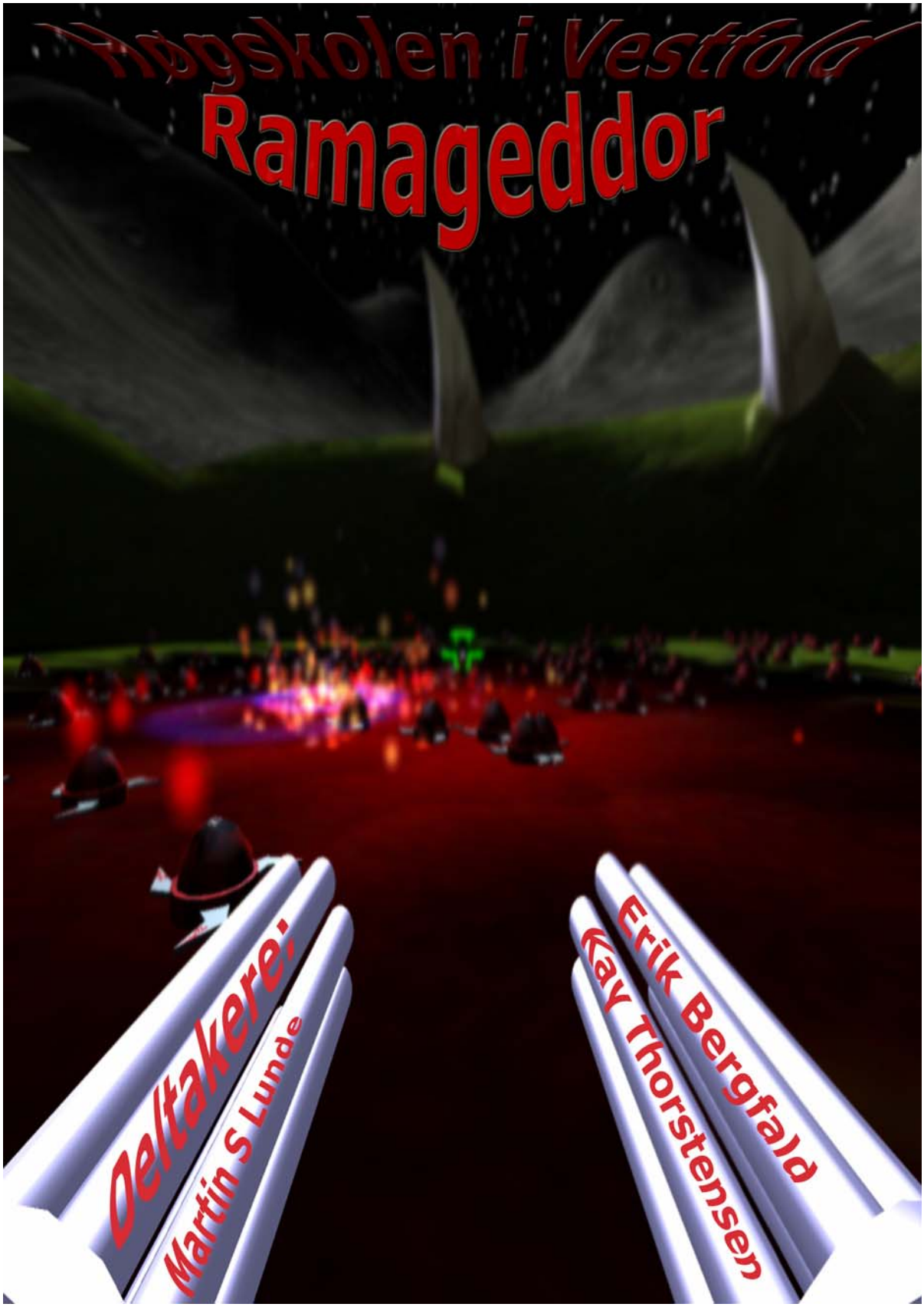

Høgskolen i Vestfold
Ramageddor
Deltakere:
Martin S Lunde
Erik Bergfald
Kay Thorstensen

HIVE INFO

Sammendrag

Hovedprosjektet vårt inneholder et 3d spill med en selvutviklet spillmotor. Vi har også lagt vekt på bruken av genetiske algoritmer når det gjelder fienden sin kunstig intelligens. Dette har vært et tema som vi ville fremme i og med at alle har valgt algoritmer som et valgfag som har gått ved siden av hovedprosjektet.

Målsetting:
Vi satte oss som mål å lage et fungerende 3d spill med en robust spillmotor. Dette innebar en del arbeid med hensyn til programmering og matematisk forståelse. Prosjektet vil kanskje vekke interesse hos andre spill utviklingsentusiaster som har lyst til å gjøre noe liknende. Derfor har vi også lagt ut spillet pluss all kildekode på nett slik at andre kan bygge videre og eventuelt lære en del programmering fra dette prosjektet.

Hensikten med dette prosjektet fra vårt synspunkt er å vise våre kunnskaper innenfor programmering, dette innebærer tankegang og systemutvikling. Hvis dette er bra utført i vårt prosjekt vil dette kanskje interessere eventuelle arbeidsgivere.

Erfaringer og resultat:
Vi sitter igjen med mange erfaringer etter dette prosjektet, spesielt når det gjelder problemløsing innenfor programmering. I tillegg har vi lært en del om samarbeid og prosjektarbeid, og ikke minst det å gi konstruktiv kritikk til hverandre.
Alt under ett synes vi at målsettingen vår er nådd. Spillet har fått mye oppmerksomhet og vi har fått overraskende positiv respons. Vi syns det er spesielt hyggelig at lokalaviser som Tønsberg Blad og Gjengangeren har vist oss interesse.

Beskrivelse av spillet:
Det er to fiender som eksisterer; stormpod og shuriken. Hver motstander har alle forskjellige forutsetninger for å ta deg. Noen er utstyrt med mye liv, og andre er flinkere til å bevege seg bort fra deg for å holde seg i live.
Hvis motstanderne skulle bli for pågående er det mulig å hoppe vekk og lande et helt annet sted på brettet. Våpenet som blir brukt er en roterende pulsrifle, som kan minne om en gattling gun. Når du har kvittet deg med første pulje med fiender kommer du til neste runde.

Innholdsfortegnelse:

Innledning

Utvikling av 3D spill inneholder mange faser og deleoppgaver. Ved større utviklingsinstitusjoner har de flere hundre ansatte som jobber daglig i flere år under hvert sitt felt. Vi er bare tre studenter som skal utvikle et fungerende 3D spill på et halvt år, og derfor sier det seg selv at omfanget av prosjektet ikke kan komme i nærheten av spill som blir gitt ut i dag. Det skal legges til at vi er ambisiøse og tar sikte på å lage et spill som skal imponere eventuelle arbeidsgivere og andre spillentusiaster. I tillegg vil vi vise frem våre kunnskaper innenfor programmering for senere å ha noe og vise til i en eventuell arbeidssituasjon.

Problemstilling
Vi skal lage et fungerende 3D spill med en robust spillmotor. Dette innebærer lyd, kollisjonsdeteksjon, kunstig intelligens ved bruk av genetisk algoritme og en rask og god grafikk.
Det kreves derfor god forståelse av objektorientert programmering, matematikk, fysikk, optimalisering og 3D modellering. Heldigvis har vi en del kunnskaper om dette fordi de fleste av disse temaene ligger under fagplanen for datateknikklinjen. I tillegg vil vi få bruk for kunnskaper innenfor programmering i C++ og DirectX.

Om spillet
Før vi satt oss for fullt ned med prosjektet, var det spesielt en ting vi ville lage som var nytt innen spill utvikling, som kalles genetiske algoritmer. Selve algoritmen er godt kjent blant utviklere, men å bruke den i spill for å lage smarte fiender har ikke vært ofte brukt.
Kollisjonsdeteksjon er en annen viktig brikke i et robust spill. Fiender skal ikke kunne gå gjennom vegger, kulene skal stoppe i fienden og objekter skal kunne kollidere realistisk.
Vi har valgt å bruke 3D Studio Max til 3D modellering. Det fine med 3dsmax er integriteten med DirectX. Utfordringen med å lage modeller til spill er å holde et lavt antall polygoner og samtidig få grafikken til å se best mulig ut. Vi bruker textures på figurene for å få dem til å virke mer detaljerte.

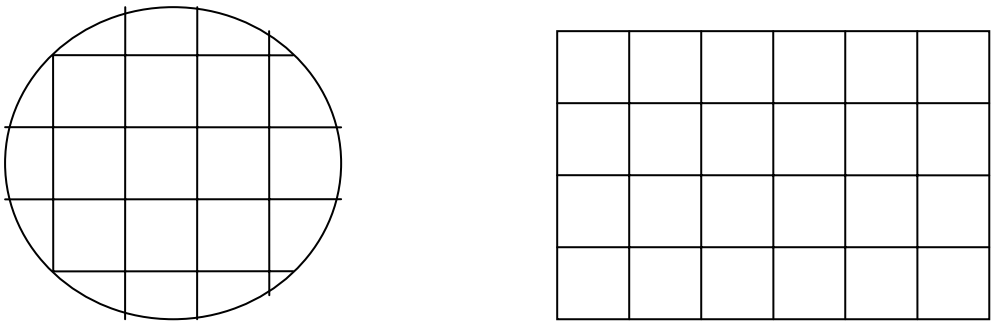
Spillmotorens funksjonalitet:

Generelt om spillmotoren:
En spillmotor i et 3D spill skal tegne selve grafikken, ta seg av blant annet kollisjonsdeteksjon og håndtere kunstig intelligens. Vi har skilt disse elementene i forskjellige klasser fordi de er tre store og uavhengige deler av spillmotoren, og ved å skille dem får man en ryddigere og mer letthåndterlig motor.
I hovedklassene CD3DApplication og CMyD3DApplication (Mental.cpp) ligger spillets "message pump" som går i en evig løkke inntil vi avslutter programmet. Message pumpen tar seg av alle input fra tastaturet, håndterer messager fra operativsystem og kaller FrameMove() og Render().

Message Pump:
while(!quit)
    user input→Framemove→Render osv…

Framemove tar seg av all animasjon mens Render tegner opp all grafikken. En iterasjon i denne løkken tilsvarer et bilde på skjermen, slik at hyppighet av bildeforfriskning kommer an på maskinvaren. Den vil også variere ettersom hvor mange objekter som befinner seg i spillet samtidig, og hvor detaljert grafikken er laget samt hvilken oppløsning man kjører i spillet.

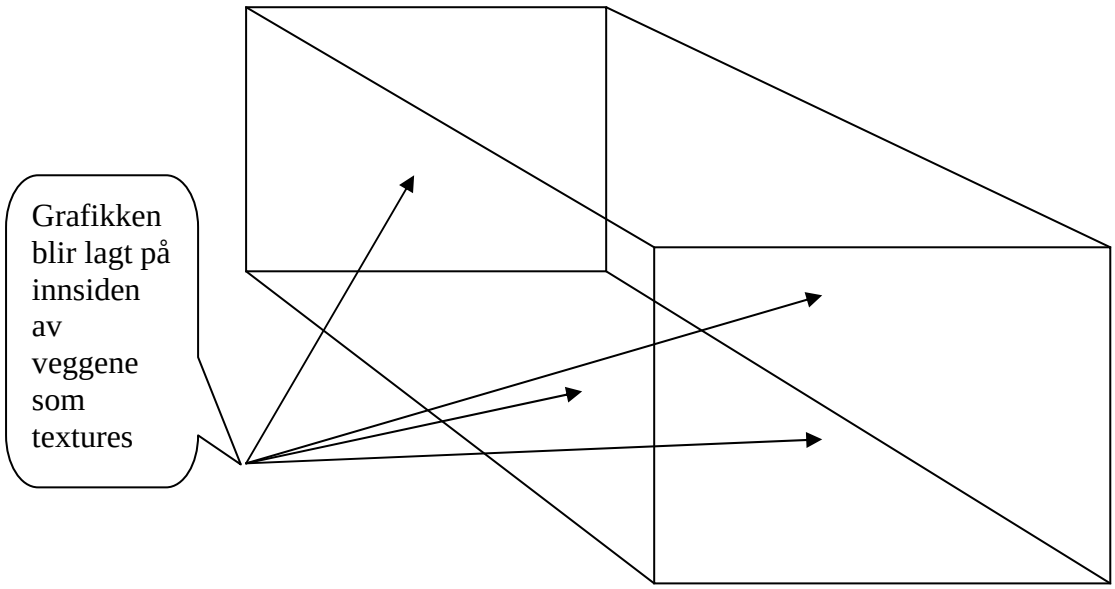
Kollisjons deteksjon/reaksjon:
Denne klassen inneholder koden som tar seg av kollisjons deteksjon/reaksjon. Dette blir gjort ved at hver spillbane blir delt opp i soner. Så blir posisjonen til hvert objekt sjekket slik at vi finner ut i hvilken sone objektet befinner seg, og dette blir lagt til i en tabell som husker hvilke soner som innholder ett eller flere objekter. Hvis en eller flere objekter befinner seg i samme sone vil en kollisjon oppstå. Dette vil føre til en reaksjon, som for eksempel at to objekter bytter hastighet og retning. Illustrasjonene i figur 1.1 viser hvordan spillbanene blir delt opp i soner sett ovenfra.
Kollisjonsdeteksjon blir nærmere forklart i senere i rapporten.



Figur 1.1(Kvadratisk og sirkulær bane delt opp i kollisjons soner sett ovenfra)

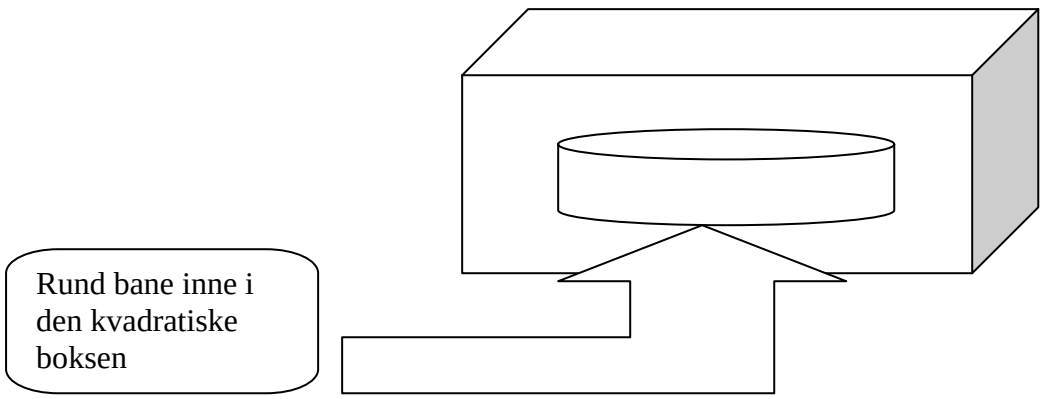
Kunstig intelligens(AI):
Denne klassen innholder all kode som har med fiendens egenskaper å gjøre. Her bestemmes hvor mye skade, liv og hvor stor hastighet hver fiende har pluss en del andre attributter. Dette bestemmes ut fra en Genetisk algoritme vi har laget, slik at fienden skal lære seg de beste egenskapene mot hver enkel spiller. Dette forklares nærmere i kapittelet om AI og genetiske algoritmer.

Grafikk:
Grafikken lages ved hjelp av DirectX SDK(Software Development Kit). Dette er ett grafikkbibliotek utviklet av Microsoft, og det blir brukt av mange ledende spillprodusenter i dag. I tillegg modellerer vi alle 3D-modellene i 3D Studio Max, som vi importerer inn i spillet ved hjelp av DirectX og C++. I vårt spill har vi laget verdener som befinner seg inne i en stor 3D boks (skybox), som vist på illustrasjonen i figur 1.2. På innsiden av denne boksen legger vi grafikk slik at det ser ut som at vi befinner oss som for eksempel i universet.



Figur 1.2 (Skybox)

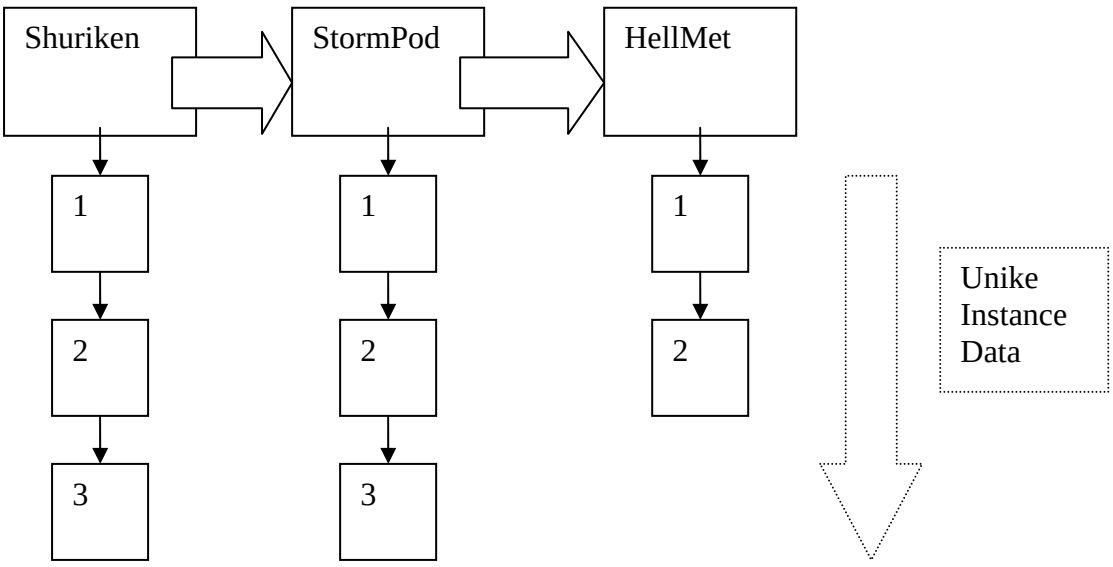
I tillegg lager vi en bane inne i boksen som er enten er rund eller firkantet, dette fordi da blir det en del enklere å holde styr på kollisjons deteksjon og grenser for bevegelighetsområde. Dette vises i figur 1.3.



Figur 1.3

Objekter i spillet:
I spillet oppretter vi mange forskjellige objekter slik som fiender, våpen og spillbaner. Alle disse objektene blir definert i PObject klassen som inneholder all informasjon om egenskapene til de forskjellige objektene. Alle objektene som arver fra PObject er polymorfe klasser. Det vil si at alle subklassenen under PObject kan ha unik kode, men har samme grensesnitt som PObject. Det betyr at vi kan håndtere subklassen som om den var en PObject-klasse, selv om den bruker koden fra subklassen. Vi kan vise et enkelt eksempel fra vårt program:
Vi har en fiende av type Shuriken, når vi skal opprette denne kan vi faktisk gjøre dette slik:
PObject m_pObject = new Shuriken.
Dette fordi Shuriken klassen arver fra PObject.
Hvis vi kjører m_pObject -> Render() vil den selv finne ut hvilken type m_pObject er, og så tegne opp objektet som en Shuriken. Fordelen ved bruk av polymorfe klasser er gjenbruk av kode, og man kan kalle det samme grensesnittet uansett hvilken type objekt det er.

Vi tar bruk av linkede lister når vi oppretter objekter, disse listene strekker seg ut horisontalt for hver unike type objekt, slik som vist på figur 1.4. Hver objekttype har igjen flere unike instances av seg selv der posisjonen, hastigheten og lignende er lagret. Disse dataene er vist vertikalt i figur 1.4.



Figur 1.4 (Første kolonne består av 3 Shuriken, andre kolonne består av 3 StormPod osv..)

Hvis vi nå skal kjøre Render funksjonen vil dette blir slik:
For hver objekttype (Shurken/StormPod) går man gjennom de unike instancene til objektet og tegner disse. Hvis en fiende blir drept, vil den bare forsvinne ut av listen. Dette gjør det også veldig enkelt når vi skal avslutte programmet, vi sletter bare alle gjenværende objekter i denne listen for å unngå minne lekkasje.

Denne strukturerte måten å organisere objektdataene på gjør det veldig enkelt å utføre samme operasjon på alle objektene i spillet, selv om de ikke skulle være av samme type. Spesielt renderingen av objektene er meget effektiv i denne strukturen, men også kollisjonsdeteksjon og kunstig intelligens blir mye lettere å håndtere.
Det er også raskt og effektivt å legge til og fjerne enheter fra spillet i sanntid siden vi kun fjerner eller legger til de dataene som er unike for hver enhet.

Framework og grafikk

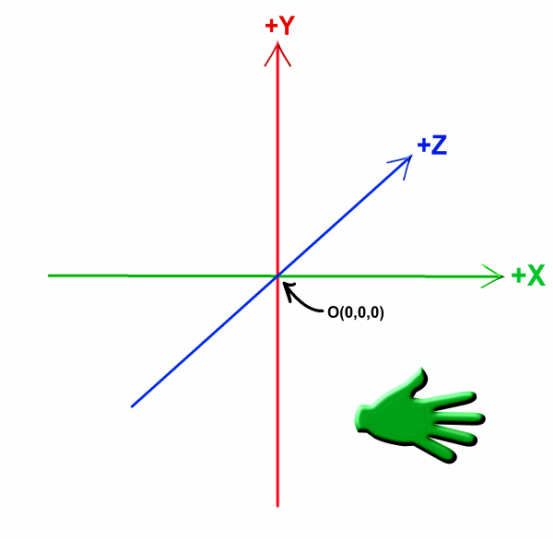
Kjernen i DirectX er HAL (Hardware Abstraction Layer). Det på mange måter en universell driver som legger seg som et skall rundt Pc-utstyret ditt (for eksempel 3dkortet), slik at kort fra forskjellige leverandører får akkurat det samme grensesnittet sett fra deg som programmerer.
Selvfølgelig medfører dette noen begrensninger. Gamle 3dkort støtter for eksempel ikke de siste nyvinningene som Vertex-shadere, men stort sett går det relativt smertefritt. Og hvis man skulle støte på problemer finnes det alltids måter å omgå dem på.

Common Files Framework
Common Files Framework er ett ferdig rammeverk som er til stor hjelp når man skal starte på ett nytt DirectX-prosjekt. Det hjelper til med å initialisere 3Dkortet, og i tillegg sørger det for en ryddig mal for hvordan hovedfunksjonene i spillet skal jobbe sammen.
Klassen CD3Dapplication er ferdig laget av Microsoft, og den håndterer blant annet oppgaver som å finne og sette opp 3dkortet og å kalle hovedfunksjonene i CMyD3Dapplication-klassen som vi jobber ut i fra.

Her er en rask oversikt over funksjonene i CMyD3Dapplication:

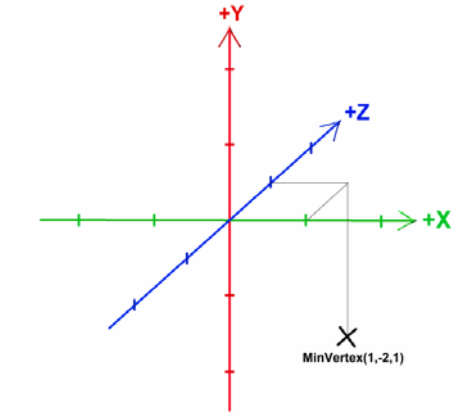
OneTimeSceneInit() kjøres rett etter konstruktøren til klassen, og skal inneholde kode som skal kjøres en gang (når programmet starter). Det er viktig å huske at 3dkortet ennå ikke er funnet, så 3dkort-avhengig kode utgår her.
InitDeviceObjects() blir kalt etter at DirectX har funnet 3dkortet ditt, og her skal all lasting av modeller, teksturer, vertexdata og lignende foregå.
Denne funksjonen kalles også hvis du bytter 3d-device under kjøring av programmet.
RestoreDeviceObjects() kalles rett etter InitDeviceObjects() og når du skifter oppløsning eller bytter til fullskjerm under kjøring. Her ligger kode som setter opp matriser, lys, RenderStates og TextureStates (brukes til å styre 3dkortet).
Etter RestoreDeviceObjects() går programmet inn i en evig løkke kjøres helt til programmet avsluttes, eller til du bytter oppløsning eller 3d-device. I denne løkken kjøres de to funksjonene FrameMove() og Render().
FrameMove() er en meget viktig funksjon som kalles for hver frame (bilde) som renderes. Den inneholder all kode som har med bevegelse og animasjon å gjøre. Den sjekker også hvilke taster du trykker på tastaturet, for du må jo ha mulighet til å styre programmet.
Render() tar selvfølgelig for seg kode som har med tegning å gjøre.
InvalidateDeviceObjects() er paret sammen med RestoreDeviceObjects(), og kalles når du bytter oppløsning. Her rydder du opp før et nytt kall på RestoreDeviceObjects(). Denne funksjonen trenger du sjelden bry deg om.
DeleteDeviceObjects() er derimot meget viktig. Den er paret med InitDeviceObjects(), og skal slette og frigjøre alle ressurser som du lastet der. Hvis du ikke gjør dette vil programmet få minnelekkasjer og krasje maskinen etter hvert. Den kalles når programmet avsluttes, og når du bytter 3d-device.
FinalCleanup() skal fjerne alt du opprettet i OneTimeSceneInit() og den kjøres rett før programmet avsluttes.

Forklaring av vertex og triangel
Direct3D benytter seg av et venstrehånds (lefthanded) koordinatsystem. Det betyr at hvis positiv X er mot høyre og positiv Y er oppover, så er positiv Z inn i skjermen (eller fra deg om du vil). Det kalles venstrehånds fordi det følger venstrehåndsregelen. Hvis du tar venstrehånda og strekker ut fingrene kan du tenke på dem som positiv X (ut mot tuppene), lukker du handa halvveis får du positiv Y, og da vet du retningen på Z ved å se på tommelen.



Slik ser koordinatsystemet ut i Direct3D.

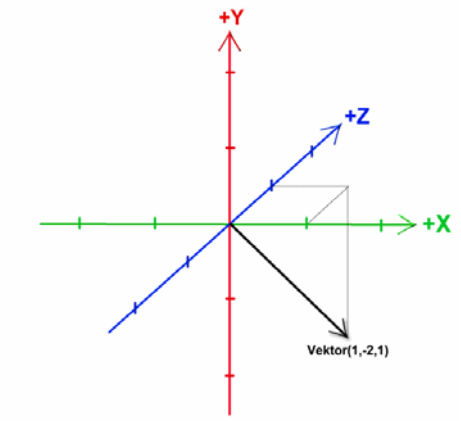
Mange har nok hørt om vertexer og har kanskje fått med seg at det har noe med 3d modeller å gjøre. En vertex er bare et punkt, der posisjonen er definert med tre koordinater (i X, Y og Z-aksen). Ved å bruke flere vertices kan man definere hvordan overflaten på en 3dmodell skal se ut.
En vertex kan se slik ut: D3DXVERTEX3 MinVertex(1,-2,1); Merk deg at alle posisjoner og koordinater skal være kommatall (float) i D3D. For enkelhets skyld vises de som heltall i eksemplene her.



MinVertex ligger i punktet (1,-2,1)

Som regel inneholder en vertex mer enn bare posisjonen. Det er fordelaktig å samle for eksempel informasjon som posisjon, farge, teksturkoordinater og normalvektor sammen i en vertex-struct, noe som vil bli nærmere forklart senere.

En annen datatype det er mye snakk om innen 3D er vektorer. En vektor er like enkel som en vertex, det er bare en pil som viser en retning i rommet. Som regel kan man tenke at denne pilen går fra origo til en eller annen posisjon i rommet. Du kan altså bruke en vektor til å definere posisjonen til MinVertex i eksempelet over.
En vektor er ikke noe du vil tegne på skjermen eller se fysisk på noen måte, det er bare et hjelpemiddel for å vise posisjoner eller retninger i 3drommet. De er også utrolig effektive i fysikkberegning som akselerasjon, kollisjon og bevegelse i et 3Dspill.



Vektoren peker til punktet (1,-2,1)

Man kan dessverre ikke lage særlig imponerende grafikk ved kun å benytte seg av punkter. Dette bringer oss over til triangler. Et triangel er en flate (trekant) bygd opp av tre vertices. Grunnen til at det akkurat er tre er at dette er det minste antall punkter man må vite for å kunne definere en flate. Hver vertex danner da et hjørne i trekanten. Man kan spørre seg hva man skal bruke dette til, men faktisk så er nesten alle 3D-objekter bare et sett med triangler som til sammen danner overflaten. Dette kommer kanskje som et sjokk på noen, men punkter, linjer og triangler er det eneste et 3dkort kan tegne.
Dagens spill benytter teksturer for å få overflaten til å se mer detaljert ut enn den egentlig er. En tekstur er som oftest bare et bilde som er lagt som en tapet på overflaten. Dette kalles 'tekstur mapping'.

Matriser
Matriser er et hendig verktøy for å flytte på objekter i 3d-rommet. En 3d-matrise er en 4 ganger 4 array, der tallene i arrayen indikerer hvordan forflytningen skal skje.
Du kan utføre tre forskjellige former for transformeringer ved hjelp av en matrise: Translasjon (lineær forflytning), rotasjon og skalering.
Det ligger en god del avansert matematikk bak hvordan en matrise egentlig fungerer, og teorien går langt utover det vi rekker i denne rapporten. Den gode nyheten er at man ikke trenger å forstå matematikken, bare hvordan man kan bruke den.

Du oppretter en matrise slik:
```
D3DXMATRIX matTranslate;
```

Følgende funksjon lager en translasjonsmatrise ut fra XYZ-koordinatene du gir inn. Den ferdige matrisen blir lagt i matTranslate.
D3DXMatrixTranslation(&matTranslate, 1.0f, 0.0f, 0.0f);

Her opprettes en rotasjonsmatrise, der matrisen vil rotere objektet ditt 180 grader rundt Z-aksen.
D3DXMATRIX matRotateZ;
D3DXMatrixRotationZ(&matRotateZ, 3.14f);
Merk at i DirectX er alle grader definert i radianer, der π (3.14) er lik 180 grader, π/2 er lik 90 grader, 2*π er lik 360 grader osv..

Det finnes også en skaleringsfunksjon. Her fordobles størrelsen på objektet i alle retninger:
D3DXMATRIX matScale;
D3DXMatrixScaling(&matScale, 2.0f, 2.0f, 2.0f);

For å animere modellene i spillet benytter vi variabelen m_fTime som vi arver fra CD3Dapplication. Den inneholder en verdi som tilsvarer tiden siden man startet programmet (i sekunder). Du kan for eksempel benytte den som innparameter i D3DXMatrixRotationZ slik: D3DXMatrixRotationZ(&matRotateZ, m_fTime);

Man får ikke noe glede av matrisen din før man forteller 3dkortet at det skal bruke den. Hvis vi ikke angir noen transformasjonsmatrise antar 3dkortet at den skal tegne objektet i origo. Sier vi derimot at det skal benytte seg av matrisen matTranslate som vi opprettet over, så vil objektet flyttes til punktet (1.0, 0.0, 0.0) før det tegnes. Dette kalles å sette WORLD-matrisen. Det vil si at de transformasjonene som ligger i WORLD-matrisen vil utføres på alle objekter som renderes, inntil du setter WORLD-matrisen på nytt.
WORLD-matrisen settes slik rett før du renderer objektet:
m_pd3dDevice->SetTransform(D3DTS_WORLD, &matTranslate);

m_pd3dDevice er en peker til 3dkortet, eller rettere sagt HAL som ligger rundt 3dkortet ditt. Klassen den peker på inneholder et vell av kjekke funksjoner som du trenger for å laste data inn i minnet, og for å rendere dem ut på skjermen igjen.

Vi bruker forskjellige matriser for hvert objekt for å flytte dem uavhengig av hverandre. Og det er bare å sette riktig matrise ved å kalle SetTransform() før vi tegner hvert objekt.

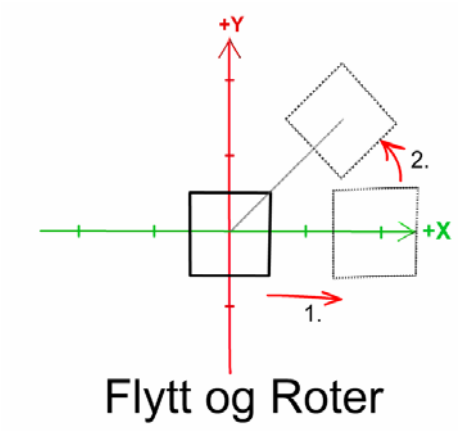
På samme måte kan vi manipulere plasseringen til kameraet (observatøren). Dette kalles å sette VIEW-matrisen. Her roteres kameraet rundt Z-aksen:
m_pd3dDevice->SetTransform(D3DTS_VIEW, &matRotateZ);

Noe av det vakreste med matriser er at vi kan legge sammen resultatet av to matriser ved å gange dem sammen. Hvis vi for eksempel både skal flytte og rotere et objekt

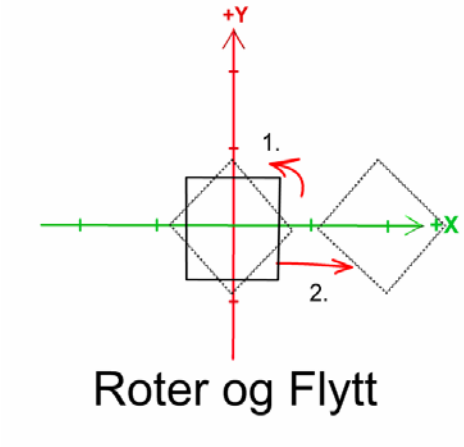
samtidig, så er det bare å gange sammen rotasjonsmatrisen og translasjonsmatrisen, og bruke resultatet som WORLD-matrise.

```
D3DXMATRIX matTransform;
D3DXMatrixMultiply(&matTransform, &matRotateZ, &matTranslate);
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matTransform);
```

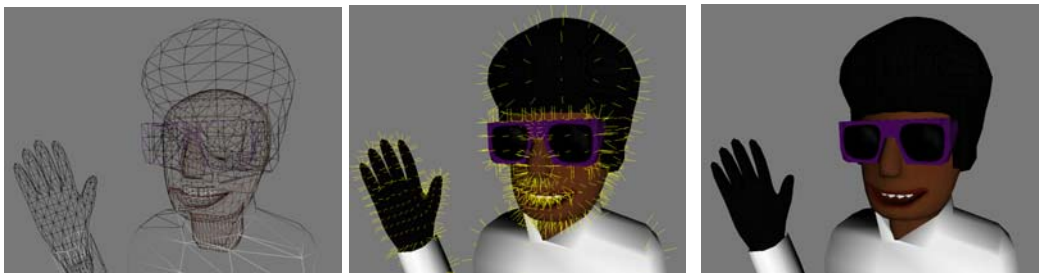
Det er veldig viktig å merke seg at matriser er IKKE kommutative. Det betyr at 'matTranslate x matRotateZ' IKKE er lik 'matRotateZ x matTranslate'. Transformasjonene virker fra venstre mot høyre i multipliseringsfunksjonen. Dvs at den først vil rotere, deretter flytte. Resultatet legger den i matTransform.
3dkortet roterer hele tiden rundt origo i koordinatsystemet, så hvis man flytter objektet før du roterer det vil det derfor ikke roteres rundt sin egen akse, men rundt origo.



Multipliseringsrekkefølgen kan spille selv den beste et puss



Avansert geometri med Mesh og X-files
Selv om manipulering av enkeltvertexer er en god metode til å lære grunnprinsippene på er det ingen seriøse spillutviklere som jobber på den måten. Grunnen er selvfølgelig at det er svært tidkrevende og vanskelig.
Hvis man skal benytte deg av mer avanserte objekter er det mye letter å modellere dem i et 3D program, og senere laste dem inn i programmet. DirectX benytter seg av et format som heter ”X-file” (har lite til felles med serien), som inneholder alle vertex-, normal- og tekstur-data som trengs.
I spillet vårt har vi for eksempel laget alle modellene i 3DStudio MAX, og benyttet en exporter for å gjøre dem om til X-fil.



Her har du et eksempel på et mesh-objekt. På bildet til venstre ser du hvordan det er bygd opp av triangler, og det midterste bildet viser hvordan normalene står ut fra overflaten.

Alle data som lastes inn fra X-filen lagres i en datatype som heter ID3DXMesh. Den deler objektet opp i mindre undergrupper, kalt ”subset”, for hver tekstur/materiale objektet har. Dette gjør den fordi man må kunne skifte tekstur og materiale med SetTexture()-funksjonen for hver gang. Hvis man ikke gjør dette får nemlig alle delene av modellen din samme tekstur.
I Render() vil det se slik ut:

```
m_pd3dDevice->SetMaterial( &m_pMaterial0 );
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pMesh->DrawSubset( 0 );
m_pd3dDevice->SetMaterial( &m_pMaterial1 );
m_pd3dDevice->SetTexture( 0, m_pTexture1 );
m_pMesh->DrawSubset( 1 );
```

For å gjøre koden mer fleksibel legges som regel teksturene og materialene inn i en array slik at vi kan bytte ut koden over med en for-løkke:

```
for(DWORD i=0; i < m_dwAntallSubset; i++)
{
        // Setter materiale og tekstur til subset’et.
        m_pd3dDevice->SetMaterial( & m_pMaterial[i] );
        m_pd3dDevice->SetTexture( 0, m_pTexture[i] );
        // Tegner subset’et.
        m_pMesh->DrawSubset(i);
}
```

Under vises ett enkelt eksempel på hvordan en modell kan lastes inn og tegnes på skjermen.

Defineres i **CD3DMyApplication()-klassen:**

```
//Oppretter en peker til Mesh-objektet.
LPD3DXMESH                  m_pMesh;
// Materialene til Mesh'en.
D3DMATERIAL9*               m_pMeshMaterials;
// Teksturene til Mesh'en.
LPDIRECT3DTEXTURE9*         m_pMeshTextures;
// Her lagres antall teksturer/materialer/subset.
DWORD                       m_dwNumMeshMaterials;
```

"Mesh.cpp":

CD3DMyApplication()-konstruktøren:

```
//Initialiserer alle pekere og variabler.
m_pMesh                 = NULL;
m_pMeshMaterials        = NULL;
m_pMeshTextures         = NULL;
m_dwNumMeshMaterials    =0;
```

// Her laster vi modellen inn fra filen
InitDeviceObjects():

```
// Et midlertidig lager for materialinformasjon
LPD3DXBUFFER pD3DXMtrlBuffer;
pD3DXMtrlBuffer = NULL;

// Her lastes filen "pcpro.x" inn i m_pMesh. Samtidig lagres data om materialer og
// teksturnavn i m_pD3DXMtrlBuffer
D3DXLoadMeshFromX( "pcpro.x", D3DXMESH_MANAGED, m_pd3dDevice,
NULL, &pD3DXMtrlBuffer,NULL, &m_dwNumMeshMaterials, &m_pMesh );


// Teksturnavnene og materialene lagres i d3dxMaterials
D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer-
>GetBufferPointer();

// Her opprettes det en array der materialene og teksturene skal lagres
m_pMeshMaterials = new D3DMATERIAL9[m_dwNumMeshMaterials];
m_pMeshTextures  = new LPDIRECT3DTEXTURE9[m_dwNumMeshMaterials];


// I denne løkken hentes materialene og teksturnavnene
// ut, og etterpå lastes hver tekstur inn i arrayen.
```

```
 for(DWORD i=0; i < m_dwNumMeshMaterials; i = i++)
{
	// Materialet lagres
	m_pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
	// Setter ambient farge lik diffuse farge
	m_pMeshMaterials[i].Ambient = m_pMeshMaterials[i].Diffuse;

 // Her lastes teksturen inn i minnet.
if( d3dxMaterials[i].pTextureFilename != NULL &&
lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
{
D3DXCreateTextureFromFile( m_pd3dDevice, d3dxMaterials[i].pTextureFilename,
&m_pMeshTextures[i] ) ;
}
else
		m_pMeshTextures[i] = NULL;

}
// Vi trenger ikke bufferen lengre, så den slettes.
SAFE_RELEASE(pD3DXMtrlBuffer);
```

I render skal teksturene settes og subset'ene renderes en etter en.
Render():

```
// Går gjennom alle subset'ene, setter tekstur og tegner
for(DWORD i=0; i < m_dwNumMeshMaterials; i++ )
{
		// Setter materiale og tekstur til subset.
		m_pd3dDevice->SetMaterial( &m_pMeshMaterials[i] );
		m_pd3dDevice->SetTexture( 0, m_pMeshTextures[i] );
		// Tegner subset.
		m_pMesh->DrawSubset( i );
}
```

Her fjernes alt vi har lastet inn fra minnet:
DeleteDeviceObjects()

```
// Sletter Mesh'en
if( m_pMesh!= NULL )
{
	SAFE_RELEASE(m_pMesh);
}

// Sletter materialene
if( m_pMeshMaterials != NULL )
{
```

```
        delete[] m_pMeshMaterials;
        m_pMeshMaterials = NULL;
}

// Sletter teksturene
if( m_pMeshTextures )
{
      for( DWORD i = 0; i < m_dwNumMeshMaterials; i++ )
      {
        if( m_pMeshTextures[i] )
        SAFE_RELEASE(m_pMeshTextures[i]);

      }

        delete[] m_pMeshTextures;
        m_pMeshTextures = NULL;
}
```

Kunstig intelligens

Generelt om GA:
Genetiske algoritmer lar oss finne løsninger på svært komplekse problem uten at vi trenger å vite den hvordan vi skal løse den. Det eneste vi trenger å vite er hvordan vi skal vurdere resultatene algoritmen gir oss. Dette gjøres ved å utvikle en fitness funksjon, alle løsingene vil få en fitness som forteller hvor god akkurat den løsningen er etter våre kriterier i fitness funksjonen. Dette fører til at de beste løsningene blir beholdt og deretter paret, slik at neste løsning arver fra de beste løsningene i forrige generasjon.
Til slutt vil vi sannsynligvis oppnå en løsning som er nær optimal, dette kommer selvfølgelig an på omfanget av problemet og hvor mange generasjoner vi lager. Hensikten med GA er jo nettopp å finne den optimale løsningen på et problem og den bruker en ”survival of fittest” metode, hvor de dårligste løsningene blir valgt bort. En ulempe med GA kan være at den bruker ganske mye ressurser.

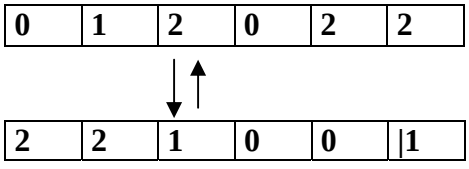
En genetisk algoritmes grunnleggende rekkefølge:

1- Start med en populasjon av kromosomer(løsninger) som får helt tilfeldige verdier
2- Bestem hvilke kromosomer som har best løsning etter kriteriene i fitness funksjonen
3- Velg foreldrene til neste generasjon
4- Par de valgte foreldrene for å produsere avkom til neste generasjon
(Forskjellige metoder)
5- Fjern kromosomene med dårligst fitness
(Forskjellige metoder)
6- Gjenta inntil optimalisert løsning er nådd (stop)

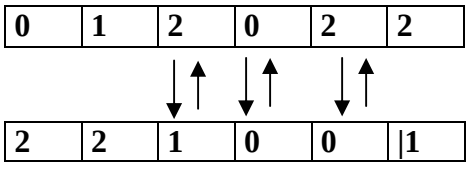
Løsning = kromosom med best fitness

Crossover:
Det finnes metoder for å pare to forskjellige kromosomer, OnePoint Crossover eller TwoPoint CrossOver er to metoder som er utbredt. Disse to metodene bytter henholdsvis en eller to egenskaper, hvilken av metodene som er best kommer helt an på problemet som skal bli løst. I eksempelet under vises hvordan disse metodene blir brukt i praksis.

OnePoint CrossOver:



TwoPoint CrossOver:



Hvis vi har to kromosomer som skal pares, kan vi bruke OnePoint Crossover eller TwoPoint Crossover. I det øverste eksempelet kan vi se at en egenskap blir byttet mellom en av foreldrene og barnet, mens nederst byttes alle mellom to tilfeldig valgte punkter.

Kunstig intelligens eller AI(Artificial Intelligence) er en stor og viktig del av vårt program. Alle fiendene vi lager har sine unike egenskaper, og ved hjelp av genetiske algoritmer utvikler de seg hele tiden for å gjøre mest mulig motstand mot spilleren.

Genetisk algoritme i spillet:
Når vi oppretter en fiende vil denne ha ulike egenskaper. I vårt program kan disse egenskapene være liv, fart og evnen til å påføre spilleren skade. Derfor må vi putte disse egenskapene inn i hver fiende, og dette gjøres ved å opprette en struct av type Genome. Denne innholder de fleste initialegenskapene som hver fiende har. Under kan man se hvordan egenskapene blir satt opp.

```
struct Genome
{
        float fSpawnHP;                 /remember spawnHP
        float fMaxHP;                   //max HP
        float fMaxVel;                  //max Velocity
        float fMaxDamage;               //max Damage

        int iMaxMissonTime;             //max  MissionTime
        float fMaxPredictionTime;       //max PredictionTime
        float fStormFront;              //MissonState(stormfront)
        float fFallBack;                //MissonState(fallback)

        Genome* pNextGenome;
        float fFitness;                 //Fitness of Genome
        float fDamageDone;              //damage done
        float fTimeLived;               //time lived
        int iType;                      //type of enemy
};
```

Arv av egenskaper:
Etter hvert som fiendene blir drept vil deres Genome bli lagt til i en liste som kalles DeadQueue, her vil de bli sortert etter to kriterier: fDamageDone og fTimeLived. Disse to er med på å bestemme hvor bra fienden har klart seg mot oss, og ut fra disse regner vi ut en fFitness. Formelen vi bruker ser slik ut:

```
fFitness = (fDamageDone*12.0f) + (fTimeLived)
```

Vi multipliserer skaden med 12 fordi dette kanskje er det viktigste kriteriet, for hensikten til fienden er jo nettopp å gjøre mest mulig skade på spilleren. Det andre kriteriet er hvor lenge fienden har overlevd, som kommer i annen rekke. Fienden kan jo ha overlevd lenge, men bare holdt seg i bakgrunnen uten å ha gjort noe skade. Disse to kriteriene blir lagt sammen til en sum, og deretter sortert i rekkefølge slik at Genomet med størst fFitness kommer først i den linkede listen. Deretter vil første Genomet og andre Genomet bli paret og avkommet vil ha arvet egenskaper fra de beste foreldrene. Det blir laget to barn som arver fra begge foreldrene, detter blir begge foreldrene og barna lagt til i en ReadyQueue. I denne køen vil de bli plukket ut til neste fiende som blir opprettet, og slik går alt i løkke og fienden blir flinkere og flinkere mot akkurat deg. I kodesnutten under kan man se hvordan to Genome pares.

```
int CppGA::Crossover(Genome* pDeadGenome1, Genome* pDeadGenome2)
{
	Genome* CrossGenome1;
	Genome* CrossGenome2;
	int iRandCrossOver;

	Genome* ChildGenome1 = new Genome;
	Genome* ChildGenome2 = new Genome;
	CrossGenome1 = pDeadGenome1;
	CrossGenome2 = pDeadGenome2;
	memcpy(ChildGenome1,pDeadGenome1,sizeof(Genome));
	memcpy(ChildGenome2,pDeadGenome2,sizeof(Genome));

	iRandCrossOver = rand()%4;
	//random hvilke egenskaper barna får
	if(iRandCrossOver == 0)
	{
		ChildGenome1->fSpawnHP = pDeadGenome2->fSpawnHP;
		ChildGenome1->fMaxDamage = pDeadGenome2->fMaxDamage;
		ChildGenome1->fMaxVel = pDeadGenome2->fMaxVel;
	}
	else if(iRandCrossOver == 1)
		ChildGenome1->iMaxMissonTime = pDeadGenome2->iMaxMissonTime;
	else if(iRandCrossOver == 2)
		ChildGenome1->fMaxPredictionTime = pDeadGenome2->fMaxPredictionTime;
	else
	{
		ChildGenome1->fStormFront = pDeadGenome2->fStormFront;
		ChildGenome1->fFallBack = pDeadGenome2->fFallBack;
	}


	iRandCrossOver = rand()%4;
	//random hvilke egenskaper barna får
	if(iRandCrossOver == 0)
	{
		ChildGenome2->fSpawnHP = pDeadGenome1->fSpawnHP;
		ChildGenome2->fMaxDamage = pDeadGenome1->fMaxDamage;
		ChildGenome2->fMaxVel = pDeadGenome1->fMaxVel;
	}
	else if(iRandCrossOver == 1)
	ChildGenome2->iMaxMissonTime = pDeadGenome1->iMaxMissonTime;
	else if(iRandCrossOver == 2)
	ChildGenome2->fMaxPredictionTime = pDeadGenome1->fMaxPredictionTime;
	else
	{
		ChildGenome2->fStormFront = pDeadGenome1->fStormFront;
		ChildGenome2->fFallBack = pDeadGenome1->fFallBack;
	}
	PrepareAndPushReady(CrossGenome1);
	PrepareAndPushReady(CrossGenome2);
	PrepareAndPushReady(ChildGenome1);
	PrepareAndPushReady(ChildGenome2);
	return(0);
}
```

Crossover funksjon som parer to Genome

I begynnelsen av spillet finnes det ingen Genomes som ligger i ReadyQueue, derfor må vi opprette nye Genomes. Disse får helt tilfeldige verdier fra starten av, og vil som

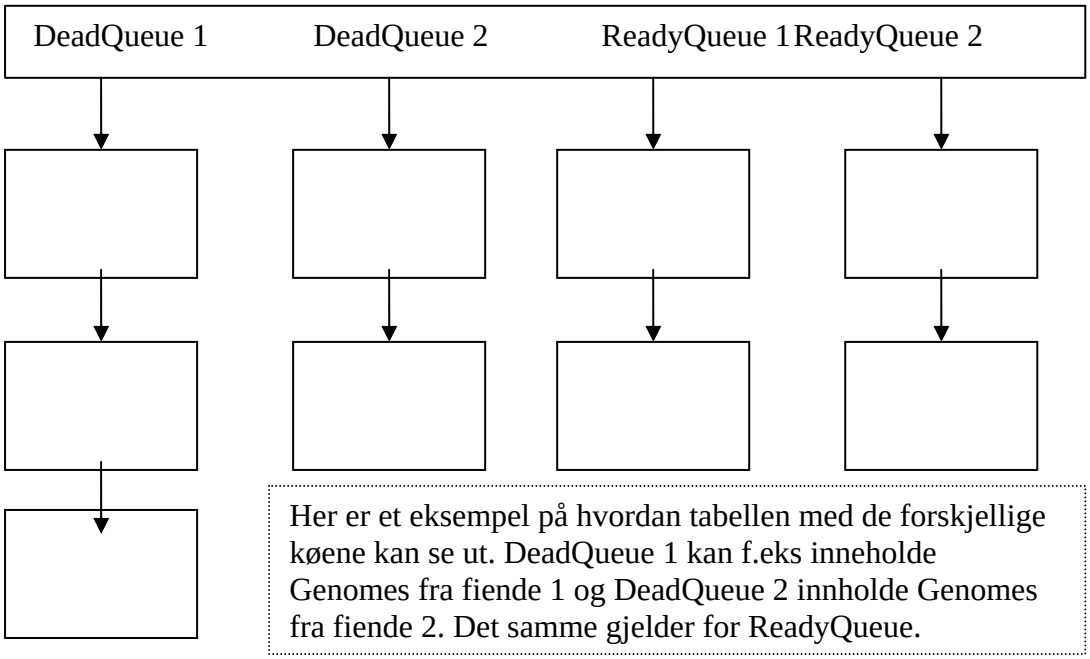
sagt utvikle seg etter hvert. Under kan man se hvordan nye Genomes blir opprettet til typen Shuriken:

```
Genome* CppGA::MakeNewGenome(int iType)
{
	int l_iType;
	l_iType = iType;

	float fMaxAbilities;
	fMaxAbilities = 200.0f;

	Genome* NewGenome = new Genome;
	NewGenome->iType = l_iType;

	//setter semirandom initisal verdier til shuriken sitt gen
	if(l_iType == TYPE_SHURIKEN)
	{
		NewGenome->fMaxDamage = (float)((rand()%10)+5);
		NewGenome->fMaxVel = (float)((rand()%40)+40);

		fMaxAbilities -= (NewGenome->fMaxDamage*8.0f);
		fMaxAbilities -= NewGenome->fMaxVel;
		NewGenome->fMaxHP = fMaxAbilities;
		NewGenome->fSpawnHP = fMaxAbilities;
//		MaxAbilities = 0;

		NewGenome->fMaxPredictionTime = (float)((rand()%7));
		NewGenome->iMaxMissonTime = ((rand()%6)+2);

		NewGenome->fStormFront		= 80.0f;
	//MissonState(stormfront)
		NewGenome->fFallBack		= 20.0f;
	//MissonState(fallback)

		NewGenome->pNextGenome = NULL;
		NewGenome->fTimeLived = 0.0f;
		NewGenome->fFitness = 0.0f;
		NewGenome->fDamageDone = 0.0f;
	}


	return(NewGenome);
}
```

Vi har valgt å la fart, liv og skade være avhengige av hverandre, dette fordi at fienden skal ikke få usannsynlig gunstige verdier på alle egenskapene. fMaxPredictionTime er en egenskap som gir fienden evnen til å vite hvor du vil befinne deg innen en gitt tid. Den vil regne ut hvilken fart og retning du har og antar hvor du vil være om for eksempel 4 sekunder. Denne verdien har vi satt til å variere fra 0 til 6 sekunder, men det er ingen fasit svar på akkurat hvilken verdi som er best, fordi dette kommer igjen an på hvordan spilleren oppfører seg under selve spillet. iMaxMissonTime er hvor lang tid det tar før fienden regner ut en ny posisjon fienden skal gå mot, dette avhenger igjen av spillerens posisjon og fiendens fMaxPredictionTime. fStormFront og fFallBack er to egenskaper som også henger tett sammen, i dette tilfellet er det 80 % sjanse for at fienden skal gå mot spilleren og 20 % sjanse for at han faller tilbake og holder seg i bakgrunnen. Dette blir avgjort hver gang fienden forandrer MissionState som avhenger av iMaxMissionTime.

ReadyQueue og DeadQueue:
I disse to køene finner vi Genomes som enten er døde eller klare for å bli lagt inn i en fiende. Det finnes flere ReadyQueues og DeadQueues ettersom det finnes flere forskjellige Genomes.
En kø kan bestå av Genome fra en type fiende og en annen kø kan inneholde Genome fra en annen type fiende. Disse linkede listene er igjen festet til en tabell, dette illustrerer figur 2.1.

Tabell som innholder linkede lister med forskjellige typer Genomes i ReadyQueue og DeadQueue:



Figur 2.1(Viser ReadyQueues og DeadQueues av forskjellige typer)

Vi har laget mange små funksjoner for å la Genomene komme seg inn og ut av de forskjellige køene. Hver gang en fiende dør vil den etterlate seg et dødt Genome, og dette vil bli lagt til i DeadQueue. Her kan vi se hvordan dette blir gjort i PushDeadGenome:

```
//DeadGenome blir lagt til i pDeadQueue når en unit/fiende dør
int CppGA::PushDeadGenome(Genome* pDeadGenome)
{
	Genome* pAddGenome;
	Genome* pCurrentGenome;
	pAddGenome = pDeadGenome;
	pCurrentGenome = NULL;
	int l_iType;


	//hvis pAddGenome er NULL returneres feil
	if(pAddGenome == NULL)
	{
		AddLog("DeadGenome == NULL Detected");
```

```
                return (-1);
        }

        //ellers blir DeadGenome
        else
        {
                l_iType = pDeadGenome->iType;
                //fitness til DeadGenome blir regnet ut
                FitnessCalc(pAddGenome);
                pCurrentGenome = pDeadQueue[l_iType];
                //hvis head finnes(listen ikke tom)
                if(pCurrentGenome)
                {
                        //legger til sortert etter fitness i DeadQueue, størst Fitness først i rekka
                        while(pCurrentGenome->fFitness > pAddGenome->fFitness)
                        {
                                pCurrentGenome = pCurrentGenome->pNextGenome;
                        }
                        pAddGenome->pNextGenome = pCurrentGenome->pNextGenome;
                        pCurrentGenome->pNextGenome = pAddGenome;


                        while(iDeadCount[l_iType] > 4)
                        {
                                SelectAndCrossTwoGenomes(l_iType);
                                iDeadCount[l_iType] -= 2;

                                AddLog("Two genome breeded");
                        }
                }
                //hvis head ikke finnes(listen tom)
                else
                {
                        pAddGenome->pNextGenome = pDeadQueue[l_iType];
                        pDeadQueue[l_iType] = pAddGenome;
                }
        }
                return(0);
}
```

Deretter blir Genomet paret hvis det har en bra nok fitness, dette er allerede vist i Crossover funksjonen tidligere i kapittelet. Når et Genome er paret og bra nok til å bli lagt til i ReadyQueuen, kjører vi PrepareAndPushReady:

```
int CppGA::PrepareAndPushReady(Genome* pPrepareGenome)
{
        Genome* pReadyGenome;
        pReadyGenome = pPrepareGenome;
        int l_iType;

        if(pReadyGenome == NULL)
        {
                AddLog("pReadyGenome == NULL Detected");
                return (-1);
        }

        else
        {
                //resetter variabler som henger igjen fra forrige populasjon
```

```
                pReadyGenome->fDamageDone = 0.0f;
                pReadyGenome->fTimeLived = 0.0f;
                pReadyGenome->fFitness = 0.0f;
                //setter spawnHP til den nye HP som blir opprettet av GA
                pReadyGenome->fMaxHP = pReadyGenome->fSpawnHP;
                l_iType = pPrepareGenome->iType;
                //legger til i ReadyQueue
                pReadyGenome->pNextGenome = pReadyQueue[l_iType];
                pReadyQueue[l_iType] = pReadyGenome;
        }
        return(0);
}
```

For at Genomene skal bli tatt ut av ReadyQueue og lagt inn i en fiende må vi kjøre PopGenome:

```
Genome* CppGA::PopGenome(int iType)
{
        int l_iType;
        l_iType = iType;
        Genome* pPoppedGenome;
        pPoppedGenome = NULL;

        //hvis ikke noen Genome er klare i ReadyQueue kalles MakeNewGenome
        if(pReadyQueue[l_iType] == NULL)
                return(MakeNewGenome(l_iType));

        else
        {
                AddLog("Popped a crossed genome");

                pPoppedGenome = pReadyQueue[l_iType];
                pReadyQueue[l_iType] = pReadyQueue[l_iType]->pNextGenome;
        }
                return(pPoppedGenome);
}
```

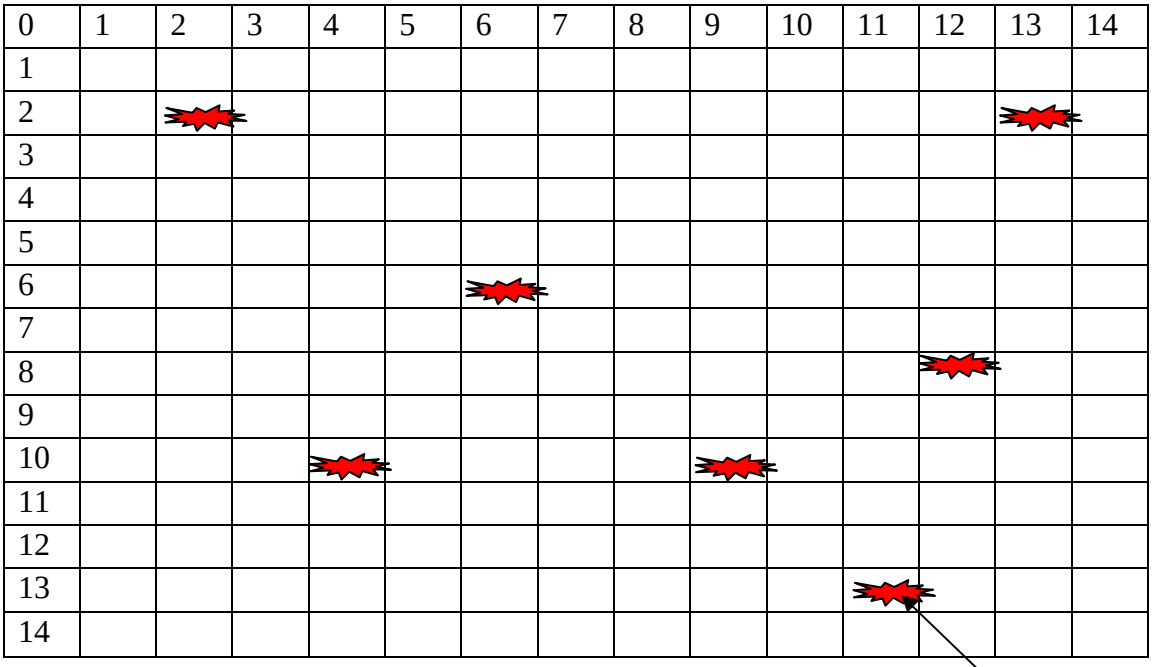
Hver gang vi skal pare to Genomes velger vi alltid de to beste som ligger øverst i DeadQueue, dette fordi vi sorterer etter hvem som har best fitness og legger dem fremst i listen. I SelectAndCrossTwoGenomes kan vi se hvordan Genomene blir valgt:

```
int CppGA::SelectAndCrossTwoGenomes(int iType)
{
        int l_iType;
        l_iType = iType;
        Genome* pFirstGenome;
        Genome* pSecondGenome;

        //velger første og andre Genome i DeadQueue
        pFirstGenome = pDeadQueue[l_iType];
        pSecondGenome = pDeadQueue[l_iType]->pNextGenome;

        //kjører Crossover funksjonen på de valgte Genomene
        Crossover(pFirstGenome,pSecondGenome);
        return(0);
}
```

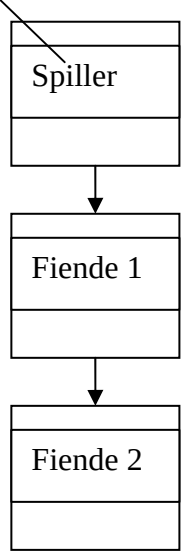
Kollisjonsdeteksjon/reaksjon

Deteksjon:

Vi har valgt å bruke en metode for kollisjons deteksjon som deler opp kollisjonsområdet i flere små soner. Dette gjør det enklere å handtere deteksjon av kollisjon i det aktuelle området, som vi har valgt skal være enten rundt eller firkantet. Vi har valgt å dele opp våre spillbaner i 15*15, som tilsvarer 225 soner. Hvert objekt som befinner seg i programmet har en posisjon, disse posisjonene blir sjekket og lagt til i en todimensjonal tabell. Hvis to eller flere objekter har samme posisjon, det vil si at de befinner seg i samme sone, vil det oppstå en kollisjon. Utefra den todimensjonale tabellen vil det bli lagt til en node for hvert objekt som er lagt til i sonen. Denne metoden for kollisjons deteksjon kalles ”Uniform Grid Collision Detection”. I figur 3.1 kan man se forskjellige soner hvor en kollisjon har oppstått.



Her kan man se et eksempel på at en kollisjon i sone (11,13) har oppstått, spilleren og to fiender befinner seg i samme sone. Spilleren vil alltid bli lagt til først i sonen og deretter alle fiendene i en viss rekkefølge. Dette har noe med hvem som skal gjøre skade på spilleren først og forenkler kollisjonsreaksjonen en del.
Første node vil bli sjekket mot andre node i listen og disse vil få en reaksjon, deretter vil andre node bli sjekket med tredje node og disse vil få sin reaksjon. Første node og tredje node vil ikke bli sjekket mot hverandre, hver node blir bare sjekket mot neste node i listen.
En svakhet med ”Uniform Grid” metoden oppstår hvis ett objekt overlapper to soner, den vil da kunne gå inn i et annet objekt uten at kollisionen blir detektert.



Figur 3.1(Soner hvor kollisjoner oppstår blir lagt til i en todimensjonal tabell)

Reaksjon:
Når alle objektene er blitt lagt til i sin sone og kollisjonstabellen er ferdig oppdatert, skal vi utføre en reaksjon for hver sone hvor en kollisjon har oppstått. Dette kommer mye an på hvilke objekter som kolliderer, her er ett eksempel på hva som skjer ved kollisjoner mellom forskjellige objekter i spillet:

- hvis en StormPod kolliderer med en Shuriken vil ingen skade bli gitt ut, men en Shuriken vil gå 90 grader til siden mens StormPod fortsetter rett frem
- hvis en Shuriken kolliderer med spilleren vil skade bli gitt ut, spiller og Shuriken vil bytte fart og retning, i tillegg kommer det an på hvor stor fart Shuriken har. Hvis den har lav fart og kommer i kontinuerlig kontakt vil livet til spilleren gå gradvis ned, ved stor fart vil ett fast beløp av HP(Health Power) bli trukket fra med en gang
- hvis Shuriken eller StormPod kolliderer med en vegg, vil de sprette tilbake fra veggen i ca et sekund, før den får en ny MissionState(finner en ny posisjon å gå mot, som regel mot spilleren)
- hvis en spiller kolliderer med en vegg, vil man enten sprette litt tilbake eller skli langs veggen
- hvis en StormPod kolliderer med en annen StormPod vil ingen skade bli gitt ut, men en StormPod vil gå 90 grader til siden mens den andre fortsetter rett frem
- hvis en StormPod kolliderer med spilleren, vil spilleren bli presset fremover av StormPod

Dette er noen eksempler på hvordan kollisjonsreaksjonen blir foretatt. Hvordan kollisjonen arter seg kommer som sagt an på egenskapene til den spesifikke fienden.

Kollisjonsdeteksjonen gjøres for hvert nye bilde på skjermen. Det er derfor veldig viktig at ikke det henger igjen noe informasjon i kollisjonstabellen, så alt dette slettes og gjøres klart til neste runde med en Init() funksjon.
Her settes alle sonene til NULL i tabellen slik at ikke et objekt kolliderer med noe som ikke er til stede under spillet.

```
//initaialiserer arrayen
HRESULT CppCollDet::Init()
{
	for(DWORD dwZX= 0; dwZX < NUMCOLROW; dwZX++)
	{
		for(DWORD dwZY= 0; dwZY < NUMCOLROW; dwZY++)
		{
			m_pZoneArray[dwZX][dwZY] = NULL;
		}
	}
	return S_OK;
}
```

Deretter kjøres kollisjonsdeteksjonsfunksjonen DoCollisionDetection og reaksjonsfunksjonen CollReaction, disse er veldig store og omfattende funksjoner som man kan finne i kilde koden under klassen pCollDet. Her kan man hvertfall se grovt hvordan de virker i denne pseudo koden:

```
while(objekt finnes)
{
        finn objektets sone
        sjekk om et annet objekt ligger i samme sone
        if(sant)
                legg til kollisjons sone til i kollisjons tabell
        else
                legg til i todimensjonal tabell

        bytt til neste objekt i listen
}
```

Deteksjon og reaksjon av kuletreff:
Hver fiende har sine har jo sine unike egenskaper, og hvor mye liv dem har kan jo variere en del. Derfor blir noen sprengt i luften fortere enn andre, men hvordan vet vi når en fiende blir truffet og hvor mye liv har den igjen? Deteksjonen av kuler mot fiender avviker ikke mye fra kollisjons deteksjon mellom andre objekter. Hver kule får en fart og retning inntil den treffer et objekt eller en vegg. Hvis den kommer inn i en sone hvor en fiende befinner seg vil et treff oppstå. Kulene vil ikke bli lagt til i sonen på samme måte som objektene gjør, men PBullets klassen vil gå inn å sjekke om det finnes noen fiender i den sonen hvor kulen befinner seg. Hver kule vil da gjøre et antall beløp med skade og dette vil bli trukket fra livet til fienden i sonen. Hvis skaden overstiger det resterende beløp som fienden har igjen av liv, vil den bli drept og gå ut av sone systemet.

Musikk og Lyd

Lyd er et stort og viktig avsnitt innen spilleutvikling. Lyden er en viktig ingrediens for å få morsom spillopplevelse, og i tillegg hjelper det spilleren til å lokalisere fiender.

Vi har benyttet DirectSound-klassen CSound() for å implementere lyd i spillet. Klassen sørger for av effekter som doppler og 3D-lyd blir gjengitt korrekt, og det eneste vi trenger sørge for er å oppdatere posisjonen og hastigheten til lyden. Dopplereffekten simulerer frekvensøkningen som oppstår når en lydkilde beveger seg mot oss, og 3D-funksjonaliteten i DirectX gir spilleren en realistisk følelse av hvor fienden befinner seg.

For hver lyd trenger vi ett sett med ”buffere” som i seg selv representerer ulike lydkilder. Det gjør at vi kan spille av samme lyd på ulike posisjoner samtidig. I tillegg må vi ha en ”listener” som fungerer som ett sett ører i spillerens posisjon.
Disse lydenhetene har en mengde innstillinger der man kan sette forsterkning eller forminskning av lyden i forhold til avstand, dopplerfaktoren, maks avstand og lignende.

Avspilling av lyden gjøres enkelt ved å kalle funksjonen Play() for den gitte lyden. Man kan velge om lyden skal repeteres flere ganger eller om den kun skal spilles en gang.

Vi støtte på noen problemer da vi skulle stoppe et enkelt buffer. Det viste seg at det ikke var mulig uten å stoppe alle bufferne til lyden. Løsningen på dette problemet var å flytte lydkilden så langt vekk at spilleren ikke oppfattet lyden lengre. Når vi fikk behov for lydkilden igjen så var det bare å flytte den tilbake på brettet. Det viste seg at denne løsningen ikke var så ulønnsom som forventet, siden DirectX sluttet å prosessere lyden hvis avstanden til den oversteg en viss lengde.

Når det kom til musikken så valgte vi å benytte DirectMusic i stedet. Dette var fordi musikkfilen er såpass stor i størrelse av det er fordelaktig å streame lyden fortløpende i stedet for å ha alt i lydbufferet.

Slik opprettes en ny lyd:

```
m_pSoundManager->Create( &m_pShurikanSound, "sound\\dronemachine1.wav",
DSBCAPS_CTRL3D, DS3DALG_NO_VIRTUALIZATION, MAX_SHURIKAN_SOUNDBUFFERS
)

 for(iSndBuffIndex = 0; iSndBuffIndex < MAX_SHURIKAN_SOUNDBUFFERS; iSndBuffIndex++)
{

        // Get the 3D buffer from the secondary buffer
        hr = m_pShurikanSound->Get3DBufferInterface( iSndBuffIndex,
        &m_pShurikanDS3DBuffer[iSndBuffIndex] )

        // Get the 3D buffer parameters
        m_dsShurikanPodBufferParams[iSndBuffIndex].dwSize = sizeof(DS3DBUFFER);
        m_pShurikanDS3DBuffer[iSndBuffIndex]->GetAllParameters(
        &m_dsShurikanPodBufferParams[iSndBuffIndex] );

        // Set new 3D buffer parameters
        m_dsShurikanPodBufferParams[iSndBuffIndex].dwMode = DS3DMODE_NORMAL  ;
        m_dsShurikanPodBufferParams[iSndBuffIndex].flMinDistance = 2.1f;
         m_dsShurikanPodBufferParams[iSndBuffIndex].flMaxDistance = 100.0f;

        m_pShurikanDS3DBuffer[iSndBuffIndex]->SetAllParameters(
        &m_dsShurikanPodBufferParams[iSndBuffIndex], DS3D_DEFERRED );

}
```

For å spille av lyden kallen funksjonen Play():

```
m_pShurikanSound->Play();
```

Debugging og testing

Kvalitetssikring:
Alle spill som blir gitt ut i dag blir alltid testet grundig for at minst mulig feil skal komme med på versjonen som blir gitt ut. Vi sier minst mulig fordi det nesten uten unntak alltid oppstår en eller annen feil på et ferdig produkt som er gitt ut. Da blir det som regel lagt ut forskjellige oppdateringer på spillprodusenten sin nettside, slik at brukerne kan laste ned disse.

Vi har fått diverse venner og bekjente til å teste ut vårt produkt under utviklingen, og dette har ført til mange fine forslag til forbedringer. De aller fleste har vært flinke til å gi tilbakemeldinger om feil og ”artifacts”(feil på grafikk) i spillet.

Her er noen betatestere som har deltatt aktivt under testperioden:

Helge Barman:

- fant ut at ”rate of fire ” var tidsavhengig, og problemer med at musa ”lagget” noen ganger
- opplevde å bli sittende fast i veggen
- konsollen fikk programmet til å krasje hvis man skrev veldig mye, funksjonen er nå slått av

Kristian Volden:

- testet spillet på ATI RADEON 9700, funket utmerket
- spillet funket ikke på et skjermkort med 8MB minne, for lite

Håkon Stangeby Lunde:

- fant en del feil med lyden,
- fant feil ved F2 skjermkort innstillingen, tekst plassert feil

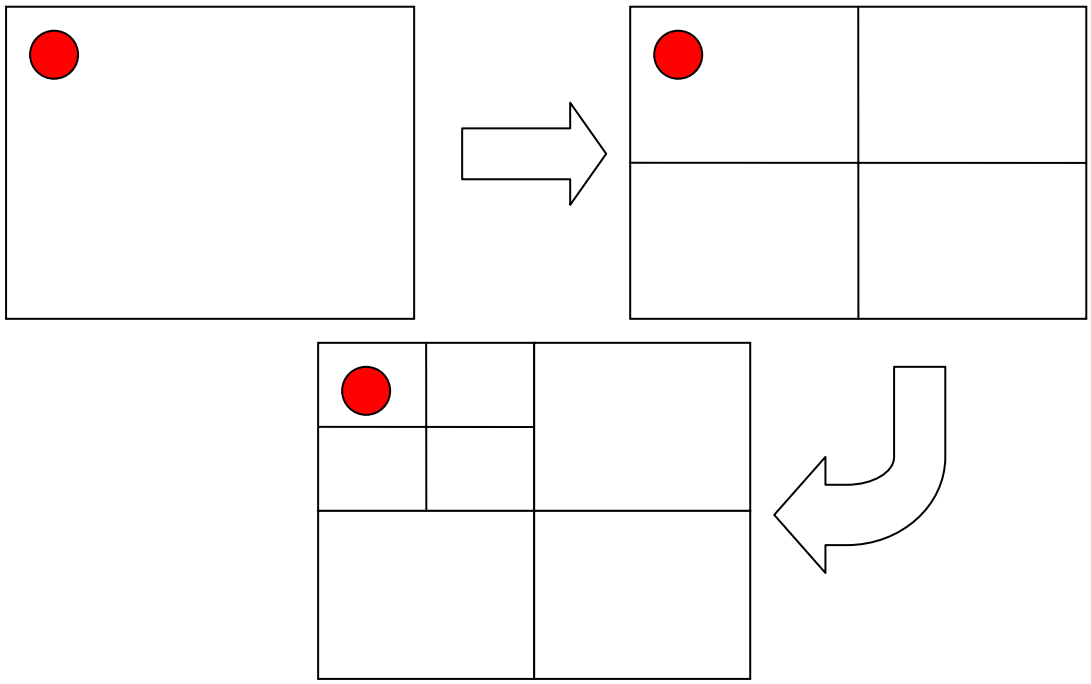
Ellers kjente feil/mangler:

- spillet fungerer meget dårlig på skjermkort uten støtte for textures over 1024x1024

Videre arbeid og utvikling

Kollisjons deteksjon/reaksjon:
Vi har basert vår kollisjons deteksjon på en metode som heter ”Uniform Grid Collision Detection”, som er en ganske enkel metode. Vi kunne tenke oss å oppdatere denne til Octrees som er en dynamisk utgave av vår metode. Octrees optimaliserer kollisjonsdeteksjonen veldig mye, fordi at den deler opp en sone i mindre sone kun hvis det finnes ett objekt der. Dette fører til at vi raskt kan forkaste store mengder søk, og i tillegg blir posisjonen mer nøyaktig. Octrees fungerer meget bra i 3dimensjonale rom, og er optimalisert for dynamiske objekter.

Vi kan dele opp soner helt til vi bare har et objekt i hver sone eller helt ned til et polygon, dette avhenger av hvor nøyaktig kollisjonsdeteksjonen skal være.
Figuren er for enkelhets skyld tegnet i 2D utgave (quadtree).



Figur 5.1: **viser hvordan vi kunne benyttet kollisjondeteksjon**

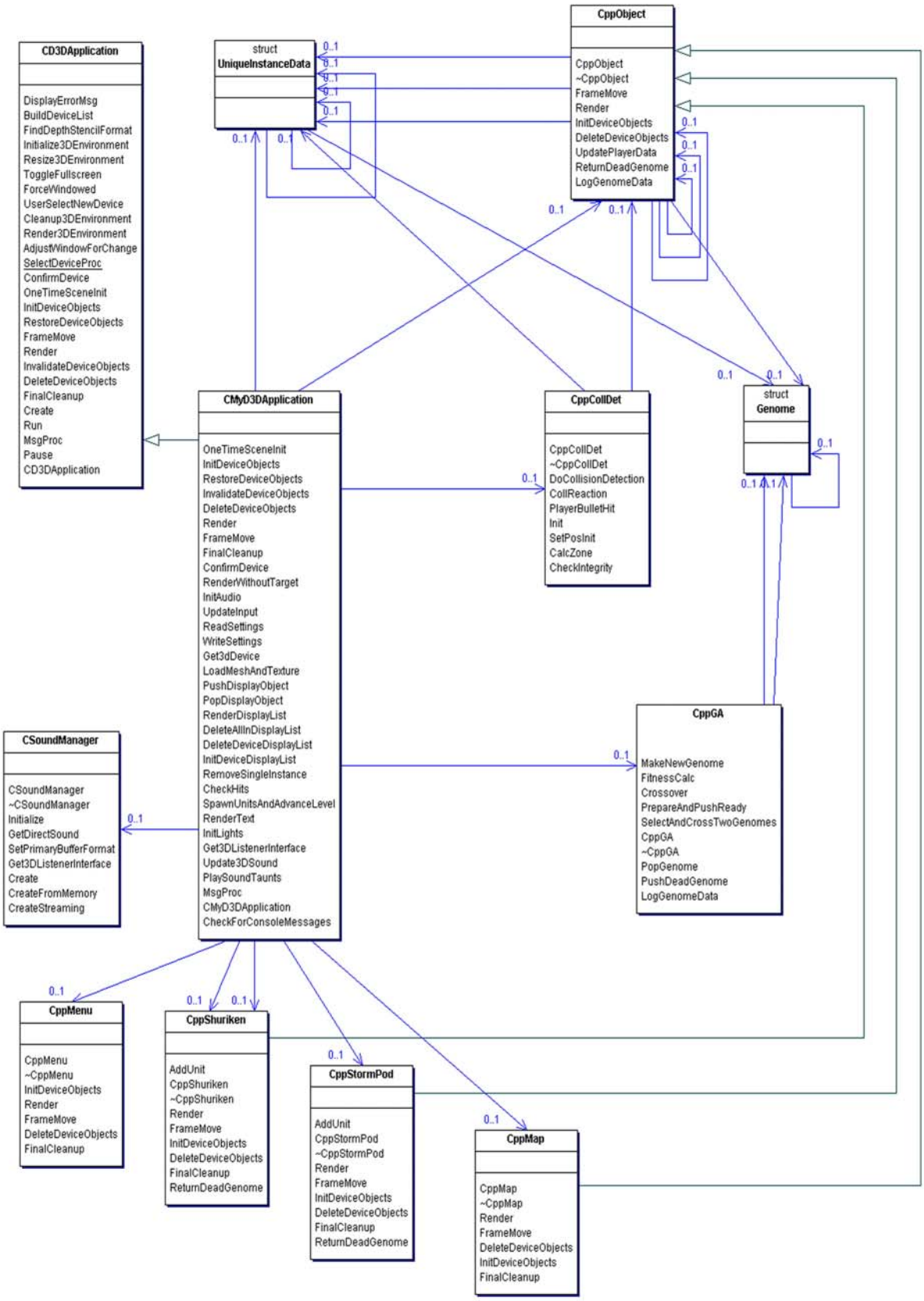
Grafikk:
Dette er kanskje feltet hvor vi ser for oss mest utvikling. Dette fordi vi kan utvikle flere fiender og legge til flere verdener(maps), i tillegg kan vi lage mer detaljert grafikk. Vi kan også optimalisere en god del ved å bruke en metode som heter Binary Space Partitioning (BSP). Dette er en metode som lar vær å tegne opp grafikk som ikke blir sett av spilleren (culling), så dette vil spare prosessoren for veldig mye arbeid. Alt annet som ikke er synlig på bildet under vil ikke bli tegnet eller kalkulert opp.

Figur 5.2: **Bildet viser hvordan spilleren oppfatter det som skjer i et bestemt øyeblikk**

Lyd og musikk:
Dette feltet la vi ikke så mye vekt på, fordi omfanget av prosjektet tvang oss til å prioritere andre felt. Derfor har vi implementert lyden ganske grov og rask måte, og dette gjør det tungvint å legge til nye lyder.

Skulle vi derimot ha utviklet Ramageddor videre, hadde vi måtte lage en smartere måte for å bruke lyden dynamisk. Når det er snakk om mange fiender og hvert enkelt objekt skulle hatt en statisk metode for å bli tildelt lyd, ville det blitt utrolig tungvint og vanskelig å holde orden på alt. En mulig løsning på dette er lydserver. Lydserver er meget effektivt å benytte når mange lyder skal avspilles samtidig og skal fremkomme ofte. Det flotte med det er at vi trenger ikke ta hensyn til når de skal spilles av eller hva slags lyd det er. Enten det er en bakgrunnslyd eller en enkel skytelyd. Det eneste vi trenger å ta hensyn til er å sende lydene til serveren når de trenger avspilles. Da vet serveren straks om det er en oppgave for en buffer eller om vi må benytte en stream til å spille det av. En buffer benyttes når det er lyder av kort varighet. Skal det spilles av større filer som bakgrunnsmusikk bør lyden spilles av ved hjelp av streaming.

UML Diagram


CD3DApplication
DisplayErrorMsg
BuildDeviceList
FindDepthStencilFormat
Initialize3DEnvironment
Resize3DEnvironment
ToggleFullscreen
ForceWindowed
UserSelectNewDevice
Cleanup3DEnvironment
Render3DEnvironment
AdjustWindowForChange
SelectDeviceProc
ConfirmDevice
OneTimeSceneInit
InitDeviceObjects
RestoreDeviceObjects
FrameMove
Render
InvalidateDeviceObjects
DeleteDeviceObjects
FinalCleanup
Create
Run
MsgProc
Pause
CD3DApplication
struct
UniqueInstanceData
CppObject
~CppObject
FrameMove
Render
InitDeviceObjects
DeleteDeviceObjects
UpdatePlayerData
ReturnDeadGenome
LogGenomeData
CMyD3DApplication
OneTimeSceneInit
InitDeviceObjects
RestoreDeviceObjects
InvalidateDeviceObjects
DeleteDeviceObjects
Render
FrameMove
FinalCleanup
ConfirmDevice
RenderWithoutTarget
InitAudio
UpdateInput
ReadSettings
WriteSettings
Get3dDevice
LoadMeshAndTexture
PushDisplayObject
PopDisplayObject
RenderDisplayList
DeleteAllInDisplayList
DeleteDeviceDisplayList
InitDeviceDisplayList
RemoveSingleInstance
CheckHits
SpawnUnitsAndAdvanceLevel
RenderText
InitLights
Get3DListenerInterface
Update3DSound
PlaySoundTaunts
MsgProc
CMyD3DApplication
CheckForConsoleMessages
CppCollDet
~CppCollDet
DoCollisionDetection
CollReaction
PlayerBulletHit
Init
SetPosInit
CalcZone
CheckIntegrity
struct
Genome
CSoundManager
CSoundManager
~CSoundManager
Initialize
GetDirectSound
SetPrimaryBufferFormat
Get3DListenerInterface
Create
CreateFromMemory
CreateStreaming
CppGA
MakeNewGenome
FitnessCalc
Crossover
PrepareAndPushReady
SelectAndCrossTwoGenomes
CppGA
~CppGA
PopGenome
PushDeadGenome
LogGenomeData
CppMenu
CppMenu
~CppMenu
InitDeviceObjects
Render
FrameMove
DeleteDeviceObjects
FinalCleanup
CppShuriken
AddUnit
CppShuriken
~CppShuriken
Render
FrameMove
InitDeviceObjects
DeleteDeviceObjects
FinalCleanup
ReturnDeadGenome
CppStormPod
AddUnit
CppStormPod
~CppStormPod
Render
FrameMove
InitDeviceObjects
DeleteDeviceObjects
FinalCleanup
ReturnDeadGenome
CppMap
CppMap
~CppMap
Render
FrameMove
DeleteDeviceObjects
InitDeviceObjects
FinalCleanup

Konklusjon

I henhold til målsettingen føler vi at prosjektet har innfridd våre forventninger.
Vi har selvfølgelig støtt på en del utfordringer underveis, uten at dette har stoppet arbeidet nevneverdig. En av de store utfordringene har vært å sette seg inn i frameworken til DirectX. Selv om det meste av frameworken er usynlig for programmereren krever det likevel god innsikt for å jobbe mot den.
Vi har også brukt mye tid og ressurser på utvikling av kunstig intelligens og kollisjonsdeteksjon, fordi dette er essensielle temaer innenfor vårt prosjekt.

Gruppearbeidet har ikke budt på noen store problemer, ettersom vi alle kjenner hverandre godt. Vi er alle tre gode kamerater utenom prosjektet og snakker mye sammen for å gi hverandre ideer underveis. Samarbeidsperioden har gitt oss mye forståelse om det å jobbe sammen i en gruppe mot samme mål.
Utfordringen er at vi har hatt alt for lite kunnskap om det å lede, styre og utforme et større prosjekt, så arbeidet har nok gått litt i feil retninger noen ganger. Sett under ett har vi klart oss tilfredstillende.
I næringslivet er prosjektarbeid et veldig sentralt begrep. Vi har heldigvis hatt en god og seriøs veileder, Helge Herheim, til å følge opp vårt arbeid og gitt oss gode råd og vinklinger. Med tanke på den retningen vi har tenkt oss videre i arbeidslivet kommer vi ikke utenom prosjektarbeid, så vi håper og tror vi er bedre rustet mot slike utfordringer etter dette prosjektet.
Prosjektet føler vi har vært delt opp i 3 hovedtemaer; programmering, rapport og web. Programmeringen kunne ikke ha vært gjennomført uten den grunnleggende og allsidige utdanningen vi har fått gjennom de mange utviklingsfagene vi har fått her på høgskolen i Vestfold. Vi tok alle tre optimering og avanserte datastrukturer som valgfag, og dette faget ga oss mye forståelse innen programmering og algoritmer.:
Prosjektrapportering er et felt som er veldig viktig under et slikt prosjekt, men er kanskje ikke like artig.
Web er sentralt i disse dager da alt skal være på nett. Det har vært spennende å jobbe med utviklingen på nett parallelt med resten av prosjektet. Vi har laget weben med tanke på at det skal være lett for oss å oppdatere feil i spillet, i tillegg til at brukerne skal ha en enkel tilgang til nye versjoner. Det går også an til å gi tilbakemeldinger via et forum, og her kan man også motta informasjon om utviklingen.

Litteraturliste

Kilder

www.gamedev.net
www.flipcode.net
AI Techniques for game programming ISBN: 1-931841-08-X
Data Structures for game programming ISBN: 1-931841-94-2
Special Effects for game programming with DirectX ISBN : 1-931841-06-3
Beginning Direct3D Game Programming ISBN: 0761531912
Code Complete ISBN: 1556154844

Vedlegg

Vedlegg 1 …….Utdrag av kildekode

Innhold på vedlagt CD:

Vedlegg 2 …….Ramageddor spill

Vedlegg 3 …….Komplett UML

Vedlegg 4 …….Kopi av rapport

Vedlegg 5 …….Kildekode komplett

Ønskes mer informasjon om oppgaven, henvend deg til Høgskolen i Vestfold, avdeling RI.