Texture-Mapping

Marc-Oliver Pahl
3. Semester
Wilhelm-Schickard-Institut für Informatik

Betreuer: Dr. Andreas Schilling

mop November MM

Inhalt:

Warum braucht man Texture-Mapping?

Wie schon gehört, ist die **Realität zu komplex**, um sie in Echtzeit mit heutigen Computern originalgetreu nachzubilden.
Das Texture-Mapping ist ein Verfahren, um mit **möglichst geringem Aufwand komplexe Oberflächen** im Computer nachzuempfinden.
Als Beispiel kann man eigentlich fast jedes 3D-Objekt heranziehen, das eine mehrfarbige bzw. strukturierte Oberfläche hat.

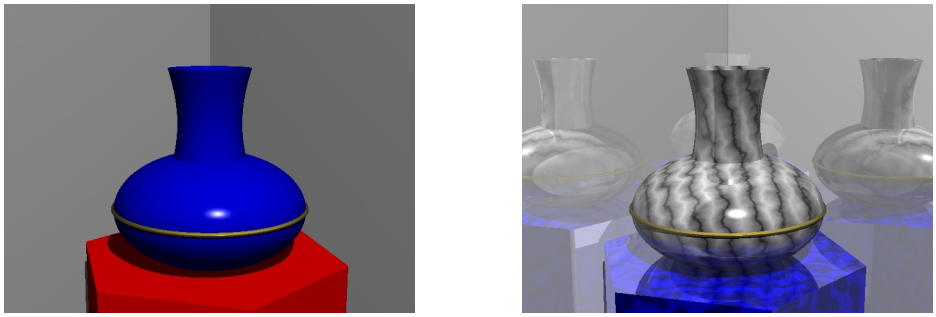


Abb. 1

Hier wurde z.B. die Vase mit einer Marmortextur überzogen, was sie gleich um einiges realistischer erscheinen läßt.

Was ist Texture-Mapping?

Was eigentlich eine Textur sein kann, wird im Folgenden deutlich werden und Mapping (engl. Abbildung) bedeutet nichts anderes, als dass die **Textur auf das Objekt abgebildet** wird; auch Verfahren dazu werden im Folgenden erläutert werden.

Was ist eine Textur?

Es gibt zwei Arten von Texturen: Die diskreten und die prozeduralen Texturen.
Diskrete Texturen sind grundsätzlich erstmal ein **Array**, im zweidimensionalen Fall also als Matrix darstellbar.

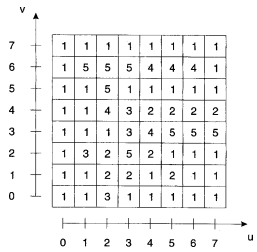


Abb. 2

Die Einträge der Matrix (TEXture-ELement kurz Texel genannt) können je nach Verwendungszweck verschieden sein.
Hier sind es nur Zahlen, die etwa für Grauwerte oder Höhen am jeweiligen Texturpunkt stehen könnten. Meistens werden aber anstelle der Skalare, wie hier, Vektoren stehen, die dann z.B. den RGB-Farbwert der Textur am entsprechenden Punkt darstellen.

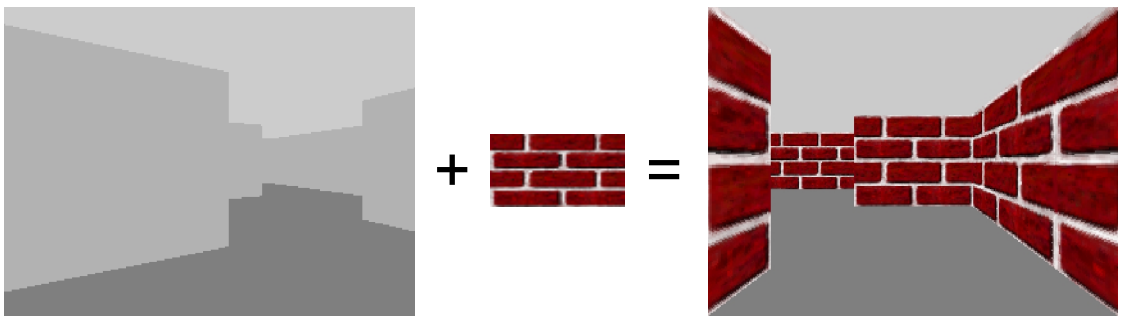


Abb. 3

Im Beispiel wird das „Foto“ des Wandstücks auf die graue Fläche gelegt, wodurch die ganze Szene gleich wesentlich „realistischer“ wirkt.
Wie schon angedeutet, müssen die Texturen nicht zweidimensional sein. Bei einer Holztextur z.B. wäre es schlecht -weil nicht realistisch-, wenn das texturierte Objekt von oben und im Querschnitt dieselbe Texturierung hätte; in solch einem Fall müsste man entweder auf zwei zweidimensionale Texturen (z.B. eine längs- und eine quergemaserte (Ringe)) zurückgreifen oder gleich eine dreidimensionale Textur, die dann auch die Farbinformationen gewisser Tiefenschnitte enthält, verwenden. (Eine Computer-Tomographie liefert z.B. eine dreidimensionale Textur.)

Die **prozeduralen Texturen** bestehen dagegen nicht aus einem Array sondern aus einer **Funktion**, mit deren Hilfe sich z.B. die Einträge der Matrix auf der linken Seite durch Einsetzen der Koordinaten errechnen ließen.

Beide Arten haben Vor- und Nachteile, doch um diese zu verstehen, ist es zuerst notwendig, zu wissen, wie das Texturieren überhaupt funktioniert.

Wie funktioniert Texture-Mapping?

Wie man schon in Abb. 3 sieht, ist die gegebene Textur (der Texturraum) zweidimensional, soll aber in die dreidimensionale Szene (Objektraum) eingebaut werden.
Würde man die Textur einfach so auf die gewünschte Fläche legen, würde das Bild wie auf der linken Seite aussehen.

Abb. 4

Die Textur muß also perspektivisch korrekt auf die Fläche „aufgebracht“ werden (rechts). Man benötigt also eine Funktion, die zu gegebenen „echten“ dreidimensionalen Koordinaten die entsprechenden Texturkoordinaten liefert:

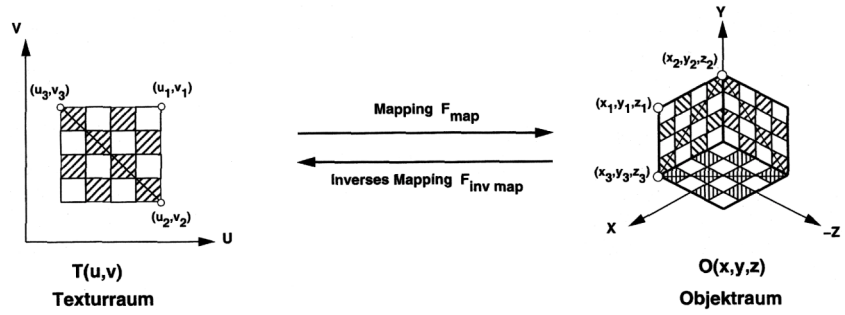


Abb. 5

Diese Aufgabe erfüllt die **inverse Mapping**-Funktion $F_{inv\ map}(x, y, z)=(u, v)$ (Bsp. für zweidimensionale Texturen).

Wenn man das Ermitteln des Texels zu den Texturkoordinaten (u, v) (also z.B. das „Nachschauen“ in der Matrix) allgemein mit der Funktion $C_{tex}(u, v)=(r, g, b)$ beschreibt, stellt folgende Abbildung die **Texturierung** dar:

$$\mathbf{C_{tex}(F_{inv\ map}(x, y, z))=(r, g, b)}$$

Die Formel gilt natürlich auch für die dreidimensionalen Texturen. C_{tex} wird dann ebenso wie $F_{inv\ map}$ dreistellig: $C_{tex}(u, v, w)=(r, g, b)$; $F_{inv\ map}(x, y, z)=(u, v, w)$.

Von der Stelligkeit her bräuchte man die inverse Mappingfunktion bei dreidimensionalem Textur- und Objektraum mathematisch nicht, dann würde aber dasselbe wie mit der Mauer im zweidimensionalen passieren: Die Perspektive wäre völlig unberücksichtigt und damit zumeist falsch.

Wie kann so eine Mapping-Funktion aussehen?

Flächen, die texturiert werden sollen, lassen sich zumeist durch Polygone annähern. Diese Polygone kann man wiederum in Dreiecke zerlegen und deren Texturierung wird im folgenden beschrieben.
Das Beispiel geht erst den Weg vom Textur in den Bildraum; letztendlich wird aber der umgekehrte ($F_{inv\ map}$) Weg gebraucht, da aber eine Matrix als Mapping-Funktion herauskommt läßt sich diese dann einfach Invertieren.

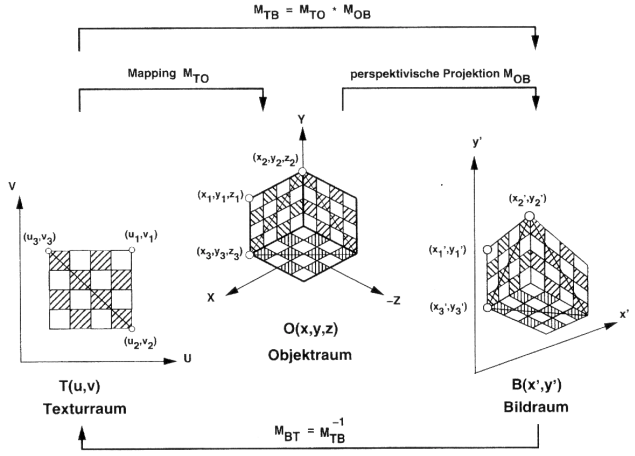


Abb. 6

Als ersten Schritt gilt es also vom Texturraum in den Bildraum zu gelangen. Das geschieht mit folgender Matrix:

$$\begin{aligned}[x, y, z, 1] &= [u, v, 1] \cdot \mathbf{M}_{TO} \\ &= [u, v, 1] \cdot \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 1 \end{bmatrix}\end{aligned}$$

Abb. 7

Durch Einsetzen der entsprechenden (u, v)-Koordinaten gelangt man also zu den korrespondierenden Koordinaten im Objektraum.
Der Bildschirm, auf dem das fertige Bild dann dargestellt werden soll ist aber nicht dreisondern nur zweidimensional, weshalb man noch die Abbildung in den Bildraum benötigt:

$$\mathbf{M}_{OB} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{1}{z_0} \\ 0 & 0 & 1 \end{bmatrix}$$

Abb. 8

Die fertige Abbildung ist nun also $M_{TB}=M_{TO} * M_{OB}$:

$$\mathbf{M}_{TB} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & 1 \end{bmatrix}$$

Abb. 9

Im Allgemeinen wird aber nicht die Mapping-Funktion, sondern ihre Inverse von Bedeutung sein. Dazu gilt es einfach noch, die Matrix zu invertieren. $M_{BT}=M_{TB}^{-1}$.

Das fertige inverse Mapping funktioniert nun also folgendermaßen:

$$[u', v', q'] = [x', y', 1] \cdot \mathbf{M}_{BT}$$
$$u = \frac{u'}{q'}, \qquad v = \frac{v'}{q'}$$

Abb. 10

Mithilfe der Matrix müsste man nun eigentlich zu jedem Pixel das zugehörige Texel bestimmen.

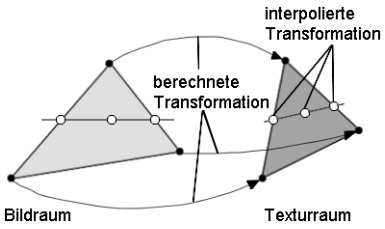


Abb. 11

Da aber eine dreieckige Textur auf ein Dreieck im Bildraum abgebildet wird, lassen sich die Texturwerte zu den Bildpunkte auch durch ihr Verhältnis zu den drei Eckpunkten bestimmen. Würde man das linear machen -also im Texturraum z.B. ein Texel weitergehen, wenn das nächste Pixel des Bildraums gesucht wird- erhielte man natürlich wieder ein Bild, wie in Abb. 4 links, in dem die Perspektive nicht berücksichtigt ist.
In das $q^{|}$ (Abb. 10), mit dessen Hilfe die perspektivische Interpolation durchgeführt wird, gehen daher nicht nur der Abstand der Bildpixel, sondern auch die Tiefen (z-) Informationen -also der Abstand des Pixels zum Betrachter- ein. Damit ist es dann möglich durch einfachere Rechnung (drei Additionen und zwei Divisionen) die Texel zum entsprechenden Pixel zu ermitteln, ohne die Matrizenrechnung jedesmal durchzuführen.

Wird die Texturierung im Zuge der Erschaffung einer neuen Szene verwendet, so ist im Allgemeinen natürlich schon bekannt, wie sich die Koordinaten transformieren. Auf den Umweg über die Dreiecke kann dann (wenn die Szene nicht sowieso aus Dreiecken besteht) verzichtet werden.

Probleme mit diskreten Texturen

Jetzt, wo das Verfahren der Texturierung näherungsweise erläutert ist, werden auch schon **Probleme** erkennbar, die sich für **diskrete Texturen** ergeben:
Das erste Problem ergibt sich aus der inversen Mapping Funktion. Die Texturfunktion C_{tex} ist **nur für bestimmte Werte** (in Abb. 2 für alle $(u, v) \in 7 \times 7$) **definiert**, weil einfach nicht mehr Daten vorliegen. Die inverse Mappingfunktion wird aber nur in den seltensten Fällen z.B. gerade Koordinaten liefern, woher soll man also die Daten nehmen?

Man könnte einfach genauere, also detailreichere Texturen verwenden. Mit steigendem Detailreichtum steigt aber auch die Datenmenge linear an und auch der heute vergleichsweise riesige Grafikkartenspeicher ist endlich. Außerdem wird die Textur nur selten im Größenverhältnis 1:1 gebraucht werden und für kleinere oder gößere Skalierungen sind dann immer noch keine exakten Daten vorhanden.

Der einfachste Lösungsansatz, Farbwerte zu nicht bekannten Texturkoordinaten zu erhalten, besteht darin, den nächsten mit der C_{tex}-Abbildung definierten Punkt zu den gesuchten Koordinaten zu ermitteln und dessen Wert zu verwenden. Dass dies nicht zu akzeptablem Ergebnis führt ist klar:

Abb. 12

Liegt bei diesem schwarzweißen Punktmuster ein gesuchter Wert etwas näher bei einem weißen Feld wird er weiß, sonst schwarz. Damit geht die Struktur verloren.

Einen besseren Weg stellt die **bilineare Interpolation** dar. Dabei werden die dem gesuchten Punkt nächsten vier Farbvektoren, gemäß ihres Abstandes gewichtet, zu einem neuen Farbvektor addiert:

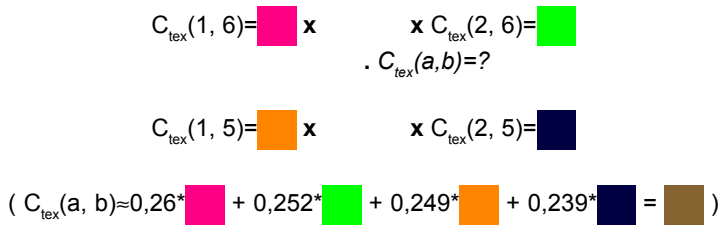


Abb. 13

Auch dieses Verfahren ist noch recht einfach, löst aber leider in der Praxis ebenfalls nicht das Problem, das durch die perspektivische Projektion entsteht:
Welche Farbe soll dargestellt werden, wenn in einiger Entfernung zum Augpunkt (dem Punkt von dem aus wir die Szene sehen) auf ein Pixel sehr viele verschiedenfarbige Texel fallen.

Außerdem gibt es Probleme, wenn sich das texturierte Objekt bewegt. Dann findet nämlich die Interpolation ständig zu neuen Punkten statt und eine Fläche, die eigentlich nur gedreht wurde hat plötzlich eine andere Farbe.

Im Übrigen ist es auch noch ineffektiv, für entfernte Objekte, die sowieso nicht genau zu sehen sind, eine große, detailreiche, Textur im Grafikspeicher zu halten, obwohl die Details gar nicht gebraucht werden.

Ein weiteres Problem, das sich aus der Endlichkeit der diskreten Texturen ergibt: **Was wenn die Textur zuende** ist, der zu füllende Platz aber noch weitergeht?
Bei dem Beispiel mit der Mauer ist das Problem so gelöst, daß die Textur sich in alle Richtungen nahtlos an sich selbst fügen läßt und somit endlos verwendet werden kann, indem Breite und Höhe der Textur so lange von den aus dem inversen Mapping gewonnenen Koordinaten abgezogen werden, bis die Texturfunktion anwendbar wird.

Schließlich kann es noch vorkommen, daß die diskrete Textur ungewollt schon Licht- oder Schattenverhältnisse enthält (z.B. ein Foto als Textur). Widersprechen die Verhältnisse auf dem Bild dann z.B. den Lichtverhältnissen in der Zielszene, wirkt das ganze nicht mehr realistisch, sondern wie ein Bild oder Plakat.

Vor und Nachteile von prozeduralen Texturen

Bei derartigen Nachteilen liegt es auf der Hand, Texturen nicht als Vektorfelder sondern gleich als Funktionen zu speichern. Dann kann man für jeden benötigten Punkt die Farbwerte ausrechnen, da die Funktion auch für noch so große oder feine Eingaben Werte liefert und spart viel Speicherplatz. Solche als Funktion gespeicherten Texturen nennt man wie anfangs erwähnt **prozedurale Texturen**.
Dieses Verfahren wäre also eigentlich optimal, wenn es so einfach wäre, die gewünschte **Funktion** zu **erstellen**.
Auch mit viel Übung wird man es wohl nicht schaffen, ein Foto, das als Textur dienen soll, mit einer Funktion zu beschreiben.

Im Gegensatz dazu gestaltet es sich einfach, diskrete Texturen zu erschaffen, indem man z.B. Fotos oder anderes Bildmaterial verwenden oder sich Muster mit dem PC erstellt.

Probleme, die durch die perspektivischen Projektion entstehen: Aliasing

Wie bei den diskreten Texturen angesprochen, ist es ein Problem, festzustellen, welcher Farbwert genommen werden soll, wenn mehrere (viele) Texel auf ein Pixel kommen. Das ist das Problem des **Aliasing**. Es tritt beim Texture-Mapping fast immer auf, da die Textur mit größerem Abstand zum Betrachter zwangsläufig verkleinert werden muß:

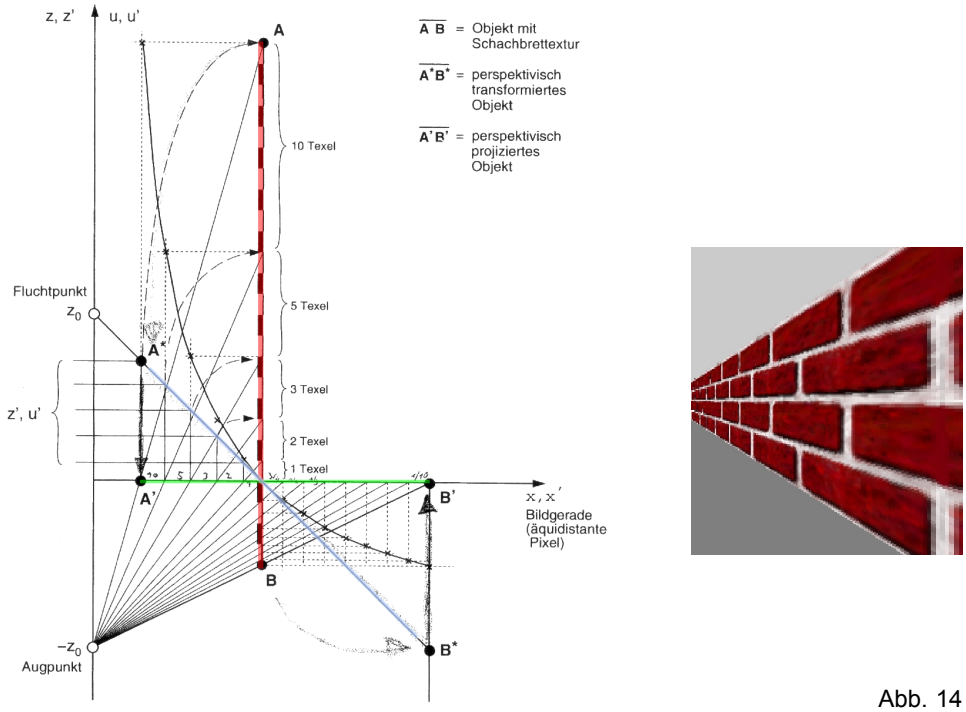


Abb. 14

Die Textur (rot) soll so dargestellt werden, als wäre sie -wie im Bild- an einer Wand rechts vom Betrachter (Augpunkt). Wie schon festgestellt, ist der Bildschirm aber flach und somit muß die Textur unter Berücksichtigung der Perspektive (siehe auch Abb. 4) auf den Bildraum (grün) projiziert werden.
Wie man an den Verbindungsgeraden sieht, müssen dazu aber ganz wenige (1/10 Texel rechts) bzw. ganz viele (10 Textel links) Texel mit einem Bildschirmpixel dargestellt werden.

„Der Computer weiß nicht, was unser Auge auf die Entfernung wahrnehmen würde."

Am einfachsten wäre es, das Raster, in dem die Texturwerte gelesen werden, gröber zu machen; also z.B. „Textur soll nur noch 1/10 groß sein" => es wird nur noch jedes zehnte Texel verwendet. Damit geschieht aber natürlich (vor allem bei kontrastreichen Texturen) folgendes:

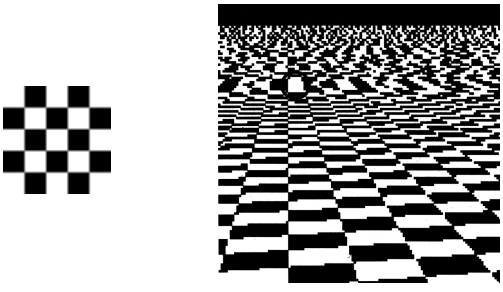


Abb. 15

Ist der zehnte Punkt zufällig gerade weiß, die neun danebenliegenden aber schwarz, so wird das Feld weiß, obwohl es schwarz sein müsste. Beim nächsten Feld verhält es sich dann genau umgekehrt.

So einfach geht es also nicht. Für eine realistischere Darstellung muß man den passenden Farbwert aus der Textur heraus**filtern**.
Verfahren dazu sind die Boxfilterung oder das Filtern mit der Gaußfunktion. Beide liefern zu einem aus mehreren Texeln bestehenden Gebiet ein neues Texel, das ungefähr dem entspricht, was unser Auge bei entsprechender Verkleinerung sehen würde.
Die Verfahren sind beide leider auch für heutige Computer noch zu rechenintensiv, um sie in Echtzeit durchzuführen. Deshalb müssen die Texturen vorgefiltert werden.

Filterung

Das verbreitetste Verfahren dazu ist das **Mip-Mapping** (MIP = multum in parvum = Vieles im kleinen). Hier werden in der Textur **verschiedene Detailstufen**, die im Voraus berechnet wurden, gespeichert:

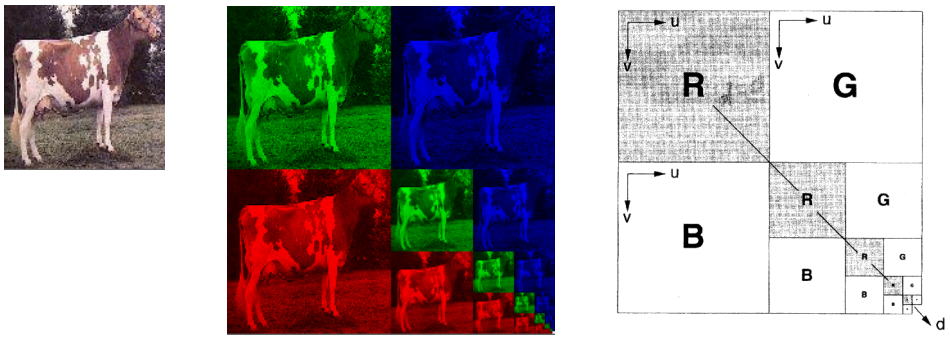


Abb. 16

Jeder Verkleinerungsschritt beträgt den Faktor zwei.

Bei der Texturierung wird nun die Detailstufe verwendet, in der das gesuchte Pixel definiert ist. Ist keine passende Stufe vorhanden wird nicht nur bilinear innerhalb der nächsten passenden, sondern **trilinear** zu den nächsten beiden Stufen (also zur „zu guten“ und „zu schlechten“) interpoliert.

Dieses Verfahren liefert ganz gute Ergebnisse, ein Problem liegt aber noch in der Geometrie der Textur: Beim Mip-Mapping ist sie quadratisch und auch die einzelnen Bildpunkte wurden quadratisch zu ihren Nachbarn heruntergerechnet.
Ein solcher quadratischer Bildpunkt wird aber im fertigen Bild zumeist nicht mehr quadratisch sein.

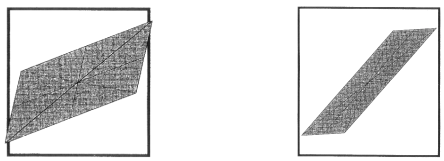


Abb. 17

Die Zurückprojektion eines quadratischen Bildpunktes auf die Textur wird Footprint (das graue) genannt. Beim Mip-Mapping wird nun, da keine anderen Daten vorhanden sind, der Farbwert des nächsten umhüllenden Quadrates verwendet.

Da setzt das **SAT-Mapping** (SAT = summed area table) an: Es verwendet keine Quadrate fester Größe, sondern Rechtecke beliebiger Dimension -die den Footprint natürlich genauer annähern- zu ermitteln.

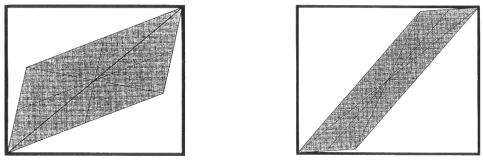


Abb. 18

Dazu werden nicht bestimmte feste Detailstufen gespeichert, sondern die Summen der Texel über alle bildbaren Rechtecke in der Textur. Also jeweils die Summen im Rechteck (0, 0) - (u, v):

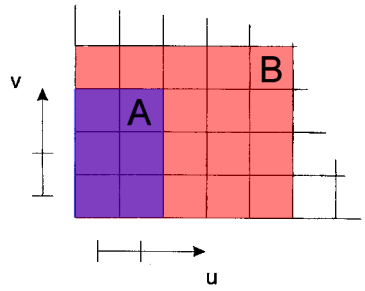


Abb. 19

In A steht dann also die Summe $C_{tex}(0, 0)+C_{tex}(0, 1)+C_{tex}(1, 0)+...+C_{tex}(2, 1)$ (lila). In B entsprechend $B=\Sigma_{u<4,\ v<3}\ C_{tex}(u, v)$ (rot und lila).

Den Texturwert für ein beliebiges Rechteck in der Textur erhält man nun durch vier Additionen und zwei Divisionen:

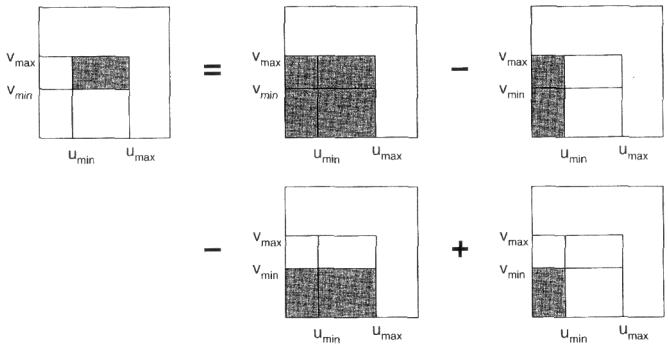


Abb. 20

Summenglieder sind jeweils die Werte in den oberen rechten Ecken. Das Ergebnis muß dann noch durch die Anzahl der verwendeten Texel (Breite x Höhe) geteilt werden.

Dieses SAT-Mapping ist genauer, aber es geht noch besser. Problem ist immer noch der zu große Weißraum zwischen Footprint und verwendetem Rechteck. Bei einer perspektivischen Projektion, wie sie beim Texture-Mapping geschieht, sind die Farbwerte aus bestimmten Blickrichtungen, die ja den Footprint ergeben, noch immer zu ungenau (Abb. 18 rechts).

Beim Elliptical Weighted Average (EWA)-Filter werden Ellipsen an Stelle der Rechtecke eingesetzt. Das schafft zwar bessere Ergebnisse, führt aber zu wesentlich erhöhtem Rechenaufwand und ist daher zur Echtzeitberechnung ungeeignet.

Einen Mittelweg zwischen EWA und Mip-Mapping stellt **Footprint-Assembly** (FPA) dar. Dazu wird die Textur ähnlich wie beim MipMap-Verfahren zerlegt und gespeichert. Um den Texturwert letztendlich zu ermitteln, wird aber nicht nur ein Quadrat berücksichtigt:

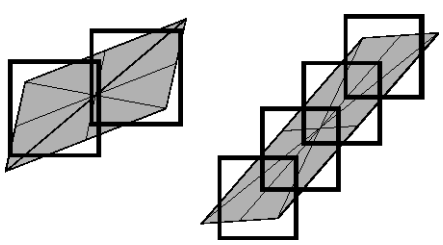
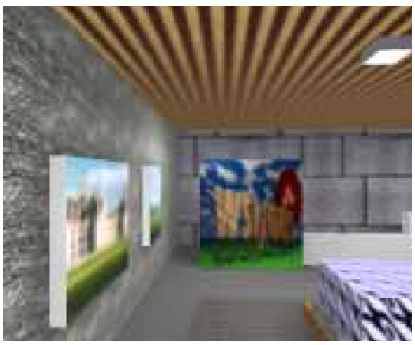
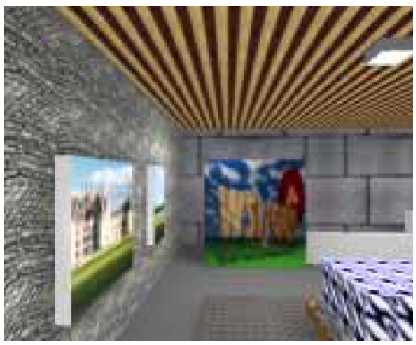


Abb. 21

In der Praxis wird der Footprint durch das arithmetische Mittel über 2^m Quadrate angenähert. Durch die Zweierpotenz läßt sich das Dividieren dann nämlich durch m in der Prozessorarithmetik eingebaute Shifts bewerkstelligen.



Standard Mip-Mapping



Footprint-Assembly

Abb. 22

Wie berücksichtigt man die Beleuchtung?

Mithilfe der beschriebenen Verfahren ist die Textur nun auf das gewünschte Objekt gelegt worden. In der fertigen Szene wird es meistens aber auch noch bestimmte Lichtverhältnisse geben, die sich auf die Textur auswirken sollten.

Am Häufigsten eingesetzt wir das Verfahren der Skalierung der Intensität **a posteriori**. Dabei werden zuerst mithilfe der Textur die Farbwerte für die Bildpunkte berechnet und anschließend mit dem zugehörigen Intensitätswert, den die Beleuchtungsrechnung ergibt, multipliziert.

Nutzen der Textur um Rechenzeit zu sparen

Wie Anfangs gesehen, ist eine diskrete Textur nichts anderes als eine Ansammlung von Zahlen, die bestimmten Ortspunkten zugeordnet werden.
Da in die Beleuchtungsrechnung auch feste Parameter (z.B. zum Material) eingehen, ist es unter Umständen sinnvoll, diese schon vorher zu berechnen und dann gleich die Ergebnisse als neue Textur abzuspeichern. Das spart dann bei der konkreten Berechnung der Szene Rechenzeit.

Auch Reflexionseffekte lassen sich a priori berechnen und als Textur speichern, die dann beim Rendern der Szene nur noch auf die Oberfläche gelegt werden muß.

Ebenso kann man Texturen verwenden, um die Lichtdurchlässigkeit eines Objekts an bestimmten Punkten zu speichern. So lassen sich z.B. Wolken oder milchige Glasscheiben darstellen, ohne komplizierte Objekte für das Raytracing definieren zu müssen. (Erinnerung: Beim Rendern einer Szene wird ein Sehstrahl vom Betrachter auf die Reise geschickt und je nachdem was er trifft die Szene moduliert. Durch die Textur kann ein Fenster als einfaches Rechteck moduliert werden. Die Durchlässigkeit ergibt sich dann aus der Textur).
Eine solche Textur kann man natürlich auch direkt vor die Lichtquelle legen und so komplizierte Lichteinstrahlung erreichen.

Mithilfe einer skalaren Textur kann man auch recht einfach dreidimensionale Oberflächen simulieren, indem die Textur z.B. die Reflexionseigenschaften für einzelne Punkte verändert (Bump-Mapping) oder auch generieren (z.B. VoxelSpacing), indem man eine skalare Textur jeweils als Länge eines Höhenvektors senkrecht zur Fläche interpretiert.

Schließlich ist es natürlich auch möglich, mehrere Texturen übereinander zu legen und so mit einfachen Texturen noch komplexere Oberflächen zu simulieren bzw. mit wenigen kleinen Texturen eine große Fläche überall unterschiedlich zu belegen (z.B. könnten man auf die Mauer (Abb. 14) noch eine zweite Textur legen, damit der Betrachter nicht andauernd am „gleichen Wandstück“ vorbeiläuft).

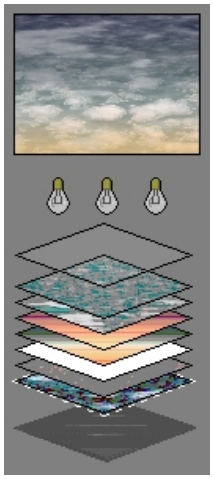


Abb. 23

Quellen:
J. Encarnacao, W. Straßer und R. Klein, *Graphische Datenverarbeitun*g, Band II, Oldenbourg, 1997 S.185-197, S.206-213, S.218f

http://wwwmayr.informatik.tu-muenchen.de/lehre/1998WS/proseminar/krause/
http://www.tu-chemnitz.de/informatik/HomePages/RA/kompendium/vortr_2000/3d_fkt/
http://www.tu-chemnitz.de/informatik/RA/kompendium/vortraege_97/grafik_3d/
http://wwwmath.uni-muenster.de/informatik/u/dollner/rtr00

Bildquellen:
Abb. 1: selbstgerendertes Pov-Ray-Beispielbild
Abb. 2: GDV II S. 219
Abb. 3: Microsoft-Windows Bildschirmschoner „Labyrinth“ Screenshot
Abb. 4: http://www.tu-chemnitz.de/informatik/RA/kompendium/vortraege_97/grafik_3d/pers.html
Abb. 5: GDV II S. 186 (leicht modifiziert)
Abb. 6: GDV II S. 193
Abb. 7-10: GDV II S. 192
Abb. 11: selbstproduziert
Abb. 12: http://www.gris.uni-tuebingen.de/gris/grdev/java/applets/texturemapping Applet Screenshot
Abb. 13: In Anlehnung an www.3dconcept.ch
Abb. 14: GDV II S. 95/ Screenshot Microsoft-Windows Bildschirmschoner „Labyrinth“
Abb. 15: GDV II S. 207
Abb. 16: http://www.gris.uni-tuebingen.de/gris/grdev/java/applets/texturemapping/hilfe/GuidedTour.html/ GDV II S. 208
Abb. 17: GDV II S. 209
Abb. 18: GDV II S. 211
Abb. 19: selbstproduziert
Abb. 20: GDV II S. 210
Abb. 21: GDV II S. 212
Abb. 22: GDV II S. 213
Abb. 23: Screenshot aus CorelTexture8
GDV II: J. Encarnacao, W. Straßer und R. Klein, *Graphische Datenverarbeitun*g, Band II, Oldenbourg, 1997