Daniel Ilett

Erstellung hochwertiger Shader für Unity®

Verwendung von Shader Graphs und HLSL Shaders

Springer Vieweg

Erstellung hochwertiger Shader für Unity®

Verwendung von Shader Graphs und HLSL Shaders

Daniel Ilett

Apress®

Erstellung hochwertiger Shader für Unity®: Verwendung von Shader Graphs und HLSL Shaders

Daniel Ilett
Coventry, UK

ISBN 979-8-8688-0429-8 ISBN 979-8-8688-0430-4 (eBook)
https://doi.org/10.1007/979-8-8688-0430-4

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über https://portal.dnb.de abrufbar.

Übersetzung der englischen Ausgabe: „Building Quality Shaders for Unity®“ von Daniel Ilett,

Dieses Buch ist eine Übersetzung des Originals in Englisch „Building Quality Shaders for Unity®“ von Daniel Ilett, publiziert durch APress Media, LLC im Jahr 2022. Die Übersetzung erfolgte mit Hilfe von künstlicher Intelligenz (maschinelle Übersetzung). Eine anschließende Überarbeitung im Satzbetrieb erfolgte vor allem in inhaltlicher Hinsicht, so dass sich das Buch stilistisch anders lesen wird als eine herkömmliche Übersetzung. Springer Nature arbeitet kontinuierlich an der Weiterentwicklung von Werkzeugen für die Produktion von Büchern und an den damit verbundenen Technologien zur Unterstützung der Autoren.

Planung/Lektorat: Spandana Chatterjee
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft APress Media, LLC und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: 1 New York Plaza, New York, NY 10004, U.S.A.

Jeder Quellcode oder anderes ergänzendes Material, auf das der Autor in diesem Buch verweist, ist für die Leser über die Produktseite des Buches auf GitHub verfügbar, zu finden unter www.apress.com/. Für detailliertere Informationen besuchen Sie bitte http://www.apress.com/source-code.

Wenn Sie dieses Produkt entsorgen, geben Sie das Papier bitte zum Recycling.

An Lewis.

Inhaltsverzeichnis

Über den Autor

Daniel Ilett ist ein ehrgeiziger und motivierter Doktorand an der Universität von Warwick. Er ist ein leidenschaftlicher Spieleentwickler, spezialisiert auf Shader und technische Kunst. Er veröffentlicht eine Reihe von Lehr- und Tutorialinhalten, einschließlich Videos und schriftlicher Arbeiten, die sich an Anfänger und fortgeschrittene Entwickler richten. Er arbeitet auch freiberuflich an Shadern und visuellen Effekten für Spiele.

Über den technischen Gutachter

Simon Jackson ist ein langjähriger Software-Ingenieur und Architekt mit langjähriger Erfahrung in der Entwicklung von Unity-Spielen sowie Autor mehrerer Titel zur Unity-Spielentwicklung. Er liebt es, sowohl Unity-Projekte zu erstellen als auch anderen zu helfen, sich weiterzubilden, sei es über einen Blog, Vlog, eine Benutzergruppe oder eine große Vortragsveranstaltung.

Sein Hauptaugenmerk liegt derzeit auf dem XRTK(Mixed Reality Toolkit)-Projekt. Dieses zielt darauf ab, ein plattformübergreifendes Mixed Reality Framework zu erstellen, um sowohl VR- als auch AR-Entwicklern zu ermöglichen, effiziente Lösungen in Unity zu erstellen und diese dann auf so vielen Plattformen wie möglich zu bauen/verteilen.

Danksagungen

Mit Dank an Lewis, der mir alle meine Tweets Sekunden nach dem Posten laut vorliest. Etwas sagt mir, dass er mir dieses ganze Buch nicht vorlesen wird. Du musstest auch mit mir am Morgen umgehen, bevor ich einen Kaffee getrunken hatte, was nicht einfach gewesen sein könnte.

Vielen Dank auch an meine Familie, insbesondere an meine Mutter, die anscheinend immer sagt „Wusstest du, dass er ein Buch schreibt?“, wann immer mein Name in Gesprächen mit Leuten erwähnt wird.

Mit Dank an die Warwick Game Design Society und ihre Mitglieder, die mir einen Raum gegeben haben, in dem ich mich entfalten und während meines Studiums an Spielen arbeiten konnte. Ihr habt mir auch gezeigt, wie man ein gigantisches Stück Schokoladenkuchen in weniger als einer Minute isst, was uns sicherlich einige seltsame Blicke von anderen Kneipenbesuchern eingebracht hat. Es gibt zu viele von euch, um sie zu zählen!

Schließlich gilt mein Dank dem Apress-Team und allen, die mit mir an diesem Buch gearbeitet haben. Von der ersten Einladung, das Buch zu schreiben, bis hin zu jedem Ratschlag und jeder Korrektur, Sie alle waren unschätzbare Partner auf dieser Reise gewesen.

Einführung

Sie haben sich also entschieden, mit dem Schreiben von Shadern in Unity zu beginnen. Das ist eine gute Entscheidung, wenn ich das so sagen darf! Bevor ich anfing, Shader zu schreiben, hörte ich viele Leute im Bereich der Spieleentwicklung in gedämpften Tönen darüber sprechen, als ob es sich um eine Art geheimnisvolles, vom Himmel gegebenes Wissen handelte, aber ich glaube, dass jeder Shader schreiben kann, wenn er behutsam eingeführt wird. Es ist wie das Erlernen jeder anderen Fähigkeit; man lernt nicht, ein Fahrrad zu fahren, indem man gleich zu Beginn Evel Knievel imitiert.

Dies ist die Art von Buch, die ich mir gewünscht hätte, als ich lernte, Shader in Unity zu schreiben. Die ersten Kapitel vermitteln die Kernkonzepte und Syntax, die Sie kennen müssen, wenn Sie Shader schreiben, und im Laufe des Buches werde ich stetig komplexere Themen einführen, bis wir einige der weniger bekannten Shader-Themen in Unity abgedeckt haben. Zu diesem Zeitpunkt sollten Sie in der Lage sein, komplizierte, vielseitige Effekte selbst zu erstellen.

Für wen ist dieses Buch gedacht?

Dieses Buch behandelt Themen für Anfänger und Fortgeschrittene. Ich gehe davon aus, dass Sie sich bereits im Unity-Editor auskennen, aber ich werde die Benutzeroberfläche von unbekannten Bereichen wie dem Shader-Graph-Editor erklären. Hier sind einige hypothetische Leser, von denen ich glaube, dass sie dieses Buch genießen werden:

- Ich habe noch nie einen Shader geschrieben.

  Wenn Sie das sind, dann empfehle ich definitiv, mit Kap. 1 zu beginnen und das Buch in der Reihenfolge durchzuarbeiten, und Sie sollten Ihr Projekt mit der Universal Render Pipeline (URP; wenn Sie nicht wissen, was das bedeutet, wird alles in Kap. 1 erklärt). Kap. 2 ist optional – es behandelt alle mit Shadern verbundenen mathematischen Aspekte, aber nicht jeder lernt Mathematik leicht, indem er sie alles im Voraus liest, also können Sie es für den

Anfang überspringen und zurückzukommen, wenn in späteren Kapiteln Mathematik eingeführt wird, mit der Sie nicht vertraut sind.

- Ich habe vor ein paar Jahren Shader für alte Unity-Versionen geschrieben, aber ich kenne mich mit Unity-Versionen nach 2018–2019 oder so nicht aus.

  Die frühen Kapitel dienen als gute Auffrischung, aber Sie können wahrscheinlich mindestens Kap. 1 überfliegen, vielleicht mehr. Wählen Sie ein paar der Beispiele aus und probieren Sie sie aus, um sich an die Syntax und Konzepte zu erinnern, bevor Sie in die späteren Kapitel eintauchen!

- Ich habe Erfahrung mit dem Schreiben von Shadern im Code, aber ich möchte auf die Verwendung von Shader Graph (oder umgekehrt) umsteigen.

  Jedes Beispiel in diesem Buch wird, soweit möglich, sowohl im Shader-Code als auch im Shader Graph dargestellt (es gibt eine kleine Anzahl von Effekten und Funktionen, die aus verschiedenen Gründen nur in einem dieser Tools möglich sind). Es kann hilfreich sein, Beispiele durchzulesen, mit denen Sie vertraut sind, und diese als Sprungbrett zu nutzen, um das Tool zu erlernen, mit dem Sie nicht so viel Erfahrung haben.

- Ich habe die Grundlagen verstanden, aber ich möchte zu schwierigeren Themen wie Compute-Shadern und Tessellation-Shadern übergehen.

  Obwohl die früheren Kapitel als Referenz nützlich sein werden, sollten Sie die Kap. 1–4, die die Grundlagen behandeln, überspringen oder überfliegen und dann die nachfolgenden Kapitel von Fall zu Fall lesen. Die fortgeschrittensten Themen beginnen ab Kap. 10, aber ich habe versucht, mittlere Funktionen im gesamten Buch zu streuen.

Empfohlene Software

Jedes Beispiel in dem Buch *wird* in **Unity 2021.3** funktionieren, welches zur Zeit der Erstellung dieses Buches die neueste Long-Term-Support(LTS)-Version der Engine ist. Wenn Sie dieses Buch in der Zukunft lesen und es gibt eine spätere LTS-Version, dann wird jedes Beispiel höchstwahrscheinlich immer noch funktionieren, obwohl Sie möglicherweise die Namen von Funktionen und Makros ändern müssen. Wenn Sie eine frühere Version als 2021.3 verwenden, werden die meisten Effekte funktionieren, aber es gibt Funktionen, die in bestimmten Versionen vollständig fehlen. Das Buch wird immer noch hilfreich sein, wenn Sie an einem Projekt arbeiten, das in einer früheren Unity-Version gestartet wurde und Sie nicht auf eine neuere Version aktualisieren können, und ich werde Sie warnen, wenn ich ein kürzlich hinzugefügtes Feature verwende.

KAPITEL 1

Einführung in Unity-Shader

Jeder möchte wunderschöne Spiele erstellen, und Shader sind eines der leistungsfähigsten Werkzeuge im Toolkit des technischen Künstlers, um genau das zu erreichen. In diesem Kapitel werden wir einen Schritt zurückgehen und kurz untersuchen, wie das Rendern in Unity funktioniert und welche Rolle Shader dabei spielen. Am Ende des Kapitels werden Sie die Grundlagen des Rendering-Prozesses verstehen, was die verschiedenen Render-Pipelines von Unity sind und wie wir Programme namens Shader verwenden, um zu steuern, wie unsere Spiele aussehen.

Rendering

Eine *Spiel-Engine* ist nichts anderes als eine Werkzeugkiste voller Software, die darauf ausgerichtet ist, Spiele zu erstellen. Die Software, die uns am meisten interessiert, ist der *Renderer* – der Code, der Meshs, Sprites und Shader nimmt und sie alle in die Bilder umwandelt, die Sie auf Ihrem Bildschirm sehen. Unity unterscheidet sich in dieser Hinsicht nicht von anderen Engines. Auf hoher Ebene verarbeitet der Renderer eine Reihe von Daten durch eine Anzahl unterschiedlicher Schritten, wie in Abb. 1-1 zu sehen. Einige der Stufen werden automatisch von Unity gehandhabt, sodass wir uns keine Sorgen um sie machen müssen, während andere vollständig durch die Verwendung von Shadern, die wir schreiben werden, gesteuert werden. Wie wir sehen werden, hat jede Stufe einen spezifischen Zweck und einige sind sogar optional. Lassen Sie uns kurz die Geschichte der Grafik-Pipeline erkunden und sehen, wie Shader überhaupt entstanden sind.

D. Ilett, *Erstellung hochwertiger Shader für Unity®*, https://doi.org/10.1007/979-8-8688-0430-4_1

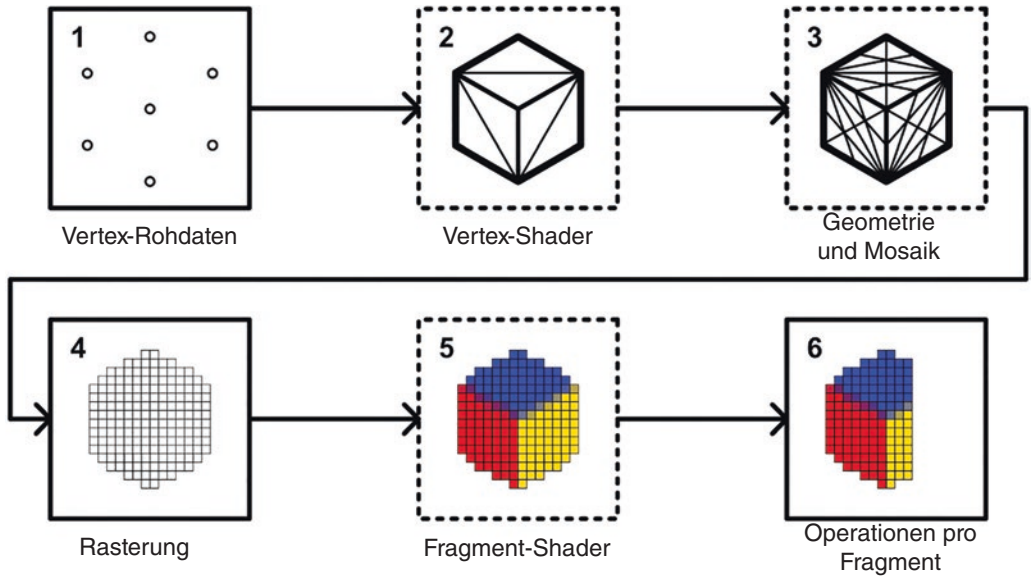

Abb. 1-1. *Die Stufen der Grafik-Pipeline. Die von einer gestrichelten Linie umgebenen Stufen zeigen Teile der Pipeline an, die wir mit Shadern anpassen können*

Eine Rendering-Grafikpipeline wandelt die rohen Vertex-Daten in fragmentbezogene Operationen um durch den Vertex-Shader, Geometrie und Tessellation, Rasterisierung und Fragment-Shader.

In alten Zeiten (den 1990er Jahren), mit dem Aufkommen von Echtzeit-3D-Grafiken, war die Funktionalität, die Grafikprogrammierern zur Verfügung stand, begrenzt. Wir hatten das, was man die *Fixed-Function-Pipeline* nennt, bei der wir eine relativ kleine Reihe von spezialisierten Funktionen (wie „Beleuchtung durchführen", „Texturen hinzufügen" und so weiter) konfigurieren konnten, die auf den GPUs der damaligen Zeit liefen. Dies war schneller als vorherige Software-Rendering-Methoden, die auf der CPU liefen, kam aber mit schweren Einschränkungen. Die Fixed-Function-Pipeline ist vergleichbar mit dem Bau eines Autos mit fünf Optionen für die Karosserieform, sieben für die Farbe und drei für die Motorgröße; sie deckte viele Anwendungsfälle ab, aber wir brauchen eine feinere Kontrolle, um ein Auto mit jeder erdenklichen Form und allen Farben des Regenbogens und darüber hinaus zu bauen.

Hier kommt die *programmierbare Pipeline* ins Spiel. In der modernen Computergrafik können wir das Verhalten bestimmter Teile des Rendering-Prozesses vollständig steuern, indem wir kleine Programme namens *Shader* verwenden. Wir sind

nicht mehr auf eine begrenzte Reihe von Funktionen beschränkt – jetzt können wir C-ähnlichen Code schreiben, um die Grafikkarte anzuweisen, jede Kombination von Effekten auszuführen. Das Unmögliche wurde möglich. Während die Fixed-Function-Pipeline wie das Betätigen einer Reihe von Hebeln ist, die jemand anderes für uns gemacht hat, ist die programmierbare Pipeline wie der Bau einer ganzen Maschine aus Rohmaterialien nach unseren genauen Spezifikationen.

Wie Sie aus Abb. 1-1 ersehen können, gibt es mehrere Stufen, von denen jede einen spezifischen Zweck erfüllt:

- Der *Vertex-Shader* ermöglicht es uns zu steuern, wie Objekte auf dem Bildschirm platziert werden. Dieser Shader-Typ erhält in der Regel Vertex-Daten von einem Mesh und verwendet mathematische Transformationen, um sie richtig auf dem Bildschirm zu positionieren. Sie können auch diese Shader-Stufe verwenden, um Vertex-Positionen im Laufe der Zeit zu animieren.
- Der *Tessellation-Shader* kann verwendet werden, um ein Mesh in kleinere Dreiecke zu unterteilen und zusätzliche Vertices zu erstellen. Dies ist nützlich, wenn es mit Vertex-Shadern kombiniert wird, die die Positionen der Vertices verschieben, da Sie detailliertere Ergebnisse erzielen können.
- Der *Geometrie-Shader* kann verwendet werden, um beliebige neue primitive Formen zu erstellen, wo immer Sie möchten. Obwohl diese Stufe einen Leistungseinbruch verursachen kann, kann sie Dinge tun, die mit den anderen Shader-Stufen schwer zu erreichen sind.
- Der *Fragment-Shader* wird verwendet, um jedes Pixel des Objekts zu färben. Hier können wir Farbtönung, Textur und Beleuchtung auf jeden Teil des Objekts anwenden – aus diesem Grund wird oft die meiste „interessante“ Arbeit im Fragment-Shader durchgeführt.
- Compute-Shader sind eine andere Art von Shader, die außerhalb der Pipeline existieren und für beliebige Berechnungen auf der GPU verwendet werden können. Diese Berechnungen müssen nicht unbedingt etwas mit Grafiken zu tun haben, obwohl Compute-Shader immer noch einen Nutzen in einigen rein grafischen Kontexten haben.

Der Vertex-Shader und der Fragment-Shader sind die beiden wichtigsten Stufen, über die wir die Kontrolle haben, daher werden wir sie in fast jedem Shader sehen, den wir schreiben. Die anderen Stufen sind optional. In Unity werden wir hauptsächlich HLSL (High-Level Shading Language) verwenden, um unseren Shader-Code zu schreiben. Es gibt mehrere Arten von Shading-Sprachen, und die Art und Weise, wie wir Shader in Unity schreiben, hat sich im Laufe der Zeit geändert, daher werden wir dies in Kap. 3 genauer untersuchen, wenn wir unsere allererste Shader-Datei schreiben. Jetzt, da wir wissen, was ein Shader ist, schauen wir uns genauer an, wie die Daten im Rendering-Prozess fließen.

Spieldaten

Am Anfang der Pipeline haben wir eine Menge Daten. Wenn wir Objekte wie Charaktere oder Landschaften in eine Szene in Unity einfügen, fügen wir implizit Daten hinzu, die an die ersten Stufen der Grafikpipeline weitergegeben werden müssen. Hier ist eine kurze Liste der Arten von Daten, die die Grafikpipeline benötigt:

- Die Position und Ausrichtung der *Kamera* bestimmt, wie jedes andere Objekt auf dem Bildschirm erscheinen wird. Einige Objekte werden verdeckt sein, wenn sie hinter der Kamera, außerhalb ihres Sichtfeldes oder zu weit von der Kamera entfernt oder zu nahe an der Kamera sind.
- *Meshs,* oder 3D-Modelle, können durch eine Reihe von *Vertices* (Punkten) definiert werden, die durch *Kanten* verbunden sind, von denen drei ein *Dreieck* bilden. Diese Vertices werden zusammen mit Daten wie Vertex-Farben und *Texturkoordinaten* an den Vertex-Shader übergeben. In Unity ist die Komponente *Mesh Renderer* dafür verantwortlich, diese Daten an den Shader zu übergeben.
- *Sprites* in 2D können als quadratisches *Quad* betrachtet werden, das aus zwei Dreiecken besteht. Die Komponente *Sprite Renderer* von Unity übermittelt die beiden Dreiecke an den Vertex-Shader.
- Objekte auf der *UI* (Benutzeroberfläche), wie Text oder Bilder, verwenden spezialisierte Komponenten, um Daten auf ähnliche Weise an den Shader zu übergeben.

Objekte werden nacheinander durch jede Stufe der Grafikpipeline verarbeitet. Vor der Verarbeitung kann Unity die Daten auf bestimmte Weisen sortieren. Beispielsweise wird eine typische Grafikpipeline zunächst alle undurchsichtigen Objekte rendern und dann halbtransparente Objekte darüber rendern, beginnend mit dem am weitesten von der Kamera entfernten transparenten Objekt (das heißt von hinten nach vorne). Sobald die Vorverarbeitungsphase abgeschlossen ist, ist der nächste Schritt, jedes Objekt nacheinander zu rendern, beginnend mit dem Vertex-Shader.

Der Vertex-Shader

Wie ich bereits erwähnt habe, sind die Renderer-Komponenten von Unity dafür verantwortlich, Daten an den Vertex-Shader zu übergeben. Hier beginnen die Dinge für technische Künstler wie uns spannend zu werden! Die meisten Vertex-Shader sehen gleich aus: Wir nehmen Vertex-Positionen, die relativ zum gerenderten Objekt starten, und verwenden eine Reihe von Matrix-Transformationen, um sie in die richtige Position auf dem Bildschirm zu bringen. Wir können auch einige Effekte auf der Vertex-Stufe implementieren. Zum Beispiel können wir Wellen erzeugen, um eine Wasserfläche zu animieren, oder das Mesh erweitern, um einen Aufblas- oder Explosionseffekt zu erzeugen.

Neben den Positionen transformiert der Vertex-Shader auch andere Vertex-Daten. Meshs können mehrere Informationen an jeden Vertex angehängt haben, wie Vertex-Farben, Texturkoordinaten und andere beliebige Daten, die wir hinzufügen möchten. Der Vertex-Shader führt nicht nur diese Transformationen durch; er übermittelt die Daten auch an nachfolgende Shader-Stufen, sodass wir die Daten nach Belieben ändern können. Zum Beispiel werden wir in einigen der später zu sehenden Shader die Weltposition der Vertices übergeben, damit wir sie im Fragment-Shader verwenden können.

Zwischen der Vertex- und Fragment-Shader-Stufe findet ein Prozess namens *Rasterisierung* statt, bei dem die Dreiecke, die ein Mesh bilden, in ein 2D-Raster von *Fragmenten* geschnitten werden. In den meisten Fällen entspricht ein Fragment einem Pixel auf Ihrem Bildschirm. Denken Sie an den Rasterisierer als eine erweiterte Version von MS Paint, der die Dreiecke des Meshs nimmt und sie in ein Bild der Größe des Spielfensters umwandelt – dieses Bild wird als *Frame Buffer* bezeichnet. Während der Rasterisierung werden andere Eigenschaften zwischen den Vertices interpoliert. Wenn

wir zum Beispiel eine Kante betrachten, an der die beiden Vertices jeweils schwarz und weiß sind, dann werden die neuen Farben der Pixel entlang dieser Kante verschiedene Grautöne sein. Sobald die Rasterisierung abgeschlossen ist, gehen wir zum Fragment-Shader über.

Der Fragment-Shader

Dies wird manchmal als *Pixel-Shader* bezeichnet und ist vielleicht die leistungsfähigste und flexibelste Stufe der grafischen Pipeline. Der Fragment-Shader ist dafür verantwortlich, jedes Pixel auf dem Bildschirm zu färben, sodass wir hier eine Vielzahl von Effekten implementieren können, von Texturen bis hin zu Beleuchtung, Transparenz und so weiter. Spezielle Arten von Shadern, die als Post-Processing-Shader bezeichnet werden und auf den gesamten Bildschirm wirken können, können für zusätzliche Effekte verwendet werden, wie einfache Farbkartierung, Bildschirm-Animationen, tiefenbasierte Effekte und spezielle Arten von Bildschirmraum-Schattierungen.

Sobald der Fragment-Shader fertig ist, findet eine letzte Runde der Verarbeitung statt. Diese Prozesse beinhalten Tiefentests, bei denen undurchsichtige Fragmente möglicherweise verworfen werden, wenn sie sonst hinter einem anderen undurchsichtigen Fragment von einem anderen Objekt gezeichnet würden, und Blending, bei dem die Farben von halbtransparenten Objekten mit Farben gemischt – geblendet – werden, die bereits auf den Bildschirm gezeichnet wurden. Wir können auch eine *Schablone* verwenden, die verhindert, dass bestimmte Pixel auf den Bildschirm gerendert werden, wie Sie in Stufe 6 von Abb. 1-1 sehen können.

Natürlich habe ich viele der Stufen für diese kurze Einführung vereinfacht. Später im Buch werden wir die Vertex- und Fragment-Shader vollständig erkunden und wir werden optionale Arten von Shadern sehen, die für hochspezialisierte Aufgaben konzipiert sind. Das heißt, die meisten der Shader, die Sie im Laufe Ihrer Shader-Karriere schreiben werden, werden sich damit beschäftigen, Vertices zu bewegen und Fragmente zu färben. Bisher haben wir gesehen, wie die Grafikpipeline im Allgemeinen funktioniert, aber es gibt noch ein paar andere Dinge, die wir beachten sollten, bevor wir eintauchen.

Unitys Render-Pipelines

Nun kommen wir zum Elefanten im Raum. Vor 2017 hatte Unity eine einzige Rendering-Pipeline für alle Anwendungsfälle: moderne High-End-PCs und Konsolen, Virtual Reality, Low-End-Mobilgeräte und alles dazwischen. Laut Unity selbst beinhaltete dies Kompromisse, die Leistung für Flexibilität opferten. Darüber hinaus war der Rendering-Code von Unity eine Art „Black Box", die ohne eine Unity-Quellcode-Lizenz unzugänglich war, selbst mit umfassender Dokumentation und einer aktiven Entwicklergemeinschaft.

Unity entschied sich für eine Überarbeitung seines Renderers durch die Einführung von *Scriptable Render Pipelines (SRPs)*. Kurz gesagt, eine SRP gibt Entwicklern die Kontrolle darüber, wie Unity alles rendert, und ermöglicht es uns, Stufen zum Rendering-Loop hinzuzufügen oder daraus zu entfernen, wie es erforderlich ist. In der Erkenntnis, dass nicht alle Entwickler Zeit damit verbringen möchten, einen Renderer von Grund auf neu zu erstellen (tatsächlich ist einer der Hauptgründe, warum viele Menschen eine Spiel-Engine wählen, dass die gesamte Arbeit am Rendering-Code für Sie erledigt ist), bietet Unity zwei Vorlagen-Render-Pipelines an: die *High-Definition Render Pipeline (HDRP)*, die sich an High-End-Konsolen- und PC-Spiele richtet, und die *Universal Render Pipeline*, die für Low-End-Maschinen und Mobilgeräte konzipiert ist, obwohl sie auch auf leistungsfähigerer Hardware laufen kann. Alle SRPs, einschließlich benutzerdefinierter SRPs, die Sie schreiben, und die beiden Vorlagen-Render-Pipelines, bringen exklusive Unterstützung für neue Systeme, die ich im Laufe des Buches erwähnen werde.

Das Legacy-Rendering-System ist auch für diejenigen verfügbar, die ihre Projekte bereits in älteren Unity-Versionen gestartet haben und wird nun als *integrierte Render-Pipeline* bezeichnet. Für die meisten neuen Projekte, die eine breite Palette von Hardware anvisieren, wird empfohlen, ein Projekt mit URP zu starten – schließlich wird Unity dies zum Standard für neue Projekte machen. Leider unterscheiden sich die Shader manchmal leicht zwischen allen drei Pipelines, weshalb ich es für wichtig halte, frühzeitig eine Unterscheidung zwischen ihnen zu treffen. In diesem Buch werde ich mein Bestes tun, um die Unterschiede zwischen den einzelnen Programmen zu erklären und Ihnen Shader-Beispiele zeigen, die in allen drei Programmen funktionieren, soweit dies möglich ist.

Hinweis Obwohl es möglich ist, die Pipelines während der Entwicklung zu wechseln, kann dies, insbesondere bei größeren Projekten, mühsam sein. Wenn Sie bereits ein Projekt gestartet haben, empfehle ich, bei der von Ihnen gewählten Render-Pipeline zu bleiben, es sei denn, es gibt eine Funktion, die nur von einer anderen Pipeline unterstützt wird und die Sie unbedingt benötigen.

Shader Graph

Mit dem Aufkommen der SRPs kamen einige exklusive Funktionen. Für uns ist keine dieser Funktionen so wirkungsvoll wie der Shader Graph, Unitys knotenbasierter Shader-Editor. Traditionell existierten Shader nur als Code, was sie fest auf die „Programmierer"-Seite des „Programmierer-zu-Künstler"-Spektrums setzte. Aber in den letzten zehn Jahren war eine der größten Innovationen im Bereich der technischen Kunst die Umstellung auf visuelle Editoren für Shader. Diese Editoren ähneln visuellen Codierungswerkzeugen, die Codezeilen durch Knoten ersetzen, die Funktionsbündel sind, die zu einem Graphen verbunden werden können. Für viele sind visuelle Editoren wie diese viel einfacher zu verstehen als Code, weil sie den Fortschritt eines Shaders bei jedem Schritt visualisieren können. Im Gegensatz zu Code-Shadern kann ein visueller Editor eine Vorschau darauf geben, wie Ihr Shader bei jedem Knoten aussieht, sodass Sie die Visuals Ihres Spiels mit Leichtigkeit debuggen können.

Hinweis Ursprünglich war Shader Graph exklusiv für SRP-basierte Pipelines. In Unity 2021.2 wurde jedoch die Unterstützung für Shader Graph auf die integrierte Pipeline portiert. Unity scheint es ein bisschen geheim zu halten, da die meiste Online-Dokumentation für Shader Graph es zu vermeiden scheint, dies zu sagen!

In diesem Buch werde ich Ihnen Beispiele sowohl im Shader-Code als auch im Shader Graph zeigen, weil ich glaube, dass beide für technische Künstler in der Zukunft wichtig sein werden. Kap. 3 wird sich auf den Shader-Code konzentrieren, während Kap. 4 Ihre Einführung in Shader Graph sein wird.

Zusammenfassung

Wir haben in diesem Kapitel viel behandelt! Sie sollten nun mit der Schlüsselterminologie vertraut sein, die im Rest dieses Buches verwendet wird. Hier ist eine Zusammenfassung dessen, was wir gelernt haben:

- Die Rendering-/Grafikpipeline ist eine Reihe von Stufen, die auf Daten arbeiten.
- Vertex-Shader werden verwendet, um Objekte auf dem Bildschirm zu positionieren.
- Dreiecksflächen werden während der Rasterisierung in Fragmente/Pixel umgewandelt.
- Daten werden während der Rasterisierungsstufe zwischen den Vertices interpoliert.

KAPITEL 2

Mathematik für die Shader-Entwicklung

Menschen lernen auf unterschiedliche Weise. Während viele Menschen, die dieses Buch lesen, jeden einzelnen Teil der Mathematik im Zusammenhang mit der Shader-Entwicklung lernen möchten, bevor sie mit der Erstellung von Shadern beginnen, werden andere gerne die wichtigen Teile überfliegen und den Rest nach und nach lesen, während sie weitermachen. In diesem Buch habe ich mich dafür entschieden, Ihnen frühzeitig einen umfassenden Einblick in die Shader-Mathematik zu geben mit der Maßgabe, dass Sie das Kapitel überspringen und jederzeit hierher zurückblättern können, wenn Sie es für richtig halten. Im gesamten Buch werde ich Verweise auf den entsprechenden Abschnitt dieses Kapitels geben, wenn ein neues Konzept eingeführt wird, damit Sie die wichtigen Teile nach und nach aufgreifen können.

In diesem Kapitel werde ich Sie in die grundlegende Mathematik einführen, die Sie beim Erstellen von Shadern treffen werden: von Vektoren und Matrizen bis hin zu Trigonometrie und Koordinatenräumen und allem dazwischen.

Vektoren

Vektoren sind ein grundlegender Baustein für Shader – fast alles, was Sie in einem Shader tun, wird irgendwo einen Vektor beinhalten. Was sind sie also? Stellen wir uns vor, wir sind auf einer Schatzsuche und bekommen eine Karte der lokalen Gegend. Wir starten im Basislager in der unteren linken Ecke und stellen dies mit dem Vektor (0, 0) dar, da diese Karte zwei Dimensionen hat. Wenn diese Karte Entfernungen in Meilen misst, sagen wir, es gibt einen extrem großen Baum zwei Meilen östlich und eine Meile

D. Ilett, *Erstellung hochwertiger Shader für Unity®*, https://doi.org/10.1007/979-8-8688-0430-4_2

nördlich, dazu einen felsigen Hügel eine Meile östlich und vier Meilen nördlich und einen sandigen Strand drei Meilen östlich und fünf Meilen nördlich. Wenn wir alles auf unserer Karte eintragen, sieht es ungefähr so aus wie in Abb. 2-1.

Position und Richtungsvektoren

Wir können Vektoren verwenden, um den Versatz zwischen unserem Ausgangspunkt und jedem dieser drei Orte darzustellen – der Vektor zwischen unserem Ausgangspunkt und dem Baum ist (2, 1), denn er liegt 2 mi nach Osten, also in *x-Richtung*, und 1 mi nach Norden, also in *y-Richtung*. Tatsächlich sind Vektoren hervorragend geeignet, um den Versatz zwischen zwei beliebigen Punkten darzustellen, einfach weil das ein Vektor ist: eine Größe, die eine Länge und eine Richtung hat. Das ist die einzeilige Beschreibung in fast jedem Lehrbuch! In diesem Beispiel zeigt die Richtung auf den Baum und die Länge beträgt etwa 2,24 mi. Sie haben vielleicht bemerkt, dass der Vektor, der am felsigen Hügel beginnt und am sandigen Strand endet, ebenfalls (2, 1) ist.

Es gibt bereits einige Dinge zu verstehen. Erstens können wir jede Position durch ihren Versatz von einem bestimmten Ursprungspunkt darstellen, der in 2D (0, 0) ist – deshalb habe ich das bequemerweise als Ausgangspunkt auf unserer Karte gewählt. Ein Vektor, der nur Nullen enthält, ist immer besonders, da er der einzige Vektor mit einer Länge von null und ohne eine bestimmte Richtung ist – beide Eigenschaften werden relevant, wenn wir Operationen mit Vektoren untersuchen. Er hat auch einen besonderen Namen: der *Nullvektor*. Zweitens können Vektoren an jedem Punkt auf der

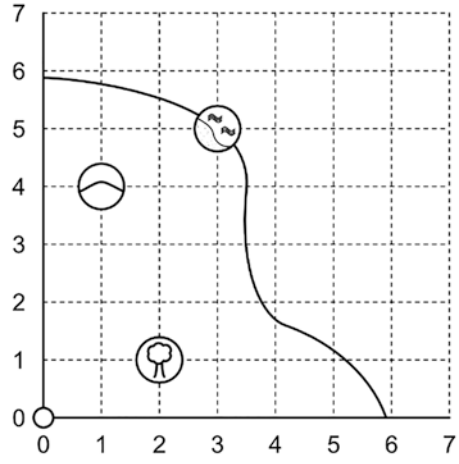

Abb. 2-1. *Eine Karte mit einigen Orientierungspunkten*

Karte beginnen. Sie sind nicht nur nützlich, um uns zu sagen, wo sich ein Punkt in Bezug auf den Ursprungspunkt befindet, sondern sie können uns auch über die Verschiebung zwischen zwei Punkten informieren, die nicht (0, 0) sind. Dies ist wichtig, weil die Aussage „der Vektor (2, 1)“ auf derselben Karte mehrere verschiedene Dinge bedeuten könnte.

Vektoraddition und -subtraktion

Jetzt gehen wir wandern. Von unserem Ausgangspunkt aus erkunden wir zuerst den felsigen Hügel, weil wir denken, dass der Schatz vielleicht auf der Spitze vergraben sein könnte. Glücklicherweise gibt es einen gut ausgetretenen Schmutzpfad, der direkt dorthin führt. Schlechte Nachrichten für uns: Nach einer Stunde graben finden wir nichts. Verdammt. Unsere nächstbeste Vermutung ist, dass der Sandstrand einige Hinweise auf den Standort des Schatzes haben könnte. Also packen wir erschöpft unsere Schaufel und den Metalldetektor ein und wandern zum Strand; diesmal gibt es einen Steinpfad ohne Biegungen, der zum Strand führt. Die Route, die wir genommen haben, ist in Abb. 2-2 zu sehen. Wie weit sind wir jetzt vom Ausgangspunkt entfernt?

Vektoren können leicht durch Hinzufügen jeder der Zahlen im ersten Vektor zu den entsprechenden Zahlen aus dem anderen Vektor addiert werden (obwohl beide Vektoren die gleiche Dimension haben müssen). Jeder der Werte in einem Vektor wird als *Komponente* bezeichnet, und sie sind wie die Achsen eines Diagramms benannt: Die erste ist die *x*-Komponente, dann *y*, dann in höheren Dimensionen *z* und dann *w*. Das

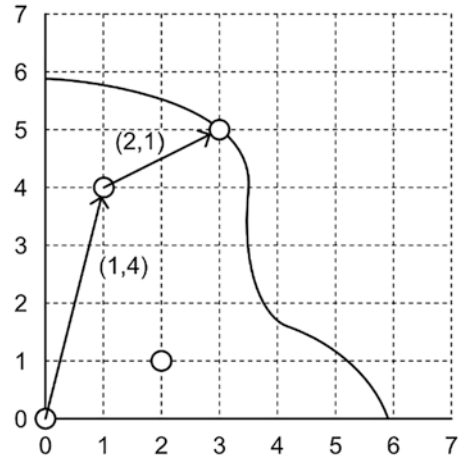

Abb. 2-2. *Die erste Etappe der Reise*