Laboration - Shaders

Martin Fitger, d00-mfi@d.kth.se

Version 1.4

Syfte

Att erbjuda studenterna möjlighet att få lära sig om olika shaderkoncept och renderingsalgoritmer samt att få prova på en visuell metod för utveckling av shaders.

Genomförande

Genomförs individuellt eller i grupp om 2 personer. Notera att labben måste göras på PC (se systemkrav). Det rekommenderas starkt att labben görs i någon av de förberedda salarna. Detta eftersom att det finns risk för att det kan förekomma variationer mellan olika grafikkortdrivrutiner som gör att vissa uppgifter i labben inte kommer att kunna kompileras, om man t ex kör på egen dator. Sal *Magenta* och *Vit* är förberedda för laborationen. Redovisning sker genom att visa en handledare avd man gjort.

Uppskattad tidsåtgång

Laboration, ca 2h

Uppgift

Ni ska prova på att skapa olika shaders med hjälp av ett grafiskt utvecklingsverktyg.

Systemkrav

- Windows XP/Vista
- OpenGL-kompatibelt grafikkort med stöd för minst ps 2.0.

DH2640/DH2323/NA8740, VT 2009
Laboration, Shaders

3D-rendering

Rendering av 3D-grafik är en process som tar en tredimensionell beskrivning av en scen och genererar en tvådimensionell bild utifrån denna som sedan kan visas på en skärm. Processen brukar kallas för *3D Rendering Pipeline* och består av en kedja av olika beräkningssteg som måste genomgås för att generera den slutliga bilden.

Till en början implementerade man hela denna pipeline i mjukvara, dvs. alla beräkningar utfördes av datorns centrala processor. Fördelen med denna typ av renderingssystem är att man har full kontroll över de operationer som skall utföras vid renderingen, men nackdelen är att dessa system har låg prestanda eftersom den centrala processorn inte är väl anpassad för att på ett effektivt sätt hantera de olika beräkningsstegen i kedjan. Många av beräkningsstegen under renderingsprocessen karakteriseras exempelvis av hög parallellitet, dvs. det är möjligt att utföra många av stegen parallellt. Koordinat-transformationer som måste utföras för varje hörn i geometrin som skall renderas kan exempelvis utföras oberoende av övriga hörn i geometrin, vilket gör att flera hörn skulle kunna processas samtidigt. Det samma gäller för beräkningar som utförs för varje pixel (bildelement) som skall renderas i en polygon - dessa beräkningar kan utföras oberoende av övriga pixlar i polygonen så det skulle vara möjligt att utföra beräkningar för flera pixlar samtidigt. Dessutom är många av de operationer som behöver utföras under renderingsprocessen ofta olika typer av vektoroperationer som kryss- och skalärprodukt, matrismultiplikation etc. vilka inte finns implementerade i centrala processorn utan måste implementeras med hjälp av flera atomära operationer.

Prestanda är speciellt viktig för system som bygger på realtidsrendering av 3D-grafik (t ex datorspel och andra interaktiva applikationer). Bilder behöver minst genereras med en hastighet av omkring 15 bilder per sekund för att de skall upplevas som rörliga, och man tvingas därför anpassa kvalitén av de bilder som renderas för att datorkapaciteten skall räcka till för att rendera bilder med en tillräcklig hastighet.

Efterfrågan på interaktiva applikationer är väldigt hög, framförallt inom datorspelsindustrin. Detta har lett till att man utvecklat speciell hårdvara som kan utföra delar an renderingsprocessen på ett effektivare sätt.

Hårdvara för rendering av 3D-grafik

I början av 90-talet blev hårdvaruaccelererad rendering av 3D-grafik för fösta gången tillgänglig på konsumentnivå i form av en ny typ av grafikkort. Specialiserad hårdvara för rendering av 3D-grafik hade funnits tidigare, men hade varir alldeles för dyr för att kunna användas för hemmabruk.

Syftet med renderingshårdvaran är att avlasta CPUn genom att utföra delar av renderingskedjan, medan CPUn kan fortsätta arbeta med andra uppgifter. Hårdvaran har stöd för de vanligaste vektor- och matrisoperationer som utförs under renderingsprocessen. Den utnyttjar också möjligheten för parallellisering av de olika stegen i renderingskedjan genom att ha flera beräkningsenheter som kan hantera flera operationer parallellt. Detta gör att hårdvaran kan utföra renderingsoperationer betydligt effektivare än den centrala processorn.

Hårdvaruaccelererad rendering innebar ett stort genombrott för datorgrafiken - eran av 3D-grafikapplikationer som rena mjukvarulösningar var ett minne blott och realtidsapplikationer och spel med 3D-grafik hade blivit en verklighet. Problemet med första generationerna av renderingshårdvaran var dock att dess funktionalitet fortfarande var väldigt begränsad. Hårdvaran erbjöd endast ett fåtal ”hårdkodade” algoritmer med tillhörande parametrar, vilket gjorde att priset för att använda denna nya snabbare teknologi var att man inte längre hade den flexibilitet som man haft i tidigare mjukvarulösningar. Allt eftersom tekniken utvecklades kunde hårdvaran erbjuda ett bredare urval av algoritmer och möjligheter för att kombinera dessa algoritmer för att åstadkomma mer avancerade effekter. Fortfarande byggde lösningen dock på ett begränsat urval av ”hårdkodade” algoritmer och belysningsmodeller vilket lämnade väldigt lite utrymme för kreativitet.

Ett område inom datorgrafiken som låg betydligt längre fram i utvecklingen mot realism var filmbranschen, där realtidsrendering inte var ett behov. *Pixar Animation Studios* hade sedan lång tid tillbaka utvecklat ett språk för beskrivning av grafikoperationer som de kallade *RenderMan*. Syftet var att ge grafiker full kontroll över renderingsresultat genom att använda ett enkelt men kraftfullt programmeringsspråk. *RenderMan* gör det möjligt att beskriva högkvalitativa fotorealistiska och icke-fotorealistiska effekter och har använts i produktionen av en rad datoranimerade filmer som t ex *Toy Story*. Dessa system byggde dock på mjukvaruimplementering av renderingsprocessen, eftersom behovet av prestanda inte var den samma som vid realtidsrendering.

Nästa stora steg i utvecklingen var en naturlig kombination av de bästa av två världar genom introduktionen av programmerbar hårdvara för rendering av 3D-grafik vid millennieskiftet. Det blev äntligen möjligt att rendera 3D-grafik i realtid med en kvalité och flexibilitet som började kunna jämföras med den hos filmindustrin. Den tidigare fixa strukturen hos hårdvaran ersattes i den programmerbara hårdvaran med en ny struktur som bygger på programmerbara processorer som kan exekvera en speciell typ av program som kallas för *shaders*. Detta gör att det idag är möjligt för en programmerare att skapa godtyckliga effekter genom att skriva dessa shaderprogram, och sedan låta dessa exekveras av hårdvaran, istället för att tvingas välja bland ett begränsat antal fixa algoritmer som tidigare.

Shaders

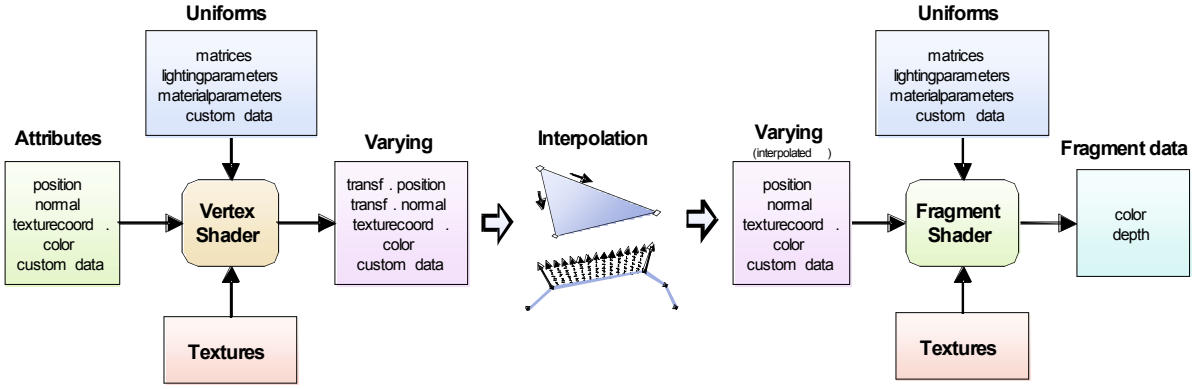
Shader är en speciell typ av program som används för att skräddarsy de operationer som dagens grafikkort skall utföra. Dessa program tar vid efter det att en tredimensionell beskrivning av en geometri som skall renderas har skickats till hårdvaran för rendering. Uppgiften för en shader är härifrån sedan att transformera och projicera denna geometri på ett tvådimensionellt plan så att den går att översätta till tvådimensionella skärmkoordinater, samt att sedan bestämma den resulterande färgen för varje bildelement på skärmen som täcks av denna geometri. En shader består av flera program (en *vertex shader*, en *fragment shader* och eventuellt även en *geometry shader*) som tar hand om olika delar av denna process.

Vertex shader är en speciell typ av shaderprogram som exekveras en gång för varje hörn i den geometri som skall renderas. Input till dessa program är dels det data som finns lagrat för det hörn som programmet behandlar (t ex hörnets position, dess normalvektor, texturkoordinater mm.) – sk *vertex attributes*. Programmet har även tillgång till variabler som renderingsapplikationen kan passa till detta program och som kan innehålla sceninformation som t ex belysningsparametrar, transformationsmatriser eller olika materialparametrar som shadern kan behöva för att den skall kunna utföra sitt arbete – sk *uniform variables*. Det är upp till programmeraren att ange vilka operationer som skall utföras av en vertex shader, men vanliga operationer som utföras med denna typ av program är transformation av koordinaten och normalvektorn för hörnet, belysningsberäkningar mm. För varje hörn som vertex shadern behandlar genererar den ett antal output-variabler – sk *varying variables*. Programmeraren anger själv vilka sådana variabler en vertex shader skall generera.

Efter att hörnen processats av en vertex shader kommer de grupperas ihop för att forma primitiver (linjer eller trianglar) som sedan skall renderas. De *varying*-variabler som vertex shadern genererat för varje hörn som ingår i primitiven kommer sedan att interpoleras över primitivens yta så att varje pixel som primitiven täcker på skärmen kommer att få en egen interpolerad uppsättning av dessa variabler (se Figur 1). Var och en av dessa pixlar kommer sedan behandlas av en annan typ av shaderprogram som kallas för *fragment shader*.

Fragment shader är en typ av program som exekveras en gång för varje bildelement (pixel) som renderas. Input till dessa program är dels resultatet från interpoleringen av de *varying*-variabler som vertex shadern genererat för varje hörn i primitiven som pixel är en del av. *Varying*-variabler används på så sätt för att kommunicera information från en vertex shader till en fragment shader. Liksom vertex shaders har fragment shaders också tillgång till sk *uniform variables* - värden som renderingsapplikationen kan passa till detta program. Det är upp till programmeraren att ange vilka operationer som skall utföras av en fragment shader, men vanliga operationer som utföras med denna typ av program är färgblandning, läsning av

färgvärden från texturer, belysningsberäkningar mm. Målet för en fragment shader är att bestämma den färg som slutligen skall appliceras för den pixel den behandlar.



Figur 1 *Dataflöde mellan olika shaderprogram.*

En shader kan exempelvis genomföra belysningsberäkningar för varje hörn i geometrin (med en vertex shader), låta den beräknade ljusintensiteten interpoleras (som en *varying*-variabel) över pixlarna i varje primitiv (linje, triangel) och slutligen använda denna interpolerade ljusintensitet för att bestämma färgen för varje pixel (i en fragment shader). Ett annat alternativ vore att låta vår vertex shader passa vidare sitt hörns normalvektor som en *varying*-variabel, låta denna vektor interpoleras över primitiven och sedan använda sig av denna interpolerade normalvektor för att utföra belysningsberäkninar i varje pixel som renderas (med en fragment shader). Det första alternativet kallas för *Gouraud shading* och det senare för *Phong shading*. Gouraud shading kräver inte lika mycket kapacitet eftersom belysningsberäkningarna endast utförs en gång för varje hörn, medan Phong shading utför belysningsberäkningarna en gång för varje pixel som renderas (vilka oftast är fler än antalet hörn i geometrin). Phong shading ger dock en bild med högre kvalitet än Gouraud shading.

Modern hårdvara för rendering av 3D-grafik stödjer idag oftast även en tredje typ av shaderprogram som kallas för *Geometry shaders*. Denna typ av program exekveras en gång för varje primitiv (linje, triangel) som skall renderas, och körs direkt efter att hörn har grupperats ihop för att forma en primitiv (alltså efter vertex shaders men för fragment shaders). Med dessa program är det möjligt att antingen förkasta primitiven, passa den vidare, eller till och med generera flera nya primitiver.

Utveckling av shaders

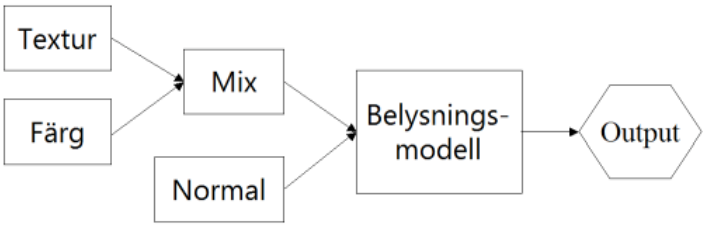
Till en början var man tvungen att skriva shaderprogram i ett för hårdvaran specifikt assemblerspråk. Processen var oftast tidsödande, krävde stor kunskap om den underliggande hårdvaran och debugging var näst intill en omöjlighet. Detta har med tiden förändrats i och med introduktionen av ett flertal högnivåspråk för shaderutveckling som *Cg* (*C for Graphics*, utvecklat av *Nvidia*), *HLSL* (*High Level Shading Language*, utvecklat av *Microsoft*) och *GLSL* (*OpenGL Shading Language*, utvecklat av *OpenGL Architecture Review Board*). Dessa språk påminner mycket om *C*, med några extra datatyper för lagring av vektorer, matriser etc. Dessa språk exponerar även en mängd hårdvaruaccelererade komplexa operationer i form av inbyggda funktioner i språket (vektor- och matrismultiplikation, kryssprodukt etc.), samt diverse ytterligare renderingsspecifika operationer som läsning från texturer mm.

Trots de stora framsteg som högnivåspråken för shaderprogrammering gjort de senaste åren har tekniken fortfarande ett flertal begränsningar. Shaders har en stor betydelse för det slutliga resultatet av det material som en grafiker skapar, vilket gör att det vore naturligt att ge grafikerna stort inflytande över designen av

shaders. För att skriva en shader krävs dock en relativt insatt programmerare vilket gör att grafiker får svårt att få utlopp för sin kreativitet då de blir helt i händerna på programmerare.

För att lösa dessa problem forskas det idag om enklare metoder för utveckling av shaders. Det finns t ex verktyg för att exportera Maya/3DSMax-material direkt som shaders. En annan teknik som kan ge mer kontroll över utvecklingen av shaders, men som fortfarande undviker komplexiteten med traditionell textuell programmering, är visuell programmering.

Shaderprogrammen består ofta av en mängd relativt välbegränsade delmoment, där resultat av ett delmoment används som indata till ett annat osv. Det faller sig därför relativt naturligt att beskriva shaders med hjälp av en graf där noder representerar olika typer av delmoment eller operationer och kanter representerar dataflöden dem emellan (se Figur 2). Denna form av visuell programmering har länge använts inom en rad olika programmeringsområden och visar sig passa bra även för shaderutveckling.



Figur 2 *Shadergraf*

Ett utvecklingsverktyg baserat på denna teknik kan exempelvis tillhandahålla ett grafiskt gränssnitt där användaren kan plocka olika komponenter från ett färdigt bibliotek och dra kopplingar mellan dem för att ange dataflöden, för att på så sätt beskriva det arbete som skall utföras av en shader. Man kan tänka sig olika varianter av dessa system med olika hög grad av abstraktion, antingen system där komponenter har en 1-1 relation till verkliga atomära operationer i programmeringsspråket eller mer abstrakta system med mer konceptuella komponenter.

Denna teknik har en hel del fördelar. Det kommer inte längre behövas en insatt programmerare för att utveckla shaders, utan grafiker kan nu få kontroll över hela utvecklingsprocessen. Programmerare kommer fortfarande att behövas för att designa de komponenter som kommer att finnas tillgängliga för användaren, men de kommer inte att behöva vara involverade i designen av individuella effekter i samma grad som tidigare. Det kommer dock fortfarande krävas att användaren har kunskap om olika grafikrelaterade koncept som normalvektorer, transformationer mm. Hur djup kunskap som krävs beror av komplexiteten och abstraktionsnivån hos systemet, där flexibilitet får vägas mot komplexitet.

Det finns idag inga fristående kommersiella verktyg för visuell shaderprogrammering, men det har börjat dyka upp en del verktyg som tillägg till kommersiella spelmotorer som är baserade på denna teknik. Dessa verktyg är dock specialiserade för att fungera ihop med sina respektive spelmotorer.

Om verktyget för laborationen

Verktyget som kommer att användas i denna laboration bygger på visuell programmering och har utvecklats som ett examensarbete på KTH. I detta verktyg kan användaren designa shaders genom att i ett grafiskt gränssnitt skapa en graf bestående av hörn som representerar olika grafikoperationer och kanter som anger sammanlänkningar av utdata från en operation som indata till en annan. Systemet kan sedan automatiskt generera källkod för shaders i shaderspråket GLSL. På så sätt slipper användaren skriva shaderprogrammen för hand.

Verktyget är fortfarande under utveckling och kan därför verka inkomplett inom vissa områden. Då verktyget av samma orsak inte testats ut i stor omfattning bör man vara beredd på att eventuella buggar kan förekomma, men hoppas att ni i så fall kan ha överseende med detta.

Laborationsuppgifter

Förberedelser

- Ladda ner filerna som behövs för labben (*shaderlabb.zip*) från:

  http://www.student.nada.kth.se/~d00-mfi/shaderlabb/

- Packa upp *shaderlabb.zip* i en ny katalog.
- Starta *sheditor.exe*.

Deluppgift 1 – Komma igång med verktyget

1. Vi börjar med att skapa en Output-nod. Varje shader-graf som skapas i systemet behöver en Output-nod som skriver ut resultatet av shadern till skärmen. Noder skapas genom att högerklicka någonstans i graf-fönstret och sedan välja den nodtyp man vill skapa från menyn som poppar upp. Det finns idag bara en typ av Output-nod att välja mellan, och det är ”Output/PixelOutput”. Denna nodtyp har en enda in-parameter - *Color* - den färg som skall skrivas till pixlarna vid rendering.

2. Vi har nu en minimal graf som består av en enda nod – vår Output-nod. För att generera en shader från grafen måste den kompileras. Detta görs med menykommandot *File/Compile* (eller med snabbknapp F7) eller motsvarande knapp i knappraden i övre delen av fönstret. När grafen kompilerats visas den genererade källkoden för shadern i *Output*-fönstret i nedre delen av applikationsfönstret. Notera att detta fönster har två flikar. Fliken *Vertex Shader* visar det program som kommer exekveras för varje hörn i geometrin, och fliken *Fragment Shader* visar det program som kommer exekveras för varje bildfragment (pixel). *Preview*-fönstret visar resultatet av den genererade shadern applicerad på en kub. Det enda vår enkla shader gör än så länge är att skriva en vit färg till pixlarna under renderingen.

3. In-parametern *Color* i vår Output-nod har fortfarande inte fått något tilldelat värde och kommer därför att tilldelas ett standardvärde vid kompilering av grafen. Standardvärdet för dess datatyp (*color*) är vit, därav den vita kuben i preview-fönstret. Vi skall nu prova att ange ett konstantvärde för *Color*-parametern. Detta görs genom att klicka på parametern (texten *Color* i noden) och sen välja *Constant* i menyn som poppar upp. Ett fönster visas nu till höger om parametern som anger dess värde. Det finns olika varianter av fönster för olika typer av parametrar. Vår parameter *Color* är en färgparameter har därför en färgkontroll som styr dess värde. Klicka i kontrollen för att ändra färgen och kompilera sedan om grafen igen för att se resultatet i preview-fönstret.

4. Vi skall nu prova att ange ett vertex-attribut som input istället för en konstant. Som bekant är vertex-attribut data som finns lagrade för varje vertex (hörn) i geometrin som skall renderas, som t ex texturkoordinater, normalvektor, färg mm. Högerklicka på *Attributes* i *Explorer*-fönstret till vänster och välj *Import Attributes from Mesh*. Ett fönster öppnas som visar de attribut som finns lagrade i den geometri som för tillfället finns inladdad i *Preview*-fönstret. Vi skall lägga till vertex-attributet *COLOR0* som är ett färgvärde som alltså finns lagrat för varje hörn i vårt kub-objekt. Markera *COLOR0* och klicka på *Import*. *COLOR0* dyker nu upp i listan av definierade attribut i *Explorer*-fönstret. Nu när vi har definierat ett vertex-attribut kan vi använda det som in-parametrar till noder i vår graf. Klicka på parametern *Color* i vår graf-nod och välj *Attributes/COLOR0*. Kompilera. Modellen i *Preview*-fönstret kommer nu renderas med resultatet av interpolering av de färgvärden som finns lagrade vid varje hörn i geometrin. Om du gjort rätt så ska du ha en kub med olika färger på varje sida.

5. Glöm inte att spara din effekt innan du börjar på nästa uppgift!

Deluppgift 2 – Texturer

1. Starta en ny effekt (*File/New*) och skapa en rot-nod *Output/PixelOutput* samt en nod av typen *Texturing/2DTexture*. *2DTexture*-noden läser ett färgvärde från en textur. Den har två input; *Texture* som anger vilken textur som skall användas och *Coords* som anger vilken koordinat som skall användas för läsningen. Noden har också två output; *Color* är färgen (RGB) från läsningen och *Alpha* är *alpha*-komponenten från läsningen (1.0 om texturen saknar *alpha*-komponent).

2. För att *2DTexture*-noden skall kunna fungera måste man ange en textur som den skall läsa från. Lägg till en ny textur till projektet med menykommandot *Effect/Textures/New Texture* (eller motsvarande knapp från knappraden i övre delena av fönstret). Ange ett namn, t ex *MyTexture* och ange en bildfil genom att tryck på knappen "…", förslagsvis *textures/face.jpg*. Typfältet ska vara *2D*. Tryck på *OK*.

3. Ange den nyskapade texturen som input för parametern *Texture* i *2DTexture*-noden i grafen genom att klicka på denna parameter och välja din nyligen skapade textur från listan som poppar upp. Vi måste sedan också ange vilka koordinater som skall användas för läsning av färgvärden från vår textur. Vi kommer att använda oss av de texturkoordinater som finns lagrade i varje hörn i geometrin. Lägg till vertex-attributet *TEXCOORD0* till projektet (se uppgift 1 för hur man lägger till vertex-attribut) och ange det som input för parametern *Coords* i *2DTexture*-noden.

4. Koppla slutligen ihop ut-parametern *Color* från *2DTexture*-noden med in-parametern *Color* i *Output*-noden genom att klicka på den tidigare, dra, och slutligen släppa över den senare. Kompilera och notera hur din textur nu kommer appliceras på objektet i *Preview*-fönstret. Textur-koordinaterna som finns lagrade i varje hörn av geometrin kommer att interpoleras över ytan av objektet. För varje pixel som renderas kommer dessa interpolerade koordinater användas för att läsa en färg från texturen, och den resulterande färgen kommer att skrivas ut till pixeln. För att se hur din effekt ser ut på andra 3D-objekt kan du högerklicka i preview-fönstret och välja ett annat objekt från undermenyn *Mesh*. Prova också att växla till preview-läge genom att trycka på F9. *Preview*-fönstret täcker då hela applikationsfönstret. Tryck på F9 igen för att återgå till normalläge.

5. Prova att lägga till en nod av typen *Color/Grayscale*. Denna nodtyp tar en färg som input och genererar en svart-vit färg som output. Använd noden för att göra så att shadern renderar din textur i svart-vit genom att låta färgen från texturläsningen passera genom Grayscale-noden innan den når Output-noden. Kompilera och kontrollera att texturen nu renderas i svart-vitt i *Preview*-fönstret.

6. Glöm inte att spara din effekt innan du börjar på nästa uppgift!

Figur 3 *Textur* (*som körts genom ett gråskalefilter*).

Deluppgift 3 – Lamberts reflektionsmodell

Lamberts reflektionsmodell är en enkel reflektionsmodell som kan användas för att simulera matta ytor:

$$I = N \bullet L$$

där **N** är ytans normal och **L** är riktningen mot ljuskällan (båda normerade).

Det finns en färdig nodtyp *Lighting/Lambert* som implementerar Lamberts reflektionsmodell. Denna nod har 2 in-parametrar *Diffuse* och *Normal*, samt en ut-parameter *Color*. Noden implementerar Lamberts reflektionsmodell på följande sätt:

$$Color = Diffuse * (Normal \bullet L)$$

där *Diffuse* är ytans färg, *Normal* är ytans normalvektor och *L* riktningen mot ljuskällan.

1. Skapa en ny effekt och lägg till noderna *Lighting/Lambert* och *Output/PixelOutput*. Lägg till vertex-attributet *NORMAL* till projektet och ange det som input till *Normal*-parametern för Lambert-noden. Koppla sedan ihop ut-parametern *Color* från *Lambert*-noden med in-parametern i *Output*-noden. Kompilera. Effekten blir tydligast om du använder något objekt med böjda ytor som t ex *Torus* (objekt väljs genom att högerklicka i *Preview*-fönstret och sen välja ett objekt från undermenyn *Meshes*). Notera att ljuskällan som används av systemet fungerar som en pannlampa och alltså följer med betraktaren när du panorerar runt objektet (se Figur 3). Det går att växla mellan fast belysning och ”pannlamp-belysning” i menyn *Preview/Lighting*.

2. Prova också att ange en färg på ytan genom att välja en konstant för *Diffuse*-parametern i *Lambert*-noden.

3. Glöm inte att spara din effekt innan du börjar på nästa uppgift!

Figur 4 *Lamberts reflektionsmodell*

Deluppgift 4 – Phongs reflektionsmodell

Till skillnad från Lamberts reflektionsmodell tar Phongs modell även hänsyn till betraktningsrikting vilket gör att det är möjligt att även simulera blanka reflektioner av ljus:

$$I = k_{ambient} + k_{diffuse}(N \bullet L) + k_{specular}(N \bullet H)^{pow}$$

Där H är vektorn

$$H = (L + E) \,/\, |(L + E)|$$

L är riktningen mot ljuskällan och E är betraktningsriktningen. $k_{ambient}$ anger intensiteten av indirekt ljus som når ytan via reflektion av andra ytor i scenen. Tänk det som en form av bakgrundsbelysning som finns överallt i scenen. $k_{diffuse}$ anger hur mycket matt ljus (Lambert-reflektion) ytan reflekterar och $k_{specular}$ anger hur mycket speglat ljus ytan reflekterar. *pow* är en skalär som anger hur ”jämnt” ljus reflekteras vid speglad reflektion (värden i intervallet 10-150 fungerar bra, ju högre värde destå blankare upplevs ytan).

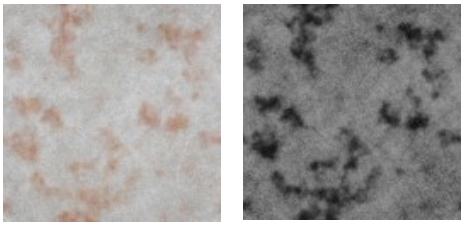
Det finns en färdig nodtyp *Lighting/Phong* som implementerar Phongs reflektionsmodell. Denna nod har 5 in-parametrar: *Ambient, Diffuse, Specular, Shininess* och *Normal*, samt en ut-parameter *Color*. Parametern *Shininess* motsvarar *pow* i funktionen ovan.

1. Skapa en ny effekt och lägg till noderna *Lighting/Phong* och *Output/PixelOutput*. Ange vertex-attributet *NORMAL* som input till in-parameter *Normal* för *Phong*-noden och koppla resultatet av *Phong*-noden till *Output*-noden. Kompilera. Notera hur ytan på objektet nu i tillägg till matt reflektion även har speglad reflektion av ljus i form av en ljuskäggla som vandrar över ytan på objektet (tydligast på objekt med böjda ytor). Experimentera med olika konstant-värden för in-parametrarna *Ambient*, *Diffuse*, *Specular* och *Shininess*.

2. Prova nu att även applicera en textur på objektet. Lägg till en nod för texturläsning (*Texturing/Texture2D*, se uppg 2) och använd bilden *textures/metal.png*. Koppla sedan färgen från texturläsningen till *Diffuse*-parametern för *Phong*-noden. Kompilera.

3. Bilden *textures/metal.png* har förutom färginformation (Röd-, Grön- och Blå-kanal) även en Alpha-kanal som lagrar blankhet för varje pixel i bilden (se Figur 5). Prova att även använda denna blankhetsinformationen från texturen genom att koppla *Alpha*-värdet från texturläsningen till *Specular*-parametern på *Phong*-noden. Notera nu hur de rostiga delarna av texturen inte inte reflekterar lika mycket speglat ljus som övriga områden (se Figur 6). Experimentera med de olika belysningsparametrarna i Phong-noden tills du är nöjd med resultatet.

4. Glöm inte att spara din effekt innan du börjar på nästa uppgift!



Figur 5 *Vänster: färg (R-, G-, och B-kanal). Höger: blankhet (Alpha-kanal).*

Figur 6 *Phongs reflektionsmodell med textur.*

Deluppgift 5 – Normal mapping

Normal mapping är en teknik för att ge intryck av att en yta har mer detalj än den egentligen har. Tekniken bygger på användet av en speciell typ av textur som istället för att lagra färger lagrar normalvektorer för varje bildelement (pixel). X-, Y- och Z- komponenterna för normalvektorn kodas i Röd-, Grön- respektive Blå-kanalen i texturen.

Eftersom komponenterna för normalvektorerna är definierade på intervallet [-1, 1] men de flesta bildformat bara kan lagra röd-, grön- och blåvärden med intervallet [0, 1] måste normalvektorns komponenter transformeras innan de kan lagras i en bildfil:

$$\text{Normal} = (X, Y, Z)$$

$$\begin{aligned} R &= (X + 1) / 2 \\ G &= (Y + 1) / 2 \\ B &= (Z + 1) / 2 \end{aligned}$$

1. Skapa en texturerad Phong shader på samma sätt som i uppgift 4, men använd istället texturen *textures/stone.jpg* (denna textur har inte någon blankhetsinformation lagrad i alpha-kanalen, så koppla heller inte alpha-kanalen till Specular-parametern som vi gjorde i tidigare uppgift). Kompilera och kontrollera att du har en fungerande Phong shader applicerad på en sten-textur.

2. Lägg nu till texturen *textures/stone_bump.png* till projektet. Denna textur har normal-vektorer lagrade för varje pixel med den metod som breskrivits ovan.

3. Skapa sedan en nod av typen *Texturing/NormalMap*. Denna nodtyp läser ett färgvärde från en textur och omvandlar R-, G- och B-komponenten till X-, Y- och Z-komponenten för en normal enligt metoden ovan. In-parametrarna för denna nodtyp är de samma som för *Texturing/2DTexture* men istället för färg som ut-parameter har den alltså en normalvektor. Ange texturen *textures/stone_bump.png* som input till NormalMap-noden (och glöm inte att även ange texturkoordinater).

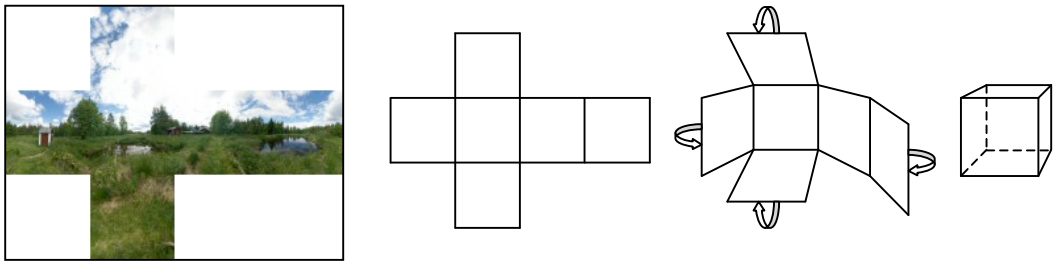
4. Använd den resulterande vektorn från *NormalMap*-noden som input till *Normal*-parametern för *Phong*-noden istället för vertex-attributet som vi använt tidigare. Kompilera. Använd gärna preview-läge för att se bättre (genom att trycka på F9). Notera hur ytan nu upplevs ha mängder av små detaljer (se Figur 7) tack vare att varje punkt på ytan får en unik tilldelad normal, till skillnad från tidigare då vi endast använt en interpolation av de normalvektorer som finns lagrade i varje hörn av geometrin. Experimentera med de olika belysningsparametrarna i Phong-noden tills du är nöjd med resultatet.

5. Glöm inte att spara din effekt innan du börjar på nästa uppgift!

Figur 7 *Normal mapping.*

Deluppgift 6 – Environment mapping

Environment mapping är renderingsalgoritmer som använder någon form av texturer för att representera ljus från omgivningen. Ofta används en speciell typ av textur som kallas för *Cube map*. Denna textur består av 6 st 2D-texturer som representerar varsin sida av en kub. De 6 texturerna lagras lämpligen i en och samma bildfil, exempelvis med den layout som visas i Figur 8.

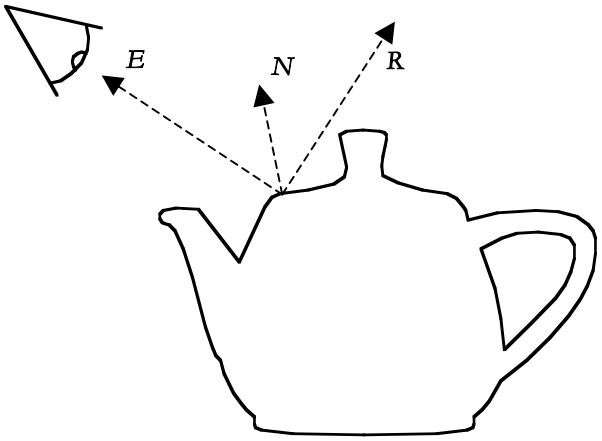


Figur 8 *Cube map. Vänster:Hur en cube map kan lagras i en bildfil. Höger: Hur de 6 delarna av bilden appliceras på en kub.* (*Texturen är nedladdad från http://www.humus.ca*)

Till skillnad från vanliga tvådimensionella texturer där man använder sig av tvådimensionella koordinater för att läsa från texturen, så används en 3D-vektor för att läsa värden från en cube map. Vid läsningen följs riktningen av den angivna 3D-vektorn med utgångspunkt från centrum av kuben tills en av kubens 6 sidor påträffas. Den 2D-textur som motsvarar denna sida av kuben används sedan för att göra en läsning med koordinaten motsvarande den punkt på ytan där vår 3D-vektor penetrerar kuben.

Reflection Mapping

Reflection mapping är en metod för att rendera ytor som speglar omgivningen (se Figur 9). Reflektionsvektorn $\boldsymbol{R}$ beräknas genom att spegla betraktningsvektorn $\boldsymbol{E}$ med ytans normalvektor $\boldsymbol{N}$ (se Figur 9). Reflektionsvektorn $\boldsymbol{R}$ används sedan för att slå upp en färg i en cube map som representerar omgivningen. Eftersom vektorn som används för uppslagningen mot vår cube map antas utgå från centrum av den tänkta kuben som beskrevs ovan, så gör algoritmen approximationen att omgivningen befinner sig på oändligt lång avstånd. Denna algoritm lämpar sig därför inte väl för att simulera spegling av närliggande föremål.



Figur 9 *Reflection mapping.*

Vi skall nu prova på att implementera Reflection mapping i verktyget.

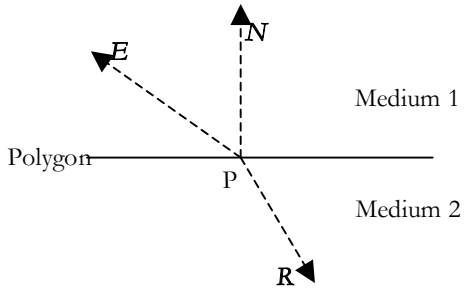
1. Börja med att lägga en cube map som bakgrund i Preview-fönstret. Detta görs genom att välja menykommandot *Preview/Background*, kryssa för *Cube Map* och sedan öppna en cube map-fil (*textures/cubemap.jpg*) med knappen "…".

2. Lägg till samma cube map som en textur *textures/cubemap.jpg* till projektet. Detta görs på samma sätt som med en vanlig tvådimensionell textur men man måste ange "Cube" istället för "2D" i fältet *Type* i dialogfönstret *New Texture*.

3. Skapa nu en shader som implementerar Reflection mapping. Använd noden *Misc/EyeVector* för att erhålla betraktningsvektorn, noden *Vector/Reflect* för att beräkna reflektionsvektorn och använd slutligen reflektionsvektorn för att slå upp en färg från vår cube map med hjälp av en nod av typen *Texturing/CubeMap*. Använd också Normal mapping med texturen *textures/face_bump.jpg* eller *textures/stone_bump.png* för att beräkna ytans normal. Koppla slutligen resultatet av läsningen från din cube map till en *Output*-nod.

Om du gjort allting korrekt skall du nu ha ett objekt som ser ut att spegla omgivningen. Tips! Du kan dölja rutnätet under objektet med meny-valet *Preview/Show Grid* om du tycker det stör mot bakgrunden.

Refraction Mapping
Med en enkel förändring av vår shader kan vi få den att simulera refraktion (brytning av ljus) istället för reflektion, vilket gör det möjligt att simulera genomskinliga material. När en ljusstråle passerar övergången mellan två medium med olika brytningsindex bryts ljusstrålen mot eller från normalen beroende på förhållandena mellan de två mediumens brytningsindex. Refraction mapping (se Figur 10) är en metod för att simulera refraktion av ljus. För en given punkt P ett objekts yta kan man beräkna refraktionsvektorn $\mathbf{R}$, som anger från vilken riktning som ljusstrålar som skulle anlända till punkten P skulle brytas så att de precis skulle sammanfalla med betraktningsvektorn $\boldsymbol{E}$. Denna refraktionsvektor kan sedan användas för att slå upp en färg från en cube map för att ta reda på vilken färg det ljus som anländer från den riktningen har. I verkligheten skulle ljusstrålarna som går genom objektet även brytas på andra sidan objektet, där det också passerar gränsen mellan två medium med olika brytningsindex. Vi kommer endast att ta hänsyn till den första brytningen, men resultatet blir ändå övertygande.



Figur 10 *Refraction mapping.*

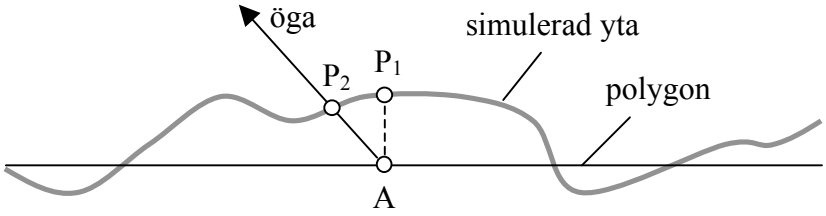
4. Gör om din shader till Refraction mapping genom att byta ut *Reflection*-noden mot en nod av typen *Lighting/Refract*. Denna nod har även en in-parameter *Eta* som anger förhållandet mellan brytningsindexen hos de två medium ljuset passerar genom. För ett objekt av glas skulle denna parameter exempelvis bli:

$$Eta = n_{luft} / n_{glas} = 1.0 / 1.5 = 0.67$$

5. Glöm inte att spara din effekt!

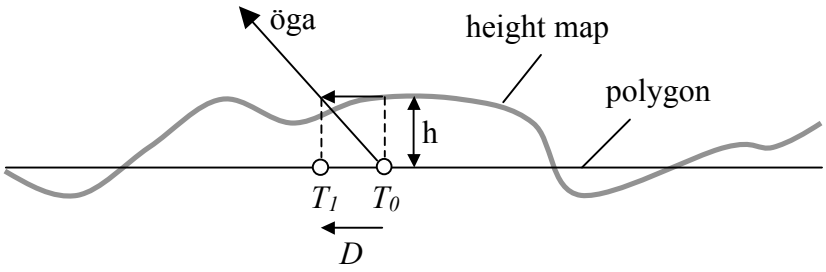
Deluppgift 7 (Valfri, i mån av tid) – Parallax mapping

Vid ren texture mapping upplevs ytan på objekt ofta som orealistiskt platt. Om materialet som representeras av texturen endast har små ojämnheter kan det räcka med att använda normal mapping för att få ett bra resultat, men om det verkliga materialet har större ojämnheter så faller illusionen så fort man vrider på objektet och ser på det från olika vinklar. Problemet är att utbuktningar i materialet borde skymma intilliggande områden av ytan då man betraktar ytan från vissa vinklar osv. Problemet åskådliggörs i Figur 11. Eftersom en textur plattas ihop på en polygon kommer punkten A på polygonen renderas med färgen från punkten P_1 på ytan/texturen, men om vi skulle betrakta den verkliga ytan längs ögonvektorn i figuren ser vi att vi egentligen skulle se punkten P_2.



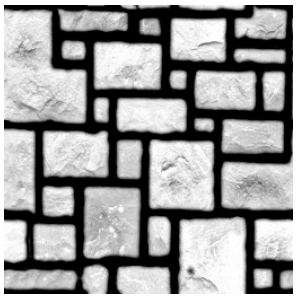
Figur 11 *Eftersom en textur appliceras helt platt på en polygon kommer punkten A på polygonen att inkorrekt renderas med färgen från texturpunkten P_1 istället för P_2.*

Parallax mapping går ut på att justera texturkoordinaten så att färgen från punkten P_2 används instället för P_1, vilket får effekten att ytan upplevs som tredimensionell fast den fortfarande är renderad på en platt polygon. Det finns olika varianter av denna algoritm som beräknar justeringen av texturkoordinaten med olika noggrannhet. Den algoritm som vi kommer att använda oss av bygger på en relativt grov approximering av punkten P_2 och finns beskriven i rapporten *Parallax mapping with Offset Limiting* (Welsh, 2004). Grundidén med algoritmen vi kommer att använder oss av beskrivs i Figur 12.



Figur 12 *Justeringsvektorn **D** som behövs för att korrigera en texturkoordinat från $\boldsymbol{T_0}$ till $\boldsymbol{T_1}$ beräknas med hjälp av ett höjdvärde **h** som läses från en height map vid $\boldsymbol{T_0}$.*

För att algoritmen skall kunna köras behövs en height map. En height map är en speciell typ av textur som lagrar höjdskillnader för varje pixel (se Figur 13). För varje fragment som skall renderas börjar algoritmen med att läsa ett höjd-värde från en height map (detta görs med vanliga interpolerade texturkoordinater). Algoritmen transformerar därefter betraktningsvektorn till *texture space* (texturkoordinater) och multiplicerar denna vektor med höjdvärdet för att få fram den vektor som motsvarar den justering av texturkoordinaten som skall göras. Därefter justeras den texturkoordinaten genom att addera den beräknade justeringsvektorn. Den justerade texturkoordinaten används slutligen för att läsa ett värde från en textur och resultatet skrivs till pixeln.



Figur 13 ***Height map.***

Vi skall nu skapa en shader som använder sig av Parallax mapping. Texturen som skall användas är *textures/stone.png*. Höjdinformation (height map) för denna textur finns lagrad i *alpha*-kanalen i bilden *textures/stone_bump.png* (se Figur 13).

Det finns en färdig nodtyp *Texture/ParallaxOffset* som implementerar Parallax mapping. Denna nod har två inparametrar - *Height* och *Scale* – samt en ut parameter - *Offset*. *Height* motsvarar parametern h i Figur 12, *Scale* anger max djup i texturen relativt texturens bredd (0,01 - 0,04 brukar vara lagom) och *Offset* motsvarar den beräknade justeringsvektorn D i Figur 12.

1. Göra en kopia av uppgift 5. Vi ska nu ändra i denna uppgift så att den använder texturkoordinater som justerats med Parallax mapping, både för läsningen från textur och Normal map.

2. Vi skall nu lägga till noder för att beräkna justerade texturkoordinater. Börja med att läsa ett höjdvärde från alpha-kanalen i bilden *textures/stone_bump.png* (med hjälp av en vanlig *Texture2D*-nod). Skapa sedan en nod *Texture/ParallaxOffset* och använd det lästa höjdvärdet som input till *Height*-parametern i denna nod. Skapa slutligen en nod av typen *Arithmetic/Add* och använd denna nod för att addera vertex-attributet TEXCOORD0 med den av *ParallaxOffset*-noden beräknade justeringsvektorn. Resultatet av denna addition är den slutliga korrigerade texturkoordinaten T_1 i Figur 12.

3. Använd den beräknade justerade texturkoordinaten i resten av grafen (från uppgift 5) istället för vertex-attributet TEXCOORD0 som använts tidigare. Kompilera. Växla till preview-läge (F9) och notera nu hur ytan upplevs som tredimensionell när objektet roteras!

4. Glöm inte att spara din effekt innan du börjar på nästa uppgift!

Deluppgift 8 (Valfri, i mån av tid) – Implementera egen nod

Denna uppgift går ut på att implementera en egen nod i programmet, med hjälp av shaderspråket GLSL. Uppgiften är helt valfri och är till för dig som blivit klar snabbt med de tidigare uppgifterna eller är sugen på en utmaning!

Gör en kopia av filen *nodes/Colors/Grayscale.xml* och kalla den för *nodes/Colors/Saturation.xml*. Försök därefter ändra i *Saturation.xml* så att den tar en input *Color* som är av typen *color*, och ytterligare en input *Amount* av typen *float*, samt en output *SatColor* av typen *color*. Försök sedan ändra i koden i *body*-blocket så att komponenten tar in-färgen och ändrar färgbalansen så att *Amount* =0 leder till gråskala kommer ut och *Amount*=1 leder till att färgen kommer ut oförändrad, och att ett värde mellan 0-1 resulterar i en skala av färgstyrka där emellan.

Varje variabel som används i GLSL-koden (inom *body*-blocket) måste skrivas som ett $-tecken direkt följt av variabelns namn. Detta för att programmet skall kunna identifiera vilka delar av koden som är variabler När programmet väver ihop koden från alla noder för att skapa ett färdigt shaderprogram så byts dessa variabler ut för att länka ihop data mellan noderna. Input- och output-parametrarna kan refereras till i programmet som vanliga variabler (även dessa med ett $-prefix) (se *nodes/Colors/Grayscale.xml*).

Ett tips är att det kan underlätta att titta på hur noden *Colors/Mix* (*nodes/Colors/Mix.xml*) är implementerad. När programmet startas om nästa gång kommer noden *Saturate* att dyka upp bland de andra i menyn (under gruppen *Colors*). Prova att använda din nya nod i en effekt och titta på den genererade koden i *Output*-fönstret för att se hur din kod vävs in i programmet.

OBS! Felhanteringen i systemet är fortfarande dålig så man får tyvärr inte mycket mer hjälp än att det inte fungerar om man har några syntax-fel i sin nod-fil! Beklagar detta. Programmet måste också tyvärr startas om för varje gång du gör ändringar i din fil.

Ett annat tips på en nod som du kan prova att skapa är en nod som inverterar en färg (för att t ex generera ett negativ av en textur). Skapa exempelvis en nod *nodes/Colors/Negative.xml* som implementerar detta, med en input *Color* av typen *color*, och en output *NegColor* av typen *color*.

Lycka till!