Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Dokumentation prozeduraler Baum

Daniel Track

Fach Real-Time Rendering

Trier, 07.08.2020

Inhaltsverzeichnis

1

Überblick und Bedienung

Dieses Programm erstellt prozedural einen Baum, indem mit einem einzelnen Dreieck gestartet wird und durch mehrere Durchläufe eines Geometry-Shaders per Transform Feedback immer neue, kleinere Äste hinzugefügt werden. Das Kernstück des Programms ist die Klasse `Application`. In `main()` wird ein Objekt dieser Klasse erstellt und initialisiert, daraufhin wird die Methode `run()` in einer Schleife aufgerufen, solange die Membervariable `is_running` auf `true` gesetzt ist.

Beim Start des Programms wird ein Fenster im Vollbildmodus geöffnet und der erste Geometry-Pass durchgeführt, sodass bereits der Stamm des späteren Baums vorhanden ist. In dem Fenster wird ein User Interface angezeigt, in dem mit einem Button weitere Geometry-Passes durchgeführt werden können. Außerdem kann durch Checkboxen das Rendering des Baums sowie die Anzeige einer Wireframe-Darstellung des Modells gesteuert werden. Außerdem gibt es die Möglichkeit, die Position der Lichtquelle zu verändern. Das Modell selbst kann durch Gedrückthalten der rechten Maustaste und Bewegen der Maus gedreht werden.

2

Klasse `Application`

Die Klasse `Application` verwaltet das Fenster in OpenGL sowie die meisten anderen Komponenten des Programms.

2.1 `Application::init()`

In der Methode `init()` wird zuerst mit Hilfe der Bibliothek SDL2 ein Fenster und OpenGL-Kontext erstellt. Dabei wird OpenGL Version 3.3 im Core Profil verwendet. Des Weiteren wird Vsync aktiviert, die Fenstergröße von SDL erfragt und als OpenGL Viewport festgelegt und einige weitere OpenGL Optionen per `glEnable` aktiviert:

- `GL_CULL_FACE` aktiviert Backface-Culling, um nur zur Kamera gerichtete Dreiecke anzuzeigen
- `GL_DEPTH_TEST` aktiviert den Tiefenpuffer, verdeckte Fragmente zu verwerfen
- `GL_MULTISAMPLE` aktiviert Multisampling, um Aliasing zu verringern

Außerdem wird im Falle einer Debug-Konfiguration OpenGls Debug-Ausgabe aktiviert und die Funktion `handle_gl_debug_output` als Callback festgelegt. Daraufhin wird noch die Bibliothek ImGui initialisiert, mit der ein Interface zur Bedienung des Programms zur Verfügung gestellt wird.

Nun werden die drei verwendeten Shader-Programme geladen und kompiliert und die benötigten Vertex Buffer und Vertex Array Objekte initialisiert. Der Buffer `start_vbo` enthält nur drei Vertices und es wird `GL_STATIC_DRAW` als Verwendung angegeben, da dieser Buffer das initiale Dreieck enthält und danach nicht mehr verwendet wird. Die beiden Buffer in `feedback_vbo` sollen genug Platz haben, um die benötigte Anzahl an Dreiecken für `MAX_GEOMETRY_ITERATIONS` Geometrie-Iterationen zu beinhalten. Für sie wird der Verwendungszweck `GL_STREAM_COPY` angegeben, da nur Shader die Daten dieser Buffer lesen und schreiben sollen.

Daraufhin wird der Start-Buffer mit den initialen Vertices gefüllt. Es handelt sich dabei um ein Dreieck auf der von der X- und Z-Achse aufgespannten Ebene, bei dem alle drei Punkte den Abstand 1 zum Ursprung des Koordinatensystems haben. Die Normalen zeigen alle in Richtung der Y-Achse (also nach oben), die initiale Länge zur Konstruktion des Baums ist 5.

Anschließend wird dieses Dreieck das erste Mal durch den Construction-Shader geschickt, und so der Stamm des Baums erstellt (weitere Details zu diesem Vorgehen im folgenden Abschnitt).

2.2 Application::run()

Diese Methode erfüllt im wesentlichen vier Funktionen: Das Empfangen von Nutzereingaben, Darstellung des graphischen Interfaces, (bei Bedarf) ausführen eines weiteren Geometry-Passes und abschließend das Zeichnen des Baums auf den Bildschirm.

Die Variablen `frame_start` und `last_frame_start` speichern die Ticks zu Beginn des aktuellen und des letzten Frames, um eine gleichmäßige Framerate zu ermöglichen. Danach werden die Events verarbeitet, die das SDL-Framework zur Verfügung stellt und bei Bedarf `is_running` auf `false` gesetzt, um nach dem aktuellen Frame das Programm zu beenden. Außerdem werden die Mausposition und der Status der Maustasten aktualisiert.

Nun wird mithilfe von ImGui ein simples Interface aufgebaut, mit dem die Darstellung des Models sowie der Wireframes an- und ausgeschaltet und die Position der Lichtquelle verschoben werden kann. Außerdem gibt es einen Button zum Ausführen eines Geometry-Passes, der bei einem Klick die Variable `run_geometry_pass` auf `true` setzt.

Wurde dieser Button geklickt und es die maximale Anzahl an Iterationen noch nicht erreicht, wird nun ein Geometry-Pass durchgeführt. Dazu wird der Counter für die Iterationen erhöht und aus seinem Wert bestimmt, welcher der beiden Feedback-Buffer in diesem Durchgang Quelle und welcher Ziel des Shader-Programms sein soll. Ersterer (bzw. das Vertex Array Objekt, zu dem er gehört) wird wie gewöhnlich eingebunden. Der Ziel-Buffer hingegen wird mit der Funktion `glBindBufferBase` als Transform Feedback Buffer gebunden. Bevor nun `glDrawArrays` aufgerufen werden kann, muss `glBeginTransformFeedback` aufgerufen werden und als Argument das Output-Primitive angegeben werden, welches der Geometry Shader auch tatsächlich ausgibt. Nach dem Call von `glDrawArrays` wird `glEndTransformFeedback` aufgerufen, um das Transform Feedback zu beenden und durch `glFlush` sichergestellt, dass die zuvor in Auftrag gegebenen Operationen auch abgeschlossen wurden. Zuletzt wird in diesem Schritt `num_triangles` aktualisiert, um ab jetzt mit der korrekten, nun höhere Zahl an Dreiecken zu arbeiten.

Der nächste Schritt ist das Rendering. Hier werden zuerst der Color- und der Depth-Buffer geleert. Falls die rechte Maustaste gedrückt ist, wird `object_rotation` entsprechend der Mausbewegung seit dem letzten Frame angepasst. Projection- und View-Matrix bleiben konstant, nur die Model-Matrix wird entsprechend der zuvor berechneten Rotation jedes Frame neu berechnet. Falls das Baum-Modell gerendert werden soll, wird nun der Render-Shader verwendet und die benötigten Uniform-Variablen werden gesetzt. Anschließend wird die aktuelle Anzahl an Drei-

ecken aus dem Feedback Buffer gerendert, der in den im letzten Geometry-Pass geschrieben wurde.

Für das Rendering der Wireframes wird sehr ähnlich verfahren: Ist die entsprechende Boolean-Variable gesetzt, wird der Line-Shader verwendet und seine Uniform-Variablen gesetzt. Daraufhin werden die Dreiecke mittels `glDrawArrays` gezeichnet.

Zuletzt wird dann das GUI gerendert und der Framebuffer getauscht, bevor auf den Start des nächsten Frames gewartet wird.

3

Shader

In der Anwendung werden drei verschiedene Shader-Programme verwendet. Zwei davon, `render` und `line`, bestehen aus einem Vertex- und einem Fragment-Shader. Sie werden verwendet, um den konstruierten Baum bzw. seine Wireframe-Repräsentation auf den Bildschirm zu zeichnen. Das dritte Shader-Programm, `construction`, besteht aus einem Vertex- und einem Geometry-Shader. Es wird zeichnet keine Pixel auf den Bildschirm, sondern nutzt Transform Feedback um neue Vertices für den Baum auf der Grafikkarte zu berechnen. Die verschiedenen Shader und ihr Code werden im Folgenden im Detail erklärt.

3.1 `Render` Shader-Programm

In diesem Shader-Programm wird im Vertex-Shader die Ausgabeposition der Vertices durch eine Model-View-Projection Matrizenmultiplikation bestimmt. `model` enthält dabei in diesem Fall nur eine Rotation um die Y-Achse. An den Fragment-Shader werden die Position und eine Normale weitergegeben, jeweils um die genannte Rotation gedreht, um so im Fragment-Shader eine Beleuchtung von einer festen Position zu ermöglichen.

Der entsprechende Fragment-Shader verwendet eine `uniform light_pos` für die Position der Lichtquelle und `camera_pos` für die Position der Kamera, um den Baum mittels Phong Shading zu beleuchten. Die Farbe des Baums ist dabei als `object_color` fest in den Shader einprogrammiert.

3.2 `Line` Shader-Programm

Dieses Shader-Programm zum zeichnen der Linien der Polygone (Wireframe) ist sehr simpel. Der Vertex-Shader bestimmt die Vertex-Position durch die Multiplikation der Model-View-Projection Matrizen; der Fragment-Shader gibt immer Rot als Farbe aus.

3.3 `Construction` Shader-Programm

Dieses Shader-Programm ist das Herzstück des Projekts. Es nutzt Transform Feedback im Geometry-Shader, um aus den bestehenden Dreiecken neue Vertices zu berechnen und so iterativ den Baum weiter aufzubauen. Der Vertex-Shader spielt dabei eine sehr geringe Rolle, denn er reicht einfach nur die Eingabewerte an den Geometry-Shader weiter.

Der Geometry-Shader bestimmt in Zeile 2 zunächst sein Input-Pimitive als `triangles`, was bedeutet, dass er jeweils die Daten eines Dreiecks erhält und mit diesen arbeitet. Zeile 3 legt `points` als Output-Primitive fest und gibt an, dass maximal 30 Punkte ausgegeben werden. Hier hätte alternativ auch mit einem Triangle-Strip als Output-Primitive gearbeitet werden können, in diesem Fall wäre es jedoch kompliziert geworden, die korrekte Reihenfolge der ausgegebenen Werte sicherzustellen.

Die Input-Variablen `normal` und `ext_length` werden als Arrays deklariert, da ein Wert für jeden der drei Vertices des zu bearbeitenden Dreiecks zur Verfügung steht. Das Ausgabeformat entspricht dem der Eingabevertices für dieses Shader-Programm.

Zur Vereinfachung des später folgenden Codes wird die Funktion `add_triangle` definiert. Sie nimmt drei Punkte und eine Erweiterungslänge `ext_length` entgegen, berechnet die Flächennormale der Punkte und gibt dann für jeden der drei Eingabepunkte einen Vertex mit der entsprechenden Position aus. Als Normale wird bei allen drei Vertices die Flächennormale verwendet, die Erweiterungslänge der Vertices entspricht dem Wert des Parameters `ext_length`.

In der `main()`-Funktion wird zunächst das Eingabedreieck ausgegeben. Ist die `ext_length` des ersten Vertices dieses Dreiecks ungleich `0`, muss ein weiterer Ast auf diesem Dreieck aufgebaut werden. Dazu wird zuerst der Mittelpunkt des Eingabedreiecks bestimmt und der `shrink_factor` definiert, welcher angibt, wie weit die Punkte des zweiten Dreiecks zum gemeinsamen Mittelpunkt hin verschoben werden. Die Positionen des zweiten Dreiecks werden dann dadurch berechnet, dass die Positionen des Eingabedreiecks erst zum Mittelpunkt hin und danach entlang der Normalen `ext_length` weit verschoben werden. Die Position der Spitze befindet sich noch 20% höher und genau über dem Mittelpunkt. Diese vier Positionen werden im Array `new_positions` gespeichert, sodass sie danach wiederverwendet werden können um die neun neuen Dreiecke zu bilden, aus denen der konstruierte Ast besteht.

Mithilfe der Funktion `add_triangle` werden dann zunächst die sechs Dreiecke für die Seiten des Asts (mit `ext_length = 0`) hinzugefügt, danach die drei Dreiecke für die Spitze des Asts mit einer 60% geringeren `ext_length` als der Eingabewert.

Zuletzt wird `EndPrimitive()` aufgerufen, um die Ausgabe von Primitiven zu beenden.

4

Weitere Files

4.1 DebugCallback.h

Enthält die Funktion `handle_gl_debug_output`, die an OpenGl als Debug Callback weitergereicht wird. Sie gibt die OpenGL Fehler auf der Konsole aus und unterbricht das Programm, wenn die Schwere des Fehlers zu hoch ist.

4.2 pch.h

Precompiled Header, um die Kompilierung zu beschleunigen. Enthält Bibliotheken und sonstige Abhängigkeiten.

4.3 Shaders.h

Enthält die Funktion `load_and_compile_shader_from_file`, die Textdateien einliest, ein Shader-Programm kompiliert und die ID dieses Programms zurück gibt, sowie ihre Helferfunktionen `checkCompileErrors` und `read_text_file`. Besonders zu erwähnen ist hier, dass beim Kompilieren des Geometry-Shaders die sogenannten Feedback Varyings angegeben werden müssen. Hierbei handelt es sich um die Namen der Variablen, die der Shader beim Transform Feedback ausgibt. Diese Namen werden mit `glTransformFeedbackVaryings` festgelegt und gelten dann für den nächsten Aufruf von `glLinkProgram`. In diesem Fall wird `GL_INTERLEAVED_ATTRIBS` als letztes Argument an `glTransformFeedbackVaryings` übergeben, da Vertices ausgegeben werden sollen, die jeweils die drei Werte `out_position`, `out_normal` und `out_ext_length` enthalten.

4.4 Types.h

Beinhaltet einige `typedef`s, um Integer-Werte einer bestimmten Bit-Größe bequemer verwenden zu können.

4.5 Vertex.h/Vertex.cpp

Enthält die Klassen `Vertex`, `ArrayBuffer` und `VertexArray`. `Vertex` definiert das verwendete Vertex-Format mit einem vierdimensionalen Vektor für die Position, einem dreidimensionalen Vektor als Normalen und einer Länge für die Erweiterung durch den nächsten Geometry-Pass. `ArrayBuffer` bietet Methoden zur bequemeren Verwendung des Buffers und erhöht die Fehlersicherheit durch Asserts, z.B. wenn ein Write-Buffer als Transform Feedback Ziel gesetzt werden soll. `VertexArray` legt bei der Erstellung das Format der Vertex-Attribute fest, sodass dieser Code nicht für jedes Vertex Array wiederholt werden muss.