

1 Einleitung

Texturen sind ein fundamentales Mittel zur Erstellung von Modellen für die Darstellung in Bildern, Videos oder Spielen. Sie werden benötigt um die Oberflächendetails eines Modells darzustellen, ohne diese explizit über Geometrie oder Materialeigenschaften modellieren zu müssen [WLKT09]. Damit wird der Detailgrad eines Modells erhöht, ohne dabei den Detailgrad der zu Grunde liegenden Geometrie zu erhöhen. Das führt unmittelbar zu einer schnelleren Berechnung des darzustellenden Bildes und einer Einsparung von Speicherplatz.

Texturen beschreiben die visuelle Oberfläche eines Modells in einem (meist) rechteckigem Bild. Mittels eines Mapping-Verfahrens wird dieses Bild anschließend auf das entsprechende Modell projiziert [KNL⁺15]. Oft werden Texturen strikter definiert, nämlich als eine Darstellung einer Oberfläche mit sich wiederholenden Mustern [WLKT09]. Der Grad der Zufälligkeit dieser Wiederholung kann dabei üblicherweise für unterschiedliche Texturen variieren. Naturgegebene Texturen besitzen in der Regel auf Grund der Unvollkommenheit der Natur eine komplett zufällige Wiederholung von sich ähnlichen Mustern, wo hingegen Menschengeschaffenes oft auch eine deterministische Komponente besitzt. Ein Fliesenmuster besitzt z.B. eine zufällige Wiederholung in den Mikrodetails seiner Fliesen, die Platzierung der Fliesen bzw. deren anliegende Kanten unterliegen dagegen aber einer deterministischen Komponente [WLKT09].

Es kann äußerst schwierig sein, eine Textur zu generieren, dessen Dimensionen und Auflösung an eine bestimmte Modellaufgabe angepasst sind [KNL⁺15]. Das zeichnet sich insbesondere dadurch wieder, dass es einen speziellen Berufszweig in der Film- und Spielebranche namens *Texture Artist* gibt, der sich ausschließlich mit der Aufgabe befasst, qualitativ hochwertige Texturen zu kreieren [WLKT09]. Diese werden meistens über Handzeichnungen oder über handbearbeitete Fotografien erstellt. Handzeichnungen können zwar ästhetisch ansprechend sein, aber sie garantieren keinerlei Fotorealismus. Fotografien hingegen werden in einem beliebigem Grafikprogramm solange bearbeitet, bis sie den Anforderungen der Textur gerecht werden. Das führt in der Regel dazu, dass die Erstellung einer Textur sehr zeitaufwändig ist und sich dennoch auffällige Wiederholungen oder Unsauberkeiten in der Textur nicht vermeiden lassen. Ferner ist nicht jeder Mensch ein Künstler und es ist äußerst schwierig wenn nicht gar unmöglich ohne Vorkenntnisse oder künstlerische Fähigkeiten qualitativ hochwertige Texturen zu erzeugen [WLKT09]. Das Verfahren der Textursynthese widmet sich diesem Problem.

2 Textursynthese

Die *Textursynthese* ist ein alternativer Ansatz für die Erstellung von Texturen, der es erlaubt, auch ohne spezielle Vorkenntnisse, qualitativ hochwertige Ergebnisse zu erzielen. Der Benutzer muss lediglich ein Beispielmuster und eventuell benötigte Konfigurationsparameter an die Textursynthese übergeben, die aus diesen Daten eine Textur synthetisiert [WLKT09]. Die resultierende Textur zeichnet sich dadurch aus, dass sie eine beliebige Größe annehmen kann, aber weiterhin sichtbare Ähnlichkeiten zu dem Beispielmuster aufweist ohne identisch zu ihm zu sein. Die Textursynthese kümmert sich dabei automatisiert und im Idealfall ohne technische Benutzereingaben um die Schwierigkeiten bei der Erstellung von Texturen. Sie berücksichtigt die visuellen Charakteristiken des Beispielmusters und vermeidet dabei zeitgleich auffällige Wiederholungen oder Unnatürlichkeiten in der synthetisierten Textur.

2.1 „Markov Random Fields“-Eigenschaft

Die „*Markov Random Field*“-Eigenschaft ist eine der populärsten Eigenschaften, die vielen Verfahren der Textursynthese zu Grunde liegt [WLKT09]. Sie motiviert eine Metrik, die Verwendung findet, um die Ähnlichkeit zwischen dem Beispielmuster und der zu synthetisierenden Textur zu beschreiben [KEBK05]. Die „*Markov Random Field*“-Eigenschaft beschreibt dabei die Synthese als eine Aneinanderreichung von *lokalen* und *stationären* Prozessen [WLKT09]. Das bedeutet, dass jede Farbe eines Pixels in der Textur ausschließlich über die Pixel in seiner direkten räumlichen Nachbarschaft charakterisiert werden kann (lokal). Diese Charakterisierung ist dabei unabhängig von der Position des betrachteten Pixels (stationär) [KEBK05]. Die Intuition dieser Eigenschaft lässt sich anhand von einem Beispiel verdeutlichen (vgl. Abbildung 1). Einem Betrachter liegt ein Bild vor, welches er aber nur durch ein kleines bewegliches Fenster betrachten kann. Er sieht demnach nie das komplette Bild auf einmal, kann aber durch Bewegungen seines Fensters einzelne Bereiche des Bildes entdecken und erschließen. Das Bild ist dann stationär, falls es unter verschiedenen Ausschnitten immer ähnlich erscheint (eine geeignete Fenstergröße vorausgesetzt). Das Bild ist lokal, falls jeder Pixel in der Mitte eines Fensters über die umliegenden Pixel in seinem Fenster bestimmt werden kann [WLKT09].

Basierend auf dieser Eigenschaft kann der Prozess der Textursynthese spezifiziert werden. Sei ein Beispielmuster gegeben. Dann lässt sich daraus eine Textur synthetisieren, sodass für jeden synthetisierten Pixel dessen räumliche Nachbarschaft zu mindestens einer Nachbarschaft im Beispielmuster ähnlich ist [WLKT09]. Die

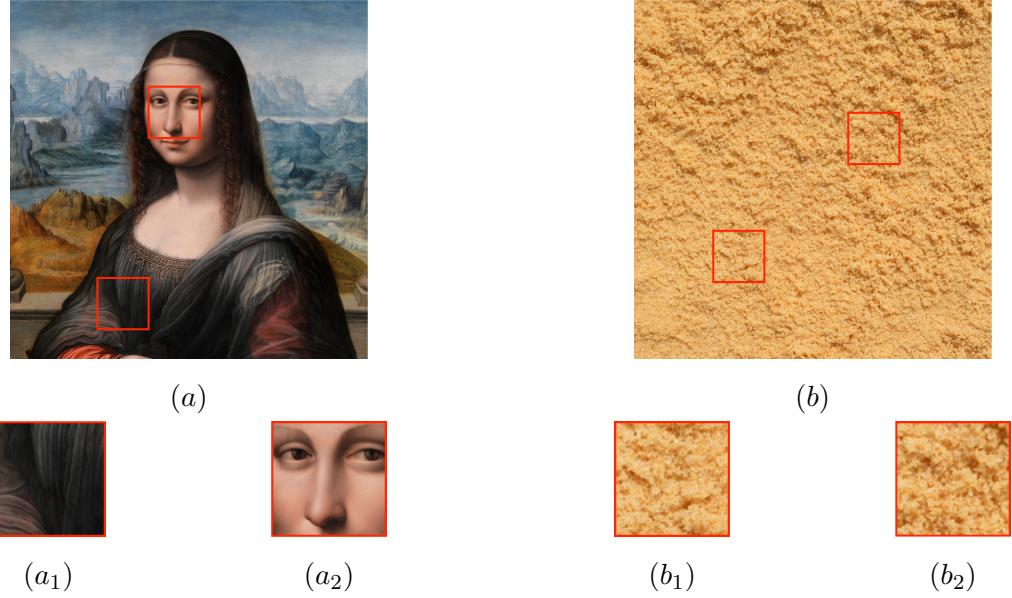


Abbildung 1: (a) ist ein generelles Bild während (b) eine Textur ist. Ein bewegliches Fenster an zwei unterschiedlichen Positionen ist als rotes Quadrat in (a) und (b) gekennzeichnet. Unterschiedliche Ausschnitte einer Textur sind sichtbar ähnlich zueinander sein (b_1, b_2). Dies ist nicht der Fall für ein generelles Bild (a_1, a_2).

Größe der betrachteten Nachbarschaften ist dabei in der Regel ein benutzerdefinierter Parameter. Die Nachbarschaftssuche basiert im Allgemeinen auf der kleinsten quadrierten farblichen Abweichung

$$\min d(\mathbf{t}_i, \mathbf{x}_j) = \|\mathbf{t}_i - \mathbf{x}_j\|^2. \quad (1)$$

\mathbf{t}_i beziehungsweise \mathbf{x}_j beschreiben dabei die Farben einer Nachbarschaft der Größe $N \times N$ als Vektor um den Pixel i in der Textur T respektive um den Pixel j in dem Beispilmuster X . Für jede Nachbarschaft \mathbf{t}_i ist dann ihre ähnlichste Nachbarschaft \mathbf{x}_j im Beispilmuster gefunden, wenn $d(\mathbf{t}_i, \mathbf{x}_j)$ minimal ist [KEBK05].

Aufgrund der Ähnlichkeiten zwischen lokalen Nachbarschaften im Beispilmuster und der Textur wird garantiert, dass die synthetisierte Textur Gemeinsamkeiten mit dem Beispilmuster aufweist [WLKT09].

2.2 Verfahren

Der Großteil an veröffentlichten Algorithmen zur Textursynthese basiert auf der „Markov Random Fields“-Eigenschaft [WLKT09]. Diese Verfahren lassen sich in der Regel einer von zwei Kategorien zuordnen: den *lokal wachsenden Verfahren* und

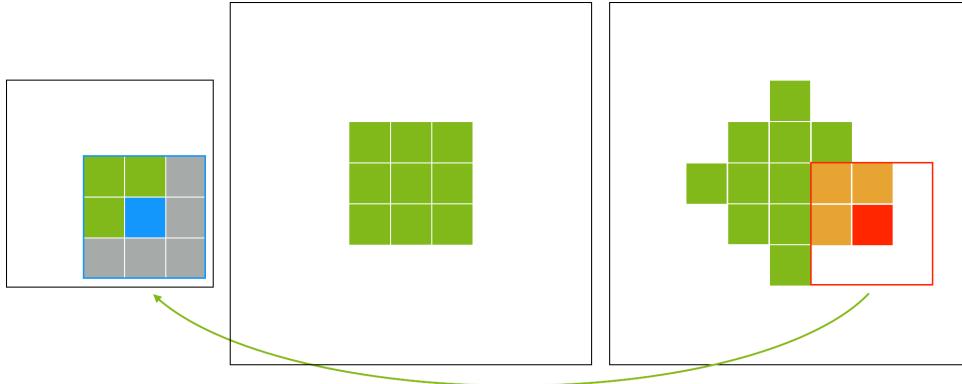


Abbildung 2: Der Algorithmus von [EL99]. Aus einem Beispielmuster (links) wird zuerst eine initiale Textur erstellt, indem eine kleine Region aus dem Beispielmuster in diese kopiert wird (Mitte). Der Algorithmus berechnet die Pixel dann sukzessive um die initiale Region auf Basis der Nachbarschaftssuche (rechts).

den *global optimierenden Verfahren*. Lokal wachsende Verfahren synthetisieren die Textur nach und nach über einzelne Pixel oder Regionen [KEBK05]. Global optimierende Verfahren hingegen synthetisieren und optimieren die Textur iterativ als Ganzes auf Basis einer Zielfunktion [KNL⁺15]. Im Folgenden sollen drei Verfahren der beiden Kategorien näher betrachtet werden.

2.2.1 Pixelbasierte Textursynthese

Einer der ersten Ansätze der Textursynthese ist die *pixelbasierte Textursynthese* (vgl. [EL99]). Er fällt in die Kategorie der lokal wachsenden Verfahren. Sei ein Beispielmuster und eine Nachbarschaftsgröße gegeben, dann funktioniert die grundlegende Idee hinter diesem Algorithmus wie folgt.

Die zu synthetisierende Textur wird zuerst initialisiert, indem eine kleine beliebige Region des Beispielmusters in die Mitte der Textur kopiert wird. Der Algorithmus berechnet dann sukzessive die einzelnen Pixelfarben der Textur kreisförmig um die initiale Region von innen nach außen. Abbildung 2 illustriert dieses Verfahren. Die Farbe eines aktuell betrachteten Pixels p (rot) in der Textur wird dann über eine Nachbarschaftssuche im Beispielmuster ermittelt. Dazu wird zuerst ein Rahmen mit der Größe der Nachbarschaft um den Pixel p gelegt (hier 3×3) und alle bereits gesetzten Pixel in diesem Rahmen gefunden (orange). Auf Basis dieser gesetzten Pixel wird im Beispielmuster die ähnlichste Nachbarschaft auf Grundlage von (1) gefunden. Der Pixel p erhält dann die Farbe des Pixels in der Mitte der ermittelten ähnlichsten Nachbarschaft im Beispielmuster (blau). Dieser Vorgang wird solange

wiederholt, bis alle Pixel der Textur gesetzt sind.

Der Algorithmus [EL99] ist relativ einfach zu verstehen sowie zu implementieren [WLKT09]. Er ist außerdem benutzerfreundlich, da dieser lediglich einen Konfigurationsparameter, die Nachbarschaftsgröße, an den Algorithmus übergeben muss [EL99]. Die Wahl der Nachbarschaftsgröße ist jedoch nicht trivial. Fällt die Wahl der Nachbarschaftsgröße zu klein aus, so kann das Ergebnis zu zufällig wirken. Ist sie auf der anderen Seite zu groß, können sichtbare Wiederholungen entstehen oder es kommt zu Unnatürlichkeiten, da die Nachbarschaftssuche auf Grund ihrer Größe wohlmöglich keine geeigneten Kandidaten finden kann [WLKT09].

2.2.2 Regionsbasierte Textursynthese

Ein weiterer Vertreter der lokal wachsenden Verfahren und eine Erweiterung bzw. ein Nachfolger der pixelbasierten Textursynthese ist die *regionsbasierte Textursynthese* (vgl. [WLKT09]). Die regionsbasierte Synthese ist sehr ähnlich zu der pixelbasierten Synthese, aber anstatt einzelne Pixel zu kopieren, werden stattdessen ganze Regionen in die Textur kopiert. Damit kann die Qualität der Textursynthese verbessert werden, denn es ist sichergestellt, dass Pixel innerhalb einer Region auch zueinander passen. Der wesentliche Unterschied der beiden Verfahren besteht letztendlich im Kopierungsprozess. In pixelbasierten Algorithmen ist die Kopie eines Pixels fest und kann nachträglich nicht mehr verändert werden. Bei den regionsbasierten Verfahren verhält es sich anders, da die Kopie einer Region üblicherweise dazu führt, dass bereits synthetisierte Bereiche der Textur durch die neue Region überdeckt werden. Der Algorithmus muss folglich eine Entscheidung treffen, wie er mit den im Konflikt stehenden Pixeln umgeht. Mehrere mögliche Szenarien sind die Folge. Neue Regionen können alte Regionen einfach überschreiben (vgl. Abbildung 3a). Das führt in der Regel aber zu sichtbaren Kanten der einzelnen Regionen [WLKT09]. Eine weitere Möglichkeit ist, neue Regionen mit den bereits überlappenden Bereichen der Textur zu überblenden (vgl. Abbildung 3b). In manchen Situationen kann dies jedoch zu verwaschenen Artefakten führen [WLKT09]. Die wohl populärste Methode ist das *GraphCut*-Verfahren (vgl. [KSE⁺03, WLKT09]). Der GraphCut-Algorithmus sucht einen optimalen Pfad zwischen zwei Regionen, der sie trennt (vgl. Abbildung 3c).

Regionsbasierte Verfahren sind in der Regel erfolgreicher als pixelbasierte Verfahren, da sie die globale Struktur des Beispilmusters besser einfangen können. Pixelbasierte Verfahren erlauben dagegen mehr Kontrolle über einzelne Pixel [KEBK05]. Ihnen beiden ist aber eine gemeinsame Schwäche zuteil. Auf Grund ihres lokalen Wachstums um eine initiale Region können sich kleine Fehler in den Anfängen der



Abbildung 3: Verschiedene Methoden, wie mit den im Konflikt stehenden Pixeln bei der Kopie einer neuen Region (grün) über bereits synthetisierte Regionen (rot) umgegangen wird.

Synthese schnell anhäufen und zu Inkonsistenzen führen [KEBK05]. Global optimierende Verfahren setzen dort an und versuchen, diesen Fehler so gering wie möglich zu halten.

2.2.3 Texturoptimierung

Die *Texturoptimierung* ist ein global wachsendes Verfahren und vereint dabei Eigenschaften der pixel- sowie regionsbasierten Verfahren. Ähnlich zur pixelbasierten Synthese generiert die Texturoptimierung eine Textur auf Grundlage von Pixeln (anstatt von Regionen). Dabei werden die Pixel jedoch nicht einzeln betrachtet, sondern es werden stets alle Pixel gemeinsam zur Berechnung der Textur mit einbezogen. Ebenso verhält sich das Setzen einer Pixelfarbe nicht *gierig* [WLKT09]. Die Texturoptimierung beschreibt einen Optimierungsprozess, der die gesamte Textur durch sukzessives Ausführen des Algorithmus verbessert. Damit können sich die Pixelfarben der Textur auch im späteren Verlauf der Optimierung ändern, solange jene Änderung zur Verbesserung bzw. Optimierung der Textur beiträgt.

Die globale Optimierung einer Textur basiert auf einer *Initialisierungstextur*. Falls eine Initialisierungstextur nicht explizit übergeben wird, so wird diese üblicherweise gleichmäßig aus zufälligen Regionen des Beispilmusters generiert [KEBK05]. Diese Initialisierung dient dann als initiale Eingabe für die Texturoptimierung. Auf Basis dieser Eingabe wird eine neue, verbesserte Textur in Abhängigkeit zur Ähnlichkeit zum Beispilmuster synthetisiert, die wiederum als Eingabe für einen weiteren Durchlauf dient. Die Texturoptimierung bricht letztendlich ab, sobald sich keine weitere Verbesserung der Eingabetextur mehr erzielen lässt.

Die Texturoptimierung basiert auf der Optimierung einer Zielfunktion E . Diese Zielfunktion beschreibt das Ungleichgewicht zwischen den Regionen der Textur und den dazu ähnlichsten Regionen des Beispilmusters. Je minimaler das Ungleichgewicht zwischen Textur und Beispilmuster, umso qualitativ hochwertiger ist folglich

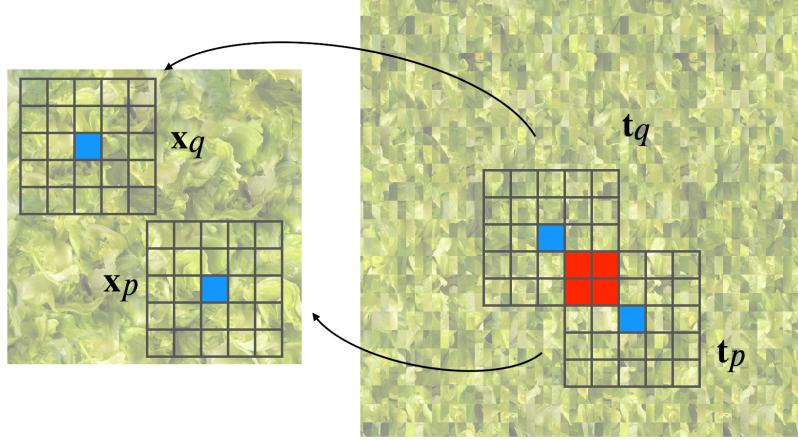


Abbildung 4: Zwei überlappende Nachbarschaften \mathbf{t}_p , \mathbf{t}_q in der Textur und deren ähnlichste Nachbarschaften \mathbf{x}_p , \mathbf{x}_q im Beispielmuster. Wenn sich zwei Nachbarschaften überlappen (roter Bereich), dann führt jede Diskrepanz zwischen \mathbf{x}_p und \mathbf{x}_q in diesem Bereich zu einem Fehler in der synthetisierten Textur.

die synthetisierte Textur. Die Ähnlichkeit zwischen zwei Regionen aus dem Beispielmuster und der Textur berechnet sich dabei analog zu den pixel- und regionsbasierten Verfahren über den Distanzterm $d(\mathbf{t}_p, \mathbf{x}_p)$ aus (1). Dann beschreibt

$$E(T, \{\mathbf{x}_p : p \in T^\dagger\}) = \sum_{p \in T^\dagger} d(\mathbf{t}_p, \mathbf{x}_p) \quad (2)$$

die zu minimierende Zielfunktion [KEBK05]. E beschreibt das Ungleichgewicht der Textur T zu einem Beispielmuster X in Abhängigkeit einer Eingabemenge $\{\mathbf{x}_p\}$. Diese Menge beschreibt die ähnlichsten Nachbarschaften im Beispielmuster für jeden betrachteten Pixel p in der Textur (vgl. Abbildung 4).

Es ist wichtig anzumerken, dass dabei nur eine Teilmenge an Pixeln $p \in T^\dagger \subseteq T$ aus der Textur berücksichtigt wird. Es zeigt sich, dass die Berachtung aller Pixel in T redundant ist und sich daher berechnungstechnisch nicht lohnt [KEBK05]. T^\dagger wird dabei so gewählt, dass sich die Nachbarschaften benachbarter Pixel überlappen. Damit beeinflussen nur eine Handvoll an Nachbarschaften die Farbe eines gegebenen Pixels. Das verhindert, dass Pixel in Regionen, die sich in vielen Nachbarschaften überlappen, im Endresultat zu verwaschen wirken [KEBK05].

Die Textursynthese nutzt die Zielfunktion (2) um T sukzessive zu optimieren, in dem E über mehrere Iterationen minimiert wird. Dafür wird die Zielfunktion alternierend zwischen ihren beiden Eingabeparametern T und $\{\mathbf{x}_p\}$ minimiert. Im ersten Schritt wird die Menge $\{\mathbf{x}_p\}$ bestimmt, sodass für alle \mathbf{t}_p mit $p \in T^\dagger$ gilt, dass $d(\mathbf{t}_p, \mathbf{x}_p)$ minimal ist. So werden die ähnlichsten Nachbarschaften im Beispi-

muster zu allen betrachteten Nachbarschaften in T gefunden. Damit kann folglich T aktualisiert werden, sodass E mit den gefundenen $\{\mathbf{x}_p\}$ minimal ist. Durch die Aktualisierung von T können sich wiederum die ähnlichsten Nachbarschaften $\{\mathbf{x}_p\}$ zu dem aktualisierten T verändert haben und die Minimierung beginnt mit den neuen Eingaben von vorne. Sie wird solange wiederholt, bis sich die Menge $\{\mathbf{x}_p\}$ nicht mehr verändert (vgl. [KEBK05]).

Die Lösung dieses Minimierungsproblems basiert auf dem „*Expectation-Maximization (EM)*“-Algorithmus (vgl. [MK97]). Der EM-Algorithmus findet Anwendung, wenn zusätzlich zu dem Problem der Optimierung der Zielfunktion dessen Parameter an die neue Zuordnung angepasst werden müssen. Das unterteilt die Lösung des Minimierungsproblems in zwei Schritte.

Der *E-Schritt* minimiert die Zielfunktion, in dem ein neues T bestimmt wird, sodass $d(\mathbf{t}_p, \mathbf{x}_p)$ minimal ist für alle $p \in T^\dagger$. Für Pixel in der Textur, die lediglich einer Nachbarschaft angehören, bedeutet das insbesondere, dass diese die Farbe des Pixels an der Position im entsprechenden Beispilmuster annehmen. Für Pixel, die sich in mehreren Nachbarschaften wiederfinden (vgl. roter Bereich in Abbildung 4), wird dessen neue Farbe über die Mittelung der verschiedenen Pixelfarben aus den zugehörigen Nachbarschaften des Beispilmusters bestimmt [KEBK05]. Diese Mittelung führt zu einer gewissen Unschärfe und erlaubt dem Optimierungsprozess, neue ähnliche Nachbarschaften im Beispilmuster zu finden.

Der *M-Schritt* ändert schließlich die Parameter für das aktualisierte T , in dem für das festgehaltene T dessen neue ähnlichste Nachbarschaften im Beispilmuster gesucht werden. Dies beinhaltet die Lösung eines klassischen „*Nearest Neighbor Search*“-Problems (vgl. [KEBK05]).

Das beschriebene Verfahren der Texturoptimierung aus [KEBK05] wiederholt die Synthese der Textur mit unterschiedlichen Nachbarschaftsgrößen. Die Synthese beginnt großflächig mit einer Nachbarschaftsgröße von 32×32 Pixeln. Das gibt der Textursynthese ein anfängliches Gefühl eines regionsbasierten Verfahrens und erlaubt damit, dass auch größere Strukturen vom Beispilmuster an die zu synthetisierende Textur übergeben werden können [KEBK05]. Die Textursynthese wird anschließend durch kleinere Nachbarschaftsgrößen verfeinert ($16 \times 16, 8 \times 8, \dots$). Damit gleicht der Optimierungsprozess einer *Scale-Pyramide* [KNL⁺15], der nach und nach über verschiedene Skalierungen der Nachbarschaftsgrößen verfeinert wird. Mittels dieses Verfahrens entfällt ebenso die benutzerdefinierte Eingabe einer passenden Nachbarschaftsgröße.

3 Schwächen der Textursynthese und deren Verbesserungsmöglichkeiten

Bisherige Ansätze demonstrieren ihre Algorithmen nur für Beispilmustern mit geringen Auflösungen bis zu 128^2 Pixeln [KNL⁺15]. Für diese Auflösungen funktionieren die vorgestellten Ansätze gut, denn die entsprechenden Beispilmuster enthalten auf Grund ihrer geringen Größe oftmals nur kleine Details, die sich leicht vervielfachen lassen, ohne dabei unnatürlich auszusehen. Höher aufgelöste Beispilmuster leiden dagegen an der „Markov-Random-Field“-Eigenschaft. Es werden stets nur kleine lokale Ausschnitte eines Beispilmusters betrachtet, wodurch die globale Ähnlichkeit zu dem benutzten Muster völlig außer Acht gelassen wird. Folglich werden Texturen aus hochauflösenden Beispilmustern synthetisiert, die den Erwartungen qualitativ hochwertigen Texturen nur selten gerecht werden.

In diesem Kapitel wird eine Textursynthese beschrieben, die sich der Texturoptimierung annimmt und sie mit mehreren Verbesserungen ausstattet, um die bisherigen Probleme der Textursynthese zu überwinden. Dazu zählen insbesondere das Erhalten von großflächigen Strukturen aus dem Beispilmuster, das Vermeiden von auffälligen Wiederholungen in der synthetisierten Textur und das Erkennen und Bewahren von Regelmäßigkeiten. Das Verfahren greift dabei Schlüsselideen vorangegangener Verbesserungsmöglichkeiten auf und modifiziert sie in so fern, so dass die Texturoptimierung weiterhin völlig autonom arbeiten kann und keine individuellen, benutzerdefinierten Anpassungen für unterschiedliche Beispilmuster von Nöten sind.

3.1 Großflächige Strukturen

Texturen bzw. Beispilmuster enthalten oft Details verschiedener Größen. So enthält eine Steintextur winzige Details in ihrer Oberfläche, aber wohlmöglich auch größere Strukturen wie etwa Risse oder Kanten. Diese Strukturen mit großer Ausdehnung werden in der bisherigen Synthese garnicht erst berücksichtigt.

Ein möglicher Ansatz dieses Problem zu lösen ist der Einsatz eines „*Guidance-Channels*“ [KNL⁺15]. Dieser „Guidance-Channel“ erweitert das Beispilmuster um einen zusätzlichen Kanal, in welchem nicht-lokale Informationen über großflächige Strukturen des Beispilmusters gespeichert werden. [LH06] zeigt, dass diese Methode besonders gut funktioniert, wenn in ihm die Distanz jedes Pixels zu dem an ihm nächstgelegenen Merkmal (z.B. eine Kontur) gespeichert wird. Dieser „Guidance-Channel“ soll im Gegensatz zu bisherigen Verfahren völlig automatisiert berechnet

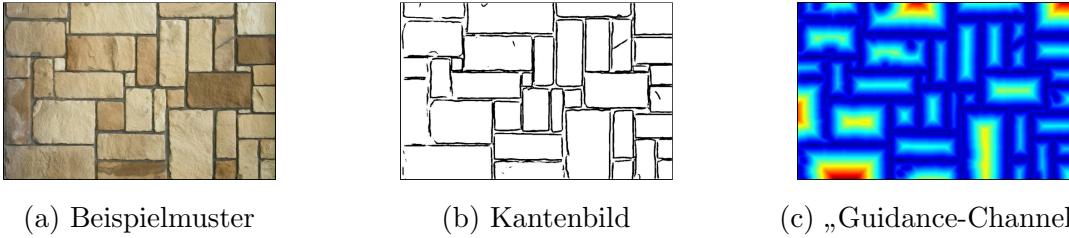


Abbildung 5: Erstellung eines „Guidance-Channels“ [KNL⁺15].

werden und nicht explizit durch den Benutzer übergeben werden müssen [KNL⁺15].

Die Erstellung eines „Guidance-Channels“ umfasst mehrere Schritte, die in Abbildung 5 illustriert sind.

Zunächst werden aus dem Beispilmuster Merkmale mit Hilfe eines Kantendetektors extrahiert. Dies generiert ein graustufiges Kantenbild des Beispilmusters mit kontinuierlichen Werten. Für die weitere Verarbeitung des Kantenbildes ist dessen Binarisierung nötig. Mittels Otsus Schwellenwertverfahren (vgl. [?]) wird dazu automatisiert ein optimaler globaler Schwellenwert ermittelt, mit dessen Hilfe das Kantenbild binarisiert werden kann. Anschließend kann aus dem binarisierten Kantenbild ein Distanzbild bestimmt werden. Dazu wird mittels MATLABs `bwdist`-Methode die euklidische Distanz jedes Pixels zu seiner nächstgelegenen Kante berechnet. Das bedeutet, dass in der Kantenbildmatrix die Distanz von jedem Eintrag mit einer 0 (keine Kante) zu dem räumlich nächstgelegenen Eintrag mit einer 1 (Kante) bestimmt wird und in einer neuen Matrix bzw. einem neuen Bild gespeichert wird (dem Distanzbild). Anschließend wird das Distanzbild entsprechend zum benutzten Farbraum normiert, so dass dessen Werte zu den restlichen Werten des Beispilmusters passen. Das normierte Distanzbild wird schließlich als „Guidance-Channel“ bzw. als vierter Kanal im Beispilmuster gespeichert (vgl. [KNL⁺15]).

Der „Guidance-Channel“ kann schließlich ohne spezielle Anpassung der Texturoptimierung für die Synthese berücksichtigt werden. Dazu müssen lediglich die Vektoren \mathbf{t}_p und \mathbf{x}_p den vierten Kanal für ihre Nachbarschaften mit aufnehmen. Dann liefert $d(\mathbf{t}_p, \mathbf{x}_p)$ (vgl. (1)) ein Ähnlichkeitsmaß zwischen Beispilmuster und Textur unter zusätzlicher Berücksichtigung des „Guidance-Channels“ [KNL⁺15].

Es zeigt sich jedoch in der Praxis, dass die feste Gewichtung des „Guidance-Channels“ zu Problemen führen kann. Durch das Hinzufügen eines vierten Kanals verringert sich die Fähigkeit der Synthese, unterschiedliche Materialien bzw. unterschiedliche Farben zwischen einer Kante zu unterscheiden. Abbildung 6 erklärt dieses Problem anhand eines Beispiels.

Um diesem Problem entgegenzuwirken kann eine *adaptive Gewichtung* des „Gu-



Abbildung 6: Synthese einer Textur bei fester Gewichtung des „Guidance-Channels“ führt zu schwachen Resultaten bei unterschiedlichen Farbverteilungen an den Kanten. Einzelne Kacheln vermischen sich in ihren Farben, da ihr Beitrag zum Gesamtresultat im Vergleich zum „Guidance-Channel“ zu gering ist [KNL⁺15].

dance-Channels“ genutzt werden. Die adaptive Gewichtung ordnet dem „Guidance-Channel“ ein geringeres Gewicht zu, wenn sich die Farben in der räumlichen Nachbarschaft des betrachteten Pixels im Beispilmuster stark unterscheiden. Damit kann der Vorteil der Berücksichtigung von nicht-lokalen Informationen weiterhin genutzt werden. Die Gewichtung wird jedoch verringert, falls sie für die Aufrechterhaltung der Qualität der synthetisierten Textur von Nöten ist.

Dazu werden zuerst die Pixelfarben in der direkten Nachbarschaft eines Pixels im Beispilmuster in einer Dichte-Karte gesammelt [KNL⁺15]. Mittels dieser Karte kann schließlich entschieden werden, ob sich die Farben in der betrachteten Nachbarschaft stark unterscheiden. Dies ist zum Beispiel der Fall, wenn sich mehrere Hügel in der Dichte-Karte wiederfinden. Unterscheiden sich die Farben nicht, so kann ein hohes Gewicht für den „Guidance-Channel“ gewählt werden. Andernfalls wird ein etwas zurückhaltenderes Gewicht gewählt [KNL⁺15].

3.2 Auffällige Wiederholungen

Bisherige Verfahren garantieren auf Grund der „Markov-Random-Field“-Eigenschaft nur eine lokale Ähnlichkeit zum Beispilmuster. Ein kleiner Ausschnitt aus der Textur ist ähnlich zu einem kleinen Ausschnitt des Beispilmusters. Das führt unmittelbar zu dem Problem, dass sich bestimmte Ausschnitte in der synthetisierten Textur häufig wiederholen, wogegen sich andere Ausschnitte des Beispilmusters nur sehr selten in der Textur wiederfinden. Zum einen wird dadurch nicht die komplette Vielfalt des Beispilmusters ausgeschöpft und es wird demnach keine globale Ähnlichkeit zum Beispilmuster erreicht werden. Zum anderen sind für den Betrachter auffällige Wiederholungen in der synthetisierten Textur zu sehen. Das führt unmittelbar da-

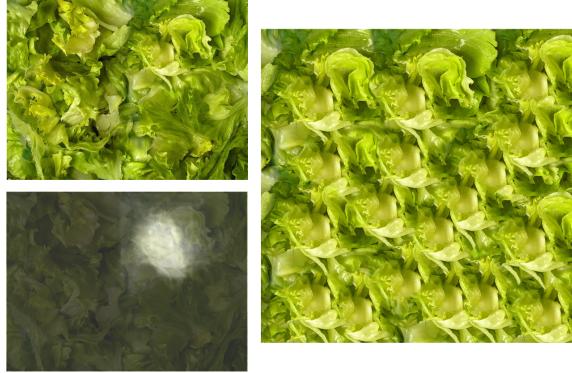


Abbildung 7: Synthese einer Textur ohne Constraints [KNL⁺15]. Es finden sich auffällige Wiederholungen in der synthetisierten Textur (rechts) wieder. Die Vorkommen-Karte Ω (unten links) des Beispielmusters (oben links) unterstreicht dies, denn es gibt einen deutlich erkennbaren (weißen) Bereich an Pixeln des Beispielmusters, der besonders oft für die Synthese der Textur verwendet wurde.

zu, dass die Ergebnisse der Textursynthese den Anforderungen an eine qualitativ hochwertige Textur nicht gerecht werden.

Zur Lösung dieses Problems können dem Auswahlprozess $\min d(\mathbf{t}_p, \mathbf{x}_p)$ (vgl. (1), (2)) der ähnlichsten Nachbarschaften im Beispielmuster zu den Nachbarschaften in der zu optimierenden Textur bestimmten Einschränkungen (*Constraints*) unterliegen. [KNL⁺15] stellt in diesem Zuge das „*Spatial-Uniformity*“ -Constraint vor, mit dem Ziel, eine globale Ähnlichkeit zum Beispielmuster zu garantieren.

Das „*Spatial Uniformity*“-Constraint soll den Auswahlprozess dazu ermutigen, alle Regionen des Beispielmuster in gleichmäßigem Verhältnis zueinander in der synthetisierten Textur zu berücksichtigen, in dem Regionen bei der Auswahl bestraft werden, die bereits zu oft verwendet wurden. Damit unterscheidet es sich insbesondere von dem „*Bidirectional Similarity*“ -Constraint, welches lediglich garantiert, dass jede Region des Beispielmusters mindestens einmal für die Synthese der Textur benutzt wird (vgl. [SCSI08]) [KNL⁺15]. Für diese Aufgabe wird für jede Iteration der Texturoptimierung eine Vorkommen-Karte Ω mit

$$\Omega(x, y) = |\{\mathbf{x}_p \mid (x, y) \in \mathcal{N}(\mathbf{x}_p)\}|$$

für die Pixel des Beispielmusters aufgebaut, die speichert, wie oft ein Pixel in den Regionen vorkommt, die die Textur bilden. $\mathcal{N}(\mathbf{x}_p)$ kennzeichnet dabei die Menge an Pixeln in der Nachbarschaft um \mathbf{x}_p . Abbildung 7 zeigt die Ausgabe der Vorkommen-Karte Ω für eine gewöhnliche Textursynthese ohne Constraints.

In einer perfekt balancierten Textur wird jeder Pixel des Beispielmusters gleich

oft benutzt. Die optimale Anzahl dafür wird über

$$\omega_{best} = \frac{|T|}{|X|} N^2$$

definiert, wobei $|T|$ bzw. $|X|$ die Anzahl der Regionen in T respektive X und N^2 die Anzahl der Pixel in jeder Region angeben.

Um den Auswahlprozess zu manipulieren wird der Distanzterm zur Nachbarschaftssuche aus (1) modifiziert zu

$$d(\mathbf{t}_p, \mathbf{x}_p) = \|\mathbf{t}_p - \mathbf{x}_p\|^2 + \lambda \frac{\Omega(\mathbf{x}_p)}{\omega_{best}}, \quad (3)$$

wobei $\Omega(\mathbf{x}_p)$ den Durchschnitt der globalen Häufigkeit aller Pixel in \mathbf{x}_p über

$$\Omega(\mathbf{x}_p) = \frac{\sum_{(x,y) \in \mathbf{x}_p} \Omega(x, y)}{N^2}$$

beschreibt. Formel 3 bestraft damit Regionen \mathbf{x}_p umso mehr, je häufiger die Pixel in ihr im Verhältnis zur optimalen Häufigkeit eines Pixels bereits in der synthetisierten Textur vorkommen. $\lambda = 10$ kontrolliert dabei die Gewichtung des Bestrafungsterms (vgl. [KNL⁺15]).

3.3 Regelmäßigkeiten

4 Zusammenfassung

Literatur

- [EL99] EFROS, Alexei A. ; LEUNG, Thomas K.: Texture Synthesis by Non-Parametric Sampling. In: *IEEE International Conference on Computer Vision*, 1999, S. 1033–1038
- [KEBK05] KWATRA, Vivek ; ESSA, Irfan ; BOBICK, Aaron ; KWATRA, Nipun: Texture Optimization for Example-based Synthesis. In: *SIGGRAPH '05*, 2005, S. 795–802
- [KNL⁺15] KASPAR, Alexandre ; NEUBERT, Boris ; LISCHINSKI, Dani ; PAULY, Mark ; KOPF, Johannes: Self Tuning Texture Optimization. In: *Computer Graphics Forum* 34 (2015), Nr. 2, S. 349–359
- [KSE⁺03] KWATRA, Vivek ; SCHÖDL, Arno ; ESSA, Irfan ; TURK, Greg ; BOBICK, Aaron: Graphcut Textures: Image and Video Synthesis Using Graph Cuts. In: *ACM SIGGRAPH 2003 Papers*, 2003, S. 277–286
- [LH06] LEFEBVRE, Sylvain ; HOPPE, Hugues: Appearance-space Texture Synthesis. In: *SIGGRAPH '06*, 2006, S. 541–548
- [MK97] McLACHLAN, Geoffrey J. ; KRISHNAN, Thriyambakam: *The EM algorithm and extensions. Wiley series in probability and statistics*. John Wiley & Sons, 1997
- [SCSI08] SIMAKOV, Denis ; CASPI, Yaron ; SHECHTMAN, Eli ; IRANI, Michal: Summarizing Visual Data Using Bidirectional Similarity. In: *IEEE Conference on Computer Vision and Pattern Recognition*, 2008
- [WLKT09] WEI, Li-Yi ; LEFEBVRE, Sylvain ; KWATRA, Vivek ; TURK, Greg: State of the Art in Example-based Texture Synthesis. In: *Eurographics 2009, State of the Art Report, EG-STAR*, 2009