Inledning

Detta är en kort sammanfattning av teorimaterialet som år 2004 ingår i examinationen i kursen TNM077 3D-grafik och animering som ges vid Linköpings tekniska universitet på Campus Norrköping. Jag har under några få timmar sammanställt denna text och vill därför reservera mig för alla typer av fel som kan ha uppstått. Jag garanterar inte heller att hela kursen täcks av detta material – för det fullständiga teorimaterialet är det säkrast att läsa boken *3D Computer Graphics* (tredje upplagan, Addison-Wesley 2000) av Alan Watt. Trots dessa försiktiga ord hoppas jag att det kan vara en nyttig och intressant text som framförallt bidrar till en övergripande förståelse för centrala grundbegrepp inom 3D-grafik och animering. Trevlig läsning!

Bézierkurvor och parametriska objektrepresentationer

Det finns fördelar med att använda parametriska beskrivningar av objekten i en 3D-scen. För det första är det möjligt att representera verkliga objekt mer exakt än med polygonmodellen, som per definition är en kantig approximation. För det andra är det mer ekonomiskt rent utrymmesmässigt – en uppsättning polynom tar mindre plats att lagra än många tusen hörnpunkter. För det tredje finns det möjlighet att på ett relativt praktiskt sätt modifiera och finjustera parametriska ytor medan man ser dem i 3D, något som kan vara besvärligt med en polygonrepresentation där uppgiften blir att flytta enskilda polygoner eller till och med hörnpunkter.

Idén med parametriska kurvor och ytor kom till på 1960-talet då flera tillverkare inom bilindustrin sökte praktiska sätt att designa nya bilmodeller. Pierre Bézier var en matematiker som arbetade på Renault i frankrike, och han tog fram en smart kurvrepresentation som är intuitiv att arbeta med och som därför används än idag, om än med något modifierat användargränssnitt.

Principen för en Bézierkurva är att man har fyra kontrollpunkter som definierar den. Kurvan börjar i den första och slutar i den fjärde, medan punkterna två och tre avgör vilken form kurvan kommer att ha. Om man tänker sig ett streck mellan punkt ett och två så kommer detta sammanfalla med tangenten till kurvan i punkten ett. Motsvarande gäller för punkt två och fyra. Detta innebär att man genom att flytta kontrollpunkterna kan göra detaljerade justeringar på kurvans utseende, och gränssnittet är enkelt att använda. För att rita mer komplicerade kurvor sätter man ihop flera segment, där man antingen kan bestämma att det ska vara en kontinuerlig övergång mellan dem (höger- och vänsterderivator i en punkt som kopplar samman två segment är identiska) eller om övergången ska tillåtas vara diskontinuerlig.

För att åstadkomma detta beteende hos kurvan använder man sig av lika många polynom som man har kontrollpunkter, och man ser till att polynomen är av grad [antal kontrollpunkter – 1]. Kurvan fås sedan genom att man adderar ihop inverkan från de olika polynomen och punkterna. En Bézierkurva *Q(u)* beskrivs som:

$$\overline{Q}(u) = \sum_{i=0}^{3} \overline{P}_i B_i(u)$$

där

$$\overline{P}_i = kontrollpunkterna$$

$$B_0(u) = (1-u)^3$$

$$B_1(u) = 3u(1-u)^2$$

$$B_2(u) = 3u^2(1-u)$$

$$B_3(u) = u^3$$

$$u \in [0,1]$$

Det kanske är svårt att se styrkan i denna beskrivning, men om man tänker efter litet så finns det flera saker som är viktiga. Principen är att en punkt på kurvan, *Q(u),* bestäms av de kontrollpunkter P_i kurvan har, var och en skalad med en faktor B_i. Sedan kan man också se att förutom i ändpunkterna kommer samtliga kontrollpunkter påverka kurvans punkter eftersom alla faktorer B_i är skilda från noll om u ligger strikt *mellan* 0 och 1.

En annan egenskap som gör Bézierkurvorna så lätta att arbeta med är att kontrollpunkternas positioner inte bara bestämmer riktningen på kurvans tangentvektorer i ändpunkterna, utan även dessa vektorers magnitud (längd). Sambandet är inte helt självklart, men det ser ut som:

$$\overline{Q}(0) = 3(\overline{P}_1 - \overline{P}_0)$$

$$\overline{Q}(1) = 3(\overline{P}_2 - \overline{P}_3)$$

Förutom Bézierkurvor så finns det andra typer av parametriska kurvbeskrivningar med liknande egenskaper. För att utnyttja dessa i 3D-grafik bygger man samman flera kurvor till ytor, så kallade *patches*.

Rendering av polygonmodeller

Renderingsprocessen för polygonmodeller kan enkelt ses som ett led i grafikarbetet där listor över hörnpunkter och trianglar matas in och färgvärden för bildskärmens pixlar kommer ut. Detta sker ofta i två steg. Först appliceras olika transformationer på geometrin – modelltransformationer, vytransformationer och så vidare. Sedan följer ljusberäkningarna, allmänt kallat *shading*. Denna del av processen består i sin tur ofta av två steg – först en algoritm som beräknar skuggningen av varje pixel och sedan en algoritm som avgör vilka pixlar som inte syns eftersom de ligger bakom något. Vanliga skuggmetoder är *Gouraud shading* och *Phong shading* medan den mest populära algoritmen för att ta bort osynliga pixlar kallas *Z-buffer*.

Rendering av parametriska ytor

När man ska rendera parametriska ytor kan man till exempel approximera dem genom att generera polygonytor utifrån den matematiska beskrivningen. Det kan verka bakvänt att till slut använda polygoner i alla fall, men man har fortfarande kvar den matematiska beskrivningen i bakgrunden och har därför tillgång till dess fördelar.

Det är även möjligt att rendera direkt från den parametriska beskrivningen, men detta är ganska komplicerat. Här måste man på något sätt ta reda på bland annat siluettkurvor och maximum- och minimumvärden för de olika ytsegmenten.

”Level of detail” (LOD)

I vissa situationer är det intressant att minska detaljrikedomen hos ett objekt. Som exempel kan vi säga att ett objekt med 500 000 polygoner kommer att ta upp onödigt mycket datorkraft under renderingen om det befinner sig så långt bort från kameran att det bara tar upp kanske 25 pixlar på skärmen. I det fallet kan maximalt 25 individuella polygoner uppfattas men 500 000 polygoner kommer ändå att finnas med under t.ex. ljusberäkningarna.

Om man representerar sina objekt med polygoner kan man i förväg bestämma vilken objektets högsta polygonupplösning ska vara (mest detaljrikt) och sedan även lagra ett antal versioner av objektet med lägre upplösning (mindre detaljrika). När objektets position i förhållande till kameran sedan förändras avgör avståndet vilken detaljnivå som ska användas, eventuellt med en interpolering (mjuk övergång) mellan de olika nivåerna.

Om man istället använder matematiska funktioner för att representera objekt behöver man inte i förväg räkna ut olika detaljnivåer – vid varje tillfälle är det möjligt att få en lämplig modell eftersom funktionerna i sig inte har något upplösningsberoende.

Interpolativa shadingmetoder

Det finns två huvudmetoder för att beräkna intensiteten över en polygon utifrån data för hörnpunkterna: Gouraud- och Phong shading. Gouraud shading utgår från beräknade intensiteter i hörnpunkterna och interpolerar fram intensiteter för varje punkt av polygonen. Detta innebär att intensiteten mitt på en polygon aldrig kan vara starkare än i hörnpunkterna vilket i sin tur medför att speglande reflexioner kommer att se konstiga ut eller till och med försvinna om man har riktigt otur. Phong shading är smartare på det sättet att man utgår från uträknade *normaler* i hörnpunkterna och sedan interpolerar fram en normal för varje punkt på polygonen. På detta sätt kan intensiteten mitt på en polygon vara starkare än i hörnpunkterna och speglande reflexioner återges mer korrekt.

Vilken metod är bäst att använda? Gouraud shading är mindre beräkningstungt men fungerar bra endast för diffus reflexion. Phong shading är mer krävande men kan hantera speglande reflexioner. Det mest logiska skulle vara att kombinera båda metoderna – och detta är precis vad man ofta gör i praktiken. Detta kan man lösa genom att för varje objekt - beroende på objektets materialegenskaper – ange vilken reflexionsmodell som ska användas.

Phongs och Blinns reflexionsmodeller

Phong utvecklade en reflexionsmodell baserad på en diffus och en speglande del. Dessutom finns en term som lättar upp skuggorna litet och bidrar med ett ”allmänljus” i scenen. Reflexionsmodellen är empirisk – den gör stora förenklingar av motsvarande fenomen i verkligheten – men resultatet ser bra ut, även om man i vissa situationer kan uppfatta felaktigheter.

En sådan är att beräkningen av den speglande reflexionens färg och magnitud inte tar hänsyn till det inkommande ljusets riktning i förhållande till ytans tangentplan i den aktuella punkten. Blinn utökade modellen så att den skulle återge speglande reflexioner fysikaliskt korrekt genom att simulera ytans mikroskopiska ojämnheter. När man använder Blinns modell kan man simulera framförallt olika metallobjekt mer verklighetstroget.

Texturmappning

Om man bara använder sig av en enkel reflexionsmodell eller skuggningsteknik blir 3D-objekten ganska ointressanta att titta på. För att något ska se spännande eller realistiskt ut krävs detaljrikedom, men detaljerade objekt kräver många polygoner och blir snabbt jobbiga att rendera beräkningsmässigt. För att komma litet billigare undan använder man sig av olika sorters *texturer*, vilket ofta innebär någon form av bild som man ”klistrar fast” på objektet eller låter inverka på objektets ljusreflexionsegenskaper.

Grundformen av texturmappning används för att kontrollera färgen på ett objekts pixlar. För att göra detta tar man fram någon form av parametrisering av 3D-ytan och använder parametriseringens variabler för att hämta motsvarande värden på en tvådimensionell bild som kan vara till exempel ett fotografi. När

man hittat en pixels motsvarighet på texturbilden tar man helt enkelt den färg som bilden har just där och sätter den på 3D-objektets pixel.

Själva parametriseringen av ytan är ett icke-trivialt steg och kommer att påverka det slutgiltiga resultatet. Det finns flera lösningar som används idag; en av dem innebär att man vid modellerandet av objektet associerar hörnpunkterna till vissa koordinatvärden på texturbilden varpå renderingsprocessen interpolerar däremellan för att få texturkoordinater svarande mot mellanliggande punkter på modellen. En annan strategi är att först mappa texturbilden på ett (osynligt) 3D-objekt vars yta är lätt att parametrisera – till exempel en sfär eller kub. Sedan tänker man sig att ens modellerade objekt ligger inuti detta mellanobjekt och att man hämtar färgvärden från texturen på mellanobjektet genom att till exempel följa objektets ytnormal i varje punkt. Detta är i sig en ganska bra metod, men valet av mellanobjekt för texturen måste ofta styras av 3D-objektets form eller texturens beskaffenhet.

En annan form av texturmappning som kan spara in många polygoner är så kallad *bump mapping*. Detta går ut på att man har en texturbild som genom att variera intensitet över sina bildpunkter beskriver hur en ytas normaler ska varieras i riktning, vilket efter applicering kommer att resultera i att ytan kan se ojämn och mer realistisk ut. Där man normalt hade varit tvungen att införa fler polygoner har man nu istället använt sig av en enkel tvådimensionell bild för att beskriva ytans ojämnheter.

I realtidsgrafik ställer man höga krav både på att miljöerna ska se realistiska ut och att det hela ska gå snabbt att rendera – ibland måste över hundra bilder per sekund kunna visas för att användaren inte ska uppfatta störande ”hack” vid snabba kamerarörelser. Ljus bidrar i hög grad till en scens upplevda realism men är samtidigt oftast en krävande process – en enda scen kan ibland ta många timmar att rendera om man vill ha en verklighetstrogen ljussimulering. Ändå är det möjligt att använda verklighetstrogna ljusberäkningsmetoder i realtidsgrafik så länge ljuskällorna inte flyttar på sig. För att åstadkomma detta utför man ljusberäkningarna i förväg, sparar resultatet som speciella bilder, så kallade *light maps*, som sedan kan mappas till objekten i realtid. På det sättet sparar man in kostsamma beräkningar för diffust ljus och kan (i huvudsak) använda sig av lokala reflexionsmodeller under simuleringens gång.

Perfekta speglande reflexioner går att approximera genom en relativt billig metod som kallas *environment mapping* (reflexionsmappar på svenska). Denna går ut på att man placerar en ”kamera” inuti objektet som ska vara speglande, låter kameran ta (rendera) bilder i alla riktningar motsvarande till exempel sidorna på en kub som man föreställer sig omsluta objektet – dessa bilder mappas sedan ned på det speglande objektet som vilken textur som helst. Detta går snabbare än att använda en mer fysikaliskt korrekt teknik som exempelvis *ray tracing*. Dock måste nya reflexionsmappar tas fram när något förändras i scenen.

En väldigt användbar form av textur är en så kallad *procedurell* textur – det vill säga en (oftast) matematisk modell som beskriver texturens pixelvärden. En sådan har inget upplösningsberoende och kan ofta vara tredimensionell om så behövs.

Om vi tänker oss att ett objekt är på ett sådant avstånd från kamerapunkten att en punkt på objektet motsvarar en mängd pixlar i texturbilden – hur ska vi då veta vilket texturvärde objektets punkt ska få? Om vi helt enkelt tar den texturpixel som ligger mitt i det täckta området kan lustiga mönster uppstå, särskilt om texturen upprepas periodiskt. Effekten är en form av *aliasing* och uppstår på grund av vårt naiva val av texturpixel – vi har gjort en *undersampling*. Det finns olika metoder för att utföra så kallad *anti-aliasing*, det vill säga undvika att aliasing framträder. En väldigt vanlig metod när det gäller anti-aliasing i texturmappning är att använda *mip-mapping*. Detta går ut på att man i förväg har lagrat varje texturbild i en högupplöst version tillsammans med mindre och mindre versioner – i varje steg minskar man bildens sidlängd med hälften. När texturmappningen sedan pågår i full fart kan hela tiden en texturbild i rimlig upplösning väljas beroende på objektets position i förhållande till kameran, och man har eliminerat problemet med aliasing.

Geometriska skuggor

Skuggor är mycket viktiga för realismen och känslan i en scen. I verkligheten kan skuggornas natur variera – de kan ha skarpa eller suddiga kanter och de kan ha både *umbra* (område som är helt skymt från ljuskällan) och *penumbra* (område som får litet ljus från ljuskällan). Hur en viss skugga ser ut beror på hur objektet ser ut, atmosfären runt omkring och hur objektet förhåller sig till ljuskällan, samt ljuskällans natur. Inom datorgrafik använder man sig ofta av förenklingar helt enkelt eftersom det är svårt att på ett korrekt sätt simulera dessa företeelser. Ofta antar man att ljuskällorna är punkter, vilket i praktiken innebär att skuggorna får skarpa kanter. En mycket simpel skuggalgoritm innebär att man projicerar det belysta objektet på planet det står på med (punkt-)ljuskällan som projiceringscentrum.

En populär teknik för skuggning använder mappnings- och z-buffertekniker. Man placerar kameran i de olika ljuskällorna i tur och ordning, så att man ser scenen från deras synvinkel. Sedan genererar man skuggmappar (*shadow maps)* som visar djupvärden för de pixlar som syns från ljuskällan. Sedan renderar man scenen från den ”vanliga” kameran med en z-bufferalgoritm. Om en viss punkt är synlig så transformerar man punktens koordinater till ljuskällans koordinatsystem. Där jämför man djupvärdet för punkten med det lagrade djupvärdet – om punktens värde är större än det lagrade i skuggmappen betyder det att punkten ligger längre bort än den yta som ljuskällan ”ser” och är därför i skugga.

På samma sätt som med texturmappning (och av precis samma orsaker) kan det uppstå aliasing i skuggor när man använder skuggmappar. Det har alltså att göra med att den pixel (på skärmen) som man vill undersöka huruvida den ligger i skugga eller ej kan motsvara många pixlar i skuggmappen, och därför är det möjligt att en del av området den täcker ligger i skugga och en del utanför.

Globala reflexionsmodeller

Gouraud- och Phong shading är exempel på lokala reflexionsmodeller, det vill säga modeller som bara tar hänsyn till ett objekts interaktion med en eller flera ljuskällor. I verkligheten reflekterar objekt ljus mot varandra på olika sätt, och detta försöker man i datorgrafiken efterlikna med så kallade *globala* reflexionsmodeller.

Det finnas endast ett begränsat antal olika typer av ljusinteraktioner som kan förekomma mellan objekt. Dessa är *diffus till diffus, diffus till speglande, speglande till diffus* och *speglande till speglande*. Det kanske inte verkar vara så komplicerat vid en första anblick, men det visar sig att det är väldigt svårt att i en och samma modell ta hänsyn till alla typer av reflexioner.

Det finns idag två stora globala reflexionsmodeller: ray tracing och radiosity. Den förstnämnda kan endast behandla reflexioner av typen speglande till speglande och den sistnämnda endast diffus till diffus.

Ray tracing

Ray tracing är precis vad det låter som – man spårar tunna ljusstrålar för att se hur de reflekteras och bryts när de går igenom scenen. Det finns olika strategier för i vilken ände man ska börja och hur det ska gå till allmänt, men en av de vanligaste kallas *Whitted ray tracing* eller *inverse ray tracing*. Denna går ut på att man följer ljusstrålar som utgår från ”ögat” och ut i scenen, vilket är rimligt eftersom man hur som helst bara är intresserad av de strålar som når ögat – de andra har ju ingen synlig inverkan på den bild man ser.

Whitted ray tracing innehåller en lokal reflexionsmodell för simulering av diffusa reflexioner (och delvis även speglande) eftersom den globalt bara kan hantera speglande reflexioner. Metoden fungerar så att man ”skjuter” strålar från kamerapunkten ut i scenen. När strålarna träffar ett objekt av något slag skjuts två nya strålar ut från träffpunkten – en reflekterad och en transmitterad (bruten). Var och en av dessa behandlas sedan rekursivt på samma sätt. Rekursionen avslutas när strålen träffar en helt diffus yta, när den lämnar scenen helt eller när dess energi har passerat ett visst tröskelvärde. Förutom de två fortsättande strålarna skickas en tredje stråle direkt till ljuskällan. Denna används för att undersöka om träffpunkten ligger i skugga eller i direkt ljus. Om ljuskällan belyser punkten direkt används den lokala reflexionsmodellen för att räkna ut en diffus komponent samt en speglande komponent som adderas till resultatet av de två andra strålarna. Denna lokala speglande komponent finns med för att man ska kunna simulera ”highlights” – det vill säga speglingen av själva ljuskällan i objektets yta. Det skulle inte vara möjligt med ”ren” ray tracing eftersom ljuskällan antas vara en punkt; chansen att en stråle träffar exakt den punkten är obefintlig och även om det skulle inträffa skulle highlight:en bli orealistiskt liten.

En annan version av samma metod är så kallad *stokastisk ray tracing*. Skillnaden är att den simulerar icke-perfekta speglingar genom att för varje träffpunkt skicka ut ett antal strålar i slumpmässiga riktningar, där slumpgenereringen styrs av materialets egenskaper. På detta sätt blir såväl spegelbilder som skuggor suddigare vilket för många material höjer realismen.

Ytterligare en variant av ray tracing kallas på engelska *two-pass ray tracing* eller *forward ray tracing* och bygger på att man först följer ljusstrålar från ljuskällan tills man träffar en diffus yta, varpå man lagrar intensiteten för träffpunkten. Sedan skjuter man som vanligt strålar från kamerapunkten och när dessa eventuellt träffar diffusa ytor så används de tidigare beräknade intensitetsvärdena. Denna metod är kapabel att simulera till exempel ljusspel som uppstår på en diffus yta när den belyses genom glasobjekt eller liknande (så kallade *caustics* på engelska).

Rent teoretiskt finns det förstås inget som hindrar att man inför diffus-diffus-reflexioner i ray tracing, men detta skulle praktiskt innebära att man vid varje träff med en diffus yta skickar ut en stor mängd strålar i alla möjliga riktningar och detta blir snabbt för tungt rent beräkningsmässigt.

Radiosity

Radiosity är en metod som i sin grundform koncentrerar sig helt på diffus-diffus-reflexioner, vilket ofta är den dominerande typen av ljusinteraktioner mellan objekt i en scen. En stor fördel med metoden är att den är oberoende av var kameran är placerad – när man väl utfört beräkningarna en gång kan man flytta runt kameran i miljön utan att behöva räkna om något. Detta gäller naturligtvis bara så länge själva objekten och ljuskällorna förblir stilla.

Radiosity bygger på att scenen delas upp i en ny uppsättning polygoner skapade utifrån ljusets variationer och detaljrikedom i olika delar av scenen. Detta är inte något enkelt problem att lösa – hur ska man kunna veta var ljuset måste vara mest detaljerat innan man beräknat ljuset?

I radiositymetoden börjar man med att låta alla ”ljuspolygoner” ”stråla ut” det ljus de ska reflektera, med start i själva ljuskällorna. Sedan fortsätter man iterativt (stegvis) genom att låta alla ljuspolygoner återigen reflektera det infallande ljuset, ända tills i princip allt ljus har fördelats i scenen. Eftersom man bara behandlar diffusa ytor absorberas ljus i varje steg, så förr eller senare har ljuset ”spridits färdigt”. Hur mycket en viss polygon belyser en annan beräknas via en så kallad *formfaktor* som baseras på polygonernas geometriska förhållande till varandra. Detta är en krävande del av processen. Ljusutsändningen B_i från en viss ljuspolygon *i* kan beräknas via ekvationen:

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

där

$E_i = \textit{självemitt ans}$

$R_i = \textit{reflektans}$

$B_j = \textit{polygonen } j\textit{ : s ljusutsändning}$

$F_{ij} = \textit{formfaktorn mellan polygon i och } j$

Med ord kan man alltså säga att mängden ljus som en viss ljuspolygon i scenen sänder ut är en summering av hur mycket själva polygonen lyser (som en ljuskälla) och hur mycket av det infallande ljuset från alla andra ljuspolygoner som ska reflekteras (beroende på materialets egenskaper).

Radiositymetoden producerar fina resultat men tar lång tid att beräkna. För att den ska bli mer praktisk att använda kan man låta den utföras i flera steg så att det är möjligt att se hur bilden byggs upp under hela processen. På det sättet kan man tidigt se om något har blivit helt fel redan från början och slipper därför vänta en längre stund på ett resultat man inte är nöjd med.

Anti-aliasing

Inom datorgrafik menar man med *aliasing* oftast de oönskade visuella fenomen som kan uppstår i samband med *undersampling* i något led av renderingen, till exempel vid texturmappningen. Litet förenklat kan man säga att när en pixel på bildskärmen motsvarar många pixlar i den virtuella scenen (på grund av vinkel och avstånd till exempel) så krävs det att man på något sätt räknar ut hur de innefattade punkterna i scenen tillsammans ska representeras på den enda skärmpixeln. Om man inte lägger någon energi vid detta steg kan onaturliga mönster uppstå eftersom man helt enkelt har samplat scenen med för liten noggrannhet – undersampling.

Den vanligaste strategin för att undvika aliasing, med andra ord *anti-aliasing*, innebär att man istället för att rendera direkt på en yta motsvarande skärmupplösningen renderar till en yta som är ett visst antal gånger större, varpå man skalar ner resultatet till rätt storlek via någon form av transformering. Detta innebär förstås att man utför mer arbete för varje pixel, men resultatet blir mer visuellt korrekt.

En smart variant av sådan här *supersampling* innebär att man istället för att göra ett stort antal samplingar med jämna intervall för varje pixel gör ett mindre antal ojämnt fördelade sampel över motsvarande yta. Hur man fördelar dem varierar från metod till metod. Slumpfördelning är ett sätt medan ett annat tar hänsyn till områdets komplexitet genom att först studera en lågupplöst bild och sedan sampla tätare där det behövs. Båda dessa tillvägagångssätt är exempel på så kallad icke-uniform sampling och resulterar oftast i väldigt bra bilder i förhållande till beräkningskraften som behövs.

Animering

Det finns olika metoder för att framställa 3D-animationer inom datorgrafik. Valet av metod beror helt på vad det är man vill åstadkomma – samtliga har fördelar såväl som begränsningar. Det är vanligt att man i en och samma animation har blandat olika tekniker för att kunna ta hänsyn till och bäst lösa olika situationer som man vill simulera.

Vid stelkroppsanimation har animeraren maximal kontroll över vad som sker. Denne definierar *keyframes* som beskriver scenen vid lämpliga tidpunkter längs tidsaxeln och mjukvaran hjälper till genom att på ett situationsbaserat sätt interpolera mellan dem för att skapa de mellanliggande bildrutorna. Som man kan

ana innebär den här tekniken stora möjligheter för artistisk kontroll samtidigt som mycket arbete måste läggas ned. Den lämpas kanske bäst för enkla, precisa rörelser. Att exempelvis försöka animera en studsande boll med hjälp av keyframes och interpolation är väldigt svårt – förmodligen behövs keyframes med väldigt täta tidsintervall eftersom det inte går att simulera fritt fall med någon enkel interpolationsteknik.

Med en hierarkisk modell av ett större objekt kan man åstadkomma leder liknande dem hos människor och djur, vilket kan vara väldigt praktiskt vid animering. Man slipper då flytta varje del av objektet för sig, eftersom olika transformationer man utför på ett visst delobjekt automatiskt appliceras på alla delobjekt som ligger längre ned i strukturen – till exempel påverkas smalben och fot när man rör på låret hos en modell av en människa.

Om man vill slippa att själv göra alla små rörelser i strukturen (detta kallas *forward* kinematics) kan man använda sig av *inverse kinematics*. Detta innebär att man talar om för mjukvaran vilket resultat man ska ha, varpå den räknar ut vilka krafter som behövs appliceras på de olika lederna för att åstadkomma den eftertraktade rörelsen. Detta kan även utnyttjas då man inte har hierarkiska strukturer för modellerna, vid så kallad *dynamisk simulering*.

Dynamisk simulering är precis vad det låter som – en simulering av en fysikaliskt (rimligt) korrekt miljö där objekten utsätts för krafter och har grundläggande attribut som exempelvis massa. Vid en vanlig simulering definierar man startpositioner för objekten och inför de krafter som ska gälla, och låter sedan datorn räkna ut vad som kommer hända. Detta passar utmärkt för scener där saker och ting ska ramla, rulla eller kastas omkring men innebär samtidigt att animeraren inte har särskilt stor kontroll. Den inverterade varianten är att man förklarar för mjukvaran vad det är man vill åstadkomma och låter den räkna ut vilka krafter som behövs för att man ska få rätt resultat.

I de flesta dynamiska simuleringar ingår ett komplicerat moment som handlar om att räkna ut om och när objekt kolliderar med varandra samt vad som ska hända när det inträffar. En väldigt simpel men otroligt beräkningstung lösning är att i varje ögonblick för varje polygon testa om den aktuella polygonen korsar en annan polygon i scenen. För att bespara sig en stor mängd beräkningar inför man oftast så kallade *bounding volumes*, vilket är tänkta, enkla geometriska objekt som till exempel sfärer som får omsluta objekten i scenen. Dessa testas mot varandra hela tiden, vilket är betydligt lättare än att titta på enskilda polygoner i varje ögonblick. Om en bounding volume inte kolliderar med någon annan så kan inte heller dess omslutna objekt kollidera med ett annat objekt. Om volymen däremot skär in i en annan så kan det inträffa, och då blir man tvungen att gå över till att testa polygon för polygon.

En ytterligare gren inom 3D-animering är *partikelanimering* där man ställer in attribut (såsom massa, huvudsaklig rörelseriktning och livslängd) för en stor mängd partiklar och sedan låter dem genomgå någon slags dynamisk simulering. Detta är en bra teknik om man vill animera företeelser såsom snö, regn eller liknande.

Inom många typer av animering går det att använda sig av så kallade *script*, där man i förväg bestämmer hur objekt ska röra och bete sig. Det kan handla om allt från enkla kurvor de ska följa (fritt fall är ett exempel) till avancerade ”sociala” beteendemönster där objekten rör sig med hänsyn tagen till andra objekt. På detta sätt kan man till exempel simulera flockbeteende hos modellerade djur utan att behöva styra varje ”individ” för sig.

När man vill animera verklighetstrogna mänskliga rörelser är tekniken *motion capture* väldigt användbar. Den går ut på att man fäster ett antal ”kontrollpunkter” på en speciell overall som en skådespelare får ta på sig. Sedan spelar man in kontrollpunkternas förflyttningar när skådespelaren utför de rörelser man vill animera, och överför dessa till motsvarande kontrollpunkter på sin 3D-modell i datorn. Det förekommer även enklare former av motion capture där man styr en modell med vardagliga instrument som till exempel datormusen. Gemensamt för de olika metoderna är att man kan skapa ”levande” rörelser utan att behöva definiera varje bildruta för sig.

Generellt gäller inom animering att ju högre grad av kontroll animeraren behöver ha – desto mer arbete måste denne lägga ner. Om man använder en dynamisk simulering eller ett beteendescript får man ett resultat som i och för sig kan se väldigt realistiskt ut, men till priset av att det inte går att styra i någon större utsträckning. Denna balansgång är anledningen till att man ofta använder sig av flera olika metoder inom samma projekt, beroende på vilka situationer man vill animera.