Fachbereich 3: Mathematik und Informatik

Masterarbeit

A Programming Language for Procedural Content Generation in 3D Scenes

Timm Decker

Matrikel-Nr. 248 563 3

14. Oktober 2016

1. Gutachter: Prof. Gabriel Zachmann
2. Gutachterin: Prof. Dr. Ute Bormann

Timm Decker

A Programming Language for Procedural Content Generation in 3D Scenes

Masterarbeit, Fachbereich 3: Mathematik und Informatik

Universität Bremen, Oktober 2016

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 14. Oktober 2016

Timm Decker

Danksagung

An dieser Stelle bedanke ich mich bei allen, die mich bei der Erstellung dieser Masterarbeit unterstützt haben. Zunächst möchte ich mich bei Patrick Lange bedanken, der mir in persönlichen Treffen neue Anregungen für diese Masterarbeit gegeben hat. Weiterhin möchte ich sowohl bei Prof. Dr. Ute Bormann als auch bei Prof. Dr. Gabriel Zachmann für die Bereitschaft danken, diese Masterarbeit zu beurteilen. Ebenfalls bedanke ich mich bei allen Teilnehmern des monatlichen Kolloquiums des Fachbereichs Computergrafik und virtuelle Realtität für das Feedback während der Zwischenpräsentationen. Abschließend möchte ich mich noch bei Xizhi Li dafür bedanken, dass er mir Geometrie-Histogramme für die Evaluation erstellt hat.

Zusammenfassung

Die vorliegende Masterarbeit behandelt die Konzipierung und Implementierung einer prozeduralen API, die auf Basis einer Merkmalanalyse von Asteroiden spezifiziert und vorrangig zur Erstellung von Asteroiden entwickelt wurde. Zur Ermittlung der API-Funktionalitäten, werden die Asteroiden Vesta, Lutetia, Itokawa und der Komet 67P/Churyumov-Gerasimenko analysiert. Weiterhin wird eine Programmiersprache namens PCGL entwickelt, die zur Erstellung prozeduraler Modelle dient.

Zentrale Fragestellungen dieser Arbeit sind:

- Welche markanten Oberflächenmerkmale haben Asteroiden und wie können sie mit prozeduralen Algorithmen erzeugt werden?
- Welche Geometriemanipulationsmethoden muss die API zur Verfügung stellen, um realitätsnahe Asteroiden zu generieren?
- Wie lassen sich diese Operationen abstrahieren und in eine domänenspezifische Programmiersprache vereinen?

Es zeigte sich, dass Lindenmayer-Systeme zwar zur Erstellung von Geometrie genutzt werden können, dessen Nutzung jedoch kaum nennenswerte Vorteile gegenüber anderen Methoden ergab. Dennoch konnten realitätsnahe prozedurale Replikate von Vesta, Lutetia, Churyumov-Gerasimenko und Itokawa mittels der implementierten Programmiersprache PCGL und der dazugehörigen prozeduralen API hergestellt werden. Das zeigt, dass eine domänenspezifische Programmiersprache wie PCGL es erlaubt, realitätsnahe Asteroiden zu erstellen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Masterarbeit behandelt die Erstellung einer API und einer dazugehörigen domänenspezifischen Programmiersprache zur Erzeugung prozeduraler Inhalte in 3D-Anwendungen. Dabei wird anhand einer Demoapplikation gezeigt, dass mittels dieser API eine Vielzahl realitätsnaher Asteroiden erzeugt werden können, die aus einem Zusammenspiel verschiedener prozeduraler Algorithmen entstehen. Die entstehenden Asteroiden werden bezüglich ihrer Realitätsnähe analysiert.

Die prozedurale Synthese von Inhalten ist eine Thematik, die schon in diversen Gebieten Einzug fand. Ein mögliches Einsatzgebiet findet sich in der Spielentwicklung. Dort bieten prozedurale Algorithmen die Möglichkeit, Spielinhalte zu generieren, um sowohl eine große Variation zu bieten, als auch das Fehlen von manuell erstellten Inhalten zu kompensieren. Gerade Letzteres findet sich bei Spielen wieder, die nicht das nötige Projektbudget besitzen, um Level-Designer zu beschäftigen [Hen+13, S. 1:2]. Prozedurale Algorithmen können so als Hilfsmittel zur effizienten Erzeugung beliebig vieler gleichartiger aber dennoch verschiedener Modelle dienen. Die dafür zugrunde liegenden Algorithmen sind keine kompletten Neuentwicklungen, sondern bedienen sich beispielsweise folgender bestehender Algorithmen und Strukturen:

- Lindenmayer-Systemen
- zelluläre Automaten
- genetische Algorithmen
- Fraktalen

Prozedurale Algorithmen finden auch bei kommerziell erfolgreichen Spielen Anwendung. „The Elder Scrolls - Oblivion“ von Bethesda Studios generiert beispielsweise Wälder mithilfe des Third-Party-Toolkit „SpeedTree“[1]. In Spielen wie „Minecraft“ und „Elite: Dangerous“ werden vollständige Welten und Universen mithilfe von prozeduralen Algorithmen geschaffen. Dabei können diese Algorithmen nicht nur visuelle Inhalte erzeugen, sondern auch Verhaltensweisen von NPCs[2] erstellen und sogar Musikstücke komponieren.

Auch die Filmindustrie nutzt prozedurale Inhalte. Das prominenteste Beispiel ist die Gruppensimulation

[1] Siehe *http://www.speedtree.com/*
[2] Non Player Characters; zu deutsch: nicht vom Spieler gesteuerten Charakteren

MASSIVE[3], die aus wenigen einzelnen Statisten umfangreiche Armeen generiert und rendert. So konnten erstmals die Schlachten in der Trilogie „Lord of the Rings“ erstellt werden, bei der sonst erheblich mehr Schauspieler nötig gewesen wären.

Prozedurale Inhalte sind aber nicht nur im Entertainment-Bereichen gefragt. Um die Wirkung eines Gebäudes innerhalb einer Skyline abzuschätzen, gibt es Programme, die diese in prozedural erstellten Städten platzieren. Das aktuell bekannteste Programm, das solche Städte erzeugt, heißt „CityEngine“[4].

1.1 Motivation

Der Bedarf an immer detailreicheren Szenerien nimmt stetig zu. Die steigende Rechenleistung von Computern ermöglicht es, immer realistischere Simulationen zu erstellen. Dadurch wächst auch der Bedarf, Inhalte zu generieren, die den Simulationen oder auch Spielen gerecht werden. Jedoch begrenzen die zeitlichen Ressourcen eines Content-Designers die Quantität und Qualität von manuell erstellten Inhalten. Diese müssen daher andere Methoden nutzen oder neue Methoden in ihren Erstellungsprozess integrieren, um dieser Nachfrage gerecht zu werden.

Ein konkreter Fall eines solchen Bedarfs ist die Erstellung von realitätsnahen Asteroiden zur Nutzung in Simulationsszenarien. Dabei ist es notwendig, dass die entstehenden Asteroiden hinsichtlich ihrer Oberflächenbeschaffenheit und Form parametrisiert sind, um verschiedene Fälle in einer Simulation abdecken zu können. Neuere wissenschaftliche Arbeiten behandeln solche Simulationen und beschäftigen sich beispielsweise mit der Rosetta-Mission der ESA, bei der die Landung des Landers Philae auf dem Asteroiden zwar gelang, dessen angestrebter Landeplatz jedoch verfehlt wurde [Ash+16]. Philae landete an einer wenig beleuchteten Stelle und erhielt so zu wenig Sonnenlicht, um dessen Akkumulatoren vollständig zu laden. Mit Simulationen kann erforscht werden, wie zukünftige Missionen mit einem ähnlichen Ziel diesem Problem ausweichen können[5].

Da sich diese Lander in einer Distanz befinden, bei der eine Echtzeitkommunikation nicht möglich ist, müssen autonom Entscheidungen getroffen werden. Erschwerend kommt hinzu, dass die Oberfläche des Asteroiden bei Start der Mission teilweise oder gänzlich unbekannt sind. Um die Gefahr eines Fehlschlags der Mission zu minimieren, ist es möglich, eine Vielzahl von Szenarien zu erstellen, so dass verschiedenste Landungen simuliert werden können. Die automatische Erstellung von Asteroiden könnte der Simulation eine nahezu unendliche Menge von möglichen realitätsnahen Asteroiden bereitstellen.

Um dies zu erreichen, können Verfahren aus der prozeduralen Inhaltserstellung angewandt werden. Es existieren viele Algorithmen, die für diese Anforderung in Frage kommen. Es soll beantwortet werden, ob bestehende Algorithmen realitätsnahe Asteroiden erzeugen können.

[3] Siehe: *http://www.massivesoftware.com/film.html*
[4] siehe *http://www.esri.com/software/cityengine*
[5] vgl. dazu [TPR15]

1.2 Ziel der Arbeit

Im Zuge dieser Masterarbeit wird eine Idee spezifiziert, implementiert und evaluiert, wie die Erstellung von Asteroiden mithilfe von prozeduralen Algorithmen unterstützt und in eine bestehende Programmierumgebung integriert werden kann. Dafür wird eine Programmiersprache entwickelt, die die Fähigkeit besitzt, eine große Menge prozeduraler Inhalte zu erzeugen. Diese Programmiersprache greift auf eine ebenfalls zu entwickelnde API zu, die die Funktionalität zur Synthese prozeduraler Inhalte bereitstellt. Dabei wird die API so entwickelt, dass sie neben Asteroiden auch noch andere Anwendungsgebiete abbilden kann. Der Schwerpunkt wird auf die Erzeugung von Vertexdaten gelegt – Texturen und Belichtungseigenschaften werden nicht behandelt. Die nötige Detailstufe soll durch eine hohe Polygonenanzahl gewährleistet werden.

Die Programmiersprache kann mittels eines Transpilers in C++ transformiert werden, der ebenfalls innerhalb dieser Masterarbeit entwickelt wird.

Schwerpunkt dieser Masterarbeit ist die Erstellung eines realitätsnahen Modells eines Asteroiden. Auf Basis dieser Entwicklung werden abschließend folgende Forschungsfragen beantwortet:

- Welche markanten Oberflächenmerkmale haben Asteroiden und wie können sie mit prozeduralen Algorithmen modelliert werden?
- Welche Geometriemanipulationsmethoden muss die API zur Verfügung stellen, um realitätsnahe 3D-Modelle von Asteroiden zu generieren?
- Wie lassen sich diese Operationen abstrahieren und in eine domänenspezifische Programmiersprache vereinen?

Zur Demonstration werden Asteroidenmodelle mit der Programmiersprache PCGL definiert. Trotz wenig Quellcode, liegt der Fokus auf der Erstellung detailreicher und variabler Modelle, die hohe Ähnlichkeit mit realen Asteroiden und Kometen aufweisen (vgl. dazu Abbildung 1.1 und Abbildung 1.2).

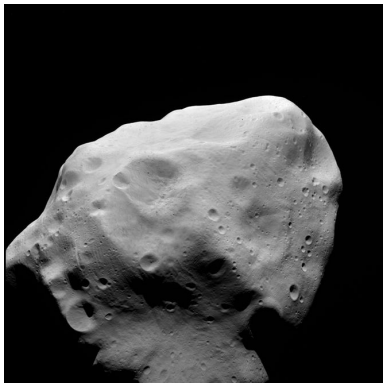

Abbildung 1.1 Asteroid Lutetia
Quelle: [Cho14]

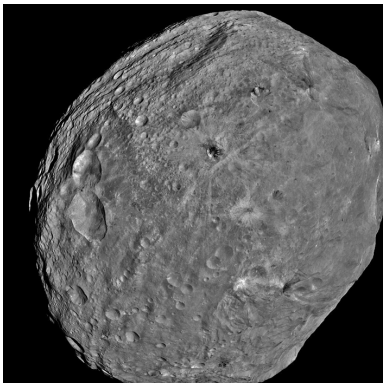

Abbildung 1.2 Asteroid Vesta
Quelle: [Gre11]

1.3 Aufbau der Arbeit

Nachdem in diesem Kapitel auf die grundlegende Motivation und das Ziel dieser Arbeit eingegangen wurde, werden im nächsten Kapitel prozedurale Ansätze in der Computergrafik thematisiert. Dazu wird anfangs auf den Begriff „prozedural“ eingegangen und dieser im Kontext der Computergrafik kurz erläutert. Dann wird anhand eines Beispiels erklärt, wie prozedurale Inhalte algorithmisch erzeugt werden können. Dafür werden Lindenmayer-Systeme eingeführt und dargestellt, wie diese genutzt werden können, um Geometrie zu erzeugen. Danach folgt mit der Vorstellung der wissenschaftlichen Arbeiten, der Schwerpunkt des Kapitels. Hier wird die Nutzung der Lindenmayer-Systeme zur Geometrierzeugung erklärt. Da ein Lindenmayer-System nicht das einzige Verfahren ist, mit dem prozedurale Algorithmen abgebildet werden können, werden noch weitere Verfahren aufgezeigt. Die anderen Verfahren werden hinsichtlich ihrer Anwendbarkeit zur Erstellung von Modellen von Asteroiden untersucht.

Das dritte Kapitel „Anforderungsanalyse“ ist in zwei Abschnitte, API und PCGL — die domänenspezifische Programmiersprache — aufgeteilt. Im Abschnitt „API“ werden reale Asteroiden merkmalspezifisch sondiert. Weiterhin wird untersucht, durch welchen prozeduralen Algorithmus verschiedene Merkmale repliziert werden können. Algorithmen, die später auch in der Implementierung Anwendung finden, werden dort genauer beschrieben. Der Abschnitt „PCGL“ beinhaltet die Spezifikation der Programmiersprache PCGL, die die Nutzung der API vereinfachen und die algorithmische Grundstruktur der Lindenmayer-Systeme hervorhebt.

Im vierten Kapitel „Implementierung“, werden Implementierungsentscheidungen der einzelnen Komponenten erläutert. Dazu gehört der Abschnitt „Transpiler“, der den Transpiler der Programmiersprache PCGL erklärt, sowie der Abschnitt „API“, der die Implementierungsentscheidungen bei der Generierung prozeduraler Inhalte beschreibt.

Das vorletzte Kapitel „Evaluation“ untersucht die entwickelten Komponenten hinsichtlich ihrer Eignung zur Erstellung von Asteroiden. Bei den zu evaluierenden Merkmalen wird zwischen der neu konzipierten und implementierten Programmiersprache PCGL und der Umsetzung eines Asteroiden mittels der entwickelten prozeduralen API unterschieden.

Am Ende dieser Arbeit folgt das Fazit, das die Ergebnisse der Evaluation zusammenträgt und die Nutzung von Lindenmayer-Systemen zur Erzeugung von Asteroiden bewertet.

1.4 Konventionen

In dieser wissenschaftlichen Arbeit wird die weibliche Form mit der männlichen Form eines Wortes gleichgestellt. Aus Gründen der Vereinfachung wird jedoch ausschließlich auf die männliche Form zurückgegriffen. Sofern es eine sachliche Form gibt, wird diese verwendet.

Da im fachlichen Umfeld dieser wissenschaftlichen Arbeit viele Fachbegriffe nur auf Englisch bekannt sind, werden diese nicht ins Deutsche übersetzt.

Abkürzungen werden bei der ersten Benutzung ausgeschrieben und als solche gekennzeichnet. Danach folgende Verwendungen treten dann in der abgekürzten Version auf.

Kapitel 2

Prozedurale Ansätze in der Computergrafik

Der Begriff „prozedural“ bezeichnet im Kontext der Computergrafik ein Vorgehen, das algorithmisch Inhalte synthetisiert [Tog+11]. Die Fachliteratur diskutiert verschiedene Ansätze, mit denen prozedurale Inhalte synthetisiert werden können. Diese werden in diesem Kapitel nach kurzer Behandlung der Grundlagen vorgestellt. Alle Ansätze haben die Gemeinsamkeit, dass sie eine Menge von Daten mittels parametrisierter Algorithmen transformieren und daraus eine neue Menge Daten generieren. Dieser Schritt kann mit den neu generierten Daten mehrfach mit den selben oder neuen Algorithmen wiederholt werden, bis die gewünschte Ausgabemenge an Daten erstellt wurde.

Der gewünschte Effekt bei prozeduralen Algorithmen ist die scheinbare Erweiterung des Informationsgehalts. Dieses wird in der Fachliteratur als „data amplification“[1] bezeichnet [Ebe03, S. 312].

Ein bekanntes Beispiel, um diesen Effekt zu verdeutlichen, ist die Koch-Flocke. Als Eingabemenge dient hierbei ein simples Dreieck, das aus drei Strecken besteht. Nun wird ein einfacher Algorithmus angewandt: Ersetze in jedem Iterationsschritt alle Strecken durch vier neue Strecken, wie sie in Abbildung 2.1 abgebildet sind. Durch diesen simplen prozeduralen Algorithmus wird nach 4 Iterationsschritten schon eine Struktur geschaffen, die einen sehr detaillierten Kantenverlauf aufweist.

Das Muster, das die Koch-Flocke erzeugt, ist ein „Fraktal“. Als Fraktale werden in der Mathematik Strukturen bezeichnet, die selbstähnlich sind. Der Begriff des Fraktales wurde erstmals 1975 vom Mathematiker Benoît Mandelbrot genutzt [Man87, S. 14].

Ein Fraktal, das eine hohe praktische Relevanz besitzt, ist der „Perlin Noise“-Algorithmus. Es generiert Noise[2], dessen Auflösung mehrfach verringert und auf das Ausgangsrauschen moduliert wird.

[1] engl. für Datenverstärkung
[2] engl. für Rauschen

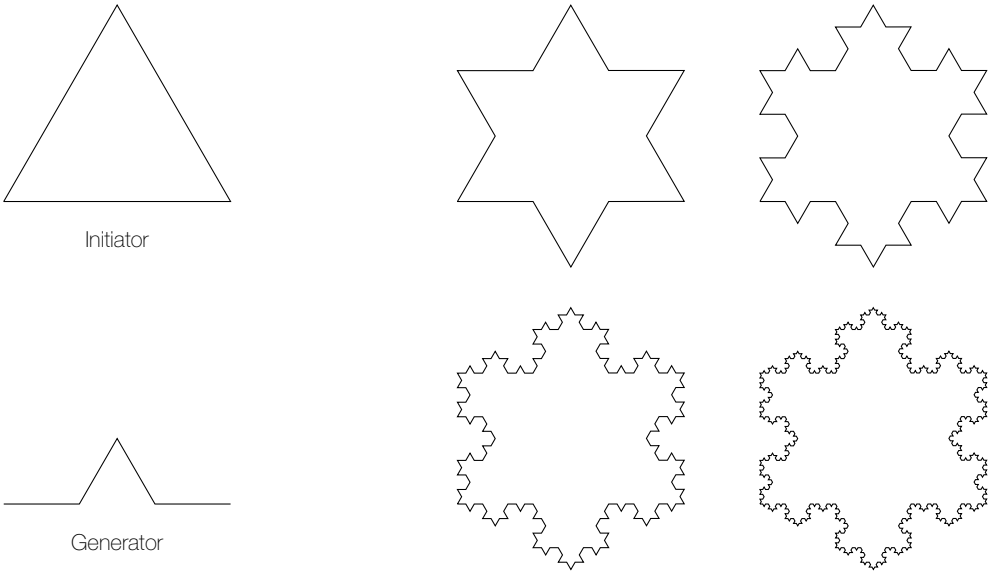

Abbildung 2.1 Konstruktion einer Koch-Kurve

2.1 Lindenmayer-Systeme

Die Konstruktion von Fraktalen lässt sich mittels „Lindenmayer-Systemen“ formalisieren.

Ein Lindenmayer-System (kurz: L-System) ist ein formalisiertes Termersetzungsverfahren. Es wurde 1968 von dem ungarischen Biologen Aristid Lindenmayer erfunden, um Pflanzenwachstum und –strukturen formal zu beschreiben. Sie lassen sich auch erweitern, um Polygondaten zu erzeugen. L-Systeme bestehen aus drei Komponenten [PL90, S. 4]:

1. Eine Menge von Symbolen (aufgeteilt in Terminal- und Nichtterminalsymbolen).
2. Eine Kette von Symbolen, mit denen gestartet wird – das Axiom [PL90, S. 5].
3. Eine Menge von Regeln, die auf die Symbole angewandt werden können.

Die einfachste Anwendung von L-Systemen ist die Entwicklung von Zeichenketten [PL90, S. 6]. Im Folgenden werden die verschiedenen Arten von L-Systemen beschrieben und deren Unterschiede erläutert.

Kontextfreie L-Systeme:

> **Beispiel 1**:
> Symbolmenge: `A, B, C`
> Axiom: `A`
> Produktionsregeln: `A` $\rightarrow$ `BB`, `B` $\rightarrow$ `ABC`

Kontextfreie L-Systeme stellen die einfachste Art dar. In Beispiel 1 werden zwei einfache Produktionsregeln aufgezeigt. Bei der ersten wird das Symbol `A` durch zwei `B` ersetzt. Bei der zweiten Produktionsregel wird das Symbol `B` durch die drei Symbole ABC ersetzt.

Die Reihenfolge, in der die Produktionsregeln angewandt werden, ist so wie die Anzahl an Anwendungen nicht definiert, sodass verschiedene Ausgaben bei der gleichen Eingabe entstehen könnten. Um dies zu verhindern, kann das L-System so modifiziert werden, dass die auszuführende Produktionsregel eindeutig ist und dass keine zirkulären Abhängigkeiten entstehen.

Kontextsensitive L-Systeme:

Beispiel 2:
Symbolmenge: `A, B`
Axiom: `A`
Produktionsregeln: `A` $\rightarrow$ `BBB`, `B[B]B` $\rightarrow$ `A`

Kontextsensitive L-Systeme erweitern kontextfreie L-Systeme hinsichtlich ihrer Produktionsregeln. In Beispiel 2 ist zu erkennen, dass sich in der zweiten Produktionsregel eine Klammer um das mittlere Symbol `B` befindet. Diese Klammerung spezifiziert den Ersetzungsrahmen. Die darum liegenden Symbole dienen dazu, die Produktionsregel nur anzuwenden, wenn diese vorhanden sind[3] [PL90, S. 30].

Somit kann sich das L-System aus Beispiel 2 wie folgt entwickeln:

$$\texttt{A} \xrightarrow[Regel\ 1]{(1)} \texttt{BBB} \xrightarrow[Regel\ 2]{(2)} \texttt{BAB} \xrightarrow[Regel\ 1]{(3)} \texttt{BBBBB}$$

Aristid Lindenmayer unterteilt L-Systeme anhand ihrer Kontextsensitivität. So ist das Beispiel 1 ein 0L-System, da es weder einen Links- noch Rechtsbezug zu einem Symbol hat. Das Beispiel 2 ist ein 2L-System, da es in mindestens einer Produktionsregel sowohl einen Bezug zu einem Symbol links und rechts vom Ersetzungssymbol herstellt [PL90, S. 30].

0L-, 1L- und 2L-Systeme lassen sich bezüglich ihrer Anwendung der Produktionsregeln weiter spezifizieren.

Stochastische L-Systeme:

Beispiel 3:
Symbolmenge: `A, B, C`
Axiom: `A`
Produktionsregeln: `A` $\rightarrow$ `BB`, `B` $\xrightarrow{50\ \%}$ `A`, `B` $\xrightarrow{50\ \%}$ `C`

Stochastische L-Systeme erweitern die Produktionsregeln von L-Systemen um Wahrscheinlichkeiten, mit denen eine Produktionsregel ausgeführt wird. In Beispiel 3 ist dies für die zweite und dritte Regel jeweils 50 %. Dabei muss laut Lindenmayer eine der Regeln ausgeführt werden, die die jeweilige Ausgangssymbolkette besitzt [PL90, S. 28].

Parametrisierte L-Systeme:

Beispiel 4:
Symbolmenge: `A`

[3] also im Kontext zu diesen Symbolen stehen

Axiom: `A(10)`
Produktionsregeln: `A(x)` : $x > 0 \rightarrow$ `A(x-1)A(x-1)`

Parametrisierte L-Systeme können sowohl kontextfreie als auch kontextsensitive L-Systeme um Bedingungen erweitern. Die definierte Bedingung muss für die Eingabevariablen erfüllt sein, bevor die Produktionsregel angewandt wird [PL90, S. 40]. In Beispiel 4 ist dies für das Axiom `A(10)` und die einzige Produktionsregel neun Mal der Fall, bevor $x = 0$ wird und keine Produktionsregel mehr angewandt werden kann.

2.2 L-Systeme zur Erzeugung von Geometrie

Durch die Anwendung einzelner Regeln eines Lindenmayer-Systems entsteht eine Hierarchie, die auch bei 3D-Modellierungs-Programmen zu finden ist. Dadurch lassen sich komplexe Geometrien in Schritten mittels einfacheren Grundobjekten aufbauen und manipulieren. Um nun konkrete Geometrien aufbauen zu können, ist es nötig, eine Interpretation für Lindenmayer-Systeme zu finden, die dies ermöglicht. Dafür eignet sich die „Turtle Interpretation“.

Bei dieser Interpretation wird zu dem Lindenmayer-System noch der Status der „Turtle“ vorgehalten. Das Lindenmayer-System wird um Befehle erweitert, die den Status der „Turtle“ verändern. Einige Umsetzungen dieser Interpretation implementieren zusätzlich einen Stapel von Status, auf den einzelne Status zwischengespeichert werden können.

Nachfolgend werden wissenschaftliche Arbeiten vorgestellt, die die „Turtle Interpretation“ nutzen und damit Geometrie erzeugen. Zudem wird gezeigt, warum Lindenmayer-Systeme sich für den jeweiligen Anwendungsfall eignen.

Prusinkiewicz et al. nutzen Lindenmayer-Systeme zur Abbildung konventioneller Algorithmen in der Geometrie. Dazu gehört beispielsweise der de-Casteljau-Algorithmus[4]. Prusinkiewicz et al. erkennen drei Eigenschaften von Lindenmayer-Systemen, die vorteilhaft für die Erzeugung von Geometrie sind [PSS10]:

- L-Systeme spezifizieren intuitiv Entwicklungen auf Basis zeitlicher und räumlicher Termersetzungsregeln, die entweder kontextfrei oder kontextsensitiv sein können.
- L-Systeme beschreiben wachsende Strukturen hinsichtlich topologischer Relationen zwischen einzelnen Strukturkomponenten. Die Relationen bleiben automatisch erhalten, wenn neue Komponenten in die Struktur eingefügt werden.
- L-Systeme referenzieren einzelne Komponenten mittels Typ, Status und Kontext und nicht mittels Position innerhalb einer Struktur.

Auch andere wissenschaftliche Ausarbeitungen haben gezeigt, dass sich Lindenmayer-Systeme für die Erzeugung von Geometrie eignen.

[4] Der de-Casteljau-Algorithmus berechnet eine Annäherung einer Bézier-Kurve.

Měch et al. beschreiben in ihrem Paper „Extensions to the graphical interpretation of L-systems based on turtle geometry“ eine Möglichkeit, wie bestehende L-Systeme um geometrieerzeugende Funktionen erweitert werden können. Dazu werden bei Anwendung der Produktionsregeln eines L-Systems Kommandos eingefügt, die Kugeln oder Zylinder erzeugen, oder die die Position der „Turtle“ anpassen [MPH97]. Zu der Geometrieerzeugung gehört auch die Berechnung von Texturkoordinaten und die Festlegung der Materialien. Texturen werden von Měch et al. nicht mit dieser Methode erzeugt.

Der Anwendungsfall von Měch et al. ist die Erstellung von organischen Strukturen. Dazu gehören neben dem eigentlichen Zweck von Aristid Lindenmayer, der Erstellung von Pflanzen, auch andere organische Objekte, wie beispielsweise Muscheln. Daher begrenzten Měch et al. die zur Verfügung stehenden Grundobjekte auf kugel- und zylinderförmige Strukturen.

Hanan beschrieb 1992 eine ähnliche Methodik zur Erstellung von Geometrien mithilfe von Lindenmayer-Systemen [Han92]. Dabei baute er auf der Arbeit von Szilard und Quinton auf, die den Symbolen im Grundalphabet eines L-Systems Befehle zugeordnet haben, die zu Grafikausgaben führten[SQ79]. Hanan erweitert die Erzeugung von einfachen Liniensegmenten, wie sie Szilard und Quinton beschrieben, um die Erzeugung von 3D-Oberflächen.

Nun werden wissenschaftliche Ausarbeitungen beschrieben und hinsichtlich ihres Anwendungsfalls analysiert. Dabei werden nur jene wissenschaftliche Ausarbeitungen betrachtet, die Lindenmayer-Systeme oder ähnliche Systeme als Grundstruktur nutzen. Es wird erläutert, inwiefern sich die jeweilige Arbeit zur Erstellung von Asteroiden eignet.

2.2.1 L-Systems in geometric modeling

Prusinkiewicz et al. beschreiben in ihrem Paper „L-Systems in geometric modeling“ die Nutzung von Lindenmayer-Systemen zur Erstellung von Bézier-Kurven mittels des de-Casteljau Algorithmus [PSS10]. Die L-Systeme, die benutzt werden, sind kontextsensitiv und parametrisiert. Dabei erweiterten sie die Parameter von einfachen Zahlen auf Vektoren, um die Position von Punkten in einem kartesischen Raum nutzen zu können. Mittels angepassten Produktionsregeln konnte der de-Casteljau-Algorithmus mit wenigen Produktionsregeln repliziert werden. Sie kommen zu dem Schluss, dass L-Systeme der verbalen Beschreibung eines Algorithmus entsprechen und eine intuitive Darstellung der mathematischen Grundlagen mittels L-Systeme umsetzbar ist.

2.2.2 Geometric programming for computer aided design

Das Fachbuch „Geometric programming for computer aided design“ von Paoluzzi und Pascucci beschreibt die Programmiersprache PLaSM.

Die Programming Language for Solid Modeling (kurz: PLaSM) ist eine Sprache zur Beschreibung von Geometrie. Es existiert eine Version, die auf der funktionalen Programmiersprache FL[5] basiert, und eine Version, die das Python-Ökosystem nutzt. Dabei werden Geometriedaten direkt erzeugt und können durch diverse Funktionen zusammengefügt werden. Diese Funktionen sind parametrisiert, um mehrere ähnliche Objekte erzeugen zu können [PP03].

Damit ähnelt der Ansatz von PLaSM den parametrisierten Lindenmayer-Systemen sehr.

Als Beispiel sei folgendes Skript gegeben, das in PLaSM auf Basis von FL geschrieben ist und Abbildung 2.2 als Ausgabe erzeugt.

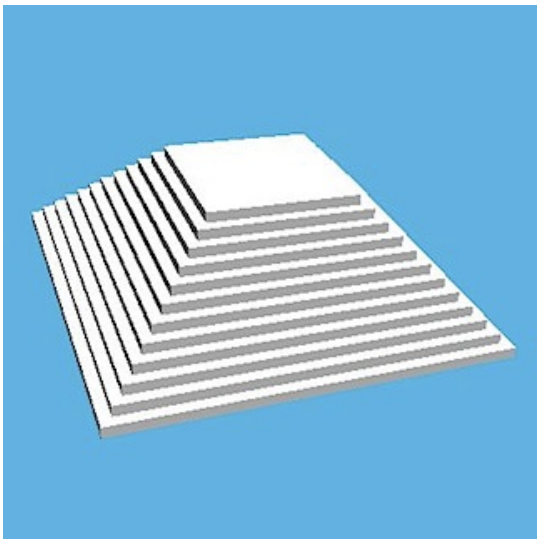

Abbildung 2.2 Eine mit PLaSM erstellte Szene
Quelle: [Pao+16]

```
DEF AztecaPyramid (hStep::IsReal; side,nStep::IsInt) =
  (T:<1,2>:<side/2,side/2> ~ STRUCT ~ CAT~DISTR):
  <(((CONS~AA:(S:<1,2>)):((TRANS~[ID,ID]):ScalingParams)):
    ScaledBox, T:3:hStep>
WHERE
  ScalingParams = (AA:/ ~ DISTR): <REVERSE:((side-nStep+1)..side),side>,
  ScaledBox = T:<1,2>:<-:(side/2),-:(side/2)>:basis,
  basis = CUBOID:<side,side,hStep>
END;

AztecaPyramid:<0.5,18,12>
```

Der eigentliche Anwendungsfall von PLaSM ist die Erzeugung von Architekturgeometrie [6]. Der Autor ist der Auffassung, dass die Syntax für Programmierer, die eher objektorientierte Programmiersprachen gewohnt sind, gewöhnungsbedürftig ist. Es ist nicht intuitiv erkennbar, woher die verschiedenen Instanzen der Pyramidenstufen herkommen. Von einer Anwendung von PLaSM zur Erzeugung von Asteroiden wird deshalb abgesehen.

[5] Abk. für programming at Function Level
[6] Siehe auch *http://www.plasm.net/docs/tutorial/*

2.2.3 HyperFun project: a framework for collaborative multidimensional F-rep modeling

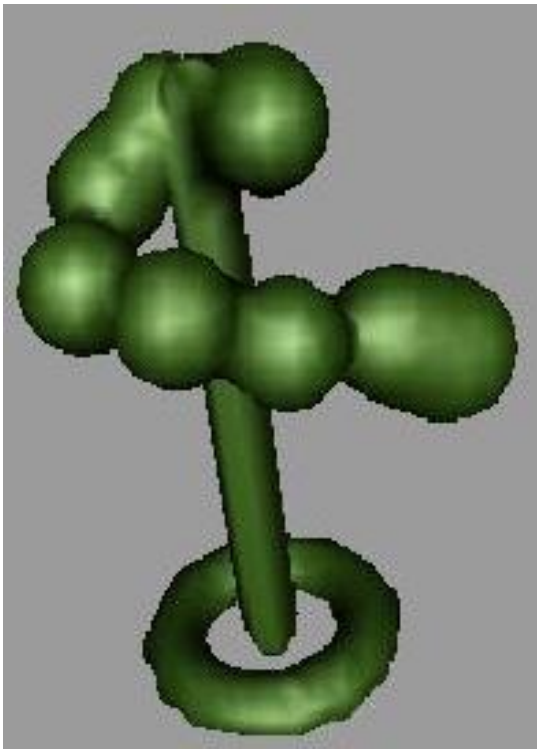

Abbildung 2.3 Ein mit HyperFun erstelltes Mesh
Quelle: [Adz+99]

Das HyperFun Projekt nutzt – wie PLaSM – ebenfalls einen funktionalen Ansatz, Geometrie zu erzeugen. Dieser wird jedoch nicht an die Programmiersprache geknüpft, sondern an die Art, wie Geometrie erzeugt wird. Geometrien werden nicht explizit mit Vertexdaten, sondern implizit per Funktionsdefinition spezifiziert. Es nutzt die sogenannte Funktionale Repräsentation (kurz: FRep) zur Definition von Geometrien [Adz+99].

Bei der funktionalen Repräsentation werden einfache Objekte mittels mathematischen Operationen verbunden, um komplexere Gebilde zu erzeugen. In Abbildung 2.4 ist eine Kugel mit dem Radius r und dem Mittelpunkt $(0, 0, 0)$ als Funktion definiert.

Zum Erhalten von Polygondaten, muss eine Grenze definiert werden, an der die Flächen für die Geometrie erzeugt werden. HyperFun definiert diese Grenze bei 0, jedoch sind auch andere Grenzwerte auswählbar. Das bedeutet, dass die Menge an Punkten, bei denen $f(x, y, z) = 0$ erfüllt ist, das Modell bilden.

$$f(x, y, z) = \sqrt{x^2 + y^2 + z^2} - r$$

Abbildung 2.4 Eine funktional repräsentierte Kugel

HyperFun beschränkt sich bei der funktionalen Repräsentation nicht nur auf die Nutzung von drei Parametern. Adzhiev et al. nutzen einen vierten Parameter, um die Zeit als Abhängigkeit in die Geometrie zu integrieren. Die erweiterte Bedingung lautet dann $f(x, y, z, t) = 0$.

Folgend ein Skript in HyperFun, das Abbildung 2.3 generiert hat.

```
--This HyperFun program consists of one object:
--union of superellipsoid, torus and soft object

my_model(x[3], a[1])
{
array x0[9], y0[9], z0[9], d[9], center[3];
x1=x[1];
x2=x[2];
x3=x[3];

-- superellipsoid by formula
superEll = 1-(x1/0.8)^4-(x2/10)^4-(x3/0.8)^4;

-- torus by library function
center = [0, -9, 0];
torus = hfTorusY(x,center,3.5,1);

-- soft object
x0 = [2.,1.4, -1.4, -3, -3, 0, 2.5, 5., 6.5];
y0 = [8, 8, 8, 6.5, 5, 4.5, 3, 2, 1];
z0 = [0, -1.4,-1.4, 0, 3, 4, 2.5, 0, -1];
d = [2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.7, 3];
sum = 0.;
i = 1;

while (i<10) loop
    xt = x[1] - x0[i];
    yt = x[2] - y0[i];
    zt = x[3] - z0[i];
    r = sqrt(xt*xt+yt*yt+zt*zt);

    if (r <= d[i]) then
        r2 = r*r; r4 = r2*r2; r6 = r4*r2;
        d2 = d[i]^2; d4 = d2*d2; d6 = d4*d2;
        sum = sum + (1 - 22*r2/(9*d2) +
        17*r4/(9*d4) - 4*r6/(9*d6));
    endif;

    i = i+1;
endloop;

soft = sum - 0.2;

-- final model as set-theoretic union
my_model = superEll | torus | soft;
}
```

Da HyperFun ausschließlich mit der funktionalen Repräsentation arbeitet, müssen alle Modelle so spezifiziert werden. Dies erschwert die Modellierung komplexerer Geometrien. Direkte Manipulationen an der Geometrie sind nicht möglich, sondern müssten umständlich durch Mengenoperationen abgebildet werden. Durch diese Komplexität kann HyperFun nur mit hohem Aufwand zur vollständigen Erzeugung von Asteroiden genutzt werden.

2.2.4 SpeedRock: procedural rocks through grammars and evolution

Das Paper „SpeedRock: procedural rocks through grammars and evolution“ von Dart et al. beschreibt ein Verfahren zur Erzeugung von Gesteinsbrocken mittels Lindenmayer-Systemen [DDT11]. Dafür definieren sie ein Lindenmayer-System mit Würfeln, das einfache, zufällig gewählte Ersetzungsregeln nutzt. Das L-System wird dann auf einen initialen Würfel angewandt, der in acht kleinere Würfel gemäß den Ersetzungsregeln aufgeteilt wird. Dieser Vorgang wird mit den neu entstandenen Würfeln solange wiederholt, bis die gewünschte Iterationstiefe erreicht wurde. Dadurch können Hohlräume entstehen, die bei SpeedRock nicht erwünscht sind. Diese werden im nächsten Schritt aufgelöst, indem alle Würfel in Richtung des Zentrums des Gesteinsbrockens geschoben werden, sofern ein Leerraum existiert. Der darauf folgende optionale Schritt dient der Simulation von Erosion. Je nachdem, wie viele Nachbarwürfel ein Würfel hat, wird entschieden, ob dieser Würfel entfernt wird. Als letzten Schritt werden die Würfel genutzt, um die endgültige Form des Gesteinsbrockens herzustellen. Eine Kugel wird um die Menge von Würfeln erzeugt, dessen Vertices auf die Oberfläche der Würfelmenge projiziert werden.

Das Verfahren eignet sich, um die Grundform eines Asteroiden zu produzieren. Merkmale, wie beispielsweise Krater und konkave Formen, lassen sich damit jedoch nicht bilden.

2.3 Weitere Verfahren

Neben Lindenmayer-Systemen existieren noch weitere Verfahren, die zur Erstellung prozeduraler Geometrien genutzt werden können. Es werden nun Arbeiten vorgestellt, die Gesteinsbrocken ohne Nutzung von Lindenmayer-Systemen erstellen.

2.3.1 Procedural Generation of Rock Piles using Aperiodic Tiling

Im Paper „Procedural Generation of Rock Piles using Aperiodic Tiling“ von Peytavie et al. wird ein Verfahren zur aperiodischen Verteilung und Erzeugung von Gesteinsbrocken beschrieben [Pey+09].

Der von Peytavie et al. entwickelte Algorithmus kann in vier Schritten beschrieben werden:

1. Erstelle eine Menge von 2^8 *corner cubes*[7], die in ein Gitter von Zellen aufgeteilt werden. Jede Zelle wird als C_{ijk} bezeichnet. Für jede Zelle wird eine Menge von *seed points* generiert, die p_k bezeichnet werden.
2. Erstelle mithilfe von Voronoï-Zellen ein Volumen V_k für jeden *seed point* p_k.
3. Wende einen einen Erosionsalgorithmus auf die erstellten Voronoï-Zellen an.
4. Erstelle Polygone anhand der impliziten Darstellung der erodierten Voronoï-Zellen.

[7] siehe dazu [LD06]

Die Nutzung dieses Algorithmus ist lediglich für Asteroiden des Typs *rubble pile* anzuwenden. *Rubble pile* Asteroiden bestehen aus mehreren kleineren Geröllteilen, die von der Gravitation lose zusammengehalten werden. Im Gegensatz dazu stehen monolithische Asteroiden, die aus einem einzigen verbundenen Stein bestehen. Eine Erweiterung des Algorithmus, um einen monolithischen Asteroiden zu erzeugen, ist möglich, indem der Teil des Algorithmus, der zur Erzeugung eines Geröllteils dient, für den gesamten Asteroiden genutzt wird. Damit wird die Generierung jedoch auf eine bestimmte Form von Asteroiden beschränkt.

2.3.2 Procedural Modeling of Multiple Rocks Piled on Flat Ground

Das Paper „Procedural Modeling of Multiple Rocks Piled on Flat Ground“ von Sakurai und Miyata beschreibt ein Verfahren, um die Verteilung von Gesteinsbrocken auf einer Fläche zu replizieren [SM10]. Zusätzlich wird auch auf die Generierung dieser Gesteinsbrocken eingegangen. Diese werden mithilfe eines dreidimensionalen Voronoï-Diagramms erzeugt. Eine Zelle in diesem Voronoï-Diagramm entspricht einem Gesteinsbrocken. Dessen Polygone werden in einem letzten Schritt mittels *polygon subdivision* noch unterteilt, sodass der Stein eine höhere Anzahl an Flächen aufweist. Diese Gesteinsbrocken werden dann zufällig in verschiedenen Schichten auf einer Ebene verteilt.

Der Anwendungsfall dieses Papers beschränkt sich ausschließlich auf die Verteilung von Gesteinsbrocken. Der Algorithmus zur Erzeugung dieser Brocken ist im Bezug auf die Generierung von Asteroiden nur zur Erstellung der Grundform benutzbar. Die Verteilung kann zur Erzeugung von *rubble pile* Asteroiden genutzt werden, sofern ein volumetrisches Modell einer Asteroidengrundform zur Verteilung der Geröllteile zur Verfügung steht.

2.3.3 Multi-Scale Modeling and Rendering of Granular Materials

Im Paper „Multi-Scale Modeling and Rendering of Granular Materials“ von Meng et al. wird ein Verfahren entwickelt, mit dem komplexe Geometrien erzeugt und gerendert werden können. Diese Geometrien beschränken sich auf Modelle, die aus granularen Materialen, wie z.B. Sand oder Zucker bestehen [Men+15]. Dabei wird eine Menge von Körnern erzeugt, die beim Rendering genutzt werden. Die Verteilung dieser Körner basiert auf einem vorher erzeugten Voxelmodell. Der im Paper vorgestellte Algorithmus versucht dabei nicht nur die zufällige Verteilung der Körner zu gewährleisten, sondern auch die Lichtparameter der Körner abzubilden.

Der Anwendungsfall in diesem Paper beschränkt sich auf Materialien, die aus granularen Materialien bestehen. Wird dieses Paper im Kontext der Asteroidenerzeugung betrachtet, so lassen sich mit diesem Algorithmus höchstens *rubble pile* Asteroiden herstellen. Da dieses Paper nur einen kleinen Teil der möglichen Asteroiden abdeckt, kann es nicht für die Erzeugung von Asteroiden im Allgemeinen genutzt werden.

2.4 Lindenmayer-Systeme zur Erzeugung von Geometrie

Die vorherig betrachteten Arbeiten, die keine Lindenmayer-Systeme nutzen, eignen sich nicht zur vollständigen Generierung von Asteroiden. Diese Verfahren beschränken sich auf die Generierung von Gesteinsbrocken als Teil von Geröll und granularen Materialien. Wie schon erwähnt, würden sich diese Algorithmen nur zur Erzeugung eines speziellen *rubble pile* Asteroiden eignen. Da in dieser Masterarbeit jedoch generische Asteroiden erstellt werden sollen, können diese Verfahren nicht genutzt werden.

Durch Anpassungen könnten einige der vorgestellten Verfahren erweitert werden, um monolithische Asteroiden zu generieren. Dazu gehört beispielsweise das Paper „Procedural Generation of Rock Piles using Aperiodic Tiling“ von Peytavie et al., das die Grundform von Asteroiden mittels Voronoï-Zellen erzeugen könnte. Der Autor sieht damit einhergehend jedoch den Verlust von Flexibilität gegenüber generischen Verfahren, wie sie L-Systeme bieten können. Das ausschließliche Nutzen von Voronoï-Zellen würde beispielsweise das Bilden von Kratern verhindern.

Das als SpeedRock bezeichnete Verfahren würde sich im Grunde zur Erzeugung eines Asteroiden eignen. Jedoch ist das Verfahren, das zur Erstellung der Asteroiden genutzt wird, durch die zufällige Auswahl der Produktionsregeln ebenfalls sehr unflexibel und wenig steuerbar. Durch die Art, wie der Gesteinsbrocken mittels einer Kugel und den Würfeln erzeugt wird, ist es auch nicht möglich konkave Modelle oder Krater zu erzeugen.

Das Paper „L-Systems in geometric modeling“ zeigt, welches Potential in Lindenmayer-Systemen zur Erzeugung von Geometrie vorhanden ist. Deshalb werden Lindenmayer-Systeme in dieser Masterarbeit genauer betrachtet. Es wurde gezeigt, dass durch kleine Erweiterungen L-Systeme zur Erzeugung komplexer Modelle genutzt werden können. Mithilfe von Annotationen in den Produktionsregeln können während der Entwicklung eines L-Systems Befehle ausgeführt werden, die Polygone generieren oder manipulieren. Somit lassen sich mit L-Systemen auch viele prozedurale Algorithmen abbilden, besonders jene, die eine hohe Selbstähnlichkeit haben.

Aus diesen Gründen wird das Potential von L-Systemen in den nachfolgenden Kapiteln bezüglich der Asteroidengenerierung untersucht.

Kapitel 3

Anforderungsanalyse

Das vorangegangene Kapitel hat diverse wissenschaftliche Arbeiten, die sich der Geometrieerzeugung gewidmet haben, vorgestellt und analysiert. Dabei wurden Lindenmayer-Systeme als geeignete Grundlage zur Erstellung der in dieser Arbeit notwendigen Asteroiden ausgewählt. Unter diesem Aspekt wird im ersten Abschnitt eine API entwickelt, die Lindenmayer-Systeme abbildet und gleichzeitig effizient ausführen kann. Durch Analyse realer Asteroiden werden zudem Merkmale herausgearbeitet, die diese API erzeugen können muss. Dazu wird zu jedem Merkmal ein Algorithmus entwickelt, der potentiell das Merkmal replizieren kann.

Im zweiten Abschnitt wird auf die Spezifikation der Sprache PCGL eingegangen. Damit einhergehend wird ein Transpiler spezifiziert, der die Ausgangssprache PCGL in eine Zielsprache umgewandelt. Der Transpiler[1] ist eine Sonderform des Compilers. Er übersetzt aus einer Quellsprache in eine andere Sprache. Diese andere Sprache ist im Rahmen dieser Masterarbeit C++.

3.1 API zur Erstellung prozeduraler Inhalte

Das Ziel dieser Masterarbeit besteht darin, Asteroiden durch Aggregation mehrerer prozeduraler Algorithmen darstellen zu können. Dazu werden Lindenmayer-Systeme genutzt. Mithilfe eines Lindenmayer-Systems können einzelne Merkmale eines Asteroiden durch Regeln ausgedrückt und mittels weiteren Regeln zusammengefasst werden.

In diesem Abschnitt werden nun Asteroiden hinsichtlich ihrer charakteristischen Merkmale untersucht.

Lutetia ist ein vergleichsweise großer Asteroid im Asteroidengürtel des Sonnensystems. Sein Durchmesser beträgt etwa 100 km.

[1] auch: Transcompiler oder source-to-source-compiler

Abbildung 3.1 Asteroid Lutetia
Quelle: [Cho14]

Die Daten und Bilder, die hier als Grundlage genutzt werden, stammen aus der von ESA geführten Rosetta-Mission, bei der die Sonde am 10. Juli 2010 an Lutetia vorbei flog.

Das Paper „Images of Asteroid 21 Lutetia: A Remnant Planetesimal from the Early Solar System“ von Sierks et al. und Abbildung 3.1 dienen hier als Basis für die Ermittlung der Oberflächenmerkmale [Sie+11].

Grundform:

Um die Grundform von Lutetia und ähnlichen Asteroiden zu erstellen, kann mit einem Ellipsoid gestartet werden. Dieser wird in mehreren Iterationen deformiert, sodass eine ähnliche Form entsteht.

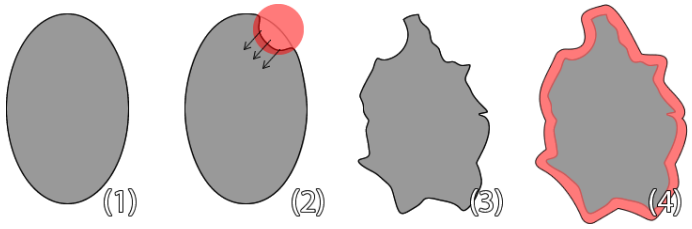

Abbildung 3.2 Grundform eines Asteroiden aus einer Ellipse

In Abbildung 3.2 wurde ein solches Verfahren grafisch für den zweidimensionalen Raum dargestellt. Aus folgenden Schritten besteht das angedachte Verfahren:

1. Erstelle eine Ellipse mit zufälligen Halbachsenlängen.
2. Wähle einen zufälligen Punkt auf der Oberfläche und selektiere alle Punkte, die sich in einem bestimmten ebenfalls zufällig gewählten Radius befinden.
3. Verschiebe die Punkte in Abhängigkeit zu der Entfernung des initial ausgewählten Punktes unterschiedlich stark in entgegengesetzter Richtung der Normale.
4. Gehe beliebig oft zu Schritt 2.
5. Je nach Anzahl der Anwendungen, ist der Asteroid nun sehr zerfurcht. Um dies zu reduzieren, müssen alle Punkte selektiert und anhand ihrer Normalen translatiert werden. Je weiter sie translatiert werden, um so weicher[2] wird der Asteroid. Da Lutetia eine augenscheinlich sehr weiche Oberfläche besitzt, sollte der Wert entsprechend gewählt sein.

Die API muss also folgende Funktionen unterstützen:

- `random_surface_point` – um einen zufälligen Vertex zu ermitteln.
- `random_float` – um einen zufälligen Wert zu ermitteln.
- `ellipsoid dx, dy, dz` – um einen Ellipsoid mit den Maßen `dx` × `dy` × `dz` zu erstellen.
- `select_sphere center, radius` – um Vertices innerhalb einer Kugel mit dem Radius `radius` und dem Mittelpunkt `center` zu selektieren.
- `translate_selection vector` – um selektierte Vertices um den Vektor `vector` zu verschieben.
- `bevel_selection amount` – um selektierte Vertices um ihren Vektor um `amount` Einheiten zu verschieben.
- `select_all` – um alle Vertices zu selektieren.
- `select_none` – um alle Vertices zu deselektieren.

Krater:

In Abbildung 3.1 fallen auch die zahlreichen Krater auf. Um sie zu bilden kann folgendes Verfahren angewandt werden.

1. Wähle einen zufälligen Kratermittelpunkt $\vec{p}_{Krater}$.
2. Wähle einen zufälligen Kraterradius r_{Krater}.
3. Selektiere alle Punkte, die sich im Kraterradius r_{Krater} um $\vec{p}_{Krater}$ befinden.

[2] Der Begriff weicher meint hierbei das Fehlen von Geröll.

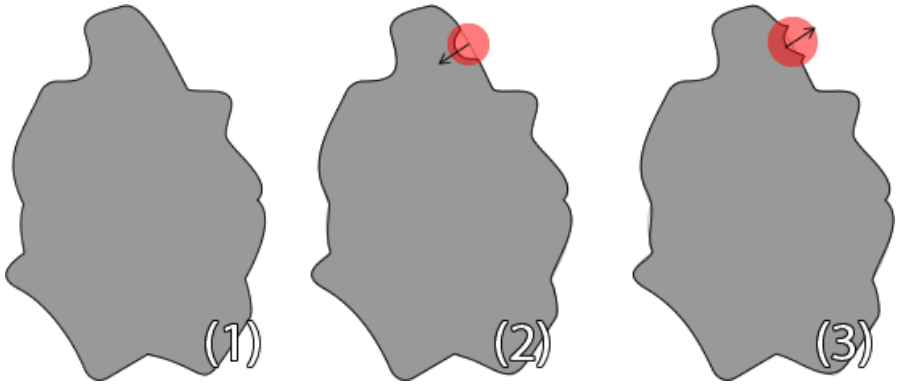

Abbildung 3.3 simulierte Kraterbildung

4. Translatiere die selektierten Punkte entgegen der Normale des Kratermittelpunkts um die Kratertiefe.
5. Wähle eine Kraterdicke w_{Krater}.
6. Selektiere alle Punkte, die sich im Radius $r_{Krater} + w_{Krater}$ um $\vec{p}_{Krater}$ befinden.
7. Translatiere die selektierten Punkte mit der Normale des Kratermittelpunkts um die Kraterlippenhöhe.
8. Wiederhole solange, bis eine bestimmte Menge Krater erstellt wurden.

Die für die Grundform benötigten API-Methoden reichen aus, um Krater bilden zu können. Es werden somit keine neuen API-Methoden als Anforderung formuliert.

Biome:

Abbildung 3.4 zeigt, dass die Oberfläche von Lutetia verschieden geartet ist. Die Oberfläche, die den Bildausschnitt dominiert (Bereich C), unterscheidet sich von den Oberflächen in Bereich A und B. In Bereich A wird deutlich, dass dort viel Geröll vorhanden ist und kleinere Schatten auf die Oberfläche geworfen werden. Bereich B hingegen ist komplett frei von Geröll und weist keinerlei Schattenbildung auf. Die Grenzen dieser Bereiche sind teils sehr scharf, können allerdings auch weich in andere Bereiche übergehen.

Um solche Bereiche zu erzeugen, eignen sich sogenannte Biome[3].

In Abbildung 3.5 wird gezeigt, wie Biome mittels einer Zufallsverteilung, die auf eine Funktion der Form $f : \mathbb{R}^n \to \mathbb{R}$ basiert, erstellt werden können.

1. Die Zufallsverteilung wird mithilfe einer Funktion definiert.
2. Biome werden an bestimmten Funktionswerten festgelegt.
3. Im zweidimensionalen Raum entsteht dann eine Verteilung der Biome, wie sie in Abbildung 3.5 abgebildet ist.
4. Nun wird ein Schwellenwert angewandt, der die Biome klar voneinander abgrenzt.
5. Die dadurch entstehende Verteilung kann auf das Modell angewandt werden.
6. Der Asteroid hat nun eine Verteilung von Biomen, auf die – je nach Art – andere Eigenschaften zugewiesen werden können.

[3] Der Begriff „Biom“ wird hier genutzt, um subjektiv unterschiedliche Landschaftstypen zu definieren.

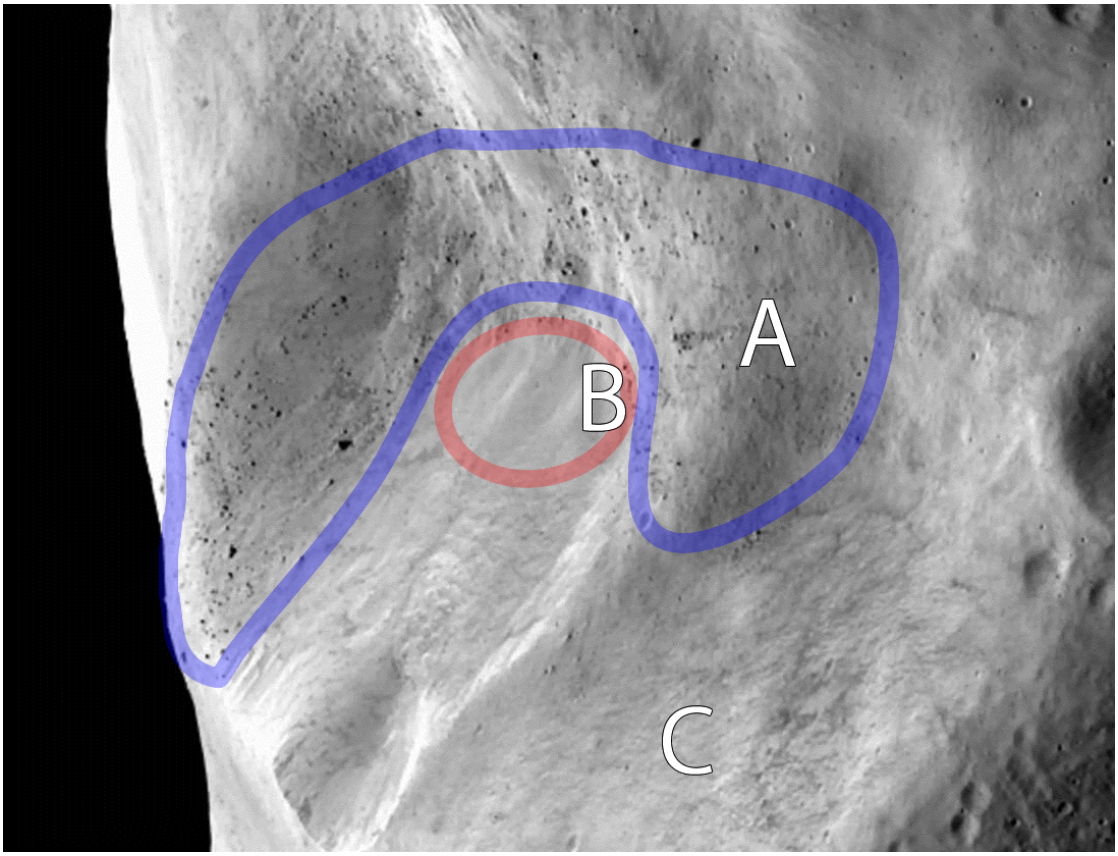

Abbildung 3.4 Nahaufnahme von Lutetia
Quelle: [al11]

Um dieses Verfahren mit der API abzubilden, sind folgende Funktionalitäten notwendig:

- `noise1 x`, `noise2 x, y`, `noise3 x, y, z` – um einen Wert mit einer Zufallsverteilung[4] zu ermitteln.

Bei der Nahaufnahme von Lutetia in Abbildung 3.4 ist besonders eine unterschiedliche Geröllverteilung zu erkennen.

Geröll:

Geröll kann nach folgendem Verfahren erstellt werden. Dabei kann zwischen großen Steinen und kleineren Geröllansammlungen unterschieden werden. Die zwei Verfahren sind in Abbildung 3.6 jeweils oben und unten abgebildet. Es wird zuerst das obere Verfahren erläutert.

1. Wähle eine zufällige Geröllposition $\vec{p}_{Geroell}$.
2. Wähle einen zufälligen Geröllradius $r_{Geroell}$.
3. Selektiere alle Punkte, die sich im Geröllradius $r_{Geroell}$ um $\vec{p}_{Geroell}$ befinden.

[4] z.B. Perlin-Noise

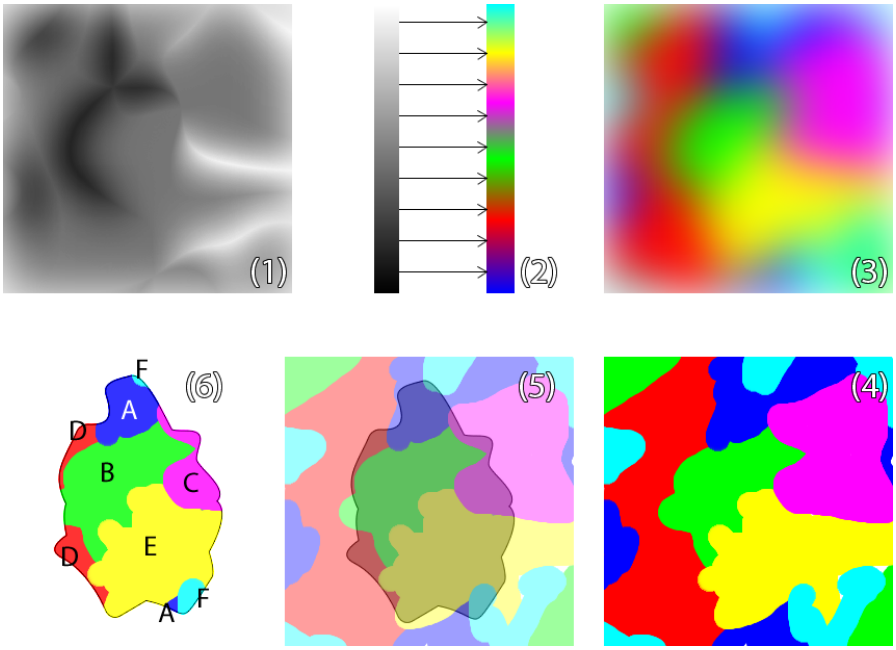

Abbildung 3.5 Biome mithilfe einer Zufallsverteilung bilden

4. Translatiere die selektierten Punkte mit der Normale der Geröllposition um den Geröllradius.
5. Wähle alle Punkte aus, deren Normalen sich maximal um 90° von der Normal von $\vec{p}_{Geroell}$ unterscheiden.
6. Translatiere diese Punkte entlang ihrer Normalen um eine bestimmte Menge.
7. Wiederhole solange bis eine bestimmte Menge von großem Geröll erstellt wurde.

Das untere Verfahren kann wie folgt erzeugt werden:

1. Wähle eine Menge von Oberflächenpositionen.
2. Generiere eine Menge an Steinen zur Verteilung an diesen Positionen.
3. Platziere diese Steine an den vorher ausgewählten Positionen und skaliere und rotiere sie.

Beide Verfahren benötigen neue Funktionalitäten in der API:

- `filter_selection_by_normal_angle base_normal, angle` – Deselektiert Vertices, deren Normalen nicht dem angegeben Kriterium entsprechen.
- `translate_selection_along_normals amount` – Translatiert Vertices entlang ihrer Normalen um `amount`.
- `push_model_state` – Speichert die Geometriedaten auf einen Stack und erzeugt eine Kopie, auf der gearbeitet werden kann. Diese und die nachfolgenden Funktionen sind notwendig, um während der Erzeugung des Modells neue Untermodelle zu erzeugen.
- `push_clean_model_state` – Erzeugt einen leeren Geometrieraum auf einem Stack, auf dem gearbeitet werden kann.
- `pop_model_state` – Löscht den aktuellen Geometrieraum und lädt den vorherigen.

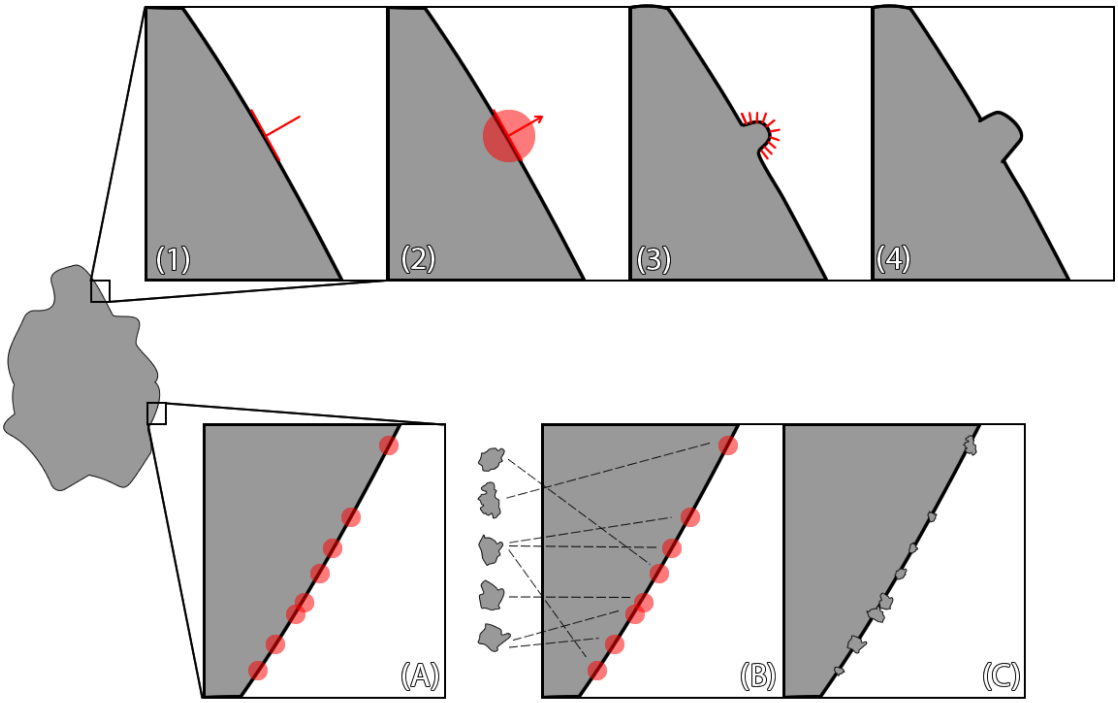

Abbildung 3.6 Geröllbildung

- `pop_and_merge_model_state` – Fügt den aktuellen Geometrieraum in den darunterliegenden ein und löscht den aktuellen Geometrieraum dann.

Vesta:

Vesta ist ein Asteroid des Asteroidengürtels, der als Planetesimal[5] eingeordnet werden kann. Er hat einen Durchmesser von etwa 500 km und ist damit nahezu 5 mal größer als Lutetia. Die ermittelten Daten und Bilder stammen von der Dawn-Mission, die während des Fluges nach Ceres, einem Zwergplaneten, an Vesta vorbeiflog und u.a. dessen Oberfläche untersuchte [Mar+12].

Vesta und Lutetia unterscheiden sich optisch kaum. Die Grundform von Vesta ähnelt eher einer Kugel. Dies liegt daran, dass Vesta ein Planetesimal ist, und durch die Gravitation in eine Kugelform gezwungen wurde [Zub+11]. Bei der Erstellung eines prozeduralen Asteroiden sollte dieser Zusammenhang berücksichtigt werden.

Zwei Merkmale können jedoch abweichend von Lutetia zusätzlich eingeführt werden.

Ungleichmäßige Kraterlippen:

Im Gegensatz zu Lutetia hat Vesta ungleichmäßige Kraterlippen. Besonders bei Zunahme des Kraterradius

[5] Als Planetesimal werden Asteroiden bezeichnet, die als Vorstufe eines Planeten angesehen werden.

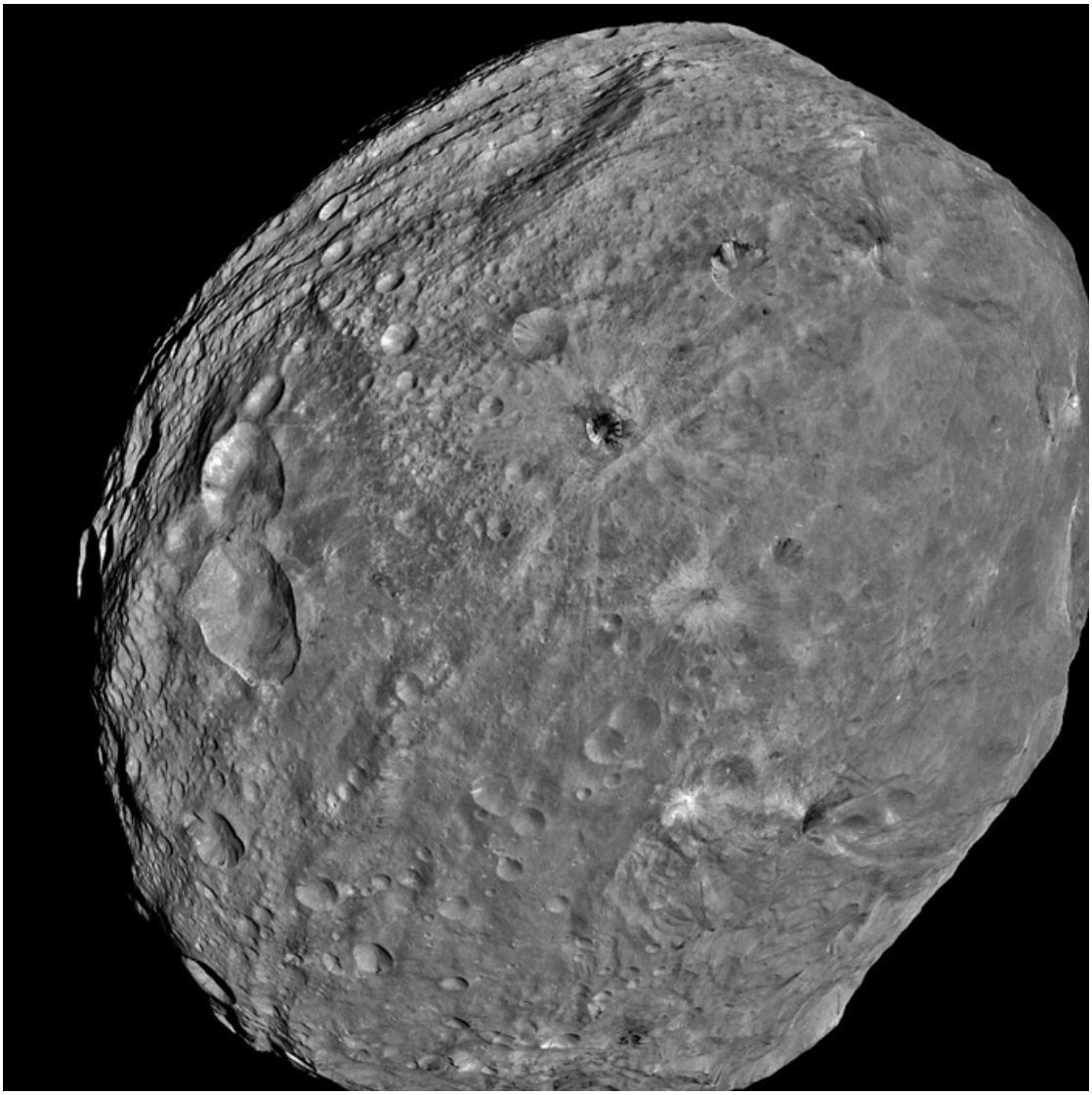

Abbildung 3.7 Asteroid Vesta
Quelle: [Gre11]

ist dieser Umstand erkennbar. Damit dies durch eine API abbildbar ist, muss die Funktion `select_sphere`, die eine Kugelselektion durchführt um einen Zufallsfaktor erweitert werden. Somit müssen die Anforderungen wie folgt formuliert werden:

- `select_sphere center, radius, turbulence` – um Vertices innerhalb einer Kugel mit dem Radius `radius` und dem Mittelpunkt `center` zu selektieren. Dabei wird der Radius mittels des Zufallsfaktors `turbulence` und einer Noise-Funktion variiert.

Raue Oberfläche:

Besonders in der oberen linken Bildecke in Abbildung 3.7 erkennt man die raue Oberfläche von Vesta. Diese lässt sich simulieren, indem die Vertices mit einem Noise-Feld moduliert werden.

Dazu muss folgende Funktion in die API integriert werden.

- `apply_noise_field_to_selection function, strength` – Moduliert die selektierten Vertices mit der angegeben Funktion `function` mit der Stärke `strength`.

Itokawa:

Der Asteroid Itokawa wurde ab dem 12. September 2005 bis in den Dezember 2005 von der Sonde Hayabusa beobachtet [Fuj+06][Age05]. Dabei wurde das Bild in Abbildung 3.8 erstellt. Itokawa ist mit den Maßen 535 m × 294 m × 209 m einer der kleinsten Asteroiden in dieser Auswahl.

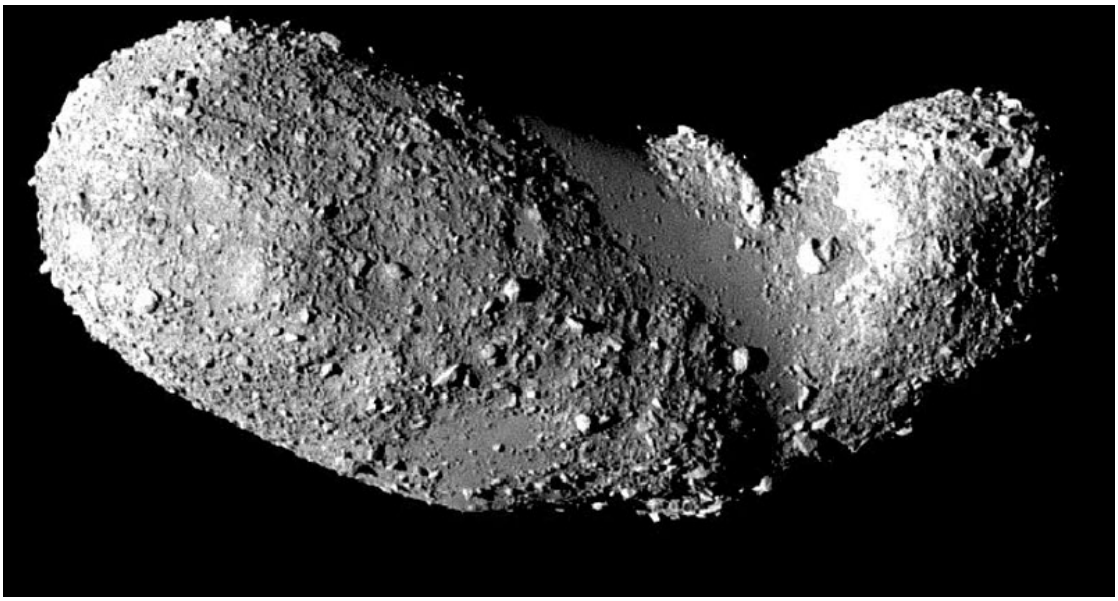

Abbildung 3.8 Asteroid Itokawa
Quelle: [Age05]

Er unterscheidet sich in der Grundform sehr stark von den beiden vorherigen. Dennoch lassen sich alle vorherigen Merkmale auch bei ihm finden.

Itokawa besitzt trotz seiner geringen Größe Biome. Mittig in Abbildung 3.8 ist eine glattere Oberfläche zu sehen, wohingegen der restliche Asteroid eine eher geröllreiche Oberfläche aufweist.

67P/Churyumov–Gerasimenko:

Der Komet Churyumov-Gerasimenko wurde ab dem 23. März 2014 während der Rosetta-Mission untersucht. Dabei wurde das in Abbildung 3.9 gezeigte Bild erstellt. Er besitzt eine Hantelform, dessen zwei größeren kugelförmigen Enden in der Mitte verbunden sind. Die beiden Enden haben jeweils eine Größe von etwa 2,6 km × 2,3 km × 1,8 km und 4,1 km × 3,3 km × 1,8 km [Age15].

Abbildung 3.9 Komet Churyumov–Gerasimenko
Quelle: [Age14]

Um seine zusammengesetzte Form in der API zu replizieren, wird ein Verfahren benötigt, das die charakteristische Hantelform erzeugen kann. Dafür bietet sich folgender Algorithmus an:

1. Erstelle einen Ellipsoid mit den Maßen a, b und c, wobei $a \gg b + c$.
2. Selektiere alle Vertices mithilfe eines Gradienten einer zufälligen Breite w und an einem zufälligen Punkt p, der optimalerweise nahe des Zentrums liegen sollte.
3. Skaliere alle selektierten Punkte um den Faktor s, wobei $s < 1$ ist.

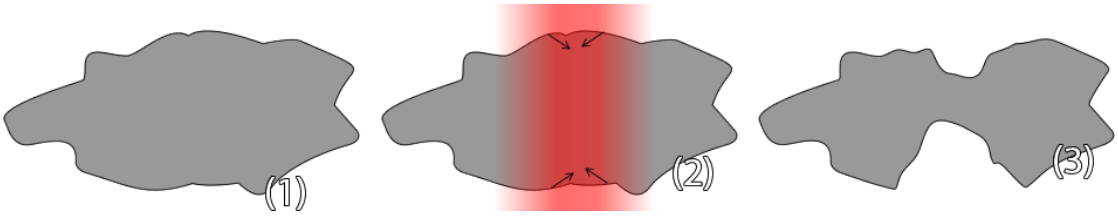

Abbildung 3.10 Bildung der Churyumov-Gerasimenko-Hantelform

Die API muss somit auf folgende Funktionalitäten hin erweitert werden:

- `select_{x,y,z}_gradient center, width` – Selektiere alle Vertices, die innerhalb des Gradienten mit dem Zentrum `center` und der Breite `width` liegen.
- `scale_selection sx, sy, sz` – Skaliere alle selektierten Vertices um die Faktoren `sx`, `sy` und `sz`.

3.2 PCGL

Die Procedural Content Generation Language (kurz: PCGL) ist eine Sprache, die es ermöglicht, geometrische Modelle im dreidimensionalen Raum algorithmisch zu beschreiben. Ihr Einsatzgebiet ist also domänenspezifisch und somit ist PCGL als domänenspezifische Sprache einzuordnen. Die Motivation der Konzipierung dieser Sprache zur Erstellung von Geometriedaten ist die Vereinfachung der Nutzung prozeduraler Algorithmen. Zudem werden prozedurale Inhalte von der eigentlichen Implementierung eines Programmes getrennt.

PCGL kann mithilfe eines Transpilers in die Kompilierung einer Applikation integriert werden. PCGL-Modelldateien können damit als Teil der sogenannten Asset-Pipeline von Spielen oder Simulationen betrachtet werden. Die unterschiedliche Behandlung von diskreten Modellen, die in einem 3D-Programm erstellt wurden (z.B. 3ds Max[6]), und prozeduralen, zur Laufzeit erstellten Modellen, fällt damit weg.

PCGL abstrahiert die Aktionen, die ein Modell-Designer in 3D-Programmen durchführt, und stellt diese als Sprachfunktionalitäten bereit. So sollen beispielsweise Selektionen und einzelne Dreiecksmanipulationen auf einem Modell algorithmisch beschreibbar und variierbar sein. Dadurch, dass PCGL für eine spezifische Domäne entworfen wird, können komplexe Befehle sehr einfach gehalten werden.

Die Funktionalität wird dabei nicht durch die Sprache selber, sondern durch eine darunterliegende API bereitgestellt.

3.2.1 Objektdefinitionen

In PCGL wird pro definiertes Objekt eine Datei angelegt. In dieser liegen die Anweisungen zu dem jeweiligen Objekt vor. Dabei gilt die erste dort vorgefundene Regel als sogenannte Startregel. Regeln sind in sich abgeschlossen und wechselwirken nur über die Geometriedaten mit anderen Regeln.

[6] 3D-Modellierungssoftware der Firma Autodesk Inc. Siehe *http://www.autodesk.de/products/3ds-max/overview*

3.2.2 Syntax

Die Syntax ist bewusst *einfach* gehalten. *Einfach* bedeutet in diesem Fall, dass folgende Merkmale erfüllt werden [Sch97]:

- **Einheitlichkeit:**
  Es existieren nur wenige und einfach zu verstehende Syntaxregeln, die für die Programmierung in PCGL nötig sind. Abweichende Syntaxregeln dürfen existieren, sollten in ihrer Logik aber kohärent sein.
- **Verallgemeinerung:**
  Zwei verschiedene Funktionen, die eine gemeinsame Teilfunktionalität bieten, sollten bei größerer Übereinstimmung zu einer Funktion verallgemeinert werden, sofern die dadurch wegfallenden Funktionen anderweitig mit der verallgemeinerten Funktion erreicht werden können.
- **Bekanntheit:**
  Die Syntax sollte von anderen Programmiersprachen inspiriert sein, sofern sie sich auch ähnlich verhält. Durch die Bekanntheit kann eine höhere Erlernbarkeit der Sprache erreicht werden.
- **Hohe Ausdrucksstärke:**
  Da die zu entwickelnde Programmiersprache eine Hochsprache[7] ist, ist es wichtig, dass die Informationsdichte maximiert wird. Dies ist wichtig, um den Lernaufwand einer neuen Sprache gegenüber der Benutzung einer nicht auf den Kontext spezialisierten Sprache aufzuwiegen.

Die Syntax von PCGL wird in EBNF[8] spezifiziert. Teile der Syntax sind aus der Programmiersprache „C“ entnommen, um den zu entwickelnden Transpiler einfach zu halten.

PCGL stellt eine Programmiersprache dar, dessen Struktur durch Einrückung definiert wird. Dies hat zwei Vorteile. Zum einen ist die Implementierung einer hierarchischen Struktur mittels Einrückung in einen Transpiler einfacher als mittels Klammern. Zum anderen entsteht leichter zu lesender Code[9].

Funktionsaufrufe:

Laut Listing 3.11 folgen Anweisungen folgendem Schema:

```
Kommandoname Parameter_1, Parameter_2, ... Parameter_n
```

Es existieren also keinerlei Klammern beim eigentlichen Funktionsaufruf. Innerhalb der Parameter können jedoch Klammern genutzt werden. Neben den integrierten Funktionsaufrufen können auch eigene Regeln aufgerufen werden.

Variablen:

Variablen können mittels der `set`-Funktion gesetzt werden:

```
1  scene
2    set my_var, random_float
```

[7] Als Hochsprache wird eine Programmiersprache bezeichnet, die nicht direkt vom Computer verstanden werden kann. Zu einer Hochsprache ist ein Compiler oder eine Menge von Compilern nötig, die die Hochsprache in Maschinensprache transformiert.
[8] Kurzform für Erweiterte Backus-Naur-Form. Hilfsmittel um systematisch Syntaxregeln zu definieren.
[9] vgl. hierzu die Entscheidung von Python: *http://effbot.org/pyfaq/why-does-python-use-indentation-for-grouping-of-statements.htm*

```
AlphabeticCharacter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                    | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                    | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                    | "V" | "W" | "X" | "Y" | "Z" ;  (* sowie alle kleinen Buchstaben *)
AlphaCharacter = AlphabeticCharacter | "_";
NonZeroDigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
Digit = "0" | NonZeroDigit;
AlphaNumericCharacter = AlphaCharacter | Digit;
Newline = ["\r"], "\n";
IntConst = NonZeroDigit, {Digit};

FloatConst = NonZeroDigit, {Digit}, ".", Digit, {Digit};
Id = AlphaCharacter, {AlphaNumericCharacter};
RuleName = Id;
RuleCondition = " : " Condition;
RuleDefinition = RuleName, [RuleCondition], Newline, RuleBody;
RuleBody = Command {Newline, Command};

CommandName = Id;

Command = CommandName, [ParameterList];
ParameterList = Parameter [",", ParameterList];

Parameter = AdditiveExp;

Condition = LogicalOrExp;
LogicalOrExp = LogicalAndExp
               | LogicalOrExp, "||", LogicalAndExp
               ;
LogicalAndExp = EqualityExp | LogicalAndExp, '&&', EqualityExp;
EqualityExp = RelationalExp | EqualityExp, '==', RelationalExp | EqualityExp, '!=', RelationalExp;
RelationalExp = AdditiveExp
               | RelationalExp, '<', AdditiveExp
               | RelationalExp, '>', AdditiveExp
               | RelationalExp, '<=', AdditiveExp
               | RelationalExp, '>=', AdditiveExp
               ;
AdditiveExp = MultExp
      | AdditiveExp '+' MultExp
      | AdditiveExp '-' MultExp
      ;
MultExp = UnaryExp
      | MultExp, '*', UnaryExp
      | MultExp, '/', UnaryExp
      ;
UnaryExp = PrimaryExp
      | unary operator, UnaryExp
      ;
UnaryOperator = '+' | '-' | '!'
      ;
PrimaryExp = Id | Const | '(', Condition, ')'
      ;
Const = IntConst
      | FloatConst
      ;
```

Abbildung 3.11 Syntaxdefinition von PCGL in EBNF
Quelle: Angelehnt an [Jin04]

Variablen sind typisiert, deren Typ jedoch implizit hergeleitet. So ist `my_var` vom Typ *float*.

Regeln:

Regeln werden in der Zero-Level-Indentation Ebene definiert und dienen zur Beschreibung von Geometrie.

Sie besitzen eine Bedingung, die erfüllt sein muss, damit diese Regel auch ausgeführt wird.

```
1  simple_rule : random_float > 50%
2    sphere random_float
```

Es können somit mehrere Regeln zu einem Regelnamen erstellt werden. Die Bedingungen der Regeln wird in der Reihenfolge evaluiert, in der sie in der Objektdefinition erstellt wurden. Es existiert zudem die Möglichkeit, eine spezielle Regel anzulegen, die bedingungslos ist. Diese Regel wird bei mehreren Bedingungen hinten angestellt und ausgeführt, wenn keine der vorhergehenden Bedingungen erfüllt sind. Damit sind parametrisierte Lindenmayer-Systeme mit PCGL abbildbar.

Um auch stochastische Lindenmayer-Systeme abzubilden, kann als Bedingung eine ganze Zahl angegeben werden. Sie dient bei der Wahl der auszuführenden Regel zur Gewichtung.

```
1  scene
2    the_rule
3
4  the_rule : 1
5    sphere 1
6
7  the_rule : 1
8    sphere 2
9
10 the_rule : 1
11   sphere 3
```

Die ganzen Zahlen werden aufsummiert und als Wahrscheinlichkeit genommen, zu der eine Regel eintritt. Im Beispiel tritt zu einer Wahrscheinlichkeit von $\frac{1}{3}$ eine der genannten Regeln ein.

3.2.3 Transpiler

Im Zuge dieser Masterarbeit wird auch ein Transpiler entwickelt, der PCGL in C++ transpiliert.

Dabei wird in der Entwicklung auf folgende Eigenschaften geachtet.

Fehlertoleranz:

Fehler, die der Benutzer bei der Bedienung des Transpilers macht, werden erkannt und mit einer geeigneten Fehlerausgabe versehen.

Der Transpiler transformiert fehlerhaften Quellcode nicht in eine fehlerhafte C++-Ausgabe. Fehlerhafte Stellen im Quellcode sollen mit ihrer Position ebenfalls ausgegeben werden.

Lesbarer Quellcode:

Der Quellcode, der bei der Nutzung des Transpilers entsteht, soll für den Nutzer verständlich sein. Dadurch ist gewährleistet, dass die zugrunde liegende API besser verstanden werden kann und Debuggingmaßnahmen aufgrund semantische Fehler einfacher zu bewerkstelligen sind.

Ebenso ist es wichtig, dass der Quellcode des Transpilers selbst simpel gehalten wird. Dies wird durch die Nutzung bewährter Entwurfsmuster erreicht.

Austauschbarkeit des Backends:

Damit PCGL auch mit anderen Programmiersprachen nutzbar ist, wird der Transpiler so implementiert, dass das Backend leicht austauschbar ist. Dafür wird das sogenannte Visitor-Entwurfsmuster genutzt, das aus der internen Darstellung des Transpilers Quellcode erzeugt. Um nun ein neues Backend zu integrieren, muss lediglich die Implementierung des Visitors ausgetauscht werden.

Kapitel 4

Umsetzung

Dieses Kapitel beschäftigt sich mit der Umsetzung der Komponenten der Masterarbeit. Dazu gehört neben dem Transpiler, der die Programmiersprache PCGL in C++-Code transformiert, auch die API zur Generierung prozeduraler Inhalte.

4.1 Transpiler

Der Transpiler dient der Transformation der vom Autor entwickelten Sprache PCGL zu C++. Der Transpiler ist eine Komponente, die optional in die Toolchain eines Entwicklers integriert werden kann, um die Modellbeschreibungen im Format `*.pcgl` von der eigentlichen Implementierung strikt zu trennen. Die Nutzung des Transpilers erzeugt C++-Code, der nur mit der in dieser Masterarbeit entwickelten API funktioniert. Die API kann jedoch ohne die Verwendung des Transpilers genutzt werden.

4.1.1 Implementierung

Zur Entwicklung des Transpilers wurde die Programmiersprache Ruby gewählt. Die Entscheidung dazu fiel aufgrund diverser Punkte:

- Ruby besitzt in den Standardbibliotheken bereits mächtige Werkzeuge, um Zeichenketten zu manipulieren und zu verarbeiten.
- Reguläre Ausdrücke sind direkt in Ruby integriert und erleichtern so die Lesbarkeit des Codes, der davon öfters Gebrauch macht.
- Der Autor des Transpilers hat bereits mehrere Jahre Erfahrung in Ruby und möchte die Implementierung des Transpilers als Referenzimplementierung angeben.

- Da Ruby eine interpretierte Sprache ist, können im Gegensatz zu C++ Erweiterungen einfacher eingefügt werden.

Beim Aufruf des Transpilers werden alle Dateien im Ordner mit der Endung `*.pcgl` ermittelt und nach folgendem Verfahren in C++-Dateien transformiert:

1. Die Datei wird geöffnet und in einzelne Zeilen unterteilt. Jede Zeile entspricht einem Statement. Dabei wird das Visitor-Entwurfsmuster benutzt, um einen Abstrakten Syntax Baum (kurz: AST) mithilfe der Einrückungstiefe aufzubauen.
2. Jedes Statement wird dann nach folgenden Kriterien zu einem Objekt verarbeitet:
   - Ist die Einrückungstiefe 0 und entspricht dem regulären Ausdruck einer einfachen Regel, einer Regel mit Doppelpunkt (also einer Regel mit Bedingung) oder einer Regel mit vorangestelltem `implicit` Schlüsselwort, so wird ein entsprechendes Rule-Objekt erstellt und im AST abgelegt.
   - Ist die Einrückungstiefe nicht 0 und entspricht dem regulären Ausdruck eines Kommandos, so wird ein Command-Objekt erstellt. Command-Objekte werden in einem später folgenden Schritt gegebenenfalls zu einem Regelaufruf umgewandelt, da diese in PCGL nicht unterschieden werden. Das spezielle Command-Objekt `set` wird später durch einen weiteren Schritt in ein Assignment-Objekt umgewandelt, da es eine spezielle Verarbeitung und Ausgabe benötigt.
3. Mehrfach vorkommende Rule Objekte mit Bedingungen werden dann in ein Rule-Objekt zusammengefasst und eine Bedingungsstruktur (If-, ElseIf- und Else-Objekte) aufgebaut. Diese wird zur späteren Ausgabe in Code-Form benötigt.
4. Das Command-Objekt mit dem Bezeichner `set` wird in ein Assignment-Objekt umgewandelt.
5. Als letztes wird auf dem AST ein Ausgabe-Visitor angewandt, der den finalen C++-Code ausgibt.

In Kapitel Anforderungen wurden einige Beispiele gegeben, die mit PCGL möglich sind. Am Beispiel von 4.1 wird nun gezeigt, wie der Transpiler die Anweisungen in C++-Code umwandelt.

Listing 4.1 Ein einfaches Skript in PCGL

```
1  scene
2    ellipsoid 5,7,5
3    apply_noise_field lines
4    apply_noise_field lines
5    apply_noise_field lines
6
7  implicit lines
8    return noise1(y * 10.0) * 10%
```

Das Aufrufen von Regeln wird mit der C++11-Funktion `bind` umgesetzt. Dies ist notwendig, um beliebige Methoden der Unterklassen von BaseModel an Methoden zu übergeben, die in BaseModel direkt definiert werden und noch nicht über die Existenz der Unterklasse informiert sind. BaseModel ist die Klasse, die die prozeduralen Algorithmen der API zur Verfügung stellt.

Die automatische Typendeduzierung mittels `auto` wird benutzt, um die explizite Angabe von Variablentypen zu entfernen. Diese Möglichkeit ist ebenfalls nur mit einem Compiler gegeben, der den C++11-Standard implementiert.

Listing 4.2 Transpiliertes Beispiel

```
#ifndef __BEEHIVE_H
#define __BEEHIVE_H

#include "BaseModel.h"
#include <functional>

using namespace std;
using namespace std::placeholders;

class BeeHive : public BaseModel
{
public:

  void __init() {
    scene();
  }
  void scene() {
      ellipsoid(5,7,5);
      apply_noise_field(bind(&BeeHive::lines, this, _1, _2, _3));
      apply_noise_field(bind(&BeeHive::lines, this, _1, _2, _3));
      apply_noise_field(bind(&BeeHive::lines, this, _1, _2, _3));
  }

  float lines(float x, float y, float z) {
      return(noise1(y * 10.0) * 0.1);
  }

};

#endif
```

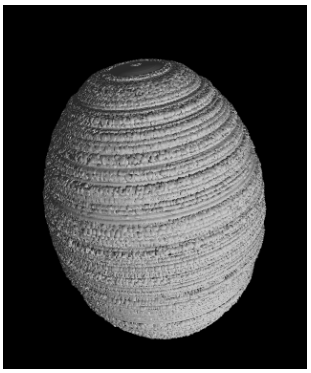

Abbildung 4.1 Grafische Ausgabe des Beispiels

4.1.2 Erweiterungsmöglichkeiten

Nach Implementierung der Grundversion des Transpilers sind einige Erweiterungsmöglichkeiten aufgefallen. Dieser Abschnitt zeigt diese möglichen Erweiterungen auf und erläutert, welche Vorteile damit verbunden wären.

Unter anderem das Sandbox-Spiel Minecraft hat gezeigt, dass auch mit der Programmiersprache Java 3D-Programme erstellt werden können. Daher ließe sich der Transpiler dahingehend erweitern, dass er als Aus-

gabesprache nicht C++- sondern Java- oder auch C#-Code generiert, der mit der geeigneten „BaseModel"-Komponente genutzt werden könnte. Dieser Austausch des Backends ist möglich, indem der Ausgabe-Visitor des Transpilers angepasst wird.

Die Nutzung von Ruby als Programmiersprache war zwar für die initiale Programmierung von Vorteil, fordert aber vom Benutzer des Transpilers, dass eine Rubyumgebung vorhanden ist. Dies ist auf Betriebssystem mit einem Paketverwaltungssystem (wie z.B. Ubuntu oder Mac OSX) einfach, stellt unter Windows aber eine deutlich höhere Hürde dar. Eine portable Entwicklung als C++-Anwendung wäre da vorteilhafter.

4.2 API

Dieses Kapitel beschreibt die Implementierungsentscheidungen, die während der Entwicklung der prozeduralen API vorgenommen wurden. Die API bietet die Basis-Funktionalitäten, die PCGL nutzt, als einfache Methoden an. Im Kontext dieser Masterarbeit wird sie in C++ implementiert.

Die zugrunde liegenden Anforderungen an die API wurden im Kapitel 3 definiert. Diese beinhalten eine Vielzahl von mathematischen und geometrischen Funktionalitäten, die der Autor komplett implementieren müsste. Unter bestimmten Voraussetzungen bietet sich die Nutzung einer oder mehrerer Bibliotheken an, die Teilfunktionalitäten bereitstellen. Bevor mit der eigentlichen Implementierung begonnen wird, werden erst einmal potentielle Bibliotheken kurz beschrieben, und in ihrer Tauglichkeit für die betreffenden Probleme analysiert. Der nachfolgende Abschnitt stellt diese vor und zeigt, welche Bibliotheken bei der Umsetzung genutzt wurden.

4.2.1 Evaluation von Bibliotheken

Da es eine Vielzahl an Bibliotheken gibt, die benötigte Teilfunktionen bereitstellen, hat der Autor nach einer ausgiebigen Recherche potentielle Bibliotheken ausgewählt. Die Auswahl basierte auf den gleichen Kriterien, wie sie zur weiteren Evaluation genutzt werden:

1. **geringe Komplexität:**
   Nur wenig Funktionalität wird benötigt. Unnötige Funktionen oder Klassen sollten demnach entweder nicht vorhanden sein oder deaktiviert werden können.
2. **einfache Integration:**
   Die Bibliothek sollte sich leicht in das bestehende Projekt integrieren lassen.
3. **hohe Performanz:**
   Vektoraddition und Matrixoperationen werden häufig benötigt. Die Bibliothek sollte die gewünschten Funktionalitäten effizient umsetzen, um eine Optimierung in dieser Hinsicht nicht im Wege zu stehen. Dies kann beispielsweise durch Einsatz von SSE-Registern passieren.

4. **wenige Abhängigkeiten:**
   Die Bibliothek sollte wenig bis keine weiteren Bibliotheken oder Abhängigkeiten benötigen.
5. **Komplett:**
   Die Bibliothek sollte in letzter Zeit nur noch wenige Erweiterungen erhalten haben[1].

Dabei wurden Frameworks hinsichtlich dieser Kriterien erst nur grob abgeschätzt, um eine kleinere Untermenge der Vielzahl an Bibliotheken zu erhalten. Diese kleinere Untermenge wird nun genauer nach den genannten Kriterien analysiert und evaluiert.

Zudem gibt es noch Anforderungen, die erfüllt sein sollten, da sie von der Implementierung der API benötigt werden. Sollte eine Bibliothek, die trotzdem genutzt wird, diese nicht erfüllen, so wäre eine Eigenimplementierung notwendig, sofern keine andere Bibliothek diese Funktionalität bereitstellt.

1. Mathematische Operationen auf Vektoren im dreidimensionalen Raum
2. Mathematische Operationen auf Matrizen im dreidimensionalen Raum
3. Strukturen zur internen Speicherung von Polygondaten
4. Boolesche Operationen auf Geometriedaten
5. Bounding Boxes
6. prozedurale Grundalgorithmen, wie z.B. Value noise
7. Pseudo-Zufallszahlengenerator

Es folgt nun eine Auswahl an Bibliotheken, die die gewünschten Kriterien in Teilen erfüllen.

GNU Scientific Library

Die GNU Scientific Library (kurz: GSL) ist eine Bibliothek, die Methoden für verschiedene mathematische Aufgabenbereiche bereitstellt. Sie bietet neben Vektor- und Matrixoperationen, Algorithmen der Lineare Algebra, Zufallszahlen, Interpolationsmethoden auch noch diverse andere Funktionalitäten an, die für die Implementierung der API nicht benötigt werden. Sie ist komplett in C geschrieben und kann so einfach in einer C++-Anwendung genutzt werden. Es existiert eine gut dokumentierte API, um auf die verschiedenen Methoden zuzugreifen.

GSL besitzt nach Durchsicht der Dokumentation und nach Installation einer sehr hohe Komplexität. Dies ist hauptsächlich der Allgemeingültigkeit der Bibliothek geschuldet. Sie versucht viele mathematische Aufgabenbereiche abzudecken.

Trotz hoher Komplexität ist sie auf einem Linux-System leicht zu installieren. Die Entwicklungsquelldateien für GSL sind in den Repositories der größten Betriebssysteme vorhanden und können über die Kommandozeile leicht installiert werden. Das in der Dokumentation beschriebene Testprogramm lief innerhalb kürzester Zeit.

GSL benutzt die CPU-Befehlssatzerweiterung SSE nicht explizit. Dadurch ist es nicht möglich, die zusätzliche Performanz von SSE mit GSL zu nutzen.

[1] → feature-completeness

GSL besitzt, neben einer Standard-C-Umgebung, keinerlei weitere Abhängigkeiten, die bei der Benutzung kompiliert oder installiert werden müssen.

Der Funktionsumfang von GSL wird weiterhin aktiv erweitert. Bestehende werden hingegen kaum geändert [Fou16].

CGAL

Die Computational Geometry Algorithms Library (kurz: CGAL) ist das geometrische Pendant zu GSL. Sie implementiert zahlreiche geometrische Algorithmen in C++. Sie bietet neben Triangulationsalgorithmen, boolesche Operationen auf Polygonen, Algorithmen zur Erstellung von geometrischen Strukturen, AABB[2] auch noch diverse andere Funktionalitäten an, die für die Implementierung der API nicht benötigt werden.

CGAL besitzt nach Durchsicht der Dokumentation und nach Evaluation ebenso wie GSL eine sehr hohe Komplexität, da viele geometrische Algorithmen angeboten werden.

Für CGAL gibt es für das Linux-Betriebssystem Ubuntu vorgefertigte und aktualisierbare Pakete, die einfach installiert werden können. Dadurch ist die Installation trotz Komplexität sehr einfach. Allerdings hat CGAL mehrere Abhängigkeiten, wie z.B. zur Boost C++ Library und zu einigen Qt-Paketen.

CGAL verfolgt das sogenannte Exact Computation Paradigm und rechnet mit beliebig präzisen Gleitkommazahlen. Dies geschieht, in dem die Fehler, die bei einer Berechnung aufgrund fehlender Präzision passieren können, überwacht werden, und eine Erweiterung der Präzision, der bei einem bestimmten Algorithmus zugrunde liegenden Zahl, eingeleitet wird. Die Performanz ist laut CGAL Dokumentation im besten Fall 20% schlechter als mit Gleitkommazahlen. Damit ist die Performanz als eher gering einzuschätzen [Pro16].

MathGeoLib

Die MathGeoLib von Jukka Jylänki ist eine Bibliothek, die Methoden und Klassen für Vektoren, Matrizen und Polygonen bereitstellt. Zu den Methoden gehören auch die benötigten Funktionen für Vektor- und Matrixmanipulation, sowie für die Erstellung und Bearbeitung von AABB. Die MathGeoLib liefert keine Algorithmen für boolesche Operationen auf Geometriedaten.

Die MathGeoLib ist nach Durchsicht aller Quellcode-Dateien simpel aufgebaut. Es bietet nur die grundlegenden Strukturen und Methoden zur Darstellung dreidimensionaler Geometrien.

Für die MathGeoLib gibt es keine fertigen Pakete. Die Bibliothek kann auf mehreren Plattformen (Linux mit cmake, Windows mit Visual Studio und OSX mit XCode und make) kompiliert werden oder direkt in ein Projekt inkludiert werden [Jyl16].

Eigenentwicklung

Neben der Benutzung einer bestehenden Bibliothek ist auch die Entwicklung einer eigenen Bibliothek möglich. Dies hat zwar den Vorteil, dass nur die nötigsten Funktionalitäten implementiert werden, führt aber auch zu folgenden Nachteilen:

[2] Die Axis-Aligned Bounding Box (kurz: AABB) ist eine geometrische Struktur, die einen Würfel mittels zwei Punkten definiert. Die Kanten dieses Würfels verlaufen parallel zu den Achsen des Koordinatensystems.

1. Mathematische Methoden müssen selbst auf Performanz optimiert werden.
2. Die Entwicklungszeit erhöht sich voraussichtlich.
3. Eine fertige Bibliothek ist von mehreren beteiligten Entwicklern programmiert und getestet worden. Häufig wurde mehr Zeit in diese Bibliothek investiert, als dies in der Bearbeitungszeit dieser Masterarbeit möglich wäre. Dieses Zeitdefizit könnte zu einer höheren Fehleranfälligkeit und somit zu einer niedrigeren Qualität der Implementierung führen.

Fazit

Nach Abwägung aller Kriterien wurde die MathGeoLib ausgewählt. Diese enthält alle nötigen mathematischen Funktionen (Vektor- und Matrixberechnungen, einfache Geometriedatenstrukturen und Bounding Boxes). Die großen Bibliotheken CGAL und GSL hätten diese Funktionen auch geliefert, aber zu einer erhöhten Komplexität und im Falle von CGAL zu einer schlechteren Performanz geführt.

Für die CSG-Funktionen wird auf Basis einer einfachen Referenzimplementierung in JavaScript (csg.js) eine Eigenimplementierung vorgenommen. csg.js benutzt BSP-Bäume[3] um boolesche Operationen auf Dreiecksdaten durchzuführen [4].

Auch prozedurale Algorithmen, wie z.B. Value noise, fließen als Eigenimplementierung in die API mit ein. Dazu wird jedoch ein Zufallszahlengenerator genutzt, der von der MathGeoLib geliefert wird.

4.2.2 Implementierung

Nach Auswahl der Bibliotheken, die die Teilfunktionalitäten bereitstellen, wird nun die eigentliche Komponente entwickelt, die die Basisfunktionalität zur Erstellung prozeduraler Inhalte liefert. Dabei wird die Komponente so entwickelt, dass sie die Basis aller Modelldefinitionen darstellt.

Die API besteht aus diversen C++-Klassen, die nun im einzelnen erläutert werden.

ModelData und ModelDataStack:

Für die Implementierung der Regeln wird ein Stack von Geometriedaten vorgehalten. In der Anforderungsanalyse wurde ermittelt, dass dieser notwendig ist, um die CSG-Operationen *union*(Vereinigung), *difference*(Differenz) und *intersection*(Schnitt) zu implementieren. Beim Aufruf des Befehls:

```
1    // Da union ein reservierter Begriff ist, wird hier auf combine zurückgegriffen
2    combine(bind(&MyModel::cup, this),bind(&MyModel::cap, this));
3    // ...
```

wird zuerst das linke Modell erstellt (`cup`), abgespeichert und dann das zweite Modell (`cap`) erstellt und abgespeichert. Bei Anwendung der CSG-Operation werden die beiden generierten Untermodelle vom Stack genommen, die Operation angewandt und dann das resultierende Modell in den obersten State integriert.

[3] BSP ist die Kurzform für binary space partitioning. Eine Methode zur Aufteilung von Geometrie in einen Binärbaum. Dadurch sind zahlreiche Berechnungsoptimierungen möglich.
[4] Siehe dazu die Implementierung von *https://github.com/evanw/csg.js*

Die Klassen, die dabei entstanden sind, sind `ModelDataStack` sowie `ModelData`. Erstere dient als Container-Klasse.

Bei der Implementierung der eigentlichen CSG-Funktionalität entstand die Klasse `ModelBSPNode`. Eine Instanz von `ModelData` kann eine Instanz von `ModelBSPNode` erstellen.

ModelBSPNode:

Die Klasse `ModelBSPNode` stellt einen Knotenpunkt im BSP-Baum dar und stellt die eigentliche CSG-Funktionalität in der API bereit. Nach Anwendung einer CSG-Funktion kann der BSP-Baum wieder in eine Instanz von `ModelData` und somit in einfache Polygondaten umgewandelt werden.

`ModelBSPNode` implementiert die Methoden `invert`, `clipTo` und `build`:

- `invert` invertiert alle Polygone, d.h. die mathematische Ebene wird invertiert und die Vertices werden in ihrer Reihenfolge umgedreht. Das Polygon zeigt somit in die entgegengesetzte Richtung.
- `clipTo` löscht Knoten im BSP-Baum, die in einem anderen BSP-Baum vorhanden sind. Dabei werden Polygone, die sich teils innerhalb und teils außerhalb eines BSP-Baumes befinden, aufgeteilt und nur der überlappende Teil gelöscht.
- `build` vereinigt zwei BSP-Knoten.

Mit diesen beiden Grundmethoden werden nun die CSG-Operationen implementiert. Um nun eine Vereinigungs-Operation auf ein Modell *A* und ein Modell *B* auszuführen, werden mithilfe von `clipTo` zuerst alle Polygone von *A* innerhalb von *B*, und dann alle Polygone von *B* innerhalb von *A* gelöscht. Außerdem werden Polygone, die sowohl innerhalb als auch außerhalb liegen, an der Schnittkante aufgeteilt. Der Sonderfall von koplanaren Polygonen wird mithilfe der `invert`-Funktion aufgelöst. Dazu wird das Modell *B* invertiert und noch einmal mittels `clipTo` auf Modell *A* angewandt. Das Ergebnis ist die Vereinigungs-Operation von *A* und *B*.

Die Operationen Differenz und Schnitt sind ebenfalls mit den Methoden `clipTo` und `invert` ähnlich implementiert. Siehe dazu Abbildung 4.2.

Vereinigung

```
1
2  A.clipTo(B);
3  B.clipTo(A);
4  B.invert();
5
6  B.clipTo(A);
7  B.invert();
8  A.build(B);
9  //
```

Differenz

```
1  A.invert();
2  A.clipTo(B);
3  B.clipTo(A);
4  B.invert();
5
6  B.clipTo(A);
7  B.invert();
8  A.build(B);
9  A.invert();
```

Schnitt

```
1  A.invert();
2
3  B.clipTo(A);
4  B.invert();
5  A.clipTo(B);
6  B.clipTo(A);
7
8  A.build(B);
9  A.invert();
```

Abbildung 4.2 CSG-Operationen mittels BSP abbilden

ModelState und ModelStates:

Die Klasse ModelStates ist eine Container-Klasse für Instanzen der Klasse ModelState und beinhaltet einen

Stack von Instanzen der Klasse ModelState. Diese repräsentiert einen Arbeitsraum für die zugrundeliegene Implementierung des Modells. Der Arbeitsraum ist eine Axis-Aligned Bounding Box (kurz: AABB). Der Stack an Arbeitsräumen wird für die Funktionen `divide_x`, `divide_y`, `divide_z` benötigt, die den Arbeitsraum an einer Achse teilen. Arbeitsräume sind notwendig, um die Wirkungsreichweite der angewandten Funktionen zu begrenzen.

ModelPolygon und ModelVertex:

Ein `ModelPolygon` ist eine spezielle Implementierung eines Polygons für die Verwendung in der API. Es beinhaltet eine Liste von Vertices auf Basis der Klasse ModelVertex und den dazugehörigen Selektionswert. Selektionen und Transformationen auf den Geometriedaten werden mithilfe dieser Klasse bewerkstelligt.

Um Polygone in 3D-Anwendungen nutzen zu können, wurde eine Methode geliefert, um ein `ModelPolygon` in ein oder mehrere Dreiecke umzuwandeln. Außerdem gehört zu einem `ModelPolygon` eine mathematische Ebene, die für die Klasse ModelBSPNode und den BSP-Baum benötigt wird.

Ein ModelVertex beinhaltet in der aktuellen Implementierung lediglich eine Position und einen Selektionswert. Eine Erweiterung um Textur- und Normalendaten wäre denkbar.

BaseModel:

Diese Klasse implementiert alle prozeduralen Funktionalitäten, die in den Anforderungen formuliert wurden. Zusätzlich wurden Funktionen implementiert, die dem Autor aus der Bedienung von 3D-Modellsoftwares bekannt sind und sich somit als nützlich erweisen könnten. Eine eigene Modelldefinition (z.B. von einem Asteroiden) muss von der Klasse BaseModel erben, damit die Funktionalitäten zur Verfügung stehen. Nachfolgend werden Besonderheiten bei der Implementierung der jeweiligen Funktionalitäten erläutert.

- **Value Noise:**
  Um Value Noise zu nutzen, wurden die Funktionen `noise1(x)`, `noise2(x,y)` und `noise3(x,y,z)` implementiert. Bei der Implementierung wurde weniger auf die gleichmäßige Verteilung der Zufallszahlen, als auf die Effizienz geachtet. Sollten in hoher Anzahl Noise-Funktionsaufrufe benötigt werden, so ist Value Noise zu bevorzugen. Es nutzt einen sehr einfachen Pseudozufallsgenerator, der an [Eli99] angelehnt ist. Zur Ermittlung von Werten zwischen den Ganzzahlgitterpunkten wird eine lineare Interpolation durchgeführt.
- **Perlin Noise:**
  Die Funktion `pnoise3(x,y,z)` implementiert Perlin Noise im dreidimensionalen Raum. Dabei wird eine erweiterte Implementierung der Referenzimplementierung von Ken Perlin genutzt[5].
- **Wiederholen von Regeln:**
  Mit `repeat (times, rule)` können Regeln wiederholt aufgerufen werden. Betrachtet man einen Asteroid, so kann beispielsweise das Hinzufügen von Geröllteilen mithilfe des `repeat`-Kommandos erledigt werden.
- **Erzeugung von Geometrie:**

[5] siehe dazu *https://github.com/sol-prog/Perlin_Noise*

`ellipsoid (rx, ry, rz, divisions)` erstellt einen Ellipsoid mit den Radien `rx`, `ry`, `rz` und der Unterteilungsiterationstiefe `divisions`. Je höher der Parameter `divisions` gewählt wird, um so mehr Polygone werden erzeugt. `sphere(radius, divisions)` ist ein Alias auf `ellipsoid(radius, radius, radius, divisions)`.

Um den Ellipsoid zu erstellen, wird ein Oktaeder als Grundform genutzt. Die Vertices des Oktaeder befinden sich auf einer Einheitskugel. Dessen Dreiecke werden `divisions` mal in vier neue Dreiecke unterteilt. Die in jedem Iterationsschritt entstehenden Vertices werden dann auf die Einheitskugel verschoben.

- **Geometrieoperationen auf Selektionen:**
  `translate_selection(vector)` verschiebt alle selektierten Vertices um den angegebenen Vektor multipliziert mit dem Selektionswert pro Vertex. Die Selektion von Vertices ist gewichtet. Das bedeutet, dass ein Vertex einen Selektionswert zwischen 0 und 1 besitzt.
  `scale_selection(vector)` skaliert die selektierten Vertices mithilfe der durch den Vektor definierten Skalierungswerte. Das Zentrum der Skalierung ist das gewichtete arithmetische Mittel aller selektierten Vertices.
  Die Funktionen `rotate_x_selection(radians)`, `rotate_y_selection(radians)` und `rotate_z_selection(radians)` rotieren die selektierten Vertices. Das Zentrum der Rotation wird wie bei der Skalierung ermittelt.
- **Selektion:**
  Mit `select_sphere(position, radius, randomness, randomness_detail)` werden Vertices innerhalb einer Kugel mit dem Radius `radius` selektiert. Die Kugel befindet sich dabei an der Position, die durch den Vektor `position` definiert wird. Die Grenze der Kugel kann mithilfe der Parameter `randomness` und `randomness_detail` zufällig verändert werden.
- **Pseudozufallsoperationen:**
  Um zufällige Positionen, Zahlen und Vektoren zu erstellen, sind Pseudozufallsoperationen in die API implementiert worden. Dazu gehört:
  - `random_vertex` – liefert einen zufälligen Vertex des Modells.
  - `random_surface_point` – erzeugt einen zufälligen Punkt auf dem Modell.
  - `random_float` – erzeugt einen zufälligen Gleitkommawert zwischen 0 und 1.
  - `random_point` – erzeugt einen zufälligen Punkt innerhalb des Modellvolumens, das beim Erstellen eines konkreten Modells definiert wird.
  - `random_vector` – erzeugt einen zufälligen Richtungsvektor[6].

Model:

Als Resultat des Erstellungsprozesses wird eine Instanz der Klasse `Model` erstellt. Diese Klasse liefert alle nötigen Funktionalitäten, um das erstellte Modell darzustellen.

[6] D.h., dass $|\vec{v}| = 1$ ist.

4.2.3 Erweiterungsmöglichkeiten

Bei der Implementierung der API und die Nutzung der API in der Demonstrationsanwendung, sind einige Erweiterungsmöglichkeiten aufgefallen. Dieser Abschnitt zeigt diese auf und gibt einen Ausblick, welche Vorteile damit verbunden wären.

Die Hauptfrage, die sich bei der Entwicklung der API gestellt hat, war, wie neue prozedurale Funktionalitäten in die Grundklasse BaseModel integriert werden können. Dies ist im Moment nur möglich, indem die Grundklasse BaseModel direkt angepasst und dort eine neue Funktion implementiert wird.

An einigen Stellen der API ist ein weiteres Parallelisierungspotential möglich. So können einige Funktionen wie `repeat`, oder `divide` neue Threads starten, die einzeln Geometrie aufbauen und diese Geometrien nach Beendigung wieder zusammenführen.

Die API könnte ebenfalls mittels Geometry Shaders beschleunigt werden. Statt eine Ausgabe von C++-Klassen, könnte PCGL auch Shader-Dateien ausgeben, die die Geometrie erstellen.

Aktuell fehlt die Möglichkeit, bestehende Modelle in den Programmablauf zu integrieren. Die API könnte dahingehend erweitert werden, dass sie Modelle aus Dateien lädt und für andere Funktionen zur Verfügung stellt.

Kapitel 5

Evaluation

Dieses Kapitel evaluiert die vorangegangenen Implementierungen. Die Evaluation wird in zwei Abschnitte aufgeteilt. Der erste Teil beschäftigt sich mit der Programmiersprache PCGL und untersucht sie im Rahmen der Forschungsfragen hinsichtlich ihrer Nutzbarkeit.

Der zweite Teil analysiert, ob die API fähig ist, die in der Anforderungsanalyse betrachteten Asteroiden und Kometen zu generieren. Anhand einer Histogrammanalyse wird die Ähnlichkeit zu den Asteroiden Lutetia, Vesta, Churyumov-Gerasimenko und Itokawa gezeigt. Zudem wird mithilfe eines Code-Analyse-Tools die syntaktische Korrektheit des C++-Codes sichergestellt und potentielle Fehlerquellen erkannt.

5.1 PCGL & Transpiler

Als Erstes der drei Hauptkomponenten PCGL, Transpiler und API wird die Sprache PCGL und der für die Nutzung der Sprache notwendige Transpiler evaluiert.

5.1.1 Ruby Code-Analyse:

Die Einhaltung eines Style-Guides ist wichtig, um sauberen, lesbaren und von Dritten wiederverwendbaren Code zu schreiben. Style-Guides beschreiben Vereinbarungen unter Programmierern, wie Code zu strukturieren und zu programmieren ist, um eine gute Lesbarkeit garantieren zu können. Auch potentielle Fehlerquellen können durch geeignete Werkzeuge entdeckt werden. Dies ist bei Ruby aufgrund von dynamischer Typisierung nicht einfach. Zur Messung der Einhaltung von Style-Guides gibt es diverse frei verfügbare Tools. Diese prüfen auch mittels statischer Code-Analyse auch auf mögliche Fehler. Für die Programmiersprache Ruby, in der auch der Transpiler geschrieben ist, gibt es das von Batsov entwickelte Programm

`rubocop` [Bat16]. Dieses wird zur Analyse herangezogen. Der Style-Guide, der in dieser Masterarbeit genutzt wird, ist in Quelle [Bat+16] ersichtlich.

Zur Messung des Wertekriteriums der Einhaltung des Ruby-Style-Guides wird das Programm `rubocop` benutzt. Während der Programmierung des Transpilers wurde versucht, möglichst viele Punkte des Style-Guides einzuhalten. Bei der Analyse mit `rubocop` schneidet der Transpiler mit folgendem Ergebnis ab:

```
[...]
26 files inspected, no offenses detected
```

Das Ergebnis besagt, dass innerhalb von 26 Dateien keine Verstöße erkannt wurden.

5.1.2 Komplexitätsverringerung

Ziel von PCGL ist es, die Komplexität der Erstellung von prozeduralen Modellen zu verringern. Dies soll unter diesem Punkt genauer betrachtet werden, indem gezeigt wird, dass die Nutzung von PCGL eine Reduzierung der Komplexität zur Folge hat.

Mit Lindenmayer-Systemen wird die Erstellung von Modellen komponentenbasiert. Durch Regeln lassen sich Teilmodelle definieren, die daraufhin durch Kombination dieser Teilmodelle zu größeren Modellen zusammengesetzt werden können. Dadurch wird auch implizit eine Parallelisierbarkeit des Erstellungsprozesses abgebildet, die die Erstellung beschleunigen kann.

Um die Reduzierung der Komplexität zu zeigen, wird die McCabe-Metrik angewandt [McC76]. Die McCabe-Metrik berechnet die zyklomatische Komplexität einer Software-Komponente. Dazu wird eine Komponente in einen gerichtetet Graph transformiert, an dem die Metrik angewandt wird.

Exemplarisch wird die Reduzierung der McCabe-Komplexität anhand des Listings 5.1 erläutert.

Dort existieren zwei Regeln, die zu unterschiedlichen Wahrscheinlichkeiten genutzt werden. Die Entscheidung, wann welche Regel eintritt, wurde durch PCGL in die Funktionsdefinition integriert. Dadurch wurde die McCabe-Komplexität der individuellen Regel reduziert.

5.1.3 Erweiterbarkeit:

Auch nach der Entwicklung des Transpilers und der Spezifizierung der Programmiersprache PCGL sollen noch Weiterentwicklungen möglich sein. Damit einhergehend muss der Transpiler erweiterbar sein.

In diesem Abschnitt werden mögliche Erweiterungen vorgestellt, und demonstriert, wie diese implementiert werden könnten. Dadurch wird gezeigt, dass bei der Implementierung des Transpilers für PCGL auf die Erweiterbarkeit Rücksicht genommen wurde.

Listing 5.1 Ein Asteroid in PCGL

```
asteroid
  sphere 5
  repeat 300, impact

impact : 2
  set p1, random_point
  select_sphere p1, random_float * 4 + 1, 0.005
  normalize p1
  translate_selection -p1 * random_float * 20%
  select_none

impact : 1
  set p1, random_surface_point
  set p2, p1
  set radius, random_float * 0.2 + 0.2
  select_sphere p2,radius, 0.2
  normalize p1
  set td, -p1 * (random_float+0.5) * 0.2
  translate_selection td * 0.2
  select_none
  select_sphere p2,radius * 1.25,0.2
  randomize_selection 0.1f
  translate_selection td * -0.2
  select_sphere p2,radius,0.2
  translate_selection td * 0.1
  select_none
```

Generell können drei verschiedene Arten von Erweiterungsmöglichkeiten betrachtet werden:

1. Auf die Sprache bezogene Erweiterungen führen zu keiner Anpassung in der zugrunde liegenden API. Dazu gehört beispielsweise die Integration von anonymen Regeln. Statt

   ```
   start_rule
     repeat 50, simple_sphere

   simple_sphere
     sphere random_float
   ```

   könnte die Regel `simple_sphere` auch direkt als anonyme Regel definiert werden:

   ```
   start_rule
     repeat 50
       sphere random_float
   ```

   Der Transpiler würde intern jedoch eine Regel erzeugen, die die anonyme Regel darstellen würde. Die API müsste dementsprechend nicht angepasst werden. Für eine solche Erweiterung müsste der `OutputVisitor` des Transpilers angepasst werden.
2. Auf die API bezogene Erweiterungen führen zu keiner Anpassung bei der Syntaxdefinition von PCGL. Dazu gehören beispielsweise neue interne Funktionen, wie z.B. `donut`, die in die API implementiert und sofort in PCGL genutzt werden können.
3. Erweiterungen, die sowohl die API als auch die Syntax von PCGL berühren, ist beispielsweise die Implementierung einer Beschränkung der Iterationstiefe von Regeln mittels eines Syntaxelements.
   So könnte folgender Code

   ```
   start_rule/50
     start_rule
   ```

zu keinem Absturz führen, da eine Beschränkung der Iterationstiefe auf 50 vorgenommen wurde. Um diese Erweiterung zu integrieren, muss sowohl eine Anpassung am `OutputVisitor` als auch am eigentlichen Parser der Eingabesprache vorgenommen werden.

5.1.4 Vor- und Nachteile der Nutzung von PCGL

PCGL bringt dem Nutzer einige Vor- und Nachteile, die in diesem Abschnitt genauer beleuchtet werden.

Da zum Zeitpunkt der Entstehung der Masterarbeit kein Interpreter entwickelt wurde, der PCGL interpretieren kann, ist es notwendig, einen Transpiler zu verwenden. Dieser Transpiler muss bei Benutzung in die Toolchain des Nutzers integriert werden. Da der aktuelle Transpiler auf Basis von Ruby entwickelt wurde, muss zunächst eine Ruby-Umgebung auf dem Nutzungscomputer installiert werden. Dies ist auf allen Unix-basierten Betriebssystemen unproblematisch, da vorkonfigurierte Pakete für die einzelnen Distributionen zur Verfügung stehen. Problematischer wird dies bei Windows, wo eine Ruby Installation nicht einfach, aber dennoch möglich ist.

Ein weiterer Nachteil ist das Erlernen einer neuen Sprache. Um diesen Nachteil zu reduzieren, ist es wichtig, dass neue Sprachen nur wenige exotische Syntaxelemente einführen. Dies ist auch Anforderung in Kapitel 3.2 gewesen.

Die Vorteile von PCGL sind neben der klaren Trennung von Modelldaten und der Logik des Ausgangsprogramms auch die Abstraktion der Ausgangssprache zu einer domänenspezifischen Sprache. Dadurch ist es möglich, wiederkehrende Elemente, wie das Definieren von parametrisierten Regeln, domänenspezifisch abzubilden.

5.1.5 Werturteil

Die Analyse hat gezeigt, dass alle Leistungsstandards erfüllt wurden. Der Autor dieser Masterarbeit möchte damit nachweisen, dass sowohl Transpiler als auch PCGL im Sinne der Verwendung einwandfrei funktionieren. Der Transpiler wurde unter dem Aspekt der guten Erweiterbarkeit entwickelt.

Der Ruby Code wurde von rubocop analysiert und enthält keine typischen Fehlerquellen. Durch die Nutzung von PCGL reduziert sich die Komplexität. Der Hauptvorteil von PCGL ist die Trennung von Quell-Code und prozeduralen Modell-Daten.

5.2 Prozedurale API und Demonstrationsanwendung

Dieser Abschnitt evaluiert die prozedurale API und die Demonstrationsanwendung. Dabei wird auch das erzeugte Modell genauer betrachtet. Unter Berücksichtigung der Eingangsfrage dieser Masterarbeit wird bewertet, ob mit der prozeduralen API realitätsnahe Modelle von Asteroiden erzeugt werden können.

Des Weiteren wird evaluiert, ob sich die API leicht erweitern lässt. Um nachträglich Funktionen hinzufügen zu können, sollte die API Möglichkeiten bieten, mit minimalem Aufwand eigene Funktionalitäten zu implementieren, die dann auch in PCGL genutzt werden können.

Dieser Abschnitt definiert die Leistungsstandards, die mit den Wertekriterien im vorherigen Abschnitt verbunden sind, und analysiert die Implementierung anhand dieser.

5.2.1 Code-Analyse der Implementierung

In dieser Masterarbeit dient eine statische Code-Analyse dazu, mögliche Fehler zu identifizieren. Bei einer statische Code-Analyse wird im Gegensatz zu einer dynamischen Code-Analyse der Programmquellcode untersucht. Dies kann der vom Programmierer erzeugte Quellcode oder auch von einem Compiler erzeugte Bytecode sein.

Ein Tool, das eine statische Code-Analyse für Projekte, die C++ als Ausgangssprache nutzen, durchführt, ist CppCheck. Es teilt Mängel in folgende Klassen ein:

- Fehler
- Warnungen
- Stilbruch
- Portabilitätswarnungen
- Performancewarnungen

CppCheck stellte bei der Code-Analyse keine Mängel fest.

5.2.2 Realitätsnähe

Ein wesentliches Ziel dieser Masterarbeit ist, die markanten Oberflächenmerkmale von Asteroiden mittels prozeduralen Algorithmen zu modellieren. Dabei soll aber auch ein breites Spektrum an Asteroiden generiert werden können. Dazu werden in diesem Abschnitt nun exemplarisch drei Asteroiden generiert und mithilfe einer Histogrammanalyse der Topologie evaluiert.

Als erstes wird ein Skript definiert, das ein Modell erzeugt, das eine hohe Ähnlichkeit zu Lutetia aufweisen soll. Dieses Skript ist in Listing 5.2 zu sehen.

Listing 5.2 Ein Asteroid in PCGL

```
asteroid
  sphere 5
  select_all
  scale_selection random_float * 0.4 + 0.7, random_float * 0.2 + 0.6, random_float * 0.2 + 0.6
  select_none

  repeat 200, impact
  repeat 100, crater
  select_all
  apply_noise_field boulder_biome


implicit boulder_biome
  noise3(x * 39.2f, y*39.2f, z*39.2f) * noise3(x*5.2f,y*5.2f,z*5.2f) * 0.2f * max(0.0f, noise3(x,y,z)-0.5f)

impact
  set p1, random_point
  select_sphere p1, random_float * 4 + 1, 0.005
  normalize p1
  translate_selection -p1 * random_float * 20%
  select_none

crater
  set p1, random_surface_point
  set p2, p1
  set radius, random_float * 0.2 + 0.2
  select_sphere p2,radius, 0.2
  normalize p1
  set td, -p1 * (random_float+0.5) * 0.2
  translate_selection td * 0.2
  select_none
  select_sphere p2,radius * 1.25,0.2
  randomize_selection 0.1f
  translate_selection td * -0.2
  select_sphere p2,radius,0.2
  translate_selection td * 0.1
  select_none
```

Der Quellcode im Listing 5.2 sind vier Regeln zu sehen. Diese beschäftigen sich mit folgenden in der Anforderungsanalyse definierten Merkmalen.

1. `asteroid`:
   In dieser Regel wird die Grundform des Asteroiden bestimmt und mittels simulierten Einschlägen verformt. Er besteht aus einer Kugel mit dem Radius 4, die dann mittels der Skalierfunktion deformiert wird. Die Skalierung ist zufällig. Danach werden 200 große und 100 kleinere Einschläge mittels der Regeln `impact` und `crater` erstellt. Zum Schluss wird auf den ganzen Asteroiden noch ein spezielles Rauschen moduliert, das die Biome mit der unterschiedlichen Gerölldichte simuliert.
2. `implicit biome`:
   Die Biome werden mittels einer speziellen Regel definiert, die pro Vertex ein mal ausgeführt wird. Dazu werden drei Noise-Funktionen miteinander moduliert, um ein unregelmäßiges Rauschen zu erzeugen. Der letzte Faktor führt zur Bildung von glatten, nicht modifizierten Oberflächen, und rauhen, von der Noise-Funktion modifizierten Oberflächen.
3. `impact`:

Diese Regel stellt einen großen Impakt dar, der die Grundform des Asteroiden maßgeblich verändert.

4. `crater`:
   Die kleineren Impakte erstellen Krater mit Kraterlippen auf der Oberfläche des Asteroiden.

Die gerenderten drei Asteroiden sind in Abbildung 5.1 zu sehen.

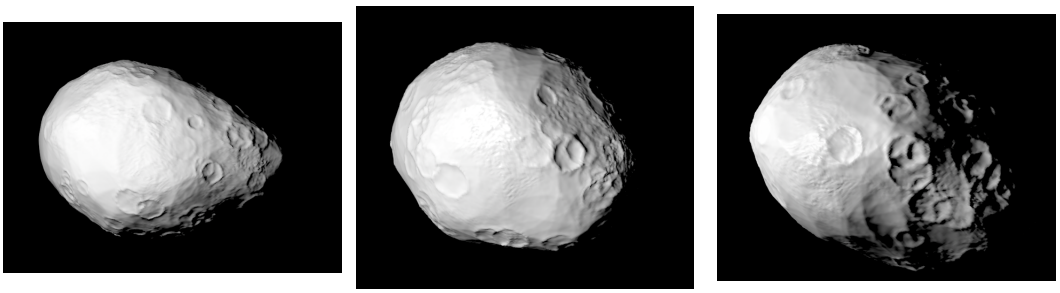

Abbildung 5.1 Generierte Asteroiden #1, #2 und #3

Die erstellten Asteroiden weisen eine subjektive Ähnlichkeit zu Lutetia auf. Um diesen subjektiven Eindruck zu messen, wird ein Modellanalysetool eingesetzt. Dieses Modellanalysetool stammt aus der Arbeitsgruppe Computergrafik der Universität Bremen und untersucht Modelle, indem es 5 Parameter (α, β, γ, δ, κ) zwischen allen Punkten samt Normalen des Modells berechnet. Diese Parameter werden daraufhin zu einer ganzen Zahl zwischen 1 und 5 quantisiert. Die einzelnen Parameter sind wie folgt zwischen zwei Punkten p_1 und p_2 aufzuschlüsseln:

α: Winkel zwischen der Normale von p_1 und p_2.
β: Winkel zwischen der Normale von p_1 und dem Vektor v, der mittels der Basis $(u = n_1, v, w)$ ermittelt wird.
γ: Winkel zwischen der Normale von p_1 und dem Vektor zwischen $p_2 - p_1$.
δ: $|p_2 - p_1|$.
κ: Die Krümmung des Punkts p_1.

Nach der Quantisierung entstehen 3125 Möglichkeiten, nach denen alle Punkte kategorisiert werden. Die Menge, die in einer solchen Kategorie enthalten ist, bilden dann zusammen mit den Möglichkeiten ein Histogramm, das als Grundlage für diese Evaluation dienen soll.

Um die Histogramme miteinander vergleichen zu können, ist es notwendig, eine Distanzfunktion zu definieren, mit der die Ähnlichkeit quantifiziert werden kann. Dafür eignet sich die Chi-Quadrat-Funktion [PW10]:

$$\chi^2(P, Q) = \sum_i \frac{(P_i - Q_i)^2}{P_i + Q_i}$$

Dadurch gilt: je kleiner $\chi^2(P, Q)$ ist, desto ähnlicher sind P und Q.

Nun werden die erzeugten drei Asteroiden auf ihre Ähnlichkeit hin zu allen anderen Asteroiden untersucht. Dies ist notwendig, um einen Grenzwert zu erhalten, bei dem ein Asteroidenpaar als ähnlich aussehend

betrachtet wird. Dabei wird auf die Auswahl der Asteroiden aus der Anforderungsanalyse zurückgegriffen:

1. Lutetia [1]
2. Vesta [2]
3. Churyumov-Gerasimenko [3]
4. Itokawa [4]

Zu allen vier Asteroiden bzw. Kometen existieren Modelldaten mit ähnlicher Polygonzahl wie die generierten Asteroiden (ca. 200.000 – 250.000 Polygone).

Nach Anwendung von χ^2 auf alle Asteroiden entsteht folgende symmetrische Matrix:

	Asteroid #1	Asteroid #2	Asteroid #3	Churyumov	Itokawa	Lutetia	Vesta
Asteroid #1	0,0	0,019	0,013	0,317	0,226	0,032	0,262
Asteroid #2	0,019	0,0	0,007	0,339	0,241	0,061	0,186
Asteroid #3	0,013	0,007	0,0	0,317	0,214	0,047	0,212
Churyumov	0,317	0,339	0,317	0,0	0,099	0,232	0,509
Itokawa	0,226	0,241	0,214	0,099	0,0	0,159	0,413
Lutetia	0,032	0,061	0,047	0,232	0,159	0,0	0,308
Vesta	0,262	0,186	0,212	0,509	0,413	0,308	0,0

Tabelle 5.1 Asteroid #1,#2 und #3 im Vergleich zu den anderen Asteroiden

Die Matrix führt zu folgenden Beobachtungen:

1. Asteroiden #1, #2 und #3 zeigen eine sehr hohe Ähnlichkeit auf ($< 0{,}02$).
2. Die generierten Asteroiden haben Ähnlichkeit mit Lutetia ($< 0{,}07$).
3. Asteroiden #1 und #3 haben eine länglichere Form als Asteroid #2. Dadurch ist die Ähnlichkeit von #1 und #3 zu Itokawa, der ebenfalls eine länglichere Form hat, höher.
4. Der Unterschied der generierten Asteroiden zu Itokawa ist geringer als der Unterschied von Vesta und Itokawa. Gleiches gilt für Vesta und Churyumov.
5. Das Spektrum der Asteroiden des erstellten Skripts ist relativ klein. Die maximale Abweichung beträgt 0,02.

Die Analyse zeigt auch, dass ähnliche Asteroiden eine χ^2-Differenz von maximal 0,07 aufweisen. Auf diese Differenz wird nachfolgend der Grenzwert gelegt, ab den ein erzeugter Asteroid zu einem anderen Asteroid unterschiedlich ist. Churyumov-Gerasimenko, Itokawa, Vesta und Lutetia sind subjektiv unterschiedlich und besitzen einen Wert, der oberhalb dieses Grenzwerts liegt.

[1] Modelldaten von *http://space.frieger.com/asteroids/data/asteroids/models/l/21_Lutetia_250k.obj*
[2] Modelldaten von *http://space.frieger.com/asteroids/data/asteroids/models/v/4_Vesta_256k.obj*
[3] Modelldaten von *http://space.frieger.com/asteroids/data/comets/models/67P_ESA_NAVCAM_Jul2015data_256k.obj*
[4] Modelldaten von *http://space.frieger.com/asteroids/data/asteroids/models/i/25143_Itokawa_200k.obj*

Um nun zu zeigen, dass die zusammengesetzten Asteroiden Churyumov-Gerasimenko, Itokawa und Vesta ebenfalls in PCGL spezifiziert werden können, wird wie folgt vorgegangen. Für jeden dieser drei Asteroiden wird ein PCGL-Skript erzeugt, das 3D-Modelle von Asteroiden generiert, die subjektive Ähnlichkeit aufweisen. Daraufhin wird ein Histogramm für diesen Asteroiden erzeugt, das wiederum mit den anderen Histogrammen verglichen wird.

Itokawa wird als erstes mittels eines PCGL-Skriptes prozedural erzeugt.

Listing 5.3 Itokawa-Replikation in PCGL

```
asteroid
  ellipsoid random_float * 2 + 6.2, random_float * 1.2 + 2.9, random_float * 1.2 + 2.9, 8
  repeat 60, impact
  compress
  repeat 40, impact
  select_all
  apply_noise_field boulder_biome, 30%

compress
  select_x_gradient vec(2,0,0), 8
  scale_selection 90%, 80%, 80%
  select_x_gradient vec(2,0,0), 4
  scale_selection 90%, 70%, 70%

deform
  select_sphere vec(0,0,0), 6
  translate_selection random_vector

implicit boulder_biome
  25% * noise3(x*32.2f,y*32.2f,z*32.2f)+0.5f * noise3(x*12.2f,y*12.2f,z*12.2f)+noise3(x*5.2f,y*5.2f,z*5.2f)) * max↩
      (0.0f, noise3(x,y,z)-0.5f)

impact
  set p1, random_surface_point
  select_sphere p1, random_float * 3 + 1, 3%
  translate_selection -p1.normal * 25%
```

Das Skript aus Listing 5.3 erzeugte den Asteroid in Abbildung 5.2. Subjektiv erkennt man, dass eine ähnliche Grundform hergestellt werden konnte. Jedoch ist die Oberflächenbeschaffenheit anders als bei dem originalen Modell von Itokawa. Das könnte auch die erhaltenen Werte aus Tabelle 5.2 erklären. Diese sagen aus, dass der erstellte Asteroid keinerlei Ähnlichkeit aufweist. Das könnte daran liegen, dass vermutlich der zugrundeliegende Algorithmus eine negativ auswirkende Gewichtung der Merkmale aufweist. Die Oberflächenbeschaffenheit wird weitaus mehr gewichtet als die Grundform, obwohl die Ähnlichkeit eines Objektes sich hauptsächlich durch die Grundform definiert. Der Autor des Modellanalyse-Algorithmus wurde dahingehend in Kenntnis gesetzt.

Ähnlich wird bei dem Asteroiden Churyumov-Gerasimenko vorgegangen. Dazu wird ein Skript in PCGL geschrieben, das die Form von Churyumov-Gerasimenko repliziert. Dieses Skript ist in Listing 5.4 zu sehen. Der Teil des Skripts, der für die Hantelform des Asteroiden zuständig ist, ist in der Regel `compress` ersichtlich. Neben einer Translation der Vertices in der Mitte des Asteroiden zum Zentrum hin, werden die nicht skalierten Vertices leicht um das Zentrum rotiert. Dadurch entsteht die charakteristische Form von Churyumov-Gerasimenko.

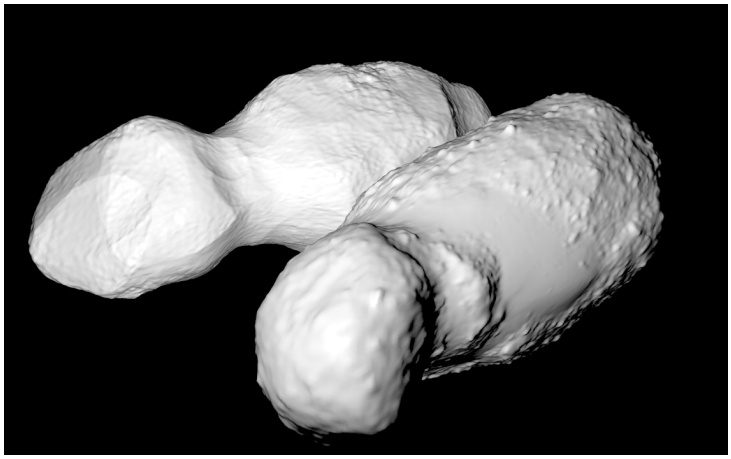

Abbildung 5.2 Generiertes Itokawa-Replikat: Asteroid #4 (links) und das Original (rechts)

	Churyumov	Itokawa	Asteroid #4	Lutetia	Vesta
Churyumov	0	0,099	0,471	0,232	0,509
Itokawa	0,099	0	0,602	0,159	0,413
Asteroid #4	0,471	0,602	0	0,616	0,853
Lutetia	0,232	0,159	0,616	0	0,308
Vesta	0,509	0,413	0,853	0,308	0

Tabelle 5.2 Asteroid #4 im Vergleich zu den anderen Asteroiden

Der erzeugte Asteroid ist mithilfe der Histogrammanalyse mit den anderen Asteroiden verglichen worden. Abbildung 5.3 zeigt den erzeugten Asteroiden zusammen mit einem Modell von Churyumov-Gerasimenko. Auch hier ist die subjektive Ähnlichkeit ersichtlich. Dies zeigt sich auch in Tabelle 5.3. Mit einer Differenz von 0,01 ist das erstellte Churyumov-Gerasimenko-Replikat sehr ähnlich zum Referenz-Modell.

	Asteroid #5	Churyumov	Itokawa	Lutetia	Vesta
Asteroid #5	0	0,01	0,135	0,276	0,542
Churyumov	0,01	0	0,099	0,232	0,509
Itokawa	0,135	0,099	0	0,159	0,413
Lutetia	0,276	0,232	0,159	0	0,308
Vesta	0,542	0,509	0,413	0,308	0

Tabelle 5.3 Asteroid #5 im Vergleich zu den anderen Asteroiden

Als letzter Asteroid wird Vesta erzeugt. Dazu wurde das Skript in Listing 5.5 erstellt.

Listing 5.4 Churyumov-Replikation in PCGL

```
asteroid
  ellipsoid random_float*2 + 8.2, random_float*1.2 + 2.9, random_float*1.2 + 2.9, 8

  repeat 60, impact
  compress
  repeat 100, impact
  repeat 200, crater

  select_all
  apply_noise_field boulder_biome, 20%

compress
  set v2, random_surface_point
  select_x_gradient vec(0,0,0), 9
  scale_selection 100%, 40%, 40%
  invert_selection
  scale_selection 50%, 100%, 100%
  rotate_z_selection 0.3
  select_x_gradient vec(0,0,0), 12
  invert_selection
  scale_selection 110%, 100%, 100%

deform
  select_sphere vec(0, 0, 0), 6
  translate_selection random_vector


implicit boulder_biome
    noise3(x*39.2f,y*39.2f,z*39.2f)*noise3(x*5.2f,y*5.2f,z*5.2f) * max(0.0f, noise3(x,y,z)-0.5f);

impact
  set p1, random_surface_point
  select_sphere p1, random_float * 3 + 1, 3%
  translate_selection -p1.normal * 25%

crater
  set p1, random_surface_point
  set radius, random_float * 0.8 + 0.2
  select_sphere p1, radius, 2%
  set td, -p1.normal * (random_float + 0.5) * 70%
  translate_selection td * 20%
  select_sphere p1, radius * 125%, 2%
  translate_selection -td * 10%
```

Abbildung 5.4 stellt den erzeugten Asteroiden #6 dar. Subjektiv zeigen sich deutliche Ähnlichkeiten. Diese Ähnlichkeit zeigt sich auch in Tabelle 5.4, die eine Histogrammdifferenz von unter 0,07 aufweist. Zufällig sind auch zwei Einschlagskrater entstanden, die eine ähnliche Verteilung aufweisen, wie auf dem Originalmodell von Vesta.

5.2.3 Werturteil

Die Evaluation hat gezeigt, dass mit PCGL und der dazugehörigen API realistische Asteroiden generiert werden können. Die in der Anforderungsanalyse eingehend betrachteten Asteroiden konnten durch verschiedene API-Funktionen nachgebildet werden. Einige Funktionen, wie z.B. die Geröllbildung, wurden jedoch

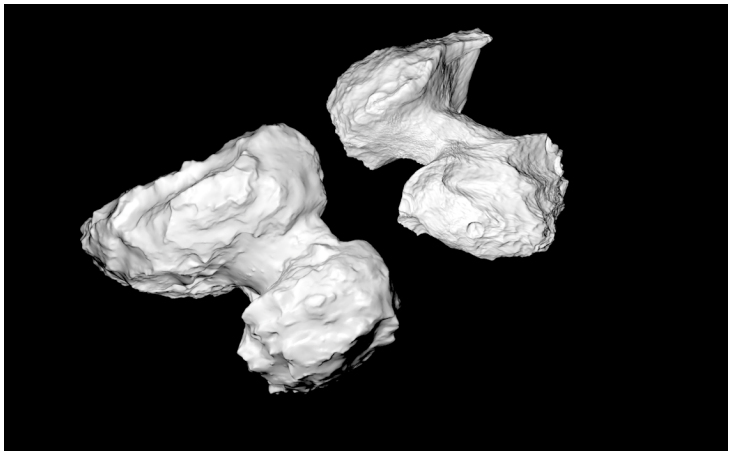

Abbildung 5.3 Generiertes Churyumov-Gerasimenko-Replikat: Asteroid #5 (rechts) und Original (links)

Listing 5.5 Vesta-Replikation in PCGL

```
asteroid
  ellipsoid random_float*2 + 5.2, random_float*2 + 5.2, random_float*2 + 5.2, 8

  repeat 50, impact
  repeat 200, crater

  select_all
  apply_noise_field boulder_biome, 20%

implicit boulder_biome
  noise3(x*39.2f,y*39.2f,z*39.2f)*noise3(x*5.2f,y*5.2f,z*5.2f) * max(0.0f, noise3(x,y,z)-0.5f);

impact
  set p1, random_surface_point
  select_sphere p1, random_float * 3 + 1, 3%
  translate_selection -p1.normal * 25%

crater
  set p1, random_surface_point
  set radius, random_float * 0.8 + 0.2
  select_sphere p1, radius, 2%
  set td, -p1.normal * (random_float + 0.5) * 70%
  translate_selection td * 20%
  select_sphere p1, radius * 125%, 2%
  translate_selection -td * 10%
```

nicht eingesetzt, da die Erstellungszeit der Histogramme mit der Zahl der Polygone exponentiell ansteigt. Die Anzahl der Polygone wurde auf 250.000 – 500.000 begrenzt. Ein Replikat von Itokawa mit Geröllbildung wurde umgesetzt, jedoch keine Evaluation dazu durchgeführt. Siehe dazu A.4.

Die Nutzung von Lindenmayer-Systemen führte dazu, dass Merkmale der Asteroiden als Regeln verfasst wurden. Diese Regeln enthalten hauptsächlich Befehlsanweisungen, die keinerlei Geometrie erzeugen, sondern diese lediglich modifizieren.

Nichtsdestotrotz konnten dadurch wiederholende Elemente, wie `impact` und `crater`, abstrahiert und wie-

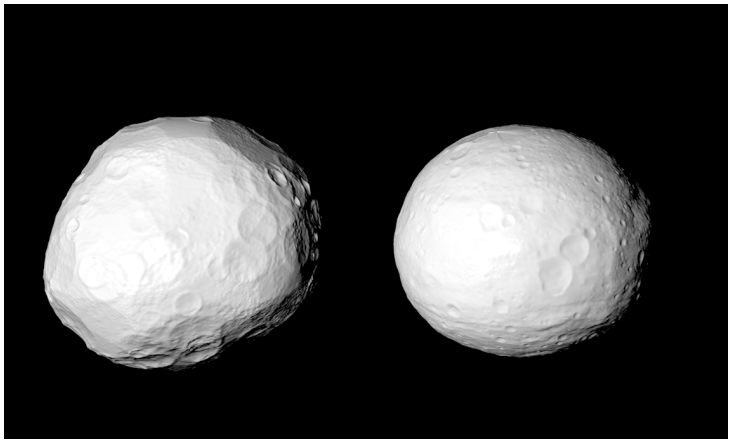

Abbildung 5.4 Generiertes Vesta-Replikat: Asteroid #6 (links) und Original (rechts)

	Churyumov	Itokawa	Lutetia	Asteroid #6	Vesta
Churyumov	0	0,099	0,232	0,44	0,509
Itokawa	0,099	0	0,159	0,314	0,413
Lutetia	0,232	0,159	0	0,274	0,308
Asteroid #6	0,44	0,314	0,274	0	0,068
Vesta	0,509	0,413	0,308	0,068	0

Tabelle 5.4 Histogrammanalyse von Asteroid #6 im Vergleich zu den anderen Asteroiden

derverwendbar gemacht werden.

Kapitel 6

Fazit

Diese Masterarbeit hat gezeigt, wie unter Zuhilfenahme von Lindenmayer-Systemen Asteroiden erzeugt werden können. L-Systeme wurden im zweiten Kapitel als potentielle Hilfsstruktur zur Erstellung von Geometrie ausgewählt, nachdem andere wissenschaftliche Arbeiten, die L-Systeme thematisieren, untersucht wurden. Dabei zeigte sich, dass die L-Systeme zwar als Grundlage genutzt werden können, sie jedoch um einige Punkte erweitert werden mussten, sodass Geometrie mit ihnen erzeugt werden kann. Dies wurde im vierten Kapitel umgesetzt. Dazu wurde die im dritten Kapitel konzipierte Sprache PCGL implementiert. PCGL setzt Lindenmayer-Systeme in C++ um, indem sie die sogenannten *functional bindings* von C++11 nutzt und damit Verknüpfungen von Regeln herstellt. Die Transformation von PCGL nach C++ benötigte einen Transpiler, der ebenfalls in Kapitel 4 umgesetzt wurde. PCGL setzt die in Kapitel 3 spezifizierte API zur Erzeugung von prozeduralen Inhalten voraus. Zur Ermittlung der Grundfunktionalität der API wurde in Kapitel 3 eine ausgiebige Anforderungsanalyse anhand realer Asteroiden durchgeführt. Dabei wurden spezifische Merkmale dieser Asteroiden ermittelt und Algorithmen zur Erzeugung dieser Merkmale beschrieben. Diese Algorithmen wurden anschließend im Detail im vierten Kapitel umgesetzt. In der Evaluation wurden darauffolgend Skripte in PCGL entwickelt, die die Asteroiden der Anforderungsanalyse erzeugten. Dies ist beim initialen Skript für Lutetia gelungen. Die Histogrammanalyse zeigte eine minimale Differenz zwischen den erstellten Asteroiden und dem 3D-Modell von Lutetia. Bei Vesta, Itokawa und Churyumov-Gerasimenko zeigte sich jedoch ein großer Unterschied. Daher wurden für diese weitere Skripte erstellt, die Modelle von Vesta, Itokawa und Churyumov-Gerasimenko generierten. Bis auf Itokawa konnte mit PCGL sowohl subjektiv überzeugende als auch objektiv mittels Histogramme übereinstimmende Modelle erzeugt werden. Das Replikat von Itokawa zeigte trotz augenscheinlicher Ähnlichkeit eine sehr hohe Differenz bei der Histogrammanalyse auf. Dies führt der Autor auf eine übermäßige Gewichtung der Oberflächenmerkmale zurück.

Zusammenfassend hat sich gezeigt, dass die Nutzung von Lindenmayer-Systemen als Grundstruktur zur Erstellung von Geometrien nur einen geringen Vorteil verschafft. Die Nutzung von Regeln zur Aufteilung der Szene in Grundobjekte eignete sich nur bedingt. Lediglich einzelne Merkmale wie Krater, oder Deformierungen der Grundform, konnten als Regeln in einem L-System abstrahiert werden. Alle weiteren Erstellungsschritte ließen sich auch mit anderen Methoden erzeugen.

Nach Betrachtung der Implementierungsentscheidungen gäbe es einige Ansatzpunkte, das Prinzip von L-Systemen besser zu nutzen. Folgende Anpassung müssten vorgenommen werden, um dem Prinzip einen entscheidenden Vorteil gegenüber anderen Methoden der prozeduralen Geometrieerzeugung zu geben.

- Jede Regel muss ihren eigenen Geometrieraum erzeugen, der beim Verlassen der Regel in den übergeordneten Geometrieraum eingefügt wird.
- Durch das Entkoppeln der Geometrieräume könnten Regeln parallel ausgewertet und Geometrie gleichzeitig erzeugt werden.

Trotzdem haben die Implementierungen der API und der Sprache PCGL gezeigt, wie die markanten Oberflächenmerkmale von Asteroiden mit prozeduralen repliziert werden können. Dazu wurden Geometriemanipulationsmethoden bereit gestellt, die laut Evaluation realitätsnahe Asteroiden generieren können. Mithilfe von Lindenmayer-Systemen wurden die Operationen abstrahiert und strukturiert in einer domänenspezifischen Sprache vereint.

Anhang A

Anhang

A.1 Abbildungsverzeichnis

A.2 Tabellenverzeichnis

A.3 Literatur

[Adz+99] Valery Adzhiev, Richard Cartwright, Eric Fausett, Anatoli Ossipov, Alexander Pasko und Vladimir Savchenko. *HyperFun project: a framework for collaborative multidimensional F-rep modeling*. 1999. URL: *http://hyperfun.org/HF_paper.pdf*.

[Age05] Japan Aerospace Exploration Agency. *Hayabusa's Scientific and Engineering Achievements during Proximity Operations around Itokawa*. 2005. URL: *http://www.isas.ac.jp/e/snews/2005/1102.shtml* (abgerufen am 18. 09. 2016).

[Age14] European Space Agency. *Comet on 19 September 2014 – NavCam*. 2014. URL: *http://www.esa.int/spaceinimages/Images/2014/09/Comet_on_19_September_2014_NavCam* (abgerufen am 10. 10. 2016).

[Age15] European Space Agency. *Comet vital statistics*. 2015. URL: *http://www.esa.int/spaceinimages/Images/2015/01/Comet_vital_statistics* (abgerufen am 22. 09. 2016).

[al11] Holger Sierks et. al. *Rosetta reveals mysterious Lutetia*. 2011. URL: *http://sci.esa.int/rosetta/49543-rosetta-reveals-mysterious-lutetia/*.

[Ash+16] Mike Ashman et al. In: SpaceOps Conferences. American Institute of Aeronautics und Astronautics, Mai 2016. Kap. The Evolution of Rosetta-Philae Science Planning Processes Following the Philae Landing. DOI: *10.2514/6.2016-2488*.

[Bat+16] Bozhidar Batsov, Wael Nasreddine, Eric Turner, Josh Baker und contributors. *ruby style guide*. 2016. URL: *https://github.com/bbatsov/ruby-style-guide* (abgerufen am 19. 09. 2016).

[Bat16] Bozhidar Batsov. *rubocop project github page*. 2016. URL: *https://github.com/bbatsov/rubocop* (abgerufen am 19. 09. 2016).

[Cho14] Charles Q. Choi. *Asteroids: Fun Facts and Information About Asteroids*. 2014. URL: *http://www.space.com/51-asteroids-formation-discovery-and-exploration.html* (abgerufen am 04. 10. 2016).

[DDT11] Isaac M. Dart, Gabriele De Rossi und Julian Togelius. »SpeedRock: Procedural Rocks Through Grammars and Evolution«. In: *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*. PCGames '11. Bordeaux, France: ACM, 2011, 8:1–8:4. ISBN: 978-1-4503-0872-4. DOI: *10.1145/2000919.2000927*.

[Ebe03] David S. Ebert. *Texturing & modeling: a procedural approach*. 3. ed. The Morgan Kaufmann series in computer graphics and geometric modeling. XXIII, 687 S ; 24 cm : Ill. (z.T. farb.), graph. Darst. Amsterdam [u.a.]: Kaufmann, 2003. ISBN: 9781558608481.

[Eli99] Hugo Elias. *Perlin Noise*. 1999. URL: *http://web.archive.org/web/19990210084510/http://freespace.virgin.net/hugo.elias/models/m_perlin.htm* (abgerufen am 11. 10. 2016).

[Fou16] Free Software Foundation. *GSL - GNU Scientific Library*. 2016. URL: *http://www.gnu.org/software/gsl/* (abgerufen am 18. 09. 2016).

[Fuj+06] A. Fujiwara et al. »The Rubble-Pile Asteroid Itokawa as Observed by Hayabusa«. In: *Science* 312.5778 (2006), S. 1330–1334. ISSN: 0036-8075. DOI: *10.1126/science.1125841*. eprint: *http:*

//science.sciencemag.org/content/312/5778/1330.full.pdf. URL: *http://science.sciencemag.org/content/312/5778/1330*.

[Gre11] Tony Greicius. *Full-Frame Image of Vesta*. 2011. URL: *http://www.nasa.gov/mission_pages/dawn/multimedia/pia14317.html* (abgerufen am 04. 10. 2016).

[Han92] James Scott Hanan. *PARAMETRIC L-SYSTEMS AND THEIR APPLICATION TO THE MODELLING AND VISUALIZATION OF PLANTS*. 1992. URL: *http://algorithmicbotany.org/papers/hanan.dis1992.pdf* (abgerufen am 11. 10. 2016).

[Hen+13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden und Alexandru Iosup. »Procedural Content Generation for Games: A Survey«. In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9.1 (Feb. 2013), 1:1–1:22. ISSN: 1551-6857. DOI: *10.1145/2422956.2422957*. URL: *http://doi.acm.org/10.1145/2422956.2422957*.

[Jin04] Pete Jinks. *C-Syntax in BNF*. 2004. URL: *http://www.cs.man.ac.uk/~pjj/bnf/c_syntax.bnf* (abgerufen am 11. 10. 2016).

[Jyl16] Jukka Jylänki. *Game Math and Geometry Library*. 2016. URL: *http://clb.demon.fi/MathGeoLib/nightly/* (abgerufen am 18. 09. 2016).

[LD06] Ares Lagae und Philip Dutré. »Abstract Poisson Sphere Distributions«. In: *Vision, Modeling, and Visualization 2006 (VMV 2006)* (2006).

[Man87] Benoît B. Mandelbrot. *Die fraktale Geometrie der Natur*. 491 S. : Ill., graph. Darst. Basel [u.a.]: Birkhäuser, 1987. ISBN: 9783764317713.

[Mar+12] S. Marchi et al. »The Violent Collisional History of Asteroid 4 Vesta«. In: *Science* 336.6082 (2012), S. 690–694. ISSN: 0036-8075. DOI: *10.1126/science.1218757*. eprint: *http://science.sciencemag.org/content/336/6082/690.full.pdf*. URL: *http://science.sciencemag.org/content/336/6082/690*.

[McC76] T. J. McCabe. »A Complexity Measure«. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dez. 1976), S. 308–320. ISSN: 0098-5589. DOI: *10.1109/TSE.1976.233837*.

[Men+15] Johannes Meng, Marios Papas, Ralf Habel, Carsten Dachsbacher, Steve Marschner, Markus Gross und Wojciech Jarosz. »Multi-scale Modeling and Rendering of Granular Materials«. In: *ACM Trans. Graph.* 34.4 (Juli 2015), 49:1–49:13. ISSN: 0730-0301. DOI: *10.1145/2766949*. URL: *http://doi.acm.org/10.1145/2766949*.

[MPH97] R. Měch, P. Prusinkiewicz und J. Hanan. *Extensions to the graphical interpretation of L-systems based on turtle geometry*. 1997. URL: *http://algorithmicbotany.org/lstudio/graph.pdf* (abgerufen am 11. 10. 2016).

[Pao+16] Alberto Paoluzzi, Giorgio Scorzelli, Simone Portuesi und Franco Milicchio. *aztecpyr*. 2016. URL: *http://www.dia.uniroma3.it/plasm/gallery/simple_examples/aztecpyr.html* (abgerufen am 10. 10. 2016).

[Pey+09] A. Peytavie, E. Galin, J. Grosjean und S. Merillou. »Procedural Generation of Rock Piles using Aperiodic Tiling«. In: *Computer Graphics Forum* 28.7 (2009), S. 1801–1809. ISSN: 1467-8659. DOI: *10.1111/j.1467-8659.2009.01557.x*.

[PL90] Przemyslaw Prusinkiewicz und Aristid Lindenmayer. *The algorithmic beauty of plants*. The virtual laboratory. XII, 228 S : Ill., graph. Darst. New York, NY [u.a.]: Springer, 1990. ISBN: 9780387972978. URL: *http://algorithmicbotany.org/papers/abop/abop.pdf*.

[PP03] Alberto Paoluzzi und Valerio Pascucci. *Geometric programming for computer aided design*. Online Ressource (xx, 776 p.) Hoboken, NJ: J. Wiley, 2003. ISBN: 9780470013885. URL: *http://dx.doi.org/10.1002/9780470013885*.

[Pro16] The CGAL Project. *The Computational Geometry Algorithms Library*. 2016. URL: *http://www.cgal.org/* (abgerufen am 18. 09. 2016).

[PSS10] Przemyslaw Prusinkiewicz, Mitra Shirmohammadi und Faramarz Samavati. »L-systems in Geometric Modeling«. In: *Proceedings Twelfth Annual Workshop on Descriptional Complexity of Formal Systems, DCFS 2010, Saskatoon, Canada, 8-10th August 2010*. 2010, S. 3–14. DOI: *10.4204/EPTCS.31.3*.

[PW10] Ofir Pele und Michael Werman. »The Quadratic-Chi Histogram Distance Family«. In: *Computer Vision – ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part II*. Hrsg. von Kostas Daniilidis, Petros Maragos und Nikos Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 749–762. ISBN: 978-3-642-15552-9. DOI: *10.1007/978-3-642-15552-9_54*.

[Sch97] Ulf Schünemann. *Requirements Specification and Development Principles for Programming Languages*. 1997. URL: *http://web.cs.mun.ca/~ulf/pld/princ.html%5C#Familiarity* (abgerufen am 12. 10. 2016).

[Sie+11] H. Sierks et al. »Images of Asteroid 21 Lutetia: A Remnant Planetesimal from the Early Solar System«. In: *Science* 334.6055 (2011), S. 487–490. ISSN: 0036-8075. DOI: *10.1126/science.1207325*. eprint: *http://science.sciencemag.org/content/334/6055/487.full.pdf*. URL: *http://science.sciencemag.org/content/334/6055/487*.

[SM10] Kaisei Sakurai und Kazunori Miyata. »Procedural Modeling of Multiple Rocks Piled on Flat Ground«. In: SA '10 (2010), 35:1–35:2. DOI: *10.1145/1900354.1900393*. URL: *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.699.5457&rep=rep1&type=pdf*.

[SQ79] A. L. Szilard und R. E. Quinton. *An interpretation for DOL systems by computer graphics*. 1979.

[Tog+11] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley und Cameron Browne. »Search-Based Procedural Content Generation: A Taxonomy and Survey«. In: *Instabilities in Spin-Polarized Vertical-Cavity Surface-Emitting Lasers* 3.3 (2011), S. 172–186. ISSN: 1943068X. URL: *http://dx.doi.org/10.1109/TCIAIG.2011.2148116*.

[TPR15] Mario Toso, Ettore Pennestrì und Valerio Rossi. »ESA multibody simulator for spacecrafts' ascent and landing in a microgravity environment«. In: *CEAS Space Journal* 7.3 (2015), S. 335–346. ISSN: 1868-2510. DOI: *10.1007/s12567-015-0081-5*. URL: *http://dx.doi.org/10.1007/s12567-015-0081-5*.

[Zub+11] Maria T. Zuber, Harry Y. McSween Jr., Richard P. Binzel, Linda T. Elkins-Tanton, Alexander S. Konopliv, Carle M. Pieters und David E. Smith. »Origin, Internal Structure and Evolution of 4

Vesta«. In: *Space Science Reviews* 163.1 (2011), S. 77–93. ISSN: 1572-9672. DOI: *10.1007/s11214-011-9806-8*. URL: *http://dx.doi.org/10.1007/s11214-011-9806-8*.

A.4 Itokawa-Replikat

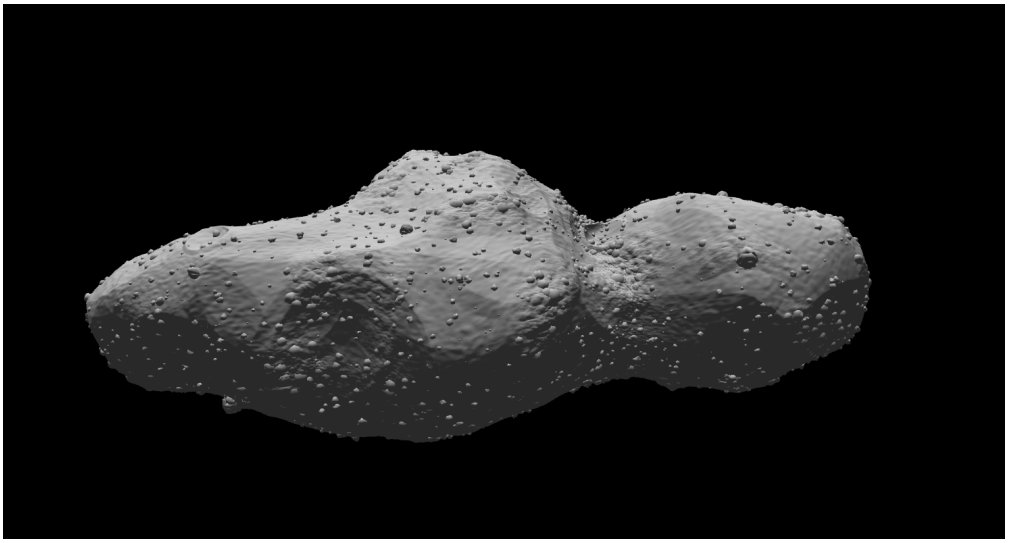

Abbildung A.1 Itokawa-Replikat mit 2,4 Millionen Polygonen und Geröll. Erzeugt mit PCGL. Gerendert in eigener Demonstrationsanwendung.

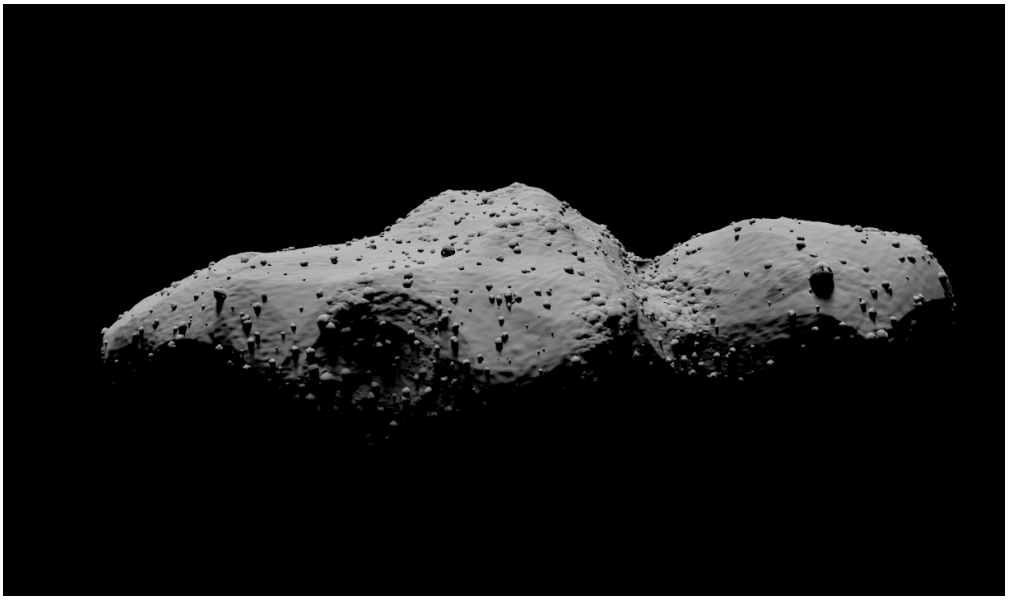

Abbildung A.2 Itokawa-Replikat mit 2,4 Millionen Polygonen und Geröll. Erzeugt mit PCGL. Gerendert in Cinema 4D.

Listing A.1 Itokawa-Replikation in PCGL

```
asteroid
  ellipsoid random_float * 2 + 6.2, random_float * 1.2 + 2.9, random_float * 1.2 + 2.9, 9

  repeat 60, impact
  compress
  repeat 40, impact
  repeat 10, big_boulder
  repeat 20, small_impact
  repeat 3000, small_boulder

  select_all
  apply_noise_field noisy, 25%

compress
  select_x_gradient vec(2, 0, 0), 8
  scale_selection 90%, 80%, 80%
  select_x_gradient vec(2, 0, 0), 4
  scale_selection 90%, 70%, 70%

deform
  select_sphere vec(0, 0, 0), 6
  translate_selection random_vector

big_boulder
    set a, random_vertex
    set b, random_float * 10% + 5%
    select_sphere a, b, 60%, 100.0
    translate_selection a.normal * b
    filter_selection_by_normal_angle a.normal, pi / 2.0
    scale_selection 200%

small_boulder
    set a, random_vertex

    push_clean_model_state
    ellipsoid (random_float + 0.5f)*5.0f, (random_float + 0.5f)*5.0f, (random_float + 0.5f)*5.0f, 2
    select_all
    apply_noise_field small_boulder_noise, 20.0
    select_all
    rotate_x_selection random_float * 4.0
    rotate_y_selection random_float * 4.0
    rotate_z_selection random_float * 4.0
    scale_selection 0.3% * random_float
    recenter_selection
    translate_selection a
    pop_and_merge_model_state

implicit small_boulder_noise
    return pnoise3(x * 302.2, y * 302.2, z * 302.2)

implicit noisy
    return (25% * pnoise3(x*32.2,y*32.2,z*32.2) + 50% * pnoise3(x*12.2,y*12.2,z*12.2)+pnoise3(x*5.2,y*5.2,z*5.2)) ⟵
          * max(0.0, pnoise3(x,y,z)-0.5f)

impact
  set p1, random_vertex
  select_sphere p1,random_float * 3 + 1, 2%
  translate_selection -p1.normal * 25%

small_impact
    set p1, random_vertex
    set radius, random_float * 0.8 + 0.2
    select_sphere p1,radius, 2%
    auto td = -p1.normal * (random_float+0.5) * 70%
    translate_selection td * 20%
    select_sphere p1, radius * 125%, 2%
    translate_selection -td * 10%
```

A.5 CD

URL: *https://github.com/nyon/pcgl*