

Stripification Tool

Projekt für das Modul Tool- und Pluginprogrammierung

Florian Wößner

Hausarbeit

Betreuer: Prof. Dr. Christof Rezk-Salama

Trier, 31.08.2024

Inhaltsverzeichnis

1	Einleitung	1
2	Implementierung des Triangels-to-Strip Konverters	2
3	Implementierung des Strip-Zeichners	5
4	Implementierung der Steuerung und der GUI.....	7
5	Weitere Anpassungen.....	9
6	Zusammenfassung und Ausblick	10

Einleitung

Das Ziel dieses Projekts ist es, nach dem Einlesen einer OBJ-Datei, die enthaltenen Dreiecke in Strips zusammenzufassen und diese effizient zu zeichnen. Die Technik der Stripification spielt dabei eine zentrale Rolle und bringt signifikante Vorteile mit sich.

Stripification bezeichnet den Prozess, bei dem Dreiecke eines 3D-Modells in sogenannte Triangle Strips umgewandelt werden. Ein Triangle Strip ist eine Sequenz von Vertices, bei der jedes aufeinanderfolgende Dreieck durch Hinzufügen eines weiteren Vertex definiert wird. Diese Methode reduziert den Speicherbedarf für die Vertices auf der GPU, da weniger redundante Daten gespeichert werden müssen. Beispielsweise wird aus den einzelnen Dreiecken mit Vertices (1, 2, 3), (3, 2, 4) und (3, 4, 5) nach der Stripification der Strip (1, 2, 3, 4, 5). Dies bedeutet, dass die Vertices 3 und 4 nicht mehrfach gespeichert werden müssen, was die Effizienz der Speicher- und Renderprozesse erhöht.

Das Projekt baut auf dem bestehenden Projekt „ObjToVertexBuffer“ auf, das in einer Übungsstunde bearbeitet wurde. Dieses Ausgangsprojekt stellt bereits einen OBJ-Datei-Parser zum Einlesen von 3D-Modellen sowie einen fertigen Canvas mit den passenden Datenstrukturen bereit, was eine solide Basis für die Weiterentwicklung darstellt.

Zur Erreichung der Projektziele wurden folgende Teilaufgaben definiert und umgesetzt:

- Implementierung des Triangles-to-Strip Konverters
- Implementierung des Strip-Zeichners
- Allgemeine Anpassung des vorhandenen Codes
- Implementierung der Steuerung und einer GUI

Die von mir ergänzten Codeabschnitte sind zur besseren Nachverfolgbarkeit mit Kommentaren markiert:

```
// <Florian>
...
// </Florian>
```

Implementierung des Triangels-to-Strip Konverters

Die Konvertierung von Dreieckspunkten zu Strips erfolgt in der Methode `void Mesh3D::generateStrips()`. Diese Methode führt die Hauptarbeit bei der Strip-Generierung durch, wie in den Vorlesungsfolien beschrieben (Abbildung 2.1).

Basis-Algorithmus

● Ablauf (SGI tomesch) :

1. Wenn es keine Dreiecke mehr zu verarbeiten gibt, beende den Algorithmus
2. Finde das Dreieck t mit den **wenigsten** Nachbarn (wenn mehrere existieren, wähle ein beliebiges)
3. Beginne einen neuen Strip
4. Füge das Dreieck t zu dem Strip hinzu und entferne es aus dem Mesh.
5. Wenn es keine benachbarten Dreiecke mehr gibt, gehe zu 1.
6. Wähle ein benachbartes Dreieck t' mit den **wenigsten** Nachbarn.
 - Wenn es mehrere Dreiecke gibt mit der gleichen Anzahl Nachbarn, schaue einen Schritt in die Zukunft.
 - Wenn es wieder gleich viele sind, wähle ein beliebiges.
7. $t' \rightarrow t$. Weiter mit Schritt 4.



Abbildung 2.1: Ablauf - SGI tomesch

Hier sind die wesentlichen Schritte und eine detaillierte Erklärung des Algorithmus.

1. Initialisierung:

Zu Beginn wird ein Vektor `std::vector<int> triangle_indices` mit Indizes initialisiert, indem er inkrementell mit natürlichen Zahlen gefüllt wird. Diese Indizes repräsentieren die Indizes der Dreiecke im Vektor `std::vector<Triangle> faces`.

2. Initialisierung der Nachbarn:

Die Methode `void initTriangleNeighbors()` wird aufgerufen, um das Attribut `std::map<int, vector<int>> triangle_neighbors_indices` zu initialisieren. Diese Methode geht durch alle Dreiecke durch und speichert jeweils alle nicht-bearbeiteten

Nachbardreiecke des jeweiligen Dreiecks in einen Vektor ab. Dieser Vektor wird an der entsprechenden Position in der Map gespeichert. Nicht-bearbeitete Dreiecke sind Dreiecke, die noch nicht Teil eines Strips sind. Ganz am Anfang gelten alle Dreiecke als nicht bearbeitet.

3. Hauptalgorithmus:

Der Algorithmus beginnt eine äußere Schleife, die nur dann endet, wenn alle Dreiecke als bearbeitet markiert wurden. Dies wird überprüft, indem die Größe von `std::unordered_set<int> processedTriangles` mit der Größe des Vektors `triangle_indices` verglichen wird. Sollten die Größen gleich sein, kann geschlossen werden, dass alle Dreiecke bearbeitet wurden.

4. Sortierung nach Nachbarn:

Am Anfang jeder Iteration der äußeren Schleife werden die Indizes im Vektor `triangle_indices` nach der Anzahl der nicht-bearbeiteten Nachbarn der jeweiligen Dreiecke in aufsteigender Reihenfolge sortiert, um sicherzustellen, dass in der aktuellen Iteration die Dreiecke mit den wenigsten Nachbarn immer vorne im Vektor stehen. Dieser Vektor wird jedes Mal neu sortiert, da sich die Anzahl der nicht-bearbeiteten Nachbardreiecke nach der Verarbeitung der Dreiecke unvorhersehbar ändert.

5. Beginn eines neuen Strips:

Nachdem ein passendes Dreieck gefunden wurde, beginnt ein neuer Strip. Es wird die Variable `std::vector<unsigned short> strip` deklariert, welche eine Liste der Vertices für einen Strip darstellt. Eine innere Schleife beginnt, die nur dann terminiert, wenn das aktive Dreieck `active_triangle` keine weiteren nicht-bearbeiteten Nachbarn mehr hat.

6. Hinzufügen von Dreiecken zu Strips:

Direkt am Anfang der Schleife wird das Dreieck `active_triangle` mit der Methode `void addTriangleToStrip(const int targetIndex, vector<unsigned short>& strip)` zum Strip hinzugefügt. Diese Methode schreibt die Vertices des übergebenen Dreiecks in den Vektor `strip`. Falls `strip` noch leer ist, werden die Vertices einfach hineingeschrieben, falls nicht, dann erfolgt eine Reihe von Überprüfungen:

- Es wird geprüft, ob die letzten zwei Vertices vom Strip im übergebenen Dreieck vorkommen. Falls ja, wird der Vertex in den Strip geschrieben, welcher nicht einer der beiden gemeinsamen Vertices war.
- Sollte es nur einen gemeinsamen Vertex geben, kann davon ausgegangen werden, dass der zweite gemeinsame Vertex auf der drittletzten Stelle im Strip steht. In diesem Fall wird ein sogenannter Swap vorgenommen. Dabei wird das drittletzte Element des Strips kopiert und zwischen dem vorletzten und letzten Element eingefügt.

7. Wahl eines neuen Nachbardreiecks:

Nach dem Abschluss der `addTriangleToStrip`-Methode wird ein Nachbardreieck von `active_triangle` gewählt, das die wenigsten Nachbarn hat und noch nicht bearbeitet wurde. Dazu wird der passende Vektor mit Nachbardreiecken aus der Map `triangle_neighbors_indices` genommen und nach der Anzahl der nicht-bearbeiteten Nachbarn der jeweiligen Dreiecke in aufsteigender Reihenfolge sortiert. Falls es mehrere Dreiecke mit der gleichen Anzahl an Nachbarn gibt, werden die Nachbarn der Nachbarn (Nachbarn zweiten Grades) verglichen. Das Dreieck mit der kleinsten Summe der Nachbarn zweiten Grades wird zum neuen `active_triangle`.

8. Abschluss der Iteration:

Nachdem ein kompletter Strip erstellt wurde und `active_triangle` keine weiteren Nachbarn mehr hat, wird die innere Schleife verlassen. Der erzeugte Strip wird zum Vektor `std::vector<vector<unsigned short>> strips` hinzugefügt, welcher eine Ansammlung von Strips darstellt.

9. Fortsetzung der äußeren Schleife:

Die äußere Schleife wird fortgesetzt, indem ein neues Dreieck mit den wenigsten nicht-bearbeiteten Nachbarn erneut gewählt und ein neuer Strip begonnen wird. Dieser Prozess wird so lange wiederholt, bis alle Dreiecke als bearbeitet markiert wurden.

Implementierung des Strip-Zeichners

Der Code, der für das Zeichnen von Strips zuständig ist, befindet sich in der Methode `void Mesh3D::draw()` in der Datei `Mesh3d_render.cpp`. Diese Methode verarbeitet und zeichnet die Strips der 3D-Mesh-Daten, die in der `Mesh3D`-Klasse gespeichert sind. Die folgende detaillierte Beschreibung erklärt die Funktionsweise dieser Methode.

Zu Beginn der Methode wird der Zufallszahlengenerator der Standardbibliothek initialisiert. Dazu wird der Seed des Generators auf 0 gesetzt, um sicherzustellen, dass die Farben bei jedem Aufruf der Methode `draw()` konsistent und wiederholbar sind. Dies bedeutet, dass trotz der zufälligen Natur der Farbwerte die gleiche Farbe für denselben Strip bei jedem Methodenaufruf verwendet wird.

Es wird eine Zählvariable `int counter` deklariert und auf 0 initialisiert. Diese Variable dient dazu, die Anzahl der gezeichneten Strips während der Schleifeniterationen zu zählen. Diese Zählung wird verwendet, um die maximale Anzahl der zu zeichnenden Strips zu begrenzen, die durch die Konstante `strip_amount_limit` definiert ist.

Die Methode beginnt dann mit einer äußeren Schleife, die über alle Strip-Daten iteriert. Der Schleifenmechanismus bricht ab, wenn die Anzahl der gezeichneten Strips die festgelegte Grenze (`strip_amount_limit`) erreicht. Dies stellt sicher, dass nicht mehr Strips gezeichnet werden als zulässig.

Innerhalb der Schleife wird eine zufällige Farbe für den aktuellen Strip generiert. Dies erfolgt durch Erzeugung von drei float-Werten `r`, `g` und `b`, die die RGB-Farben repräsentieren. Diese Werte werden durch die Division der vom Zufallszahlengenerator zurückgegebenen Werte durch `RAND_MAX` berechnet. Dies sorgt dafür, dass die Werte im Bereich $[0, 1]$ liegen, was der Farbdarstellung in OpenGL entspricht. Die Funktion `glColor3f(r, g, b)` wird verwendet, um diese Farbe für den aktuellen Strip festzulegen.

Anschließend wird der Strip gezeichnet. Dies beginnt mit dem Aufruf von `glBegin(GL_TRIANGLE_STRIP)`, welcher OpenGL mitteilt, dass ein neuer Strip gezeichnet wird. Innerhalb einer inneren Schleife werden die Indizes des aktuel-

len Strips durchlaufen. Für jeden Index wird der entsprechende Vertex aus dem `vertices`-Array geladen und dessen Koordinaten an die GPU übergeben, indem `glVertex3f(float x, float y, float z)` aufgerufen wird.

Sobald alle Vertices für den aktuellen Strip verarbeitet wurden, wird die innere Schleife beendet, und mit `glEnd()` wird der Strip abgeschlossen und gezeichnet. Die Zählvariable `counter` wird inkrementiert, und die äußere Schleife fährt mit dem nächsten Strip fort, bis entweder alle Strips gezeichnet wurden oder die Grenze `strip_amount_limit` erreicht ist.

Zusammengefasst, bereitet die Methode `draw()` die Farbwerte vor, verwendet OpenGL-Funktionen zum Zeichnen der Strips und sorgt durch die Zählvariable dafür, dass nur eine begrenzte Anzahl von Strips gezeichnet wird. Die Verwendung eines festen Seeds für den Zufallszahlengenerator sorgt dafür, dass die Farben konsistent bleiben, während der Code in der Schleife die Strips effizient an die GPU sendet und zeichnet.

Implementierung der Steuerung und der GUI

Die Methode `renderGui()` und die Methode `keyEvent(unsigned char key, int x, int y)` sind zentrale Bestandteile der Benutzeroberfläche und Steuerung für das **ViewerWindow**-Objekt. Im Folgenden wird detailliert beschrieben, wie diese Methoden implementiert sind und welche Aufgaben sie erfüllen.

Die Methode `renderGui()` ist verantwortlich für die Darstellung und Aktualisierung der grafischen Benutzeroberfläche unter Verwendung von `ImGui`, einer weit verbreiteten und leistungsfähigen Bibliothek für sofortige GUI-Entwicklung (siehe `ImGui`). Diese Methode verwaltet die Benutzeroberfläche, die dem Nutzer ermöglicht, verschiedene Interaktionen und Einstellungen vorzunehmen.

1. Ladefenster:

Wenn der `isLoading`-Status auf `true` gesetzt ist, zeigt die Methode ein Ladefenster an, das den Benutzer darüber informiert, dass eine Aktion (wie das Laden eines neuen Meshes) im Gange ist. Dies geschieht durch den Aufruf von `ImGui::Begin()` mit entsprechenden Flags, um das Fenster ohne Titel, Größe und Bewegung anzuzeigen. `ImGui::Text()` wird verwendet, um den Ladehinweis darzustellen.

2. Haupt-Benutzeroberfläche:

Sobald `isLoading` auf `false` gesetzt ist, zeigt die Methode die Haupt-GUI an, die mehrere Abschnitte enthält:

- Unter dem Abschnitt 'Control' wird dem Benutzer erklärt, welche Tastenkombinationen zur Steuerung der Anzahl der anzuzeigenden Strips verwendet werden können. Die Beschreibungen dieser Steuerungen werden durch `ImGui::Text()` bereitgestellt.
- Im Abschnitt 'Mesh' kann der Benutzer zwischen verschiedenen Mesh-Dateien auswählen. Hierzu wird `ImGui::BeginCombo()` verwendet, um ein Dropdown-Menü zu erstellen. Die Auswahl wird durch `ImGui::Selectable()` ermöglicht, wobei die Variable `selectedIndex` die Auswahl speichert.

- Im Abschnitt 'Info' werden verschiedene Informationen über das aktuelle Mesh angezeigt. Dazu gehören die Anzahl der einzigartigen Vertices, der Dreiecksflächen und der Strips sowie die Anzahl und Einsparungen von Vertices, die an die GPU gesendet wurden. Berechnungen zur Anzahl der gesparten Vertices werden durchgeführt, und die Ergebnisse werden mittels `ImGui::Text()` angezeigt.
- Ein Button mit der Beschriftung 'Process' ermöglicht das Laden eines neuen Meshes. Beim Klicken auf diesen Button wird der Status `isLoading` auf `true` gesetzt und ein neuer Thread gestartet, der das Mesh aus einer .obj-Datei lädt. Der alte Mesh wird durch das neue ersetzt, und `isLoading` wird auf `false` zurückgesetzt, sobald der Ladevorgang abgeschlossen ist.

3. Rendering und Abschluss:

Nach der Erstellung und Aktualisierung der GUI wird `ImGui::Render()` aufgerufen, um die GUI zu rendern, und `ImGui_ImplOpenGL3_RenderDrawData()` wird verwendet, um die gerenderten GUI-Daten mit OpenGL anzuzeigen.

Die Methode `KeyEvent(unsigned char key, int x, int y)` wurde angepasst, um Tastatureingaben zu verarbeiten und entsprechende Änderungen in der Anzeige der Strips vorzunehmen. Der Parameter `key` gibt die gedrückte Taste an, und basierend auf diesem Wert wird eine der folgenden Aktionen ausgeführt:

- Wenn die Taste '1' gedrückt wird, wird die Anzahl der anzuzeigenden Strips um eins reduziert, vorausgesetzt, die Anzahl ist nicht bereits auf 0 reduziert.
- Bei Drücken der Taste '2' wird die Anzahl der anzuzeigenden Strips um eins erhöht, sofern sie nicht bereits die maximale Anzahl der verfügbaren Strips erreicht hat.
- Wenn die Taste '3' gedrückt wird, wird entweder die Anzahl der anzuzeigenden Strips auf 0 gesetzt (was alle Strips ausblendet) oder auf die Gesamtanzahl der verfügbaren Strips, um alle anzuzeigen.

Weitere Anpassungen

In diesem Kapitel werden die Änderungen an weiteren Code-Stellen erläutert.

- In der **main.cpp** wurden Anpassungen vorgenommen, um die Callback-Funktionen für das Fenster-Management zu aktualisieren und die Interaktionen zwischen ImGui und der Hauptfensterinstanz zu ermöglichen. Die Funktionen **reshape**, **keyboard**, **mouse** und **move** wurden so modifiziert, dass sie sowohl die ImGui-spezifischen Anforderungen erfüllen als auch die entsprechenden Methoden des ViewerWindow-Objekts aufrufen.
Zusätzlich wurde ImGui in der **main()**-Funktion initialisiert, indem die ImGui-Version überprüft, ein ImGui-Kontext erstellt, das Dark-Theme angewendet und die ImGui-Backends für GLUT und OpenGL3 konfiguriert wurden.
- In der **CGlutWindow**-Klasse wurden spezifische Anpassungen für die Integration von ImGui vorgenommen. Beim Destruktor der Klasse wurde der Code hinzugefügt, um ImGui-Ressourcen zu bereinigen.
Zusätzlich wurde in der **renderFrame()**-Methode der Code ergänzt, um einen neuen ImGui-Frame zu beginnen und zu rendern.
- Alle Funktionalitäten zum Einlesen und Verarbeiten von Normalen und Texturkoordinaten wurden entfernt, da sie nicht mehr benötigt werden.
- Entsprechende Header-Dateien wurden angepasst, um die neuen Funktionalitäten zu integrieren.

Zusammenfassung und Ausblick

In diesem Kapitel soll die Arbeit noch einmal kurz zusammengefasst werden. Insbesondere sollen die wesentlichen Ergebnisse Ihrer Arbeit herausgehoben werden. Erfahrungen, die z.B. Benutzer mit der Mensch-Maschine-Schnittstelle gemacht haben oder Ergebnisse von Leistungsmessungen sollen an dieser Stelle präsentiert werden. Sie können in diesem Kapitel auch die Ergebnisse oder das Arbeitsumfeld Ihrer Arbeit kritisch bewerten. Wünschenswerte Erweiterungen sollen als Hinweise auf weiterführende Arbeiten erwähnt werden.