Laboration 5 – Shaders

Syfte

Erbjuder studenterna en möjlighet att lära sig *grunderna i GPU-programmering*.

Genomförande

Genomförs *individuellt* eller i grupp om *2 personer*. ***Notera att denna labb måste göras på PC, på CSC är det förberett för labben i Matsalen och Konsthallen.***

Uppskattad tidsåtgång

6-12 timmar effektiv arbetstid per gruppmedlem.

Informations- och bakgrundsmaterial

- *En kort introduktion till OpenGL*.
- Utdrag ur *Cg User's Manual* (kursbunten).
- Avsnitt i kursboken om transformationer.
- *C för den som kan Java*.
- Utdelat material om JPEG och om hur man kompilerar OpenGL-program på Mac/PC.
- Föreläsnings- och övningsanteckningar, ev. annat utdelat material.
- http://www.opengl.org
- Google!

Uppgift

Skriva en uppsättning shaders som implementerar olika reflektionsmodeller.

Redovisning

Redovisas per e-post till gustavt@csc.kth.se senast **fredag 20 april kl. 23.59**. Följande ska finnas med i redovisningen:

1. Studentens(ernas) namn och personnummer.
2. C- eller C++-källkoden till uppgift 2-6! (Om du gjort labben på PC, lägg gärna med färdigkompilerade versioner också!)
3. Kopior av de texturer du/ni använt!

Förslag på uppdelning (om ni jobbar i par)

Den här labben har ingen naturlig uppdelning – det bästa är om ni arbetar sida vid sida och hjälper varandra. ***Dela inte upp deluppgifterna mellan er och genomför dem ensamma – det betraktas som fusk!*** Gör labben individuellt om ni inte har tid att sitta tillsammans.

Deluppgift 1: Få igång Visual Studio och Cg

Vertex- och fragmentprogram är något som ersätter delar av OpenGLs pipeline. Det betyder att du själv måste ansvara för en del funktioner som OpenGL annars utför automatiskt. Den viktigaste av dessa är transformationer av hörn och normaler. ***Det är alltså viktigt att du har koll på hur OpenGL hanterar transformationer innan du börjar***. Läs i kursboken om *transformationsmatriser*, *objektkoordinater*, *ögonkoordinater* och *klippkoordinater*!

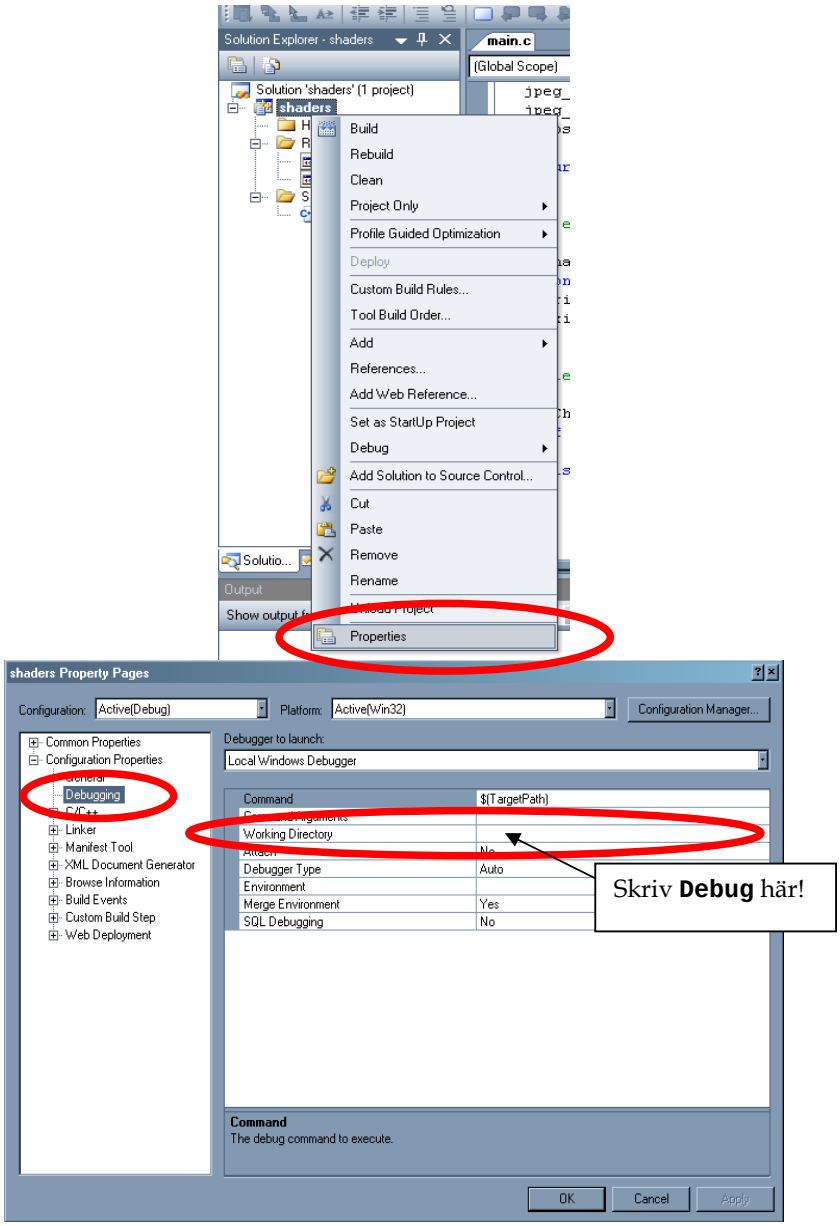
Trots att möjligheter till GPU-programmering har funnits i flera år är kompilatorerna inte alls så långt utvecklade som till Java, C och C++. Dessutom kan grafikkortens drivrutiner vara bristfälliga. Det betyder att man då och då får fel resultat trots att man skrivit programmet rätt. Tyvärr är det inte mycket att göra i de lägena annat än att pröva en alternativ lösning. Ett annat problem är att dagens grafikkort inte kan ha hur långa vertex- och fragmentprogram som helst. ***Det är viktigt att du så fort som möjligt tar kontakt med en handledare om du misstänker att du har sådana här fel***, annars riskerar du att ”fastna” i onödan.

Information om Cg finns i *Start→nVidia Corporation→Cg Toolkit→User's Manual*. Du behöver inte läsa hela manualen, men sidorna 11-23, 33-41 och 43-85 är bra att ha gått igenom. På sidorna 146-152 finns en bra tutorial. Referensmaterial finns i *Start→nVidia Corporation→Cg Toolkit→Reference Manual*.

Börja med att kopiera den katalog i kurskatalogen som heter `shaders` till din arbetsdator. Du behöver ungefär 15Mb ledigt utrymme. Använd programmet *Start→SSH Secure Shell→Secure File Transfer Client* för att komma åt kurskatalogen `/afs/nada.kth.se/info/grip07/`. (Om du vill göra labben hemma och använder Visual Studio 2003, använd `shaders-Visual_Studio_2003.zip` istället.)

Dubbelklicka på `shaders.sln` för att öppna Visual Studio. Se till att fliken ”Solution Explorer” är vald i rutan till vänster och klicka på +-ikonen invid ”shaders”-raden. Öppna upp de tre foldrarna ”Source Files”, ”Header Files” och ”Resources”. Du bör nu kunna se de filer som ingår i projektet. Dubbelklicka på `main.c` för att öppna den i editorn.

Viktigt: Nästa sak du måste göra är att ange vilken katalog programmet ska startas i när det körs. Högerklicka på `shaders`-projektet i ”Solution Explorer” och välj *Properties…* Klicka på *Debugging* i rutan till vänster och skriv in katalogen `Debug` som *Working Directory* i rutan till höger:



Pröva nu att kompilera! Välj *Build→Build Solution* i menyn (eller tryck <Ctrl>+<Shift>+<B>). Du bör inte få några kompileringsfel. Om du får det och inte själv kan klura ut vad det är som är fel, kontakta en handledare. För att köra programmet, välj *Debug→Start* (eller tryck <Ctrl>+<F5>). Du bör få ett blått, tomt fönster. Tryck <q> eller <Esc> för att stänga det.

Skalprogrammet du just startat laddar ett vertexprogram och ett fragmentprogram, men ritar ingenting. Läs igenom koden ordentligt så att du förstår vad det är som händer. Utöka sedan programmet så att det ritar en polygon. För att det ska fungera måste du:

- Ändra i filen `vertex.cg` (du hittar den under ”Resource Files” i ”Solution Explorer”) så att den verkligen konverterar hörn från objektkoordinater till klippkoordinater. Det innebär att du måste multiplicera hörnets position med matrisen **A** = **M**·**P** där **M** är OpenGLs `MODELVIEW`-matris och **P** är OpenGLs `PROJECTION`-matris. Skapa en `float4x4` uniform-variabel för **A** i

vertexprogrammet. Exempel på hur det kan se ut finns i Cg-manualen på sidan 146-152!

- I filen `main.c` måste du se till att **A** skickas till vertexprogrammet. Läs i Cg-manualen på sid 82-84 eller i Cg-referensmanualen på sidan 320 hur man gör (eller sök i den interaktiva referensmanualen efter `cgGLSetStateMatrixParameter()`)!
- Se till att `MODELVIEW` och `PROJECTION`-matriserna sätts till något vettigt. Du bör ha lärt dig hur det går till i OpenGL-labben!
- Se till att programmet ritar en polygon. Det har du också gjort i OpenGL-labben!
- Ändra i filen `fragment.cg` (du hittar den under ”Resource Files” i ”Solution Explorer”) så att polygonen ritas ut med en färg du tycker är snygg. (Just nu blir den vit.)

Eftersom det är lika vanligt att man skriver fel i ett vertex- eller fragmentprogram som i andra programspråk är Visual Studio-projektet anpassat för att kompilera även `vertex.cg` och `fragment.cg`. Det som händer är att Visual Studio anropar Cg-kompilatorn via ett s.k. ”custom build step”. Det är inte viktigt att känna till detaljerna kring hur det går till. Men kontakta en handledare om du vill lägga till ett nytt Cg-program eller om du misstänker att Visual Studio inte rapporterar fel som Cg-kompilatorn hittar!

Deluppgift 2: Lamberts reflektionsmodell

Nästa steg är att implementera Lamberts reflektionsmodell:

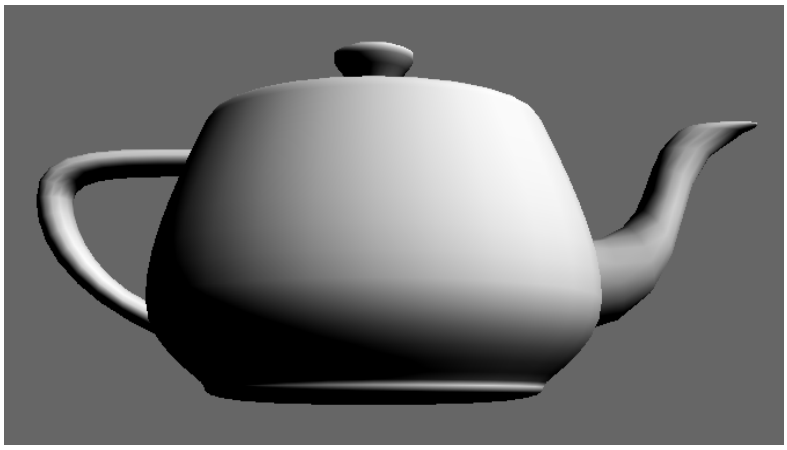
$$\boldsymbol{I} = \boldsymbol{N} \bullet \boldsymbol{L}$$

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

Gör följande:

- För att det ska vara någon idé att arbeta med reflektionsmodeller som denna behöver du en approximation av en böjd yta. Så ersätt din polygon med en `glutSolidTeapot()`! (Du kan behöva rotera den för att den ska hamna upprätt.)
- Lägg till en `float3` uniform-variabel i ditt fragmentprogram och gör så att ditt `main`-program skriver in riktningen till ljuskällan (i ögonkoordinater) i denna. Glöm inte att normera den först!
- Utöka vertexprogrammet så att det tar emot normaler i binding semantic `NORMAL` och skickar vidare transformerade normaler till fragmentprogrammet i binding semantic `TEXCOORD0` (fragmentprogram har ingen binding semantic för normaler). (Läs om binding semantics i *Cg User's Guide* om du inte redan gjort det!) Tänk på att du vill ha normalerna i ögonkoordinater, och att du *inte* ska multiplicera dem med **M**, utan med $(\mathbf{M}^{-1})^{T}$. Du måste alltså lägga till en `float4x4`-variabel i ditt vertexprogram där du skriver in invers-transponat av `MODELVIEW`-matrisen. Läs dokumentationen för Cg-funktionen `cgGLSetStateMatrixParameter()` för att ta reda på hur det går till (du behöver inte beräkna inversen själv).
- Ändra fragmentprogrammet så att det beräknar skalärprodukten mellan ***N*** och ***L***, och sätter detta värde till fragmentets färg. Använd Cg-funktionerna `dot()` och `saturate()` (kolla i manualen vad de gör). **Viktigt**: Eftersom *x*-, *y*- och *z*-värdena i normalerna interpoleras linjärt var för sig över polygoner måste du normera normalen i fragmentprogrammet innan du använder den. Annars är risken stor att du får bandeffekter i din reflektionsmodell.
- Lägg till en `idle()`-funktion i ditt program som roterar tekannan så att du ser att belysningen funkar som den ska.

Om du gjort allt rätt bör du få något i stil med detta:



Deluppgift 3: Phong shading

Nästa reflektionsmodell du ska implementera är Phongs:

$$\boldsymbol{I} = k_d(\boldsymbol{N} \bullet \boldsymbol{L}) + k_s(\boldsymbol{N} \bullet \boldsymbol{H})^{pow}$$

Där $\boldsymbol{H}$ är vektorn

$$\boldsymbol{H} = (\boldsymbol{L} + \boldsymbol{E}) \; / \; |(\boldsymbol{L} + \boldsymbol{E})|$$

$\boldsymbol{E}$ är riktningen från hörnet mot kamerapositionen (normerad) och *pow* är en skalär (värden i intervallet 10-150 brukar fungera bra). k_d och k_s viktar mellan matt och speglande reflektion. $\boldsymbol{H}$ approximerar reflektionsriktningen i hörnet (se kursboken för detaljer). $\boldsymbol{E}$ är enkel att beräkna i vertexprogrammet: OpenGLs kamera sitter som bekant alltid i origo, vilket betyder att

$$\boldsymbol{E} = (-\boldsymbol{P}) \; / \; |\boldsymbol{P}|$$

där $\boldsymbol{P}$ är hörnets position i ögonkoordinater. För att få hörnets position i ögonkoordinater måste du multiplicera det med `MODELVIEW`-matrisen, vilket betyder att du måste skicka också den matrisen som en `float4x4`-variabel till vertexprogrammet.

Om du gjort allt rätt bör du få något i stil med detta (i bilden har k_d och k_s värdet 0.5 och *pow* värdet 64.0):

Deluppgift 4: Cook-Torrance

Den sista reflektionsmodellen du ska pröva är Cook-Torrance. En god approximation av denna modell är följande:

$$D = \frac{1}{m^2 (N \bullet H)^4} e^{-\left(\frac{1-(N \bullet H)^2}{m^2 (N \bullet H)^2}\right)}$$

$$F = n + (1-n)(1-(N \bullet E))^5$$

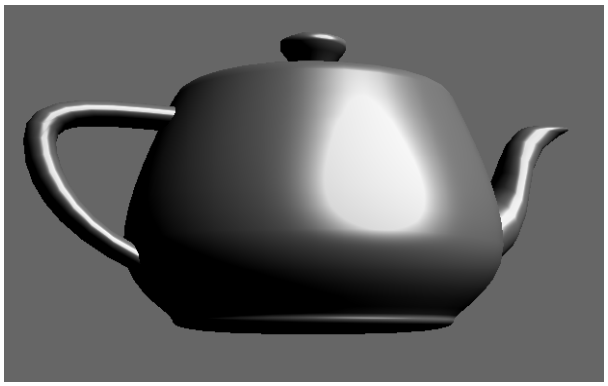
$$\gamma = \frac{2(N \bullet H)}{(E \bullet H)}$$

$$G = \min(\gamma (N \bullet E), \gamma (N \bullet L))$$

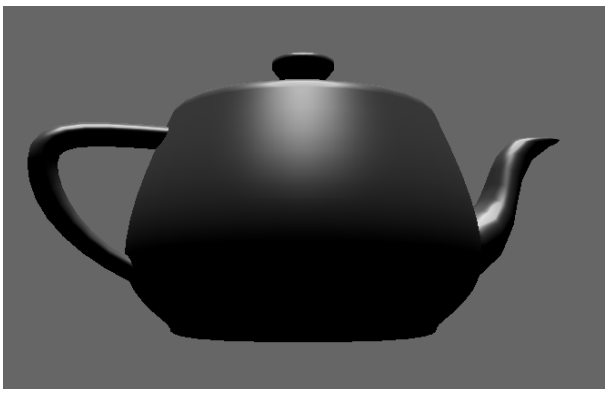
$$\rho = \frac{1}{\pi (N \bullet E)}$$

$$I = k_d (N \bullet L) + k_s D \cdot G \cdot F \cdot \rho$$

där m är en skalär (”roughness”) i intervallet [0.1, 1.0] och n är en skalär (”index of refraction”) i intervallet [0.1, 1.0]. Ett tips är att implementera varje komponent (D, F, γ, G och ρ) för sig och kolla att de verkar vettiga. Om du gjort allt rätt så ska du få något i stil med detta (här är k_d=0.5, k_s=0.5, m=0.3 och n=0.4):



Det kan tyckas som om Cook-Torrance är ekvivalent med Phong, men försök att flytta runt ljuskällan och experimentera lite med m- och n-parametrarna så kan du åstadkomma en rad olika ”metalleffekter” som Phongmodellen inte klarar av att återskapa. Här är ett exempel (samma parametrar som ovan, men med ljuskällan rakt ovanför tekannan):



Deluppgift 5: Texturer

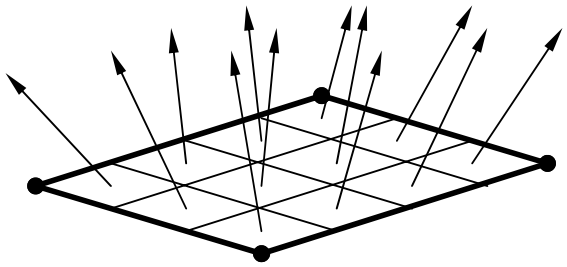
Nu när du har en uppsättning bra reflektionsmodeller är det dags att använda sig av texturering.

- Utöka din pixel- och vertex-shader så att programmet tar emot ”vanliga” texturkoordinater i binding semantic `TEXTURE0`. Flytta alltså ned alla andra parametrar du tar emot till närmsta lediga `TEXTURE`-semantic. Glöm inte att vidarebefordra texturkoordinaterna till fragmentprogrammet – gör det i binding semantic `TEXTURE0`!
- Pröva att allt stämmer genom att ge tekannan färgen (s, t, u, 0) där s, t och u är texturkoordinaterna. Du bör få en röd-grön ”storrutig” tekanna.
- Nästa steg är att lägga till en `uniform sampler2D`-variabel i fragmentprogrammet och sätta denna till din OpenGL-textur. Läs i Cg-manualen på sidan 82-84 för information om hur det går till (och/eller titta i referensmanualen på sid 322 och 202)! Använd sedan RGB-värdena i din textur som värdet på k_d! Information om hur man hämtar data från en textur finns i tutorial-exemplet i Cg-manualen på sid 146-152.
- Du behöver en textur också! Det finns en som heter `wood.jpg` i projektkatalogen. Lägg filen i katalogen `...\shaders\shaders\Debug` på din PC (samma katalog som du angav som *Working Directory* i början av labben). Det är rätt katalog om filen `shaders.exe` finns i den.

Du bör få något i stil med följande:

Deluppgift 6: Normal mapping

Normal mapping är en teknik som används för att ge intrycket av att en yta har mer detalj än den egentligen har. Skillnaden mot vanlig texturering är att man lagrar normaler i texturen och inte färger. En sådan textur kallas *normal map*. Bilden nedan föreställer en polygon med en normal map.



En normal har som bekant *x*-, *y*- och *z*-värden i intervallet [-1, 1], men de flesta bildformat (som t.ex. JPEG) stöder bara värden >0. Det betyder att man i texturen lagrar RGB-värdena

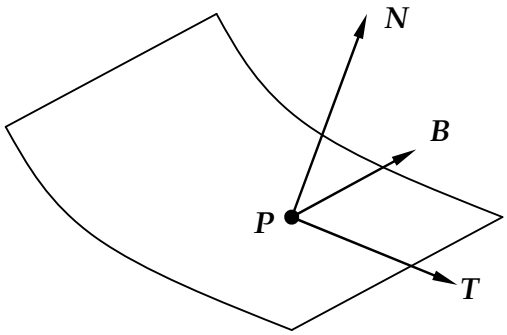
$$\boldsymbol{N}_{\text{normal map}} = (N_x, N_y, N_z)$$

$$\begin{aligned} R &= (N_x + 1) / 2 \\ G &= (N_y + 1) / 2 \\ B &= (N_z + 1) / 2 \end{aligned}$$

vilket i sin tur betyder att du i ditt fragmentprogram måste ”konvertera tillbaka” värdena till [-1, 1]:

$$\begin{aligned} N_x &= 2R - 1 \\ N_y &= 2G - 1 \\ N_z &= 2B - 1 \end{aligned}$$

Eftersom man vill kunna rotera och flytta sina polygoner i förhållande till ljuskällan kan man inte lagra normalerna i ögonkoordinater i texturen – man lagrar dem antingen i objektkoordinater eller i något som man kallar för *tangent space*. Tangent space i en punkt $\boldsymbol{P}$ på en yta definieras så här:



$\boldsymbol{N}$ är ytans normal i $\boldsymbol{P}$, $\boldsymbol{T}$ är en normerad tangentvektor till ytan i $\boldsymbol{P}$. $\boldsymbol{B}$ kalls *bitangent* (ibland används den felaktiga termen *binormal*, vilket egentligen är något annat) och definieras av

$$\boldsymbol{N} = (\boldsymbol{B} \times \boldsymbol{T}) \; / \; |(\boldsymbol{B} \times \boldsymbol{T})|$$

Notera att alla tre vektorerna är i objektkoordinater. Notera också att bitangenten och tangenten inte behöver vara ortogonala för att $\boldsymbol{N}$ ska vara väldefinierad (men de får inte vara lika). Det vanligaste är dock att man utgår från $\boldsymbol{N}$ och $\boldsymbol{T}$, d.v.s.

$$\boldsymbol{B} = (\boldsymbol{N} \times \boldsymbol{T}) \; / \; |(\boldsymbol{N} \times \boldsymbol{T})|$$

När man använder detta till ljussättning gör man följande. Först transformerar man ytans tangent, bitangent och normal till ögonkoordinater:

$$\begin{aligned} \boldsymbol{T}_e &= (\mathbf{M}^{-1})^{\mathrm{T}} \boldsymbol{T} \\ \boldsymbol{B}_e &= (\mathbf{M}^{-1})^{\mathrm{T}} \boldsymbol{B} \\ \boldsymbol{N}_e &= (\mathbf{M}^{-1})^{\mathrm{T}} \boldsymbol{N} \end{aligned}$$

Nästa steg är att transformera ljusriktningen $\boldsymbol{L}$ från ögonkoordinater till rummet som spänns upp av $\boldsymbol{T}_e$, $\boldsymbol{B}_e$ och $\boldsymbol{N}_e$:

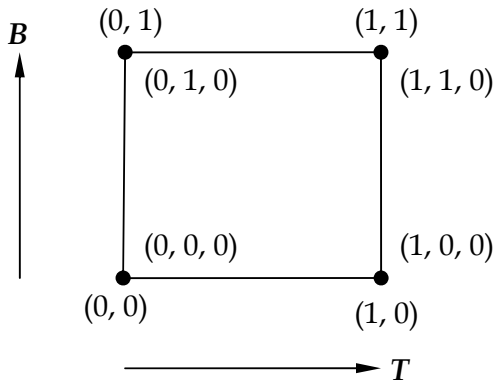
$$\boldsymbol{L}' = (\mathrm{L}'_x, \mathrm{L}'_y, \mathrm{L}'_z)$$

$$\begin{aligned} \mathrm{L}'_x &= \boldsymbol{L} \bullet \boldsymbol{T}_e \\ \mathrm{L}'_y &= \boldsymbol{L} \bullet \boldsymbol{B}_e \\ \mathrm{L}'_z &= \boldsymbol{L} \bullet \boldsymbol{N}_e \end{aligned}$$

Sedan kan man använda normalerna i normal map, t.ex. för Lamberts reflektionsmodell:

$$\boldsymbol{I} = \boldsymbol{N}_{\text{normal map}} \bullet \boldsymbol{L}'$$

Alltså måste man definiera en tangent $\boldsymbol{T}$ till varje hörn i sin modell (eller beräkna den på något sätt). Oftast låter man tangent och binormal ligga i linje med modellens texturkoordinatsystem eftersom det brukar ge bäst resultat rent visuellt. Det är dock överkurs att gå in på hur man genererar tangenter ur godtyckliga texturkoordinater, så vi gör det enkelt för oss i den här labben och arbetar med en enda kvadratisk polygon med följande objektkoordinater och texturkoordinater:



Vi låter alltså normalen i varje hörn vara polygonens normal (0, 0, 1), tangenten vara vektorn (1, 0, 0), och bitangenten vara vektorn (0, 1, 0). Texturkoordinaterna sätter vi som i bilden.

Sammanfattningsvis:

- Ändra ditt `main`-program så att det ritar ut en polygon med hörn och texturkoordinater som i bilden ovan. Se till att den också syns i kameran med hjälp av `gluLookAt()`.

- Ändra ditt vertexprogram så att det transformerar hörnets normal, tangent och bitangent från objektkoordinater till ögonkoordinater enl. ovan. Transformera ljusriktningen till rummet som spänns upp av $\boldsymbol{N}_e$, $\boldsymbol{B}_e$ och $\boldsymbol{T}_e$, också enl. ovan.
- Skicka vidare den transformerade ljusriktningen till fragmentprogrammet. Skicka vidare texturkoordinater precis som i föregående deluppgift.
- I fragmentprogrammet, använd texturkoordinaterna för att hämta normalen från din normal map (det finns en i kurskatalogen) och använd Lamberts reflektionsmodell för att beräkna fragmentets färg. Glöm inte att RGB-värdena i din normal map måste konverteras till intervallet [-1, 1]!
- Ändra ljusriktningen i din `idle()`-funktion så att du ser att allt funkar som det ska.

Du bör få något i stil med följande:

I den vänstra bilden kommer ljuset från höger, i den högra bilden uppifrån.

Om du har tid och lust…

- Kombinera normal mapping med Phongs reflektionsmodell!
- Kombinera normal mapping med en vanlig detaljtextur!