Informatik
Hauptcampus

HOCHSCHULE TRIER

Konzeption und Umsetzung von zweidimensionalen Beleuchtungseffekten in der Unity Engine

Conception and Implementation of 2D Illuminationeffects in Unity

Oliver Biwer

Master-Abschlussarbeit

Betreuer: Prof. Dr.-Ing. Christoph Lürig

Trier, 01.06.2019

Kurzfassung

In der heutigen Zeit gewinnen Echtzeit-Beleuchtungsverfahren immer mehr Aufmerksamkeit. Erst im August 2018 hat Nvidia Grafikkarten mit Hardware Einheiten zur Beschleunigung von Raytracing vorgestellt. Im Bereich des Machine-Learning entstehen Verfahren, um mittels Raytracing gerenderte Bilder von Bildrauschen zu befreien. Raytracing und viele andere Verfahren basieren auf physikalischen Gesetzgebungen (oder deren Approximation) und Spiele Engines wie Unreal oder Unity verwenden Physically Based Materials, um eine photorealistische Optik zu erzielen. Dieses realitätsgetreue Vorgehen ist in 2D Spielen durch die fehlende Dimension nicht zwingend möglich.

Doch wie sieht es bei 2D Spielen aus? Die meisten 2D Spiele berechnen Schatten und Licht nicht in Echtzeit sondern verwenden vorgefertigte Grafiken um den Eindruck einer Lichtquelle zu schaffen. Diese Arbeit befasst sich mit dynamischer Licht- und Schattenberechnung in 2D Spielen. Dazu werden bekannte 3D Techniken und ihre Umsetzbarkeit und Leistung in 2D Spielen betrachtet. Die Umsetzung erfolgt in der Unity Engine dabei werden verschiedene Integrationsmöglichkeiten der Verfahren in der Unity Engine betrachtet.

Inhaltsverzeichnis

1

Einleitung

Diese Arbeit befasst sich mit Beleuchtungseffekten in 2D Videospielen, mit einem Schwerpunkt auf dynamisch berechneten Beleuchtungseffekten. Dabei werden Methoden aus dem Bereich des 3D-Renderings betrachtet und eine Umsetzung in einer 2D Welt analysiert.

Die Unity Engine ist primär für 3D Spiele ausgelegt, dennoch bietet sie die Möglichkeit 2D Spiele zu erstellen. Dazu übernimmt die Engine vor allem Aufgaben wie 2D Physik und das Rendern von Sprites. Lichtquellen in Unity sind jedoch ausschließlich 3D Objekten vorbehalten. Dieses Defizit hat dafür gesorgt, dass es einige Erweiterungen im Unity AssetStore gibt, die sich mit dieser Thematik beschäftigen [APP] [Huk] [FUN]. Außerdem wurde seitens Unity-Technologies im November 2018 angekündigt, Unterstützung für 2D Lichter und Schatten in den Unity Editor zu integrieren [Tec].

Der Quellcode des Projektes ist hier einzusehen:
https://github.com/soraphis/Unity-2D-Lighting

Teil I

Grundlagen

2

2D Spiele

Im Gegensatz zu 3D Spielen, welche meist eine realistische grafische Darstellung haben, stellen 2D Spiele immer eine Abstraktion dar. Typischerweise fallen 2D Spiele entweder in die Kategorie der Sidescroller oder der Top-Down Spiele.

Bei Sidescrollern betrachten wir einen Querschnitt der Welt. Oft bestehen solche Spiele aus mehreren "Layern" (Schichten), um in einem gewissen Maße eine dritte Dimension zu simulieren.

Top-Down Spiele unterteilen sich in Spiele mit 90° Winkel und angeschrägter Perspektive. Bei 2D Top-Down Spielen mit angeschrägter Perspektive ist eine reine 2D Beleuchtung nicht ohne Weiteres möglich, da Grafiken in diesen Spielen mehr als zwei Dimensionen darstellen. Daher werden die in dieser Arbeit betrachteten Methoden ausschließlich auf Sidescroller und reine Top-Down Spiele betrachtet.

2.1 Beleuchtung in 2D Spielen

Die meisten 2D Videospiele verzichten auf dynamische Beleuchtungseffekte. Wenn überhaupt Licht und Schatten visualisiert werden, so sind dies meistens elliptische, halb-transparente, dunkle Grafiken unter den Spielfiguren, um eine Art Schatten darzustellen oder halb-transparente, helle Grafiken, um Licht darzustellen (Beide Effekte in Abb. 2.1 zu sehen).

In einigen Fällen sind diese Grafiken nicht vollständig statisch, sondern animiert. Eine dynamische Berechnung erfolgt jedoch nicht.

Das Spiel Dead Cells (Figure 2.2) erzielt seine Ästhetik aus einer Mischung von unterschiedlichen Beleuchtungseffekten. Charaktere in der Welt haben ihre Beleuchtung in die Sprites integriert und sind daher nicht dynamisch beleuchtet. Von der Spielfigur geht ein dynamisch berechnetes Punktlicht aus, welches von der statischen Weltgeometerie absorbiert wird. Dadurch entstehen Lichtkegel. Lichtquellen im Level verwenden die oben beschriebenen statischen/animierten halb-transparenten Grafiken. Lichtquellen im Levelhintergrund verwenden eine 3D Lichtquelle, um halb-transparente Dunst/Staub Grafiken im Levelvordergrund zu beleuchten.

Abb. 2.1. Hyperlight Drifter. Statische Grafiken für Schatten unter den Charakteren und Lichteinfall. *quelle:* `https://heartmachine.com/hyper-light`

2.2 Ähnliche Veröffentlichungen

Efficient Computation of Visibility Polygons [BHH+14] beschäftigt sich mit verschiedenen Algorithmen, um Sichtbarkeitspolygone zu bilden.
Der Web-Blog Eintrag von Red Blob Games "*2d Visibility*" [red12] beschäftigt sich ebenfalls mit dieser Thematik.

Programme wie *SpriteIlluminator*, *Sprite-DLight* und *SpriteLamp* sind darauf spezialisiert, *normal maps* zu 2D Grafiken zu generieren.

- SpriteIlluminator: `https://www.codeandweb.com/spriteilluminator`
- Sprite-DLight: `http://www.2deegameart.com/p/sprite-dlight.html`
- SpriteLamp: `http://www.snakehillgames.com/spritelamp/`

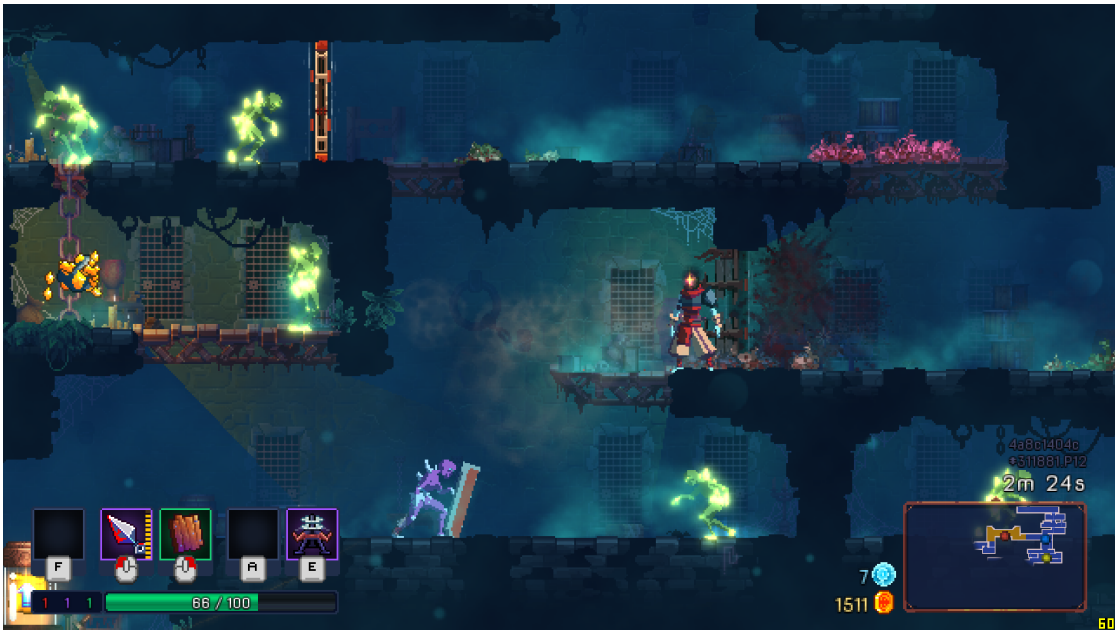

Abb. 2.2. Dead Cells. verschiedene Licht/Schatten effekte. *quelle:* *https://www.mobygames.com/game/windows/dead-cells*

3

Unity Engine

Die Unity Spieleengine zeichnet sich vor allem durch ihre Erweiterbarkeit aus. Vor allem die Implementierung der *Scriptable Render Pipeline*, welche im zweiten Quartal 2019 die Preview-Phase verlassen hat, ermöglicht es eigene Lichtquellen zu definieren, während es zuvor nur möglich war, dass 2D Grafiken auf 3D Lichter reagieren.

3.1 Grundlagen

Eine Szene in Unity entspricht einer Sammlung von *GameObject*s. Ein GameObject entspricht dabei einer Entity in klassischen Entity-Component-Frameworks. Das GameObject ist ein fundamentales Objekt in Unity, welches zwar aus sich heraus nur wenig Funktionalität liefert, aber als Container für Komponenten dient. [unic]

Als Komponenten in Unity können alle C# Klassen verwendet werden, die von der Klasse `MonoBehavour` erben. Durch diese Komponenten wird das Spiel programmiert.

3.2 CommandBuffer

Die *Camera* Komponente, die von der Unity Engine zur Verfügung gestellt wird, bietet die Möglichkeit so genannte *CommandBuffer* zu registrieren. Ein CommandBuffer enthält eine Liste von Rendering Kommandos, die an verschiedenen Punkten des Renderns ausgeführt werden können. [unia]

3.3 Scriptable Render Pipeline

Render Pipeline bezeichnet die Abfolge von Schritten, die zum Rendern der Szene nötig sind. Die Unity Engine besitzt eine 'built in' Pipeline, welche sich um Dinge wie Frustum Culling, Occlusion Culling, Forward oder Deferred Rendering und Post Processing kümmert. Diese eingebaute Pipeline funktioniert wie eine

'Black Box' und ist dafür ausgelegt, für möglichst viele Spiele und Plattformen zu funktionieren.[unib]

Bei der *Scriptable Render Pipeline* handelt es sich um die Möglichkeit diese Rendering Schritte anzupassen, zu erweitern, zu entfernen oder auszutauschen. Dies ermöglicht es, eigene Lichtquellen zu definieren, Schattenberechnungen anzupassen und grundsätzlich das Rendering bezüglich des Spiels zu optimieren.[unid]

Teil II

Implementierung

4

Übergreifende Funktionalität

4.1 Geometry Collector

Der `GeometryCollector` ist eine static Helferklasse, die Methoden bereitstellt um statische und dynamische Grafiken, welche Schatten werfen sollen, zu einem Mesh hinzuzufügen.

Sprites in Unity verfügen über ein *Physics Shape*, welches manuell im Unity Editor angepasst werden kann. Der `GeometryCollector` nutzt dieses *Physics Shape*, um die lichtblockierenden Kanten eines Sprites zu erfassen. Außerdem unterstützt der `GeometryCollector` Unity *Tilemaps*, in dem die Tilemap einzelne Tiles zu einem *CompositeCollider* zusammenfasst.

Da Unity keine integrierte Möglichkeit bietet, beim Erzeugen neuer oder Löschen vorhandener Objekte benachrichtigt zu werden, muss dieses Mesh in jedem Frame neu aufgebaut werden. Unity bietet für interne Optimierungen die Möglichkeit unbewegte Objekte als *static* zu markieren. Dieselbe Optimierung wird beim `GeometryCollector` genutzt.

Die `CollectStatic` Methode des `GeometryCollector` iteriert über alle *GameObject*s, die eine Renderer Komponente besitzen und bei denen das *static*-Feld auf `true` gesetzt ist. Diese Iteration ist vergleichsweise teuer, da sie über jede Komponente iteriert, die sich auf einem beliebigen *GameObject* befindet. Diese Methode sollte nur beim Laden einer neuen Szene verwendet werden. Eine mögliche Modifizierung wäre es, diese Methode nur im Editor zur Verfügung zu stellen und das sich daraus ergebene Mesh in der Szene abzuspeichern. Dadurch würden die Kosten von CPU-Zeit beim Laden einer neuen Szene verringert.

Die `CollectDynamic` Methode des `GeometryCollector` nutzt die `Runtime2DShadowcaster`-Komponente. Diese verfügt über eine static Liste aller *GameObject*s, die diese Komponente nutzen. Dadurch wird das Iterieren über

unnötige *GameObject*s (also solche, die z.B. keinen *SpriteRenderer* besitzen) verhindert. Diese Methode ist dazu gedacht in jedem Frame verwendet zu werden, um bewegliche Schattenwerfer zu ermitteln.

Die Methoden des `GeometryCollector` sind darauf optimiert keine Heap-Memory-Allocations zu erzeugen.

5

Meshgeneration

Diese Methode basiert darauf, ausgehend von der Lichtquelle Lichtstrahlen auszusenden. Die Endpunkte dieser Lichtstrahlen werden dann zu einem Mesh verbunden.

5.1 Implementierung

Zu Beginn wird mit Hilfe des `GeometryCollector`s ein Mesh in einer static Variable erzeugt, das alle Lichtquellen lesend verwenden. Jede Lichtquelle baut nun ein eigenes Mesh auf, welches die beleuchtete Fläche darstellt. Die Generierung dieses Meshs gliedert sich in 3 Schritte.

Zuerst werden alle Kanten, die der GeometryCollector bereitstellt, gefiltert. Dabei werden alle Kanten aussortiert, die außerhalb der Reichweite der Lichtquelle liegen.

Im zweiten Schritt werden die Lichtstrahlen ausgesendet. Zunächst werden kreisförmig in regelmäßigen Abständen ein paar Strahlen ausgesendet, um die Lichtquelle kreisförmig darzustellen, auch wenn keine Geometrie in ihrer Reichweite ist (siehe Abbildung 5.1 links).

Danach werden für alle gefundenen Kanten Lichtstrahlen zu allen Eckpunkten innerhalb der Lichtreichweite gesendet. (siehe Abbildung 5.1 rechts)

Nun werden für jede Ecke zwei weitere Strahlen im und gegen den Uhrzeigersinn sowie um ein paar Grad versetzt, ausgesendet. Dabei wird einer dieser Strahlen auf die Kante treffen und der andere wird die Kante verfehlen. Dadurch ergibt sich eine gerade Kante dort, wo die Geometrie Schatten wirft. (siehe Abbildung 5.2 links)

In einem letzten Schritt, um die Lichtquelle gleichmäßig rund zu machen, werden nun weitere Strahlen ausgesendet. Dazu werden auf jeder Kante Punkte gesucht, die auf dem Kreis um die Lichtquelle liegen. Der Radius entspricht hierbei der Reichweite des Lichtes. (siehe Abbildung 5.2 rechts)

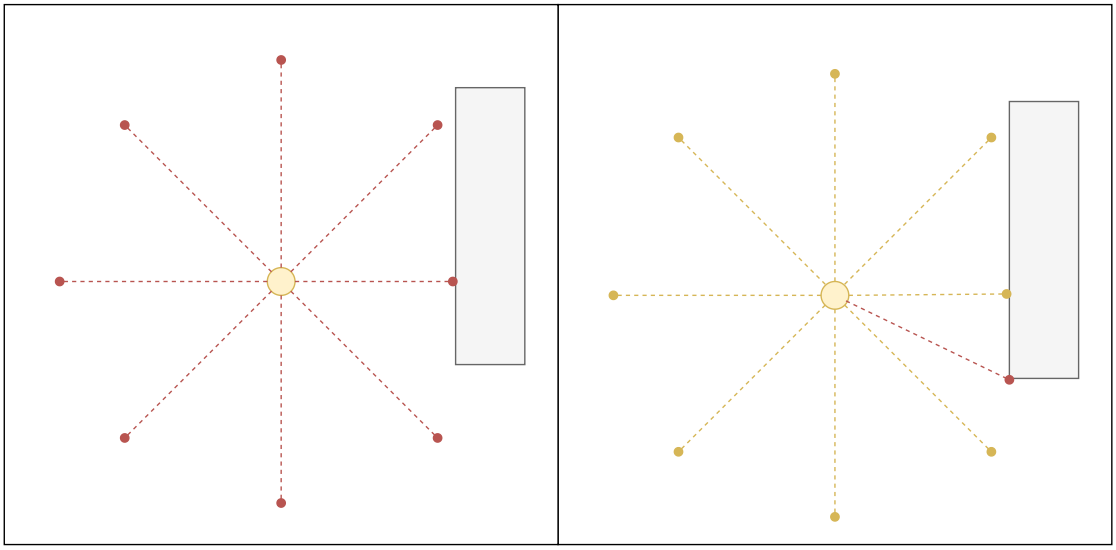

Abb. 5.1. links: In regelmäßigen Abständen werden Strahlen ausgesendet. **rechts**: Es werden Strahlen zu allen Ecken innerhalb der Reichweite gesendet.

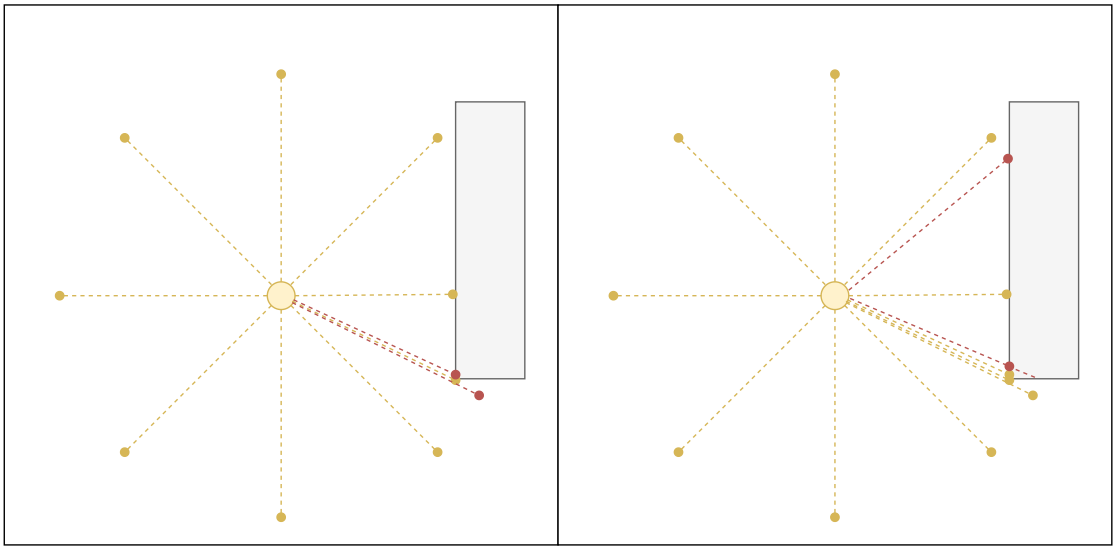

Abb. 5.2. rechts: Die roten, gestrichelten Linien verbinden die Lichtquelle mit Punkten auf den Kanten der Geometrie, die genau um die Reichweite des Lichtes von der Lichtquelle entfernt sind.

Im dritten und letzten Schritt, um das Mesh der Lichtquelle zu generieren, werden alle Endpunkte der Lichtstrahlen nach ihrem Winkel (von -180° bis +180°) sortiert. Anhand dieser Reihenfolge wird das Mesh aus Dreiecken zusammengebaut, in dem je zwei benachbarte Punkte mit dem zentralen Punkt der Lichtquelle selbst verbunden wird.

Ein großer Vorteil dieser Technik ist es, dass das generierte Mesh ebenfalls zur Kollisionsabfrage genutzt werden kann. Dies kann in Spielen notwendig sein, wenn überprüft werden soll, ob sich eine Spielfigur im Licht befindet.

#Lichtquellen	Average FPS
0	240
1	240
4	200
8	140
12	110
16	90
20	76
24	65
28	60

Diese Technik könnte man dadurch verbessern, dass man die Lichtmeshs parallel aufbaut. Eventuell ist auch ein Geometry Shader möglich.

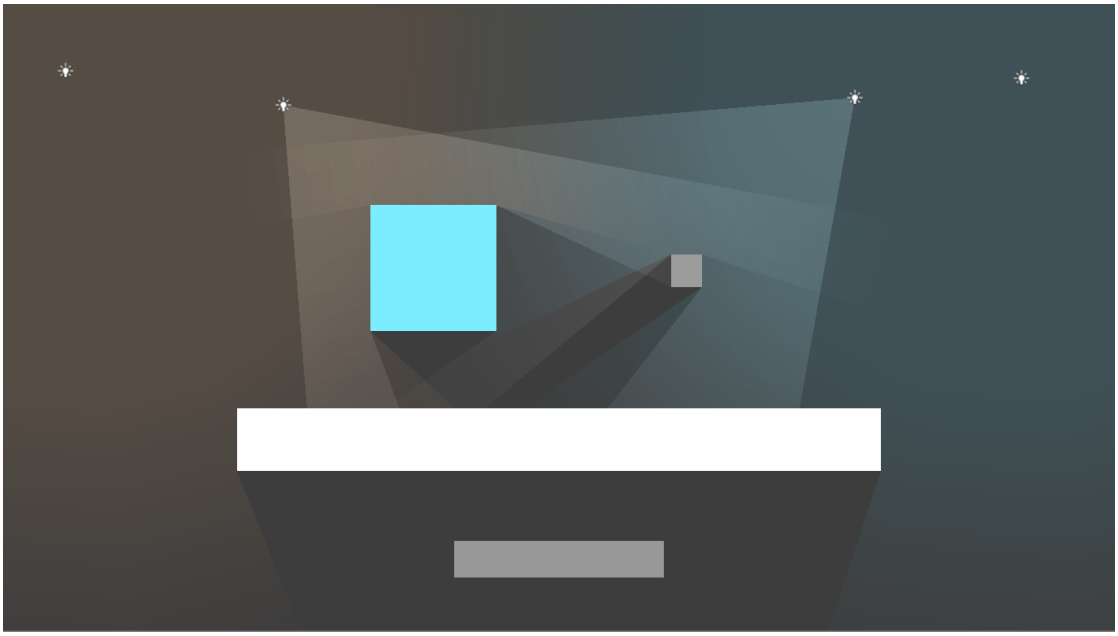

Abb. 5.3. 4 Lichtquellen in einer Testszene

6

Shadow Mapping

Beim Shadowmapping handelt es sich um eine vergleichsweise alte Technik. Sie wurde 1978 in 'Casting Curved Shadows on Curved Surfaces'[Wil78] vorgestellt. Die darin beschriebene Technik ist vergleichsweise einfach:

In einem ersten Schritt wird die Szene aus Sicht der Lichtquelle gerendert, dabei werden die Tiefeninformationen in einer Textur gespeichert, der Shadowmap. In einem zweiten Schritt wird danach erst die Szene aus sich der Kamera gerendert, dabei werden die Informationen der Shadowmap genutzt um zu bestimmen, ob ein Pixel von einer Lichtquelle zu sehen ist (kein Schatten), oder nicht (Schatten). In der 3D Grafik wird für jede Lichtquelle eine 2D Textur erstellt. Da in 2D Spielen die Z-Achse jedoch keine Rolle spielt, ist es ausreichend eine 1D Textur zu erstellen. Diese 1D Textur speichert die Tiefeninformationen in Polarkoordinaten. Das heißt, die X-Position auf der Shadowmap gibt einen Winkel um die Lichtquelle an und der Farbwert an dieser Position die Entfernung zur Lichtquelle. Für jede 2D Lichtquelle genügt also eine ein Pixel hohe Textur oder etwas performanter: Jede ein Pixel hohe Reihe in der Textur steht für eine andere Lichtquelle. Das heißt mit einer 256 Pixel hohen Textur könnten für 256 Lichtquellen Shadowmaps generiert werden.

Mit Hilfe des in Kapitel 4.1 beschriebenen *GeometryCollector*s wird jede Kante der Levelgeometrie für jede Lichtquelle gerendert. Dadurch ergibt sich allerdings ein Problem für Kanten, deren Endpunkte einerseits im Bereich 0..90 Grad und andererseits im Bereich 270..360 Grad liegen. Der Lösungsansatz ist, dass die Shadowmap Polarkoordinaten von 0 bis 540 (statt 360) Grad enthält, da eine Kante höchstens 180 Grad in Polarkoordinaten abdecken kann. Der Nachteil dieses Lösungsansatzes liegt darin, dass der Bereich von 0..180 und von 360..540 denselben Bereich der Weltgeometrie adressiert. [War] Deshalb wird in einem zweiten Schritt aus dieser Shadowmap eine finale Shadowmap von 0 bis 360 Grad generiert.

Bei der Verwendung dieser Technik sollte die Shadowmap in der Horizontalen mindestens 512 Pixel fassen, denn durch die Verwendung von Polarkoordinaten entspricht jeder Pixel in der Horizontalen der Shadomap einem Lichtkegel. Das bedeutet, dass mit zunehmender Reichweite die Breite eines Kegels stark zunimmt. Dies hat zur Folge, dass Licht an geraden Kanten in Zacken abschließt, dieser Effekt ist in Figure 6.1 zu sehen. Je größer die Shadowmap ist, desto kleiner sind

die Lichtkegel und im Idealfall erreicht man so, dass die Zacken kleiner als 1/2 Pixel werden.

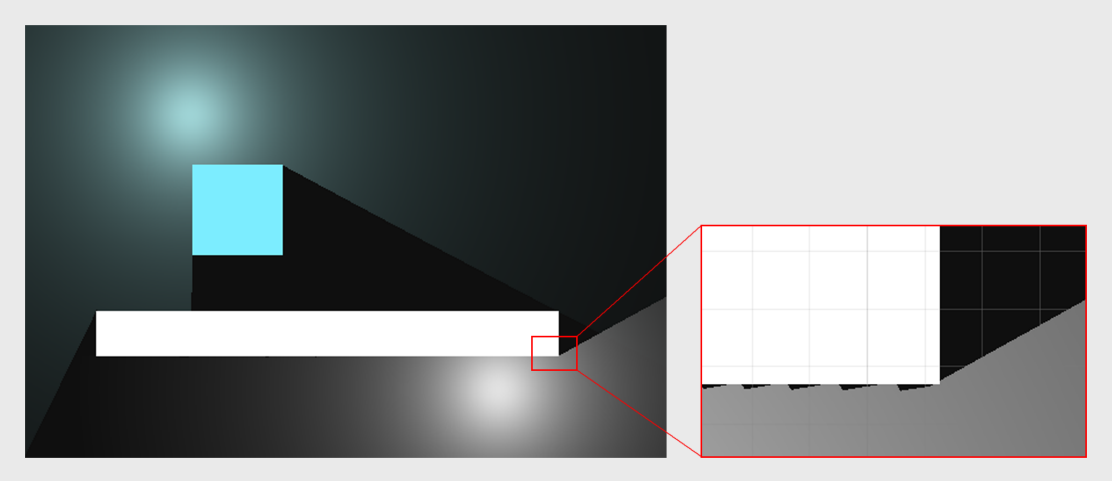

Abb. 6.1. Lichtstrahlen bilden Zacken an der Geometrie.

6.1 Implementierung: Pseudo-Shadowmapping

Da es in Unity nicht ohne Weiteres möglich ist, eigene Lichtquellen zu definieren, wird in dieser Implementierung eine Textur erzeugt, die durch ein Alpha-Blending über die 2D Szene gelegt wird.

Kern der Implementierung ist die `LightLayer`-Komponente. Diese wird dem Kamera GameObject hinzugefügt.
Das ist notwendig, weil die `LightLayer`-Komponente CommandBuffer registriert, um die Shadowmap zu generieren.

#Lichtquellen	Average FPS
0	245
1	240
4	240
8	240
12	240
16	230
20	195
24	160
28	140

Getestet wurde mit einer horizontalen Shadowmap Auflösung von 2048 Pixel.

6.2 Implementierung: SRP 2D Shadowmapping

Diese Implementierung ist erst durch neuere Versionen der Unity Engine und die Entwicklung der *Scriptable Render Pipeline* möglich. Dadurch können nun eigene Lichtquellen definiert werden. Dies hat zwei Vorteile: Zum einen wird die `LightLayer`-Komponente überflüssig, diese Aufgabe übernimmt die Render Pipeline.

Dies ist ohne die Scriptable Render Pipeline nur bedingt möglich, bisherige Lösungen verwenden 3D Lichtquellen, um 2D Grafiken zu beleuchten.

Mit Unity 2019.2 wird es möglich sein, Sprites mehrere Texturen zu hinterlegen. Dadurch wird es möglich *normal maps* einfacher zu referenzieren, wodurch diese Technik so erweitert werden kann, dass mit minimalem Aufwand 2D Grafiken plastisch wirken.

Zur Zeit können nur 16 Lichtquellen gleichzeitig gerendert werden. Diese Limitierung liegt daran, dass die Grafik API die Größe der Arrays, die die Shader entgegennehmen, auf 16 limitiert. Dieses Limit kann dadurch umgangen werden, dass Objekte basierend auf den betreffenden Lichtquellen gruppiert und gemeinsam gerendert werden. Dann liegt das Limit bei 16 Lichtquellen pro Objekt.

#Lichtquellen	Average FPS
0	245
1	245
4	240
8	230
12	160
16	130

Getestet wurde mit einer horizontalen Shadowmap Auflösung von 2048 Pixel.

Teil III

Ausblick

7

Dynamic Realtime Lighting

7.1 Voxel Floodfill

Floodfill Algorithmen kommen zumeist in Sandbox Spielen vor (z.B. Minecraft, Terraria, Starbound). Diese Spiele sind meist an einem Gitter orientiert. Befindet sich in einer Zelle eine Lichtquelle, wird der Lichtwert in dieser Zelle auf einen Wert größer 0 gesetzt. Alle benachbarten Zellen erhalten dann rekursiv diesen Wert um 1 dekrementiert (sofern dieser Wert höher ist als der aktuelle Wert der Zelle). [Arn]

Eine *Screen Space* Implementierung dieses Verfahrens ist ebenfalls möglich. [AHT04]

7.2 Shadow Volumes

Shadow Volumes ist eine Technik, die durch das Spiel DOOM 3 bekannt wurde.

In 3D Spielen wird für diese Technik der *Stencil Buffer* verwendet wird, um zu entscheiden ob ein Objekt in einem Schattenkegel, davor oder dahinter liegt. Um diese Technik auf 2D Spiele anzuwenden, kann dieses Vorgehen vereinfacht werden, da es keine Punkte gibt, die durch einen Schattenkegel hindurch betrachtet werden können.

Der Ablauf kann wie folgt aufgebaut werden:

Für jede Lichtquelle:

- Stencil Buffer auf 0 setzen.
- *Stencil Buffer* auf 1 setzen, falls sich der Punkt in Reichweite und Blickrichtung der Lichtquelle befindet.
- Rendern des Meshs aus dem Geometry Collector:
  - Mit dem *Geometry Shader* für jede Kante ein Quad erzeugen, welches dem Schattenvolumen der Kante bezüglich der Lichtquelle entspricht.
  - Stencil Buffer für alle Punkte innerhalb des Quads auf 0 setzen
- Rendern der Szene mit *Alpha Blending* für alle Punkte, an denen der Stencil Buffer nicht 0 ist.

7.3 Realtime Radiosity

> The radiosity algorithm is a method for evaluating light intensity at discrete points of ideal diffuse surfaces in a closed environment
>
> "Radiosity ist ein Algorithmus um die Lichtintensität an diskreten Punkten auf ideal-diffusen Oberflächen in geschlossenen Umgebungen zu ermitteln." [DP09]

Für 3D Spiele ist es ungeeignet in Echtzeit berechnet zu werden, da Radiosity ein iteratives und aufwendiges Verfahren ist. In 2D Spielen ist dies aufgrund der fehlenden Dimension allerdings möglich.

8

Vorberechnete Verfahren

Während sich diese Arbeit ausschließlich auf Echtzeitverfahren konzentriert, existieren Beleuchtungstechniken die auf 2D Spiele angewendet werden können.

8.1 Lightmapping

Beim Lightmapping werden statische Lichtquellen und statische Geometrie betrachtet. In 3D Spielen werden dann für jede statische Geometrie Lightmaps erzeugt. Dies sind Texturen, welche mit der Diffusemap (oder Albedomap) multipliziert werden.

Der Vorteil dieser Technik ist es, dass das Licht mit allen Reflexionen und Brechungen vorberechnet werden kann. Dadurch ist eine indirekte Beleuchtung und *Color Bleeding* möglich, welche in den in Part II beschriebenen Verfahren nicht behandelt wurde.

Für gewöhnlich wird der Radiosity Algorithmus verwendet, um diese Lichtinformationen zu generieren.

Literaturverzeichnis

AHT04. Arvo, Jukka, Mika Hirvikorpi und Joonas Tyystjärvi: *Approximate Soft Shadows with an Image-Space Flood-Fill Algorithm*. Comput. Graph. Forum, 23:271–280, 2004.

APP. APPTOUCH: *2DDL Pro : 2D Dynamic Lights and Shadows*. https://assetstore.unity.com/packages/tools/particles-effects/2ddl-pro-2d-dynamic-lights-and-shadows-25933.

Arn. Arnold, Benjamin: *Fast Flood Fill Lighting in a Blocky Voxel Game*. https://www.seedofandromeda.com/blogs/29-fast-flood-fill-lighting-in-a-blocky-voxel-game-pt-1.

BHH+14. Bungiu, Francisc, Michael Hemmer, John Hershberger, Kan Huang und Alexander Kröller: *Efficient Computation of Visibility Polygons*. CoRR, abs/1403.3905, 2014.

DP09. Dimitri Plemenos, Georgios Miaoulis: *Visual Complexity and Intelligent Computer Graphics Techniques Enhancements*. Springer, 2009.

FUN. FUNKYCODE: *Smart Lighting 2D*. https://assetstore.unity.com/packages/tools/particles-effects/smart-lighting-2d-112535.

Huk. Hukha: *Lumbra : 2D Dynamic Lights and Field of view* . https://assetstore.unity.com/packages/tools/particles-effects/lumbra-2d-dynamic-lights-and-field-of-view-128759.

red12. *2D Visibility*, März 2012. https://www.redblobgames.com/articles/visibility/.

Tec. Technologies, Unity: *Unite Los Angeles 2018 - Unity 2019 Roadmap*. Slide 90 https://www.slideshare.net/unity3d/unite-los-angeles-2018-unity-2019-rd-roadmap/90.

unia. *Unity ScriptReference: CommandBuffer*. https://docs.unity3d.com/ScriptReference/Rendering.CommandBuffer.html.

unib. *Scriptable render pipeline*. https://docs.unity3d.com/2017.4/Documentation/Manual/ScriptableRenderPipeline.html.

unic. *Unity Manual: GameObject*. https://docs.unity3d.com/Manual/class-GameObject.html.

unid. *What is a Scriptable Render Pipeline.* https://github.com/Unity-Technologies/ScriptableRenderPipeline/wiki/What-is-a-Scriptable-Render-Pipeline.

War. Ware, Rob: *Fast 2D shadows in Unity using 1D shadow mapping.* https://www.gamasutra.com/blogs/RobWare/20180226/313491/Fast_2D_shadows_in_Unity_using_1D_shadow_mapping.php.

Wil78. Williams, Lance: *Casting Curved Shadows on Curved Surfaces.* SIGGRAPH Comput. Graph., August 1978.

A

Glossar

Rendern	Erzeugung eines Bildes aus Rohdaten.
Sprite	2D Grafik, die von der Grafikhardware über das Hintergrundbild eingeblendet wird
Mesh	Gitternetz eines 3D Models
Unity-Technologies	Entwicklerstudio; Entwickelt die Unity Engine
Unity	Spiele Engine
Engine	Grundgerüst für Videospiele
AssetStore	Virtueller Marktplatz für zusätzliche Inhalte für die Unity Engine
Ambient Light	Allgegenwärtiges Licht; Eine Art minimale Beleuchtung die überall existiert

B

Erklärung der Kandidatin / des Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

Datum	Unterschrift der Kandidatin / des Kandidaten