UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung einer Beispielapplikation mit Hilfe von Geometrie-Shadern

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Andreas Michael von Arb

Betreuer: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Mai 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)

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die OpenGL-Pipeline hat sich mit der Zeit ein wenig gewandelt. Von der relativ starren fixed-function Pipeline, wie sie früher üblich war, über hardwarebeschleunigtes Transform & Lightning hin zu programmierbaren Funktionseinheiten, die mit Vertex und Fragment-Shadern ihren Anfang nahmen.

Immer noch relativ neu zur Zeit der Abfassung dieser Arbeit sind die Geometrie-Shader. Durch die Einführung von Direct X 10 wurden die neuen Möglichkeiten der Öffentlichkeit insbesondere den Entwicklern im Bereich der Computergrafik vorgestellt [Har07].

Aber der Einsatz von Geometrie-Shadern ist nicht auf Direct X 10 Anwendungen beschränkt. Auch mit dem herstellerunabhängigen Grafik-API OpenGL lassen sich Geometrie-Shader in Form einer Extension nutzen. Die Extension wurde von Nvidia entwickelt und mit den Treibern der GeForce 8 Serie ausgeliefert.

Betrachtet man die Publikationen, die zur Zeit der Recherche zu dieser Arbeit zugänglich waren, ist der allgemeine Wissensstand was den Einsatz der Geometrie-Shader angeht nicht besonders hoch. Vereinzelte Arbeiten beschäftigen sich bereits mit komplexeren Themen der Computergraphik unter Verwendung von Geometrie-Shadern, wie das Marching Cubes Beispiel von Cyril Crassin [Cra07]. In der Recherchephase waren aber hauptsächlich kleine Beispielanwendungen zugänglich [Bha08], manche davon lange Zeit nicht im Quelltext, wie der früh im Internet veröffentlichte Artikel von Yongming Xie [Yon08]. Diese Arbeit soll den Einstieg erleichtern und die Grundlagen der Geometrie-Shader erklären und versucht dies im Rahmen der zu erstellenden Beispielapplikation mit Geometrie-Shadern die im folgenden vorgestellt werden soll.

1.2 Zielsetzung

Ziel dieser Arbeit ist es die Technologie der Geometrie-Shader vorzustellen und diese Anhand einer geeigneten Bespielapplikation zu demonstrieren. Dabei sollen die persönlichen Kenntnisse im Umgang mit C++, OpenGL und der Shaderprogrammierung mit GLSL vertieft werden. Es wird angestrebt eine Anwendung mit grafischer Benutzungsoberfläche zu erstellen um dem Benutzer mehr Komfort zu bieten und soweit möglich die 3D Szene oder ein dargestelltes Objekt zu manipulieren. Als Einsatzgebiet für die Geometrie-Shader wurde ein multi-pass Verfahren zur Erzeugung der Geometrie eines Baumes ausgewählt. Die Parameter des Baummodells sollen dabei über das GUI interaktiv veränderbar sein.

2 Grundlagen

2.1 OpenGL-Pipeline

Grundlegende Kenntnisse der OpenGL-Pipeline sind notwendig, um bestimmte Details dieser Arbeit zu verstehen. Die OpenGL-Pipeline ist eine State Machine. Das bedeutet, dass die Grafikpipeline in OpenGL ihren Status behält und mit diesen Einstellungen der einzelnen logischen Einheiten auf den Eingabedaten arbeitet.

Eingabedaten sind Grafikprimitive wie Linienzüge, Polygone, Dreiecke und Dreiecksnetze. Die Primitive bestehen dabei aus einzelnen Punkten, genannt Vertices. Zu den Vertices lassen sich in OpenGL eine große Anzahl an Attributen angeben – wie zum Beispiel Position, Farbe und Normale – um nur die wichtigsten zu nennen. Ein Vertex durchläuft dann mitsamt den angegebenen Attributen die OpenGL-Pipeline. In dieser Arbeit ist der Teil der Pipeline vor der perspektivischen Division von besonderem Interesse. Dieser soll nun nochmals genauer betrachtet werden. Eine Ausführlichere Beschreibung der gesamten OpenGL-Pipeline findet sich zum Beispiel in [Kor08] auf Seite 21.

- Anfangs liegen die Vertices in Objektkoordinaten vor (Rechtssystem).
- Die Vertices werden durch die Modelview-Matrix in das Betrachterkoordinatensystem transformiert. Hier findet die Beleuchtung statt.
- Durch die Projection-Matrix werden die Vertices ins kanonische Volumen vor der perspektivischen Division transformiert. Nun liegt ein Linkssystem vor. Die Normalen sind nun nicht mehr brauchbar.

Die genannten ersten drei Schritte in der Pipeline können durch einen Vertex-Shader durchgeführt werden. Ein einfacher Vertex-Shader führt daher auch nur die Modelview- und Projection-Transformation durch.

Nach der Transformation durch die Projektionsmatrix werden die einzelnen Vertices wieder zu Primitiven zusammengesetzt (Primitive Assembly) abhängig vom aktuellen Primitivtyp. Danach folgt u.a. das Clipping, die perspektivische Division sowie die Viewport-Transformation und die Rasterisierung wie auf Abbildung 1 zu sehen ist.

2.2 Vertex-Buffer

Normalerweise werden Primitive in OpenGL von einem `glBegin/glEnd`-Paar gekapselt gezeichnet. Die Vertexattribute werden dabei im Arbeitsspeicher des Rechners (*OpenGL-Client*) gehalten und an die Grafikhardware (*OpenGL-Server*) gesendet. Wird die begin-end Semantik verwendet, kann

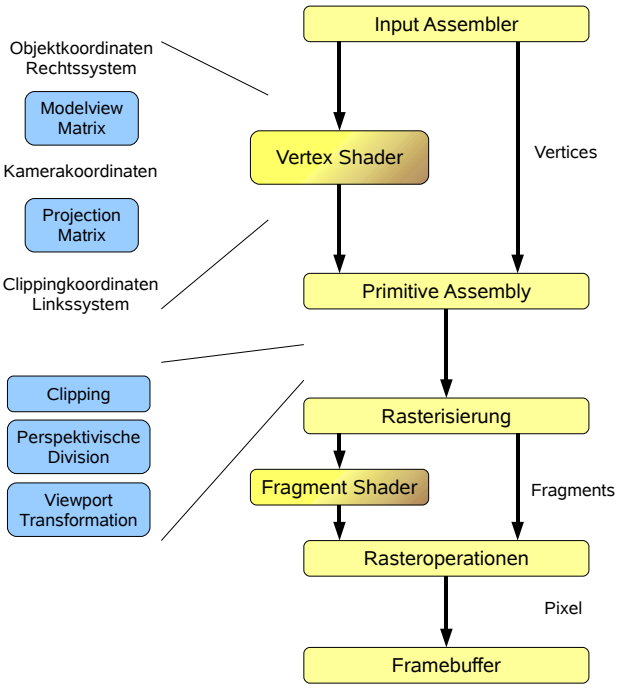

Abbildung 1: OpenGL-Pipeline

der GL-Client 3D-Modelle durch Manipulation der Vertexattribute des Modells leicht verändern. Andererseits sind die Geometriedaten für jedes Frame und jedes Objekt – das von einem Modell durch OpenGL dargestellt werden soll – erneut zur Grafikhardware zu senden. Wenn ein Modell mehrfach an verschiedenen Positionen gezeichnet werden soll, werden die Primitive und alle zugehörigen Attribute erneut an die Grafikkarte gesendet.

Im folgenden wird sich zeigen, dass Vertexdaten direkt im Speicher der Grafikkarte gehalten werden können. Dazu werden dort Pufferspeicher alloziert. Dabei wird auf Vertex-Arrays (siehe Kapitel 2.8 in [SA06]) zurückgegriffen, was eine performante Art ist größere Mengen an Vertexdaten der Pipeline zu übergeben [WH08]. Die Vertexdaten werden aber nicht im Arbeitsspeicher des Computers abgelegt, sondern direkt im Speicher der Grafikkarte.

Üblicherweise lässt sich ein Dreieck wie folgt zeichnen:

```
glBegin(GL_TRIANGLES);
glVertex3f( -1.0f, 0.0f, 0.0f);
glVertex3f( 1.0f, 0.0f, 0.0f);
glVertex3f( 0.0f, 1.4f, 0.0f);
glEnd();
```

Unter Verwendung von Vertex-Buffer-Objects (VBOs) (siehe Kapitel 2.9 in [SA06]) wird dieses einfache Beispiel aber ein wenig komplizierter. Zunächst muss über `glGenBuffer` ein gültiges Buffer Handle vom GL-Server erfragt werden. Soll das VBO angelegt werden, geschieht dies über `glBindBuffer`. Dieser Befehl erwartet ein gültiges Target (in unserem Fall `GL_ARRAY_BUFFER`) und das genannte Handle, welches in OpenGL als vorzeichenlose Ganzzahl gespeichert wird.

Der Puffer lässt sich nun durch `glBufferData` mit Daten füllen. Dazu wird auf ein Vertex-Array im Client-Speicher verwiesen, welcher dann auf den Server kopiert wird. Mittels `glBufferSubData` lässt sich ein Teilbereich des Puffers mit Daten füllen. Er läßt sich über `glMapBuffer` wieder in den Speicher des Clients mappen. Dort sind Veränderungen der Daten durch das laufende Programm möglich.

Sind die Vertexdaten im Speicher der Grafikkarte, können sie über den Befehl `glVertexPointer` adressiert werden. Für die verschiedenen Vertexattribute exisitieren entsprechende Pointer Funktionen über welche die zu zeichnenden Daten ausgewählt werden können.

Das Zeichnen der Daten selbst erledigt der Aufruf der Funktion `glDrawArrays`, welche eine angebene Anzahl von Vertices samt selektierter Vertexattribute zeichnet.

Mit den beschriebenen Methoden lässt sich ein Puffer auf der Grafikkarte anlegen, mit Daten füllen, verändern und auch durch OpenGL zeichnen. Das folgende Listing zeichnet ein Dreieck unter Verwendung eines Puffers auf der Grafikkarte.

```
uint vbo;
uint vertexCount = 9;
float vertexData[9] = { -1.0f, 0.0f, 0.0f,
                         1.0f, 0.0f, 0.0f,
                         0.0f, 1.4f, 0.0f };
GlGenBuffers(1, &vbo);
glBindBuffer( GL_ARRAY_BUFFER, vbo);
glBufferData( GL_ARRAY_BUFFER_ARB,
        vertexCount*sizeof(float),
                       &vertexData,
               GL_STATIC_DRAW_ARB);
glVertexPointer(3, GL_FLOAT, 0, NULL);
glDrawArrays( GL_TRIANGLES, 0, vertexCount);
```

Es lassen sich für Vertexattribute verschiedene Puffer als Quelle für die zu zeichnenden Daten nutzen (*separated*-Modus). So lassen sich zum Beispiel die Normalen in einem Puffer und die Positionen in einem anderen Puffer speichern. Beide Puffer werden – wie gezeigt – gebunden und die Vertex- bzw. Normalen-Pointer entsprechend gesetzt.

Vertexattribute lassen sich aber auch nacheinander in einem Puffer abspeichern (*interleaved*-Modus). OpenGL bietet dazu eine Reihe vordefinierter Formate. Um die Pointer nicht von Hand setzen zu müssen, kann eines dieser Formate benutzt werden. Zum Zeichnen von interleaved-Puffern existiert die Funktion `glInterleavedArrays`, die auch in der Anwendung verwendet wird im Zusammenhang mit Transform Feedback.

2.3 GLSL

GLSL ist die Hochsprache, die in OpenGL verwendet wird, um Shader zu schreiben. Ihre Syntax ist an C angelehnt. GLSL-Shader werden meist vom Grafikkartentreiber kompiliert und dann in den dafür vorgesehenen Funktionseinheiten auf der Grafikkarte ausgeführt. Man unterscheidet zwischen Vertex- und Fragment-, neuerdings auch Geometrie-Shadern. Gemeinsam werden diese in Form von Programmobjekten auf der GPU vorgehalten und auf Anforderung aktiviert. Es können dabei ohne weiteres mehrere Shaderprogrammobjekte im Speicher der GPU vorhanden, jedoch nur eines aktiv sein. Sind keine Shader aktiv übernimmt die OpenGL fixed-function Pipeline das Rendering der darzustellenden 3D Szene. Nachdem mit Vertex- und Fragment-Shadern die Transformation und Beleuchtung der Eckpunkte einerseits, die Rasterisierung und auch Texturierung der 3D Szene andererseits flexibel programmierbar geworden sind, lassen sich nun mit GLSL auch Geometrie-Shader programmieren, die auf der GPU laufend wesentlich weitergehende Möglichkeiten bieten Geometriedaten zu verändern (siehe Kapitel 2.5).

2.4 Transform Feedback

2.4.1 Status der Erweiterung

Transform Feedback stand zum Beginn dieser Arbeit als proprietäre Erweiterung des Grafikkarten Herstellers Nvidia zur Verfügung und ist in der OpenGL-Extension-Registry [ER08] als Extension mit der Nummer 341 GL_NV_transform_feedback [LBW08b] verzeichnet. Diese Arbeit bezieht sich auf die genannte Extension.

Seit einiger Zeit liegt Transform Feedback auch als Hersteller übergreifende Erweiterung vor. Die Änderungen gegenüber der Nvidia Extension werden in der EXT_transform_feedback im Abschnitt „Interactions with NV_transform_feedback“ erläutert. Die neue Extension [LBW08a] ist in der Registry unter der Nummer 352 zu finden und wurde in wenigen Punkten gegenüber der alten Version von Nvidia vereinfacht. Das allgemeine Verhalten der Erweiterung hat sich indes nicht geändert. Wenn im folgenden Bezeichner wie Enums und Funktionsnamen auf NV enden (für die Nvidia

Extension), so ist davon auszugehen, dass die selbe Funktionalität gewährleistet ist, wenn die Funktionen und Enums mit Suffix EXT benutzt werden.

2.4.2 Funktionalität

Neue Anwendungen lassen sich realisieren, wenn Geometrie-Shader zusammen mit Transform Feedback eingesetzt werden. Transform Feedback ermöglicht es Daten aufzuzeichnen, die durch die Pipeline gehen. Dies geschieht jeweils vor dem Clipping und funktioniert mit der fixed-function Pipeline sowie wenn Shader aktiv sind.

Daten die OpenGL im Transform Feedback Modus aufzeichnet, können in einem oder in mehreren Puffern im Speicher der Grafikkarte abgelegt werden. Soll ein Puffer für alle aufzuzeichnenden Vertexattribute verwendet werden, so werden die verschiedenen Vertexattribute interleaved abspeichert. Sollen mehrere Puffer Verwendung finden, so wird jedem Attribut ein Puffer zugeordnet.

Transform Feedback ist ein Status der OpenGL-Pipeline. Ist es aktiv, werden alle selektierten Daten in einen Puffer geschrieben. Optional kann die weitere Verarbeitung abgeschaltet werden, so dass die Daten nicht zur Anzeige gelangen. Dies geschieht, indem die Daten verworfen und damit in der Pipeline nicht weiter verarbeitet werden.

Da die Daten als ein Array von Vertexattributen (siehe Kapitel 2.2) direkt im Speicher des OpenGL-Servers aufgezeichnet werden, lassen sie sich ohne aufwändiges Kopieren durch die CPU direkt im GL-Server weiterverwenden. So gesehen ist Transform Feedback eine gute Möglichkeit VBOs zu erzeugen.

Wird Transform Feedback verwendet, kann durch *asynchrone Querys* bestimmt werden wie viele Vertices und Primitive in Puffer gestreamt wurden. Sollen die in den Puffer aufgezeichneten Geometriedaten gerendert werden, ist es wichtig zu wissen, wie viele gültige Daten der Puffer enthält. Ansonsten kann es zu Darstellungsfehlern kommen, wenn OpenGL versucht Speicherbereiche als Vertexdaten zu interpretieren, welche ganz andere Daten enthalten.

Zusammen mit GLSL-Shadern (siehe Kapitel 2.3) kann Transform Feedback wie folgt verwendet werden:

- Vorbereitung
  - Zunächst wird ein VBO angelegt, in welches die noch auszuwählenden Daten gestreamt werden sollen.
  - Die Positionen der aufzuzeichnenden varying-Variablen im Programmobjekt werden ermittelt (`glGetVaryingLocationNV`).
  - Für das aktive Programmobjekt wird festgelegt, welche Variablen aufgezeichnet werden sollen (`glTransformFeedback-`

VaryingsNV). Dazu werden die Namen der aufzuzeichnenden aktiven Varyings benötigt. Hier wird der zu verwendende Puffermodus festgelegt (interleaved oder separated).

- Aufzeichnung starten
  - Der als Ziel für Transform Feedback angelegte Vertexpuffer wird an den Binding-Point GL_TRANSFORM_FEEDBACK_BUFFER_NV gebunden.
  - Transform Feedback wird gestartet (glBeginTransformFeedbackNV). Dabei muss angegeben werden welcher Primitivtyp verwendet wird.
  - Optional kann die weitere Verarbeitung der Daten abgeschaltet werden (GL_RASTERIZER_DISCARD_NV).
- Eingabedaten in die Pipeline schicken (Objekte zeichnen).

Solange die Aufzeichnung mittels Transform Feedback aktiv ist kann die Query GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV ausgeführt werden, welche die Anzahl der aufgezeichneten Primitive ermittelt. Die Query ist zu beenden bevor der Transform Feedback Modus beendet wird.

Die Gesamtzahl der Komponenten der Vertexattribute, die maximal in einen Puffer gestreamt werden können, ist begrenzt. So können zum Beispiel auf einer GeForce 8800 GTX mit Treibern der Version 169.21 im interleaved-Modus maximal 64 Komponenten aufgezeichnet werden, also z.B. 16 volle Vektoren mit X, Y, Z und W Komponente. Werden die Attribute in getrennte Puffer gespeichert, gibt es andere Einschränkungen. Zunächst kann auf genannter Hardware in maximal 4 Pufferspeicher gleichzeitig geschrieben werden. Dabei darf jedes zu speichernde Attribut maximal 4 Komponenten besitzen. Sehr wahrscheinlich ändern sich diese Werte je nach verwendeter Hardware und Treiberversion.

Mit Transform Feedback lassen sich also gezielt einzelne Attribute der Pipeline bzw. varying-Variablen der Vertex- oder Geometrie-Shader zur weiteren Verwendung auf der Grafikhardware in VBOs aufzeichnen.

2.5 Geometrie-Shader

Geometrie-Shader können als herstellerübergreifende Erweiterung – wie sie im Treiber von Nvidias GeForce 8 Reihe enthalten ist – verwendet werden. In dieser Arbeit liegt das Augenmerk auf der Erweiterung Nummer 324 GL_EXT_geometry_shader4 [BL08].

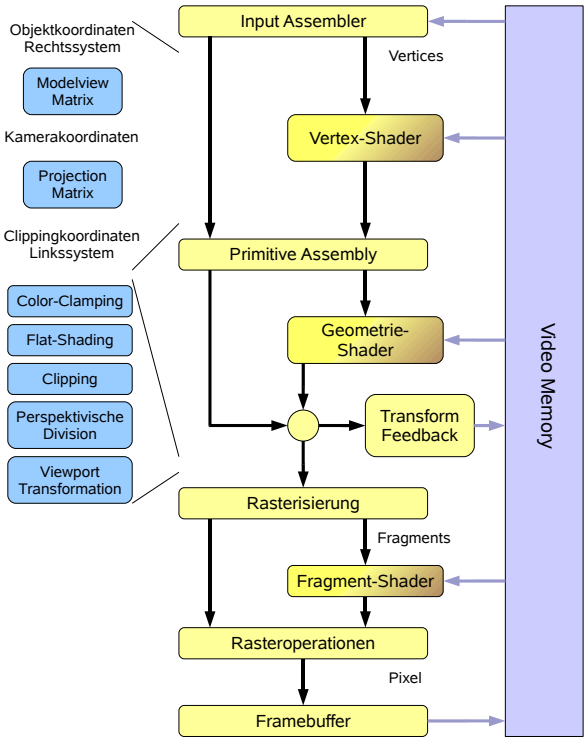

Abbildung 2: OpenGL-Pipeline mit Geometrie-Shader Einheit

2.5.1 Einordnung in die Pipeline

Geometrie-Shader werden in der OpenGL-Pipeline praktisch direkt nach den Vertex-Shadern aber vor Color-Clamping, Flat-Shading, Clipping, perspektivischer Division und Viewport-Transformation ausgeführt (siehe Abb. 2). Daher arbeiten Geometrie-Shader normalerweise auf transformierten Vertexdaten – und damit in einem Linkssystem – was zu beachten ist, wenn weitere Transformationen der Vertexdaten durchgeführt werden sollen. Da nach der `ModelViewProjection`-Transformation kein orthogonales Koordinatensystem mehr gegeben ist, sind Veränderungen an Geometriedaten mitunter schwerer vorstellbar. In Kapitel 3.4.1 wird dargestellt, was zu tun ist, um im Geometrie-Shader auf nicht transformierten Daten arbeiten zu können.

2.5.2 Funktionalität

Im Gegensatz zu den schon länger bekannten Shaderarten wie Vertex- und Fragment-Shadern, arbeitet der Geometrie-Shader nicht auf Vertices oder Pixeln, bzw. Fragments, sondern auf ganzen Primitiven. Der Geometrie-Shader bekommt von der Primitive Assembly ein Primitiv übergeben und

kann ein oder mehrere Primitve oder auch gar kein Primitiv ausgeben. Damit entsteht die neue Möglichkeit, Primitive innerhalb der Pipeline von Shaderprogrammen erzeugen zu lassen bzw. diese zu löschen.

Die Ein- und Ausgabeprimitive des Geometrie-Shaders sind dabei festgelegt. Die Erweiterung `GL_EXT_geometry_shader4` unterstützt als Eingabeprimitive u.a. Punkte, Linien und Dreiecke. Auf die einzelnen Vertices der Primitive kann im Geometrie-Shader frei zugegriffen werden. Auch muss ein Eingabeprimitiv nicht zwingend wieder ausgegeben werden, sondern kann abhängig von Berechnungen, die im Geometrie-Shader durchgeführt werden, oder aufgrund von Daten, die dem Shader übergeben werden, verworfen werden. In diesem Fall wird für das betreffende Eingabeprimitiv zur Laufzeit der Shaders einfach kein entsprechendes Ausgabeprimitiv erzeugt.

Zusätzlich zu den erwähnten Eingabeprimitiven kann der Geometrie-Shader auf Primitiven mit Nachbarschaften arbeiten, die in der Erweiterung neu definiert wurden. Es wurden vier weitere Primitivtypen spezifiziert:

- Linien mit Nachbarschaften
- Linienzüge mit Nachbarschaften
- (einzelne) Dreiecke mit Nachbarschaften
- Dreiecksnetze mit Nachbarschaften

Die Nachbarschaften müssen im OpenGL-Programm explizit angegeben werden und sind nicht über herkömmliche Primitive realisiert. Die Nachbarschaftsinformation steht somit nur dem Geometrie-Shader zur Verfügung und evtl. zukünftigen oder bestehenden Funktionseinheiten, die für die Verarbeitung der Nachbarschaftinformation ausgelegt sind. Für den Rest der Pipeline erscheinen die Primitive ohne Nachbarschaftsinformation.

Da beim Geometrie-Shader die Ein- und Ausgabeprimitivtypen vor dem Linken des Programmobjekts festgelegt werden, tritt ein Fehler auf, wenn versucht wird bei aktiviertem Shader andere als die angegebenen Primitivtypen zu rendern.

Die Ein- und Ausgabeprimitivtypen sind beim Geometrie-Shader voneinander unabhängig. So kann der Shader als Eingabe z.B. Punkte erhalten und ein oder mehrere Dreiecke ausgeben. Primitive, die den Geometrie-Shader verlassen, durchlaufen das Clipping und erscheinen allen weiteren Funktionseinheiten wie gewöhnliche Primitive. Standardmäßig erwartet ein Geometrie-Shader einzelne Dreiecke als Eingabe und gibt Dreiecksnetze aus. Die Primitivtypen sind dem Programmobjekt als Programmparameter zu übergeben.

Einem GLSL-Shaderprogrammobjekt, das einen Geometrie-Shader enthält, muss auch vor dem Linken mitgeteilt werden, wie viele Ausgabeprimitive pro Eingabeprimitiv vom Shader erzeugt werden. Dies geschieht über den Programmparameter `GEOMETRY_VERTICES_OUT_EXT` der bei jedem Programmobjekt mit Geometrie-Shader zu setzen ist, da das Shader Programmobjekt sonst nicht linkt. Alle über den genannten Wert im Shader erzeugten Vertices und Primitive werden von OpenGL ignoriert.

Das Eingabeprimitiv wird vom Geometrie-Shader nach dessen Ausführung verworfen. Im folgenden Listing ist ein Geometrie-Shader zu sehen, der nichts weiter tut, als die Eingabevertices auch wieder auszugeben.

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main(void) {
   for (int i=0; i<gl_VerticesIn; i++) {
      gl_Position = gl_PositionIn[i];
      EmitVertex();
   }
   EndPrimitive();
}
```

Zu beachten ist hierbei, dass die vordefinierten varying-Variablen `gl_Position` und `gl_PositionIn[]` verwendet werden. Dabei dient `gl_Position` als Ausgabevariable und `gl_PositionIn[]` als Eingabevariable. Da der Geometrie-Shader auf Primitiven und nicht auf einzelnen Vertices arbeitet, entsteht aus der Position, die im Vertex-Shader geschrieben wurde, ein Array von Vertices, das genau so lang ist, wie der Primitivtyp Vertices besitzt. Diesen Wert gibt `gl_VerticesIn` an. Allgemein erhöht sich die Dimension jeder varying-Variable vom Vertex- zum Geometrie-Shader um eins. Aus Texturkoordinaten, die im Vertex-Shader als eindimensionaler Array benutzt werden, wird so ein zweidimensionaler Array im Geometrie-Shader.

Neue Syntax-Elemente sind die Funktionen `EmitVertex` und `EndPrimitive`, welche dazu dienen neue Vertices zu generieren und diese zu Primitiven zusammenzufassen. Über die beiden Kommentare am Beginn des Shaders wird die minimal benötigte GLSL-Version angegeben sowie die Geometrie-Shader Extension aktiviert.

Für benutzerdefinierte varying-Variablen wurden in GLSL neue Schlüsselwörter eingeführt, da im Geometrie-Shader zwischen Eingabe- und Ausgabevariablen unterschieden werden muss. Eingabevariablen müssen – wie schon erwähnt – stets als Array definiert werden. Die Angabe der Arraylänge ist dabei optional. Um Eingabevariablen zu definieren wurde das Schlüsselwort `varying` zu `varying in` erweitert, analog für Ausgabevariablen `varying out` eingeführt.

Eine vordefinierte varying-Variable würde explizit, wie im folgenden Beispiel zu sehen, definiert.

```
varying in gl_PositionIn[];
varying out gl_Position;
```

Ebenfalls neu ist die Primitive-ID, die von der Primitive Assembly erzeugt wird, bevor ein Primitiv dem Geometrie-Shader zugeführt wird. Lesend und schreibend kann auf die ID mittels `gl_PrimitiveIDIn` und `gl_PrimitiveID` zugegriffen werden. An dieser Stelle ist darauf hinzuweisen, dass `gl_PrimitiveIDIn` kein Array ist, sondern ein Integerwert, der die Primitive-ID speichert. Primitive-IDs werden ab Null gezählt und nach jedem Punkt, jeder Line bzw. jedem Dreieck, das den Geometrie-Shader durchläuft, um Eins erhöht. Die Primitive-ID steht später dem Fragment-Shader für weitere Berechnungen zur Verfügung. Primitive-IDs können verwendet werden, um Primitive zu unterscheiden. Dazu kann sie im Geometrie-Shader über die Ausgabevariable `gl_PrimitiveID` einen beliebigen Integer-Wert annehmen und so die Primitive klassifizieren.

Neben der Hauptaufgabe und eigentlichen Innovation der Geometrie-Shader Einheit – nämlich neue Primitive zu erzeugen und bestehende Primitive zu verwerfen – bietet der Geometrie-Shader auch Funktionen für das Layered Rendering. Im Geometrie-Shader kann in mehrere Layer einer Cube Map, dreidimensionaler Texturen, sowie ein- und zweidimensionale Texture-Arrays gerendert werden. Diese Möglichkeit kann im Rahmen dieser Arbeit aber nur erwähnt und nicht erschöpfend dargestellt werden.

2.5.3 Einschränkungen

Neben der maximalen Anzahl der Vertices, die in einem Geometrie-Shader Durchlauf erzeugt werden können, existieren weitere Einschränkungen. Die Anzahl der varying-Variablen die im Geometrie-Shader verwendet werden können ist durch die maximale Komponentenanzahl limitiert, also durch die Summe aller Komponenten varying-Variablen. Das gilt für Eingabe- wie Ausgabevariablen. Wie üblich sind die Grenzwerte abzufragen.

Auch ist der Texturzugriff im Geometrie-Shader durch die Anzahl der zur Verfügung stehenden Textureinheiten im Geometrie-Shader beschränkt. Ob Texturzugriff im Geometrie-Shader überhaupt möglich ist, ist abhängig von der GL-Implementierung.

Da die varying-Variablen zwischen den Vertices eines Primitives normalerweise interpoliert werden und dazu in der Grafikhardware Funktionseinheiten existieren, ist auch beim Geometrie-Shader die Anzahl der interpolierbaren Komponenten der varying-Variablen beschränkt. Auch hier zählt wieder die Komponentensumme der aktiven varying-Variablen.

Ein wichtiger beschränkender Wert, der hier noch genannt werden soll, ist `MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS_EXT`. Er ist definiert als Produkt aller im Geometrie-Shader ausgegebener Vertices und der Komponentensumme aller aktiven varying-Variablen. Auf der zu Verfügung stehenden GeForce 8800 GTX mit bereits genannter Treiberversion (siehe Kapitel 2.4.2 auf Seite 7) liegt auch hier das Limit bei 1024. Soll im Geometrie-Shader ein `Vec4` verarbeitet werden liegt dieser als `varying in` und `-out` vor. Es wird vermutet dass diese 8 Komponenten das Vertex-Limit der genannten Definition folgend auf 128 verringert. Bedenkt man die große Anzahl vordefinierter varying-Variablen, ist es verständlich, dass der Geometrie-Shader nicht beliebig viele Primitive pro Eingabeprimitv erzeugen kann. Würden aus allen Primitiven einer Szene wirklich 100 bis 1000 neue erzeugt, kommt die Hardware an ihre Grenzen. So stören die aufgeführten Beschränkungen auch in der ersten Generation Geometrie-Shader fähiger Hardware nicht wirklich. Durch geeignete multi-pass Verfahren lassen sich auch größere Mengen an Geometriedaten prozedural auf der GPU erzeugen, was in Kapitel 3.3 gezeigt wird.

3 Umsetzung

Im folgenden Kapitel wird beschrieben, wie das Programm geomShader-Demo realisiert wurde – eine Beispielapplikation die Geometrie-Shader einsetzt. Zunächst wird ein grober Überblick über die Softwarearchitektur des Programms gegeben. Es folgt eine Beschreibung des gewöhnlichen Programmablaufs. Schließlich sollen noch die umgesetzten Shader Konzepte vorgestellt und die Shader dargestellt werden.

3.1 Softwarearchitektur

Die Applikation geomShaderDemo wurde als Windows Anwendung unter Verwendung von MS Visual C++ und OpenGL erstellt. GLSL wird als hersteller- und plattformunabhängige Shaderhochsprache verwendet. Die zur Anbindung der GLSL-Shader implementierten Klassen werden im folgenden Kapitel genauer betrachtet.

3.1.1 Shader

Die Klasse Shader und die davon abgeleiteten Klassen `Vertex-`, `Geometry-` und `FragmentShader` (siehe Abb. 3) bieten die Funktionalität den Quellcode eines entsprechenden GLSL-Shaders aus einer Datei zu laden und von OpenGL daraus ein Shaderobjekt anlegen zu lassen. Dazu wird dem Konstruktor einer der abgeleiteten Shader Klassen der Pfad zur Shaderdatei übergeben. Es gibt die Möglichkeit das Handle des ShaderObjects abzufragen und eine Methode die für Debuging-Zwecke den Shader InfoLog ausgibt. Dabei wird angegeben um welchen Shader Typ es sich handelt. Die abgeleiteten Klassen unterscheiden sich nur im Shadertyp und benutzen die Methoden der Basisklasse, um den Shaderquelltext aus der Datei zu lesen, auf die Grafikkarte hochzuladen und ihn zu einem Shader-Object zu kompilieren.

3.1.2 Geometrieklassen

Die von der Klasse Geometry abgeleiteten Klassen Teapot, Tree und Tetraeder (siehe Abb. 4) dienen der Applikation zur Darstellung von Testobjekten für die Shader. Sie sind sehr einfach gehalten und verwalten keine Geometriedaten, sondern zeichnen diese vielmehr direkt mit OpenGL-Befehlen.

Die verschiedenen Geometriedaten dienen als Eingabewerte für die GLSL-Shader, die in Kapitel 3.4 detaillierter beschrieben werden.

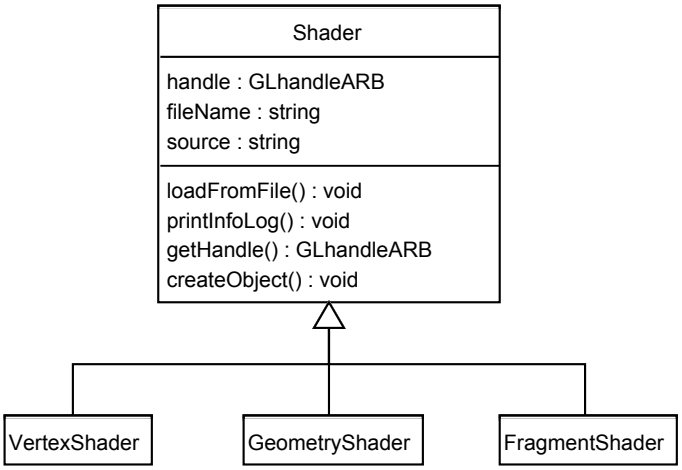

Abbildung 3: Shader Klassen

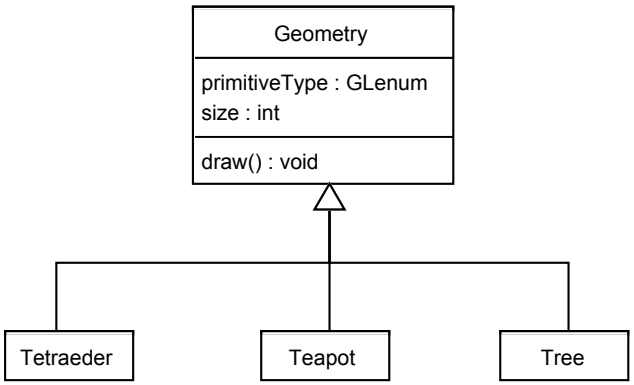

Abbildung 4: Klassen zur Darstellung von 3D-Objekten

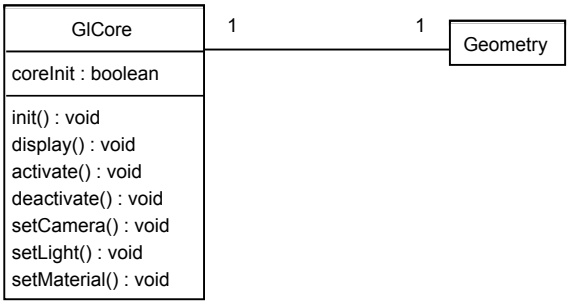

Abbildung 5: Basisklasse `GlCore`

3.1.3 Views

Um die verschiedenen Beispiel-Szenen bzw. verschiedene Shader gut in einer Applikation unterbringen zu können, wurden Klassen erstellt, die einen oder mehrere Shader sowie die dazugehörige Geometrie als eine `View` kapseln. Alle View-Klassen sind dabei von `GlCore` (Abb. 5) abgeleitet.

`GlCore` initialisiert den OpenGL-Kontext, setzt Licht, Kamera und ein Standardmaterial für die Anwendung. Jede View hat ihre eigene `init`-Methode. Da das allgemeine Setup des OpenGL-Kontexts in der Klasse `GlCore::init` geschieht, rufen alle davon abgeleiteten Klassen in ihrer `init`-Methode zunächst die Initialisierung von `GlCore` auf. Neben der Initialisierung bietet jede View eine `display`-Methode, die die Geometrie der View darstellt. Um zwischen den Views wechseln zu können, existieren pro View `activate`- und `deactivate`-Methoden. Diese stellen sicher, dass notwendige Shader ein- und ausgeschaltet werden. Desweiteren erben alle Views von `GlCore` Methoden, um Kamera, Licht und Material zu verändern.

Herkömmliche Shader Von `GlCore` abgeleitet ist u.a. die Klasse `ShaderView` (siehe Abb. 6). Diese soll nun genauer betrachtet werden.

`ShaderView` ist die einfachste View-Klasse, die implementiert wurde. Wie alle folgenden Views besitzt sie ein Handle für das Programmobjekt des Shaders, der in der View verwendet werden soll. Die `ShaderView` erstellt in ihrer `init`-Methode durch den Aufruf von `loadShader` das Shaderprogrammobjekt. Dazu wird ausgehend vom Shader – der durch einen Namen identifiziert wird – die benötigten Instanzen der Klasse `VertexShader` und `FragmentShader` erstellt. `ShaderView` untersützt nur herkömmliche Shader ohne Geometrie-Shader. Sind diese geladen und kompiliert, werden sie zu einem Shaderprogrammobjekt zusammengefügt

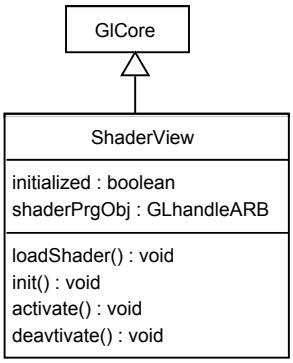

Abbildung 6: Eine View die Vertex- und Fragment-Shader lädt

(gelinkt) und im Speicher der GPU abgelegt. Identifiziert über ein Handle – welches die View-Klasse speichert – lässt sich der Shader nun aktivieren.

Typischerweise werden die Views wie folgt benutzt; dies wird teilweise durch Fehlermeldungen bei falscher Verwendung erzwungen:

- `init`
- `activate`
- `display`
- `deactivate`

Eine bereits aktivierte View muss dabei nicht für jeden display Aufruf erneut aktiviert werden, da activate nur den Shader einschaltet. Sobald der Shader aktiv ist, kann das Geometry-Objekt gezeichnet werden, welches der View über den Konstruktor übergeben wurde. Einfache Shader lassen sich ganz gut mit der `ShaderView` Klasse testen.

Geometrie-Shader Von der Klasse `ShaderView` erben zumindest indirekt alle weiteren Shader-Klassen. Die grundlegende Funktionalität der Shader-Klassen wird schon von `ShaderView` abgedeckt. Die Spezialisierung `GeomShaderView` für den Einsatz von GLSL Geometrie-Shadern wird erreicht, indem die Methode `loadshader` überschrieben wird. Dabei wird zusätzlich zu den anderen beiden Shader Instanzen ein Objekt der Klasse `GeometryShader` angelegt. Eine entsprechende Quelltextdatei muss auf der Festplatte vorliegen, damit das um den Geometrie-Shader erweiterte Programmobjekt erfolgreich gelinkt werden kann. Wie in den Grundlagen in Kapitel 2.5.2 auf Seite 9 schon erwähnt, werden die benötigten Parameter für den Geometrie-Shader gesetzt. Diese sind Eingabe- und

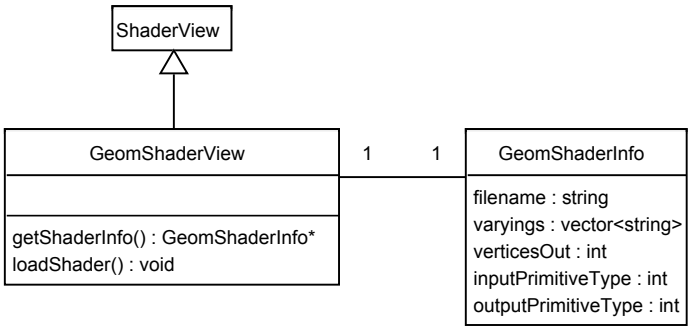

Abbildung 7: `GeomShaderView` mit ShaderInfo

Ausgabeprimitivtyp, sowie die maximale Anzahl an Primitiven, die pro Durchlauf aus einem Eingabeprimitiv durch den Shader erzeugt werden.

Die View mit Geometrie-Shader Unterstützung wurde um eine *ShaderInfo*-Klasse erweitert (siehe Abb. 7). Diese kapselt alle Daten, die erforderlich sind, um ein Shaderprogrammobjekt mit Geometrie-Shadern erfolgreich in Betrieb zu nehmen. Die Daten wurden für die Verwendung der View in einer Transform-Feedback-View ausgelagert. Die Aufgabe des ShaderInfo-Objekts wird deutlich in Kapitel 3.2.3. Dort wird das Erstellen der Views beschrieben.

Transform Feedback Auch Transform Feedback wurde als View realisiert. Die `FeedbackView` ist, wie die `ShaderView`, direkt von `GlCore` abgeleitet und ist die komplexeste View, die für die Applikation geomShaderDemo realisiert wurde. Die Grundidee der `FeedbackView` ist eine `ShaderView` mehrfach im Feedback Loop zu verwenden, wie dies im Nvidia OpenGL SDK 10 Beispiel „Transform Feedback Fractal“ [sdk08] geschieht. Es wird ein Shader auf Eingabedaten ausgeführt und die Ausgabedaten mittels Transform Feedback gespeichert. Die Ausgabedaten des ersten Duchgangs dienen dann als Eingabedaten des nächsten. So wird ein einfacher Feedback Loop realisiert.

Die `FeedbackView` wurde, ausgehend von einem ersten Entwurf, dahingehend angepasst, den Geometrie-Shader einer `GeomShaderView` mehrfach auf die Geometriedaten seiner View anzuwenden. Die zunehmende Komplexität des Datenflusses durch die OpenGL-Pipeline hat dazu geführt, dass die `FeedbackView` zwei weitere Views benötigt, um ihre Aufgabe zu erfüllen. Einmal eine View deren Geometrie-Shader mehrfach ausgeführt werden soll und dann noch eine View, um die Ergebnisse darzustellen. Dazu genügt eine einfache `ShaderView`. Für die mehrfache Anwendung eines Geometrie-Shaders wurde schließlich eine eigene View von `GeomShaderView` und eine erweiterte ShaderInfo-Klasse von

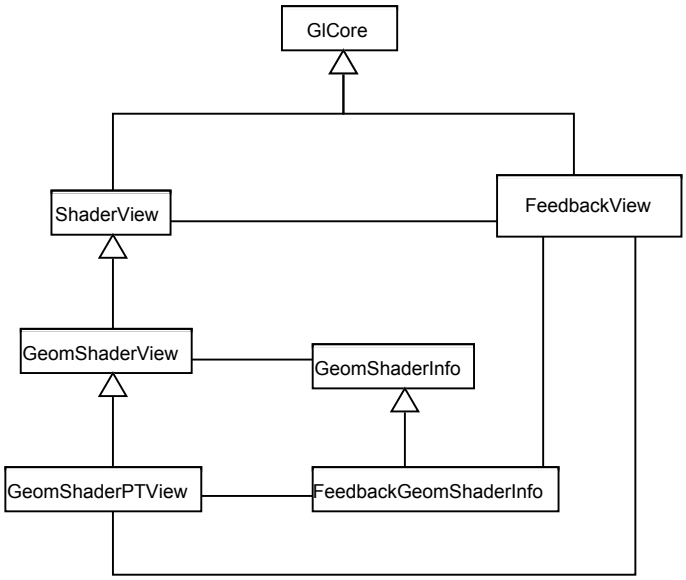

Abbildung 8: Zusammenhänge `FeedbackView`

`GeomShaderInfo` abgeleitet. Abbildung 8 zeigt vereinfacht wie die Views rund um die `FeedbackView` zusammenhängen und in der Beispielanwendung verwendet werden. Die genaue Funktionsweise der `FeedbackView` wird im Kapitel 3.3 deutlich.

3.1.4 Grafische Benutzungsoberfläche

Um die Anwendung ansprechender und leichter bedienbar zu machen wurde im Rahmen dieser Arbeit ein GUI mit dem cross-plattform Toolkit FLTK [flt08] realisiert. FLTK bietet sich an, weil es eine vergleichsweise geringe Komplexität besitzt und eine gute OpenGL Anbindung. So lassen sich zum Beispiel bestehende GLUT Programme Dank des in FLTK vorhandenen GLUT-Emulation leicht mit einem GUI versehen. Da die FLTK Bibliothek ziemlich schlank ist, wird sie normalerweise statisch gelinkt.

FLTK liefert einen grafischen Editor zur Erstellung der Benutzungsoberfläche mit, genannt *Fluid* (Abb. 9). Aus den Fluid Dateien erstellt Fluid kompilierbaren C++ Code, der sich direkt in den Build-Prozess integrieren lässt. Es existieren verschiedene Arbeiten, die den Umgang mit Fluid erleichtern. Im Rahmen der FLTK Dokumentation [SEMS08] wird mit FLUID ein kleines GUI erstellt. Auch enthält der Quellcode von FLTK viele kleine Beispiel-Anwendungen. Bei einigen wird Fluid verwendet.

Die Benutzungsoberfläche für die in dieser Arbeit erstellte Anwendung baut zu großen Teilen auf dem in der FLTK Distribution enthaltenen

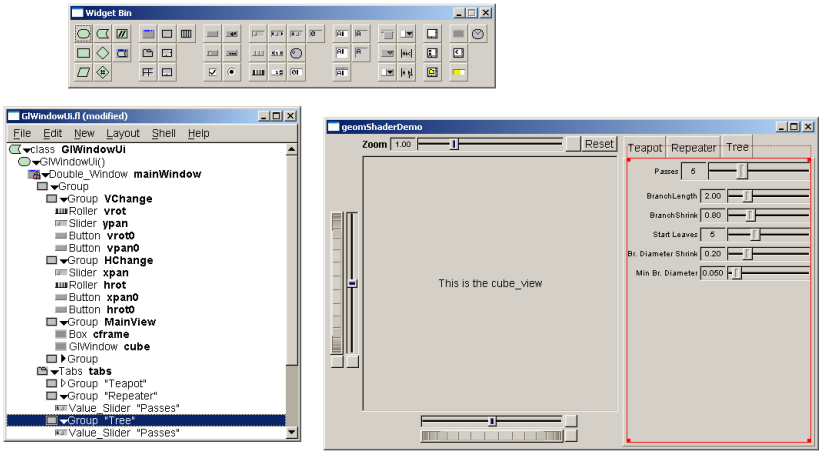

Abbildung 9: FLTK GUI-Designer Fluid

Beispiel *cubeview* auf. Das Konzept der cubeview ist, um einen OpenGL-Kontext herum ein GUI aufzubauen, über das die Szene im GL-Fenster manipuliert werden kann, indem eine Instanz des benutzerdefinierten GL-Fensters im GUI instanziert wird.

Die Interaktion zwischen GUI und Anwendung erleichtert Fluid, indem es automatisch callback-Funktionen erzeugt, die Benutzerfunktionen aufrufen. Von dort sind die Methoden aus der eigenen Klassenhierarchie aufrufbar.

Dem Konzept der cubeview Anwendung folgend, kann eine OpenGL-Anwendung leicht mit einer Benutzungsoberfläche versehen werden, wobei GUI-Logik und Anwendungslogik getrennt bleiben und über klar definierte Schnittstellen kommunizieren.

Über die in Abbildung 9 gezeigten Regler lässt sich die dargestellte Geometrie in Position, Rotation und Größe verändern, damit die Ergebnisse der Shader genau betrachtet werden können. Die Parameter der einzelnen Shader sind in Registerkarten rechts neben dem GL-Kontext untergebracht. Über die Registerkarten schaltet man auch zwischen den in die Applikation integrierten Views um. Die Änderung der Shader- bzw. Geometrieparameter auf der rechten Seite lassen sich in Echtzeit im GL-Kontext nachvollziehen.

In der Beispielanwendung erzeugt die Klasse `GlWindowUi` die grafische Benutzungsoberfläche. In `GlWindowUi` wird die Anwendung selbst instanziert, die von der Klasse `GlWindow` gekapselt wird. `GlWindow` enthält die Views und die Geometriedaten. Gesteuert wird geomShaderDemo über `GlWindowUi`, welches dazu viele Widgets enthält (siehe Abb. 10).

`GlWindow` enthält einen STL `Vector` von Views und einen `Vector` von Objekten der Klasse `Geometry`. Sie kümmert sich um die Konstrukti-

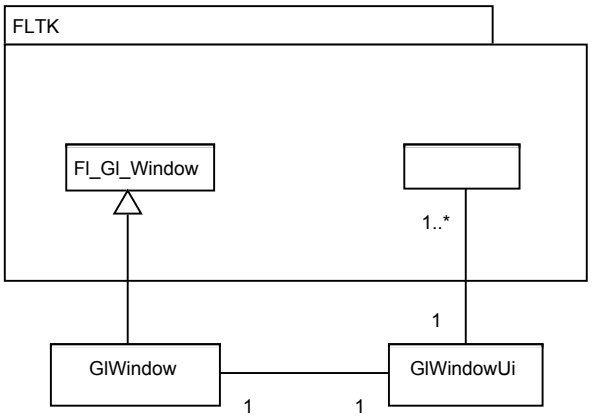

Abbildung 10: Anbindung von FLTK

on der Views sowie deren Initialisierung. Hier werden auch beim Umschalten zwischen den verschiednenen Views die Shader an und abgeschaltet. Dies geschieht über die bereits erwähnten `activate` und `deactivate` Methoden die alle Views von `GlCore` erben. Wie die verschiedenen Klassen zusammenhängen wird auch im folgenden Kapitel verdeutlicht, das den Programmablauf der Beispielanwendung nachzeichnet.

3.2 Programmablauf

Das folgende Kapitel veranschaulicht den Aufbau der Anwendung durch die Beschreibung des Ablaufs bei ihrer Ausführung. Während das vorherige Kapitel 3.1 die Architektur der Applikation grob beschrieben hat, um einen Überblick zu verschaffen, beschreibt das aktuelle Kapitel den Ablauf und die Architektur noch ausführlicher, bevor im darauf folgenden Kapitel die dazugehörigen Shader (3.4) beschrieben werden.

3.2.1 Benutzungsoberfläche und GL-Kontext

In der `main`-Funktion wird die Klasse `GlWindowUi` instanziert, die wiederum im Konstruktor alle GUI-Elemente anlegt und damit auch den durch `GlWindow` repräsentierten OpenGL-Kontext der Anwendung. Der Konstruktor initialisiert neben den Attributen der Klasse die Variablen, die für die Anbindung der FLTK Widgets benötigt werden. Dies sind unter anderem Variablen für horizontale und vertikale Position und Rotationswinkel der Geometrie, sowie einen Geometriegrößenfaktor. Nach der Initialisierung enthalten die genannten Variablen die Werte, die auch das GUI anfänglich anzeigt. Nachfolgende Objekte werden soweit nicht anders erwähnt im Konstruktor von `GlWindow` angelegt.

3.2.2 Geometrie anlegen

Als nächstes wird in einen STL-`Vector` die der Anwendung zur Verfügung stehende Geometrie eingefügt, indem Instanzen der Klassen `Teapot`, `Tetraeder` und `Tree` gebildet werden. Die ersten beiden benutzen in ihrer `draw`-Methode die entsprechenden GLUT-Methoden, um die Geometrie zu zeichnen. Die Klasse `Tree` hingegen zeichnet lediglich ein einzelnes Dreieck, welches die Komponenten der Vertexattribute Position, Normale und Farbe benutzt, um die Parameter für den zu erstellenden Baum zu speichern. Die einzelnen Parameter des Baumes liegen als Attribute der Klasse vor und können verändert werden.

3.2.3 Views erstellen

Nach der Konstruktion der Geometrie werden die Views angelegt. Wie auch die verschiedenen Geometrieklassen werden die Views in einem STL-`Vector` gespeichert. Eine View ganz ohne Shader wird durch Instanzierung von `GlCore` erstellt und erhält dabei als Geometrie einen `Teapot`, der in dieser Beispielapplikation nicht fehlen darf. Die restlichen Views sind komplexer und für den Einsatz von Transform Feedback und Geometrie-Shader ausgelegt.

Für Transform Feedback muss ein Puffermodus festgelegt werden. Diese Arbeit beschränkt sich darauf Transform Feedback im interleaved-Modus zu verwenden. Der im Kapitel 2.4 schon erwähnte separated-Puffermodus wird in der Beispielapplikation nicht eingesetzt. Um die für Transform Feedback ausgewählten Attribute speichern zu können, ist neben dem gewählten Modus auch ein Format für den Puffer auszuwählen. Die zur Verfügung stehenden Formate können dabei in [SA06] auf Seite 32 nachgeschlagen werden.

„Repeater“ Für die im GUI „Repeater“ genannte View, werden die Normalen und die Vertices per Transform Feedback aufgezeichnet. Beide können gemeinsam in ein VBO des Formates `GL_N3F_V3F` abgelegt werden, die 3 komponentigen Normalen zuerst und die ebenfalls 3 komponentigen Vertices danach. Wie in Kapitel 2.2 beschrieben, kann ein VBO im interleaved-Format direkt gerendert werden.

Bevor die View selbst angelegt wird, wird im Konstruktor von `GlWindow` das ShaderInfo-Objekt (genauer `FeedbackGeomShaderInfo`) angelegt, das die Metadaten der View aufnimmt. Neben dem Pufferformat für Transform Feedback speichert es den Namen des zu verwendenden Shaders. Zusätzlich müssen die varying-Variablen bezeichnet werden, die nach der Geometrie-Shader Einheit per Transform Feedback aufgezeichnet werden sollen. Damit sind die ShaderInfo-Daten vollständig und die View selbst kann erstellt werden.

```
view.push_back(
  new FeedbackView(
    new GeomShaderPTView(
      geometry.at(1),
      (GeomShaderInfo*)gsi ),
    new ShaderView(NULL, "phong")
  )
);
```

In den STL-`Vector` `view` wird dann die `FeedbackView` eingefügt. Als im Feedback-Loop mehrfach auszuführende View wird ihr im Konstruktor eine `GeomShaderPTView` übergeben. Die `GeomShaderPTView` erhält ihre Geometrie und das ShaderInfo-Objekt. Schließlich erwartet die `FeedbackView` im Konstruktor neben der View, die mehrfach ausgeführt werden soll, eine weitere View, die nach dem Feedback-Loop die erzeugte Geometrie rendert. Hierzu wird eine `ShaderView` ohne Geometriedaten und ein einfacher Phong-Shader verwendet. Die Shader der „Repeater"-View werden in Kapitel 3.4.1 ausführlicher beschrieben.

„Tree" Schließlich wird die komplexeste View der Anwendung erstellt. Diese wird prozedural Geometrie per Shader erzeugen mit Hilfe eines Feedback-Loops. Da dazu mehr Informationen verwendet werden als in der vorherigen View, wird auch ein komplexeres Pufferformat (`GL_C4F_N3F_V3F`) für das Feedback benötigt. Neben Normalen und Positionen (je 3 komponentig `float`) wird zusätzlich die Farbe aufgezeichnet die 4 Komponenten umfasst. Es stehen also für die Erzeugung der Geometrie im Shader insgesamt 12 `float` Werte als Parameter zur Verfügung. Das ShaderInfo-Objekt – welches wieder vor der View selbst erstellt wird – enthält die drei Namen der varying-Variablen, die nach der Geometrie-Shader Einheit aufgezeichnet werden sollen.

Die als „Tree" bezeichnete View ist eine `FeedbackView`. Sie enthält eine `GeomShaderPTView` mit Geometriedaten der Klasse `Tree` sowie das erwähnte ShaderInfo-Objekt. Zur Darstellung wird ihr zusätzlich eine `GeomShaderView` mit angepassten Phong-Shader übergeben.

```
view.push_back(
  new FeedbackView(
    new GeomShaderPTView(
      geometry.at(2),
      (GeomShaderInfo*)treeGsi
    ),
    new GeomShaderView(
      NULL,
      new GeomShaderInfo("treePhong2")
```

```
    )
  )
);
```

Bevor die `FeedbackView` im STL-`Vector` abgelegt werden kann, müssen die beiden ShaderInfo-Objekte (`treeGsi` und die anonyme `GeomShaderInfo`), die Geometrie-Objekte, sowie die beiden von der `FeedbackView` benötigten Views (`GeomShaderPTView` und `GeomShaderView`) angelegt sein. Der „Tree"-Shader selbst wird in Kapitel 3.4.2 beschrieben.

Im Konstruktor von `GlWindow` wurden also die gewünschten Views erzeugt. Die eigentliche Initialisierung ist getrennt von der Konstruktion des Objektes und wird erst ausgeführt wenn der OpenGL-Kontext bereit ist.

3.2.4 Views initialisieren

Nach dem Erzeugen von `GlWindowUi` und dem damit verbundenen Anlegen des OpenGL-Kontexts durch `GlWindow`, wodurch die Geometrie- und die View-Objekte erstellt werden, können die Views initialisiert werden. Dies geschieht nachdem in der `main`-Funktion das GUI sichtbar geschaltet und der FLTK mainloop betreten wurde. Zeichnet FLTK das GUI zum ersten Mal, ist auch der OpenGL-Kontext bereit initialisiert zu werden. Die FLTK-Dokumentation empfiehlt in der `draw`-Methode des GL-Fensters zu Überprüfen, ob der OpenGL-Kontext gültig ist und diesen falls nötig zu initialisieren. Da bei Initialisierung der Views `GlCore::init` ausgeführt und damit der OpenGL-Kontext initialisiert wird, können die Views nicht im Konstruktor der View-Objekte initialisiert werden. Leider führt die späte Initialisierung der Views zu einer kurzen Verzögerung bei der erstmaligen Darstellung des GUI.

```
if (!context_valid()) {
  for (unsigned int i=0; i<view.size(); i++) {
    view.at(i)->init();
  }
  //activate current view
  view.at(activeView)->activate();
}
```

Über den STL-`Vector view` werden alle der Applikation verfügbaren Views erstmalig initialisiert. `GlCore::init` erledigt dabei die Einrichtung des OpenGL-Kontexts. `FeedbackView::init` führt die notwendigen Schritte zur Erstellung des Feedback-Loops durch, wie es in Kapitel 3.3.2 noch ausführlicher dargestellt wird.

3.2.5 View darstellen

Nachdem am Anfang von `GlWindow::draw` die Views erstmalig eingerichtet wurden, wird in der `draw`-Methode überprüft, ob das Fenster in seiner Größe verändert wurde und eine entsprechende Anpassung des Viewports notwendig ist.

Anschließend kann die eigentliche Darstellung beginnen. Der Bildspeicher und Tiefenpuffer wird gelöscht. Die aktuelle View kann `setCamera` überschreiben und damit die Position der Kamera verändern, wenn sie in der `draw`-Methode der aktuellen View aufgerufen wird. Nun werden die aktiven Shader vorübergehend deaktiviert, um eine Ebene unter dem Testobjekt der Szene zu zeichnen. Schließlich wird nach Anwendung der über das GUI gesteuerten Verschiebung und Rotation der darzustellenden View-Geometrie die `display`-Methode der aktuellen View nach erneuter Aktivierung ihrer Shader aufgerufen. Im einfachsten Fall wird dann das Geometrie-Objekt der aktuellen View gezeichnet. Wenn eine `FeedbackView` aktiv ist, wird das in dieser Arbeit umgesetzte System für multi-pass Rendering mit Geometrie-Shadern aktiv.

3.3 Multi-pass Geometrie-Shader

Im Folgenden wird das Verhalten der Klasse `FeedbackView` bei Ausführung des Programms beschrieben, da es den Hauptteil der Arbeit ausmacht. Dabei wird im Speziellen auf das multi-pass Rendering mit Geometrie-Shadern eingegangen.

3.3.1 Konstruktor

Es werden die Membervariablen initialisiert. Darunter befindet sich die child-View, die mehrfach ausgeführt werden soll, und die so genannte *simpleView*, die nach dem Feedback-Loop die Daten, auf denen gearbeitet wurde, zur Darstellung aufbereitet.

3.3.2 init-Methode

Die `init`-Methode initialisiert zunächst `GlCore` und anschließend die child-View, welche ihre Shader lädt und linkt. Danach werden die Puffer für das Transform Feedback angelegt (siehe Kapitel 2.4). Diese dienen abwechselnd als Lese- und Schreibpuffer und realisieren damit den Feedback-Loop.

`init` bestimmt zudem die Positionen der varying-Variablen im Shaderprogrammobjekt der child-View, wozu sie deren ShaderInfo-Objekt heranzieht, das die Namen der varying-Variablen enthält, die aufgezeichnet werden sollen. Ihre Positionen werden dabei durch die Funktion `glGetVaryingLocationNV` abgefragt.

3.3.3 activate-Methode

Damit der Feedback-Loop ausgeführt werden kann, muss einer der im Loop verwendeten VBOs mit initialen Geometriedaten gefüllt werden. Das erledigt in der `activate`-Methode die Funktion `initGeometry`. Dazu wird ein einzelner Render-Pass mit aktiviertem Transform Feedback ausgeführt, der den speziell zu diesem Zweck eingeführten pass-through Modus in `GeomShaderPTView` verwendet. Der pass-through Modus der `GeomShaderPTView` besitzt einen zusätzlichen Geometrie-Shader, der dafür zuständig ist, die regulären Attribute wie `glNormal`, `glPosition` usw. in benutzerdefinierte varying-Variablen zu kopieren, die im Feedback-Loop benutzt werden. Dieser Geometrie-Shader wird zusammen mit den anderen Shadertypen zu einem Shaderprogrammobjekt zusammengefügt, das nur in diesem Modus verwendet wird.

Um die Startgeometrie im VBO bereitzustellen werden konkret folgende Schritte in `initGeometry` durchgeführt:

- Der pass-through Modus der child-View wird eingeschaltet.
- Die child-View wird aktiviert, wodurch das Programmobjekt mit pass-through Shader zum Einsatz kommt.
- Die bereits ermittelten varying-Positionen des pass-through Shaders werden Transform Feedback übergeben und damit die entsprechenden varying-Variablen zur Aufzeichnung selektiert.
- Transform Feedback wird gestartet.
- Die `display`-Methode der child-View wird aufgerufen.

Durch den pass-through Modus werden die Zeichenbefehle der child-View nun im richtigen Format für den Feedback-Loop im VBO abgelegt.

Nachdem die Transformation der Eingabedaten bei der Initialisierung des VBOs abgeschlossen wurde, kann Transform Feedback gestoppt werden.

- Transform Feedback wird gestoppt.
- Die Shader der child-View werden deaktiviert.
- Der pass-through Modus der child-View wird abgeschaltet.

Mit Beendigung von Transform Feedback, konnte mit Hilfe der Query (siehe Kapitel 2.4) ermittelt werden, wie viele Primitive durch die child-View erzeugt wurden. Damit ist bekannt, wie viele im nächsten Render-Pass zu zeichnen sind. Abschließend müssen die beiden Puffer getauscht

werden. Der per Transform Feedback gefüllte Puffer wird im ersten Pass des Feedback-Loops als Eingabe verwendet.

Bei der Initialisierung der Eingabedaten für den multi-pass Geometrie-Shader findet eine Transformation der Eingabedaten statt. OpenGL-Standard-Attribute werden benutzt, um die Eingabedaten in den Shader zu bekommen. Diese werden aber an genannter Stelle in ein benutzerdefiniertes Format überführt. Dies beschränkt sich bei den implementierten Shadern auf die Verwendung von eigenen varying-Variablen. Bei der Aufbereitung der Eingabedaten – ausgehend von OpenGL-Zeichenbefehlen bzw. Attribute der verwendeten Eingabeprimitive – ist man durch den extra Render-Pass zur Aufbereitung der Daten nur unwesentlich eingeschränkt. Es wären daher auch komplizierte Umwandlungen der Daten denkbar.

3.3.4 display-Methode

Nachdem die View bei der Initialisierung und bei ihrer Aktivierung vorbereitet wurde, kann sie nun das Ergebnis liefern – Geometriedaten die durch einen multi-pass Ansatz mit Geometrie-Shadern erstellt wurden.

Dazu wird zunächst die child-View aktiviert. Damit sind die Shader der `geomShaderPTView` bereit für den Feedback Loop. Die folgenden Schritte beschreiben den Feedback-Loop und werden so oft wiederholt wie im GUI ausgewählt:

- Transform Feedback wird gestartet, also
  - der Puffer gebunden, der die Transform Feedback Daten aufnimmt (VBO-1),
  - der Transform Feedback Modus aktiviert,
  - die Rasterisierung abgeschaltet
  - und eine Query angelegt, welche die erzeugten Primitive mitzählt.
- Die aktuelle Geometrie gezeichnet, das bedeutet
  - das aktuelle VBO wird als `GL_ARRAY_BUFFER` gebunden (VBO-0),
  - die zum Pufferformat passenden Vertex-Pointer gesetzt,
  - der Puffer über `glDrawArrays` gezeichnet
  - und danach VBO-0 wieder gelöst.
- Transform Feedback wird gestoppt, dabei wird unter anderem
  - die Query ausgewertet
  - und die Rasterisierung wieder eingeschaltet.

- VBO-0 und VBO-1 werden vertauscht.

Danach kann die child-View wieder deaktiviert werden. Damit das Ergebnis des Feedback-Loops wieder eine darstellbare Form erhält – zweckentfremdete Vertexattribute können außergewöhnliche Bilder produzieren, werden die Geometriedaten ohne Korrektur gezeichnet – werden die Daten durch die simpleView wenn nötig umgewandelt und dargestellt. Dazu wird abschließend

- die SimpleView aktiviert,
- der aktuelle Puffer gezeichnet
- und die SimpleView wieder deaktiviert.

Nachdem die Ausgabe der SimpleView nicht per Transform Feedback aufgezeichnet wird, erscheint das Ergebnis der multi-pass Geometrie-Shader jetzt auf dem Bildschirm.

3.4 Shader

Die GLSL-Shader, die im multi-pass Verfahren in der `FeebackView` ihre Anwendung finden, sollen im folgenden Kapitel genauer betrachtet werden. Im Rahmen dieser Arbeit wurden verschiedene Shader implementiert. Es werden jedoch nur die für den Feedback-Loop relevanten Shader ausführlicher dargestellt.

3.4.1 Repeater

Die Grundidee des im GUI „Repeater“ bezeichneten Shaders ist leicht erklärt: Es sollen Dreiecke unterteilt werden. Dabei wird ein Dreieck in jeweils drei neue Dreiecke unterteilt. Dabei sollen die neuen Dreiecke aus der Ebene hinaus wachsen, die von dem ursprünglichen Dreieck aufspannt wird. Die Länge und Richtung in welche die Dreiecke in den Raum wachsen, werden durch die Normale des ursprünglichen Dreiecks definiert.

Der Vertex-Shader ist erdenklich einfach. Er reicht lediglich die Normale weiter, da sie sonst verworfen würde. Die Modelview- und Projection-Transformation der Eingabevertices findet nicht statt. Für die Erzeugung von Geometriedaten ist es vorteilhaft im Weltkoordinatensystem zu bleiben, da es orthogonal ist, die Normalen vorhanden sind und ein Rechtssystem vorliegt. Dies erleichtert auch die mehrfache Ausführung des Geometrie-Shaders, da die `ModelViewProjection`-Transformation sonst ungewollt für jeden Pass erneut durchgeführt werden würde.

```
#version 120
varying vec3 normal;

void main()
{
  normal = gl_Normal;
  gl_Position = gl_Vertex;
}
```

Der dazugehörige Geometrie-Shader nimmt die Normale als Eingabevariable entgegen und besitzt 2 Ausgabevariablen: Normale und Position. Die Funktion `outputTriangle` gibt ein Dreieck aus, inklusive einer für das Dreieck neu berechneten Normale.

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

varying in vec3 normal[];

varying out vec3 oNormal;
varying out vec3 oVert;

void outputTriangle(vec3 a, vec3 b, vec3 c) {

  vec3 norm = normalize( cross(a - b, b - c) );

  oNormal = norm;
  oVert = a;
  EmitVertex();

  oNormal = norm;
  oVert = b;
  EmitVertex();

  oNormal = norm;
  oVert = c;
  EmitVertex();

  EndPrimitive();
}

void main(void)
{
  vec3 a = vec3( gl_PositionIn[0] );
```

```
  vec3 b = vec3( gl_PositionIn[1] );
  vec3 c = vec3( gl_PositionIn[2] );
  vec3 d = ((a+b+c)/3) + normal[0];

  outputTriangle(c, a, d);
  outputTriangle(b, d, a);
  outputTriangle(b, c, d);
}
```

Der zur Initialisierung verwendete pass-through Shader (vergleiche Kapitel 3.3.3) dient dazu, die Eingabevertexattribute auf die im Shader verwendeten benutzerdefinierten Variablen abzubilden. Die in der Anwendung zur Aufzeichnung mit Transform Feedback selektierten varying-Variablen des Shaders lauten oNormal und oVert und bezeichnen die Ausgabevariablen des Geometrie-Shaders, da nur diese aufgezeichnet werden können.

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

varying in vec3 normal[];

varying out vec3 oNormal;
varying out vec3 oVert;

void main(void ) {
  for (int i=0; i<3; i++) {
    oNormal = normal[i];
    oVert = vec3(gl_PositionIn[i]);
    EmitVertex();
  }
  EndPrimitive();
}
```

Damit alle varying-Variablen vom System als im Shaderprogrammobjekt aktiv erkannt werden, hat es sich als nützlich erwiesen, diese explizit im Fragment-Shader zu verwenden. Dies ist nötig – obwohl er im Programm gar nicht zum Einsatz kommt – da die Rasterisierung abgeschaltet wird.

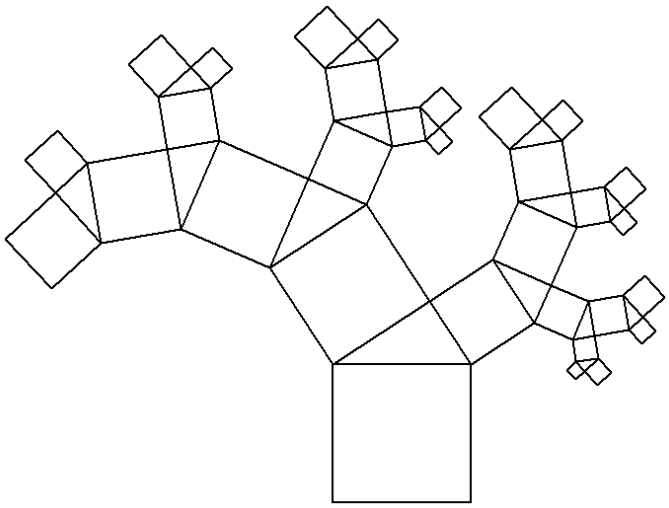

Abbildung 11: Pythagorasbaum, Quelle: Wikipedia

```
#version 120

varying vec3 oNormal;
varying vec3 oVert;

void main()
{
  gl_FragColor = vec4((oNormal+oVert)/2, 1);
}
```

Abbildung 15 auf Seite 35 zeigt die Ausgabe des Shaders mit N=0, 1, 2, 3 und 4 Passes und einem Tetraeder als Eingabegeometrie. 0 Passes bedeutet dabei, dass nur die initiale Geometrie dargestellt wird und der Geometrie-Shader nicht zur Anwendung kommt.

Der im abschließenden Render-Pass zur Beleuchtung und zum Rendering zum Einsatz kommende Phong Shader wurde von António Ramires Fernandes in einem GLSL-Tutorial [Fer08] vorgestellt.

3.4.2 Tree

Der „Tree“ genannte Shader erzeugt prozedural einen Baum mit Ästen und Blättern. Die Idee zu diesem Shader basiert auf zweidimensionalen Phytagorasbäumen (siehe Abb. 11).

Die Konstruktionsvorschrift wurde dahingehend verändert, dass ein Dreieck in Richtung der Normale extrudiert wird. Auf das abschließende Dreieck werden drei neue Dreiecke aufgesetzt, die zusammen mit dem ab-

schließenden Dreieck einen Tetraeder bilden. Die Höhe des Tetraeders bestimmt dabei, in welchem Winkel die Äste des entstehenden Baumes weiterwachsen.

Das Baummodell wurde um zusätzliche Parameter erweitert, die im GUI der Anwendung kontrollierbar sind. Die Parameter wurden als Attribute der Klasse `Tree` implementiert. Dort werden hauptsächlich `GlColor`-Komponenten zweckentfremdet, um die Nutzdaten zu transportieren. Aktuell umfassen die Baumparameter unter anderem

- einen Parameter zur Unterscheidung zwischen (nicht) finalem Blatt bzw. Ast,
- die aktuelle Rekursionstiefe,
- die Astlänge,
- einen Astverkürzungsfaktor,
- den minimalen Astumfang
- und ein Faktor zur Verringerung des Astumfanges.

Der Vertex-Shader erhält alle Eingabedaten über die Vertexattribute Position, Normale und Farbe. Wie in Kapitel 3.2.3 Seite 22 beschrieben, kommt als Transform Feedback Pufferformat `GL_C4F_N3F_V3F` zum Einsatz. Der Vertex-Shader hat – wie beim „Repeater"-Shader – keine andere Aufgabe als die Eingabedaten unverändert weiter zu reichen. Es findet keine `ModelViewProjection`-Transformation statt.

```
#version 120

varying vec4 color;
varying vec3 normal;

void main()
{
  color = gl_Color;
  normal = gl_Normal;
  gl_Position = gl_Vertex;
}
```

Die schrittweise Erzeugung der Baumgeometrie findet im Geometrie-Shader statt. Es wird eine Farbkomponente benutzt, um Dreiecke als final zu markieren. So kann unterschieden werden, ob der Baum an einer Stelle weiterwachsen soll oder nicht. Ähnlich wie beim „Repeater"-Shader existiert eine Funktion `outputTriangle`, die neue Primitive erzeugt. Sie erhält aber in einem zusätzlichen Argument die Baumparameter.

In der `main`-Funktion des Shaders werden zunächst die Positionen des Eingabedreiecks sowie die restlichen Daten entgegengenommen und in lokale Variablen kopiert. Zumindest die eingebauten Eingabevariablen sind nur lesbar und können somit nicht verändert werden, was diesen Schritt notwendig macht. Die drei Farbwerte der Eckpunkte werden in eine 4×4 Matrix geschrieben und enthalten die Parameter für das Baumwachstum. Für jedes nicht finale Dreieck wird in der `main`-Funktion der Rekursionszähler erhöht und die Astlänge mit dem Verkürzungsfaktor multipliziert. Während ein finales Eingabedreieck lediglich durchgereicht wird und keine Veränderung an seinen Vertexattributen erhält, werden alle nicht finalen Dreiecke extrudiert.

```
void main(void)
{
  vec3 a = vec3(gl_PositionIn[0]);
  vec3 b = vec3(gl_PositionIn[1]);
  vec3 c = vec3(gl_PositionIn[2]);

  mat4 data = mat4(color[0],color[1],color[2],vec4(0));

  if ( data[0][0] < 0.001 ) {
    //Rekursionstiefe
    data[0][3] += 1;
    // Ast-Segment Länge * Segment-Shrink
    data[0][1] *= data[0][2];

    extrude(a, b, c, normal[0], data);
  } else {
    //it's final, pass-through
    outputTriangle(a, b, c, data);
  }
}
```

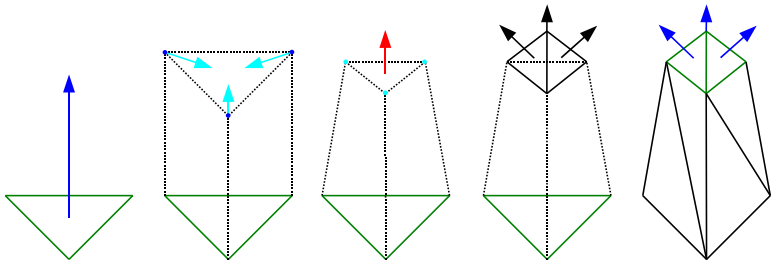

Abbildung 12: Extrudieren der Äste aus einem Eingabedreieck

Das Extrudieren läuft wie folgt ab (vergleiche Abb. 12):

- Die Eckpunkte des Eingabedreiecks werden um die aktuelle Astlänge in Richtung der Normale verschoben.
- Die entstandenen Punkte werden in Richtung des Dreieckmittelpunktes verschoben. Als Maß dafür dient der Faktor zur Verringerung des Astumfangs.
- Es wird ein weiterer Punkt gewählt über dem Mittelpunkt der verschobenen Eingabedaten.
- Aus den vier neuen Punkten werden drei abschließende Dreiecke erzeugt, die als nicht final markiert sind.
- Die Seitenflächen des Astes hingegen werden als finale Dreiecke ausgegeben.

Durch das Extrudieren können zunehmend dünnere Äste für den Baum erstellt werden. Wird ein angegebener Umfang unterschritten, wird kein neuer Ast mehr erzeugt. Ab einer benutzerdefinierten Rekursionstiefe wachsen an den Kanten des Astes Blätter, wie in Abbildung 13 zu sehen ist. Damit sie von den Ästen visuell unterschieden werden können, werden sie als finales Blatt markiert.

Rasterisiert wird der Baum erst nach der im GUI eingestellten Anzahl von Feedback-Loop-Durchläufen, in denen die Geometrie des Baumes erzeugt wird. Der Render-Pass, der für die Darstellung des Baumes verantwortlich ist, muss zuvor die zweckentfremdeten Vertexattribute auf gültige Werte setzen. Über eine `GlColor`-Komponente wird – wie schon erwähnt – zwischen (nicht) finalen Ästen und Blättern unterschieden. Obwohl die Szene des Programms nur ein Material enthält, werden Äste und Blätter farblich unterschiedlich gerendert, was durch zwei im Darstellungs-Shader definierte Farben erreicht wird.

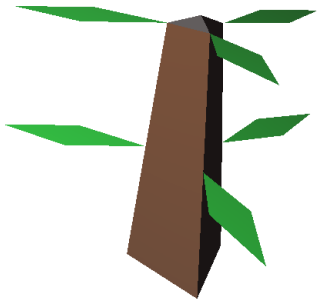

Abbildung 13: Einzelner Ast mit Blättern

4 Ergebnisse

In der vorliegenden Arbeit wurde eine Beispielanwendung (Abb. 14) entwickelt, die Geometrie-Shader einsetzt. Die Anwendung wurde mit Visual C++ 2005 erstellt unter Verwendung von OpenGL und GLSL zur Ansteuerung der Grafikhardware.

Implementiert wurde ein GUI, welches das Toolkit FLTK verwendet. Es wurde Wert darauf gelegt Funktionen der Grafikhardware in Klassen zu kapseln, wodurch ein kleines Framework mit knapp 20 Klassen entstanden ist. Um verschiedene Shader mit unterschiedlichen Objekten demonstrieren zu können, wurden Klassen – die sogenannten Views – entwickelt, welche Shader und Geometrieobjekte kapseln und eine Szene im OpenGL-Kontext der Anwendung darstellen. Zwischen den einzelnen Views kann zur Laufzeit der Anwendung über das GUI hin- und hergeschaltet werden.

Zur Demonstration der Geometrie-Shader wurde ein multi-pass Ansatz gewählt. Unter Verwendung der Transform Feedback OpenGL-Extension wurde ein Verfahren entwickelt, das rekursiv prozedurale Algorithmen iterativ auf den Geometrie-Shader Einheiten ablaufen lässt. Zum Einsatz kommt dieses Verfahren in zwei vorgestellten Shadern, einem primitiven Subdivision-Shader (siehe Abb. 15) und einem Shader, der Pflanzengeometrie auf der GPU erzeugt und darstellt (siehe Abb. 16). Ein einfaches parametrisches Baummodell wurde umgesetzt, dessen Parameter sich in Echtzeit über die grafische Benutzungsoberfläche steuern lassen. Die Parameter des Modells werden über Vertexattribute dem Shader zugänglich gemacht. Per Transform Feedback werden die Ergebnisse eines Geometrie-Shader Durchlaufs in ein VBO gespeichert und zur weiteren Erzeugung von Geometrie innerhalb eines Feedback-Loops erneut verwendet.

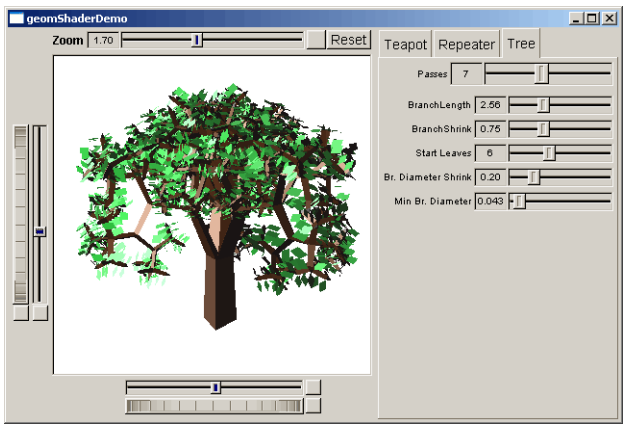

Abbildung 14: Benutzungsoberfläche Beispielapplikation

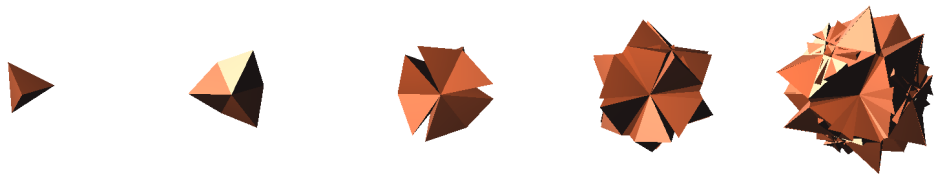

Abbildung 15: Ergebnisbilder „Repeater“-Shader, von links nach rechts 0–4 Geometrie-Shader Passes

Abbildung 16: Ergebnisbilder „Tree“-Shader, von links nach rechts 0–7 Geometrie-Shader Passes

4.1 Fazit

Die Erzeugung und Manipulation von Geometriedaten direkt auf der GPU ist ein interessantes und zum Zeitpunkt der Erstellung dieser Arbeit noch relativ junges Forschungsgebiet. Wie in dieser Arbeit gezeigt werden konnte, lassen sich mit Geometrie-Shadern unter Verwendung von Transform Feedback auch relativ komplexe Algorithmen zur Erzeugung von Geometrie auf die Grafikhardware verlagern. Durch die Verwendung des modularen klassenbasierten Ansatzes, wurde die Anwendung flexibel aber auch deutlich komplexer, als ein einfaches Beispielprogramm hätte ausfallen können.

Die Wahl von FLTK als Toolkit zur Erzeugung von grafischen Benutzungsoberflächen stellt einen Kompromiss aus Komplexität des zu erlernenden Frameworks und den erzielbaren visuellen Ergebnissen dar. So lassen sich zum Beispiel einzelne FLTK-Widgets ausrichten, gruppieren und auch mit Fluid visuell anordnen. FLTK skaliert per Voreinstellung die Widgets in ihrer Größe, wenn die Fenstergröße verändert wird, was zu großen Widgets führen kann, wird das Fenster maximiert. Das wirkt sich unter Umständen störend aus. Wie schon erwähnt ist die Anbindung von OpenGL im Gegenzug vergleichsweise einfach.

Das in der Anwendung umgesetzte Baummodell hat sich als nicht ausgereift erwiesen. Die Äste und Blätter werden stets durch möglichst einfache Vektoroperationen erzeugt. Die einzelnen Parameter und das Konstruktionsprinzip wurden nicht bedacht genug ausgewählt, um zu verhindern, dass Änderungen an verschiedenen Parametern des Modells ähnliche visuelle Auswirkungen hervorrufen. Zum Beispiel kann durch mehrere Parameter die Richtung verändert werden, in die ein neuer Ast weiter wächst.

4.2 Ausblick

Die realisierte Anwendung kann an verschiedenen Punkten verbessert werden. Interessant wäre es, weitere Baummodelle umzusetzen und das bestehende weiter zu entwickeln. Zur Veranschaulichung der Geometrie-Shader kann der „Repeater“-Shader ohne Transform Feedback umgesetzt werden. Nach Modifikation des Shaders ist ein Vergleich der erzielbaren Performance beider Ansätze durch Messungen möglich. Wie viel Geometrie in einer 3D-Szene per Geometrie-Shader erzeugt werden kann, ohne dass die Framerate spürbar nachlässt, ist ein Ansatz für weitere Untersuchungen.

Die gesammelten Erfahrungen legen Nahe, dass Geometrie-Shader und Transform Feedback eine weite Verbreitung im GPGPU-Bereich finden werden. Durch die Erweiterung der Pipeline ergebenden sich neue Möglichkeiten Daten in der Pipeline zu erzeugen, zu verwerfen und in multi-

pass Verfahren wiederzuverwenden. Dadurch lassen sich rekursive Algorithmen, die von GLSL momentan nicht unterstützt werden, dennoch iterativ auf der GPU umsetzen. Dies schafft neue Implementierungsmöglichkeiten bekannter Problemstellungen der Informatik und im Speziellen der Computergrafik.

Literatur

[Bha08] BHATTACHARJEE, Shiben. *Example Codes for Shader Model 4.0*. http://cvit.iiit.ac.in/index.php?page=resources. März 2008

[BL08] BROWN, Pat ; LICHTENBELT, Barthold. *OpenGL Extension Registry, EXT_geometry_shader4*. http://opengl.org/registry/specs/EXT/geometry_shader4.txt. Mai 2008

[Cra07] CRASSIN, Cyril. *OpenGL Geometry Shader Marching Cubes*. http://www.icare3d.org/blog_techno/gpu/opengl_geometry_shader_marching_cubes.html. November 2007

[ER08] *OpenGL Extension Registry*. http://opengl.org/registry/. Mai 2008

[Fer08] FERNANDES, António R. *OpenGL Shading Language @ Lighthouse 3D - GLSL Tutorial*. http://www.lighthouse3d.com/opengl/glsl/. Mai 2008

[flt08] *Fast Light Toolkit (fltk)*. http://www.fltk.org. Mai 2008

[Har07] HART, Evan. *GeForce8800 OpenGL Extensions*. präsentiert auf der GDC 2007. 2007

[Kor08] KORN, Matthias: *Artefaktfreie Level of Detail Darstellung prozedural erzeugter Geometrie am Beispiel von Pflanzen*, Universität Koblenz-Landau, Diplomarbeit, 2008

[LBW08a] LICHTENBELT, Barthold ; BROWN, Pat ; WERNESS, Eric. *OpenGL Extension Registry, EXT_transform_feedback*. http://opengl.org/registry/specs/EXT/transform_feedback.txt. Mai 2008

[LBW08b] LICHTENBELT, Barthold ; BROWN, Pat ; WERNESS, Eric. *OpenGL Extension Registry, NV_transform_feedback*. http://opengl.org/registry/specs/NV/transform_feedback.txt. Mai 2008

[SA06] SEGAL, Mark ; AKELEY, Kurt. *The OpenGL Graphics System: A Specification (Version 2.1)*. http://www.opengl.org/registry/doc/glspec21.20061201.pdf. Dezember 2006

[sdk08] *Nvidia OpenGL SDK 10, Code Samples*. http://developer.download.nvidia.com/SDK/10/opengl/samples.html. Mai 2008

[SEMS08] SWEET, Michael ; EARLS, Craig P. ; MELCHER, Matthias ; SPITZAK, Bill. *FLTK 1.3.0 Programming Manual - Programming with FLUID*. http://www.fltk.org/documentation.php/doc-1.3/fluid.html. Mai 2008

[WH08] WILLIAMS, Ian ; HART, Evan. *Efficient rendering of geometric data using OpenGL VBOs in SPECviewperf*. http://www.spec.org/gwpg/gpc.static/vbo_whitepaper.html. März 2008

[Yon08] YONGMING, Xie. *Geometry Shader Tutorials*. http://appsrv.cse.cuhk.edu.hk/~ymxie/Geometry_Shader/. März 2008