Inhalt

1.0 Pipeline

Dieses Kapitel erläutert den Aufbau und die Funktionsweise der aktuellen Grafikpipeline, d.h. welche Stufen durchlaufen die Vertices und die daraus resultierenden Pixel eine Objektes, bis sie schließlich auf den Backbuffer gezeichnet werden. Dazu wird zuerst die ältere und mittlerweile ausgemusterte Fixed Function Pipeline erläutert, daran anschließend die neuere Programmable Pipeline. Das Kapitel soll ein Verständnis dafür schaffen, wann die verschiedenen Shader zum Einsatz kommen und welches Zahnrad diese innerhalb des gesamten Uhrwerks bilden.

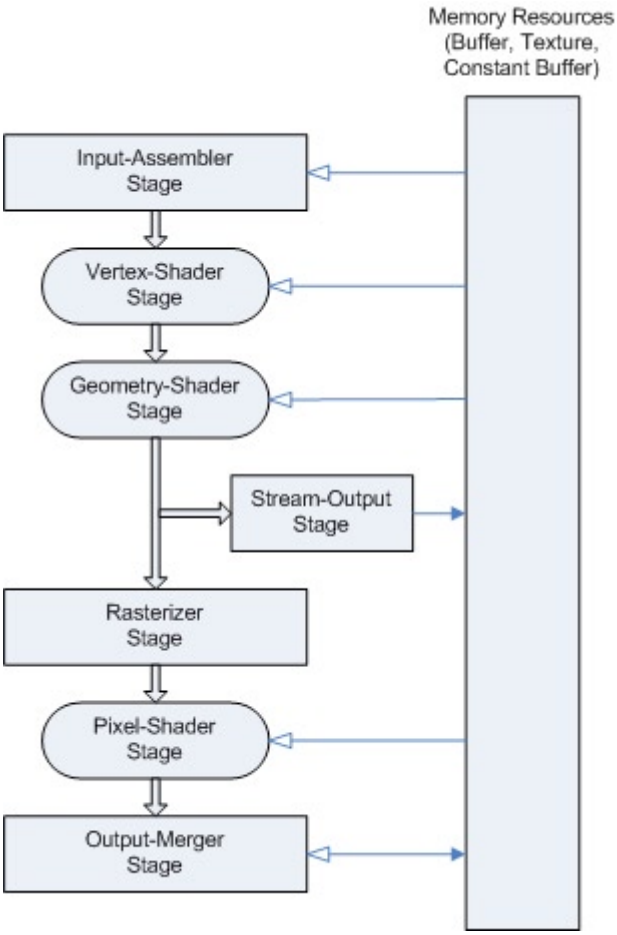

Abb. 1.0 *Pipeline [DX07]*

1.1 Fixed Function Pipeline

Wie aus dem Namen hervorgeht, sind in der Fixed Function Pipeline (FFP) die Transformation sowie die Lichtberechnung (T&L) fest vorgegeben und können nur über einige Zustandsänderungen geringfügig modifiziert werden. D.h. die FFP ist aufgrund dessen zwar schnell, aber unflexibel. Die Transformationen der Vertices, die Lichtberechnungen sowie die Berechnung der Farbe eines Pixel können nur eingeschränkt beeinflusst werden. Aus diesem Grund wurden Teile der Pipeline durch programmierbare Bausteine ersetzt, die sogenannten Shader. Durch den Einsatz der Shader haben die Programmierer mehr Einfluss auf die Berechnungen der Vertices, Primitive und Pixel.

1.2 Programmable Pipeline

Wie bereits erwähnt wurde, sind einige Stufen der Pipeline mittlerweile programmierbar, allerdings gilt dies nicht für alle. Zum Beispiel können Blending und Rasterung nicht frei programmiert werden, sondern werden wie bei der FFP lediglich über Zustände eingestellt. Es existieren drei programmierbare Teile innerhalb der Pipeline, dies sind der Vertex-Shader (Vertex-Processor), der Geometry-Shader (Geometry-Processor) sowie der Pixel-Shader (Pixel-Processor, Fragment-Processor). Die drei programmierbaren Bausteine sowie die restlichen Stufen der Pipeline, werden in den nachfolgenden Abschnitten genauer erklärt und in Abbildung 1.0 dargestellt. Wie die Abbildung verdeutlicht besitzt die Pipeline ein strickte Reihenfolge in der ihre Komponenten (Stufen) ausgeführt werden. Während eines Durchlaufs können die Komponenten auf Informationen (Memory Resources) innerhalb des Grafikspeichers zugreifen, genaueres dazu folgt in den nächsten Abschnitten.

1.2.1 Memory Resources

Die in Abbildung 1.0 gezeigten Memory Resources symbolisieren den Arbeitsspeicher der Grafikkarte, welcher für verschiedene Ressourcen verwendet werden kann. Zu den Ressourcen zählen unter anderem Vertex- und Indexbuffer, Texturen, Rendertargets, Shaderkonstanten, Stateblocks (Rasterizer States, Blend States, ...). In den meisten Fällen liest die Grafikkarte vom Speicher, zum Beispiel beim Lesen einer Textur oder beim Laden der Vertexdaten. An zwei Stellen, dem Stream-Output und dem Output-Merger, kann auch auf den Speicher geschrieben werden, zum Beispiel durch zeichnen der einzelnen Pixel auf den Backbuffer.

1.2.2 Input-Assembler

Der Input-Assembler (IA) regelt die Eingabe der Daten in die Pipeline. D.h. vom Benutzer angelegte Indexbuffer und ein oder mehrere Vertexbuffer werden zu Dreiecken, Linien oder Punkten zusammengesetzt (Engl. „assemble“). Weiterhin fügt der IA jedem Vertex systemgenerierte Daten hinzu (z.B. Vertex ID) und übergibt schließlich die einzelnen Vertices an der Vertex-Shader.

1.2.3 Vertex-Shader

Der Vertex-Shader (VSH) bearbeitet die vom Input-Assembler übergebenen Vertices. Normalerweise gehört dazu die Transformation der Position in den Projection Space (siehe *Kapitel 4.3 Projection Space*), Erstellung, Bearbeitung oder Generierung von Texturkoordinaten, Skinning (Animationen), per Vertex Lichtberechnungen, uvm.. Als Eingabe erhält der Vertex Shader immer einen einzigen Vertex und als Ausgabe liefert der Shader die transformierten Daten des Eingabe-Vertex an die Pipeline weiter.

1.2.4 Geometry-Shader

Der Geometry-Shader (GSH) bearbeitet Primitive (Dreiecke, Linien oder Punkte), deren Vertices bereits vom Vertex-Shader transformiert wurden. D.h. Eingabe sind entweder drei Vertices für ein Dreieck, zwei Vertices für eine Linie oder ein Vertex für ein Punkt-Primitiv. Eine Besonderheit ist, dass der Geometry-Shader zusätzliche Geometrie hinzufügen bzw. aus der Pipeline entfernen kann, im Engl. wird diese Eigenschaft als limited geometry amplification und de-amplification bezeichnet. Damit kann der GSH beispielsweise aus einem Punkt-Primitiv ein Dreieck erzeugen und somit eine variable Tesselierung der Objekte ermöglichen.

1.2.5 Stream Output

Ausgabe von Geometrie Daten in den Grafikkartenspeicher. Über den Stream Output ist es möglich Daten direkt in einen Buffer zu schreiben, auf welchen wiederum zu einem später Zeitpunkt zugegriffen werden kann. Dies kann neben der Übergabe zum Rasterizer geschehen und führt zu nicht Unterbrechnung der Pipeline.

1.2.6 Rasterizer

Der Rasterizer ist für das Clipping (zu Deutsch „abschneiden“) von Primitiven zuständig, d.h. nicht sichtbare Primitive werden vom Rasterizer erkannt und nicht weiter bearbeitet. Weiterhin bereitet der Rasterizer die Daten der transformierten Vertices für den Pixel Shader vor, dies beinhaltet die Abbildung auf den Viewport, die Rasterung in Pixel sowie die Interpolation der Vertex-Daten.
Nach dem die Rasterung des Primitiv‘s erstellt wurde, wird für die einzelnen Pixel ein z-Test ausgeführt (siehe *Kapitel 2.1.2 Depth-Stencil States*). Der z-Test entscheidet darüber, ob der Pixel-Shader für den Pixel ausgeführt werden soll oder der Pixel nicht weiter von der Pipeline behandelt wird.

1.2.7 Pixel-Shader

Der Pixel-Shader (PSH) berechnet für jeden Pixel eines Primtiv‘s verschiedene Farbtransformationen und eventuelle Tiefentransformationen. Dazu zählen beispielsweise per Pixel Lichtberechungen, Texturen laden oder sonstige Farbveränderungen. Als Eingabe erhält der PSH die interpolierten Daten eines Pixels, als Ausgabe generiert er ein oder mehrere Farben, die auf ein oder mehrer Rendertargets/Buffer geschrieben werden. Zusätzlich zu den Farbausgaben kann der PSH die Tiefe des Pixels verändern und ausgeben, dies führt allerdings dazu, dass der z-Test erst nach dem PSH ausgeführt werden kann.

1.2.8 Output Merger

Kombiniert die Ausgabe des Pixel-Shaders mit den Daten der Rendertargets/Backbuffers sowie Depth-Stencil-Buffers. D.h. die Pixel werden auf die verschiedene Buffer geschrieben und evtl. mit den bereits vorhanden Daten der Buffer geblendet, dies passiert je nach Einstellung der Pipeline.

2. Ressourcen

2.1 Zustände

Zustände beschreiben Einstellungen der Pipeline, zum Beispiel ob die Pipeline für jeden Pixel einen z-Test durchführen soll. Die Zustände der Pipeline werden in vier logische Kategorien unterteilt: Rasterizer States, Depth-Stencil States, Blend States und Sampler States. Die vier Kategorien werden in dem nachstehenden Kapitel genauer beschrieben.

2.1.1 Rasterizer States

Die Rasterizer States kontrollieren welche Primitive gezeichnet werden und falls diese gezeichnet werden, wie deren Rasterung durchgeführt wird. Dazu gehört beispielweise das Abschneiden (im Engl. „culling“) von Primitiven, falls diese gegen den Uhrzeigersinn oder mit dem Uhrzeigersinn gezeichnet werden. Außerdem kann über die Zustände entschieden werden, ob ein Primitiv komplett ausgefüllt (Solid), nur dessen Verbindungslinien (Wireframe) oder lediglich die Eckpunkte (Point) gerastert werden.

2.1.2 Depth-Stencil States

Über die Depth-Stencil States werden zwei Teile der Pipeline konfiguriert, zu diesen zählen Konfigurationen bzgl. des Tiefen- sowie des Stencilbuffers.
Der Tiefenbuffer enthält für jeden Pixel einen Tiefenwert, kurz z-Wert. Der Tiefen-Wert repräsentiert die Entfernung bis zur Near-Clip-Plane (siehe *Kapitel 4.3 Projection Space*). Je größer demnach der z-Wert eines Pixels ist, desto größer ist die Entfernung zur Clip-Plane. Bevor die Farbe eines Pixels auf den Backbuffer geschrieben wird bzw. bevor der Pixel-Shader für einen Pixel ausgeführt wird, wird der z-Wert des Pixels mit dem z-Wert auf dem Tiefenbuffer verglichen. Im Normalfall wird ein Pixel nur dann gezeichnet, wenn der z-Wert des neuen Pixels kleiner ist, als der äquivalente z-Wert aus dem Tiefenbuffer. In diesem Fall liegt der Pixel vor dem bereits existierenden Pixel. Zum besseren Verständnis zeigt Abbildung 2.0 die genaue Durchführung des Tiefen-Tests. Der Vergleichsoperator und die Aktivierung des z-Tests lassen sich über die Depth-Stencil States modifizieren.
Der zweite Teil kontrolliert Einstellungen bzgl. des Stencilbuffers (Stencil zu Deutsch „Schablone“). Mit Hilfe des Stencilbuffers kann jeder Pixel markiert und über einen speziellen Test, dem sogenannten Stencil-Test, abgefragt werden.

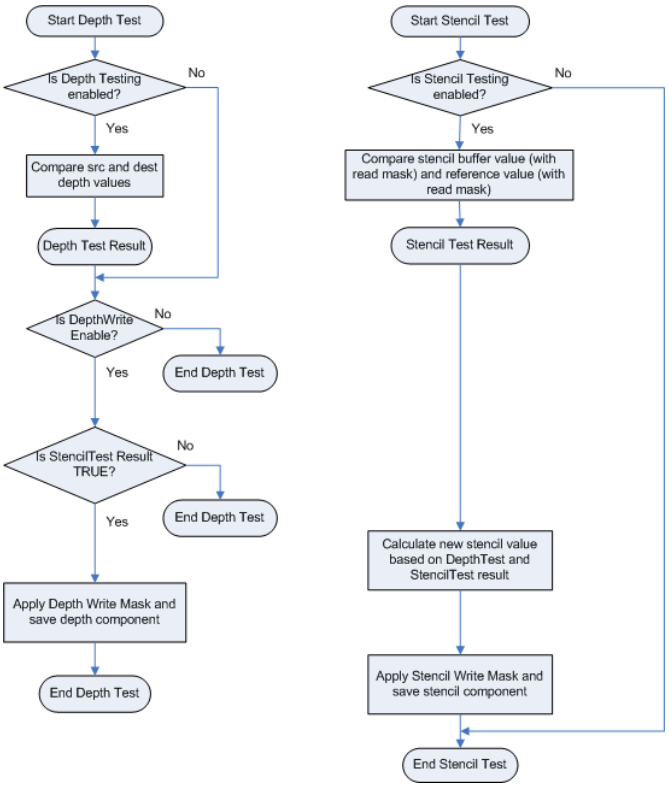

Abb. 2.0 *Depth- und Stencil-Test [DX07]*

2.1.3 Blend States

Mit Hilfe der Blend States können Einstellungen am Blending (zu Deutsch „Vermischung“) vorgenommen werden. Das Blending bestimmt wie neu zu zeichnende Pixel mit den bereits vorhandenen Pixeln auf dem Backbuffer interagieren. Die Zustände beinhalten zum Beispiel aktivieren und deaktivieren des Blendings oder festlegen der Blend-Operation (siehe Abb. 2.1).

2.1.4 Sampler States

Alle Zustände die beim Lesen von Texturen verwendet werden, gehören zu der Kategorie der Sampler States. Unter anderem bestimmen die Zustände mit welchem Filter (siehe *Kapitel 2.4.6 Filter)* eine Textur gesamplet wird oder wie die Adressierung vorgenommen wird, mehr dazu wird in den *Kapitel 2.4 Texturen* beschrieben.

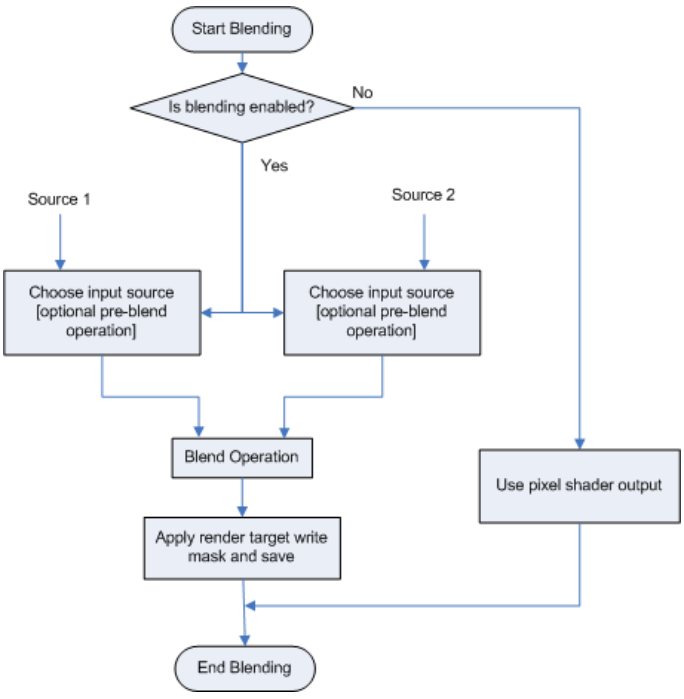

Abb. 2.1 *Blending [DX07]*

2.2 Buffer

2.2.1 Vertexbuffer

Der Vertexbuffer bildet einen Speicherbereich, in dem alle Vertices eines Objektes enthalten sind. Oder anders gesagt, ein Vertexbuffer ist eine Ansammlung von Vertices. Ein Vertex innerhalb des Buffers enthält wiederum verschiedene Informationen, meist gehören dazu die Position, die Farbe, die Normale und die Texturkoordinate des Vertex.

2.2.2 Indexbuffer

Der Indexbuffer enthält eine Ansammlung von Indizes. Jeder Index wird verwendet, um einen Vertex innerhalb des Vertexbuffers zu laden. D.h. der Input Assembler lädt nicht direkt drei aufeinanderfolgende Vertices aus dem Vertexbuffers und setzt diese zu einem Dreieck zusammen, sondern lädt zuerst drei aufeinanderfolgende Indizes aus dem Indexbuffer. Jeder Index wiederum bestimmt einen Vertex aus dem Vertexbuffer, diese werden dann zu einem Dreieck zusammengesetzt. Daraus ist ersichtlich, dass der Indexbuffer hauptsächlich das Auftreten redundanter Daten vermeiden soll.

2.3 Shaderkonstanten

Shaderkonstanten werden verwendet um Daten zu übertragen, die für alle Vertices und Pixel eines Objektes gleich sind, siehe *Kapitel 3.1 Eingabedaten*. Dazu zählen zum Beispiel die unterschiedlichen Transformationsmatrizen, Licht Einstellungen, Objektfarbe, uvm.. Jede Shaderkonstante besitzt vier Werte und bildet damit einen vierdimensionalen Vektor.

2.4 Texturen

Ähnlich dem Vertexbuffer bzw. Indexbuffer enthalten Texturen eine Ansammlung von Pixeln, die in diesem Fall allerdings als Texel bezeichnet werden. Damit ist ein Texel das kleinste Element aus einer Textur und besteht aus 1 bis 4 Komponenten bzw. Farben. So besitzt ein Texel entweder nur einen Alpha-Kanal, oder drei Farbkanäle sowie einen Alpha-Kanal, oder eins der vielen weiteren Formate (siehe *Kapitel 2.4.7 Formate*).
Im Gegensatz zu den bereits beschriebenen Vertex- und Indexbuffer (siehe *Kapitel 2.2 Buffer*) können Texturen im Shader beliebig oft gelesen (Engl. „fechted“ oder „sampled“) werden. Es kann allerdings immer nur ein gesamter Texel gelesen werden, das heißt ein Zugriff auf die einzelnen Farbkanäle kann erst nach dem Samplen des kompletten Texels erfolgen.

2.4.1 Eindimensionale Texturen

Eindimensionale Texturen enthalten ein eindimensionales Array von Texel. Um auf einen Texel zugreifen zu können, wird somit eine einzige Texturkoordinate benötigt. Die Koordinate wird meist als *u* bezeichnet und besitzt einen Wert von 0, wenn der erste Texel ausgewählt werden soll, bzw. einen Wert von 1, wenn der letzte Texel adressiert werden soll. Alle Werte zwischen 0 und 1 adressieren den entsprechenden Texel zwischen dem ersten und letzten Texel. Liegt die Koordinate zwischen zwei Texel, können diese über einen Filter (siehe *Kapitel 2.4.6 Filter*) gemischt werden.
Abbildung 2.2 verdeutlicht den Aufbau einer eindimensionalen Textur, die in dem gezeigten Beispiel 5 Texel enthält. Die weiteren Stufen stellen die einzelnen Mipmaps dar, mehr dazu im *Kapitel 2.4.5 Mipmaps*.

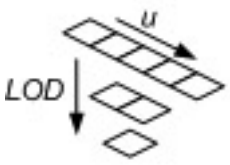

Abb. 2.2 *1D-Textur samt Mipmaps [DX07]*

2.4.2 Zweidimensionale Texturen

Falls die Textur ein zweidimensionales Array aus Texeln enthält, handelt es sich um eine zweidimensionale Textur. Jeder Texel wird diesmal über zwei Koordinaten angesprochen, wobei die beiden Koordinaten meistens als *u* und *v* bezeichnet werden. Hier gibt *u* die Verschiebung innerhalb einer Zeile und *v* die Adressierung der einzelnen Zeilen an. Wie bei der eindimensionale Textur reichen die Koordinaten *u* und *v* von (0, 0) (für den ersten Texel in der ersten Zeile) bis (1, 1) (für den letzten Texel in der letzten Zeile). Auch hier wird Aufbau inklusive der Mipmap-Levels durch Abbildung 2.3 verdeutlicht.

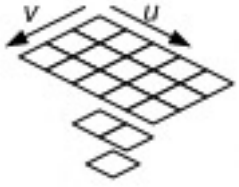

Abb. 2.3 *2D-Textur samt Mipmaps [DX07]*

2.4.3 Dreidimensionale Texturen

Dreidimensionale Texturen, auch als „Volumetextures“ bekannt, bilden ein 3D Volumen aus Texel. Eine Volumetexture besteht damit aus einer Ansammlung zweidimensionaler Texturen gleicher Größe, siehe Abbildung 2.4. Jede der verschiedenen zweidimensionalen Texturen innerhalb der Volumetexture, wird im weiteren Verlauf als Schicht (im Engl. „Slices“) bezeichnet.
Ein Texel wird hier über drei Koordinaten adressiert, die *u*, *v* und *w* Koordinate. Wie bei einer zweidimensionalen Textur adressieren *u* und *v* die einzelnen Spalten und Zeilen innerhalb einer einzelnen Schicht. Während die neue Koordinate *w,* die Auswahl der Schicht bestimmt.
Ebenfalls lesen hier die Koordinaten (u,v,w) = (0,0,0) den ersten Texel innerhalb der ersten Schicht und die Koordinaten (u,v,w) = (1,1,1) den letzten Texel aus der letzten Schicht.

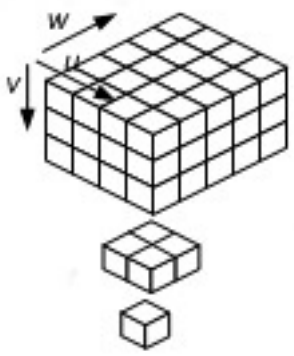

Abb. 2.4 *3D-Textur samt Mipmaps [DX07]*

2.4.4 Cubemaps

Eine Cubemap enthält sechs Texturen, wobei jede dieser Texturen jeweils die Seite eines Würfels bildet. Wie bei einer dreidimensionalen Textur müssen die Texel über die drei Koordinaten *u*, *v*, und *w* angesprochen werden. Allerdings bestimmen *u* und *v* nicht die Spalte und Zeile des zu adressierenden Texels, und *w* bestimmt nicht die auszuwählende Schicht. Vielmehr bilden die drei Koordinaten einen Vektor, der vom Zentrum des Würfels zu dessen inneren Seiten zeigt. Es wird der Texel adressiert auf den der Strahl des Vektors trifft.

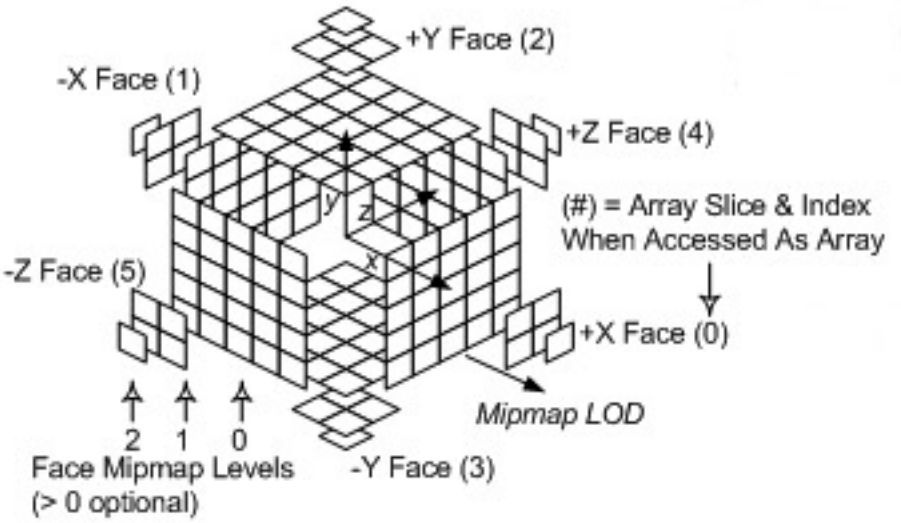

Abb. 2.5 *Cubemap samt Mipmaps [DX07]*

2.4.5 Mipmaps

Der Begriff Mipmap stammt aus dem lateinischen „multum in parvo“, was übersetzt soviel wie „Viel in Kleinem“ bedeutet. Mipmaps bilden eine Anreihung von Texturen, bei der jede Folgetextur zwar immer das gleiche Bild repräsentiert, dies allerdings in einer geringeren Auflösung. Genauer gesagt halbiert sich die Auflösung mit jedem Mipmap-Level, somit besitzt die letzte Mipmap eine Auflösung von 1x1 Pixel.
In Abhängigkeit davon wie nahe sich der Betrachter an einem Primitiv befindet (also je nach Abbildung der Texel auf die Pixel, siehe *Kapitel 2.4.6 Filter*), wird eine passende Mipmap ausgewählt. Die Hardware kann demnach feststellen welcher Mipmap-Level die nahste Auflösung zur gewünschten Ausgabe (Auflösung des Primitiv‘s) besitzt, und darauf hin den entsprechenden Mipmap-Level verwenden. Liegt die Auflösung zwischen zwei Mipmap-Levels, kann zwischen den beiden Mipmap-Stufen interpoliert/geblendet werden.
Mipmapping verringert das Auftreten von Aliasing und trägt somit zur Verbesserung der Bildqualität bei, dies allerdings auf Kosten des zusätzlich benötigten Speicherberdarfs.

2.4.6 Filter

Wird ein Primitiv zusätzlich durch eine Textur eingefärbt, so wird die Textur im Normalfall entweder „magnified“ oder „minified“ adressiert. D.h. die Auflösung der Textur entspricht nicht der Auflösung des Primitiv‘s und besitzt entweder eine größere oder kleinere Auflösung.
Im Fall einer Magnification wird ein einzelner Texel auf mehrere Pixel abgebildet. Obwohl die Pixel eine feinere Auflösung als die Textur besitzen, wird durch die Textur ein grobes Muster auf dem Primitiv erzeugt. Das Problem entsteht dadurch, dass jeder Pixel einfach dem ihm nächsten Texel auswählt (Nearest-Point Sampling). Anstatt den nahsten Texel auszuwählen, kann ein Pixel (der sich zwischen mehreren Texeln befindet) die Informationen aller benachbarten Texel verwenden und diese linear (bzw. bilinear im Fall einer zweidimensionalen Textur) filtern. Durch diesen Filter wird das grobe Muster der Textur verfeinert, was zu einer deutlichen Verbesserung der Qualität führt.

Im zweiten Fall tritt das genaue Gegenteil ein, d.h. es wird ein Pixel auf mehrere Texel abgebildet. Demnach besitzt die Textur eine höhere Auflösung, als das zu zeichnende Primitiv. Diesmal entsteht zwar kein grobes Muster durch eine zu geringe Auflösung der Textur, allerdings entsteht ein Aliasing Effekt, da der Pixel „zufällig" einen der vielen Texel auswählen muss. Eine Lösung des Problems wird durch die bereits erwähnten Mipmaps erreicht (siehe Kapitel *2.4.5 Mipmaps*). Durch den Einsatz von Mipmaps wird nicht zwangsläufig die feinste Auflösung einer Textur verwendet, stattdessen wird auf eine geringer aufgelöste Mimap-Stufe zurückgegriffen. Die ausgewählte Mipmap besitzt somit eine Auflösung, die der Auflösung des Primitv's nahe kommt und dadurch das Aliasing verringert.

2.4.7 Adressierung

Wie bereits beschrieben wurde, wird über die *u*, *v* und *w* Koordinaten jeweils ein Texel aus einer der verschiedenen Texturen (siehe *Kapitel 2.4.1 - 2.4.4*) adressiert. Dabei wurde der zulässige Bereich der Koordinaten von (0, 0, 0) bis (1, 1, 1) definiert.
Allerdings blieb bisher ungeklärt welcher Texel adressiert wird, wenn die Koordinaten kleiner als 0 oder größer als 1 ausfallen. Welche Adressierung in diesen speziellen Fällen durchgeführt werden soll, kann über die sogenannten Sampler States (Siehe *Kapitel 2.1.4 Sampler States*) bestimmt werden. Dazu existieren vier verschiedene Zustände (Wrap, Mirror, Clamp, und Border) innerhalb der Sampler States.
An dieser Stelle sei angemerkt, dass die folgenden Zustände nur für eindimensionale Texturen, zweidimensionale Texturen und dreidimensionale Texturen gelten. Da bei einer Cubemap (siehe *Kapitel 2.4.4 Cubemaps*) die Koordinaten einen Vektor bilden und keine direkte Stelle innerhalb der Textur definieren, bleiben die Koordinaten beim Adressieren einer Cubemap unverändert, egal welcher Zustand eingestellt wurde.

2.4.7.1 Wrap

Die Adressierung beginnt erneut von Vorne bzw. Hinten, falls die Koordinaten den zulässigen Bereich von (1, 1, 1) bzw. (0, 0, 0) verlassen. D.h. sobald die Koordinate die obere Grenze von 1 überschreitet, springt sie auf 0 zurück und wird von dort weiter hochgezählt. Falls im umgekehrten Fall, die Koordinate die untere Grenze von 0 unterschreitet, springt sie auf 1 hoch und wird von dort weiter heruntergezählt. Bei einer solchen Adressierung samplen demnach die Koordinaten (**0.8**, 0, 0) den gleichen Texel, wie die Koordinaten (**1.8**, 0, 0), (**2.8**, 0, 0), (**-0.2**, 0, 0), usw..

2.4.7.2 Mirror

Die Adressierung wird ähnlich der bereits beschriebenen Wrap-Adressierung (siehe *Kapitel 2.4.7.3 Wrap*) durchgeführt.
***

2.4.7.3 Clamp

Die Koordinaten werden auf den zulässigen Bereich abgeschnitten. D.h. alle Koordinaten werden auf 1 abgeschnitten, falls diese einen größeren Wert als 1 besitzt. Das Gleiche gilt für alle Koordinaten, die einen Wert kleiner als 0 aufweisen. Allerdings werden diese nicht auf 1, sondern auf 0 abgeschnitten.
Damit adressieren die folgenden Beispielkoordinaten immer den gleichen Texel: (0.2, **1.0**, 0.4), (0.2, **3.3**, 0.4), (0.2, **51.2**, 0.4), (0.2, **1.2**, 0.4) und alle weiteren Koordinaten deren *v* Komponente größer als 1 ist.

2.4.7.4 Border

Wie der Name bereits erahnen lässt, soll mit Hilfe dieses Zustands ein Rahmen für die Textur erzeugt werden. Erreicht wird dies, in dem alle Koordinaten außerhalb des zulässigen Bereichs einen vordefinierten Farbwert erhalten. Liegt also einer der Koordinaten (*u*, *v* oder *w*) nicht innerhalb des Intervalls von 0 bis 1, wird kein Texel aus der Textur adressiert, sondern lediglich die festgelegte Rahmenfarbe gelesen.

2.4.7 Formate

***

3. Shader

Die folgenden Kapitel werden noch einmal etwas genauer auf die einzelnen Shader eingehen, allerdings werden vorher ein paar grundlegende Begriffe bzgl. der Eingabe- und Ausgabedaten erklärt.

3.1 Eingabedaten

Daten die von einem Shader gelesen werden, werden in zwei Kategorien unterteilt: Uniform Inputs und Varying Inputs. Wie aus dem Namen bereits hervorgeht, zählen zu den Varying Inputs, Daten die für jeden Vektor bzw. Pixel variieren. Bei einem Vertex Shader sind dies beispielsweise alle Daten eines Vertex (Position, Normale, Texturkoordinate, ...). Varying Inputs werden zu Beginn des Shaders in spezielle Eingaberegister übertragen und können über diese abgefragt bzw. verwendet werden.
Uniform Inputs hingegen sind für alle Vektoren, Primitive und Pixel gleich und variieren nicht beim Durchlaufen der Pipeline. Zum Beispiel zählen alle Shaderkonstanten (siehe *Kapitel 2.3 Shaderkonstanten*) und Texturen (siehe *Kapitel 2.4 Texturen*) zur Kategorie der Uniform Inputs. Weiterhin werden Uniform Inputs nicht in spezielle Eingaberegister übertragen, sondern sind ständig in den Memory Resources vorhanden und können demnach „beliebig“ oft abgefragt werden.

3.2 Ausgabedaten

Ausgaben an die Pipeline können nur über spezielle Ausgaberegister erfolgen, d.h. ein Shader kann die Daten nicht direkt auf die Memory Resources (z.B. Shaderkonstanten oder Texturen) schreiben. Die Ausgaberegister unterscheiden sich je nach Typ des Shaders und werden in den Kapiteln des jeweiligen Shaders (siehe *Kapitel 2.5.3 Vertex Shader* und *Kapitel 2.5.4 Pixel Shader*) genauer erklärt.

3.3 Vertex Shader

Wie in Kapitel *1.2.3 Vertex Shader* beschrieben, bearbeitet der Vertex Shader die vom Input Assembler übergebenen Vertices. Aus diesem Grund wird das Vertex Shader Programm für jeden Vertex genau einmal ausgeführt. Die Aufgabe des Vertex Shader besteht darin Transformationen und sonstige Berechnungen für alle Vertices eines Objektes auszuführen.

Als variierende Eingabe erhält der Vertex Shader die Daten des zu bearbeitenden Vertex, diese beinhalten beispielsweise dessen Position, Normale und Texturkoordinaten. Als Ausgabe muss der Vertex Shader mindestens eine Position im Projection Space ausgeben (siehe *Kapitel 4.3 Projection Space*), diese benötigt die Pipeline zur Rasterung des Primitiv‘s. Weiterhin bedeutet dies, dass die Transformation der Vertexposition nicht mehr automatisch von der FFP ausgeführt wird, sondern nun vom Vertex Shader übernommen werden muss. Das gleiche gilt für per Vertex Lichtberechnungen oder sonstige Transformationen.
Neben der Position kann der Vertex Shader zusätzliche Daten an die Pipeline weiterreichen, zum Beispiel die Texturkoordinaten oder das Ergebnis der bereits erwähnten per Vertex Lichtberechnung. Diese Daten werden wie die Position in die speziellen Ausgaberegister übergeben und bilden damit die Varying Inputs für den Geometry Shader bzw. den Pixel Shader.

3.4 Pixel Shader

Für jeden zu zeichnenden Pixel eines Primitiv‘s, ruft der Rasterizer einmal den Pixel Shader auf. Der Pixel Shader bietet damit die Möglichkeit für jeden Pixel ein komplexe Berechnung durchzuführen, diese beinhaltet zum Beispiel eine per Pixel Lichtberechnung oder einen Bump Mapping Effekt.

Alle Daten die vom Vertex Shader ausgegeben wurden, zum Beispiel Texturkoordinaten oder Farben, erhält der Pixel Shader als variierende Eingabe. Dabei liest der Pixel Shader die Daten über das gleiche Register ein, welches der Vertex Shader bereits als Ausgaberegister verwendet hat. Man könnte sagen, dass der Pixel Shader als Eingabedaten alle Ausgabendaten des Vertex - bzw. Geometry Shaders erhält. Wie bereits erwähnt wurde (siehe *Kapitel 1.2.6 Rasterizer*) werden die Ausgabedaten des Vertex Shaders vom Rasterizer interpoliert (dies geschieht anhand der Pixel-Position innerhalb des Primitiv‘s) und danach schließlich als variierende Eingabedaten an den Pixel Shader weitergereicht.
Als Ausgabe muss der Pixel Shader mindesten den Farbwert des Pixels an die Pipeline ausgeben. Zusätzlich kann der Pixel Shader den z-Wert (siehe *Kapitel 2.1.2 Depth-Stencil States*) des Pixels berechnen und ebenfalls über ein spezielles Ausgaberegister an die Pipeline weiterleiten. Durch die Ausgabe des Tiefenwerts, kann der Tiefen-Test allerdings nicht vor dem Pixel Shader erfolgen, da zu diesem Zeitpunkt noch kein konkreter z-Wert für den Pixel existiert. D.h. der z-Test erfolgt erst nach dem ausführen des Pixel Shaders, dies kann zu einer Verringerung der Performanz führen.

4. Transformationen

Da die Anwendung in der Lage sein soll dreidimensionale Objekte zu erzeugen und diese des Weiteren auf einem (zweidimensionalen) Bildschirm ausgeben soll, werden verschiedene Systeme benötigt. Zur Repräsentation des dreidimensionalen Raumes werden meistens Kartesische-Koordinatesysteme verwendet, diese besitzen drei Achsen (X, Y und Z), welche jeweils senkrecht aufeinander stehen. Um eine möglichst einfache und verständliche Transformation aus dem dreidimensionalen Raum, in dem sich die Objekte befinden, in den zweidimensionalen Raum zur Ausgabe des Bildes zu erreichen, werden fünf verschiedene Räume bzw. deren Transformationen eingesetzt.

4.1 Object Space und World Space

Der Object Space, oft auch als Model Space bekannt, ist der bevorzugte Raum in dem 3D-Objekte generiert und bearbeitet werden. Dabei wird das Objekt meistens um den Ursprung des Raumes erzeugt, wodurch spätere Transformationen (z.B. eine Rotation oder Skalierung) leichter ausgeführt werden können.
Damit alle Objekte in Relation zueinander stehen, müssen sie in einen neuen, gemeinsamen Raum transformiert werden, den World Space. Im World Space (siehe Abb. 4.2) befinden sich neben den Objekten ebenfalls alle weiteren Komponenten, beispielsweise die Kamera und die verschiedenen Lichtquellen.

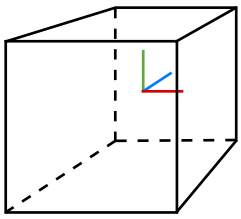

Abb. 4.1 *Object Space*

4.2 View Space

Nach dem sich nun alle Objekte im World Space befinden, wird im nächsten Schritt ein Raum benötigt, der die gesamte Welt aus der Perspektive einer definierten Kamera (Betrachter) darstellt. Dazu werden die Objekte zuerst vom World Space in den sogenannten View Space abgebildet. Der View Space kontrolliert die äußeren Kameraeinstellungen, d.h. Position und Ausrichtung der Kamera. Die inneren Einstellungen (Field of View, Clip-Planes) einer Kamera werden im darauf folgenden Schritt durch die Transformation in den Projection Space (siehe *Kapitel 4.3 Projection Space*) simuliert.
Im View Space bildet die Kamera den Ursprung des Raums und die Blickrichtung der Kamera definiert die positive z-Achse. Diese beiden Bedingungen sind notwendig, damit die darauf folgende perspektivische Transformation möglichst einfach umgesetzt werden kann.

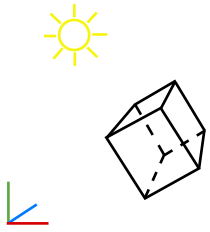

Abb. 4.2 *World Space*

4.3 Projection Space

In dem beschriebenen View Space (siehe *Kapitel 4.2 View Space*) werden bereits alle Objekte relativ zur Position und Ausrichtung der Kamera angegeben, allerdings fehlen zwei weitere wichtige Bedingungen. Zum einen soll der sichtbare Bereich der Kamera, definiert durch ein Viewing Frustum (siehe Abb. 4.3), auf einen Quader (siehe Abb. 4.4) abgebildet werden. Durch diese Abbildung kann die Grafikkarte schnell und effizient Vertices bzw. Pixel des Objektes entfernen, falls diese außerhalb des sichtbaren Bereiches liegen. Zum anderen soll eine perspektivische Transformation bzw. Projektion ermöglicht werden. Eine Projektion läßt Objekte größer erscheinen, je näher sich diese an der Kamera befinden. Diese Eigenschaft ist vergleichbar mit der Linse einer Kamera.

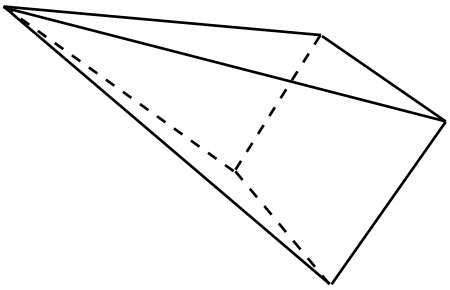

Abb. 4.3 *Viewing Frustum*

Die im *Kapitel 4.1 Object Space und World Space* und *Kapitel 4.2 View Space* beschriebenen kartesischen Räume, können jeweils durch einfache Translation, Rotation und Skalierung von einem Raum in den nächsten Raum abgebildet werden. Dabei wurde die vierte Komponente (w) eines Vektors lediglich dazu verwendet, die Ausführung einer Translation zu ermöglichen. Wie sich im weiteren Verlauf herausstellen wird, wird die w-Komponente nun ebenfalls benötigt, um die gewünschte Projektion zu erreichen. Hierbei spielt der

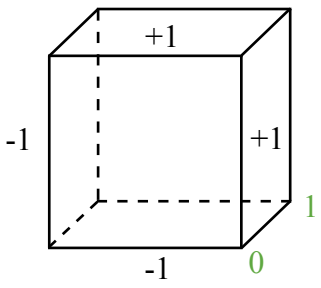

Abb. 4.4 *Projection Space*

Rasterizer (siehe *Kapitel 1.2.6 Rasterizer*) eine wichtige Rolle. Dieser dividiert die Position (gegeben durch x, y, z, w) eines jeden Pixels durch dessen w-Komponente. Nach dieser Division erhält der Rasterzier die endgültige Position des Pixels und kann diese schließlich in den Screen Space (siehe *Kapitel 4.4 Screen Space*) transformieren. Da die Projektion in Form einer Matrix, sich als etwas komplexer erweist, wird nun genauer auf deren Aufbau eingegangen.

Durch Anwendung einer Projektion auf das View Frustum, wird das pyramidenartige Frustum auf eine Ebene abgebildet. Das Prinizp der Transformation basiert auf der Entfernung (z im View Space) der einzelnen Vertices zur Kameraposition (Ursprung im View Space). D.h. die Position des Vertex im Projection Space ist abhängig von dem Verhältnis zwischen x bzw. y zu dessen z-Wert. Das Verhältnis (x/z bzw. y/z) gibt die Steigung der Geraden an, die vom Ursprung durch den Vertex führt und verschiebt damit alle Vertices auf der Geraden, in den selben Punkt innerhalb der Projektionsebene.
Der erste Schritt besteht demnach darin die Projektion in Form einer Matrix zu erstellen, hierfür wird die bereits erwähnte w-Komponente benötigt. Alle Punkte innerhalb des Viewing Frustum (in Abbildung 4.5 als eine Seitenansicht dargestellt) sollen auf eine Ebene projiziert werden. Die Projektionsebene wird in der Abbildung rot gekennzeichnet.

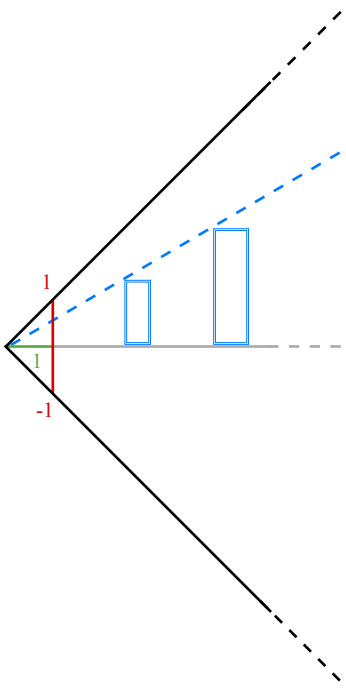

Abb. 4.5 *Projektion*

Der projizierte Vertex (x', y', z', w') kann erreicht werden, indem die drei Komponenten des Punktes durch die z-Komponente dividiert werden. Daraus ergeben sich folgende drei Formeln.

$$x' = x / z$$
$$y' = y / z$$
$$z' = z / z = 1$$

Die passende Matrix zu den aufgeführten Formeln sieht folgendermaßen aus:

$$(x', y', z', w') = (x, y, z, 1) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Indem w' den Wert von z annimmt und der Rasterizer alle Komponenten (x', y' und z') durch w' dividiert, kann mithilfe der aufgestellten Matrix indirekt eine Division erreicht werden.
Allerdings ist erkennbar das z' immer einen Wert von 1 erhält. Dies wird auch aus Abb. 4.5 ersichtlich, da alle Punkte auf die Projektionsebene abgebildet werden und diese sich genau um eine Einheit entfernt vom Ursprung befindet. Damit die z-Komponente nicht ihre Information verliert, muss die aufgestellte Matrix noch bearbeitet werden.

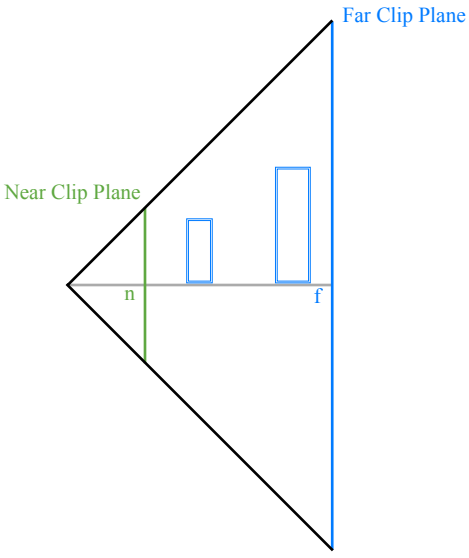

Abb. 4.6 *Clip-Planes*

Wie in der Einleitung des Kapitels beschrieben wurde, soll der Bereich des Viewing Frustums auf einen Quader abgebildet werden, siehe Abb. 4.4. Der Quader besitzt entlang der z-Achse ein Ausmaß von 0 bis 1.
Damit diese Abbildung durchgeführt werden kann, müssen zwei Ebenen für das Frustum definiert werden, die sogenannte Near- und Far-Clip-Plane (siehe Abb. 4.6). Alle Punkte zwischen den beiden Ebenen, die parallel zur XY-Ebene verlaufen, sollen im Projection Space einen z-Wert zwischen 0 und 1 besitzen. Dabei werden die Punkte den Wert 0 annehmen, falls diese genau auf der Near-Clip-Plane liegen, und den Wert 1, falls sie sich auf der Far-Clip-Plane befinden.

$$(x', y', z', w') = (x, y, z, 1) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -Q \cdot n & 0 \end{bmatrix}$$

$$Q = \frac{f}{f - n}$$

n = Near-Clip-Plane
f = Far-Clip-Plane

Die neu aufgestellte Matrix bewirkt eine veränderte Skalierung und Verschiebung für den z-Wert des Vertex. Dabei wird z mit dem reziproken Abstand ($\frac{1}{f-n}$) der beiden Clip-Planes skaliert. Durch diese Transformation wird der Abstand zwischen Near-Clip-Plane und Far-Clip-Plane auf eine Größe von 1 gestaucht (evtl. gestreckt). Allerdings reichen diese noch nicht von 0 bis 1, dazu muss die gestauchte Near-Clip-Plane auf den Ursprung verschoben werden (- Q • n). Da bei der Skalierung berücksichtigt werden muss, dass der Rasterizer später alle Komponenten eines Vertex (damit auch z) durch die w-Komponente (damit durch z) dividiert. Um diese Division zu negieren wird nicht einzig durch den Abstand der beiden Ebenen dividiert, sondern noch mit der Far-Clip-Plane (den größten Wert den z annehmen kann) mutlipliziert ($\frac{f}{f-n}$).

4.4 Screen Space

Im letzten Schritt transformiert der Rasterizer die einzelnen Pixel in den Screen Space. Diese Position wird schließlich verwendet um einen Verweis auf den Backbuffer zu erhalten. Da der Backbuffer im Grunde einer zweidimensionalen Textur entspricht, wird ein zweidimensionaler Raum benötigt, um jede Stelle innerhalb des Backbuffers zu erreichen. Die linke, obere Ecke bildet den Ursprung des Screen Space. Die positive x-Achse verläuft nach rechts und die positive y-Achse nach unten. Damit besitzt der letzte Pixel innerhalb des Backbuffers die Koordinate (w-1, h-1), was zugleich der rechten, unteren Ecke des Backbuffers enstpricht.

Quellen:

[DX07] DirectX Documentation 2007