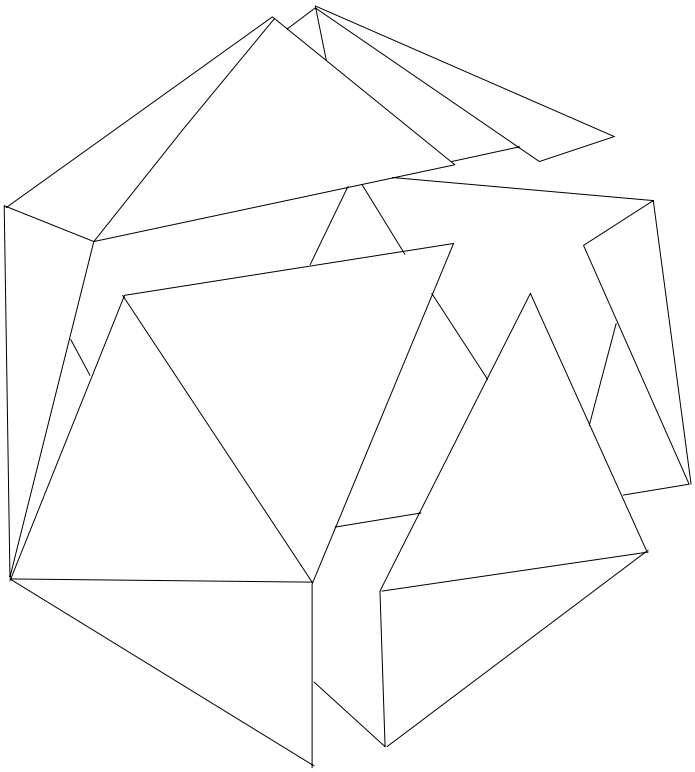
Geometric Compression Through Topological Surgery



Peter Wepf
GDV Fachseminar 97

Inhaltsverzeichnis

Literatur:

[1] Gabriel Taubin, Jarek Rossignac: Geometric Compression Through Topological Surgery, IBM Research Report Nr. 7990, 1996

[2] Michael Deering: Geometric Compression,
Computer Graphics (Proc. SIGGRAPH) August 1995

1. Einführung

In der modernen Computergraphik bedient man sich der Geometrie zur Beschreibung von 3D Objekten. Obwohl moderne Modellierungssysteme das Design von Freiformflächen unterstützen, wird in vielen Fällen auf einfache Primitive wie z.B. Polygone zurückgegriffen. Bevorzugt werden dabei Dreiecksflächen, da sie immer konvex und planar und daher von Algorithmen einfach zu verarbeiten sind. Ausserdem lassen sich allgemeine Polygone leicht in Dreiecke zerlegen.
Moderne Graphikhardware unterstützt das Rendern von Polygonen bzw. Dreiecken. Durch NURBS beschriebene Flächen können meist nicht direkt verarbeitet werden. Solche Objekte müssen von der Software zuerst trianguliert werden, bevor sie von der Hardware gerendert werden können. So gelangen Polyedermodelle dort zum Einsatz, wo es auf eine schnelle Darstellung von 3D Objekten ankommt.
Im Vergleich zu Video- und Bildkompression wurde der Komression von 3D Objekten bisher wenig Aufmerksamkeit geschenkt. Im industriellen CAD ist der steigende Bedarf an Haupt- und Sekundärspeicher für komplexe Modelle ein nicht zu vernachlässigbarer Kostenfaktor. Sollen 3D Modelle durch ein Netz übertragen werden, so ist die Übertragungsrate durch die begrenzte Bandbreite des Netzes eingeschränkt. Der Zeitaufwand für das Übermitteln komplexer Objekte ist deshalb sehr gross, was gleichzeitig die Wahrscheinlichkeit eines Übertragungsfehlers erhöht. Soll das Objekt angezeigt werden und kann der interne Speicher einer Graphikkarte nicht die gesamte Beschreibung eines Modells aufnehmen, muss dieses in mehreren Teilen nachgeladen werden, was sich negativ auf die Verarbeitungsgeschwindigkeit auswirkt. Dies sind Gründe, die eine kompakte Darstellung von 3D Objekten wünschenswert machen.

2. Methoden zur Komression von 3D-Meshes

Ein Mesh M = (V,K) wird beschrieben durch dessen Geometrie und Konnektivität. Im Falle von Dreiecksmeshes beschreiben die Eckpunkte die Geometrie des Meshes und die einzelnen Dreiecke deren Konnektivität.
Es existieren drei grundsätzlich verschiedene Ansätze ein 3D-Mesh zu komprimieren:

2.1 Vereinfachung des Polyedernetzes

Beim Vereinfachungsansatz wird versucht, die Anzahl Punkte aus denen ein Mesh besteht, zu reduzieren. Dazu wird die Konnektivität des Modells verändert und die Lage der Eckpunkte gegebenenfalls angepasst. Durch diese Vereinfachung verliert man die Information über die ursprüngliche Konnektivität des Netzes.

2.2 Kodierung der Geometrie

Diese Techiken reduzieren den Platz zur Speicherung von Geometrieinformationen, d.h. von Eckpunktkoordinaten, Normalen und Texturkoordinaten. Allgemeine binäre Kompressionsalgorithmen wie z.B. JPEG liefern dabei nur suboptimale Lösungen.
Mögliche Ansätze liegen im Ausnützen der endlichen Geometrie eines Modells. Oft werden die Positionskomponenten mittels 32 bit IEEE Gleitkommazahlen dargestellt, wobei 24 bit für die Mantisse und 8 bit für den Exponenten verwendet werden. Fixiert man den Exponenten für ein bestimmtes Modell, so muss er für dieses nur noch einmal gespeichert werden. Durch den Exponenten wird somit ein Objektraum festgelegt, in welchem die Geometrie des Modells durch die 24 bit-Mantisse bestimmt wird.
Deering [2] hat gezeigt, dass 16 bit zur Speicherung der Positionskomponenten für die meisten Modelle genügen. Eine genauere Beschreibung eines Objektes kann durch das Zusammensetzen mehrerer 16 bit-Objekträume erreicht werden.
Besitzen die Eckpunkte eine Ordnung, so kann anstelle der absoluten Position eine Punktes die Differenz zu seinem Vorgänger gespeichert werden. Aufgrund der Lokalität der Geometrien genügen zur Darstellung der Differenz meist weniger als 16 bit.

Die hier gewählte Art der Kompression geht noch weiter, indem ein Schätzfunktion für die Lage der Eckpunkte eingeführt wird. Die Position eines Eckpunktes wird dabei anhand seiner Vorgänger geschätzt. Kodiert wird dann nur noch die Differenz von der geschätzen zur wirklichen Lage des Punktes.

2.3 Kodierung der Konnektivität

Techniken zur Kodierung der Konnektivität versuchen die Redundanz, welche die Repräsentation eines 3D Meshes besitzt, zu reduzieren. Bei Dreiecksmeshes versucht man die Anzahl bit, die zur Beschreibung der Dreiecke nötig sind, zu minimieren.
In einer einfachen Implementierung kann ein Dreieck als ein Tripel von Pointern zu seinen Eckpunkten dargestellt werden. Die Beschreibung eines Dreiecks in einem einfachen Mesh mit 1000 Eckpunkten benötigte dabei 30 bit (3 x 10 bit Adressen). Nun besitzen Meshes mit einer einfachen Topologie etwa zweimal soviele Dreiecke wie Eckpunkte, alleine zur Kodierung der Konnektivitätsinformation würden so 60 bit pro Eckpunkt benötigt.
In Graphik APIs (z.B. OpenGL) werden häufig Triangle strips zur Kodierung der Konnektivität verwendet. Hier werden die einzelnen Eckpunkte so abgespeichert, dass jeweils 3 aufeinanderfolgende Eckpunkte implizit ein Dreieck definieren. Die Information über die Konnektivität liegt somit alleine in der Ordnung der Eckpunkte. Nun setzt sich ein Mesh aus verschiedenen Triangle strips zusammen, so wird im Durchschnitt jeder Eckpunkt zweimal gebraucht, entweder innnerhalb desselben oder in zwei benachbaten strips. Derselbe Punkt muss also mehrmals gespeichert oder referenziert werden.
Verwendet man Triangle strips mit Referenzierung der einzelnen Eckpunkte zur Kompression, wobei alle Koordinaten zur Dekompressionszeit zur Verfügung stehen müssen, so benötigt man zur Speicherung eines Triangle strips zwei Verweise zu den Anfangspunkten und je einen Verweis pro Dreieck. Ein zusätzliches bit pro Dreieck gibt an, welche offene Seite des vorangegangenen Dreiecks die Basis des neuen bildet. Die Länge der einzelnen strips und deren Anzahl muss ebenfalls gespeichert werden. Triangle strips lohnen sich nur, wenn es gelingt, ein Mesh in möglichst wenige lange strips zu zerlegen. Deering schlägt einen Stack Buffer zur Speicherung von 16 zuletzt verwendeten Punkten vor und benötigt so in der Dekompressionsphase keinen Zugriff auf sämtliche Punkte des Meshes. Hardwarelösungen können so auch mit begrenzten Speicher realisiert werden. Deerings Implementation benötigt zur Speicherung der Konnektivität durchschnittlich 11 bit pro Eckpunkt. Unter der Voraussetzung, dass zur Dekompressionszeit auf alle Eckpunktkoordinaten zugegriffen werden kann, liefert der nachfolgende Algorithmus eine 2-3 mal bessere Kompressionsrate.

3. Kompression

Durch den Kompressionsalgorithmus sollen Dreiecksmeshes in möglichst kompakter Form dargestellt werden. Der hier vorgestellte Komressionsalgorithmus gilt für orientierte Meshes mit Euler-Charakteristik χ=2. Die Oberfläche eines solchen Meshes lässt sich aus einem einfach verbundenen Polygon erstellen, indem man je zwei Kanten des Randes einander zuordnet. Durch Falten des Polygons an den inneren Kanten und Zusammenfügen aller korrespondierenden Kantenpaare des Randes, kann die Meshoberfläche konstruiert werden. Der Algorithmus geht umgekehrt vor, er zerlegt ein gegebenes Mesh in ein passendes Polygon und identifiziert die korrespondierenden Kantenpaare des Polygonrandes.
Zur Kodierung der Konnektivität wird das Mesh mit einem Subset seiner Kanten geschnitten. Diese Kanten werden im folgenden *cut edges* genannt. Die *cut edges* müssen so gewählt werden, dass sie in einer Baumstruktur alle Eckpunkte des Meshes erreichen. Dieser von den *cut edges* aufgespannte Baum heisst *Vertex Spanning Tree*. Die Kanten des *Vertex Spanning Tree* entsprechen den cut edges, die Knoten den Eckpunkten des Meshes.

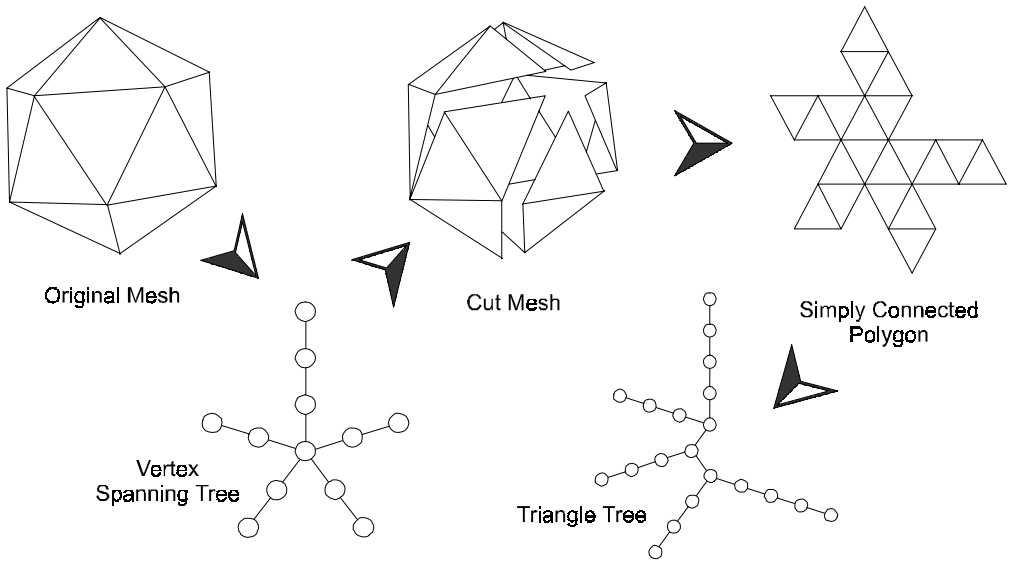


Abb. 3.1: Ablauf der Kompression

Zur Kodierung der Konnektivität wird das Mesh mit einem Subset seiner Kanten geschnitten. Diese Kanten werden im folgenden *cut edges* genannt. Die *cut edges* müssen so gewählt werden, dass sie in einer Baumstruktur alle Eckpunkte des Meshes erreichen. Dieser von den *cut edges* aufgespannte Baum heisst *Vertex Spanning Tree*. Die Kanten des *Vertex Spanning Tree* entsprechen den cut edges, die Knoten den Eckpunkten des Meshes.
Die Blatt- und Gabelknoten (*Y-vertices*) sind durch sogennante *Vertex Runs* verbunden. Innerhalb eines Runs besitzt jeder Knoten nur einen Nachkommen.
Durch die Wahl eines Blattknotens als Wurzel erhalten die Kanten des Baumes eine Orientierung und es entstehen Eltern-Kind Beziehungen zwischen den Knoten. Ist der Wurzelknoten bestimmt, kann der Baum linearisiert werden. Dazu wird der Vertex Spanning Tree pre-order traversiert, wobei die Äste eines branching nodes alle entweder im Uhrzeiger- oder Gegenuhrzeigersinn, bezüglich des entsprechenden Elternknotens, besucht werden. Die einzelnen Knoten werden in der Reihenfolge der Traversierung gespeichert. Zusätzlich muss die Struktur des Baumes gespeichert werden, dazu werden die Längen der durchlaufenen Runs in einem Array abgelegt. Zur Kodierung der Topologie des Baumes dienen zwei zusätzliche bits pro Run. Das *leaf bit* gibt an, ob der Endknoten des entsprechenden Runs ein Blattknoten ist oder nicht. Startet der Run in einem Gabelknoten, so gibt ein gesetztes *branching bit* an, ob noch weitere Runs vom selben Gabelknoten aus starten.
Betrachtet man die cut edges als topologische Grenze, so unterteilen sie das Mesh in eine Menge von Triangle Runs, die durch branching Triangles verbunden sind. Die Kanten, welche die einzelnen Dreiecke innerhalb eines Runs verbinden, heissen marching edges. Die verbundenen Triangle Runs können ebenfalls als Baum dargestellt werden. Dabei werden die Dreiecke durch Knoten repräsentiert, die Kanten entsprechen den marching edges. Der so gebildete Baum ist im Gegensatz zum Vertex Tree binär, da jedes Dreieck nur einen Vorfahren und höchstens zwei Nachkommen haben kann. Zur Kodierung der Topologie wird deshalb nur 1 bit pro Run benötigt, es gibt an, ob der Run in einem Blattknoten endet oder nicht. Ansonsten wird die Baumstruktur gleich kodiert.
Es folgen genauere Angaben zu den einzelnen Phasen der Kompression.

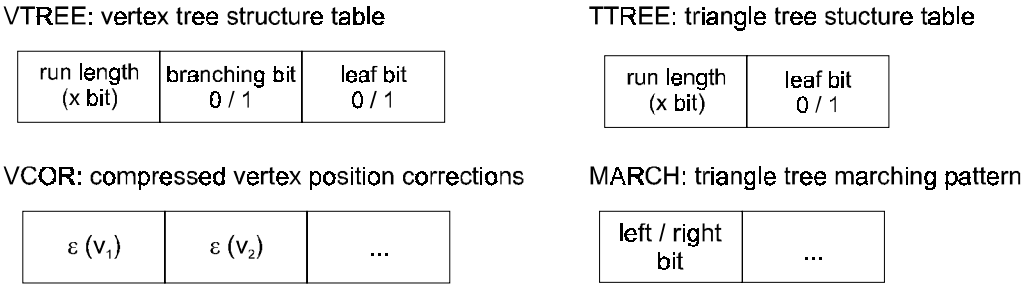


Abb. 3.2: Datenstruktur

3.1 Vertex Spanning Tree

Da die einzelnen Runs der beiden Bäume je einen Record in der VTREE bzw. TTREE Tabelle benötigen, hängt die Kompressionsrate des Algorithmus von der totalen Anzahl Runs beider Bäume ab. Optimale Kompression wird durch minimieren dieser Zahl erreicht. Der Algorithmus bedient sich dabei einer Art Distanztransfortmation, um ein Mesh in möglichst wenige Runs zu zerlegen. Zuerst wird ein Eckpunkt des Meshes als Wurzelknoten des Vertex Spanning Tree gewählt. Alle Dreiecke, die sich diesen Punkt als Eckpunkt teilen, bilden einen ersten Dreiecksgürtel um den Wurzelknoten. Die restlichen Eckpunkte dieser Dreiecke bilden zusammen mit den sie verbindenden Kanten die Grenze zwischen dem ersten und zweiten Dreiecksgürtel. Die Punkte dieser ersten Grenze sind über eine Kante mit dem Wurzelpunkt verbunden, der kürzeste Weg vom Startpunkt zu den Punkten der zweiten Grenze führt entlang zweier Kanten, zu den Punkten der dritten sind es drei usw...
Alle diese Grenzen unterteilen das Mesh in Dreiecksgürtel, die sich auf einfache Weise in eine Spirale verwandeln lassen. Als erstes wird der Wurzelknoten mit einem seiner Nachbarn in der ersten Grenze verbunden und die entsprechende Kante zum Vertex Spanning Tree hinzugefügt. Dann werden die Kanten und Punkte der ersten Grenze zum Vertex Tree hinzugefügt, wobei eine minimale Anzahl Kanten weggelassen wird, um die Baumstruktur zu erhalten. Nun wird eine Kante, welche die erste mit der zweiten Grenze verbindet, ausgewählt und zum Vertex Tree hinzugefügt. Danach werden die Kanten und Punkte der zweiten Grenze hinzugefügt, wobei es wiederum die Baumstruktur zu erhalten gilt. Bei der Wahl der Kanten ist jeweils darauf zu achten, dass sie die Anzahl Gabelungen (Äste) in den beiden Bäumen minimieren. Dieser Vorgang ist fortzuführen bis das ganze Mesh traversiert bzw. der Vertex Spanning Tree erstellt ist.

3.2 Kompression der Eckpunktkoordinaten

Zusätzlich zur Struktur des Vertex Tree, welche in der VTREE Tabelle gespeichert wird, müssen auch die Koordinaten der Eckpunkte abgelegt werden. Die Reihenfolge in welcher die komprimierten Koordinaten in VCOR abgelegt werden, entspricht einer top-down Traversierung des Baumes.
Zur Kompression werden die Koordinaten zuerst innerhalb der Bounding Box des Meshes normiert und auf die gewünschte Anzahl bits gerundet. Danach wird eine Schätzfunktion eingeführt, welche die Koordinaten der einzelnen Punkte anhand ihrer Vorgänger im Vertex Tree nähert. In der VCOR Tabelle werden nur noch die Korrekturterme von den geschätzten zu den eigentlichen Koordinaten abgelegt.
Innerhalb des Vertex Tree gibt es von jedem Punkt genau einen Pfad zum Wurzelknoten. Die Tiefe eines Knoten entspricht der Länge dieses Pfades. Die Tiefe des Wurzelknotens ist null. Die Position eines Eckpunktes der Tiefe n ist gegeben durch:

$$v_n = \varepsilon(v_n) + P(\lambda, v_{n-1}, v_{n-2}, \ldots, v_{n-K})$$

wobei $\varepsilon(v_n)$ den Korrekturterm und *P* die Schätzfunktion bezeichnen. λ und K sind Parameter des Schätzers und $v_{n-1},\ldots, v_{n-K}$ sind die K Vorfahren des Punktes auf dem Weg zum Wurzelknoten. Während der rekursiven Traversierung des Vertex Tree müssen ihre Werte jeweils in einer Tabelle gehalten werden. Da nicht alle Punkte des Baumes K Vorfahren besitzen, wird K bei einer kleineren Anzahl Vorfahren entsprechend angepasst. Zur Vorhersage der Koordinaten wird eine lineare Schätzfunktion verwendet:

$$P(\lambda, v_{n-1}, v_{n-2}, \ldots, v_{n-K}) = \sum_{i=1}^{K} \lambda_i v_{n-i}$$

Die Kompressionsrate hängt von der Wahl der Variablen λ und K ab. Sie können in verschiedener Weise gewählt werden, hier werden sie durch Minimieren des quadratischen Fehlers über allen Punkten mit Tiefe grösser als K festgelegt.

$$\sum_{n \geq K} \left\| \varepsilon_n \right\|^2$$

Die aneinander gereihten Korrekturwerte der einzelnen Punktpositionen werden in einem letzten Schritt noch Huffman oder JPEG/MPEG komprimiert.
Die Koeffizienten der Schätzfunktion sowie die Rundungsinformation gehören ebenfalls zu der komprimierten Darstellung des Meshes.

3.3 Erstellen des Bounding Loop

Da die Kanten des Vertex Tree den cut edges entsprechen und diese den Rand des aufgeklappten Polygons bilden, kann dessen Bounding Loop (Rand) ebenfalls mit Hilfe des Vertex Tree konstruiert werden. Besteht das Mesh aus V Punkten, so besteht der Bounding Loop aus 2V-2 Dreieckseckpunkten und ebenso vielen Kanten. Zu dessen Darstellung wird eine Tabelle mit 2V-2 Verweisen in die Tabelle der Eckpunktkoordinaten aufgebaut. Die Tabelle wird während der Traversierung des Vertex Tree erstellt. Dazu werden alle besuchten Knoten an die Tabelle angehängt und gleichzeitig auf einen Stack gelegt. Gelangt man bei der Traversierung zu einem Blattknoten, so wird zuerst sein Verweis an die Tabelle angehängt, um danach den Stack bis zum letzten Verweis auf einen Y-Vertex, von wo aus der nächste Run startet, abzuräumen und ebenfalls an die Tabelle anzuhängen. Ist der Vertex Tree traversiert und der Stack leer, so ist die Abfolge der Eckpunkte auf dem Rand des aufgeklappten Polygons bestimmt.

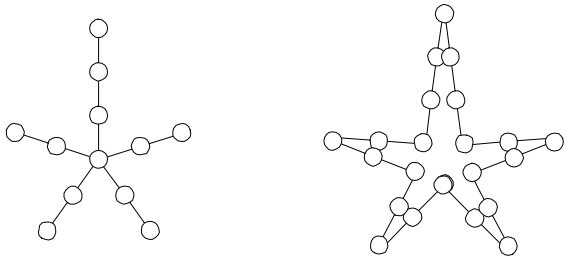


Abb. 3.3: Vertex Tree mit entsprechendem Bounding Loop

3.4 Kodierung des Triangle Tree

Während der Traversierung des Triangle Tree werden die Strukturtabelle des Triangle Tree sowie das marching pattern der Triangle Runs erstellt. Ein Blatt des Triangle Tree wird als Wurzeldreieck gewählt. Zwei Kanten dieses Dreiecks liegen auf dem Bounding Loop. Der

gemeinsame Punkt dieser Kanten ist der Wurzelpunkt des Triangle Tree. Sein Index auf dem Bounding Loop gehört ebenfalls zur komprimierten Darstellung des Baumes. Er wird ermittelt, indem man der cut edge von der Wurzel des Vertex Tree folgt, bis man die Spitze des ersten Blattdreiecks erreicht und dabei die besuchten Punkte zählt.
Zur Linearisierung wird der Triangle Tree pre-order traversiert. Jeder Triangle strip, der zwei leaf- oder branching Triangles miteinander verbindet, erhält einen Eintrag in der Triangle Tree structure Table. Die Länge eines Run entspricht dabei der Anzahl innerer Kanten bzw. der Anzahl Dreiecke plus eins. Das leaf bit gibt an, ob der Run in einem Blattknoten endet oder nicht.
Das marching pattern wird ebenfalls während der Traversierung des Triangle Tree erstellt. Dabei wird von jedem Dreieck mit gegebener Basis geschaut, auf welcher Seite des Bounding Loop der dritte Eckpunkt des Dreiecks liegt. Die entsprechende Seite wird mit einem bit in der MARCH Tabelle kodiert (1 für rechts, 0 für links). Da die Blattdreiecke durch die letzte marching edge des Run und den Bounding Loop bereits bestimmt sind, benötigen sie keinen Eintrag im marching pattern. Die Länge des pattern für einen Run entspricht deshalb der Anzahl seiner Dreiecke minus 1.
Ist das marching pattern des Baumes erstellt, wird es wie die Eckpunktpositionen noch komprimiert.

4. Dekompression

Die Kombination des Vertex Spanning Tree und des Triangle Tree mit den komprimierten Eckpunktpositionen erlaubt es uns, für jeden Triangle Run dessen Länge, seinen Rand sowie die Eckpunkte eines jeden Dreiecks zu bestimmen. Dazu wird zuerst eine Tabelle mit Eckpunktindices erstellt, welche die Reihenfolge der Eckpunkte auf dem Bounding Loop angibt.
Die top-down Traversierung des Triangle Tree definiert uns den linken und rechten Rand eines jeden Triangle Run. Da sowohl der linke wie der rechte Rand eines Triangle Run verbundene Subsets des Bounding loops bilden, ist der Rand eines Triangle Run gegeben durch je einen Startpunkt auf jeder Seite des Run, sowie der Anzahl Eckpunkte auf jeder Seite. Die Anzahl marching edges eines Run kann aus der TTREE Tabelle entnommen werden, diese Zahl entspricht der Anzahl Eckpunkte auf beiden Seiten des Run. Die Anzahl Nullen im marching pattern des Run entspricht der Anzahl Punkte auf seinem linken Rand. Mit den Verweisen auf den linken und rechten Startpunkt des Run und der Information des marching pattern können die Dreiecke eines Run konstruiert werden.
Am Ende eines Run folgt entweder ein Blatt des Dreiecksbaumes oder ein branching Triangle. In letzterem Fall bildet die letzte marching edge des Run die Basis des branching Triangle. Der dritte Eckpunkt des branching Triangle wird Y-Vertex genannt. Er bildet den linken Startpunkt des nächsten Triangle Run. Sein Offset zum letzten Punkt auf dem linken Rand des vorhergegangenen Run wird in einem Preprocessing Schritt berechnet. Für jeden branching Triangle werden die entsprechenden Offsets in einer Tabelle abgelegt.
Der Dekompressions Algorithmus durchläuft den gesamten Triangle Tree rekursiv und rekonstruiert die entsprechenden Triangle Runs.
Nachfolgend einige ausführlichere Angaben zu den einzelnen Phasen der Dekompression:

4.1 Rekonstruktion der Geometrie

Im ersten Schritt der Dekompression werden die Koordinaten der Eckpunkte des Meshes rekonstruiert und in einer Tabelle abgelegt. Auf diese Tabelle muss während der gesamten Phase der Dekompression wahlfrei zugegriffen werden können.
Die totale Anzahl der Mesh-Eckpunkte kann aus der Vertex Tree Tabelle errechnet werden, sie entspricht der Summe der Längen aller Vertext Runs plus 1. Mit Hilfe der Position des Wurzelknotens des Vertex Tree und der Tabelle mit den Korrekturtermen zu den einzelnen Eckpunkten, können ihre Koordinaten rekonstruiert werden. Die VCOR Tabelle liegt in komprimierter Form vor und muss deshalb zuerst dekomprimiert werden. Danach wird der Vertex Tree depth first traversiert und die Koordinaten der einzelnen Punkte anhand ihrer

Vorgänger im Vertex Tree berechnet. Dazu muss während der Traversierung ein Array mit Indizes zu den Koordinaten der Vorgängerpunkte verwaltet werden.

4.2 Startpunkte der einzelnen Triangle Runs

Während der Traversierung des Vertex Tree wird der Bounding Loop des Polygons erstellt. Dies geschieht wie in 3.3 beschrieben. Mit dem Bounding Loop besitz man den Rand der zusammenhängenden Triangle Runs, weiss aber, bis auf den ersten Run, welcher zum Wurzeldreieck gehört, nicht, wo die einzelnen Runs starten. Um die einzelnen Dreiecke mit Hilfe des Triangle Trees zu rekonstruieren, benötigt man für jeden Run die Indices seines rechten bzw. linken Startpunktes auf dem boundig Loop.
Da alle Runs, bis auf den ersten von branching Triangles aus starten, wird für jeden branching Triangle der offset seines dritten Eckpunktes (y-Vertex) bezüglich des letzten Punktes auf der linken Seite des vorangegangenen Triangle Run berechnet und in der y-Vertex Look-up Table abgelegt. In Abb. 3.4 sind die Offsets der y-Vertices (10,14,18) alle bezüglich des Punktes 1 zu berechnen.

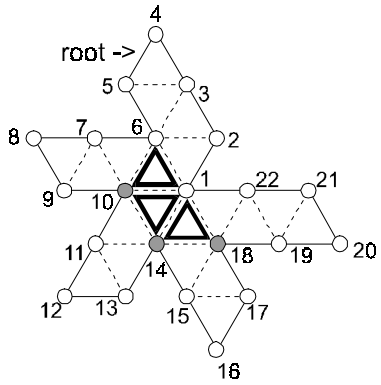


Abb 3.4: branching Triangles mit entsprechenden y-vertices

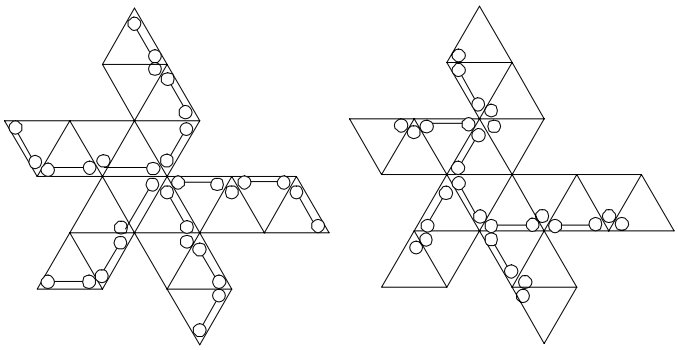
4.3 Rekonstruktion der Dreiecke

Ein Pointer zu einem Wurzelpunkt identifiziert das Wurzeldreieck im Triangle Tree. Seine linken und rechten Eckpunkte sind die Vorgänger bzw. Nachfolger auf dem Bounding Loop. Die restlichen Dreiecke werden rekursiv mit Hilfe zweier Stacks rekonstruiert.
Besitzt man die beiden Startpunkte auf jeder Seite eines Runs, gibt einem das marching pattern des entsprechenden Runs an, wie viele Punkte auf welcher Seite des Runs in welcher Reihenfolge folgen. Die Dreiecke des Runs können so mit Hilfe der Bounding Loop Tabelle rekonstruiert werden.
Endet ein Run in einem branching Triangle, so wird dieses Dreieck durch die letzen beiden Punkte des Runs und dem zum branching Triangle gehörenden y-Vertex gebildet. Der y-Vertex wird dabei auf den right Vertex Stack, der letzte auf der linken Seite des Runs liegende Punkt wird auf den left Vertex Stack gelegt. Die beiden Startpunkte des nächsten Runs entsprechen dem aktuallen rechten Randpunkt und dem y-Vertex. Endet ein Run in einem leaf Triangle, so sollte der Nachfolger des aktuell rechten Randpunktes gleich dem Nachfolger des aktuell linken Randpunktes sein. Das Blattdreieck des Runs kann somit rekonstruiert werden. Die Startknoten des nächsten Runs können dann vom right bzw. left Vertex Stack geholt werden. Sind beide Stacks leer, so wurden alle Dreiecke rekonstruiert.

5. Kodierung der Photometrie Daten

Normalen, Farben und texture mapping liefern Zusatzinformationen, die zur realistischeren Darstellung eines Meshes benötigt werden.
Eine Normale ist ein dreidimensionaler Floating Point Einheitsvektor, eine Farbe wird durch einen dreidimensionaler Floating Point Vektor innerhalb $[0,1]^3$ dargestellt. Ein texture

mapping Vektor ist ein zweidimensionaler Vektor innerhalb des Einheitsquadrats $[0,1]^2$. Normalen, Farben und texture mapping Vektoren können in ähnlicher Weise kodiert werden. Nachfolgend ist dieser Prozess für Normalen beschrieben.
Für ein Flat Shading benötigt man eine Normale pro Dreieck. Für ein Gouraud- oder Phong Shading benötigt man eine Normale pro Eckpunkt. Beim Gouraud Shading werden zuerst die Farbintensitäten der Eckpunkte errechnet, die Farbintensität innerhalb eines Dreiecks erhält man durch Interpolation der Eckpunktintensitäten, beim Phong Shading interpoliert man zuerst die Normalen und errechnet danach die Farbintensität. Zur Darstellung scharfer Kanten benötigt man drei Normalen pro Dreieck.
In allen drei Fällen werden die Normalen als Baum organisiert. Im Falle von einer Normalen pro Eckpunkt entspricht dieser dem Vertex Tree, bei einer Normalen pro Dreieck hat er die Struktur des Triangle Tree. Die einzelnen Daten werden wie bei den vorherigen Fällen in der Reihenfolge einer depth-first Traversierung in einer Tabelle abgelebt, die Struktur der Bäume ist bereits vorhanden und muss nicht nochmals kodiert werden.
Zur Anordnung von drei Normalen pro Dreieck, bedient man sich des Bounding Loop. Folgt man dem Rand des einfach verbundenen Polygons, so besucht man während eines Umlaufs jeden Knoten jedes Dreiecks genau ein Mal. Damit bei der Traversierung des Triangle Tree direkt auf die Normalen eines Dreiecks zugegriffen werden kann, werden sie in einen linken und rechten Normal Tree aufgeteilt. Im rechten Normal Tree werden jene Normalen abgelegt, die zu Eckpunkten auf der rechten Seite der Triangle Runs gehören, die anderen entsprechend im Linken. Die Normalen der Blattknoten von Blattdreiecken werden dem linken Normal Tree zugeordnet. Beim Linearisieren der Normal Trees verfährt man gleich, wie bei den anderen Bäumen.
Um den Speicherbedarf zu reduzieren, werden die Normalen ebenfalls quantisiert und kodiert.



6. Zusammenfassung

Zum Schluss noch eine grobe Zusammenfassung des Vorgehens bei der Kompression bzw. Dekompression von 3D Dreiecksmeshes:

Kompression:

1. Konstruktion des Vertex Tree und des Bounding Loop
2. Kodierung der Struktur des Vertex Tree
3. Kompression der Lage der Eckpunkte mit Hilfe eines linearen Schätzers
4. Konstruktion und Kodierung des Triangle Tree mit Hilfe des Bounding Loop

Dekompression:

1. Rekonstruktion der Eckpunktkoordinaten
2. Erstellen des Bounding Loop
3. Berechnung der Lage der y-vertices auf dem Bounding Loop
4. Rekonstruktion der Dreiecke anhand der Triangle Tree structure Tabelle und des Bounding Loop