Vorläufige Dokumentation des CGA-Frameworks

Robert Giacinto
Prof. Dr. Horst Stenzel

Eine kurze Einführung in den Aufbau und die Verwendung des neuen CGA-Frameworks.

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung grafischer Systeme, die Visualisierung von großen Datenmengen oder der Entwurf von Algorithmen innerhalb der Computergrafik als Fachgebiet der Informatik nimmt mit der wachsenden Leistungen und günstigen Verfügbarkeit von Computern eine zunehmende Rolle sowohl in der Industrie als auch in der Ausbildung von Informatikern ein. Ein fundiertes Verständnis der Zusammenhänge und Grundlagen der Methoden der Computergrafik sind daher essentiell und finden sich in unterschiedlicher Gewichtung im Curriculum der verschiedenen Informatikstudiengänge. Innerhalb dieser Ausarbeitung soll die Entwicklung eines modularen Grafiksystems vorgestellt werden, das im Rahmen der Veranstaltung "Computergrafik und Animation" an der Fachhochschule Köln eingesetzt wird, um die theoretischen Kenntnisse aus der Vorlesung in selbstständig zu lösenden Praktikumsaufgaben zu vertiefen.

Der Aufbau der Arbeit gliedert sich wie folgt. In Kapitel 2 soll ein Überblick über die Grundlagen, die der Implementierung zugrunde liegen, gegeben werden. Kapitel 3 konzentriert sich auf das Design der Engine, die Anforderungen, die an sie gestellt werden und wie diese praktisch implementiert werden können. Kapitel 4 gibt einen Überblick über den aktuellen Stand der Arbeit und präsentiert anhand eines kurzen Beispiels die Arbeitsweise der Engine. Kapitel **??** schließt mit einem Fazit ab und gibt Ausblicke auf mögliche Erweiterungen in zukünftigen Versionen.

2 Grundlagen

Innerhalb der Veranstaltung "Computergrafik und Animation" wird das theoretische Fundament der Computergrafik beschrieben. Die Studenten lernen die grundlegenden Algorithmen, Datenstrukturen und Vorgänge, die zu einer grafikschen Ausgabe führen. Parallel zur Vorlesung sollen wesentliche Aspekte in praktischen Übungen nachvollzogen werden können und so den Einblick zu vertiefen. In diesem Kapitel soll auf die vermittelten Grundlagen in einem kurzen Überblick eingegangen werden. Kapitel 3 baut auf diesen Grundlagen auf und versucht die Frage zu klären, wie ein System aussehen sollte und welche Anforderungen erfüllt werden müssen, um die praktische Arbeit mit den gelernten Grundlagen zu unterstützen.

2.1 Grafikpipeline

Bei der Beschreibung der Architektur von (Echtzeit-) Grafiksystemen wird gerne auf die Form der Pipeline zurückgegriffen, um die Funktionskomponenten und deren Zusammenhang zu beschreiben. Eine Pipeline besteht dabei aus einer beliebigen Anzahl von Stufen, die (in der Regel) sequentiell durchlaufen werden. Eine erste grobe Aufteilung der Verarbeitungsstufen ist in Abbildung 1 dargestellt. Die drei Hauptstufen, die in den folgenden Abschnitten näher beschrieben werden sollen, sind die Anwendungsstufe, die Geometriestufe und die Rasterstufe.

Die folgenden Beschreibungen lehnen sich an den Ausführungen in [1] an. Für tiefergehende Informationen und eine Auswahl an weiterführenden Themen sei auf dieses Buch verwiesen.

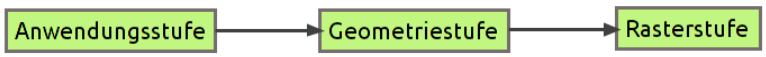

Abbildung 1: Übersicht der Grafikpipeline.

2.1.1 Anwendungsstufe

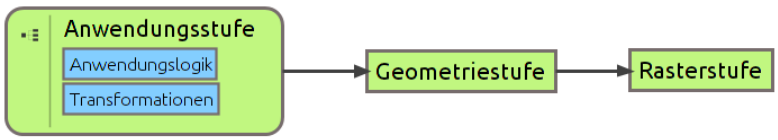

Abbildung 2: Die Anwendungsstufe der Grafikpipeline im Detail.

Bei dieser Stufe handelt es sich um die Stufe, bei der ein Entwickler einer grafischen Anwendung den größten Spielraum und Einfluss hat. Hierbei handelt es sich um die Anwendung, die als Software implementiert die Geometrie generiert oder zur Verfügung stellt, die der Ausgangspunkt für die nächste Phase, die Geometriephase, ist. Beispiele für die Aktivitäten, die in dieser Stufe durchgeführt werden, sind die Animation von Polygonmodellen oder von Texturen, der Kollisionserkennung zwischen einzelnen Entitäten oder dem Morphing von Geometrien.

2.1.1.1 Transformationen

Transformationen sind ein grundlegendes Tool in der Computergrafik, mit denen Objekte verändert werden können. An dieser Stelle sollen die wichtigsten Vertreter von Transformationen kurz vorgestellt werden. Alle folgenden Transformationen können als Matrixmultiplikation dargestellt werden. Es wird davon ausgegangen, dass die Punkte bzw. Vektoren, die transformiert werden sollen, in homogenen Koordinaten[1] der Form (x, y, z, w) mit $w = 0$ für Vektoren und $w = 1$ für Punkte dargestellt werden. Auf diese Weise können die Transformationen sowohl bei Punkten als auch bei Vektoren durch einfache Multiplikationen mit $4x4-$Matrizen durchgeführt werden.

Diese Art von Transformationen wird im Bereich der Anwendungsstufe besprochen, da sie hier für die Positionierung von Objekten innerhalb der Welt oder ihrer Animation verwendet wird. Sie finden darüber hinaus natürlich auch ihre Anwendung in der Geometriestufe.

2.1.1.2 Translation

Translationen sind Positionsverschiebungen eines Punktes im Raum und sind somit nur für Punkte relevant. Durch die Überführung von Punkten und Vektoren in ihre homogenen Koordinaten ist es möglich alle Transformationen, auch wenn sie wie im Beispiel der Translation nur für Punkte gelten, über 4x4-Matrixmultiplikationen darzustellen.

Diese Verschiebung in Richtung des Verschiebungsvektors $\mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_y \end{pmatrix}$ kann als Translationsmatrix

[1] Homogene Koordinaten sind ein vielbenutztes Hilfsmittel in der Computergrafik bei dem Vektoren und Punkten im $\mathbb{R}^3$ um eine Dimension erweitert werden. Diese zusätzliche Komponente ermöglicht affine Abbildungen durch Matrixmultiplikationen abzubildenen, bei denen Vektoren Vektoren und Punkte Punkte bleiben. Bei Projektionen kann diese vierte Komponente einen anderen Wert als 1 oder 2 annehmen. In diesem Fall erhält man den Vektor oder Punkt durch Homogenisierung, indem durch die w-Komponente der homogenen Koordinate dividiert wird. Eine genaue Herleitung und Beschreibung homogener Koordinaten findet sich in [2], S. 14ff.

$$\mathbf{T}(\mathbf{t}) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Die Multiplikation eines Punktes $p = (p_x, p_y, p_z, 1)$ mit der Matrix $\mathbf{T}(\mathbf{t})$ führt zu dem neuen Punkt $p^{'} = (p_x + t_x,\, p_y + t_y,\, p_z + t_z,\, 1)$. Die inverse Operation $\mathbf{T^{-1}}(\mathbf{t})$ ist definiert als $\mathbf{T}(-\mathbf{t})$.

2.1.1.3 Rotation

Eine Rotation $\mathbf{R}(\phi)$ um den Winkel ϕ im Bogenmaß kann in Form einer Rotationsmatrix um eine der drei orthogonalen Achsen des Koordinatensystems beschrieben werden. Somit ergeben sich die drei Rotationsmatrizen:

$$\mathbf{R_x} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R_y} = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R_z} = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Inverse kann auf zwei Arten hergeleitet werden. Einmal als Rotation um den Winkel ϕ in die entgegengesetzte Richtung, also $\mathbf{R^{-1}}(\phi) = \mathbf{R}(-\phi)$, oder durch die Eigenschaft orthogonaler Matrizen, dass die Inverse einer orthogonalen Matrix ihre Transponierte ist, also $\mathbf{R^{-1}}(\phi) = \mathbf{R^T}(\phi)$. Möchte man eine Rotation um eine beliebige Achse durchführen, muss diese zurückgeführt werden auf eine Verkettung von Rotationen um die drei orthogonalen Achsen.

Eine weitere Möglichkeit Rotationen elegant darzustellen sind die sogenannten Quaternionen, die im 19. Jhd. von Sir William Rowan Hamilton als Erweiterung der komplexen Zahlen eingeführt wurden und in einem Artikel von Shoemake im Jahr 1985 [3] für die

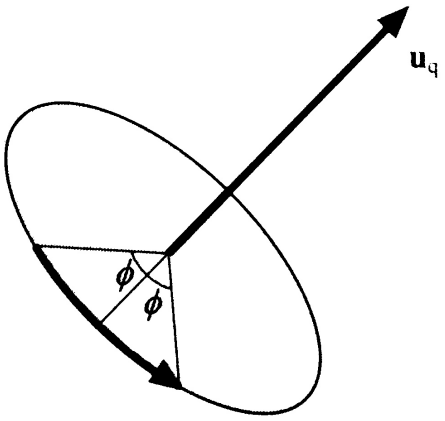

Abbildung 3: Eine Rotation mithilfe eines Einheitsquaternion. (Aus [1], S.47)

Berechnung von Rotationen in der Computergrafik vorgeschlagen wurden. Quaternionen können als Vektoren mit vier Komponenten dargestellt werden, wobei für die Rotationsberechung Einheitsquaternionen (Quaternionen der Länge 1) verwendet werden. An dieser Stelle sollen nicht die mathematischen Grundlagen für das Rechnen mit Quaternionen gelegt werden[2], sondern ein erster Einblick in den Nutzen geschaffen werden.

Die Rotation eines Punktes $p = (p_x, p_y, p_z, p_w)$ um eine beliebige Achse $\mathbf{u}_q = (u_x, u_y, u_z)$, mit $\|\mathbf{u}_q\| = 1$, lässt sich mithilfe eines Einheitsquaternion berechnen. Dafür wird als erstes der Punkt in ein Quaternion transformiert, indem der Punkt komponentenweise in ein Quaternion $\hat{p}$ übertragen wird. Das Einheitsquaternion, das die gewünschte Rotation repräsentiert hat die Form $\hat{q} = (\sin\phi u_q, \cos\phi)$. Die Transformation ist dann definiert als $\mathbf{R}(\hat{p}) = \hat{q}\hat{p}\hat{q}^{-1}$, mit der Konjugierten $\hat{q}^{-1}$ zu $\hat{q}$. In dieser Form rotiert die Transformation den Punkt um den Wert 2ϕ. Abbildung 3 stellt diese Rotation noch einmal grafisch da.

Aus Optimierungssicht soll an dieser Stelle ein weiterer Vorteil der Quaternionen genannt werden. In einiger Hardware ist die Matrixmultiplikation direkt implementiert, so dass es sinnvoll ist, ein Quaternion in Form einer Matrix darzustellen. Diese Matrix ermöglicht es, sobald innerhalb des verwendeten Quaternions die trigonometrischen Funktionen berechnet wurden, auf diese im weiteren Verlauf aller Rechnungen verzichten zu können, was einen großen Vorteil in Bezug auf die Rechengeschwindigkeit hat.

2.1.1.4 Skalierung Als letzte Transformation soll hier noch die Skalierung angesprochen werden. Hierbei wird ein Objekt um die skalaren Faktoren $\mathbf{s} = (s_x, s_y, s_z)$ skaliert. In Matrixform kann dies geschrieben werden als

[2] Herleitung und Beschreibung von Quaterionen finden sich zum Beispiel in [2], S. 439ff.

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Die Inverse ist definiert als $\mathbf{S^{-1}}(\mathbf{s}) = \mathbf{S}(\frac{\mathbf{1}}{\mathbf{s}})$.

2.1.2 Geometriestufe

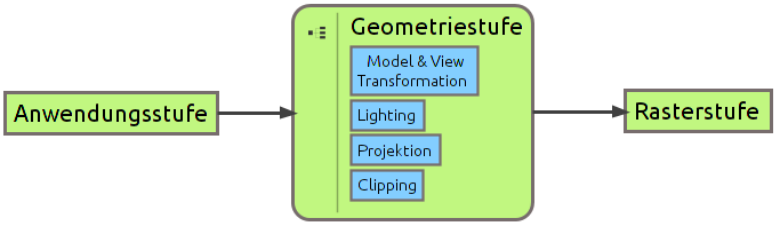

Abbildung 4: Die Geometriestufe der Grafikpipeline im Detail.

Diese Stufe beinhaltet die meisten polygonbasierten Aktivitäten innerhalb der Architektur der Grafikpipeline. Sie kann in weitere Unterstufen aufgeteilt werden, die wahlweise in Software aber auch in Hardware implementiert sein können. Sie sind in Abbildung 4 dargestellt.

2.1.2.1 Model & View Transform

Jedes Objekt existiert in einem eigenen Modellkoordiantenraum. Um das Arbeiten mit diesen Modellen zu vereinfachen werden sie in Bezug zu unterschiedlichen Koordinatenbasen dargestellt. Jedem Modell können dabei mehrere Transformationen zugewiesen werden. Auf diese Weise ist es möglich einem Geometrieobjekt beliebig viele Instanzen zuzuweisen, die an unterschiedlichen Positionen platziert werden können, die der Instanz einen Ort in Weltkoordinaten definieren.

Eine virtuelle Kamera wird als Abstraktion der sich nun anschließenden Stufe dazu verwendet, die durch die Kamera sichtbaren Objekte in einen normalisierten Raum zu transformieren, der es erlaubt Projektionen und Clipping-Algorithmen effizient durchführen zu können. Hierfür wird die Kamera in den Ursprung des Weltkoordinatensystems bewegt und so orientiert, dass sie in negative z-Richtung blickt, die y-Achse nach oben ausgerichtet ist und die x-Achse nach rechts zeigt. Die Transformation führt zu einer Darstellung des Raums im Kameraraum (*camera space* oder *eye space*) und wird mit Matrixmultiplikationen in homogenen Koordinaten realisiert.

2.1.2.2 Lighting

Um den Realismus der durch die Kamera sichtbaren Objekte zu erhöhen können Beleuchtungsalgorithmen auf die sichtbare Szene angewandt werden. Dabei wird auf jedes Objekt, das von Beleuchtungsquellen beeinflusst werden soll, eine Beleuchtungsfunktion angewandt, die die Farbe und Helligkeit in der Szenerie bestimmt. Die Beleuchtungsfunktion nähert dabei die Interaktion von Photonen mit ihrer Umwelt an, um einen realistischen Beleuchtungseindruck zu verschaffen.

In vielen Echtzeitsystemen ist diese Berechnung aber zu zeitaufwändig und es müssen Kompromisse eingegangen werden. So, dass die Beleuchtung zwischen den Vertizes eines Oberflächendreiecks des Objekts interpoliert und man dadurch den Aufwand der Berechnungen reduzieren kann.

2.1.2.3 Projektion

Die Projektion ist dafür verantwortlich, dass bei der Computergrafik etwas auf einem Ausgabegerät ausgegeben werden kann. Grundlegend muss, damit etwas darstellbar ist, die sichtbare Szene auf einer Projektionsebene abgebildet werden. Abhängig von der Art der Projektion, die verwendet wird, werden unterschiedliche Eigenschaften der Objekte verändert oder nicht. So verändert die orthographische Projektion die w-Komponente der homogenen Koordinaten nicht. Anders die perspektivische Projektion, hier bleiben parallele Geraden nicht unbedingt parallel und im Extremfall können Geraden zu Punkten konvergieren.

In dieser Stufe werden die Koordinaten der Objekte, die durch die Kamera sichtbar sind, in ein Einheitssichtvolumen transformiert, das durch die Extrempunkte $(-1, -1, -1)$ und $(1, 1, 1)$ aufgespannt wird. Diese Normalisierung hilft im weiteren Verlauf bei den Clippingberechnungen, die sich an die Projektion anschließen.

2.1.2.4 Clipping

Beim Clipping werden die alle Primitive eines Objekts entfernt, die sich weder ganz noch teilweise im Sichtvolumen der Kamera befinden. So sollten Primitive, die teilweise außerhalb des Sichtvolumens liegen, so in der Cippingstufe verändert werden, dass die außerhalb liegenden Vertizes durch Vertizes ersetzt werden, die direkt an den Grenzen des Einheitswürfels des Sichtvolumens liegen. Ein wichtiger Vertreter der Clipping-Algorithmen ist der Cohen-Sutherland-Algorithmus. Er kann auf zweidimensionale Strecken angewandt werden, die durch ein Window teilweise außerhalb des Sichtbereichs befinden.

Der Algorithmus untersucht die drei möglichen Fälle, die bei einer Strecke auftreten können.

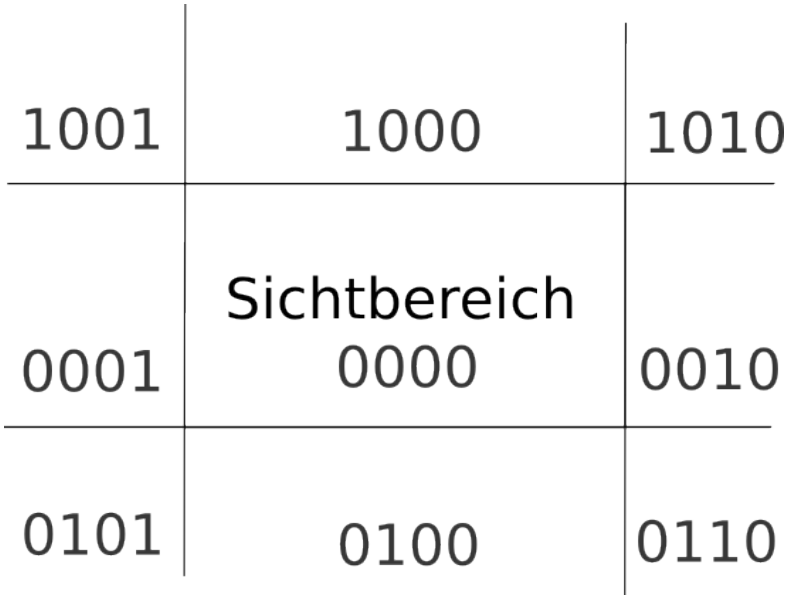

Abbildung 5: Die acht an den Sichtbereich angrenzenden Bereiche und ihre Binärcodierung.

1. Die beiden Endpunkte der Strecke liegen innerhalb des Sichtbereichs. In diesem Fall wird die Strecke vollständig gerastert und angezeigt.
2. Die beiden Endpunkt liegen beide außerhalb des Sichtbereichs und schneiden diesen nicht. In diesem Fall wird die Strecke nicht gerastert.
3. Einer der beiden Endpunkt liegt außerhalb des Sichtbereichs oder die Strecke durchkreuzt den Sichtbereich. In dem Fall muss an den Fenstergrenzen geclippt und nur der sichtbare Bereich der Strecke gerastert werden.

Durch das Verlängern der vier Fensterseiten ins Unendliche erhält man neun Bereiche, wie sie in Abbildung 5 gezeigt sind. Jedem Bereich und jedem Endpunkt der Strecke wird ein 4-Bit-Code zugewiesen, der die Lage relativ zum Sichtfenster beschreibt. Nachdem jedem Endpunkt ein Bereichscode zugewiesen wurde, vergleicht der Algorithmus die Codes der beiden Punkte. Zuerst wird geprüft, ob die Strecke überhaupt sichtbar ist. Dies geschieht durch eine logische Und-Verknüpfung der beiden Code. Ist das Ergebnis der Verknüpfung der Code 0000, so geht (mindestens) ein Teil der Strecke durch den Sichtbereich, ist das Verknüpfungsergebnis ein anderes, so kann die Strecke verworfen werden, da sie den Sichtbereich nicht schneidet. Eine darauf folgende Oder-Verknüpfung der beiden Strecken wird dazu verwendet, zu entscheiden, ob und welcher Teil der Strecke geclippt werden muss.

Eine Erweiterung auf 2D-Polygone findet sich im Sutherland-Hodgman-Algorithmus (vgl. hierzu auch [2], S. 49ff).

2.1.2.5 Window-Viewport-Transformation

Ein Window ist ein Auswahlbereich in der Projektionsebene, ein Viewport ist ein Auswahlbereich in Bildschirmkoordinaten auf dem Ausgabegerät. Die Window-Viewport-Transformation bildet entsprechend ein beliebiges Window auf einem beliebigen Viewport ab. Es handelt sich dabei um eine Aneinanderreihung von einer Translation, einer Skalierung und einer weiteren Translation.

Die erste Translation verschiebt das Window in den Koordinatenursprung. Dort wird es auf die Größe des Viewport skaliert. Nach erfolgter Skalierung kann der Viewport an die Ursprungsposition zurückverschoben werden.

2.1.3 Rasterstufe

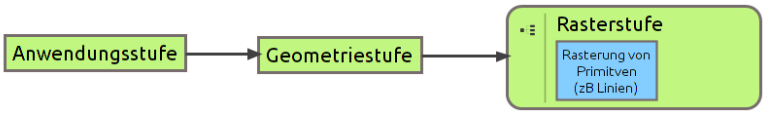

Abbildung 6: Die Rasterstufe der Grafikpipeline im Detail.

In dieser letzten großen Stufe werden die Koordinaten der Objekte im Einheitssichtvolumen genommen und gerastert, damit sie auf einem punktbasierten Ausgabegerät wir einem Bildschirm ausgegeben werden können. Als Beispiel für diesen Prozess soll in Abschnitt 2.1.3.1 die Rasterung von Linien besprochen werden. Ein wichtiger Vertreter dieser Rasteralgorithmen ist der Mid-Point-Algorithmus oder auch Bresenham-Algorthmus.

2.1.3.1 Line Drawing

Eine Linie kann mathematisch auf unterschiedliche Arten dargestellt werden. Zu nennen sind hier die Punkt-Steigungsform $y = m \cdot x + b$, mit der Steigung $m = \frac{\Delta y}{\Delta x}$ und dem y-Achsenabschnitt b oder die parametrische Geradengleichung aus der Vektorrechnung $\vec{x} = \vec{o} + t \cdot \vec{r}$, mit dem Ortsvektor $\vec{o}$, dem Richtungsvektor $\vec{r}$ und dem Skalar t.

In der Computergrafik müssen die kontinuierlichen Geraden zwischen einem Start- und einem Endpunkt gerastert werden, damit sie auf einem pixelbasierten Ausgabegerät dargestellt werden können. Ein naiver Algorithmus wäre die Verwendung der Steigung zwischen den beiden Punkten, die die Linie definieren. Dieses Vorgehen, wie auch die Verwendung der parametrischen Geradengleichung zur Darstellung von Linien ist aber aus mehreren Gründen nicht zu empfehlen.

1. Durch Rundungsfehler und Schrittweiten > 1 während der Rechnungen kann es zu Darstellungsfehlern kommen.

2. Die Punkt-Steigungsform besitzt Singularitäten, wenn Δx gegen 0 strebt.

3. Multiplikationen sind für einen Computer „teurer" auszuführen als Additionen.

Ein Algorithmus, der diese Probleme adressiert und hier vorgestellt werden soll, ist der Mid-Point Algorithmus, der auf den Bresenham-Algorithmus zurückgeht. Ziel dieses Verfahrens ist ein effizientes Verfahren zur Rasterung von Linien bereitzustellen, das auf alle Multiplikationen während der Rasterung der Linie verzichten kann.

Der Mid-Point-Algorithmus beginnt bei einem Startpunkt und geht über alle x-Werte in Richtung des Endpunkts der zu zeichnenden Geraden. Für jeden Rechtsschritt berechnet der Algorithmus über einen Entscheidungswert, wie weit sich ein weiterer Rechtsschritt von der Idealgeraden entfernt. Erreicht die Entscheidungsgröße einen Grenzwert, so geht Algorithmus einen Diagonalschritt und nähert sich somit der Geraden wieder an.

Auf diese Weise nähert der Mid-Point-Algorithmus die über Start- und Endpunkt definierte Gerade ohne eine Multiplikation in der Zeichenroutine an.

3 Anforderungen und Implementierung

Dieses Kapitel soll die wesentlichen Entscheidungen und Anforderungen präsentieren, die Grundlage der Implementierung des Frameworks sind. Hier steht der Einsatz innerhalb der Veranstaltung *Computergrafik und Animation* im Vordergrund und die Erfahrungen, die während der letzten Jahre gesammelt wurden, sollten sich in der Gestaltung der Anwendung entsprechend widerspiegeln.

3.1 Anforderungen

Im Rahmen der Veranstaltung sollen die Studenten die Grundlagen der Computergrafik kennenlernen und diese in praktischen Aufgabenstellungen anwenden und vertiefen. Die Neugestaltung des Frameworks soll die wesentlichen Gedanken moderner Grafiksysteme präsentieren, ohne deren Komplexität aufzuweisen, zudem sollte es die Möglichkeit der Erweiterbarkeit bieten, um zusätzliche Anforderungen in der Zukunft mit abdecken zu können.

Eine wesentliche Kernkompetenz, die mit dem Framework gefördert werden soll, ist die Abstraktion und Modellierung von Zusammenhängen in einer objektorientierten Art und Weise, um so auf das Gelernte der ersten beiden Semester aktiv zurückzugreifen. Zudem werden Grundgedanken bekannter und etablierter Frameworks (JMonkeyEngine oder OpenSceneGraph) aufgegriffen und in vereinfachter Form zur Verfügung gestellt.

Tabelle 1 fasst die wichtigsten Anforderungen aus den Erfahrungen der letzten Jahre, der Evaluation der API anderer Open-Source Projekte und Gesprächen mit Herrn Prof. Dr. Horst Stenzel zusammen.

Anforderung	Begründung	Priorität
Objektorientierte API	Anlehnung an etablierte Frameworks und Übungsmöglichkeit für Studenten ihre Kenntnisse in der Objektorientierung zu vertiefen.	hoch
Modularisierung in Anlehnung an Grafikpipeline	Bietet Vertiefungsmöglichkeit der in der Vorlesung gelernten Konzepte und greift auf eine vertraute Abstraktion zurück.	hoch
Flache Lernkurve und einfacher Einstieg	Einstieg sollte möglichst einfach und überschaubar sein. Gerade in Bezug auf die kurze Zeit zwischen den einzelnen Praktikumsterminen ist eine lange Einführung nicht praktikabel.	hoch
Implementierung sollte unabhängig von externen Abhängigkeiten sein	Es sollte weitesgehend auf externe Abhängigkeiten und Bibliotheken verzichtet werden. Das Rad soll nicht immer neu erfunden werden, soll durch entsprechende Implementierungen und Reduktion auf das wesentliche den Verständnisprozess fördern.	mittel

Tabelle 1: Anforderungen an das neue CGA-Framework.

3.2 Implementierung

Wie in den Anforderungen auch gefordert ist ein wesentlicher Aspekt der Implementierung die Anlehnung an die bereits vorgestellte Grafikpipeline. Abbildung 7 zeigt das Modell der Pipeline, wie sie im Framework abgebildet wird und welche Komponenten eine wesentliche Rolle spielen. In den folgenden Abschnitten soll auf diese Komponenten genauer eingegangen werden. Kapitel 4 präsentiert anhand eines Beispiels den aktuellen Stand des Frameworks und gibt eine Einführung in seine Benutzung.

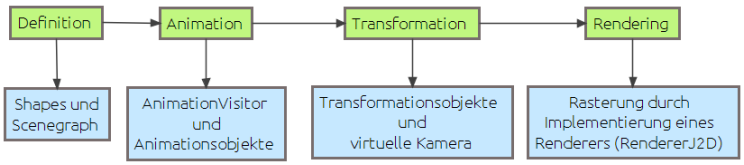

Abbildung 7: Die Grafikpipeline, wie sie im Framework abgebildet ist. Die grün unterlegten Blöcke sind dabei die Phasen der Pipeline, die durchlaufen werden, die blau hinterlegten geben die in dieser Stufe beteiligten Komponenten des Frameworks an.

Dieser Ablauf wird über die Klasse Framework gesteuert. Sie kontrolliert den Szenengraph und seine Aktualisierung sowie die Visitors, die Aktionen auf dem Szenegraphen ausführen. Diese Klasse muss vom Nutzer erweitert werden.

3.2.1 Definition

In dieser Phase werden die Geometrieobjekte der Anwendung implementiert und dem Szenengraphen hinzugefügt. Der Szenengraph ist als einfacher Baum implementiert, d.h. es werden keine Optimierungen vorgenommen. Der Szenengraph dient als Datenstruktur zur Speicherung der Szene. In ihr werden die Geometrieobjekte, Animationen und Transformationen gehalten. Angelehnt an die Arbeitsweise des OpenSceneGraphs werden Aktionen auf der Datenstruktur über das Visitor Pattern implementiert. Dafür besitzt jeder Knoten eine *accept*-Methode, über die ein Visitor Zugriff auf den Knoten bekommt.

Die für den Nutzer wichtigen Methoden des Szenengraphs sind

- *void add(Node node)*
  Fügt dem Wurzelknoten des Szenengraphs einen neuen Kindsknoten hinzu.
- *void add(Shape shape)*
  Fügt dem Wurzelknoten des Szenengraphs ein neues Geometrieobjekt hinzu.

Jedem Knoten kann ein Bezeichner mitgegeben werden. Über diese Bezeichner kann der Knoten wiedergefunden und verändert werden. Daher gibt es die oben beschriebenen Methoden auch in einer Variante, in der ein Objekt (Knoten oder Geometrie) einem bezeichneten Knoten und nicht dem Wurzelknoten hinzugefügt werden kann.

Alle Geometrieobjekte, die innerhalb des Frameworks verwendet werden, erben von der abstrakten Klasse Shape. Sie bietet die grundlegenden Funktionen eines Geometrieobjekts. Eine Klasse, die die Klasse Shape erweitert muss die abstrakte Methode *render(Matrix*

transformation, Camera camera, Renderer renderer) implementieren. Sie wird in der letzten Stufe der Pipeline verwendet, um das Objekt zu rastern. Der Methode werden von außen drei Parameter mitgegeben, die zur korrekten Darstellung benötigt werden.

1. Eine Transformationsmatrix, die eine mögliche Transformation durch übergeordnete Transformationsgruppen innerhalb des Szenengraphs an das Objekt darstellt. Existiert keine Transformationsgruppe, die sich auf dieses Objekt auswirkt, so ist sie die 4x4 Einheitsmatrix.

2. Eine virtuelle Kamera, die die Projektion und das Clipping durchführt.

3. Eine Renderer Implementierung, die das gerasterte Objekt auf einem Ausgabegerät darstellt.

Zusätzlich ist es möglich über die Methode *setAnimation* einem Geometrieobjekt eine Animation zuzuordnen, die die Eigenschaften beeinflussen kann. Dies kann entweder zur Laufzeit oder vorher geschehen. Auf die Details und unterschiedlichen Möglichkeiten die Szene zu animieren geht der folgenden Abschnitt ein.

3.2.2 Animation

Geometrieobjekte lassen sich über Animationsobjekte verändern. Ihr Code wird durch einen *AnimationVisitor* auf dem Geometrieobjekt ausgeführt. Da ein Animationsobjekt in der Regel nur für eine spezielle Klasse von Shapes funktioniert, bietet es sich an, diese als anonyme Klasse in der *setAnimation(Animation animation)* Methode zu setzen. Zu beachten ist dabei, dass diese Art von Animationen für ein Objekt gelten. Dies ermöglicht die individuelle Animation einzelner Komponenten.

Eine zweite Möglichkeit eine gewisse Dynamik in eine Szene zu bekommen ist der Einsatz von animierten Transformationsgruppen, die in Abschnitt 3.2.3 besprochen werden. Diese wirken sich nicht auf einzelne Geometrieteile aus, sondern transformieren die Position oder Lage im Raum des gesamten Shapes.

Damit man innerhalb der Animationsklasse Zugriff auf die Datenfelder der Unterklasse von Shape hat, werden Generics in der Deklaration der abstrakten Klasse Animation verwendet. Die Signatur der Klasse sieht folgendermaßen aus: `abstract class Animation<T extends Shape>`. Dadurch werden alle Unterklassen von Shape innerhalb des Templates zugelassen.

Zur Laufzeit besucht bei jeder Aktualisierung des Szenengraphs durch das das Framework eine Instanz der Klasse *AnimationVisitor* jeden Knoten und ruft für jede Animation, die in einem Shape gefunden wird, die dazugehörige *animate*-Methode auf.

3.2.3 Transformation

Transformationen wie sie in Abschnitt 2.1.1.1 angesprochen wurden, implementieren im Framework das Interface *Transformation* und werden einem Knoten innerhalb des Szenengraphs zugewiesen. Sie wirken sich auf alle Shapes des Knotens und aller Kindsknoten. Auf diese Weise ist es möglich, Transformationen zu schachteln und so die Position und Lage ganzer Gruppen von Shapes auf einmal zu verändern.

Wie schon in 3.2.2 angesprochen, ist es möglich diese Transformationen zu animieren, in dem der zu der Transformation gehörige Parameter (zum Beispiel das ϕ der Rotationsmatrizen) zwischen einem Start- und einen Endwert interpoliert wird. Für solche Interpolationen gibt es verschiedene Implementierungen des Interface *Interpolation*, zum Beispiel *LinearInterpolation* oder *EaseOutInterpolation*[3], die an die Transformationsobjekte übergeben werden können.

Neue Transformationen und Interpolationen lassen sich bei Bedarf leicht selbst schreiben, sollte eine spezielle Art für eine bestimmte Anwendung fehlen. Dies ist in den Code Listings 1 und 2 für eine Rotation und eine neue Interpolationsart dargestellt.

[3] Diese Interpolation beginnt mit großen Schrittweiten zwischen zwei Interpolationsschritten zu Beginn und endet mit kleinen Schrittweiten am Ende. Es stellt eine Art Abbremsung dar.

Algorithmus 1 Implementierung einer Rotationsmatrix.

```
/**
 * Eine Rotation entlang der x-Achse.
 */
public class RotationX implements Transformation {
  /**
   * der aktuelle Winkel der Rotation.
     */
    private double phi;

  /**
   * die Transformationsmatrix, die bei den Transformationsrechnungen verwendet wird.
     */
    private Matrix transformMatrix;

    /**
     * Erstellt ein Rotationsobjekt, das eine Rotation, um den Winkel phi beschreibt.
     * @param phi der Winkel phi der Rotation
     */
    public RotationX(double phi) {
    this.phi = phi;
    updateMatrix();
    }

    /**
     * Aktualisiert die Transformationsmatrix.
     */
    private void updateMatrix() {
    double[][] values = {
        {1, 0, 0, 0},
        {0, Math.cos(phi), -Math.sin(phi), 0},
        {0, Math.sin(phi), Math.cos(phi), 0},
        {0, 0, 0, 1}
    };
    transformMatrix = Matrix.constructWithCopy(values);
    }

    @Override
    public void update() {
    if (interpolationPhi != null) {
        phi = interpolationPhi.nextValue();
        updateMatrix();
    }
    }
}
```

Algorithmus 2 Implementierung einer neuen Interpolationsart. Hier wird das *EaseIn* implementiert. Also eine sich „beschleunigende" Interpolation.

```
/**
 * Eine Interpolation mit kleinen Schrittweiten zu Beginn, die zum Ende hin größer
 * werden. Auf diese Weise ist es zum Beispiel möglich Beschleunigungen zu
 * animieren.
 *
 * @author Robert Giacinto
 */
public class EaseInInterpolation extends Interpolation {
    /**
     * Konstruktor der Interpolation.
     * @param start der Startwert der Interpolation
     * @param end der Endwert der Interpolation
     * @param stepCount die Anzahl der Schritte zwischen Start- und Endwert
     * @param cyclic true, wenn Interpolation zyklisch laufen soll
     */
    public EaseInInterpolation(double start, double end, int stepCount, boolean cyclic) {
        super(start, end, stepCount, cyclic);
    }

    @Override
    public double nextValue() {
        if (stepCounter++ < stepCount) {
            double x = (stepCounter / stepCount);
            x = Math.pow(x, 2);
            return min + max * x;
        } else if (cyclic) {
            stepCounter = 0;
        }
        return max;
    }
}
```

Neben der Transformation durch Transformationsobjekte werden die unterschiedlichen Shapeobjekte durch die virtuelle Kamera transformiert. Wie schon in Kapitel 2 angesprochen realisiert die virtuelle Kamera eine Koordinatentransformation von Weltkoordinaten in das normierte Clipping Einheitsvolumen. Die Implementierung der virtuellen Kamera im Framework ermöglicht orthographische und perspektivische Projektionen. Es ist auch möglich eigene Implementierungen einzubinden. Dafür muss die neue Kamera das Interface *Camera* implementieren. Kameras müssen folgende drei Methoden implementieren:

1. *Matrix getProjection()*
   Diese Methode gibt die implementierte Projektionsmatrix zurück. Dies ist sinnvoll, wenn andere Module direkt mit der Projektionsmatrix umgehen müssen, um eine Funktionalität realisieren zu können. In der Regel sind aber die beiden folgenden Methoden für den Nutzer relevanter.

2. *CVPoint getClippingSpaceCoordinates(Vector3d vector3d)*
   Diese Methode projiziert den übergebenen dreidimensionalen Vektor auf einen Punkt im Clipping Space. Dies kann nützlich sein, wenn eigene Clippingalgorithmen implementiert werden sollen.

3. *Pixel getImageSpaceCoordinates(Vector3d vector3d)*
   Diese Methode projiziert den Vektor in den Bildraum und gibt die Position in Bild-

schirmkoordinaten als Pixel zurück. Diese Methode wird in der Regel in den render-Methoden der Shapes eingesetzt.

3.2.4 Rendering

In der Renderingstufe wird jeder Knoten von einer Instanz des *RenderVisitors* besucht. Dieser ruft in jedem Shapeobjekt, das er beim Traversieren des Szenengraphs findet, die Rendermethode auf und weist die Objekte so an, sich selbst auf dem mit dem Renderer verbundenen Ausgabegerät bzw. Ausgabemedium auszugeben.

Das Framework bietet hier derzeit nur eine Implementierung an, nämlich den *RendererJ2D*, der Java2D für die Ausgabe verwendet. Es sind aber auch andere Implementierungen denkbar. Zum Beispiel eine hardwarebeschleunigte mit OpenGL oder eine, die die Renderausgabe in eine Bilddatei umleitet. Alles, was dazu gemacht werden muss, ist die abstrakte Klasse *Renderer* erweitern, die die Unterklassen folgende Methoden implementieren lässt:

- *void putPixel(Pixel pixel)*
  Diese Methode gibt den Pixel in einer Standardfarbe aus.
- *void putPixel(Pixel pixel, Color color)*
  Diese Methode gibt den Pixel in der Farbe *color* aus.
- *void show()*
  Diese Methode weist den Renderer an, die Ausgabe anzuzeigen / zu speichern.

Code Listing 3 zeigt die Implementierung für eine Ausgabe auf Basis von Java2D. Neben den durch die abstrakte Klasse geforderten Methoden werden hier noch weitere Methoden zur Verfügung gestellt, die spezifisch für die verwendete Plattform sind.

Hiermit ist das Ende der Pipeline erreicht und das Ergebnis wird ausgegeben. Neben Erweiterungen der existierenden Stufen können auch neue einfach hinzugefügt werden. Durch das implementierte Visitor Pattern muss dazu ein zu der Stufe passender Visitor erzeugt werden, die die entsprechende Funktionalität bereitstellt.

Algorithmus 3 Java2D Implementierung eines Renderers, der die Ausgabe des Frameworks in einem JFrame darstellt.

```
public class RendererJ2d extends Renderer {

    private JFrame frame;
    private BufferStrategy bs;
    private int width;
    private int height;
    private int offsetX;
    private int offsetY;

    /**
     * Erzeugt eine Java2D Ausgabe in einem JFrame der Größe width x height.
     * @param width die Breite des Fensters
     * @param height die Höhe des Fensters
     */
    public RendererJ2d(int width, int height) {
        super(width, height);
        this.width = width;
        this.height = height;
        this.offsetX = width >> 1;
        this.offsetY = height >> 1;

        frame = new JFrame("Java2D Renderer");
        frame.setSize(width, height);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setBackground(java.awt.Color.WHITE);
        frame.setVisible(true);

        frame.createBufferStrategy(2);
        bs = frame.getBufferStrategy();
    }

    @Override
    public void putPixel(Pixel pixel) {
        Graphics g = bs.getDrawGraphics();
        g.setColor(java.awt.Color.BLACK);
        g.fillRect(offsetX + (int) pixel.x, (int) (-pixel.y) + offsetY, 1, 1);
        g.dispose();
    }

    @Override
    public void putPixel(Pixel pixel, cga.framework.renderer.Color color) {
        Graphics g = bs.getDrawGraphics();
        g.setColor(color.color);
        g.fillRect(offsetX + (int) pixel.x, (int) (-pixel.y) + offsetY, 1, 1);
        g.dispose();
    }

    @Override
    public void show() {
        bs.show();
        bs.getDrawGraphics().clearRect(0, 0, frame.getWidth(), frame.getHeight());
    }

    /**
     * Fügt dem JFrame einen neuen MouseListener hinzu.
     */
    public void addMouseListener(MouseAdapter mouseAdapter) {
        frame.addMouseListener(mouseAdapter);
        frame.addMouseMotionListener(mouseAdapter);
        frame.addMouseWheelListener(mouseAdapter);
    }

    /**
     * Fügt dem JFrame einen neuen KeyEventListener hinzu.
     */
    public void addKeyListener(KeyAdapter keyAdapter) {
        frame.addKeyListener(keyAdapter);
    }
}
```

4 Aktueller Stand der Arbeit

In diesem Kapitel soll der aktuelle Stand des Frameworks und die ersten Schritte bei seiner Verwendung demonstriert werden. Im praktischen Teil dieses Abschnitts wird eine kleine Anwendung beispielhaft implementiert werden. Es handelt sich dabei um ein Dreieck, das um eine beliebige Achse rotiert. Anhand dieses Beispiels werden die wichtigsten Aspekte noch einmal aus praktischer Sicht dargestellt. Folgende Fragen sollen in diesem Bereich beantwortet werden:

- Wie sieht ein lauffähiges Programm aus, das das Framework benutzt?
- Wie wird ein eigenes Shape erstellt?
- Wie werden Shapes dem Szenengraph hinzugefügt?
- Wie fügt man einem Shape eine Animation hinzu?

Abbildung 8 zeigt einen Screenshot der Ausgabe des Programms, das im Laufe dieses Kapitels entwickelt werden soll und anhand dessen die Arbeitsweise und Funktionskomponenten vorgestellt werden.

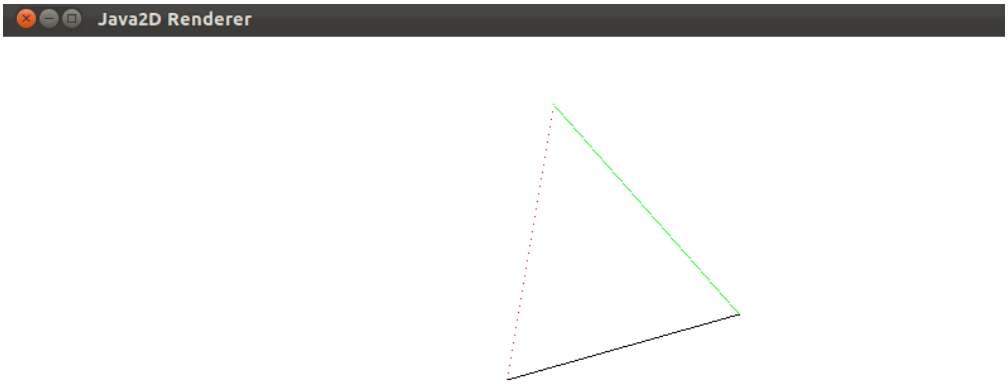

Abbildung 8: Java2D Ausgabe des Programms, das in Kapitel 4 entwickelt wird.

4.1 Minimalprogramm

Das Minimalprogramm zur Verwendung des Frameworks ist in Listing 4 gezeigt.

Um das Framework einsetzen zu können erstellt man eine Unterklasse der Klasse Framework, die sich um die meisten Aufgaben zur Laufzeit kümmert. Die Unterklasse von Framework muss die abstrakte Methode *initGraph* implementieren. In dieser Methode werden alle Aktionen durchgeführt, die den Szenengraph vor der Ausführung des Programms mit Inhalt füllen. Im einfachsten Fall bleibt diese Methode leer.

Algorithmus 4 Eine Minimalimplementierung.

```
import cga.Framework;

/**
 * Dieses Beispiel zeigt die Minimalkonfiguration des Frameworks.
 */
public class Minimal extends Framework {

    /**
     * Neues Framework, das eine Java2D Ausgabe der Größe width x height hat.
     * @param width die Breite des Fensters
     * @param height die Höhe des Fensters
     */
    public Basics(int width, int height) {
        super(width, height);
    }

    @Override
    public void initGraph() {
    }

    public static void main(String[] args) {
        Minimal example = new Minimal(800, 600);
        example.start();
    }
}
```

4.2 Eigene Objekte mit der Klasse Shape

Eigene Objekte, die in der Ausgabe dargestellt werden, müssen die abstrakte Klasse *Shape* erweitern. Für das Beispielprogramm benötigt man ein Dreieck, dem den einzelnen Seiten unterschiedliche Farben zugewiesen werden können. Für diese Implementierung greifen wir auf eine bereits existierende Klasse des Frameworks zurück. Die Klasse *Line2d* kann zum Zeichnen von zweidimensionalen Linien verwendet werden. Der Konstruktor nimmt die Start- und Endkoordinaten der Linie entgegen.

Algorithmus 5 Listing der Klasse Triangle.

```
import cga.framework.camera.Camera;
import cga.framework.math.Matrix;
import cga.framework.renderer.Color;
import cga.framework.renderer.Renderer;
import cga.framework.shape.Line2d;
import cga.framework.shape.Point2d;
import cga.framework.shape.Shape;

/**
 * Ein Dreieck, dessen Seiten unterschiedliche Farben haben.
 */
public class Triangle extends Shape {

    private Line2d l1, l2, l3;

    /**
     * Erzeugt ein neues Dreieck mit den Eckpunkten p1, p2 und p3.
     * @param p1 Eckpunkt p1
     * @param p2 Eckpunkt p2
     * @param p3 Eckpunkt p3
     */
    public Triangle(Point2d p1, Point2d p2, Point2d p3) {
        l1 = new Line2d(p1.x, p1.y, p2.x, p2.y);
        l2 = new Line2d(p1.x, p1.y, p3.x, p3.y);
        l3 = new Line2d(p3.x, p3.y, p2.x, p2.y);

        l1.color = Color.BLACK;
        l2.color = Color.RED;
        l3.color = Color.GREEN;
    }

    @Override
    public void render(Matrix transformation, Camera camera, Renderer renderer) {
        l1.render(transformation, camera, renderer);
        l2.render(transformation, camera, renderer);
        l3.render(transformation, camera, renderer);
    }
}
```

Das Dreieck wird über den Konstruktor der Klasse *Triangle* mithilfe der drei Eckpunkte des Dreiecks definiert. Die Punkte können mit der Klasse *Point2d* dargestellt werden. Es ist aber genauso gut möglich, die einzelnen Werte in Form von Zahlenwerten zu übergeben. Die Klasse Triangle speichert die drei Kanten des Dreiecks als $l_1 := \overline{P_1P_2}$, $l_2 := \overline{P_2P_3}$, $l_3 := \overline{P_1P_3}$ mit den Eckpunkten P_1, P_2 und P_3. Die einzelnen Linienobjekte können dann in der Rendermethode einzeln für die Ausgabe gerastert werden. Dafür wird die entsprechende Methode der Objekte aufgerufen. Der Code für die Klasse *Triangle* findet sich in Listing 5.

In der Main-Klasse des Beispiels kann das Dreieck nun dem Szenengraph hinzugefügt werden. Dafür verwendet man die *add(Shape shape)* Methode, die das Framework zur Verfügung stellt.

In der Methode *initGraph* sollte die Zeile `add(new Triangle(new Point2d(0,0), new Point2d(200,0), new Point2d(100,100));` hinzugefügt werden. Wird das Programm jetzt ausgeführt sollte man das Dreieck in der Ausgabe sehen.

4.3 Animierte Rotation

Im letzten Teil dieses kurzen Einführungsbeispiels soll gezeigt werden, wie Objekte durch die Verwendung von Interpolatoren und Transformationsgruppen animiert werden können.

Um das Dreieck aus 4.2 animiert um eine beliebige Achse rotieren zu lassen, benötigt man einen Knoten, dem eine animierte Transformation zugewiesen wird. Diesem Knoten wird dann das Dreieck hinzugefügt. Die Klassen, die dafür benötigt werden sind:

- *Node*
  Es wird eine neue Unterklasse von Node erzeugt, die *RotatingTriangle* heißt und ein Dreieck als Parameter im Konstruktor entgegennimmt.
- *EaseInInterpolation*
  Dieser Interpolator beschreibt eine Beschleunigung. Das Dreieck beginnt mit einer langsamen Rotationsbewegung und beschleunigt dabei.
- *RotationZ*
  Eine Rotation um die z-Achse des Weltkoordinatensystems.

Algorithmus 6 Die Klasse *RotatingTriangle*.

```
import cga.framework.animation.interpolation.EaseInInterpolation;
import cga.framework.scenegraph.Node;
import cga.framework.scenegraph.transform.RotationZ;

/**
 * Knoten, der ein Dreieck um die z-Achse rotieren lässt.
 */
public class RotatingTriangleNode extends Node {

    public RotatingTriangleNode(Triangle triangle) {
        super("Rotating Triangle");

        RotationZ rotationZ = new RotationZ();
        rotationZ.setInterpolationPhi(new EaseInInterpolation(0, 6 * Math.PI, 1000, true));
        setTransformation(rotationZ);

        addShape(triangle);
    }
}
```

In Listing 6 sind die wenigen Schritte aufgezeigt, die man für eine angepasste Nodeklasse benötigt, um eine animierte Rotation zu implementieren. Drei Schritte müssen dabei durchgeführt werden:

1. Die Klasse *Node* erweitern und innerhalb des Konstruktors den Konstruktor der Oberklasse aufrufen, um dem Knoten einen Namen zu geben. Würde man mehr als eine Instanz von dieser Klasse benötigen wäre es sinnvoll den Namen vom Nutzer über den Konstruktor selbst bestimmen zu lassen.
2. Es wird eine Instanz einer Rotation um die z-Achse als Transformation dieses Knotens gesetzt. Die Interpolation, die für die Animation der Rotation sorgt, wird über die Methode *setInterpolationPhi* der Klasse *RotationZ* festgelegt.

3. Am Ende des Konstruktors wird das Dreieck dem Knoten als Shape hinzugefügt.

In der Mainklasse muss jetzt nur noch der Aufruf in der *initGraph*-Methode angepasst werden. Die Zeile sollte so verändert werden, dass das Framework einen Knoten vom Typ *RotatingTriangleNode* dem Szenengraph hinzufügt:

```
add(new RotatingTriangleNode(new Triangle(new Point2d(0,0), new Point2d(200,0), new Point2d(100,100))));
```

Wird das Programm ausgeführt, rotiert das Dreieck um die z-Achse und beschleunigt dabei, bis es nach drei Umdrehungen am Ende der Interpolation angekommen ist, das Dreieck kurz stoppt und mit der Rotationsbewegung von neuem beginnt.

4.4 Allgemeiner Aufbau

Für den Nutzung des Frameworks sind nach außen nur wenige Klasse nötig. Dies vereinfacht den Einstieg und macht ein leichtes Erweitern der Klassen möglich. Für einfache Aufgaben reicht die Grundkonfiguration des Frameworks aus, so dass man mithilfe der Klassen (bzw. Interfaces) *Shape, Interpolation, Transformation* und *Animation* schon viele Problemstellungen ansprechen kann. Im Hintergrund sind noch einige Klassen mehr beteiligt, wie man im UML-Klassendiagramm des Beispiels aus den letzten Abschnitten in Abbildung 9 erkennen kann.

Die Klasse Framework initialisiert die wichtigsten Klassen (*UpdateVisitor, RenderVisitor, Camera, SceneGraph*) bevor der Code der Unterklasse ausgeführt wird. So kann sichergestellt werden, dass die wesentlichen Funktionen und Objekte durch das Framework bereitgestellt werden. Gleichzeitig bietet diese Klasse einen zentralen Einstieg in das System, um sich einen eigenen Überblick für individuelle Erweiterungen zu verschaffen.

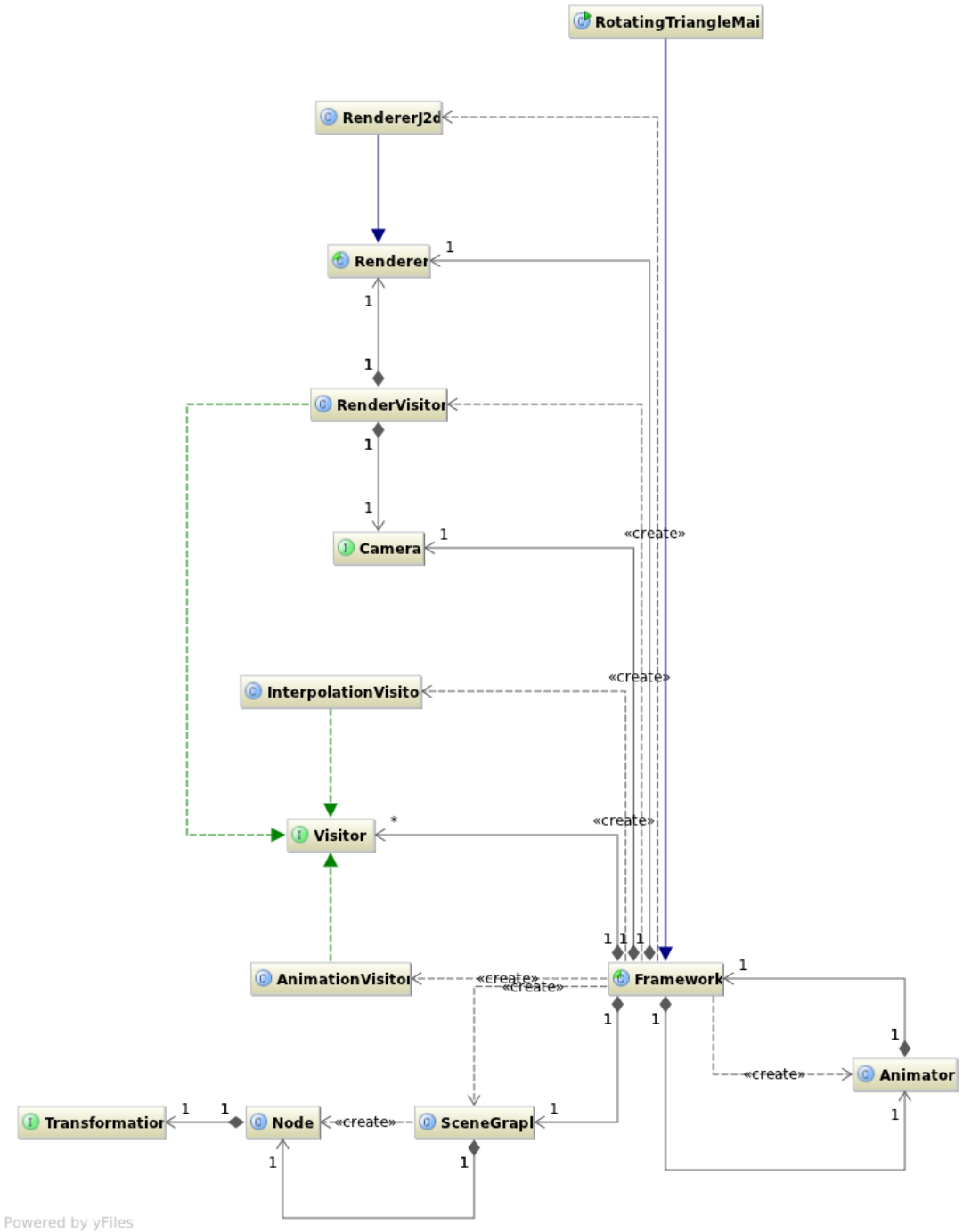

Abbildung 9: UML-Klassendiagramm des RotatingTriangle Beispiels.

Literatur

[1] T. Akenine-Möller, *Real-time rendering.*, 2nd ed. Natick Mass.: A K Peters, 2002.

[2] M. Bender, *Computergrafik: Ein anwendungsorientiertes Lehrbuch*. München: Hanser, 2003.

[3] K. Shoemake, "Animating rotation with quaternion curves." in *SIGGRAPH*, P. Cole, R. Heilman, and B. A. Barsky, Eds. ACM, 1985, pp. 245–254. [Online]. Available: http://dblp.uni-trier.de/db/conf/siggraph/siggraph1985.html#Shoemake85