samt de tilhørende linjer.

om de to linjer indeholdende a hhv. b kolliderer. Dette gør vi ved testen (hvor % er krydsproduktet og $\cdot$ prik-produktet)

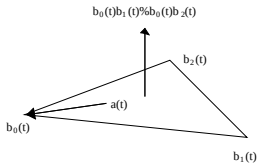
$$a_0(t)b_0(t) \cdot (a_0(t)a_1(t)\%b_0(t)b_1(t)) = 0.$$

Ligningen ovenfor er lig nul når vektoren $a_0(t)b_0(t)$ (læs vektoren fra $a_0(t)$ til $b_0(t)$) ligger i det plan de to kanter a og b tilsammen giver. Hvis vi finder at de to linjer kolliderer første gang til tiden t (vi viser senere i dette kapitel hvordan dette tidspunkt t findes), så kan vi foretage en alm. diskret kant/kant test [1] og afgøre om de to kanter rammer hinanden.

7.2 Test for kollision imellem flade og punkt

Bemærk at denne test skal benyttes i tilfældet (V, F) og i tilfældet (F, V), testen er den samme i begge tilfælde grundet symmetri.

Lad punktet a og fladen b være givet. Antag at b er angivet ved de tre hjørner b_0,b_1 og b_2. Som ovenfor er vi nødt til at lade disse fire punkter være funktioner af tiden, vi skriver dette ved $a(t)$, $b_0(t)$, $b_1(t)$ og $b_2(t)$, se Figur 11 for opstillingen. Vi starter med at afgøre om punktet $a(t)$ ligger i det plan som



Figur 11: Punktet $a(t)$ og fladen givet ved $b_0(t)$, $b_1(t)$ og $b_2(t)$.

er givet ved fladen b. Dette gøres ved testen

$$a(t)b_0(t) \cdot (b_0(t)b_1(t)\%b_0(t)b_2(t)) = 0,$$

ligningen ovenfor er nul når vektoren $a(t)b_0(t)$ (læs vektoren fra $a(t)$ til $b_0(t)$) ligger i planet givet ved normalen $b_0(t)b_1(t)\%b_0(t)b_2(t)$. Når vektoren $a(t)b_0(t)$ til tiden t (vi viser senere i dette kapitel hvordan dette tidspunkt t findes) ligger i dette plan kan vi bruge en traditionel diskret test (f.eks. vha. barycentriske koordinater) og afgøre om $a(t)$ ligger i fladen b.

7.3 Brug af interval aritmetik til at finde tidligst kollisionstid

I de to foregående kapitler har vi fundet følgende to ligninger

$$\begin{aligned} a_0(t)b_0(t) \cdot (a_0(t)a_1(t)\%b_0(t)b_1(t)) &= 0, \\ a(t)b_0(t) \cdot (b_0(t)b_1(t)\%b_0(t)b_2(t)) &= 0. \end{aligned}$$

Vi så at når en af disse ligninger er opfyldt, så er der potentielle for en kollision imellem den tilhørende geometri. Lad os nu beskrive hvordan vi kan finde en tid t der opfylder en af disse ligninger.

Lad os uden tab af generalitet først antage at $t \in [0, 1]$. Vi bemærker at begge ligninger er på formen

$$f(t) = 0, \quad t \in [0, 1]. \tag{7.3.b}$$

Vores ønske er at finde den mindste værdi af t så (7.3.b) er opfyldt. Vi ønsker at finde den mindste værdi fordi det er her den første kollision potientielle kollision forekommer.

Vi kan vha. interval aritmetik afgrænse $f(t)$ for $t \in [0,1]$ - under antagelse af at de indgående udtryk i f er en kombination af operatorene i Kapitel 5 og funktioner hvor afgrænsning er kendt på forhånd (dvs. f.eks. sin og cos). Hvis intervallet der afgrænser f ikke indeholder nul (dvs. at de to grænseværdier har samme fortegn) kan f ikke antage nul for $t \in [0,1]$.

Hvis intervallet derimod indeholder nul er det muligt at f antager værdien nul for et passende t. Vi kan ikke med sikkerhed vide om f antager værdien nul; hvis afgrænsningen ikke er tæt eller hvis funktionen ikke er kontinuert (alle funktioner vi benytter vil dog være kontinuerte) kan denne situation forekomme. I tilfælde af at intervallet indeholder nul gør vi som følger; tidsintervallet $[0,1]$ deles i to lige store delintervaller $I_1 = [0,1/2]$ og $I_2 = [1/2,1]$. Herefter undersøges det om $f(t)$, afgrænset for $t \in I_1$ indeholder nul (vi tjekker $[0,1/2]$ først idet vi ønsker det tidligst mulige nulpunkt for f), hvis f ikke kan antage nul på I_1 tjekkes $f(t)$ for $t \in I_2$.

Denne proces kan gentages til en vilkårlig fin inddeling af tidsintervallet $[0,1]$, når opdelingen er blevet tilstrækkelig fin og f har potientielle for et nulpunkt på et interval $[a,b]$ kan vi bruge den tilsvarende diskrete test til at tjekke om den underliggende geometri kolliderer til tiden a. Bemærk at denne test skal benytte sig af en vis fejltolerance grundet afrunding og hvor fint tidsintervallet bliver delt op. Hvis geometrien viser sig at kollidere har vi fundet en kollision og testen er slut. Hvis geometrien derimod ikke kolliderer foresætter vi med vores test for nulpunkter af f i næste delinterval.

Bemærk at vi tester for kollision til tiden a, der er her en vigtig pointe: Grundet den måde vi har gennemløbet det oprindelige tidsinterval $[0,1]$ (ved altid at tjekke det "lave" interval først) kan *ingen* på, kan ingen kollisioner forekomme før tiden a. Dette medfører at vi kan garantere at de to objekter ikke har penetreret hinanden til tiden a: hvis dette var tilfældet ville der havde været kollisioner før tiden a og disse skulle så være blevet fanget da tidligere intervaller blev tjekket.

Vi bemærker desuden at vi, med tilpas store fejltolerancer, kan finde en kollision til tiden a når kollisionen i virkeligheden forekommer for en værdi $a < x \leq b$. Vi kalder derfor a en konservativ kollisionstid.

8 Programmeringsovervejelser

Vores implementation af den gennemgåede metode vil blive skrevet i C++. Vi vil bruge OpenTissue [8] som en basis for vores løsning; f.eks. vil OpenTissue stille matrix- og vektor-klasser til rådighed, OpenTissue vil også gøre det nemt at hente objekter fra filer til at teste med.

Lad os starte med at overveje hvordan vi vil implementere vores imellembevægelse.

8.1 Imellem-bevægelser

Lad os først overveje hvilke krav vi stiller til vores implementation af disse imellem-bevægelser.

1) Vi ønsker nemt at kunne tilføje nye bevægelser senere. F.eks. kan det være relevant at tilføje en ny bevægelse fordi den er hurtigere f.eks. at afgrænse eller evaluere, dette vil typisk være fordi bevægelsen ikke er så generel som den vi implementerer.

2) Idet de nye bevægelser kan benytte meget forskelligt til at interpolere positioner er det ikke nemt at skrive kode til at afgrænse disse bevægelser generelt. Vi ønsker derfor at nye bevægelser selv skal tilbyde kode til at foretage disse afgrænsninger.

Det første krav kan opfyldes ved at bruge virtuelle funktioner i C++. Vi laver en klasse `arbitrary_motion` som er et skelet for en bevægelse (svarer til et interface i Java). I vores implementation benyttes udelukkende de funktioner der er angivet i denne basis klasse. Hvis nye bevægelser skal tilføjes skal de blot nedarve vores klasse og tilbyde de virtuelle funktioner klassen har defineret.

Krav nummer to er også opfyldt ved at bruge denne basis klasse. Vi angiver blot i denne klasse at der skal være funktioner til at afgrænse bevægelsen til rådighed. Klassen `arbitrary_motion` kan ses i denne rapports Bilag C.

8.2 Interval aritmetik

Vi vælger at benytte boost-bibliotekets implementation af intervaller (se dog Kapitel 9). Et interval med reelle endepunkter angives ved

```
typedef boost::numeric::interval<double> interval_type;
```

Eftersom OpenTissues vektor- og matrix-klasse bruger templates kan vi nemt definerer vektorer og matricer hvor de grundlæggende elementer er intervaller, dette er blot at skrive

```
typedef OpenTissue::vector<interval_type>    vector_type;
typedef OpenTissue::matrix3x3<interval_type> matrix_type;
```

Vores vektortype er nu en tredimensionel vektor hvor hver indgang er et interval med endepunkter af typen `double`. På samme måde indeholder vores matricer elementer af typen intervaller med endepunkter af typen `double`. Idet vi benytter klasserne i OpenTissue kan vi allerede nu f.eks. multiplicere vektorer indeholdende intervaller; der skal ikke kodes specifikke funktioner for at opnå dette.

Lad os nu vise hvor let det er f.eks. at afgrænse de indgående variable i den "adskillende-akse-test" beskrevet tidligere.

Antag vi er givet et tidsrum $[t_0, t_1]$ Antag endvidere vi er givet to OBBer $O_1(d_1, \mathbf{e}, a_1, b_1, c_1)$ og $O_2(d_2, \mathbf{f}, a_2, b_2, c_2)$ som skal undersøges for kollision indenfor tiden $[t_0, t_1]$, antag at bevægelserne af O_1 og O_2 er givet ved de to bevægelser m_1 og m_2.

Vi benytter først at de to klasser m_1 og m_2 tilbyder funktioner der afgrænser bevægelsen indenfor et givent tidsinterval, her $[t_0, t_1]$.

Lad R_0 være en matrice af typen `matrix_type` der afgrænser m_1s rotation af O_1 i tidsintervallet $[t_0, t_1]$. Lad T_0 være en vektor af typen `vector_type` der afgrænser m_1s translation af O_1 i tidsintervallet $[t_0, t_1]$. Lad R_1 og T_1 være defineret som ovenfor, blot for OBB O_2.

Vi kan nu nemt afgrænse vores to OBBer: afgrænsningen af de to centrum d_1 og d_2 er givet ved

```
vector_type boundD1 = R_0 * (vector_type)(d_1) + T_0;
vector_type boundD2 = R_0 * (vector_type)(d_2) + T_1;
```

hvor `(vector_type)(d_1)` er vektoren $([d_{1_x}, d_{1_x}], [d_{1_y}, d_{1_y}], [d_{1_z}, d_{1_z}])^T$, tilsvarende for `(vector_type)(d_2)`.

Akserne i **e** hhv. **f** afgrænses på samme måde, blot skal der her ikke translateres med T_0 hhv. T_1 idet akserne er vektorer givet ved de to centrum.

Lad os desuden vises hvordan vi kan foretage den "adskillende-akse-test" for en akse h under antagelse af at samtlige indgående variable nu er blevet afgrænset. Vi benytter følgende terminologi. `boundH` er den afgrænsede akse h. `boundEi` er den afgrænsede akse e_i, tilsvarende er `boundFi` den afgrænsede akse f_i. Højre side af (6.3.a) kan afgrænses ved

```
double lower_limit_left =
    (boundH * (boundD2 - boundD1)).get_abs_lower();
```

hvor * er prik-produktet som defineret i OpenTissue og `get_abs_lower()` returnerer den nedre grænse af den absolutte værdi af det givne interval. Venstre side af (6.3.a) kan afgrænses ved

```
double upper_limit_right =
    a_1 * (boundH * boundE1).get_abs_upper() +
    b_1 * (boundH * boundE2).get_abs_upper() +
    c_1 * (boundH * boundE3).get_abs_upper() +
    a_2 * (boundH * boundF1).get_abs_upper() +
    b_2 * (boundH * boundF2).get_abs_upper() +
    c_2 * (boundH * boundF3).get_abs_upper();
```

hvor `get_abs_upper()` returnerer den øvre grænse af den absolutte værdi at det givne interval. Som beskrevet i Kapitel 6.3, så er h en adskillende akse i hele tidsintervallet $[t_0, t_1]$ når `lower_limit_left` er større end `upper_limit_right`. Vi skal i Kapitel 9 optimere på denne kontinuerte test.

Vi har nu vist hvor let det er at benytte interval aritmetik. Dette skyldes i høj grad den generalitet OpenTissue tilbyder i form af templates.

9 Optimeringer

Vi har i forbindelse med dette projekt brugt meget tid på at optimere vores implementation. Vi beskriver nedenfor nogle af de optimeringer vi har imple-

menteret, samt nogle der ikke blev implementeret grundet tidsmangel. Vi har i forbindelse med vores optimering draget meget nytte af artiklerne [9] og [11].

9.1 Optimeringer af test for kollision imellem to OBBer

Vi minder om at vores kollisionstest er som følger (vi skriver her den diskrete version). For alle akser h i mængden

$$\{e_i, f_j, e_i \% f_j \, 1 \leq i \leq 3, 1 \leq j \leq 3\},$$

undersøges uligheden

$$|h \cdot (d_1 - d_2)| > \sum_{i=1}^{3} a_i |a \cdot e_i| + \sum_{i=1}^{3} b_i |a \cdot f_i|.$$

Vi ser at når $h = e_1$ så udregnes værdierne $a_1|e_1 \cdot e_1| = a_i$ samt $a_2|e_1 \cdot e_2| = a_3|e_1 \cdot e_3| = 0$. Disse udregninger kan i vores implementation udelades når vi benytter disse identiteter. Vi kan gøre dette for alle akser e_i og f_i for $1 \leq i \leq 3$.

Som det ses i Kapitel 6 er testen imellem omringende kugler væsentlig simplere og hurtigere end testen imellem omringende OBBer. Vi vælger derfor at knytte både en OBB og en kugle til vores objekter: Først tester vi for kollision imellem kugler og kun imellem de tilsvarende OBBer hvis kuglerne kolliderer.

9.2 Optimeringer af gennemløbning af træ af omringende volumener

- Antag at vi leder efter kollisioner i tidsintervallet $[t_0, t_1]$ og at vi i vores gennemløb af to givne OBB-træer har fundet en kollision af geometri til tiden t. Det er klart at vi ønsker at finde den tidligst mulige kollision imellem geometri, derfor er kollisioner efter tiden t ikke relevante. Fremtidige test imellem både omringende volumener og geometri kan derfor optimeres ved kun at teste tidsintervallet $[t_0, t]$.

- Denne optimering er mulig fordi vi som ovenfor kan se bort fra kollisioner der sker efter den nuværende tidligste fundet kollisionstid t. Optimeringen er som følger: Antag at vi i vores gennemløbning af vores OBB-træ er givet to kolliderende OBBer A og B. Antag at A skal splittes idet den har større volume end B. Vi vælger her at tilføje det barn a_i af A hvis centrum er tættest centrummet af B forrest i vores kø Q (jf. tidligere pseudokode). Vi gør dette fordi a_i har størst mulighed for også at kollidere med B. Dette betyder at vi sandsynligvis hurtigt vil kunne finde et tidspunkt hvor geometri kolliderer og herved få indskrænket det tidsinterval vi skal undersøge.

- Da vores kode under opsplitning af A tilføjer børnene a_i én efter én vælger vi at sortere listen efter afstand fra det pågældende barn til B. Dette kræver ikke meget ekstra tid idet vi kan indsætte de nye tubler (a_i, B) i

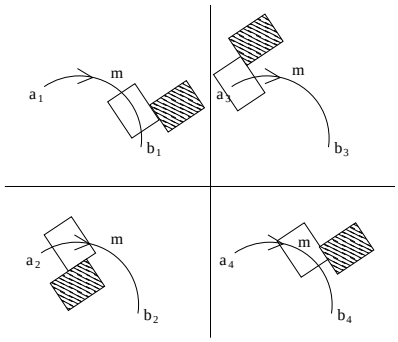
konstant tid. Denne sortering er blot en viderudvikling af optimeringen ovenfor.

- Antag at vi søger kollisioner i tidsrummet $[t_0, t_1]$. Antag at to OBBer A og B kolliderer til tid t. Det følger umiddelbart at hvis geometrien i A og B skal kollidere skal dette ske efter tid t, dette skyldes at geometrien er omringet af enten A eller B. Når vi tester for kollision imellem geometri eller omringende volumener senere *i samme gren som A hhv. B* kan vi derfor indskrænke os til intervallet $[t, t_1]$. Med samme gren menes at der skal være tale om børn af A og B, den nedre grænse er unik per sæt (A, B) af omringende volumener: for to andre OBBer A' og B' kan vi ikke vide om geometrien kolliderer før t.

9.3 Optimeringer af imellem-bevægelser

- Afgrænsning af imellem-bevægelser sker tit i vores program: Hver gang to omringende bokse eller to geometrier skal tjekkes for kollision i tidsrummet $[t_0, t_1]$, så skal de tilsvarende bevægelser afgrænses i samme tidsrum. I forbindelse med træer af OBBer kan der forekomme mange tusinder af sådanne tests, og derfor mange tusinde afgrænsninger.

  Generelt kan vi ikke vide noget om de tidsintervaller der testes imod, dvs. vi kan ikke vide hvor store de er og hvor de to endepunkter ligger. Antag f.eks. at den samme bevægelse m bruges til at interpolere positionen af en boks O_1 imellem fire sæt af kendte punkter $(a_i, b_i), 1 \leq i \leq 4$ i tidsrummet $[t_0, t_1]$. Som det ses nedenfor i Figur 12 kan evt. kollisioner imellem O_1 og en anden boks O_2 i de fire tilfælde ske på vidt forskellige tidspunkter. På figuren rammer de to bokse O_1 og O_2 hinanden på fire



Figur 12: Den samme bevægelse m bruges til at interpolere imellem fire kendte sæt af positioner, (a_i, b_i). Bevægelsen af den hvide boks O_1 interpoleres vha. m, positionen af den skraverede boks O_2 interpoleres af en anden bevægelse (som ikke behøver at være den samme for alle fire tilfælde).

forskellige tidspunkter $t_i, 1 \leq i \leq 4$. Grundet optimeringen ovenfor vil test

for kollisioner af børn af O_1 og O_2 ske i tidsrummene $[t_i, t_1], 1 \leq i \leq 4$. Disse fire intervaller har intet tilfældes udover det ene endepunkt.

Det er altså ikke muligt én gang for alle at udregne samtlige afgrænsninger der skal bruges i hele det pågældende programs levetid.

Lad os nu optimere dette så vi kan nøjes med en mængde afgrænsninger der kan udregnes én gang og herefter genbruges. Optimeringen går ud på at opdele det givne tidsinterval i et forudbestemt antal mindre intervaller og kun benytte disse til at afgrænse med.

Vi lader et tal n angive hvor fint vi vil inddele vores tidsinterval som vi uden tab af generalitet vil være $[0, 1]$.

Nu opdeler vi vores interval $[0, 1]$ i 2^n lige store dele af formen

$$[i/2^n, (i+1)/2^n], \quad 0 \leq i \leq 2^n - 1.$$

For hver af disse intervaller afgrænses m, dette sker én gang. Bemærk at vi ikke afgrænser for mindre eller større intervaller.

Lad os nu antage at vi skal teste for kollision imellem O_1 og O_2 i et givent interval $[t_0, t_1]$. Lad positionen af O_1 være givet ved m_1 og positionen af O_2 være givet ved m_2. Vi finder i_0 så $t_0 \in [i_0/2^n, (i_0+1)/2^n]$. På samme måde lader vi $i_1 \geq i_0$ være det indeks hvor $t_1 \in [i_1/2^n, (i_1+1)/2^n]$.

For at teste imellem kollision lader vi nu i løbe fra i_0 til i_1. For alle intervaller $[i/2^n, (i+1)/2^n]$ har vi tidligere udregnet afgrænsningen af m_1 og vi benytter denne til at afgrænse O_1. Tilsvarende kan vi benytte den tidligere afgrænsning af m_2 til at afgrænse O_2. Nu kan vi teste for kollision imellem O_1 og O_2 i vores tidsintervaller $[i/2^n, (i+1)/2^n], i_0 \leq i \leq i_1$.

Det er klart at vores intervaller ikke altid passer præcist til t_0 og t_1, f.eks. hvis $t_1 - t_0 = 1/4^n$. Vores metode kan derfor medføre flere kollisioner (idet vi tester over større tidsintervaller) end hvis man brugte de korrekte tidsintervaller . I vores implementation har optimeringen dog alligevel hjulpet på udførselstiden.

Vi bemærker at vi kun kan teste for kollisioner i tidsintervaller af længde $1/2^n$. Antag f.eks. to OBBer slet ikke rammer hinanden i tidsintervallet $[0, 1]$, her ville man hurtigt kunne sortere de to OBBer fra ved først at teste *hele* intervallet $[0, 1]$. Vi er med vores optimering nødt til at teste samtlige 2^n intervaller. Afhængigt af det overordnede tidsintervals størrelse og de indgående objekter og imellem-bevægelse har det for os været muligt at finde passende n så optimeringen overordnet set har hjulpet på hastigheden af vores implementation.

- I denne rapport og [10] benyttes matricer i imellem-bevægelser. Vi ser i den forrige optimering at afgrænsningerne af vores imellem-bevægelser kun skal udregnes én gang. Det er derfor ikke relevant hvor hurtigt vi er i stand til at afgrænse. Derfor kan vi vælge en vilkårlig teknik til at interpolere vores positioner, deres tidskompleksitet kan ignoreres under

forudsætning af at det er tilladt at programmet har længere opstartstid end ved matricer.

Vi vælger at bruge quaternioner istedet for matricer til at interpolere orienteringer af objekter. Quaternioner er pladsmæssigt mindre end matricer (en afgrænset matrix er i vores program $3 \cdot 3 \cdot 2 = 18$ doubles hvorimod en afgrænset quaternion er $4 \cdot 2 = 8$ doubles). Den største grund til at vi vælger at bruge quaternioner er dog fordi de nemt kan bruges til at interpolere imellem to kendte orienteringer: Antag at vi givet to orienteringer q_1 og q_2 hvor vi ønsker at interpolere fra q_1 til q_2 over tidsintervallet $[0, 1]$. Dette kan gøres ved [3]

$$q(t) = \frac{\sin(1-t)\omega}{\sin\omega} q_1 + \frac{\sin t\omega}{\sin\omega} q_2, \quad t \in [0, 1],$$

hvor $\omega = \cos^{-1}(q_1 q_2)$. Denne form for interpolering kaldes slerp (engelsk spherical linear interpolation). Interpoleringen vil altid være "pæn" idet orienteringen ikke vil hakke grundet f.eks. afrundningsfejl.

9.4 Optimeringer ved hjælp af programmet gprof

Gprof [6] er et værktøj der tilbyder profilering af et givent program. Vi viser nedenfor de øverste linjer af resultatet af gprof på et af vores testprogrammer (se Kapitel 10).

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call name
 24.98     86.77    86.77 86949434     0.00     0.00
OpenTissue::vector3<OpenTissue::interval<double>>::operator%
(OpenTissue::vector3<OpenTissue::interval<double> > const&) const
```

Outputtet skal læses som følger: Operatoren der udregner krydsproduktet imellem to vektorer med interval elementer bruger ca. 25% af den samlede tid programmet har kørt. Tiden "cumulative seconds" er hvor lang tid operatoren og de funktioner der er angivet over operatoren har brugt. Tiden "self seconds" er hvor lang tid operatoren selv har benyttet. De to tider er ens fordi operatoren er den første funktion i listen. Tallet "calls" er antallet af gange operatoren er blevet kaldt. "self s/call" angiver hvor mange millisekunder operatoren bruger per kald, "total s/call" er hvor mange millisekunder operatoren bruger plus hvor mange millisekunder funktioner kaldt af operatoren bruger. Tiderne er her nul idet operatoren er hurtigere end 1/100 millisekunder.

Vi har draget stor nytte af gprof i forbindelse med intervaller. F.eks. fandt vi at intervaller generelt er det langsommeste i vores program. Ved at implementere vores egen optimerede interval klasse istedet for boosts opnåede vi et radikalt speedup. Vores intervalklasse inklusiv optimeringer kan ses i Bilag C.

9.5 Fremtidige optimeringer

I vores implementation af test imellem geometri tester vi som bekendt typerne (E, E) og (V, F). Vores geometri er trekanter og vi skal derfor teste i alt $3^2 = 9$ kollisioner af typen (E, E). Vi skal også teste om en eller flere af de 3 punkter af den ene trekant ligger i den anden trekant og omvendt, samlet er dette 6 tests.

Vores nuværende implementation af dette tester hver af disse potentielle kollisioner *en af gangen*, dvs. vi undersøger det givne tidsinterval i alt 15 gange.

En optimering er at undersøge tidsintervallet én gang mens samtlige 15 kollisionstests tjekkes. Vi kan desuden benytte os af "Improved Newton Interval Method" som beskrevet i [11]. Ideen er som følger.

Antag at vi vil finde et nulpunkt for en funktion f på intervallet $[t_0, t_1]$ (dette er hvad vi ønsker i Kapitel 7.3). Antag at afgrænsningen af f på et interval har endepunkter med forskellige fortegn (dvs. at der er potentielle for et nulpunkt).

Lad $[a, b]$ være et interval der afgrænser f' på tidsintervallet $[t_0, t_1]$. Hvis $ab > 0$ så kan vi indskrænke det interval hvor f måske antager værdien nul til intervallet

$$\left([m, m] - \frac{[f(m), f(m)]}{[a, b]}\right) \cap [t_0, t_1],$$

hvor $m \in [t_0, t_1]$. Kravet $ab > 0$ siger at der kun er potientielle for ét nulpunkt for f.

Ideen med metoden ovenfor er at vi hurtigt kan pejle os ind på hvornår f antager værdien nul. Hvis f.eks. f antager værdien nul til tiden t og t er sidst i tidsintervallet $[t_0, t_1]$ (altså tæt på t_1) kan vi relativt hurtigt sortere tidsrummet $[t_0, t[$ fra.

10 Afprøvning

Vi vil i dette afsnit foretage en "black-box" afprøvning af vores implementation af kontinuerligt kollisionsdetektering. Med "black-box" menes at vi vil undersøge om det *overordnede* resultat af en kollisionstest er korrekt. Vi vil altså ikke kigge specifikt på enkelte funktioner af vores program under afprøvning.

Vi har i vores afgrænsninger valgt ikke at garantere at alle "rigtige" kollisioner imellem to givne objekter skal findes. Med "rigtige" menes de kollisioner der forekommer hvis objekterne bevæges langs deres korrekte bane. Vi minder om at dette skyldes at vores imellem-bevægelser er en erstatning for disse ukendte rigtige baner og derfor ikke nødvendigvis bevæger objekterne langs den "rigtige" bane.

I vores afgrænsninger har vi også valgt ikke at garantere at finde kollisioner imellem to objekter hvor det ene er fuldstændigt indeholdt i det andet og hvor objekterne ikke rører hinanden.

I vores afprøvning vil vi først afprøve vores imellem-bevægelse, dvs. vi vil teste at den interpolerer korrekt. Under antagelse af at imellem-bevægelsen er korrekt ved vi nu at vores geometri og omringende volumener vil blive bevæget korrekt. Herefter vil vi afprøve vores kollisionstest imellem to OBBer mod

hinanden. Når dette er gjort afprøver vi vores kollisionstest imellem geometrier. Til sidst afprøver vi vores kollisionstest imellem træer af OBBer.

10.1 Afprøvning af imellem-bevægelse

Lad imellem-bevægelsen m være givet. Antag at m skal interpolere imellem de to punkter p_0 og p_1 over tiden $[0, 1]$. Vi afprøver m ved at bede om den interpolerede position (en quaternion og en vektor) til tiderne $i/8, 0 \leq i \leq 8$. Disse positioner udskrives på skærmen og vi verificerer at de er korrekte.

Lad os skrive quaternioner på formen (w, x, y, z) hvor (x, y, z) er rotationsaksen og w er rotationen. Vi tester følgende situationer

- AB00. $p_0 = p_1$.

  Dette anses for et specieltilfælde og afprøves. Samtlige positioner skal være lig $p_0 = p_1$.

- AB01. p_0 er vektoren $(0, 0, 0)$ og orienteringen er identiteten (quaternionen $(1, 0, 0, 0)$). p_1 er vektoren $(10, 0, 0)$ og samme orientering som til tiden t_0.

  Her testes translation. De udskrevne vektorer skal indeholde $x-$værdier som tilsammen er en strengt voksende følge endende med værdien 10, $y-$ og $z-$værdierne skal forblive nul. Orienteringen skal til alle tider være lig identiteten.

- AB02. p_0 er vektoren $(0, 0, 0)$ og orientering er identiteten. p_1 er vektoren $(0, 0, 0)$ og orienteringen $(0, 1, 0, 0)$ (som er rotationen 180^o omkring $x-$aksen).

  Her testes rotation. De udskrevne translationer skal alle være nul-vektoren. Orienteringen skal løbe fra identiteten og til $(0, 1, 0, 0)$.

10.2 Afprøvning af kollisionstest imellem to OBBer

Lad to OBBer O_1 og O_2 være givet. Lad m_1 og m_2 være de tilhørende imellembevægelser og lad tidsintervallet hvor der skal tjekkes for kollision være $[t_0, t_1]$.

- AO00. O_1 og O_2 er adskilt til tiden t_0. Vi lader m_1 og m_2 være givet så O_1 og O_2 ikke rammer hinanden i hele tidsintervallet. Vores kollisionstest må ikke rapportere nogle kollision.

- AO01. O_1 og O_2 er adskilt til tiden t_0. Vi lader m_1 og m_2 være givet så O_1 og O_2 rammer hinanden i tidsintervallet. Vores kollisionstest skal rapportere en kollision.

- AO02. O_1 og O_2 rammer hinanden til tiden t_0. Vores kollisionstest skal rapportere en kollision.

10.3 Afprøvning af kollisionstest imellem to geometrier

Som nævnt tidligere tester vi for to typer af kollisioner; (E, E) og (V, F). Vi tester de to typer hver for sig og opstiller følgende testsituationer

- Lad k_1 og k_2 være to kanter. Lad m_1 og m_2 være de tilhørende bevægelser.
  - AE00. k_1 og k_2 rammer ikke hinanden til tid t_0. m_1 og m_2 er angivet så k_1 og k_2 ikke rammer hinanden i tidsintervallet $[t_0, t_1]$. Vores test skal ikke rapportere en kollision.
  - AE01. k_1 og k_2 rammer ikke hinanden til tid t_0. m_1 og m_2 er angivet så k_1 og k_2 rammer hinanden i tidsintervallet $[t_0, t_1]$. Vores test skal rapportere en kollision.
  - AE02. k_1 og k_2 rammer hinanden til tid t_0. Vores test skal rapportere en kollision.
- Lad f være en flade og v være et punkt. Lad m_1 og m_2 være de tilhørende bevægelser.
  - AVF00. f og v rammer ikke hinanden til tiden t_0. m_1 og m_2 er angivet så f og v ikke rammer hinanden i tidsintervallet $[t_0, t_1]$. Vores test skal ikke rapportere en kollision.
  - AVF01. f og v rammer ikke hinanden til tiden t_0. m_1 og m_2 er angivet så f og v rammer hinanden i tidsintervallet $[t_0, t_1]$. Vores test skal rapportere en kollision.
  - AVF02. f og v rammer hinanden til tiden t_0. Vores test skal rapportere en kollision.

10.4 Afprøvning af kollisionstest imellem to træer indeholdende OBBer

Vi tester nu samme opstilling som ved afprøvningen af to OBBer, her benyttes blot træer indeholdende OBBer.

Lad T_1 og T_2 være to træer indeholdende OBBer. Lad m_1 og m_2 være de tilhørende bevægelser. Vi afprøver nu følgende

- AT01. T_1 og T_2 rammer ikke hinanden til tiden t_0. m_1 og m_2 er angivet så T_1 og T_2 ikke rammer hinanden i tidsintervallet $[t_0, t_1]$. Vores test skal ikke rapportere en kollision.
- AT02. T_1 og T_2 rammer ikke hinanden til tiden t_0. m_1 og m_2 er angivet så T_1 og T_2 rammer hinanden i tidsintervallet $[t_0, t_1]$. Vores test skal rapportere en kollision.
- AT03. T_1 og T_2 rammer hinanden til tiden t_0. Vores test skal rapportere en kollision.

Samtlige testprogrammer er kompileret med GCC version

`g++-4.0 (GCC) 4.0.3 (Ubuntu 4.0.3-1ubuntu5)`,

og med parameterne `-O3 -Wall -Werror -march=pentium4`. Testprogrammerne er tilgængelige i forfatterens DIKU-hjemmemappe

`/home/disk17/jackj/contcoll/afproevning`.

Resultaterne af samtlige afprøvninger kan ses i Bilag B. Testprogrammerne kan ses i Bilag A.

11 Konklusion

Vi har i denne rapport nu gennemgået den metode Stephane Redon beskriver i hans artikel [10]. Overordnet synes vi at brugen af interval aritmetik til at afgrænse med er en meget intuitiv og nem måde at angribe kontinuert kollisiondsdetektering på.

Det har været yderst interresant og lærerigt at skrive denne rapport samt at implementere vores løsning. Vi har f.eks. lært en masse omkring computergrafik i gennemgangen af metoden og implementationen af den. F.eks. har vi lært om quaternioner, interval aritmetik, OBBer, gprof, C++ biblioteket boost og i særdeleshed hvordan OpenTissue skal bruges. OpenTissue har været et nyttigt redskab i vores implementation - selvom vi ikke vil ligge skjul på at templates kan være svære at forstå og bruge når man første gang stifter bekendtskab med dem.

Vi har som nævnt i Kapitel 9 brugt lang tid på at optimere vores kode. De færdige køretider er væsentligt bedre end ved de første kørsler af vores programmer, men der er stadig langt til de tider der kort nævnes i f.eks. [9].

Vi ville gerne havde haft tid til at teste vores implementation imod andre implementationer, f.eks. BULLET biblioteket der er tilgængeligt på hjemmesiden `http://www.continuousphysics.com/Bullet/`. Vi ville også gerne havde haft tid til at teste vores metode i en simulation. Det er muligt at vi gør dette på et senere tidspunkt, det er yderst interresant at se hvordan vores metode klarer sig hastighedsmæssigt imod andre implementationer nu hvor vi har brugt lang tid på optimeringer.

Det er vores forhåbning at vi, efter denne rapport er afleveret, vil kunne optimere yderligere på vores implemenentation: Det ville være meget tilfredsstillende hvis andre senere kan drage nytte af vores arbejde, f.eks. ved at vores kode inkluderes i OpenTissue.

12 Bilag A, Testprogrammer

Samtlige testprogrammer benytter følgende skabelon. I vores testprogrammer nedenfor nøjes vi derfor med at vise vores

```
cApplication::init()
```

funktion idet det er her vi afprøver vores testcases.

12.1 main.cpp

```
/*
* Basic GL stuff.
*
* Jackj.
*/
#include <iostream>
#include <fstream>
#include <OpenTissue/utility/GL/gl_util.h>
#include <OpenTissue/image/image.h>
#include <OpenTissue/image/util/screen_capture.h>
#include <OpenTissue/image/io/image_write.h>
#include "application.hpp"


float transx,transy, oldTransY, oldTransX;
int rotatex,rotatey, oldRotateY, oldRotateX;

bool mouseLeftDown, mouseMiddleDown, drawPath;
bool f1down;
int beginMoveLeftY, beginMoveLeftX, beginMoveMiddleY, beginMoveMiddleX;
float wscale;

GLfloat specularColor[3] = {1.0f, 1.0f, 1.0f};
GLfloat shine = 100.0f;
GLfloat center_lightposition[] = {0.0, 0.0, 0.0, 1.0f};
GLfloat extra_lightposition[] = {50.0, 50.0, 50.0, 1.0f};

bool animate = false;
bool saveScreen = false;
int screenNr = 0;

using namespace OpenTissue;

// The actual application
cApplication theApp;

void UpdateScreen() {
  glClearColor(1,1,1,0);
  glClear(GL_DEPTH_BUFFER_BIT);
  glClear(GL_COLOR_BUFFER_BIT);

  glLoadIdentity();

  //rotate the coordinate space for viewing from the (-, +, +) octant
  glRotatef(45.0f, 1.0f, 0.0f, 0.0f);

  //rotate the coordinate space for viewing from the (-, +, +) octant
  glRotatef(45.0f, 0.0f, 1.0f, 0.0f);

  //translate out into coordinate space for viewing from the (-, +, +) octant
  glTranslatef(20.0f, -30.0f, -20.0f);

  glTranslatef(transx, transy, 0);
  glRotatef(rotatey, 1, 0, 0);
  glRotatef(rotatex, 0, 1, 0);

  // Draw x,y,z axis
  glDisable(GL_TEXTURE_2D);
  glEnable(GL_LIGHTING);
```

```
    glColor3f(0xff, 0, 0);
    glBegin(GL_LINES);
    glVertex3f(0,0,0);
    glVertex3f(100,0,0);
    glEnd();
    glColor3f(0, 0xff, 0);
    glBegin(GL_LINES);
    glVertex3f(0,0,0);
    glVertex3f(0,100,0);
    glEnd();
    glColor3f(0, 0, 0xff);
    glBegin(GL_LINES);
    glVertex3f(0,0,0);
    glVertex3f(0,0,100);
    glEnd();

    glScalef(wscale,wscale,wscale);
    theApp.Draw();
    if (saveScreen)
    {

      image_write("screen.png", *screen_capture() );

      saveScreen = false;
    }
    glFinish();
    glutSwapBuffers();
}

void ResizeWindow(int w, int h)
{
  if (w == 0 || h == 0) return;  //Nothing is visible then, so return

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0,(GLdouble)w/(GLdouble)h,0.5,400.0);

  glMatrixMode(GL_MODELVIEW);
  glViewport(0,0,w,h);  //Use the whole window for rendering
}

void UpdateMouseButton(int button, int state, int x, int y) {

        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
    mouseLeftDown = true;
    beginMoveLeftX = x; beginMoveLeftY = y;
    oldTransX = transx; oldTransY = transy;
  }
  if (button == GLUT_LEFT_BUTTON && state == GLUT_UP)
    mouseLeftDown = false;

  if (button == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) {
    mouseMiddleDown = true;
    beginMoveMiddleX = x; beginMoveMiddleY = y;
    oldRotateX = rotatex; oldRotateY = rotatey;
  }
  if (button == GLUT_MIDDLE_BUTTON && state == GLUT_UP)
    mouseMiddleDown = false;

  if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN
      && f1down) {
    wscale *= 0.5;
    UpdateScreen();
  }
  if (button == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN
      && f1down) {
    wscale *= 1.5;
    UpdateScreen();
  }
}

void UpdateMouseMotion(int x, int y) {
  if (mouseMiddleDown) {
    rotatex = (oldRotateX + x-beginMoveMiddleX) % 360;
    rotatey = (oldRotateY + y-beginMoveMiddleY) % 360;
    UpdateScreen();
```

```
  }

  if (mouseLeftDown) {
    transx = oldTransX + (x-beginMoveLeftX) / 40.0;
    transy = oldTransY - (y-beginMoveLeftY) / 20.0;
    UpdateScreen();
  }
}

void Loop() {
  theApp.Run();
  glutPostRedisplay();

  // FIXME: -time spent on drawing, obviously
#ifdef WIN32
  Sleep(1000/theApp.fps);
#else
  usleep(1000000/theApp.fps);
#endif
}

void Key(unsigned char key, int /*x*/, int /*y*/) {
  switch (key) {
      case 'q':
      case 27:
          glutLeaveGameMode();
          exit(0);
          break;
        case ' ':
          animate = !animate;
          if (animate)
            glutIdleFunc(Loop);
          else
            glutIdleFunc(NULL);
          break;
        case 'y':
          saveScreen = true;
          break;
        default:
          theApp.MenuAction(key);
  }
}

void KeySDown(int key, int /*x*/, int /*y*/) {
    if (key == GLUT_KEY_F1)
        f1down = true;
}

void KeySUp(int key, int /*x*/, int /*y*/) {
    if (key == GLUT_KEY_F1)
        f1down = false;
}


void Menu(int entry) {
  Key(entry, 0, 0);
}

int main(int argc, char **argv)
{

  glutInit(&argc, argv);
  if (!theApp.Init()) {
      std::cerr << "Error initializing application." << std::endl;
      return -1;
  }

  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutGameModeString("1024x768");
  glutEnterGameMode();

  int mmenu = glutCreateMenu(Menu);
  int toggles = glutCreateMenu(Menu);
  glutAddMenuEntry("Animation [space]", ' ');
  glutAddMenuEntry("Screen capture [y]", 'y');
  theApp.AddMenu(mmenu, Menu);
```

```
    glutSetMenu(mmenu);
    glutAddSubMenu("Toggles", toggles);
    glutAddMenuEntry("Quit [esc]", 27);
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    glutDisplayFunc(UpdateScreen);
    glutMouseFunc(UpdateMouseButton);
    glutMotionFunc(UpdateMouseMotion);
    glutReshapeFunc(ResizeWindow);
    glutKeyboardFunc(Key);
    glutSpecialUpFunc(KeySUp);
    glutSpecialFunc(KeySDown);

    glShadeModel(GL_SMOOTH);

    glEnable(GL_DEPTH_TEST);
    glFrontFace(GL_CCW);          // Counter clock-wise polygons face out
    glEnable(GL_CULL_FACE);       // Do not calculate inside of objects
    glEnable(GL_NORMALIZE);

    float ambience[4] = {0.3f, 0.3f, 0.3f, 1.0};
    float diffuse[4] = {0.9f, 0.9f, 0.9f, 1.0};
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambience);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, center_lightposition);
    glLightfv(GL_LIGHT1, GL_POSITION, extra_lightposition);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHTING);
    glEnable(GL_COLOR_MATERIAL);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specularColor);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, &shine);

    transy = transx = 0;
    rotatey = rotatex = 0;
    wscale=5.0;
    f1down = false;
    mouseLeftDown = mouseMiddleDown = false;

    glutMainLoop();
    return 0;
}
```

12.2 application.cpp

```
/*
 * Example application.
 *
 * Jackj.
 */
#include <string>
#include <OpenTissue/utility/GL/gl_util.h>
#include <OpenTissue/utility/high_res_timer.h>
#include <OpenTissue/math/constants.h>
#include <OpenTissue/mesh/mesh.h>
#include <OpenTissue/mesh/common/util/mesh_drawing.h>
#include <OpenTissue/collision/bvh/util/top_down_constructor/bvh_top_down_constructor.h>
#include <OpenTissue/collision/bvh/util/bvh_get_nodes_at_height.h>
#include <OpenTissue/math/matrix.h>
#include <OpenTissue/math/coordsys.h>
#include <OpenTissue/math/vector.h>
#include <OpenTissue/math/rotation.h>
#include "collision/continuous/arbitrary_constant.h"
#include "collision/continuous/cont_coll_edge.h"
#include "application.hpp"
#include "obb.h"

using namespace OpenTissue;
using namespace std;

cApplication::cApplication()
{
    mesh = false;
```

```
    drawBV = false;
    bvLevel=0;
    test = 'o';
    anim_time = 0.0;
}

cApplication::~cApplication() {}

bool cApplication::Init()
{
}

void cApplication::draw_point_face(double time)
{
    coordsys<double> c = e1_motion.pos(time);
    rotation<double> r = c.Q();

    glPushMatrix();
    glTranslatef(c.T()[0], c.T()[1], c.T()[2]);
    glRotatef(r.angle()*OT_M_RADIAN, r.axis()[0], r.axis()[1],
            r.axis()[2]);
    glTranslatef(e1a[0], e1a[1], e1a[2]);
    glutSolidSphere(0.2,10,10);
    glPopMatrix();

    c = f1_motion.pos(time);
    r = c.Q();
    glPushMatrix();
    glTranslatef(c.T()[0], c.T()[1], c.T()[2]);
    glRotatef(r.angle()*OT_M_RADIAN, r.axis()[0], r.axis()[1],
            r.axis()[2]);
    glBegin(GL_LINE_LOOP);
    glVertex3f(f1a[0], f1a[1], f1a[2]);
    glVertex3f(f1b[0], f1b[1], f1b[2]);
    glVertex3f(f1c[0], f1c[1], f1c[2]);
    glEnd();
    glPopMatrix();
}

void cApplication::draw_edges(double time)
{
    coordsys<double> c = e1_motion.pos(time);
    rotation<double> r = c.Q();

    glPushMatrix();
    glTranslatef(c.T()[0], c.T()[1], c.T()[2]);
    glRotatef(r.angle()*OT_M_RADIAN, r.axis()[0], r.axis()[1],
            r.axis()[2]);
    glBegin(GL_LINES);
    glVertex3f(e1a[0], e1a[1], e1a[2]);
    glVertex3f(e1b[0], e1b[1], e1b[2]);
    glEnd();
    glPopMatrix();

    c = e2_motion.pos(time);
    r = c.Q();
    glPushMatrix();
    glTranslatef(c.T()[0], c.T()[1], c.T()[2]);
    glRotatef(r.angle()*OT_M_RADIAN, r.axis()[0], r.axis()[1],
            r.axis()[2]);
    glBegin(GL_LINES);
    glVertex3f(e2a[0], e2a[1], e2a[2]);
    glVertex3f(e2b[0], e2b[1], e2b[2]);
    glEnd();
    glPopMatrix();
}

void cApplication::print_string(int x, int y, char *s)
{
    glRasterPos2f(x, y);
    while (*s)
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *s++);
}

void cApplication::draw_obbs(double time, bool drawcollobbs)
{
    if (mesh)
```

```
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        else
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

110     coordsys<double> c = w_motion.pos(time);
        rotation<double> r = c.Q();

        glPushMatrix();
        glTranslatef(c.T()[0], c.T()[1], c.T()[2]);
115     glRotatef(r.angle()*OT_M_RADIAN, r.axis()[0], r.axis()[1],
                r.axis()[2]);
        glDrawMesh(wMesh, GL_POLYGON, false, true, false);

        float curcol[4];
120
        glGetFloatv(GL_CURRENT_COLOR, curcol);

        if (drawBV) {
            bvh_type::bv_ptr_container nodes;
125         bvh_get_nodes_at_height(bBVH,bvLevel,nodes);
            glColorPicker(1,0,0);
            bvh_type::bv_iterator node = nodes.begin();
            bvh_type::bv_iterator end = nodes.end();
            for (;node!=end;++node )
130             node->volume().draw( GL_POLYGON );
        }

        glPopMatrix();
        glColor4f(curcol[0], curcol[1], curcol[2], curcol[2]);
135
        c = b_motion.pos(time);
        r = c.Q();
        glPushMatrix();
        glTranslatef(c.T()[0], c.T()[1], c.T()[2]);
140     glRotatef(r.angle()*OT_M_RADIAN, r.axis()[0], r.axis()[1],
                r.axis()[2]);
        glDrawMesh(bMesh, GL_POLYGON, false, true, false);

        glGetFloatv(GL_CURRENT_COLOR, curcol);
145
        if (drawBV) {
            bvh_type::bv_ptr_container nodes;
            bvh_get_nodes_at_height(wBVH,bvLevel,nodes);
            glColorPicker(1,0,0);
150         bvh_type::bv_iterator node = nodes.begin();
            bvh_type::bv_iterator end = nodes.end();
            for (;node!=end;++node )
                node->volume().draw( GL_POLYGON );
        }
155
        glPopMatrix();
        glColor4f(curcol[0], curcol[1], curcol[2], curcol[3]);
    }

160 void cApplication::Draw()
    {
        bool tempM;
        bool temp = drawBV;

165
        switch(test) {
            case 'e': // EDGE/EDGE
                tempM = mesh;
                mesh = false;
170             // start pos
                glColor3f(0.0,1.0,0.0);
                draw_edges(0);

                // end pos
175             glColor3f(1.0,0.0,0.0);
                draw_edges(1);

                // first collision time
                glColor3f(0.0, 1.0, 1.0);
180             draw_edges(edgecolltime);
```

```
                // current pos
                mesh = tempM;
                glColor3f(0.0, 0.0, 1.0);
                draw_edges(anim_time);
                break;
            case 'o': // OBB/OBB

                tempM = mesh;
                mesh = true; drawBV = false;

                // start pos
                glColor3f(0.0, 1.0, 0.0);
                draw_obbs(0);

                // end pos
                glColor3f(1.0, 0.0, 0.0);
                draw_obbs(1);

                // collision time
                glColor3f(1.0, 1.0, 0.0);
                draw_obbs(obbcolltime, true);

                // current pos
                drawBV = temp;
                mesh = tempM;
                glColor3f(0.0, 0.0, 1.0);
                draw_wall_bullet(anim_time);

                break;
            case 'p': // FACE/POINT

                // start pos
                glColor3f(0.0, 1.0, 0.0);
                draw_point_face(0.0);

                // end pos
                glColor3f(1.0, 0.0, 0.0);
                draw_point_face(1.0);

                // collision time
                glColor3f(1.0, 1.0, 0.0);
                draw_point_face(facepointcolltime);

                // current pos
                glColor3f(0.0, 0.0, 1.0);
                draw_point_face(anim_time);

                break;

            default:
                break;
        }
    }

    void cApplication::AddMenu(int mmenu, void (*Menu)(int))
    {
    }

    void cApplication::MenuAction(int entry)
    {
        switch (entry) {
            case 'd':
                drawBV = !drawBV;
                glutPostRedisplay();
                break;
            case '+':
                bvLevel++;
                glutPostRedisplay();
                break;
            case '-':
                bvLevel--;
                if (bvLevel<0) bvLevel=0;
                glutPostRedisplay();
                break;
            case 'm':
                mesh = !mesh;
```

```
                glutPostRedisplay();
                break;
            case 'e':
            case 'o':
            case 'p':
                test = entry;
                anim_time = 0.0;
                glutPostRedisplay();
                break;
            default:
                break;
        }
}


void cApplication::Run()
{
    anim_time += 0.01;
    if (anim_time > 1.0) anim_time = 0.0;
}
```

12.3 application.h

```
#ifndef __CCAPPLICATION_HPP__
#define __CCAPPLICATION_HPP__

/*
 * Application class.
 *
 * Jackj.
 */

#include <stdarg.h>
#include <stdio.h>
#include <OpenTissue/math/vector.h>
#include <OpenTissue/mesh/mesh.h>
#include <OpenTissue/geometry/obb.h>
#include <OpenTissue/geometry/sphere.h>
#include <OpenTissue/collision/bvh/bvh.h>
#include <OpenTissue/collision/bvh/util/top_down_constructor/bvh_top_down_constructor.h>
#define DEFMESHROOT "/home/jackj/src/DataTissue/"
#define DEFWALL "demos/common/mesh/turbine-blades-1200.msh"
#define DEFBULLET "demos/common/mesh/turbine-blades-1200.msh"

typedef OpenTissue::vector3<double>   vector3_type;

// This is needed in our OBB fitting classes
class OBBTreeFaceTraits : public OpenTissue::DefaultFaceTraits
{
    public:

        vector3_type * m_v0;
        vector3_type * m_v1;
        vector3_type * m_v2;
        vector3_type * m_n0;
        vector3_type * m_n1;
        vector3_type * m_n2;

    public:

        OBBTreeFaceTraits()
            : m_v0()
            , m_v1()
            , m_v2()
            , m_n0()
            , m_n1()
            , m_n2()
            {}


};

typedef OpenTissue::PolyMesh<OpenTissue::PolyMeshListKernel,
        OpenTissue::DefaultVertexTraits, OpenTissue::DefaultHalfEdgeTraits,
```

```
    OpenTissue::DefaultEdgeTraits, OBBTreeFaceTraits> mesh_type;
    typedef mesh_type::face_type *        face_ptr_type;
    typedef OpenTissue::OBB<double>       obb_type;
55  typedef mesh_type::face_type          face_type;
    typedef mesh_type::vertex_type        vertex_type;
    typedef mesh_type::halfedge_type      halfedge_type;


60  #include "obb.h"
    #include "collision/continuous/arbitrary_constant.h"
    #include "collision/continuous/cont_coll_edge.h"
    #include "collision/continuous/cont_coll_face.h"
    #include "collision/continuous/cont_coll_obb.h"
65  #include "collision/continuous/cont_coll_bvh.h"
    #include "collision/continuous/cont_coll_bv_traits.h"


    /*
70   * OBB collision policy, used in our BVH code
     */

    class bv_coll;

75  template <typename bvh_type_>
    class obb_coll_policy
    {

        public:
80

            typedef bvh_type_                               bvh_type;
            typedef double                                  real_type;
            typedef std::vector<bv_coll>                    result_type;
            typedef typename bvh_type::bv_type              bv_type;
85          typedef typename bvh_type::bv_ptr               bv_ptr;
            typedef typename bvh_type::annotated_bv_ptr     annotated_bv_ptr;
            typedef typename bvh_type::annotated_bv_type    annotated_bv_type;

        private:
90

            OpenTissue::cont_coll_obb<double, obb_type>
                m_obb_checker;                              // Our OBB/OBB checker
            OpenTissue::cont_coll_edge<double>
                m_edge_edge_checker;                        // Our EDGE/EDGE checker
95          OpenTissue::cont_coll_face_point<double>
                m_face_point_checker;                       // Our FACE/POINT checker
            OpenTissue::arbitrary_constant<double> m_m1; // A's motion
            OpenTissue::arbitrary_constant<double> m_m2; // B's motion

100     public:

            obb_coll_policy() {}

            void set_m1(OpenTissue::arbitrary_constant<double> const &m1)
105         {
                m_m1 = m1;
            }

            void set_m2(OpenTissue::arbitrary_constant<double> const &m2)
110         {
                m_m2 = m2;
            }

            void set_obb_checker(OpenTissue::cont_coll_obb<double,obb_type>
115                 const &obb)
            {
                m_obb_checker = obb;
            }

120         void set_edge_edge_checker(OpenTissue::cont_coll_edge<double> const
                &edgeedge)
            {
                m_edge_edge_checker = edgeedge;
            }
125

            void set_face_point_checker(OpenTissue::cont_coll_face_point<double>
                const &facepoint)
```

```
        {
            m_face_point_checker = facepoint;
130     }

        obb_coll_policy(OpenTissue::cont_coll_obb<double,obb_type> &obb,
                OpenTissue::arbitrary_constant<double> &m1,
                OpenTissue::arbitrary_constant<double> &m2)
135         : m_obb_checker(obb)
            , m_m1(m1)
            , m_m2(m2) {}

        void reset(result_type &r) // policy
140     {
            r.clear();
        }

        void report(bv_ptr A, bv_ptr B, double &t,
145             result_type &r) // policy
        {
            bv_coll coll(&(A->volume()), &(B->volume()), t);
            r.push_back(coll);
        }
150
        bool bv_overlap(bv_ptr const A, bv_ptr const B, double &t0, double &t1) // policy
        {
            real_type t =
                m_obb_checker.check_pair(A, m_m1, B, m_m2, t0, t1);
155
            if (t < -0.5) return false;
            t0 = t;
            return true;
        }
160
        bool geometry_overlap(bv_ptr A, bv_ptr B, double &t0,
                double const &t1)
        {
            annotated_bv_ptr a1 = boost::static_pointer_cast<annotated_bv_type>(A);
165         annotated_bv_ptr a2 = boost::static_pointer_cast<annotated_bv_type>(B);

            face_type *f1 = *(a1->geometry_begin());
            face_type *f2 = *(a2->geometry_begin());

170         OpenTissue::vector3<double> a[3];

            mesh_type::const_face_vertex_circulator v1(*f1), v1end;
            unsigned int i = 0;
            for (;v1!=v1end;++v1,++i) {
175             assert(i <= 2 || !"Only do triangles at the moment");
                a[i].set(v1->m_coord(0), v1->m_coord(1), v1->m_coord(2));
            }
                assert(i == 3 || !"Only do triangles at the moment");

180         OpenTissue::vector3<double> b[3];

            mesh_type::const_face_vertex_circulator v2(*f2), v2end;
            for (i =0;v2!=v1end;++v2,++i) {
                assert(i <= 2 || !"Only do triangles at the moment");
185             b[i].set(v2->m_coord(0), v2->m_coord(1), v2->m_coord(2));
            }
            assert(i == 3 || !"Only do triangles at the moment");

            bool coll=false;
190         real_type earliest_time = t1;
            real_type t;

    #define OT_TEST_EDGE_EDGE(x,y,z,v)\
            t = m_edge_edge_checker.check_interval(a[x], a[y], m_m1,\
195                     b[z], b[v], m_m2, t0, earliest_time);\
            if (t < 4.0) { coll=true; if (t<earliest_time) earliest_time=t; }

            // EDGE/EDGE tests (9 of these)
            OT_TEST_EDGE_EDGE(0,1,  0,1);
200         OT_TEST_EDGE_EDGE(0,1,  1,2);
            OT_TEST_EDGE_EDGE(0,1,  2,1);
            OT_TEST_EDGE_EDGE(1,2,  0,1);
            OT_TEST_EDGE_EDGE(1,2,  1,2);
```

```
            OT_TEST_EDGE_EDGE(1,2,  2,1);
            OT_TEST_EDGE_EDGE(2,1,  0,1);
            OT_TEST_EDGE_EDGE(2,1,  1,2);
            OT_TEST_EDGE_EDGE(2,1,  2,1);

#undef OT_TEST_EDGE_EDGE

#define OT_FACE_POINT_A(x)\
            t = m_face_point_checker.check_face_point(a[x],\
                    m_m1, b[0], b[1], b[2], m_m2, t0, earliest_time);\
            if (t < 4.0) { coll=true; if (t < earliest_time) earliest_time = t; }

            // Check if any of A's corners lie inside face B
            OT_FACE_POINT_A(0);
            OT_FACE_POINT_A(1);
            OT_FACE_POINT_A(2);
#undef OT_FACE_POINT_A

#define OT_FACE_POINT_B(x)\
            t = m_face_point_checker.check_face_point(b[x],\
                    m_m2, a[0], a[1], a[2], m_m1, t0, earliest_time);\
            if (t < 4.0) { coll=true; if (t < earliest_time) earliest_time = t; }

            // Check if any of B's corners lie inside face A
            OT_FACE_POINT_B(0);
            OT_FACE_POINT_B(1);
            OT_FACE_POINT_B(2);
#undef OT_FACE_POINT_B

            if (coll) {
                t0 = earliest_time;
                std::cout << "I found collision at time: "<< t0 <<std::endl;
                return true;
            }

            return false;
        }

};

/*
 * Class to contain OBB collisions
 */
class bv_coll
{

    protected:

    obb_type *m_A, *m_B;
    double m_time;

    public:

    bv_coll(obb_type *A, obb_type *B, double t)
        : m_A(A)
        , m_B(B)
        , m_time(t) {}

    obb_type const *getA()
    {
        return m_A;
    }

    obb_type const *getB()
    {
        return m_B;
    }

    double gettime()
    {
        return m_time;
    }
};

class cApplication {

    public:
```

```
            typedef double                real_type;
            typedef OpenTissue::cont_coll_bv_traits<double> bv_traits;
            typedef OpenTissue::BVH<obb_type, face_ptr_type,bv_traits> bvh_type;
285         typedef OpenTissue::BVHTopDownConstructor<bvh_type,
                    obb_base_tree_top_down_policy<bvh_type, real_type> >    constructor_type;

        public:
            cApplication();
290         ~cApplication();
            bool Init();
            void Draw();
            void Run();
            void MenuAction(int entry);
295         void AddMenu(int, void (*Menu)(int));

            int fps;

        private:
300
            // OBB/OBB stuff
            mesh_type wMesh;
            mesh_type bMesh;
            OpenTissue::arbitrary_constant<double> b_motion;
305         OpenTissue::arbitrary_constant<double> w_motion;
            double obbcolltime;
            unsigned int obbcolls;
            bvh_type bBVH, wBVH;
            void draw_wall_bullet(double time,bool drawcollobbs=false);
310         std::vector<bv_coll> obb_colls;

            // EDGE/EDGE stuff
            OpenTissue::arbitrary_constant<double> e1_motion;
            OpenTissue::arbitrary_constant<double> e2_motion;
315         OpenTissue::vector3<double> e1a, e1b, e2a, e2b;
            OpenTissue::cont_coll_edge<double> edge_check;
            OpenTissue::vector3<double> edgecolln, edgecollp;
            double edgecolltime;
            void draw_edges(double time);
320
            // FACE/POINT stuff
            OpenTissue::vector3<double> f1a, f1b, f1c;
            OpenTissue::arbitrary_constant<double> f1_motion;
            OpenTissue::cont_coll_face_point<double> face_check;
325         double facepointcolltime;
            void draw_point_face(double time);

            // basic stuff
            void print_string(int x, int y, char *s);
330         bool mesh, drawBV;
            int bvLevel;
            char test;
            double anim_time;
    };
335
    #endif
```

12.4 AB00

```
1   bool cApplication::Init()
    {
        matrix3x3<double> wrs = diag(1.0); // Rotation is identity
        matrix3x3<double> wre = diag(1.0); // Rotation is identity
5
        vector3<double> wts(0,0,0);
        vector3<double> wte(0,0,0);
        w_motion.m_p0 = coordsys<double>(wts, wrs);
        w_motion.m_p1 = coordsys<double>(wte, wre);
10      w_motion.init();

        // Print interpolated positions
        double t = 0.;
        for (int i=0;i <= 8; i++) {
15          coordsys<double> c = w_motion.pos(t);
```

```
            std::cout << "Vektor: " << c.T() << " orientering: " <<
                c.Q() << std::endl;
            t += 1. / 8.;
        }

        return true;
    }
```

12.5 AB01

```
bool cApplication::Init()
{
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity

    vector3<double> wts(0,0,0);
    vector3<double> wte(10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Print interpolated positions
    double t = 0.;
    for (int i=0;i <= 8; i++) {
        coordsys<double> c = w_motion.pos(t);
        std::cout << "Vektor: " << c.T() << " orientering: " <<
            c.Q() << std::endl;
        t += 1. / 8.;
    }

    return true;
}
```

12.6 AB02

```
bool cApplication::Init()
{
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = Rx(OT_M_PI); // Rotation is 180 degress around X-axis

    vector3<double> wts(0,0,0);
    vector3<double> wte(0,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Print interpolated positions
    double t = 0.;
    for (int i=0;i <= 8; i++) {
        coordsys<double> c = w_motion.pos(t);
        std::cout << "Vektor: " << c.T() << " orientering: " <<
            c.Q() << std::endl;
        t += 1. / 8.;
    }

    return true;
}
```

12.7 AO00

```
bool cApplication::Init()
{
    // Motion for first OBB
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity
    vector3<double> wts(0,0,0);
    vector3<double> wte(-10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();
```

```
    // Motion for second OBB
    matrix3x3<double> brs = diag(1.0); // Rotation is identity
    matrix3x3<double> bre = diag(1.0); // Rotation is identity
    vector3<double> bts(10,0,0);
    vector3<double> bte(20,0,0);
    b_motion.m_p0 = coordsys<double>(bts, brs);
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    // Our OBB/OBB checker
    // Our OBB checker takes the OBBs as boost shared pointers
    OpenTissue::cont_coll_obb<double, obb_type> obb_checker;

    // First OBB
    vector3<double> obb_center(0,0,0);       // Center of OBB
    matrix3x3<double> obb_orien = diag(1.0); // Orientation of OBB
    vector3<double> obb_extends(1,1,1);      // Extends of OBB
    boost::shared_ptr<OBB<double> > O1(
            new obb_type(obb_center, obb_orien, obb_extends));

    // Second OBB
    vector3<double> obb_center2(10,0,0);   // Center of OBB
    boost::shared_ptr<OBB<double> > O2(
            new obb_type(obb_center2, obb_orien, obb_extends));

    // The two OBBs moves away from eachother: O1 startes at x=0 and moves to
    // x=-10. O2 starts at x=10 and moves to X=20. They can therefore not
    // collide since they both have half-extends of 1 along the x-axis.

    // In case of collision, t is the conservative collision time
    double t = obb_checker.check_pair(O1, w_motion, O2, b_motion, 0., 1.);
    if (t>= 0.0)
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;

    return true;
}
```

12.8 AO01

```
bool cApplication::Init()
{
    // Motion for first OBB
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity
    vector3<double> wts(0,0,0);
    vector3<double> wte(10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Motion for second OBB
    matrix3x3<double> brs = diag(1.0); // Rotation is identity
    matrix3x3<double> bre = diag(1.0); // Rotation is identity
    vector3<double> bts(10,0,0);
    vector3<double> bte(0,0,0);
    b_motion.m_p0 = coordsys<double>(bts, brs);
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    // Our OBB/OBB checker
    // Our OBB checker takes the OBBs as boost shared pointers
    OpenTissue::cont_coll_obb<double, obb_type> obb_checker;

    // First OBB
    vector3<double> obb_center(0,0,0);       // Center of OBB
    matrix3x3<double> obb_orien = diag(1.0); // Orientation of OBB
    vector3<double> obb_extends(1,1,1);      // Extends of OBB
    boost::shared_ptr<OBB<double> > O1(
            new obb_type(obb_center, obb_orien, obb_extends));

    // Second OBB
    vector3<double> obb_center2(10,0,0);   // Center of OBB
```

```
    boost::shared_ptr<OBB<double> > O2(
            new obb_type(obb_center2, obb_orien, obb_extends));

    // The two OBBs collides: O1 startes at x=0 and moves to
    // x=10. O2 starts at x=10 and moves to X=0.

    // In case of collision, t is the conservative collision time
    double t = obb_checker.check_pair(O1, w_motion, O2, b_motion, 0., 1.);
    if (t>= 0.0)
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;

    return true;
}
```

12.9 AO02

```
bool cApplication::Init()
{
    // Motion for first OBB
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity
    vector3<double> wts(0,0,0);
    vector3<double> wte(-10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Motion for second OBB
    matrix3x3<double> brs = diag(1.0); // Rotation is identity
    matrix3x3<double> bre = diag(1.0); // Rotation is identity
    vector3<double> bts(0,0,0);
    vector3<double> bte(10,0,0);
    b_motion.m_p0 = coordsys<double>(bts, brs);
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    // Our OBB/OBB checker
    // Our OBB checker takes the OBBs as boost shared pointers
    OpenTissue::cont_coll_obb<double, obb_type> obb_checker;

    // First OBB
    vector3<double> obb_center(0,0,0);       // Center of OBB
    matrix3x3<double> obb_orien = diag(1.0); // Orientation of OBB
    vector3<double> obb_extends(1,1,1);      // Extends of OBB
    boost::shared_ptr<OBB<double> > O1(
            new obb_type(obb_center, obb_orien, obb_extends));

    // Second OBB
    vector3<double> obb_center2(10,0,0);   // Center of OBB
    boost::shared_ptr<OBB<double> > O2(
            new obb_type(obb_center2, obb_orien, obb_extends));

    // The two OBBs collides at time 0: they both start at position (0,0,0)

    // In case of collision, t is the conservative collision time
    double t = obb_checker.check_pair(O1, w_motion, O2, b_motion, 0., 1.);
    if (t>= 0.0)
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;

    return true;
}
```

12.10 AE00

```
bool cApplication::Init()
{
    // Motion for first edge
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity
```

```
        vector3<double> wts(0,0,0);
        vector3<double> wte(-10,0,0);
        w_motion.m_p0 = coordsys<double>(wts, wrs);
        w_motion.m_p1 = coordsys<double>(wte, wre);
        w_motion.init();

        // Motion for second edge
        matrix3x3<double> brs = diag(1.0); // Rotation is identity
        matrix3x3<double> bre = diag(1.0); // Rotation is identity
        vector3<double> bts(10,0,0);
        vector3<double> bte(20,0,0);
        b_motion.m_p0 = coordsys<double>(bts, brs);
        b_motion.m_p1 = coordsys<double>(bte, bre);
        b_motion.init();

        // Our EDGE/EDGE checker
        OpenTissue::cont_coll_edge<double> edge_edge_checker;

        // First edge
        vector3<double> e1a(0,0,0);
        vector3<double> e1b(1,0,0);

        // Second edge
        vector3<double> e2a(0,0,0);
        vector3<double> e2b(1,0,0);

        // In case of collision, t is the conservative collision time
        vector3<double> not_used1; // normal
        vector3<double> not_used2; // point
        double not_used3;          // time
        bool coll = edge_edge_checker.check(e1a, e1b, w_motion,
                e2a, e2b, b_motion, not_used1, not_used2, not_used3);

        // The edges do not collide. They start seperated and move in different
        // directions
        if (coll)
            std::cout << "Collision found" << std::endl;
        else
            std::cout << "Collision not found" << std::endl;

        return true;
    }
```

12.11 AE01

```
1   bool cApplication::Init()
    {
        // Motion for first edge
        matrix3x3<double> wrs = diag(1.0); // Rotation is identity
5       matrix3x3<double> wre = diag(1.0); // Rotation is identity
        vector3<double> wts(0,0,0);
        vector3<double> wte(10,0,0);
        w_motion.m_p0 = coordsys<double>(wts, wrs);
        w_motion.m_p1 = coordsys<double>(wte, wre);
10      w_motion.init();

        // Motion for second edge
        matrix3x3<double> brs = diag(1.0); // Rotation is identity
        matrix3x3<double> bre = diag(1.0); // Rotation is identity
15      vector3<double> bts(10,0,0);
        vector3<double> bte(0,0,0);
        b_motion.m_p0 = coordsys<double>(bts, brs);
        b_motion.m_p1 = coordsys<double>(bte, bre);
        b_motion.init();
20
        // Our EDGE/EDGE checker
        OpenTissue::cont_coll_edge<double> edge_edge_checker;

        // First edge
25      vector3<double> e1a(0,0,-1);
        vector3<double> e1b(0,0,1);

        // Second edge
        vector3<double> e2a(-1,0,0);
```

```
30      vector3<double> e2b(1,0,0);

        // In case of collision, t is the conservative collision time
        vector3<double> not_used1; // normal
        vector3<double> not_used2; // point
35      double not_used3;          // time
        bool coll = edge_edge_checker.check(e1a, e1b, w_motion,
                e2a, e2b, b_motion, not_used1, not_used2, not_used3);

        // The edges do collide. They start seperated and move towards
40      // eachother, they will pass thru eatchother
        if (coll)
            std::cout << "Collision found" << std::endl;
        else
            std::cout << "Collision not found" << std::endl;
45
        return true;
    }
```

12.12 AE02

```
1   bool cApplication::Init()
    {
        // Motion for first edge
        matrix3x3<double> wrs = diag(1.0); // Rotation is identity
5       matrix3x3<double> wre = diag(1.0); // Rotation is identity
        vector3<double> wts(0,0,0);
        vector3<double> wte(10,0,0);
        w_motion.m_p0 = coordsys<double>(wts, wrs);
        w_motion.m_p1 = coordsys<double>(wte, wre);
10      w_motion.init();

        // Motion for second edge
        matrix3x3<double> brs = diag(1.0); // Rotation is identity
        matrix3x3<double> bre = diag(1.0); // Rotation is identity
15      vector3<double> bts(0,0,0);
        vector3<double> bte(10,0,0);
        b_motion.m_p0 = coordsys<double>(bts, brs);
        b_motion.m_p1 = coordsys<double>(bte, bre);
        b_motion.init();
20
        // Our EDGE/EDGE checker
        OpenTissue::cont_coll_edge<double> edge_edge_checker;

        // First edge
25      vector3<double> e1a(0,0,-1);
        vector3<double> e1b(0,0,1);

        // Second edge
        vector3<double> e2a(-1,0,0);
30      vector3<double> e2b(1,0,0);

        // In case of collision, t is the conservative collision time
        vector3<double> not_used1; // normal
        vector3<double> not_used2; // point
35      double not_used3;          // time
        bool coll = edge_edge_checker.check(e1a, e1b, w_motion,
                e2a, e2b, b_motion, not_used1, not_used2, not_used3);

        // The edges do collide. They start off coliding
40      if (coll)
            std::cout << "Collision found" << std::endl;
        else
            std::cout << "Collision not found" << std::endl;

45      return true;
    }
```

12.13 AVF00

```
1   bool cApplication::Init()
    {
```

```
    // Motion for point
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity
    vector3<double> wts(0,0,0);
    vector3<double> wte(-10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Motion for face
    matrix3x3<double> brs = diag(1.0); // Rotation is identity
    matrix3x3<double> bre = diag(1.0); // Rotation is identity
    vector3<double> bts(10,0,0);
    vector3<double> bte(20,0,0);
    b_motion.m_p0 = coordsys<double>(bts, brs);
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    // Our FACE/POINT checker
    OpenTissue::cont_coll_face_point<double> face_point_checker;

    // Point
    e1a = vector3<double>(0,0,0);

    // Face
    f1a = vector3<double>(0,0,-1);
    f1b = vector3<double>(0,0,1);
    f1c = vector3<double>(0,1,0);

    // In case of collision, t is the conservative collision time
    double t = face_point_checker.check_face_point(e1a, w_motion,
            f1a, f1b, f1c, b_motion, 0, 1);

    // The face and point does not collide. They start off seperated
    // and move away from eatchother
    if (t>=0.0)
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;

    return true;
}
```

12.14 AVF01

```
bool cApplication::Init()
{
    // Motion for point
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity
    vector3<double> wts(0,0,0);
    vector3<double> wte(10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Motion for face
    matrix3x3<double> brs = diag(1.0); // Rotation is identity
    matrix3x3<double> bre = diag(1.0); // Rotation is identity
    vector3<double> bts(10,0,0);
    vector3<double> bte(0,0,0);
    b_motion.m_p0 = coordsys<double>(bts, brs);
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    // Our FACE/POINT checker
    OpenTissue::cont_coll_face_point<double> face_point_checker;

    // Point
    e1a = vector3<double>(0,0,0);

    // Face
    f1a = vector3<double>(0,0,-1);
    f1b = vector3<double>(0,0,1);
    f1c = vector3<double>(0,1,0);
```

```
    // In case of collision, t is the conservative collision time
    double t = face_point_checker.check_face_point(e1a, w_motion,
            f1a, f1b, f1c, b_motion, 0, 1);

    // The face and point does collide. They start off seperated
    // and move towards eatchother, they move thru eatchother
    if (t>=0.0)
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;

    return true;
}
```

12.15 AVF02

```
bool cApplication::Init()
{
    // Motion for point
    matrix3x3<double> wrs = diag(1.0); // Rotation is identity
    matrix3x3<double> wre = diag(1.0); // Rotation is identity
    vector3<double> wts(0,0,0);
    vector3<double> wte(10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Motion for face
    matrix3x3<double> brs = diag(1.0); // Rotation is identity
    matrix3x3<double> bre = diag(1.0); // Rotation is identity
    vector3<double> bts(0,0,0);
    vector3<double> bte(10,0,0);
    b_motion.m_p0 = coordsys<double>(bts, brs);
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    // Our FACE/POINT checker
    OpenTissue::cont_coll_face_point<double> face_point_checker;

    // Point
    e1a = vector3<double>(0,0,0);

    // Face
    f1a = vector3<double>(0,0,-1);
    f1b = vector3<double>(0,0,1);
    f1c = vector3<double>(0,1,0);

    // In case of collision, t is the conservative collision time
    double t = face_point_checker.check_face_point(e1a, w_motion,
            f1a, f1b, f1c, b_motion, 0, 1);

    // The face and point does collide. They start colliding
    if (t>=0.0)
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;

    return true;
}
```

12.16 AT00

```
bool cApplication::Init()
{
    // Load mesh to build OBB tree around
    if (!mesh_default_read("/home/jnj/src/DataTissue/mesh/teapot.msh",
              bMesh)) return false;
    // Note: this teapot has a maximum spread of 40 units

    // Build OBB BVHs
    constructor_type BVHconstructor1;
    create_obb_tree(bMesh, wBVH , BVHconstructor1); // First tree
```

```
        constructor_type BVHconstructor2;
        create_obb_tree(bMesh, bBVH, BVHconstructor2); // Second tree

        // Motion for first tree
        matrix3x3<double> wrs = diag(1.0);
        matrix3x3<double> wre = diag(1.0);

        vector3<double> wts(0,0,0);
        vector3<double> wte(-10,0,0);
        w_motion.m_p0 = coordsys<double>(wts, wrs);
        w_motion.m_p1 = coordsys<double>(wte, wre);
        w_motion.init();

        // Motion for second tree
        matrix3x3<double> brs = diag(1.0);
        matrix3x3<double> bre = diag(1.0);

        vector3<double> bts(40,0,0);
        vector3<double> bte(50,0,0);
        b_motion.m_p0 = coordsys<double>(bts, brs);
        b_motion.m_p1 = coordsys<double>(bte, bre);
        b_motion.init();

        OpenTissue::cont_coll_obb<double, obb_type> obb_check;
        cont_coll_bvh_world<obb_coll_policy<bvh_type> > bvh_obb_check;
        bvh_obb_check.set_m1(w_motion);
        bvh_obb_check.set_m2(b_motion);
        bvh_obb_check.set_obb_checker(obb_check);
        bvh_obb_check.set_edge_edge_checker(edge_check);
        bvh_obb_check.set_face_point_checker(face_check);

        bvh_obb_check.run_world_check(wBVH, bBVH, obb_colls);

        // The objects does not collide: they start off seperated and move away
        // from eatchother
        if (obb_colls.size())
            std::cout << "Collision found" << std::endl;
        else
            std::cout << "Collision not found" << std::endl;

        return true;
    }
```

12.17 AT01

```
1   bool cApplication::Init()
    {
        // Load mesh to build OBB tree around
        if (!mesh_default_read("/home/jnj/src/DataTissue/mesh/teapot.msh",
                    bMesh)) return false;
        // Note: this teapot has a maximum spread of 40 units

        // Build OBB BVHs
        constructor_type BVHconstructor1;
        create_obb_tree(bMesh, wBVH , BVHconstructor1); // First tree
        constructor_type BVHconstructor2;
        create_obb_tree(bMesh, bBVH, BVHconstructor2); // Second tree

        // Motion for first tree
        matrix3x3<double> wrs = diag(1.0);
        matrix3x3<double> wre = diag(1.0);

        vector3<double> wts(0,0,0);
        vector3<double> wte(-10,0,0);
        w_motion.m_p0 = coordsys<double>(wts, wrs);
        w_motion.m_p1 = coordsys<double>(wte, wre);
        w_motion.init();

        // Motion for second tree
        matrix3x3<double> brs = diag(1.0);
        matrix3x3<double> bre = diag(1.0);

        vector3<double> bts(40,0,0);
        vector3<double> bte(0,0,0);
        b_motion.m_p0 = coordsys<double>(bts, brs);
```

```
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    OpenTissue::cont_coll_obb<double, obb_type> obb_check;
    cont_coll_bvh_world<obb_coll_policy<bvh_type> > bvh_obb_check;
    bvh_obb_check.set_m1(w_motion);
    bvh_obb_check.set_m2(b_motion);
    bvh_obb_check.set_obb_checker(obb_check);
    bvh_obb_check.set_edge_edge_checker(edge_check);
    bvh_obb_check.set_face_point_checker(face_check);

    bvh_obb_check.run_world_check(wBVH, bBVH, obb_colls);

    // The objects does collide: they start off seperated and move
    // towards eatchother, they move thru eatchother
    if (obb_colls.size())
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;

    return true;
}
```

12.18 AT02

```
bool cApplication::Init()
{
    // Load mesh to build OBB tree around
    if (!mesh_default_read("/home/jnj/src/DataTissue/mesh/teapot.msh",
              bMesh)) return false;
    // Note: this teapot has a maximum spread of 40 units

    // Build OBB BVHs
    constructor_type BVHconstructor1;
    create_obb_tree(bMesh, wBVH , BVHconstructor1); // First tree
    constructor_type BVHconstructor2;
    create_obb_tree(bMesh, bBVH, BVHconstructor2); // Second tree

    // Motion for first tree
    matrix3x3<double> wrs = diag(1.0);
    matrix3x3<double> wre = diag(1.0);

    vector3<double> wts(0,0,0);
    vector3<double> wte(10,0,0);
    w_motion.m_p0 = coordsys<double>(wts, wrs);
    w_motion.m_p1 = coordsys<double>(wte, wre);
    w_motion.init();

    // Motion for second tree
    matrix3x3<double> brs = diag(1.0);
    matrix3x3<double> bre = diag(1.0);

    vector3<double> bts(15,0,0);
    vector3<double> bte(40,0,0);
    b_motion.m_p0 = coordsys<double>(bts, brs);
    b_motion.m_p1 = coordsys<double>(bte, bre);
    b_motion.init();

    OpenTissue::cont_coll_obb<double, obb_type> obb_check;
    cont_coll_bvh_world<obb_coll_policy<bvh_type> > bvh_obb_check;
    bvh_obb_check.set_m1(w_motion);
    bvh_obb_check.set_m2(b_motion);
    bvh_obb_check.set_obb_checker(obb_check);
    bvh_obb_check.set_edge_edge_checker(edge_check);
    bvh_obb_check.set_face_point_checker(face_check);

    bvh_obb_check.run_world_check(wBVH, bBVH, obb_colls);

    // The objects does collide: they start off colliding
    if (obb_colls.size())
        std::cout << "Collision found" << std::endl;
    else
        std::cout << "Collision not found" << std::endl;
```

```
50      return true;
    }
```

13 Bilag B, Testresultater

13.1 AB00

```
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [1,0,0,0]
```

Korrekt.

13.2 AB01

```
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [1.25,0,0] orientering: [1,0,0,0]
Vektor: [2.5,0,0] orientering: [1,0,0,0]
Vektor: [3.75,0,0] orientering: [1,0,0,0]
Vektor: [5,0,0] orientering: [1,0,0,0]
Vektor: [6.25,0,0] orientering: [1,0,0,0]
Vektor: [7.5,0,0] orientering: [1,0,0,0]
Vektor: [8.75,0,0] orientering: [1,0,0,0]
Vektor: [10,0,0] orientering: [1,0,0,0]
```

Korrekt.

13.3 AB02

```
Vektor: [0,0,0] orientering: [1,0,0,0]
Vektor: [0,0,0] orientering: [0.980785,-0.19509,0,0]
Vektor: [0,0,0] orientering: [0.92388,-0.382683,0,0]
Vektor: [0,0,0] orientering: [0.83147,-0.55557,0,0]
Vektor: [0,0,0] orientering: [0.707107,-0.707107,0,0]
Vektor: [0,0,0] orientering: [0.55557,-0.83147,0,0]
Vektor: [0,0,0] orientering: [0.382683,-0.92388,0,0]
Vektor: [0,0,0] orientering: [0.19509,-0.980785,0,0]
Vektor: [0,0,0] orientering: [-6.12303e-17,-1,0,0]
```

Korrekt idet -6.12303e-17 kan anses som værende nul.

13.4 AO00

```
Collision not found
```

Korrekt.

13.5 AO01

```
Collision found
```

Korrekt.

13.6 AO02

```
Collision found
```

Korrekt.

13.7 AE00

```
Collision not found
```

Korrekt.

13.8 AE01

```
Collision found
```

Korrekt.

13.9 AE02

```
Collision found
```

Korrekt.

13.10 AVF00

```
Collision not found
```

Korrekt

13.11 AVF01

```
Collision found
```

Korrekt.

13.12 AVF02

```
Collision found
```

Korrekt.

13.13 AT00

```
Collision not found
```

Korrekt

13.14 AT01

```
I found collision at time: 0.984375
I found collision at time: 0.950195
I found collision at time: 0.934814
I found collision at time: 0.922318
I found collision at time: 0.867243
I found collision at time: 0.859403
I found collision at time: 0.84767
I found collision at time: 0.796388
I found collision at time: 0.788819
I found collision at time: 0.781128
I found collision at time: 0.757782
I found collision at time: 0.749985
I found collision at time: 0.74218
I found collision at time: 0.736322
I found collision at time: 0.729976
Collision found
```

Korrekt (de udskrevne tider er fundne kollisionstider imellem geometri).

13.15 AT02

```
I found collision at time: 0.0703125
I found collision at time: 0.0585938
I found collision at time: 0.0317383
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
```

```
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
I found collision at time: 0
```

Korrekt (de udskrevne tider er fundne kollisionstider imellem geometri).

14 Bilag C, Kode

14.1 arbitrary_motion.h

```
#ifndef OPENTISSUE_COLLISION_CONTINOUOS_ARBITRARY_MOTION_H
#define OPENTISSUE_COLLISION_CONTINOUOS_ARBITRARY_MOTION_H
/*
 * An arbitary motion.
 *
 * This is the interface a "real" motion must implement.
 *
 * Jackj.
 */

#include "interval.h"
#include <OpenTissue/math/coordsys.h>
#include <OpenTissue/math/quaternion.h>
#include <OpenTissue/math/vector.h>
#include <OpenTissue/math/matrix.h>

namespace OpenTissue
{

    template<typename real_type_>
    class arbitrary_motion
    {

        public:

            typedef vector3<real_type_>                 vector_type;
            typedef matrix3x3<real_type_>               matrix_type;

            typedef OpenTissue::interval<real_type_>    int_type;
            typedef vector3<int_type>                   vector_i_type;
            typedef quaternion<int_type>                quaternion_i_type;
            typedef matrix3x3<int_type>                 matrix_i_type;

        public:

            virtual ~arbitrary_motion() {}

        public:
```

```
40          /*
             * Function can set up needed variables, will be called
             * before anything else.
             */
            virtual void init() = 0;
45
            /*
             * Calculate interpolated position at time t.
             */
            virtual coordsys<real_type_> pos(real_type_ const &t) const = 0;
50
            /*
             * Returns precalced position (translation part)
             * at time int_step *n
             */
55          virtual vector_type pos_translationsprec(unsigned int const n)
                const = 0;

            /*
             * Returns precalced position (orientation part)
60           * at time int_step *n
             */
            virtual matrix_type pos_rotationprec(unsigned int const n)
                const = 0;

65          /*
             * Bound rotation in interval [t0, t1]
             */
            virtual quaternion_i_type bound_rotation(real_type_ const &t0,
                    real_type_ const &t1) const = 0;
70
            /*
             * Bound rotation in interval [int_step*n, int_step*(n+1)],
             * these values are considered pre-calced (in init()).
             * See below for an explanation of int_step
75           */
            virtual matrix_i_type bound_rotation_prec(unsigned int const n)
                const = 0;

            /*
80           * Bound translation in interval [t0, t1]
             */
            virtual vector_i_type bound_translation(real_type_ const &t0,
                    real_type_ const &t1) const = 0;

85          /*
            * Bound translation in interval [int_step*n, int_step*(n+1)],
            * these values are considered pre-calced (in init()).
            * See below for an explanation of int_step
            */
90          virtual vector_i_type bound_translation_prec(unsigned int const n)
                const = 0;

            /*
             * Return pos'(t)
95           */
            virtual coordsys<real_type_> derived_pos(real_type_ const &t)
                const = 0;

            /*
100          * 2^refinement level = number of subintervals [0,1] is split
             * into when pre-calcing bounds
             */
            virtual unsigned int refinement_level() const = 0;

105         /*
             * 1 / (2 ^refinement level)
             *
             * In other words, the length of the subintervals we split
             * [0,1] into when pre-calcing bounds
110          * Considered pre-calced
             */
            virtual real_type_ int_step() const = 0;

            /*
115          * Returns n so that
```

```
             * t-int_step<=n*int_step<=t
             */
            virtual unsigned int bucket_lower(real_type_ const &t) const = 0;

            /*
             * Returns n so that
             * t<=n*int_step<=t+int_step
             */
            virtual unsigned int bucket_upper(real_type_ const &t) const = 0;

    }; // arbitrary_motion class

    // this is mostly for debugging.
    // Print an interval "[a,b]"
    template<typename V>
        std::ostream & operator<<
        (std::ostream & o,boost::numeric::interval<V> const & i)
    {
        o << "[" << i.lower() << "," << i.upper() << "]";
        return o;
    }

} // OpenTissue namespace

#endif // OPENTISSUE_COLLISION_CONTINOUOS_ARBITARY_MOTION_H
```

14.2 arbitrary_constant.h

```
#ifndef OPENTISSUE_COLLISION_CONTINUOUS_ARBITRARY_CONSTANT_H
#define OPENTISSUE_COLLISION_CONTINUOUS_ARBITRARY_CONSTANT_H

/*
 * Arbitrary motion with constant translation and rotation.
 *
 * Parameters for this motion:
 *
 * p0: Object's position at time 0.
 * p1: Object's position at time 1.
 *
 * Jackj.
 */

#include <math.h>
#include <boost/numeric/interval.hpp>
#include "interval.h"
#include <OpenTissue/math/constants.h>
#include <OpenTissue/math/coordsys.h>
#include <OpenTissue/math/vector.h>
#include <OpenTissue/math/matrix.h>
#include <OpenTissue/math/quaternion.h>
#include "arbitrary_motion.h"

namespace OpenTissue
{

    template<typename real_type_>
    class arbitrary_constant : public arbitrary_motion<real_type_>
    {
        public:

            typedef real_type_                          real_type;
            typedef matrix3x3<real_type>                matrix_type;
            typedef vector3<real_type>                  vector_type;
            typedef quaternion<real_type>               quaternion_type;

            typedef OpenTissue::interval<real_type>     int_type;
            typedef vector3<int_type>                   vector_i_type;
            typedef quaternion<int_type>                quaternion_i_type;
            typedef matrix3x3<int_type>                 matrix_i_type;

        public:

            coordsys<real_type> m_p0;    // Object's position at time 0
```

```
    coordsys<real_type> m_p1;    // Object's position at time 1

protected:

    // These variables are calculated in init()
    vector_type              m_trans;        // Object's translation in [0,1]
    real_type                m_sin_w_inv;    // 1/sin(w)
    real_type                m_w;            // acos(m_A*m_B) (below)
    int                      m_type_slerp;   // type of slerp
    quaternion_type          m_A;            // Start orientation
    quaternion_type          m_B;            // End orientation
    bool                     m_slerp_way;    // This is used in derived_pos

    template<typename real_type>
    class precalced {

        // Contains either bounds or a position

        friend class arbitrary_constant;

        protected:

            vector3<real_type>   m_trans;
            matrix3x3<real_type> m_ort;

    }; // precalced class

    real_type                          m_int_step;
    std::vector<precalced<int_type> >  m_precalc_bounds;
    std::vector<precalced<real_type> > m_precalc_values;

public:

    arbitrary_constant()
        : m_p0()
        , m_p1() {}

    arbitrary_constant(coordsys<real_type> p0, coordsys<real_type> p1)
        : m_p0(p0)
        , m_p1(p1) {}

    // this one would make sense for traditional discrete
    // collision detection
    arbitrary_constant(coordsys<real_type> p0)
        : m_p0(p0)
        , m_p1(p0) {}

    arbitrary_constant(arbitrary_constant const &m)
        : m_p0(m.m_p0)
        , m_p1(m.m_p1)
        , m_trans(m.m_trans) {}

    virtual ~arbitrary_constant() {}

private:

    /*
     * Interpolates orientation like slerp2 in quaternion.h,
     * uses some precalced constants.
     */
    quaternion_type precalced_slerp2(real_type const &t) const
    {
        using std::sin;

        if (!m_type_slerp)
            return ((1.-t)*m_A + t*m_B);
        if (m_type_slerp == 1) {
            real_type t_pi = t*OT_M_PI;
            return (sin((OT_M_PI/2.)-t_pi)*m_A + sin(t_pi)*m_B);
        }

        real_type t_w = t*m_w;
        return (sin(m_w-t_w)*m_sin_w_inv*m_A + sin(t_w)*m_sin_w_inv*m_B);
    }

public:
```

```
            void init()
            {
125             m_trans = m_p1.T() - m_p0.T();

                //
                // This is very much like slerp2 in quaternion.h
                //
130
                m_A = m_p0.Q();
                m_B = m_p1.Q();
                real_type q_tiny = 10e-7;

135             real_type cos_omega = m_A * m_B;

                if ((1.0-cos_omega)<q_tiny) {
                    // Linear
                    m_type_slerp = 0;
140             } else
                if ((1.0+cos_omega)<q_tiny) {
                    // Normal, but we go q0->hat(q0)->-q0=q1
                    m_type_slerp = 1;
                    m_B = hat(m_A);
145             } else {
                    // Normal
                    m_type_slerp = 2;

                    // Figure out which way is shortest around hypersphere
150                 quaternion<real_type> C = m_A+m_B;
                    real_type l1 = C*C;
                    C = m_A-m_B;
                    real_type l2 = C*C;
                    m_slerp_way = l2<l1;
155                 if (m_slerp_way)
                        m_B = -m_B;

                    using std::acos;
                    using std::sin;
160                 m_w = acos(cos_omega);
                    m_sin_w_inv = 1. / sin(m_w);
                }

                // Now precalc motion bounds at some fixed times:
165             // 0, m_int_step, 2*m_int_step,...,
                // 2^(refinement level)*m_int_step=1
                using std::pow;

                unsigned int r =
170                 (unsigned int)(pow(2., (real_type)refinement_level()));
                m_precalc_values.reserve(r);
                m_precalc_bounds.reserve(r);

                m_int_step = 1. / (real_type)r;
175
                // Precalced bounds
                for (real_type t = 0.; t < 1; t+=m_int_step) {
                    precalced<int_type> b;

180                 b.m_trans = bound_translation(t, t+m_int_step);
                    b.m_ort = bound_rotation(t, t+m_int_step);
                    m_precalc_bounds.push_back(b);
                }

185             // Precaled positions
                for (real_type t = 0.; t <= 1; t+=m_int_step) {
                    precalced<real_type> b;

                    coordsys<real_type> post = pos(t);
190                 b.m_trans = post.T();
                    b.m_ort = post.Q();
                    m_precalc_values.push_back(b);
                }
            }
195
            unsigned int refinement_level() const
            {
```

```
    return 7;
}

real_type int_step() const
{
    return m_int_step;
}

unsigned int bucket_lower(real_type const &t) const
{
    using std::floor;
    return (unsigned int)floor(t / m_int_step);

}

unsigned int bucket_upper(real_type const &t) const
{
    using std::ceil;
    return (unsigned int)ceil(t / m_int_step);
}

coordsys<real_type> pos(real_type const &time) const
{
    quaternion<real_type> rot = precalced_slerp2(time);
    return coordsys<real_type>(m_p0.T() + time*m_trans, // trans.
            rot);                                       // rot.
}

vector_type pos_translationsprec(unsigned int const n) const
{
    return m_precalc_values[n].m_trans;
}

matrix_type pos_rotationprec(unsigned int const n) const
{
    return m_precalc_values[n].m_ort;
}

// Bounds on rotation in time interval [t0, t1]
quaternion_i_type bound_rotation(real_type_ const &t0,
        real_type_ const &t1) const
{
    quaternion<real_type> startr = precalced_slerp2(t0);
    quaternion<real_type> endr = precalced_slerp2(t1);

    return quaternion_i_type(

            startr.s() > endr.s() ?
            int_type(endr.s(),startr.s()) :
            int_type(startr.s(), endr.s()),

            startr.v()[0] > endr.v()[0] ?
            int_type(endr.v()[0],startr.v()[0]) :
            int_type(startr.v()[0], endr.v()[0]),

            startr.v()[1] > endr.v()[1] ?
            int_type(endr.v()[1],startr.v()[1]) :
            int_type(startr.v()[1], endr.v()[1]),

            startr.v()[2] > endr.v()[2] ?
            int_type(endr.v()[2],startr.v()[2]) :
            int_type(startr.v()[2], endr.v()[2])

            );
}

// Bounds on rotation, uses values calculated
// in init()
matrix_i_type bound_rotation_prec(unsigned int const n)
    const
{
    return m_precalc_bounds[n].m_ort;
}

// Bounds on translation in time interval [t0, t1]
vector_i_type bound_translation(real_type_ const &t0,
        real_type_ const &t1) const
```

```
275         {
                // Our translation is linear and so highest/lowest
                // bounds must occour at time t0 / t1
                vector3<real_type> start = m_p0.T() + m_trans*t0;
                vector3<real_type> end = m_p0.T() + m_trans*t1;
280
                vector_i_type boundsT;

                if(start[0]>end[0]) // X
                    boundsT[0] = int_type(end[0], start[0]);
285             else
                    boundsT[0] = int_type(start[0], end[0]);
                if(start[1]>end[1]) // Y
                    boundsT[1] = int_type(end[1], start[1]);
                else
290                 boundsT[1] = int_type(start[1], end[1]);
                if(start[2]>end[2]) // Z
                    boundsT[2] = int_type(end[2], start[2]);
                else
                    boundsT[2] = int_type(start[2], end[2]);
295
                return boundsT;
            }

            // Bounds on translation
300         // Uses values calculated in init()
            vector_i_type bound_translation_prec(unsigned int const n)
                const
            {
                return m_precalc_bounds[n].m_trans;
305         }

            /*
             * Calculates dT/dt and dQ/t (where T is translation
             * and Q is orientation).
310          *
             * Calculates DT/dt (t) and dQ/t (t)
             */
            coordsys<real_type> derived_pos(real_type const &t) const
            {
315
                // ---- Translation
                //
                // T(t) = m_p0.T() + t*m_trans)
                //
320             // dT/dt (t) = m_trans(t) = m_trans

                // ---- Orientation
                //
                // We recall that slerp2(q0,q1,u) is defined as
325             //
                // q(u) = q0*sin((1-u)w)/sin(w) + q1*sin(uw)/sin(w)
                //
                // where w = acos(q0q1)
                //
330             // Diff. of (1) with respect to u gives us
                //
                // dq/du (u) = -(q0*cos((u-1)*w)*w) / sin(w)
                //       + (q1*cos(t*w)*w) / sin(w)                    (1)
                //
335             // We notice that the "w-pitfalls" stays the same
                // (we are still only dividing by sin(w)).
                //
                // In case of w==0 slerp2 uses linear interpolation,
                // q(u) = (1-u)q0 + uq1
340             // which, diff. with respect to u, is,
                //
                // dq/du (u) = -q0 + q1 = q1-q0                       (2)
                //
                // In case of w==pi slerp2 "cheats" (there is no
345             // unique solution afaik, since it's pretty much
                // undefined what w=pi implies) and goes from q0
                // to hat(q0) to -q0=q1, this is
                //
                // q(u) = q0 * sin (pi/2 - u*pi) + qh * sin(u*pi)
350             //
```

```
                // where qh = hat(q0),
                //
                // diff. with respect to u gives
                //
                // dq/du (u) = -q0 * sin(u*pi)*pi + qh * cos(t*pi)*pi (3)
                //

                quaternion<real_type> rot;

                using std::cos;
                using std::sin;
                real_type tpi;
                real_type tw;

                switch(m_type_slerp) {
                    case 0:
                        // Linear, equation (2)
                        rot = m_p1.Q() - m_p0.Q();
                        break;
                    case 1:
                        // w==pi, equation (3)
                        tpi = t*OT_M_PI;
                        rot = (hat(m_p0.Q())*cos(tpi)
                            - m_p0.Q()*sin(tpi)) * OT_M_PI;
                    default:
                        // Normal, equation (1)

                        tw = t*m_w;

                        m_slerp_way ?
                            rot = -(m_p0.Q()*cos(tw-m_w)*m_w * m_sin_w_inv)
                            -(m_p1.Q() * cos(tw)*m_w) * m_sin_w_inv
                            :
                            rot = -(m_p0.Q()*cos(tw-m_w)*m_w * m_sin_w_inv)
                            +(m_p1.Q() * cos(tw)*m_w) * m_sin_w_inv;
                }

                return coordsys<real_type> (m_trans, rot);
            }

    }; // arbitrary_constant class

} // OpenTissue namespace
#endif // OPENTISSUE_COLLISION_CONTINUOUS_ARBITRARY_CONSTANT_H
```

14.3 cont_coll_bvh.h

```
#ifndef OPENTISSUE_COLLISION_CONTINUOUS_CONT_COLL_BVH_H
#define OPENTISSUE_COLLISION_CONTINUOUS_CONT_COLL_BVH_H

/*
 * Collision detection using BVH.
 *
 * Jackj.
 */

#include <list>
#include <OpenTissue/math/bounds.h>
#include <OpenTissue/utility/high_res_timer.h>

namespace OpenTissue {
    /*
     * The policy the classes below inherit must
     * provide the following:
     *
     * typedef bvh_type
     * typedef bv_type
     * typedef result_type
     * typedef real_type
     *
     * -------
     * bool bv_overlap(bv_type *a, bv_type *b, real_type &t0,
     *                 real_type &t1)
     *  - Checks if a and b collide in time interval [t0, t1]
     *    true if they collide, false if not.
```

```
     *
     *    t0 is set to conservative collision time
     *
     * -------
     * void reset(result_type &r)
     *  - reset whatever r contains.
     *
     * -------
     * bool geometry_overlap(bv_type *a, bv_type *b, real_type &t0,
     *                       real_type &t1)
     *  - Checks if geometry in a and b collides at some point
     *    in time [t0, t1].
     *    If function finds they do, it returns true and sets t0
     *    to time of collision.
     *
     *    Please note: geometries tested do not have to be in the
     *    final list of colliding geometries: We may find geometries
     *    with earlier collision time later on.
     *
     * -------
     * void report(bv_type *a, bv_type *b, real_type &t, result_type &r)
     *  - a and b contains geometries with earliest collision
     *    time: they collide at time t
     *
     * Function parameters may vary, check description before each
     * class.
     */

    /*
     * cont_coll_bvh_world
     * ---
     *
     * Checks for collision between two BVHs
     * in >same< CS (here called 'world' coordinate
     * space)
     */
    template <typename collision_policy>
    class cont_coll_bvh_world : public collision_policy
    {

        public:

            typedef typename collision_policy::bvh_type     bvh_type;
            typedef typename collision_policy::bv_type      bv_type;
            typedef typename bvh_type::bv_ptr               bv_ptr;
            typedef typename collision_policy::result_type  result_type;
            typedef typename collision_policy::real_type    real_type;
            typedef typename bvh_type::bv_ptr_iterator      bv_ptr_iterator;

        private:

            // This is for debug only
            HighResTimer<double>  obb_timer;
            HighResTimer<double>  geom_timer;

        public:

            // This is for debug only
            double                obb_time_spent;
            int                   number_obb;
            double                geom_time_spent;
            int                   number_geom;

        private:

            class bv_to_check
            {
                friend class cont_coll_bvh_world;

                protected:

                    real_type m_t;
                    bv_ptr m_A, m_B;

                public:
```

```
105                 bv_to_check(real_type t, bv_ptr A, bv_ptr B)
                        : m_t(t)
                        , m_A(A)
                        , m_B(B) {}

110                 bv_to_check() {}

            }; // bv_to_check class


115     public:

            /*
             * Do the actual collision test.
             *
120          * @param bvhA   A's BVH
             * @param bvhB   B's BVH
             * @param r      container to hold result(s)
             *
             * Remember: A and B are considered to be in
125          * the same CS.
             *
             * We maintain two times:
             *
             * One is global- earliest_time.
130          * This variable is the earliest time a collision
             * between >geometries< have been detected. We do not
             * care about BV collisions if they occour after this
             * time.
             *
135          * The other bvt.m_t is local. This time is a conversative
             * time for when two given BVs >parents< collided.
             * Geometry and OBB tests for two given BVs will
             * have lower bound bvt.m_t
             *
140          * So: Given two BVs a and b we check for a collision
             * in time interval [bvt.m_t,earliest_time]
             * If overlap finds a collision and both BVs are leaves,
             * we call geometry_overlap. geometry_overlap checks if the
             * geometries in the two BVs (two triangles for example)
145          * collide. If they do our earliest_time time is set
             * to the time indicated by geometry_overlap.
             *
             * We store a list of BV leave pairs containing colliding
             * gemometries and at the end we report all that has
150          * collision time <= earliest_time.
             *
             */
            template<typename results_container>
            void run_world_check(bvh_type const &bvhA, bvh_type const &bvhB,
155                     results_container &r)
            {

                reset(r); // policy
                obb_time_spent = geom_time_spent = 0.0;
160             number_obb = number_geom = 0;
                real_type earliest_time = (real_type)1.0; // Earliest recorded
                                                          // geometry collison

                std::list<bv_to_check> Q;                 // BVs yet to be checked
165
                std::list<bv_to_check> colls;             // Leaf BV pairs containing
                                                          // geometry that
                                                          // collides, bvt.m_t is
                                                          // collide time
170
                bv_ptr root_A =  boost::const_pointer_cast<bv_type>( bvhA.root() );
                bv_ptr root_B =  boost::const_pointer_cast<bv_type>( bvhB.root() );

                bv_to_check bvt(0, root_A, root_B);
175             Q.push_back(bvt);

                while (!Q.empty())
                {
                    bvt = Q.front(); Q.pop_front();
180
```

```
                    if (bvt.m_t > earliest_time) {
                        continue;
                    }

                    obb_timer.start();
                    // policy
                    bool bv = bv_overlap(bvt.m_A, bvt.m_B, bvt.m_t, earliest_time);
                    obb_timer.stop();
                    number_obb++;
                    obb_time_spent += obb_timer();

                    if (!bv)
                        continue;

                    if (bvt.m_A->is_leaf() && bvt.m_B->is_leaf()) {
                        geom_timer.start();
                        // policy
                        bool geom = geometry_overlap(bvt.m_A, bvt.m_B, bvt.m_t,
                                earliest_time);
                        geom_timer.stop();
                        number_geom++;
                        geom_time_spent += geom_timer();
                        if (!geom)
                            continue;

                        earliest_time = bvt.m_t; // Geometry in A/B collides
                                                 // at earliest time
                        colls.push_front(bvt);
                        continue;
                    }

                    // We wish to find an 'earliest_time' as soon as possible.
                    // We do this by making sure the BV with the center
                    // closest to the other BV's center is processed first

#define OT_BV_C_LEN(a,b) (((a)->volume().center() - (b)->volume().center())*\
                          ((a)->volume().center() - (b)->volume().center()))

                    if (bvt.m_B->is_leaf()  || (!bvt.m_A->is_leaf() &&
                        (bvt.m_A->volume().volume() > bvt.m_B->volume().volume()))) {

                        // (A is not leaf)
                        bv_ptr_iterator a=bvt.m_A->child_ptr_begin();
                        bv_to_check nbvt(bvt.m_t, *a, bvt.m_B);
                        real_type cur_shortest = OT_BV_C_LEN(nbvt.m_B, nbvt.m_A);
                        Q.push_front(nbvt);
                        for (++a;a!=bvt.m_A->child_ptr_end();++a) {
                            nbvt.m_A = *a;
                            real_type len = OT_BV_C_LEN(nbvt.m_B, nbvt.m_A);
                            if (len < cur_shortest) {
                                cur_shortest = len;
                                Q.push_front(nbvt);
                            } else
                                Q.push_back(nbvt);
                        }
                    } else {

                        //(B is not leaf)
                        bv_ptr_iterator b=bvt.m_B->child_ptr_begin();
                        bv_to_check nbvt(bvt.m_t, bvt.m_A, *b);
                        real_type cur_shortest = OT_BV_C_LEN(nbvt.m_B, nbvt.m_A);
                        Q.push_front(nbvt);
                        for (++b;b!=bvt.m_B->child_ptr_end();++b) {
                            nbvt.m_B = *b;
                            real_type len = OT_BV_C_LEN(nbvt.m_B, nbvt.m_A);
                            if (len < cur_shortest) {
                                cur_shortest = len;
                                Q.push_front(nbvt);
                            } else
                                Q.push_back(nbvt);
                        }
                    }
#undef OT_BV_C_LEN
                }

                // Our colls queue may (very well) contain BV leafs with
                // geometry that collides >after< earliest time.
```

```
                // We loop thru the list and report all BVs that have
                // collision time == earliest (we use <= just for kicks)
                //
                // Since we add new geometry collisions to the start
                // of the list, we can stop once we find the first
                // time > earliest_time
                bv_to_check geom_coll;
                while (!colls.empty()) {
                    geom_coll = colls.front(); colls.pop_front();
                    if (geom_coll.m_t <= earliest_time)
                        report(geom_coll.m_A, geom_coll.m_B, geom_coll.m_t, r);
                    else
                        break;
                }

            }

    }; // cont_coll_bvh_world class

} // OpenTissue namespace

#endif // OPENTISSUE_COLLISION_CONTINUOUS_CONT_COLL_BVH_H
```

14.4 cont_coll_bv_traits.h

```
#ifndef OPENTISSUE_COLLISION_CONTINUOUS_CONT_COLL_BV_TRAITS_H
#define OPENTISSUE_COLLISION_CONTINUOUS_CONT_COLL_BV_TRAITS_H

/*
 * Traits needed in a BV if it is to be
 * used in a BVH test.
 *
 * Jackj.
 */

#include <OpenTissue/geometry/sphere.h>

namespace OpenTissue {

    template <typename real_type>
    class cont_coll_bv_traits {

        public:

            Sphere<real_type> m_bounding_sphere;

    }; // cont_coll_bv_traits class

} // OpenTissue namespace

#endif // OPENTISSUE_COLLISION_CONTINUOUS_CONT_COLL_BV_TRAITS_H
```

14.5 cont_coll_edge.h

```
#ifndef OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_EDGE_H
#define OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_EDGE_H

/*
 * Class to check for collision between two edges.
 *
 * Jackj.
 */

#include <boost/numeric/interval.hpp>
#include "interval.h"
#include <OpenTissue/math/quaternion.h>
#include <OpenTissue/math/vector.h>
#include <OpenTissue/math/matrix.h>
#include "arbitrary_motion.h"

namespace OpenTissue {

    /*
```

```
20       * Checks for collision between edges
         */
        template<typename real_type_>
        class cont_coll_edge
        {
25          public:

                typedef real_type_                          real_type;
                typedef OpenTissue::interval<real_type>     int_type;
                typedef vector3<real_type>                  vector_type;
30              typedef matrix3x3<real_type>                matrix_type;
                typedef arbitrary_motion<real_type>         motion_type;
                typedef vector3<int_type>                   vector_i_type;
                typedef matrix3x3<int_type>                 matrix_i_type;
                typedef quaternion<int_type>                quaternion_i_type;
35
            private:

                real_type m_time_pres; // How small must time interval be before
                                       // we check the actual primitives?
40
            public:

                cont_coll_edge()
                    : m_time_pres(0.01) {}
45
                cont_coll_edge(real_type time_pres)
                    : m_time_pres(time_pres) {}

                ~cont_coll_edge() {}
50
            protected:

    #define OT_LINE_SEG_EPSILON ((real_type)1.e-6)

55              /*
                 * This function checks if two line segments given by
                 *
                 * @param a    First line segment, first endpoint,
                 * @param b    First line segment, second endpoint,
60               * @param c    Second line segment, first endpoint,
                 * @param d    Second line segment, second endpoint,
                 *
                 * intersect. We handle situations where the two segments
                 * are parallel (they are considered intersecting
65               * when parallel and touching).
                 *
                 * This test is based on
                 * "Faster line segment intersection" by Franklin Antonio,
                 * 1992.
70               *
                 */
                bool faster_line_segment_intersection(vector_type const &a,
                        vector_type const &b, vector_type const &c,
                        vector_type const &d)
75              {
                    using std::fabs;
                    real_type e = (b[0]-a[0])*(d[2]-c[2]) -
                        (b[2]-a[2])*(d[0]-c[0]);

80                  if (fabs(e)<OT_LINE_SEG_EPSILON) {
                        // FIXME: Check if they are co-incident
                        return false;
                    }

85                  real_type f = (a[2]-c[2])*(d[0]-c[0]) -
                        (a[0]-c[0])*(d[2]-c[2]);

                    if (e>-OT_LINE_SEG_EPSILON) {
                        if (f<OT_LINE_SEG_EPSILON || f>e)
90                          return false;
                    } else {
                        if (f>-OT_LINE_SEG_EPSILON || f<e)
                            return false;
                    }
95
```

```
                real_type g = (a[2]-c[2])*(b[0]-a[0]) -
                    (a[0]-c[0])*(b[2]-a[2]);

                if (e>-OT_LINE_SEG_EPSILON) {
                    if (g<OT_LINE_SEG_EPSILON || g>e)
                        return false;
                } else {
                    if (g>-OT_LINE_SEG_EPSILON || g<e)
                        return false;
                }

                return true;
            }

        public:

            /*
             * Returns >4 if no collision, otherwise returns time
             * in [0,1]
             */
            real_type check_interval(
                    vector_type &a, vector_type &b, motion_type &m1,
                    vector_type &c, vector_type &d, motion_type &m2,
                    real_type t0, real_type t1)
            {
                // bounds on translation and rotation
                quaternion_i_type bounds1R = m1.bound_rotation(t0, t1);
                vector_i_type     bounds1T = m1.bound_translation(t0, t1);
                quaternion_i_type bounds2R = m2.bound_rotation(t0, t1);
                vector_i_type     bounds2T = m2.bound_translation(t0, t1);

                // bounds on a,b,c and d
                vector_i_type boundsA = bounds1R.rotate(vector_i_type(
                        int_type(a[0]), int_type(a[1]), int_type(a[2]))) + bounds1T;
                vector_i_type boundsB = bounds1R.rotate(vector_i_type(
                        int_type(b[0]), int_type(b[1]), int_type(b[2]))) + bounds1T;
                vector_i_type boundsC = bounds2R.rotate(vector_i_type(
                        int_type(c[0]), int_type(c[1]), int_type(c[2]))) + bounds2T;
                vector_i_type boundsD = bounds2R.rotate(vector_i_type(
                        int_type(d[0]), int_type(d[1]), int_type(d[2]))) + bounds2T;

                // bound f(t)=c(t)a(t) dotp ( b(t)a(t) crossp d(t)c(t) )
                int_type F =
                    (boundsC-boundsA)*((boundsB-boundsA)%(boundsD-boundsC));

                // interval does not contain 0
                if (F.upper() < 0.0 || F.lower() > 0.0) return (real_type)5.0;

                if (t1-t0 <= m_time_pres) {

                    coordsys<real_type> p1 = m1.pos(t0);
                    coordsys<real_type> p2 = m2.pos(t0);
                    vector_type ap = p1.Q().rotate(a)+p1.T();
                    vector_type bp = p1.Q().rotate(b)+p1.T();
                    vector_type cp = p2.Q().rotate(c)+p2.T();
                    vector_type dp = p2.Q().rotate(d)+p2.T();

                    if (faster_line_segment_intersection(ap,bp,cp,dp))
                            return t0; // Geometry collision detected

                    return (real_type)5.0;
                }

                real_type m = 0.5 * (t0+t1);
                real_type ret = check_interval(a,b,m1, c,d,m2,  t0, m);
                if (ret<4.0) return ret;

                return check_interval(a,b,m1, c,d, m2, m, t1);
                // Assume this will be "tail-optimized"
            }

            /*
             * @param a endpoint of first edge
             * @param b endpoint of first edge
             * @param m1 motion for first edge
```

```
             *
             * @param c endpoint of second edge
             * @param d endpoint of second edge
             * @param m2 motion for second edge
             *
             * @param cp contact point (if any)
             * @param cn contact normal (if any)
             * @param t  contact time (if any)
             *
             * Checks for collision between two
             * edges. Returns true if they collide.
             */
            bool check(
                    vector_type &a, vector_type &b, motion_type &m1,
                    vector_type &c, vector_type &d, motion_type &m2,
                    vector_type &cp, vector_type &cn, real_type &t)
            {

                t = check_interval(a,b,m1, c,d,m2, 0.0, 1.0);
                if (t > 4.0) { t = 0.0; return false; }

                cp = cn = vector_type(0,0,0); // Avoid warnings
                return true;
            }
    }; // cont_coll_edge class


} // OpenTissue namespace

#endif // OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_EDGE_H
```

14.6 cont_coll_face.h

```
#ifndef OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_FACE_H
#define OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_FACE_H

/*
 * Class to check for collision between two faces.
 *
 * Jackj.
 */

#include <boost/numeric/interval.hpp>
#include "interval.h"
#include <OpenTissue/math/vector.h>
#include <OpenTissue/math/quaternion.h>
#include <OpenTissue/math/matrix.h>
#include <OpenTissue/mesh/trimesh/trimesh_face.h>
#include <OpenTissue/mesh/polymesh/polymesh_face.h>
#include <OpenTissue/geometry/util/barycentric.h>
#include <OpenTissue/geometry/plane.h>
#include "arbitrary_motion.h"

namespace OpenTissue {

    template<typename real_type_>
    class cont_coll_face_point
    {
        public:

            typedef real_type_                          real_type;
            typedef OpenTissue::interval<real_type>     int_type;
            typedef vector3<real_type>                  vector_type;
            typedef matrix3x3<real_type>                matrix_type;
            typedef arbitrary_motion<real_type>         motion_type;
            typedef vector3<int_type>                   vector_i_type;
            typedef matrix3x3<int_type>                 matrix_i_type;
            typedef quaternion<int_type>                quaternion_i_type;

        private:

            real_type m_time_pres; // Time precision, determines how small
                                   // we split a given interval
```

```
        public:

            cont_coll_face_point(real_type timepres)
                : m_time_pres(timepres) {}

            cont_coll_face_point()
                : m_time_pres((real_type)0.001) {}


        protected:

            /*
             * Checks if
             *
             * @param p    Point
             *
             * is inside triangle given by
             *
             * @param a    First corner
             * @param b    Second corner
             * @param c    Third corner
             *
             * We use barycentric coordinates to check this.
             */
#define OT_POINT_IN_TRI_EPS (10e-5)
            bool point_in_triangle(vector_type const &p,
                    vector_type const &a, vector_type const &b,
                    vector_type const &c)
            {
                plane<real_type> pl(a,b,c);

                if (pl.get_distance(p) > OT_POINT_IN_TRI_EPS)
                    return false;

                real_type w1, w2, w3;
                barycentric(a, b, c, p, w1, w2, w3);
                real_type lower = -OT_POINT_IN_TRI_EPS;
                real_type upper = 1.+OT_POINT_IN_TRI_EPS;
                return (w1>lower)&&(w1<upper) &&
                       (w2>lower)&&(w2<upper) &&
                       (w3>lower)&&(w3<upper);
            }

        public:

            /*
             * Gets time precision
             */
            real_type get_time_precision() const {
                return m_time_pres;
            }

            /*
             * Sets time precision
             */
            void set_time_precision(real_type const &t) {
                m_time_pres = t;
            }

            /*
             * Checks if
             * @param p    Point
             * moving using
             * @param m1   Motion for point
             *
             * intersects with triangle given by
             * @param a    Corner
             * @param b    Corner
             * @param c    Corner
             * moving using
             * @param m2   Motion for triangle
             *
             * in interval
             * @param t0
             * @param t1
             */
```

```
            real_type check_face_point(vector_type const &p,
                    motion_type &m1, vector_type const &a,
                    vector_type const &b, vector_type const &c,
                    motion_type &m2, real_type t0, real_type t1)
            {

                quaternion_i_type boundsAR = m1.bound_rotation(t0, t1);
                vector_i_type     boundsAT = m1.bound_translation(t0, t1);
                quaternion_i_type boundsBR = m2.bound_rotation(t0, t1);
                vector_i_type     boundsBT = m2.bound_translation(t0, t1);

                // Bounds on point
                vector_i_type boundsP = boundsAR.rotate(
                    vector_i_type(int_type(p[0]), int_type(p[1]), int_type(p[2])))
                    + boundsAT;

                // Bounds on face corners
                vector_i_type boundsA = boundsBR.rotate(
                    vector_i_type(int_type(a[0]), int_type(a[1]), int_type(a[2])))
                    + boundsBT;
                vector_i_type boundsB = boundsBR.rotate(
                    vector_i_type(int_type(b[0]), int_type(b[1]), int_type(b[2])))
                    + boundsBT;
                vector_i_type boundsC = boundsBR.rotate(
                    vector_i_type(int_type(c[0]), int_type(c[1]), int_type(c[2])))
                    + boundsBT;

                int_type boundsF =
                    (boundsA - boundsP) * ((boundsB-boundsA)%
                                           (boundsC-boundsA));

                if (boundsF.upper() < 0.0 || boundsF.lower() > 0.0)
                    return (real_type)5.0; // F does not contain zero

                if (t1-t0 <= m_time_pres) {
                    coordsys<real_type> p1 = m1.pos(t0);
                    coordsys<real_type> p2 = m2.pos(t0);
                    vector_type pp = p1.Q().rotate(p)+p1.T();
                    vector_type ap = p2.Q().rotate(a)+p2.T();
                    vector_type bp = p2.Q().rotate(b)+p2.T();
                    vector_type cp = p2.Q().rotate(c)+p2.T();

                    if (point_in_triangle(pp, ap, bp, cp))
                        return t0; // Geometry collision detected

                    return (real_type)5.0;
                }

                real_type m = 0.5 * (t1+t0);

                real_type t = check_face_point(p, m1, a, b, c, m2,
                        t0, m);
                if (t < 4.0)
                    return t;

                return check_face_point(p, m1, a, b, c, m2, m, t1);
                // Assume this is tail-optimized
            }


        }; // cont_coll_face_point class

        template<typename real_type_>
        class cont_coll_face_face
        {
            public:

                typedef real_type_                  real_type;


        }; // cont_coll_face_face class

    } // OpenTissue namespace
    #endif // OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_FACE_H
```

14.7 cont_coll_obb.h

```
#ifndef OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_OBB_H
#define OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_OBB_H

/*
 * Class to check for collision between two OBBs.
 *
 * Jackj.
 */

#include <OpenTissue/utility/high_res_timer.h>
#include <vector>
#include <boost/numeric/interval.hpp>
#include "interval.h"
#include <OpenTissue/math/vector.h>
#include <OpenTissue/math/quaternion.h>
#include <OpenTissue/math/coordsys.h>
#include <OpenTissue/math/matrix.h>
#include <OpenTissue/geometry/obb.h>
#include "arbitrary_motion.h"

namespace OpenTissue {

    /*
     * Checks for collision between OBBs
     */
    template<typename real_type_,typename obb_type_>
    class cont_coll_obb
    {
        public:

            typedef real_type_                          real_type;
            typedef obb_type_                           obb_type;
            typedef arbitrary_motion<real_type>         motion_type;
            typedef vector3<real_type>                  vector_type;
            typedef matrix3x3<real_type>                matrix_type;
            typedef coordsys<real_type>                 coordsys_type;

            typedef OpenTissue::interval<real_type>     int_type;
            typedef vector3<int_type>                   vector_i_type;
            typedef matrix3x3<int_type>                 matrix_i_type;
            typedef quaternion<int_type>                quaternion_i_type;

        public:

            cont_coll_obb() {}
            ~cont_coll_obb() {}

        private:

            class constant_obb_info {
                friend class cont_coll_obb;

                protected:

                    motion_type  const *m_m1;  // A motion
                    motion_type  const *m_m2;  // B motion

                    real_type    m_rad;  // Radius between two current BVs spheres

                    vector_type  m_acen; // A OBB center
                    vector_type  m_bcen; // B OBB center

                    vector_type  m_aext; // A OBB extends
                    vector_type  m_bext; // B OBB extends

                    vector_type  m_bo1;  // B OBB orientation axis 1
                    vector_type  m_bo2;  // B OBB orientation axis 2
                    vector_type  m_bo3;  // B OBB orientation axis 3

                    vector_type  m_ao1;  // A OBB orientation axis 1
                    vector_type  m_ao2;  // A OBB orientation axis 2
                    vector_type  m_ao3;  // A OBB orientation axis 3

            }; // constant_obb_info class
```

```
    constant_obb_info m_obb_info;

protected:

    /*
     * Bounds two matrices
     */
    inline void bound_matrix(matrix_type const &A,
            matrix_type const &B, matrix_i_type &dst) const
    {
        for (unsigned int i = 0; i < 3; i++)
            for (unsigned int j = 0; j < 3; j++)
                if (A[i][j] > B[i][j])
                    dst[i][j] = int_type(B[i][j], A[i][j]);
                else
                    dst[i][j] = int_type(A[i][j], B[i][j]);
    }

    /*
     * Bounds two vectors
     */
    inline void bound_vector(vector_type const &A, vector_type const &B,
            vector_i_type &dst) const
    {
        for (unsigned int i = 0; i < 3; i++)
            if (A[i]>B[i])
                dst[i] = int_type(B[i], A[i]);
            else
                dst[i] = int_type(A[i], B[i]);
    }

public:

    /*
     * Checks if aobb in abv and bobb inb bbv collides in time
     * interval [t0, t1].
     *
     * @return t\in[0,1] if they do, -1 otherwise
     */
    template<typename bv_ptr>
    real_type check_pair(bv_ptr const abv, motion_type const &m1,
            bv_ptr const bbv, motion_type const &m2,
            real_type const &t0, real_type const &t1)
    {

        static unsigned int id = 0;

        id++;

        obb_type const * aobb = &abv->volume();
        obb_type const * bobb = &bbv->volume();

        // This is (r1+r2)^2,
        // r1 = A's bounding sphere radius
        // r2 = B's bounding sphere radius
        real_type const rad =
            (abv->m_bounding_sphere.radius() +
             bbv->m_bounding_sphere.radius()) *
            (abv->m_bounding_sphere.radius() +
             bbv->m_bounding_sphere.radius());

        // OBB's centers
        vector_type const bcen = bobb->center();
        vector_type const acen = aobb->center();

        // OBB's extends
        vector_type const aext = aobb->ext();
        vector_type const bext = bobb->ext();

        // OBB's orientations
        matrix_type ori = bobb->orientation();
        vector_type const bo1(ori(0,0), ori(1,0), ori(2,0));
        vector_type const bo2(ori(0,1), ori(1,1), ori(2,1));
        vector_type const bo3(ori(0,2), ori(1,2), ori(2,2));
        ori = aobb->orientation();
        vector_type const ao1(ori(0,0), ori(1,0), ori(2,0));
```

```
                    vector_type const ao2(ori(0,1), ori(1,1), ori(2,1));
                    vector_type const ao3(ori(0,2), ori(1,2), ori(2,2));

155                 // Used to hold dot-products (see below)
                    static real_type dots[3][3];

                    real_type const int_step = m1.int_step();
                    assert(int_step == m2.int_step() ||
160                        ! "Different refinement levels in motions");

                    bool loop = true;
                    bool hit = false;

165                 unsigned int n = m1.bucket_lower(t0);

                    real_type t = int_step*(real_type)n;
                    for(; loop; t+=int_step) {

170                 if (t+int_step >= t1) { loop = false; } // last loop

                    matrix_i_type boundAR = m1.bound_rotation_prec(n);
                    vector_i_type boundAT = m1.bound_translation_prec(n);

175                 matrix_i_type boundBR = m2.bound_rotation_prec(n);
                    vector_i_type boundBT = m2.bound_translation_prec(n++);

                    // Bounds on OBB A's center
                    vector_i_type boundTA = boundAR * acen + boundAT;
180
                    // Bounds on OBB B's center
                    vector_i_type boundTB = boundBR * bcen + boundBT;

                    vector_i_type TBmTA = boundTB - boundTA;
185
                    // First check between the two BV's spheres
                    // The spheres have same center as our OBBs
                    int_type sphere_dist = TBmTA * TBmTA;
                    if (sphere_dist.lower() > rad) {
190                     continue;
                    }

                    // --------
                    // SAT TEST
195
                    // Need to check:
                    //
                    // |aT_AT_B|>\sum_{i=1}^3 a_i|a*e_i| + \sum_{i=1}^3 b_i|a*f_i|.
                    // a\in\{e_i, f_j, e_i^f_j, 1\leq i \leq 3, 1 \leq j \leq 3\}.
200                 //
                    // Notice that since our dot-product is commutative
                    // (well.. assuming real_type is) we have
                    //        f_i*e_i = e_i*f_i
                    //
205                 // we will be testing with axe e_1 for example where
                    // e_1f_1, e_1f_2 and e_1f_3 is calculated
                    //
                    // so when we check f_1,f_2 and f_3 we don't need
                    // to calculate these values again. Same applies
210                 // for e_2 and e_3.
                    //
                    // Notice also that
                    //     e_1*e_1 = 1
                    // and
215                 //     e_1*e_2=e_1*e_3=0
                    //
                    // this applies for e_1,e_2,e_3,f_1,f_2 and f_3.

                    // Calculates |e*f|.upper()
220 #define OT_CALC_DOT(x,y, e, f)\
                    dots[x][y] = ((e)*(f)).get_abs_upper()

                    // First we check E1-E3 and F1-F3, ie. face normals
                    // for A's/B's OBB.
225                 //
                    // We write a macro to do this
```

```
#define OT_SAT_TEST(expr1, expr2) \
                if (((expr1) * TBmTA).get_abs_lower() >\
                        (expr2)) continue

                // Notice that our rotations below
                // use our matrix<int_type> * vector<real_type>
                // defined in interval.h
                //

                // Bounds on B's OBB axes. Note that we don't translate
                // the axes, they are vectors, not vertices.
                vector_i_type boundF1 = boundBR * bo1;
                vector_i_type boundF2 = boundBR * bo2;
                vector_i_type boundF3 = boundBR * bo3;

                // Bounds on A's first axis. Note that we don't
                // translate, it's a vector, not a vertex
                vector_i_type boundE1 = boundAR * ao1;

                OT_CALC_DOT(0,0, boundE1, boundF1);
                OT_CALC_DOT(0,1, boundE1, boundF2);
                OT_CALC_DOT(0,2, boundE1, boundF3);
                OT_SAT_TEST(boundE1, aext(0)
                        + bext(0)*dots[0][0]
                        + bext(1)*dots[0][1]
                        + bext(2)*dots[0][2]);

                // Bounds on A's 2nd axis. Note that we don't
                // translate, it's a vector, not a vertex
                vector_i_type boundE2 = boundAR * ao2;

                OT_CALC_DOT(1,0, boundE2, boundF1);
                OT_CALC_DOT(1,1, boundE2, boundF2);
                OT_CALC_DOT(1,2, boundE2, boundF3);
                OT_SAT_TEST(boundE2, aext(1)
                        + bext(0)*dots[1][0]
                        + bext(1)*dots[1][1]
                        + bext(2)*dots[1][2]);

                // Bounds on A's 3rd axis. Note that we don't
                // translate, it's a vector, not a vertex
                vector_i_type boundE3 = boundAR * ao3;

                OT_CALC_DOT(2,0, boundE3, boundF1);
                OT_CALC_DOT(2,1, boundE3, boundF2);
                OT_CALC_DOT(2,2, boundE3, boundF3);
                OT_SAT_TEST(boundE3, aext(2)
                        + bext(0)*dots[2][0]
                        + bext(1)*dots[2][1]
                        + bext(2)*dots[2][2]);

#undef OT_CALC_DOT
                // If we are this far, all dot-products have
                // been calculated

                OT_SAT_TEST(boundF1, bext(0)
                        + aext(0)*dots[0][0]
                        + aext(1)*dots[1][0]
                        + aext(2)*dots[2][0]);

                OT_SAT_TEST(boundF2, bext(1)
                        + aext(0)*dots[0][1]
                        + aext(1)*dots[1][1]
                        + aext(2)*dots[2][1]);

                OT_SAT_TEST(boundF3, bext(2)
                        + aext(0)*dots[0][2]
                        + aext(1)*dots[1][2]
                        + aext(2)*dots[2][2]);

#undef OT_SAT_TEST

                // Now we have to check e_i^f_j 1\leq i \leq 3, 1\leq j\leq3
                //
                // We know that
                //    (u%v)*u = (u%v)*v = 0,
                // so we can skip two calculations in each test
```

```
305
                real_type lBoundLeft;
                real_type uBoundRight;
                vector_i_type axis;

310             // FIXME: does "axis" actually make this faster?

    #define OT_SAT_EXPR(ext,expr)\
                ((ext)*(axis*(expr)).get_abs_upper())

315 #define OT_SAT_TEST(expr,expr2) \
                axis = expr;\
                lBoundLeft = ((axis * TBmTA)).get_abs_lower();\
                uBoundRight = expr2;\
                if (lBoundLeft > uBoundRight)\
320             continue

                OT_SAT_TEST(boundE1 % boundF1,
                        OT_SAT_EXPR(aext(1), boundE2)+
                        OT_SAT_EXPR(aext(2), boundE3)+
325                     OT_SAT_EXPR(bext(1), boundF2)+
                        OT_SAT_EXPR(bext(2), boundF3));

                OT_SAT_TEST(boundE2 % boundF1,
                        OT_SAT_EXPR(aext(0), boundE1)+
330                     OT_SAT_EXPR(aext(2), boundE3)+
                        OT_SAT_EXPR(bext(1), boundF2)+
                        OT_SAT_EXPR(bext(2), boundF3));

                OT_SAT_TEST(boundE3 % boundF1,
335                     OT_SAT_EXPR(aext(0), boundE1)+
                        OT_SAT_EXPR(aext(1), boundE2)+
                        OT_SAT_EXPR(bext(1), boundF2)+
                        OT_SAT_EXPR(bext(2), boundF3));

340             OT_SAT_TEST(boundE1 % boundF2,
                        OT_SAT_EXPR(aext(1), boundE2)+
                        OT_SAT_EXPR(aext(2), boundE3)+
                        OT_SAT_EXPR(bext(0), boundF1)+
                        OT_SAT_EXPR(bext(2), boundF3));
345
                OT_SAT_TEST(boundE2 % boundF2,
                        OT_SAT_EXPR(aext(0), boundE1)+
                        OT_SAT_EXPR(aext(2), boundE3)+
                        OT_SAT_EXPR(bext(0), boundF1)+
350                     OT_SAT_EXPR(bext(2), boundF3));

                OT_SAT_TEST(boundE3 % boundF2,
                        OT_SAT_EXPR(aext(0), boundE1)+
                        OT_SAT_EXPR(aext(1), boundE2)+
355                     OT_SAT_EXPR(bext(0), boundF1)+
                        OT_SAT_EXPR(bext(2), boundF3));

                OT_SAT_TEST(boundE1 % boundF3,
                        OT_SAT_EXPR(aext(1), boundE2)+
360                     OT_SAT_EXPR(aext(2), boundE3)+
                        OT_SAT_EXPR(bext(0), boundF1)+
                        OT_SAT_EXPR(bext(1), boundF2));

                OT_SAT_TEST(boundE2 % boundF3,
365                     OT_SAT_EXPR(aext(0), boundE1)+
                        OT_SAT_EXPR(aext(2), boundE3)+
                        OT_SAT_EXPR(bext(0), boundF1)+
                        OT_SAT_EXPR(bext(1), boundF2));

370             OT_SAT_TEST(boundE3 % boundF3,
                        OT_SAT_EXPR(aext(0), boundE1)+
                        OT_SAT_EXPR(aext(1), boundE2)+
                        OT_SAT_EXPR(bext(0), boundF1)+
                        OT_SAT_EXPR(bext(1), boundF2));
375 #undef OT_SAT_EXPR
    #undef OT_SAT_TEST

                // ----------------------------------------
                // OBB might be colliding, we can't find
380             // any axes that seperate during the entire
                // time interval
```

```
                // ----------------------------------------
                hit = true;
                break;
                }
                return hit ? t : -1.;
            }

    }; // cont_coll_obb class

} // OpenTissue namespace

#endif // OPENTISSUE_COLLISION_CONTINOUOS_CONT_COLL_OBB_H
```

14.8 interval.h

```
#ifndef OPENTISSUE_MATH_INTERVAL_H
#define OPENTISSUE_MATH_INTERVAL_H

/*
 * Interval class.
 *
 * A *simple naive* speed test, no -DNDEBUG used.
 *
 * Intervals of type [x,y] where x<0 and y>0 used in testing.
 *
 * this interval, 10^6 muls: 2.85913 seconds
 * boost interval, 10^6 muls: 5.13671 seconds
 *
 * this interval, 10^6 adds: 1.94612 seconds
 * boost interval, 10^6 adds: 3.86221 seconds
 *
 * this interval, 10^6 subs: 2.00829 seconds
 * boost interval, 10^6 subs: 3.8229 seconds
 *
 * Intervals of type [x,y] where 0<x<y used in divs.
 *
 * this interval, 10^6 divs: 3.41608 seconds
 * boost interval, 10^6 divs: 4.78257 seconds
 *
 * This class contains >no< error-checking.
 * You might want to use boost::numeric::interval
 * until you know your interval-code is working.
 *
 * Jackj.
 */

namespace OpenTissue
{

    template <typename real_type_>
        class interval
        {
            public:

                typedef real_type_ real_type;

            protected:

                real_type l, u;

            public:

                interval()
                    : l(0)
                    , u(0) {}

                interval(interval const &i)
                    : l(i.l)
                    , u(i.u) {}

                interval(real_type const &val)
                    : l(val)
                    , u(val) {}
```

```
60              interval(real_type const &l_val,
                         real_type const &h_val)
                    : l(l_val)
                    , u(h_val) {}

65              ~interval() {}

            public:

                real_type & lower()
70              {
                    return l;
                }


75              real_type const & lower()const
                {
                    return l;
                }

80
                real_type & upper()
                {
                    return u;
                }
85              real_type const & upper() const
                {
                    return u;
                }
90              inline real_type & operator() (unsigned int const index)
                {
                    if (!index) return l;
                    return u;
95              }

                inline real_type const & operator() (unsigned int const index)
                    const
                {
100                 if (!index) return l;
                    return u;
                }

                inline real_type & operator[] (unsigned int const index)
105             {
                    if (!index) return l;
                    return u;
                }

110             inline real_type const & operator[] (unsigned int const index)
                    const
                {
                    if (!index) return l;
                    return u;
115             }

                interval & operator=(interval const &cpy) {
                    l = cpy.l;
                    u = cpy.u;
120                 return *this;
                }

            public:
                /*
125              * The following operators are implemented as they are in
                 * interval aritchmetic, see for example (18) in
                 * "Continuous Collision Detection for Rigid and Articulated Bodies"
                 * by Stephane Redon
                 *
130              * or
                 *
                 * (2)-(6) in
                 * "Interval Computations: Introduction, Uses, and Resources"
                 * by R.B. Kearfott
135              */
```

```
    bool operator<(interval const &i) const
    {
        return u < i.l;
    }

    bool operator>(interval const &i) const
    {
        return l > i.u;
    }

    bool operator<=(interval const &i) const
    {
        return u <= i.l;
    }

    bool operator>=(interval const &i) const
    {
        return l >= i.u;
    }

    bool operator==(interval const &i) const
    {
        return l == i.l && u == i.u;
    }

    bool operator!=(interval const &i) const
    {
        return l != i.l || u != i.u;
    }

    interval &operator+=(interval const &i)
    {
        l += i.l;
        u += i.u;
        return *this;
    }

    interval const operator+(interval const &i) const
    {
        return interval(l+i.l, u+i.u);
    }

    interval &operator-=(interval const &i)
    {
        l -= i.u;
        u -= i.l;
        return *this;
    }

    interval const operator-(const interval &i) const
    {
        return interval(l-i.u, u-i.l);
    }

    interval operator-() const
    {
        return interval(-u, -l);
    }

    interval &operator*=(real_type const &val)
    {
        if (val < (real_type)0.) {
            real_type x = l;
            l = u * val;
            u = x * val;
        } else {
            u *= val;
            l *= val;
        }

        return *this;
    }

    interval &operator*(real_type const &val)
    {
        if (val < (real_type)0.)
            return interval(u*val, l*val);
```

```
    return interval(l*val, u*val);
}

inline interval &operator*=(interval const &i)
{
    register real_type nl, nu; // -O4 probably makes register

    if (l <= 0.0 && i.u >= 0.0) {
        nl = l * i.u;
    } else if (u >= 0.0) {
        if (i.l <= 0.0) {
            nl = u*i.l;
        } else {
            nl = l*i.l;
        }
    } else {
        nl = u*i.u;
    }

    if (l <= 0.0 && i.l <= 0.0) {
        nu = l * i.l;
    } else if (u >= 0.0) {
        if (i.u >= 0.0) {
            nu = u * i.u;
        } else {
            nu = l * i.u;
        }
    } else {
        nu = u * i.l;
    }

    u = nu; l = nl;
    return *this;
}

/*
 *
 * "Redon"-version
 */
/*
inline interval operator*(interval const &i) const
{
    register real_type nl, nu, x; // -O4 probably makes register

    nu = nl = l*i.l;
    x = l*i.u; if (x < nl) nl = x; else if (x > nu) nu = x;
    x = u*i.l; if (x < nl) nl = x; else if (x > nu) nu = x;
    x = u*i.u; if (x < nl) nl = x; else if (x > nu) nu = x;

    return interval(nl, nu);
}
*/

/*
 * Optimized version, more IFs, two less mul
 */
inline interval operator*(interval const &i) const
{
    register real_type nl, nu;

    if (l <= 0.0 && i.u >= 0.0) {
        nl = l * i.u;
    } else if (u >= 0.0) {
        if (i.l <= 0.0) {
            nl = u*i.l;
        } else {
            nl = l*i.l;
        }
    } else {
        nl = u*i.u;
    }

    if (l <= 0.0 && i.l <= 0.0) {
        nu = l * i.l;
    } else if (u >= 0.0) {
        if (i.u >= 0.0) {
```

```
290                         nu = u * i.u;
                        } else {
                            nu = l * i.u;
                        }
                    } else {
295                     nu = u * i.l;
                    }

                    return interval(nl, nu);
                }
300
                interval const operator/=(interval const &i)
                {
                    // Assuming i.l > 0 || i.u < 0
                    *this *= interval(1.0 / i.u, 1.0 / i.l);
305                 return *this;
                }

                interval operator/(interval const &i) const
                {
310                 // Assuming i.l > 0 || i.u < 0
                    return *this * interval(1.0 / i.u, 1.0 / i.l);
                }

                real_type get_abs_lower() const
315             {
                    if (l >= 0.0) return l;              // -------0-[   ]- <-R
                    if (u >= 0.0) return (real_type)0.0; // -----[ 0 ]----- <-R
                    return -u;                           // -[   ]-0------- <-R
                }
320
                real_type get_abs_upper() const
                {
                    if (l+u >= 0.0) return u;            // -------0-[   ]- <-R
                                                         // -----[ 0 ]----- <-R
325
                    return -l;                           // -[   ]-0------- <-R
                }

        }; // class interval
330
#include <iosfwd>
    template<typename I>
        std::ostream &operator<<(std::ostream &o, interval<I> const &i) {
            o << "[" << i[0] << "," << i[1] << "]";
335         return o;
        }


    /*
340  * The following two operators was made to speed up
     * multiplication with intervals of type [x,x].
     *
     * The normal * in interval will use 3*six 'if's to do
     * this.
345  * This uses 3.
     */

#include <OpenTissue/math/vector.h>
    /*
350  * [[x1,x2],[y1,y2],[z1,z2]] * [x,y,z] =
     * [x1,x2]*x+[y1,y2]*y+[z1,z2]*z.
     *
     * This is used below.
     */
355 template<typename T>
        interval<T> operator*(vector3<interval<T> > const &vi,
                vector3<T> const &vr)
        {

360         register T min, max; // -O4 probably makes these register

            if (vr[0] < 0) {
                min = vi[0].upper()*vr[0];
                max = vi[0].lower()*vr[0];
365         } else {
                min = vi[0].lower()*vr[0];
```

```
                max = vi[0].upper()*vr[0];
            }

            if (vr[1] < 0) {
                min += vi[1].upper()*vr[1];
                max += vi[1].lower()*vr[1];
            } else {
                min += vi[1].lower()*vr[1];
                max += vi[1].upper()*vr[1];
            }

            if (vr[2] < 0) {
                min += vi[2].upper()*vr[2];
                max += vi[2].lower()*vr[2];
            } else {
                min += vi[2].lower()*vr[2];
                max += vi[2].upper()*vr[2];
            }

            return interval<T>(min, max);
        }

#include <OpenTissue/math/matrix.h>
    /*
     * Normal matrix*vector mul,
     * uses * operator above
     *
     * This is used when we rotate a point
     * with a bounded orientation matrix.
     */
    template<typename T>
        vector3<interval<T> > operator*(matrix3x3<interval<T> > const &m,
                vector3<T> const &v)
        {
            return vector3<interval<T> >(
                    m.m_row0 * v,
                    m.m_row1 * v,
                    m.m_row2 * v);
        }

} // namespace OpenTissue
#endif // header check
```

14.9 obb.h

```
#ifndef CONTCOLL_OBB_H
#define CONTCOLL_OBB_H

#include <OpenTissue/geometry/util/obb_fit.h>
#include <OpenTissue/geometry/plane.h>
#include "application.hpp"

/*
 * This is the base OBB top-down policy,
 * it is much like OT/collision/obb_tree/obb_tree_top_down_policy.h
 *
 * We extend this class in three classes, each with a unique
 * splitting policy;
 *
 *  * min-max, TODO
 *  * along OBB-axis,
 *  * min-sum. TODO
 *
 */
template<typename bvh_type, typename real_type_>
class obb_base_tree_top_down_policy
{
    public:

        typedef real_type_                                          real_type;
        typedef obb_base_tree_top_down_policy<bvh_type,real_type_>  top_down_type;
        typedef typename bvh_type::bv_ptr                           bv_ptr;
        typedef typename bvh_type::annotated_bv_type                annotated_bv_type;
        typedef typename bvh_type::annotated_bv_ptr                 annotated_bv_ptr;
        typedef typename bvh_type::volume_type                      volume_type;
```

```
        typedef typename bvh_type::geometry_type                    geometry_type;
        typedef typename std::vector<geometry_type>                 geometry_container;
        typedef OpenTissue::plane<real_type>                        plane_type;

public:

    geometry_container m_geometry;

public:

    class partition_type
    {
        public:

            friend class obb_base_tree_top_down_policy<bvh_type, real_type>;

        public:

            typedef std::vector<partition_type>             partition_container;
            typedef typename partition_container::iterator  partition_iterator;

        public:

            partition_container    m_sub_partitions;
            unsigned int           m_left;
            unsigned int           m_right;
            top_down_type        * m_owner;
            bv_ptr                 m_bv;

        public:

            partition_type()
                : m_left(0)
                , m_right(0)
                , m_owner(0)
                , m_bv()
                {}

            partition_type(top_down_type * owner,unsigned int left,
                    unsigned int  right)
                : m_left(left)
                , m_right(right)
                , m_owner(owner)
                , m_bv()
                {}

            bool annotated() const { return size()==1; }
            unsigned int size() const { return (m_right-m_left+1); }
            bool empty() { return (size()==0); }
            void split() { m_owner->split((*this)); }
            partition_iterator sub_partition_begin() {
                return m_sub_partitions.begin(); }
            partition_iterator sub_partition_end() { return m_sub_partitions.end(); }

            void fit(bv_ptr bv)
            {
                m_bv = bv;
                m_owner->fit(bv,(*this));
            }

    };

public:

    typedef typename partition_type::partition_container   partition_container;
    typedef typename partition_type::partition_iterator    partition_iterator;

protected:

    class vector3_iterator :
        public std::iterator<std::forward_iterator_tag, vector3_type>
    {
        protected:

            top_down_type * m_owner;
            unsigned int    m_idx;
```

```
                unsigned int    m_sub_idx;

            public:

                vector3_iterator(top_down_type * owner,unsigned int idx)
                    : m_owner(owner)
                    , m_idx(idx)
                    , m_sub_idx()
                    {}

            public:

                bool operator==(vector3_iterator const& other) const {
                    return (m_idx==other.m_idx && m_sub_idx==other.m_sub_idx); }
                bool operator!=(vector3_iterator const& other) const {
                    return !(m_idx==other.m_idx && m_sub_idx==other.m_sub_idx); }

                vector3_iterator & operator++()
                {
                    if(m_sub_idx==3)
                    {
                        ++m_idx;
                        m_sub_idx = 0;
                    }
                    else
                    {
                        ++m_sub_idx;
                    }
                    return *this;
                }

                vector3_type & operator*() const
                {
                    face_ptr_type  f = m_owner->m_geometry[m_idx];
                    if(m_sub_idx==0)
                        return *(f->m_v0);
                    else if (m_sub_idx==1)
                        return *(f->m_v1);
                    return *(f->m_v2);
                }

        };

    public:

        partition_type all() { return partition_type(this,0,m_geometry.size()-1); }

        template<typename iterator>
            void init(iterator begin,iterator end)
            {
                m_geometry.clear();
                std::copy( begin, end, std::back_inserter( m_geometry ) );
            }

    public:

        virtual ~obb_base_tree_top_down_policy() {}

        unsigned int degree() const {return 2;}

        void split(partition_type & partition)
        {
            using std::min;

            assert(partition.m_bv);
            obb_type & obb = partition.m_bv->volume();
            vector3_type  center = obb.center();

            real_type value = obb.ext()(0);
            unsigned int column = 0;

            if(obb.ext()(1) > value)
            {
                value = obb.ext()(1);
                column = 1;
            }
            if(obb.ext()(2) > value)
```

```
            {
                column = 2;
            }

            vector3_type  axis   = vector3_type(
                    obb.orientation()(0,column),
                    obb.orientation()(1,column),
                    obb.orientation()(2,column));
            plane_type    split_plane(axis,center);

            unsigned int first = partition.m_left;
            unsigned int last  = partition.m_right;
            unsigned int i     = first;
            for(;i<last;)
            {
                mesh_compute_face_center( (*this->m_geometry[i]), center);
                if(split_plane.get_signed_distance(center) > 0)
                {
                    swap(this->m_geometry[i],this->m_geometry[last]);
                    --last;
                }
                else
                {
                    ++i;
                }
            }
            unsigned int sub_right = min(last+1,partition.m_right);
            unsigned int sub_left  = sub_right - 1;

            partition.m_sub_partitions.resize(2);
            partition.m_sub_partitions[0] =
                partition_type( this, partition.m_left, sub_left          );
            partition.m_sub_partitions[1] =
                partition_type( this,   sub_right     , partition.m_right );
        }

        void fit(bv_ptr bv,partition_type & partition)
        {
            partition.m_bv = bv;

            if(partition.annotated())
            {
                annotated_bv_ptr A = boost::static_pointer_cast<annotated_bv_type>(bv);
                A->insert( m_geometry[partition.m_left] );
            }
            vector3_iterator begin( this, partition.m_left     );
            vector3_iterator end(   this, partition.m_right+1  );

            // Fit OBB
            obb_fit(begin,end,bv->volume());

            // Fit sphere
            real_type rad2 = OpenTissue::Math::lowest<real_type>();
            for (vector3_iterator i=begin;i!=end;++i) {
                vector3_type &v = *i;

                real_type dist2 =
                    (v(0) - bv->volume().center()(0))*
                    (v(0) - bv->volume().center()(0))+
                    (v(1) - bv->volume().center()(1))*
                    (v(1) - bv->volume().center()(1))+
                    (v(2) - bv->volume().center()(2))*
                    (v(2) - bv->volume().center()(2));

                if (dist2 > rad2) rad2 = dist2;
            }

            using std::sqrt;
            bv->m_bounding_sphere.set(bv->volume().center(), sqrt(rad2));
        }
};

template<typename mesh_type, typename bvh_type, typename constructor_type>
void create_obb_tree(mesh_type& mesh, bvh_type& tree, constructor_type &con)
{
    unsigned int F = mesh.size_faces();
```

```
260         std::vector<face_ptr_type> geometry(F,0);
            typename mesh_type::face_iterator f     = mesh.face_begin();
            typename mesh_type::face_iterator f_end = mesh.face_end();
            for(unsigned int i=0;f!=f_end;++f,++i)
            {
265             geometry[i] = &(*f);
                assert(valency(*f) || !"create_obb_tree(): Only triangle faces are supported");
                typename mesh_type::face_vertex_circulator v(*f);
                f->m_v0 = &(v->m_coord);
                f->m_n0 = &(v->m_normal);
270             ++v;
                f->m_v1 = &(v->m_coord);
                f->m_n1 = &(v->m_normal);
                ++v;
                f->m_v2 = &(v->m_coord);
275             f->m_n2 = &(v->m_normal);
            }

            tree.clear();
            con.run(geometry.begin(), geometry.end(), tree);
280
        }

        #endif
```

Litteratur

[1] F. Antonio. Faster line segment intersection. *Graphics Gems III*, pages 199–202 og 500–501, 1992.

[2] Gino Van Den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Department of Mathematics and Computing Science, Eindhoven University of Technology*, 1998.

[3] Erik B. Dam, Martin Koch, and Martin Lillholm. Quaternions, interpolation and animation, 1998.

[4] S. Gottschalk. Collision queries using oriented bounding boxes, 1998.

[5] S. Gottschalk, M.C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. *Proc. SIGGRAPH*, pages 171–180, 1996.

[6] S. Graham, P. Kessler, and M. McKusick. Gprof: A call graph execution profiler. *Proceeedings of the SIGPLAN '82 Symposium on Compiler Construction*, 17(6):120–126, June 1982.

[7] Knud Henriksen Kenny Erleben, Jon Sporring and Henrik Dohlmann. *Phycics-based Animation*. Charles River Media, 2005.

[8] OpenTissue. Opensource Project, Physical based Animation and Surgery Simulation. http://www.opentissue.org.

[9] S. Redon, A. Kheddar, and S. Coquillart. Fast continuous collision detection between rigid bodies, 2002.

[10] Stephane Redon. Continuous collision detection for rigid and articulated bodies. *Department of Computer Science, University of North Carolina at Chapel Hill*, 2004.

[11] Stephane Redon, Young J. Kim, Ming C. Lin, and Dinesh Manocha. Fast continuous collision detection for articulated models, 2004.