UNIVERSITETET I OSLO
Institutt for informatikk

Visualisering av trianguleringer og triangulerings-prosesser

Hovedfagsoppgave

Per-Idar Evensen
(periev@ifi.uio.no)

November 2004

Forord

Denne hovedfagsrapporten er skrevet som en del av min hovedfagsoppgave til Cand. Scient. graden ved Universitetet i Oslo. Arbeidet med denne hovedfagsoppgaven har vært utført ved Simula Research Laboratory og Institutt for informatikk i perioden fra mars 2002 til november 2004. Jeg vil først og fremst takke veilederne mine Morten Dæhlen og Øyvind Hjelle for nyttig veiledning og inspirasjon. Jeg vil også takke Thomas Sevaldrud for all hjelp underveis, spesielt i startfasen.

Målet med denne hovedfagsoppgaven har vært å utvikle en interaktiv trianguleringsapplikasjon, som vi har kalt for *TriangTutor*. Det er laget en egen hjemmeside til denne applikasjonen. Her finnes selve programmet med kildekode, samt en del demoanimasjoner og bilder. Hjemmesiden til TriangTutor er foreløpig å finne under mine egne hjemmesider (http://heim.ifi.uio.no/~periev/triangtutor), men vil etter hvert også bli å finne under hjemmesidene til trianguleringsbiblioteket TTL – The Triangulation Template Library (http://www.simula.no/ogl/ttl). Ved siden av å anvende TTL i denne hovedfagsoppgaven, er det også jeg som har jobbet med å tilrettelegge TTL for å lansere dette biblioteket som Open Source Software (OSS). TriangTutor med kildekode og animasjoner finnes også på en CD-ROM som følger med denne hovedfagsrapporten.

Oslo, 1. november 2004

Per-Idar Evensen

Innhold

Kapittel 1

Innledning

Vi vil begynne dette første kapitlet med en beskrivelse av selve hovedfagsoppgaven. Deretter vil vi gi en kort presentasjon av de øvrige kapitlene. Vi vil se på hvordan disse er organisert og hva leseren kan vente å finne i de forskjellige kapitlene.

1.1 Beskrivelse av oppgaven

Bakgrunnen for denne hovedfagsoppgaven var ønsket om en *interaktiv trianguleringsapplikasjon* som primært skulle brukes i undervisnings-sammenheng. Denne applikasjonen skulle ta utgangpunkt i pensumet som undervises i trianguleringskurset ved Institutt for Informatikk, INF-MAT5370 (tidligere INF-TT), og være et hjelpemiddel til å illustrere og demonstrere sentrale deler av dette. I tillegg skulle applikasjonen anvende *trianguleringsbiblioteket TTL* (Triangulation Template Library) og dermed også være en slags demoapplikasjon for dette biblioteket. Arbeidet med denne hovedoppgaven startet like etter at Simula Research Laboratory flyttet inn på Fornebu, og det var også ønskelig at applikasjonen skulle kunne kjøres på visningsveggen som ble bygget der.

Dette skulle hovedsakelig være en oppgave innenfor visualisering. Applikasjonen skulle visualisere Delaunay-trianguleringer i planet, med og uten føringer, som blir beregnet av TTL. I tillegg skulle vi plukke ut noen andre konsepter relatert til Delaunay-trianguleringer, som for eksempel Voronoi-diagram og omskrivende sirkler, og visualisere disse. Applikasjonen skulle bruke OpenGL til å generere datagrafikken.

Det vi har skrevet til nå, er stort sett de rammene eller kravene vi startet

med når vi begynte utviklingen av applikasjonen. Hovedproblemstillingen i oppgaven har vært å finne hensiktsmessige måter å visualisere Delaunay-trianguleringer og teorien rundt disse. Vi har valgt å visualisere Voronoi-diagram, omskrivende sirkler og influensregionen til nyinnsatte noder, i tillegg til selve Delaunay-trianguleringene. Applikasjonen vi har laget har, som vi skal se, høy grad av brukerinteraksjon. Den har blant annet en nyttig funksjon hvor brukeren kan ta tak i og flytte en node rundt i trianguleringen, som da oppdateres fortløpende i sanntid. Dette gir brukeren mulighet til fritt å utforske forskjellige situasjoner som kan oppstå i en Delaunay-triangulering.

Vi har kalt applikasjonen for *TriangTutor*, fordi den først og fremst er en applikasjon som skal brukes til å lære trianguleringsteori. Det er meningen at TriangTutor både skal kunne brukes av en lærer eller foreleser til å illustrere og demonstrere teori og av studenter for å prøve ut forskjellige situasjoner og kanskje få en bedre forståelse av teorien. Det er lagt vekt på at applikasjonen skal være brukervennlig og enkel å ta i bruk. Den er dessuten plattformuavhengig, og det foreligger både en Windows-versjon og en Linux/Unix-versjon. Det er også lagt vekt på å lage en godt strukturert kode som er fyldig kommentert, slik at hele eller deler av denne kan brukes som basis for nye hovedoppgaver.

Det finnes mange forskjellige interaktive trianguleringsapplikasjoner med liknende funksjonalitet som TriangTutor. Noen av disse er laget som java-appleter, som for eksempel VoroGlide [VOR], utviklet ved universitetet i Bonn, og Mocha [MOC].

1.2 Organisering av kapitlene

Hoveddelen i denne hovedfagsrapporten, som består av de seks neste kapitlene, er delt inn i to deler som er omtrent like store. Den første delen tar for seg teorien og bakgrunnsstoffet som vi har basert applikasjonen på, og den andre delen beskriver selve applikasjonen.

Teoridelen består av tre kapitler. Det første kapitlet, som er Kapittel 2, tar for seg grunnleggende trianguleringsteori. Vi definerer her Delaunay-triangulering og Voronoi-diagram og ser på sammenhengen mellom disse. I Kapittel 3 beskriver vi TTL, samt litt teori som er relevant for å bruke dette biblioteket. Kapittel 4, som er det siste teorikapitlet, handler om datagrafikk og OpenGL.

Den delen som beskriver TriangTutor er også delt inn i tre kapitler. De

første to kapitlene, Kapittel 5 og Kapittel 6, beskriver henholdsvis grunnleggende og mer avansert funksjonalitet i programmet. Vi tar her for oss alt brukeren kan gjøre i TriangTutor og ser på hvordan dette er implementert. Kapittel 7, som er det siste kapitlet i denne delen, beskriver hvordan koden er organisert og hvilke klasser den består av.

Til slutt i hovedfagsrapporten er det et kort oppsummerings- og konklusjonskapittel. I dette kapitlet tar vi et tilbakeblikk på arbeidet med TriangTutor og kommer med noen kommentarer på hvordan resultatet har blitt. Vi presenterer også noen ideer til videre utvidelser av TriangTutor.

Kapittel 2

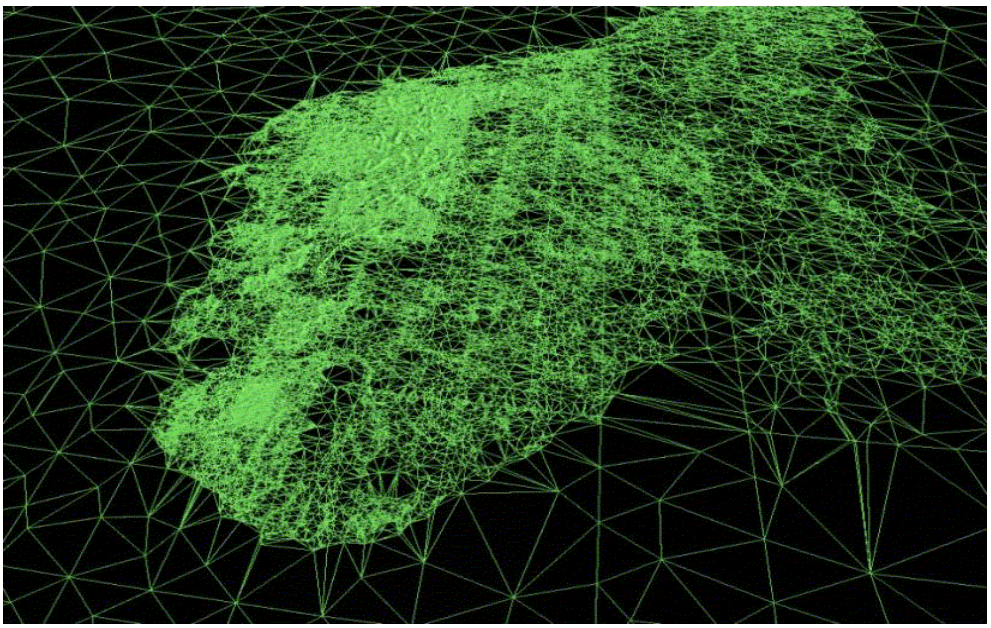
Trianguleringsteori

Dette kapitlet gir en kort innføring i trianguleringsteori, og beskriver hovedsakelig den trianguleringsteorien som er relevant for denne hovedoppgaven. Vi gir først en introduksjon som beskriver hva en triangulering er og hva trianguleringer kan brukes til. Videre ser vi på noen grunnleggende egenskaper hos trianguleringer i planet i seksjon 2.2, før vi definerer Delaunay-triangulering og Voronoi-diagram i seksjon 2.3. Til slutt ser vi på Delaunay-trianguleringer med føringer i seksjon 2.4. Teorien i dette kapitlet er hentet fra [Pre85], [Ber00] samt [Hje03], og er mer utfyllende og grundig presentert der.

2.1 Introduksjon til trianguleringer

Å triangulere vil si å dele opp geometriske flater eller plane polygoner i trekanter. Dette har mange praktiske fordeler, særlig når vi ønsker å representere en flate i en datamaskin. Bruk av trianguleringer og trekant-baserte flater er derfor svært utbredt og benyttes i mange forskjellige typer applikasjoner. De brukes innenfor kartografi, i digitale terrengmodeller og i geografiske informasjonssystemer (GIS), hovedsakelig for å representere deler av jordoverflaten. Vi finner dem i elementmetoden (FEM, finite element methods) for løsing av partielle differensiallikninger, som basis for numeriske beregninger og simuleringer, og i datamaskinassisterte tegne- og designsystemer (CAD). Sist, men ikke minst, er trianguleringer svært mye brukt innenfor datagrafikk og visualisering.

Innenfor mange områder er det vanlig å hente inn måledata i form av spredte punkter på overflaten til et fysisk objekt. Slike data kan man få fra



Figur 2.1: Hierarkisk triangulering av Sør-Norge. (Rune Aasgaard/Thomas E. Sevaldrud, Metaphor Project)

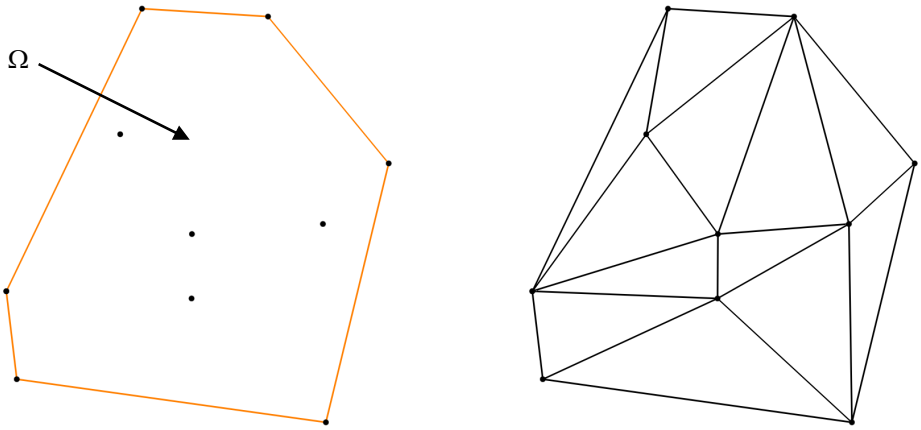
forskjellige typer skannere eller for eksempel fra en satellitt, hvis objektet det måles på er jorden. Ved hjelp av disse målepunktene er det ofte ønskelig å konstruere den underliggende geometrien til objektet i en datamaskin, og dette kan gjøres ved hjelp av en triangulering eller trekantbasert flate. Et eksempel på en triangulering av geografiske data ser vi på *Figur 2.1*, som viser en hierarkisk triangulering av Sør-Norge.

2.2 Trianguleringer i planet

I denne hovedoppgaven er det først og fremst trianguleringer i planet som står i sentrum. Generelt starter vi da med en *punktmengde* i planet,

$$P = \{p_1, p_2, \ldots, p_N\}, \text{ der } p_i = (x_i, y_i),$$

og et *domene* Ω som inneholder alle punktene i P. Vi antar at randen til Ω består av et eller flere *enkle* lukkede polygoner. At et polygon er enkelt betyr at det ikke skjærer seg selv. Ofte er Ω den *konvekse innhylningen* til P,



Figur 2.2: Et domene Ω med punkter i planet (a), og en mulig triangulering av disse (b).

som for eksempel i Delaunay-trianguleringer, men den behøver ikke nødvendigvis å være konveks. En triangulering Δ av punktene i P er grovt sagt en oppdeling av Ω i trekanter t_i,

$$\Delta(P) = \{t_1, t_2, \ldots, t_T\},$$

som ikke overlapper hverandre og hvor hjørnene i alle trekantene t_i er punktene i P. *Figur 2.2* (*a*) viser et domene Ω med punkter i planet, og *Figur 2.2* (*b*) viser en mulig triangulering av disse.

Før vi går videre og ser på noen grunnleggende egenskaper for trianguleringer, trenger vi litt notasjon. En trekant i en triangulering Δ, som utspennes av nodene v_i, v_j og v_k assosiert med punktene p_i, p_j og p_k, betegnes $t_{i,j,k}$. Indeksene i, j og k ordnes slik at de tilsvarende nodene orienteres *mot klokka*. En sidekant eller kant mellom to noder v_i og v_j, betegnes $e_{i,j}$ eller ekvivalent $e_{j,i}$.

For at en triangulering Δ skal regnes som *gyldig*, må følgende fire krav være oppfylt:

(1) Δ må ikke inneholde noen degenererte trekanter, det vil si at for en trekant $t_{i,j,k}$ kan ikke punktene p_i, p_j og p_k ligge på samme rette linje.

(2) Det indre av to trekanter $t_{i,j,k}$ og $t_{l,m,n}$ i Δ kan ikke overlappe, det vil si

$$Int(t_{i,j,k}) \cap Int(t_{l,m,n}) = \phi.$$

(3) Snittet mellom to trekanter $t_{i,j,k}$ og $t_{l,m,n}$ i Δ $t_{i,j,k} \cap t_{l,m,n}$ må være enten en felles sidekant, en felles node eller tomt.

(4) Unionen av alle trekantene i Δ må dekke hele domenet Ω, det vil si

$$\bigcup_{i=1}^{T} t_i = \Omega.$$

Videre kalles en triangulering Δ *regulær* hvis den i tillegg oppfyller følgende tre krav:

(5) Domenet Ω, og dermed også Δ, må være sammenhengende.

(6) Δ (og domenet Ω) kan ikke inneholde noen hull.

(7) Hvis v_i er en randnode i Δ, så finnes det nøyaktig to randkanter som møtes i v_i. Dette innebærer at antall randnoder er lik antall randkanter.

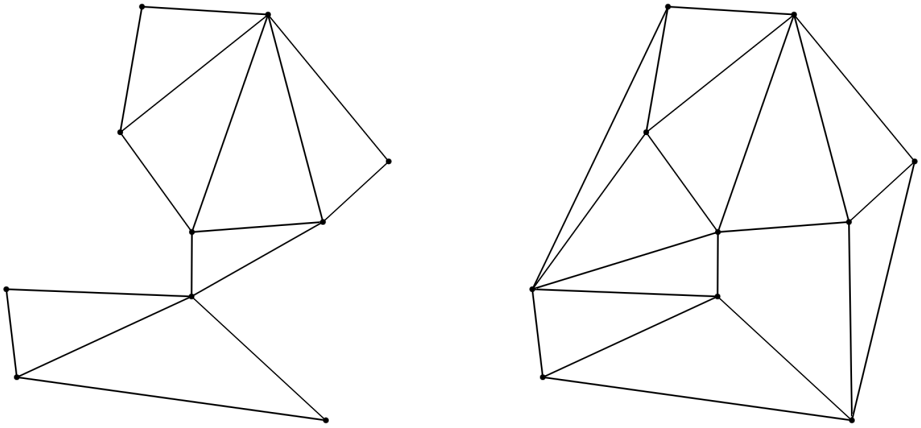
Trianguleringen på *Figur 2.2* (*b*) er gyldig og regulær i henhold til definisjonen over. *Figur 2.3* (*a*) viser en triangulering som er gyldig men ikke regulær, siden mer enn to randkanter møtes i en randnode. På *Figur 2.3* (*b*) ser vi en triangulering som ikke er gyldig på grunn av krav (4), under forutsetning av at domenet trianguleringen er definert over er det samme som vist på *Figur 2.2* (*a*).

La V, E og T betegne henholdsvis antall noder, antall sidekanter og antall trekanter i en triangulering, og la subskriptene $_B$ og $_I$ betegne henholdsvis randelementer og indre elementer. For en regulær triangulering Δ har vi da følgende tre relasjoner:

$$T = 2V_I + V_B - 2, \tag{2.1}$$

$$E = 3V_I + 2V_B - 3, \tag{2.2}$$

$$E_I = 3V_I + V_B - 3. \tag{2.3}$$



(a) (b)

Figur 2.3: En gyldig triangulering som ikke er regulær (a), og en triangulering som ikke er gyldig (b).

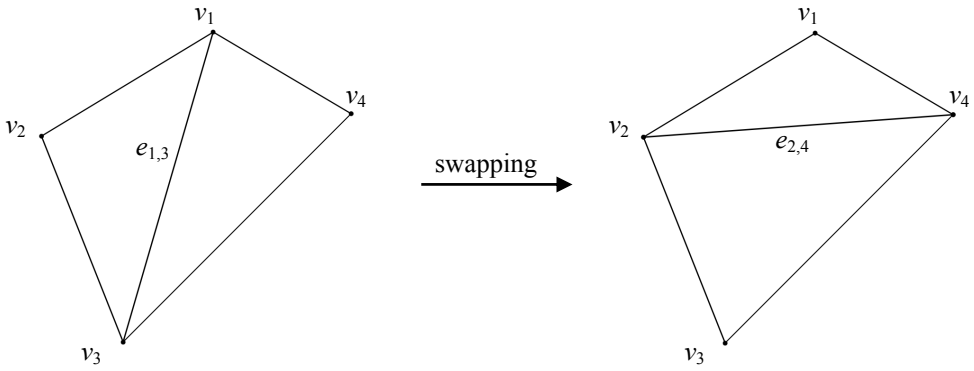
Kombinerer vi (2.2) og (2.3) får vi *Euler-Poincaré* formelen

$$T = E - V + 1. \tag{2.4}$$

Vi definerer *graden* til en node v_i, $\deg(v_i)$, i en triangulering Δ som antall kanter som møtes i v_i, og for en triangulering Δ (regulær eller ikke) har vi at

$$\sum_{i=1}^{V} \deg(v_i) = 2E. \tag{2.5}$$

For en gitt punktmengde P i planet, som inneholder mer enn tre punkter, finnes det flere forskjellige trianguleringer $\Delta(P)$. En enkel måte å komme fra en triangulering $\Delta(P)$ til en annen triangulering $\Delta'(P)$, er å *swappe* eller flippe en eller flere indre sidekanter i $\Delta(P)$. Swapping er en operasjon som kan utføres på sidekanter som er diagonaler i strengt konvekse kvadrilateral, som vist på *Figur 2.4*. Har man først funnet en triangulering $\Delta(P)$, så kan det vises at alle andre mulige trianguleringer av P med samme rand, kan nås gjennom en sekvens med ombyttinger av sidekanter [Law72]. Mange trianguleringsalgoritmer baserer seg på swapping av sidekanter, og det finnes forskjellige kriterier for å avgjøre om en kant skal swappes eller ikke. Delaunay-kriteriet, som beskrives i neste seksjon, er et slikt kriterium.



Figur 2.4: Swapping av en kant i et strengt konvekst kvadrilateral.

En triangulering Δ kan også innholde *føringer*, det vil si predefinerte sidekanter som må være med i Δ. Disse kan for eksempel brukes til å representere elver, veier eller geologiske forkastninger i terrengmodeller. Teorien for Delaunay-trianguleringer er utvidet til å håndtere føringer, og seksjon 2.4 tar for seg dette.

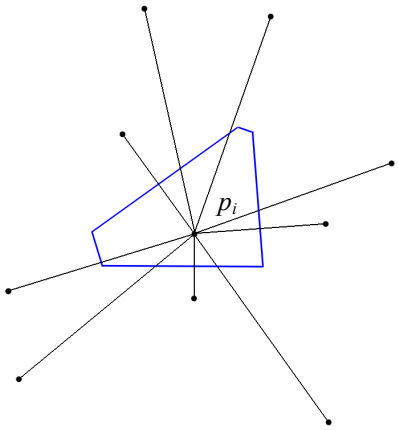
2.3 Delaunay-trianguleringer og Voronoi-diagram

Siden en punktmengde i planet kan trianguleres på mange forskjellige måter, trenger vi kriterier å triangulere etter. Som oftest er det ønskelig å unngå trekanter som er lange og smale eller nesten degenererte, siden disse kan gi numeriske problemer. Av alle mulige trianguleringer vil man derfor ofte foretrekke en triangulering der trekantene er mest mulig likesidede. En måte å oppnå dette på, er å velge en triangulering hvor den minste vinkelen i trianguleringen er størst mulig. Dette kalles for *MaxMin*-vinkel kriteriet.

Vi kan til enhver mulig triangulering $\Delta^k(P)$ av en punktmengde P, med samme rand, tilordne en *indikatorvektor*,

$$I(\Delta^k) = (\alpha_1, \alpha_2, \ldots, \alpha_T),\ \alpha_i \leq \alpha_j,\ i < j,$$

der α_i er den minste vinkelen i trekant t_i. Elementene i indikatorvektoren sorteres i stigende rekkefølge. Vi sier at en vektor $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ er *leksikografisk* større enn en vektor $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, dersom vi for et heltall



Figur 2.5: Voronoi-regionen til et punkt i planet.

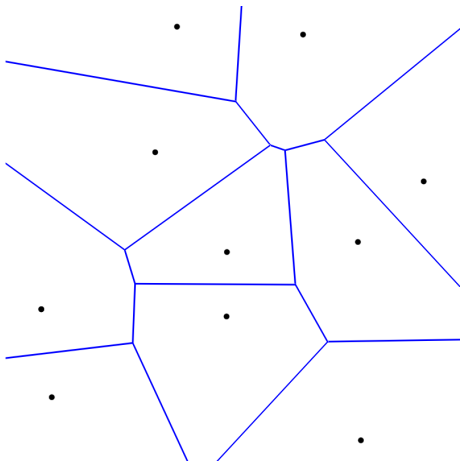
m har at $u_i = v_i$ for $i = 1, 2, \ldots, m-1$, mens $u_m > v_m$. En triangulering Δ^i med en indikatorvektor $I(\Delta^i)$ som er leksikografisk større enn en triangulering Δ^j med en indikatorvektor $I(\Delta^j)$, vil nå ha flere trekanter som er omtrent likesidede enn Δ^j. Den optimale trianguleringen i henhold til MaxMin-vinkel kriteriet vil være den med størst mulig indikatorvektor, og ut i fra dette kan vi nå gi en av flere ekvivalente definisjoner på en *Delaunay-triangulering*:

> **Definisjon 2.1:** (Delaunay-triangulering, MaxMin-vinkel kriteriet) *En triangulering Δ av en punktmengde P i planet som er optimal i henhold til MaxMin-vinkel kriteriet, og som er definert på den konvekse innhylningen til P, kalles en Delaunay-triangulering av P.*

Trianguleringen på *Figur 2.2* (*b*) er en Delaunay-triangulering. Det finnes alltid minst en slik optimal triangulering siden antall mulige trianguleringer av P er endelig, men denne er ikke nødvendigvis entydig. Vi kan ha trianguleringer hvor de samme vinklene opptrer selv om diagonalen i et kvadrilateral swappes. Dette kalles et *nøytralt tilfelle*, og vi skal snart se nærmere på dette.

En annen definisjon av en Delaunay-triangulering baserer seg på *Voronoi-diagram*, så la oss først se litt nærmere på dette. Anta at vi har en mengde med distinkte punkter i planet

$$P = \{p_1, p_2, \ldots, p_N\},$$



Figur 2.6: Voronoi-diagrammet til en punktmengde i planet.

og la $d(p_i, p_j)$ være den Euklidske avstanden mellom p_i og p_j. Til hvert punkt p_i i P kan vi tilordne et område i planet

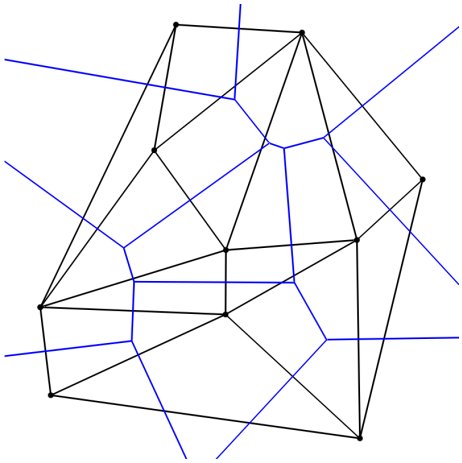
$$V(p_i) = \{x \mid d(x, p_i) \leq d(x, p_j), j = 1, 2, \ldots, N\},$$

som kalles *Voronoi-regionen* til p_i. Det indre av $V(p_i)$ består av alle punktene x i planet som ligger nærmere p_i enn noen av de andre punktene i P. Et eksempel på en Voronoi-region er vist på *Figur 2.5*. Dersom vi setter sammen Voronoi-regionene til alle punktene i P, får vi *Voronoi-diagrammet* eller Dirichlet-tesselleringen til P

$$VD(P) = \bigcup_{i=1}^{N} V(p_i).$$

Voronoi-diagrammet til en punktmengde P dekker hele planet R^2 og er alltid *entydig*. *Figur 2.6* viser en punktmengde og det tilsvarende Voronoi-diagrammet. Kantene i et Voronoi-diagram kalles *Voronoi-kanter*, og nodene kalles *Voronoi-punkter*. To punkter p_i og p_j kalles *Voronoi-naboer* hvis deres respektive Voronoi-regioner, $V(p_i)$ og $V(p_j)$, deler en felles Voronoi-kant.

Dersom vi trekker rette linjestykker mellom alle Voronoi-naboene i P, som vist på *Figur 2.7*, får vi en Delaunay-triangulering. Vi kan ut i fra dette formulere en annen definisjon på en Delaunay-triangulering:



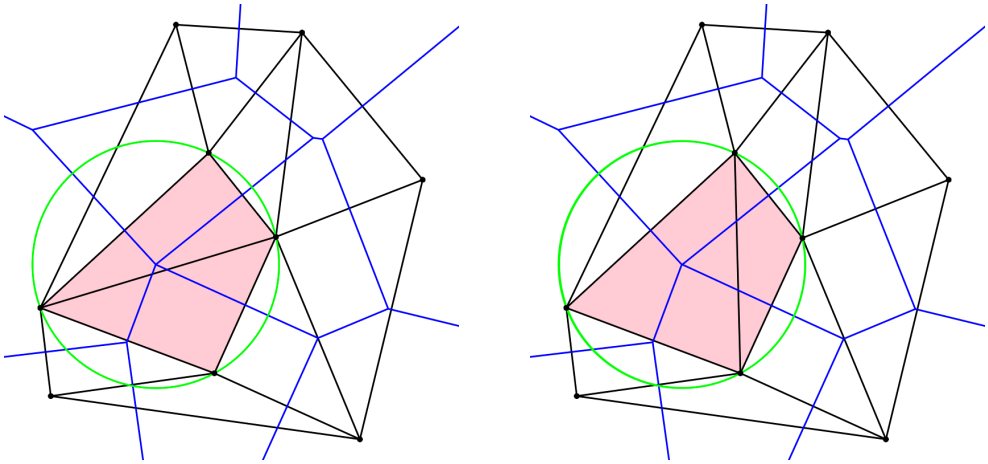
Figur 2.7: Rettlinjedualen til et Voronoi-diagram gir en Delaunay-triangulering.

Definisjon 2.2: (Delaunay-triangulering, rett linje dual) *En Delaunay-triangulering Δ av en punktmengde P i planet, er rettlinjedualen til Voronoi-diagrammet til P.*

Kantene mellom Voronoi-naboene i P kalles *Delaunay-kanter*, og trekantene som dannes kalles *Delaunay-trekanter*.

Selv om Voronoi-diagrammet til en punktmengde P alltid er entydig, er ikke nødvendigvis den avledede Delaunay-trianguleringen det. Dersom det finnes fire eller flere punkter fra P som ligger på samme omskrivende sirkel rundt en trekant, eller ekvivalent at fire eller flere Voronoi-kanter møtes i et Voronoi-punkt, så har vi et nøytralt tilfelle. Det finnes da flere alternative måter å lage en Delaunay-triangulering på. *Figur 2.8* viser et eksempel på et nøytralt tilfelle. Vi ser her to forskjellige Delaunay-trianguleringer av samme punkmengde. I et slikt nøytralt tilfelle kalles to punkter p_i og p_j, hvor de respektive Voronoi-regionene $V(p_i)$ og $V(p_j)$ kun deler et felles Voronoi-punkt, for *svake* Voronoi-naboer. Voronoi-naboer i ikke-nøytrale tilfeller, hvor de respektive Voronoi-regionene deler en felles Voronoi-kant, kalles *sterke* Voronoi-naboer.

Et Voronoi-punkt er ekvidistant fra tre sterke Voronoi-naboer i P, og er dermed senteret i den *omskrivende sirkelen* til trekanten som dannes av de tre Voronoi-naboene. Hvert Voronoi-punkt kan derfor assosieres med en trekant i Delaunay-trianguleringen av P. I et nøytralt tilfelle er et Voronoi-punkt ekvidistant fra fire eller flere svake Voronoi-naboer, og kan derfor



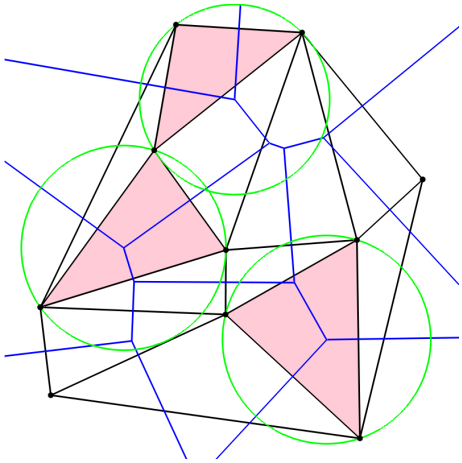
Figur 2.8: Nøytralt tilfelle med to forskjellige Delaunay-trianguleringer av samme punktmengde P.

assosieres med to eller flere Delaunay-trekanter. Dette kan vi bruke til å beregne Voronoi-diagrammet for en gitt Delaunay-triangulering. *Figur 2.9* viser Delaunay-trianguleringen og Voronoi-diagrammet for en punktmengde P, samt den omskrivende sirkelen til noen av trekantene. Midtpunktene i sirklene sammenfaller med Voronoi-punktene. En omskrivende sirkel til en trekant i en Delaunay-triangulering av en punktmengde P har den viktige egenskapen at den ikke omslutter noen punkter fra P. Denne egenskapen kalles *sirkelkriteriet*, og den gir oss en tredje definisjon på en Delaunay-triangulering:

> **Definisjon 2.3:** (Delaunay-triangulering, sirkelkriteriet) *En Delaunay-triangulering Δ av en punktmengde P i planet er en triangulering der ingen omskrivende sirkel av noen trekant i Δ inneholder noen punkter fra P.*

Det er denne siste definisjonen som er den mest brukte definisjonen på en Delaunay-triangulering.

Vi har nå gitt tre ulike definisjoner på en Delaunay-triangulering med hvert sitt kriterium. Det kan vises at alle disse tre kriteriene er ekvivalente [Hje03].



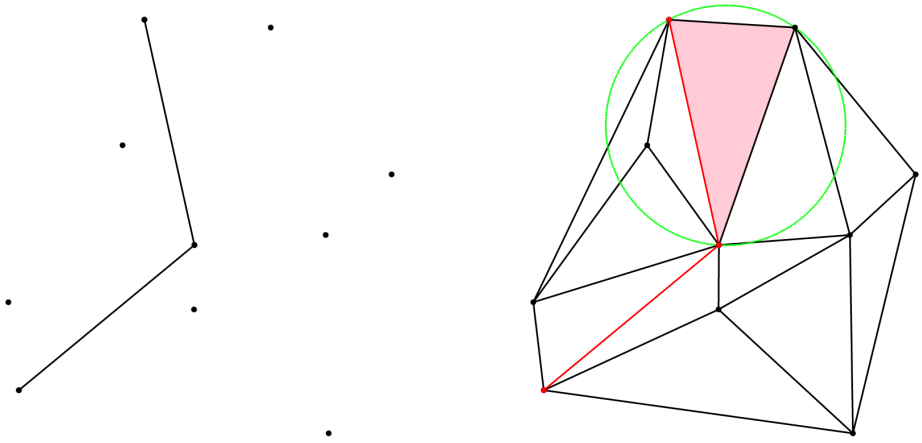
Figur 2.9: Voronoi-diagrammet og Delaunay-trianguleringen, med noen omskrivende sirkler, for en punktmengde P.

2.4 Delaunay-trianguleringer med føringer

En triangulering Δ med føringer innholder, som nevnt tidligere, predefinerte sidekanter som må være med i Δ. Føringer kan for eksempel brukes til å definere elver, veier eller randen på innsjøer i terrengmodeller.

Teorien for Delaunay-trianguleringer kan generaliseres til å håndtere føringer. Vi får da en *Delaunay-triangulering med føringer* eller en *CDT* (Constrained Delaunay triangulation). Med denne teorien kan vi også lage mer generelle trianguleringer som inneholder hull, og som har en ytre rand som ikke er konveks, mens vi fortsatt bevarer Delaunay-egenskapene i de indre delene av trianguleringen.

For å utvide teorien vi så på i forrige seksjon til å kunne håndtere føringer, må vi innføre begrepet *synlighet*. Vi antar nå at vi i tillegg til en punktmengde P i planet, også har en mengde med predefinerte sidekanter E_c. Endepunktene til kantene i E_c må være inneholdt i P, og kantene i E_c må være en delmengde av kantene i den endelige trianguleringen $\Delta(P, E_c)$. To punkter p_i og p_j er *synlige* for hverandre dersom det rette linjestykket mellom p_i og p_j ikke skjærer det indre av noen kanter i E_c eller noe område som ikke er triangulert, som for eksempel et hull eller et område utenfor den



(a) (b)

Figur 2.10: En punktmengde P og en mengde med predefinerte sidekanter E_c (a), og en Delaunay-triangulering med føringer $\Delta(P, E_c)$ av P og E_c.

ytre randen til trianguleringen. Ved å modifisere sirkelkriteriet i *Definisjon 2.3* av en Delaunay-triangulering til å inkludere synlighet, kan vi nå gi en definisjon på en Delaunay-triangulering med føringer:

> **Definisjon 2.4:** (Delaunay-triangulering med føringer, modifisert sirkelkriterium) *En Delaunay-triangulering med føringer* $\Delta(P, E_c)$ *av* P *og* E_c *er en triangulering som inneholder sidekantene i* E_c*, og som er slik at ingen omskrivende sirkel av en trekant* t *i* $\Delta(P, E_c)$ *inneholder noen punkter fra* P *i det indre som er synlige fra alle tre nodene til* t*.*

Figur 2.10 (*a*) viser en punktmengde P og en mengde med predefinerte sidekanter E_c, og *Figur 2.10* (*b*) viser en Delaunay-triangulering med føringer $\Delta(P, E_c)$ av P og E_c. En omskrivende sirkel er også tegnet inn for å illustrere det modifiserte sirkelkriteriet. Noden innenfor den omskrivende sirkelen er ikke synlig fra alle tre nodene til den merkede trekanten.

I forrige seksjon hadde vi også to andre ekvivalente definisjoner på en Delaunay-triangulering. Ser vi på *Definisjon 2.1*, kan denne overføres til også å gjelde for Delaunay-trianguleringer med føringer, når vi begrenser alle mulige trianguleringer til å være de som inneholder kantene i E_c. *Definisjon 2.2* derimot, som baserer seg på Voronoi-diagram, kan ikke uten videre overføres til også å gjelde for Delaunay-trianguleringer med føringer.

Kapittel 3

Trianguleringsbiblioteket TTL

Etter først å ha sett på grunnleggende trianguleringsteori, vil vi i dette kapitlet se på hvordan trianguleringer kan bygges opp og behandles i datamaskiner. Først ser vi generelt på det å programmere trianguleringer. Deretter introduserer vi generaliserte maps eller G-maps i seksjon 3.2, som det generiske trianguleringsbiblioteket TTL baserer seg på. Selve TTL, som er det trianguleringsbiblioteket som anvendes i denne hovedoppgaven, beskrives i seksjon 3.3. Seksjon 3.4 tar for seg inkrementell oppbygning av Delaunay-trianguleringer, som er måten trianguleringer lages på i TTL. I seksjon 3.5 gir vi en beskrivelse av halvkant-datastrukturen for trianguleringer. Det er denne datastrukturen som anvendes i denne hovedoppgaven. Til slutt ser vi litt på generiske trianguleringsbibliotek generelt i seksjon 3.6. TTL og den generiske programmeringsfilosofien som brukes i dette trianguleringsbiblioteket, er mer grundig beskrevet i rapporten [Hje00].

3.1 Å programmere trianguleringer

For å bygge opp og behandle trianguleringer i en datamaskin trenger vi en underliggende *datastruktur*. Denne brukes til å representere de *topologiske* enhetene i en triangulering: noder, kanter og trekanter. En datastruktur inneholder videre den topologiske naboskapsinformasjonen mellom de topologiske enhetene, samt den *geometriske* informasjonen assosiert til disse. Det finnes mange forskjellige datastrukturer, og forskjellige typer applikasjoner har ulike behov når det gjelder egenskapene til en data-struktur. For eksempel så trenger interaktive visualiseringsapplikasjoner en

datastruktur som gir rask tilgang til geometriske data og effektiv traversering av topologien for å hente ut alle trekantene. Raske datastrukturer betyr nødvendigvis flere pekere og dermed mer bruk av minne. Valg av datastruktur blir derfor alltid et kompromiss mellom effektivitet på den ene siden og minnebruk på den andre. Enkelte applikasjoner kan også trenge mer enn bare en datastruktur. En detaljert beskrivelse og analyse av en del forskjellige datastrukturer finnes i [Hje03].

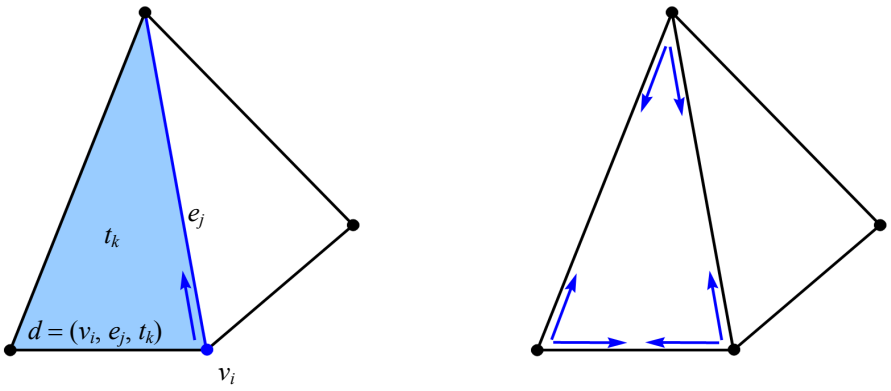
Applikasjoner som inneholder trianguleringer trenger også en del topologiske og geometriske operasjoner, som kan anvendes på de topologiske og geometriske enhetene representert ved datastrukturen til trianguleringen. Slike operasjoner kan hovedsakelig deles inn i to grupper: operasjoner som *ekstraherer* informasjon fra trianguleringen, også kalt *spørringer*, og operasjoner som *endrer* topologien eller geometrien til trianguleringen. Et eksempel på en spørring kan være å finne alle nabonodene til en node, mens swapping av en kant er et eksempel på en operasjon som endrer trianguleringen.

Det har vært mest vanlig å implementere slike operasjoner eller funksjoner slik at de tilpasses den datastrukturen som benyttes. Dette medfører at alle slike funksjoner må skrives om dersom datastrukturen endres. Bruker man flere datastrukturer, må man implementere flere versjoner av funksjonene. Muligheten til å gjenbruke kode i nye applikasjoner blir også sterkt redusert når det gjøres på denne måten.

I neste seksjon skal vi se på et verktøy som gjør det mulig å konseptuelt skille mellom datastrukturer og operatorer, såkalte generaliserte maps eller *G-maps*. G-maps lar oss gi en *algebraisk* beskrivelse av randbaserte strukturer som for eksempel trianguleringer, som er uavhengig av underliggende datastrukturer. Dette blir dermed et kraftig verktøy som kan brukes til å implementere et generelt grensesnitt mot datastrukturer. Det er nettopp denne tilnærmelsen som anvendes i det generiske trianguleringsbiblioteket TTL.

3.2 Generaliserte maps (G-maps)

Generaliserte maps eller G-maps er et generelt verktøy for å modellere topologien til randbaserte geometriske strukturer, som blant annet har vært anvendt innenfor geologisk modellering. G-maps defineres algebraisk ut i fra noen få enkle og klare konsepter. Topologien beskrives gjennom et enkelt topologisk element som kalles en *dart* og en mengde med funksjoner



Figur 3.1: En dart $d = (v_i, e_j, t_k)$ (a), og de seks forskjellige dartposisjonene i en trekant (b).

eller operatorer som anvendes på dartene i den topologiske strukturen. Her vil vi se på hvordan G-maps kan anvendes for å lage en algebraisk beskrivelse av topologien i trianguleringer, som de topologiske operatorene kan operere på. En grundig teoretisk gjennomgang av G-maps finnes i [Lie89] og [Ber94].

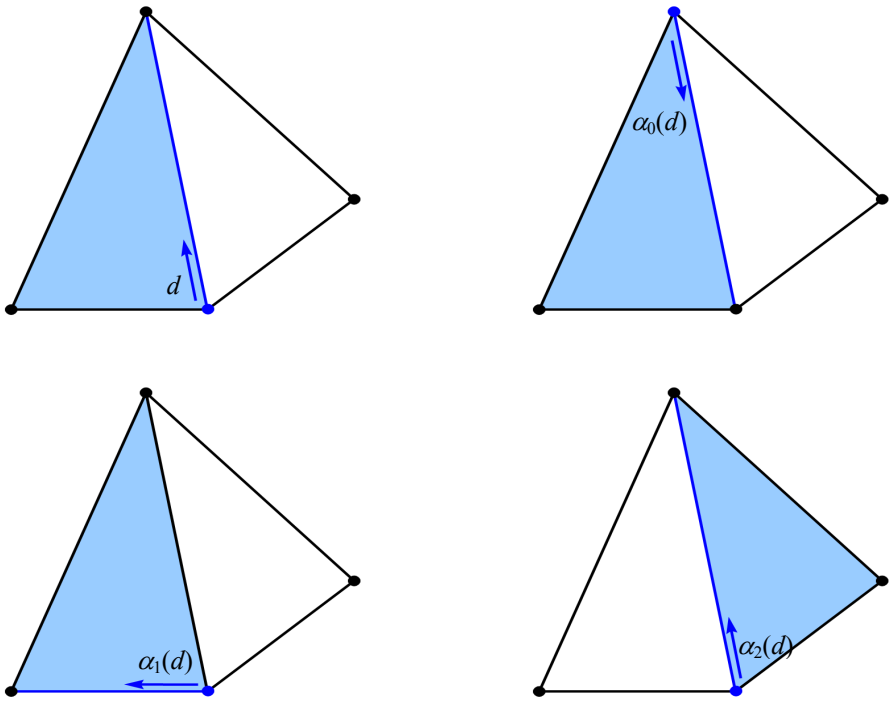
En dart i en triangulering kan ses på som et unikt *trippel* $d = (v_i, e_j, t_k)$, hvor v_i er en node i sidekanten e_j, og e_j er en sidekant i trekanten t_k. Dette er vist på *Figur 3.1* (*a*), hvor darten er markert med en pil. For hver trekant finnes det altså seks forskjellige tripler som kan definere en dart, og disse er tegnet inn på *Figur 3.1* (*b*). Vi kan videre uttrykke hele topologien i en triangulering gjennom en entydig mengde med tripler eller darter D.

Vi kan definere tre entydige funksjoner α_0, α_1 og α_2 som opererer på mengden av tripler D i en triangulering i planet gjennom en en-til-en avbildning:

$$\alpha_i : D \rightarrow D, \text{ der } i = 0, 1, 2.$$

Funksjonene er også *bijektive*, det vil si at $\alpha_i(\alpha_i(d)) = d$. Vi har kalt disse tre operatorene for α-iteratorer, og anvendt på trianguleringer definerer vi de som følger:

- $\alpha_0(d)$ avbilder d til en dart med en annen node, men samme sidekant og trekant.



Figur 3.2: Resultatet av å anvende hver av de tre α-iteratorene på en dart d.

- $\alpha_1(d)$ avbilder d til en dart med en annen sidekant, men samme node og trekant.
- $\alpha_2(d)$ avbilder d til en dart med en annen trekant, men samme node og sidekant.

Figur 3.2 viser resultatet etter at hver av de tre α-iteratorene har blitt anvendt på en dart d. Dersom en sidekant e_j i en dart $d = (v_i, e_j, t_k)$ ligger på randen av trianguleringen, definerer vi $\alpha_2(d)$ til å være uendret. En triangulering kan nå representeres som en *graf*

$$G(D, \alpha_0, \alpha_1, \alpha_2),$$

som kalles en G-map.

En *komposisjon* av α-iteratorer $\alpha_i(\alpha_j(\cdots\alpha_r(d)\cdots))$ skriver vi på formen $\alpha_i \circ \alpha_j \circ\cdots\circ \alpha_r(d)$. Vi har følgende interessante komposisjoner som er nyttige når vi skal lage iteratorer for å operere på trianguleringer:

- $\alpha_0 \circ \alpha_1$ anvendt gjentatte ganger itererer gjennom alle nodene og sidekantene i en trekant.
- $\alpha_1 \circ \alpha_2$ anvendt gjentatte ganger itererer gjennom alle sidekantene og trekantene som deler en felles node.
- $\alpha_0 \circ \alpha_2$ anvendt gjentatte ganger itererer rundt en sidekant gjennom å besøke begge nodene og begge trekantene som ligger inntil sidekanten.
- Dersom rekkefølgen i en komposisjon $\alpha_i \circ \alpha_j$ byttes om til $\alpha_j \circ \alpha_i$, endres også retningen til iterasjonen.

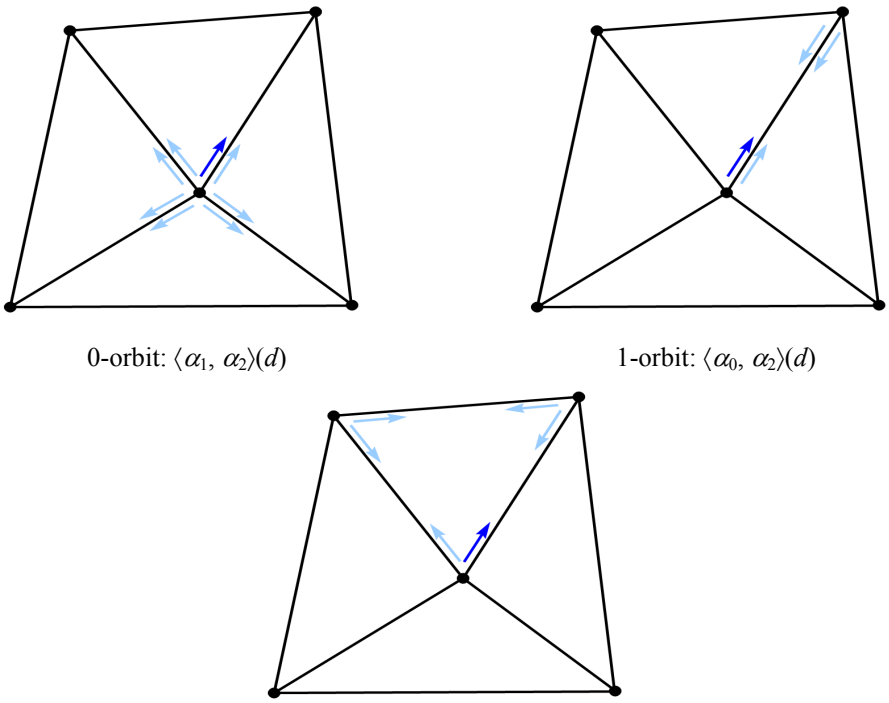
Mer formelt kan vi definere en orbit og en k-orbit:

> **Definisjon 3.1:** (Orbit) *La $\{\alpha_i\}$ være en, to eller alle tre α-iteratorene til en G-map $G(D, \alpha_0, \alpha_1, \alpha_2)$, og la $d \in D$. En orbit $\langle\{\alpha_i\}\rangle(d)$ av d er mengden av alle darter i D som kan nås med komposisjoner av $\{\alpha_i\}$ i vilkårlig rekkefølge ved å starte fra d.*

> **Definisjon 3.2:** (k-orbit, $k = 0, 1, 2$) *k-orbiten til en dart d i en G-map $G(D, \alpha_0, \alpha_1, \alpha_2)$ defineres som orbiten $\langle\alpha_i, \alpha_j\rangle(d)$* der $i, j \neq k$ og $i \neq j$.

En 0-orbit $\langle\alpha_1, \alpha_2\rangle(d)$ er altså mengden av alle darter rundt en node, en 1-orbit $\langle\alpha_0, \alpha_2\rangle(d)$ er mengden av alle darter rundt en sidekant, og en 2-orbit $\langle\alpha_0, \alpha_1\rangle(d)$ er mengden av alle darter innenfor en trekant. Dette er vist på *Figur 3.3*. Vi observerer videre at dersom vi har en regulær triangulering, så er orbiten $\langle\alpha_0, \alpha_1, \alpha_2\rangle(d)$, der d er en vilkårlig dart i D, mengden av alle darter i D. Dette vil si at alle de topologiske elementene: nodene, kantene og trekantene i en triangulering kan nås gjennom å anvende komposisjoner av $\{\alpha_i\}$, der $i = 0, 1, 2$.

Gjennom α-iteratorene har vi nå et effektivt verktøy til å traversere trianguleringer med. Disse kan brukes som et grensesnitt mot en hvilken som helst datastruktur, og generiske algoritmer for å navigere rundt i topologien til en triangulering kan baseres på disse iteratorene. Algoritme 3.1 i [Hje00] viser hvordan α-iteratorene kan brukes til å lokalisere den trekanten i en triangulering som innholder et gitt punkt. I den neste seksjonen skal vi se mer i detalj på selve trianguleringsbiblioteket TTL.



Figur 3.3: k-orbitene, $k = 0, 1, 2$ til en dart d.

3.3 Triangulation Template Library (TTL)

Triangulation Template Library eller TTL er et generisk triangulerings-bibliotek hovedsakelig for generering av Delaunay-trianguleringer. TTL ble utviklet i forbindelse med Dynamap-prosjektet [DYN] på SINTEF Anvendt Matematikk (SAM). Vedlikeholdet av TTL er nå underlagt Oslo Graphics Lab, og biblioteket er i ferd med å bli formelt lansert som Open Source Software (OSS) under Trolltechs Qt Public License (QPL), som er godkjent av Open Source Initative (OSI). TTL brukes i dag i diverse sammenhenger både på SINTEF og på Simula Research Laboratory, og i forbindelse med hovedoppgaver og prosjektoppgaver ved Universitetet i Oslo.

TTL er generisk ved at det ikke avhenger av en spesiell underliggende

datastruktur. Dermed kan brukeren anvende sin egen datastruktur og dra nytte av en rekke generiske algoritmer i TTL som kan arbeide direkte mot en hvilken som helst datastruktur for trianguleringer. Det er også mulig å bruke en av datastrukturene som distribueres med TTL. Foreløpig er det bare halvkant-datastrukturen som følger med i TTL-distribusjonen, men det er planlagt at flere datastrukturer skal følge med i fremtidige versjoner av TTL. Halvkant-datastrukturen beskrives nærmere i seksjon 3.5.

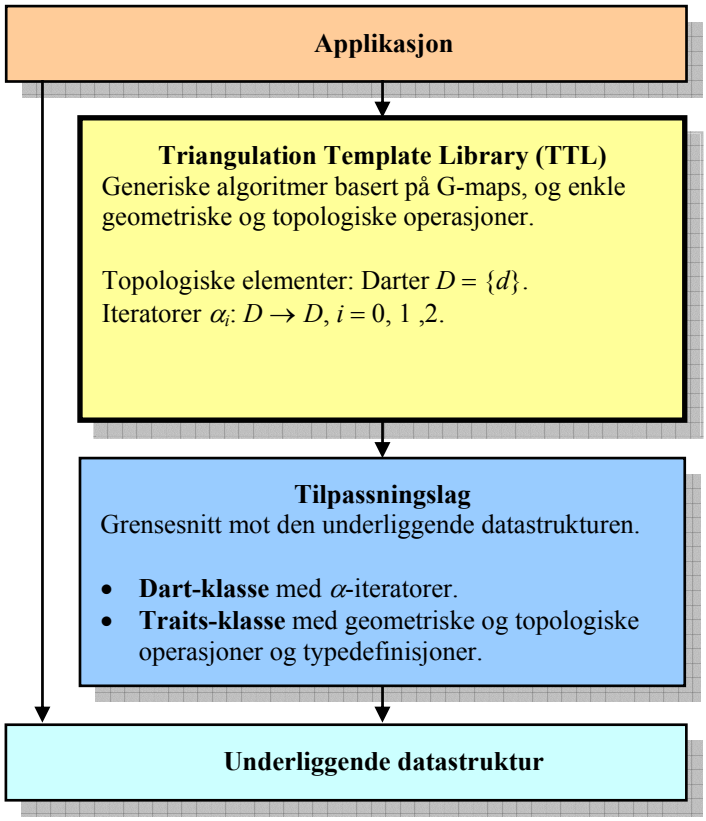
TTL er programmert i programmeringsspråket C++ og gjør hyppig bruk av *funksjonstemplates* [Str97] for å oppnå et generisk programmerings-grensesnitt eller API (application programmer's interface). Det bruker G-maps med dartalgebra og α-iteratorer til å modellere trianguleringene på et abstrakt nivå, som forklart i seksjon 3.2. Programmeringsfilosofien bak TTL minner mye om programmeringsfilosofien bak The Standard Template Library eller STL [STL], med generiske iteratorer og pekeraritmetikk.

Målet med TTL har vært å implementere en kompakt samling av generiske algoritmer for triangulering, som er uavhengige og klart separerte fra den underliggende datastrukturen. TTL kommuniserer med data-strukturen til applikasjonen gjennom et grensesnitt eller tilpassningslag, som må programmeres av applikasjonsprogrammereren. Tilpassningslaget må implementere følgende to komponenter, tilpasset den underliggende data-strukturen som benyttes av applikasjonen:

- en *Dart-klasse* med α-iteratorer, og
- en *Traits-klasse* med grunnleggende geometriske og topologiske operasjoner samt typedefinisjoner.

Hvordan TTL fungerer, er vist skjematisk på *Figur 3.4*. Som vi ser er selve TTL-biblioteket fullstendig uavhengig av den underliggende datastrukturen. Dette står i kontrast til hvordan tradisjonell trianguleringsprogramvare virker, med algoritmer som arbeider direkte på den aktuelle datastrukturen som brukes, som vist skjematisk på *Figur 3.5*.

TTL inneholder forskjellige topologiske og geometriske spørringer for å ekstrahere informasjon om trianguleringen. Videre inneholder TTL topologiske og geometriske modifiserere for å lage Delaunay-trianguleringer med og uten føringer, som vi så på i Kapittel 2.



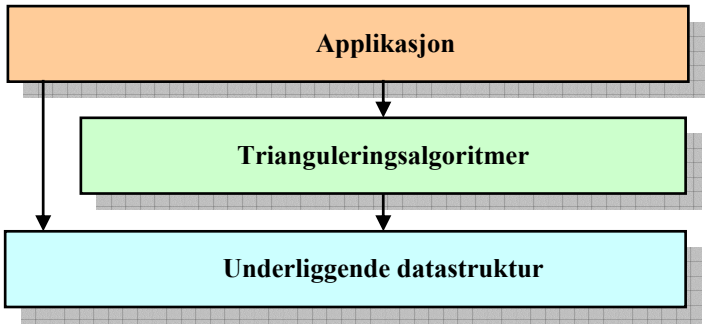
Figur 3.4: Skjematisk skisse av en applikasjon som bruker TTL.

De topologiske spørringene henter ut informasjon om den topologiske strukturen til trianguleringen, og disse er implementer generiske ved hjelp av darter og α-iteratorer i G-maps. Eksempler på topologiske spørringer som finnes i TTL er:

- `bool isBoundaryNode(const DartType& dart);`
- `bool isBoundaryEdge(const DartType& dart);`
- `bool isBoundaryFace(const DartType& dart);`
- `int getDegreeOfNode(const DartType& dart);`
- `void getBoundary(const DartType& dart,`
  `list<DartType>& boundary);`

Alle funksjonene bruker en dart-type som eneste parametertype. Dette gir et mer uniformt grensesnitt.

Den andre typen spørringer som finnes i TTL, geometriske spørringer,



Figur 3.5: Skjematisk skisse av en tradisjonell trianguleringsapplikasjon.

brukes til å hente ut informasjon vedrørende den geometriske posisjonen til nodene i en triangulering. Eksempler på geometriske spørringer som finnes i TTL er:

- `bool locateTriangle(const PointType& point, DartType& dart);`
- `bool inTriangle(const PointType& point, const DartType& dart);`

De grunnleggende geometriske operasjonene som trengs i template-funksjonene til disse spørringene implementeres i Traits-klassen i tilpassningslaget (se *Figur 3.4*). Ved å overlate ansvaret for de geometriske beregningene til tilpassningslaget og applikasjonsprogrammereren, er det også applikasjonsprogrammereren som avgjør hvor høyt presisjonsnivå disse skal ha.

I tillegg til de topologisk og geometriske spørringene, som lar trianguleringene være uendret, har TTL en mengde operasjoner som modifiserer trianguleringene. Disse er mer kompliserte enn spørringene. Vi kan grovt sett dele slike topologiske operasjoner inn i tre kategorier:

(1) Operasjoner som legger til noder, kanter og trekanter i en triangulering.
(2) Operasjoner som fjerner noder, kanter og trekanter i en triangulering.
(3) Operasjoner som bevarer antall noder, kanter og trekanter, for eksempel swapping av en kant.

Følgende funksjoner er eksempler på slike operasjoner som finnes i TTL:

- `bool insertNode(DartType& dart, PointType& point);`
- `void removeNode(DartType& dart);`

Template-funksjonene for disse operasjonene trenger også en del grunnleggende funksjonalitet som må implementeres i Traits-klassen.

3.4 Inkrementell Delaunay-triangulering

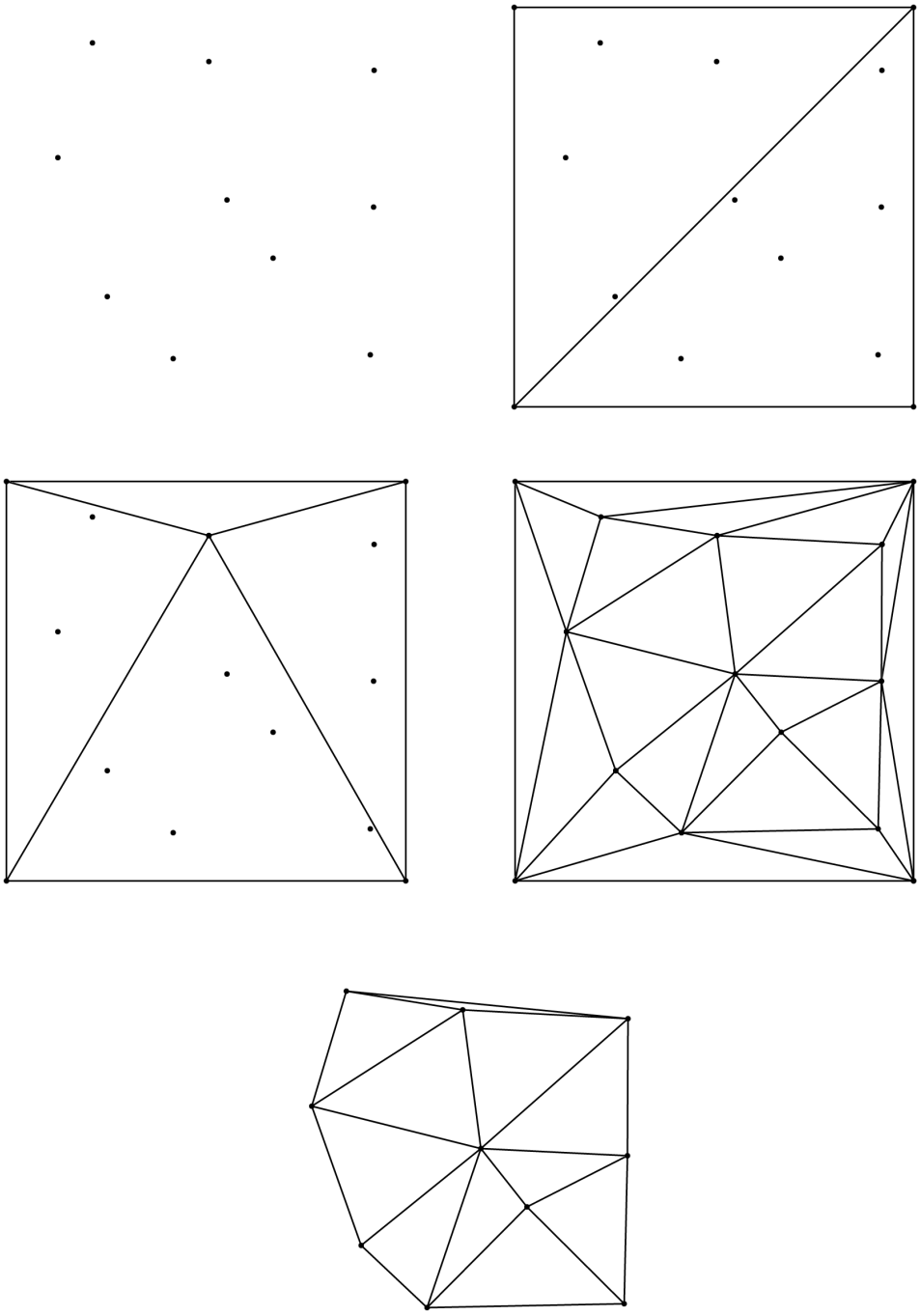
TTL benytter en såkalt *inkrementell* algoritme til å lage Delaunay-trianguleringer. Denne algoritmen starter med en kunstig initiell triangulering. Dette kan for eksempel kan være et rektangel delt inn i to store trekanter, som omslutter alle punktene som skal trianguleres. Deretter settes alle punktene inn i trianguleringen, ett etter ett. Etter hver innsetting oppdateres trianguleringen til å være en Delaunay-triangulering. Til slutt, når alle punktene er satt inn i trianguleringen, fjernes den kunstige randen. Algoritmen er illustrert på *Figur 3.6*.

Selve operasjonen å sette inn et punkt p i en eksisterende Delaunay-triangulering Δ_N som består av N noder, og få en ny Delaunay-triangulering Δ_{N+1} med $N+1$ noder, kan deles inn i tre steg:

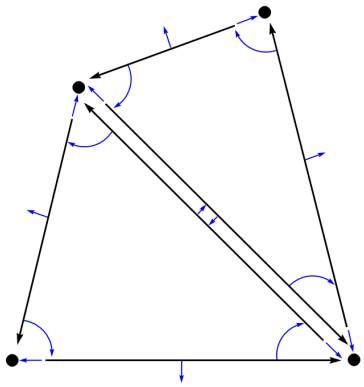
(1) Lokaliser trekanten T_i i Δ_N som inneholder p.
(2) Splitt T_i i tre trekanter ved å opprette tre nye kanter mellom p og nodene til T_i, og få en ny triangulering Δ'_{N+1}.
(3) Anvend en *swappeprosedyre* basert på sirkelkriteriet for å swappe kanter i Δ'_{N+1} som ikke er lokalt optimale, inntil alle kantene er lokalt optimale, og den endelige trianguleringen Δ_{N+1} er en Delaunay-triangulering.

Detaljene i denne algoritmen er beskrevet nærmere i [Hje00].

Det kan vises at tidsforbruket til den inkrementelle algoritmen for Delaunay-trianguleringer i verste tilfelle er $O(N^2)$, hvor N er antall punkter som skal settes inn. Den teoretisk optimale kjøretiden for algoritmer for Delaunay-trianguleringer er $O(N \log N)$, og det finnes *splitt-og-hersk* algoritmer som har slike kjøretider. Imidlertid er det bare i helt spesielle tilfeller, hvor store deler av trianguleringen må retrianguleres for hvert punkt som settes inn, at den inkrementelle algoritmen har et tidsforbruk som er $O(N^2)$. Eksperimenter har vist at i praktiske applikasjoner vil kjøre-



Figur 3.6: Inkrementell Delaunay-triangulering.



Figur 3.7: Pekerstrukturen til halvkant-datastrukturen.

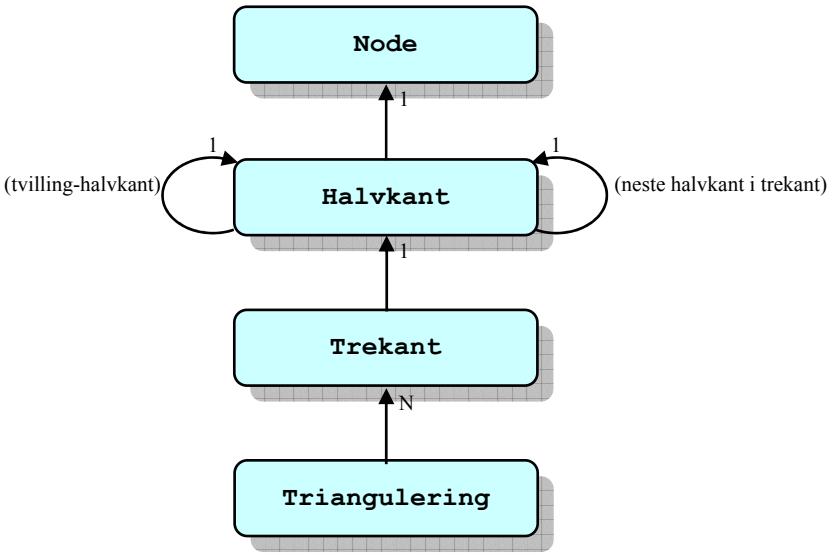
tidene gjennomsnittlig være $O(N \log N)$ eller bedre [Hje03].

For at algoritmen skal kjøre fortest mulig, kan punktene sorteres i leksikografisk økende orden, slik at $p_i = (x_i, y_i) < p_j = (x_j, y_j)$ hvis og bare hvis $x_i < x_j$, eller $x_i = x_j$ og $y_i < y_j$. Det neste punktet som skal settes inn vil da være i nærheten av det forrige, og dette kan TTL utnytte til å raskere finne trekanten som inneholder dette punktet.

3.5 Halvkant-datastrukturen

I skrivende stund er halvkant-datastrukturen for trianguleringer den eneste datastrukturen som følger med i TTL-distribusjonen. Prinsippet bak halvkant-datastrukturen ble introdusert av Weiler [Wei85] og går ut på å splitte hver sidekant i to rettede halvkanter, som er orientert motsatt vei i forhold til hverandre. Vi kan derfor se på dette som om en halvkant tilhører nøyaktig en trekant. De tre halvkantene til en trekant orienteres mot klokka, når vi ser på trekanten ovenfra.

Figur 3.7 viser pekerstrukturen for halvkant-datastrukturen som brukes i TTL. Hver halvkant har en peker til noden den starter i (kildenoden), en peker til den neste halvkanten i samme trekant når vi følger retningen mot klokka, og en peker til den motsatt rettede tvillinghalvkanten sin i trekanten ved siden av.



Figur 3.8: Klassediagram for halvkant-datastrukturen.

Når en datastruktur implementeres i et *objektorientert* programmeringsspråk som C++ eller Java, er det naturlig å tenke på de topologiske enhetene: noder, sidekanter og trekanter som *klasser*. Halvkant-datastrukturen vil da representeres gjennom en **Node**-klasse, en **Halvkant**-klasse og en **Trekant**-klasse, som vist på klassediagrammet på *Figur 3.8*. **Trekant**-klassen innholder en peker til en av halvkantene i trekanten. Denne kalles den *ledende halvkanten* til trekanten. Hver trekant har nøyaktig en ledende halvkant. **Halvkant**-klassen inneholder en peker til kildenoden, en peker til den neste halvkanten i samme trekant og en peker til tvillinghalvkanten i trekanten siden av. **Node**-klassen inneholder den geometriske posisjonen til noden. I tillegg har vi også en **Triangulering**-klasse, som inneholder en liste med pekere til alle trekantene i trianguleringen. Implementasjonen av halvkant-datastrukturen som følger med TTL har ingen egen **Trekant**-klasse, men lar heller hver trekant bli representert gjennom en ledende halvkant.

Vi merker oss at en gitt halvkant i en trekant kan assosieres med en dart d. Den neste halvkanten i trekanten kan derfor enkelt finnes gjennom komposisjonen $\alpha_1 \circ \alpha_0(d)$. Tilsvarende kan tvillinghalvkanten enkelt finnes gjennom komposisjonen $\alpha_0 \circ \alpha_2(d)$.

3.6 Generiske trianguleringsbiblioteker

Vi har sett at ved hjelp av G-maps, kan vi lage generiske algoritmer for trianguleringer som er konseptuelt uavhengige av den underliggende datastrukturen. Effektiviteten til slike generiske algoritmer er heller ikke betylig lavere enn algoritmer som virker direkte på den underliggende datastrukturen [Hje00]. Spesielt ikke når de implementeres som template-funksjoner i C++, slik som det er gjort i TTL. For å øke effektiviteten kan de topologiske traverseringsoperatorene α_0, α_1 og α_2 i G-maps, som dominerer kjøretiden, implementeres i C++ som *inline-funksjoner* [Str97] i en dart-klasse.

Det finnes også flere andre trianguleringsbiblioteker som benytter generiske algoritmer. Et eksempel er GrAL (Grid Algorithms Library) [Ber02], [GRA]. GrAL kan lage forskjellige typer av grid og er laget spesielt med tanke på numerisk løsning av partielle differensiallikninger. Et annet eksempel er CGAL (Computational Geometry Algorithms Library) [CGA], som er et relativt stort bibliotek med generiske algoritmer innenfor geometri.

Kapittel 4

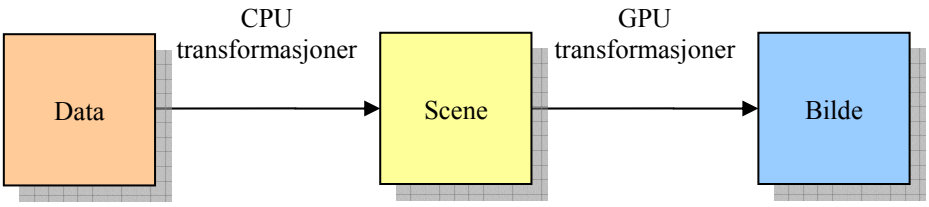
Datagrafikk og OpenGL

Vi har i de to foregående kapitlene sett på trianguleringsteori og hvordan vi kan implementere trianguleringer i en datamaskin. I mange tilfeller ønsker vi også å vise resultatet av denne implementasjonen i form av datagrafikk. Dette vil vi ta for oss i dette kapitlet. Først vil vi se generelt på hva visualisering er. Deretter vil vi i seksjon 4.2 beskrive litt grunnleggende teori rundt hvordan datagrafikk genereres. Seksjon 4.3 handler om OpenGL, og vi vil si litt om det å visualisere trianguleringer i seksjon 4.4. I den siste seksjonen i dette kapitlet vil vi se nærmere på hva antialiasing er. En grundig innføring i diverse emner innenfor datagrafikk finnes i [Fol90], eller [Ang00] som er litt mer rettet mot OpenGL-brukere.

4.1 Visualisering

Å visualisere vil si å synliggjøre. Innenfor datateknologi innebærer dette å transformere numeriske data, eller annen informasjon, til bilder generert ved hjelp av datagrafikk. Vi mennesker egner oss dårlig til å trekke ut nyttig informasjon fra data i form av tall, men så fort vi får omgjort dette til bilder, som for eksempel kurver i to dimensjoner eller flater og volumer i tre dimensjoner, kan vi bruke synssansen i samarbeid med hjernen til å tolke og forstå dataene. Det viser seg også at høy grad av interaktivitet og rask respons er viktig i visualiseringsapplikasjoner for at vi best mulig skal oppfatte relasjoner mellom objektene som visualiseres. Dette innebærer at det bør være mulig å panorere, rotere og zoome ut og inn i sanntid.

Mengden av numeriske data vi får fra beregninger og simuleringer blir stadig større, ettersom datamaskinene hele tiden blir kraftigere og kraftigere.



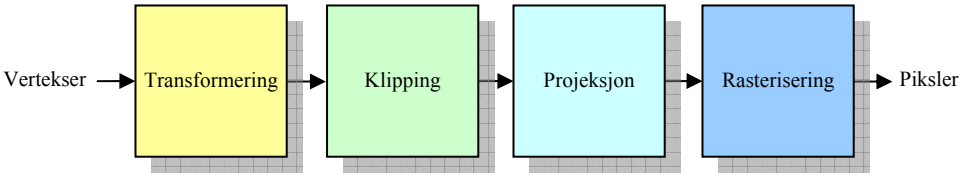
Figur 4.1: Visualiseringsprosessen.

Det samme gjelder datamengden vi får fra ulike typer skannere, siden oppløsningen på disse også stadig øker. Når mengden med numeriske data hele tiden blir større og større, stilles det også høyere og høyere krav til programvare og maskinvare innenfor visualisering. Det skal også nevnes at underholdningsindustrien er en stor pådriver til utvikling av kraftigere maskinvare til å håndtere datagrafikk, siden det hele tiden utvikles mer avanserte dataspill med datagrafikk som blir mer og mer virkelighetstro. Dette har ført til at de fleste nyere datamaskiner til hjemmebruk, også har ganske kraftige grafikkort.

Visualiseringsprosessen kan oppsummeres med diagrammet på *Figur 4.1*. Den begynner med å lese inn data på diskret form. Dataene transformeres til geometriske objekter i en *virtuell scene*. Dette er en generell prosess som kan avhenge av dataene og interaksjon fra brukeren, og den må derfor gjøres av prosessoren. Den virtuelle scenen transformeres videre til et bilde gjennom en *rasteriseringsprosess*. Denne prosessen er veldig spesialisert, og de samme beregningene utføres mange ganger. Den kan derfor gjøres veldig effektivt av en spesialisert grafikkprosessor eller GPU (Graphics Processing Unit). Vi vil se nærmere på denne siste transformasjonen i neste seksjon.

4.2 Datagrafikk

Generering av datagrafikk består av et relativ lite antall enkle operasjoner som utføres gjentatte ganger. Disse operasjonene kan implementeres effektivt i hardware og parallelliseres for å få best mulig ytelse. En GPU er organisert som en *pipeline*, eller et samlebånd, med forskjellige stasjoner. Denne pipelinen inneholder de transformasjonene som er nødvendige for å transformere en 3-dimensjonal scene til et 2-dimensjonalt bilde på dataskjermen. Den kalles den geometriske pipelinen og inneholder fire trinn.



Figur 4.2: Den geometriske pipelinen.

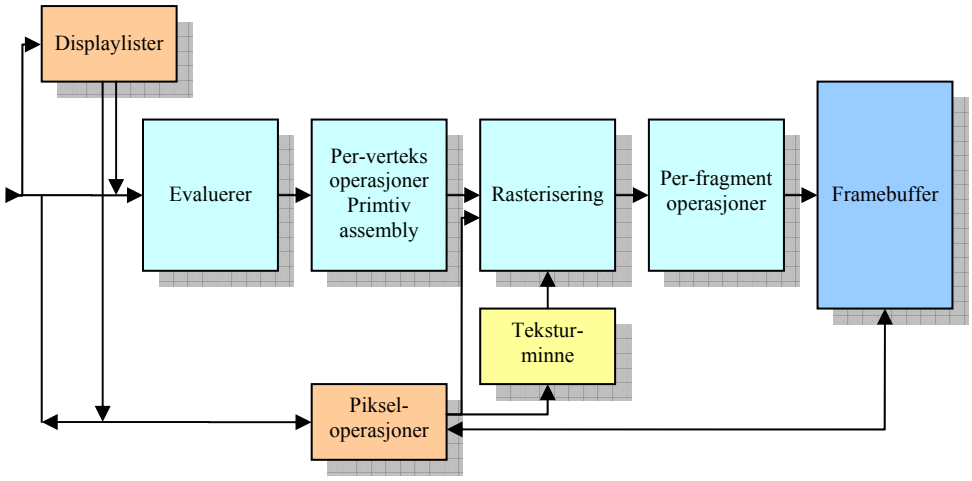
Disse er vist skjematisk på *Figur 4.2*. Det første trinnet er transformasjoner mellom forskjellige koordinatsystemer. Neste trinn er *klipping*, det vil si å fjerne det som i øyeblikket ikke er synlig på skjermen. Det tredje trinnet projiserer den 3-dimensjonale scenen ned i et plan. Til slutt, i det siste trinnet som er rasterisering, omgjøres den projiserte scenen til piksler i *framebufferet*, som så kan vises på skjermen.

Den store fordelen med pipelineorganiseringen er at prosesseringen av en ny *verteks* eller punkt kan starte så fort den første verteksen har passert det første trinnet. En pipeline som er delt i fire vil hele tiden inneholde fire vertekser som er under prosessering. Gjennomstrømningen vil dermed være fire ganger større, enn hvis det bare hadde vært et stort trinn som prosesserte en verteks av gangen.

4.3 OpenGL

OpenGL er en API for å utvikle interaktive 2D og 3D grafikkapplikasjoner. Det ble opprinnelig introdusert av SGI (Silicon Graphics, Inc.) i 1992, men har senere blitt en åpen standard. OpenGL er plattformuavhengig, og bruken av det er svært utbredt.

OpenGL grensesnittet er direkte knyttet opp mot selve grafikk-maskinvaren og opererer derfor på et relativt lavt nivå. Det inneholder noen hundre prosedyrer og funksjoner, som gir applikasjonsprogrammereren detaljert kontroll over hvordan grafikken bygges opp. OpenGL tegner *grafiske primitiv* inn i framebufferet. Et grafisk primitiv kan være et punkt, et linjesegment, et polygon eller et pikselrektangel. OpenGL kan settes i en rekke forskjellige moduser eller tilstander, som bestemmer hvordan de grafiske primitivene skal tegnes.



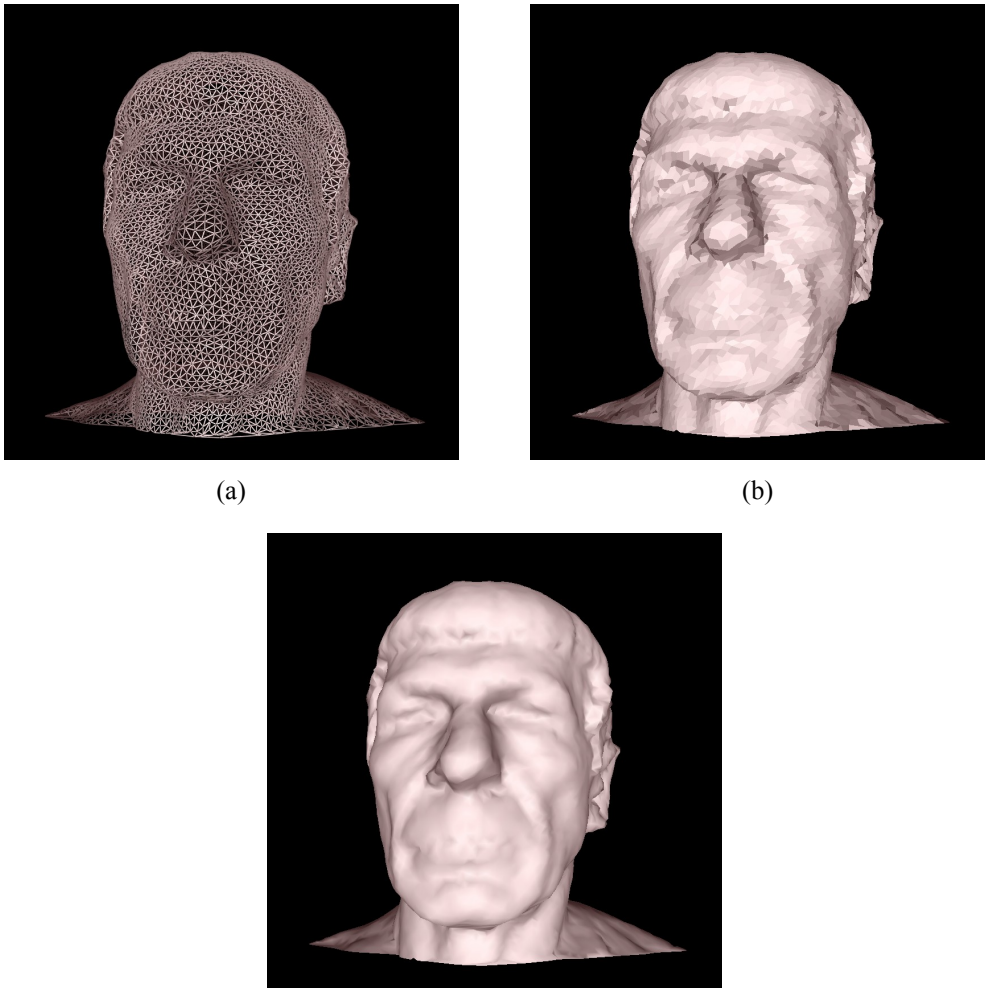
Figur 4.3: Pipelinen i OpenGL.

På *Figur 4.3* ser vi et skjematisk diagram over OpenGL. OpenGL-kommandoene kommer inn fra venstre. Dette kan være kommandoer som spesifiserer et geometrisk objekt eller som forteller hvordan et objekt skal behandles i de forskjellige stegene i pipelinen. De fleste slike kommandoer kan også samles opp i såkalte *displaylister*, som så prosesseres ved et senere tidspunkt. Ellers vil kommandoene bli sendt gjennom pipelinen fortløpende.

Alle funksjonskallene i OpenGL er grundig beskrevet i referanse-manualen [Ope04]. Ellers finnes det mange bøker som tar for seg programmering i OpenGL, for eksempel [Ope03]. OpenGL har også sitt eget nettsted (http://www.opengl.org).

4.4 Visualisering av trianguleringer

I denne hovedoppgaven skal vi først og fremst visualisere trianguleringer i planet. Trianguleringsbiblioteket TTL, som vi så på i forrige kapittel, tar seg av oppbygningen av selve trianguleringen. Det vi trenger å gjøre, er å hente ut informasjon fra TTL om hvordan trianguleringen ser ut, og visualisere den ved hjelp av datagrafikk. Den enkleste måten å gjøre dette på er å hente ut en liste med alle kantene i trianguleringen og tegne opp disse som linjestykker, en etter en. Vi vil da få ut et bilde av trianguleringen. Det er en slik metode vi benytter i TriangTutor.



(a) (b)

(c)

Figur 4.4: Triangulering i 3D vist som wireframe (a), og fylte trekanter med flat (b) og Gouraud skyggelegging (c).

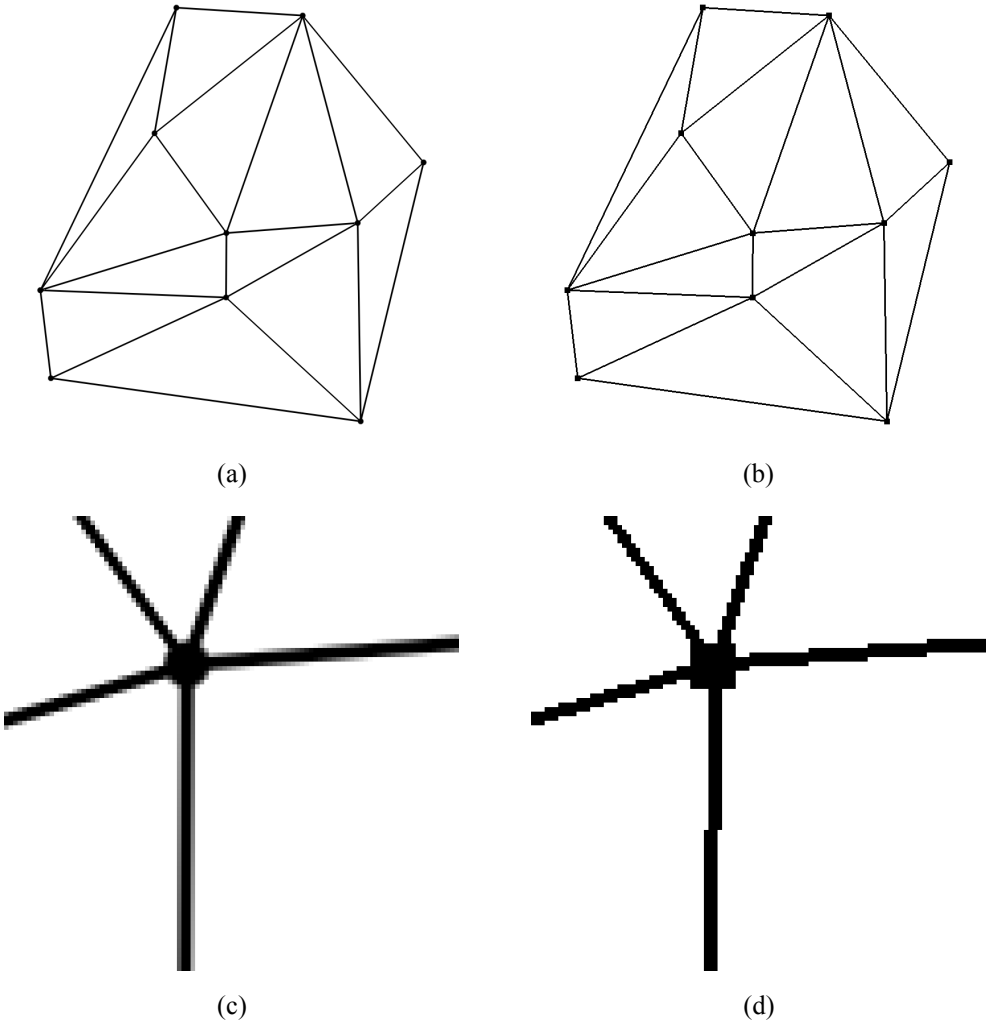
Denne metoden vil i utgangspunktet også fungere dersom vi ønsker å visualisere trianguleringer i 3D. Ved visualisering av trianguleringer i rommet, er det derimot vanlig å la hver trekant utgjøre en fylt flate. Dette er det ikke mulig å få til ved metoden beskrevet over. En metode som egner seg bedre til å visualisere tredimensjonale trianguleringer vil være å i stedet hente ut en liste med alle trekantene og tegne opp disse, en etter en. En trekant er et grafisk primitiv i OpenGL. For store trianguleringer kan opptegningen effektiviseres ved å finne lange *trekantstriper* i

trianguleringen, og tegne opp disse. *Figur 4.4* viser et eksempel på en tredimensjonal triangulering. På *Figur 4.4* (*a*) ser vi rammeverket (wireframe) til trianguleringen, mens på *Figur 4.4* (*b*) og (*c*) ser vi fylte trekanter med henholdsvis *flat* og *Gouraud* skyggelegging. TriangTutor har ikke støtte for å lage trianguleringer i 3D.

4.5 Antialiasing

En begrensning med *rasterdisplayet*, som er et grid med piksler, er at punkter, linjer og andre grafiske primitiv må approksimeres ved hjelp av disse pikslene. Dette fører til at en linje som går på skrå ikke vil være glatt, men ha en slags trappeform. Denne effekten, som kalles for *aliasing*, oppstår fordi vi prøver å gå fra den kontinuerlige representasjonen av et geometrisk objekt, som i prinsippet har uendelig høy oppløsning, til en diskret representasjon med begrenset oppløsning. Grafikkort i nyere datamaskiner har innebygget støtte for å redusere problemene med aliasing. Dette kalles for *antialiasing*. Antialiasing går ut på å blende inn bakgrunnsfargen i kantene til det geometriske objektet, slik at det ikke blir noen skarpe fargeoverganger.

På *Figur 4.5* (*a*) og (*b*) ser vi en hel triangulering tegnet opp både med og uten bruk av antialiasing. Vi ser at antialiasing gjør at linjene og punktene blir mye glattere. Tar vi ut en liten del av bildet av trianguleringen og forstørrer denne, som vi har gjort på *Figur 4.5* (*c*) og (*d*), ser vi tydelig hvordan antialiasing fungerer. Detaljene rundt teknikkene som brukes i antialiasing står det mer om i [Fol90]. Å bruke antialiasing er beregningsmessig ganske tungt, og for å oppnå antialiasing over hele skjermen i sanntid, må dette implementeres i hardware.



Figur 4.5: En triangulering tegnet opp med (a)(c) og uten (b)(d) bruk av antialiasing.

Kapittel 5

Grunnleggende funksjonalitet

Etter å ha gått gjennom grunnleggende trianguleringsteori i Kapittel 2, introdusert trianguleringsbiblioteket TTL og konseptene rundt dette i Kapittel 3 og sett litt på visualisering og datagrafikk i Kapittel 4, er vi nå klare til å se på selve trianguleringsapplikasjonen, TriangTutor. I dette kapitlet og neste kapittel beskrives funksjonaliteten i TriangTutor. Dette kapitlet tar for seg den mest grunnleggende funksjonaliteten. Det vil si funksjonalitet for å lage Delaunay-trianguleringer med og uten føringer. I neste kapittel vil vi se på mer avansert funksjonalitet, som influensregion, Voronoi-diagram og omskrivende sirkler. Vi starter med å gi en kort oversikt over funksjonaliteten i programmet. Deretter tar vi en kikk på menysystemet og hvordan dette er organisert, i seksjon 5.2. Videre beskrives innsetting og fjerning av noder i henholdsvis seksjon 5.3 og seksjon 5.4, og interaktiv flytting av noder i seksjon 5.5. I seksjon 5.6 ser vi på innsetting av føringer, og seksjon 5.7 tar for seg filformatet som brukes. Vi runder av dette kapitlet med litt om zooming og panorering. I Kapittel 7 vil vi se nærmere på hvordan koden til applikasjonen er strukturert, og da spesielt hvordan den er delt inn i objektklasser.

5.1 Oversikt over funksjonalitet

I og med at TriangTutor baserer seg på funksjonaliteten i trianguleringsbiblioteket TTL, som benytter en inkrementell algoritme til å bygge opp Delaunay-trianguleringer, er det også denne typen trianguleringer man først og fremst kan lage med TriangTutor. Når applikasjonen startes finnes det tre forskjellige måter å opprette en triangulering på. Den første måten er å lage

en ny Delaunay-triangulering helt fra bunnen gjennom å sette inn en og en node. En annen måte er å laste inn en fil som inneholder datapunkter, og få laget en Delaunay-triangulering av disse. Den tredje og siste måten er å la TriangTutor selv generere en Delaunay-triangulering av et antall vilkårlige punkter i planet. Det er selvfølgelig også mulig å forandre på en triangulering som er lastet inn eller generert av programmet, enten ved å sette inn nye punkter, å fjerne punkter eller å flytte på punkter. Videre er det mulig å legge inn føringer eller predefinerte kanter i trianguleringene. Dette gjøres ved enten å markere allerede eksisterende kanter som føringer, eller å tegne inn føringene direkte mellom to punkter. En føring kan også fjernes igjen. Til slutt kan Delaunay-trianguleringen, med eventuelle føringer, lagres til fil. Det er også mulig å lagre en kopi av framebufferet til fil i form av en bildefil (.bmp). Dette er nyttig for å lage figurer. For eksempel er de fleste trianguleringsfigurene i denne hovedfagsrapporten laget på denne måten.

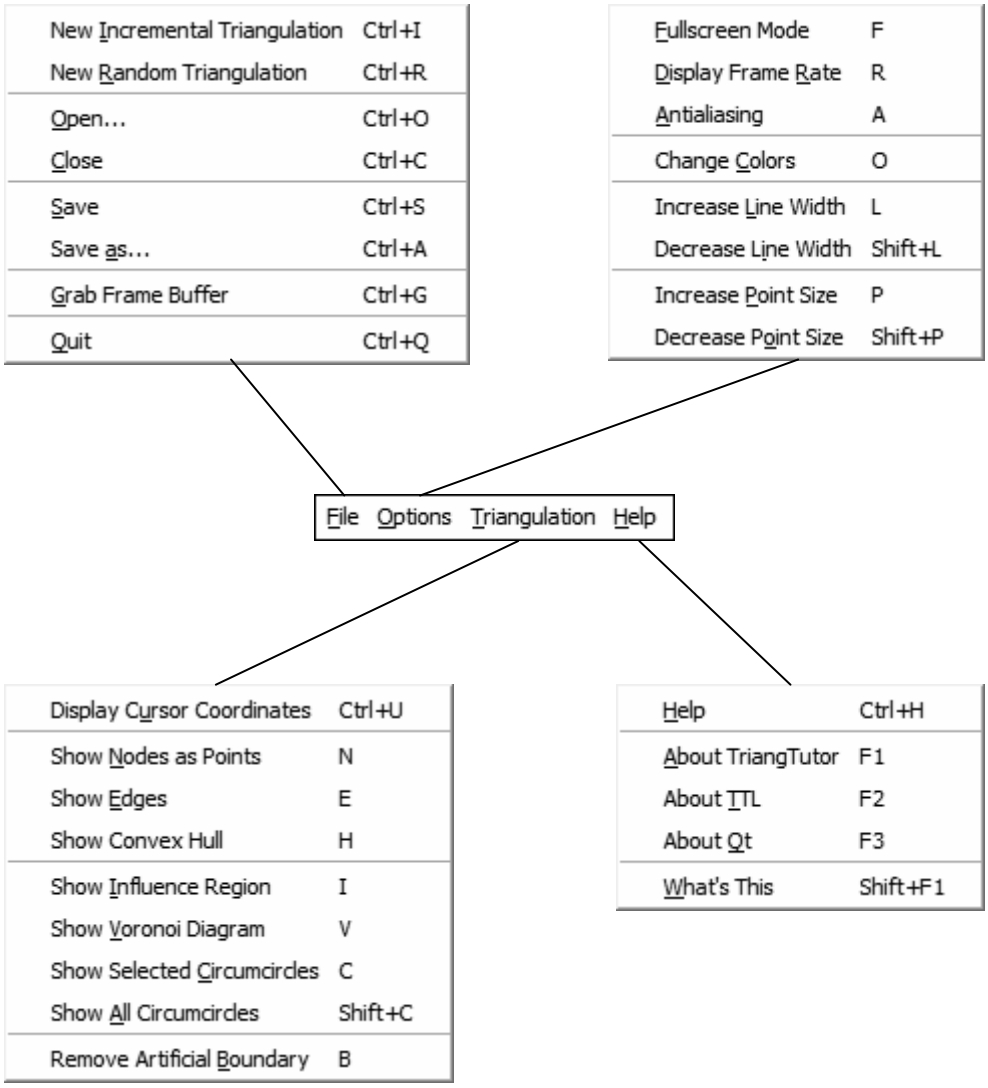
Funksjonaliteten som er nevnt i avsnittet over, er den vi har kalt for *grunnleggende* funksjonalitet i TriangTutor. Det er denne vi skal beskrive i dette kapitlet. Dette er funksjonalitet som er tett knyttet opp til funksjonaliteten i TTL, og som har direkte med det å lage og visualisere trianguleringer å gjøre. Funksjonaliteten som beskrives i neste avsnitt, er den vi har kalt for *avansert* funksjonalitet. Dette er funksjonalitet som visualiserer forskjellige egenskaper hos Delaunay-trianguleringer. Denne beskrives nærmere i neste kapittel.

Når et punkt settes inn eller fjernes i en Delaunay-triangulering, kan TriangTutor vise influensregionen til punktet. Det vil si det området i trianguleringen som forandrer seg. Det er videre mulig å vise Voronoi-diagrammet som svarer til en Delaunay-triangulering. En annen funksjon er å vise den omskrivende sirkelen for en mengde med utvalgte trekanter, eller for alle trekantene dersom dette skulle være ønskelig.

Det er også en del funksjonalitet som går mer på det visuelle som for eksempel linjetykkelse, punktstørrelse og bakgrunnsfarge. Mer om dette i neste seksjon. Sist, men ikke minst har TriangTutor funksjonalitet for å zoome ut og inn, samt å panorere horisontalt og vertikalt.

5.2 Menysystemet og statuslinjen

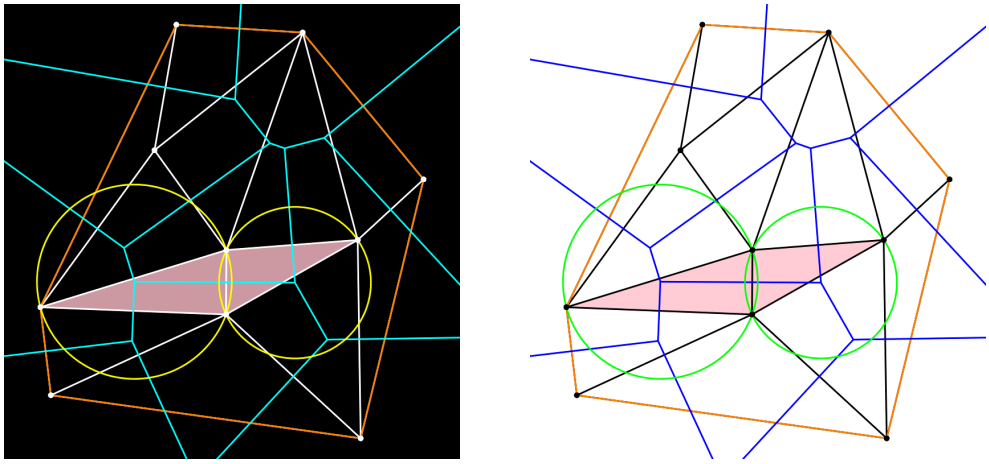
For at TriangTutor skal være mest mulig brukervennlig og enkel å ta i bruk, har applikasjonen et menysystem. Dette menysystemet er laget med Qt, som



Figur 5.1: Oversikt over menysystemet.

vi skal bli bedre kjent med i seksjon 7.4. Alle funksjonene som finnes i programmet har også *hurtigtaster*, som brukerne kan lære seg etter hvert.

Det første som møter brukeren av TriangTutor er et tomt vindu med en *menylinje* øverst. Menylinjen består av fire kategorier med hver sin undermeny: ”File”, ”Options”, ”Triangulation” og ”Help”. Å utforske de forskjellige undermenyene gir også en god oppsummering over hva slags funksjonalitet som finnes i applikasjonen. En skjematisk oversikt over hele menysystemet er gitt på *Figur 5.1*.



(a) (b)

Figur 5.2: To forskjellige fargetabeller: en med svart (a) og en med hvit bakgrunnsfarge (b).

For å opprette en ny Delaunay-triangulering må brukeren først gå inn på ”File”-undermenyen. Her finner vi igjen de tre alternative måtene som programmet tilbyr for å bygge opp en triangulering, som vi nevnte i forrige seksjon. Vi finner også funksjonalitet for å lagre trianguleringen, lagre en kopi av framebufferet som et bilde, lukke trianguleringen og avslutte programmet.

Undermenyen ”Options” inneholder en del innstillinger som går på det visuelle. Øverst har vi en menyknapp for å skifte mellom fullskjerm- og vindusmodus. Under denne har vi en knapp som gjør at applikasjonen viser *frameraten*, det vil si antall skjermbilder som tegnes opp per sekund. Denne funksjonen er beregnet for å måle ytelsen til programmet og TTL. TriangTutor bruker som standard antialiasing, som vi så på i seksjon 4.5, for at linjene og punktene skal se glatte ut. Dette krever imidlertid grafikkmaskinvare som har støtte for antialiasing. Hvis programmet kjøres på maskinvare uten slik støtte, bør antialiasing skrus av gjennom ”Antialiasing”-knappen på menyen, for at ytelsen ikke skal bli dramatisk redusert. Menyknappen ”Change Colors” lar brukeren forandre *fargetabell* for applikasjonen. Vi har definert to fargetabeller som brukeren kan velge mellom: en med svart bakgrunn og en med hvit bakgrunn. Et eksempel på hvordan programmet tar seg ut med de to fargetabellene er vist på *Figur 5.2*. De fire nederste knappene i denne menyen lar brukeren justere tykkelsen på linjene, og størrelsen på punktene eller nodene. For eksempel kan det være

greit å bruke ganske tykke linjer og store noder, dersom TriangTutor brukes på en stor skjerm i undervisningssammenheng.

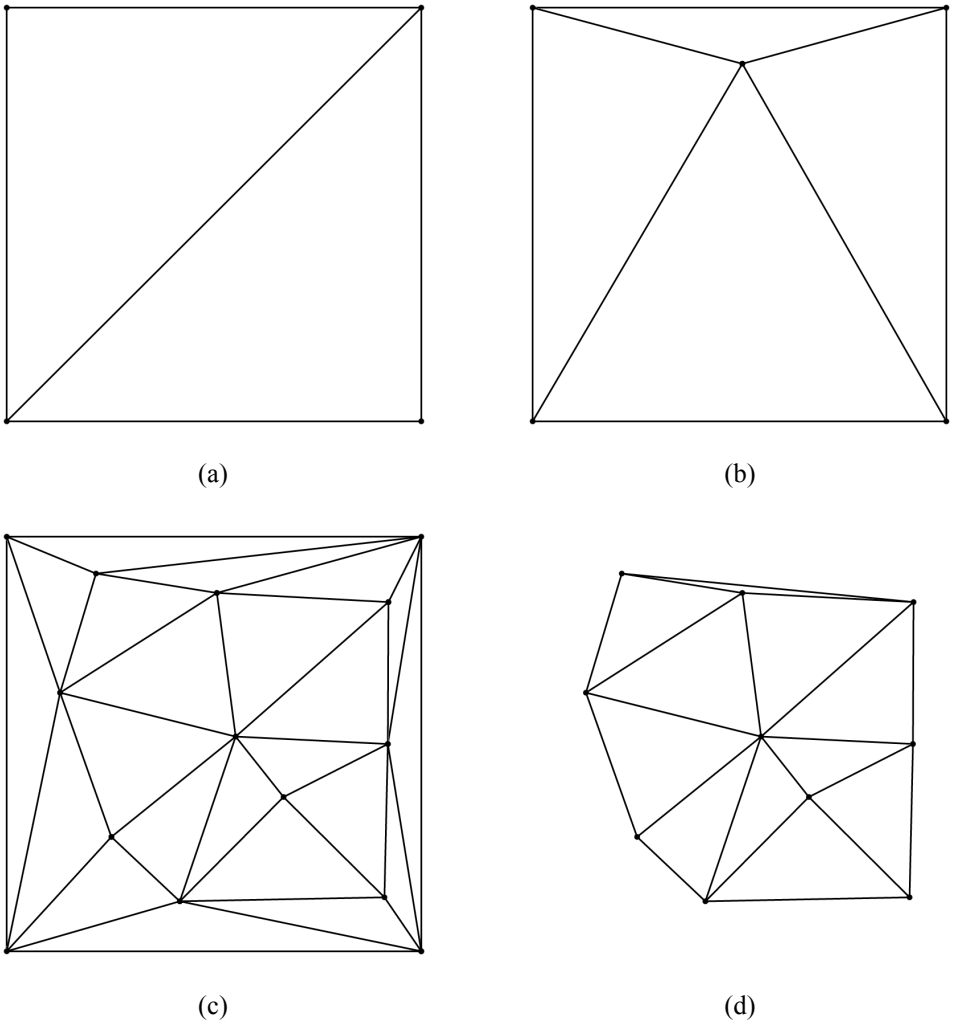
”Triangulation”-undermenyen inneholder funksjonalitet som er relatert til selve trianguleringen. ”Display Cursor Coordinates” lar brukeren velge om x- og y-koordinatene til musepekeren skal vises eller ikke. De tre neste menyknappene brukes til å styre hvilke deler av trianguleringen som skal tegnes til enhver tid. Brukeren kan velge om både kantene og nodene skal tegnes eller ikke, og videre om den konvekse innhylningen til mengden med noder skal markeres spesielt. Den neste gruppen med knapper brukes til å skru av og på visning av det vi har kalt for avansert funksjonalitet, altså influensregion, Voronoi-diagram og omskrivende sirkler. Nederst har vi en menyknapp som fjerner en eventuelt kunstig rand rundt en triangulering. Det blir mer om denne funksjonen i neste seksjon.

Den siste undermenyen ”Help” inneholder først og fremst en ”Help”-knapp, som åpner et vindu med en kort oversikt over all funksjonaliteten i programmet, og en beskrivelse av hvordan de forskjellige operasjonene utføres. De øvrige knappene gir litt informasjon til brukeren om selve TriangTutor, TTL og Qt.

Nederst i vinduet finner vi *statuslinjen* til TriangTutor. Her angis til enhver tid antall noder, antall kanter og antall trekanter i trianguleringen som vises. Det er også her koordinatene til musepekeren og frameraten blir vist. Alle meldinger fra TriangTutor til brukeren dukker også opp i statuslinjen. Dette kan for eksempel være informative meldinger eller feilmeldinger. Meldingene vises i noen få sekunder før de forsvinner.

5.3 Å bygge opp trianguleringer

Uansett hvilken av de tre måtene TriangTutor lager en Delaunay-triangulering på, så bygges trianguleringen opp etter den inkrementelle algoritmen vi så på i seksjon 3.4. Vi vil nå gå gjennom i detalj hvordan dette gjøres i TriangTutor, dersom brukeren velger ”New Incremental Triangulation” fra ”File”-undermenyen. Først opprettes fire noder som hver utgjør et av hjørnene i et rektangel. Disse sendes til TTL som lager en gyldig Delaunay-triangulering av disse. Resultat blir da en kunstig initiell triangulering som består av to trekanter, som vist på *Figur 5.3* (*a*). Randen på denne kunstige trianguleringen blir en slags ramme rundt den Delaunay-trianguleringen vi egentlig ønsker å lage. Innenfor denne rammen bygger vi opp trianguleringen vår, ved å sette inn de nodene vi ønsker, en etter en. Det



Figur 5.3: Stegene i oppbygningen av en inkrementell Delaunay-triangulering i TriangTutor.

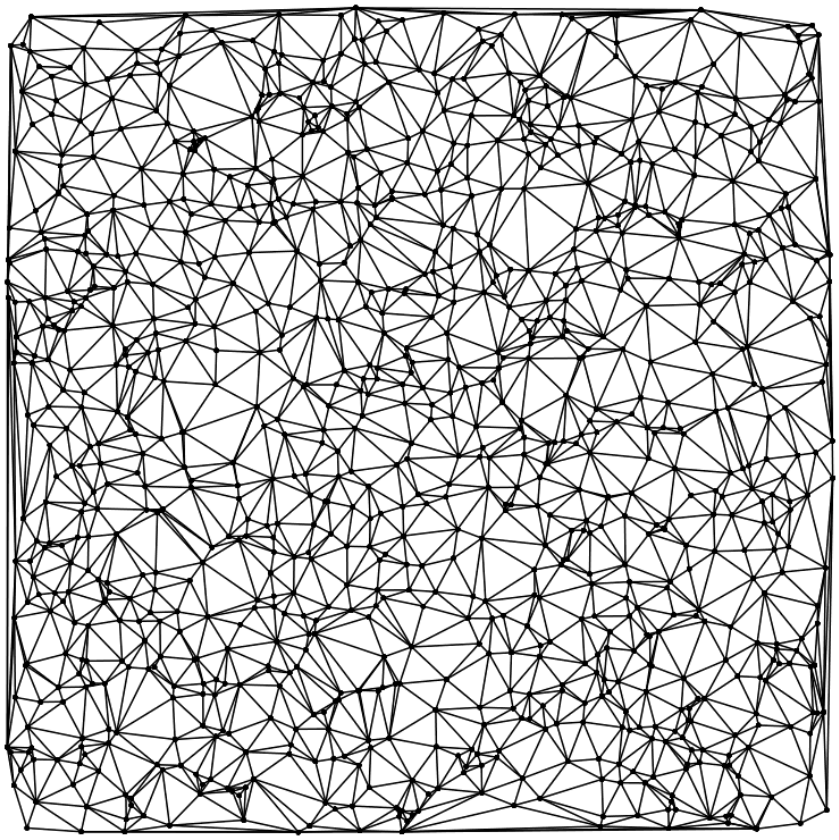
er ikke mulig å sette inn noder utenfor denne kunstige randen. En node settes inn ved å trykke på venstre museknapp. Applikasjonen oppretter da en node på posisjonen til musepekeren, og sender denne til TTL. Til dette anvendes funksjonen `insertNode()` i TTL-kjernen, som setter inn en ny node i en eksisterende Delaunay-triangulering ved å bruke den inkrementelle algoritmen. Vi får da ut en ny Delaunay-triangulering med fem noder, som vist på *Figur 5.3* (*b*). Samme prosedyre gjentas for alle nodene som skal settes inn. Den resulterende Delaunay-trianguleringen kan

for eksempel se ut som den på *Figur 5.3* (*c*). Det oppstår numeriske problemer i TTL dersom to noder settes inn på samme sted. TriangTutor må derfor sjekke at det er en minimumsavstand til nærmeste node, før en ny node kan settes inn. Etter at alle nodene som skal være med i trianguleringen har blitt satt inn, kan vi fjerne den kunstige randen ved å velge ”Remove Artificial Boundary” fra ”Triangulation”-menyen. TTL-kjernen har en innebygget funksjon som gjør nettopp dette (`removeRectangularBoundary()`). Randen til trianguleringen blir da den konvekse innhylningen til de nodene brukeren har satt inn. Dette er vist på *Figur 5.3* (*d*).

Måten vi har valgt å bygge opp en inkrementell Delaunay-triangulering på i TriangTutor, har den ulempen at alle nodene må settes inn innenfor den kunstige randen. Etter at denne randen har blitt fjernet må nye noder settes inn innenfor randen av selve trianguleringen. Det er ikke mulig å sette inn den kunstige randen igjen, dersom denne først har blitt fjernet. Bakgrunnen for at vi har valgt å gjøre det på denne måten, er hovedsakelig at TTL ikke har støtte for innsetting av noder utenfor randen av en triangulering. Det er imidlertid mulig at støtte for dette kan bli implementert i TTL senere.

En måte å omgå dette problemet på, uten å utvide selve TTL-kjernen, er å la trianguleringen starte med en veldig stor trekant som ikke tegnes opp, og som brukeren dermed ikke ser. Når brukeren legger inn noder vil alle disse befinne seg innenfor den store trekanten, og vi kan fortsatt bruke den inkrementelle algoritmen for å bygge opp Delaunay-trianguleringen. Hvis vi da heller ikke tegner opp noen av kantene som går ut mot de tre nodene i den store trekanten, vil bare de nodene som brukeren selv har lagt inn og Delaunay-trianguleringen av disse være synlig på skjermen. For at randen på denne trianguleringen skal se riktig ut, er det viktig at starttrekanten lages så stor som mulig.

Når TriangTutor i stedet leser inn en mengde med datapunkter fra fil, eller genererer en mengde med vilkårlige punkter, vil som nevnt tidligere Delaunay-trianguleringen bygges opp etter samme inkrementelle algoritme. For at algoritmen her skal gå fortest mulig, sorterer vi først punktene i leksikografisk økende orden, som vi var inne på i seksjon 3.4. Vi må også her passe på at ingen punkter er like. Det vil i praksis si at alle punkter må ha en minimumsavstand til nærmeste nabopunkt. Halvkant-datastrukturen, som følger med TTL, har en innebygget funksjon (`createDelaunay()`) som tar inn en mengde med punkter eller noder og lager en Delaunay-triangulering av disse. Funksjonen hvor TriangTutor genererer et gitt antall vilkårlige punkter innenfor enhetskvadratet, og lager en Delaunay-triangulering av disse, er nyttig for å lage trianguleringer med et stort antall



Figur 5.4: Delaunay-triangulering av 1000 vilkårlig plasserte noder innenfor enhetskvadratet.

noder. *Figur 5.4* viser en vilkårlig generert Delaunay-triangulering som består av 1000 noder.

5.4 Fjerning av noder

Punkter kan også fjernes fra en Delaunay-triangulering. Dette gjøres ved å trykke på høyre museknapp, med musepekeren over noden som skal fjernes. TriangTutor finner da den noden som ligger nærmest musepekeren, forutsatt at denne ligger innefor en fastsatt radius. For å fjerne noden brukes

funksjonen **`removeNode()`** i TTL-kjernen. Denne funksjonen swapper først kanter vekk fra noden til den har grad tre. Det vil si, som vi husker fra Kapittel 2, at antall kanter som møtes i noden er tre. Noden og de tre kantene befinner seg nå inne i en trekant, og disse kan fjernes fra trianguleringen. Trianguleringen oppdateres deretter til å være en Delaunay-triangulering gjennom en swappeprosedyre i TTL (**`optimizeDelaunay()`**).

Vi har valgt å ikke tillate å fjerne randnoder eksplisitt i TriangTutor, siden det ikke er mulig å sette disse inn igjen. Det er heller ikke tillatt å fjerne noder som er endepunkter i en føring, siden vi regner disse som fastlåste.

5.5 Interaktiv flytting av en node

En viktig funksjon, som gjør TriangTutor nyttig for forståelse i undervisningssammenheng, er muligheten til å ta tak i en node og dra denne rundt i trianguleringen. Hvis for eksempel en node dras fra den ene siden av en Delaunay-triangulering til den andre, vil trianguleringen hele tiden forandres og oppdateres i henhold til Delaunay-kriteriet underveis. Vi vil derfor hele tiden se at kanter swappes i området rundt noden som flyttes.

En enkel måte å flytte en node på, er å fjerne den, for så å sette den inn igjen like ved. Gjør vi dette gjentatte ganger ettersom musepekeren flytter seg, vil det se ut som noden beveger seg. Ved første forsøk på å implementere interaktiv flytting av en node prøvde vi denne metoden, og dette fungerte bra. Imidlertid så blir det noe unødvendig arbeid for datamaskinen, siden trianguleringen oppdateres til å være en Delaunay-triangulering både når noden fjernes, og når den settes inn igjen. TTL blir også nødt til å lage et nytt nodeobjekt hver gang noden settes inn, og dette kan i praksis skje flere titalls ganger i sekundet.

En bedre måte å flytte en node på vil derfor rett og slett være å bare endre posisjonen til noden, siden denne kan endres eksplisitt i TTL. Det er da bare nødvendig å oppdatere trianguleringen til å være en Delaunay-triangulering en gang for hver lille bit noden flyttes. Oppdateringen skjer, som nevnt før, gjennom en swappeprosedyre i TTL (**`optimizeDelaunay()`**). Det er denne siste metoden TriangTutor bruker nå, men i praksis er det ikke noen merkbar ytelsesforskjell mellom de to metodene.

Vi tar tak i en node ved å holde venstre museknapp nede, samt trykke på Shift-tasten. Noden vil deretter følge musepekeren til vi slipper opp venstre

museknapp igjen. På samme måte som ved fjerning av en node velger TriangTutor den noden som ligger nærmest musepekeren, så lenge denne ligger innefor den fastsatte radiusen.

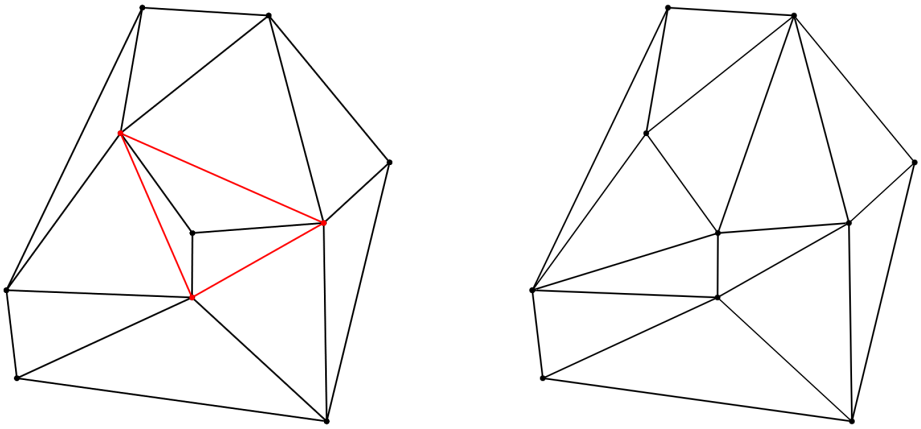
Av samme årsak som nevnt i forrige seksjon, er det ikke mulig å flytte noder som er endepunkter i en føring. Å flytte på randnoder er heller ikke mulig, siden dette ville krevd funksjonalitet i TTL for å håndtere noder utenfor randen.

5.6 Føringer

I TriangTutor finnes det to forskjellige måter å legge inn føringer eller fastlagte kanter i en Delaunay-triangulering på. Trianguleringen blir da en Delaunay-triangulering med føringer (Constrained Delaunay triangulation). Føringene tegnes opp med rød farge de to fargetabellene vi har definert i TriangTutor.

Den enkleste måten å sette inn en føring på, er å gjøre om en allerede eksisterende sidekant i trianguleringen til en føring. Dette gjøres ved å trykke på midtre museknapp over den kanten som skal bli en føring. Applikasjon finner den kanten som er nærmest musepekeren, og markerer denne som en føring gjennom funksjonen `setConstrained()` i halvkant-datastrukturen. En kant i halvkant-datastrukturen har et flagg som angir om den er en føring eller ikke. Trykker man på midtre museknapp over en kant som allerede er en føring, blir denne til en vanlig kant igjen. Etter denne operasjonen må trianguleringen oppdateres til å være en Delaunay-triangulering.

Den andre måten å sette inn føringer på, er å tegne disse direkte inn mellom to noder. Dette gjøres ved å plassere musepekeren over startnoden, og så trykke ned midtre museknapp samt Shift-tasten. Musepekeren flyttes deretter til endenoden til føringen, før man slipper opp museknappen. En hjelpelinje kommer fram underveis og viser hele tiden hvor føringen vil bli tegnet inn. TTL har en innebygget funksjon for å sette inn en føring mellom to noder (`(insertConstraint()`). Denne funksjonen swapper sidekanter helt til føringen blir en del av trianguleringen. Deretter markeres denne som en føring, og `optimizeDelaunay()` anvendes til å oppdatere trianguleringen til å bli en Delaunay-triangulering med føringer. TriangTutor må her passe på at brukeren ikke forsøker å tegne inn en føring som krysser en annen føring, ettersom dette vil føre til en feilsituasjon i TTL.



(a) (b)

Figur 5.5: En Delaunay-triangulering med (a) og uten føringer (b).

Et eksempel på en Delaunay-triangulering med føringer er vist på *Figur 5.5* (*a*). *Figur 5.5* (*b*) viser samme triangulering uten føringer.

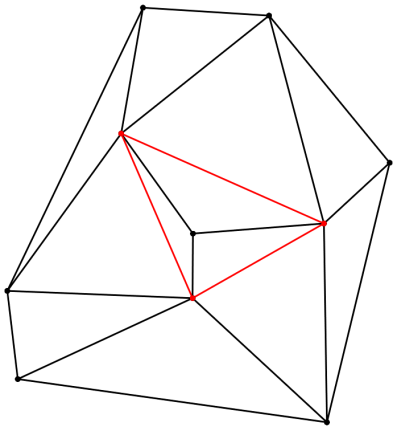
5.7 Filformatet

Filformatet som TriangTutor bruker er tekstbasert. Dette gjør det enkelt å lage testfiler ved hjelp av en vanlig teksteditor. Først i filen må det være et heltall som angir antall punkter eller noder som finnes i filen. Deretter følger koordinatene til alle punktene, i form av tre flyttall. Vi har valgt å inkludere *z*-koordinatene til punktene også, selv om ikke programmet gjør bruk av disse. En av grunnene til dette er at programmet kanskje en gang i fremtiden vil bli utvidet til å la punktene også ha høydeverdier. Dessuten gjør dette at TriangTutor enklere kan lese datafiler fra mange andre applikasjoner, som bruker tredimensjonale datapunkter.

Etter koordinatene til alle punktene kan filen enten være slutt eller den kan inneholde et nytt heltall, som angir antall føringer som finnes i filen. På samme måte som med nodene, følger deretter koordinatene til start- og endenodene til alle føringene. Etter at filen er lest inn bygges trianguleringen opp og føringene settes inn, på samme måte som ved manuell innsetting. Vi har dermed et enkelt filformat for å lagre og laste inn Delaunay-trianguleringer med og uten føringer. *Figur 5.6* viser et eksempel

```
10
0.470523 0.462514 0
0.0613491 0.336495 0
0.469683 0.320492 0
0.312366 0.682532 0
0.6383 0.941928 0
0.759264 0.484687 0
0.904226 0.61824 0
0.0844194 0.142494 0
0.360458 0.959289 0
0.765853 0.0475029 0
3
0.312366 0.682532 0
0.759264 0.484687 0
0.312366 0.682532 0
0.469683 0.320492 0
0.469683 0.320492 0
0.759264 0.484687 0
```

(a)



(b)

Figur 5.6: Et eksempel på en datafil for TriangTutor (a), og den resulterende trianguleringen (b).

på en enkel datafil for TriangTutor og den resulterende trianguleringen.

5.8 Zooming og panorering

TriangTutor har også funksjonalitet til å zoome inn og ut i en triangulering, samt å kunne flytte kameraet eller panorere i alle retninger. *Tabell 5.1* gir en oversikt over hvilke muligheter brukeren har til å stille inn og flytte på kameraet.

Tast	Beskrivelse
←	Flytter kameraet mot venstre.
→	Flytter kameraet mot høyre.
↑	Flytter kameraet opp.
↓	Flytter kameraet ned.
Page Up	Zoomer inn (forstørrer)
Page Down	Zoomer ut (forminsker)
End	Tilbakestiller zoom.
Home	Tilbakestiller kameraet.

Tabell 5.1: Oversikt over kamerabevegelser.

Kapittel 6

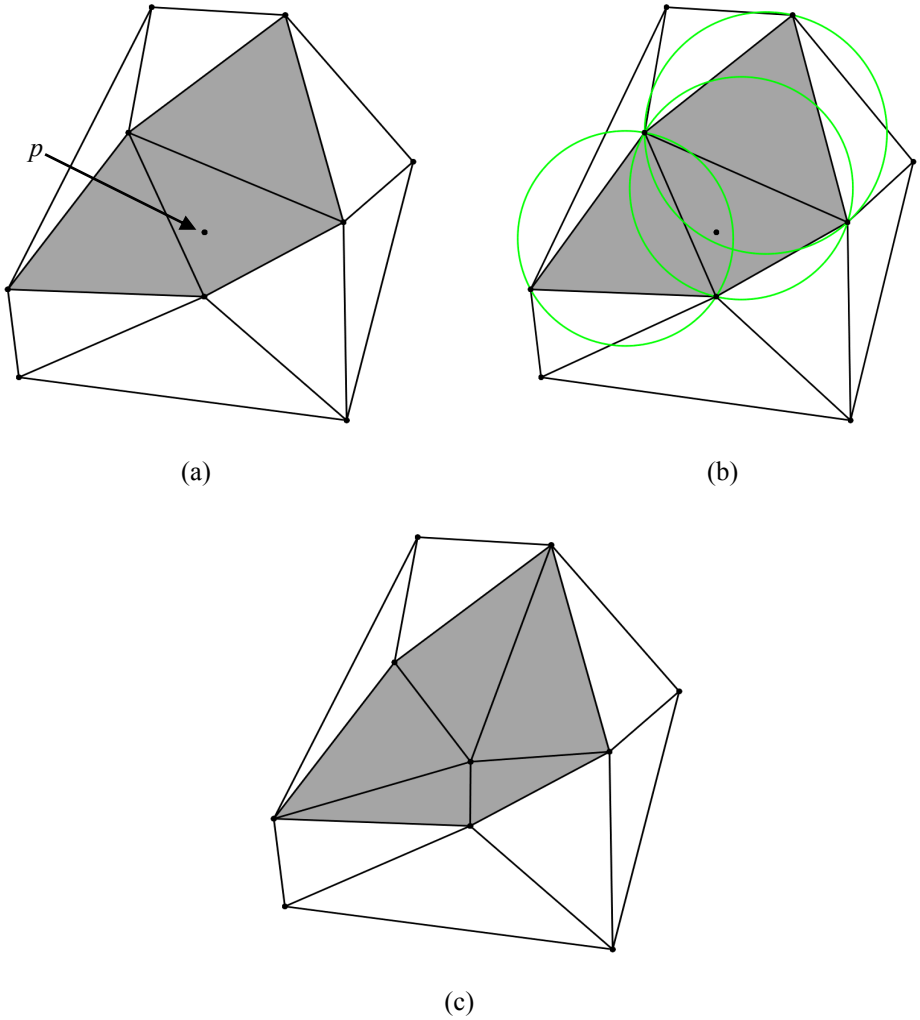
Avansert funksjonalitet

Vi har nå kommet fram til kapitlet hvor den mer avanserte funksjonaliteten i TriangTutor beskrives. I første seksjon skal vi se på hva en influensregion er og hvordan vi har valgt å visualisere denne. Neste seksjon vil ta for seg hvordan vi genererer Voronoi-diagram, og til slutt i dette kapitlet ser vi på funksjonaliteten for å vise omskrivende sirkler.

6.1 Influensregion

Når man setter inn et nytt punkt p i en Delaunay-triangulering Δ, er det som regel bare et begrenset område rundt p som må oppdateres eller retrianguleres for at trianguleringen fortsatt skal være en Delaunay-triangulering. Vi kaller dette området for *influensregionen* til p i Δ, som betegnes R^p. Randen til R^p kaller vi for *influenspolygonet* til p i Δ. Dersom p fjernes fra trianguleringen igjen er det også bare det samme området R^p som må retrianguleres.

Hvordan finner vi så influensregionen til et punkt p som skal settes inn i en Delaunay-triangulering Δ? Vi husker *Definisjon 2.3* eller sirkelkriteriet fra seksjon 2.3, som fastslo at de omskrivende sirklene til alle trekantene i Δ må være punktfrie. Ut i fra dette er det klart at de trekantene i Δ som har omskrivende sirkler som inneholder p, ikke lenger er Delaunay-trekanter og derfor må modifiseres. Dette er også de eneste trekantene som må forandres, og det er disse trekantene som utgjør influensregionen til p i Δ. *Figur 6.1* (*a*) viser en Delaunay-triangulering, samt et punkt p som skal settes inn i denne. Influensregionen til p er også markert på figuren. Som vi ser av *Figur 6.1* (*b*) er p inneholdt i de omskrivende sirklene til alle trekantene i



Figur 6.1: Influensregionen til et punkt *p* som skal settes inn i en Delaunay-triangulering (a) og (b), og den nye Delaunay-trianguleringen med *p* innsatt (c).

influensregionen til *p*. *Figur 6.1* (*c*) viser den nye Delaunay-trianguleringen etter at *p* har blitt satt inn. Vi legger her merke til at alle trekantene i influensregionen til *p* har *p* som felles node. Det er nettopp denne egenskapen vi vil utnytte, når vi nå skal visualisere influensregioner i TriangTutor.

TriangTutor kan vise influensregionen både når et punkt settes inn i en Delaunay-triangulering og fjernes fra en Delaunay-triangulering. Vi ønsker

ikke at TriangTutor selv skal beregne influensregioner, men å dra nytte av den informasjonen vi kan få fra TTL. I det første tilfellet setter vi inn punktet på vanlig måte. Etter at TTL har oppdatert trianguleringen finner vi enkelt influensregion ved hjelp av observasjonen over. Vi tar utgangpunkt i noden som akkurat ble satt inn, og finner alle kantene rundt denne. Dette kan gjøres ved å finne 0-orbiten til noden. Alle punktene som influenspolygonet består av ligger da i andre enden av disse kantene. Det er viktig å være klar over at en influensregion ikke nødvendigvis er konveks, når man skal tegne den opp.

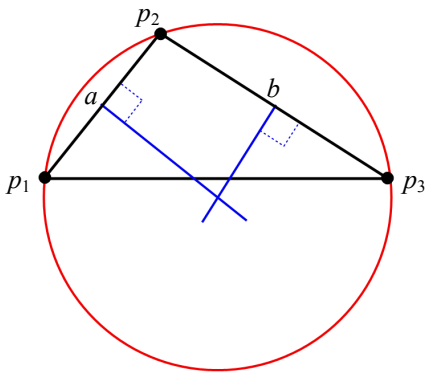
I tilfellet når en node skal fjernes fra Delaunay-trianguleringen finner vi influensregionen på samme måte som ved innsetting, men nå må vi ta utgangspunkt i noden og finne 0-orbiten, før den fjernes. Influensregionen kan også vises når en node flyttes rundt, og viser da området som til enhver tid forandrer seg. Vi finner her influensregionen på samme måtte som ved innsetting, rett etter at noden har blitt flyttet og Delaunay-trianguleringen har blitt oppdatert. TriangTutor markerer en influensregion med lysegrå farge i de to fargetabellene vi har definert, og denne vises bare i noen få sekunder før den forsvinner.

6.2 Voronoi-diagram

Vi så i seksjon 2.3 at en Delaunay-triangulering og det tilsvarende Voronoi-diagrammet er dualer. Det vil si at gitt en Delaunay-triangulering så kan det tilsvarende Voronoi-diagrammet avledes fra denne og omvendt. Vi skisserte også hvordan dette kan gjøres i seksjon 2.3. Det er nettopp på denne måten TriangTutor beregner et Voronoi-diagram. TTL generer Delaunay-trianguleringen, så det TriangTutor trenger å gjøre, er å avlede Voronoi-diagrammet fra denne.

Vi vil nå gå mer i dybden på hvordan TriangTutor foretar denne avledningen. For hver trekant i Delaunay-trianguleringen finnes det et Voronoi-punkt, og dette ligger i sentrum av den omskrivende sirkelen til trekanten. Det første vi trenger er altså en metode for å beregne senteret i den omskrivende sirkelen til en trekant.

Geometrisk sett kan vi finne dette senteret ved å konstruere midtnormalen til to av kantene i trekanten, og deretter finne skjærings-punktet mellom disse som vist på *Figur 6.2*. Vi har kalt de tre punktene i trekanten for p_1, p_2 og p_3 med koordinater (x_1, y_1), (x_2, y_2) og (x_3, y_3). Linja som går gjennom p_1 og p_2 kaller vi for linje a, mens linje b går gjennom



Figur 6.2: Senteret i den omskrivende sirkelen til en trekant.

punktene p_2 og p_3. Likningene for disse to linjene er da:

$$y_a = m_a(x - x_1) + y_1$$

$$y_b = m_b(x - x_2) + y_2,$$

der m_a og m_b er stigningstallene gitt ved:

$$m_a = \frac{y_2 - y_1}{x_2 - x_1} \text{ og } m_b = \frac{y_3 - y_2}{x_3 - x_2}.$$

Likningene til de to linjene som står vinkelrett på linje a og b, og som går gjennom midtpunktene på linjestykkene p_1p_2 og p_2p_3, blir da:

$$y'_a = -\frac{1}{m_a}\left(x - \frac{x_1 + x_2}{2}\right) + \frac{y_1 + y_2}{2}$$

$$y'_b = -\frac{1}{m_b}\left(x - \frac{x_2 + x_3}{2}\right) + \frac{y_2 + y_3}{2}.$$

Senteret til den omskrivende sirkelen til trekanten som utspennes av punktene p_1, p_2 og p_3 befinner seg i skjæringspunktet mellom disse to linjene. Vi kan finne dette ved å sette de to linkningene lik hverandre, og løse med hensyn på x. Vi kan deretter finne y-verdien ved å sette inn for x i

en av de to linkningene. Resultatet av disse beregningene gir:

$$x = \frac{m_a m_b (y_1 - y_3) + m_b (x_1 + x_2) - m_a (x_2 + x_3)}{2(m_b - m_a)}$$

$$y = \frac{m_b (y_2 - y_3) + m_a (y_1 + y_2) - x_1 + x_3}{2(m_b - m_a)}.$$

Disse likningene egner seg derimot ikke til å brukes i et dataprogram, fordi det oppstår problemer dersom en av linjene y_a eller y_b er loddrett. Vi kan derfor ikke basere generelle beregninger av skjæringen mellom to linjer på disse eksplisitte utrykkene for y_a og y_b.

Vi har i TriangTutor valgt å bruke følgende likninger for å finne senteret i en omskrivende sirkel for en trekant:

$$x = x_1 - \frac{(y_2 - y_1)d_{31}^2 - (y_3 - y_1)d_{21}^2}{2((x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(y_3 - y_1))}$$

$$y = y_1 + \frac{(x_3 - x_1)d_{21}^2 - (x_2 - x_1)d_{31}^2}{2((x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(y_3 - y_1))},$$

der d_{ij}^2 er kvadratet av avstanden mellom to punkter p_i og p_j:

$$d_{ij}^2 = (x_j - x_i)^2 + (y_j - y_i)^2.$$

Disse likningene, som blant annet brukes av Jonathan R. Shewchuk (http://www.ics.uci.edu/~eppstein/junkyard/circumcenter.html), er numerisk stabile og dessuten raske å beregne på grunn av relativt få divisjoner og multiplikasjoner.

Utledningen av disse likningene tar utgangspunkt i likningen for en sirkel med sentrum (a, b) og radius r:

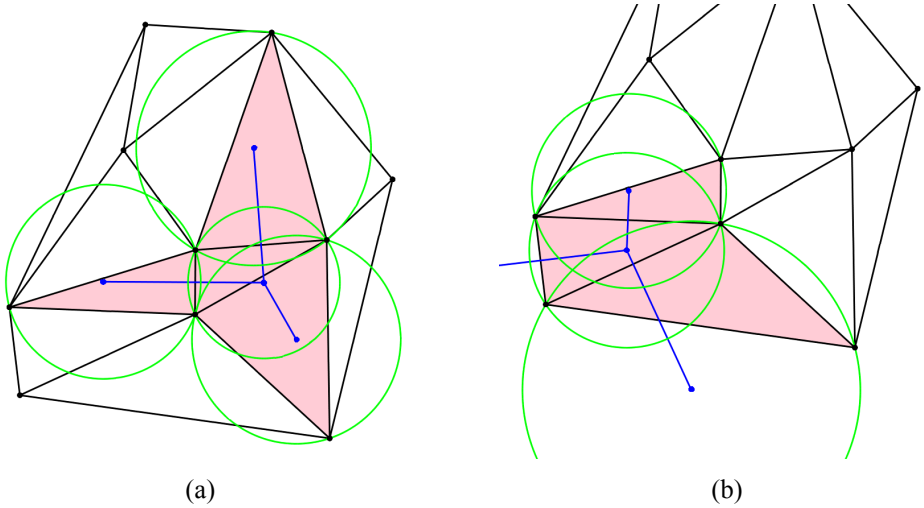
$$(x - a)^2 + (y - b)^2 = r^2.$$

Denne ekspanderes til

$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 = r^2.$$

Algoritme 6.1: `createVoronoiDiagram(triangulation)`

```
01    for each triangle in triangulation
02      source_node <- findCircumcenter(triangle)
03      for each half_edge in triangle
04        if twin_edge != 0 // Interior edge
05          if for twin_edge:
              (x_source > x_target) or
              (x_source = x_target and y_source > y_target)
06            target_node <- findCircumcenter(neighbour_triangle)
07            drawVoronoiEdge(source_node, target_node)
08        else                // Boundary edge
09          vector <- findPerpendicularUnitVector(half_edge)
10          target_node <- voronoi_source_node + 100 * vector
11          drawVoronoiEdge(source_node, target_node)
```



(a) (b)

Figur 6.3: Beregning av Voronoi-kanter ut i fra en trekant som ikke ligger på randen (a), og en trekant som ligger på rander (b).

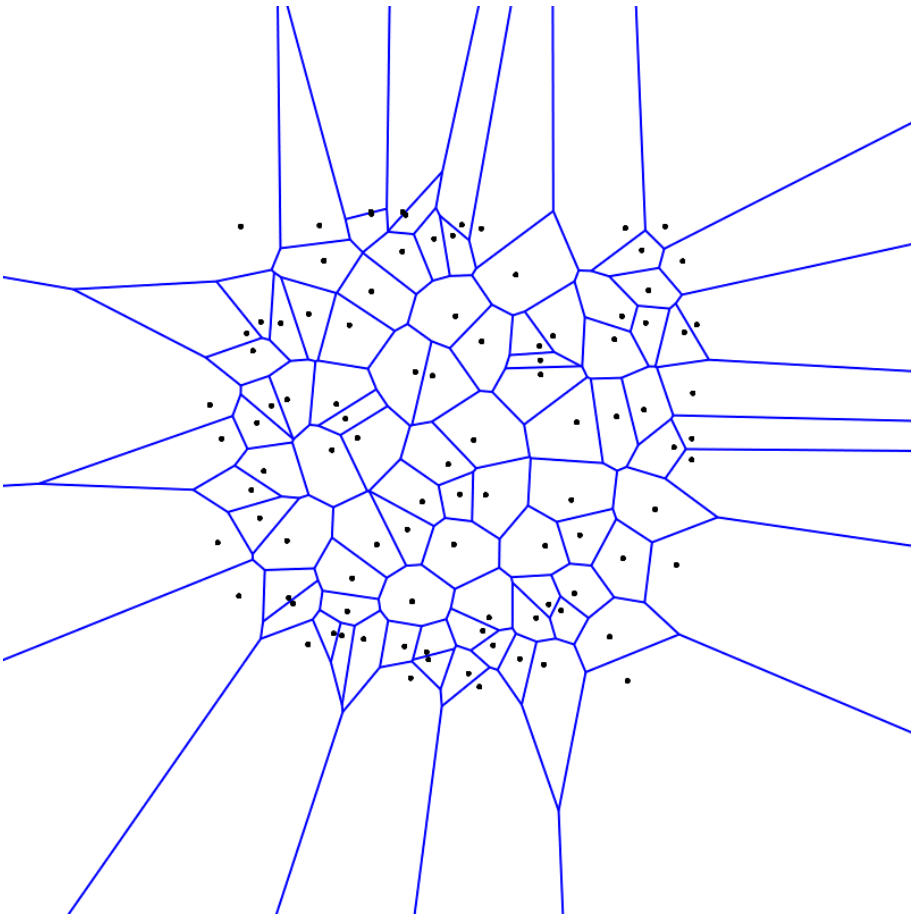
Siden denne likningen ikke er lineær lar vi

$$q = r^2 - a^2 - b^2.$$

Etter litt omstokking får vi da:

$$(2x)a + (2y)b + q = x^2 + y^2,$$

som er lineær med hensyn på a, b og q. Setter vi inn for de tre punktene p_1,



Figur 6.4: Eksempel på et Voronoi-diagram for 100 punkter.

p_2, og p_3 får vi et likningssystem, som kan skrives på matriseform og løses ved hjelp av *Cramers regel* (se f.eks. [Leo98], s. 101 for Cramers regel).

Nå som vi har en numerisk stabil metode for å beregne senteret i den omskrivende sirkelen til en trekant, kan vi gå videre med å forklare hvordan TriangTutor avleder Voronoi-diagrammet fra en Delaunay-triangulering. For hver trekant i trianguleringen beregnes senteret i den omskrivende sirkelen, og sentrene i de omskrivende sirklene til de tre nabotrekantene som den deler en sidekant med. Det tegnes så tre Voronoi-kanter mellom disse som vist på *Figur 6.3* (*a*). Dette fungerer bare dersom trekanten ikke ligger på randen. Dersom den ligger på randen tegner vi en Voronoi-kant som står vinkelrett på randkanten og strekker seg langt vekk fra trianguleringen. Denne situasjonen ser vi på *Figur 6.3* (*b*). I teorien vil en slik Voronoi-kant være uendelig lang, men vi nøyer oss med å la den være 100 ganger større

enn lengden på selve trianguleringen. Etter at dette er gjort for alle trekantene, har vi tegnet ferdig Voronoi-diagrammet som svarer til Delaunay-trianguleringen. Vi har summert opp denne framgangsmåten i *Algoritme 6.1*. For å unngå at samme Voronoi-kant beregnes og tegnes opp to ganger bruker vi også en avskjæringstest. Denne finner vi på linje 05.

TriangTutor lagrer ikke noe av det den beregner i forbindelse med genereringen av Voronoi-diagrammet, bortsett fra en displayliste med selve tegningen. Når trianguleringen forandres forkastes denne, og alt beregnes på nytt. *Figur 6.4* viser et eksempel på et Voronoi-diagram laget av programmet. Hvis det settes inn føringer i en triangulering, slik at den ikke lenger er noen Delaunay-triangulering, så er det selvsagt ikke mulig for TriangTutor å vise noe Voronoi-diagram.

6.3 Omskrivende sirkler

Vi har i forrige seksjon sett hvordan TriangTutor kan beregne senteret i den omskrivende sirkelen til en trekant, og det er også dette vi må gjøre når vi skal tegne opp omskrivende sirkler. Siden ikke OpenGL har noen innebygd funksjon for å tegne sirkler, lager TriangTutor først en statisk displayliste med en enhetssirkel. Denne genereres ved hjelp av de vanlige parametriske likningene for en sirkel med radius r:

$$x = r \cos t \text{ og } y = r \sin t, \text{ der } 0 \leq t < 2\pi.$$

Selve sirkelen tegnes opp ved hjelp av korte linjestykker mellom punkter som beregnes ved hjelp av disse likningene. For lange og tynne trekanter kan vi få veldig store omskrivende sirkler. Det er derfor viktig å bruke tilstrekkelig antall punkter, slik at sirkelen også i disse tilfellene ser rundt ut og ikke kantete. 512 har vist seg å være et passe antall punkter.

Det TriangTutor må gjøre for å tegne den omskrivende sirkelen til en trekant, er først å beregne senteret i trekanten. Til dette brukes, som nevnt, metoden fra forrige seksjon. Videre beregnes radiusen til sirkelen ved å finne avstanden mellom en node i trekanten og sentrum i sirkelen. Sirkelen tegnes deretter opp ved først å skalere displaylisten med den statiske enhetssirkelen i henhold til den beregnede radiusen. Deretter forflyttes den til riktig posisjon og eksekveres. Vi har summert opp dette i *Algoritme 6.2*.

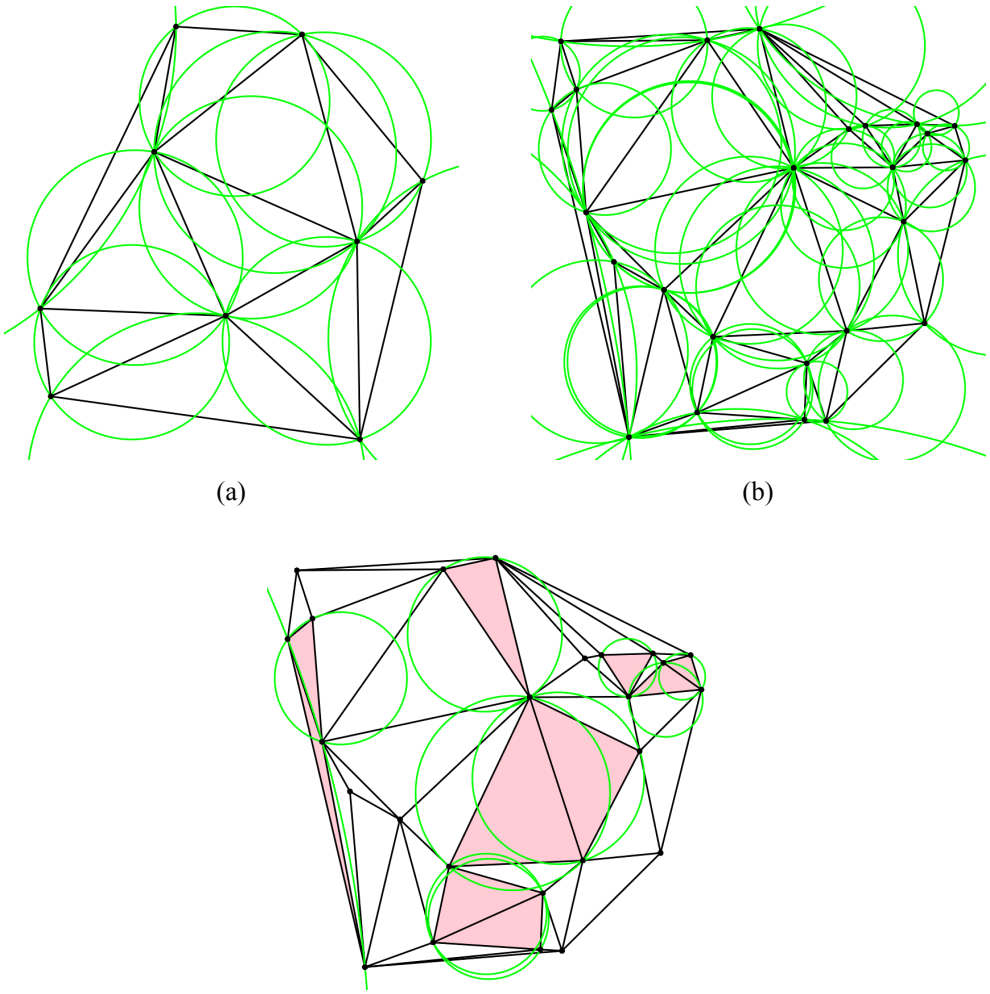
I TriangTutor er det mulig å velge å vise de omskrivende sirklene til alle trekantene i en triangulering. Dette fungerer veldig dårlig dersom

Algoritme 6.2: `createCircumcircle(triangle)`

```
01    if unitcircle = 0
02      unitcircle <- createUnitCircle() // This is done only once
03    center <- findCircumcenter(triangle);
04    radius <- findCircumradius(triangle, center);
05    drawCircumcircle(center, radius);
```

trianguleringen består av mer enn noen få trekanter. Som vi ser går det greit å vise alle de omskrivende sirklene til trianguleringen på *Figur 6.5* (*a*), siden denne bare består av 10 trekanter. Trianguleringen på *Figur 6.5* (*b*) derimot inneholder 40 trekanter, og her gjør alle sirklene at bildet blir veldig uoversiktlig. For også å kunne vise bare noen omskrivende sirkler i store trianguleringer, har vi lagt til funksjonalitet som gjør det mulig å merke trekanter, slik at det bare er den omskrivende sirkelen til disse som vises.

Når vi merker en trekant i TriangTutor, ønsker vi at denne trekanten skal være merket så lenge den eksisterer. Det vil si at selv om vi flytter på nodene i trekanten så skal denne fortsatt være merket, så lenge ikke en av kantene i trekanten swappes og nye trekanter dannes i stedet. For å få til dette må TriangTutor lagre nodene og sidekantene til trekantene som merkes. De merkede trekantene lagres i en liste, og denne må hele tiden oppdateres ettersom trianguleringen endrer seg. Når nye noder settes inn, noder fjernes eller noder flyttes på, må TriangTutor sjekke om noen av de merkede trekantene forandrer seg eller blir borte. En trekant merkes ved å holde Control-tasten inne samt trykken på venstre museknapp. Gjøres dette med en trekant som allerede er merket forsvinner merkingen igjen. Merkede trekanter farges rosa i de to fargetabellene vi har definert i TriangTutor. På *Figur 6.5* (*c*) ser vi den samme trianguleringen som på *Figur 6.5* (*b*), men nå vises bare de omskrivende sirklene til de 10 merkede trekantene. Bildet blir nå mye mer oversiktlig.



(a) (b)

(c)

Figur 6.5: Visning av alle omskrivende sirkler for en triangulering med 10 trekanter (a), og for en triangulering med 40 trekanter (b), samt visning av omskrivende sirkler for 10 av trekantene i trianguleringen med 40 trekanter (c).

Kapittel 7

Programstrukturen

Til nå har vi ikke sagt noe særlig om hvordan koden til TriangTutor er bygget opp og organisert. Dette vil vi se på i dette kapitlet. Først vil vi se på hvilke klasser applikasjonen består av, og hva de forskjellige klassene gjør. I seksjon 7.2 vil vi se på hva det som vi har kalt for grafiske objekter inneholder, og i seksjon 7.3 gir vi en beskrivelse den mer generelle geometriske verktøygruppen som TriangTutor bruker. Qt, som er et verktøy for å lage grafiske brukergrensesnitt, ser vi nærmere på i seksjon 7.4. Vi har i tillegg til den vanlige versjonen av TriangTutor, som bruker Qt som rammeverk, også laget en versjon som kun bruker OpenGL. I den siste seksjonen i dette kapitlet skal vi komme inn på bakgrunnen for dette.

7.1 Klasseinndelingen

Under arbeidet med TriangTutor har det hele tiden vært viktig å dele opp programmet i mindre deler, slik at det blir mer oversiktlig og dermed lettere å arbeide med. Denne oppdelingen vil også være nyttig for senere utvidelser og vedlikehold av applikasjonen. Det blir også lettere å gjenbruke deler av programmet. Siden TriangTutor er implementert i C++, som er et *objektorientert* programmeringsspråk, har det vært naturlig å dele opp applikasjonen i *klasser*. Det meste i applikasjonen som kan ses på som egne objekter har blitt skilt ut i egne klasser. Som et resultat av denne oppdelingen består programmet av 17 klasser. I tillegg har programmet en *gruppe* med generelle geometriske verktøyfunksjoner, som vi skal ta en kikk på i seksjon 7.3. Vi vil nå se nærmere på hvilke klasser TriangTutor består av, og hva disse inneholder. De to første klassene vi skal se på er *subklasser* av klasser i Qt. Qt er et verktøy for å lage grafiske brukergrensesnitt eller

GUI (Graphical user interface) for applikasjoner, som vi skal komme tilbake til i seksjon 7.4.

Noe som er typisk for mange av klassene i TriangTutor er at de består av objekter som vi kan kalle for *grafiske objekter*. Disse objektene kan vi se på skjermen i form av datagrafikk. I neste seksjon vil vi se nærmere på hvordan slike grafiske objekter er bygget opp, og hvilke funksjoner de inneholder. Nå følger en kort presentasjon av alle klassene.

ApplicationWindow

Denne klassen er en subklasse av **QMainWindow** i Qt, og styrer selve vinduet til TriangTutor. Den inneholder hele menysystemet, og har en peker til et **statusBar**-objekt. **ApplicationWindow** fungerer som et rammeverk rundt hele applikasjonen. I vinduet som **ApplicationWindow** styrer befinner det seg en **GLScene**-objekt, som vi nå skal se på.

GLScene

Dette er klassen for OpenGL-scenen hvor all grafikken tegnes opp. Den er en subklasse av **QGLWidget** i Qt. I **GLScene** er funksjonen som eksekveres hver gang grafikken i vinduet oppdateres implementert. Videre finnes det funksjoner som styrer hva TriangTutor skal gjøre når brukeren anvender musa i grafikkvinduet. **GLScene** har en peker til et **Camera**-objekt, et **ColorTable**-objekt og et **Triangulation2D**-objekt. Disse vil vi nå se på i tur og orden.

Camera

Dette er en generell klasse for et virtuelt kamera. Den inneholder funksjoner for å bevege kameraet i x- og y-retning over et plan, samt å zoome inn og ut. Disse funksjonene er veldig korte og er implementert som *inline-funksjoner* [Str97] for å lage mest mulig effektiv kode, slik at kamerabevegelsene går så glatt som mulig.

ColorTable

Klassen **ColorTable** holder oversikten over hva slags farge som skal brukes på de forskjellige grafiske objektene. At alt som har med farger å gjøre skilles ut i en egen klasse gjør at det blir enklere å gå inn i koden for å forandre på fargen til et objekt. Det er implementert to forskjellige fargetabeller i denne klassen, som brukeren kan velge mellom.

Triangulation2D med subklasser

Dette er en abstrakt klasse som implementerer alle de funksjonene som er felles for de tre måtene å bygge opp en Delaunay-triangulering på. Når en ny triangulering skal lages opprettes det ikke et **Triangulation2D**-objekt, men enten et **IncrementalTriangulation**-objekt, et **FileTriangulation**-objekt eller et **RandomTriangulation**-objekt. Disse er alle er subklasser av **Triangulation2D**-klassen. Måten disse tre objekttypene opprettes på er forskjellig, så de har blant annet hver sin *konstruktør*. Felles for disse objektene er at de inneholder en peker til et **Triangulation**-objekt i TTL. De inneholder dessuten pekere til alle de seks andre grafiske objektene som programmet består av, og som er de neste objektene vi skal se på. **Triangulation2D**-objektene er også grafiske objekter i seg selv. Alle de grafiske objektene inneholder en peker til et **ColorTable**-objekt.

HelpLine

HelpLine-klassen inneholder det som trengs for å kontrollere og tegne opp hjelpelinjen som kommer fram når brukeren skal tegne inn en føring.

ConvexHull

Denne klassen inneholder funksjoner for å finne og tegne opp den konvekse innhylningen til nodene i Delaunay-trianguleringen. Dette vil i praksis si randen til trianguleringen.

InfluenceRegion

I denne klassen har vi samlet funksjonene som brukes til å beregne og tegne opp influensregionen til noden som sist ble satt inn, fjernet eller flyttet på. Detaljene rundt beregningen av influensregioner så vi på i seksjon 6.1.

Voronoi

Vi husker fra seksjon 6.2 framgangsmåten for å avlede et Voronoi-diagram fra en Delaunay-triangulering. Koden for å gjøre dette er implementert i **Voronoi**-klassen.

Circumcircle

Denne klassen har to konstruktører. Hvilken som brukes avhenger av om vi vil vise den omskrivende sirkelen til en bestemt trekant eller alle trekantene i en triangulering. Det finnes altså to varianter av dette objektet. Klassen implementerer *Algoritme 6.2* fra seksjon 6.3 for å beregne og tegne opp

omskrivende sirkler.

MarkedTriangles

Alle trekantene som markeres i en triangulering lagres i en liste i et **MarkedTriangles**-objekt. Hver markerte trekant er et **MarkedTriangle**-objekt, som blir den neste klassen vi skal se på. Det denne klassen gjør er å opprette og vedlikeholde listen av pekere til **MarkedTriangle**-objekter. Som vi nevnte i seksjon 6.3 skal en trekant som er markert fortsatt være markert så lenge ingen kanter i trekanten swappes. Dette gjelder også dersom nodene i trekanten blir flyttet på. For å få til dette må programmet gå igjennom listen med markerte trekanter hver gang trianguleringen endres. Når for eksempel en node fjernes må også eventuelle markerte trekanter som inneholder noden fjernes. Dersom en node settes inn eller flyttes på må det sjekkes at kantene i hver av de markerte trekantene fortsatt danner en trekant.

MarkedTriangle

I denne klassen lagres kantene, nodene og punktene til en markert trekant. Dessuten har et **MarkedTriangle**-objekt en peker til den omskrivende sirkelen til trekanten. Som vanlig i grafiske objekter, har vi også funksjoner for å tegne opp trekanten.

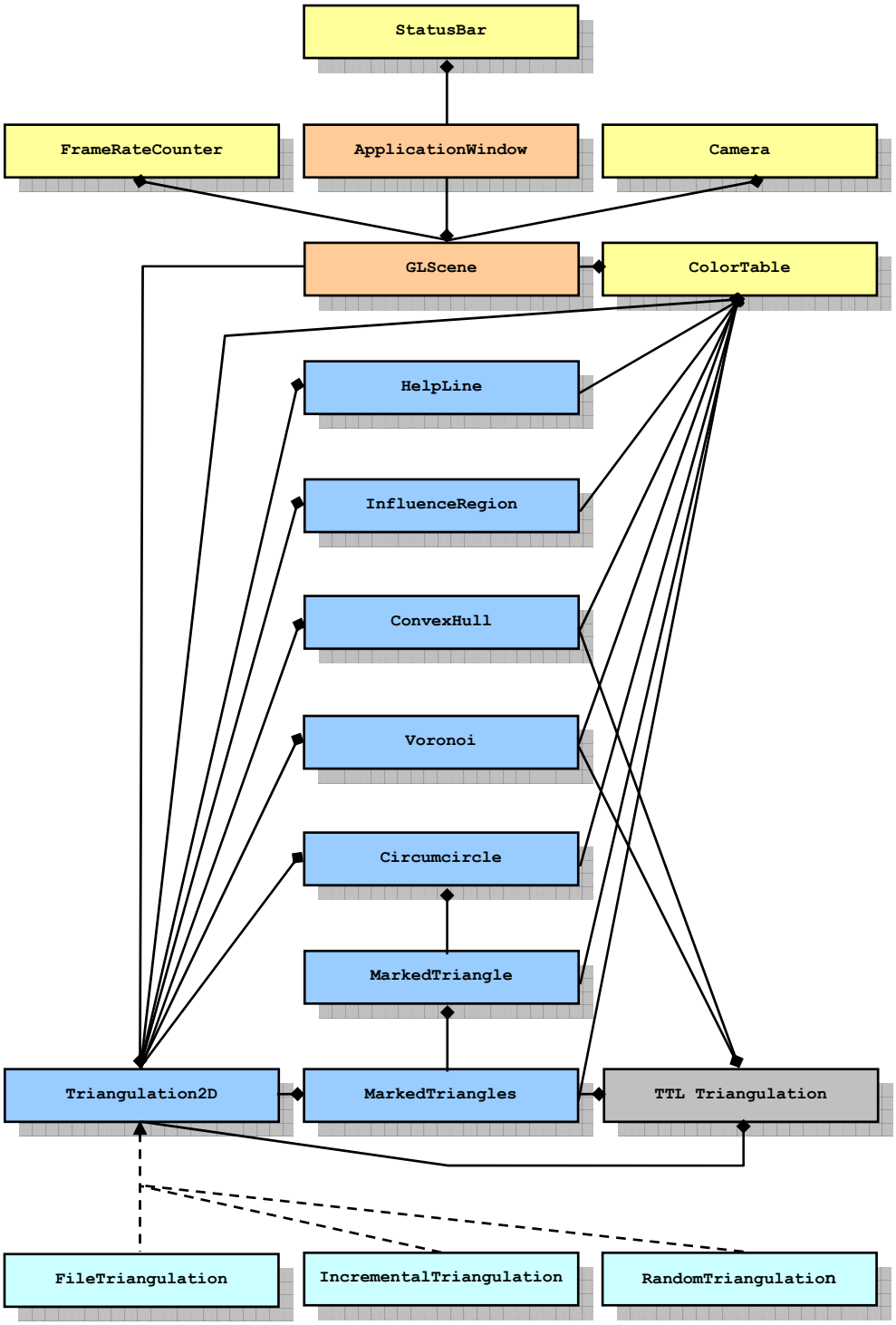
StatusBar

Denne klassen er en subklasse av **QStatusBar** i Qt og inneholder funksjoner for å vise og oppdatere informasjon i statuslinja nederst i vinduet.

FrameRateCounter

Dette er en enkel klasse for å beregne antall bilderammer TriangTutor klarer å generere i sekundet. Denne funksjonaliteten brukes stort sett til testing av ytelsen til programmet.

Vi har nå gått igjennom alle klassene applikasjonen består av og gitt en kort presentasjon av disse. På *Figur 7.1* har vi laget et forenklet klassediagram som viser en oversikt over alle klassene og forbindelsene mellom disse.



Figur 7.1: Klassediagram for TriangTutor. Heltrukne piler viser pekere mellom klassene, mens stiplede piler angir subklasser.

```
class GraphicalObject {

public:
  GraphicalObject(ttl::Triangulation* tri, ColorTable* ct);
  ~GraphicalObject();

  void callDisplayList();
  void updateObject();

private:
  GLuint createDisplayList();

private:
  ttl::Triangulation* triangulation_;
  ColorTable* color_table_;
  GLuint display_list_;

}
```

Figur 7.2: Innholdet i et grafisk objekt.

7.2 Grafiske objekter

Vi har kalt de objektene som vi kan se som datagrafikk på skjermen for grafiske objekter. Innholdet i klassene for disse objektene er stort sett likt. Når et slikt objekt opprettes sendes det med en peker til enten hele **Triangulation**-objektet i TTL, eller en del av det, for eksempel en trekant. Dette avhenger av hvor mye informasjon objektet trenger for å opprette seg selv. Videre sendes det med en peker til et **Colortable**-objekt. Objektene inneholder funksjoner til å beregne og tegne seg selv opp på skjermen. Dessuten har de fleste av dem en oppdateringsfunksjon, som brukes når trianguleringen og dermed kanskje også objektet forandrer seg. *Figur 7.2* gir en oversikt over hva et typisk grafisk objekt inneholder.

Fordelen med å ha grafiske objekter som tegner opp seg selv, er at hovedopptegningsfunksjonen i TriangTutor blir mye kortere og mer oversiktlig. Denne trenger da bare å kalle opptegningsfunksjonene til de grafiske objektene, som til enhver tid skal vises på skjermen. *Algoritme 7.1* viser litt forenklet hvordan hovedopptegningsfunksjonen ser ut.

Nesten alle de grafiske objektene i TriangTutor bruker displaylister når de tegner seg selv. I stedet for å ha en `draw()`-funksjon som tegner opp det samme objektet flere ganger, har de en `createDisplayList()`-funksjon og en `callDisplayList()`-funksjon. Den første av disse tegner opp objektet

Algoritme 7.1: `paintGL()`

```
01    resetOpenGL();
02    setCameraPosition()
03    if Triangulation2D != 0
04      if show_nodes
05        Triangulation2D->callNodesDisplayList()
06        Triangulation2D->callConstrainedNodesDisplayList()
07      if show_edges
08        Triangulation2D->callEdgesDisplayList()
09        Triangulation2D->callConstrainedEdgesDisplayList()
10      if show_convex_hull
11        ConvexHull->callDisplayList()
12      if show_influence_region
13        InfluenceRegion->callDisplayList()
14      if show_voronoi_diagram
15        Voronoi->callDisplayList()
16      if MarkedTriangles != 0
17        MarkedTriangles->callDisplayList()
18        if show_marked_circumcircles
19          MarkedTriangles->showCircumcircles()
20      if show_all_circumcircles
21        Circumcircles->callDisplayList()
22      if show_helpline
23        HelpLine->draw()
24    swapBuffers()
```

og legger det i en displayliste. Den andre viser innholdet i displaylista på skjermen. Vi så nærmere på hva en displayliste er i seksjon 4.3. Denne måten å tegne opp grafikk på er i de fleste tilfeller mye mer effektiv, enn å tegne opp objektet på nytt hver gang skjermen oppdateres. Innholdet i en displayliste kan ikke forandres på, så hvis objektet forandrer seg må det bygges opp en ny displayliste.

7.3 Verktøygruppen

De mer generelle geometriske beregningsfunksjonene som brukes i TriangTutor har vi skilt ut i en egen fil, hvor vi har definert en verktøygruppe (namespace **utils**). Disse funksjonene kan da brukes på tvers av klassene. For eksempel bruker både **Voronoi**-klassen og **Circumcircle**-klassen funksjonen som beregner senteret i den omskrivende sirkelen til en trekant. Andre eksempler på funksjoner som brukes av flere klasser, er funksjonen som finner avstanden mellom to noder og funksjonen som finner normalavstanden mellom en node og en linje. En oversikt over alle

Funksjon	Beskrivelse
`bool eqPoints(Node* p1, Node* p2)`	Sjekker om p_1 og p_2 ligger på samme sted (innenfor en angitt toleranse).
`bool ltLexPoint(Node* p1, Node* p2)`	Sjekker om p_1 er leksikografisk mindre enn p_2.
`double nodeDistance2D(Node* n1, Node* n2)`	Beregner avstanden mellom n_1 og n_2.
`double normalDistance2D(Node* p, Node* n1, Node* n2)`	Beregner normalavstanden fra p til linja mellom n_1 og n_2.
`bool findCircumcenter(Node* n1, Node* n2, Node* n3, double& x, double& y)`	Beregner senteret i den omskrivende sirkelen til trekanten utspent av n_1, n_2 og n_3.

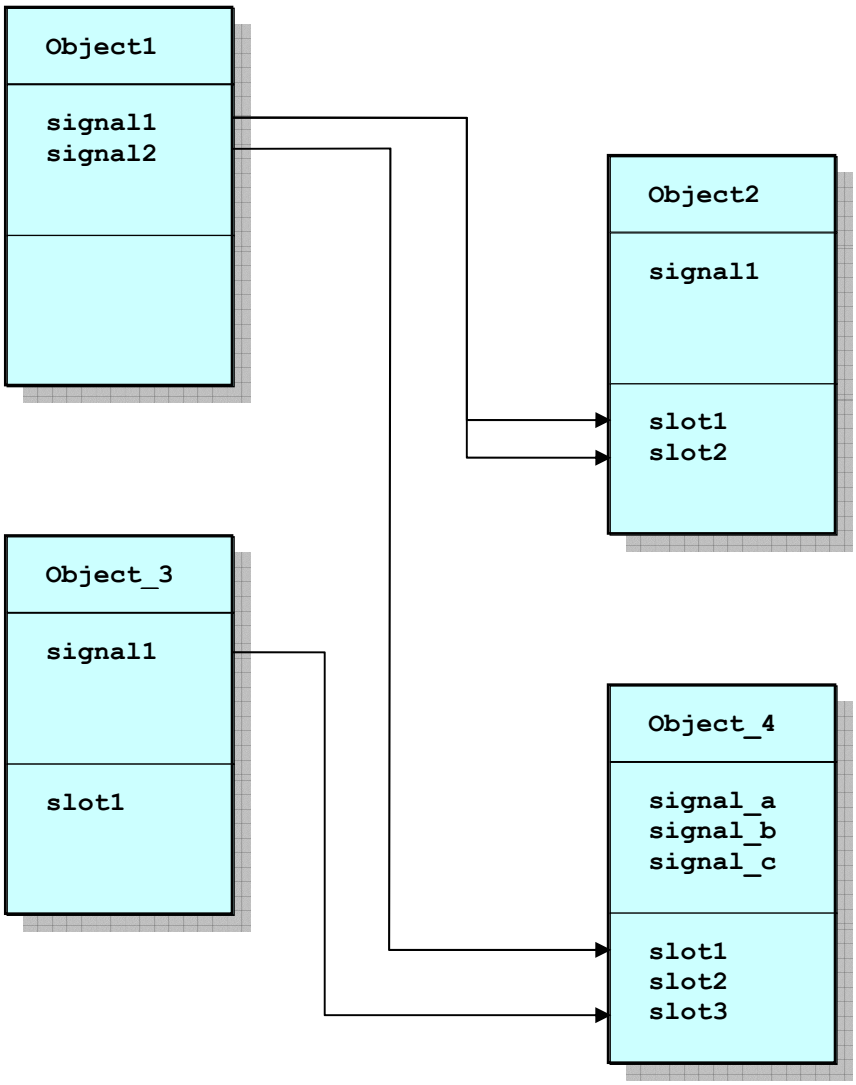
Tabell 7.1: Oversikt over funksjonene i namespace **utils**.

funksjonene i verktøygruppen er gitt i *Tabell 7.1*.

7.4 Qt

Qt er et komplett C++ rammeverk for utvikling av applikasjoner med grafisk brukergrensesnitt, utviklet av Trolltech AS. En viktig egenskap med Qt er at det er plattformuavhengig. Dette vil si at applikasjoner som er utviklet ved hjelp av Qt kan kjøres på en rekke plattformer. Qt inneholder en stor mengde med komponenter (widgets) til å bygge opp grafiske brukergrensesnitt. Applikasjonene får samme utseende som GUI-komponentene i vindussystemet til plattformen det kjøres under.

Qt bruker en ny mekanisme for å kommunisere mellom objekter, som kalles *"signals and slots"*. Denne mekanismen går ut på at en klasse kan inneholde signaler og slots eller kontaktpunker. Signalene kan den sende ut til andre klasser, mens med kontaktpunktene kan den ta imot signaler utenfra. Signalene og kontaktpunktene til de forskjellige klassene må kobles sammen et sted i koden. En komponent eller widget sender typisk ut et signal når det skjer en hendelse eller en brukerinteraksjon. En slik hendelse kan for eksempel være når brukeren trykker på en knapp på menyen. Signalet vil være koblet til en slot i en annen klasse. En slot er egentlig bare en funksjon i en klasse, som angir hva som skal skje når det kommer et signal inn på denne. "Signals and slots"-mekanismen gjør at de forskjellige klassene ikke trenger å vite om hverandre når de implementeres, og dette



Figur 7.3: Eksempel på bruk av ”signals and slots”.

gjør det mye lettere å utvikle mer elegant og generisk kode. Akkurat som med vanlige funksjoner, kan det overføres parametere fra et signal til en slot.

Tradisjonelt har GUI-applikasjoner inneholdt en mengde *callback-funksjoner*, som har angitt hva som skal skje ved forskjellige hendelser. Disse måtte som regel også være globale eller statiske, noe som gjorde det vanskelig å lage fullgod objektorientert design. Callback-funksjonene slipper man ved å bruke ”signal and slots”. *Figur 7.3* viser et eksempel på

bruk av ”signals and slots” mellom fire klasser. Pilene angir sammenkoblinger mellom signaler og kontaktpunkter (slots).

Grunnene til at vi har valgt å bruke Qt er mange. Først og fremst er det enkelt å lage et grafisk brukergrensesnitt, som gir applikasjonen et profesjonelt utseende. Qt er dessuten plattformuavhengig, og har innebygget støtte for utvikling av OpenGL-applikasjoner. Videre fører ”signals and slots”-mekanismen til mer elegant og robust kode i motsetning til å bruke callback-funksjoner, som ellers er vanlig. Qt er dessuten gratis å bruke dersom all kildekoden til applikasjonen distribueres sammen med applikasjonen.

Qt er grundig beskrevet i Qt 3.3 Whitepaper [QtW], som kan lastes ned fra Trolltech sine hjemmesider (http://www.trolltech.com). Det finnes ellers flere bøker som handler om programmering i Qt, for eksempel [Bla04].

7.5 To applikasjoner

I tillegg til den versjonen av TriangTutor som bruker Qt, har vi også laget en versjon som er implementert i ren OpenGL. Grunnen til at vi har valgt å lage to versjoner, er for å kunne kjøre applikasjonen på visningsveggen på Simula-senteret. Applikasjonen må da kjøres gjennom et system som heter Chromium [Hum02], som gjør at OpenGL-applikasjoner kan kjøres på store sammensatte skjermer. Vi vil beskrive visningsveggen og Chromium nærmere i Appendiks A. I stedet for Qt har vi da brukt GLUT (GL Utility Toolkit) som rammeverk. GLUT er OpenGL sitt eget rammeverk for kommunikasjon med vindussystemet. OpenGL-versjonen av TriangTutor har også et slags menysystem hvor alle menyene rendres i ren OpenGL. Vi har brukt PLIB (Portable Game Library, [PLI]) for å få til dette. Dette menysystemet er på langt nær så avansert som det vi kan lage med Qt. Det er heller ikke helt stabilt. Menyene vises dessuten ikke riktig på visningsveggen gjennom Chromium, så når vi har kjørt programmet på visningsveggen har det vært best å deaktivere menyene og bare bruke hurtigtastene.

Det har vist seg at interessen for å bruke visningsveggen ikke har vært så stor som vi hadde regnet med i begynnelsen av denne hovedoppgaven. Vi har dermed prioritert utviklingen av Qt-versjonen av TriangTutor, framfor den rene OpenGL-versjonen. Store deler av koden i de to applikasjonene er den samme, for eksempel klassene for alle de grafiske objektene. Det er stort sett bare koden som har med menysystemet og brukerinteraksjonen

som er forskjellig. Disse delene av koden ligger hovedsakelig i klassene **ApplicationWindow** og **GLScene**. En sammenlikning av koden til disse to klassene i de to versjonene av TriangTutor, viser at Qt med ”signal and slots” gir mye mer elegant og modulær kode enn GLUT og PLIB med sine callback-funksjoner.

Til vanlig bruk er versjonen som bruker Qt best å bruke, fordi denne har et mer stabilt og mer avansert menysystem. Denne er derfor mer brukervennlig og har dessuten et mer profesjonelt utseende.

Kapittel 8

Oppsummering og konklusjon

I dette siste kapitlet vil vi summere opp arbeidet med TriangTutor. Først vil vi ta et tilbakeblikk på hva vi har gjort, og hvordan resultatet har blitt, før vi helt til slutt skal se på mulige utvidelser av TriangTutor.

8.1 Et oppsummerende tilbakeblikk

Vi har gjennom denne hovedfagsrapporten gått grundig igjennom hvordan vi har laget TriangTutor. Denne hovedfagsoppgaven har vært ganske praktisk rettet, i den forstand at det har vært mye programmering. TriangTutor har etter hvert blitt et ganske stort program. Kildekoden er til sammen på nesten 10.000 linjer. Når kildekoden blir såpass stor er det viktig å dele den opp så mye som mulig, for ikke å miste oversikten. Det hjelper også at mange av klassene er bygget opp på samme måte. Kildekoden til TriangTutor har underveis gjennomgått tre store omstruktureringer, etter hvert som den har vokst. Viktig har det også vært at koden er godt kommentert, ikke minst med tanke på videre utvikling. Deler av kildekoden til TriangTutor er allerede i bruk som basis for en ny hovedfagsoppgave ved Simula-senteret.

En tidligere versjon av TriangTutor ble brukt i undervisningen i INF-MAT5370 høsten 2003. Programmet brukes også i INF-MAT5370 nå i høst (2004). Det er også meningen at en mindre versjon av TriangTutor skal følge med i distribusjonen av TTL, som en demoapplikasjon. Denne versjonen vil bare inneholde funksjonalitet som er tett knyttet opp mot TTL, det vil si den funksjonaliteten vi beskrev i Kapittel 5.

TriangTutor har også blitt et brukbart verktøy for å lage figurer og

illustrasjoner. For eksempel er de fleste av figurene i denne hovedfagsrapporten laget med TriangTutor, som vi var inne på tidligere.

8.2 Robusthet og systemkrav

Robusthet er viktig for at et program skal kunne brukes i en seriøs sammenheng. Vi har derfor forsøkt å luke ut alle feilene i TriangTutor som vi har oppdaget underveis. Når det gjelder programmer med stor grad av brukerinteraksjon, så er dette en stor utfordring. Programmet må lages slik at det så godt som mulig fanger opp alle mulige situasjoner som brukeren kan tenkes å komme i, og unngå at det oppstår feilsituasjoner. Et program som krasjer i tide og utide er ikke særlig brukervennlig. Den eneste feilsituasjonen vi i dag vet om i den nyeste versjonen av TriangTutor, er hvis brukeren prøver å laste inn en datafil som ikke har riktig format.

Den funksjonaliteten i TriangTutor som er mest ressurskrevende, og som dermed stiller størst krav til systemet programmet kjøres på, er å flytte på noder. Det må da hele tiden gjøres beregninger for å oppdatere Delaunay-trianguleringen. Hvis brukeren i tillegg har valgt å vise Voronoi-diagram og omskrivende sirkler, blir det ganske mange beregninger som må utføres. Første gang vi prøvde ut dette, var vi ikke sikre på om dette i hele tatt ville fungere i sanntid på en standard PC. Det viste seg at dette fungerte bra på vanlige datamaskiner, som ikke er eldre enn fire til fem år gamle. Til den bruken TriangTutor er beregnet for, opererer brukeren sjelden med trianguleringer som inneholder mer enn noen få titalls noder. For å ha fullt utbytte av programmet anbefaler vi derfor at brukeren har en PC med en CPU på 1 GHz eller mer, samt et grafikkort som har støtte for antialiasing. Det kan nevnes at de raskeste datamaskinene for hjemmemarkedet i dag, klarer å håndtere flytting av noder med visning av Voronoi-diagram og omskivende sirkler i trianguleringer med opp mot 5.000 noder.

8.3 Videre arbeid

Vi kan jo si at et dataprogram egentlig aldri blir ferdig, men etter hvert så er det bare ingen som oppdaterer det lenger. Under arbeidet med TriangTutor har vi også samlet opp en del ideer om videre utvidelser som kan være

aktuelle. Vi vil nå se på de viktigste av disse.

Det kunne vært nyttig å implementere en *skripttolker* i TriangTutor. Brukeren ville da hatt muligheten til å laste inn skriptfiler som innholdt kommandoer som TriangTutor kunne utføre. Dette ville da fungert på samme måte som en animasjon. Vi kunne også implementert muligheten for å generere en skriptfil under kjøring av programmet. Denne filen kunne så blitt lastet inn senere, og TriangTutor ville da utført de samme operasjonene som brukeren gjorde når skriptfilen ble generert. For å få til alt dette måtte vi ha definert et enkelt skriptspråk, som inneholdt all funksjonaliteten i TriangTutor.

En annen utvidelse vi kunne hatt lyst til å implementere, er støtte for å assosiere høydeverdier til alle nodene. Brukeren kunne da også valgt å se på trianguleringene i 3-dimensjoner. Editeringen av trianguleringene kunne fortsatt foregått i planet, men vi kunne for eksempel hatt et ekstra vindu hvor trianguleringene ble vist i 3D. Høydeverdiene kunne vi fått fra en funksjon, eller vi kunne interpolert disse ut i fra et grid i planet med tilhørende z-verdier. En slik utvidelse ville ikke minst vært nyttig i forhold til den delen av pensumet i INF-MAT5370 som tar for seg dataavhengige trianguleringer (Kapittel 5 i [Hje03]).

I forhold til pensumet i INF-MAT5370, ville det også vært nyttig å implementere funksjonalitet for å visualisere teorien i Kapittel 7 i [Hje03]. Dette kapitlet handler om forfining av trianguleringer eller *gridding*, som brukes i elementmetoden for løsing av partielle differensiallikninger (FEM). I dette kapitlet defineres det en del operasjoner og algoritmer, som vi kunne implementert i TriangTutor, kanskje i form av en egen ”gridding-modus”.

Videre hadde det også vært greit å ha implementert den alternative måten å starte oppbygningen av en inkrementell Delaunay-triangulering på, som vi nevnte i seksjon 5.3. Det vil si at vi i stedet for å starte med et rektangel delt inn i to trekanter, starter med en gigantisk trekant som vi ikke tegner opp.

Figurene vi kan lage med TriangTutor blir lagret som bildefiler som er en direkte kopi av framebufferet. Slike bildefiler tar ganske stor plass. Siden figurene vi får fra TriangTutor stor sett består av punkter, streker og sirkler og ikke noe spesielt avansert datagrafikk, hadde det også vært nyttig å kunne lagre disse som vektorgrafikk. For å få til dette, måte vi ha implementert støtte for å tegne ut all grafikken til fil i form av et vektorgrafikkformat. Aktuelle vektorgrafikkformater kunne vært PostScript eller SVG (Scalable Vector Graphics), som bygger på XML (Extensible Markup Language).

Til slutt kan vi nevne at det hadde vært nyttig om TriangTutor ”husket” innstillingene sine fra gang til gang. Dette kan gjøres ved at TriangTutor

lagrer innstillingene sine til en fil hver gang det avsluttes. Denne filen leses så inn igjen hver gang programmet startes opp.

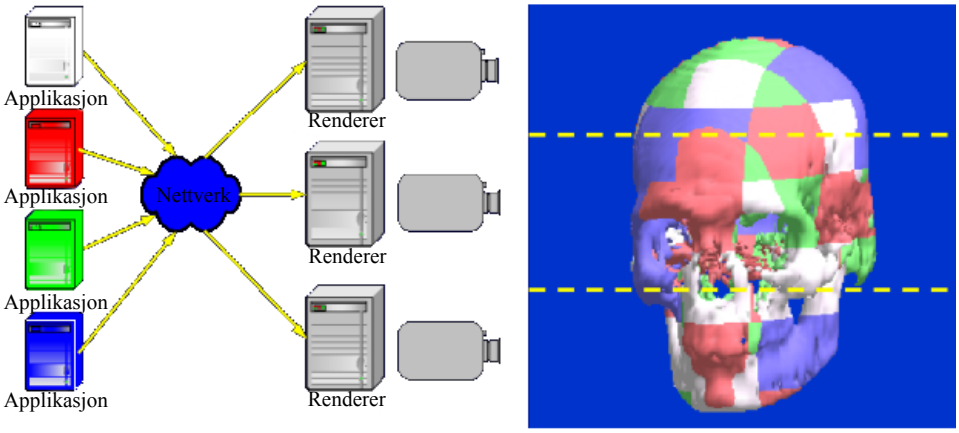
Appendiks A

Visningsveggen og Chromium

A.1 Visningsveggen

På Simula Research Laboratory har vi en visningsvegg som er 3,2 meter bred og 2.4 meter høy. Den har en total oppløsning på 4096 x 3072 piksler. Visningsveggen er basert på et bakprojektorsystem som består av 4 x 4 projektorer. Disse 16 projektorene er koblet opp mot hver sin datamaskin

Figur A.1: Visningsveggen på Simula Research Laboratory.



Figur A.2: Eksempel på bruk av Chromium. (G. Humphreys)

(visualiseringsnode), med et nVidia GeForce 3 grafikkort. Datamaskinene er koblet sammen til et Linux-cluster. Et bilde av visningsveggen ser vi på *Figur A.1*.

A.2 Chromium

Chromium er et skalerbart grafikksystem for interaktiv rendering på clustere av datamaskiner, hovedsakelig utviklet ved Universitetet i Stanford. Systemet er plattformuavhengig, og tilbyr et OpenGL-grensesnitt slik at man skal kunne kjøre eksisterende OpenGL-applikasjoner på clustere, uten modifikasjoner.

Chromium benytter standard teknikker for parallell rendering for å minimalisere overflødig overføring av grafiske data, noe som muliggjør interaktiv rendering ved høy oppløsning fordelt over flere skjermer eller projektorer. Detaljene i hvordan dette gjøres er beskrevet på hjemmesiden til Chromium (http://chromium.sourceforge.net) og i [Hum02].

Figur A.2 viser en enkel skisse over hvordan Chromium fungerer når det benyttes flere maskiner i parallell til å generere grafiske data. Disse dataene visualiseres her på en høyoppløsningsskjerm delt inn i ruter, hvor flere maskiner med hvert sitt grafikkort tilknyttet hver sin projektor, har ansvaret for å avbilde hver sin del.

Bibliografi

[Ang00] E. Angel. *Interactive Computer Graphics: A top-down approach with OpenGL*. 2nd edition , Addison Wesley Longman, 2000.

[Ber00] M. de Berg, M. van Kreveld, M. Overmars og O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. 2nd edition, Springer-Verlag, 2000.

[Ber02] Guntram Berti , GrAL – The Grid Algorithms Library, *Proceedings of ICCS 2002*, 2002.

[Ber94] Y. Bertrand og J. F. Dufourd. Algebraic specification of a 3D-modeler based on hypermaps. *Graphical Models and Image Processing*, 56(1):29-60, 1994.

[Bla04] J. Blanchette og M. Summerfield. *C++ GUI Programming with Qt 3*. Prentice Hall, 2004.

[CGA] The CGAL Home Page.
<http://www.cgal.org>

[DYN] DYNAMAP Homepage.
<http://www.sintef.no/static/AM/dynamap/index.html>

[Fol90] J. D. Foley, A. van Dam, S. K. Feiner og J. F. Hughes. *Computer Graphics, Principles and Practice*. 2nd edition, Addison-Wesley, 1990.

[GRA] GrAL - Grid Algorithms Library.
<http://www.math.tu-cottbus.de/~berti/gral>

[Hje00] Ø. Hjelle. A Triangulation Template Library (TTL): Generic Design of Triangulation Software. Teknisk rapport STF42 A00015, SINTEF, 2000.

[Hje03] Ø. Hjelle og M. Dæhlen. Triangulations and Applications. Lecture Notes, University of Oslo, August 2003.

[Hum02] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern and P. D. Kirchner og J. T. Klosowski. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters, *ACM Transactions on Graphics*, 21(3):693-702, 2002.

[Law72] C. L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3:365-372, 1972.

[Leo98] S. J. Leon. *Linear algebra with applications*. 5th edition, Prentice Hall, 1998.

[Lie89] P. Lienhardt. Subdivision of n-dimensional spaces and n-dimensional generalized maps. *5th ACM Symposium on Computational Geometry*, 228-236, Saarbrucken, Germany, 1989.

[MOC] Mocha Homepage.
<http://loki.cs.brown.edu:8080/pages>

[Ope03] OpenGL Architecture Review Board. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*. 4th edition, Addison-Wesley Longman, 2003.

[Ope04] OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4*. 4th edition, Addison-Wesley Longman, 2004.

[PLI] PLIB: A Portable Games Library.
<http://plib.sourceforge.net>

[Pre85] F. P. Preparata og M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[QtW] Trolltech. *Qt 3.3 Whitpaper*, 2004.
<http://www.trolltech.com/pdf/whitepapers/qt33-whitepaper-a4.pdf>

[STL] Standard Template Library Programmer's Guide.
<http://www.sgi.com/tech/stl>

[Str97] B. Stroustrup. *The C++ Programming Language*. 3rd edition, Addison-Wesley Longman, 1997.

[VOR] VoroGlide, interactive Voronoi diagrams. <http://web.informatik.uni-bonn.de/I/GeomLab/VoroGlide/index.html.en>

[Wei85] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21-40, 1985.