

Environment Mapped Bump Mapping

In diesem Demo wird eine einfache Szene mit einem Objekt (zurzeit ein Würfel, kann aber durch ein beliebiges anders Objekt ausgetauscht werden) dargestellt. Auf das Objekt wird eine Normalmap als Textur angewendet, welche die Oberflächendetails bereitstellt. Das Objekt reflektiert die Umgebung und die Normalvektoren aus der Normalmap werden verwendet um den Reflexionsvektor zu modifizieren.

Die Umgebung wird durch eine vorgefertigte Cubemap Textur dargestellt, welche direkt in eine OpenGL Cubemap geladen und über die Geometrie eines Würfels dargestellt wird.

Das Objekt selbst wird mittels eines Vertex und Fragment Shaders dargestellt. Der Vertex Shader leitet die 3 Basis Vektoren für den Tangent-Space direkt an den Fragment Shader weiter. Der Bitangent Vektor wird dabei im Vertex Shader aus dem Tangent und Normalvektor berechnet (<http://www.terathon.com/code/tangent.php>). Der Vertex Shader berechnet auch den View-Vektor, welche ebenfalls an den Fragment Shader für weitere Berechnungen weitergeleitet wird.

Der Fragment Shader ist sehr einfach. Die interpolierten Tangent Space Basisvektoren werden normalisiert (um größere Fehler zu vermeiden). Aus der Normalmap Textur wird ein Sample gelesen, welches eine Normale repräsentiert (diese wird nicht normalisiert, obwohl dies eventuell die Qualität erhöhen kann). Die gelesene Normale wird mit der Tagnent Space Matrix transformiert um die Normale im Worldspace zu erhalten. Mit dem View-Vektor und der Normalen kann mit einem einfachen reflect() der Reflexionsvektor berechnet werden, welcher direkt als Index in die Environment-Cubemap dient.

Screen Space Ambient Occlusion

In diesem Demo versuche ich einen Ambient Occlusion Effekt ausschließlich aus Screen Space Daten zu erzeugen. Die Firma Crytek hat diesen Effekt in ihrem Spiel Crysis eingesetzt, doch sind die genauen Implementierungsdetails nicht bekannt, deshalb musste ich etwas probieren.

Als ersten Schritt rendere ich die Szene in eine Textur wobei ich die linearen z-Werte dort abspeichere (Im Kamera Raum). Dafür könnte man in DirectX eine Float Textur mit nur einer Komponente verwenden. Das gestaltet sich in OpenGL etwas schwieriger, es dort nur Texturformate mit mehreren Komponenten gibt in welche man auch rendern kann. (ausgenommen Herstellerspezifische Extensions). In meinem ersten Versuch habe ich hier einfach eine RGB16F Textur verwendet, aber da man die z-Werte auf den Bereich $[0, 1]$ normalisieren kann ist es auch möglich eine RGBA8 Textur zu verwenden und die Werte entsprechend zu kodieren (das mache ich jetzt und es ist schneller als mit einer Float Textur).

Nachdem man die linearen z-Werte in einer Textur abgespeichert hat kann man daraus die SSAO-Werte berechnen. Dazu wird ein Fullscreen Quad mit einem speziellen Fragment Shader gerendert. Für jeden Pixel wird der z-Wert aus der Textur gelesen und mit den z-Werten der Pixel in einem bestimmten Radius verglichen um einen Block-Faktor zu errechnen. Damit ein wenig Zufall in das ganze kommt verwende ich eine 4x4 Noise-Textur, um die Sample Positionen für die einzelnen Pixel zu randomisieren (Crysis macht das auch). Für genauere Details bitte den Shader Code ansehen. Am Ende erhalte ich einen Wert im Bereich $[0, 1]$. Gerade Flächen erhalten dabei ungefähr einen Wert von 0.5.

Da das ganze Resultat ziemlich verrauscht und deshalb noch unbrauchbar ist wende ich einen Blur-Filter darauf an. In meinem Fall tut das eine 6-pass Kawase Bloom Filter.

Als letzten Durchgang zeichne ich die Szene ganz normal und wende eine einzelne Lichtquelle (Richtungslicht) auf die Szene an. Die Berechnete SSAO Textur wird mittels Projektive-Texturing auf die Szene gemappt. Der Wert aus der SSAO Textur wird noch in den richtigen Wertebereich gebracht (von $[0, 1]$ in $[0, 2]$ und dann geclampt; weil 0.5 -> kein Occlusion) und direkt als Ambient verwendet (auch noch mal gewichtet).

Software Rendering mit Demo

Mein drittes Projekt umfasste die Implementierung eines Software Renderers, welcher die gesamte Rendering Pipeline wie in der Vorlesung besprochen implementiert. Dazu habe ich auch eine Demo entwickelt welche die Fähigkeiten des Software Renderers darstellt.

Der Software Renderer sollte auch auf der GP2X Konsole (Linux basiert auf ARM mit 200Mhz Prozessor) laufen und ausreichend Performance liefern. Da auf dem ARM Prozessor keine nativen Floating-Point Operationen zur Verfügung stehen und Software Emulation zu langsam ist kommt der Renderer mit Fixed Point Arithmetik aus.

Die Vertex Transformationen können durch einen in C++ programmierten Vertex Shader beeinflusst werden. Der Renderer übernimmt das Clipping in homogenen Koordinaten und macht auch den „Perspective Divide“ und „Viewport Transform“ (siehe OpenGL Spezifikation). Der Rasterizer kann Punkte, Linien und Dreiecke mit beliebig vielen Interpolanten rasterisieren. Bei Dreiecken kann neben der affinen Interpolation der verschiedenen Parameter auch die perspektivisch Korrekte Interpolation gewählt werden. Was mit den einzelnen Fragmenten geschehen soll kann durch einen in C++ programmieren Fragment Shader bestimmt werden.

Das zum Renderer gehörende Demo zeigt Effekte wie Point rendering, Line rendering, Lighting, Texture Mapping, Environment Mapping und Additive Blending.