

Bump mapping med varianter

SUPPLEMENT 1 till "So How Can We Make Them Scream?"

Mellan *Polygons Feel No Pain* och *So How Can We Make Them Scream* finns det en lucka, en del material om bump mapping som blev överhoppat i farten. *Polygons Feel No Pain* introducerar begreppet, inte fel direkt (möjligen något teckenfel i en formel) men inte riktigt komplett, eftersom det inte framgår var tangentvektorerna (basvektorerna för texturkoordinaterna) kommer från. Normal mapping bara nämns i förbigående. Bortom detta saknas vissa uppgifter som behövs för parallaxmapping och så vidare.

1. En titt till på Bump mapping

1.1 Basvektorer för texturkoordinater

Vi antar att normalvektorn n levereras från världprogrammet den vanliga vägen, i modellkoordinater, varvid vi transformerar den till vykoordinater med

$$n = \text{gl_NormalMatrix} * \text{gl_Normal}$$

Dessutom behöver vi två basvektorer till, \mathbf{V}_s och \mathbf{V}_t , som anger hur s- och t-koordinaterna varierar, basvektorer för s och t. Dessa måste också leveras från världprogrammet. Vi kan klara oss enbart med \mathbf{V}_s och kryssa fram \mathbf{V}_t .

I fallet med en enkel rektangel så är \mathbf{V}_s och \mathbf{V}_t triviala att beräkna, de går helt enkelt längs kanten på rektangeln. I det generella fallet får vi ett par (s, t) för varje vertex och måste beräkna \mathbf{V}_s (och ev \mathbf{V}_t) från detta.

Om man genererar texturkoordinater själv så är det oftast enkelt att bestämma \mathbf{V}_s och \mathbf{V}_t på samma gång.

Ett "fulhack" som kan fungera är att ansätta en vektor (t.ex. x), och kryssa den med normalvektorn för att få \mathbf{V}_s . Detta ger en vektor som inte har någon som helst koppling till den variation som s och t verkligen har över ytan, men för enkla bump maps i rent demo- bruk kan det duga. Det är dock mer tur än skicklighet.

En seriös lösning skall alltså beräkna en tangentvektor \mathbf{V}_s som ligger längs s-koordinatens basvektor. Denna kan beräknas enligt följande:

Enligt Angel är

$$\mathbf{V}_s = \begin{bmatrix} \frac{\delta x}{\delta s} \\ \frac{\delta y}{\delta s} \\ \frac{\delta z}{\delta s} \end{bmatrix} \quad \mathbf{V}_t = \begin{bmatrix} \frac{\delta x}{\delta t} \\ \frac{\delta y}{\delta t} \\ \frac{\delta z}{\delta t} \end{bmatrix}$$

Därmed är problemet att på ett enkelt sätt beräkna dessa differentier. Dietrich föreslår i Game Programming Gems 1 en lösning med planets ekvation i alternativa rymder. Detta är en trevlig, rättfram lösning. Problemet med GPG-lösningen är att det inte görs helt uppentat hur man finner planets ekvation. Detta är dock lika enkelt som vanligt.

En triangel anger tre punkter $\mathbf{p}_1 = (x_1, y_1, z_1, s_1, t_1)$, $\mathbf{p}_2 = (x_2, y_2, z_2, s_2, t_2)$, $\mathbf{p}_3 = (x_3, y_3, z_3, s_3, t_3)$. Samtliga tre parametrar varierar linjärt relativt varandra! Därför kan vi uttrycka triangeln i vilken 3-dimensionell rymd vi önskar. Dietrich väljer (x, s, t) .

Normalvektorn beräknas precis som vanligt, med kryssprodukten mellan $\mathbf{p}_2 - \mathbf{p}_1$ och $\mathbf{p}_3 - \mathbf{p}_1$, men med (x, s, t) i stället för (x, y, z) .

$$\mathbf{n}_{xst} = (A, B, C)$$

Att variationen i x med avseende på s är $-B/A$ anar vi redan från normalvektorn. Från Dietrich hämtar vi följande metod att beräkna lutningen:

$$A \cdot x_1 + B \cdot s_1 + C \cdot t_1 + D = 0$$

$$A \cdot x_4 + B \cdot s_4 + C \cdot t_4 + D = 0$$

Båda är noll, så vi utnyttjar likheten för att eliminera D :

$$A \cdot x_1 + B \cdot s_1 + C \cdot t_1 + D = A \cdot x_4 + B \cdot s_4 + C \cdot t_4 + D = 0$$

$$A \cdot x_1 + B \cdot s_1 + C \cdot t_1 = A \cdot x_4 + B \cdot s_4 + C \cdot t_4$$

Nu kommer tricket, och orsaken till att jag använde en fjärde punkt: Vi kräver att $t_1 = t_4$! Poängen är att om t varierar alls mellan \mathbf{p}_2 och \mathbf{p}_3 så kan vi interpolera/extrapolera fram en punkt t_4 längs samma linje där $t_1 = t_4$!

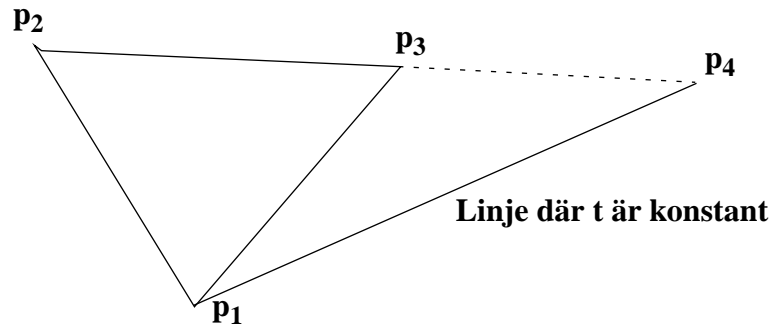


FIGURE 1. Extrapolation till punkt med samma t som p_1

$$A \cdot x_1 + B \cdot s_1 = A \cdot x_4 + B \cdot s_4$$

$$A \cdot x_1 - A \cdot x_4 = B \cdot s_4 - B \cdot s_1$$

men eftersom det är linjära funktioner så är detta ekvivalent med

$$A \cdot dx = -B \cdot ds$$

vilket ger

$$dx/ds = -B/A$$

Punkten p_4 måste alltså inte bestämmas, utan var bara en hjälppunkt på vägen till detta enkla samband. Allt vi behöver är därmed normalvektorn, som ger oss

$$dx/ds = -B/A$$

$$dx/dt = -C/A$$

Övriga differentier dy/ds , dy/dt , dz/ds , dz/dt kan beräknas på motsvarande sätt.

Observera också att V_s och V_t typiskt beräknas i modellkoordinater, och därför måste transformeras till vykoordinater, om igen med `gl_NormalMatrix`. Vi kallar de transformerade vektorerna P_s och P_t . (Kallas även ofta t och b i litteraturen.)

$$P_s = gl_NormalMatrix * V_s$$

$$P_t = gl_NormalMatrix * V_t$$

1.2 Konvertering mellan koordinatsystem

Det finns inte mindre än tre koordinatsystem att hålla reda på när vi jobbar med bump mapping. Vi anger geometrin i *modellkoordinater*. Vi transformerar till *vykoordinater* för att det skall bli möjligt att ta med ljusriktning såväl som betraktningsriktning. Men dessu-

tom finns *texturkoordinater*, vilket i detta fall är en 3-dimensionell rymd definieras av \mathbf{P}_s , \mathbf{P}_t och \mathbf{n} . Vi kommer att se att det finns tillfällen då vi vill arbeta direkt i texturkoordinater.

Från modellkoordinater till vykoordinater kommer vi som vanligt med `gl_NormalMatrix`. Aningen mindre uppenbart är hur vi transformerar från vykoordinater till texturkoordinater. Detta sker med en rotationsmatrix byggd av \mathbf{P}_s , \mathbf{P}_t och \mathbf{n} :

$$M_{vt} = \begin{bmatrix} \mathbf{P}_s \\ \mathbf{P}_t \\ \mathbf{n} \end{bmatrix} = \begin{bmatrix} P_{sx} & P_{sy} & P_{sz} \\ P_{tx} & P_{ty} & P_{tz} \\ n_x & n_y & n_z \end{bmatrix}$$

Matrisen kan trivialt utökas till 4x4 om man behöver det.

Några ord om terminologin här. \mathbf{P}_s brukar (helt korrekt) kallas *tangentvektorn* och ges oftare symbolen \mathbf{t} (trots uppenbar förvirring med texturkoordinaten t , som är längs \mathbf{P}_t). \mathbf{P}_t kallas mer överraskande binormal (symbolen \mathbf{b}), vilket är uppenbart felaktigt, det är snarare *bitangent*. Jag har valt att undvika symbolerna \mathbf{t} och \mathbf{b} , och i stället hålla fast vid \mathbf{P}_s och \mathbf{P}_t (som jag ursprungligen hämtat från Hearn&Baker under namnen \mathbf{P}_u och \mathbf{P}_v).

Texturkoordinater enligt ovan (inklusive normalriktningen) kallas ofta *tangent space*. Man kan skilja på *tangent space* och *texture space*, och låta tangent space vara en ortonormal bas, medan texture space är strikt riktad efter texturriktningen. Jag väljer att inte göra den distinktionen här. Se Dietrich i GPG1. I praktiken är skillnaden i riktningen på \mathbf{P}_t .

Några ord om texturer i detta sammanhang. Med "bump map" avser jag höjdkartan. Den kan naturligtvis lika gärna kallas höjdkarta, height map, men eftersom den används för belysning och inte för geometri så tycker jag det kan passa med ett separat begrepp. En "normal map", normalkarta, är den förgenererade textur med normalvektorer som behandlas nedan. Även denna kallas ibland "bump map" för att göra förvirringen fullständig.

1.3 Modifiering av normalvektorn

När vi arbetar i vykoordinater med \mathbf{P}_s , \mathbf{P}_t , \mathbf{n} , så beräknas den modifierade normalvektorn med

$$b_s = db/ds$$

$$b_t = db/dt$$

$$\mathbf{n}' = \mathbf{n} + b_s \cdot \mathbf{P}_s + b_t \cdot \mathbf{P}_t$$

Detta är helt enligt Polygons Feel No Pain.

En alternativ definition är

$$\mathbf{n}' = \mathbf{n} - b_s \cdot \mathbf{P}_s - b_t \cdot \mathbf{P}_t$$

som är samma sak så när som på tecken, vilket är en definitionsfråga, beror på om bump-mappen går in eller ut ur ytan. Den första är att föredra, då en bumpmap som går in i ytan ger mindre synliga artefakter än en som sticker ut, men den senare är lite mer intuitiv.

Denna definition kräver att $\mathbf{P}_s, \mathbf{P}_t, \mathbf{n}$ är ortogonala. Om de inte är det, eller inte nära nog för att kunna anta det, så är formeln

$$\mathbf{n}' = \mathbf{n} + b_s \cdot (\mathbf{P}_t \times \mathbf{n}) + b_t \cdot (\mathbf{n} \times \mathbf{P}_s)$$

Åter samma som i PFNP så när som på ett teckenfel i en term. Observera också att \mathbf{n}' avslutningsvis måste normeras.

Beräkningen av b_s och b_t görs helt enkelt med en differens mellan två grantextlar i bump-mappen.

$$b_s = b[s+1, t] - b[s, t]$$

$$b_t = b[s, t+1] - b[s, t]$$

Man kan också modifiera normalvektorn i texturkoordinater! Där är den modifierade normalvektorn (före normering)

$$\begin{bmatrix} b_s \\ b_t \\ 1 \end{bmatrix}$$

eller (beroende på bumpmappens riktning)

$$\begin{bmatrix} -b_s \\ -b_t \\ 1 \end{bmatrix}$$

Det är så oförskämt enkelt att trasslet ovan känns lite onödigt. Dock är det inte så onödigt som man kan tro; ljusriktningen måste transformeras till texturkoordinater för att denna vektor skall bli meningsfull. Därmed är förenklingen inte fullt så lönsam, men den kommer att bli viktigare nedan.

2. Normalmappning

Ämnet normalmappning förtjänar att inte hoppas över som en trivial variant av bumpmappning, speciellt som det är den gängse metoden som används i praktiska implementationer.

Tricket med normalmappning är att vi förberäknar normalvektorn, i texturkoordinater. Enligt ovan kan den beräknas som

$$\begin{bmatrix} -b_s \\ -b_t \\ 1 \end{bmatrix}$$

(alt varianten utan minustecken). Detta beräknas i bildplanet, som

$$-b_s = b[s, t] - b[s+1, t]$$

$$-b_t = b[s, t] - b[s, t+1]$$

$$1$$

Normering görs, varefter denna information placeras i en textur, dock packad för att ligga i intervallet [0..1]:

$$R = (x+1)/2$$

$$G = (y+1)/2$$

$$B = (z+1)/2$$

Detta görs i förväg, sparas i en textur. Observera att normeringen av normalvektorn garanterar att alla komponenter är i intervallet [-1..1] så att det garanterat kan lagras. Denna procedur genererar en normalkarta från en bump map i stil med figuren nedan.

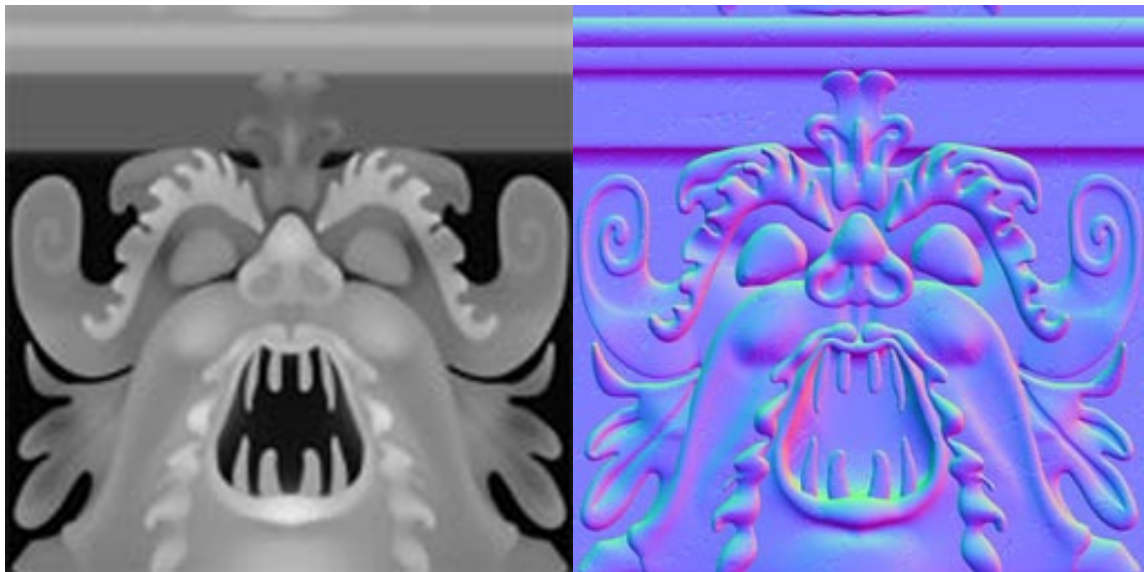


FIGURE 2. En av de vanligaste bump mapparna (vänster) och dess normal map (höger).

Vid rendering packas detta upp med

$$x = 2R - 1$$

$$y = 2G - 1$$

$$z = 2B - 1$$

vilket ger oss normalvektor i texturkoordinater som $\mathbf{n}_t = (x, y, z)$.

Ljusriktningen transformeras till texturkoordinater

$$\mathbf{L}_t = M_{vt} \cdot \mathbf{L}$$

Därmed har vi normalvektor och ljusriktning i samma koordinatsystem, varvid ljussättning kan utföras efter önskad formel.

3. Parallaxmappning

Nästa steg är uppenbart parallaxmappning. I *So How Can We Make Them Scream?* presenteras principen, att vi utläser bumpmappens höjd och tar ett steg längs ytan/bumpmappen längs betraktningsriktningens projektion på ytan.

Detta är enkelt nog i princip. Det gäller bara att hålla reda på koordinatsystemen igen.

Betraktningsriktningen fås i vertexshadern som vertexpositionen. Den kan sedan interpoleras med en varying-variabel så vi får den fragmentvis. Denna interpolerade version kallar vi \mathbf{V}_v .

Transformera betraktningsriktningen till texturkoordinater, med M_{vt} . Den resulterande vektorns st-komponent ger oss en offset i texturplanet som sedan kan användas.

$$\mathbf{V} = M_{vt} \cdot \mathbf{V}_v$$

Denna delas upp i en x,y-komponent (egentligen s,t) som vi betecknar \mathbf{V}_{xy} och z-komponent (egentligen n) som betecknas V_z .

Om motsvarande texturposition är $\mathbf{T}_0 = (i, j)$ så fås en justerad position som

$$\mathbf{T}_n = \mathbf{T}_0 \pm b(i, j) \cdot \mathbf{V}_{xy} / V_z$$

Plus eller minus ges om igen av huruvida bump mappen anger höjd över ytan eller djup ner i den.

Detta ger större offset ju närmare ytan vi är. En variant, kallad *offset limiting*, tar helt enkelt bort nämnaren:

$$\mathbf{T}_n = \mathbf{T}_0 \pm b(i, j) \cdot \mathbf{V}_{xy}$$

Detta gör texturen "plattare" i branta vinklar, vilket verkar huvudlöst tills man betänker alternativet. Det är i branta vinklar som parallax mapping ger allra störst fel. När vi plattar till i stället för att överdriva felen så blir resultatet mindre störande.

4. Slutsatser

Med detta supplement hoppas jag ha givit en mer komplett bild av problemet och dess lösningar. Jag har valt att inte gå djupare in på reliefmappning och displacementmappning. Dessa ligger på en solidare grund med detta som bakgrund och är därmed indirekt stärkta.

5. Referenser

Jag har konsulterat följande källor:

Astle, More OpenGL Game Programming

Angel, Interactive Computer Graphics, 5th ed

Hearn&Baker, Computer Graphics with OpenGL, 3rd ed

Game Programming Gems 1