3D projekt
DV1541

Mattias Lunderot **940815-7592**
Jonas Andersson **940527-4417**

Height-map terrain rendering, user can walk on the terrain.

Jonas Andersson

För att uppfylla denna punkt krävdes förutom att själva terrängen skapas och laddas korrekt också att det finns en kamera som kan röra sig i världen. För att kameran ska kunna röra sig krävs input från tangentbordet och även en timer som används för att räkna ut acceleration och deceleration.

Jag började med att skapa terrängen, först som en platt grid av trianglar. Då bestod en vertex bara av xyz-värden. Det som behövdes efter det var själva terrängen i form av en heightmap. Denna heightmap är av formatet Bitmap, som håller värden mellan 0-255 och information om bredd och höjd. Denna data läses in i en array och används för att skapa terrängen.

Ä heightmapen 256x256 i storlek så är även terrängen det. En array av vertiser med storlek 256x256 skapas där höjdvärdet på varje vertex mappas med värdet i heightmapen på motsvarande position.

Terrängen ritades nu ut som en grid fast med höjdvärden satta på varje vertex utifrån heightmapen.

Det som behövdes därefter var en textur, och då också texturkoordinater. En vertex består nu av xyz-värden och uv-koordinater. Dessa räknas ut genom att ta vertisens x och z värde och få dessa till texturkoordinaternas [0,1] range. Ligger vertisen i 128,128 blir uv koordinaterna (0.5,0.5), ligger den i 128,256 blir de (0.5, 1) osv. Värdena är alltså direkt mappade till 1/terrängens bredd och höjd. För att lägga till texture repeat ändrar man ettan mot hur många gånger man vill att den ska upprepas, dvs om den ska upprepas 8ggr blir det 8/bredd (då terrängen är kvadratisk spelar det ingen roll om det delas på bredd eller höjd). Värdet som läggs till på uv-coordinaterna mellan varje vertex måste också ändras därefter till att vara bredd/8.

Terrängen ritades nu ut med en textur som upprepas med ett givet antal upprepningar.

För att ljusberäkningar ska kunna göras på terrängen måste varje vertex också ha en normal. Denna fås genom att först räkna ut normalen för varje face, alltså varje triangel i griden för terrängen. Detta görs enligt principen med vektorerna (p1-p0) och (p2-p0) och sedan ta kryssprodukten mellan dessa.
Slutgiltiga normalen för varje vertex fås genom att ta medelvärdet var alla face-normaler som vertisen i fråga är en del i.
Terrängen ritas nu ut som den ska, med höjd från heightmap, texturkoordinater och normaler.

För att kunna röra sig i terrängen krävdes en kamera och input från tangentbord och mus. Input från tangentbordet fås genom att helt enkelt kolla tillståndet på de tangenterna som är av intresse. Med musen läses också tillståndet av och jämförs med tillståndet förra framen för att få reda på nuvarande position. Denna position använd för att räkna ut rotationen av kameran.

Kameran skapas med en View och en Projection matris. View innehåller kamerans position, rotation, en vektor som pekar uppåt och en som pekar i kamerans riktning, alltså vart kameran ska fokusera. Denna matris uppdateras varje frame med ny position och rotation som fås när man använder tangenterna för att gå runt och musen för att rotera.
Projection innehåller Field of View, aspect ratio, near och far plane.

För att kameran ska kunna gå på marken används en GetY funktion. Denna funktion hämtar y-värdet på de fyra hörnen i den quad som xz-värdet mappar till, sen interpoleras detta med hjälp av bilinear interpolation för att få reda på hur mycket varje hörns höjd ska räknas till slutgiltiga höjden. Den returnerar sedan detta höjdvärde och används för att sätta kamerans y-värde.

Back-face culling using geometry shader

Jonas Andersson

Efter att terrängen ritades ut korrekt och kameran fanns och kunde röra sig i världen var denna punkt väldigt lätt att lägga till. Terrängen ritas ut med en geometry shader som tar emot en data i form av trianglar och kamerans position i en constant buffer.

Två vektorer skapas, v1 från p0 till p1 och v2 från p0 till p2. Kryssprodukten mellan v1 och v2 ger triangelns normal. Sedan räknar man ut kamerans riktning genom att skapa en vektor mellan kamerans position och triangelns position. Om skalärprodukten mellan triangelns normal och kamerans riktning är större än 0 betyder det att triangeln är riktad från kameran.

Deferred rendering and lighting

Jonas Andersson

För att lägga till deferred rendering behövdes det bara skapas två nya render targets, ett för att rita ut färgen till (själva scenen) och ett för att rita ut normalerna till, som används i lighting-steget för ljussättning.

Texturerna skapas genom att varje typ av shader ritar ut till två render targets i pixel shadern. Alltså shadern som används för att rita ut terrängen med blend map skapar sitt resultat och ritar ut till de två render targets. Sedan när objekten ska ritas ut utan tex. Blend map aktiveras deras shader där pixel shadern också ritar ut sitt resultat till dessa render targets.

I lighting-steget samplas dessa två texturer. Ljusberäkningar görs för ljusen i scenen baserat på normalerna från normaltexturen och läggs ihop med samplingen från texturen för färgen och ritas ut till quad-objektet som sedan ritas ut på skärmen.

Shadow maps

Jonas Andersson

När deferred rendering och lighting var implementerat var det ganska enkelt att lägga till denna punkt.

Ett nytt render target lades till som används för att rita ut de skuggade objektens position innan WVP-transformeringen. Det görs för att kunna sampla och jämföra med depth-mapen för skuggorna.

Denna depth map skapas genom att en depth buffer aktiveras utan något render target, sedan ritas alla objekt med skugga ut med en shader som bara består av en enkel vertex shader som transformerar input-position med modelWorld * lightView * lightProjection. Därifrån genereras en depth map för dessa positioner.

I pixel shadern i lighting-steget används nu dessa två nya texturer, världspositioner och shadow map, till att räkna ut skuggorna. Världspositionerna samplas och transformeras med lightView * lightProjection för att sedan kunna jämföras med samplade värden i depthmapen för skuggorna.

Detta läggs sedan ihop med tidigare ljusberäkningar för slutresultat.

Picking

Jonas Andersson

Eftersom vi har valt att ha muspekaren låst till mitten av skärmen blir denna punkt lite enklare eftersom muspekarens position alltid kommer vara (0,0) och behöver därför inte göras om med Projection-matrisen.

Varje objekt som ska gå att klicka på innehåller en egen intersection sphere som är definierad av center-position i världen och radie. Det är mot denna som intersektionstest görs.

En vektor mellan positionen (0,0,0) och (0,0,1) skapas, alltså en som pekar rakt in i skärmen. Denna transformeras med inversen av View matrisen för att hamna i world space.

Därefter görs ett test (samma som i labb 1) mot varje intersection sphere hos objekt man kan klicka på. Vid testet returneras avståndet till objektet där intersektion sker, och av alla objekt som har intersektion väljs det som har kortast avstånd.

Parsing and rendering of an existing model format

Mattias Lunderot

Jag valde att ladda in OBJ filer till projektet. Filformatet är ganska enkelt och kan läsas in som text, rad för rad.

Om den känner av att ”v” är först på raden så läser den in en position, alltså tre stycken flyttal. Om den läser in ”vn” är det en normal som också är tre stycken flyttal. Om det är ”vt” är det en texturkoordinat med två flyttal. All data som kommer in konverteras ifrån höger till vänster-koordinatsystem för att det ska fungera med DirectX.

Till sist är ”f” som betyder att programmet ska sätta ihop data den samlat in innan för att skapa en vertexpunkt. Först läses index för positionen för vertexpunkten in. Sedan kollar programmet om den kan läsa in texturkoordinat och samma för normalen. Detta görs tre gånger för att läsa tillräcklig data för att skapa en triangel. Alla vertexpunkter kommer till slut att ligga i en stor lista som sedan används för att skapa en vertex buffer.

Material laddas också in från filen om den känner av att ”mtllib” är i filen. Filnamnet läses in och sedan läses en textur in om den hittar den.

Texture loading

Mattias Lunderot

Texturer laddas in från ett enkel RAW fil där färgerna är lagrade i RGBA format så en pixel tar fyra byte totalt. Programmet mäter först hur stor filen är och får fram bredden och höjden på texturen. Jag valde att programmet antar att texturen alltid har samma bredd som höjd. Sen laddas texturen över på grafikkortet.

Blend-mapping

Mattias Lunderot

För att få blend-mapping att fungera så laddas fyra texturer in, tre stycken är de texturer som kommer ritas ut i resultatet och den fjärde är själva blend-map texturen. Alla texturer skickas sedan till pixel shadern som sedan färgar beroende på vad värdet i blend-map:en är. T.ex. om värdet är 100% röd så kommer shadern visa en färg som är bara den första texturen som syns. Om värdet på blend-map:en är 50% röd och 50% blå kommer färgen bli en kombination av den första och andra

texturen. Detta gör att man får fina övergångar mellan olika texturer och det blir inga skarpa kanter mellan texturer. Det finns också ett repetitionsvärde och den anger hur många gånger texturen ska upprepa på hela terrängen.

View frustum culling against a quadtree

Mattias Lunderot

Jag valde att använda ett separat program för att generera quadträdet först för att sedan spara till en fil. För att generera ett quadträd måste programmet ha en lista med modeller och världsmatriser - det laddas in från filen modellist.txt. Programmet laddar in modellerna från en obj-fil och försöker sätta in modellen in quadträdet. Om modellen får plats i den nuvarande noden så kommer den testa om den får plats i deras barn. Sedan upprepas detta rekursivt till modellen har nått sin minsta omslutande box eller maxdjupet har uppnåtts.

När alla modeller har lagts in i trädet så sparas den till fil. Storleken skrivs ut och sedan skrivs noderna ut rekursivt.

Formatet är ett enkelt textformat. Om första bokstaven är ”o” så indikerar det att en modell ska läggas till på den nuvarande noden. Om bokstaven är ”c” så betyder det att noden har ett barn och siffran efter indikerar vilken kvadrant barnet är. Programmet skriver sedan ut barnet rekursivt. Bokstaven ”e” betyder att den nuvarande noden inte har mer data.

Programmet kan sedan läsa in filen på nästan samma sätt, också det sker rekursivt.

När rendering sker så räknar programmet ut sex olika plan med hjälp av view och projection matrisen. Dessa plan är precis på kanten mellan vad kameran kan och inte kan se.

Programmet räknar sedan ut åtta punkter som bildar en box, som ska testas mot de olika planen. Om alla punkterna ligger bakom samma plan kan vi konstatera att kameran inte kommer se något av vad som ligger innanför boxen och behöver då inte rendera modellerna som ligger innanför boxen. Programmet gör sedan samma för nodens barn rekursivt.

Particle system with billboarded particles

Mattias Lunderot

Programmet skickar in en textur, kameraposition och partikelposition till en geometry shader. Shadern konstruerar sedan en vektor mellan partikelpositionen och kamerapositionen. Sedan tar man kryssprodukten mellan en nya vektorn och vektorn som pekar uppåt och får ut en vektor som alltid pekar till höger och partikeln kommer alltid att vara vinklad mot kameran så man alltid ser den. Genom att lägga ihop vektorn som pekar uppåt och den som pekar höger multiplicerat med en skalär, så får man fram fyra olika punkter som skapar två trianglar.