Datavetenskap

Mattias Bergström, Magnus Bohman

Generell beskrivning av 3D-motorer med fokus på kärnfunktionalitet

---

Examensarbete

2004:12

Generell beskrivning av 3D-motorer med fokus på kärnfunktionalitet

Mattias Bergström, Magnus Bohman

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Mattias Bergström

Magnus Bohman

Godkänd, 2004-06-03

Handledare: Robin Staxhammar

Examinator: Tim Heyer

Sammanfattning

I takt med att dagens datorer får bättre prestanda och grafiska möjligheter strävar programutvecklare efter att skapa program som kan simulera verkligheten så realistiskt som möjligt. En 3D-motor är en central byggsten i applikationer som skapar realistisk tredimensionell grafik.

Ämnet datavetenskap vid Karlstads universitet funderar på att utveckla en kurs i spelprogrammering där 3D-programmering behandlas. En generell beskrivning av 3D-motorer med fokus på dess kärnfunktionalitet kan bidra till en klarare överblick av 3D-motorer och skulle därför vara användbar i en sådan kurs.

Uppsatsen tar upp frekvent använda och inom området välkända tekniker för att lösa problem relaterade till 3D-motorer. Exempel på det är hur 3D-motorn undviker att bearbeta grafik som inte ska synas genom att använda sig av tekniker som culling och clipping.

Uppsatsen är ett resultat av en undersökning för att hitta kärnfunktionaliteten i 3D-motorer. Resultatet är en introduktion till teorierna bakom 3D-motorer och ger klarhet i vad de gör samt vilka dess viktigaste delar är.

General description of 3D Engines focusing on core functionality

Abstract

While present day computers are getting better performance and graphical possibilities software engineers are striving to create programs that can simulate our reality in the best way possible. A 3D-engine is an important building block of an application in the creation of realistic three dimensional graphics.

The department of computer science at Karlstad University is considering developing a course in game programming, treating 3D-programming. A general description of 3D-engines focusing on core functionality would give a good overview of 3D-engines, and would be usefull in such a course.

The report covers some frequently used and within the area well known techniques to solve problems related to 3D-engines. Examples of this are the various culling and clipping techniques that helps a 3D-engine avoid processing graphics that are not supposed to appear.

This report is the result of an investigation aimed at finding the core functionality of a 3D-engine. The result is an introduction to the theories behind 3D-engines and shows what a 3D-engine does as well as it's most important parts.

Tack

Tack till Robin Staxhammar för hjälp när vi kört fast och för all vägledning under arbetets gång.
Tack till Alex Tjusling för att hon har hjälpt oss att utforma och skriva vår uppsats på ett korrekt sätt.

Innehåll

Figurer

1 Inledning

1.1 Bakgrund

I takt med att dagens datorer får bättre prestanda strävar programutvecklare efter att skapa program som efterliknar vår verklighet på ett så trovärdigt sätt som möjligt. Ett sätt att förbättra representationen av verkligheten är att använda tredimensionell grafik istället för tvådimensionell.

Kravet på representationerna är bakgrunden till varför de flesta nya spel bygger på en slags tredimensionell verklighet. Spel som bygger på tre dimensioner kan representera en virtuell värld som användaren upplever på ett mer verklighetstroget sätt.

En jämförelse mellan två och tredimensionella spel tydliggör skillnaderna mellan spelen. I ett tvådimensionellt spel kan objekt, se avsnitt 5.1, röra sig i sidled och höjdled. I ett tredimensionellt spel kan objekt även röra sig i djupled, precis som objekt i verkligheten. Därför kan tredimensionella spel användas för att bättre representera verkligheten.

Användaren får en bra verklighetskänsla när denne spelar tredimensionella spel som använder sig av ett förstapersons perspektiv.

Ett förstapersonsperspektiv innebär att användaren ser den virtuella världen, den värld som spelet utspelar sig i, på det sätt som han/hon skulle ha upplevt världen med sina egna ögon. Det är till stor del på grund av spel som använder sig av ett förstapersonsperspektiv som 3D-spel har blivit så populära som de är idag. Upplevelsen av spelen blir stor eftersom användaren får en ökad verklighetskänsla.

Det är inte bara spelindustrin som använder sig av 3D-grafik. Det har länge använts med simuleringsprogram och Computer Aided Engineering (CAE). CAE är en samling av verktyg som används för solidmodellering, se avsnitt 3.2.

En 3D-motor är den del hos en 3D-applikationen som sköter den grafiska hanteringen. 3D-motorn bestämmer vilka delar av den virtuella världen som skall ritas upp, samt ritar upp dem på skärmen. För att kunna använda en 3D-motor i en applikation krävs ytterligare funktionalitet, som exempelvis en spelmotor och en fysikmotor, se avsnitt 4.1.1.

I dagsläget finns det inte någon kurs vid Karlstads universitet där 3D-programmering behandlas. Ämnet datavetenskap funderar på att utveckla en kurs i spelprogrammering, där 3D-programmering kommer att ingå.

3D-programmering är ett brett och komplext område. Eftersom det ingår så många olika delar i en 3D-applikation är det svårt att få en överblick över vad 3D-motorn egentligen har för ansvar hos 3D-applikationen. En generell beskrivning av 3D-motorer med fokus på dess kärnfunktionalitet kan bidra till att ge en klarare överblick av 3D-motorer, och skulle därför vara användbar i en kurs som handlar om spelprogrammering.

1.2 Mål

Målet med arbetet är att framställa både en generell och en detaljerad beskrivning av 3D-motorer med fokus på dess kärnfunktionalitet. Beskrivningen är främst tänkt att fungera som ett underlag för ämnet datavetenskap vid Karlstads universitet, inför utvecklandet av en eventuell kurs inom 3D-programmering.

Uppsatsen ska också övergripande presentera en 3D-motors olika delar, samt hur 3D-motorer används i olika applikationer.

Innehållet i uppsatsen skall motiveras med att visa att den kärnfunktionalitet som uppsatsen tar upp och beskriver är väsentlig vid användandet av en 3D-motor.

1.3 Avgränsning

Vi har valt att avgränsa oss till att utröna och beskriva kärnfunktionaliteten hos 3D-motorer. Det ger inte en komplett beskrivning av en 3D-applikation, utan en djupare inblick i 3D-motorer som är en av applikationens delar.

1.4 Metod

Uppsatsen bygger på en undersökning med avseende att finna kärnfunktionaliteten hos 3D-motorer. Undersökningen består av inläsning och litteraturstudier för kunna skapa en helhetsbild över 3D-motorer och för att kunna plocka ut den relevanta informationen till uppsatsen.

1.5 Målgrupp

Uppsatsen är avsedd för läsare med samma baskunskaper som författarna. Det betyder att läsaren bör ha studerat minst två år på en datavetenskaplig universitetsutbildning eller motsvarande. En stor del av den teori som 3D-motorer bygger på består av matris- och vektoralgebra varvid sådan kunskap också är att föredra. För läsare som saknar eller önskar repetera sina algebrakunskaper finns ett kapitel där algebran är beskriven. I bilagorna finns kodexempel med källkod skriven i programspråken C och C++. Det är därför en fördel om läsaren har kunskaper i de programspråken, även om det ändå är möjligt att tillgodogöra sig exemplen utan de kunskaperna.

1.6 Disposition

- **Kapitel 1** beskriver bakgrunden till varför vi skriver uppsatsen, målet med uppsatsen samt det planerade resultatet. Här specificeras även hur uppsatsen är avgränsad, till vilken målgrupp uppsatsen riktar sig samt uppsatsens övergripande struktur.

- **Kapitel 2** beskriver grunderna i den matematik som används i 3D-motorer. Här beskrivs vektor, matrisalgebra, objekttransformeringar, geometri samt geometriska skärningar på den nivå som är nödvändig för att läsaren skall kunna ta till sig informationen som finns i uppsatsen.

  Under avsnittet vektoralgebra tas addition och multiplikation av vektorer upp. Här beskrivs även hur en vektors längd räknas ut samt hur skalärprodukten mellan två vektorer räknas ut. Avsnittet täcker även uträkningen av kryssprodukten mellan två vektorer. Avsnittet matriser beskriver hur matriser adderas och multipliseras. Avsnittet beskriver även transponatet av en matris, enhets- och nollmatris, samt inversmatris.

  Objekttransformationsavsnittet tar upp skalning, rotation samt förflyttning av objekt. Geometriavsnittet beskriver linjer, plan och sfärer. Avsnittet geometriska skärningar beskriver skärningar mellan linje och plan samt mellan linje och sfär.

- **Kapitel 3** beskriver olika användningsområden för en 3D-motor. Kapitlet beskriver olika spel som använder sig av 3D-motorer, övriga användningsområden för en 3D-motor samt innehåller en översikt av olika delar i ett 3D-spel. Översikten beskriver virtuella världar, objekt i 3D-spel, karaktärer samt miljöeffekter.

- **Kapitel 4** ger en översikt av en 3D-motors olika delar. Kapitlet beskriver övergripande de olika delarna som en 3D-motor består av. Här beskrivs först vad en 3D-motor gör och hur en 3D-motor fungerar. Kapitlet presenterar även koncepten culling, clipping,

texturering samt kollisionsdetektering.

- **Kapitel 5** definierar och beskriver olika begrepp och koncept som hör till 3D-motorer, men som inte faller under kärnfunktionaliteten. Informationen i kapitlet anser vi vara nödvändig för att kunna förstå och ta till sig informationen i kapitel 6, 7 och 8. Här beskrivs objekt, koordinatsystem, transformationer, synlighet, gränsvolymer, bounding hierarchial volumes, quadTrees och binaryspace partition.

- **Kapitel 6** Kapitel sex, sju och åtta innehåller den generella beskrivningen av 3D-motorer med fokus på dess kärnfunktionalitet. I detta kapitel tas rendering, omritningsfrekvens, rastrering, sortering, z-buffert samt texturhantering upp.

- **Kapitel 7** behandlar begreppen culling och clipping. Kapitlet tar upp tre olika sätt att använda culling, backface culling, frustum culling samt occlusion culling. Här beskrivs även optimering av culling med hjälp av bounding hierarchial volumes.

- **Kapitel 8** beskriver kollisionsdetektering hos 3D-motorer. Kapitlet beskriver kollisionsdetektering med hjälp av gränsvolymerna sfär och cylinder. Kapitlet tar även upp optimering av kollisionsdetekteringen med hjälp av binaryspace partition.

- **Kapitel 9** beskriver det genererade resultatet. Kapitlet beskriver även rekommendationer för de som vill fördjupa sig i ämnet 3D-motorer.

- **Kapitel 10** innehåller sammanfattningen. Sammanfattningen består av vår syn på arbetet och hur vi tyckte att uppsatsen har gått. Den innehåller även kommentarer om hur vi upplevt ämnet och vad som kunde gjorts annorlunda under examensarbetets gång.

2 Grundläggande matematik för 3D-motorer

Kapitlet presenterar grunderna i matris- och vektoralgebra. Kapitlet är inte primärt tänkt som en introduktion utan mer som en repetition för läsare med redan befintliga algebrakunskaper.

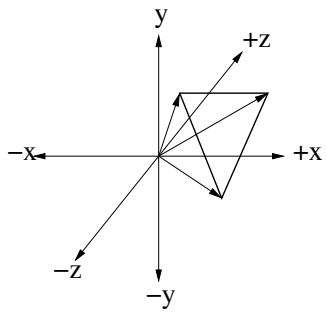
Vidare presenteras några av de grundläggande algoritmer som används inom tredimensionell grafikprogrammering för att bestämma skärningar mellan objekt. Läsaren behöver den kunskapen som presenteras i detta kapitel för att kunna ta till sig delar av den information som presenteras i resten av uppsatsen. En stor del av den funktionalitet som beskrivs i uppsatsen använder sig av matematik som beskrivs i kapitlet. Teorierna i kapitlet är hämtade ur, Lengyl [5], samt, Vretblad [6].

2.1 Vektoralgebra

Vektorer beskriver bland annat positioner och rörelseriktiningar. Vektorer används av 3D-motorer samt inom fysik och mekanik.

En n-dimensionell vektor $\vec{a}$ består av komponenterna $a_1, a_2, \ldots, a_n$. Vanligtvis används vektorer i två och tre dimensioner, ytan och rummet. Vektorns komponenter motsvarar då (x,y)-planet som är ytans koordinataxlar och (x,y,z) som är koordinataxlarna i rummet. En vektor har både storlek och riktning. Riktingen utgörs av vektorns komponenter.

Exempel på en tredimensionell vektor $\vec{a} = (1, 2, 3)$
Det finns en speciell vektor som kallas *ortsvektor*. Ortsvektorn utgår alltid från origo. Den egenskapen gör att ortsvektorns komponenter tolkas som en punkt i koordinatsystemet.



Figur 2.1: Visar hur ortsvektorer kan beskriva en triangel i rummet.

2.1.1 Addition

För att addera två vektorer $\vec{a}$ och $\vec{b}$ adderas varje komponent i vektorerna var och en för sig enligt följande:
$\vec{a} + \vec{b} = (a_1, a_2, a_3) + (b_1, b_2, b_3) = (a_1 + b_1, a_2 + b_2, a_3 + b_3)$
Exempel:
Anta att $\vec{a} = (1, 3, -4), \vec{b} = (5, -5, 0)$. Då blir $\vec{a} + \vec{b} = (6, -2, -4)$

2.1.2 Multiplikation

En vektor $\vec{a}$ kan multipliceras med en skalär c. En skalär är ett godtyckligt reellt tal. Varje komponent i vektorerna multipliceras med c enligt följande:
$c * \vec{a} = (c * a_1, c * a_2, c * a_3)$ Exempel:
Anta att $\vec{a} = (1, 3, -4)$, $c = 10$. Då blir $c * \vec{a} = (10, 30, -40)$

2.1.3 Längd

Längden av en vektor $\vec{a}$ beräknas med hjälp av Pytagaros sats och brukar betecknas med $|\vec{a}|$. Där $|\vec{a}| = \sqrt{a_1^2 + a_2^2 + a_3^2}$

En vektor $\vec{a}$ där $|\vec{a}| = 1$ kallas för enhetsvektor. Exempel:
Anta att $\vec{a} = (1, 2, 3)$. Då är $|\vec{a}| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$

2.1.4 Skalärprodukt

Namnet skalärprodukt kommer ifrån att resultatet är en skalär och inte en vektor. Andra namn på skalärprodukt som förekommer i litteratur är vektorprodukt samt inre produkt. Här används uteslutande ordet skalärprodukt.

Det finns två metoder för att beräkna skalärprodukten av två vektorer $\vec{a}$ och $\vec{b}$. I den första metoden summeras produkterna av komponenterna i vektorerna enligt följande:
$\vec{a} \bullet \vec{b} = a_1 * b_1 + a_2 * b_2 + a_3 * b_3$
Den andra metoden beräknas kryssprodukten som produkten av vektorernas längder och cos för vinkeln mellan vektorerna.
$\vec{a} \bullet \vec{b} = |\vec{a}||\vec{b}| \cos \alpha$ där α är vinkeln mellan vektorerna.

Om $\vec{a}$ och $\vec{b}$ är vinkelräta är skalärprodukten 0. Om vinkeln mellan $\vec{a}$ och $\vec{b}$ är < 90 grader är skalärprodukten < 0. Om vinkeln mellan $\vec{a}$ och $\vec{b}$ är > 90 grader är skalärprodukten > 0.

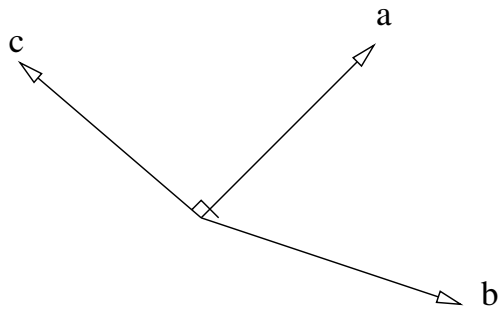
För att beräkna vinkeln α används följande formel: $\alpha = \arccos \frac{\vec{a} \bullet \vec{b}}{|\vec{a}||\vec{b}|}$

Exempel:

Anta att $\vec{a} = (1, 3, -4), \vec{b} = (5, -5, 0)$ då är $\vec{a} \bullet \vec{b} = 1 * 5 + 3 * (-5) + (-4) * 0 = -10$. Detta ger $\alpha = \arccos \frac{-10}{\sqrt{26} * \sqrt{50}} = 1,94$ radianer.

2.1.5 Kryssprodukt

Kryssprodukten av två vektorer $\vec{a} \times \vec{b} = \vec{c}$ ger en vektor $\vec{c}$ som är vinkelrät mot både $\vec{a}$ och $\vec{b}$. Kryssprodukten är bara definierad för vektorer av tre dimensioner.



Figur 2.2: Illustration som visar kryssprodukten av två vektorer.

$$\vec{c} = \vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \vec{i} \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} - \vec{j} \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} + \vec{k} \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} = \vec{i}(a_2 * b_3 - a_3 * b_2) -$$

$\vec{j}(a_1 * b_3 - b_1 * a_3) + \vec{k}(a_1 * b_2 - b_1 * a_2)$ Exempel:

Anta att $\vec{a} = (1, 3, -4), \vec{b} = (5, -5, 0)$. Resultatet blir då $\vec{c} = \vec{a} \times \vec{b} = (20, 20, -20)$

2.2 Matriser

En matris är en samling av element uppdelade i rader och kolumner. En $m \times n$-matris anger en matris med m rader och n kolumner.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

2.2.1 Addition

Addition av två matriser $A + B = C$ är definierad enligt följande.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \dots & a_{mn} + b_{mn} \end{pmatrix}$$

Exempel:

$$\begin{pmatrix} 1 & 3 & 6 \\ 2 & 4 & 7 \\ 3 & 5 & 8 \end{pmatrix} + \begin{pmatrix} 3 & 3 & 3 \\ 2 & 2 & 2 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 4 & 6 & 9 \\ 4 & 6 & 9 \\ 4 & 6 & 9 \end{pmatrix}$$

2.2.2 Multiplikation

För att multiplicera två matriser A och B och A är en $m \times n$-matris, så krävs det att B är en $n \times r$-matris. Resultatet C blir en $m \times r$-matris. Resultatet har samma antal rader som i A och samma antal kolumner som B.

Notera särskilt att $A * B \neq B * A$.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1r} \\ b_{21} & b_{22} & \dots & b_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nr} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} \dots a_{1n} * b_{n1} & a_{11} * b_{12} + a_{12} * b_{22} \dots a_{1n} * b_{n2} & \dots & a_{11} * b_{1r} + a_{12} * b_{2r} \dots a_{1n} * b_{nr} \\ a_{21} * b_{11} + a_{22} * b_{21} \dots a_{2n} * b_{n1} & a_{21} * b_{12} + a_{22} * b_{22} \dots a_{2n} * b_{n2} & \dots & a_{21} * b_{1r} + a_{22} * b_{2r} \dots a_{2n} * b_{nr} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} * b_{11} + a_{m2} + b_{21} \dots a_{mn} * b_{n1} & a_{m1} * b_{12} + a_{m2} + b_{22} \dots a_{mn} * b_{n2} & \dots & a_{m1} * b_{1r} + a_{m2} + b_{2r} \dots a_{mn} * b_{nr} \end{pmatrix}$$

Exempel: En 3×3-matris gånger en 3×2-matris ger en 3×2-matris

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \times \begin{pmatrix} 9 & 6 \\ 8 & 5 \\ 7 & 4 \end{pmatrix} = \begin{pmatrix} 1*9+4*8+7*7 & 1*6+4*5+7*4 \\ 2*9+5*8+8*7 & 2*6+5*5+8*4 \\ 3*9+6*8+9*7 & 3*6+6*5+9*4 \end{pmatrix} = \begin{pmatrix} 90 & 54 \\ 114 & 69 \\ 138 & 84 \end{pmatrix}$$

2.2.3 Transponat

Transponatet A^T av en matris A är samma som att skriva kolumnerna som rader och raderna som kolumner. En $m \times n$ matris blir då en $n \times m$.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}$$ Exempel:

Om $M = \begin{pmatrix} 90 & 54 \\ 114 & 69 \\ 138 & 84 \end{pmatrix}$, då är $M^T = \begin{pmatrix} 90 & 114 & 138 \\ 54 & 69 & 84 \end{pmatrix}$

2.2.4 Enhets- och Nollmatris

En enhetsmatris E är en $n \times n$- matris där alla element i diagonalen är 1 och alla andra element är 0.

$$E = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

I nollmatrisen är alla element i matrisen M 0.

$$M = \begin{pmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}$$

2.2.5 Invers

En $n \times n$ matris M är inverterbar om och endast om det finns en matris M^{-1} sådan att $M * M^{-1} = M^{-1} * M = E$ Exempel:

Om $A = \begin{pmatrix} 2 & -5 \\ -1 & 3 \end{pmatrix}$ är $B = \begin{pmatrix} 3 & 5 \\ 1 & 2 \end{pmatrix}$ en invers till A eftersom $A \times B = E$ och $B \times A = E$. Inversen till en matris beräknas fram med hjälp av radreduktion, ett moment som av utrymmesskäl inte täcks vidare av uppsatsen.

2.3 Objekttransformering

För att ändra storlek, position eller rotation på ett objekt används transformeringar. Transformeringen utförs på alla nodpunkter, se avsnitt 5.1.2, i ett objekt. Eftersom nodpunkterna i objektet är lagrade som ortsvektorer är det möjligt att använda matrismultiplikation för att transformera.

2.3.1 Skalning

Skalningsmatrisen definieras enligt följande: $M = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}$ Där a, b, c är skalningsfaktor för respektive komponent.

$$M * \vec{p} = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix} * \begin{pmatrix} x \\ y \\ x \end{pmatrix} = \begin{pmatrix} ax \\ by \\ cx \end{pmatrix}$$

2.3.2 Rotation

Vi har tre olika rotationsmatriser. De roterar kring varsin koordinataxel.

$$M_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$M_y = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

$$M_z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

För att rotera kring alla axlar används multiplikationen av dessa matriser $M_x \times M_z \times M_z$.

2.3.3 Förflyttning

När ett objekt förflyttas adderas alla nodpunkter i objektet med respektive förflyttningsvektor. Det blir matematiskt som att addera objektets positionsvektor med en förflyttningsvektor.

2.4 Geometri

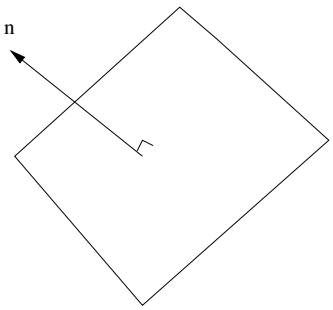
2.4.1 Linjer

En linje i ett tredimensionellt rum ges av följande funktion $P(\vec{t}) = \vec{P_0} + t\vec{V}$. $\vec{P_0}$ är en punkt på linjen, t är en parameter och $\vec{V}$ är en riktningsvektor parallell med linjen.

Exempel: $P(\vec{t}) = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + t * \begin{pmatrix} -1 \\ 4 \\ 5 \end{pmatrix}$

2.4.2 Plan

Ett plan är oändligt bred och oändligt lång, samt är oändligt platt. Ett plan definieras enligt ekvationen $Ax + By + Cx + D = 0$ där $(A, B, C) = \vec{N}$. $\vec{N}$ är planets normalvektor. $D = -\vec{N} \bullet \vec{P_0}$, där $\vec{P_0}$ är en punkt i planet.



Figur 2.3: Ett plan med normalvektor

$d = \frac{|D|}{|\vec{N}|}$ ger det mot planet vinkelräta avståndet till origo. En alternativ definition för ett plan är mängden av alla punkter $\vec{P}$ så att följande uppfylls.$\vec{N} \bullet (\vec{P} - \vec{P}_0) = 0$.

2.4.3 Sfär

En sfär med origo i punkten $\vec{P}_0$ och radie r definieras som mängden av alla punkter P där följande olikhet är uppfylld.$|\vec{P} - \vec{P}_0| \leq r$. Ekvationen för en sfär med medelpunkten origo är $r^2 = x^2 + y^2 + z^2$

2.5 Geometriska skärningar

I tredimensionell grafik är det viktigt att avgöra om vissa geometriska figurer skär varandra och var en eventuell skärningen sker. Skärningar används i 3D-motorer till culling, clipping, se kapitel 7, och kollisionsdetektering, se kapitel 8.

2.5.1 Linje och plan

För att avgöra om och var en linje i rummet skär ett plan används följande samband: Skärningen sker i punkten $\vec{P} = \vec{P_0} - \frac{\vec{N} \bullet \vec{P_0} + D}{\vec{N} \bullet \vec{V}} * \vec{V}$

Om $\vec{N} \bullet \vec{V} = 0$ är planet och linjen parallella. Om planet och linjen är parallella skär linjen planet om och endast om $\vec{N} \bullet \vec{P_0} + D = 0$ vilket innebär att linjen ligger i planet.

Exempel:
Anta ett plan $y = 0$, det vill säga xz-planet med normalvektor $\vec{n} = (0, 1, 0)$. Linjen som ska undersökas är $P\vec{(t)} = 1, \vec{1}, 1 + t{-}1, \vec{-1}, -1$. Det kan utläsas att skärning kommer att ske när $t = 1$. Skärningen sker i punkt $(0, 0, 0)$.

Eftersom planets ekvation är $y = 0$ medför det att $D = 0$. $\vec{N} \bullet \vec{V} = (0, 1, 0) \bullet (-1, -1, -1) = -1$. Linjen är alltså inte i planet, och skärningspunkten kan beräknas. Enligt formeln sker skärningen i punkten $\vec{P} = (1, 1, 1) - \frac{(0,1,0) \bullet (1,1,1) + 0}{(0,1,0) \bullet (-1,-1,-1)} * (-1, -1, -1) = \vec{P} = (1, 1, 1) - \frac{1}{(-1)} * (-1, -1, -1) = (0, 0, 0)$

2.5.2 Linje och Sfär

För att avgöra om en linje skär en sfär kan följande samband användas: Linjen i det tredimensionella rummet $P\vec{(t)} = \vec{Q} + t\vec{V}$. Då ekvationen för en sfär skrivs om med hjälp av ekvationen för linjen kan följande samband härledas:

Först beräknas $D = 4((\vec{Q} \bullet \vec{V})^2 - (\vec{V} \bullet \vec{V})((\vec{Q} \bullet \vec{Q}) - r^2))$

- Om $D > 0$ skär linjen sfären i två punkter.
- Om $D = 0$ tangerar linjen sfären.
- Om $D < 0$ skär linjen inte sfären.

Sedan beräknas parametern t, för att sedan åter användas i linjens ekvation, för att ge den söka punkten. $t = \frac{-2(\vec{Q} \bullet \vec{V}) - \sqrt{D}}{2 * (\vec{V} \bullet \vec{V})}$ Exempel:

Anta att en sfär med radien 1 belägen i origo och en linje $P(t) = (2, 0, 0) + t * (-1, 0, 0)$.

För att avgöra om linjen skär sfären räknas först D ut.

$D = 4((2, 0, 0) \bullet (-1, 0, 0))^2 - ((-1, 0, 0) \bullet (-1, 0, 0) * (2, 0, 0) \bullet (2, 0, 0) - 1^2)) = 4((-2)^2 - 1 * (4 - 1)) = 4$

Eftersom $D > 0$ skär linjen sfären i två punkter. Nu beräknas t för att få reda på var linjen skär sfären.

$t = \frac{-2((2,0,0) \bullet (-1,0,0)) - \sqrt{4}}{2 * (-1,0,0) \bullet (-1,0,0)} = \frac{2}{2} = 1.$

$t = -1$ ger $P(-1) = (2, 0, 0) + 1(-1, 0, 0) = (1, 0, 0)$ Skärningen sker alltså i punkten $(1, 0, 0)$.

3 Användningsområden för 3D-motorer

Kapitlet beskriver översiktligt olika kategorier av datorprogram som använder 3D-motorer. Översikten beskriver 3D-spel, simuleringsprogram samt solidmodellering, se avsnitt 3.2. Kapitlet är till för att läsaren skall se vad en 3D-motor kan användas till. Kapitlet introducerar även begrepp som används hos 3D-applikationer. Dessa begrepp används även senare i uppsatsen och behövs därför kännas till.

3.1 Spel

De flesta spel som idag kommer ut på marknaden är 3D-spel. Dessa spel använder sig av tre dimensioner för att skapa den miljö och de karaktärer, se avsnitt 3.1.5, som spelet använder.

3.1.1 Förstapersonspel

Ett förstapersonspel innebär ett 3D-spel som använder sig av en kameravy, se avsnitt 5.2.1, som utgår från karaktärens egna ögon. Denna form av kameravy är vanlig inom actionspel. De actionspel som använder sig av förstapersonsperspektiv har även fått ett eget namn nämligen *first person shoot them up* spel. Figur 3.1 visar hur den virtuella världen kan se ut genom förstapersonsperspektivet.

Figur 3.1: En bild från spelet Half Life 2 från Sierra [21]

3.1.2 Tredjepersonsspel

Ett tredjepersonsspel innebär ett 3D-spel som använder sig av en kameravy där användaren ser sin karaktär ur någon annans ögon. Detta perspektiv använder sig av en kamera som är placerad bakom karaktären. Kameran kan även rotera runt karaktären beroende på hur denna förflyttar sig.

Ett exempel på ett tredjepersonsspel är ett motorspel. Ett motorspel innebär spel som använder sig av motorfordon i någon slags tävlingsform. Ett motorspel, eller *racingspel* som de även kallas använder sig av ett förstapersonsperspektiv, men inte i lika stor utsträckning som shoot them up spel. Figur 3.2 visar hur tredjepersonsperspektiv användas i ett motorspel.

Figur 3.2: En bild från spelet Grand Turismo 4 från Polyphony Digital [21]

3.1.3 Virtuella världar

Den verkliga världen är fylld av många olika intryck, som skuggor, ljus, mörker, former och färger. Intrycken ändrar upplevelsen av verkligheten på olika sätt. [9]

3D-spel använder sig av virtuella världar för att efterlikna dessa upplevelser. En virtuell värld visar detaljerade scener med känsla för hur djup och ser ut, samt beter sig som den verkliga världen gör. Verklighetskänslan förstärks av att spelen använder sig av ljudeffekter av olika slag. [9]

3.1.4 Objekt i 3D-spel

De Goes [9] anser att det existerar två olika typer av objekt i 3D-spel, nämligen statiska och dynamiska objekt. Den funktionalitet hos en 3D-motor som tas upp i uppsatsen tar inte hänsyn till om ett objekt är statiskt eller dynamiskt.

Med statiska objekt menas objekt som har en fast position och form. Statiska objekt är lättare att hantera än dynamiska objekt eftersom det inte krävs några uträkningar hur förflyttningar och transformeringar av objektet skall ske. Statiska objekt brukar användas för att skapa miljön i den tredimensionella världen. Byggnader, väggar och berg är exempel

på statiska objekt.

Med dynamiska objekt avses objekt som kan ändra form och förflyttas. Exempel på dynamiska objekt är de som som rör sig i en virtuell värld. Spelarkaraktärer, hissar och bilar för att nämna några av alla tänkbara dynamiska objekt.

3.1.5 Karaktärer

En karaktär kan definieras som en intelligent varelse, vanligtvis är objektet som representerar karaktären kapabel till animation och rörelse [9]. Det finns oftast två olika typer av karaktärer. Huvudkaraktären är den karaktär som användaren kontrollerar. Övriga karaktärer kontrolleras inte av användaren, men kan kontrolleras av andra användare eller av datorn.

Multiplayerspel är ett exempel på spel som använder sig av olika karaktärer. Användaren kontrollerar sin karaktär och spelar med eller mot andra användare som i sin tur kontrollerar sina karaktärer. Användarna befinner sig oftast på olika platser och använder sig av ett nätverk för att kunna spela mot eller med varandra.

3.1.6 Miljöeffekter

Miljöeffekter är icke-objekt som är nära bundna till spelets virtuella miljö [9]. Exempel på miljöeffekterna som kan simuleras i 3D-spel är ljud, ljus, skuggor, dimma och andra effekter som turbulens och jordbävningar. Vad för slags miljöeffekter som återfinns i ett spel är helt och hållet upp till konstruktörerna av spelet.

3.2 Övriga användningsområden

Solidmodellering är programvara för modellering inom Computer Aided Design (cad), design och mekanik. Med solidmodellering skapar en konstruktör eller designer olika modeller i en 3D-miljö. Modellen kan exempelvis vara designen på en ny bil.

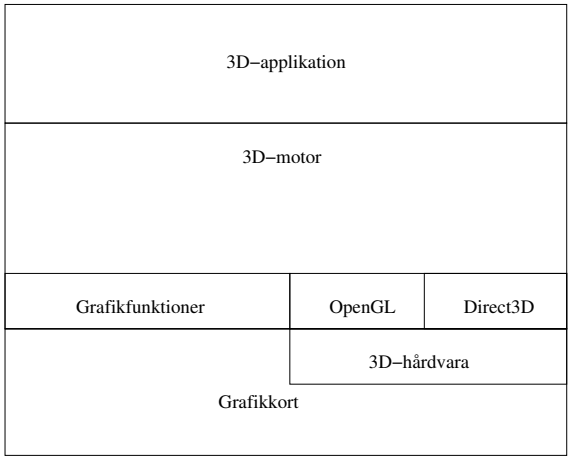
Tredimensionella simuleringsprogram kan användas för att visualisera en robots rörelser. På så vis kan exempelvis en produktionslina, ett antal robotar som samverkar, simuleras i en dator för att kontrollera att allt är korrekt och produktionen är redo att starta.

4 Översikt av 3D-motorer

Kapitlet ger en kort översikt av 3D-motorer. Kapitlet är till för att läsaren skall få en helhetsbild över vad en 3D-motor gör och hur den fungerar. Kapitlet presenterar och beskriver även culling, clipping, fysikmotor, spelmotor, texturering samt kollisionsdetektering.

4.1 Vad gör en 3D-motor

En 3D-motor är ett ramverk för att organisera, kontrollera och generera tredimensionell grafik. Ett ramverk är en uppsättning funktioner eller klasser avsedda att förenkla applikationsutveckling. En 3D-motor kan ses som ett lager mellan applikationen och den grafik som applikationen skall visa på bildskärmen. Funktionerna som används för att rita upp grafik kan vara en del av 3D-motorn, eller funktioner implementerade på ett grafikkort, se figur 4.1. Ett 3D-accelererat grafikkort hjälper datorns processor att rita upp grafik. 3D-



Figur 4.1: 3D-motorns roll mellan applikation och hårdvara

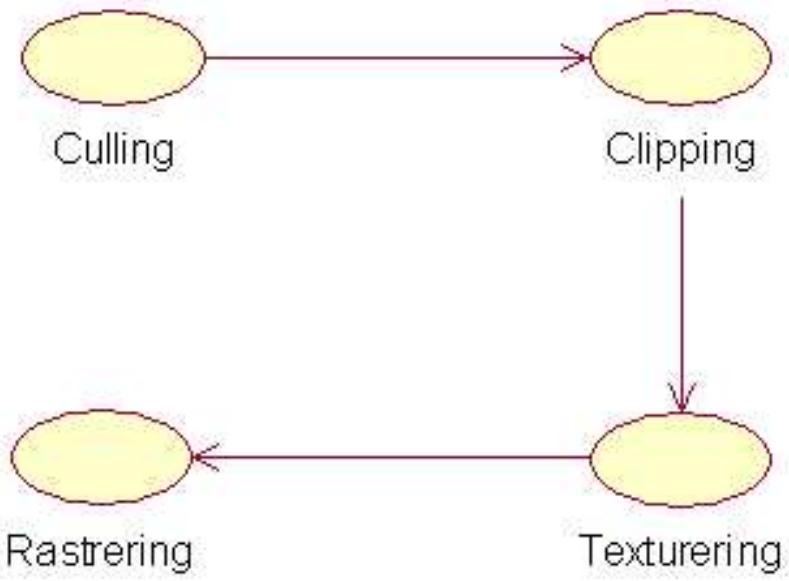
accelererade grafikkort är åtkomliga från 3D-motorn via en av API:erna (Appliction Programming Interface), OpenGL från Silicon Graphics Institute, eller Microsofts Direct3D API.

4.1.1 Fysikmotor och spelmotor

En fysikmotor är frikopplad från den grafiska representationen som sköts av 3D-motorn. En fysikmotor kontrollerar vilka händelser som objekt i ett spel utför eller inte utför och agerar utefter de regler som applikationen specificerat. Fysikmotorn hjälper 3D-motorn att upprätthålla realismen i en virtuell värld. Med realism menas exempelvis vad som ska ske när två obejekt kolliderat. [8] En spelmotor används vid kommunikationen mellan användaren och 3D-motorn. Spelmotorn hanterar interaktionen med användaren, samt sköter logiken för applikationen.

4.2 Så fungerar en 3D-motor

Detta avsnittet förklarar översiktligt hur en 3D-motor arbetar. Figur 4.2 visar övergripande vad en 3D-motor gör och i vilken ordning den gör det. Först undersöker 3D-motorn vilka



Figur 4.2: De viktigaste delarna i en 3D-motor

objekt och polygoner som skall ritas upp. Detta görs genom att på olika sätt undersöka om objekten är synliga eller inte ur den givna kameravyn. Culling, se kapitel 7, är ett samlingsnamn på olika tekniker för att avgöra objekts synlighet. När de icke synliga objekten har valts bort undersöker 3D-motorn om de objekt som finns kvar är helt eller delvis synliga. 3D-motorn ser till att endast rita upp den del av objekten som syns. Om ett objekt delvis kommer att synas på skärmen skall 3D-motorn bara bearbeta den synliga delen. Clipping, se kapitel 7, är ett samlingsnamn på olika tekniker som sköter hanteringen av delvis synliga objekt. Texturering, se avsnitt 4.4, innebär att 3D-motorn ger objekten i den virtuella världen ett utseende. Varje polygon hos ett objekt associeras till en textur. Slutligen sker rastreringen, se avsnitt 4.5 som ritar upp de objekt som skall synas på datorns bildskärm. För beskrivning av objekt, polygoner och kameravy se kapitel 5.

4.3 Culling och Clipping

Culling är samlingsnamnet på olika metoder för att avgöra vilka objekt och polygoner som är synliga ur en given kameravy. Målet med culling och clipping är att minska antalet objekt som renderaren ska hantera [8].

Frustum culling avgör om ett objekt befinner sig inom kameravyn. Om det inte befinner sig i kameravyn ska det inte heller synas på skärmen. Renderaren bearbetar inte objekt som inte kommer att synas. Occlusion culling avgör om ett objekt är dolt av andra objekt. Är objektet dolt behöver inte renderaren bearbeta det objektet. Rendering tas upp mer deteljerat i avsnitt 6.1.

Backface culling är en metod för att få renderaren att enbart hantera de polygoner, se avsnitt 5.1.2, hos ett objekt som är riktad mot kameravyn. Eftersom baksidan av ett objekt inte skall synas på skärmen skall renderaren inte bearbeta dem [8]. Clipping används precis som culling för att undvika att renderaren arbetar i onödan. Clipping undersöker

om objekt delvis befinner sig inom kameravolymen. Om så är fallet klipps objektet och endast den delen av objektet som befinner sig inom kameravolymen bearbetas. De olika cullingteknikerna tas detaljerat upp i kapitel 7. [8]

4.4 Texturering

En textur är en bild som är associerad till en polygon hos ett objekt. Texturer används för att ge objekten ett realistiskt utseende. En textur kan vara allt från en bild bestående av en färg till ett porträtt av en människa. Texturmapping, se avsnitt 6.5.1, innebär att associera en textur till en polygon. Exempelvis kan en bild av en människa mappas till en polygon. Bilden på människan kommer då att vara synligt när polygonen visas. [3]

4.5 Rastrering

I denna fas ritas grafik upp på datorns bildskärm. Rastreringen innebär att de geometriska figurerna som skall visas på skärmen ritas upp. Ofta används Direct3D eller OpenGL för att rastrera med hjälp av 3D-grafikkort. Alternativt kan 3D-motorn använda egna rastreringsfunktioner för att rastrera i mjukvaruläge.

4.6 Kollisionsdetektering

Kollisionsdetektering finns inte med i figur 4.2 eftersom att det är fristående från uppritningen. Kollisionsdetektering är viktig att behandla i uppsatsen.Kollisionsdetektering är den komponent som ansvarar för att kontrollera om objekt har kolliderat Eftersom 3D-motorn är den del av en 3D-applikation som vet var objekten i en virtuell värld befinner sig är det 3D-motorn som avgör om objekt kolliderat eller inte. En kollision inträffar när två objekt

samtidigt upptar samma utrymme i den virtuella världen.Det existerar många metoder för att detektera kollisioner. Detekteringsmetoderna skiljer sig åt i utförande och prestanda. Det leder till att metodval ofta bestäms av de krav applikationen som ska använda 3D-motorn ställer. En metod för kollisionsdetektering är att undersöka objektens respektive gränsvolymer mot varandra. Om det finns en skärning i någon gränsvolym, det vill säga om volymerna går i varandra, har objekten kolliderat. Uppsatsen är avgränsad till att beskriva kollisionsdetektering med hjälp av gränsvolymer. Kollisionsdetektering med hjälp av gränsvolymer räcker till en generell beskrivning eftersom det räcker för att visa funktionaliteten och syftet med kollisiondetektering.

När kollisionen upptäcks hanteras den av spelmotorn eller fysikmotorn. De bestämmer vad som ska inträffa efter att objekten kolliderat. Hur kollisionshanteringen sköter kollisionen beror på vilken typ av objekt som har kolliderat, samt var i världen och när kollisionen inträffade.

5 Begrepp och koncept

Kapitlet beskriver olika begrepp och koncept som är relevanta för 3D-motorer. Begreppen är centrala att förstå eftersom de i fortsättningen används frekvent.

Först definieras objekt, polygoner, kameravolym, främre och bortre plan, synplan samt koordinatsystem. Därefter beskrivs hur objekt transformeras från sitt lokala koordinatsystem till skärmkoordinater genom ett antal olika transformeringar. Sist i kapitlet förklaras gränsvolymer som används i culling, se kapitel 7, samt binaryspace paritioning träd.

5.1 Objekt

LaMothe [8] definierar ett objekt som en ordnad mängd av polygoner som bygger upp en tredimensionell modell. Det enklaste tredimensionella objekt som kan genereras med hjälp av polygoner är en pyramid. Pyramiden består av fem nodpunkter och fem polygoner. Kuben är den näst enklaste av alla objekt och består av åtta nodpunkter samt sex polygoner.

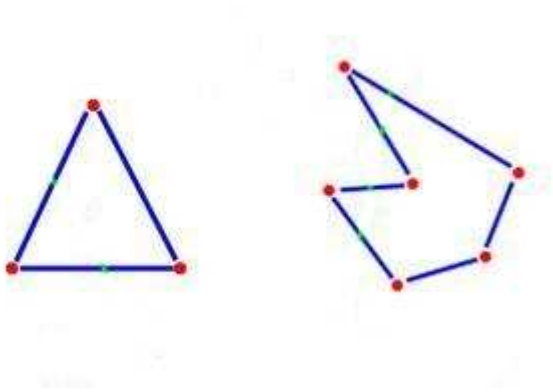
Sammansatta objektet består av en mängd andra delobjekt som tillsammans bygger upp det sammansatta objektet. Med sammansatta objekt kan en hel objektstruktur byggas upp. Ett exempel på ett sammansatt objekt är ett tåg. Ett tåg är ett objekt, men kan också ses som en uppsättning av ett lokobjekt samt flera tågvagnsobjekt [8]. Ett 3D-spel är fyllt med saker som inte räknas som objekt. Föremål som en ljusstrimma räknas inte som ett objekt eftersom ljusstrimman saknar substans [9]. Med substans i verkligheten menas att objekt är saker som går att ta på. En ljusstrimma går inte att ta på och saknar därför substans. Ett exempel på hur en kub kan skapas till en 3D-motor i språket C++ finns i bilaga A.

5.1.1 Att skapa objekt

3D-objekt skapas i program för 3D-modellering. Ett exempel på sådant program är 3D Studio Max från Discreet. [19] Användandet av modelleringsprogram ger kreatören stora möjligheter att påverka objektets utseende och rörelser. 3D-modelleringsverktygen kan spara objekt i ett format som 3D-motorn kan använda. [8]

5.1.2 Polygon

Den huvudsakliga primitiven i ett objekt är en polygon. En primitiv är en enkel geometrisk form från vilka mer komplexa former kan skapas. Varje polygon är uppbyggd av ett antal nodpunkter. En nodpunkt är en punkt hos en polygon. En vanlig polygon är den geometriska figuren triangel som består av tre nodpunkter. [8] [9] Det existerar två olika



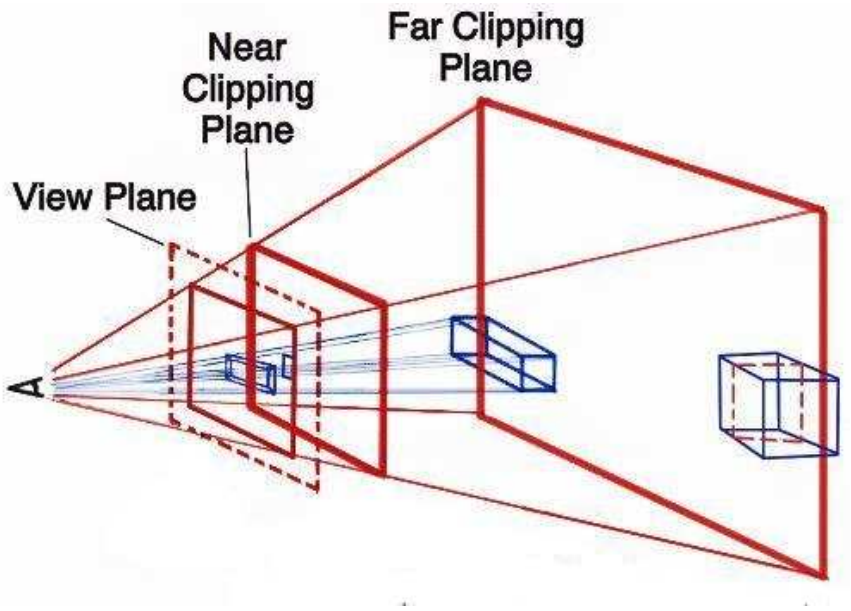
Figur 5.1: En konvex och en konkav polygon

typer av polygoner, konvexa polygoner och konkava polygoner, se figur 5.1. Skillnaden ligger i att hos konvexa polygoner existerar det inte två punkter där en linje kan dras genom utan att linjen passerar igenom polygonen. Hos konkava polygoner existerar det minst två punkter där en linje som går igenom punkterna passerar utanför polygonen. [9]

5.2 Synlighet

5.2.1 Kameravolym

Med kameravolym eller *view frustum* menas den trunkerade pyramid som är definierad av främre, bortre och fyra sidoplan. I figur 5.2 syns det främre och det bortre planet, se avsnitt 5.2.2, som bestämmer början och slut på kameravolymen. Figuren visar även synplanet, se avsnitt 5.2.3. Synplanet eller kameravyn ger en två dimensionell bild av kameravolymen. Sidoplanen definierar synfältet hos kameravolymen. Hos människan är synfältet 180 grader.



Figur 5.2: Kameravolym [25]

De synplan som avgränsar synen åt sidan är i 90 graders lutning mot kameravolymen. Det främre planet är det plan som trunkerar den närmsta spetsen hos pyramiden och det bortre planet markerar slutet på kameravolymen i djupled. [8]

5.2.2 Främre och bortre plan

De främre och bortre planen i kameravolymen definierar gränserna för den volym vars objekt kommer att hanteras av renderaren. Det främre planet är placerat vid $near_z$ och det bortre vid far_z. $near_z$ och far_z innebär det främre samt det bortre planens position på z-axeln. De främre och bortre planet har samma normalvektor $\vec{n} = (0, 0, 1)$. Det ger följande definition på planen:

- **Främre plan:** $0 * x + 0 * y + 1 * z = near_z$
- **Bortre plan:** $0 * x + 0 * y + 1 * z = far_z$

De objekt som befinner sig längre bort från kameran än det bortre planet eller är framför det främre planet kommer inte att renderas. [8]

5.2.3 Synplan

Ett synplan eller viewplane, är det plan som ligger framför kamerans främre plan. Objektens tredimensionella kamerakoordinater projiceras på synplanet. Koordinaterna övergår då till tvådimensionella koordinater. Synplanet representerar en tvådimensionell bild av världen. Synplanet fungerar som ett fönster in i den virtuella tredimensionella världen. I stället för ordet synplan används fortsättningsvis kameravy. När synplanet specifikt åsyftas kommer ordet synplan att användas. Avståndet mellan kamerans utgångspunkt och synplanet avgör perspektivgraden. Perspektivgraden avgör hur de tredimensionella objekten ska representeras i två dimensioner. Kamerakoordinater beskrivs i avsnitt 5.3.3. [8]

5.3 Koordinatsystem

Inom tredimensionell grafik används olika koordinatsystem. Varje objekt i den virtuella världen är byggt i ett eget koordinatsystem. Koordinatsystemet brukar benämnas modellkoordinater eller lokala koordinater. Alla objekt existerar i sin tur i ett världskoordinatsystem. Världskoordinater definierar den virtuella världen. Kameran har också ett eget koordinatsystem, kallat kamerakoordinatsystem.

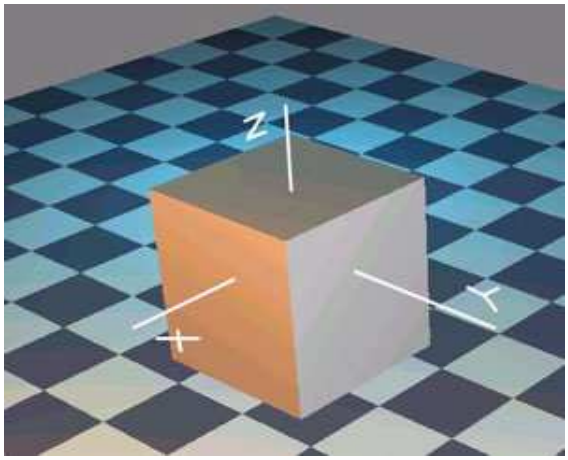
1. **Modellkoordinater** beskriver ett objekts form.
2. **Världskoordinater** beskriver positioner i en virtuell värld.
3. **Kamerakoordinater** beskriver positioner inom kameravolymen.
4. **Perspektivkoordinater** är tvådimensionella koordinater i ett synplan.
5. **Skärmkoordinater** definierar x och y koordinaterna på datorns bildskärm.

5.3.1 Modellkoordinater

Modellkoordinater används av objekt i en 3D-motor som dess eget koordinatsystem. Ett objekt är uppbyggt av en mängd nodpunkter definierade i objektets lokala koordinatsystem. Vanligtvis är objektets centerpunkt placerad i origo. [8] Figur 5.3 visar ett objekt med dess lokala koordinatsystem utmarkerat. Centrumpunkten hos objektet indikerar den punkt som används för att placera ut objektet i världen, se 5.4.1. Detta är även den punkt som ett objekt roterar runt.

Några exempel som förtydligar rotationspunkt:

Anta två objekt, en jordglob och en arm hos en människa. Rotationspunkten hos jordgloben befinner sig i globens geometriska centrum då detta är den punkten jordgloben ska rotera



Figur 5.3: Ett objekt med x, y och z axlar som utgör det lokala koordinatsystemet [24]

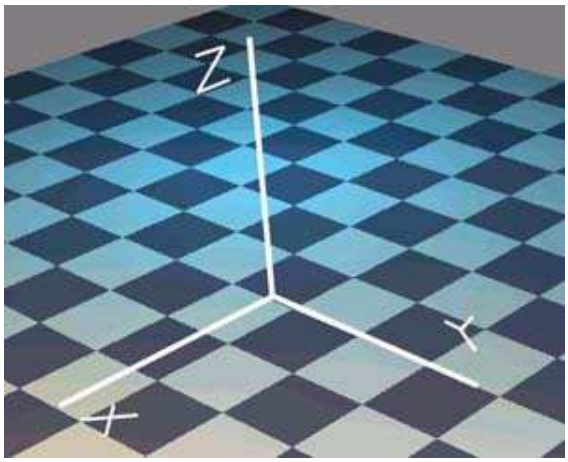
runt när någon snurrar på den. Rotationspunkten hos en arm befinner sig på två ställen. En människa kan röra sin arm vid armbågen samt vid axeln. Därför har en mänsklig arm två rotationspunkter. I fortsättningen av avsnittet kommer rotationspunkt användas för att indikera centrumpunkten.

När ett objekt roterar sker rotationen i objektets lokala koordinatsystem. Rotationerna utförs med hjälp av matrismultiplikationer, se avsnitt 2.3.2.

Rotationspunkten för ett objekt bör inte alltid placeras i objektets geometriska centrum. Valet av rotationspunkt hos objektet beror på hur objektet ser ut och vad det representerar. I följande exempel används ett objekt som representerar en människa. Objektet har delats upp i flera objekt. Anta att varje ben och varje arm är ett objekt. Huvudet samt överkroppen blir också var sitt objekt. Anta att varje objekt hos människan har en rotationspunkt, den punkt som objektet skall röra sig runt. Rotationspunkten för benen bör inte vara placerad i objektets geometriska centrum eftersom en människas ben inte rör sig runt sitt geometriska centrum. Ett ben bör kunna rör sig vid höften på människan snarare än vid benets geometriska centrum. Därför skall objektets rotationspunkt vara vid höften, i benets ände snarare än i mitten på benet. [8]

5.3.2 Världskoordinater

Världskoordinater är de koordinater som definierar den virtuella världen. När objekt skall positioneras i den virtuella världen, ges det en uppsättning världskoordinater [8]. Figur



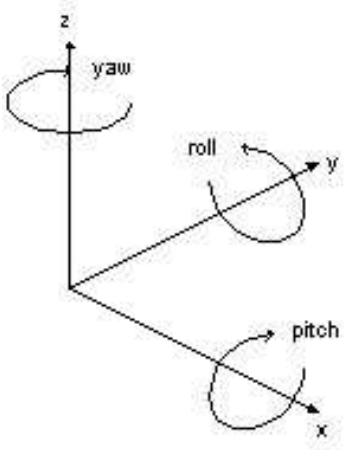
Figur 5.4: x, y och z axlarna utgör världskoordinaterna [24]

5.3.2 visar de positiva värdena hos x, y och z axlarna. Världskoordinaterna består även av negativa värden på axlarna.[8]

5.3.3 Kamerakoordinater

Kamerakoordinater används för att representera den kamera som användaren ser världen igenom. Kameran placeras i den virtuella världen genom att ge den en position (cam_x, cam_y, cam_z) i världen. Kamerans riktning definieras av (ang_x, ang_y, ang_z). Kameran ser den virtuella världen genom en kameravolym. [8]

En kamera kan röra sig på tre olika sätt. De olika sätten att rotera en kamera kallas för roll, pitch och yaw. Figur 5.5 visar hur en kamera kan röra sig längs x, y och z-axlarna. För att förtydliga hur pitch, rool och yaw fungerar anta hur en människa kan röra sitt huvud. När människan rör sitt huvud upp och ner kallas det pitch. När människan rör huvudet så



Figur 5.5: Hur en kamera kan röra sig längs koordinataxlarna

vänster öra placeras på vänster axel eller höger öra placeras på höger axel kallas det roll. När människan rör sitt huvud horisontalt åt sidorna kallas det yaw. [16] När rotationerna hos en kamera skall räknas ut används tre rotationsmatriser, se avsnitt 2.3.2, en för varje koordinataxel.

5.3.4 Perspektivkoordinater

Perspektivkoordinater används för att beskriva en tvådimensionell bild av de tredimensionella objekten. Projektionen är ett steg i att omvandla objektens punkter till skärmkoordinater, se avsnitt 5.4.4. [8]

5.3.5 Skärmkoordinater

Skärmkoordinater identifierar en pixels placering på datorns bildskärm. Skärmkoordinaterna definieras i två dimensioner och består av ett koordinatsystem med en x- och en y-axel. En pixel på datorns bildskärm har ett specifikt (x,y) värde. [8]

5.4 Transformationer

För att byta från ett koordinatsystem till ett annat används transformeringar. I avsnittet beskrivs de transformeringar som behövs för att flytta ett objekt i sitt lokala koordinatsystem till en tvådimensionell representation som används vid uppritningen.

5.4.1 Modellkoordinater till världskoordinater

Modell till världstransformationer används när modellkoordinaterna skall överföras till koordinater i den virtuella världen. När ett objekt placeras ut i den virtuella världen transformeras objektets lokala koordinater till världskoordinater [8]. En kub består av åtta nodpunkter. De åtta punkterna finns sparade i en datastruktur av något slag. När objektet placeras in i en virtuell värld skall dess modellkoordinater överföras till världskoordinater. För att placera ett objekt i en virtuell världen behövs endast kunskapen om var i den virtuella världen objektet skall placeras. Det anges genom att definiera ett värde på x, y och z axeln där rotationspunkten hos objektet skall placeras i den virtuella världen. Sen överförs modellkoordinaterna till världskoordinater. [8]

Anta en virtuell värld med dimensionerna $1000x1000x1000$. Anta att den kub som beskrevs tidigare skall sättas in i den virtuella världen på koordinaterna:

```
(worldCorr_x, worldCorr_y, worldCorr_z)
```

Världskoordinaterna, $worldCoor_x, worldCoor_y$ och $worldCorr_z$ anger vart i den virtuella världen objektets rotationspunkt skall placeras.

Följande pseudokod beskriver placeringen av ett objekt i den virtuella världen. Koden visas för att ge ett exempel på hur transformationerna kan implementeras:

```
ModelToWorld()
```

```
{
   struct cube_vertex      // Innehåller kubens punkter
   struct world_vertex // Här hamnar objektet i världen

   for(int i = 0; i < 8; i++)
   {
      world_vertex[i].x = cube_vertex[i].x + worldCorr_x
      world_vertex[i].y = cube_vertex[i].y + worldCorr_y
      world_vertex[i].z = cube_vertex[i].z + worldCorr_z
   }
}
```

Informationen om varje punkt i objektet överförs till en annan datastruktur som används för att spara undan informationen om objektet. Det görs för att lämna objektets datastruktur orörd.

Vidare associeras varje undansparat x, y och z värde med det x, y eller z värde där objektet skall sättas in i världen. [8]

5.4.2 Världskoordinater till kamerakoordinater

Värld till kameratransformering innebär att världen flyttas efter kameran. Världen roteras så att kamerans utgångspunkt är $(0, 0, 0)$ och kameran är riktad rakt ner längs den positiva z-axeln. [8]

Det innebär att alla nodpunkter i världen först måste flyttas så de får nya koordinater definierade av (cam_x, cam_y, cam_z). Sedan transformeras varje nodpunkt med inverserna, se avsnitt 2.2.5, till rotationsmatriserna, se avsnitt 2.3.2.

Vinklarna till de inversa rotationsmatriserna är vinklarna som beskriver hur kameran är roterad kring kamerakoordinatsystemets axlar. [8]

Appendix B visar hur det kan se ut när världskoordinater överförs till kamerakoordinater.

5.4.3 Kamerakoordinater till perspektivkoordinater

För varje nodpunkt n av ett objekt i kamerakoordinater beräknas dess perspektivkoordinater ut baserat på objektets z värde. Följande pseudokod visar transformationen:

```
void CamToPers()
{
   struct object; // objektets punkter
   struct camera; // kamerans position
   struct proj; // projicerade koordinater

   for(int i = 0; i < object.NrOFNode; i++)
   {
      ar = width / height;
      d = 0.5* width * tan(v/2) ;
      proj[i].x = d * camera.x / object.Node[i].z ;
      proj[i].y = d * ar * camera.y / object.Node[i].z;
      proj[i].z = camera.z;
   }
}
```

Variabeln *ar* används för bibehålla proportionerna mellan skärmens och projektionens höjd och bredd. Annars skulle resultatet bli en skev bild. Variabeln v är vinkeln på synfältet, det vill säga vinkeln mellan synvolymens sidoplan. Avståndet d är beräknat från kamerans

ursprung till synplanet. Detta värde bestämmer projektionen. [8]

5.4.4 Perspektivkoordinater till skärmkoordinater

Då bildskärmen använder skärmkoordinater måste perspektivkoordinaterna konverteras till skärmkoordinater innan de kan bli uppritade. [8] [7] Perspektivkoordinaterna konverteras på följande vis:

```
n_screen.x = (screen_widh -1) * ( proj_n.x +1) / 2;
n_screen.y = (screen_height -1) * ( proj_n.y +1) / 2;
```

5.5 Gränsvolym

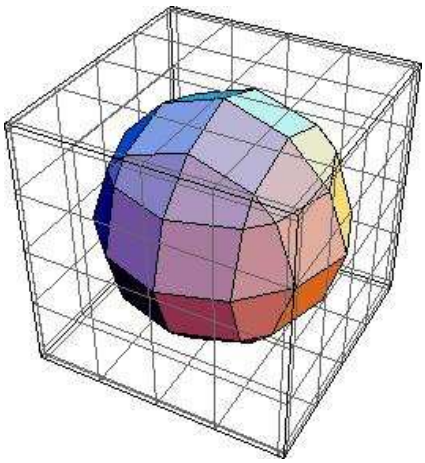
En gränsvolym är en geometrisk volym som omsluter ett objekt. Gränsvolymer används för att undersöka om objekt har kolliderat med varandra. Gränsvolymer används även för frustum culling. [7]

Idén bakom gränsvolymer är att ha en geometrisk figur som kan testas mot skärningar på ett effektivare sätt än att behöva kontrollera objektet självt. Antag ett objekt på 10000 polygoner. För att undersöka om objektet kolliderat med ett annat behövs varje polygon kontrolleras var för sig. Med gränsvolym undersöks om volymen kolliderat. Med de gränsvolymer som används är det matematiskt enklare att avgöra om kollision inträffat, då endast en operation behövs göras.

Anta två objekt som är omslutna av vad sin gränsvolym. Om inte objektens gränsvolymer har skurit varandra har inte heller objekten skurit. Dock måste inte objekten skära varandra bara för att dess gränsvolymer skär, vilket visas i figur 8.1. [7]

5.5.1 Gränskub

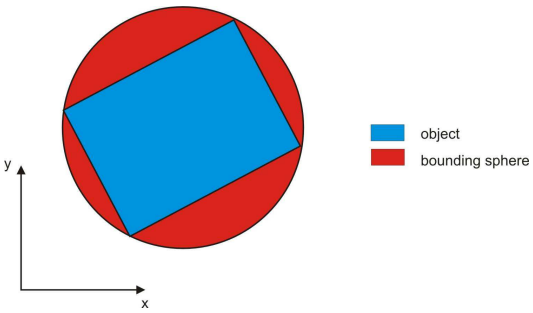
En gränskub är en kub som är placerad så att mitten av kuben är i objektets centrum. Sidorna i kuben är precis så stora att hela objektet omsluts.



Figur 5.6: Bild som exemplifierar en kub som gränsvolym [27]

5.5.2 Gränssfär

En gränssfär, se avsnitt 2.4.3 är en sfär som omsluter ett objekt. Sfären har en radie som är precis så stor för att kunna omsluta hela objektet.



Figur 5.7: Bild som exemplifierar en sfär som gränsvolym [28]

5.5.3 Gränscylinder

En cylinder kan ses som en bit av ett rör som har en radie, r. Cylinder har även en höjd h som markerar dess höjd. En gränscylinder är en cylinder som har placerats runt ett objekt. En gränscylinder har en höjd och radie som är stora nog för att precis omsluta ett objekt. Gränscylindrar använd i 3D-motorer till långsmala objekt därför att de kan bättre omsluta långsmala objekt än exempelvis gränssfärer. [8]

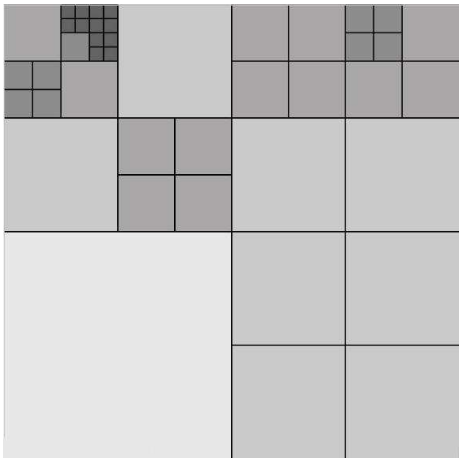
5.6 Bounding Hierarchical Volumes

Bounding hierarchical volumes (BHV) är ett sätt att gruppera objekt. BHV används vid culling och kollisionsdetektering. Anta att det existerar ett antal objekt som representeras av deras gränsvolym. Anta att någon slags trädstruktur används för att representera de grupperade objekten. BHV delar då in objekten i större grupper för att kunna culla hela grupper med objekt istället för enstaka objekt. BHV grupperar alla objekts gränsvolymer i världen samt insluter hela världen i en stor sfär. Sfären som innehåller hela världen fungerar som rot i trädet. [8]

Varje nod i trädet har en centerpunkt och en radie. Varje nod har även en lista som innehåller sina barnnoder samt en lista som innehåller objekten som är inslutna i noden. Det innebär att om en värld består av 100 objekt kommer rotnoden att ha en lista som består av alla 100 objekten. Anta att nästa nivå i BHV-trädet består av fem noder. Då kommer varje nod på den nivån att ha en lista bestående av 20 objekt. [8]

5.7 QuadTrees

Ett quadträd är en trädstruktur där varje nod har fyra barn. Med ett quadträd skapas celler, där varje cell är uppdelade i fyra mindre celler. Den egenskapen kan användas för att gruppera objekt i världen beroende på dess position. På så vis kan culling utföras, om kameran inte har en viss cell i sin vy kommer inga av dess barnceller heller vara i kameravyn. [7] I figuren 5.8 visas en bild av hur en stor yta delas in i ett antal mindre



Figur 5.8: Exempel på Quadtree [13]

celler. Varje cell delas i sin tur upp i fyra celler. Det innebär att varje förälder i quadträdet har fyra barn. Quadträd används i uppsatsen främst för culling. Eftersom quadträd är en hierarkisk struktur används den för att slippa culla objekt som inte kan befinna sig innanför kameravolymen.

Till skillnad från BHV används inga gränsvolymer för culling med quadträd. De behövs inte eftersom quadträdet avgör vad som kan ses. [13]

Skillnaden mellan quadträd och BHV är att med quadträd har varje förälder fyra barn, medans med BHV kan varje förälder ha n stycken barn. Vid användning av *breadth first* traversering kan det vara en fördel att använda BHV. Om trädet har mer än fyra barn blir

det effektivare att använda BHV träd än quadträd vid traverseringen.

5.8 Binary Space Partitioning

Binaryspace partitioning (BSP) är en trädstruktur för att gruppera en mängd polygoner.

Till skillnad från quadträd arbetar BSP med polygoner, medans quadträdet använder hela objekt. Vidare är quadträdet anpassat för culling, eller kollisionsdetektering av objekt. BSP kan användas för culling samt kollisionsdetektering på polygonnivå.

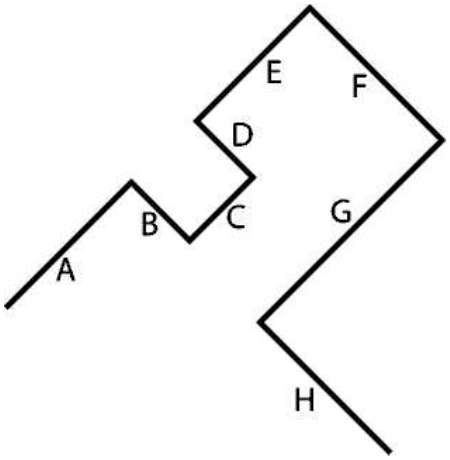
BSP delar upp en mängd polygoner med hjälp av partitioneringsplan. Partitioneringsplanen delar in polygonerna så att de antingen är på främre eller bakre sidan av planet. Om polygonen är framför eller bakom planet bestäms av dess och polygonens respektive normalvektorer. [8]

Indelningen i främre och bakre polygoner gör BSP till en bra metod för culling. Om kameran inte tittar mot ett plan i ett BSP då tittar den inte heller på några av de planen som ligger bakom. Då kan de helt utelämnas från renderingen. [8]

5.8.1 Skapa ett BSP-träd

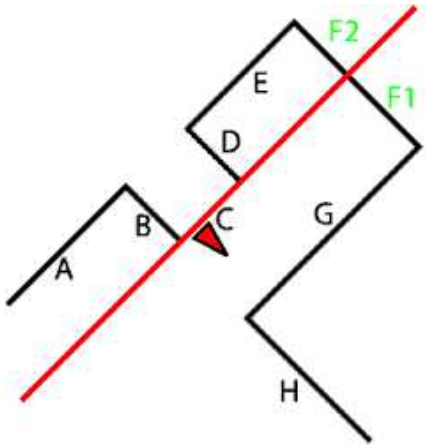
Ett BSP-träd skapas genom att ett plan i rummet väljs som partioneringsplan. Sedan läggs alla polygoner på främre sidan av partioneringsplanet till i en främre lista och alla polygoner bakom planet läggs till i en bakre lista. Om en polygon sträcker sig igenom planet, delas det upp i mindre polygoner på varsin sida om partioneringsplanet. Algoritmen utförs rekursivt på både bakre och främre listor. [8]

Följande exempel visar hur ett BSP växer fram. Figure 5.9 visar ursprungs polygonerna utan BSP.



Figur 5.9: Bilden visar de polygoner som BSP ska skapas för

I figur 5.8.1 väljs polygon C till partionerings plan. Det medför att polygon F måste delas upp på två polygoner, F1 och F2. Efter ett antal iterationer har den trädstruktur som

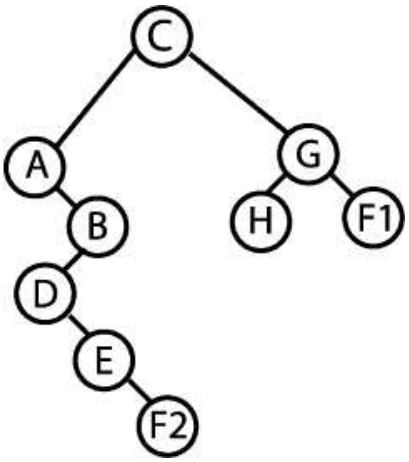


Figur 5.10: Bilden visar Polygon C som vald partioneringsplan

återfinns i figur 5.11 skapats.

Pseudokoden visar hur ett BSP träd kan skapas:

```
funktion createBSP(list polygons)
{
```



Figur 5.11: Bilden visar det skapade BSP-trädet

```
  list front
  list back
  partition_plane = select_best_part_plane(polygons)

  if partition_plane == empty  then exit

  add_front(front, partition_plane, polygons)
  add_back(back, partition_plane, polygons)

  createBSP(front)
  createBSP(back)
}
```

Med hjälp av heuristik, som är metoder baserade på statistik kan partitioneringsplanet väljas på ett så bra sätt som möjligt i varje steg. Ett bra valt partitioneringsplan delar på så få polygoner som möjligt. [8]

6 Uppritning

Det här kapitlet är den första delen i den generalla beskrivningen av 3D-motorer. Beskrivningen har delats upp i tre kapitel. Det första kapitlet förklarar 3D-motorns arbete ur ett helhetsperspektiv. De två resterande kapitlen går djupare in i de olika delarna hos en 3D-motor. Kapitlet behövs för att få en helhetsbild över den process som en 3D-motor gör.

Kapitlet beskriver renderingsprocessen som är den huvudsakliga uppgiften för en 3D-motor. Renderingsprocessen består av flera delar. Processens delar som rastrering, sortering, z-buffert och texturer beskrivs i detta kapitel.

6.1 Rendering

Att rita upp en 3D-bild kallas för att rendera. Hela renderingsprocessen utförs varje gång grafiken på skärmen ska uppdateras. Därför måste renderaren vara tillräckligt effektiv för att kunna ge applikationen en tillräckligt hög omritningsfrekvens. Begreppet omritningsfrekvens förklaras vidare i avsnitt 6.2. [8]

Nedan presenteras renderingsprocessens olika steg. Beroende på implementationstekniska krav som prestanda och val av algoritmer för respektive steg kan ordningen på stegen variera för olika 3D-motorer.

Det första steget är att transformera alla objekt som skall placeras ut i den virtuella världen till världskoordinater. Det innebär att objekten placeras ut i världen. [8]

I steg två avgörs med hjälp av frustum culling, vilka objekt som befinner sig inom kameravolymen. Renderaren fortsätter att bearbeta enbart objekten inom kameravolymen medans övriga ignoreras. Det tredje steget är backface culling. Backface culling tar bort de polygoner som är riktade bort från kameravyn. Steg två och tre är inte nödvändiga för

renderingen men de bidrar till att minska på det arbete som renderaren måste utföra och bidrar därmed till att 3D-motorn kan arbeta snabbare. [8]

I nästa steg omvandlas världskoordinaterna till kamerakoordinater. Genom 3D-clipping tas sedan polygoner bort från varje objekt som skär kameravolymen. Sedan projiceras objektens polygoner till perspektivkoordinater och transformeras vidare till skärmkoordinater.

Innan rastreringen tar vid bestäms polygonens texturutseende samt ljusegenskaper för varje polygon. Rastreringen ritar upp de polygoner som 3D-motorn har bestämt skall synas på skärmen. I renderingens avslutande steg kallad rastrering, ritas de olika polygonernas upp. Det sker genom att tvådimensionella representationer av polygonerna ritas upp en i taget. [8]

6.2 Omritningsfrekvens

Omritningsfrekvens eller *frames per second, FPS* är ett mått på hur många bilder renderaren genererar per sekund. En hög omritningsfrekvens gör att grafiken inte hackar. Ett exempel på det är om ett 3D-spel exekveras på en dator som inte uppfyller spelets hårdvarukrav, det vill säga den rekommendationen som speltillverkarna anser måste uppfyllas för att spelet skall vara spelbart, kommer spelet antagligen att hacka eftersom 3D-motorn inte klarar av att rita upp tillräckligt många bilder per sekund.

6.3 Rastrering

Rastrering innebär att rita upp de pixlar som renderaren tagit fram. Rastreringen ritar upp tvådimensionella projektioner av de tredimensionella objekten på skärmen, komplett med textur- och ljuseffekter.

Det är i rastreringen användandet av ett 3D-accelererat grafikkort märks, då många funktioner för rastrering är inbyggda i hårdvara [7]. Grafikkortet avlastar datorns processor från jobbet att rita upp grafik. Om ett 3D-accelererat grafikkort saknas utförs rastreringen med hjälp av datorns processor. Det leder till att processorn måste arbeta mycket mer och resultatet blir då ofta en lägre uppritningsfrekvens. 3D-grafik som ritas i mjukvaruläge genererar ofta en sämre grafisk kvalitet. Det beror på att många grafiska effekter som till exempel olika textur- och ljuseffekter inte används, då det skulle ta alldeles för mycket kraft av datorns processor.[8]

6.4 Sortering

Idén bakom sortering är att undvika att rita upp en pixel på skärmen mer än en gång [7]. För att få renderingsprocessen så effektiv som möjligt eftersträvas att varje pixel bara ska bearbetas en gång för varje omritning av bilden.

Termen djupkomplexitet anger hur många gånger en pixel ritas upp. Det teoretiskt bästa djupkomplexitetsvärdet är ett, vilket innebär att varje pixel endast ritas en gång.[7]

En metod för sortering är att använda en z-buffer. En annan metod är djupsortering med hjälp av BSP, se avsnitt 5.8. BSP sorterar på hela polygoner, till skillnad från z-buffert som jämför pixlar.

6.4.1 Z-buffer

En metod för djupsortering är z-buffert. Då avgörs vilka pixlar som ska ritas baserat på pixlarnas djup. Pixeldjupet är avståndet från en given punkt i kameravolymen till kamerans främre plan.

Vid användning av en z-buffer kontrolleras en pixels djup mot det tidigare djupvärdet för samma pixel. Om pixelns djupvärde är lägre, det vill säga ligger närmare kameravyn kommer den pixeln att ritas upp, och det gamla djupvärdet uppdateras. [8]

Principen för z-buffer beskrivs nedan i psuedokod.

```
zbuffer float[screen_width][screen_height];

bool compare(float x,  float y, float z)
{
  if (zbuffer[x,y] > z) {
    zbuffer[x,y] = z ;
    return true ;
  }
  return false;
}
```

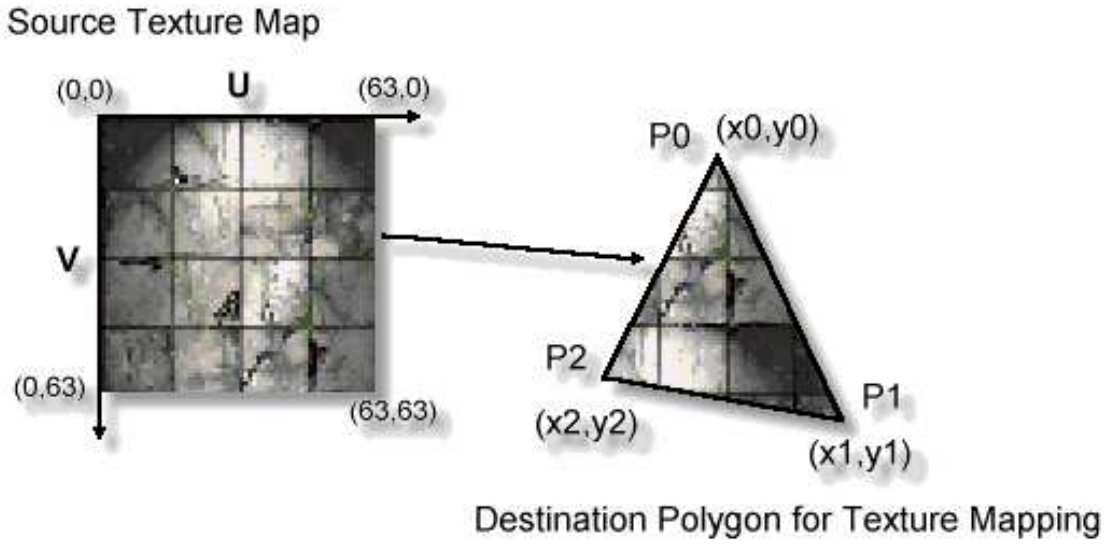
6.5 Texturer

Texturer består oftast av en bild. De används för att ge polygoner i ett objekt ett mer realistiskt utseende. Texturer är en tvådimensionell array av värden. Den minsta enheten i en textur kallas *textel*. En textel är en texturs motsvarighet till en pixel. Varje pixel på en skärm har ett unikt x och y värde. Dessa par (x,y) används för att referera till en enskild pixel. En enskild textel i en textur kan refereras till genom ett unikt u och v värde. [17]

Texturer använder ett koordinatsystem [0..1]x[0..1] där punkten 0,0 är nedre vänstra hörnet på texturen. Texturens mitt är punkten (0,5 , 0,5). [3]

6.5.1 Texture Mapping

Texture mapping innebär att en textur kopplas till en polygon. Figur 6.1 visar en textur



Figur 6.1: Texture Mapping [29]

som mappas till en pixel. Texturen representeras av en 64 * 64 pixlar stor bitmapbild. En bitmapbild är en bild som är uppbyggd av rader med olikfärgade pixlar. Dessa raderna skapar tillsammans en bild. [20] U och V anger koordinaterna hos texturen. P0, P1 och P3 är de tre punkterna som utgör polygonen. När texturen kopplas till polygonen anges vilka koordinater i texturen som ska bindas till vilken del av polygonen. En textur kan kopplas mot flera olika polygoner.

6.5.2 Mipmaps

En mipmap är en serie av texturer, där varje textur har en lägre upplösning än texturen innan. När objekt befinner sig långt bort från kameran väljs en textur med låg upplösning

för att representera objektet. När objektet närmar sig kameran väljs en textur med högre upplösning. [18]

Det går fortare för renderaren att bearbeta en textur med lägre upplösning än en med hög upplösning. Skillnaden i grafisk kvalitet är nästan obefintlig, men det besparar rendraren arbete.

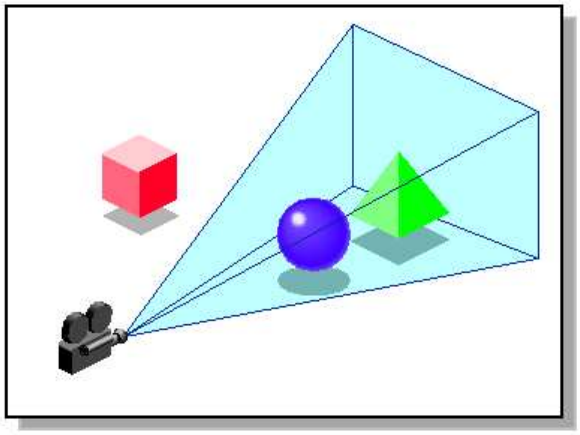
När polygonen däremot befinner sig nära kameran, används ursprungstexturen. Om ursprungstexturen inte används uppstår en viss försämring av bildkvaliteten. Mipmap använder Fourieranalys för att framställa texturer baserat på originaltexturen. [8] [3]

7 Culling och Clipping

Kapitlet tar upp begreppen culling och clipping som är två viktiga koncept för 3D-motorer. De används för att minska på renderingsarbetet, genom att ta bort objekt och polygoner som inte kommer att synas på skärmen.

Culling ser till att 3D-motorn inte bearbetar de objekt som av olika anledningar inte skall ritas upp. Det existerar ett flertal olika cullingalgoritmer som hanterar detta. Valet av algoritm beror på var objektet befinner sig i förhållande till kameravolymen. Kapitlet beskriver frustum culling som kontrollerar vilka objekt som befinner sig i kameravolymen, backface culling som hanterar de polygoner som är vända från kameravyn. Det tar även upp occlusion culling som undersöker vilka polygoner som döljs av andra polygoner.

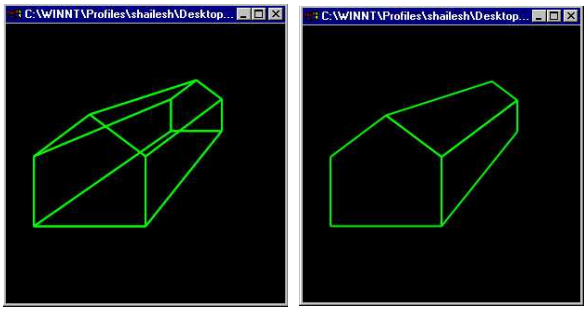
Clipping undersöker om polygoner skär kameravolymen. Om polygonerna skär kameravolymen måste de klippas och den delen som är inom kameravolymen skall bearbetas och ritas upp. Figur 7.1 visar ett exempel på frustum culling. Kuben som ligger utanför kameravolymen väljas bort från renderingen.



Figur 7.1: Exempel på Frustum culling [22]

7.1 Backface culling

Backface culling eller backface removal som det också heter används för att se till att inte slösa processorkraft på att rita upp polygoner som inte är synliga ur den givna kameravyn. Figur 7.2 visar en tredimensionell presentation av ett hus med och utan backface culling.



Figur 7.2: Effekterna av backface culling [26]

Bilden till vänster visar huset utan backface culling. Alla polygoner syns i figuren, även de som inte skall vara synliga ur den givna kamerayvn. Bilden till höger visar samma hus fast med backface culling. Här syns de polygoner som användaren ser ur den givna kameravyn.

Backface culling sker innan värld till kamera transformeringen, se avsnitt 5.4.2. Backface culling begränsar därför antalet polygoner som transformeras från världskoordinater till kamerakoordinater. [8]

Följande exempel visar hur backface culling fungerar. Anta att alla polygoner i ett objekt har numrerats. Nu beräknas polygonens normalvektor n_s, se avsnitt 2.4.2, och testas mot synvektorn v. Synvektorn är den vektor som definierar den riktning som kameravolymen har. Om vinkeln mellan synvektorn och polygonens normalvektor är mindre eller lika med 90 grader är polygonen synlig.

Själva testerna utförs med hjälp av uträkning av vektorernas skalärprodukt, se avsnitt 2.1.4. Testet använder sig av ett antal egenskaper som gäller för skalärprodukter.

- Om $n \bullet v = 0$ innebär det att vinkeln mellan vektorerna n och v är 90 grader.
- Om $n \bullet v > 0$ innebär det att vinkeln mellan vektorerna n och v är mindre än 90 grader
- Om $n \bullet v < 0$ innebär det att vinkeln mellan vektorerna n och v är större än 90 grader

Nu undersöks om skalärprodukten av de två vektorerna är mindre än 0. Om så är fallet är polygonens normalvektor vänd bort från kameravyn och ska därför inte bearbetas.

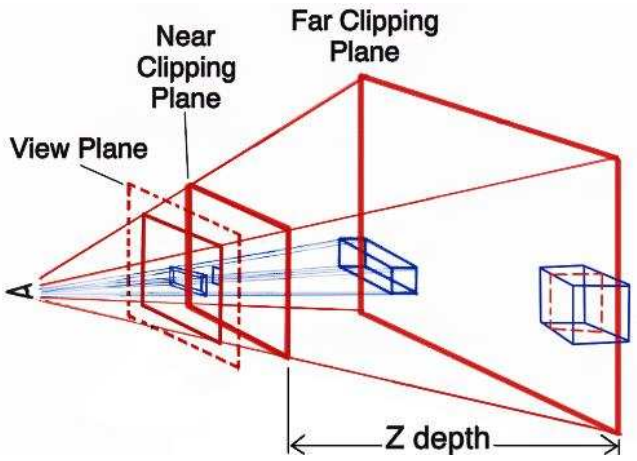
Backface culling sett i pseudokod:

```
if( Dotproduct(n, v) > 0 )
{
   // Polygonen är synlig
}
```

7.2 Frustum Culling

Frustum culling innebär att undersöka om ett objekt befinner sig inom kameravolymen. För att utföra frustum culling testas om centerpunkten hos objektets gränsvolym, se avsnitt 5.5, existerar inom någon av kameravolymens plan och där igenom avgörs om objektet skall bearbetas eller inte. Om gränsvolymen existerar inom kameravolymen, skall objektet behandlas av renderaren.

Uppsatsen beskriver tre olika gränsvolymer, se avsnitt 5.5, gränssfär, gränskub och gränscylinder. Avsnittet täcker Frustum culling för gränskuber och gränssfärer. Frustum culling kallas ibland också för object culling eftersom culling sker på hela objekt istället för som på enstaka polygoner som i exempelvis backface culling. [7] Figur 7.3 visar hur kameravolymen ser ut.



Figur 7.3: Exempel på Frustum culling i kameravyn [25]

I figuren finns det främre och det bortre planet, se avsnitt 5.2.2, med, z djupet vilket indikerar de värdena på z-axeln som ligger mellan de två planen. Figuren visar även synplanet, se avsnitt 5.2.3. Normalvektorn hos de sex planen i kameravolymen pekar alltid inåt i kameravolymen.

Figuren visar även två objekt. Ett av objekten befinner sig utanför kameravolymen. Det objektet kommer inte att bearbetas vidare och projekteras på synplanet som figuren visar.

Vilken ordning testningen mot kameravolymens plan sker kan ändras beroende på vilken miljö kameravyn befinner sig i. Ett landskap exempelvis expanderar mer på x och z axlarna än på y axeln. Därför kan frustum culling först testas mot de högra och vänstra planen innan det testas mot det övre och det undre planet.

Först undersöks frustum culling med en sfär som gränsvolym. Frustum culling med gränssfärer går ut på att undersöka om gränssfären, se avsnitt 5.5.2, ligger framför ett plan, bakom ett plan eller om det skär ett plan. [13]

Undersökningen görs genom att först räkna ut avståndet från centrumpunkten hos en given gränssfär till ett plan. Om absolutbeloppet av det värdet är mindre än radien hos sfären

skär sfären planet. Om värdet är större än noll är sfären på framsidan av planet. Om värdet är mindre än noll är sfären på baksidan av planet. När sfären är på baksidan av ett plan i kameravolym kan den inte befinna sig inom kameravolymen. [13]

Anta en ortsvektor som definierar centrumpunkten, $\vec{C}$ hos sfären. En normalvektor hos ett plan i kameravolymen $\vec{N}$ och avståndet från planet längs normalvektorn till kameravolymens origo D. Avståndet fås genom att beräkna skalärprodukten, Distance = Skalärprodukt$(\vec{C}, \vec{N})$ $+D$.

Ett exempel på hur frustum culling med gränssfärer kan se ut finns i bilaga C.

Undersökningen är annorlunda när det gäller gränskuber. 3D-motorn testar då varje punkt i kuben mot varje plan i kameravolymen. Om alla punkterna i kuben befinner sig framför alla sex planen anses objektet befinna sig innanför kameravolymen. Om minst en till sju punkter befinner sig inom kameravolymen anses objektet skära kameravolymen. [13]

Frustum culling för gränskuber använder sig av skalärprodukt vid beräkningen av en given punkts position relativt till ett givet plan. Skalärprodukten av varje punkt i kuben tillsammans med normalvektorn för det givna planet räknas ut. Avståndet från planets normalvektor till kameravolymens origo adderas till resultatet av skalärprodukten.

Om värdet av uträkningen är större än noll befinner sig punkten innanför kameravolymen. Om värdet är mindre än noll befinner sig den utanför.

Ett exempel på hur frustum culling med gränskuber kan se ut finns i bilaga D.

Maximal effektivitet hos frustum culling fås vid användandet av någon slags hierarkisk struktur för att strukturera objekten i världen. Det beror på att då behöver 3D-motorn inte undersöka alla objekten i världen. [13]

Quadtreese, se avsnitt 5.7 strukturen används i exemplifieringen av frustum culling.

Anta en värld som består av 100 objekt och att 3D-motorn använder sig av quadtree strukturen för att gruppera objekten. När frustum culling skall göras undersöks först om den översta noden i trädet, rotnoden kan cullas. Om så är fallet behövs inte några andra gränsvolymer i världen undersökas eftersom de befinner sig längre bort ifrån kameravolymen än rotnoden. [13]

I exemplet räckte det med en undersökning för att välja bort 100 objekt. Om 3D-motorn inte hade använt sig av en quadtrees skulle den behöva undersöka alla hundra objekten var och en för sig.

Anta vidare att det existerar ett objekt i världen som befinner sig inom kameravolymen. När 3D-motorn gör frustum culling på de 100 objekten behövs sammanlagt 6 undersökningar göras. Efter första undersökningen har antalet objekt reducerats till 50, vid nästa undersökning har antalet objekt reducerats till 27, nästa till 13 och så vidare tills antalet objekt att undersöka är nere i ett. [13]

Exempel i pseudokod på hur en binär trädstruktur kan användas för att effektivisera frustum culling:

```
FrustumCulling(binaryTree Tree, ViewFrustum Frustum )
{
   if ( Tree.volume is outside Frustum )
      then return OUTSIDE

   if (Tree.volume is leaf)
      render Tree.contents
   else
      FrustumCulling ( Tree.left, Frustum )
      FrustumCulling ( Tree.right, Frustum )
```

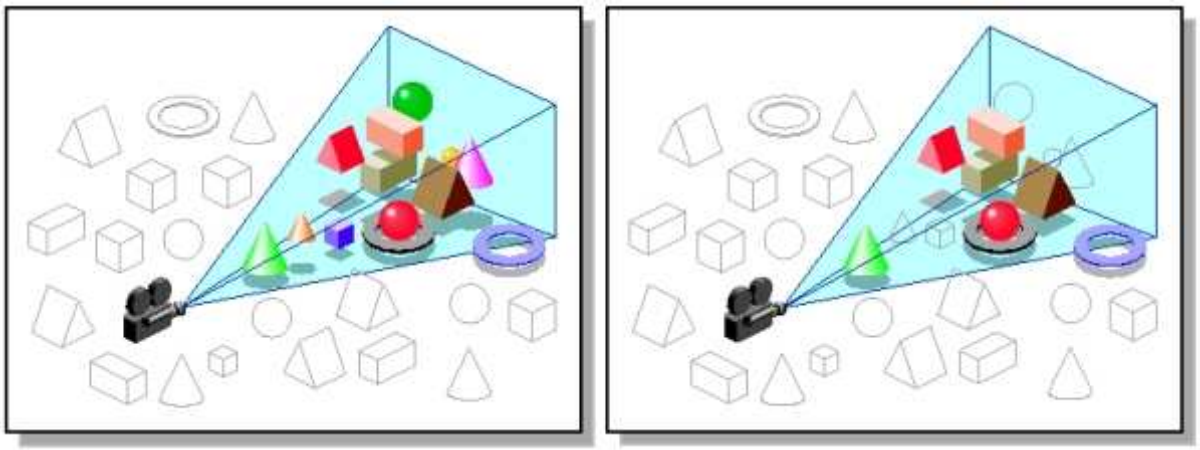
```
}
```

Koden undersöker först om den gränsvolym som undersöks för tillfället befinner sig utanför kameravolymen. Om så är fallet returneras OUTSIDE som indikerar att den delen av trädet är utanför kameravolymen. Om gränsvolymen är ett löv anropas rendraren med gränsvolymens innehåll. Om inget av ovanstående uppfyllts anropas frustum culling rekursivt först med trädets vänstra sida och sen med trädets högra sida.

7.3 Occlusion Culling

Occlusion culling är ett annat sätt att hantera synligheten i 3D-motorer. Occlusion culling hanterar de polygoner i kameravolymen som döljs av andra polygoner. [4] [7]

I exempelvis en stadsmiljö bestående av höghus bör det vara omöjligt att från en kameravy som är positionerad på marken helt och hållet kunna se alla hus. Användaren skall inte kunna se alla hus i världen eftersom de befinner sig närmare utgångspunkten hos kameravolymen helt eller delvis täcker de hus som befinner sig längre bort. BSP, se avsnitt 5.8,



Figur 7.4: Effekterna av Occlusion culling syns i den högra bilden [23]

kan användas vid implementeringen av occlusion culling. Eftersom uppsatsen tar upp BSP

sortering av polygoner används de i exemplifieringen av occlusion culling.

Eberly [7] beskriver två olika sätt att göra occlusion culling. Han beskriver occlusion culling i världskoordinater och i skärmkoordinater.

Anta ett BSP träd, A som representerar den virtuella världen. Trädet används för *front-to-back* sortering. *Front-to-Back* sortering innebär att de polygoner som befinner sig närmast kameran ritas upp först. [7]

Anta även ett BSP träd, B som används för att spara information om vad som är synligt ur den givna kameravyn [7]. Först undersöks occlusion culling i världskoordinater. BSP trädet, B representerar kameravolymen. Partitionsplanen i trädet, se avsnitt 5.8, är de sex planen som definierar kameravolymen. Träd A traverseras utifrån den givna ögonvyn. Den givna ögonvyn är utifrån origo hos kameran. Varje polygon som påträffas i traverseringen hanteras av trädet B och faktoriseras in i delpolygoner. [7]

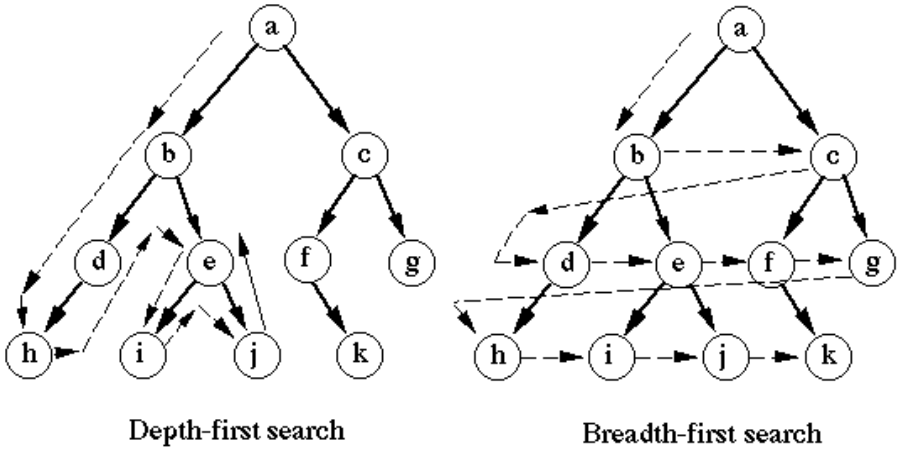
Varje delpolygon anses vara helt synlig eller helt osynlig. Varje synlig delpolygon används sen för att definiera nya partitionsplan som formas med hjälp av ögonvyn och kanterna hos delpolygonerna. De nya partitionsplanen formar en pyramid med ögonvyn som origo. Den del av världen som existerar i pyramiden men är bakom en delpolygon anses vara osynlig för användaren. [7]

Occlusion culling i skärmkoordinater sker i två dimensioner. Trädet A traverseras på nytt. Varje påträffad polygon delas upp i delpolygoner och anses vara helt synlig eller helt osynlig ur den givna kameravyn. Eftersom trädet A sorterade polygonerna i *front-to-back* ordning kommer de polygoner som är synliga efter clipping, se avsnitt 7.5, att fortsätta vara synliga efter B trädets beräkningar. [7]

7.4 Optimering

Optimering av culling kan ske med hjälp av ett bounding hierarchical volume träd. För en definition av bounding hierarchical volumes se avsnitt 5.6. Anta ett BHV-träd som är konstruerat för att hantera gränssfärer. När BHV träd skall användas för culling undersöks först om det går att culla alla objekten i världen. Det görs genom att utföra culling på rotnoden. Om rotnoden kan cullas, behöver inga objekt i världen bearbetas. Resten av noderna i trädet undersöks på samma sätt. Ett sätt att visa vilka noder i trädet som skall cullas och vilka som skall bearbetas är att sätta en flagga på de noder som skall cullas.

Kontrollen av vilka noder som skall cullas eller inte sker på följande sätt. Om den översta noden inte kan cullas kommer en rekursiv algoritm att traversera varje kant hos den aktiva noden och försöker culla varje nod den stöter på. Traverseringen kan ske på två olika sätt. Antingen traverserar algoritmen trädet *depth first* eller så traverseras trädet *breadth first*. Figur 7.5 visar hur depth first och breadth first sökning går till. De streckade pi-



Figur 7.5: depth first och breadth first sökning

larna indikerar i vilken ordning trädet traverseras. Traverseringen börjar från rotnoden i träden. Depth first sökning innebär att ett träd traverseras hela vägen ner till en lövnod. När algoritmen nått en lövnod går den upp till nodens förälder och försöker fortsätta tra-

verseringen där. Om det inte går försöker algoritmen fortsätta traverseringen ytterligare en nivå upp. Breadth first sökning innebär att undersökningen sker på alla noderna på den nivån i trädet som undersöks för tillfället innan undersökningen sker på nästa nivå. [8]

När en nod markerats som cullad kommer inga av den nodens barn att kontrolleras eftersom de inte skall kunna existera inom kameravolymen om deras förälder inte gör det [8]. Eftersom varje nod har en lista som innehåller nodens barn markeras alla noderna i listan som cullade. Algoritmen låter bli att traversera de markerade noderna

Följande exempel är hämtat ur LaMothe [8]. Anta en virtuell värld som består av 10000 objekt som är jämt utspridda i världen. Rotnoden i trädet innehåller alla objekten i världen. Noderna på nästa nivå är minst hälften så stora som rotnoden och innehåller tillsammans alla objekten i världen. Anta att det existerar följande BHV trädstruktur:

- **Roten:** Består av en BHV som har radien r som innebär radien på sfären. Rotsfären inkapslar alla objekten i världen. Rotsfären har alltså en lista som består av 10000 objekt.
- **Första nivån:** Består av fem stycken BHV med radien r/2. Det innebär att varje sfär på denna nivå är hälften så stor som rotnoden. Varje BHV har en lista bestående av 2000 objekt.
- **Andra nivån:** Består av BHV med radien r/4. Det innebär att varje sfär är hälften så stor som på föregående nivå. Anta att nivån består av 20 BHV, där varje BHV har 500 objekt.
- **Tredje nivån:** Består av BHV med radien r/8. Även här är varje sfär hälften så stor som på föregående nivå. Anta att nivån består av 1000 BHV, där varje har 10 objekt.

Anta att 3D-motorn skall testa objekt i världen med frustum culling. Motorn börjar med

att försöka culla roten i BHV-trädet. Om det lyckas behövs inga mer undersökningar. Alla objekten i världen har just cullats och motorn besparades från att behöva undersöka all 10000 objekt.

Anta att 3D-motorn måste traversera hela trädet med frustum culling och att traverseringen sker med breadth first. Algoritmen kommer att markera de objekt som skall cullas under traverseringen. De noder som algoritmen markerat behöver inte traverseras vidare. Hur många objekt som måste undersökas under vägen beror på hur många objekt som är synliga. Anta att endast en BHV på varje nivå behöver undersökas ytterligare. Då kommer antalet undersökningar att bli 20 istället för 10000. [8]

7.5 Clipping

Clipping undersöker om polygoner delvis existerar inom ett område. Om en polygon delvis existerar inom kameravolymen skall 3D-motorn se till att bearbeta den del som existerar inom kameravolymen. Avsnittet syftar att förklara den clipping som är relevant för 3D-motorer, i två och tre dimensioner.

Clipping hos 3D-motorer delas upp i två olika kategorier, 3D-clipping och 2D-clipping. 2D-clipping inträffar vid det sista steget av rendering. 2D-clipping klipper de primitiver, se avsnitt 5.1.2, som inte kommer att synas på skärmen när scenen ritas upp. 2D-clipping sker i skärmkoordinater, se avsnitt 5.3.5. [14]

3D-Clipping sker i världskoordinater, se avsnitt 5.3.2. I 3D-clipping används kameravolymen som gräns för vad som skall ritas upp. Clipping påverkar de primitiver som skär kameravolymen. 3D-clipping skär eller klipper bort den del av primitiverna som befinner sig utanför kameravolymen.

Först undersöks clipping av linjer mot kameravolymen. Anta en linje definierad av två

punkter, A och B. Först undersöks på vilken sida av ett plan i kameravolymen varje punkt i linjen är. Det görs genom att beräkna avståndet från punkten till planet. Skalärprodukten, se avsnitt 2.1.4, används för att beräkna avståndet. Uträkningen liknar den som gjordes i avsnitt 7.2, när centerpunkten hos gränssfären undersöktes mot planen i kameravolymen. [13]

Anta en ortsvektor $\vec{P}$ som definierar punkten P i en linje. En normalvektor $\vec{N}$ hos ett plan i kameravolymen och avståndet från planet till kameravolymens origo D. Avståndet från punkten till planet fås genom att beräkna Distance = Skalärprodukt$(\vec{P}, \vec{N})$ $-D$.

Pseudokod på hur undersökningen vart punkterna i linjen befinner sig finns i bilaga E.

Om $Distance < 0$ befinner sig punkten på baksidan av planet och därför inte inom kameravolymen.

Om $Distance = 0$ befinner sig punkten på planet.

Om $Distance > 0$ befinner sig punkten innanför kameravolymen. Om båda punkterna befinner sig utanför kameravolymen innebär det att linjen skall cullas och den inte behöver tas hand om här.

Om båda punkterna befinner sig innanför kameravolymen behöver 3D-motorn inte bearbeta linjen eftersom den varken skär planet och behöver clippas eller befinner sig utanför planet och behöver cullas.

Om punkterna befinner sig på olika sidor av planet skär linjen planet. 3D-motorn räknar då ut vart på linjen planet skärs. [14]

Punkten där linjen skär planet kan räknas ut på följande sätt:

```
typedef struct Point{
   float x, y, z;
```

```
} POINT;

LineClipping()
{
   Point dotA;   // Avståndet från planet till punkt A i linjen
   Point dotB;   // Avståndet från planet till punkt B i linjen
   Point intersectionpoint;

   float s = dotA/(dotA-dotB);   // ger ett värde mellan 0 och 1

   intersectpoint.x = dotA.x + s*(dotB.x-dotA.x);
   intersectpoint.y = dotA.y + s*(dotB.y-dotA.y);
   intersectpoint.z = dotA.z + s*(dotB.z-dotA.z);
}
```

När skärningspunkten har räknats ut behövs den punkten som befinner sig utanför kameravolymen ersättas med skärningspunkten, så har linjen klippts.

När en polygon skall klippas mot ett plan skall varje linje i polygonen clippas mot planet. När en polygon skall klippas mot kameravolymen undersöks varje linje i polygonen mot varje plan i kameravolymen och klipps därefter. [14]

8 Kollisionsdetektering

Kollisionsdetektering är en väsentlig del hos 3D-motorer. Ett 3D-spel är vanligtvis fyllt av objekt som på något sätt kommer att interagera med varandra.

Det kan handla om bilar som tävlar mot varandra på en racingbana, två karaktärer som slåss mot varandra eller *shoot-them-up* spel där användaren styr sin karaktär genom en värld fylld med fiender.

Kapitlet beskriver kollisionsdetektering med hjälp av gränssfärer och gränscylindrar. Det existerar fler kollisionsdetekteringsmetoder än de som presenteras i uppsatsen, men i nittionio procent av fallen fungerar de enkla algoritmerna lika bra som komplexa kollisionsdetekterings algortimer. [8]

8.1 Kollisionsdetektering

Om ett spel skall simulera verkligheten, måste spelet kunna efterlika verkligheten på så många sätt som möjligt. Kollisionsdetektering är en viktig del av en 3D-motors verklighetssimulering. Kollisionsdetektering bygger på att 3D-motorn skall tillhandahålla realistiska interaktioner mellan objekt. Dessa interaktioner tillhandahåller spelaren med grunderna för verklighetsuppfattningen, spelaren kan bättre förstå och navigera i en värld där saker beter sig som deras verkliga motsvarigheter gör. [2]

Algoritmerna för att skapa en kollisionsdetekterare för dynamiska objekt anses vara svårare än för statiska objekt. Det beror delvis på att med dynamiska objekt ökar antalet dimensioner ökar till fyra [7]. Tiden är den fjärde dimensionen som måste beaktas i kollisionsdetektering för dynamiska objekt. statiska och dynamiska objekt är beskrivna i avsnitt3.1.4.

Kollisionsdetektering med gränsvolymer, se avsnitt 5.5, används för att kollisionssystemet

ska kunna kontrollera om några objekt upptar samma utrymme i den virtuella världen.

Till skillnad mot frustum culling som används för att jämföra gränsvolymer mot planen i kameravolymen jämförs nu två gränsvolymer mot varandra, och om de skär varandra, måste detekteraren undersöka respektive objekt vidare. [8]

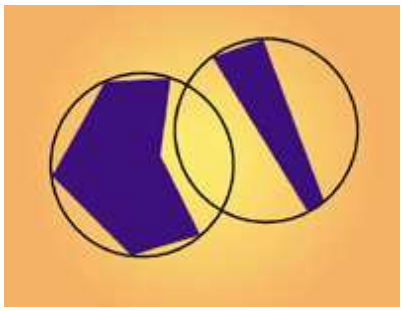
8.1.1 Kollisionsdetektering med gränssfärer

Den gränsvolym som är enklast att använda för kollisionsdetektering är gränssfären. Sfären anses vara enkel att använda eftersom matematiken för att räkna ut kollisionen mellan två sfärer är lätt jämfört med exempelvis gränscylindrar. Vid användning av gränssfärer undersöks om avståndet mellan sfärernas centrumpunkter är mindre än summan av deras radie [2]. Vid användning av gränscylindrar undersöks också om avståndet mellan cylindrarnas centrumpunkter är mindre än summan av dess radie. Användet av gränscylindrar kräver även att cylindrarnas position relativt till varandra på y-axeln undersöks. [8]

Följande pseudokod beskriver kontrollen om två gränssfärer för objekt a och b kolliderat.

```
if (distance(c.position, d.position) < (c.radius + d.radius )) {
  //Collision occurred
}
```

Anta två gränssfärerer c och d som används för att precis omsluta objekten A och B. c har radien $c.radius$ och d har radien $d.radius$. Anta att gränssfärerna har var sin centrumpunkt, definierad av $c.position$ och $d.position$. De två objekten skär varandra om avståndet mellan $c.position$ och $d.position$ är mindre än summan av $c.radius$ och $d.radius$. Figur 8.1 visar två objekt vars gränssfärer har krockat. Notera att objekten i sig inte har krockat. Valet av gränsvolym är viktigt för att få en så effektiv kollisionsdetektering som möjligt. Gränsvolymen bör vara vald så att den omsluter objektet så precist som möjligt.



Figur 8.1: Två objekt med gränssfärer som har krockat

8.1.2 Kollisionsdetektering med gränscylindrar

Cylindrar går också att använda som gränsvolymer.Cylindrar är att föredra som gränsvolymer till objekt som har en långsmal geometri. Anta ett objekt som representerar en lång pinne. En sfär är inte den bästa gränsvolymen till det objektet. Om två pinnar skulle använda sig av sfärer som gränsvolymer kan en kollision detekteras långt innan objekten i sig skulle kolliderat.

Så länge cylindrarna hos de kolliderande objekten är orienterade på samma sätt, fungerar detekteringen på liknande som för gränssfärer. [8]

Anta två objekt A och B, samt deras gränscylindrar c och d. Vid kollisionsdetektering behövs två saker testas. Först om cylindrarna överlappar varandra på y axeln samt om cylindrarna skär varandra.

Följande kod beskriver kontrollen om två gränscylindrar för objekt a och b kolliderat.

```
struct Point
{
   float, x, y, z;
};POINT
```

```
struct BoundingCylinder
{
   Point position;
   float radius;
   float y;
};BOUNDINGCYLINDER

void CheckCyl()
{
   BoundingCylinder c, d;

   if ( (distance(c.position, d.position) < (c.radius + d.radius )) &&
      ( absoluteValue(c.y - d.y) < ((c.hight + d.hight) \ 2))
   {
      //Collision occurred
   }
}
```

Position definierar vart i den virtuella världen gränscylindrarna befinner sig. Position består av en punkt i världskoordinater, se avsnitt 5.3.2. Funktionen distance räknar ut avståndet mellan gränscylindrarna i världkoordinater. Om avståndet är mindre än summan av cylindrarnas radie och om cylindrarna överlappar varandra på y-axeln har cylindrarna krockat.

8.1.3 Optimering

Kollisionsdetekteringen kan optimeras med hjälp av BSP-träd, se avsnitt 5.8. Följande exempel är taget ur LaMothe [8]. Anta en virtuell värld som består av 1000 objekt. Kollitionsdetektering på dessa 1000 objekten skulle innebära att 3D-motorn behöver göra 1000

* 1000 beräkningar, det vill säga 1,000,000 stycken. Anta en värld med 10000 objekt. Om den virtuella världen innehåller 1000 objekt krävs det 100,000,000 beräkningar för att göra kollisionsdetektering på alla objekt.

Om 3D-motorn först hirarkiskt sorterar världen med BSP eller octtree, se avsnitt 5.6, kan motorn dela upp världen i regioner eller celler. Varje cell eller region i sig kan innehålla ett par hundra objekt. Anta att världens 10000 objekt delas upp i celler där varje cell innehåller 50 st objekt. Världen består då av 10000/50 = 200 celler. 3D-motorn behöver genomföra 50 * 50 beräkningar för varje cell. Det innebär att 3D-motorn sammanlagt behöver genomföra 200*(2500) beräkningar. Det krävs 500,000 beräkningar för att utföra kollisionsdetekteringen med en hirarkisk struktur, vilket är 5% av det ursprungliga antalet beräkningar. [8]

9 Resultat och rekommendationer

Det huvudsakliga målet med uppsatsen var att framställa både en generell och detaljerad beskrivning av 3D-motorer med fokus på kärnfunktionalitet. Dessa mål har vi nått i och med färdigställandet av uppsatsen

För att kunna skapa en generell beskrivning var vi först tvungna att beskriva de bakomliggande och underliggande koncepten hos 3D-motorer. Det är därför uppsatsen består av ett kapitel som beskriver olika matematiska teorier samt ett kapitel som beskriver olika koncept och begrepp som rör 3D-motorer.

Den generella beskrivningen består av kapitel sex, sju och åtta. Kapitel sex fungerar som en helhetsbild över 3D-motorer med avseende på kärnfunktionalitet. Här beskrivs den grafikgenererande process som en 3D-motor utför. Kapitlet beskriver även olika funktionella aspekter vid uppritning av grafik.

Kapitel sju beskriver culling och clipping. Vi har funnit att de culling- och clipping-tekniker som tas upp i kapitlet är viktiga för att en 3D-motor skall kunna fungera optimalt. Det leder till att prestandan ökar då motorn inte behöver arbeta i onödan för att bearbeta objekt och polygoner som ändå inte kommer att synas.

Dessa tekniker är används för att optimera. Om 3D-motorn endast kommer att använda ett litet antal objekt behövs inte culling användas eftersom optimeringen ändå inte kommer att märkas på prestanda.

Det sista kapitlet i den generella beskrivningen tar upp kollisionsdetektering. Kapitlet tar inte upp kollisionshantering eftersom det ansvaret ligger hos fysik- eller spelmotorn, se avsnitt 4.1.1. Kollisionsdetekteringen som beskrivs i kapitlet anses vara en del av kärnfunktionaliteten eftersom den bidrar till realismen hos den 3D-applikation som 3D-motorn är en del av.

Vi har beskrivit kollisionsdetektering med hjälp av gränsvolymer. Den metoden ger ingen garanti för att objekten faktiskt kolliderat. För sådana kravs krävs ytterligare kontroll där varje polygon i de två objekten kontrolleras mot varandra för att avgöra om de skär varandra. Det finns ytterligare aspekter inom kollisiondetektering som var ett objekt har kolliderat och när har det inträffat. Dessa aspekter är betydligt mer komplicerade och har därför lämnats ute ur uppsatsen.

Vi rekommenderar att de som vill utveckla en 3D-motor först sätter sig ner och läser in sig ordentligt i ämnet. Ämnet 3D-motorer visade sig vara större och komplexare än vi först trodde när vi bestämde oss för att försöka sig på att göra en 3D-motor. Därför rekommenderar vi att de som vill koda en egen 3D-motor först skapar sig en bra helhetsbild över allt som skall ingå i motorn och alla problem som kan uppstå vid implementationen.

3D-motorer är idag ett så avancerat område att det inte är rekommendera att skapa en egen motor från grunden, i annat än utbildningssyfte. Ska den användas kommersiellt är det bättre att köpa en licens på en redan befintlig 3D-motor.

Vi har använt ett antal bra böcker, men främst är det Eberly [7] och LaMothe [8] som varit oss till mycket stor hjälp och rekommenderas varmt.

10 Summering av uppsats

När vi började skriva uppsatsen visste vi inte hur omfattande området var. Området var bredare då det existerar en stor mängd olika tekniker och metoder som används.

Den första tanken var att vi skulle skriva en egen 3D-motor från grunden. Det visade sig vara ett för stort åtagande för ett projekt på 20 veckor. Uppsatsen avgränsades snabbt istället till en beskrivning. Det krävdes mycket läsande för att förstå de olika begreppen och de olika teknikerna som rör 3D-motorer. Det krävs en djupare kunskap inom området för att lyckas med att skriva en egen 3D-motor. Att skriva en 3D-motor från början kräver många timmars studier samt många timmars testning och utveckling.

Det finns mycket information och litteratur angående 3D-motorer. Det gäller bara att veta var man skall leta. Internet är en bra informationskälla där det finns många bra tutorials och exempel. Ibland är det dock svårt att hitta relevant information i den uppsjö av länkar och sidor som finns.

Vår uppsats har bara skrapat lite på ytan inom ämnet. Det finns många fler grafiska effekter, samt metoder för att lösa de problemen som är beskrivna i uppsatsen. 3D-motorer och framförallt 3D-programmering har visat sig vara ett område som kräver mycket arbete för att lära sig.

Skrivandet av uppsatsen tog ungefär så lång tid som vi hade planerat. Vi la tyvärr för lite tid på arbetet i början. Det fick vi ta igen i slutet. Det kan ses som en erfarenhet i sig då vi förhoppningsvis har lärt oss att bättre planera ett större arbete. Om vi skulle göra samma sak igen, skulle vi antagligen förbereda oss mycket bättre innan vi startade att skriva.

Referenser

[1] Per Foreby. *Att skriva rapporter med LaTeX.* Lunds Tekniska Högskola ,24 nov 2003.

[2] Mark Deloura. *Game programming gems* Charles River Media, INC 2000.

[3] Kevin Hawkins *OpenGl game programming* Prima Publishing, 2001.

[4] Dante Treglia. *Game programming gems 3* Charles River Media 2002.

[5] Eric Lengyel. *Mathematics for 3d game programming & computer graphics* Charles River Media 2002

[6] Anders Vretblad. *Algebra och Geometri* Gleerups Förlag AB, 1999

[7] David H. Eberly. *3D Game Engine Design* Morgan Kaufmann,30 Sep 2000.

[8] André LaMothe. *Tricks of the 34 game programming gurus* Sams Publishing, 2003.

[9] John De Goes. *3D game programming with C++* Coriolis, 2000.

[10] Nationalencyklopedin. *Nationalencyklopedins Internettjänst* http://www.ne.se

[11] Royal Institute of Technnology. *Nästa generations CAD/CAE - nya möjligheter och nya krav* http://www.md.kth.se/ ulfs/Publications/artikel.pdf

[12] *Frustum Culling exempel* http://viz.aset.psu.edu/gho/sem_notes/color_3d/index.html/

[13] *View frustum och frustum culling* http://www.flipcode.com/articles/article_frustumculling.shtml

[14] *3D-Clipping for Realtime Graphics* http://www.cubic.org/šubmissive/sourcerer/3dclip.htm#ma2

[15] *spacesimulator.net* http://www.spacesimulator.net/tut1_3dengine.html

[16] *Camera Movement* http://gamecode.tripod.com/tut/tut03.htm

[17] *TestureMapping* http://www.geocities.com/siliconvalley/2151/tmap.html

[18] *TestureMapping2* http://www.gamedev.net/hosted/blackart/gldetut4.html

[19] *3DStudioMax* http://www.discreet.com/

[20] *Bitmap* http://www.tfe.umu.se/courses/systemteknik/Multimed1/00/Graphics/

[21] *Half Life 2 och Grand Turismo 4 bilder* http://www.gamespy.com/

[22] *Frustum culling bild* http://wwweic.eri.u-tokyo.ac.jp/computer/manual/lx/SGI_Developer/books /Perf_GetStarted/sgi_html/ch15.html

[23] *Occlusion Culling bild* http://www.fh-wedel.de/ ko/Galerie/2000-WS-Seminar/Boege/optimizer/index.html

[24] *Kooordinatsystem bilder* http://www.msu.edu/user/ionescua/pioneer/cpwk2_1b.htm

[25] *Kameravolym bild* http://viz.aset.psu.edu/gho/sem_notes/color_3d/html/rendering.html

[26] *Backface culling bild* http://student.seas.gwu.edu/ shailesh/lab0.html

[27] *Bounding Box bild* http://cis.k.hosei.ac.jp/ F-rep/Tut_HTML_j/a_little_bit_mathematics.html

[28] *Bounding Sfär bild* http://www.ifp.uni-stuttgart.de/lehre/diplomarbeiten/firchau/firchau.html

[29] *TextureMap bild* http://www.gamedev.net/reference/articles/article852.asp

A Definition av ett objekt

Koden visar hur ett objekt representerande en kub kan skapas i C

```
#define MAX_VER 1000           //Max antal punkter hos en polygon
#define MAX_POL 1000         //Max antal polygoner hos ett objekt

polygon pol[MAX_POL];        //En array som skall fyllas med objektets polygoner
vertex ver[MAX_VER];         //En array som skall fyllas med punkterna i polygonerna

/* Strukturen definierar en punkt som har tre värden x, y och
z. Värderna mostvarar de olika axlarna i världkoordinater */

typedef struct{
   float x,y,z;
}vertex;

// Strukturen defenierar en polygon som består de tre punkterna a, b och c.

typedef struct{
   int a,b,c;
}polygon;

/*Strukturen defenierar ett objekt som består av maximalt 1000 punkter
och maximalt 1000 polygoner. */

typedef struct{
   vertex ver[MAX_VERTICES];
```

```
   polygon pol[MAX_POLYGONS];
}obj,*obj_ptr;

/* Här ges de åtta punkterna i objekten ett x, y och z värde. Objektet
   är en kub i tre dimensioner */

obj object;
object.ver[0].x=0;  object.ver[0].y=0;  object.ver[0].z=0;
object.ver[1].x=1;  object.ver[1].y=0;  object.ver[1].z=0;
object.ver[2].x=1;  object.ver[2].y=0;  object.ver[2].z=1;
object.ver[3].x=0;  object.ver[3].y=0;  object.ver[3].z=1;
object.ver[4].x=0;  object.ver[4].y=1;  object.ver[4].z=0;
object.ver[5].x=1;  object.ver[5].y=1;  object.ver[5].z=0;
object.ver[6].x=1;  object.ver[6].y=1;  object.ver[6].z=1;
object.ver[7].x=0;  object.ver[7].y=1;  object.ver[7].z=1;

/*Eftersom polygonerna i objektet är defenierade med tre punkter kuben
delas upp i tranglar. Varje yta hos kuben består av en fyrkant. Det
existerar sex stycken ytor hos kuben, så objekten kommer att ha 12
stycken polygoner.*/


/*Koden nedan tilldelar varje polygon i objektet tre punkter. pol[0] exempelvis
  kommer att bestå av punkterna ver[0], ver[1] och ver[4]. */
object.pol[0].a=0;  object.pol[0].b=1;  object.pol[0].c=4;
object.pol[1].a=1;  object.pol[1].b=5;  object.pol[1].c=4;
object.pol[2].a=1;  object.pol[2].b=2;  object.pol[2].c=5;
```

```
object.pol[3].a=2;  object.pol[3].b=6;  object.pol[3].c=5;
object.pol[4].a=2;  object.pol[4].b=3;  object.pol[4].c=6;
object.pol[5].a=3;  object.pol[5].b=7;  object.pol[5].c=6;
object.pol[6].a=3;  object.pol[6].b=0;  object.pol[6].c=7;
object.pol[7].a=0;  object.pol[7].b=4;  object.pol[7].c=7;
object.pol[8].a=4;  object.pol[8].b=5;  object.pol[8].c=7;
object.pol[9].a=5;  object.pol[9].b=6;  object.pol[9].c=7;
object.pol[10].a=3; object.pol[10].b=2; object.pol[10].c=0;
object.pol[11].a=2; object.pol[11].b=1; object.pol[11].c=0;
```

B Värld till kameratransformering

Följande kod visar hur värld till kamera transformering kan ske. Koden är skriven i C++

```
//Strukturen definierar en punkt

typedef struct Vertex
{
    float x, y, z;  // Indikerar punktens position på x, y och z-axlarna

} VERTEX;

 /* Strukturen definierar en polygon.Strukturen kan spara upp till 16 polygoner.
    world är till för att spara
    polygonerna i världskoordinater och camera är till för att spara de
    i kamerakoordinater. Count innehåller antalet polygoner */

typedef struct Polygon
{
   VERTEX world[16], camera[16];
   int Count;

} POLYGON;

/*Strukturen definierar en kamera. x, y och z indikerar kamerans position
  i världen. Roll, Pitch och Yaw indikerar hur mycket är roterad kring
  koordinataxlarna. */
```

```
typedef struct Camera
{
   float x, y, z;
   float Roll, Pitch, Yaw;

} CAMERA;

/* Funktionen transformerar världskoordinaterna till kamerakoordinater med hjälp av
   rotationsmatriser som finns beskrivna i kapitel 2. */

int WorldToCamera(POLYGON *Polygon, CAMERA *Camera;)
{
   float x2, y2, z2;
   int i;

   for (i=0; i<Polygon->Count; i++)
   {
      Polygon->camera[i] = Polygon->world[i];

      x2 = Polygon->world[i].x - Camera->x;
      y2 = Polygon->world[i].y - Camera->y;
      z2 = Polygon->world[i].z - Camera->z;

      Polygon->camera[i].z = z2 * sin(Camera->Yaw) + x2 * cos(Camera->Yaw);
      Polygon->camera[i].y = y2;
      Polygon->camera[i].z = z2 * cos(Camera->Yaw) - x2 * sin(Camera->Yaw);
```

```
        x2 = Polygon->camera[i].x;
        y2 = Polygon->camera[i].y * cos(Camera->Pitch) - Polygon->camera[i].z
             * sin(Camera->Pitch);
        z2 = Polygon->camera[i].y * sin(Camera->Pitch) + Polygon->camera[i].z
             * cos(Camera->Pitch);

        Polygon->camera[i].x = y2 * sin(Camera->Roll) + x2 * cos(Camera->Roll);
        Polygon->camera[i].y = y2 * cos(Camera->Roll) - x2 * sin(Camera->Roll);
        Polygon->camera[i].z = z2;
    }
}
```

C Frustum Culling med gränssfär

```
// Koden nedan testar om gränssfären existerar inom kameravyn

const int NROFPLANES         //Antalet plan i kameravyn
struct planes;               //En struct innehållande planen
Sphere sphere;      //Sfären som testas

inte frustumCull()
{
float Distance;

for(plane = 0; plane < NROFPLANES; ++plane)
        {
//Koden nedan räknar ut avståndet från sfärens centrumpunkt till planet

Distance = planes[i].Normal().dotProduct(refSphere.Center()) + planes[i].Distance();

/* Om avståndet är mindre än sfärens radie befinner sig
                   sfären utanför kameravyn */

if(Distance < sphere.Radius())
return(OUTSIDE);

   /*Om Absolutbeloppet av skalärprodukten är mindre än
                     sfärens radin skärs kameravyn*/
```

```
if(absolut(Distance) < sphere.Radius())
return(INTERSECT);
}

// Om ingen av ovan stämmer är sfären innanför kameravyn

return(IN_FRUSTUM);
}
```

D Frustum Culling med gränsskub

```
// Koden nedan testar om gränskuben existerar inom kameravyn

const Integer NROFPLANES 6
const Integer NROFVERTEX 8
const Integer INFRUSTRUM 0
Vector KubeVertex[8];
Integer NrOfVertexInFrustum = 0;
Integer ToTal = 0;
Struct KubeModel;

int frustumCull()
{

   // Hämtar punkterna hos kuben
   Kube.GetVertices(KubeVertex);



   for(int plane = 0; plane < NROFPLANES; ++plane)
   {
      int iPtIn = 1;

      for(int i = 0; i < NROFVERTEX; ++i)
      {

      // Testar en punkt i taget mot det aktuella planet
```

```
         if(plane[plane].SideOfPlane(KubeVertex[i]) > INFRUSTRUM)
         {
            NrOfVertexInFrustum++;
 }
      }

      //Var alla punkterna i kuben utanför planet
      If(NrOfVertexInFrustum == 0)
         return(OUT);
      If(NrOfVertexInFrustum == 8)
         ToTal++
   }

/* Om alla punkterna var innanför alla 6 planen befinner kuben sig
            innanför kameravyn */

if(ToTal == 6)
return(IN);

// Annars räknas kuben som delvis inom kameravyn
return(INTERSECT);
}
```

E Hitta punkterna i en linje för Frustum clipping

```
//Följande exempel är skrivet i pseudokod.

Integer pos = 0;
Integer neg = 0;

for (Integer i equals 0; i smaller then V.length; i++)
{

   /* Om punkten är synlig befinner sig punkten på
      framsidan av det givna planet */

   if( V[i] is visible )
   {

      V[i].distance = Dotproduct(clipplanes normalvector ,V[i].point) - clipplane.c;
   }

   if ( V[i].distance is equal or greater then epsilon )
   {
      add pos by one;
   }

   else if ( V[i].distance is less then -epsilon )
   {
```

```
      add neg by one;

      V[i] is not visible;
   }

   else
   {
      // point on the plane within floating point tolerance
      V[i].distance = 0;
   }
}

if ( neg is equal to zero )
{
   // all vertices on nonnegative side, no clipping
   return +1;
}

if ( pos is equal to zero )
{
   // all vertices on nonpositive side, everything clipped
   return -1;
}
```