TECHNISCHE UNIVERSITÄT DRESDEN

Computergraphik und Visualisierung

Fakultät Informatik

Institut für Software- und Multimedia-Technik
Professur für Computergrafik und Visualisierung

Bachelor-Arbeit

Prozedurale Dekoration von 2D Plots

Luis Renner

Geboren am: 11. April 2000 in Chemnitz
Matrikelnummer: 4765427

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Erstgutachter
Prof. Dr. Stefan Gumhold

Zweitgutachter
Dr. Dmitrij Schlesinger

Betreuer
Benjamin Russig

Eingereicht am: 8. August 2022

TECHNISCHE UNIVERSITÄT DRESDEN

Computergraphik und Visualisierung

Fakultät Informatik Professur für Computergrafik und Visualisierung

Aufgabenstellung für die Anfertigung einer Bachelor-Arbeit

Studiengang:	Informatik
Name:	Luis Renner
Matrikelnummer:	4765427
Immatrikulationsjahr:	2018
Titel:	Prozedurale Dekoration von 2D Plots

Motivation

Immersive Visualisierungen ermöglichen intuitives Erkunden und verstehen komplexer Daten, jedoch müssen häufig auch klassische Ansichten wie Zeitreihenplots zu Navigations- und Übersichtszwecken eingebettet werden. Die Teilelemente dieser Plots werden traditionell mittels eigener Geometrie dargestellt. Mit dieser Technik muss bei jeder Veränderung des Plots mindestens die Geometrie der betroffenen Teilelemente neu erzeugt werden. Bei Performanz-kritischen Echtzeitanwendungen, wie dem Streamen von Live-Daten oder der dynamischen Interaktion mit dem Plot, will man jedoch diese Neuerzeugungen vermeiden. Ein weiteres Problem sind visuelle Artefakte, die durch z-Fighting beim Layering der Geometrien entstehen können. Eine mögliche Alternative ist das prozedurale Erzeugen der Plotstrukturen im Texturraum, was sich auf modernen GPUs effizient auf Fragmentbasis implementieren lässt. Durch das Erzeugen im Texturraum kann die dynamische Anpassung von Skalierung und Granularität der Achsen zur Interaktion mit dem Plot, sowie das Nachladen von Live-Daten, effizient implementiert werden. Bei diesem Ansatz werden alle Plotstrukturen auf einer einzigen Geometrie dargestellt, wodurch z-Fighting vermieden wird. Ein weiterer Vorteil ist die triviale Umsetzung von Antialiasing beim Arbeiten auf Fragmentbasis.

Ziel der Arbeit

Ziel der Bachelorarbeit ist es, eine Bibliothek zur Texturraum basierten Darstellung von Plots mit OpenGL zu implementieren und abschließend die Implementierung in Bezug auf Funktion und Performance zu analysieren. Dazu wird auch nach Strategien zur Dekoration von Plots in existierenden Plotbibliotheken gesucht und die Umsetzbarkeit dieser im Texturraum diskutiert. Die Arbeit beschränkt sich dabei auf 2D Plots.

Schwerpunkte der Arbeit

- Literaturrecherche:
  - Prozedurales Rendering mit Fokus auf Texturraum
  - Strategien und Algorithmen zur Dekoration von Plots in etablierten Tools wie GnuPlut und Excel oder Algebrasystemen wie Matlab, Mathematica und Maple
- Analyse und Diskussion zur Anwendbarkeit existierender Methoden im Fragment Shader
- Auswahl (ggf. Neudesign) und Implementierung geeigneter Methoden in Form eines Renderers für das CGV-Framework mit folgenden Eigenschaften:
  - Konfiguration über Render Styles
  - Layer-Prinzip zur Organisation der Plotkomponenten
  - Punkt- und linienbasierte Dekorationen
  - Schriftunterstützung für Achsen- und Tickmark-Labels
- Evaluation und Diskussion des entwickelten Ansatzes
  - qualitative Analyse der Renderingergebnisse (Artefakte etc.)
  - quantitative Analyse der Renderperformanz der Implementierung

Optionales

- blickpunktabhängige Adaption von Tickmarks und Gittern
- blickpunktabhängige Textausrichtung
- Titel, Untertitel oder komplett frei platzierbare Label mit beliebigem Text

Erstgutachter:	Prof. Dr. Stefan Gumhold
Zweitgutachter:	Dr. Dmitrij Schlesinger
Betreuer:	Benjamin Russig
Ausgehändigt am:	25. April 2022
Einzureichen am:	11. Juli 2022

Prof. Dr. Stefan Gumhold
Betreuender Hochschullehrer

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich das vorliegende Dokument mit dem Titel *Prozedurale Dekoration von 2D Plots* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in diesem Dokument angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung des vorliegenden Dokumentes beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 8. August 2022

Luis Renner

Zusammenfassung

In dieser Arbeit wird ein 2D-Plot-Renderer implementiert. Dieser wird Plot-Elemente im Texturraum erzeugen und zusammensetzen. Dazu werden wichtige Aspekte der Plot-Darstellung untersucht und Abschließend die Ergebnisse Ausgewertet. Das Zeichnen von kontinuierlichen Graphen, sowie die Darstellung eines Plot übergreifenden Grids sind möglich.

Inhaltsverzeichnis

Symbole und Abkürzungen

CGVF	CGV-Framework
SDF	Signed-Distance-Fields
RCR	Rectangle-Renderer
SSB	Shader-Storage-Buffer

1 Einleitung

Traditionell werden Plots in der Computergraphik aus Einzelteilen zusammen gebaut. Dabei hat jedes Element im Plot eine eigene Geometrie, eine Achse ist zum Beispiel ein sehr langes Rechteck. Diese Darstellung hat einige schwierige Probleme, wie z.B. Z-Fighting. Dabei entstehen visuelle Artefakte bei Elementen die sich direkt Überlappen. Ein anderes Problem ist, dass bei der Darstellung von Echtzeitdaten, sämtliche Geometrien neu berechnet werden müssen. Weniger ein Problem als eine neutrale Tatsache ist, dass diese Traditionellen Plot-Renderer die CPU stärker beanspruchen, um die Geometrieformen zu berechnen.

Die in dieser Arbeit betrachtete Alternative, ist das vollständige Generieren von Plots im Fragment-Shader. Dabei werden alle Elemente des Plots im Texturraum und auf Fragmentbasis erzeugt. Z-Fighting ist damit komplett irrelevant und die Performance bleibt beim nachladen von neuen Elementen gleich. Zudem verlagert dieser Ansatz fast alle Arbeit von CPU auf die GPU, was für manche Anwendungsfälle von Nutzen sein kann.

2 Grundlagen

2.1 CGV Framework

Die zu dieser Arbeit begleitende Implementation wird als Plugin für das CGV Framework[Gum21] der Professur für Computergraphik und Visualisierung an der Technischen Universität Dresden entwickelt. Im weiteren Verlauf der Arbeit wird CGV-Framework mit CGVF abgekürzt. Das Framework ist im Kern eine OpenGL basierte Render-Engine mit eingebauter Mausteuerung und einer Schnittstelle zur einfachen Umsetzung von rudimentären User-Interface-Elementen. Zudem verfügt das Framework über zahlreiche, modulare Zusatzbibliotheken.

2.2 Signed-Distance-Fields

Signed-Distance-Fields (SDFs) sind Mathematische Felder, die die Distanz zur Umrandung einer geometrischen Form angeben. Null wäre dabei genau auf der Kante, negative Werte bilden die Distanz ins Innere der Form ab und positive Werte die Distanz nach Außen. Diese SDFs können mit Hilfe einer Signed-Distance-Function angegeben werden. Das ist eine Funktion, die eine Position im Raum als Eingabe nimmt und den entsprechenden Distanzwert als Ausgabe zurück gibt.

3 Verwandte Arbeiten

Sowohl das Erzeugen von Vector-Graphiken auf Fragmentbasis als auch die Darstellung von Daten in Graphen (Plots) finden in der Wissenschaft Anwendung und werden auch in vielen Arbeiten erforscht. Die Kombination hingegen findet bisher keine große, wissenschaftliche Aufmerksamkeit. Das kann möglicherweise daran liegen, dass in allen Anwendungsfällen mit statischen Daten geometriebasierte Plots sinnvoller wären, da die Geometrien nicht in jedem Frame neu berechnet werden müssen. Somit habe Ich keine direkt Verwandten Arbeiten gefunden, die dasselbe Ziel verfolgen würden, sondern Arbeiten die in Teilen Verwandt sind.

3.1 Vektor-Graphik

In der Arbeit *Random-access rendering of general vector graphics*[NH08] wird die Darstellung von Vektor-Graphiken beschrieben. Beschriebene Techniken beinhalten Prefiltering, SDFs und die Rasterisierung der Graphiken in Teilbereiche. Die Arbeit ist in soweit relevant, dass sie sich mit das performanten Darstellung von SDFs im Texturraum befasst. Die Plots in dieser Arbeit werden mit der selben Technik dargestellt.

4 Konzeption

4.1 Framework

Die gesamte Implementation wird ein Plugin für das CGVF. Dabei will Ich möglichst viele Schnittstellen auf dem Framework nutzen. Eine der wichtigsten Schnittstellen ist das eingebaute GUI-System, das für die Demo-Anwendung zum Einsatz kommen wird.

4.2 SDFs

Große Teile der Arbeit werden sich auf SDFs stützen, da diese der beste Weg sind um Vektor-Graphiken auf Fragmentbasis darzustellen. Außerdem gibt es Möglichkeiten komplexere Graphiken mit Rasterisierung in Teil Graphiken zu unterteilen, wie in [NH08] beschrieben.

4.3 Referenzsystem

Alle Elemente des Plots, die auf der zugrunde liegenden Geometrie dargestellt werden brauchen zur Bestimmung ihrer Position ein Referenzsystem. Dieses Referenzsystem ist im Texturraum der Geometrie. Also können wir z.B. die UV-Koordinaten nutzen, die uns GLSL bereitstellt. Der UV-Koordinatenraum erstreckt sich von (0.0, 0.0) zu (1.0, 1.0). Ich bevorzuge jedoch ein Referenzsystem, das sich von (-1.0, -1.0) bis (1.0, 1.0) erstreckt und bei dem der Punkt (0.0, 0.0) genau in der Mitte liegt. Das ist auch das Referenzsystem von dem Ich in der restlichen Arbeit ausgehen werde.

4.4 Graphen

Es soll die Möglichkeit geben, statt nur einzelner Punkte, auch Kontinuierliche Graphenlinien zu Zeichnen. Punkte und Linien sollten auch kombiniert eingesetzt werden können.

Abbildung 4.1 veranschaulicht diese Darstellungsmöglichkeiten. Der einfachste Weg so eine Linie darzustellen ist, das einfache erzeugen eines Liniensegments zwischen zwei, nebeneinander liegenden Punkten im Datensatz für den Graphen. Ein Beispiel für so eine Darstellung sieht man in Abbildung 4.2.

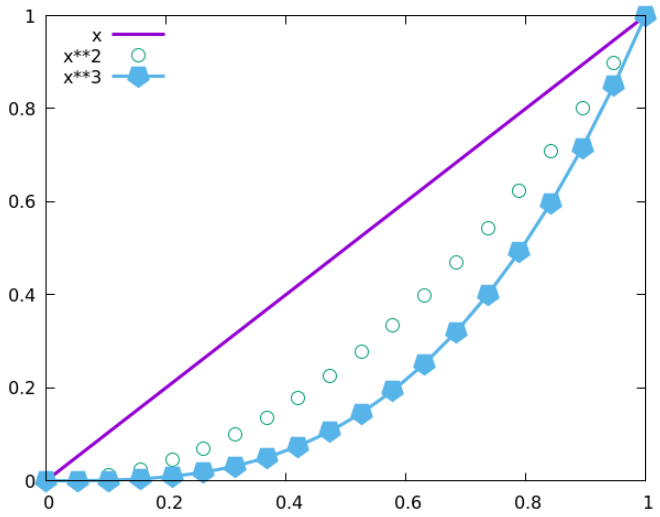

Abbildung 4.1: Gnuplot 5 [Phi20] p.47

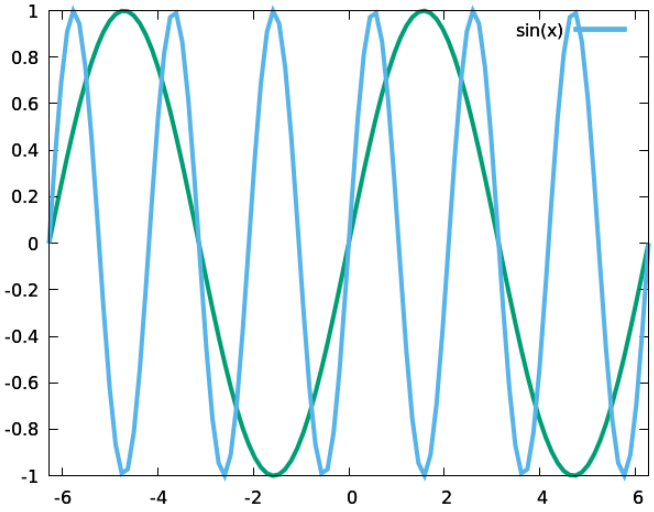

Abbildung 4.2: Gnuplot 5 [Phi20] p.41

4.5 Dekoration

4.5.1 Grid

Der Plot soll noch weitere Bestandteile haben außer den Graphen. Eine wichtige Dekoration ist ein Grid auf dem gesamten Plot zur Orientierung, wie in Abbildung 6.7.

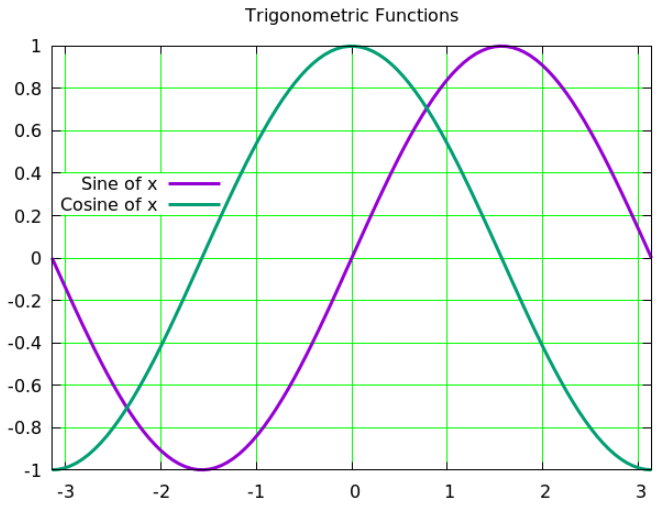

Abbildung 4.3: Gnuplot 5 [Phi20] p.31

Natürlich gibt es neben einem statischen Raster auch andere Möglichkeiten für ein Grid. Für ein dynamisches Grid, das seine Auflösung an die Zoomstufe anpassen kann braucht man einen Referenzpunkt. Ich werde versuchen ein Grid zu Implementieren, das sich and der Position im 3D-Raum orientiert. Dabei sollen mehr Grid-Linien erscheinen, desto näher man heran geht. Die Auflösung des Rasters sollte großer werden wenn weiter weg geht und feinen Rasterlinien sollten verschwinden.

4.5.2 Tickmarks

An beiden Hauptachsen sollte es Tickmarks geben die eine alternative Orientierungsmöglichkeit zum Grid bieten. Diese Tickmarks könnten, genau wie das Grid ebenfalls dynamisch sein und sich an die Zoomstufe anpassen. Dabei würde die Unterteilung zwischen den Tickmarks beim heranzoomen immer kleiner.

4.5.3 Färbung

Alle Dekorationen sollten die Möglichkeit haben, sie beliebig einzufärben, um verschiedene Aspekte visuell hervorheben zu können.

4.5.4 Beschriftung

Beschriftungstext an Achsen, Graphen oder sogar einzelnen Tickmarks ist wichtig für das herauslesen von Größen und anderen Informationen von Plots. Der Text könnte dabei auch 'prozedural' im Fragment-Shader erzeugt werden. Dazu müssen Glyphen aus einem SDF-Atlas geladen werden. Die Erstellung eines solchen Atlas ist z.B. mit der Implementierung 'msdfgen' [Chl21] möglich. Um bessere Ecken und Kanten bei niedrig aufgelösten SDF-Texturen zu haben, bieten sich Multi-Channel-SDFs an, wie in diesem Paper beschrieben [Gre07].

4.6 Raumtransformation

Eine weitere nützliche Funktion, wäre eine Auswahl aus verschiedenen Achsenräumen. So könnte man neben dem normale Raum auch logarithmisch oder exponentiell wachsende Achsen haben.

4.7 Interaktion

Die Implementierte Demo soll dem Nutzer die Möglichkeit geben beliebig viele Graphen im selben Plot darzustellen und diese auch individuell anpassen zu können. Der Nutzer muss auch in der Lagen sein im 3D-Raum mit dem Plot zu interagieren durch einen adaptive Grid-Auflösung, die sich an der Distanz zur Kamera orientiert.

5 Methoden und Umsetzung

5.1 Einbindung in das Framework

Die Plot-Demo wird mit dem Plugin-Interface des CGVF implementiert. Dieses Plugin kann dann von der Viewer Anwendung aufgerufen werden. Das Plugin wird dabei von einer Hauptklasse repräsentiert, diese nutzt das Node-Interface des Frameworks. Wird das Plugin aufgerufen, wird auch der Konstruktor der Klasse aufgerufen.

Zudem nutzt das Plugin das Render-Interface, das die Methoden 'init' und 'draw' zur Verfügung stellt. Dieses Interface macht es möglich eine Interne Render-Pipeline vom Framework zu nutzen, sodass am Ende nur die Shader modifiziert werden müssen. Die Methode 'init' wird vor dem Rendern des ersten Frames aufgerufen und die 'draw' Methode beim Rendern jedes Frames.

Zuletzt nutzt das Plugin das GUI-Interface, das die Erstellung einer einfachen Nutzeroberfläche aus vorgefertigten Elementen in der Viewer Anwendung ermöglicht.

```
class procedural_plot :
        public cgv::base::node,
        public cgv::render::drawable,
        public cgv::gui::provider {
...
bool init(cgv::render::context &ctx) override { ... }
...
void draw(cgv::render::context &ctx) override { ... }
...
void create_gui() override { ... }
... }
```

Abbildung 5.1: procedural_plot.cxx Plugin Aufbau

5.2 Rectangle-Renderer

Der Rectangle-Renderer (RCR) ist eine Hilfsklasse des CGVF, zur Darstellung einer einfachen Rechteckgeometrie. Dieser hat bereits ein vorgegebenes Shader-Programm, dass heißt

es müssen nur noch Position und Dimension des Rechtecks angegeben werden und der RCR kann das Rechteck in der 'draw' Methode darstellen. In Abbildung 5.2 wird statt dem Vorgabeprogramm ein eigenes Shader-Programm genutzt, das die Funktionalität zur Darstellung der Plots enthalten wird.

```
auto &rcr = cgv::render::ref_rectangle_renderer(ctx, 0);
rcr.set_prog(shader_program);
rcr.set_position(ctx, RECT_POSITION);
rcr.set_extent(ctx, RECT_EXTEND);
rcr.render(ctx, 0, 1);
```

Abbildung 5.2: procedural_plot.cxx Rectangle-Renderer

5.3 Shader-Programm

Ein Shader-Programm im CGVF ist eine Sammlung von allen Shadern in der Shader-Pipeline die zum Darstellen einer Szene im Viewer gebraucht werden. Mit einer 'glpr' Datei werden alle Teildateien des Shader-Programms definiert. In Abbildung 5.3 wird die 'rectangle.glpr' Datei, das Vorgabeprogramm auf dem RCR, als Grundlage genutzt und um die Dateien 'sdf.glfs' und 'procedural_plot.glfs' erweitert.

```
files:rectangle
vertex_file:group.glsl
vertex_file:quaternion.glsl
vertex_file:view.glsl
geometry_file:side.glsl
geometry_file:view.glsl
fragment_file:sdf.glfs
fragment_file:procedural_plot.glfs
fragment_file:side.glsl
fragment_file:lights.glsl
fragment_file:bump_map.glfs
fragment_file:surface.glsl
fragment_file:brdf.glsl
```

Abbildung 5.3: procedural_plot.glpr

Mit der Member-Funktion 'build_program' werden in der Abbildung 5.4 alle, in der 'glpr' definierten Shader-Dateien kompiliert und dem Shader-Programm hinzugefügt.

```
cgv::render::shader_program shader_program;
shader_program.build_program(ctx, "procedural_plot.glpr", true, defines);
```

Abbildung 5.4: procedural_plot.cxx Shader-Programm Erstellung

5.4 Übermittlung an den Shader

Da sich der Kern der Implementation für die Plot-Darstellung im Fragment-Shader befindet, muss es einen Weg geben, die Eingaben aus dem Nutzer-Interface und die Graph-Daten in

den Shader zu laden, um sie dort nutzen zu können. Für diesen Zweck werden tatsächlich drei verschiedene Wege genutzt: Compiler-Defines, Uniform-Variablen und Shader-Storage-Buffer.

5.4.1 Compiler-Defines

Compiler-Defines im Shader werden hier genutzt, um bestimmte Teile des Codes nur dann zu, wenn ein Define gesetzt ist. Dadurch können verschachtelte If-Statements vermieden und so Performance gespart werden. Zudem können Defines auch genutzt werden um einfache Integer-Werte zu übermitteln, die nur beim erneuten bauen des Shaders aktualisiert werden müssen. Wenn im Shader eine Funktion aktiviert oder deaktiviert werden soll, dann wird in Abbildung 5.5 der Define gesetzt oder nicht gesetzt und das Shader-Programm wird neu gebaut.

```
bool is_hidden = false;
const cgv::render::shader_define_map defines = {
            {"IS_HIDDEN",    std::to_string(is_hidden)}
};
if (defines != current_defines) {
        shader_program.destruct(ctx);
        shader_program.build_program(ctx, "procedural_plot.glpr", true, defines);
        current_defines = defines;
}
```

Abbildung 5.5: procedural_plot.cxx Compiler-Defines werden gesetzt

Im Shader muss zunächst der Default-Define gesetzt werden, damit der Compiler weiß, dass dieser Define existiert. In Abbildung 5.6 werden die Bereiche des Codes zwischen '#if ...' und '#endif' abhängig vom Define geladen oder nicht geladen.

```
#define IS_HIDDEN 0
color = vec4(vec3(0.3), 1.0);
#if IS_HIDDEN == 1
color.a = 0.0f;
#endif
frag_color = color;
```

Abbildung 5.6: procedural_plot.glfs Compiler-Defines im Shader

5.4.2 Uniform-Variablen

Uniform-Variablen werden genutzt um Konfigurationsdaten an den Shader zu senden, die nicht mit einem einfachen ja/nein ausgedrückt werden können, bzw. häufiger aktualisiert werden müssen. Es können auch Arrays mit Festgröße bzw. Vektoren übermittelt werden.

```
template<typename T>
void set_uniform(cgv::render::context &ctx, const std::basic_string<char> &name, const T &value) {
    shader_program.set_uniform(ctx, shader_program.get_uniform_location(ctx, name), value);
}

set_uniform(ctx, "alpha_test_val", domain_alpha);
set_uniform(ctx, "domain_extent", RECT_EXTEND);
```

Abbildung 5.7: procedural_plot.cxx Setzen zweier Uniform-Variablen

```
uniform float alpha_test_val;
uniform vec2 domain_extent;
```

Abbildung 5.8: procedural_plot.glfs Definition der Uniform-Variablen im Shader

5.4.3 Shader-Storage-Buffer

Shader-Storage-Buffer (abgekürzt mit SSB) sind Buffer mit einer nicht vordefinierten Größe die unabhängig von den Uniform-Variablen aktualisiert werden können. Durch die dynamische Größe bieten sich SSBs an, um die vollständigen Graph-Daten in Sets in den Shader zu laden. Ein SSB lädt dabei alle Daten von beliebig vielen Graphen, sequenziell aneinander gereiht. Ein zweiter SSB lädt zusätzliche die Anzahl der Datenpaare, die Färbung des Graphen und die Skalierung des Graphen als Parameter für jeden Graphen, ebenfalls sequenziell aufgereiht. Diese Parameter sind in dem Struct 'Graph_Param' definiert. Das CGVF bietet bereits implementierte SSBs an. In Abbildung 5.9 wird die vordefinierte Klasse 'vertex_buffer' genutzt.

```
cgv::render::vertex_buffer vec_sbo, segment_sbo;
std::vector<cgv::math::fvec<float, 2>> graph_data;
std::vector<Graph_Param> graph_param;

template<typename T>
void load_data_buffer(cgv::render::context &ctx, cgv::render::vertex_buffer &buffer, T &data) {
    cgv::render::vertex_buffer sbo(cgv::render::VBT_STORAGE, cgv::render::VBU_STATIC_READ);
    if (!sbo.create(ctx, data)) throw;
    buffer = std::move(sbo);
}

segment_sbo.destruct(ctx);
vec_sbo.destruct(ctx);
load_data_buffer(ctx, segment_sbo, graph_param);
load_data_buffer(ctx, vec_sbo, graph_data);

const int vec_sbo_h = vec_sbo.handle ? (const int &) vec_sbo.handle - 1 : 0,
                      segment_sbo_h = segment_sbo.handle ? (const int &) segment_sbo.handle - 1 : 0;
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, segment_sbo_h);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, vec_sbo_h);
```

Abbildung 5.9: procedural_plot.cxx Erstellen und Binden von SSBs

```
struct Graph_Param
{
        int vertex_count;
        float color[4];
        float scale[2];
};

layout(std430, binding = 0) readonly buffer GP { Graph_Param graph_param[]; };
layout(std430, binding = 1) readonly buffer GD { vec2 graph_data[]; };
```

Abbildung 5.10: procedural_plot.glfs Definition von SSBs im Shader

5.5 SDF-Implementationen

Alle SDFs sind in einer eigenen Shader-Datei 'sdf.glfs'. Diese SDFs stammen aus Artikeln von Inigo Quilez [Qui22], da seine Implementationen die effizientesten sind, die ich gefunden habe. Effizient bedeutet hier die wenigsten Instruktionen des kompilierten Shaders auf der Grafikkarte. In Abbildung 5.11 ist die Linien-Segment SDF als Beispiel angegeben.

```
float sdf_segment(in vec2 p, in vec2 a, in vec2 b, float wi)
{
    vec2 pa = p-a;
    vec2 ba = b-a;
    float h = clamp(dot(pa,ba)/dot(ba,ba), 0.0, 1.0);
    return length(pa-ba*h);
}
```

Abbildung 5.11: sdf.glfs Linien-Segment SDF

Weiterhin werden SDFs für die Darstellung von Parallelogrammen, Kreisen und Boxen von [Qui22] genutzt. Für die Distanz zu den Achsen wird die SDF aus Abbildung 5.13 genutzt.

```
float sdf_axis(in vec2 p)
{
    return min(abs(p.x), abs(p.y));
}
```

Abbildung 5.12: sdf.glfs Achsen SDF

5.5.1 Anti-Aliasing

Anti-Aliasing bei der Darstellung der SDFs wird mit der eingebauten 'smoothstep' Funktion von GLSL umgesetzt.

```
segment = 1.0 - smoothstep(0, -fwidth(sdf), sdf);
frag_color = mix(sdf_color, frag_color, segment);
```

Abbildung 5.13: sdf.glfs Achsen SDF

5.6 Koordinatensystem

Der Geometry-Shader auf dem Shader-Programm des RCR, den wir in der 'procedural_plot.glpr' immer noch nutzten, stellt das 'RECTANGLE_FS' Objekt für den Fragment-Shader zur Verfügung, zusehen in Abbildung 5.14. Darin befinden sich nützliche Daten aus früheren Berechnungen in der Shader-Pipeline, wie der 2D-Vektor 'splatcoord'. Dieser Vektor gibt uns die Koordinaten des Fragments auf der Oberfläche des Rechtecks an. Praktischer Weise sind diese Koordinaten nicht im UV-Koordinatensystem von 0.0 bis 1.0, sondern in einem Splat-Koordinatensystem von -1.0 bis +1.0 mit dem Mittelpunkt bei 0.0, so wie es in der Konzeption geplant war.

```
in RECTANGLE_FS {
        vec3 position_eye;
        vec3 normal_eye;
        vec2 texcoord;
        vec4 color;
        vec4 secondary_color;
        vec4 border_color;
        float depth_offset;
        flat int side; // 0 is back facing and 1 is front facing
        vec2 splatcoord;
        vec2 percentual_splat_size;
        vec2 percentual_blend_width;
        vec2 percentual_rectangle_size;
        vec2 percentual_core_size;
} fi;
```

Abbildung 5.14: procedural_plot.glfs RECTANGLE_FS

Um nun die tatsächliche Position im Plot-Raum zu bestimmen, wird die Splat-Position mit dem Streckungs-Vektor 'domain_extend' Multipliziert. Dieser Vektor kann Manuell angepasst werden und gibt die Anzahl von Schritten in X- und Y-Richtung an. Dieser Wert gilt für die positiven und negativen Richtungen, der Ursprung bleibt vorerst in der Mitte. Um den Ursprung zu verschieben wird als nächstes der Offset subtrahiert, negativer Offset schiebt den Ursprung in die negative Richtung und positiver Offset in die positive Richtung. Anschließend wird die Position mit der Skalierung des gesamten Plots Multipliziert. Die gesamte Berechnung ist in Abbildung 5.15 zusehen.

```
vec2 tex_pos = fi.splatcoord*domain_extent;
tex_pos -= offset;
tex_pos *= graph_scale;
```

Abbildung 5.15: procedural_plot.glfs Position im Plot-Raum

Alle weiteren Implementationen werden diese Plot-Raum-Position des Fragments nutzen.

5.7 Darstellung von Graphen

Alle Graphen, die durch den SSB in den Shader geladen wurden, werden übereinander dargestellt. Ein Graph wird dabei aus einzelnen Segmenten zusammengesetzt, diese Segmente bestehen jeweils aus einer SDF. Dafür muss für jedes Fragment geprüft werden,

ob sich das Fragment in einem der Segmente befindet. Da es ineffizient wäre, alle Segmente für jedes Fragment zu prüfen, wird vorher idealer Weise eine Binärsuche vorgenommen um festzustellen, in welchem Segmentabschnitt auf der X-Achse das Fragment liegt.

5.7.1 Liniensegmente

Eine Möglichkeit ist es Liniensegmente, wie in Abbildung 5.16 und implementiert in Abbildung 5.11, zu nutzten.

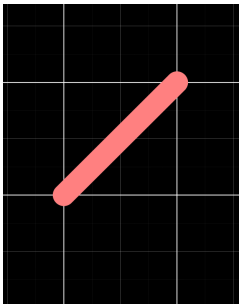

Abbildung 5.16: Liniensegment

Bei den Liniensegmenten ist eine einfache Binärsuche nicht möglich. Am Start- und Zielpunkt des Segments sieht man, dass das Segment über die X-Grenzen hinaus ragt. Das heißt, dass sich die Segmente überlappen, darauf gehe Ich in der Auswertung in Abschnitt ?? näher ein. Das schlimmste ist, dass sich die Überlappung beim vergrößern der Linienstärke mit vergrößert. Daher reicht eine einfache Binärsuche nicht und es muss überprüft werden wie viele Segmente sich überlappen können. Die Berechnung zu dieser Anzahl 'd' ist in Abbildung 5.17 gezeigt. Eine Überlappung mit den direkten Nachbarn gibt es dabei immer.

```
ddx = x_next-x;
int d = int(max(ceil(segment_width/ddx), 1))+1;
```

Abbildung 5.17: procedural_plot.glfs maximale Überlappungen an einer Seite

5.7.2 Parallelogrammsegmente

Eine andere Möglichkeit ist es Parallelogrammsegmente, wie in Abbildung 5.18 zu nutzen.

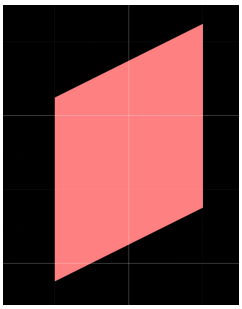

Abbildung 5.18: Parallelogrammsegment

Bei den Parallelogrammsegmenten funktioniert auch eine einfache Binärsuche.

6 Ergebnisse

Im Folgenden werden die Ergebnisse der zuvor beschriebenen Umsetzung Ausgewertet. Dabei wird insbesondere auf die, mit der Funktionalität des Plugins erreichten Ziele in der Aufgabenstellung eingegangen. Anschließend gibt es eine kurze Performance-Evaluation.

6.1 Software-Interface

Hier wird kurz beschrieben wie man die Funktionen des Plugins auch außerhalb des Demo-Interfaces nutzen kann.

Durch das befüllen der Vektoren 'graph_data' und 'graph_param' in der Hauptklasse, können beliebige Graphdaten geladen werden. Mit 'graph_param' lässt sich zudem die Färbung und relative Skalierung der Graphen zueinander anpassen. Als Referenz kann die Funktion 'gen_graph_data' genutzt werden, die die Demo-Daten generiert und damit die Vektoren befüllt.

Der Integer 'graph_count' stellt die Anzahl der Graphen dar, die aus den Vektoren tatsächlich dargestellt werden. Normalerweise wäre diese Anzahl genau die Zahl an Graphen die gerade in den Vektoren geladen sind.

Der 'segment_type' gibt an, welcher Segmenttyp genutzt wird, entweder 'LINE_SEGMENT' oder 'PARALLEL'. Ohne weitere Modifikationen sollte möglichst nur der Typ 'LINE_SEGMENT' genutzt werden, die Gründe werden in Abschnitt 6.3.1 erläutert.

Sämtliche weitere Einstellungen können mit den Slidern im Demo-Interface ausprobiert werden.

6.2 Nutzer-Interface

Das Nutzer Interface des Plugins nutzt wie zuvor beschrieben das interne GUI-System des CGVF. In Abbildung 6.1 ist die Nutzeroberfläche einmal dargestellt. Dieses Interface ist

sehr rudimentär und dient eher zum Testen der verschiedenen Funktionalitäten, als der endgültigen Nutzung. Es gibt Schalter zum aktivieren/deaktivieren von Tickmarks, dem Grid, den hervorgehobenen Hauptachsen, den Graphen und dem Hintergrund des Grids. Zudem gibt es Slider für andere Einstellungen. Die minimalen und maximalen Werte der Slider sind oft in einem 'Testbereich' gewählt, dass heißt viele Werte können auch außerhalb dieser Grenzen Liegen.

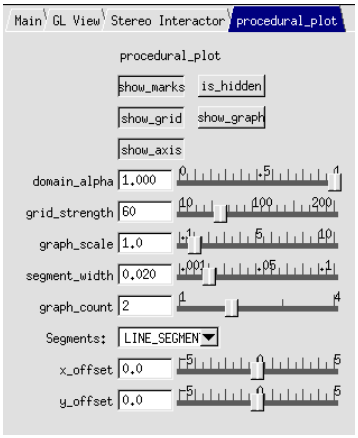

Abbildung 6.1: Nutzer-Interface

6.3 Graphen-Darstellung

Die generelle Darstellung von ununterbrochenen Graphen funktioniert zuverlässig und gut bei der Nutzung von Liniensegmenten. Ein Beispiel mit allen Testgraphen ist in Abbildung 6.2 zu sehen.

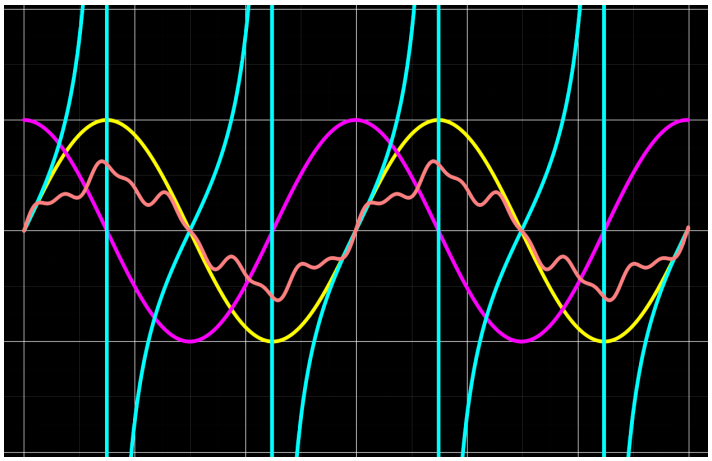

Abbildung 6.2: Test von vier Graphen mit Linien-Segmenten

6.3.1 Liniensegmente und Parallelogrammsegmente

Auch wenn bei der Umsetzung zwei verschiedene Segmenttypen zum Einsatz kamen, gibt es de facto nur einen Typ der wirklich funktioniert. Nur die Liniensegmente schaffen eine

einfache und problemfreie Darstellung. Im Folgenden wird auf die zwei größten Probleme mit der Alternative, den Parallelogrammsegmenten eingegangen.

Zunächst das Problem mit der gleichmäßigen Linienstärke. In den Abbildungen 6.3 und 6.4 werden jeweils die Sinus- und Kosinus-Kurven gezeigt. Bei der Nutzung von Liniensegmenten sieht man Graphen mit gleichmäßiger Zeichenstärke, doch bei Nutzung von Parallelogrammsegmenten sieht man eine ungleichmäßige Stärke. Dass kommt daher, dass sich ein Parallelogramm bei starkem Anstieg oder Abstieg, vertikal verzerrt. Zudem müssen die Seitenflächen an den Übergängen zu Nachbarsegmenten gleich zu denen der Nachbarn sein, damit der Graph keine Stufen hat. Ich habe während der Bearbeitungszeit keine Lösung dafür gefunden und bin mir auch nicht sicher ob es eine (effiziente)Lösung gibt.

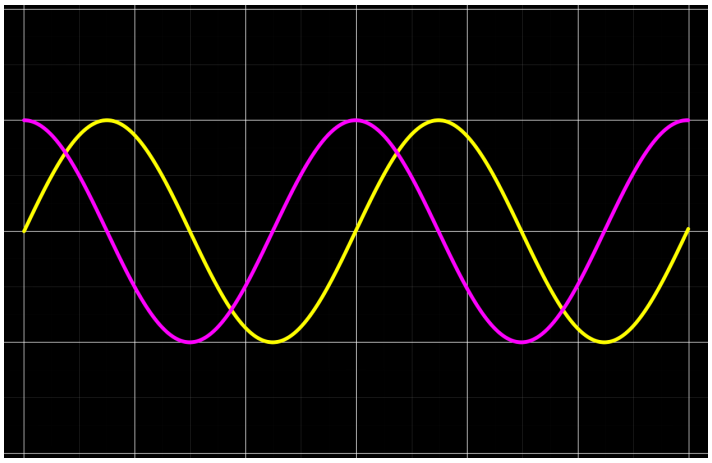

Abbildung 6.3: Test von zwei Graphen mit Linien-Segmenten

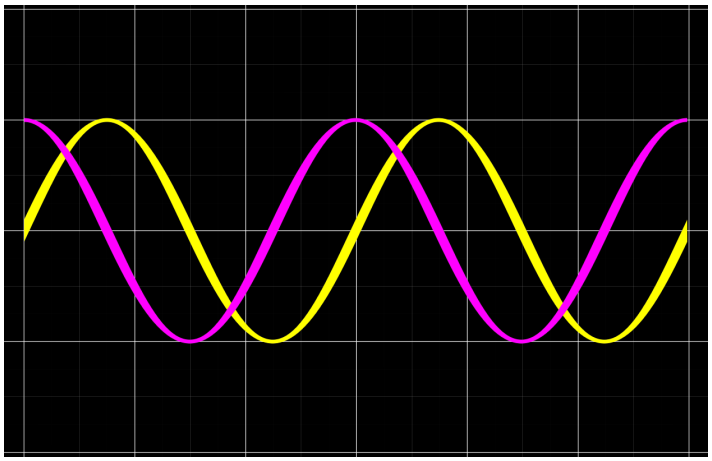

Abbildung 6.4: Test von zwei Graphen mit Parallelogramm-Segmenten

Ein weiteres Problem mit den Parallelogrammsegmenten ist ein Fehler, der die Seitenflächen leicht voneinander abweichen lässt. Das hat wahrscheinlich damit zu tun, dass eine Seite eine andere Höhe braucht als die Andere. Mit diesem Fehler konnte Ich mich noch gar nicht auseinandersetzen, wollte ihn aber dennoch erwähnen. Die Abbildung 6.5 und 6.6 zeigen deutlich das Problem und den Vergleich zur Nutzung der Liniensegmente.

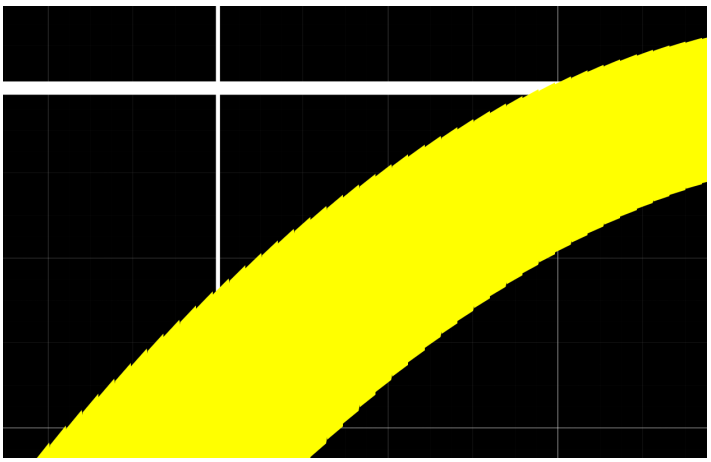

Abbildung 6.5: Parallelogramm-Segment vergrößert

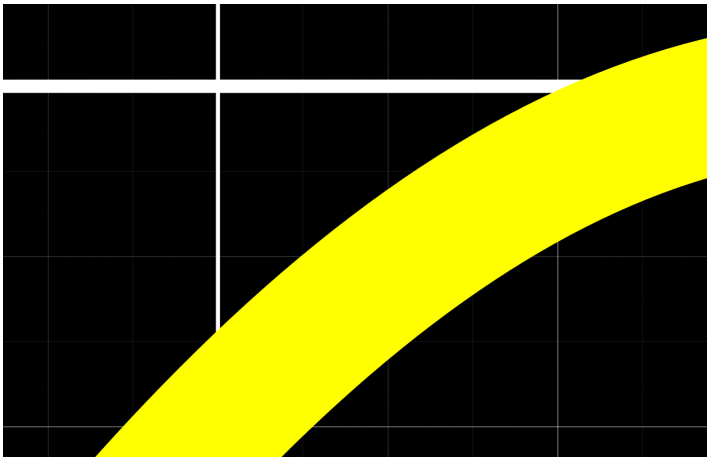

Abbildung 6.6: Linien-Segment vergrößert

6.4 Grid

Das statische Grid als wichtigst Dekoration bei der Plot-Visualisierung funktioniert ohne Probleme und hat genau eine Unterteilung pro Achseneinheit. Die Stärke des Grids kann angepasst werden und der Ursprung orientiert sich immer am Ursprung des Plot-Raums.

Das dynamische Grid funktioniert zwar, ist aber noch ein Prototyp. Wie in Abbildung 6.7 zu sehen, werden die Grid-Linien dünner mit feinerer Auflösung.

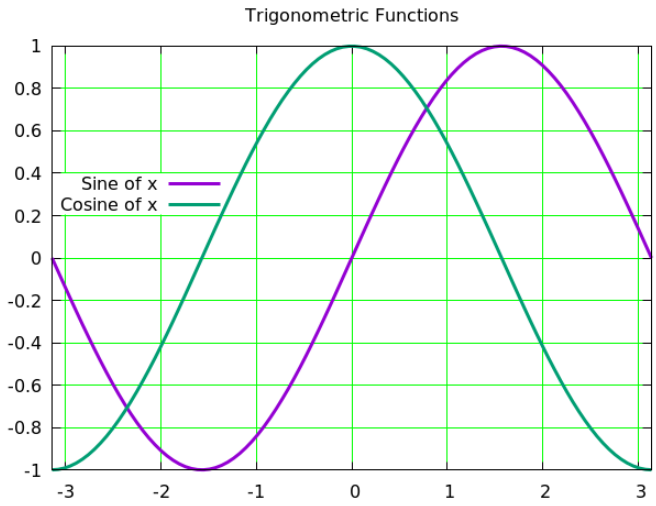

Abbildung 6.7: Dynamisches Grid

Die Sichtbarkeit der Ebenen hängt von der Position der Kamera im 3D-Raum ab. Ist die Kamera näher dran, sind auch mehr Linien sichtbar. Die maximale Anzahl der Ebenen ist auf momentan auf 9 beschränkt und Ebene mit der gröbsten Auflösung ist immer das statische Grid, mit einer Unterteilung pro Achseneinheit. Aufgrund dieser Einschränkungen betrachte Ich es als Prototyp.

6.5 Andere Dekorationen

6.6 Performance

Zuletzt in der Auswertung wird die Performance im Bezug auf das Darstellen von durchgängigen Graphen im Plot evaluiert.

6.6.1 SDF

Die beiden SDFs, die für die Graphensegmente genutzt werden stammen beide von Inigo Quilez [Qui22] und soweit Ich dass beurteilen kann, sind Beide optimal und Sie lassen sich nicht weiter vereinfachen. Das von mir testweise Erstellte 'sdf_parallelogram_segment' ist spürbar Langsamer als die Implementation von Quilez, weswegen früh klar war, dass es sich nicht lohnt eigene SDF-Implementationen weiter zu erforschen.

6.6.2 Binärsuche bei Segmenten

Der größte Performance-Faktor beim Darstellen der Graphen ist die 'quasi' Binärsuche nach relevanten Segmenten. Meine Implementation der Suche ist sicherlich nicht optimal, funktioniert aber zuverlässig und ist erheblich schneller als das Verzichten auf eine Suchfunktion.

Bei Parallelogrammsegmenten ist die Suchzeit immer in der Klasse von $O(\log n)$. Bei den Liniensegmenten gibt es eine zusätzliche Abhängigkeit von der Linienstärke. Das kommt daher, dass sich die Liniensegmente an den Messpunkten in den Graph-Daten überlappen. Diese Überlappung wird größer, sobald die Linienstärke wächst. Ist die Linienstärke groß genug, kann ein Segmentrand sogar ein anderes Segment komplett in der X-Richtung überdecken. Da die anderen Segmente nicht auf der gesamten Fläche überdeckt werden, entstehen visuelle Artefakte wenn man überdeckte Segmente einfach weglässt. Daher muss berechnet werden, wie viele Segmente bei der Momentanen Linienstärke überlappt werden können; mehr dazu in Abschnitt 5.7. Bei jedem Schritt in der Suche muss jetzt in beide Richtungen geprüft werden, ob nicht noch Andere Segmente überlappen könnten. Bestenfalls erreicht die Binärsuche dabei eine Suchzeit in der Klasse von $O(3 * \log n)$, da direkte Nachbarsegmente immer überlappen. Schlimmstenfalls kommt es zu einer Suchzeit in der Klasse von $O(n^2)$. Dieser schlimmste Fall tritt aber in der realen Nutzung praktisch nie auf, da man mit einer so großen Linienstärke nichts mehr vom Graphen erkennen würde. Daher betrachte Ich diese Abhängigkeit nicht als ein großes Problem.

7 Fazit und Ausblick

Zusammenfassend lässt sich sagen das die Kernfrage aus der Motivation, ob die prozedurale Generierung von Plots im Fragment-Shader möglich und umsetzbar ist, mit einem deutlichen ja beantwortet wurde. Dennoch haben es viele Teilziele und optionale Ziele gar nicht oder nur angedeutet in die Arbeit geschafft. Am Ende war das Thema wohl einfach zu groß, bzw. mein Zeit-Management einfach nicht gut genug. Dass heißt aber auch, dass es noch viel Platz für weitere Arbeiten gibt, die auf die Erforschten Konzepte weiter eingehen.

Literatur

[QMK06] Zheng Qin, Michael D. McCool und Craig S. Kaplan. „Real-time texture-mapped vector glyphs“. In: *Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06*. ACM Press, 2006. DOI: 10.1145/1111411.1111433.

[Gre07] Chris Green. „Improved alpha-tested magnification for vector textures and special effects“. In: *ACM SIGGRAPH 2007 courses on - SIGGRAPH '07*. ACM Press, 2007. DOI: 10.1145/1281500.1281665.

[NH08] Diego Nehab und Hugues Hoppe. „Random-access rendering of general vector graphics“. In: *ACM Transactions on Graphics* 27.5 (Dez. 2008), S. 1–10. DOI: 10.1145/1409060.1409088.

[Rou14] Nicolas P. Rougier. „Antialiased 2D Grid, Marker, and Arrow Shaders“. In: *Journal of Computer Graphics Techniques* 3.4 (2014), S. 52. URL: https://hal.archives-ouvertes.fr/hal-01081592.

[Chl15] Viktor Chlumsky. „Shape decomposition for multi-channel distance fields“. Magisterarb. Czech Technical University in Prague, Mai 2015. URL: https://dspace.cvut.cz/bitstream/handle/10467/62770/F8-DP-2015-Chlumsky-Viktor-thesis.pdf.

[Phi20] Lee Phillips. *Gnuplot 5*. Second. Alogus Publishing (a division of the Alogus Research Corporation), 2020. ISBN: 978-0-692-92716-8. URL: https://alogus.com/publishing/.

[Chl21] Viktor Chlumsky. *Multi-channel signed distance field generator*. Dez. 2021. URL: https://github.com/Chlumsky/msdfgen.

[Gum21] Stefan Gumhold. *The Computer Graphics and Visualization Framework*. Juni 2021. URL: https://github.com/sgumhold/cgv.

[Qui22] Inigo Quilez. *Inigo Quilez Articles*. https://iquilezles.org/articles/. 2022. URL: https://iquilezles.org/articles/.

Abbildungsverzeichnis