UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Evaluierung verschiedener 3D Painting Verfahren

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
Jan Robert Menzel

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter: Dipl.-Inform. Matthias Raspe
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2008

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Diplomarbeit von Jan Robert Menzel (Matr.-Nr. 203110424)

Thema: **Evaluierung verschiedener 3D Painting Verfahren**

Virtuelle dreidimensionale Objekte werden in der Computergrafik praktisch immer mit einer Textur dargestellt um den Detailgrad zu erhöhen. Die Erstellung dieser Textur ist allerdings nicht trivial. Die Oberfläche des Objektes wird in der Regel auf eine ebene Textur abgebildet, wodurch es bei allen nicht trivialen Modellen zu Verzerrungen sowie Nähten kommt. Diese Textur wird nun in einem Zeichenprogramm für zweidimensionale Bilder bemalt, wobei der Künstler eben auf diese Verzerrungen und Nähte achten muss.

Ein alternatives Vorgehen bieten 3D Painting Programme an. Bei diesen ist es dem Künstler erlaubt, die Textur direkt auf dem Modell zu erstellen und zu bearbeiten. Technische Details der Textur-Map wie Verzerrungen und Nähte können so vor dem Benutzer verborgen werden, wodurch der Einstieg in die Texturierung erleichtert und auch Anfängern zugänglich gemacht wird.

Es sind verschiedene Ansätze bekannt, 3D Painting zu realisieren und mit den entstehenden Problemen und Artefakten umzugehen. Allerdings ist die Praxistauglichkeit mancher Lösungen noch unklar.

Das Ziel dieser Diplomarbeit besteht darin, geeignete Verfahren auszuwählen, mit denen die Texturierung direkt auf dem Modell erfolgen kann. Diese sollen implementiert und miteinander verglichen werden. Insbesondere interessiert die Art des Pinsels, die Behandlung von Artefakten an Nähten und die zugrunde liegende Datenstruktur der Textur.

Schwerpunkte dieser Arbeit sind:

1. Einarbeitung in vorhandener Literatur zu den Themen Texturierung und 3D Painting
2. Auswahl, bzw. Entwicklung der zu untersuchenden Verfahren
3. Konzeption und Entwicklung einer 3D Painting Anwendung
4. Vergleich der Verfahren und Bewertung der Ergebnisse
5. Dokumentation

Koblenz, den 17. März 2008

- Prof. Dr. Stefan Müller -

Inhaltsverzeichnis

1 Einleitung und Motivation **1**
- 1.1 Problemstellung . . . 1
- 1.2 Zielvorgabe . . . 1

2 Grundlagen **3**
- 2.1 Textur-Map . . . 3
- 2.2 Nähte . . . 4
- 2.3 Verzerrungen . . . 5
- 2.4 3D Painting . . . 7

3 Existierende Ansätze **8**
- 3.1 Vorstellung der Verfahren . . . 8
  - 3.1.1 Micro Polygone . . . 8
  - 3.1.2 Projektives Malen . . . 9
  - 3.1.3 Vertexcolor . . . 14
  - 3.1.4 Octree Texturen . . . 14
  - 3.1.5 NURBS . . . 17
  - 3.1.6 Punktwolken . . . 18
  - 3.1.7 Weitere Verfahren . . . 20
  - 3.1.8 Haptisches Feedback . . . 20
- 3.2 Bewertung der Verfahren . . . 20

4 Implementierung **23**
- 4.1 Funktionalität . . . 23
  - 4.1.1 Dateiformate . . . 23
  - 4.1.2 Ebenen . . . 23
  - 4.1.3 Unterstützung von Grafiktabletts . . . 24
  - 4.1.4 Unterstützte Plattformen . . . 24
  - 4.1.5 Performance . . . 24
- 4.2 Wahl der Bibliotheken . . . 25
- 4.3 Beschreibung der Softwarearchitektur . . . 26
  - 4.3.1 Klassendiagramm . . . 26
  - 4.3.2 Texturhandling . . . 31
  - 4.3.3 Shaderanbindung . . . 31
  - 4.3.4 Umsetzung der Mal-Algorithmen . . . 33
  - 4.3.5 Umsetzung der Nahtfilter . . . 50
- 4.4 Umgesetzte Funktionen . . . 54
  - 4.4.1 Die GUI . . . 54
  - 4.4.2 Die Werkzeuge . . . 57
  - 4.4.3 Die Ebenenpalette . . . 58
  - 4.4.4 Die Pinselpalette . . . 59
  - 4.4.5 Die 2D Ansicht . . . 60

4.5 Octree Texturen 62
4.5.1 GPU Umsetzung 63

5 Ergebnisse **65**
5.1 Vergleich der Nahtfilter 65
5.2 Nutzertest 69
5.2.1 Umfrage zur Texturierung 70
5.2.2 Test der Anwendung MeshPaint 74
5.3 Anforderungen an die Textur-Map 81
5.4 Betrachtung der Performance 82
5.4.1 Mal-Algorithmen 82
5.4.2 Ebenen 83
5.4.3 Einfluss der Modell- und Texturgröße 85
5.5 Octree Texturen 85
5.6 Beispiele 86

6 Ausblick **89**

7 Fazit **92**

A Modellverzeichnis **93**

B Benchmark Ergebnisse **96**

Abbildungsverzeichnis

1 Vier mögliche Texture-Maps für das gleiche Modell 3
2 Automatisch erstellte Textur-Map 4
3 Visualisierung des Verlaufes der Nähte der Textur-Maps . . 5
4 Zwei Punkte der Oberfläche können nicht einfach... 6
5 Pyramide mit unterschiedlich verzerrten Textur-Maps 7
6 ID-Puffer 8
7 Projektives Malen aus der Sicht des Nutzers... 10
8 Ein Puffer enthält zunächst die Pinselstriche... 10
9 Um alle Texel der Ausgangstextur auf diesen Puffer... 10
10 Auf diesem Weg erhält man für jeden Vertex eine... 11
11 Da nun bekannt ist, welchen Ausschnitt der... 11
12 Die Polygone werden in diesem Renderpass mit der... 12
13 Artefakte an Nähten 13
14 Bemalen einer Octree Textur [GJDN02]. 15
15 Normalen bei Octree Texturen 16
16 Eine naive Erstellung der Mip-Map... 16
17 Direktes Bemalen einfacher NURBS Modelle 17
18 Surface Graph Sketching 18
19 Painting auf Punktwolken 19
20 Visualisierung der Pinselposition und -ebene... 19
21 Projektion der Pinselstriche bei Igarashi [IC01]. 20
22 SensAble PHANTOM. 21
23 Sechs Textur-Ebenen mit unterschiedlichen Blendmodi... . . 24
24 Das Programm MeshLab 27
25 Automatisches Kantenbild der Pinsel 28
26 Klassendiagramm (1/2) 29
27 Klassendiagramm (2/2) 30
28 Vorschauprobleme bei geringer Texturauflösung 33
29 In Kombination mit Ebenenmodi kann die Vorschau... 34
30 Die Texturen beim Malen... 37
31 Das Werkzeug brush ist ein Volumen... 43
32 Malen ohne (links) und mit Interpolation (rechts). 48
33 Der Linefilter 51
34 Der Pointfilter 51
35 Nachbar Polygone 52
36 Nachbar Polygone. 52
37 Der Nearest Neighbour Filter 54
38 Das Hauptfenster von MeshPaint mit geladenem Modell. . . 55
39 Die verschiedenen Ansichten des Modells... 56
40 Durch die Laseroption des *airbrush* kann symmetrisch... . . . 57
41 Wirkung von color burn und color dodge 58
42 Dialog zur Wahl der aktiven Ebene. 59

43 Dialog zur Auswahl von Pinseln. 60
44 Der 2D Modus 61
45 Details wie dieses Auge lassen sich gut im 2D... 61
46 Octree Textur 62
47 Wie ein Octree in eine Textur gespeichert wird. 63
48 Links sind die vielen Nähte der Textur-Map... 65
49 Bei sehr kleinen Polygonen... 66
50 Bei größeren Polygonen versagt der Punktfilter. 66
51 Der Linienfilter hat unabhängig von der... 67
52 Bei einem Texel Nearest Neighbour... 67
53 An den Kanten der duplizierten Polygone... 68
54 Der Zeitbedarf des Nearest Neigbour Nahtfilters 68
55 Seite 1 der Nutzerumfrage 71
56 Seite 2 der Nutzerumfrage 72
57 Die Hälfte der Nutzer verzichtete ganz auf den 2D Modus . 73
58 Ergebnis vom Nutzer 1 75
59 Ergebnis vom Nutzer 2. 76
60 Das erste Modell vom Nutzer 3 (3a in den Diagrammen). . . 76
61 Das erste Modell vom Nutzer 3 (3b in den Diagrammen). . . 77
62 Ergebnis vom Nutzer 4. 77
63 Ergebnis vom Nutzer 5. 78
64 Ergebnis vom Nutzer 6. 78
65 Ergebnis vom Nutzer 7. 79
66 Ergebnis vom Nutzer 8. 80
67 Ergebnis vom Nutzer 9. 80
68 Nutzung der verschiedenen Werkzeuge. 81
69 Bei zu kleinen Polygonen versagt das projektive Malen. . . . 81
70 Die Länge eines Striches. 83
71 Die Anzahl der gemalten Striche zwischen zwei Rotationen. 84
72 Zeitbedarf zur Erstellung einer Vorschautextur 84
73 Bunny mit einfacher Octree Textur und Phong Beleuchtung 86
74 Unterschiedliche Traversierung des Octree... 86
75 Stanford Bunny mit automatisch erstellter Textur-Map. . . . 87
76 Killeroo 88
77 Das Stanford Bunny im Stil eines Clownfisches. 88
78 Prozeduraler Holzfilter 89
79 Die Hose links wurde mit einem Rauschen... 90
80 Das Wischfinger-Werkzeug 91
81 Stanford Bunny mit Textur-Map. 93
82 Killeroo mit Textur-Map. 94
83 Nasenmann mit Textur-Map. 94
84 Asian Dragon und Horse 95
85 Zeitbedarf zur Erstellung einer Vorschautextur 97
86 Der Zeitbedarf des Nearest Neigbour Nahtfilters 100

Tabellenverzeichnis

1 Antworten des Nutzertests 70
2 Zeiten zum Ebenenmixen 96
3 Vorberechnung ohne Nahtfilter 98
4 Vorberechnung mit Linienfilter 98
5 Vorberechnung mit Punktfilter 98
6 Vorberechnung mit Kombifilter 99
7 Vorberechnung mit Zusatzpolygonen 99
8 Projizieren der Puffer-Textur 99
9 Anwendung des Brush Werkzeuges ohne erneute Berechnung der Vorschau Textur. 100

Listings

1 Eine XML Datei für ein *ShaderSet* 32
2 Defaultwerte für Uniformparameter können im Shader als Kommentar angegeben werden. Beim Laden des Shaders werden diese speziell nach Uniformparametern durchsucht. . . 32
3 Pseudocode des *airbrush*. 35
4 Vertex-Shader zur Vorberechnung der Texelzuordnung. . . . 39
5 Fragment-Shader zur Vorberechnung der Texelzuordnung. . 39
6 Fragment-Shader zur Projektion. 41
7 Funktion zum Mischen von zwei RGBA Werten im Fragment-Shader. 41
8 Die Werkzeuge *color dodge* und *color burn* im Fragment-Shader. 42
9 Pseudocode des *brush*. 44
10 Fragment-Shader vom *brush*. 45
11 Fragment-Shader vom *brush*. 48
12 Pseudocode des Nearest Neighbour Filter. 53

1 Einleitung und Motivation

In der Computergrafik dienen Texturen dazu, die visuelle Komplexität von Modellen und Szenen zu verbessern, ohne die geometrische Komplexität zu erhöhen. Eine Textur ist in der Regel ein zweidimensionales Bild, das auf ein dreidimensionales Objekt abgebildet wird.

Zur Bearbeitung der Textur kann prinzipiell ein Bildbearbeitungsprogramm genutzt werden, dies stellt den Künstler aber vor eine Reihe von Herausforderungen: Texturbereiche, die auf dem Modell aneinander grenzen, können auf der Textur getrennt sein (so genannte Nähte, siehe Abschnitt 2.2) und die zweidimensionale Abwicklung kann in Relation zur 3D Oberfläche verzerrt sein (siehe Abschnitt 2.3). Zudem fehlen Beleuchtungseffekte, die erst beim Rendern erzeugt werden.

Wünschenswert ist es daher, das 3D Modell direkt bemalen zu können, wie man es von 2D Bildern gewohnt ist. Hierdurch erhält man ein direktes optisches Feedback (dies wird oft als *WYSIWYG* bezeichnet: *What You See Is What You Get*). Der Nutzer muss sich so nicht mit den Details der komplexen Textur-Abwicklung beschäftigen.

1.1 Problemstellung

Zunächst erscheint die Texturierung eines Modells eine ähnliche Aufgabenstellung zu sein, wie etwa das Malen in eine zweidimensionale Bilddatei. Anders als bei der Bearbeitung eines flachen Bildes ergibt sich bei der Erstellung einer Textur aber das Problem der Zuordnung zwischen den Nachbartexeln im 3D und den korrespondierenden Texeln im 2D. Dieses Problem wird in Abschnitt 2.2 genauer beschrieben. Zunächst genügt es zu wissen, dass Nachbartexel auf dem Modell nicht zwangsläufig auch in der Textur Nachbarn sein müssen und dass diese Stellen als *Nähte* bezeichnet werden.

Beim Rendern des Modelles sind die Bereiche der Nähte aufgrund des Texturfilterings anfällig für Artefakte. Daher müssen auch Texel an den Nähten, die nicht direkt einem Punkt der Modelloberfläche zugeordnet werden, mit bemalt werden. Aufgrund einer direkten Vorschau muss diese Behandlung im besten Falle in Echtzeit geschehen. Zudem soll die optische Qualität hoch genug sein, so dass dem Nutzer gar nicht erst bewusst wird, wo Nähte vorhanden sind.

1.2 Zielvorgabe

Das Ziel dieser Diplomarbeit ist es, eine Beispielapplikation zu implementieren, mit der es auch ungeübten Nutzern möglich sein soll, leicht 3D Modelle zu texturieren. In der Anwendung sollen die Probleme der Parametri-

sierung und der Nähte vor dem Nutzer verborgen bleiben. Hierfür müssen zunächst die möglichen Ansätze recherchiert und bewertet werden.

Die Nähte müssen geeignet behandelt werden, damit keine Artefakte auftreten. Es sollen daher verschiedene Möglichkeiten in Hinblick auf ihre Performance und optische Qualität hin untersucht werden.

Wie im Abschnitt 3 noch erklärt werden wird, gibt es Verfahren, die nicht auf einer 2D Textur arbeiten, sondern in einem Zwischenschritt etwa Vertexfarben oder Volumentexturen bearbeiten. Eine direkte Bearbeitung der Zieltextur kann aber wünschenswert sein, da der Nutzer so die realistischste Vorschau erhält. Hierbei wird davon ausgegangen, dass am Ende der Bearbeitung eine zweidimensionale Textur gewünscht wird. Diese hat den Vorteil, dass sie von gängiger 3D Beschleunigerhardware direkt unterstützt wird.

Aus diesem Grund soll untersucht werden, in wie fern sich die direkte Bearbeitung der Textur eignet und wo hierbei die Grenzen liegen.

Als Alternative soll die Nutzung von Octree Texturen betrachtet werden. Diese setzen keinerlei Parametrisierung des Modells voraus und erscheinen als interessante Option zur Texturierung unparametrisierter Modelle.

2 Grundlagen

In diesem Kapitel sollen die grundlegenden Begriffe, die in dieser Arbeit Verwendung finden, definiert werden.

2.1 Textur-Map

Eine Textur-Map ist eine Abbildung, die jedem Punkt der Oberfläche eines Modells einen Wert aus einer Textur zuordnet (vom englischen *map*: Abbildung). Die Textur selbst kann ein-, zwei- oder auch drei-dimensional sein, sowie aus einer mathematischen Beschreibung bestehen[Hec86]. Die Abbildung vom Modell auf die Textur wird in der Regel als Abbildung von jedem Vertex auf die Textur gespeichert. Zwischen diesen Textur-Koordinaten wird dann für die dazwischen liegenden Punkte der Oberfläche interpoliert.

Wenn es nicht anders angegeben wird, sind im Folgenden 2D Texturen gemeint. Für ein Modell kann es diverse, unterschiedliche Textur-Maps geben, Abbildung 1 zeigt mehrere mögliche Textur-Maps für das gleiche Modell.

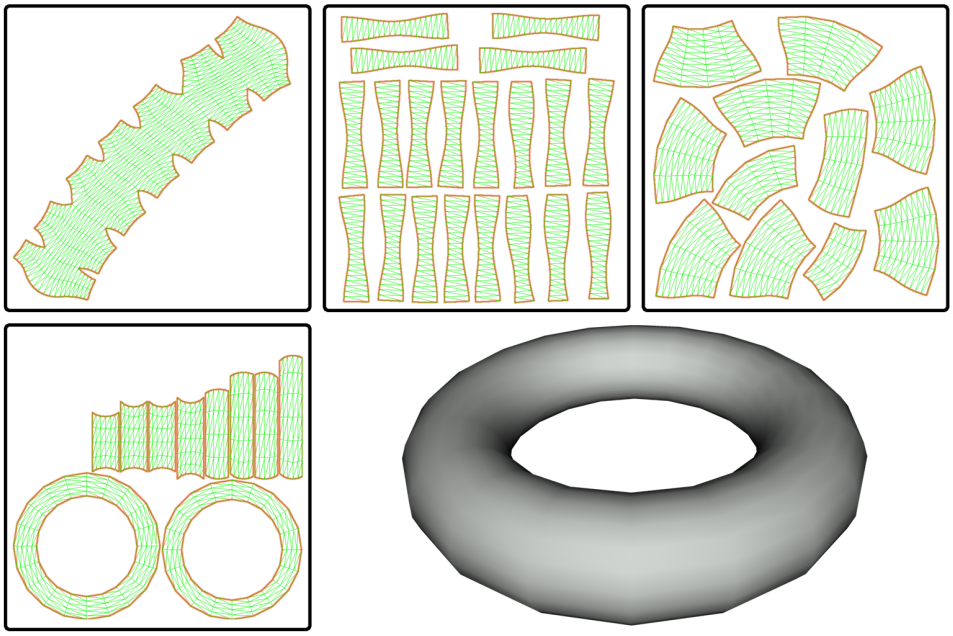


Abbildung 1: Vier mögliche Texture-Maps für das gleiche Modell. Die Nähte sind jeweils rot markiert.

Die Textur-Map wird nur selten automatisch erstellt, dies liegt vor allem daran, dass der Künstler oft die Textur in einem Bildbearbeitungsprogramm erstellen möchte. Hierfür ist es hilfreich, wenn die Textur in der

Textur-Map aus möglichst wenigen Einzelteilen besteht und zudem wenige Verzerrungen enthält. Der Nachteil dieser Textur-Maps liegt in der ungenutzten Fläche der Textur.

Alternativ lassen sich Textur-Maps auch automatisch erstellen um den Anteil ungenutzten Texturspeichers zu reduzieren. So kommen Carr et al. auf eine fast vollständige Abdeckung der gegebenen Fläche der Textur-Map (vergleiche Abbildung 2) [CH02] [CH04b] [CH04a].

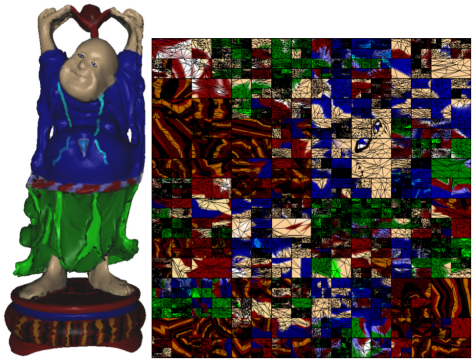


Abbildung 2: Automatisch erstellte Textur-Map, optimiert um möglichst wenig ungenutzte Fläche zu erzeugen [CH04a].

Solch optimierte Textur-Maps haben bei gleicher Anzahl sichtbarer Texel einen kleineren Speicherbedarf, können aber nicht mehr im 2D bemalt werden, da die Zuordnung der Texturbereiche zum Modell für den Künstler zu schwierig wird und es zu viele Nähte gibt. Wenn aber das 3D Painting die "manuelle" Bearbeitung der Textur im 2D überflüssig macht, können solche Textur-Maps eingesetzt bzw. Alternativen wie Octree Texturen genutzt werden.

Teilweise wird dies auch *Textur-Atlas* genannt [CH04a] [MYV93]. Im Gegensatz hierzu wird oft auch die Sammlung mehrerer unabhängiger Texturen in einer *Textur-Map* als Textur-Atlas bezeichnet [nVi04].

Die einzelnen Elemente einer Textur werden analog zum Pixel bei Bildern (vom englischen *Pic*ture *El*ement, wobei das *c* zu *x* wird) als *Texel* (*Tex*ture *El*ement) bezeichnet [JIK+99].

2.2 Nähte

In der Regel soll einem Punkt der Textur-Map genau ein Punkt auf der Objektoberfläche zugeordnet werden. Will man nun das Objekt auf die 2D Textur-Map abbilden, so muss das Objekt unweigerlich "zerschnitten" werden. Das Modell kann, muss aber nicht immer, in mehrere unzusammenhängende Teile zerlegt werden. Diese Teile werden dann auf der Textur-Map positioniert. Die Ränder der Einzelteile werden *Nähte* genannt.

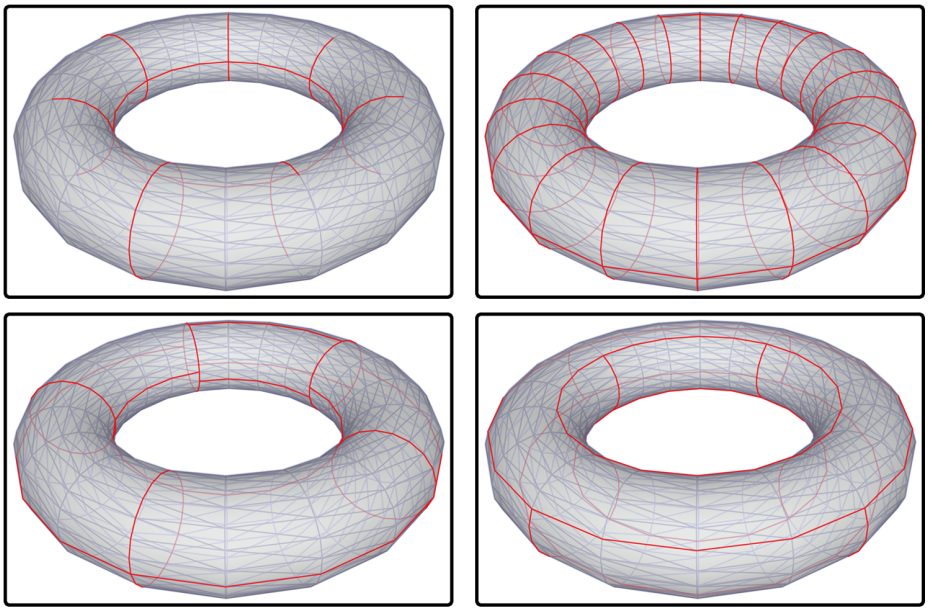


Abbildung 3: Visualisierung des Verlaufes der Nähte der Textur-Maps aus Abb. 1.

Aufgrund dieser Nähte kann zunächst keine Aussage dazu getroffen werden, ob ein Nachbartexel in der Textur auch auf dem Modell ein Nachbar ist. Umgekehrt können Nachbarn auf der Modelloberfläche in der Textur weit auseinander liegen.

Wenn auf die Oberfläche eine Linie gemalt werden soll, so genügt es daher nicht, Anfangs- und Endpunkt in die Textur abzubilden und diese Punkte in der Textur zu verbinden. Abbildung 4 zeigt, dass dies funktionieren kann, wenn zwischen den Punkten keine Nähte verlaufen und die Textur nicht nennenswert verzerrt ist (links). Im allgemeinen kann hiervon aber nicht ausgegangen werden und man erhält ein unerwartetes Ergebnis (rechts).

2.3 Verzerrungen

Bei der Abwicklung eines Objektes auf die Textur-Map können Teile vom Modell verzerrt werden. Dies ist der Fall, wenn die Innenwinkel des Polygons nicht mit denen in der Textur-Map übereinstimmen. Auch wenn gleich große Dreiecke im Modell auf der Textur auf unterschiedlich große Flächen abgebildet werden spricht man von einer Verzerrung.

Abbildung 5 zeigt eine Pyramide mit drei denkbaren Textur-Maps. In der linken Map entsprechen alle Winkel und Flächen denen des Modells. Bei der mittleren wurde das Pyramidendach quasi “von oben“ betrachtet und die Tiefe ignoriert, hierdurch werden die Polygone verzerrt und ver-

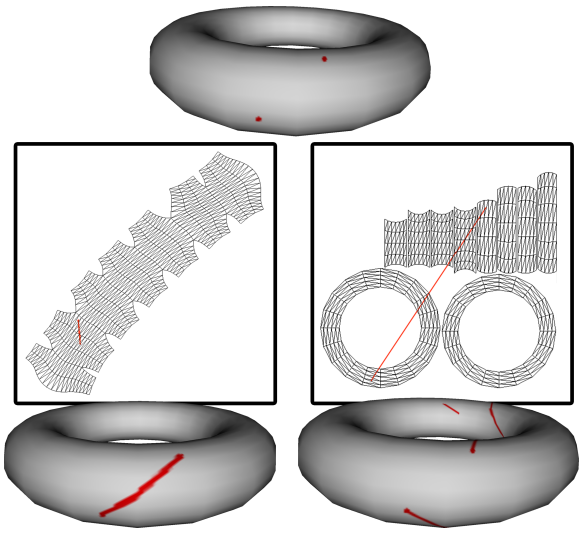


Abbildung 4: Zwei Punkte der Oberfläche können nicht einfach in der Textur miteinander verbunden werden: Links gelingt dies zufällig, da keine Nähte zwischen den Punkten liegen. Rechts hingegen werden (auf dem Objekt) weit entfernte Bereiche bemalt, die gewünschte Linie wird nicht erzeugt.

kleinert dargestellt. In der rechten Map stimmen zwar die Flächenverhältnisse aller Polygone, zwei Dreiecke haben allerdings stark verzerrte Innenwinkel.

Wie an den farbig markierten Nähten zu erkennen ist, "erkauft" sich die erste Textur-Map den Vorteil unverzerrter Oberflächen durch den Nachteil mehr Nähte zu enthalten, die zweite Map den Vorteil gleichmäßiger Verzerrung durch eine kleinere Fläche.

Sowohl Nähte, als auch Verzerrungen stellen eine Herausforderung bei der Texturierung im 2D dar. Dies ist mit ein Grund dafür, dass Künstler die Textur-Maps selbst erstellen, da sie so auf diese Faktoren Einfluss nehmen können. So können etwa Nähte an die Grenzen von Strukturen gelegt werden, an denen im Modell keine kontinuierliche Struktur gewünscht wird. Beispielsweise bietet es sich bei einem Modell eines Menschen an, die Nähte an den Grenzen der Kleidung entlang laufen zu lassen. Analoges gilt für Verzerrungen: Sollen etwa Teile des Modells beschriftet werden, so wird darauf geachtet, diese Bereiche nicht zu verzerren, da sonst auch die Schrift in der gleichen Art verzerrt werden müsste.

Der Künstler muss sich also, wenn er im 2D texturiert, bereits bei der Erstellung der Textur-Map Gedanken über die gewünschte Textur machen.

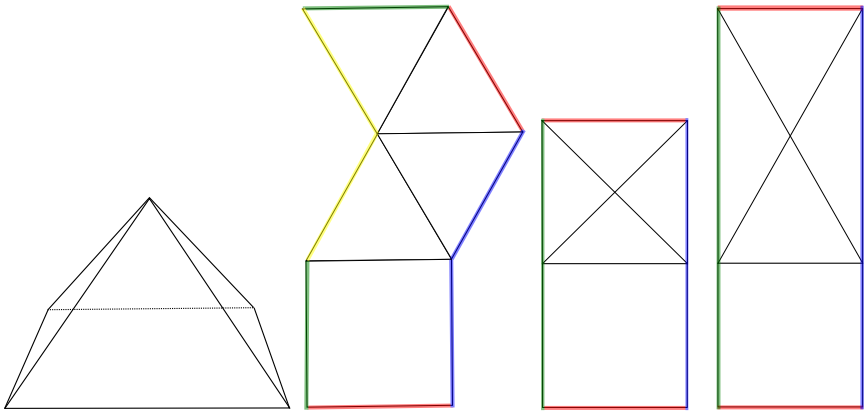


Abbildung 5: Eine Pyramide (links) mit drei möglichen Textur-Maps. Die Nähte sind farbig markiert.

2.4 3D Painting

Die Oberflächenbeschaffenheit eines dreidimensionalen Objektes durch direktes Bemalen im 3D zu editieren soll im Folgenden als 3D Painting bezeichnet werden. Dies umfasst im speziellen die Farbe, muss aber nicht auf diese beschränkt sein. Ebenso ließen sich Reflexionseigenschaften oder Normalen durch das Bemalen ändern. *In 3D* bedeutet in diesem Falle, dass direkt auf die Oberfläche gemalt werden kann und nicht auf eine Textur-Map oder eine andere Parametrisierung der Oberfläche zurück gegriffen werden muss. Auch wenn die Materialeigenschaften letztendlich in einer Textur-Map gespeichert werden, soll dies vor dem Nutzer verborgen bleiben.

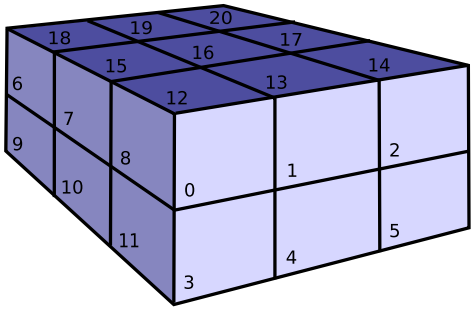
3 Existierende Ansätze

In diesem Kapitel soll eine Übersicht über die Literatur zum Thema 3D Painting gegeben werden. Bekannte Verfahren werden vorgestellt und bewertet. Das Ziel soll sein, einen viel versprechenden Ansatz zu finden, der implementiert und genauer untersucht werden soll.

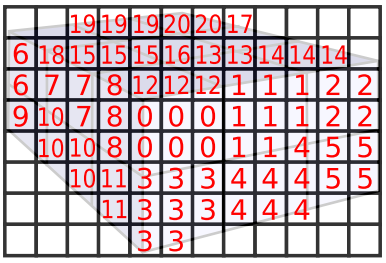
3.1 Vorstellung der Verfahren

3.1.1 Micro Polygone

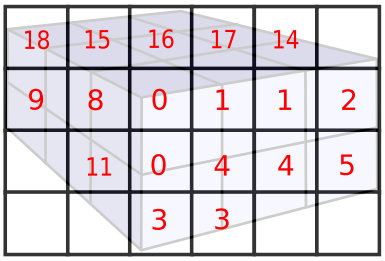
Der erste Ansatz für 3D Painting stammt von Pat Hanrahan und Paul Haeberli aus dem Jahr 1990. In ihrer Veröffentlichung *Direct WYSIWYG Painting and Texturing on 3D Shapes* [HH90] beschreiben sie die Texturierung mit Hilfe von *micropolygons*. Hierbei unterteilten sie das Modell in kleine Patches, die durch vierseitige Polygone mit jeweils einer Farbe und Materialeigenschaft beschrieben wurden. Die Unterteilung geschah allerdings aus technischen Gründen, da zu dem Zeitpunkt noch kaum Hardware-unterstütztes Texture Mapping möglich war.



(a) Alle Texel eines Objektes haben eine eindeutige ID



(b) Ausreichend hoch aufgelöster ID-Puffer



(c) Zu gering aufgelöster ID-Puffer

Abbildung 6: Nicht alle Texel-IDs finden sich im ID-Puffer wieder, wenn dieser zu gering aufgelöst ist. Die Texel-IDs sind in den ID-Puffern jeweils rot eingetragen.

Um zu ermitteln, welche Micro Polygone sich unter dem Pinsel befinden, wurden diese zuvor in einen Offscreenpuffer gerendert. Als Farbe wurde hierfür die Objekt ID genutzt. So kann beim Malen im ID-Puffer an der Bildschirmkoordinate des Pinsels nachgeschlagen werden, welches Micro Polygon bemalt werden muss.

Dieser Ansatz ließe sich leicht auf texturierte Objekte erweitern, indem die vom Micro Polygon gespeicherten Eigenschaften (diffuse Farbe, spekulare Farbe, Normale und Rauheit) auf mehrere Texturen verteilt werden. Er hat allerdings einen entscheidenden Nachteil: Ist die Auflösung der Textur im Verhältnis zur ID-Pufferauflösung sehr hoch, werden einzelne Texel nicht mehr in den ID-Puffer gerendert. Dieses Problem ist zudem von der Kameraposition abhängig: "Zoomt" der Nutzer zu weit hinaus, kommt es zu Artefakten beim Malen aufgrund fehlender Einträge im ID-Puffer. Abbildung 6 verdeutlicht dieses Problem.

Abbildung 6a zeigt die einzelnen Texel, bzw. Micro Polygone mit ihren zugehörigen IDs. Abhängig von der Kameraposition und der Auflösung des ID-Puffers finden sich alle sichtbaren Texel im Puffer wieder (Abb. 6b), oder nur eine Untermenge aller Texel (Abb. 6c). Texel, deren IDs nicht im Puffer auftauchen, werden beim Bemalen nicht berücksichtigt, denn beim Malen wird in diesem Puffer nachgeschlagen, welche Texel geändert werden müssen.

3.1.2 Projektives Malen

Die Probleme einer zu ungenauen Abtastung bei [HH90] lässt sich dadurch umgehen, dass nicht mehr die Texel auf den Bildschirm projiziert und in einen Puffer gespeichert werden. Statt dessen wird zunächst in einen bildschirmfüllenden Puffer gemalt und anschließend werden die Texel auf diesen Puffer projiziert um den neuen Farbwert zu ermitteln.

Der wesentliche Unterschied bei diesem Vorgehen ist der, dass auch mehrere Texel auf die gleiche Stelle des Puffers abgebildet werden können, ohne sich gegenseitig zu verdecken.

Damit nur die Vorderseite des Modelles bemalt wird, muss zudem ein Verdeckungstest der Texel durchgeführt werden. Die Abbildungen 7 bis 12 verdeutlichen den Vorgang.

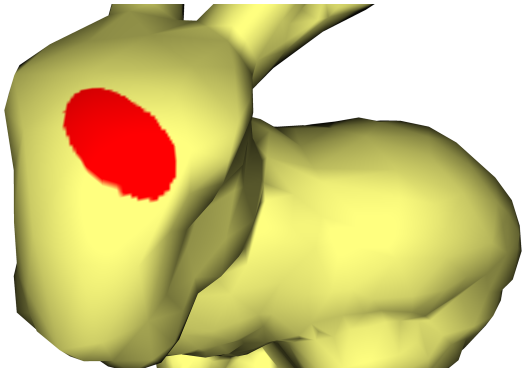


Abbildung 7: Projektives Malen aus der Sicht des Nutzers: Der rote Pinselstrich erscheint direkt auf dem Modell.

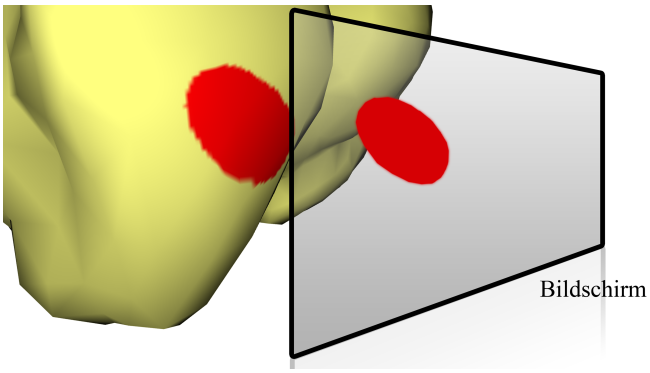


Abbildung 8: Ein Puffer enthält zunächst die Pinselstriche, dieser kann als Textur implementiert werden und liegt gedacht genau vor dem Bildschirm.

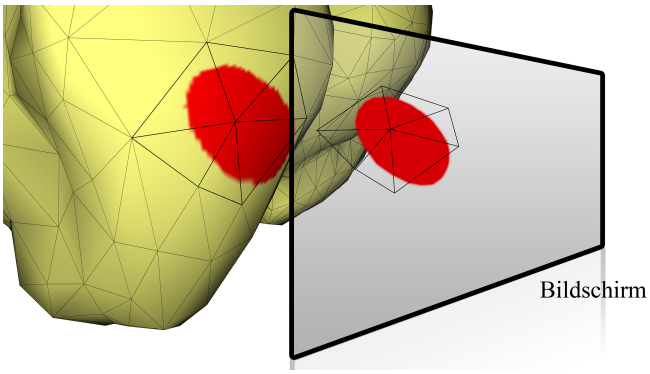


Abbildung 9: Um alle Texel der Ausgangstextur auf diesen Puffer zu projizieren, werden zunächst die Polygone des Modells auf diesen abgebildet.

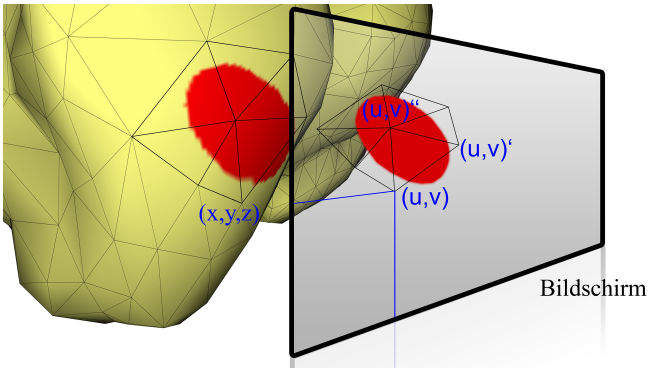


Abbildung 10: Auf diesem Weg erhält man für jeden Vertex eine Bildschirmkoordinate. Diese kann als Textur-Koordinate in der Puffer-Textur interpretiert werden. Hierbei handelt es sich um eine Projektion analog zur aktuellen Kameraeinstellung, kann also sowohl perspektivisch, als auch orthografisch geschehen.

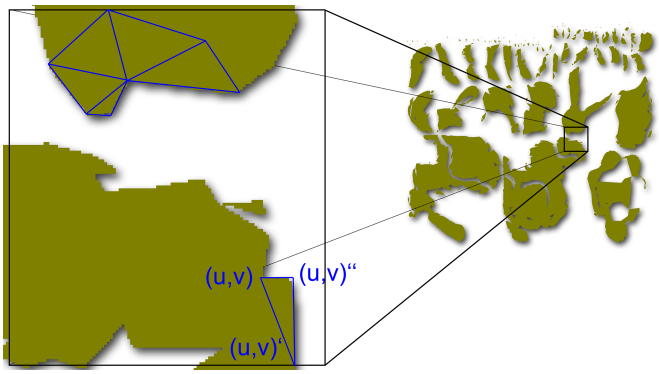


Abbildung 11: Da nun bekannt ist, welchen Ausschnitt der Puffer-Textur jedem Polygon des Modells zugeordnet werden kann, ist es möglich, die Textur zu aktualisieren. Hierfür muss das Modell in Textur-Koordinaten gerendert werden. Die X-,Y-Koordinaten der Vertices werden durch die Textur-Koordinate ersetzt, die Z-Koordinate wird auf eine Konstante gesetzt. Die Darstellung erfolgt orthographisch, die Kamera blickt entlang der Z-Achse, daher ist der konkrete Wert für Z irrelevant. Dieses Rendering muss in der Auflösung der Textur durchgeführt werden und geschieht am besten Offscreen.

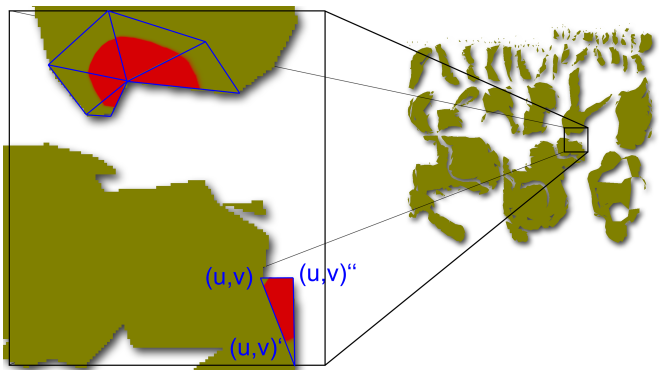


Abbildung 12: Die Polygone werden in diesem Renderpass mit der ursprünglichen Textur und der Puffer-Textur gerendert. Überall dort, wo nichts gemalt wurde, ist die Puffer-Textur transparent, daher bleibt die Textur im Original bestehen. In diesem Schritt muss zudem der Verdeckungstest durchgeführt werden.

In den Abbildungen 7 bis 12 fehlt noch der Verdeckungstest. Hierfür wird jedes Texel in die Kamera projiziert, um den Punkt auf dem Bildschirm zu erhalten, an dem man ihn sehen würde. Punkte die außerhalb des Bildschirmes, und somit außerhalb des Malbereiches liegen, werden nicht modifiziert. Bei allen anderen wird der Tiefenwert des Punktes mit einer Depth-Map der Szene aus dieser Kameraposition verglichen. Nur wenn der Tiefenwert gleich dem Wert in der Depth-Map ist, wird er geändert, denn nur dann handelt es sich um die gesehene Vorderseite des Objektes. Dieses Verfahren ähnelt stark den in der Computergrafik weit verbreiteten Shadow-Maps. Details zur Implementierung des Verdeckungstests folgen in Kapitel 4.3.4.

Dieser Ansatz wird in von Carr et al [CH04a] im Zusammenhang mit automatisch erstellen Textur-Maps angesprochen. Foskey et al. [FOL02] verwenden ein ähnliches Verfahren, allerdings ermitteln sie zunächst alle Polygone, die vom Pinselstrich betroffenen sind und rendern nur diese in Textur-Koordinaten. Die Aktualisierung der Textur geschieht im Wesentlichen in Software. Der Pinsel wird hier als Volumen angenommen.

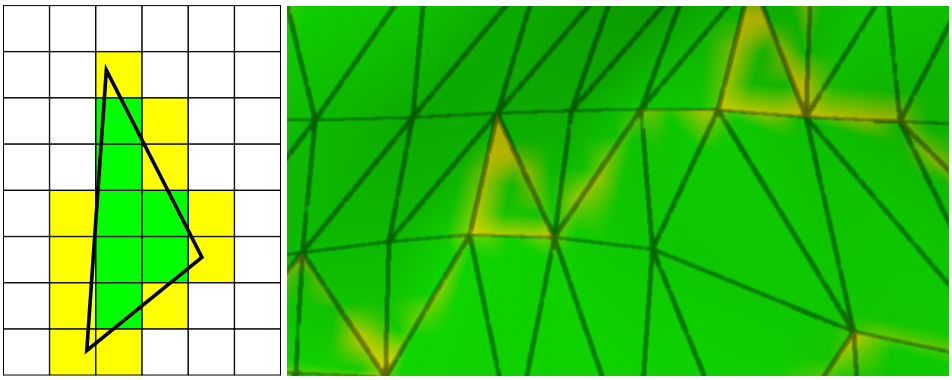
Das gesamte Vorgehen lässt sich auf der GPU durchführen:

- Zunächst wird eine Depth-Map für den Verdeckungstest erstellt.
- Die Textur wird als Ziel eines Renderpasses angegeben.
- Das Modell wird mit drei Texturen gerendert: Der eigenen Textur und der Puffer-Textur in dem sich die Pinselstriche befinden, sowie der Depth-Map.
- Im Vertex-Shader werden die Vertices in die Kamera projiziert. Diese Koordinaten werden dann an den Fragment-Shader übergeben.

Danach wird die Projektion verworfen, stattdessen werden die ausgehenden Vertexpositionen auf die Textur-Koordinaten gesetzt. Der Viewport dieses Renderpasses muss so eingestellt sein, dass auf diesem Weg das ganze Modell (und somit die ganze Textur) Bildschirmfüllend gerendert werden kann.

- Im Fragment-Shader wird der Verdeckungstest durchgeführt: Hierbei wird der Tiefenwert der vom Vertex-Shader übergebenen Projektion mit der Depth-Map verglichen.
- Ist das Fragment sichtbar, so wird die Farbe der Textur mit der Farbe des Pinselstriches gemischt.
- Nun muss das Modell noch einmal für den Nutzer mit der neuen Textur gerendert werden.

Artefakte an Nähten Beim projektiven Malen entstehen an den Nähten Artefakte, die beim Rendern des Modells sichtbar werden. Die Ursache liegt in der Art, wie das Modell in die Textur gerendert wurde. Wird etwa ein kleines Dreieck gerendert, so werden nur die in Abbildung 13 grün eingefärbten Texel erzeugt. Das Polygon liegt aber auch zu einem kleinen Teil über den gelb eingefärbten Texel. Diese behalten allerdings ihre Ursprungsfarbe.



(a) Gerastertes Dreieck. **(b)** Artefakte im gerenderten Modell.

Abbildung 13: Links: Nur die grünen Fragmente werden erzeugt. Beim Rendern werden hingegen auch die gelben Texel gelesen (rechts).

Bei der Darstellung des Modells werden allerdings auch die gelben Texel genutzt. Diese müssen daher auch bei der Texturierung erzeugt werden.

3.1.3 Vertexcolor

Ein weiterer Ansatz verzichtet zunächst völlig auf Texturen und beschränkt sich auf die Änderung von Vertexfarben [ABL95]. Dadurch, dass das Modell nicht mehr auf eine zweidimensionale Ebene parametrisiert werden muss, fallen Nähte und Verzerrungen ganz weg. Zudem existieren allein durch das Modell bereits alle nötigen Nachbarschaftsinformationen. Wenn also der aktuelle Vertex unter dem Pinsel ermittelt wurde, können anhand der Nachbarschaften und der Größe des Pinsels alle weiteren betroffenen Vertices ermittelt werden.

Hierfür kann der Raum um das Modell zum Beispiel in Voxel unterteilt werden, in denen dann die enthaltenen Vertices gespeichert werden. Dadurch können schnell zu einer gegebenen Pinselposition alle Vertices in der Nachbarschaft ermittelt werden [ABL95].

Der wesentliche Nachteil dieses Verfahrens liegt in der Abhängigkeit zur Geometrieauflösung. Um die optische Qualität zu erhöhen, müssen die Polygone weiter unterteilt werden. Hierdurch kann das Modell schnell zu groß werden, um es noch interaktiv zu rendern. Zwar gibt es Möglichkeiten, nur die geänderten Teile des Modelles neu zu rendern und sehr große Geometrien darzustellen, diese *out of core* Verfahren verkomplizieren diesen painting Ansatz allerdings.

3.1.4 Octree Texturen

Die Idee, die hinter Texturen steht, ist prinzipiell nicht auf zweidimensionale Bilder beschränkt. So kann zum Beispiel auch ein 3D Volumen als Textur genutzt werden. Aus den Vertex-Koordinaten kann direkt durch eine geeignete Skalierung eine Textur-Koordinate errechnet werden. So kann man zum Beispiel das Modell auf einen Einheitswürfel skalieren um Textur-Koordinaten im Bereich 0..1 zu erhalten.

Der Vorteil von Volumentexturen liegt darin, dass kein Mapping auf eine zweidimensionale Ebene nötig ist. Der offensichtliche Nachteil liegt in dem enormen Speicherbedarf von Volumentexturen, zumal nur ein kleiner Anteil der Texel für das Rendering genutzt wird.

Dieses Problem lässt sich dadurch umgehen, dass nur die Texel definiert werden, die tatsächlich zum Rendering genutzt werden, also das Modell schneiden. Das Speichern nicht benötigter Voxel lässt sich umgehen, indem die Textur in einem Octree gespeichert wird. Hierdurch verringert sich der Speicherbedarf dramatisch. Mit programmierbaren Grafikkarten lassen sich diese Texturen sogar in Echtzeit darstellen. DeBry et al. [GJDN02] nennen einen um 33% höheren Speicherbedarf im Vergleich zu 2D Texturen und eine um 68% längere Zeit zum Rendern bei Octree Texturen.

Die erhöhte Renderingzeit wurde bei DeBry in einem Software Renderer gemessen, Lefebvre et al. stellen eine Möglichkeit vor, Octree Texturen mit Hilfe von Fragment-Shadern auf modernen Grafikkarte zu nutzen [LHN05].

Painting Bei DeBry [GJDN02] wird in einen adaptiven Octree gemalt, der sich dem Detailgrad des Gemalten anpasst. Abbildung 14 zeigt, wie Farbe dem Octree hinzugefügt wird: Zunächst werden die Knoten gesucht, deren Größe in etwa einem Pixel des Monitors entsprechen. Diese werden ähnlich dem Vorgehen beim projektiven Malen 3.1.2 auf den Bildschirm projiziert um die Position gegen den Z-Puffer des gerenderten Modells zu testen. So sollen nur die sichtbaren Bereiche bemalt werden. Von diesen Knoten geht man nun zu allen Blättern und speichert hier die Farbe des Pinsels.

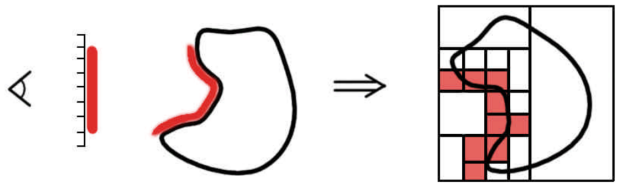


Abbildung 14: Bemalen einer Octree Textur [GJDN02].

Adaptiv wird das Verfahren durch die Erweiterung, dass ein Blatt zur Laufzeit weiter unterteilt werden kann, wenn dieses wesentlich größer als ein Bildschirmpixel ist, wenn dieses Blatt bemalt werden soll. Umgekehrt kann ein Baum wieder schrumpfen, wenn alle Blätter eines Knotens die gleiche Farbe haben.

Einen anderen Ansatz verfolgt Lefebvre [LHN05]: Hier wird die 3D Koordinate des Pinsels, sowie sein Radius genutzt um alle Voxel zu finden, die sich mit dem Pinsel schneiden. Eine Projektion der im 2D gemalten Linie ist hierbei nicht nötig.

Limitierungen Probleme bereiten Octree Texturen wenn das Modell Strukturen aufweist, die kleiner als ein Voxel sind. Abbildung 15 zeigt einen Flugzeugflügel und die umgebenen Voxel. Am rechten Ende wird sowohl die Farbe der Oberseite, als auch die der Unterseite durch das gleiche Voxel bestimmt.

Dieses Problem kann durch die Einführung von Normalen in den Blättern behoben werden. Wird ein Modell aus zwei ausreichend verschiedenen Winkeln bemalt, werden zusätzliche Blätter mit unterschiedlichen Normalen im Octree gespeichert. Bei DeBry müssen die Normalen sich um mindestens 90 Grad unterscheiden, damit ein weiteres Blatt erzeugt wird.

(a) Die gelben Voxel gelten für Ober- und Unterseite.

(b) Je zwei Voxel mit unterschiedlichen Normalen.

Abbildung 15: Um der Ober- und Unterseite des Flügels unterschiedliche Farben zuordnen zu können, müssen die Blätter Normalen erhalten [GJDN02].

Auch die Erstellung von Mip-Maps kann bei Octree Texturen problematisch werden. Dünne Teile des Modells, die in der Originalauflösung an beiden Seiten unterschiedliche Voxel schneiden und somit unterschiedliche Farben haben können, können in einem geringer aufgelösten Octree sich ein Voxel teilen. Bei Benson et al. [BD02] findet sich ein anschauliches Beispiel, bei dem die innere Fläche blau und die äußere rot gefärbt ist (siehe Abbildung 16). Da die größeren Voxel der Mip-Map aber beide Seiten einfärben würden, darf die Farbe nicht gemittelt werden. Auch hier wird sich mit Normalen und mehreren Blättern beholfen. Hierdurch steigt natürlich der Speicherbedarf.

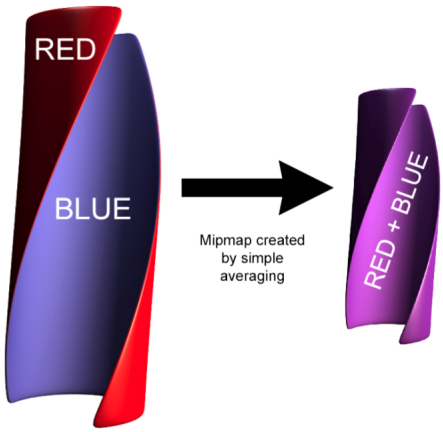


Abbildung 16: Eine naive Erstellung der Mip-Map kann bei dünnen Strukturen scheitern [BD02]. Links das Original, rechts die Mip-Map.

Wie Octree Texturen auf der GPU umgesetzt werden können, wird in Abschnitt 4.5.1 detailliert erläutert.

Umwandlung in eine 2D Textur Eine Octree Textur kann in eine reguläre 2D Textur umgewandelt werden, wenn eine Textur-Map für das Modell existiert. Dies kann nötig werden, falls das Modell in einem System dargestellt werden soll, dass keine Octree Texturen unterstützt. Hierbei

wird das Modell mit den Textur-Koordinaten anstelle der Vertex-Koordinaten gerendert. Die Texturierung erfolgt allerdings weiterhin über die Octree Textur. Das so gerenderte Bild kann nun als 2D Textur genutzt werden. Dieser Vorgang funktioniert nach dem Prinzip des projektiven Malens aus Abschnitt 3.1.2 mit dem einzigen Unterschied, dass keine neue Farbe aufgetragen wird.

Wie schon beim projektiven Malen entstehen auch hier Artefakte an den Nähten.

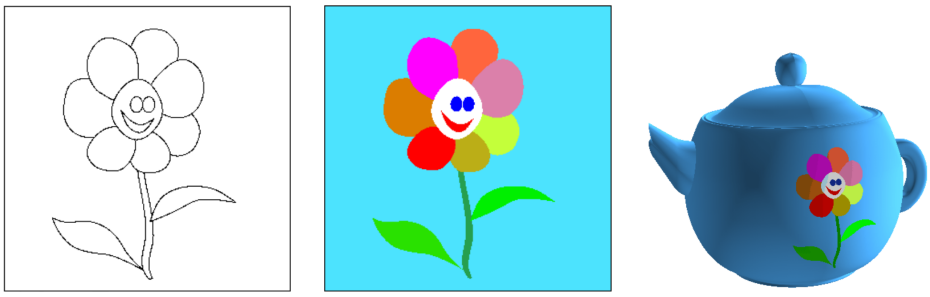
3.1.5 NURBS

Johnson et al. beschäftigte sich mit dem Problem des Malens auf NURBS Modellen [JIK+99]. Dadurch, dass nur einfache Modelle verwendet werden, bei denen eine Textur nahtfrei auf das Modell abgebildet wurde, wird die Zuordnung vom 3D in die Textur stark vereinfacht. Der Pinsel wird direkt im Parameterraum der NURBS getracked, diese Koordinaten sind zugleich die Textur-Koordinaten.

Abbildung 17: Direktes Bemalen einfacher NURBS Modelle [JIK+99]. Links die Textur, rechts das Modell.

Auch Hutchinson et al. projizieren Kurven auf NURBS um so ein Objekt zu kolorieren [HLH96]. Aus dieser Projektion kann auch eine "klassische" Textur erstellt werden. Ein wesentlicher Nachteil liegt hierbei darin, dass eben nur Pfade erstellt werden können, keine Pinsel im Sinne einer 2D Bildbearbeitung.

Da man es meistens mit Modellen aus Polygonen und nicht mit aus NURBS erstellten Objekten zu tun hat, eignen sich diese Painting Verfahren nur in Spezialfällen. Daher wurde in dieser Arbeit auf eine nähere Betrachtung dieser Ansätze verzichtet.



(a) Kurven... **(b)** ...mit Farbfüllung... **(c)** ...werden auf das Objekt projiziert.

Abbildung 18: Surface Graph Sketching nach Hutchinson et al. [HLH96]

3.1.6 Punktwolken

Zwicker et al. schlagen in ihrem Programm Pointshop3D folgendes Vorgehen zum Bemalen von Modellen vor, die aus unzusammenhängenden Punkten bestehen [ZPKG02]: Zunächst wählt der Benutzer eine Untermenge der Punkte aus, nämlich den Bereich, der bearbeitet werden soll. Dieses *Patch* wird dann auf eine Ebene projiziert, auf dieser Ebenen findet dann ein Resampling der Punkte statt, so dass die Punkte auf einem regulären Raster angeordnet sind. In diesem Raster entsprechen die Punkte nun einer zweidimensionalen Abbildung, können also wie ein normales Bild bearbeitet werden. In Abbildung 19 wird dieses Vorgehen verdeutlicht. Das Modell kann aufgrund des Resamplingschrittes modifiziert werden.

Alternativ zur manuellen Selektion eines *Patches* wird beim Malen mit einem Pinsel die Selektion automatisch vorgenommen. Um die Pinselposition herum werden die Punkte, die sich im Radius des Pinsels befinden, orthogonal auf die Pinselebene projiziert. Ihnen wird dann der Farbwert des Pinsels an dieser Stelle zugeordnet (vergleiche Abbildung 20).

Punktwolken können auch zur Texturierung beliebiger Modelle genutzt werden. Reuter et al. [RSSP04] erzeugen hierfür zum Beispiel aus einem Polygonmesh eine Punktwolke, die das gleiche Modell repräsentieren. Nach der Texturierung wird die Textur exportiert um sie mit dem Polygon-basierten Modell nutzen zu können. Ihre Arbeit setzt auf Pointshop3D von Zwicker [ZPKG02] auf, so dass die Texturierung an sich analog funktioniert.

Boubekeur et al. stellen ebenfalls eine Erweiterung für Pointshop3D vor [BS06]. Ihnen geht es darum, Modelle zu bemalen, die zu viele Punkte enthalten um sie komplett interaktiv bearbeiten zu können. Hierfür wird zunächst grob auf eine reduzierte Version gemalt. Um Details hinzuzufügen, muss der Nutzer die gewünschte Region markieren, die dann verfeinert wird.

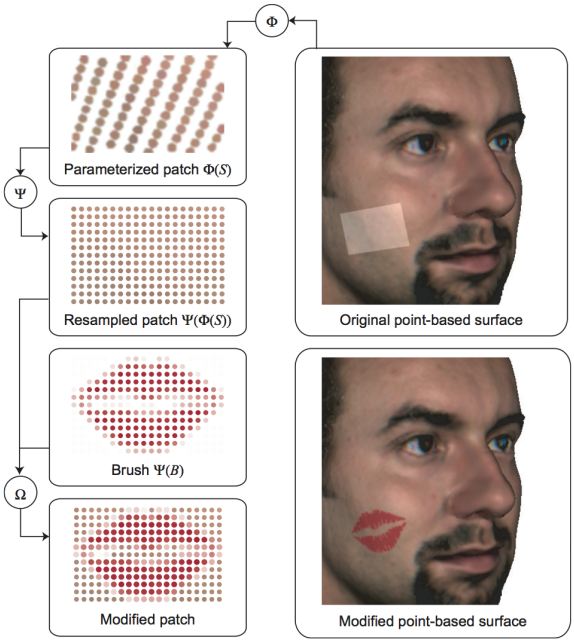


Abbildung 19: Auf dem Modell (rechts oben) wird ein *Patch* selektiert (links oben), dieses wird in einem regulären Raster neu gesampelt (links Mitte). Dieses geänderte *Patch* kann wie eine Bilddatei editiert werden um dann wieder in das Modell eingefügt zu werden (rechts unten) [ZPKG02].

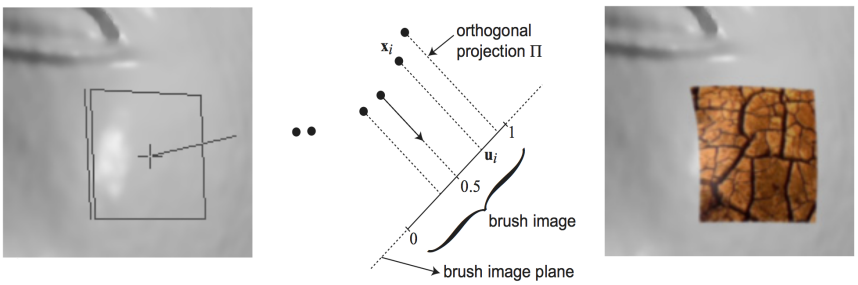


Abbildung 20: Links: Visualisierung der Pinselposition und -ebene. Mitte: Projektion der Punkte in Pinselnähe auf die Pinselebene. Rechts: Das Modell mit geänderten Farbwerten.

3.1.7 Weitere Verfahren

Igarashi et al. stellen in ihrer Arbeit einen 3D Painter vor, der während des Malens gleichzeitig die Textur-Map erstellt [IC01]. Abbildung 21 zeigt, wie ein neuer Pinselstrich der Textur hinzugefügt wird.

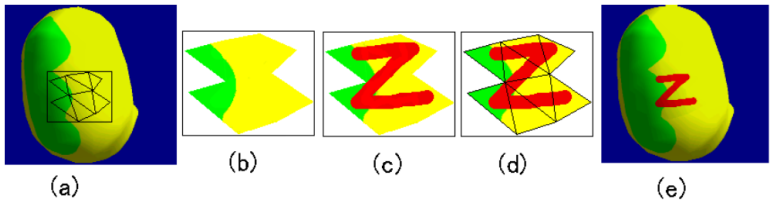


Abbildung 21: Projektion der Pinselstriche bei Igarashi [IC01].

Zunächst werden die Polygone des Modells ermittelt, die sich unter dem Strich befinden (Abb. 21 a). Diese Polygone werden dann mit ihrer bisherigen Textur in der Größe gerendert, in der sie aktuell zu sehen sind (Abb. 21 b). Auf diese 2D Fläche wird dann der Pinselstrich gemalt (Abb. 21 c). Dieses Bild bildet einen neuen *Textur Patch*, die Polygone bekommen neue Textur-Koordinaten zugewiesen, die in diesen *Patch* verweisen. Am Ende der Texturbearbeitung müssen dann alle erstellten *Patches* in eine Textur-Map zusammengefasst werden.

Wird ein Texturbereich mehrfach übermalt, so wird er auch mehrfach in unter Umständen unterschiedlichen Auflösungen neu gerendert und abgespeichert. Hierunter kann die Texturqualität leiden, wie die Autoren einräumen. Zudem weisen sie darauf hin, dass ihr System nicht für professionelle Arbeiten geeignet ist.

3.1.8 Haptisches Feedback

In einigen Arbeiten wird vorgeschlagen, die Texturierung mit einem haptischen Interface [JIK+99] durchzuführen, etwa mit dem PHANTOM [EGL01] [FOL02]. Hierbei handelt es sich um einen im Raum aufgehängten Stift, der frei bewegt werden kann und zudem durch Force-Feedback den Eindruck ermitteln kann, ein reales Objekt zu bemalen.

Der wesentliche Nachteil dieses Gerätes besteht in der geringen Verbreitung, auch aufgrund der hohen Kosten. Aus diesem Grund wurde von einer Verwendung eines PHANTOM abgesehen.

3.2 Bewertung der Verfahren

Verschiedene Arbeiten gehen von einem runden, bzw. kugelförmigen Pinsel aus. Es wird nur das Zentrum und der Radius des Pinsels benötigt. In der Praxis werden aber beliebig geformte Pinsel genutzt. Dies lässt sich am

Abbildung 22: SensAble PHANTOM.

einfachsten bei den Verfahren integrieren, die zunächst in einen "Malpuffer" rendern.

Um eine breite Auswahl an Modellen zu unterstützen, fallen auf NURBS oder Punktwolken basierende Techniken weg. Es sollten Modelle aus Polygonen mit gegebenenfalls bereits vorhandener Textur-Map bearbeitet werden können. Fortgeschrittene Modellierer erstellen die Textur-Map manuell und wünschen oft eine zumindest teilweise Texturierung im 2D. Automatisch generierte Maps, wie in Abbildung 2 eignen sich hierfür nicht. Die Bearbeitung im 2D soll daher ermöglicht werden. Verfahren, die keine gegebenen Textur-Maps nutzen, werden daher auch nicht weiter betrachtet, denn nur wenn eine 2D Bearbeitung angeboten wird, kann eine Aussage über die Akzeptanz der 3D Bearbeitung getroffen werden.

Das projektive Malen bearbeitet ohne Zwischenschritte direkt die Textur, wie sie auch zur Darstellung benötigt wird. Dadurch unterliegt dieses Verfahren aber auch den Nachteilen der Textur-Map: Nähte und Verzerrungen erschweren die Zuordnung vom 3D ins 2D. Bildverarbeitungsfilter, die Nachbarschaftsinformationen benötigen, werden daher nur umständlich umzusetzen sein. Zudem leidet dieses Verfahren unter Artefakten an den Nähten. Auch dieses Problem muss gelöst werden.

Die Bearbeitung der Vertexfarben hat im Wesentlichen mit der extremen Geometriegröße zu kämpfen. Nach dem Bemalen muss aus dem hochaufgelösten Modell für die meisten Anwendungen eine Textur erstellt werden. Es ist also ein Zwischenschritt nötig, bei dem das Modell anders aussehen kann, als das Endergebnis (nämlich dann, wenn die Zieltextur einen geringeren Detailgrad als das hochaufgelöste Modell hat). Wenn nach der Texturierung also eine reguläre 2D Textur für das Ausgangsmodell gewünscht wird, muss für dieses eine Textur-Map vorliegen. Unter solchen Umständen entfällt also der Vorteil, mit unparametriesierten Modellen arbeiten zu können.

Octree Texturen benötigen ebenfalls keine Textur-Map, im Gegensatz zur Bearbeitung der Vertexfarben wird hier auch das Modell nicht geändert. Zudem ist es möglich, ein Objekt mit einer Octree Textur direkt interaktiv darzustellen. Dies ist allerdings weniger performant als reguläre

Texturen, da letztere direkt von der Hardware unterstützt werden, zudem bietet sich diese Möglichkeit nur bei Grafikkarten mit programmierbaren Shadern an.

Wegen der einfachen Möglichkeit, die Textur auch im 2D bearbeiten zu können, habe ich mich daher entschieden, primär das projektive Malen zu untersuchen um herauszufinden, wo die Grenzen bei diesem Ansatz liegen. Wenn sich das Verfahren als robust genug gegenüber Textur-Maps mit vielen Nähten herausstellt, kann sogar auf manuell erstellte Textur-Maps verzichtet werden. Unter den Umständen fiele auch der Nachteil weg, eine Textur-Map zum Modell bereitstellen zu müssen.

Zudem sollen Octree Texturen näher betrachtet werden. Sie erscheinen als interessante Alternative zu klassischen Texturen, da sie auf modernen Grafikkarten direkt dargestellt werden können.

4 Implementierung

Im folgenden Kapitel soll die Software-Architektur der Anwendung MeshPaint beschrieben werden. Dies ist eine Beispielapplikation, die das projektive Malen aus Kapitel 3.1.2 implementiert. Zudem soll diese Anwendung dazu dienen, verschiedene Möglichkeiten der Nahtbehandlung zu untersuchen. Eine Implementierung von Octree Texturen wurde ebenfalls umgesetzt.

Zunächst werden die angestrebten Funktionen aufgelistet um anschließend die Wahl der zugrunde liegenden Software-Bibliotheken zu begründen. Anschließend werden die Architektur im Groben, sowie interessante Bereiche im Detail beschrieben.

4.1 Funktionalität

4.1.1 Dateiformate

Um Texturen auf 3D Modellen bearbeiten zu können, muss zunächst einmal mindestens ein gängiges 3D Format eingelesen werden können. Weit verbreitet ist das *Wavefront .obj* Format. Es kann von vielen Anwendungen geschrieben und gelesen werden, zudem ist es gut dokumentiert und basiert auf reinem ASCII Text. Ein Importer hierfür ist relativ schnell implementiert.

Ebenso müssen Bilddateien eingelesen und geschrieben werden können. Aus Gründen der Qualität kommen beim Speichern nur unkomprimierte, bzw. verlustfrei komprimierte Bildformate in Frage. Werden mehrere Ebenen eingesetzt, so besteht eine einzelne Ebene aus vier Kanälen (Rot, Grün, Blau, Alpha). Zusätzlich zu den Ebenen müssen aber noch weitere Informationen gespeichert werden, etwa das zur Textur gehörende Modell, sowie Licht und Kamerapositionen. Somit genügen selbst Dateiformate, die mehrere Ebenen unterstützen nicht aus um alle nötigen Informationen aufzunehmen. Solche Formate wären zum Beispiel die nativen Formate von *Adobe Photoshop* [Inc08b] oder *The GIMP* [Tea08]. Zudem können diese Formate zu reinen Bitmap-Ebenen noch zusätzliche Elemente enthalten, die eine Unterstützung erschweren, etwa Vektorgrafiken, Textebenen oder Farbprofile.

Daher habe ich mich für .png Dateien für die einzelnen Ebenen, sowie eine XML Datei für die zusätzlichen Informationen entschieden.

4.1.2 Ebenen

In der Bildbearbeitung wird viel mit so genannten Ebenen gearbeitet. Auf diese Weise können verschiedene Elemente des Bildes von einander getrennt bearbeitet und nachträglich einfacher manipuliert werden. Auch der

Beispiel 3D Painter soll das Konzept von Ebenen unterstützen, hierzu gehören auch eine einstellbare Deckkraft und verschiedene Ebenenmodi. Über die Deckkraft können Ebenen teilweise “ausgeblendet“ werden, die Ebenenmodi ändern die Art, wie Bildbestandteile vermischt werden. Abbildung 23 macht das Konzept deutlich.

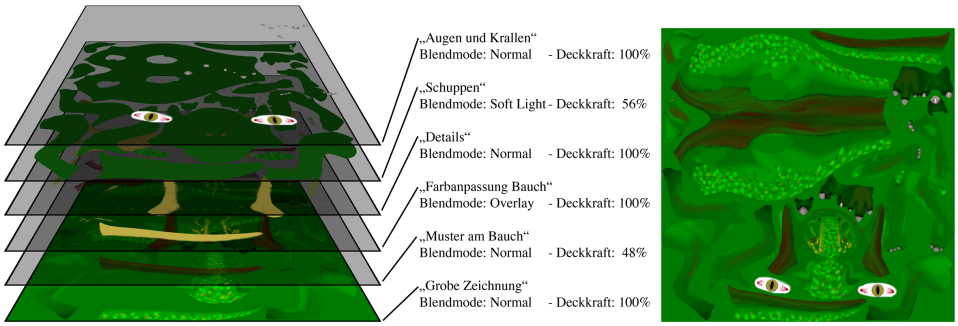


Abbildung 23: Sechs Textur-Ebenen mit unterschiedlichen Blendmodi und Transparenzen. Dargestellt ist eine Textur für das Killeroo Modell, dieses Modell ist mit der hier dargestellten Textur auf Seite 88 zu sehen.

4.1.3 Unterstützung von Grafiktabletts

Unter Grafikern weit verbreitet sind Grafiktabletts. Diese bestehen aus einem Tablett, dessen Größe etwa von DIN A6 bis DIN A3 variiert sowie einem Stift, mit dem auf diesem Tablett der Cursor gesteuert werden kann. Im Gegensatz zur Maus kann ein Tablett auch den Druck, mit dem gemalt wird, auswerten. Der Druck steuert dann die Transparenz oder Größe des Pinsels. Dies soll auch in der Applikation unterstützt werden.

4.1.4 Unterstützte Plattformen

Um möglichst flexibel zu bleiben, soll die Anwendung plattformunabhängig gestaltet werden. Zumindest die Betriebssysteme Microsoft Windows XP sowie Mac OS X von Apple sollen unterstützt werden. Diese Systeme haben auf dem Desktop die größte Verbreitung.

4.1.5 Performance

Eine Texturierung im 3D ist erst dann sinnvoll, wenn auch große Modelle sowie große Texturen interaktiv bemalt werden können. “Groß“ bedeutet in beiden Fällen, dass es sich um Modelle handeln soll, die auf einer aktuellen Grafikkarte auch interaktiv darstellbar sind. Es sollen also Modelle texturiert werden können, die etwa für Spiele verwendbar sind.

Somit sollten mindestens 20.000 Polygone sowie Texturen von 1024^2, besser 2048^2, Texeln interaktiv bemalt werden können.

4.2 Wahl der Bibliotheken

Da die Performance sehr entscheidend ist und sich der grundlegende Mal-Algorithmus komplett auf der Grafikkarte implementieren lässt, fiel die Wahl der Grafikbibliothek auf OpenGL. Der Vorteil von OpenGL im Gegensatz zu DirectX liegt in der Plattformunabhängigkeit. Als Shadersprache sollte GLSL eingesetzt werden, da diese Sprache ein Bestandteil von OpenGL ist und somit keine zusätzlichen Abhängigkeiten erzeugt.

Für die grafische Oberfläche fiel die Wahl auf QT 4 von Trolltech. Dieses GUI-Toolkit ist ebenfalls plattformunabhängig. Zudem unterstützt es sehr gut OpenGL und bietet Routinen zum Laden und Speichern gängiger Bildformate. So unterstützt QT unter anderem auch das bereits angesprochene PNG Format. Des weiteren bietet dieses Toolkit eine plattformübergreifende Anbindung an Grafiktabletts. Leider gibt es zum Auslesen von Grafiktabletts sonst nur plattformspezifische Lösungen.

Als Alternative wurde GTK erwogen, auch diese GUI-Bibliothek ist plattformübergreifend und bietet ähnliche Features. Allerdings war die Unterstützung von Mac OS X zu Beginn der Implementierungsphase noch nicht sehr weit fortgeschritten. So lies sich GTK nur über eine zusätzliche Zwischenschicht zwischen dem Toolkit und dem Betriebssystem nutzen (X11, ein auf Mac OS X portierter Unix-Windowserver). Ausschlaggebend für QT war die sehr gute online Dokumentation und die Verwendung von QT in MeshLab (siehe unten).

Zum Einlesen von gängigen 3D Formaten wurden eine Reihe von Programmen und Bibliotheken betrachtet.

- Ein Plug-in für Adobe Photoshop [Inc08b] hätte den Vorteil, eine vielen Künstlern bekannte Umgebung zu bieten und gleichzeitig auf eine Große Anzahl an Malwerkzeugen zurück greifen zu können. Es stellte sich bei genauerer Untersuchung des SDK heraus, dass ein Plug-in immer nur Zugriff auf eine Textur-Ebene hat. Nach dem Start des Plugins könnte zwar auf die akuelle Textur-Ebene gemalt werden, eine Voransicht wäre hingegen nicht möglich gewesen, da hierfür alle Ebenen gelesen werden müssten.

- G3D [Hil08] ist eine plattformübergreifende 3D Engine unter der BSD Lizenz. Sie kann verschiedene Bild- und 3D-Formate einlesen, unter anderem JPEG und PNG für Bilder, sowie 3DS, MD2 und BSP für Modelle. Das weit verbreitete OBJ Format fehlt hingegen.

- Bei Irrlicht [Geb08] handelt es sich um eine weitere 3D Engine. Auch Irrlicht ist open source und plattformübergreifend, darüber hinaus

kann diese Bibliothek eine große Anzahl an Bild- und Modellformaten verarbeiten, auch OBJ. In einem Prototypen des 3D Painters stellte sich allerdings heraus, dass bei Irrlicht der Entwicklungsaufwand auch bei einfachen Aufgaben bereits sehr hoch ist. Die Bibliothek schien daher für diese Aufgabe kaum geeignet zu sein.

- Für die 3D Engine Ogre [Ltd08] sprach, dass sie sehr gut dokumentiert ist. So existiert auch ein Buch zu dieser Bibliothek (*Pro OGRE 3D Programming*). Allerdings unterstützt sie nur ihr eigenes 3D Format.

- Die Integration der Texturerstellung in ein Modellierungs Tool bietet den Vorteil, dass der Künstler nicht zwischen zwei Programmen wechseln muss. Es hat aber den Nachteil, sich auf ein Programm fest zulegen. Trotzdem wurde eine Integration in das open source Programm K-3D erwogen. Dieses nutzt für die Erstellung der GUI allerdings das bereits besprochene Toolkit GTK. Darüber hinaus läuft es nur unter Windows und Linux stabil.

- Bei MeshLab [Cig08] handelt es sich um ein Programm zum Editieren von 3D Modellen mit Hilfe von Filtern. Es kann daher eine große Anzahl an Formaten sowohl lesen, als auch schreiben, unter anderem OBJ, 3DS und PLY. Das Programm steht unter der GPL und nutzt als GUI Toolkit QT 4, zudem ist es unter Windows, Mac OS X und Linux lauffähig. Da sowohl die Filter, als auch die Import/Export Module als Plug-ins realisiert sind, erhält man durch Weglassen dieser eine relativ schlanke und überschaubare Anwendung. Hierdurch wird es erleichtert, die Anwendung an die eigenen Bedürfnisse anzupassen. Aus diesen Gründen habe ich mich für eine Anpassung von MeshLab entschieden. Abbildung 24 zeigt einen Screenshot der Anwendung.

4.3 Beschreibung der Softwarearchitektur

Im folgenden soll die Architektur der Anwendung beschrieben werden. Zunächst gebe ich eine Übersicht über alle Bestandteile und gehe darauf ein, welche von dem Programm MeshLab "ererbt" wurden. Dann sollen interessante Teilbereiche näher erläutert werden.

4.3.1 Klassendiagramm

Eine der zentralsten Klassen ist die *GLArea*, welche von dem QT Widget *QGLWidget* abgeleitet wurde. Hierdurch erhält die Klasse einen OpenGL Kontext und das Aussehen wird durch OpenGL Zeichenbefehle gesteuert. *GLArea* wurde aus MeshLab übernommen und stark angepasst, da das Malen auf dem 3D Objekt, sowie die Vorschau in dieser Klasse statt findet.

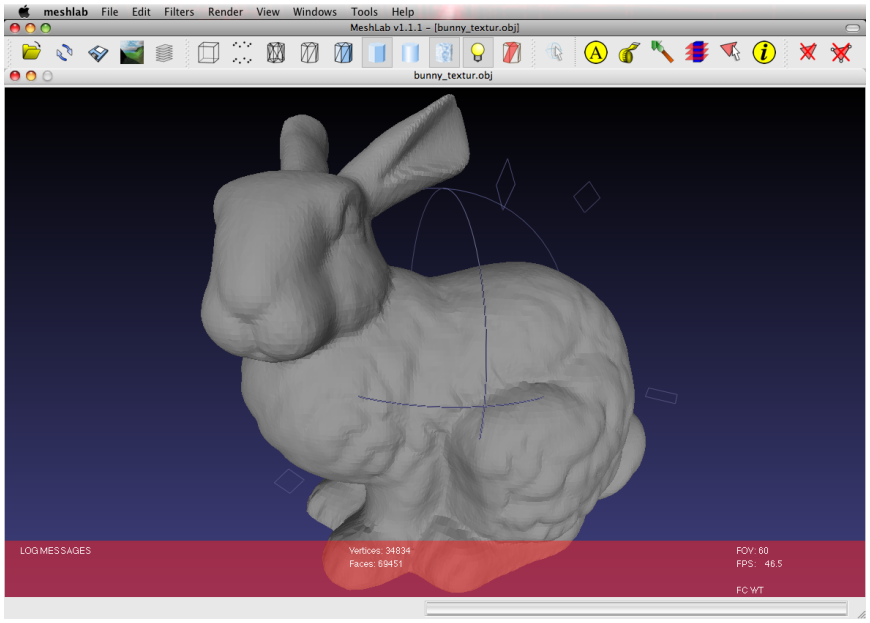


Abbildung 24: Das Programm MeshLab auf dem in dieser Arbeit aufbaut wurde.

Die *GLArea* ist in einem kleinen Fenster innerhalb des Programmfensters untergebracht, was durch ein *QMdiSubWindow* implementiert wurde. Dieses ist das Hauptelement des von *QMainWindow* abgeleiteten *MainWindow*. Das *MainWindow* erstellt die Menus, Toolbars und Short-Cuts und wurde ebenfalls von MeshLab übernommen. Da in dieser Anwendung aber andere Menus und Buttons benötigt wurden, wurde auch diese Klasse stark angepasst. Die Icons für die Buttons wurden teilweise von MeshLab übernommen, teilweise stammen sie von der Grafikapplikation GIMP.

Das *MainWindow* ist zudem für das Laden von Plug-Ins verantwortlich. Von MeshLab wurden hier die Plug-Ins zum importieren verschiedener 3D Formate übernommen. Wird ein Modell geladen, so wird ein Objekt vom Typ *GLArea* erzeugt und diesem ein Objekt vom Typ *MeshDocument* übergeben. Letzteres ist der MeshLab Container für ein 3D Modell. Nachdem der OpenGL Kontext erstellt wurde, erzeugt *GLArea* Objekte der Typen *DrawPlane*, *BrushManager*, *ShaderManager* und *TextureStack*. All diese Klassen erzeugen GLSL Shader und/oder Texturen, die nur in diesem OpenGL Kontext gültig sind.

Die *DrawPlane* verwaltet die Bildschirmfüllende Textur, in die die Pinselstriche zunächst gemalt werden, bevor die komplette Linie auf die Textur projiziert wird. Wenn der Nutzer malt, werden die Bildschirmkoordinaten eines Maus Events dieser Klasse übergeben. Die Pinsel Textur wird dann in der eingestellten Größe an dieser Stelle in die Puffer-Textur gerendert. Um auch bei räumlich weit auseinander liegenden Koordinaten

eines Pinselstriches eine durchgezogene Linie zu erhalten (und nicht etwa einzelne “Pinseltupfer“) wird zwischen den Koordinaten gegebenenfalls interpoliert.

Der *BrushManager* lädt aus einem Verzeichnis alle *.png Bilddateien und erstellt aus ihnen Pinsel, wobei die Helligkeit in den Bildern als Transparenz des Pinsels interpretiert wird. Er erstellt zudem Vorschaubilder der Pinsel, hierbei handelt es sich um Kantenbilder der Pinsel (siehe Abbildung 25).

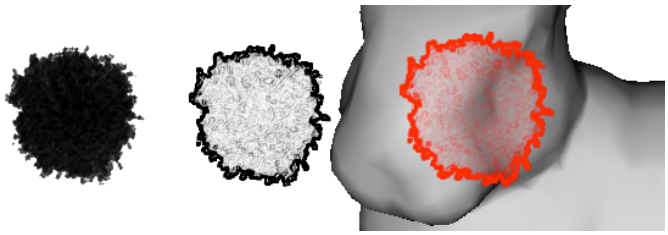


Abbildung 25: Von links nach rechts: Der Pinsel, das zur Laufzeit erstellte Kantenbild, die mit der Pinselfarbe eingefärbte Vorschau in der Anwendung MeshPaint.

Der *ShaderManager* liest die GLSL Shaderprogramme ein und verwaltet sie. Genaueres zum Umgang mit Shadern folgt in Abschnitt 4.3.3.

Der *TextureStack* verwaltet alle Ebenen der Textur und erstellt bei Bedarf eine Vorschautextur. Im Abschnitt 4.3.2 wird dies im Detail erläutert.

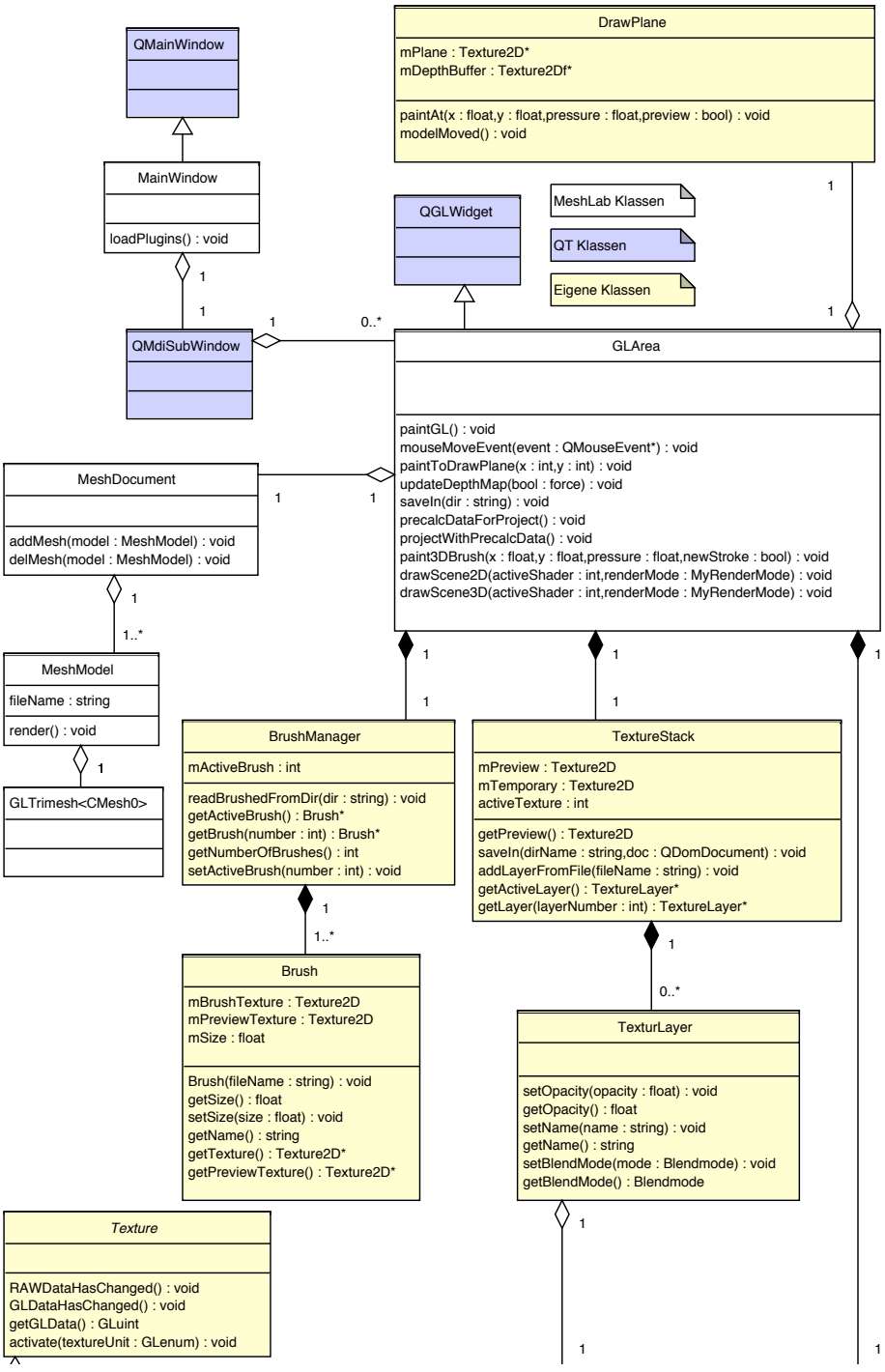


Abbildung 26: Das Klassendiagramm der Anwendung (1/2)

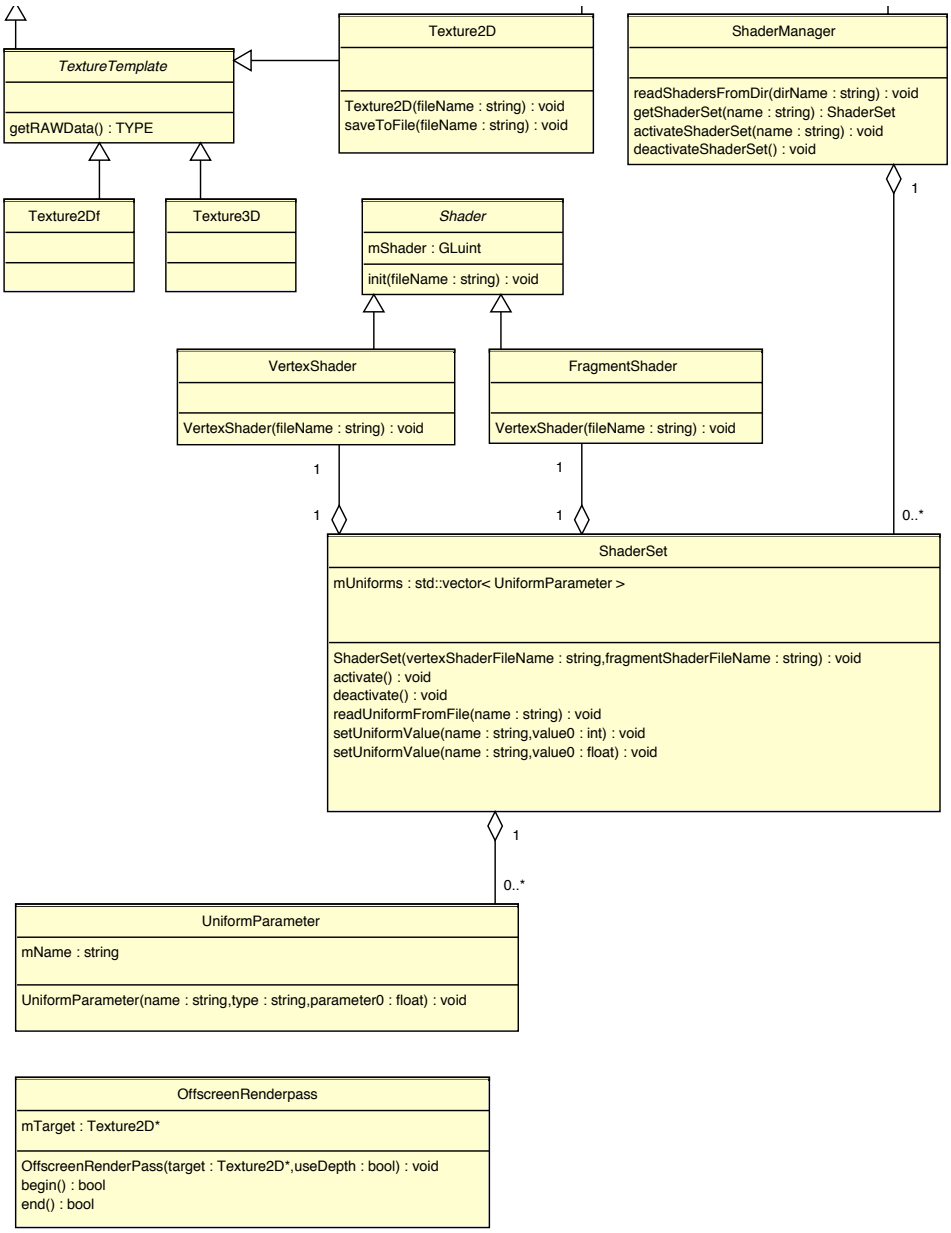


Abbildung 27: Das Klassendiagramm der Anwendung (2/2): Lila unterlegte Klassen gehören zum GUI Toolkit QT, weiß unterlegte Klassen sind Teil von MeshLab, gelb unterlegte Klassen wurden in dieser Arbeit neu hinzugefügt. Von MeshLab wurden nur die relevanten Klassen ins Diagramm übernommen.

4.3.2 Texturhandling

Die Handhabung der Texturen ist ein wichtiger Aspekt dieser Software, schließlich handelt es sich primär um einen Textureditor. Die Textur wird beim Malen mehrfach in der Sekunde geändert. Zur Anzeige muss die Textur in den Speicher der Grafikkarte kopiert werden, viele Änderungen können zudem direkt auf der Grafikkarte mit Hilfe von Offscreen Renderpasses durchgeführt werden. Es kann allerdings auch vorkommen, dass für einen Algorithmus kein geeignetes Vorgehen auf der GPU gefunden werden kann. Dann müsste die Textur in den Hauptspeicher zurückkopiert werden um sie zu editieren und anschließend zur Anzeige wieder zurückkopiert werden.

Gerade dieses Kopieren stellt einen Flaschenhals dar. Daher wird jede Textur zunächst im RAM erzeugt und diese Kopie als *aktuell* markiert. Wird auf die dazugehörige OpenGL Textur zugegriffen, so wird die Textur auf die Grafikkarte kopiert und auch diese Kopie als *aktuell* markiert. Wenn nun eine dieser Kopien geändert wird, wird die jeweils andere als *veraltet* markiert, aber noch nicht aktualisiert. Erst wenn auf eine veraltete Kopie zugegriffen wird, wird die Textur aktualisiert (also von der Grafikkarte in den Hauptspeicher kopiert, bzw. umgekehrt vom Hauptspeicher auf die Grafikkarte geladen). Auf diesem Weg werden unnötige Kopien vermieden. Trotzdem kann jederzeit auf die Textur im Hauptspeicher, bzw. auf die OpenGL Textur zugegriffen werden, ohne wissen zu müssen, welche Version der Textur aktuell ist und ob unter Umständen eine Kopie benötigt wird. Dieses Vorgehen wird als *lazy evaluation* bezeichnet.

Die Textur, die der User malt, kann in der Tat aus beliebig vielen Ebenen bestehen, die mit verschiedenen Blendmodi und Transparenzen ineinander gerechnet werden. Dieser Ebenenstack kann aber nicht direkt angezeigt werden, da in OpenGL maximal acht Texturen zur gleichen Zeit genutzt werden können. Daher bietet die Klasse *TextureStack* eine Vorschau des gesamten Stacks an. Dies ist eine einzelne Textur, in die alle Ebenen zuvor zusammengefügt wurden. Die Vorschau wird mit Hilfe eines oder mehrerer Offscreen Renderpasses erzeugt. Der *TextureStack* errechnet allerdings nur dann eine neue Vorschau, wenn sich mindestens eine Ebene seit der letzten Vorschauerzeugung geändert hat.

Die einzelnen Ebenen bestehen aus einer Textur, dem aktiven Blendmodus, der Deckkraft und einem Namen.

4.3.3 Shaderanbindung

Da diese Anwendung sehr shaderlastig ist und oft zwischen verschiedenen Shadern umgeschaltet werden muss, war es wichtig, die Nutzung von Shadern möglichst einfach und dennoch flexibel zu gestalten.

Die Kombination von Vertex- und Fragment-Shader, sowie weitere zur Laufzeit benötigte Informationen zu diesen Programmen (etwa zu setzende Uniformparameter) werden im weiteren als *ShaderSet* bezeichnet. Um ein neuen *ShaderSet* einzubinden muss im Unterordner *meshpaint-shaders* der Anwendung eine neue XML-Datei angelegt werden, gegebenenfalls auch die Dateien mit dem Vertex-Shader bzw. Fragment-Shader Quellcode. In der XML-Datei stehen die Dateinamen vom Vertex- bzw. Fragment-Shader, daher können sich verschiedene *ShaderSets* die gleichen Shader teilen. Aus diesem Grund muss es nicht zwangsläufig für einen neuen *ShaderSet* auch neue Shader geben.

Ein Beispiel für die XML-Dateien zeigt Listing 1. Neben den Shadern selbst wird hier noch angegeben, wie viele Textureinheiten genutzt werden, damit automatisch die richtige Anzahl an Textureinheiten aktiviert wird.

Listing 1: Eine XML Datei für ein *ShaderSet*

```
<meshpaintshader>
  <vertexshader>mixing.vert</vertexshader>
  <fragmentshader>mixing.frag</fragmentshader>
  <neededtextureunits>4</neededtextureunits>
</meshpaintshader>
```

Der *ShaderManager* liest automatisch alle *.xml Dateien im Unterordner *meshpaint-shaders* ein und lädt die dazugehörigen Shader, um aus diesen je ein Objekt der Klasse *ShaderSet* zu erstellen. Zu diesem Zeitpunkt werden die Shaderquellcodes nach dem Schlüsselwort *uniform* durchsucht, um alle verwendeten Uniformvariablen zu ermitteln und dem *ShaderSet* hinzuzufügen. Der Typ des Uniform wird anhand des im Quellcode angegebenen Typs ermittelt und Startwerte für diesen Uniform können im Quellcode des Shaders angegeben werden. Da dies in GLSL nicht vorgesehen ist, stehen diese als Kommentar direkt hinter der Deklaration. Listing 2 zeigt ein Beispiel. Mit `//` wird ein Kommentar in GLSL eingeleitet, der *ShaderManager* sucht in Zeilen mit einer Uniformdeklaration nach einem Kommentar und in diesem nach den ersten Zahlen, die dann als default Werte genutzt werden. Eine Raute beendet dieses Parsen, so dass weiterer Text auch für diesen Parser als Kommentar erscheint.

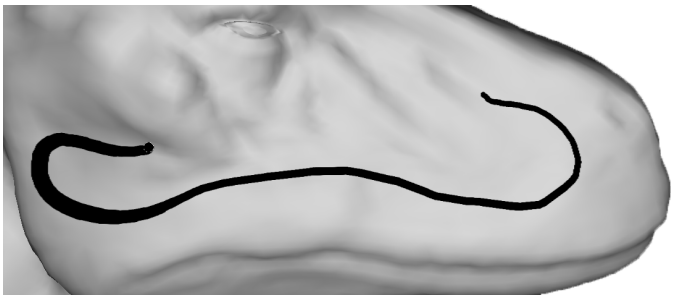
Listing 2: Defaultwerte für Uniformparameter können im Shader als Kommentar angegeben werden. Beim Laden des Shaders werden diese speziell nach Uniformparametern durchsucht.

```
uniform vec4 brushcolor; // 1.0 1.0 1.0 1.0 # the color of the brush as RGBA
uniform float brushOpacity; // 1.0 # the opacity in a range from 0 to 1
```

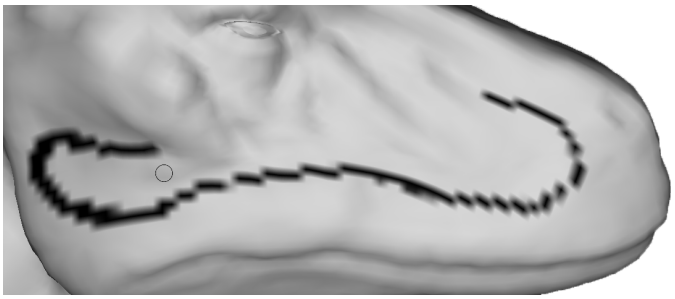
4.3.4 Umsetzung der Mal-Algorithmen

Es wurden zwei unterschiedliche Werkzeuge zum Malen umgesetzt, den *brush* und den *airbrush*. Die weitere Werkzeuge (*eraser*, *color burn*, *color dodge*) basieren technisch auf dem *airbrush* und werden in Kapitel 4.4 näher erläutert. Der Unterschied zwischen dem *brush* und dem *airbrush* liegt in der Abbildung vom 3D Modell zur 2D Textur-Map.

Airbrush Beim *airbrush* malt der Nutzer seinen Pinselstrich auf eine gedachte Scheibe vor dem Modell. Dieses bereits jetzt zweidimensionale Bild wird dann auf das Modell projiziert. Dies hat zur Folge, dass der Nutzer erst nach dieser Projektion das exakte Ergebnis sieht. Malt er zum Beispiel mit einer (in Relation zur Texturauflösung) sehr kleinen Pinselspitze, so sieht er diese feine Linie solange sie nicht projiziert wurde. Nach der Projektion hingegen kann diese Linie dünner als ein Texel sein und somit nicht dem erwarteten Ergebnis entsprechen. Dieses Problem wird in Abbildung 28 veranschaulicht.



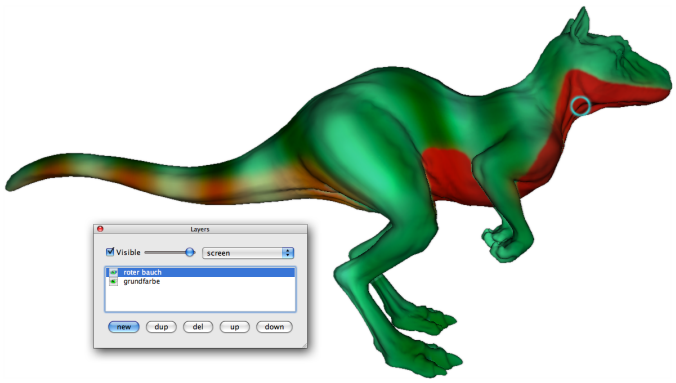
(a) Beim Malen



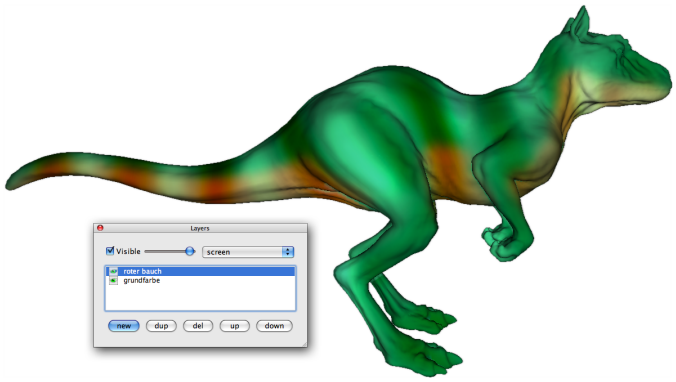
(b) Nach der Projektion

Abbildung 28: Ist die Texturauflösung sehr gering und der Pinsel in Relation sehr klein, so täuscht die Vorschau des *airbrush* einen zu hohen Detailgrad vor.

Das Zwischenspeichern der gemalten Striche hat noch einen zweiten Nachteil: Solange sie nicht projiziert wurden, liegen sie vor der Textur des Modells, dies bedeutet auch, dass der Ebenenmodus sowie die Transparenz der aktuellen Ebene in der Vorschau nicht berücksichtigt werden. Das Gleiche gilt für Ebenen, die über der aktuellen Ebene liegen und das Gemalte unter Umständen verdecken. Abbildung 29 zeigt ein Beispiel einer fehlerhaften Vorschau: In der unteren Ebene sind die senkrechten grünen Streifen gemalt worden, in die obere Ebene soll die Einfärbung des Bauches gemalt werden. Damit diese die senkrechten Streifen nicht überdeckt, sondern nur rot einfärbt, wurde der Ebenenmodus *screen* gewählt. Während des Malens sieht der Nutzer nur die Vorschau des Pinselstriches, der Ebenenmodus wird ignoriert.



(a) Beim Malen



(b) Nach der Projektion

Abbildung 29: In Kombination mit Ebenenmodi kann die Vorschau vom *airbrush* falsch sein.

Der wesentliche Vorteil dieses Vorgehens liegt darin, dass der Nutzer interaktiv malen kann und die aufwändige Projektion nicht interaktiv durchgeführt werden muss. Zwischen dem Nachteil der nicht exakten Vorschau und der teuren Projektion muss ein sinnvoller Kompromiss gefunden wer-

den, da neben der Projektion an sich auch jedes mal eine Neuberechnung der Vorschautextur aus dem Texturstack erfolgen muss. In MeshPaint wird ein Pinselstrich jeweils nach dem Beenden einer durchgezogenen Linie, also nach dem Loslassen der linken Maustaste, auf die aktuelle Textur projiziert.

Umsetzungsdetails Vereinfacht arbeitet der *airbrush* wie in Listing 3 dargestellt. Die Bewegungen der Maus, bzw. des Grafiktabletts erzeugen Maus Events in der Klasse *GLArea*.

Listing 3: Pseudocode des *airbrush*.

```
1  if ( mouseEvent == MouseMoved ) {
2    DrawPlane->paintAt( mouseEvent.x, mouseEvent.y, ↩
       1.0 );
3    paintGL();
4
5  } else if ( mouseEvent == MouseReleased ) {
6    projectDrawPlane();
7    DrawPlane->reset();
8    paintGL();
9
10 }
```

Wird die Maus bei gedrückter linker Maustaste bewegt, so wird in die *DrawPlane* an der aktuellen Position des Cursors gemalt. Der dritte Parameter stellt den Druck dar, dieser ist bei der Maus immer 1, beim Grafiktablett entspricht dieser dem Druck, mit dem der Nutzer den Stift aus das Tablett drückt. Mit dem Druck kann zum Beispiel die Pinselgröße beim Malen geändert werden.

Wenn der Benutzer die Maus sehr schnell bewegt, so liegen die einzelnen Punkte der aufeinanderfolgenden Maus Events weiter auseinander, als der Pinsel breit ist. Dies bewirkt, dass keine durchgezogene Linie gemalt wird, sondern nur einzelne Punkte. Um dies zu verhindern, merkt sich die *DrawPlane* die letzte Position an der gemalt wurde und erzeugt zwischen dieser und der neuen Position gegebenenfalls weitere Punkte. Auf diese Weise wird immer eine Linie erzeugt, unabhängig von der Malgeschwindigkeit des Nutzers.

Die *DrawPlane* verwaltet eine Puffer-Textur, in die der Pinselstrich gemalt wird. Hierbei dient diese (anfangs komplett transparente) Textur als Ziel eines *OffscreenRenderPass*, in welche die aktuelle Pinseltextur in der eingestellten Größe (u.U. modifiziert durch den Druck des Tabletts) an der übergebenen Position hinein gerendert wird.

Um die bereits gemalte Linie anzuzeigen, wird nun `paintGL()` aufgerufen. Diese Methode zeichnet das ganze Modell neu und projiziert die Puffer-Textur zur Vorschau auf das Modell. Hierdurch wird die neue Linie bereits korrekt beleuchtet.

Wird die Maustaste losgelassen, bzw. der Stift des Grafiktabletts angehoben, so wird die Puffer-Textur auf die aktuelle Textur-Ebene projiziert. Dies geschieht wie in Abschnitt 3.1.2 beschrieben mit Hilfe eines weiteren *OffscreenRenderPass*. Nun muss die Puffer-Textur mit einer vollständig transparenten Farbe gefüllt werden, damit sie den nächsten Strich aufnehmen kann. Die Methode `projectDrawPlane()` informiert den *TextureStack* darüber, dass sich die aktuelle Ebene geändert hat, dies bewirkt, dass die Vorschautextur verworfen wird.

Das Modell wird nun erneut dargestellt. An dieser Stelle kann der *TextureStack* nicht mehr auf eine vorbereitete Vorschautextur zurückgreifen, sondern muss eine neue errechnen. Auch dies geschieht auf der Grafikkarte.

Wenn bisher davon gesprochen wurde, dass die Projektion des gemalten Striches aufwändig ist und daher nur einmal pro Linie gemacht wird, dann ist dies nicht ganz korrekt. Die Projektion ist in der Tat performant genug, dass sie bei jeder Anzeige vom Modell zur Vorschau durchgeführt wird. Der Grund für die Verwendung der Puffer-Textur liegt darin, dass die Projektion in die Textur-Ebene eine Neuberechnung der Vorschautextur nach sich zieht, welche verhältnismäßig langsam ist (vergleiche Kapitel 5.4). Zudem kann der Puffer nicht direkt der Textur hinzugefügt werden, da zum mischen der Farben sowohl die aktuelle Textur, als auch der Puffer im mischenden Shader gelesen werden müssen. Eine Textur, aus der gelesen wird, darf allerdings nicht gleichzeitig das Renderziel darstellen. Hierbei handelt es sich um eine generelle Einschränkung der Grafikkarten. Daher wird in eine zusätzliche Textur-Ebene gerendert und diese wird dann mit der eigentlichen Textur-Ebene ausgetauscht um eine Kopie der Daten zu vermeiden. Die Klasse *TextureStack* kümmert sich sowohl um diese zusätzliche Textur, als auch darum, dass diese nach der Projektion mit der aktuellen Ebene ausgetauscht wird.

Als positiven Nebeneffekt enthält diese Zusatztextur zu jeder Zeit eine Kopie der aktuellen Textur-Ebene bevor der letzte Strich hineingerechnet wurde. Daher bewirkt ein erneuter Tausch der Zusatztextur mit der aktuellen Ebene ein *undo*.

Die Projektion der Puffer-Textur auf die Textur-Ebene erfordert ein Rendern des Modells, bei dem die Vertex-Koordinaten durch die zugehörigen Textur-Koordinaten ersetzt werden. Hierdurch wird die Projektion auch abhängig von der Größe des Modells. Hinzu kommt, dass die Ausführungszeiten einiger Nahtfilter ebenfalls von der Anzahl an Polygonen abhängen.

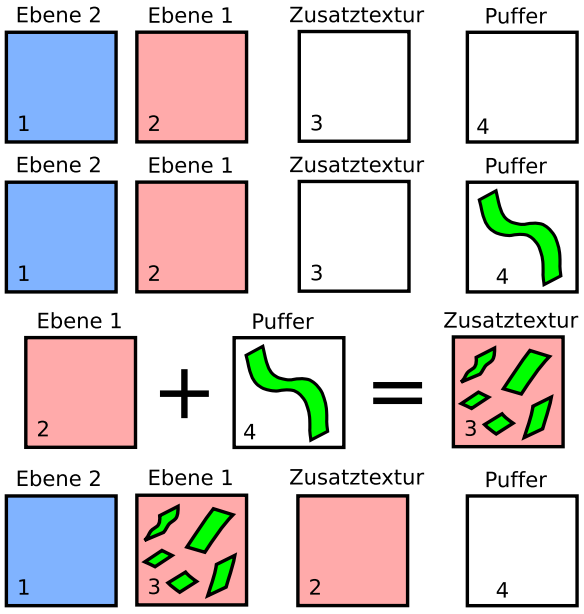


Abbildung 30: Die Texturen beim Malen, von oben nach unten: Zustand vor dem Malen mit zwei Textur-Ebenen, der noch leeren Zusatztextur und dem leeren Puffer. Nun wird in den Puffer gemalt. Bei der Projektion werden die Ebene 1 und der Puffer kombiniert und in die Zusatztextur geschrieben. Die wird dann mit der Ebene 1 ausgetauscht, der Puffer wird im Gegensatz zur Zusatztextur gelöscht. Im Falle eines *undo* werden die Texturen zurückgetauscht.

Die Zuordnung der Texel zur dargestellten Position auf dem Bildschirm und somit zur Position in der Puffer-Textur ist allerdings nur von der Kameraposition abhängig und ändert sich zwischen den einzelnen Pinselstrichen nicht. Auch das Ergebnis vom Verdeckungstest hängt nur vom aktuellen Blickwinkel ab. Je länger der Nutzer malt, ohne die Kamera zu bewegen, umso lohnender wird es, diese Zuordnung nur einmal zu berechnen und zwischen zu speichern. Die Nutzertests haben ergeben, dass durchschnittlich 11,9 Striche gemalt wurden, bevor erneut rotiert wurde. Testkandidat Nr. 8 malte durchschnittlich sogar 44,0 mal, bevor der Blickwinkel auf das Modell geändert wurde (Details zum Test und den Ergebnissen stehen in Kapitel 5.2).

Zur Zwischenspeicherung der Texelzuordnungen wird ein weiterer Renderpass benötigt in dem die Zuordnung ermittelt wird. Dieser Renderpass wird durchgeführt, nach dem die Kamera geändert, und erneut ein Malwerkzeug gewählt wurde. Zwischen der Wahl des Werkzeuges und der Positionierung der Maus über der Stelle, an der gemalt werden soll, vergeht ein kurzer Moment. Dieser reicht im günstigsten Falle bereits aus, um die Vorberechnungen durchzuführen, so dass der Nutzer hiervon gar nichts mit bekommt.

- Erstelle die Depth-Map für die aktuelle Kameraposition
- Erstelle eine temporäre Textur mit der gleichen Auflösung wie die Modelltextur
- Beginne einen Offscreen Renderpass mit der temporären Textur als Ziel:
  - Render gegebenenfalls einen Nahtfilter mit dem Shader *preproject*
  - Render das Modell mit dem Shader *preproject*
- Beende den Offscreen Renderpass

Die Depht-Map wird für den Verdeckungstest benötigt und wäre sonst vor der Projektion erstellt worden. Die Nahtfilter, welche interaktiv genutzt werden können, können in diesem Schritt bereits berechnet werden. Auf die Filter selbst wird im Abschnitt 4.3.5 eingegangen. Hier genügt es zu wissen, dass es die Vorberechnung erlaubt, diese Filter nur einmal pro Kamerarotation auszuführen und nicht mehr einmal pro Projektion.

Im Folgenden soll auf den *preproject* Shader eingegangen werden. Listing 4 zeigt den Vertex-Shader, Listing 5 den zugehörigen Fragment-Shader.

Listing 4: Vertex-Shader zur Vorberechnung der Texelzuordnung.

```
1  varying vec4 projectedVertex;
2
3  void main()
4  {
5    projectedVertex = gl_ModelViewProjectionMatrix*↩
       gl_Vertex;
6    projectedVertex.xyz /= projectedVertex.w;
7
8    // from -1..1 to 0..1
9    projectedVertex.xyz += 1.0;
10   projectedVertex.xyz *= 0.5;
11
12   vec4 myflat = vec4( gl_MultiTexCoord0.x, ↩
       gl_MultiTexCoord0.y, 0.0, 1.0 );
13   // from 0..1 to -1..1
14   myflat.xy = myflat.xy * 2.0;
15   myflat.xy = myflat.xy - 1.0;
16   gl_Position = myflat;
17 }
```

Der `varying vec4 projectedVertex` Parameter dient dazu, die auf den Monitor projizierte Vertex-Koordinate an den Fragment-Shader zu übergeben. Die Projektion findet in den Zeilen 5 bis 10 statt. Die Ausgabeposition des Vertex soll die Textur-Koordinate sein. Diese liegt im Wertebereich von 0 bis 1, die Ausgabe soll den Wertebereich -1 bis 1 haben (Zeilen 12 bis 16).

Der Fragment-Shader bekommt zusätzlich zum `projectedVertex` die Depht-Map übergeben, mit der der Verdeckungstest vorgenommen wird. In der Ausgabetextur soll in den Rot-, Grün-, Blau- und Alpha-Werten die Bildschirmkoordinate des projezierten Texels, sowie eine Aussage über die Sichtbarkeit des Texels stehen.

Rot und Grün nehmen die X- und Y-Koordinate auf dem Bildschirm und somit in der Puffer-Textur auf. Zwar wird der Tiefenwert in den Blaukanal geschrieben, dieser wird für den *airbrush* allerdings nicht verwendet, der *brush* hingegen benötigt auch diesen Wert. Der Alphakanal soll anzeigen, ob das Texel verdeckt wird (es bekommt den Wert 0,0), oder nicht (1,0).

Listing 5: Fragment-Shader zur Vorberechnung der Texelzuordnung.

```
1  uniform sampler2D depthMap; // 0
2  varying vec4 projectedVertex;
```

```
3
4  void main(void)
5  {
6    vec4 outputColor;
7    vec4 depth = texture2D( depthMap, projectedVertex.↩
       xy );
8
9    if ((projectedVertex.x >= 0.0) && (projectedVertex↩
       .x <= 1.0) &&
10       (projectedVertex.y >= 0.0) && (projectedVertex↩
           .y <= 1.0)) {
11
12     if (( projectedVertex.z > depth.r ) || ( ↩
         projectedVertex.z < depth.r-0.1 ) ) { // ↩
         hidden
13       outputColor = vec4(projectedVertex.xyz, 0.0);
14     } else { // front
15       outputColor = vec4(projectedVertex.xyz, 1.0);
16     }
17
18   } else {
19     outputColor = vec4( 0.0, 0.0, 0.0, 0.0 );
20   }
21
22   gl_FragColor = outputColor;
23 }
```

Der Test in den Zeilen 9 und 10 prüft, ob das aktuelle Texel auf dem Bildschirm sichtbar wäre oder nicht. Nicht sichtbare Modellteile sollen auch nicht bemalt werden, daher wird der Wert `vec4(0.0, 0.0, 0.0, 0.0)` in Zeile 19 zugewiesen, entscheidend ist hier nur der Wert für Alpha. Der Z-Wert des Texels wird nun gegen die Depht-Map getestet, da in dieser der Tiefenwert des der Kamera nächstgelegenen Punktes steht. Ein Texel ist sichtbar, wenn der Tiefenwert größer als der in der Depht-Map ist. Allerdings würde dieser Test auch für alle Texel *hinter* der Kamera gültig sein. Daher muss auch dies ausgeschlossen werde. Abhängig von der Sichtbarkeit wird der Alpha-Wert gesetzt (Zeile 13 bzw. 15).

Die Projektion nach jedem Pinselstrich geschieht nun folgendermaßen: In einem Offscreen Renderpass wird wie in Abbildung 30 bereits erläutert in die Zusatztextur gerendert. Gerendert wird ein bildschirmfüllendes Viereck, so dass jedes Texel einmal im Fragment-Shader behandelt wird. Ein minimaler Fragment-Shader steht in Listing 6. Dieser benötigt drei Texturen: Die Textur-Ebene, in die gemalt werden soll (hier: `sourceTexture`

genannt), die Puffer-Textur (`projectionPlane`) und die eben erstellte Textur mit den Zuordnungen (`preprocess`).

Listing 6: Fragment-Shader zur Projektion.

```
1  uniform sampler2D sourceTexture; // 0
2  uniform sampler2D projectionPlane; // 1
3  uniform sampler2D preprocess; // 2
4
5  void main(void)
6  {
7    vec4 outputColor;
8    vec4 source = texture2D( sourceTexture, ↩
       gl_TexCoord[0].st);
9    vec4 prepro = texture2D( preprocess, gl_TexCoord↩
       [0].st);
10   vec4 project = texture2D( projectionPlane, prepro.↩
       st);
11
12   if ( prepro.a > 0.5 ) {
13     outputColor = blendColors( source, project );
14   } else {
15     outputColor = source;
16   }
17
18   gl_FragColor = outputColor;
19 }
```

Für dieses Texel wird die ursprüngliche Farbe `source` aus der Textur an der aktuellen Position gelesen. Ebenfalls an der aktuellen Position wird der Eintrag in der `preprocess` Textur ermittelt. Da in diesem Wert in den Farben Rot und Grün die projizierte Position auf dem Bildschirm steht, muss aus der `projectionPlane` auch an dieser Stelle gelesen werden (Zeile 10).

Ist dieses Texel nicht sichtbar (`prepro.a < 0.5`), so wird die ursprüngliche Texturfarbe beibehalten (Zeile 15). Ansonsten werden die alte und neue Farbe ineinander gemischt. Dies geschieht in einem Aufruf der Funktion `blendColors`.

Listing 7: Funktion zum Mischen von zwei RGBA Werten im Fragment-Shader.

```
1  vec4 blendColors( vec4 base, vec4 blend )
2  {
3    vec4 white = vec4(1.0, 1.0, 1.0, 1.0);
```

```
4
5     // resulting alpha:
6     float alpha = base.a + blend.a - base.a * blend.a;
7
8     // mix base and white:
9     vec3 tmp = base.a * base.rgb + white.rgb * ( 1.0 -↩
          base.a );
10
11    // mix blend and tmp:
12    tmp = blend.a * blend.rgb + tmp.rgb * (1.0 - blend↩
          .a);
13
14    // remove white:
15    tmp -= white.rgb * ( 1.0 - alpha );
16    tmp /= alpha;
17
18    vec4 result = vec4( tmp, alpha );
19    return result;
20 }
```

Der tatsächliche Shader enthält folgende Erweiterungen: Zum einen kann der Verdeckungstest im Listing 6, Zeile 12 durch den *Laser Modus* deaktiviert werden. Hierfür wird ein weiterer Uniform Parameter übergeben. Weiterhin sind in diesem Shader auch die Werkzeuge *color dodge*, *color burn* und *eraser* implementiert. Letzterer verändert den Alpha-Wert der vorhandenen Textur (`outputColor = source*( 1.0 - project.a );`). Bei den anderen zwei Werkzeugen wird die Funktion zum Mischen der Farbwerte ersetzt. Diese beiden Funktionen ändern die vorhandene Farbe, mischen allerdings keine neue hinein, lediglich die Deckkraft des Pinsels ist entscheidend (siehe Listing 8).

Listing 8: Die Werkzeuge *color dodge* und *color burn* im Fragment-Shader.

```
1  vec4 dodge( vec4 base, float dodge )
2  {
3    return (base / vec4( dodge, dodge, dodge, 1.0 ) );
4  }
5  vec4 burn( vec4 base, float burn )
6  {
7    vec3 white = vec3( 1.0, 1.0, 1.0 );
8    vec3 result = (white - (white - base.rgb) / vec3( ↩
       burn, burn, burn ));
9    return vec4( result, base.a );
10 }
```

Dadurch, dass zunächst auf eine Ebene gemalt wird, wird die Geometrie des Modells nicht berücksichtigt. Hierdurch können leicht versehentlich Teile des Modells bemalt werden, die zwar aufgrund der aktuellen Kameraposition nah beieinander liegen, im Modell aber weiter von einander entfernt sind. So betrifft ein Pinselstrich entlang der Kante des vorderen Ohres vom Stanford Bunny in Abbildung 56 (Seite 72) auch das hintere Ohr. Da dieses Verhalten selten erwartet wurde (vergleiche die Auswertung vom Nutzertest in Abschnitt 5.2), wurde nach einer Alternative gesucht, die beim Malen die Geometrie des Objektes berücksichtigt.

Brush Hierbei entstand der *brush*. Diesen kann man sich als Volumen vorstellen, der auf der Oberfläche des Objektes entlangfährt und nur dort Texel einfärbt, wo das Volumen das Objekt schneidet (siehe Abbildung 31). Das Volumen entspricht von vorne (also vom Nutzer aus) betrachtet der Pinselform, wie sie auch vom *airbrush* verwendet wird. In die Tiefe wird diese Form um einen bestimmten Wert verlängert, der nur von der Pinselgröße abhängig ist. Dieser Pinsel hat den Vorteil, dass nur solche Teile des Objektes bemalt werden können, die sich auch im 3D in der Nähe des Pinselmittelpunktes befinden (also in der Nähe des Punktes, den der Künstler mit der Maus angewählt hat).

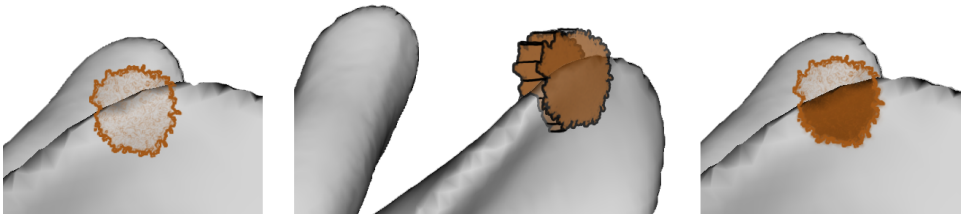


Abbildung 31: Das Werkzeug *brush* ist ein Volumen, das das Modell schneidet. Links die Sicht des Nutzers, in der Mitte sieht man das Volumen, das nur das vordere Ohr schneidet, rechts das Ergebnis des Malens.

Es wird also der Mittelpunkt des Pinselvolumens benötigt, wobei sich die X- und Y-Koordinate aus der Bildschirmkoordinate der Maus direkt ergeben. Die Z-Koordinate, also die Entfernung des Pinsels zum Bildschirm kann ermittelt werden, indem an der Bildschirmkoordinate in den Depht-Buffer des Vorschaubildes nachgeschlagen wird um den vordersten Schnittpunkt mit dem Modell zu ermitteln.

Würde nur das Pinselvolumen mit dem 3D Modell geschnitten und die Texel im Schnittvolumen eingefärbt, könnten auch Rückseiten an dünnen Objekten mit bemalt werden. Um dies zu verhindern muss auch bei diesem Pinsel, wie auch schon beim *airbrush*, ein Verdeckungstest durchgeführt werden.

Der Schnitttest des Pinselvolumens mit dem Objekt ist abhängig vom Pinselmittelpunkt, der sich während eines Pinselstriches mehrfach in der Sekunde ändert. Der gesamte Strich kann also nicht zwischengespeichert werden um am Ende vom Strich erst in die Textur hineingerechnet zu werden. Statt dessen wird bei diesem Mal-Algorithmus die Textur während des Zeichnens aktualisiert. Dies hat den Vorteil, dass der Nutzer eine direkte Ansicht des Ergebnisses erhält, die kurzzeitig ungenaue Vorschau vom *airbrush* entfällt hier also. Der Nachteil liegt allerdings in der langsameren Ausführungsgeschwindigkeit. Ein Vergleich der Performance beider Ansätze folgt in Kapitel 5.4.

Umsetzungsdetails Auch der *brush* nutzt die Vorberechnung der Texelzuordnungen, die im vorherigen Abschnitt vorgestellt wurden. Der Pseudocode dieses Werkzeuges sieht folgendermaßen aus:

Listing 9: Pseudocode des *brush*.

```
1  static bool newStroke = false;
2  if ( mouseEvent == MouseMoved ) {
3    paint3DBrush( mouseEvent.x, mouseEvent.y, 1.0, ↩
       newStroke);
4    newStroke = true;
5    paintGL();
6
7  } else if ( mouseEvent == MouseReleased ) {
8    newStroke = false;
9
10 }
```

Bei jeder Bewegung der Maus wird die aktuelle Pinselposition in die Textur gerendert (Zeile 3), daraufhin wird das Modell neu dargestellt (Zeile 5). Da die aktuelle Textur-Ebene im Aufruf `paint3DBrush` geändert wird, muss beim Rendern des Modells auch eine neue Vorschautextur errechnet werden. Dies ist ein wesentlicher Unterschied zum *airbrush*, bei dem eine neue Vorschau erst am Ende vom Pinselstrich erzeugt werden muss.

Da auch dieser Pinsel bei zu schnellen Bewegungen der Maus keine durchgehende Linie, sondern nur einzelne Punkte malen würde, muss auch hier zwischen zwei Punkten interpoliert werden. Hierfür wird der Methode `paint3DBrush` ein Flag übergeben, das angibt, ob eine neue Linie begonnen wird, oder ein neuer Punkt zu einer alten Linie hinzugefügt wird. Im letzteren Fall wird zwischen der neuen und der letzten Koordinate interpoliert. Die jeweils letzte Koordinate wird in der Methode

`paint3DBrush` gespeichert. Für diese Unterscheidung wird der Boolean `newStroke` benötigt.

Der Dritte Parameter der Methode `paint3DBrush` stellt wie schon beim *airbrush* den Druck dar, welcher bei der Maus als 1,0 angegeben wird, bei einem Grafiktablett hingegen ausgelesen werden kann und zwischen 0 und 1 liegt.

Wie bereits erwähnt wurde, wird neben der X- und Y-Koordinate im Bildschirmkoordinatensystem auch der Z-Wert an dieser Stelle benötigt. Dieser liegt in der Depth-Map bereits vor, welche zur Erstellung der Zuordnungstextur benötigt wurde. Das Auslesen der Textur oder einzelner Werte aus dieser vom Programm aus ist allerdings zu unperformant. Daher wird der benötigte Z-Wert erst im Shader ausgelesen, so wird ein Auslesen der Textur in den RAM gespart.

Da also die Z-Werte ausschließlich im Fragment-Shader zur Verfügung stehen, muss auch die Interpolation zwischen dem aktuellen und dem letzten Punkt einer Linie im Fragment-Shader erfolgen. Diesem Shader muss also die aktuelle X,Y Koordinate, sowie die letzte Koordinate übergeben werden. Des weiteren der aktuelle Druck des Grafiktabletts, sowie der vorhergegangene (da sich der Druck beim Tablett zwischen zwei Punkten ändern kann). Konkret kann der Druck die Größe und die Deckkraft des Pinsels modifizieren, daher genügt es, für diese Werte je den neuen und den alten Wert zu übergeben. Auch die Pinselfarbe und die Information, ob es sich um einen neuen Strich handelt muss als Uniform Parameter übergeben werden.

Der Shader benötigt die ursprüngliche Textur-Ebene, in die gemalt werden soll. Zudem die Zuordnungstextur, sowie die Depth-Map. Anstelle der Puffer-Textur, die hier ja nicht verwendet wird, muss die Textur des Pinsels übergeben werden.

Im Listing 10 kann auch erkannt werden, wie die default Werte der Uniform Parameter direkt im Shader angegeben werden. Bei den Koordinaten, der Deckkraft und der Pinselgröße werden die neuen und alten Werte jeweils in einem Uniform übergeben. Wenn die alte Koordinate negativ ist, so beginnt ein neuer Strich, es darf also nicht interpoliert werden.

Listing 10: Fragment-Shader vom *brush*.

```
1 uniform sampler2D sourceTexture; // 0
2 uniform sampler2D preprocess; // 1
3 uniform sampler2D brushTexture; // 2
4 uniform sampler2D depthMap; // 3
5
6 uniform vec3 brushcolor; // 1.0 1.0 1.0
```

```
7  uniform vec4 brushXY; // -1 -1 1 1 # the old brush position (negative if there was no old brush), new brush position
8  uniform vec2 brushOpacity; // 1 1 # opacityOld, opacityNew
9  uniform vec2 brushSize; // 0.2 0.2 # sizeOld sizeNew
10
11 void main(void)
12 {
13   vec4 source  = texture2D( sourceTexture, gl_TexCoord[0].st );
14   vec4 prepro  = texture2D( preprocess, gl_TexCoord[0].st);
15   vec4 finalColor = source;
16
17   float maximumBrushAlpha = 0.0;
18
19   vec4 depth   = texture2D( depthMap, brushXY.xy );
20   vec3 mybrushxyz = vec3( brushXY.xy, depth.r );
21
22   vec3 brushDimension = vec3( brushSize.x, brushSize.x, brushSize.x );
23   vec3 brushpos = mybrushxyz - brushDimension*0.5;
24   vec3 indexInBrush = vec3( prepro.xyz - brushpos );
25   indexInBrush /= brushDimension;
26
27   if ( all( greaterThanEqual( indexInBrush, vec3( 0.0, 0.0, 0.0 ) ))) {
28     if ( all( lessThanEqual( indexInBrush, vec3( 1.0, 1.0, 1.0 ) ))) {
29       if ( prepro.a > 0.5 ) { // visible
30         vec4 brushAlpha = texture2D( brushTexture, indexInBrush.st );
31         maximumBrushAlpha = brushOpacity.y * (1.0 - brushAlpha.r);
32       }
33     }
34   }
35
36   vec4 brushColorWithAlpha = vec4( brushcolorNew.r, brushcolorNew.g, brushcolorNew.b, maximumBrushAlpha );
37   finalColor = blendColors( source, brushColorWithAlpha );
```

```
38
39    gl_FragColor = finalColor;
40  }
```

Das Listing 10 zeigt die vereinfachte Implementierung des *brush* ohne Interpolation. Zunächst soll ermittelt werden, ob das aktuelle Texel im Pinselvolumen liegt. Ist dies der Fall, so muss abhängig von der allgemeinen Deckkraft des Pinsels und der Deckkraft an dieser Stelle des Pinsels die Pinselfarbe in die Textur hineingemischt werden. Liegt der Texel außerhalb des Pinselvolumens, so soll die alte Texturfarbe übernommen werden.

Hierfür wird die 3D Koordinate des Pinsels ermittelt: Die X- und Y-Position wird über den Uniform `brushXY` übergeben, die Tiefe wird in Zeile 19 aus der Depht-Map ausgelesen. Dann wird die linke untere Ecke des Pinselvolumens errechnet (Zeile 23) um so eine relative Position des aktuellen Texels im Volumen zu erhalten (Zeile 24). Anschließend wird diese Koordinate so normiert, dass alle Punkte im Volumen im Bereich 0..1 liegen.

In den Zeilen 27 und 28 wird getestet, ob das Texel im Pinselvolumen liegt. Schlägt dieser Test fehl, so wird trotzdem in den Zeilen 36, 37 die Pinselfarbe in die Textur gemischt, allerdings ist noch aus Zeile 17 eine Deckkraft von 0 aktuell, so dass letztendlich die alte Texturfarbe übernommen wird. Ob das Texel überhaupt von der Kamera aus sichtbar war, wird wie schon beim *airbrush* aus der Vorberechnung entnommen (Zeile 29).

`brushAlpha` in Zeile 30 ist die aus der Pinseltextur ausgelesene Deckkraft des Pinsels. Diese muss noch mit der vom Nutzer eingestellten allgemeinen Deckkraft multipliziert werden, um die absolute Deckkraft des Pinsels an dieser Stelle zu erhalten. Dies geschieht in der Zeile 31. In der Pinseltextur bedeutet Weiß transparent und Schwarz opak, da so neue Pinsel intuitiv erstellt werden können. Dies ist der Grund für das Invertieren der Pinselfarbe in der Rechnung `brushOpacity.y * (1.0 - brushAlpha.r)`.

Die Funktion `blendColors` ist analog der Funktion beim *airbrush*.

Da schnelle Malbewegungen keine durchgezogene Linie erzeugen, wird wie auch schon beim *airbrush* zwischen dem alten und dem aktuellen Punkt interpoliert. Der Unterschied wird in Abbildung 32 deutlich: Links sieht man das Ergebnis des Shaders 10, rechts mit der Erweiterung aus Listing 11. Der Absatz bei der linken Hinterpfote im rechten Bild erklärt sich aus dem großen Unterschied der Tiefe.

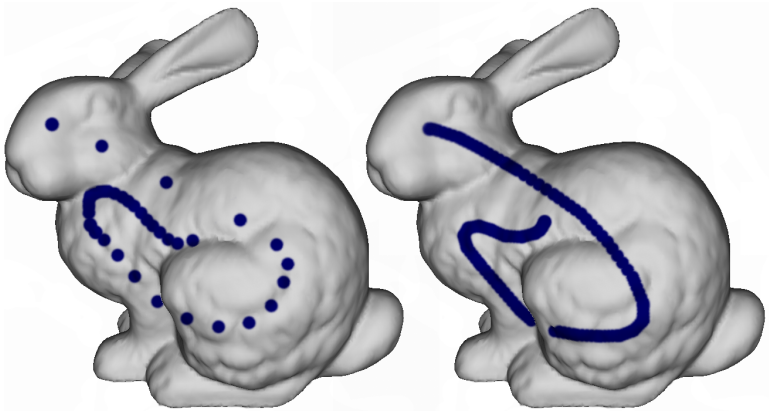


Abbildung 32: Malen ohne (links) und mit Interpolation (rechts).

Listing 11: Fragment-Shader vom *brush*.

```
1  void main(void)
2  {
3    vec4 source  = texture2D( sourceTexture, ↩
       gl_TexCoord[0].st );
4    vec4 prepro  = texture2D( preprocess,    ↩
       gl_TexCoord[0].st);
5    vec4 finalColor = source;
6
7    float startValue = 0.2;
8    float oldZ = 0.0;
9    float newZ = 0.0;
10
11   if ( brushXY.x < 0.0 ) {
12     startValue = 1.0;
13   } else {
14     oldZ = texture2D( depthMap, vec2( brushXY.xy ) )↩
         .r;
15     newZ = texture2D( depthMap, vec2( brushXY.zw ) )↩
         .r;
16   }
17
18   float maximumBrushAlpha = 0.0;
19
20   if ( abs(oldZ - newZ) <= 0.05 ) {
21     for ( float mixValue = startValue; mixValue <= ↩
         1.001; mixValue += 0.2 ) {
22
```

```
23        vec4 depth    = texture2D( depthMap, vec2( mix(↩
              brushXY.xy, brushXY.zw, mixValue ) ) );
24        vec3 mybrushxyz = vec3( mix(brushXY.xy, ↩
              brushXY.zw, mixValue ), depth.r );
25
26        float brSize = mix( brushSize.x, brushSize.y, ↩
              mixValue );
27        vec3 brushDimension = vec3( brSize, brSize, ↩
              brSize );
28        vec3 brushpos = mybrushxyz - brushDimension↩
              *0.5;
29        vec3 indexInBrush = vec3( prepro.xyz - ↩
              brushpos );
30        indexInBrush /= brushDimension;
31
32        if ( all( greaterThanEqual( indexInBrush, vec3↩
              ( 0.0, 0.0, 0.0 ) ))) {
33          if ( all( lessThanEqual( indexInBrush, vec3(↩
                1.0, 1.0, 1.0 ) ))) {
34            if ( prepro.a > 0.5 ) { // visible
35              vec4 brushAlpha = texture2D( ↩
                    brushTexture, indexInBrush.st );
36              maximumBrushAlpha = max( ↩
                    maximumBrushAlpha, brushOpacity.y * ↩
                    (1.0 - brushAlpha.r) );
37            }
38          }
39        }
40
41      }
42
43      vec4 brushColorWithAlpha = vec4( brushcolorNew.r↩
            , brushcolorNew.g, brushcolorNew.b, ↩
            maximumBrushAlpha );
44      finalColor = blendColors( source, ↩
            brushColorWithAlpha );
45    }
46
47    gl_FragColor = finalColor;
48  }
```

Gab es einen vorherigen Punkt in der Linie, beginnt also mit diesem Aufruf keine neue Linie, so ist der Test in Zeile 11 wahr und es werden

die Tiefenwerte der alten und neuen Position ausgelesen. In Zeile 20 von Listing 11 wird getestet, ob diese Z-Werte sich zu sehr unterscheiden. Ist dies der Fall, wird nicht gemalt.

Andernfalls wird für eine Reihe von Punkten zwischen den zwei Koordinaten der Pinsel gemalt. Hierfür wird die Schleife in Zeile 21 eingeführt. Die Laufvariable dient zur Interpolation zwischen den zwei Z-Werten (Zeile 23), den Koordinaten (Zeile 24) und der Pinselgröße (Zeile 26). In Zeile 36 wird die maximale Deckkraft der Schleife ermittelt. Diese dient am Ende dazu, den neuen Farbwert zu errechnen. Sollte es keinen vorhergegangenen Punkt gegeben haben, so wird der Startwert der Schleife so gewählt, dass sie nur ein mal für den neuen Punkt durchlaufen wird (Zeile 12).

Die Namensgebung der Werkzeuge *brush* und *airbrush* basiert auf der Umfrage zum Nutzertest (vergleiche Abschnitt 5.2).

4.3.5 Umsetzung der Nahtfilter

Wie in Abschnitt 3.1.2 gezeigt, entstehen an den Nähten beim projektiven Malen Artefakte. Um diese zu vermeiden, wurden verschiedene Wege untersucht, die im Folgenden vorgestellt werden sollen. Gewünscht wurde zum einen eine hohe optische Qualität, zum anderen ein geringer zusätzlicher Aufwand, da diese Filter im Idealfall interaktiv bei der Texturbearbeitung angewendet werden sollen.

Im Wesentlichen erzeugen diese Filter um die Polygone in der Texur-Map einen zusätzlichen "Rand" um die Artefakte zu beseitigen. Hierbei muss sichergestellt werden, dass dieser Rand nicht andere Polygone überdeckt.

Linefilter Um die Ränder der Polygone "dicker" zu machen und so die bisher nur teilüberdeckten Texel komplett zu überdecken, wurde das Modell in einem weiteren Renderpass als Drahtgittermodell dargestellt. In der Wireframedarstellung kann die Liniendicke in OpenGL vergrößert werden, so dass dieser Ansatz ohne hohen Aufwand implementiert werden kann. Um nicht versehentlich den Inhalt anderer Dreiecke zu überdecken, wurde zunächst das Wireframe gerendert und anschließend das Modell normal drüber gerendert.

Abbildung 33 veranschaulicht die Idee: Wird nur das Polygon gerendert, so werden nur die grünen Texel erzeugt, obwohl auch die gelben leicht überdeckt werden und somit bei der Darstellung Verwendung finden. Dieses Beispiel wurde in Abbildung 13 bereits ausführlicher behandelt. Die dicken Linien auf der rechten Seite der Abbildung stellen das Wireframe mit einer großen Linienbreite dar. Die blauen Texel werden nun zusätzlich erzeugt. Alle benötigten Texel werden allerdings nicht erzeugt, da die Linien des Drahtgittermodells nicht vollständig umschlossen werden.

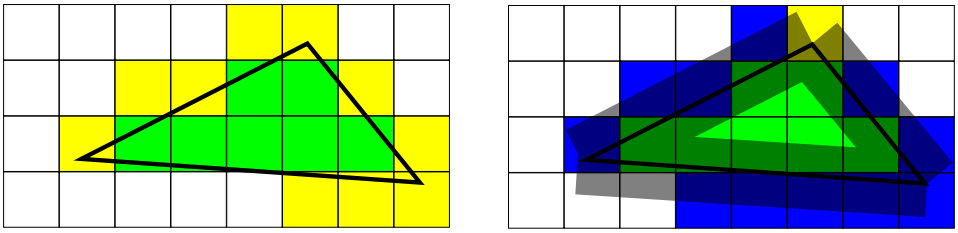


Abbildung 33: Links: Das Polygon aus Abbildung 13. Rechts: Die blauen Texel werden zusätzlich erzeugt.

Ein vollständiges Umschließen lässt sich allein über die OpenGL Pipeline nicht realisieren, allerdings könnte man diese breiten Linien durch ein neues Polygon erzeugen [BGK03] [LHN05]. In diesem Fall lässt sich das Problem allerdings auch einfacher lösen.

Pointfilter Da es beim Linefilter Probleme an den Eckpunkten gibt, bietet es sich an, nur die Vertices in einem weiteren Renderpass darzustellen. Auch diese lassen sich in OpenGL gezielt vergrößern. Abbildung 34 zeigt rot umrandet die vergrößerten Eckpunkte und die hierdurch erzeugten Texel (blau).

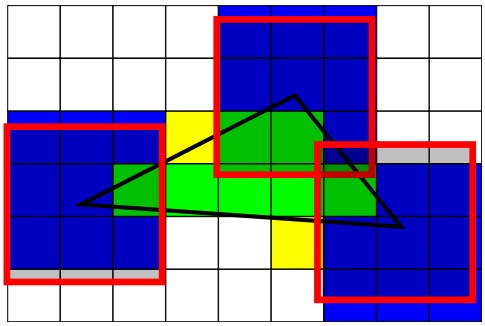


Abbildung 34: Der Pointfilter: Die roten Punkte erzeugen an den Eckpunkten der Polygone neue Texel.

Kombination aus Line- und Pointfilter Dieser Pointfilter allein genügt nur bei relativ kleinen Dreiecken. Daher wurde er mit dem Linefilter kombiliert, das heißt, es wurde zunächst das Modell als Punktwolke gerendert, darauf als Drahtgitter und zuletzt als gefüllte Polygone. Diese Kombination erzeugt sehr gute optische Ergebnisse, ist aber auch durch die drei Renderpasses langsam.

Nachbar Polygone Eine Naht ist eine Kante, an der im Modell zwei Dreiecke aneinander stoßen, wobei diese Dreiecke keine gemeinsamen Textur-

Koordinaten haben. Daher existiert zu jeder Naht in der Textur-Map ein Polygon, das andere Textur-Koordinaten hat, aber an zwei Punkten die gleichen Vertex-Koordinaten.

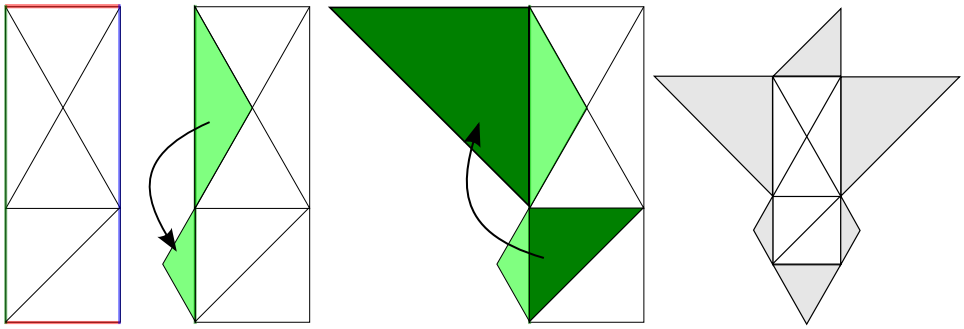


Abbildung 35: Von links nach rechts: Die Textur-Map aus Beispiel 5; an der grünen Naht wird das obere Polygon dupliziert, verkleinert und unten angesetzt; analog wird das dunkelgrüne Polygon oben angesetzt; alle duplizierten Dreiecke in der Textur-Map (grau).

Nun kann man zu allen Nähten die jeweiligen Nachbarpolygone suchen und duplizieren. Das Duplikat behält seine Vertex-Koordinaten, bekommt aber neue Textur-Koordinaten, so dass die im 3D aneinander liegenden Seiten auch in der Textur-Map aneinander stoßen und das duplizierte Dreieck nicht zusätzlich verzerrt wird.

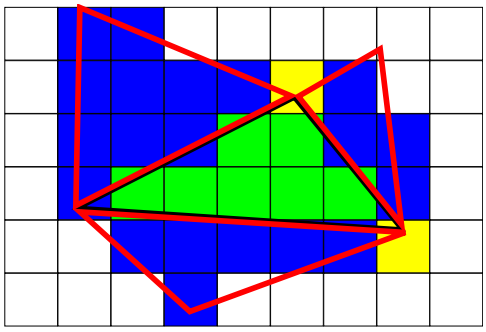


Abbildung 36: Nachbar Polygone.

Der Vorteil liegt bei diesem Ansatz darin, dass nicht nur die Randfarbe der Polygone auf mehr Texel ausgeweitet wird, sondern die exakte Nachbarfarbe erzeugt wird. Wird nämlich auf ein dupliziertes Polygon gemalt, wird die Farbe an mehrere Stellen der Textur-Map gemalt. Doch auch hier kann es an den Ecken zu Artefakten kommen.

Nearest Neighbour Die bisherigen Filter erzeugten direkt beim Malen zusätzliche Texel, indem zusätzliche Geometrie gerendert wurde. Aufgrund

der Vorberechnung der Texelzuordnungen muss dies nicht bei jedem Strich, sondern nur nach jeder Änderung der Kamera erfolgen (vergleiche Kapitel refpreprocess).

Der Nearest Neighbour Filter hingegen erweitert nachträglich die gerenderten Regionen. Ein Pseudocode zu diesem Vorgehen steht in Listing 12.

Listing 12: Pseudocode des Nearest Neighbour Filter.

```
forall (texel) {
  if ( texel != empty ) {
    return;
  }

  numberOfNeighbours = 0;
  sumOfColor = 0;

  forall ( neigbours of texel ) {
    if ( neigbours != empty ) {
      numberOfNeighbours++;
      sumOfColor += neigbours;
    }
  }

  if ( numberOfNeighbours > 0 ) {
    newColor = sumOfColor / numberOfNeighbours;
  }
}
```

Von allen Texeln sollen die betrachtet werden, die nicht durch das Rendern der Geometrie gefüllt wurden. Bei diesen wird in der Nachbarschaft (in der Praxis die direkte 3*3 Nachbarschaft) nach Texeln gesucht, welche gefüllt wurden. Wenn welche gefunden wurden, so wird als neuer Farbwert der Durchschnitt der gefundenen Farben genommen.

Auf diesem Weg wird der Bereich der definierten Texel um einen Texel erweitert (vergleiche Abbildung 37 links). Bei sehr dünnen Dreiecken kann eine Erweiterung um nur einen Texel nicht ausreichend sein (Abbildung 37 rechts), hier kann zum einen die Filtermaske erweitert werden, oder der Filter mehrfach angewendet werden.

Implementiert ist dieser Filter als Fragment-Shader. Zur Unterscheidung zwischen Texeln, die durch das Rendering erzeugt wurden und denen, die geändert werden dürfen, wird eine Maske verwendet. Diese wird erstellt, in dem das Modell in weiß in eine schwarze Textur gerendert wird (auch hier mit den Textur-Koordinaten als Vertex-Koordinaten). Bei mehre-

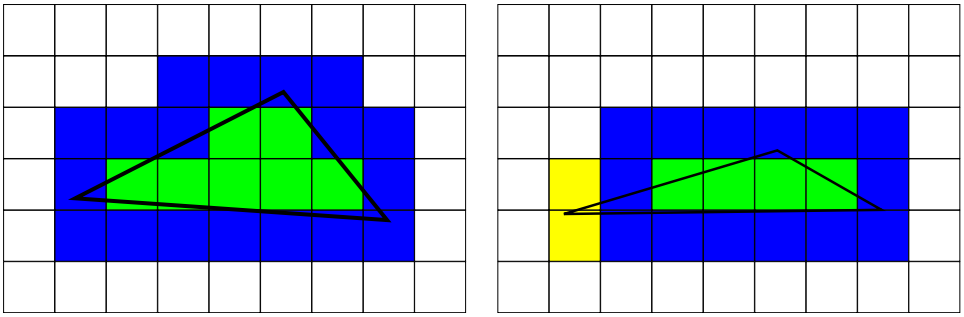


Abbildung 37: Nearest Neighbour Filter: Die Erweiterung um einen Texel reicht bei sehr schmalen Polygonen unter Umständen nicht aus.

ren Durchläufen des Filters muss auch die Maske um einen Texel erweitert werden.

Auf die optische Qualität sowie die Performance dieser Filter wird in Kapitel 5.1 näher eingegangen.

4.4 Umgesetzte Funktionen

Im folgenden soll eine Übersicht über die Funktionen der Anwendung gegeben werden, die dem Nutzer zur Auswahl bereit stehen. Zwar ist es möglich den verwendeten Nahtfilter zur Laufzeit zu ändern, doch sollte der Nutzer im Normalfall nicht mit diesen technischen Details in Berührung kommen.

4.4.1 Die GUI

Abbildung 38 zeigt die Anwendung MeshPaint mit einem geladenen Modell.

Links sind senkrecht aufgereiht die Werkzeuge sowie Farbfelder für Vorder- und Hintergrundfarbe. Oben finden sich von links nach rechts: Öffnen, Speichern, Screenshot speichern, Ebenenpalette ein-/ausblenden, Pinselpalette ein-/ausblenden sowie Einstellungen zur Modellansicht (siehe Abbildung 39). Rechts sieht man die Ebenenpalette sowie die Pinselpalette.

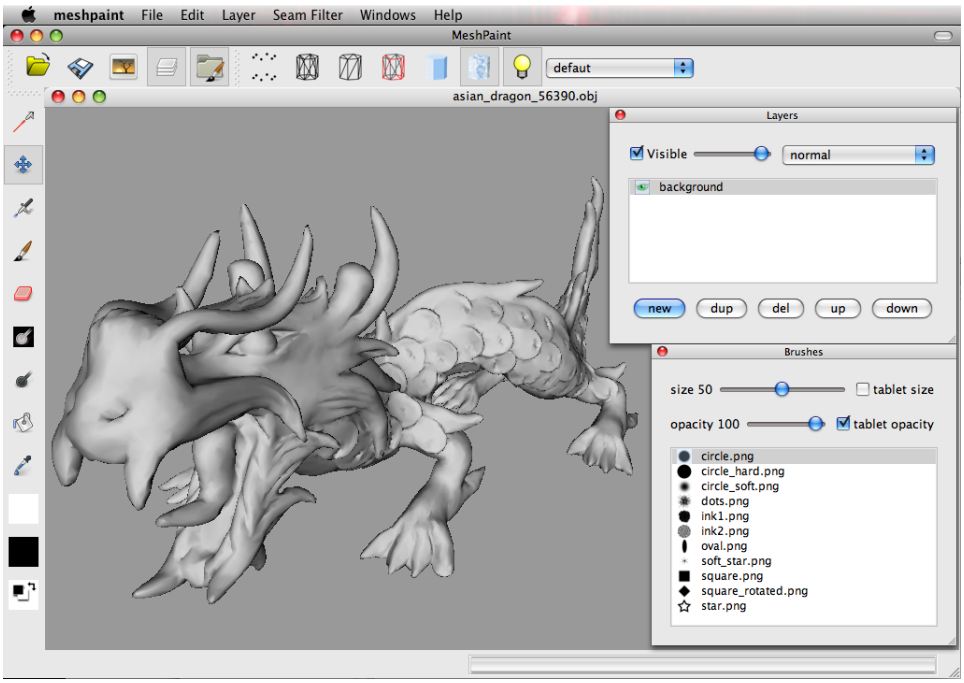
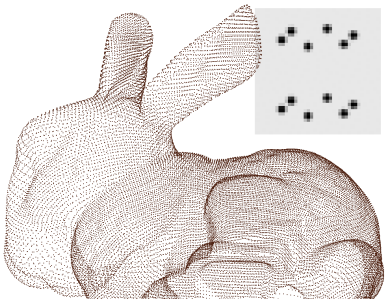
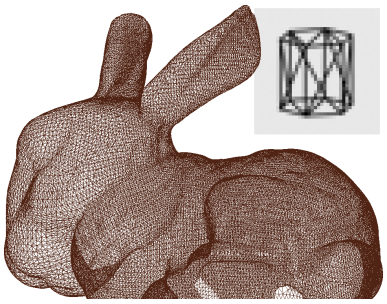


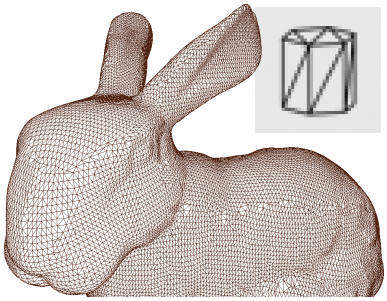
Abbildung 38: Das Hauptfenster von MeshPaint mit geladenem Modell.



(a) Nur Vertices rendern

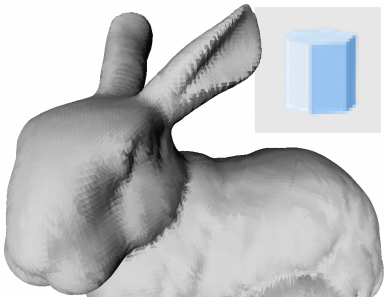


(b) Wireframe

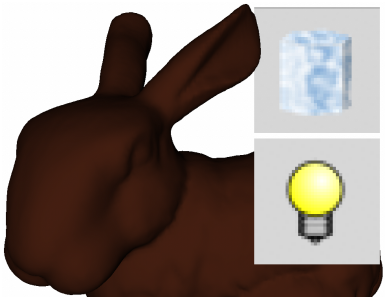


(c) Wireframe ohne Rückseite

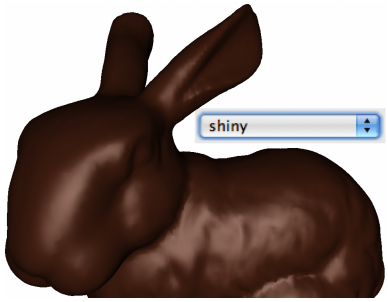
(d) Visualisierung der Nähte der Textur-Map



(e) Ohne Textur



(f) Textur mit per-Fragment Phong-Beleuchtung



(g) Phong-Beleuchtung mit spekularem Term

Abbildung 39: Die verschiedenen Ansichten des Modells mit dem entsprechenden Button aus der oberen Toolbar.

4.4.2 Die Werkzeuge

Kamera Mit dem Kamerawerkzeug kann die Ansicht auf das Modell geändert werden. Translationen der Kamera sowie Rotationen des Modells funktionieren ähnlich wie bei anderen 3D Modellierungsprogrammen. Mit diesem Werkzeug ist es zudem möglich, zwischen einer perspektivischen und orthographischen Projektion hin und her zu schalten.

Airbrush und Brush *Airbrush* und *brush* wurden bereits ausführlich vorgestellt und sind die wesentlichen Malwerkzeuge der Anwendung. Das dritte Icon symbolisiert einen Laser. Die Laseroption kann mit beiden Malwerkzeugen kombiniert werden und schaltet den Verdeckungstest ab. Auf diese Weise kann komplett durch das Modell hindurch gemalt werden. Kombiniert mit einer orthographischen Ansicht auf das Modell können auf diese Art symmetrische Muster auf ein symmetrisches Modell gemalt werden, etwa die Streifen eines Zebras. Die Lasermetapher für diese Funktion findet sich auch bei Igarashi et al. [IC01].

(a) In dieser orthographischen Ansicht wurden die Streifen des Zebras gemalt.

(b) Die Streifen erscheinen nahtlos auch auf der Rückseite.

Abbildung 40: Durch die Laseroption des *airbrush* kann symmetrisch auf Vorder- und Rückseite des Modells gemalt werden.

Radiergummi Das *Radiergummi* kann bereits gemaltes ausradieren, also transparent machen. Dieses Werkzeug ist wie alle anderen Malwerkzeuge mit jeder Pinselspitze kombinierbar. Umgesetzt ist es dadurch, dass schlicht eine komplett transparente Farbe gemalt wird.


color burn und color dodge *color burn* und *color dodge* sind Werkzeuge, die die vorhandene Farbe ändern. *color dodge* hellt die Farbe auf, *color burn* dunkelt sie ab. Abbildung 41 verdeutlicht die Wirkung. Diese Werkzeuge arbeiten analog zu den gleichnamigen Blendmodi für Ebenen.

Fülleimer Der *Fülleimer* setzt die Farbe der gesamten Ebene auf den Wert der Vordergrundfarbe.

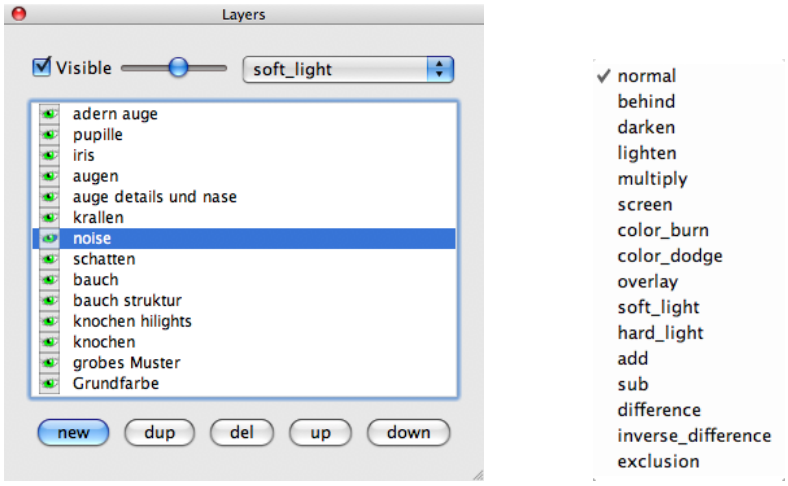
Farbwähler Mit der *Pipette* kann die Farbe unter dem Werkzeug aufgenommen werden. Dies geschieht indem an der gewählten Position die Farbe aus dem Frontbuffer ausgelesen wird. Hierdurch wird die beleuchtete Farbe, also die, die der Nutzer auch tatsächlich sieht, zurückgegeben. Möchte der Nutzer hingegen die an dieser Stelle gemalte Farbe aufnehmen, so muss er zuvor die Beleuchtung des Modells deaktivieren.

Abbildung 41: Links das Ausgangsbild, in der Mitte die Wirkung des *color burn* Werkzeuges, rechts die von *color dodge*.

4.4.3 Die Ebenenpalette

Die Ebenenpalette zeigt alle Ebenen der Textur an. Die blau markierte Ebene ist die aktive, in welche gemalt wird. In diesem Dialog kann die Transparenz in 100 Stufen geändert werden, zudem kann die Ebene mit einer Checkbox temporär ganz deaktiviert werden. Rechts oben kann der Blendmodus der Ebene gewählt werden, es stehen die üblichen, aus Bildbearbeitungsprogrammen bekannten, Modi bereit. Eine Beschreibung der Modi und deren Umsetzung als Fragment-Shader findet sich in [Fer04].

Die Buttons unten erlauben es eine neue, leere Ebene zu erstellen, die aktuelle Ebene zu dublizieren oder zu löschen. Um die Reihenfolge der Ebenen zu ändern kann die aktuelle Ebene mit den Buttons *up* und *down* um eine Ebene nach oben bzw. unten verschoben werden. Der Name der Ebene lässt sich über das Menu *Layer* ändern. Über dieses Menu lässt sich auch die ausgewählte Ebene dauerhaft in die darunter liegende hinein mi-



(a) Ebenenpalette **(b)** Blendmodi

Abbildung 42: Dialog zur Wahl der aktiven Ebene.

schen. So lässt sich die Textur etwa auf eine einzige Ebene reduzieren wenn die Bearbeitung abgeschlossen ist.

4.4.4 Die Pinselpalette

Die aktive Pinselspitze, das heißt die Form des Pinsels, lässt sich in der Pinselpalette wählen. Zudem kann über zwei Schieberegler die Größe sowie die Deckkraft des Pinsels eingestellt werden. Über die Checkboxen kann zusätzlich gewählt werden, dass diese Eigenschaften durch den Druck des Stiftes auf das Grafiktablett gesteuert werden soll. In dem Falle gibt der Wert des Schiebereglers den Wert bei Maximaldruck an.

Die Vorschau wird direkt aus den Bilddateien der Pinselspitzen erstellt, so dass ohne weitere Änderungen neue Pinsel hinzugefügt werden können. Dies geschieht, indem eine weitere .png Datei dem Pinsel Ordner hinzugefügt wird.

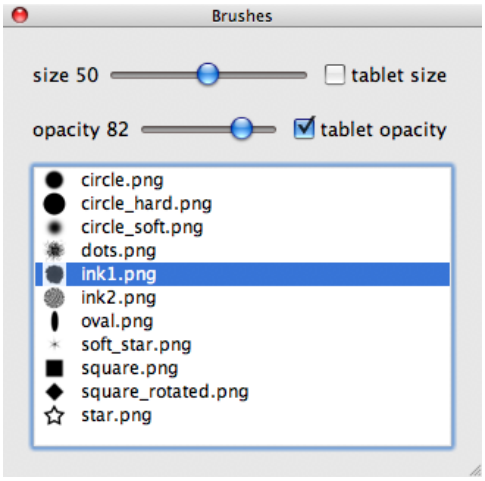


Abbildung 43: Dialog zur Auswahl von Pinseln.

4.4.5 Die 2D Ansicht

Da in Vorgesprächen mit potenziellen Nutzern oft gewünscht wurde, die Textur auch im 2D bearbeiten zu können, wurde eine entsprechende Funktion integriert. In diesem Modus kann die Textur verschoben und gedreht werden, zudem lässt sich wie im 3D ein- und auszoomen. Alle Werkzeuge funktionieren wie im 3D, da in diesem Modus das Modell einfach nur durch ein Viereck mit gleicher Textur ersetzt wurde. Lediglich die Kamera wurde dahingehend eingeschränkt, dass sie nur senkrecht auf die Textur blicken kann. So ergibt sich eine Ansicht wie in einer normalen Bildbearbeitung.

Neben dem Kamerawerkzeug wurde die Funktion der Buttons zur Umstellung des Rendermodus geändert: Das Rendern der Vertices ist ohne Funktion, die Einstellung zur Darstellung des Wireframe zeigt das Dreiecksgitter über der Textur. Dies erleichtert die Orientierung, da sonst nicht ersichtlich ist, welche Texturbereiche an welcher Stelle des Modells abgebildet werden. Analog hierzu zeigt die Nahtansicht nur die Umrisse der einzelnen Bereiche, so wird die zu bemalende Textur nicht durch das Wireframe selbst verdeckt. Eine Beleuchtung findet im 2D nicht statt.

Beim “mechanischen Killeroo“ des Nutzers 7 hat dieser die Augen in der 2D Ansicht bearbeitet. Die Augen stellen in der Textur-Map einzelne Bereiche dar, die Nähte laufen also entlang der Augenlieder. Hierdurch kann der Künstler im 2D beliebig “übermalen“ und hat es so einfacher. Zudem handelt es sich bei den Killeroooaugen um eine realtiv “flache“ Region, die geringen Verzerrungen in der Textur-Map können vernachlässigt werden.

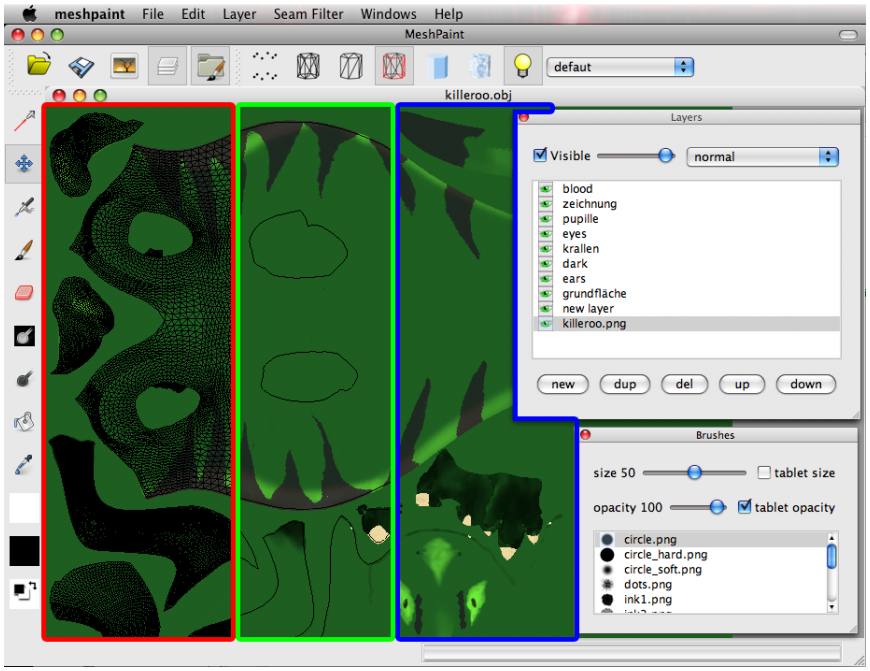


Abbildung 44: Der 2D Modus: Rot umrandet mit eingeblendetem Wireframe, grün umrandet mit der eingeblendeten Naht und ohne Hilfsmittel (blau).

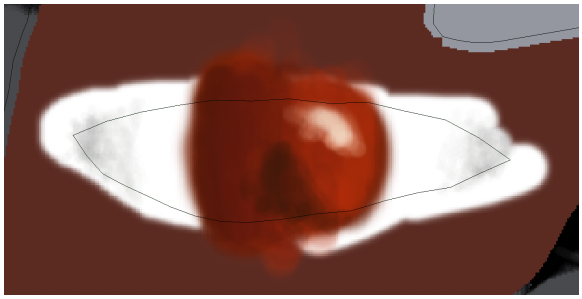


Abbildung 45: Details wie dieses Auge lassen sich gut im 2D bearbeiten, wenn die Naht als Auswahl fungiert.

Um diese Vorteile der direkten Textur Bearbeitung nutzen zu können, muss der Modellierer die Textur-Map so gestalten, dass voneinander getrennte Regionen der späteren Textur schon in der Map einzelne Regionen darstellen.

4.5 Octree Texturen

Zu jedem Modell, auch einem ohne Textur-Koordinaten, kann eine Octree Textur erstellt werden. Diese Textur besteht aus einer Wurzel und bis zu acht Kindknoten, die auch jeweils wieder bis zu acht Kindknoten und/oder Blätter haben können. In den Blättern werden die Farbwerte der Textur gespeichert.

Der Octree wird erstellt, indem um das Modell ein Würfel aufgespannt wird, der das gesamte Objekt umfasst. Dieser Würfel wird in acht gleich große Würfel unterteilt, in denen nach Polygonen des Objektes gesucht wird, die in diesem Würfel liegen, oder es schneiden. Ist dies nicht der Fall, der Würfel also leer, wird an dieser Stelle kein Kindknoten erstellt, ansonsten wird einer erstellt und dieser Knoten rekursiv weiter unterteilt.

Dies wird bis zu einem Abbruchkriterium durchgeführt, bei MeshPaint bis zu einer maximalen Rekursionstiefe. Dann werden anstelle weiterer Kindknoten Blätter erstellt.

Jedem Punkt der Modelloberfläche ist somit eindeutig ein Blatt des Octrees zugeordnet, der allein aufgrund der Koordinate des Punktes ermittelt werden kann, in dem der Baum traversiert wird.

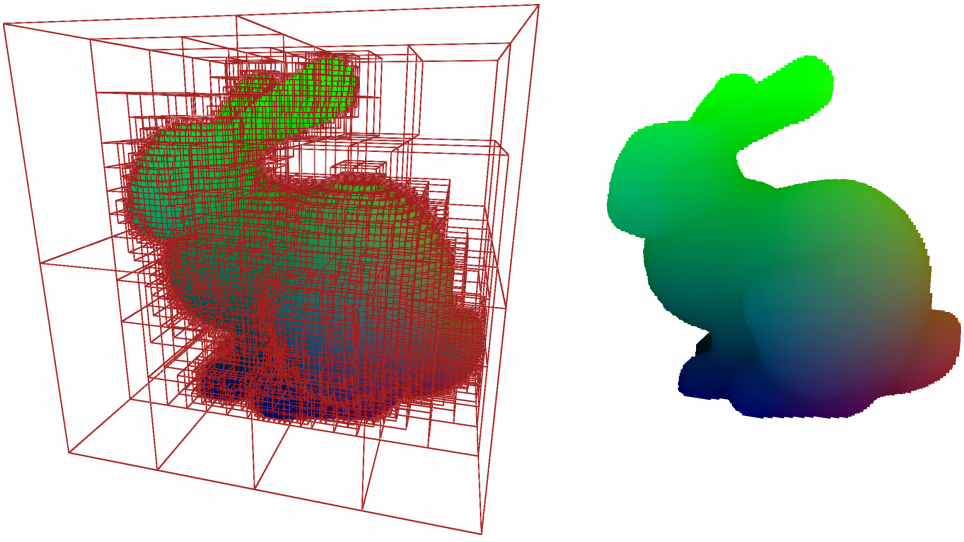


Abbildung 46: Links: Visualisierung des Octree, Rechts: Nur die Blätter.

Zunächst wurde zu dem Modell ein Octree mit festgelegter Rekursionstiefe erstellt. Die hierfür benötigten Schnitttests von Polygonen mit Qua-

dern finden sich bei Möller [AMH02] [AM08]. Dieser Baum wurde dann in Texturen überführt, um die Octree Textur auf der Grafikkarte darstellen zu können.

4.5.1 GPU Umsetzung

Für die Darstellung wurde ein Ansatz gewählt, der auf dem von Lefebvre et al. [Fer04] aufbaut. Hierbei dient eine Volumentextur dazu, den gesamten Octree aufzunehmen. Zunächst soll dieses Verfahren erklärt werden, bevor auf die Anpassungen eingegangen wird.

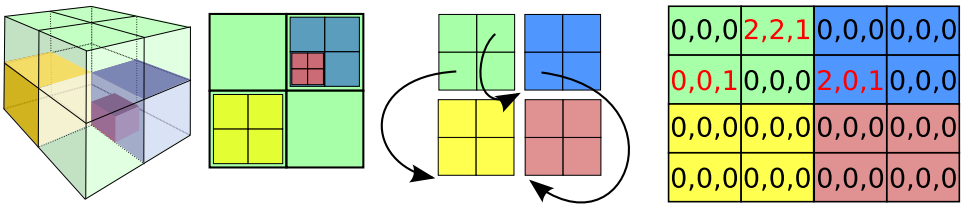


Abbildung 47: Links: Ein einfacher Octree; daneben ein ähnlicher Quadtree, der in eine zweidimensionalen Textur gespeichert werden kann.

Abbildung 47 zeigt links einen Octree, daneben einen Quadtree mit ähnlichem Aufbau. An diesem soll im 2D erklärt werden, wie ein Baum in einer Textur untergebracht werden kann, da auf der GPU nicht auf komplexe Datenstrukturen, welche zum Beispiel Pointer voraussetzen, zugegriffen werden kann. Der Quadtree ist eine gedachte verschachtelte Struktur, die auseinander genommen aus vier Elementen besteht mit je vier Feldern. An den Stellen, an denen ein Feld aus weiteren Elementen bestehen soll, wurden Pfeile auf das jeweilige Element eingefügt. Setzt man nun die vier Elemente zu einem zweidimensionalen Array zusammen, so kann man die Indirektionen durch die jeweiligen Koordinaten der unteren linken Ecke des Zieles ersetzen.

Der Koordinaten-Ursprung soll unten links sein. Das untere, linke, grüne Feld verweist nun auf den Ursprung, da dort das gelbe Feld beginnt. Die dritte Koordinate dient der Unterscheidung zwischen Indirektionen und leeren Feldern. Hier wurden die Indirektionen zur besseren Unterscheidung rot hervorgehoben.

Ist die Position des Wurzelknotens im Array, bzw. in der Textur bekannt, so kann anhand der Koordinaten der Baum traversiert werden.

Für einen Octree werden nicht vier, sondern je acht Felder benötigt. Diese werden bei Lefebvre in $2*2*2$ Texel großen Würfeln angeordnet, so dass zur Speicherung des Octrees eine Volumentextur verwendet werden muss. Entsprechend werden drei Koordinaten für die Indirektionen benötigt, in diesem Falle die RGB-Werte des Texels. Zwischen einer Indirektion und ei-

nem Blatt wird mithilfe des Alpha-Wertes unterschieden. In den Blättern können so direkt die Farbwerte gespeichert werden.

Textur-Ebenen Der Transparenz-Wert der Textur ist aber ein wichtiger Bestandteil der Textur-Ebene, ohne den nicht mit mehreren Ebenen gearbeitet werden kann. Aus diesem Grund wurde die GPU-Octree Repräsentation modifiziert: Weiterhin soll der Alpha-Wert zwischen einer Indirektion und einem Blatt unterscheiden, allerdings soll die Farbe des Blattes nicht in der Volumentextur des Octree gespeichert werden, sondern in einer weiteren Textur, die ausschließlich die Farbwerte speichert. Gibt also der Alpha-Wert des Octree an, dass ein Blatt erreicht wurde, so muss in der Farbtextur an einer Koordinate nachgeschlagen werden, welche aus den Rot- und Grün-Werten des Octree Texels ermittelt werden können. Die Farbe dient direkt als Koordinate innerhalb der Farbtextur.

Dieses Vorgehen hat noch einen zweiten entscheidenden Vorteil neben dem, dass für die Farbe ein Alpha-Wert zur Verfügung steht: Dadurch, dass in dieser Farbtextur ausschließlich Farbwerte gespeichert werden, lassen sich diese Textur-Ebenen mit den gleichen Techniken mischen wie übliche 2D Texturen. Diese Art der Textur-Ebenen für Octrees konnte in MeshPaint problemlos umgesetzt werden.

Malen Um in die Octree Texturen zu malen wurde folgender Ansatz umgesetzt: Bei der Erstellung der Volumentextur, mit der der Octree auf der GPU dargestellt werden soll, sowie der Farbtextur, wurde eine weitere Hilfstextur erstellt. Diese hat die gleichen Dimensionen wie schon die Farbtextur. An der Position, an der in der einen Textur die Farbe eines Voxels gespeichert wird, wird in der Hilfstextur die 3D Koordinate des Voxelmittelpunktes gespeichert.

Das Ziel ist es, in einem Renderpass, der die Depht-Map des Modells sowie diese Hilfstextur als Eingabe hat, im Fragment-Shader alle Voxel auf den Bildschirm zu projizieren und einen Sichtbarkeitstest durchzuführen. Das Ergebnis ist eine Textur, in der die projizierten Bildschirmkoordinaten der Octree Voxel an den Stellen gespeichert sind, an denen in der Farbtextur die Voxelfarben gespeichert sind.

Somit verhält sich diese Vorberechnung analog zur Vorberechnung der Texelzuordnungen beim projektiven Malen bei 2D Texturen. Es stellt also ein alternatives "Backend" zum bereits implementierten *airbrush* bzw. *brush* dar.

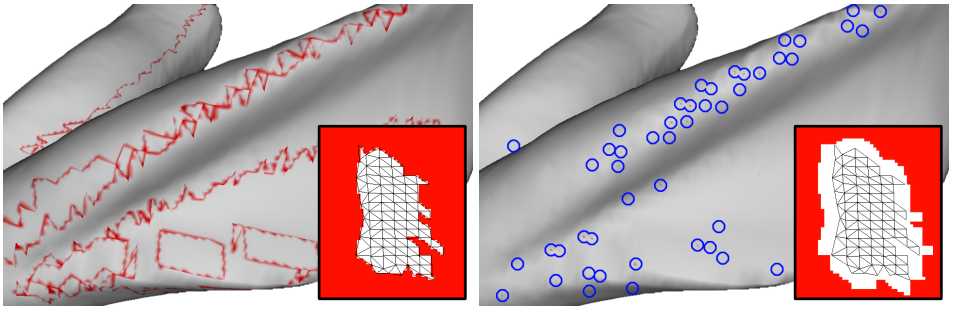
5 Ergebnisse

In diesem Kapitel sollen zunächst die Nahtfilter verglichen werden, danach wird auf den Nutzertest eingegangen. Zuletzt soll ein Blick auf die Performance der Anwendung geworfen werden.

5.1 Vergleich der Nahtfilter

Zum optischen Vergleich der Nahtfilter wurde ein rotes Modell mit vielen Nähten komplett weiß angemalt. Überall dort, wo noch rote Farbe durchschimmert, hat der Nahtfilter also nicht ausgereicht. Die Beispiele wurden mit dem Stanford Bunny und einer Textur von 2048^2 Texeln erstellt. Hier nehmen die einzelnen Polygone nur einen sehr kleinen Bereich der Textur-Map ein. Zum Vergleich folgen Bilder mit einem auf 5000 Polygone reduziertem Stanford Bunny.

Da die meisten verbliebenen Artefakte sehr klein sind und sich bei den verschiedenen Filtern im Wesentlichen in der Anzahl unterscheiden, wurden sie in den Abbildungen mit blauen Kreisen markiert.



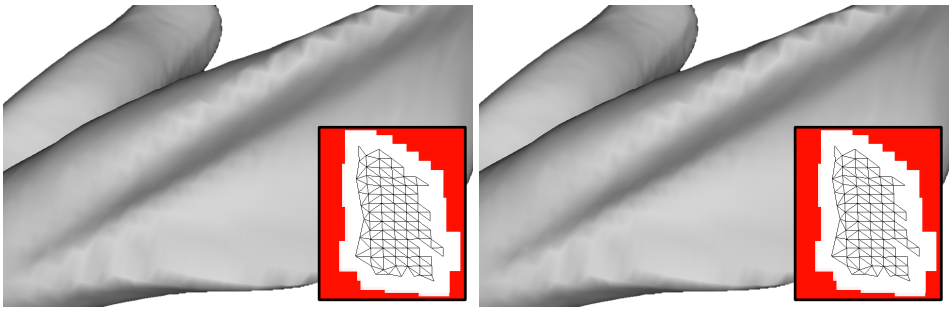
(a) Ohne Filter **(b)** Linienfilter

Abbildung 48: Links sind die vielen Nähte der Textur-Map in rot dargestellt. Beim Linienfilter ist noch eine Vielzahl an roten Punkten sichtbar.

In Abbildung 48a sind die vielen Nähte gut erkennbar. Unten rechts ist jeweils ein Ausschnitt der Textur-Map mit drauf geblendetem Drahtgitter zu sehen. Beim Linienfilter bleiben allein in diesem Ausschnitt vom Ohr über 40 kleine Artefakte übrig. Dieses Verhalten zeigt sich unabhängig von der Textur- oder Modellauflösung.

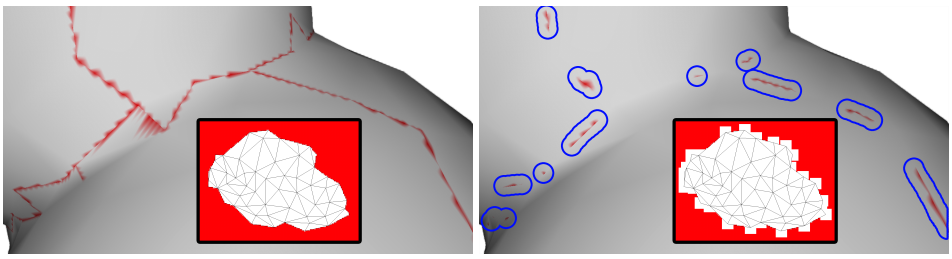
Die 69451 Polygone des Bunnys in Abbildung 49 sind auf der Textur-Map so klein, dass die vergrößerten Punkte des Punktfilters nahezu überall die Nähte abdecken. Der kombinierte Filter kommt wegen der Dominanz der Punkte zum gleichen optischen Ergebnis. Hier sind keine Artefakte mehr auszumachen.

Vergleicht man dies mit den Nähten des kleinen 5000 Polygone Bunny in Abbildung 50, so sieht man deutliche zusammenhängende Artefakte



(a) Punktfilter **(b)** Kombination aus Punkt und Linienfilter

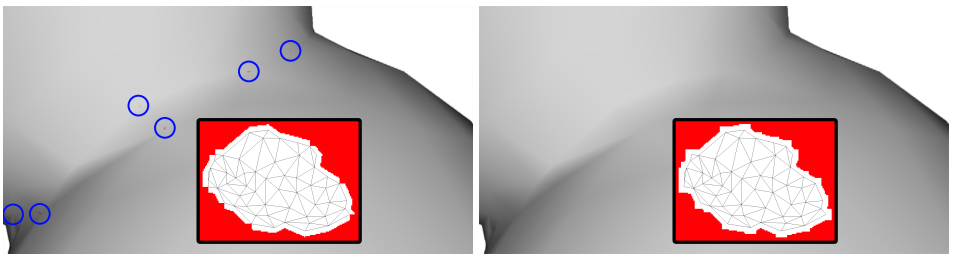
Abbildung 49: Bei sehr kleinen Polygonen genügt der Punktfilter, ansonsten muss er mit dem Linienfilter kombiniert werden.



(a) Ohne Filter **(b)** Punktfilter

Abbildung 50: Bei größeren Polygonen versagt der Punktfilter.

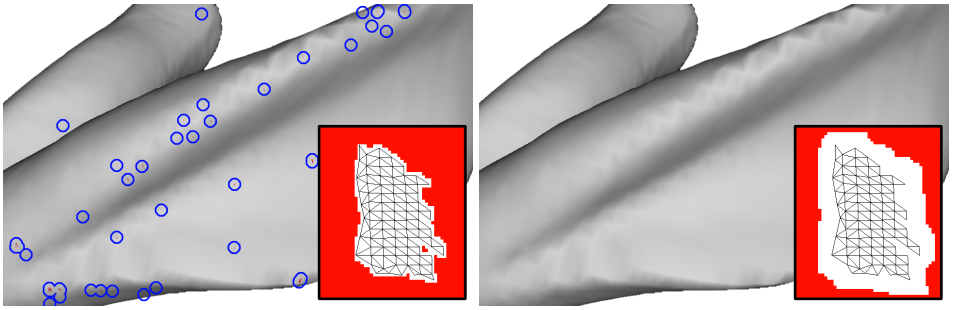
beim Punktfilter. Der Linienfilter erzeugt auch auf diesem Modell die gleichen kleinen Flecken, erst der kombinierte Filter verdeckt die Nähte vollständig (Abbildung 51).



(a) Linienfilter **(b)** Kombination aus Punkt und Linienfilter

Abbildung 51: Der Linienfilter hat unabhängig von der Polygongröße kleinere Artefakte.

Der Nearest Neighbour Filter ist von der Anzahl der Polygone unabhängig, da hier nur die Texel betrachtet werden. Wird nur in einer ein Texel Nachbarschaft gefiltert, so bleiben aufgrund sehr spitzer Polygone noch eine Reihe von Artefakten übrig, bei fünf Filterdurchgängen konnten keine Artefakte mehr entdeckt werden.

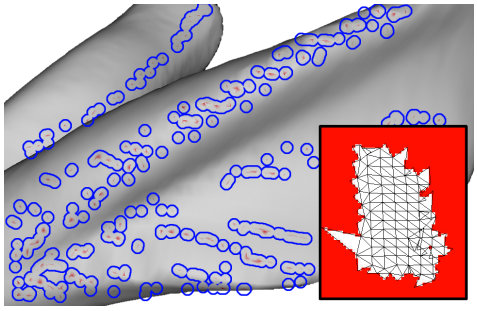


(a) 1 Texel Nearest Neighbour **(b)** 5 Texel Nearest Neighbour

Abbildung 52: Bei einem Texel Nearest Neighbour bleiben vereinzelt noch Artefakte an sehr dünnen Polygonen, diese sind meist kleiner als die beim Linienfilter. Bei fünf Texeln sind bereits keine Artefakte mehr sichtbar.

Das duplizieren der Polygone an den Nähten erwies sich überraschend als schlechteste Lösung, hier blieben sehr viele Artefakte erhalten. Dieser Filter war zudem am aufwändigsten zu implementieren und braucht bei großen Modellen eine lange Zeit zur Ermittlung der Nachbarpolygone.

Es erzeugen also die Kombination aus Linien- und Punktfilter gute Ergebnisse, sowie ein mehrfaches Nearest Neighbour filtern. Schon bei nur drei Durchläufen benötigt der Nearest Neighbour Filter über 30 Millise-



(a) Duplizierte Polygone

Abbildung 53: An den Kanten der duplizierten Polygone blieben erstaunlich viele Artefakte erhalten.

kunden bei einer 2048^2 großen Textur. Dies macht ihn zu langsam um ihn interaktiv beim Malen anzuwenden. Dies wird besonders beim *brush* deutlich, der bei dieser Texturgröße zum Malen eines Punktes nur 3,5 Millisekunden benötigt. Dennoch ist er auch bei sehr großen Texturen und mehr Durchläufen immer performant genug, um ihn am Ende der Bearbeitung anzuwenden.

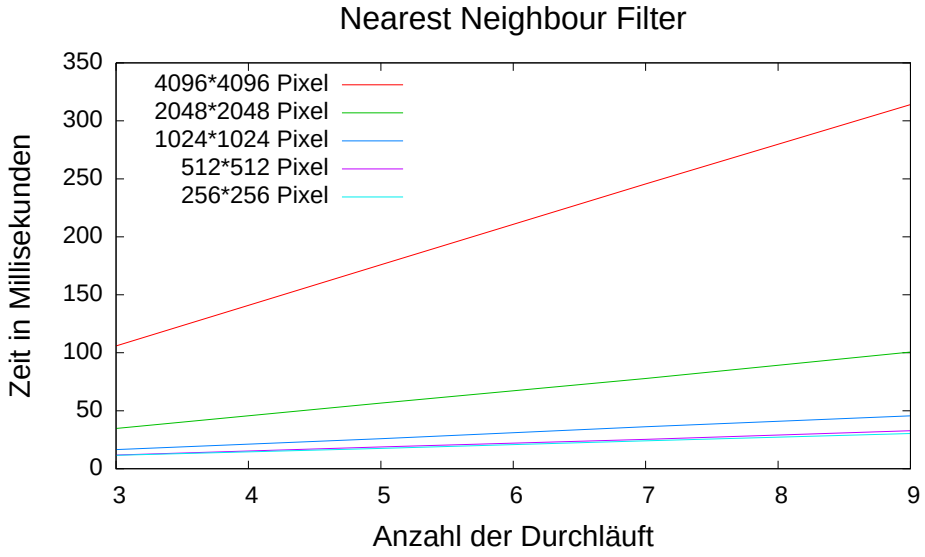


Abbildung 54: Der Zeitbedarf des Nearest Neigbour Nahtfilters ist nur von der Texturgröße abhängig.

Die Alternative ist der Kombinationsfilter, dessen Geschwindigkeit bei der Vorberechnung der Texelzuordnungen eine Rolle spielt. Mit ihm dauert die Vorberechnung bei Modellen unter 25.000 Polygonen 2 bis 4 mal so lange. Bei 277.000 Polygonen wird sie bereits 15 bis 33 mal langsamer

(abhängig von der Texturgröße). Bei Modellen die größer sind als eine halbe Million Polygone erreicht die Vorberechnung mit diesem Filter die ein-Sekunden Grenze und wird uninteressant.

Die Vorberechnungszeit vergrößert sich beim Punktfilter ähnlich rasant, wohingegen der Linienfilter bei 277.000 Polygonen nur 4 mal so lange benötigt als die Vorberechnung ohne Nahtfilter und selbst bei 1,1 Millionen Polygonen noch unter 300 Millisekunden liegt.

Daher eignet sich der Linienfilter am besten für eine interaktive Bearbeitung des Modells, da er sehr performant ist und nur wenige Artefakte übrig bleiben. Verbleiben noch Fehler, so lässt sich die Textur jederzeit manuell mit dem Nearest Neighbour filtern.

Die duplizierten Polygone schnitten optisch sehr schlecht ab, bei der Zeit zur Vorberechnung der Texelzuordnungen fallen sie gerade bei großen Modellen positiv auf. Bei 277.000 Polygonen war dieser Filter nur knapp halb so schnell wie gar kein Filter. Allerdings dauert die einmalige Suche nach den Nachbarpolygonen bei großen Modellen sehr lange.

Beim Stanford Bunny mit 69451 Polygonen dauert die Suche knappe 8 Sekunden auf einem Intel CoreDuo 2,13 Ghz Prozessor. Das 277.000 Polygone Modell brachte es bereits auf 14 Minuten, bei 1,1 Millionen Dreiecken dauert es 48 Minuten.

Detaillierte Benchmark-Ergebnisse finden sich im Anhang ab Seite 96.

5.2 Nutzertest

In einem Nutzertest sollten folgende Fragen beantwortet werden: Genügt das Bemalen im 3D Modus, oder benötigt man auf jeden Fall noch eine direkte Bearbeitung der 2D Texture-Map? Wenn dies der Fall ist, wo liegen die Grenzen des 3D Paintings? Diese Frage wurde zunächst in einem Fragebogen (siehe Abbildung 55) gestellt. Anschließend wurde die tatsächliche Nutzung der 2D Ansicht auf die Texture-Map im praktischen Teil des Testes gemessen.

Eine weitere Fragestellung lag in der Metapher des Malwerkzeuges: Soll der Pinsel die Geometrie des 3D Objektes berücksichtigen oder nicht? Abbildung 56 zeigt die konkrete Fragestellung: Die Ohren des Hasen liegen im Raum deutlich auseinander, verdecken sich aber in der aktuellen Kameraansicht. Wird der Nutzer nun erwarten, dass auch das hintere Ohr angemalt wird oder nur das vordere? Ist diese Erwartung abhängig von der Bezeichnung des Werkzeuges?

Wie in Kapitel 4.3.4 bereits beschrieben wurde, ist das in *Abbildung 4* des Nutzertestes gezeigte Verhalten wesentlich leichter zu implementieren. Wird dieses Verhalten von den meisten Nutzern auch erwartet, lohnt der Aufwand zur Implementierung und Optimierung des Geometrie berücksichtigenden Pinsels nicht.

Zusätzlich zu diesen zentralen Fragestellungen wurde nach dem Vorwissen in Bezug auf die Texturierung im Allgemeinen und die Texturierung mit 3D Paintern im Speziellen gefragt.

Im praktischen Teil sollten die Nutzer ein Modell nach eigenen Vorstellungen bemalen. Hierbei wurde darauf geachtet, dass das Modell eine "gute" Texture-Map hat. "Gut" in dem Sinne, dass eine Texturierung bei dieser Map auch allein anhand der 2D Abbildung erfolgen könnte. Nur so konnte getestet werden, ob die Benutzer auch auf die Möglichkeit der Bearbeitung in der 2D Ansicht zurückgreifen. Die Nutzung der einzelnen Werkzeuge, hier insbesondere der zwei Pinselarten, die Verwendung von Ebenen und die Häufigkeit und Länge der Nutzung der 2D Ansicht wurden automatisch gemessen. Zusätzlich wurde gemessen, wie viele Pinselstriche gemalt wurden, bevor das Modell rotiert wurde und wie lang diese Pinselstriche sind. Hierbei sollte ermittelt werden, wo sich Optimierungen lohnen.

5.2.1 Umfrage zur Texturierung

An der Umfrage nahmen 15 Personen teil, nicht alle von diesen fanden die Gelegenheit, auch am Test der Anwendung teilzunehmen. Ein Großteil hatte nur geringe Erfahrungen im Modellieren und keine im Texturieren, aber es waren auch ein paar "Experten" dabei (vergleiche Tabelle 1). Zwei Personen hatten zudem schon Erfahrungen mit 3D Paintern gemacht. Die übrigen hatten ihre Texturen bisher in Bildbearbeitungsprogrammen wie Photoshop oder GIMP erstellt.

Tabelle 1: Antworten des Nutzertests

Vorwissen der Nutzer					
Modellierung	1	8	3	1	2
Texturierung	8	3	2	1	1

Dass die Texturierung im 3D die Bearbeitung im 2D vollständig ersetzen kann glaubte nur ein Befragter, zwei waren der Ansicht, dass die Meiste Arbeit im 2D erfolgen müsse und nur wenig im 3D gearbeitet wird. Allerdings empfand niemand die Möglichkeit des 3D Paintings als überflüssig, die große Mehrheit sagte aus, dass der 2D Modus nur für wenige Arbeiten nötig ist.

Folgende Gründe wurden genannt, warum auf die direkte Bearbeitung der Textur-Map nicht verzichtet werden sollte:

- Unbeleuchtete Farben kann man im 2D besser beurteilen als auf dem Modell.
- Großflächige Strukturen (zum Beispiel eine Orangenschale) seien im 2D einfacher zu erstellen.

Umfrage zur dreidimensionalen Texturierung
Teil der Diplomarbeit von Jan Robert Menzel

Allgemeine Informationen: ☐ männlich ☐ weiblich

Wie häufig modellieren sie 3D Objekte? noch nie ☐ ☐ ☐ ☐ ☐ regelmäßig
Wie häufig texturieren sie 3D Objekte? noch nie ☐ ☐ ☐ ☐ ☐ regelmäßig

Welche Programme nutzen sie für die Texturierung?

Haben sie bereits Erfahrungen mit 3D-Paintern? ☐ Ja ☐ Nein

Wenn ja, mit welchen:

Im folgenden ist ihre subjektive Meinung gefragt. Ein richtig oder falsch gibt es bei diesen Fragen nicht.
Kann die Texturierung in 3D die Bearbeitung der 2D Texture-Map ersetzen?

- ☐ Ja, eine Bearbeitung in 2D wird überflüssig.
- ☐ Nicht vollständig, in manchen Fällen ist eine Bearbeitung der 2D Map nötig.
- ☐ Nein, der Hauptteil der Arbeit wird in 2D nötig sein.
- ☐ Nein, 3D-Painting ist völlig überflüssig.

Falls eine der letzten drei Antworten gewählt wurde:
Wann/Warum ist eine Bearbeitung in 2D nötig?

Sie möchten einen Hasen (Abb. 1) texturieren. Dabei malen sie mit dem ***Pinsel Werkzeug*** über den in Abbildung 2 markierten Bereich. Welches Programmverhalten erwarten sie:

- ☐ Nur das vordere Ohr wird bemalt (Abb. 3)
- ☐ Das vordere und das hintere Ohr werden bemalt (Abb. 4)
- ☐ Ein ganz anderes Verhalten, nämlich: _____

Dieses mal wollen sie das ***Airbrush Werkzeug*** benutzen, um über den in Abbildung 2 markierten Bereich zu malen. Welches Programmverhalten erwarten sie:

- ☐ Nur das vordere Ohr wird bemalt (Abb. 3)
- ☐ Das vordere und das hintere Ohr werden bemalt (Abb. 4)
- ☐ Ein ganz anderes Verhalten, nämlich: _____

Abbildung 55: Seite 1 der Nutzerumfrage

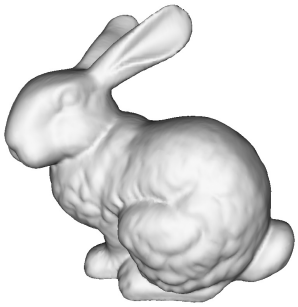


Abbildung 1: Der Hase

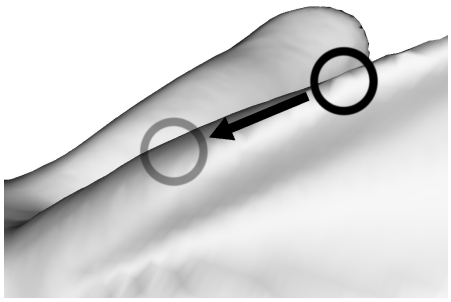


Abbildung 2: Der Schwarze Kreis symbolisiert die Werkzeugspitze, von rechts oben nach links unten soll gemalt werden.

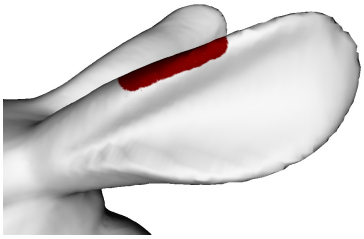


Abbildung 3: Nur das vordere Ohr wird bemalt.

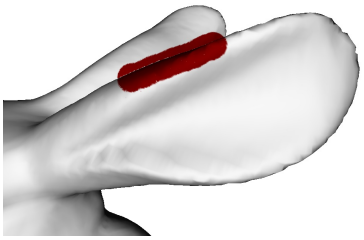


Abbildung 4: Beide Ohren werden entsprechend der Pinselgröße bemalt.

Abbildung 56: Seite 2 der Nutzerumfrage

- Es ist einfacher genau entlang einer Polygongrenze zu malen, wenn diese eine geometrische Bedeutung hat. Besonders dann, wenn die Naht der Textur bewusst hier entlang gelegt wurde. Ein Beispiel sind die Augen des Killeroo (Abbildung 45).
- Da man die gesamte Textur im Blick hat geht die Texturierung schneller, da die Rotation und Translation der Kamera entfällt.
- 2D eignet sich besser für einfache, glatte Strukturen, etwa eine Backsteinmauer.
- Platzierung von Fotos wird im 2D als einfacher eingeschätzt.
- Details können im 2D zum Teil leichter zu erstellen sein.

Die Bearbeitung von Details, die Platzierung von Fotos und die Texturierung einfacher Modelle wurden mehrfach genannt.

Diese Einschätzung der Notwendigkeit der direkten Bearbeitung der Textur-Map änderte sich nach dem praktischen Teil des Testes nicht und war zudem von dem Vorwissen der Nutzer unabhängig.

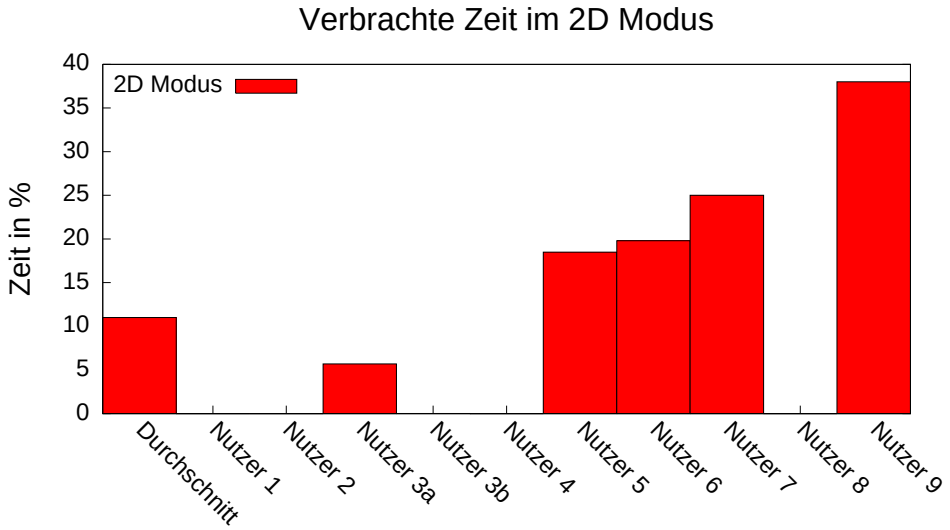


Abbildung 57: Die Hälfte der Nutzer verzichtete ganz auf den 2D Modus, die andere Hälfte nutzte ihn zum Teil sehr intensiv.

Die Grafik 57 zeigt an, wie viel Zeit die Nutzer in der 2D Ansicht verbracht haben. Vier Probanden versuchten fast komplett im 3D zu malen, weitere vier verbrachten zwischen $18,5\%$ und 38% ihrer Bearbeitungszeit im 2D. Testkandidat 3 nutzte die 2D Ansicht nur beim Killeroo, der Stanford Bunny wurde komplett im 3D texturiert. Im Durchschnitt wurde in

rund 10% der Zeit auf die direkte Bearbeitung der Textur-Map zurückgegriffen.

Somit kann das 3D Painting die "klassische" Texturierung nicht vollständig ersetzen, ist aber mehr als nur eine Ergänzung. Im Gegenteil stellt die Bearbeitung der Textur-Map eine wichtige Ergänzung zum 3D Painting dar, auf die allerdings auch nicht verzichtet werden sollte.

Werkzeugmetapher Die zweite Seite der Umfrage (Abbildung 55) zeigt ein Modell mit räumlich weit auseinander liegenden Elementen, den Ohren. Malt der Künstler nun über das vordere Ohr, wobei der Radius des Pinsels auch das hintere Ohr überdeckt, so stellt sich die Frage, ob auch das hintere Ohr bemalt werden soll. Erwartet der Nutzer also, dass die Geometrie des Objektes berücksichtigt wird, oder nicht?

Dies sollten die letzten zwei Fragen beantworten. Beim Pinselwerkzeug waren 13 Personen der Meinung, dass nur das vordere Ohr betroffen sein müsste, lediglich zwei Personen gingen davon aus, dass beide Ohren bemalt werden. Ein Kandidat war zusätzlich der Auffassung, dass nicht nur der sichtbare Teil des vorderen Ohres bemalt werden müsse, sondern auch die Rückseite des vorderen Ohres. Es solle also zwar nur das vordere Ohr betroffen sein, allerdings kein Verdeckungstest stattfinden.

Nachdem sich diese doch recht eindeutige Meinung im Verlauf der Befragungen abgezeichnet hatte, wurde der *brush* implementiert, um dieses Verhalten umsetzen zu können.

Wird das verwendete Werkzeug hingegen "Airbrush" genannt, so ändern sich die Ergebnisse: Nur drei Personen waren hier der Meinung, der Airbrush bemalt nur das vordere Ohr. Diese Kandidaten erwarteten dieses Verhalten im Übrigen bei beiden Werkzeugen. Die Mehrheit erwartete also, dass auf die Geometrie keine Rücksicht genommen wird. Fünf Kandidaten gaben zusätzlich zur Einschätzung, dass beide Ohren bemalt werden an, dass sie zudem eine Abschwächung der Farbe auf dem hinteren Ohr bzw. eine Körnung der Farbe erwarten, wie dies durch einen realen Airbrush geschieht.

Aufgrund dieser Ergebnisse wurde das vorher *brush* genannte Werkzeug in *airbrush* umbenannt. Die bei einem Drittel der Nutzer erwartete Abschwächung bei weiter entfernten Objekten wurde nicht implementiert. Daher passt diese Metapher zwar wesentlich besser zu den Erwartungen, ist aber noch nicht die Ideale.

5.2.2 Test der Anwendung MeshPaint

Testperson 1 wollte aus dem Stanford Bunny einen Leoparden Bunny machen. Der Test sollte in zwei Sitzungen durchgeführt werden, leider konnte der Tester die Textur krankheitsbedingt nicht zu Ende stellen.

Abbildung 58: Nutzer 1: Modelliererfahrung: 2/5, Texturiererfahrung: 2/5. Gesamtdauer: 95 Minuten. Die Textur besteht aus 2 Ebenen.

Den Pinguin Tux nahm sich der Testkandidat 2 als Vorbild zur Texturierung des Killeroo in Abbildung 59. Dieser Anwender rotierte sehr häufig, im Durchschnitt wurde alle $2,8$ Pinselstriche die Kameraposition geändert. Allerdings malte er auch sehr große Flächen ohne abzusetzen aus.

Testperson 3 hat zwei Modelle bemalt, zunächst das Killeroo, später den Bunny. Von ihr wurden bessere Werkzeuge zur nachträglichen Editierung der Farbe gewünscht. Dies gab den Anlass die Werkzeuge *color burn* und *color dodge* zu implementieren. Die Übersicht der verwendeten Werkzeuge (Abbildung 68) zeigt, dass diese Testperson beim zweiten Modell hiervon starken Gebrauch gemacht hat.

Es fällt in dieser Übersicht zusätzlich auf, dass hier erstmals das Werkzeug *brush* Verwendung findet. Dies liegt aber schlicht daran, dass erst zu diesem Zeitpunkt auch aufgrund der Umfrageergebnisse der *brush* implementiert wurde.

Testperson 5 hat sich bei der Texturierung des Nasenmannes auf das Gesicht und Details wie die Fingernägel konzentriert. Die Kleidung wurde bewusst ausgespart, da er diese durch weitere Geometrie dem Modell hinzufügen wollte.

Hemd und Hose verfügen über einige Details beim Nasenmann vom Nutzer 6. Hier wurde viel mit dem Laser Modus gearbeitet um die Symmetrie der Kleidung zu erzeugen. Zudem hat dieser Testkandidat oft vom 2D Modus gebrauch gemacht.

Abbildung 59: Nutzer 2: Modelliererfahrung: 2/5, Texturiererfahrung: 1/5. Gesamtdauer: 45 Minuten. Die Textur besteht aus 6 Ebenen.

Abbildung 60: Nutzer 3 (3a in den Diagrammen): Modelliererfahrung: 2/5, Texturiererfahrung: 1/5. Gesamtdauer: 94 Minuten. Die Textur besteht aus 10 Ebenen.

Abbildung 61: Nutzer 3 (3b in den Diagrammen): Modelliererfahrung: 2/5, Texturiererfahrung: 1/5. Gesamtdauer: 67 Minuten. Die Textur besteht aus 8 Ebenen.

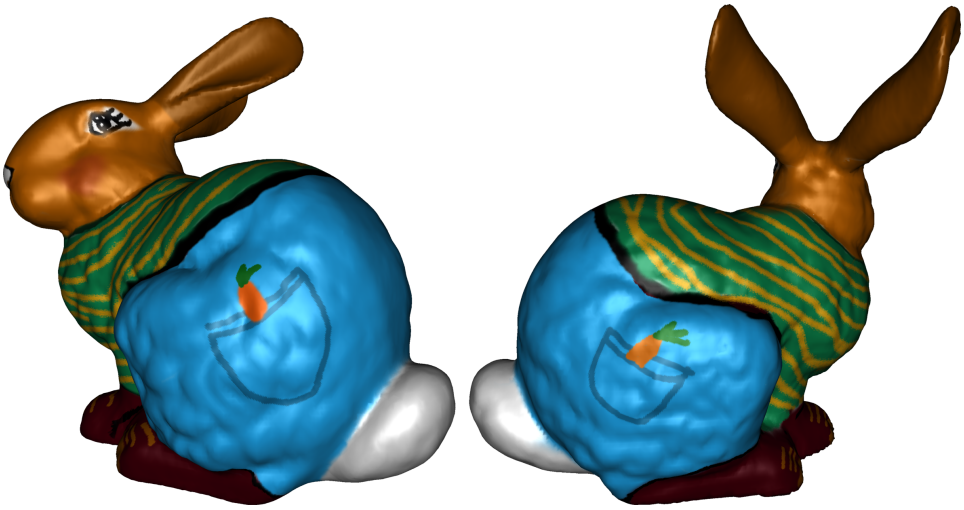


Abbildung 62: Nutzer 4: Modelliererfahrung: 3/5, Texturiererfahrung: 3/5. Gesamtdauer: 52 Minuten. Die Textur besteht aus 5 Ebenen.

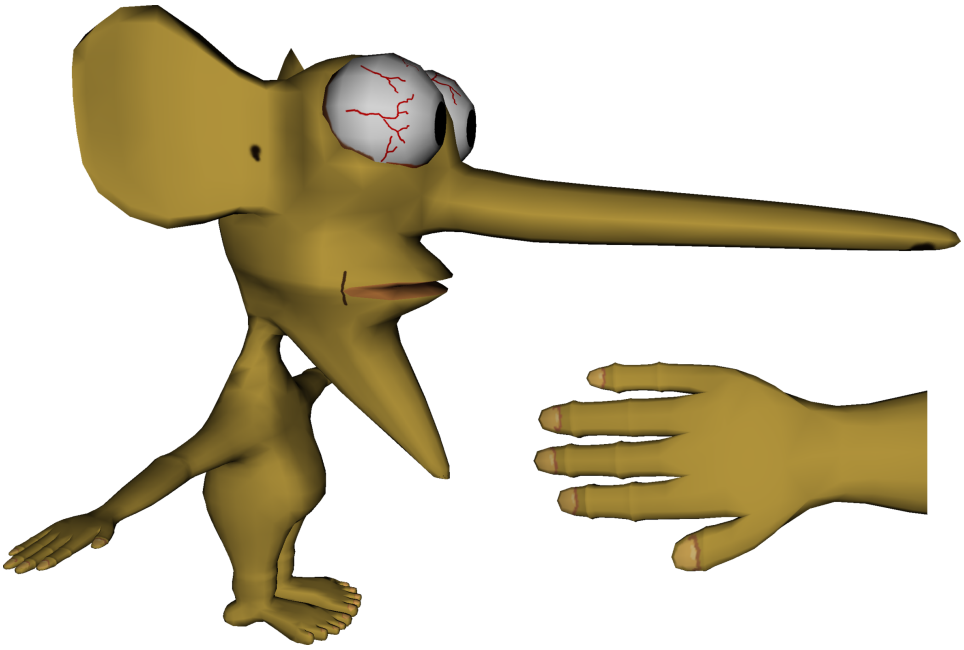


Abbildung 63: Nutzer 5: Modelliererfahrung: 3/5, Texturiererfahrung: 1/5. Gesamtdauer: 51 Minuten. Die Textur besteht aus 6 Ebenen.

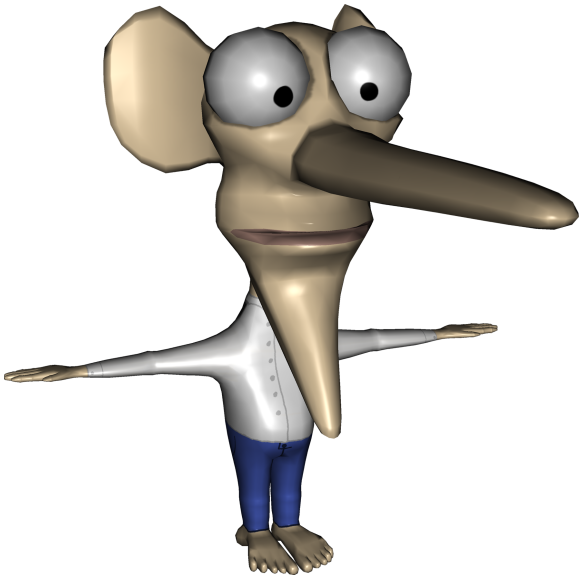


Abbildung 64: Nutzer 6: Modelliererfahrung: 5/5, Texturiererfahrung: 4/5. Gesamtdauer: 36 Minuten. Die Textur besteht aus 5 Ebenen.

Das mechanische Killeroo vom Tester 7 wurde fast vollständig in orthographischer Projektion bemalt. Viele Elemente wurden mit dem Laser Modus gleichzeitig auf beide Seiten des Modells gemalt, zum Beispiel die Gelenke an den Beinen und die Schrauben am Schwanz. Die Augen entstanden im 2D Modus.

Gerade die Gelenke und Schrauben sind sehr genau ausgearbeitet worden, wodurch sich die hohe Anzahl von Durchschnittlich 44 Strichen zwischen zwei Rotationen erklärt.

Abbildung 65: Nutzer 7: Modelliererfahrung: 2/5, Texturiererfahrung: 1/5. Gesamtdauer: 69 Minuten. Die Textur besteht aus 7 Ebenen.

Insgesamt ist aufgefallen, dass besonders die symmetrischen Modelle Killeroo und Nasenmann die Tester dazu verleitet haben, mit dem Laser Modus des *airbrush* zu arbeiten. Da nur ein Undo Schritt zur Verfügung stand, mussten Fehler wegradiert werden, daher liegt dieses Werkzeug noch knapp vor dem *brush* auf dem zweiten Platz.

Von den Ebenen und Blendmodi wurde ausgiebig Gebrauch gemacht, bis zu 10 Ebenen wurden pro Textur genutzt.

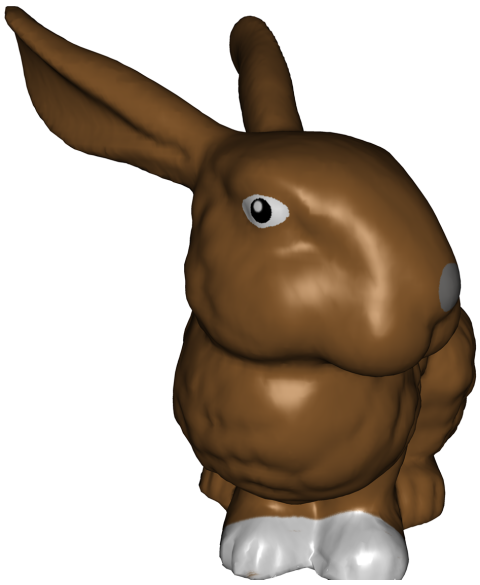


Abbildung 66: Nutzer 8: Modelliererfahrung: 2/5, Texturiererfahrung: 1/5. Gesamtdauer: 26 Minuten. Die Textur besteht aus 3 Ebenen.

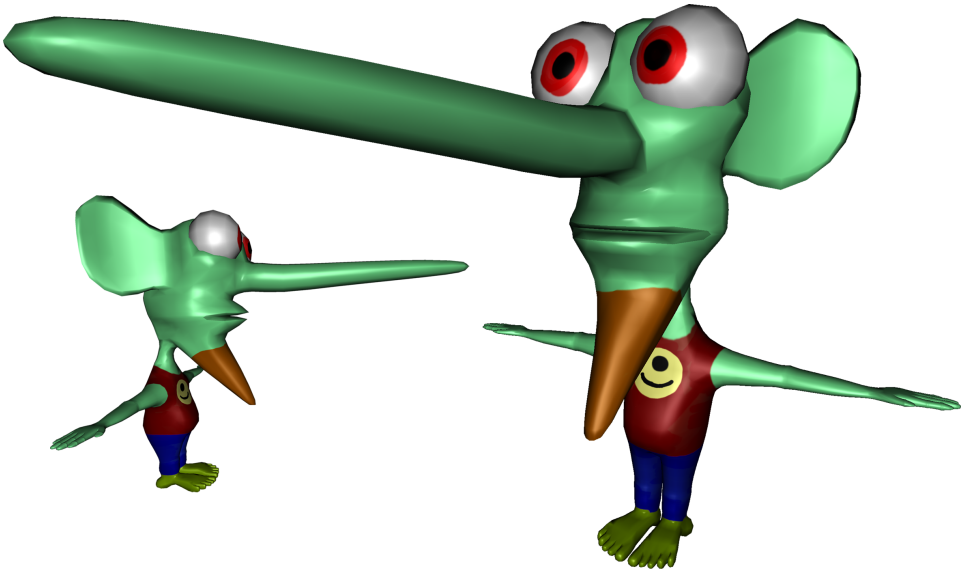


Abbildung 67: Nutzer 9: Modelliererfahrung: 2/5, Texturiererfahrung: 1/5. Gesamtdauer: 75 Minuten. Die Textur besteht aus 5 Ebenen.

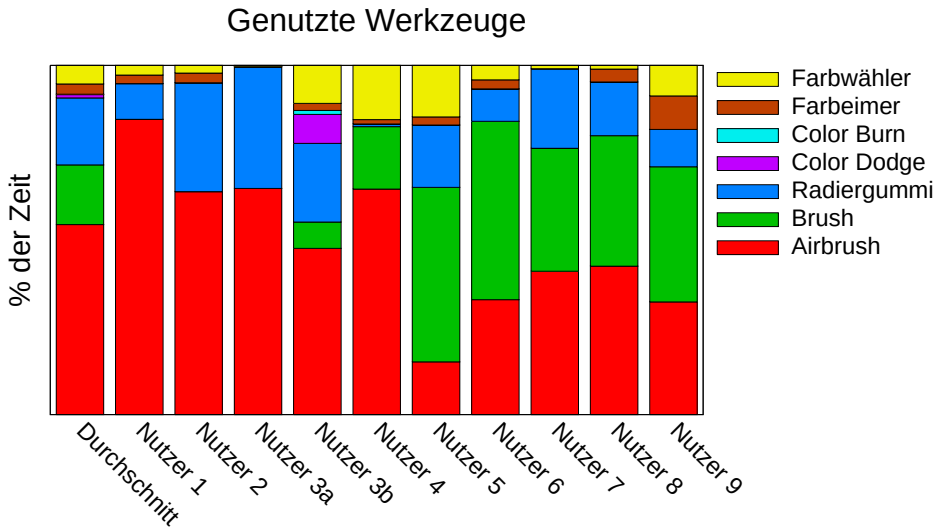


Abbildung 68: Nutzung der verschiedenen Werkzeuge.

5.3 Anforderungen an die Textur-Map

Eine manuell erstellte Textur-Map besteht aus wenigen, großen Bereichen zusammenhängender Polygone. Wird die Textur-Map hingegen automatisch erstellt, kann es vorkommen, dass mehrere kleine Bereiche entstehen, bis hin zu einzelnen Polygonen.

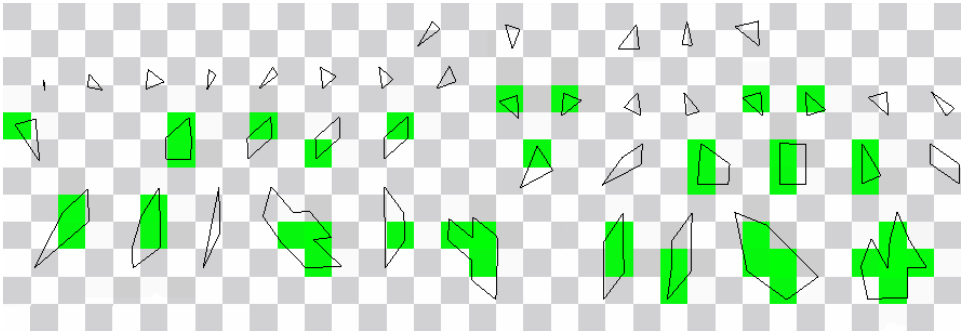


Abbildung 69: Bei zu kleinen Polygonen versagt das projektive Malen: Das Modell wurde ohne Nahtfilter komplett grün angemalt.

Abbildung 69 zeigt einzelne Polygone und kleine Bereiche in der von Autodesk Maya [Inc08a] automatisch erstellten Textur-Map des Stanford Bunny. In diesem Fall wurde eine sehr niedrig aufgelöste Textur gewählt (256^2). Wenn Dreiecke kleiner als die Texel sind, kann es sein, dass an dieser Stelle keine Farbe erstellt wird, da diese Dreiecke nicht gerendert werden. Zwar vergrößern Nahtfilter den Bereich, in dem Farbe erzeugt wird, doch

es gibt keine Garantie, dass nicht doch ein sehr kleines Polygon "durchschlüpft".

Das Modell darf also keine, in Relation zur Texturauflösung, zu kleinen Bereiche in der Texur-Map enthalten. Bei entsprechender Auflösung gab es auch bei diesen Textur-Maps keine Probleme.

5.4 Betrachtung der Performance

Alle Benchmarks wurden auf einem Intel CoreDuo Prozessor mit 2,13 GHz und einer nVidia GeForce 8800 Grafikkarte mit 640 MB Grafikkartenspeicher unter Windows XP durchgeführt. Ausführliche Messergebnisse finden sich im Anhang ab Seite 96.

5.4.1 Mal-Algorithmen

In Hinblick auf die Geschwindigkeit unterscheiden sich die zwei Pinsel *airbrush* und *brush* in zwei Punkten. Zum ersten nutzt der *airbrush* eine Puffer-Textur, in den die Striche vor der Projektion gemalt werden. Diese Textur muss nach jeder Projektion "geleert" werden. Dies entfällt beim *brush*, weshalb eine *einzelne* Anwendung dieses Werkzeugs schneller ist. Dafür muss die Projektion wesentlich seltener durchgeführt werden, nämlich nur einmal pro gemalter Linie. Wie oft der *brush* pro Linie aufgerufen werden muss, sieht man in der Abbildung 70. Im Durchschnitt werden 105 Koordinaten der Maus, bzw. des Tabletts an die Anwendung übergeben, bevor der Benutzer die Linie beendet hat. Ebenso oft müsste der *brush* aufgerufen werden.

Der zweite Punkt, in dem sich die Werkzeuge unterscheiden, ist eigentlich eine Gemeinsamkeit: Nach jedem Aufruf muss die Vorschautextur aktualisiert werden. Dies passiert aber wie eben erläutert beim *brush* sehr viel häufiger. Zudem fällt eine kurze Verzögerung aufgrund der Berechnung nach einem gemalten Strich kaum auf, während des Malens allerdings schon.

Daher wird die Geschwindigkeit vom *brush* vor allem durch die Erstellung der Vorschautextur limitiert. Der Aufruf vom *brush* selbst dauert je nach Texturgröße zwischen 2 und 3,5 Millisekunden (4096^2 Texel große Texturen fallen mit 10 Millisekunden aus dem Rahmen, siehe unten). Das Projizieren der Puffer-Textur dauert hingegen bis zu 78 Millisekunden bei 2048^2 Texel Texturen. Dieser Wert schließt das Löschen der Linie aus der Puffer-Textur mit ein. Schon ohne die Errechnung einer neuen Vorschautextur ist der *airbrush* bei durchschnittlicher Linienlänge doppelt so schnell.

Beide Werkzeuge nutzen die vorberechnete Texelzuordnung, die nur einmal nach einer Änderung der Kamera vorgenommen werden muss. Im Durchschnitt wurden 11,9 einzelne Striche zwischen zwei Rotationen gemalt (Abbildung 71). Wenn man den Testkandidaten 7 auslässt, der mit 44

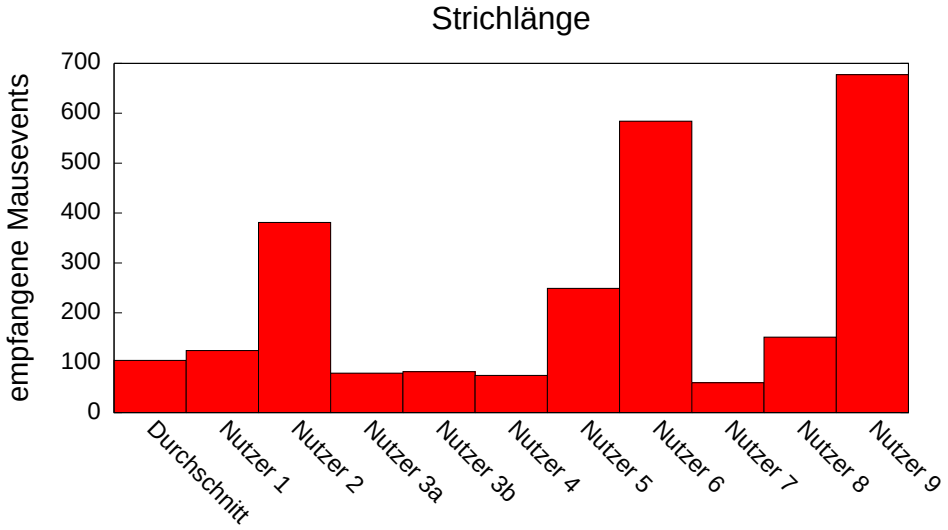


Abbildung 70: Die Länge eines Striches.

Strichen zwischen zwei Kamerabewegungen aus dem Rahmen fällt, so erhält man 7,4 Striche als Durchschnittswert. Diese Vorberechnung lohnt sich somit, da die Zuordnung sonst mit jeder Werkzeuganwendung erneut berechnet werden müsste, beim *airbrush* also mehr als 6 mal häufiger, beim *brush* sogar mehr als 600 mal häufiger.

5.4.2 Ebenen

Zur Darstellung des Modells wird eine Vorschautextur erstellt, damit Modelle mit beliebig vielen Textur-Ebenen dargestellt werden können. Dies geschieht in einem oder mehreren Offscreen Renderpasses, damit die Texturen nicht zwischen der Grafikkarte und dem Hauptspeicher hin und her kopiert werden müssen.

Es werden zunächst die vier untersten, sichtbaren Ebenen zu einer zusammengefasst. Diese wird dann mit den nächst höheren 3 zusammengefasst. Dieses Vorgehen spiegelt sich in den Ausführungszeiten in Abbildung 72 wieder: Bei den Texturgrößen bis 1024^2 erkennt man regelmäßig ein Ansteigen der benötigten Zeit, zunächst bei vier Ebenen, danach alle drei Ebenen.

Die Anzahl der Texturen, auf die ein Fragment-Shader gleichzeitig zugreifen kann, wird durch die Grafikkarte limitiert. Zum Zeitpunkt dieser Arbeit können aktuelle Grafikkarten auf vier bzw. acht Texturen zugreifen. Daher wurde entschieden, in jedem Durchlauf maximal vier Texturen zu mischen.

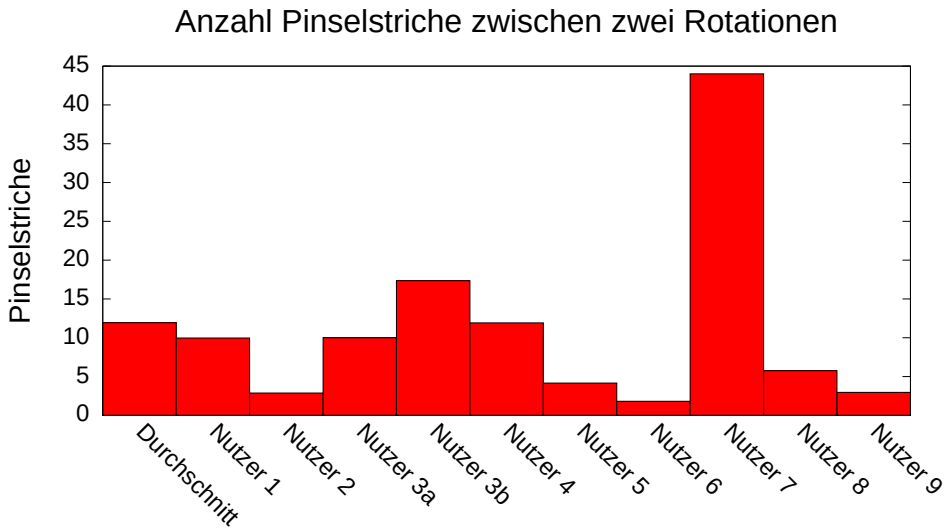


Abbildung 71: Die Anzahl der gemalten Striche zwischen zwei Rotationen.

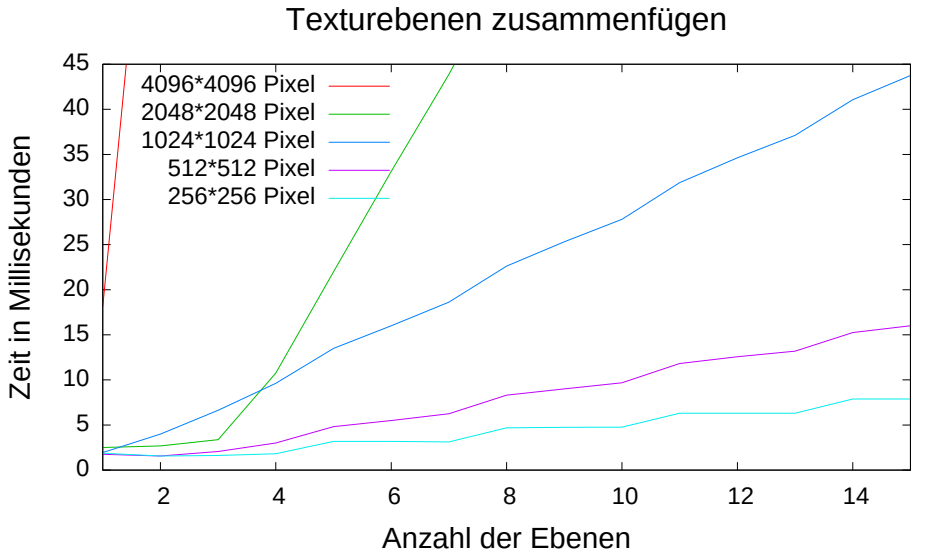


Abbildung 72: Zeitbedarf zur Erstellung einer Vorschautextur aus mehreren Textur-Ebenen. Die Werte steigen weiter linear an (vergleiche die vollständigen Werte in Abbildung 85).

5.4.3 Einfluss der Modell- und Texturgröße

Die Anzahl der Polygone des Modells macht sich an zwei Stellen bemerkbar: Zunächst einmal in der Vorschau, da das Modell natürlich erneut gerendert werden muss. Beim Malen muss nach jedem empfangenen Maus-Event, also nach jedem neuen Teilstück einer Linie, das Modell zur Voransicht erneut dargestellt werden. An dieser Stelle limitiert in erster Linie die Grafikkarte durch die maximale Menge an darstellbaren Polygonen pro Sekunde.

Zudem verlangsamt eine hohe Polygonanzahl die Errechnung der Texelzuordnungen von der Textur-Map auf Bildschirmkoordinaten. Diese Berechnung ist allerdings nur noch nach einer Rotation von Nöten, das Malen wird nicht behindert. Daher können auch sehr hoch aufgelöste Modelle bearbeitet werden.

Die Texturgröße hingegen macht sich deutlich stärker bemerkbar. Dies liegt im Wesentlichen an der Berechnung der Vorschautextur aus den Textur-Ebenen. Eine Vorschau muss beim *airbrush* nach jeder Linie neu errechnet werden, beim *brush* nach jedem Maus-Event, da hier öfters die Textur geändert wird. Im Nutzertest waren dies im Durchschnitt 105 Einzel-Events pro Linie.

Bei sehr großen Texturen von 4096^2 dauert die Vorschaugenerierung aus vier Ebenen schon rund 175 Millisekunden, in der gleichen Zeit lassen sich 18 Ebenen der Größe 2048^2 oder über 50 Ebenen der Größe 1024^2 zusammenfügen. Auch die Anwendung des *brush* ist bei dieser Texturgröße wesentlich langsamer als bei Kleineren: Bis zur Texturgröße 1024^2 benötigt dieser rund 2 Millisekunden, bei 2048^2 rund 3,5, bei 4096^2 schon 10 Millisekunden. Zusammen mit der Vorschaugenerierung der Textur, die bei diesem Werkzeug nach jeder Anwendung erneut durchgeführt werden muss, wird das Malen in dieser Texturgröße spürbar "träge".

Alle Nutzertests wurden mit einer Textur der Größe 2048^2 durchgeführt, dies ging auch bei mehr als 10 Ebenen noch problemlos.

5.5 Octree Texturen

Zwar konnte die Darstellung der Octree-Texturen auf der GPU implementiert werden, doch hat sich dies als weniger robust herausgestellt, als erwartet. Die Traversierung des Baumes erfordert, dass mehrfach hintereinander die Volumentextur ausgelesen wird und die gelesenen Werte mit der errechneten relativen Position in diesem Octree-Knoten zu einer neuen Koordinate für einen Textur-Zugriff verrechnet werden. Hierbei war zu beobachten, dass mit steigender Rekursionstiefe häufiger Rechenungenauigkeiten auftraten, die dazu führten, dass für einzelne Bereiche der Modelloberfläche kein zugehöriger Voxel im Octree gefunden werden konnte und somit keine korrekte Farbe ermittelt werden konnte.

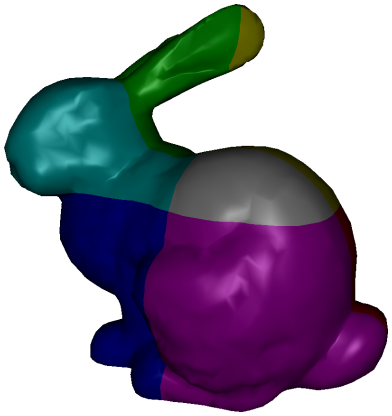


Abbildung 73: Bunny mit einfacher Octree Textur und Phong Beleuchtung

Diese Fehler waren nicht immer auf anderen Grafikkarten gleich reproduzierbar. Abbildung 74 zeigt die gleiche Octree-Textur links auf einer ATI X1900 Grafikkarte, rechts auf einer nVidia GeForce 8800. Die hellgrünen Streifen stellen Bereiche der Modelloberfläche dar, an denen kein zugehöriger Farbwert ermittelt werden konnte.

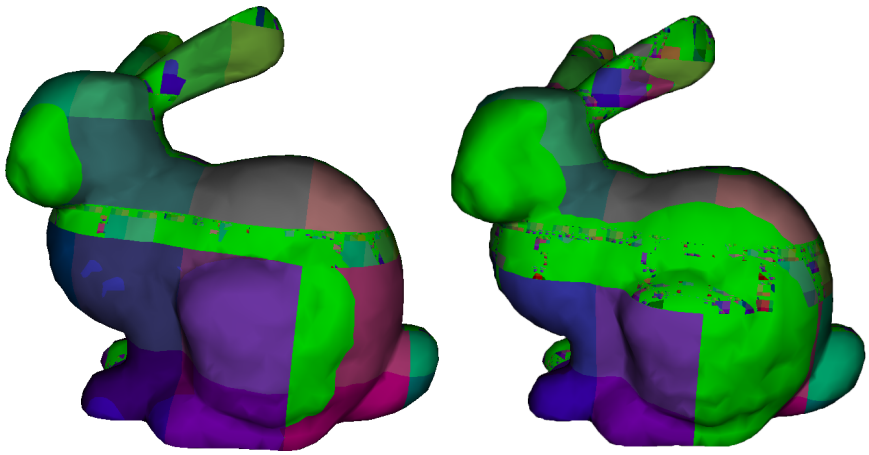
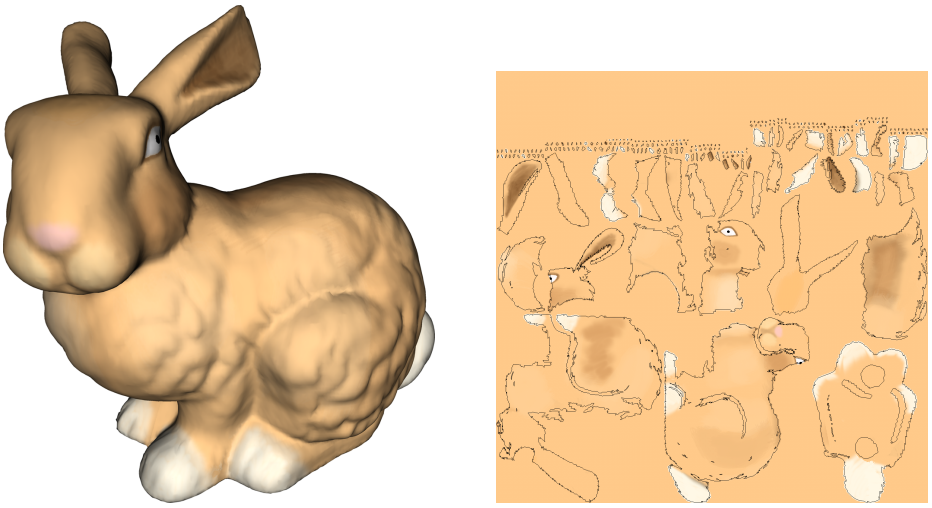


Abbildung 74: Unterschiedliche Traversierung des Octree auf verschiedenen Grafikkarten.

5.6 Beispiele

Die folgenden Texturen sind wie schon die Texturen der Nutzertests komplett in MeshPaint entstanden. Beim ersten Stanford Bunny wurde eine Textur-Map ohne Nutzerinteraktion von Autodesk Maya erstellt. Die größeren Bereiche am unteren Ende der Map enthalten zum Teil vermeidbare Löcher. Bereiche, die Details enthalten wurden zerschnitten (deutlich zu

sehen am linken Auge). Am oberen Ende der Textur-Map sammeln sich eine Vielzahl kleiner Texturstückchen, die zum Teil aus nur einem Polygon bestehen.



(a) Das Bunny. **(b)** Die Textur.

Abbildung 75: Dieser Stanford Bunny hat eine von Autodesk Maya automatisch erstellte Textur-Map.

Nähte sind auf dem Modell trotzdem nicht erkennbar. Mit dieser Textur-Map wurden die Testbilder der Nahtfilter erstellt (Kapitel 5.1).

Die Textur des Killeroo enthält 14 Ebenen, eine auf sechs Ebenen reduzierte Version diente als Beispiel für Abbildung 23 auf Seite 24. Die Augen wurden in der 2D Ansicht erstellt, der Rest im 3D. Die Augen nehmen in der Textur-Map einen überproportional großen Bereich ein. Sie sind fast so groß, wie die Füße. Hierdurch können die Augen so detailliert texturiert werden (siehe Abbildung 76). Bestimmten Regionen einen größeren Teil der Textur zuordnen zu können ist ein Vorteil der manuellen Textur-Map Erstellung.

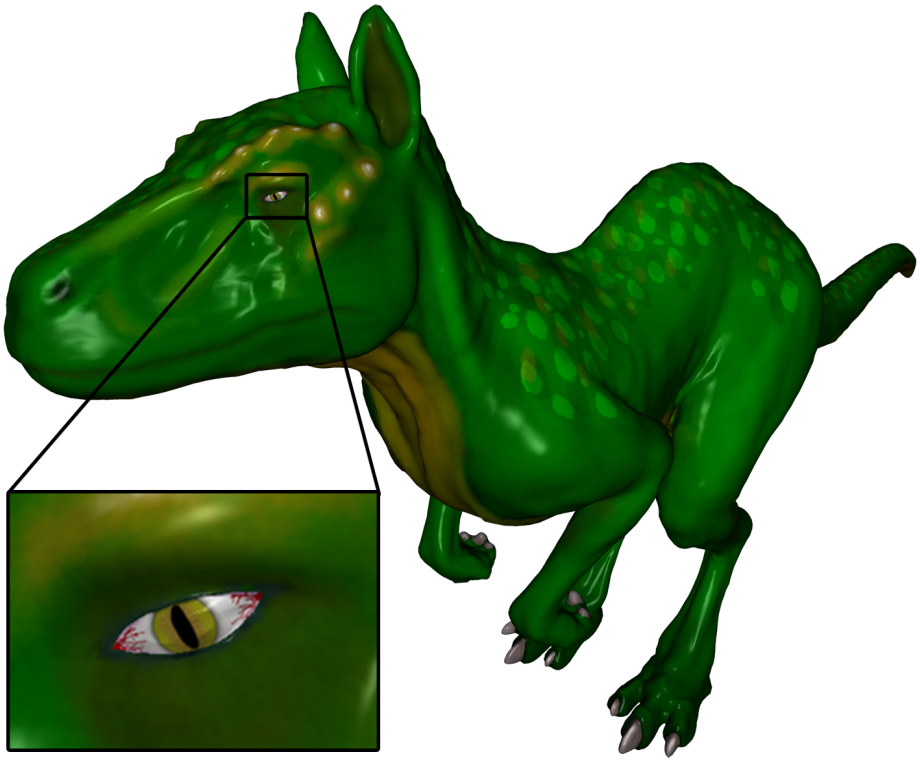


Abbildung 76: Killeroo

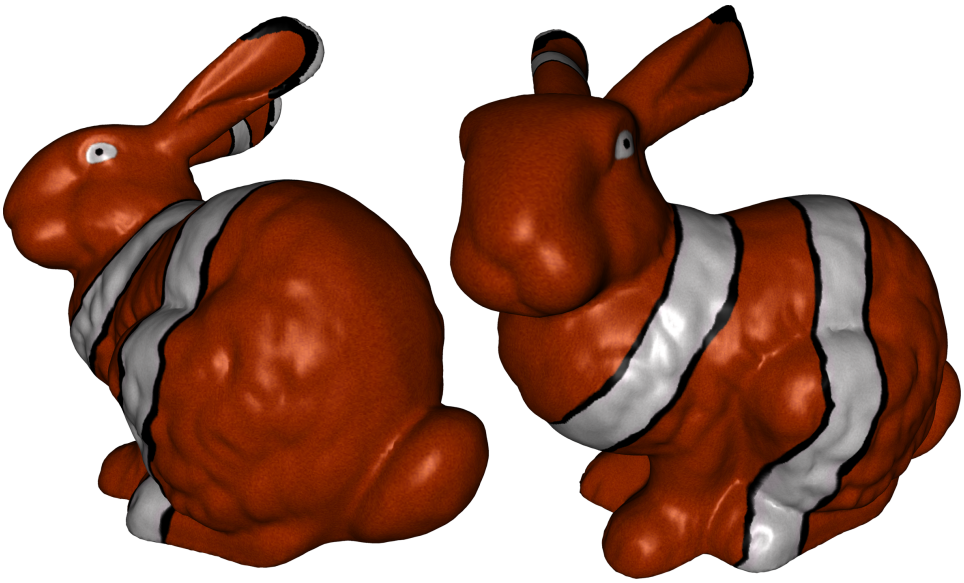


Abbildung 77: Das Stanford Bunny im Stil eines Clownfisches.

6 Ausblick

Das projektive Malen auf Texturen lässt sich noch in einigen Punkten erweitern. Zunächst einmal sind weitere Funktionen aus der klassischen Bildbearbeitung möglich, wie zum Beispiel Selektionen und Ebenenmasken. Durch beide lassen sich die Bereiche einer Ebene, in die gemalt werden können, einschränken. Beides lässt sich mit Hilfe von Graustufen Texturen umsetzten, die für jedes Texel festlegen, wie sehr es "selektiert", also änderbar ist. Bei der Anwendung einer Projektion muss hierfür lediglich ein Zusätzlicher Texturzugriff in die Selektionstextur durchgeführt werden, um zu ermitteln, wie stark dieses Texel geändert werden darf. Im Shader würde man also zunächst aus der alten Texelfarbe a und der zu projizierenden Farbe p wie bisher die neue Farbe n errechnen. Anschließend würde man in der Selektionstextur die Stärke der Selektion s auslesen. Der tatsächliche neue Texelwert würde sich dann als $a*(1-s)+n*s$ ergeben. Hierbei würde ein Selektionswert von 0 nicht selektiert, also nicht änderbar und ein Wert von 1 selektiert, also änderbar bedeuten.

Bestehende Bilder oder Fotos auf das Modell abzubilden kann oft hilfreich sein. Etwa wenn Werbung auf das Modell eines Sportwagens "geklebt" werden soll. Dies ist mit dem projektiven Ansatz leicht realisierbar, da schon die Puffer-Textur nichts anderes als ein Bild ist, das auf das Modell projiziert wird.

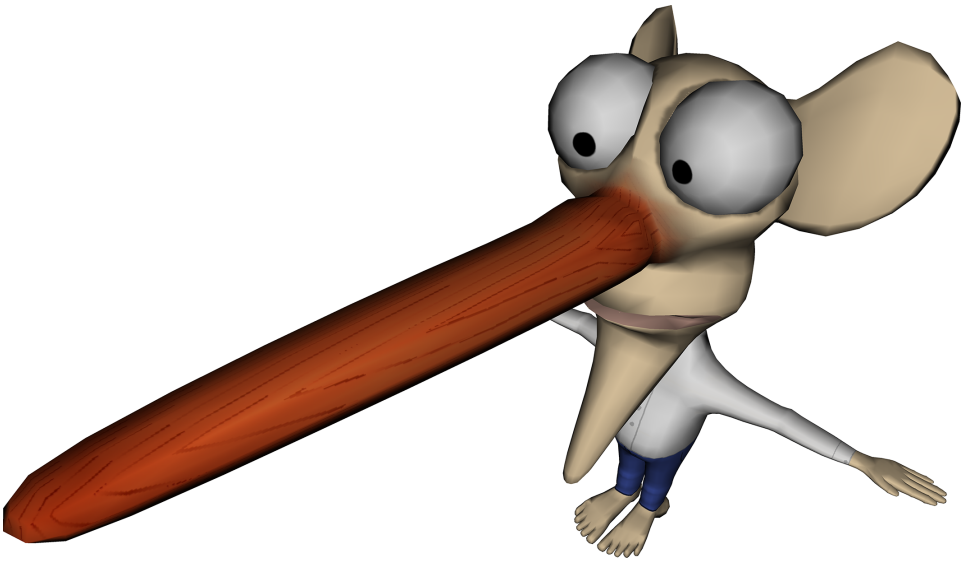


Abbildung 78: Die Nase wurde durch einen prozeduralen Holzfilter erstellt, der als Beispiel in MeshPaint implementiert wurde.

Bildbearbeitungsfilter, die den Farbwert einzelner Pixel ändern, ohne die Nachbarschaft betrachten zu müssen, lassen sich sehr leicht als Fragment-

Shader umsetzen. So sind zum Beispiel Helligkeits- und Kontrastanpassungen leicht zu implementieren [Bjo04]. Auch Farbverfälschungen lassen sich so erreichen. Hierfür muss die Textur lediglich mit diesem Filter behandelt werden, als wäre es ein reguläres Bild.

Oftmals dienen prozedurale Texturen als einfache Beispiele für die Shaderprogrammierung. Hierbei wird zum Beispiel eine Marmor- oder Holzstruktur auf ein Modell gelegt. Dies lässt sich mit geringen Anpassungen auch direkt in eine Textur-Ebene hinein rendern. Hierdurch ergibt sich der Vorteil, schnell eine realistische Struktur auf ein Objekt legen zu können, diese Textur aber trotzdem noch weiter bearbeiten zu können. Abbildung 78 zeigt den Nasenmann aus dem Nutzertest, der durch einen prozeduralen Holzfilter zu Pinocchio gemacht wurde. Die Holzmaserung muss nicht manuell erstellt werden, lediglich die Farbe vom Holz und der Maserung müssen gewählt werden.

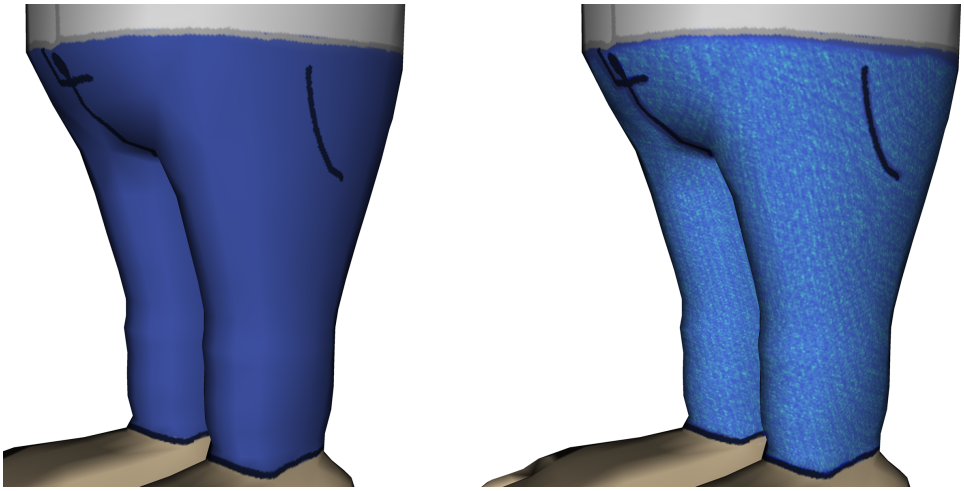


Abbildung 79: Die Hose links wurde mit einem Rauschen im Ebenenmodus *color dodge* überlagert. So entsteht ein Eindruck von Stoff.

Abbildung 79 zeigt ein weiteres Beispiel: Durch eine zusätzliche Textur-Ebene, in die mit einem Filter ein Rauschen gerendert wurde, wird Stoff imitiert. Hierfür wird aus dieser Ebene alles, was nicht über der Hose liegt, ausradiert und die Ebene im Modus *color dodge* auf die Hose geblendet. Auch dieser Filter wurde exemplarisch in MeshPaint integriert.

Filter, die die Nachbarschaften eines Texels betrachten müssen, stellen hingegen ein Problem dar. Hierunter fallen bereits Faltungen zum Weich- oder Scharfzeichnen. Auch Werkzeuge, wie der “Wischfinger“ aus Bildbearbeitungsprogrammen benötigt zur Errechnung eines neuen Texels die Farbwerte der Nachbartexel, aber gerade diese Information fehlt an Nähten.

Abbildung 80: Das Wischfinger-Werkzeug aus Adobe Photoshop: Über den Text oben wurde einmal von links nach rechts “gewischt“ (unten).

Um dennoch solche Werkzeuge zu implementieren, kann man sich eines Tricks bedienen: Zunächst wird das Modell aus der aktuellen Kameraposition ohne Beleuchtung in eine Textur gerendert. In dieser können dank vorhandener Nachbarschaftsinformationen beliebige Bildverarbeitungs Operationen durchgeführt werden. Anschließend wird diese modifizierte Textur auf das Modell zurück projiziert (analog der Puffer-Textur, die Pinselstriche zwischengespeichert hat). Der Nachteil liegt bei diesem Vorgehen darin, dass die Genauigkeit dieser Operationen durch die Auflösung eben dieser Zusatztextur limitiert wird.

7 Fazit

Ein Nachteil der Octree Texturen liegt in der Limitierung auf das 3D Painting, doch wie die Nutzertests gezeigt haben, wird die Möglichkeit der direkten Editierung der Textur-Map gewünscht und genutzt. Des weiteren ist der Aufwand zur Darstellung höher als bei regulären Texturen. Die Umsetzung des Malens in Octree Texturen erwies sich als wesentlich aufwändiger, als erwartet. Octree Texturen stellen somit keine geeignete Alternative zur Texturierung dar.

Die direkte Bearbeitung der Textur durch projektives Malen funktionierte hingegen erstaunlich gut. Selbst große Modelle lassen sich so bemalen, auch die Größe der Textur wird vor allem durch die Grafikkarte begrenzt.

Es konnte gezeigt werden, dass sich die Probleme mit den Artefakten an Nahtstellen mit Filtern umgehen lassen. So eignet sich dieses Verfahren auch zur Texturierung mit automatisch erstellten Textur-Maps. Hierdurch fällt die Notwendigkeit einer manuell erstellten Map weg, es können also auch unparametrisierte Modelle automatisch für die Texturierung durch projektives Malen vorbereitet werden. Die Möglichkeit, die Textur im 2D bearbeiten zu können, bleibt allerdings weiterhin erhalten.

Projektives Malen eignet sich somit gut, wenn das Modell in Echtzeit gerendert werden kann. Um extrem große Modelle zu texturieren eignet es sich hingegen nicht mehr, da hierfür ein zusätzlicher Aufwand getrieben werden müsste, um echtzeitfähig zu bleiben. Dieser Aufwand wird bei der Texturierung mit Vertex Color allerdings bereits betrieben, nämlich aus der Notwendigkeit heraus, dass auch kleine Modelle hier extrem vergrößert werden müssen. Durch die vorhandenen Nachbarschaftsinformationen im Mesh können manche Werkzeuge hier einfacher implementiert werden (wie zum Beispiel der Wischfinger).

Daher lohnt sich der Aufwand nicht, das projektive Malen für extrem große Modellen anzupassen. Hier kann direkt die Texturierung über Vertex Color umgesetzt werden.

Es konnte gezeigt werden, dass sich das projektive Malen sehr gut für die Bearbeitung von Texturen eignet. Sowohl für manuell erstellte, die der Künstler im 2D bearbeiten möchte, als auch für automatisch erstellte.

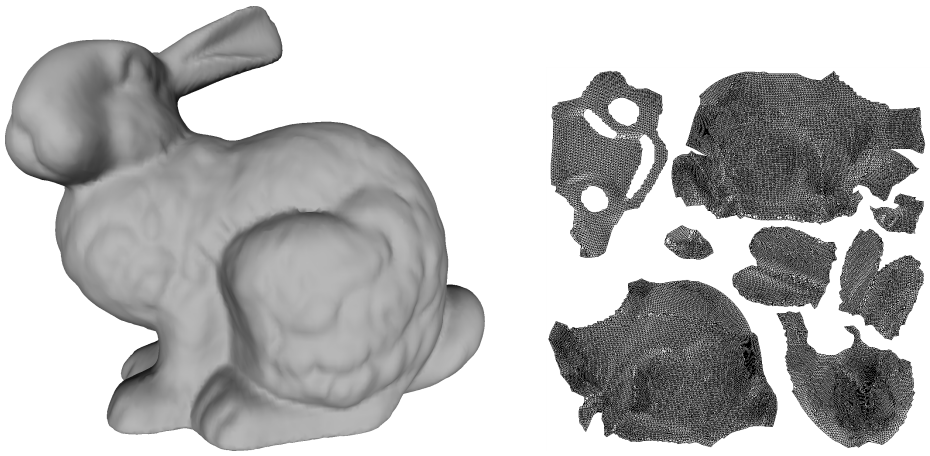
A Modellverzeichnis

Modelle, die nicht als Dreiecksgitter vorlagen, wurden entsprechend konvertiert. Für die aufgeführten Modifikationen, Konvertierungen und die Erstellung von Textur-Maps wurden Autodesk Maya [Inc08a], MeshLab [Cig08] und UVLayout [hmPL08b] verwendet.

Folgende Modelle wurden für den Nutzertest verwendet:

Stanford Bunny [Lev08]

Das Stanford Bunny wurde im Nutzertest in der Originalversion mit 69451 Polygonen angeboten. Für die Benchmarks wurden Versionen mit 5000 bis 1,1 Millonen Polygonen erzeugt.

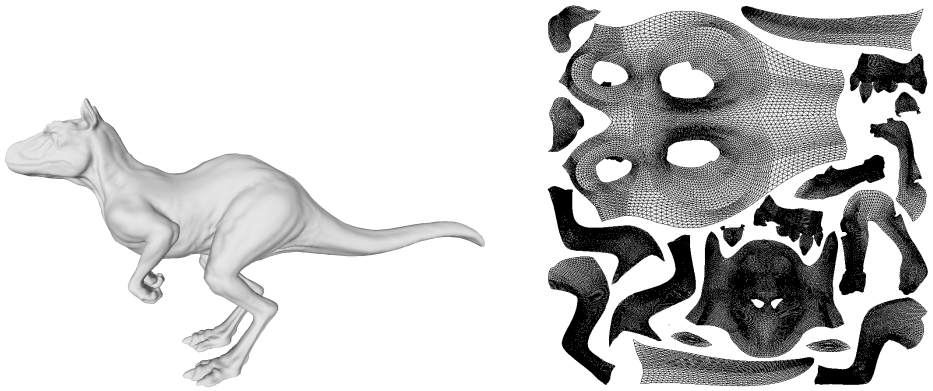


(a) Stanford Bunny **(b)** Textur-Map

Abbildung 81: Stanford Bunny mit Textur-Map.

Killeroo [hmPL08a]

Das Killeroo Modell wurde nachbearbeitet, da das Original keine Augen enthielt. Diese bearbeitete Version hat 92783 Polygone.

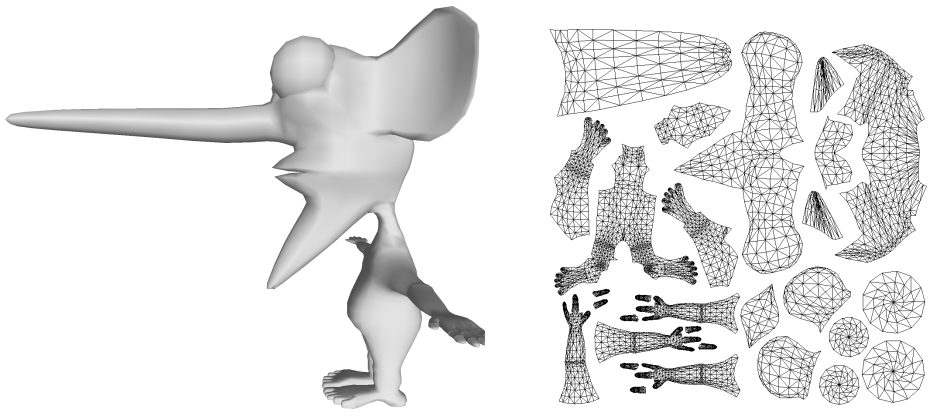


(a) Killeroo **(b)** Textur-Map

Abbildung 82: Killeroo mit Textur-Map.

Nasenmann

Das Modell Nasenmann wurde freundlicherweise von Stefan Rilling zur Verfügung gestellt, er besteht aus nur 9376 Polygonen.



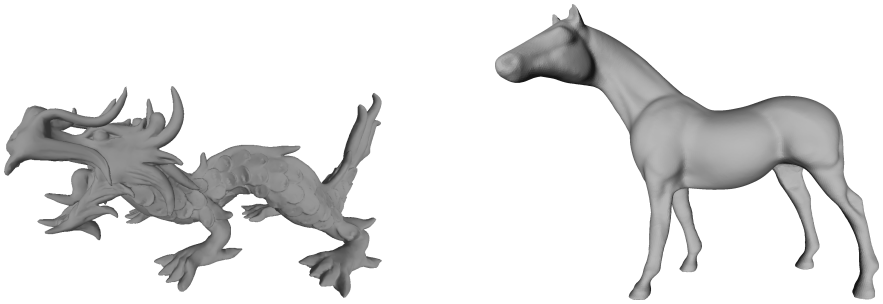
(a) Nasenmann **(b)** Textur-Map

Abbildung 83: Nasenmann mit Textur-Map.

Weitere Modelle:

Der Asian Dragon [Lev08] wurde in einer Version mit 56390 Polygonen verwendet.

Das Modell Horse [TM08] besteht aus 96964 Polygonen.



(a) Asian Dragon **(b)** Horse

Abbildung 84: Asian Dragon und Horse

B Benchmark Ergebnisse

Tabelle 2: Dauer zum Mischen mehrerer Ebenen in Millisekunden. Angegeben sind die Durchschnittswerte aus 250 Durchläufen.

Vorschau-Textur Erzeugung					
Ebenen	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
1	1.872	1.748	1.94	2.504	18.124
2	1.564	1.564	3.996	2.688	84.0
3	1.624	2.064	6.624	3.376	141.188
4	1.812	3.0	9.624	10.752	176.188
5	3.188	4.812	13.496	22.004	225.252
6	3.188	5.5	15.996	33.128	162.128
7	3.124	6.252	18.62	43.884	-
8	4.688	8.312	22.624	55.756	-
9	4.748	9.0	25.308	70.136	-
10	4.752	9.688	27.808	88.076	-
11	6.312	11.812	31.872	100.516	-
12	6.312	12.564	34.62	110.58	-
13	6.312	13.184	37.124	124.952	-
14	7.876	15.252	41.06	134.196	-
15	7.876	16.0	43.744	145.572	-
16	7.876	16.624	46.432	155.636	-
17	9.436	18.812	50.368	165.76	-
18	9.5	19.436	53.12	175.884	-
19	9.44	20.188	55.808	188.636	-
20	11.0	22.188	59.68	200.136	-
21	11.06	22.876	62.432	207.636	-
22	11.188	23.564	65.056	223.884	-
23	12.94	25.624	68.932	229.324	-
24	13.128	26.372	71.748	243.264	-
25	13.188	27.064	74.368	249.636	-
26	15.0	29.192	78.248	260.008	-
27	15.188	29.876	80.996	273.448	-
28	15.312	30.564	83.62	282.132	-
29	17.064	32.688	87.556	290.756	-
30	17.188	33.372	90.368	301.444	-
31	17.312	34.064	93.06	314.508	-
32	19.0	36.188	96.808	327.572	-
33	19.188	36.816	99.616	-	-
34	19.376	37.5	102.244	-	-

Tabelle 2 – Fortsetzung

Ebenen	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
35	21.064	39.624	106.184	-	-
36	21.188	40.376	108.932	-	-
37	21.376	41.128	111.496	-	-
38	23.128	43.188	115.496	-	-
39	23.188	43.876	118.996	-	-
40	23.312	44.564	121.436	-	-
41	25.064	46.752	124.624	-	-
42	25.312	47.376	127.12	-	-
43	25.252	48.0	129.68	-	-
44	26.876	50.188	133.684	-	-
45	27.064	50.876	136.56	-	-
46	27.188	51.504	138.932	-	-
47	28.816	53.5	142.936	-	-
48	29.0	54.248	145.744	-	-
49	29.312	55.064	148.136	-	-
50	30.94	57.188	151.96	-	-

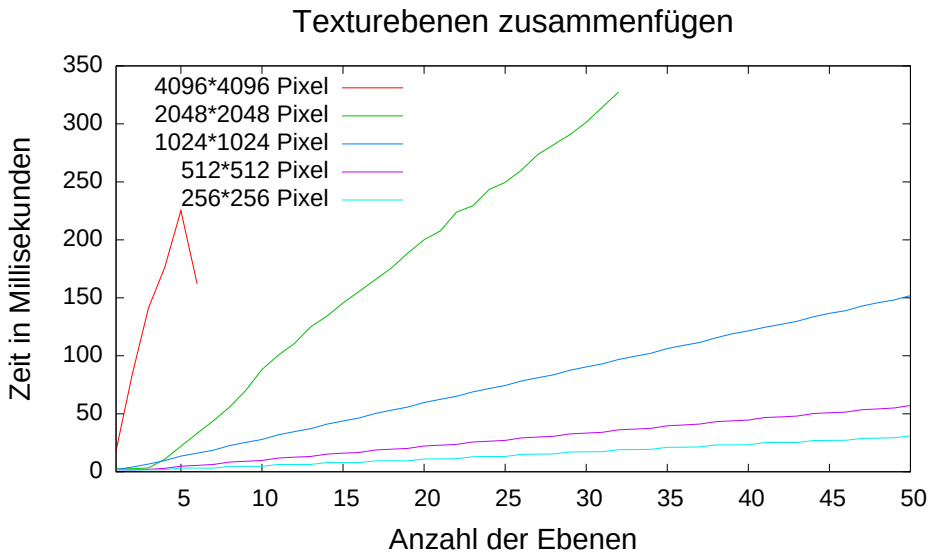


Abbildung 85: Zeitbedarf zur Erstellung einer Vorschautextur aus mehreren Textur-Ebenen.

Tabelle 3: Vorberechnung ohne Nahtfilter, alle Werte in Millisekunden.

Vorberechnung ohne Nahtfilter					
Polygonanzahl	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
5000	10,62	10,94	10,94	10,94	10,64
24999	11,24	11,26	11,24	11,24	11,24
69451	11,88	11,56	11,56	11,58	11,56
277804	14,7	14,68	14,38	14,38	15,32
625059	19,38	20,0	20,0	20,6	21,56
1111216	29,68	30,32	31,56	31,56	32,5

Tabelle 4: Vorberechnung mit Linienfilter, alle Werte in Millisekunden.

Vorberechnung mit Linienfilter					
Polygonanzahl	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
5000	10,62	10,62	10,92	10,94	13,76
24999	11,88	11,88	11,56	13,12	18,74
69451	21,86	21,26	20,32	20,92	26,26
277804	59,38	65,94	63,1	59,98	62,86
625059	118,76	141,26	151,28	145,28	134,06
1111216	201,24	231,86	273,44	274,7	258,44

Tabelle 5: Vorberechnung mit Punktfilter, alle Werte in Millisekunden.

Vorberechnung mit Punktfilter					
Polygonanzahl	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
5000	10,94	10,94	10,94	10,94	37,18
24999	35,32	27,5	20,32	17,82	46,26
69451	101,26	82,8	57,8	40,64	63,74
277804	420,0	372,52	291,16	193,72	163,88
625059	997,9	941,34	846,0	672,6	456,82
1111216	1819,0	1754,06	1650,66	1441,58	1179,7

Tabelle 6: Vorberechnung mit Kombifilter, alle Werte in Millisekunden.

Vorberechnung mit Kombifilter					
Polygonanzahl	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
5000	24,68	24,38	25,62	33,14	38,12
24999	53,42	46,24	40,32	44,68	51,56
69451	130,62	111,26	86,56	75,62	75,32
277804	490,3	448,78	364,58	268,1	229,86
625059	1135,44	1100,72	1014,02	834,14	688,4
1111216	2047,76	2011,26	1942,2	1726,26	1431,6

Tabelle 7: Vorberechnung mit Zusatzpolygonen, alle Werte in Millisekunden.

Vorberechnung mit Zusatzpolygonen					
Polygonanzahl	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
5000	29,08	29,38	29,68	30,0	332,5
24999	30,0	29,68	30,0	30,62	265,94
69451	30.64	30,32	30,32	30,94	313,44
277804	34,06	33,76	33,74	34,06	38,74
625059	17,5	-	-	-	-
1111216	25,94	-	-	-	-

Tabelle 8: Projizieren der Puffer-Textur, alle Werte in Millisekunden.

Projizieren der Puffer-Textur				
Texturgröße in Pixel				
256^2	512^2	1024^2	2048^2	4096^2
29,06	29,36	38,12	77,82	935,64

Tabelle 9: Anwendung des Brush Werkzeuges ohne erneute Berechnung der Vorschau Textur.

Brush					
Polygonanzahl	Texturgröße in Pixel				
	256^2	512^2	1024^2	2048^2	4096^2
5000	2,18	1,88	2,5	3,74	11,26
24999	2,2	2,2	2,18	3,76	9,68
69451	2,18	1,88	2,18	3,74	10,62
277804	2,2	1,88	2,18	3,44	9,06
625059	2,18	1,88	2,2	3,42	10,3
1111216	1,88	2,18	2,18	3,44	12,2

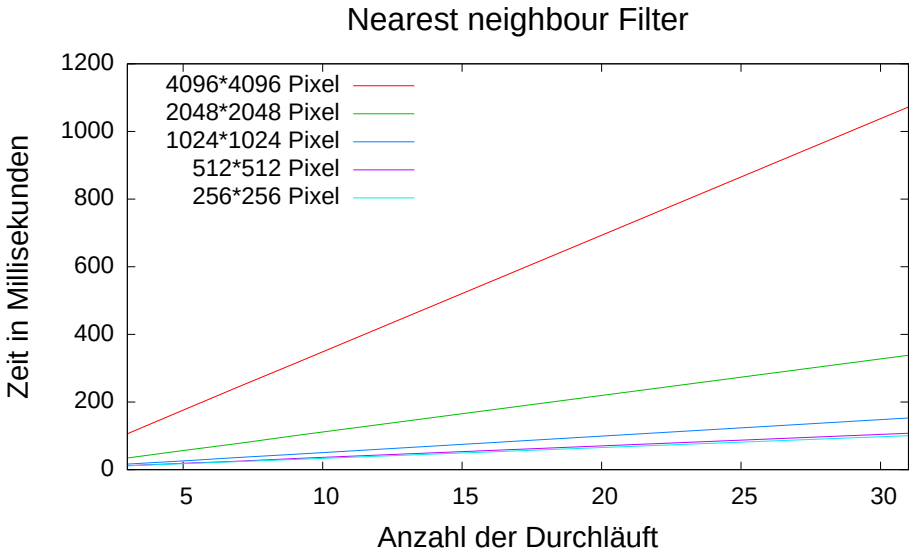


Abbildung 86: Der Zeitbedarf des Nearest Neigbour Nahtfilters ist nur von der Texturgröße abhängig und skaliert linear auch bei vielen Durchläufen.

Literatur

[ABL95] Maneesh Agrawala, Andrew C. Beers, und Marc Levoy. *3D Painting on Scanned Surfaces*. 1995.

[AM08] Tomas Akenine-Möller. Real-time rendering home page. `http://www.realtimerendering.com/`, Stand: 15.9.2008.

[AMH02] Tomas Akenine-Möller und Eric Haines. *Real-Time Rendering (2nd Edition)*. AK Peters, Ltd., July 2002.

[BD02] D. Benson und J. Davis. *Octree textures*. 2002.

[BGK03] A. Balazs, M. Guthe, und R. Klein. Fat borders: Gap filling for efficient view-dependent lod rendering. 2003.

[Bjo04] Kevin Bjorke. Color controls. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter 22, pages 363–373. Addison-Wesley Professional, March 2004.

[BS06] Tamy Boubekeur und Christophe Schlick. *Interactive Out-Of-Core Texturing with Point-Sampled Textures*. *Eurographics Symposium on Point-Based Graphics*, 2006.

[BS07] Jasmin Blanchette und Mark Summerfield. *C++ GUI Programmierung mit Qt 4*. Addison-Wesley Verlag, 2007.

[BVIG91] Chakib Bennis, Jean-Marc Vézien, Gérard Iglésias, und André Gagalowicz. Piecewise surface flattening for non-distorted texture mapping. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 237–246, 1991.

[CH02] Nathan A. Carr und John C. Hart. *Meshed atlases for real-time procedural solid texturing*. *ACM Trans. Graph.*, 21(2):106–131, 2002.

[CH04a] Nathan A. Carr und John C. Hart. Painting detail. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 845–852, New York, NY, USA, 2004. ACM.

[CH04b] Nathan A. Carr und John C. Hart. Two algorithms for fast reclustering of dynamic meshed surfaces. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 224–234, New York, NY, USA, 2004. ACM.

[Cig08] Paolo Cignoni. Main page - vcgmediawiki. `http://vcg.sourceforge.net/index.php/Main_Page`, Stand: 28.8.2008.

[DK95] Julie Daily und Kenneth Kiss. 3d painting: paradigms for painting in a new dimension. In *CHI '95: Conference companion on Human factors in computing systems*, pages 296–297, New York, NY, USA, 1995. ACM.

[DMK03] P. Degener, J. Meseth, und R. Klein. An adaptable surface parametrization method. In *The 12th International Meshing Roundtable 2003*, September 2003.

[EGL01] Stephen A. Ehmann, Arthur D. Gregory, und Ming C. Lin. *A touch-enabled system for multi-resolution modeling and 3D painting*. *The Journal of Visualization and Computer Animation*, 12(3):145–157, 2001.

[ESG01] Ilya Echstein, Vitaly Surazhsky, und Craig Gotsman. *Texture Mapping with Hard Constraints*. 2001.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, March 2004.

[FOL02] M. Foskey, M. Otaduy, und M. Lin. *Artnova: Touchenabled 3d model design*. 2002.

[gDGPR02] David (grue) DeBry, Jonathan Gibbs, Devorah DeLeon Petty, und Nate Robins. *Painting and rendering textures on unparameterized models*. *ACM Trans. Graph.*, 21(3):763–768, 2002.

[Geb08] Nikolaus Gebhardt. Irrlicht engine - a free open source 3d engine. `http://irrlicht.sourceforge.net/index.html`, Stand: 28.8.2008.

[GJDN02] D. Grue, D. Jonathan, G. Devorah, und P. Nate. *Painting and rendering textures on unparameterized models*. 2002.

[GK03] M. Guthe und R. Klein. Automatic texture atlas generation from trimmed nurbs models. In *Eurographics 2003*, September 2003.

[Hec86] Paul S. Heckbert. Survey of texture mapping. pages 207–212, 1986.

[HH90] Pat Hanrahan und Paul Haeberli. *Direct WYSIWYG painting and texturing on 3D shapes*. *In Proceedings of ACM SIGGRAPH 90*, pages 215–223, 1990.

[Hil08] Daniel Hilferty. G3d engine. http://g3d-cpp.sourceforge.net/, Stand: 28.8.2008.

[HLH96] Dave Hutchinson, F. Lin, und Terry Hewitt. *Surface Graph Sketching*. *Computer Graphics Forum*, 15(3):301–310, 1996.

[hmPL08a] headus (metamorphosis) Pty Ltd. headus (metamorphosis) - killeroo 3d scan. http://www.headus.com/au/samples/killeroo/index.html, Stand: 28.8.2008.

[hmPL08b] headus (metamorphosis) Pty Ltd. headus uvlayout - home. http://www.uvlayout.com/, Stand: 28.8.2008.

[IC01] Takeo Igarashi und Dennis Cosgrove. Adaptive unwrapping for interactive texture painting. 2001.

[Inc08a] Autodesk Inc. Autodesk - autodesk maya. http://www.autodesk.de/adsk/servlet/index?siteID=403786&id=11473933, Stand: 28.8.2008.

[Inc08b] Adobe Systems Incorporated. Adobe - photoshop developer center. http://www.adobe.com/devnet/photoshop/, Stand: 28.8.2008.

[JIK+99] David Johnson, Thomas V. Thompson II, Matthew Kaplan, Donald D. Nelson, und Elaine Cohen. Painting textures with a haptic interface. In *VR*, pages 282–285, 1999.

[Lev08] Marc Levoy. The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/, Stand: 28.8.2008.

[LHN05] Sylvain Lefebvre, Samuel Hornus, und Fabrice Neyret. Octree textures on the GPU. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 37, pages 595–613. Addison Wesley, March 2005.

[LM94] Peter Litwinowicz und Gavin Miller. *Efficient Techniques for Interactive Texture Placement*. *Computer Graphics Proceedings*, pages 119–122, 1994.

[Low01] Kok-Lim Low. Simulated 3D painting. Technical Report TR01-022, 8 2001.

[Ltd08] Torus Knot Software Ltd. Meshlab. http://meshlab.sourceforge.net/, Stand: 28.8.2008.

[MYV93] Jérôme Maillot, Hussein Yahia, und Anne Verroust. Interactive texture mapping. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 27–34, New York, NY, USA, 1993. ACM.

[nVi04] nVidia. Improve batching using texture atlases. `http://download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf`, 2004.

[PF05] Matt Pharr und Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005.

[Ros06] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[RSSP04] Patrick Reuter, Benjamin Schmitt, Christophe Schlick, und Alexander Pasko. *Interactive Solid Texturing Using Point-based Multiresolution Representations*. 2004.

[Sch99] Günter Schuler. *Photoshop Filter Factory*. SmartBooks Publishing AG, Kilchberg, erste Auflage, 1999.

[SGR96] Marc Soucy, Guy Godin, und Marc Rioux. *A texture-mapping approach for the compression of colored 3D triangulations*. *The Visual Computer*, 12(10):503–514, 1996.

[SWND07] Dave Shreiner, Mason Woo, Jackie Neider, und Tom Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2.1 (6th Edition)*. Addison-Wesley Professional, July 2007.

[Tea08] The GIMP Team. Gimp - the gnu image manipulation program. `http://www.gimp.org/`, Stand: 15.9.2008.

[TM08] Greg Turk und Brendan Mullins. Model : Horse. `http://www.cc.gatech.edu/projects/large_models/horse.html`, Stand: 28.8.2008.

[ZKY05] Xinyu Zhang, Young J. Kim, und Xiuzi Ye. Versatile 3d texture painting using imaging geometry. 2005.

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll, und Markus Gross. *Pointshop 3D: An Interactive System for Point-Based Surface Editing*. 2002.