ARM® Unity 開発者ガイド

バージョン 4.2

モバイル ゲームのグラフィックスの最適化

arm

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
100140_0402_01_ja

ARM® Unity 開発者ガイド

モバイル ゲームのグラフィックスの最適化

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.

リリース情報

ドキュメント履歴

発行	日付	機密保持ステータス	変更点
0100-00	05 9 月 2014	非機密扱い	バージョン 1.0 初版
0200-00	23 6 月 2015	非機密扱い	バージョン 2.0 初版
0201-00	28 7 月 2015	非機密扱い	バージョン 2.1 初版
0300-00	18 9 月 2015	非機密扱い	バージョン 3.0 初版
0300-01	05 11 月 2015	非機密扱い	バージョン 3.0 第 2 版
0301-00	07 4 月 2016	非機密扱い	バージョン 3.1 初版
0301-01	25 4 月 2016	非機密扱い	バージョン 3.1 第 2 版
0302-00	31 5 月 2016	非機密扱い	バージョン 3.2 の初回リリース
0303-00	17 3 月 2017	非機密扱い	バージョン 3.3 の初回リリース
0303-01	01 6 月 2017	非機密扱い	バージョン 3.3 第 2 版
0400-00	19 9 月 2019	非機密扱い	バージョン 4.0 初版
0401-00	28 2 月 2020	非機密扱い	バージョン 4.1 初版
0402-00	14 5 月 2020	非機密扱い	バージョン 4.2 初版
0402-01	17 3 月 2021	非機密扱い	バージョン 4.2 第 2 版

関連参照

Non-Confidential Proprietary Notice(2 ページ)
機密保持ステータス(4 ページ)
製品ステータス(4 ページ)
Web アドレス(4 ページ)

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at *http://www.arm.com/company/policies/trademarks*.

Copyright © 2014–2017, 2019–2021 Arm Limited (or its affiliates).All rights reserved.

Arm Limited.Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

非機密著作権情報

本書は著作権などの権利により保護されており、本書に含まれる手順または実装に関する情報は、1 つ以上の特許または申請中の特許により保護されている可能性があります。本書のいかなる部分も、Arm から事前に書面による明示的な承諾なく、何らかの形式や方法で無断複製することは許可されていません。**特に記載がない限り、明示的であるか黙示的であるかを問わず、また禁反言やその他いかなる知的財産権のライセンスを許諾するものではありません。**

本書の情報には、実装により、いかなる第三者の特許も侵害されないことを確認する目的のために情報を使用せず、第三者にもそれを許可しないと承諾することを条件としてアクセスすることができます。

本書は、「現状」のまま提供されます。Arm は、明示的、黙示的、または制定法上のいずれを問わず、いかなる表明も保証も行いません。これには、本書に関連した商品性、品質基準、非侵害、または特定目的への適合性に関する黙示的保証を含むが、これに限定されません。疑義を避けるため、Arm は第三者の特許、著作権、営業機密、または他の権利の範囲および内容に関して、いかなる表明も行わず、識別や理解のための分析も行いません。

本書には、技術的に不正確な箇所および誤記が含まれる場合があります。

法により禁止されていない限りにおいて、Arm は本書の使用により生じた直接的、間接的、特別、付随的、懲罰的、または結果的損害などを含むすべての損害に対して、たとえそのような損害の可能性が事前に告知されていた場合でも、その原因および責任理論の如何に関わらず一切の責任を負わないものとします。

本書には、商品のみが含まれています。本書の使用、複製、または開示が関連するあらゆる輸出法および輸出規制に完全に準拠し、本書が全体であれ一部であれ、該当する輸出法に違反して直接的または間接的に輸出されることがないことを保証する責任を負うものとします。ARM のお客様に関連して「パートナー」という言葉が使用されている場合でも、他会社と提携関係を設立することや、言及することを意図するものではありません。Arm は、通知することなくいつでも本書を変更することができます。

本契約のいずれかの規定と、Arm と締結された本書の内容を含むクリックスルー契約または署名済みの書面契約の間に矛盾がある場合、クリックスルー契約または署名済みの書面契約を本契約の規定より優先するものとします。本書は、便宜上、他言語に翻訳される場合がありますが、本書の英語版と翻訳との間に矛盾がある場合、契約書の英語版に含まれる規定を優先することに同意するものとします。

Arm のロゴおよび ® または ™ のマークが付いた語は、Arm Limited (またはその子会社) の米国およびその他の国における登録商標または商標です。All rights reserved. 本書に記載されている他の製品名は、各社の所有する商標です。Arm の 商標の使用に関する次のガイドラインに従ってください。*http://www.arm.com/company/policies/trademarks*

Copyright © 2014–2017, 2019–2021, Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

関連参照

リリース情報(2 ページ)
機密保持ステータス(4 ページ)
製品ステータス(4 ページ)
Web アドレス(4 ページ)

機密保持ステータス

本書は非機密扱いであり、本書を使用、複製、および開示する権利は、ARM および ARM が本書を提供した当事者との間で締結した契約の条項に基づいたライセンスの制限により異なります。

無制限アクセスは、ARM 社内による分類です。

関連参照

リリース情報(2 ページ)
Non-Confidential Proprietary Notice(2 ページ)
製品ステータス(4 ページ)
Web アドレス(4 ページ)

製品ステータス

本書の情報は最終版であり、開発済み製品に対応しています。

関連参照

リリース情報(2 ページ)
Non-Confidential Proprietary Notice(2 ページ)
機密保持ステータス(4 ページ)
Web アドレス(4 ページ)

Web アドレス

developer.arm.com

関連参照

リリース情報(2 ページ)
Non-Confidential Proprietary Notice(2 ページ)
機密保持ステータス(4 ページ)
製品ステータス(4 ページ)

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

目次
ARM® Unity 開発者ガイド モバイル ゲームのグラフィックスの最適化

Preface

本書について 9
ご意見、ご感想 11

第 章 1 概要

1.1 Unity について 1-13
1.2 ARM® Mali™ GPU について 1-14
1.3 最適化について 1-15
1.4 Ice Cave デモについて 1-16

第 章 2 アプリケーションの最適化

2.1 最適化プロセス 2-18
2.2 Unity の品質設定 2-19

第 章 3 パフォーマンス解析

3.1 パフォーマンス解析について 3-23
3.2 Unity ゲームサンプルのプロファイリング 3-24

第 章 4 最適化リスト

4.1 アプリケーション プロセッサの最適化 4-43
4.2 GPU の最適化 4-49
4.3 アセットの最適化 4-70

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

4.4 Mali™ オフライン シェーダ コンパイラによる最適化 4-72

第 章 5 リアルタイム 3D アートのベスト プラクティス: ジオメトリ
5.1 ジオメトリとは 5-79
5.2 三角形とポリゴンの使用 5-80
5.3 詳細レベル 5-86
5.4 ジオメトリに関するその他のベスト プラクティス 5-91

第 章 6 リアルタイム 3D アートのベスト プラクティス: テクスチャリング
6.1 テクスチャ アトラス、フィルタリング、ミップマップ テクスチャ アトラス 6-95
6.2 テクスチャ フィルタリング 6-96
6.3 ミップマッピング 6-100
6.4 テクスチャ サイズ、色空間、および圧縮 6-101
6.5 UV アンラップ、ビジュアル インパクト、テクスチャ チャネル パッキング 6-104
6.6 アルファ チャネルおよび法線マップのベスト プラクティス 6-108
6.7 法線マップ ベーキングのベスト プラクティス 6-110
6.8 テクスチャ設定の編集 6-114

第 章 7 リアルタイム 3D アートのベスト プラクティス: マテリアルとシェーダ
7.1 シェーダとマテリアルの概要 7-116
7.2 モバイル プラットフォーム用に最適化されたシェーダの使用 7-118
7.3 テクスチャの最適化 7-119
7.4 ライティングなしシェーダとライティングありシェーダの比較 7-120
7.5 透明度を適用する場合の注意点 7-122
7.6 透明度の実装方法のプロファイリングと比較 7-124
7.7 補足資料とシェーダのベスト プラクティス 7-126

第 章 8 リアルタイム 3D アートのベストプラクティス: ライティング
8.1 ライティングの概要 8-128
8.2 レンダーパイプライン 8-129
8.3 ライトモード 8-130
8.4 静的なライトの使用(可能な場合) 8-131
8.5 可能な限りベイクする 8-132
8.6 ライトマップの最適化 8-134
8.7 可能な限りフェイクする 8-138
8.8 ライトプローブ 8-139
8.9 メッシュレンダラーの設定 8-141
8.10 リアルタイムライトとライトのタイプ 8-142

第 章 9 高度なグラフィックス手法
9.1 カスタム シェーダ 9-144
9.2 ローカル キューブマップによる反射の実装 9-157
9.3 反射の結合 9-174
9.4 ローカル キューブマップに基づく動的なソフト シャドウ 9-180
9.5 ローカル キューブマップに基づく屈折 9-188
9.6 Ice Cave デモの鏡面反射エフェクト 9-194
9.7 Early-Z を使用する 9-197
9.8 ダーティ レンズ エフェクト 9-198
9.9 ライト シャフト 9-201

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.10 フォグ エフェクト 9-205
9.11 ブルーム 9-212
9.12 氷壁のエフェクト 9-219
9.13 手続き型スカイボックス 9-225
9.14 ホタル 9-233
9.15 接空間からワールド空間への法線変換ツール 9-237

第 章 10 バーチャル リアリティ
10.1 Unity のバーチャル リアリティ ハードウェア サポート 10-245
10.2 Unity VR への移植プロセス 10-246
10.3 VR への移植時に考慮すべき事項 10-249
10.4 VR での反射 10-251
10.5 結果 10-256

第 章 11 高度な VR グラフィックス手法
11.1 エイリアシング 11-258
11.2 マルチサンプル アンチエイリアシング 11-260
11.3 ミップマッピング 11-261
11.4 詳細レベル 11-263
11.5 色空間 11-266
11.6 テクスチャ フィルタリング 11-267
11.7 アルファ合成 11-270
11.8 レベル設計 11-274
11.9 バンディング 11-276
11.10 バンプ マッピング 11-278
11.11 影 11-280

第 章 12 Vulkan
12.1 Vulkan について 12-283
12.2 Unity における Vulkan について 12-285
12.3 Unity における Vulkan の有効化 12-286
12.4 Vulkan のケース スタディ 12-287

付録 A リビジョン
A.1 リビジョン 付録-A-292

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

Preface

この前書きでは、次について紹介します。*ARM® Unity 開発者ガイド モバイル ゲームのグラフィックスの最適化*.

このドキュメントは、次で構成されています。

- *本書について*(9 ページ).
- *ご意見、ご感想*(11 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

本書について

本書は、特に Mali™ GPU を搭載したモバイル プラットフォームで、Unity を最大限に活用するアプリケーションとコンテンツを作成するために役立つように設計されています。

製品リビジョン ステータス

r*m*p*n* の識別子は、本書で説明されている製品のリビジョン ステータスを示します。たとえば、r*1*p*2* の場合:

r*m* 製品のメジャー リビジョンを示します。たとえば、r1 などです。

p*n* 製品のマイナー リビジョンまたは変更ステータスを示します。たとえば、p2 などです。

対象読者

本書の構成

本書は以下の章から構成されています。

第 章 1 概要
この章では、ARM モバイル ゲーミング グラフィックスを最適化するための Unity 開発者ガイドについて概説します。

第 章 2 アプリケーションの最適化
本章では、Unity でアプリケーションを最適化する方法について説明します。

第 章 3 パフォーマンス解析
本章では、アプリケーションのプロファイリングについて説明します。

第 章 4 最適化リスト
本章では、Unity アプリケーションのいくつかの最適化について説明します。

第 章 5 リアルタイム 3D アートのベスト プラクティス: ジオメトリ
この章では、3D アセットに対する主なジオメトリの最適化について説明します。ジオメトリ最適化は、ゲームの効率を高めるだけでなく、モバイル プラットフォームでのゲームのパフォーマンスを向上させるという総合的な目標を達成するうえで役立ちます。

第 章 6 リアルタイム 3D アートのベスト プラクティス: テクスチャリング
この章では、ゲームをスムーズに実行し、外観を向上させるために役立ついくつかのテクスチャ最適化について説明します。

第 章 7 リアルタイム 3D アートのベスト プラクティス: マテリアルとシェーダ
この章では、ゲームのより効率的な実行とより良い外観を実現するための、複数の異なるマテリアルとシェーダの最適化について説明します。

第 章 8 リアルタイム 3D アートのベストプラクティス: ライティング
この章では、ゲームの見栄えを良くして、スムーズに実行するのに役立ついくつかのライティングの最適化について説明します。

第 章 9 高度なグラフィックス手法
この章では、使用可能ないくつかの高度なグラフィックス手法を説明します。

第 章 10 バーチャル リアリティ
この章では、バーチャル リアリティ ハードウェア上で実行されるアプリケーションまたはゲームの適用プロセスと、バーチャル リアリティにおける反射の実装のいくつかの違いについて説明します。

第 章 11 高度な VR グラフィックス手法
この章では、バーチャル リアリティ アプリケーションのグラフィック性能を向上させるために使用できるさまざまな技法について説明します。

第 章 12 Vulkan
本章では、Vulkan とそれを有効にする方法について説明します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

付録 A リビジョン

この付録では、本書の改訂版に加えられた変更について説明しています。

用語集

「ARM® 用語集」は、ARM マニュアルで使用されている用語とその定義のリストです。一般に認められている意味と ARM での意味が異なる場合を除いて、「ARM 用語集」に業界標準の用語は含まれていません。

詳細については、「*ARM® 用語集*」を参照して下さい。

表記規則

italic

重要用語、相互参照、引用箇所を示します。

bold

メニュー名などのユーザインタフェース要素を太字で記載しています。また、必要に応じて記述リスト内の重要箇所、ARM プロセッサの信号名、重要用語、および専門用語にも太字を使用しています。

`monospace`

コマンド、ファイル名、プログラム名、ソースコードなど、キーボードから入力可能なテキストを示しています。

`monospace`

コマンドまたはオプションに使用可能な略語を示しています。コマンド名またはオプション名をすべて入力する代わりに、下線部分の文字だけを入力することができます。

`monospace italic`

引数が特定の値で置き換えられる場合のモノスペーステキストの引数を示しています。

`monospace bold`

サンプルコード以外に使用される言語キーワードを示しています。

`<and>`

コードまたはコードの一部のアセンブラ構文で置換可能な項が使用されている場合に、その項を囲みます。以下はその一例です。

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

スモールキャピタル

「*ARM® 用語集*」で定義されている専門的な意味を持つ用語について、本文中で使用されます。たとえば、IMPLEMENTATION DEFINED、IMPLEMENTATION SPECIFIC、UNKNOWN、および UNPREDICTABLE などです。

参考資料

ARM の出版物

他の出版物

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ご意見、ご感想

本製品に関するフィードバック

本製品についてのご意見やご提案がございましたら、以下の情報を添えて購入元までお寄せ下さい。

- 製品名
- 製品のリビジョンまたはバージョン
- 説明にはできるだけ多くの情報を含めて下さい。適宜、症状と診断手順も含めて下さい。

関連参照

内容に関するフィードバック(11 ページ)

内容に関するフィードバック

内容に関するご意見につきましては、電子メールを *errata@arm.com* まで送信してください。その際には、以下の内容を記載してください。

- *タイトル ARM Unity 開発者ガイド モバイル ゲームのグラフィックスの最適化。*
- 文書番号(100140_0402_01_ja)。
- 該当する場合は、問題のあるページ番号。
- 問題点の簡潔な説明

ARM では、補足すべき点や改善すべき点についての全般的なご提案もお待ちしております。

注

ARM では、この PDF を Adobe Acrobat および Acrobat Reader でのみテストしており、その他の PDF リーダーを使用した場合の表示品質については、保証いたしかねます。

関連参照

本製品に関するフィードバック(11 ページ)

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 1 章
概要

この章では、ARM モバイル ゲーミング グラフィックスを最適化するための Unity 開発者ガイドについて概説します。

以下のセクションから構成されています。

- *1.1 Unity について*(1-13 ページ).
- *1.2 ARM® Mali™ GPU について*(1-14 ページ).
- *1.3 最適化について*(1-15 ページ).
- *1.4 Ice Cave デモについて*(1-16 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

1.1 Unity について

Unity は、2D ゲーム、3D ゲーム、その他のアプリケーションを作成して配布できる、ソフトウェア プラットフォームです。

本書は、特に Mali™ GPU を搭載したモバイル プラットフォームで、Unity を最大限に活用するアプリケーションとコンテンツを作成するために役立つように作成されています。アプリケーションのパフォーマンスを向上させるために使用できる手法とベストプラクティスについて説明しています。

注

特に記載されていない限り、説明されている手法は他のプラットフォームでも使用できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

1.2 ARM® Mali™ GPU について

ARM Mali GPU は、モバイルまたは組み込みデバイス向けに設計されています。ARM Mali GPU は、以下のファミリに区分されます。

Bifrost GPU

Bifrost GPU には、頂点、フラグメント、ジオメトリ、テッセレーション、およびコンピュート処理を実行する統合されたシェーダ コアがあります。Vulkan、OpenGL ES 1.1 から OpenGL ES 3.2 まで、および OpenCL 1.2 Full Profile で作成されたグラフィックスおよびコンピュート アプリケーションに使用されます。

Midgard GPU

Midgard GPU には、バーテックス、フラグメント、ジオメトリ、テッセレーション、および計算処理を実行するシェーダコアが統合されています。これらは、Vulkan、OpenGL ES 1.1 から OpenGL ES 3.2、OpenCL 1.2 フルプロファイルを使ったグラフィックスや計算アプリケーションに使用されます。

Utgard GPU

Utgard GPU には、1 つの頂点プロセッサと 1 つ以上のフラグメント プロセッサがあります。OpenGL ES 1.1 と 2.0 で作成されたグラフィックス限定アプリケーションに使用されます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

1.3 最適化について

グラフィックスは物事を良く見せることが重要です。最適化では、最小の計算量で見栄えを良くすることが重要です。これは、計算能力とメモリ帯域幅を制限して消費電力を抑える必要のあるモバイル デバイスにとって特に重要です。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

1.4 Ice Cave デモについて

Ice Cave デモは、ARM によって作成されたデモンストレーション アプリケーションです。モバイル デバイス用の高品質のビジュアル コンテンツを生成するために、いくつかの最適化された手法を使用しています。

このドキュメントでは、Ice Cave デモで使用されているグラフィックス手法と、プロジェクトの開発中に生じた問題の解決方法について説明します。

Ice Cave デモは、ARM Cortex® -A57 MP4 プロセッサと ARM Mali -T760 MP8 GPU を搭載したモバイル デバイスで開発およびテストされています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 2 章
アプリケーションの最適化

本章では、Unity でアプリケーションを最適化する方法について説明します。

以下のセクションから構成されています。

- *2.1 最適化プロセス*(2-18 ページ).
- *2.2 Unity の品質設定*(2-19 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

2.1 最適化プロセス

最適化とは、アプリケーションをより効率的にするプロセスです。グラフィックス アプリケーションでは通常、アプリケーションがより速く動作するように変更することを意味します。

例えば、フレーム レートが低いゲームは、カクカクしているように見えることがあります。これだと印象が悪く、プレイしづらくなる可能性があります。最適化を使用すると、ゲームのフレーム レートを上げて、見栄えを良くし、スムーズにプレイすることができます。

コードを最適化するには、最適化のプロセスを使用します。これは、パフォーマンス上の問題を発見して除去する反復的なプロセスです。

最適化プロセスは、次の手順で構成されます。

1. プロファイラでアプリケーションの測定を行います。
2. データを分析してボトルネックを特定します。
3. 適用する必要な最適化を決定します。
4. 最適化が機能することを確認します。
5. パフォーマンスが望ましくない場合は、手順 1 に戻り、プロセスを繰り返します。

次に、最適化プロセスのサンプルを示します。

1. 求められるパフォーマンスを発揮していないゲームがある場合は、プロファイラを使って計測することができます。
2. パフォーマンス問題の原因を切り分けて特定できるように、プロファイラを使って測定値を分析します。
3. 一例として、レンダリングする頂点の数が多すぎることが、このゲームの問題であることがわかりました。
4. メッシュ内の頂点の数を減らします。
5. ゲームを再度実行し、最適化が確実に機能していることを確認します。

このプロセスを実行しても、ゲームのパフォーマンスが望ましくない場合は、アプリケーションのプロファイリングをもう一度行い、問題の原因を特定するプロセスを再度行います。

このプロセスは何度も繰り返す必要がある場合があります。最適化は、さまざまなところでパフォーマンス上の問題を見つける場合がある、反復的なプロセスです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

2.2 Unity の品質設定

Unity の品質設定を理解することは、アプリケーションの正しい設定を選択するために役立ちます。

Unity には、ゲームの画質を変える、さまざまなオプションがあります。これらのオプションの中には、計算コストが高く、ゲームのパフォーマンスに悪影響を与えるものもあります。

次の図は、**インスペクタ**の品質設定を示しています。

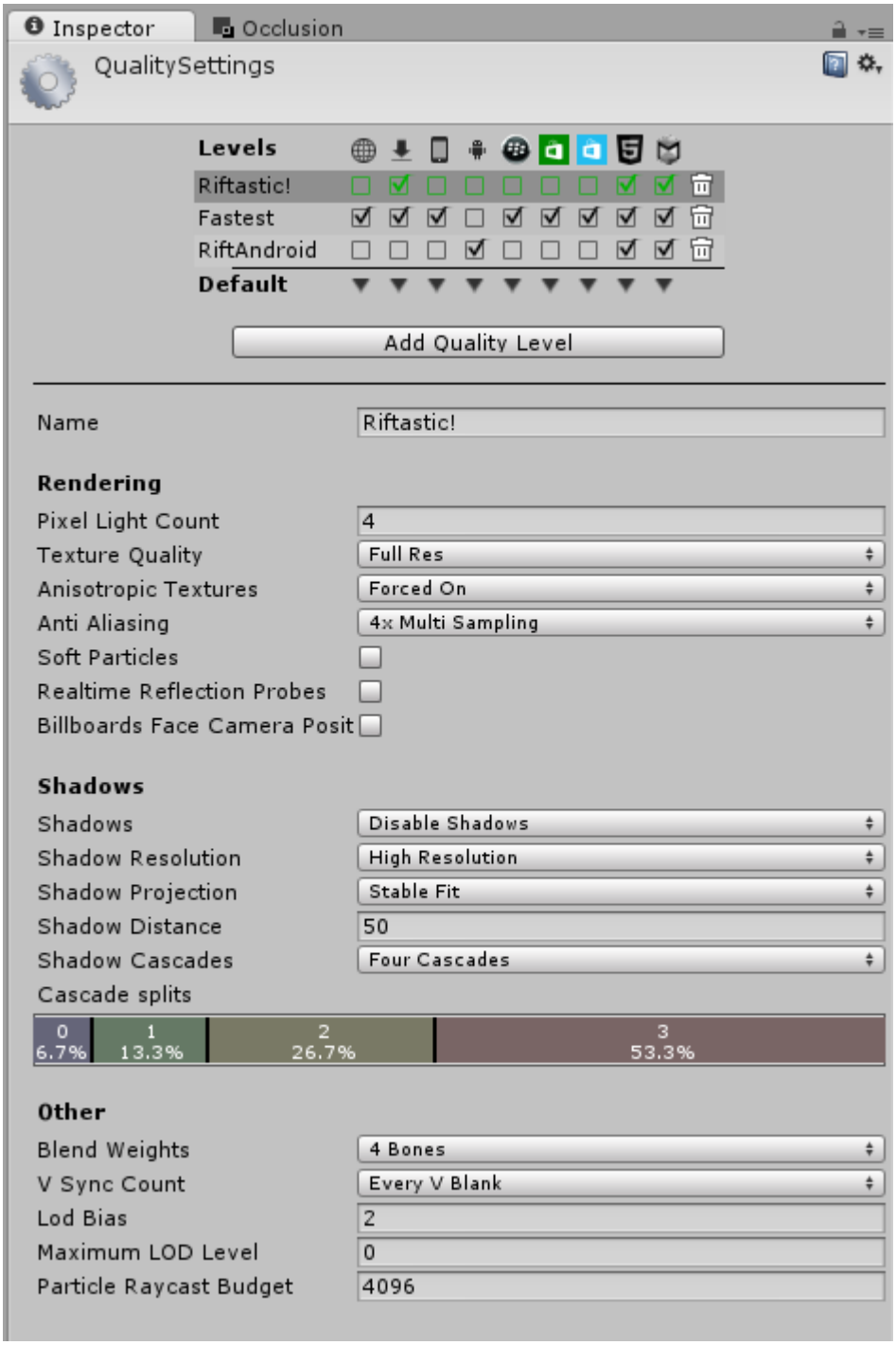


図 2–1 クオリティ設定

パフォーマンスのトレードオフをわずかに抑えて、ゲームの画質を向上できる、さまざまなオプションがあります。たとえば、ゲームのフレームレートが低い場合に、複雑なグラフィックス エフェクトを実行する

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ときに GPU の処理情報量が多すぎることがあります。影やライティングなどのグラフィックス エフェクトが複雑でないバージョンを実行して、画質への影響を比較的小さくすることができます。より簡単なエフェクトを使用することにより、GPU の負荷が大幅に軽減され、フレームレートが向上します。

ライティングのデフォルト設定は、モバイル デバイスにとっては複雑すぎることがあるため、モバイル プラットフォーム用に作成されたゲームの中には、複雑な手法を避け、ゲーム固有の手法を使用するものもあります。これには、ライトマップへのライティングのプリベーキングや、影を投影する代わりにテクスチャを投影するなどの手法が含まれます。

[Project Settings] > [Quality]には、ゲームのパフォーマンスに大きな影響を与える場合がある、いくつかのオプションがあります。

Pixel Light Count

ピクセルライト数は、あるピクセルに影響を与える可能性のあるライトの数です。Pixel Light Count の値が大きい場合、多くの計算を必要とします。多くのゲームでは、ダイナミック ライトとリアルタイム ライトをあまり使用しなくても、画質への影響を最小限に抑えることができます。ライティングによってゲームのパフォーマンスの問題を生じている場合は、ライトマップや投影テクスチャなどの手法の使用を検討します。

Texture Quality

テクスチャ品質は GPU を読み込むことができますが、通常はパフォーマンスの問題を引き起こしません。Texture Quality を下げると、ゲームの視覚的品質に悪影響を与える可能性があります。不可避な場合のみ下げるようにします。Ice Cave デモでは、**Texture Quality** は Full Res (フル解像度) に設定されています。
テクスチャによってパフォーマンス上の問題が生じている場合には、ミップマッピングを使用してみてください。ミップマッピングは、画質に影響を与えることなく、計算および帯域幅の要件を削減します。

AntiAliasing

アンチエイリアシングは、ピクセルを三角形のエッジの周りにブレンドするエッジ スムージングの手法です。これにより、ゲームの画質が大幅に向上します。アンチエイリアスにはいくつかの方法がありますが、これは MSAA (*Multi-Sampled Anti-Aliasing*) のトグルです。MSAA の 4x は Mali GPU 上では非常に低コストで動作するため、可能な場合は常に使用してください。

Soft Particles

ソフト パーティクルでは、デプス テクスチャにレンダリングするか、遅延モードでレンダリングする必要があります。これは GPU の負荷を増加させますが、パーティクルに現実的なビジュアルを実現するという点では価値があります。モバイル プラットフォームでは、デプス テクスチャへのレンダリングと読み込みは貴重な帯域幅を消費します。また遅延パスを使用したレンダリングでは MSAA を活用できません。ゲームでソフト パーティクルが重要であるかどうかを十分に検討して使用します。

Anisotropic Textures

異方性テクスチャは、高い勾配で描かれたテクスチャから歪みを取り除く手法です。これは画質を改善に役立ちますが、コストの高い手法です。歪みが特に目立たない限り、この手法は使用しないようにします。

Shadows

影が高品質である場合、計算コストが大きくなる可能性があります。影によってパフォーマンス上の問題が生じる場合は、単純な影にするか、またはオフにします。ゲームで影が重要な場合は、投影テクスチャなどの簡単な動的シャドウイングの手法の使用を検討します。

Realtime Reflection Probes

リアルタイム リフレクション プローブ オプションは、ランタイム パフォーマンスに重大な悪影響を及ぼす場合があります。

リフレクション プローブがレンダリングされると、キューブマップのすべての面が、プローブの原点にあるカメラによって、別々にレンダリングされます。相互反射が考慮される場合、このプロセスはすべての反射バウンス レベルに対して行われます。光沢のある反射の場合、ぼかし処理を適用するためにキューブマップのミップマップも使用されます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

キューブマップのレンダリングに影響を与える要因は以下のとおりです。

Cubemap Resolution

高解像度のキューブマップでは、レンダリング時間が長くなります。必要なだけの最も低い品質のキューブマップを使用します。

Culling Mask

キューブマップをレンダリングするときは、カリング マスクを使用して、反射に関係のないジオメトリをレンダリングしないようにします。

Cubemap Updating

Refresh Mode オプションは、キューブマップの更新頻度を定義します。

- Every Frame オプションは、フレームごとにキューブマップをレンダリングします。これは計算コストが最も高いオプションです。必要がない限り使用しないようにします。
- The On Awake オプションは、実行時にシーンが開始されるときに 1 回、キューブマップをレンダリングします。
- Via Scripting オプションを使うと、キューブマップの更新タイミングを制御できます。このオプションを使用すると、更新の実行時の条件を指定でき、ランタイム リソースの使用を制限できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 3 章
パフォーマンス解析

本章では、アプリケーションのプロファイリングについて説明します。

以下のセクションから構成されています。

- *3.1 パフォーマンス解析について*(3-23 ページ).
- *3.2 Unity ゲームサンプルのプロファイリング*(3-24 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

3.1 パフォーマンス解析について

モバイルデバイスでのゲームのパフォーマンスをテストするには、プロファイラやグラフィックスデバッガーなどの評価ツールを使用できます。これらのツールは、デバイスのリソースがゲームによってどのように使用されたかを正確に詳しく表示します。

*Unity の組み込みプロファイラ*や *Arm Mobile Studio の Streamline* などのプロファイラは、接続されたデバイスで実行されているゲームのシーンを分析して、アプリケーションがどこで時間を費やしているかを示します。このようなツールはデータをグラフで表示し、デバイスが CPU と GPU のワークロードをどのように処理しているかを視覚化します。ここで、ゲームの問題のある領域を特定し、パフォーマンスを最適化する機会を見つけることができます。

*Unity の組み込みフレームデバッガー*や *Arm Mobile Studio の Graphics Analyzer* などのグラフィックスデバッガーを使用すると、ゲームが行う OpenGL ES コールや Vulkan API コールをドローコールごとに見つけることができるため、レンダリングの欠陥を特定し、シーンの詳細レベルを評価して、最適化が必要なコストのかかっているシェーダを見つけることができます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

3.2 Unity ゲームサンプルのプロファイリング

Streamline は、*ARM Mobile Studio* (AMS) 最適化およびデバッグ ソフトウェア ツール スイートの一部を形成する多目的パフォーマンス解析ツールです。

Streamline は、Android デバイス上の複数のソースから、サンプルとイベントベースの両方のパフォーマンス データを収集します。タイムライン ビューをはじめとする各種ビューには、集計結果が表示されます。タイムライン ビューの上半分には収集されたシステム パフォーマンス カウンタが示され、下半分には同じタイムラインに関するさまざまな情報が示されます。

次のスクリーンショットは、ビューの下半分にヒートマップが表示されているタイムライン ビューを示しています。

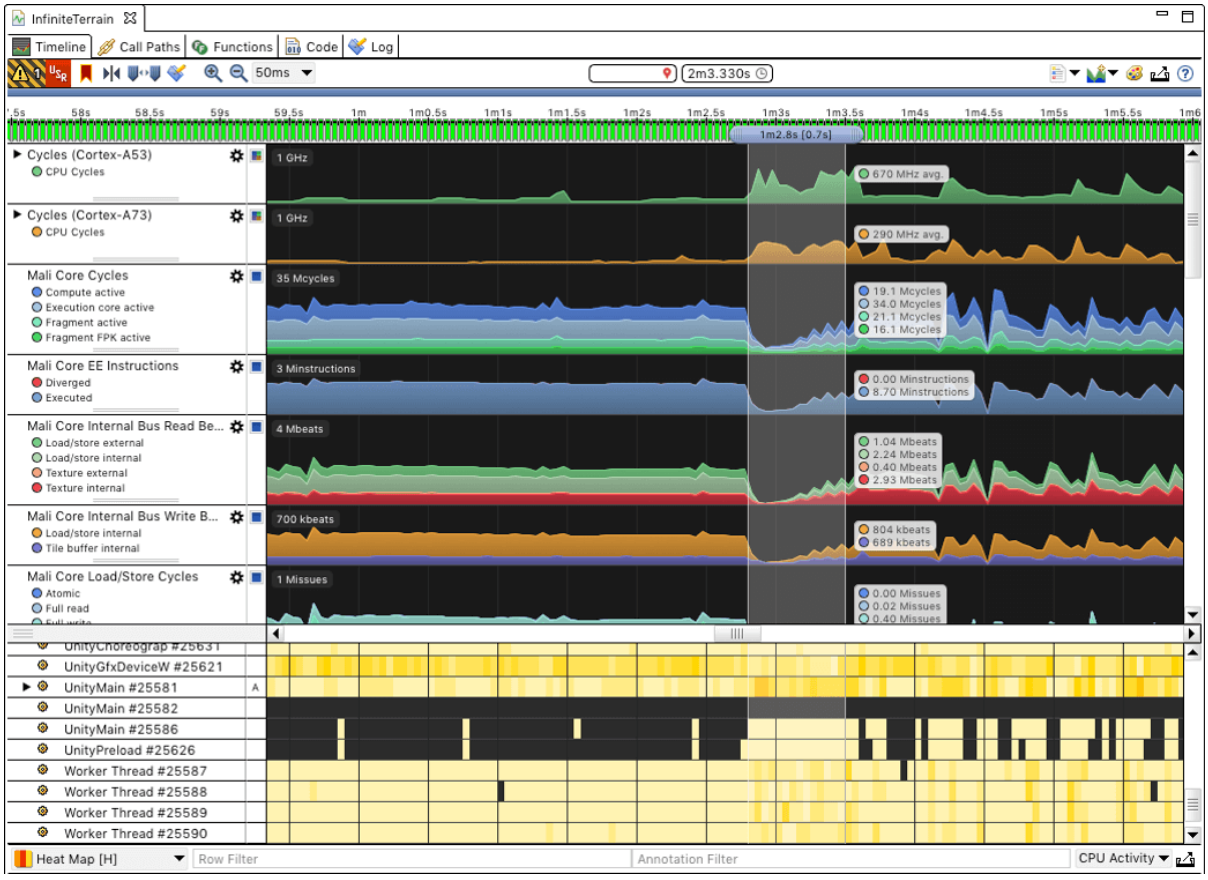


図 3-1 Streamline タイムラインのヒート マップ ビュー

以下のサブセクションから構成されています。

- *3.2.1 サンプル シーンの解析*(3-24 ページ).
- *3.2.2 Unity でのプロファイリング*(3-26 ページ).
- *3.2.3 Streamline を使用した Unity のプロファイリング機能*(3-27 ページ).
- *3.2.4 ワーカー スレッド*(3-30 ページ).
- *3.2.5 Streamline アノテーション*(3-35 ページ).
- *3.2.6 Unity プロジェクトの効果的なセットアップ*(3-36 ページ).

3.2.1 サンプル シーンの解析

次の解析は、カメラが紆余曲折しながら移動する、手順を踏んで生成された地形と、オンザフライで生成された新しい地形で構成されています。

シーンをレンダリングする際、カメラから遠すぎるタイルは削除されます。タイルを削除することで、時間の経過に伴ってシーンの複雑度をほぼ同じ程度に保つことができます。カメラの移動が遅く、新しい地

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

形の生成速度が低下することもあります。カメラの速度が増したら、地形生成の速度も高速度に合わせて増大する必要があります。

以下のスクリーンショットでは、各タイルのエッジが濃い色でレンダリングされ、各タイルのサイズが明らかになっています。

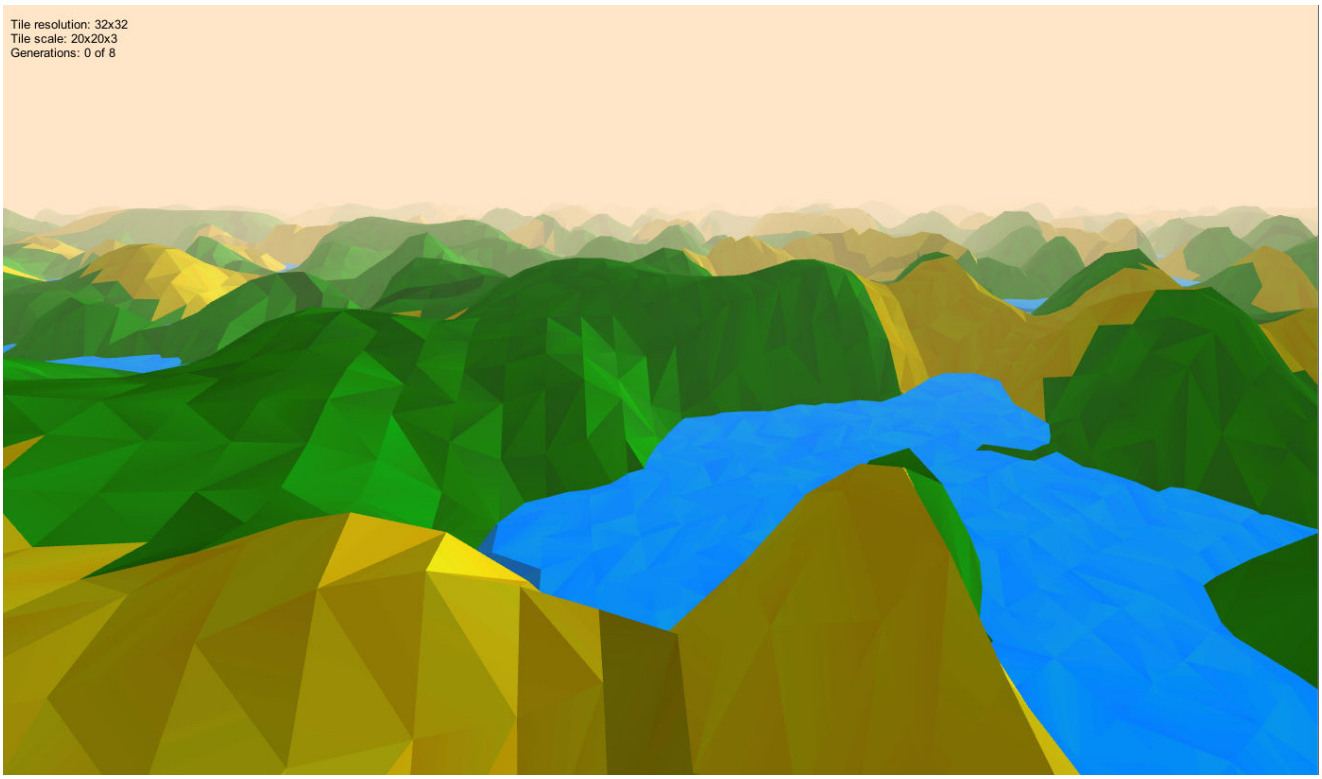


図 3-2 レンダリングされたタイル

オンザフライでの地形タイルの生成は計算量が多くなります。計算量を最小限に抑えるには、*Unity Job Scheduler* (UJS) を使用してメインの Unity スレッドの完了を妨げないバックグラウンド スレッドをディスパッチします。

これによってユーザーは、新しい地形が生成されるたびに、減速されることなく安定したフレーム レートを体験できます。

サンプル デモは、外観は同じでもタイルの生成方法が異なる 4 種類のシーンで実行されるように設定されています。

次の図に示されているように、地形は固定サイズの多くの地形ブロックで構成されます。各ブロックは、固定解像度の複数のメッシュを構成します。メッシュの 1 つは緑色の地形、1 つは黄色の地形、そしてもう 1 つは水です。[Render Distance] は、プレイヤーの周囲に生成されるタイルの数を制御します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

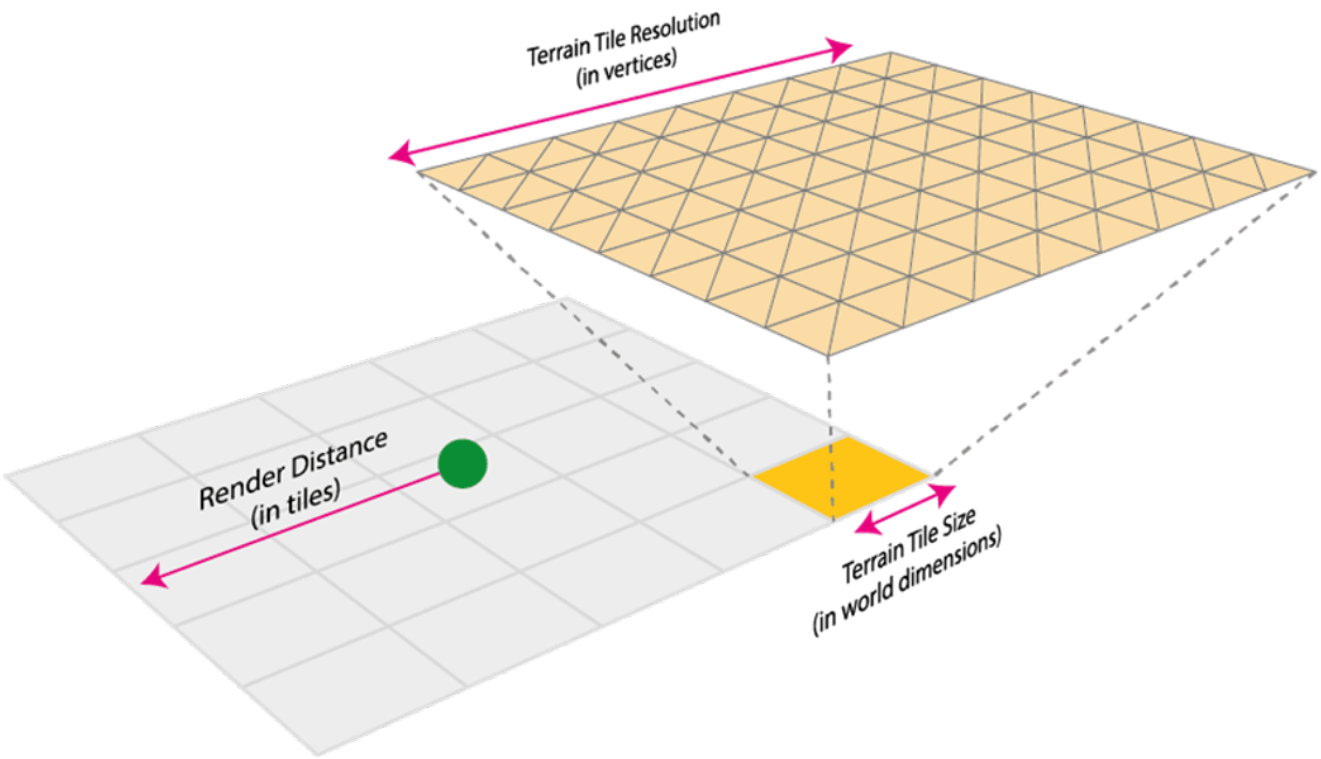


図 3-3 地形タイルの解像度:

4 つのシーンは、次のように設定します。

表 3-1 シーン コンフィギュレーション設定のサンプル

シーン	レンダリング距離	地形タイルのサイズ	地形の解像度	並列地形生成数
1	3	20x20	32x32	8
2	3	20x20	32x32	1
3	6	10x10	16x16	8
4	6	10x10	16x16	1

プロファイリングは、2017 Huawei P10 スマートフォンで実行されます。Huawei P10 には、以下で構成される HiSilicon Kirin 960 チップが含まれています。

- 高性能の ARM Cortex -A73 CPU コア 4 個。
- 高効率の ARM Cortex -A53 CPU コア 4 個。
- ARM Mali -G71 MP8 GPU 1 つ。

3.2.2 Unity でのプロファイリング

Unity 内のプロファイラの詳細が表示されます。ただし、バッテリの使用状況などの重要なデータポイントは表示されません。そのため、Unity のプロファイラを使用して Streamline に情報を渡し、さらに分析することができます。

Unity 内のプロファイラは、ジョブがスケジュールされている場合に表示されます。ただし、プロファイラには、使用可能な CPU または GPU リソースの詳細や、リソースの使用状況は表示されません。

ターゲット アプリケーションのフレーム レートは 60 *フレーム/秒* (FPS) に達する可能性がありますが、他の領域を監視することも重要です。これらの領域には、CPU コアの使用率や GPU のパフォーマンスに関する情報が含まれます。Streamline を使用して、重要な領域を監視できます。

次の図は、Unity に組み込まれているプロファイラ内で使用可能な情報のスクリーンショットを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

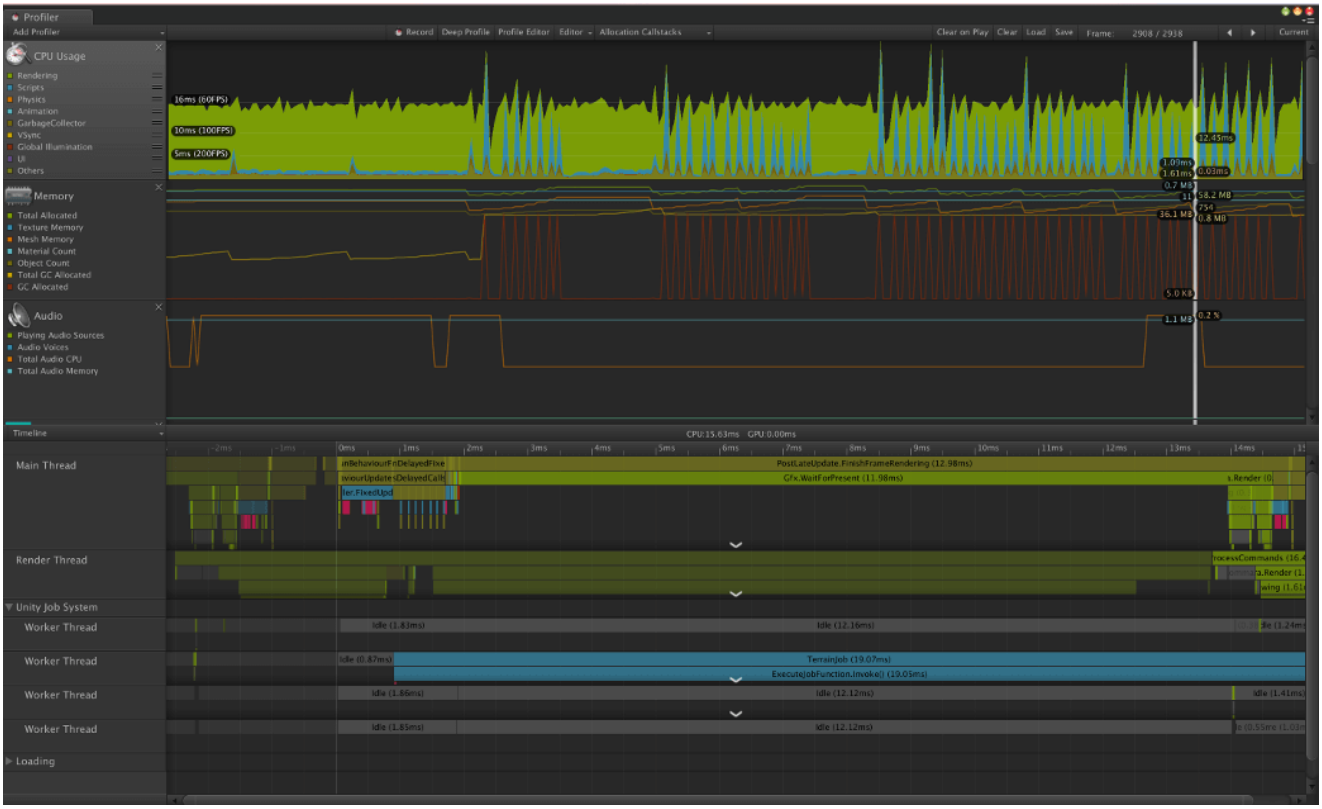


図 3-4 Unity プロファイラ

3.2.3 Streamline を使用した Unity のプロファイリング機能

Streamline には、アプリケーションのパフォーマンスをスレッド単位まで最適化するための主要なプロファイリング機能がいくつか備えられています。

このサンプル シーンは、Streamline 内で 3 つの異なるタイプのアノテーションを使うように変更されています。アノテーションは以下のとおりです。

1. [Markers] – マーカーは、[Timeline] ビューの一番上に表示されるラベル付きの 1 つの時点です。
2. [Channels] – チャネルは、各スレッドに沿って別々の情報行を提供します。
3. *[Custom Activity Maps]*(CAM) – CAM は、複雑な依存関係のある可能性のあるクロススレッド アクティビティを表示できます。各 CAM は、ユーザー インターフェイス (UI) の下半分に固有のビューとして表示されます。

次のスクリーンショットは、[Timeline] ビューに表示された不規則なフレーム レートのサンプルを示しています。

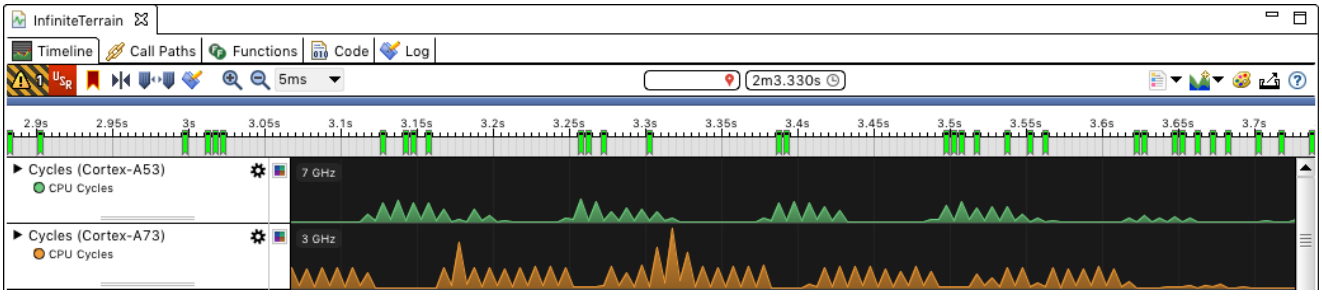


図 3-5 フレーム レート

フレーム レートが意図したよりもスムーズでないこと、および各 CPU コアでかなり多くの集中的なアクティビティがあることがわかります。フレーム レートは減速したり加速したりし、その間に一時停止が時折

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

発生しています。ただし、このような結果は、地形生成から予想されることと一貫しています。問題をさらに詳しく調べる価値はあります。

Streamline では、[Timeline] ビューは 2 つの部分に分けられています。上のビューはメトリック グラスを表示し、下半分にさまざまなプロファイルを表示できます。ヒート マップなどのプロファイルは、システム全体で作業がどのように分散されているのかを表示します。上のタイムラインは、特定のプロセスまたはスレッドに伴う作業のみを表示するようにフィルタ処理できます。

まず [UnityMain] スレッドを選択し、次にすべての [Worker Thread] スレッドを選択すると、ヒート マップを調べることができます。メイン Unity スレッドとジョブ スケジューラ内のスレッドの両方において CPU アクティビティがどのように分割されるかを確認できます。

以下の図は、[UnityMain] スレッドの CPU プロファイルのスクリーンショットです。Cortex -A73 CPU 上にアクティビティの大きなバーストがあります。

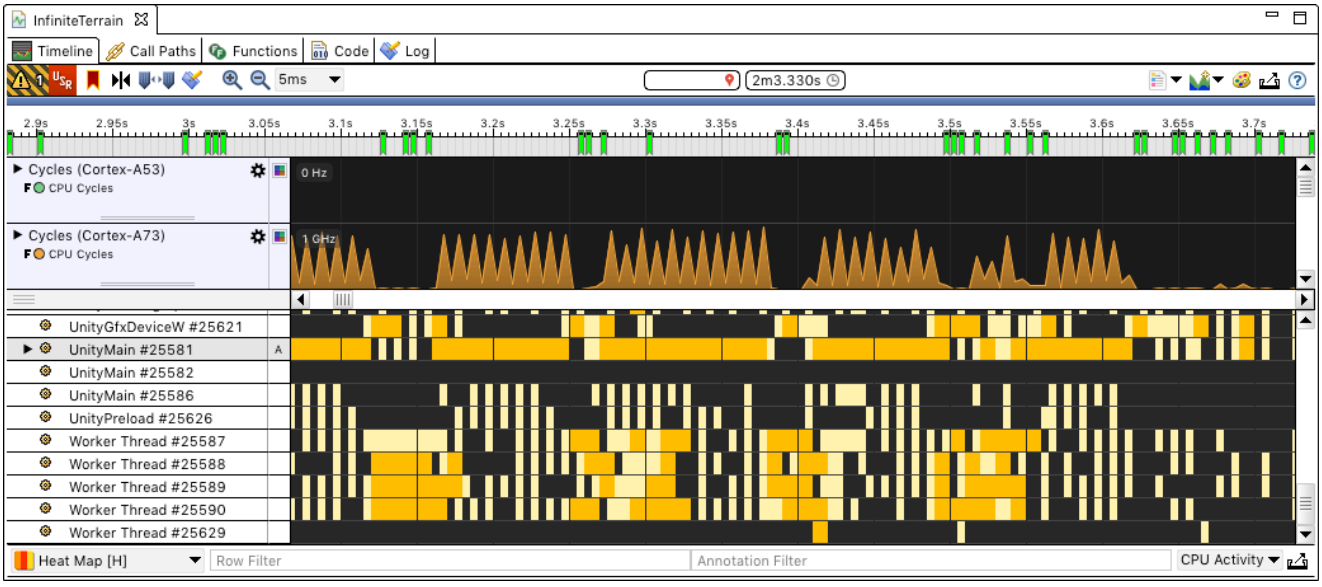


図 3-6 [Unity Main] スレッドの CPU プロファイル

以下のスクリーンショットは、[Worker Thread] スレッドの CPU プロファイルを示しています。これらのスレッドには、Cortex -A73 CPU、および Cortex -A53 CPU の両方でアクティビティの小さなバーストがあることがわかります。

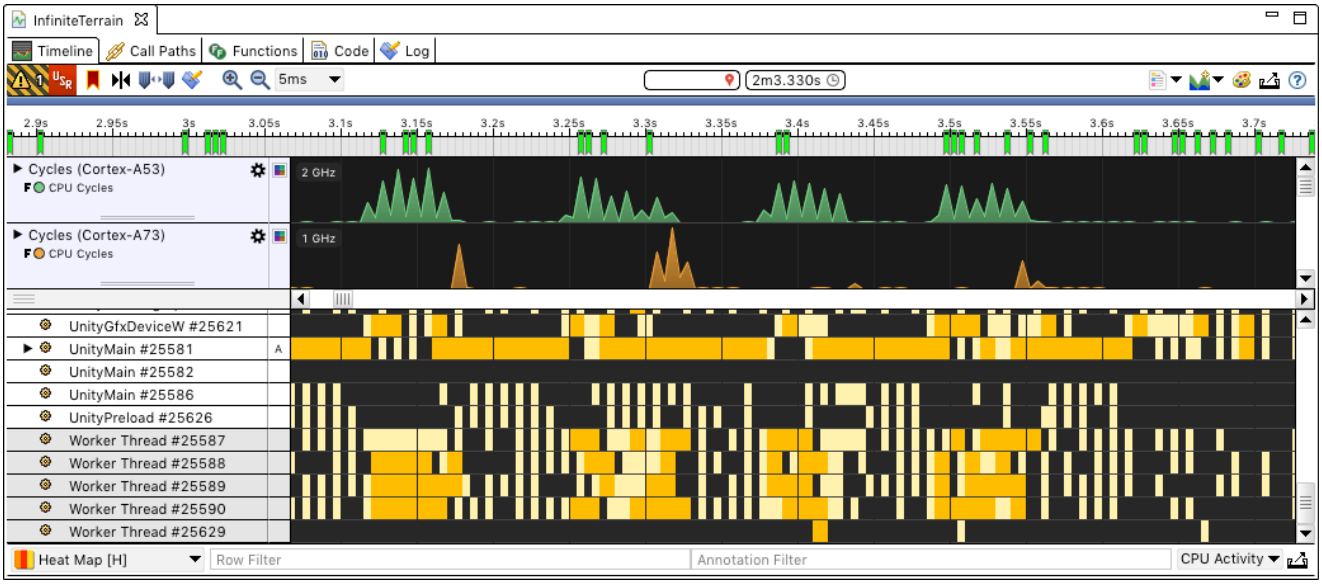


図 3-7 [Worker Thread] スレッドの CPU プロファイル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

以下のスクリーンショットは、[UnityMain] スレッドの拡大ビューです。[UnityMain] 番号の右側にあるスクリーンショットの左側には、[A] アイコンがあります。[A] アイコンは、*Streamline アノテーション チャネル* (SAC) があることを意味します。

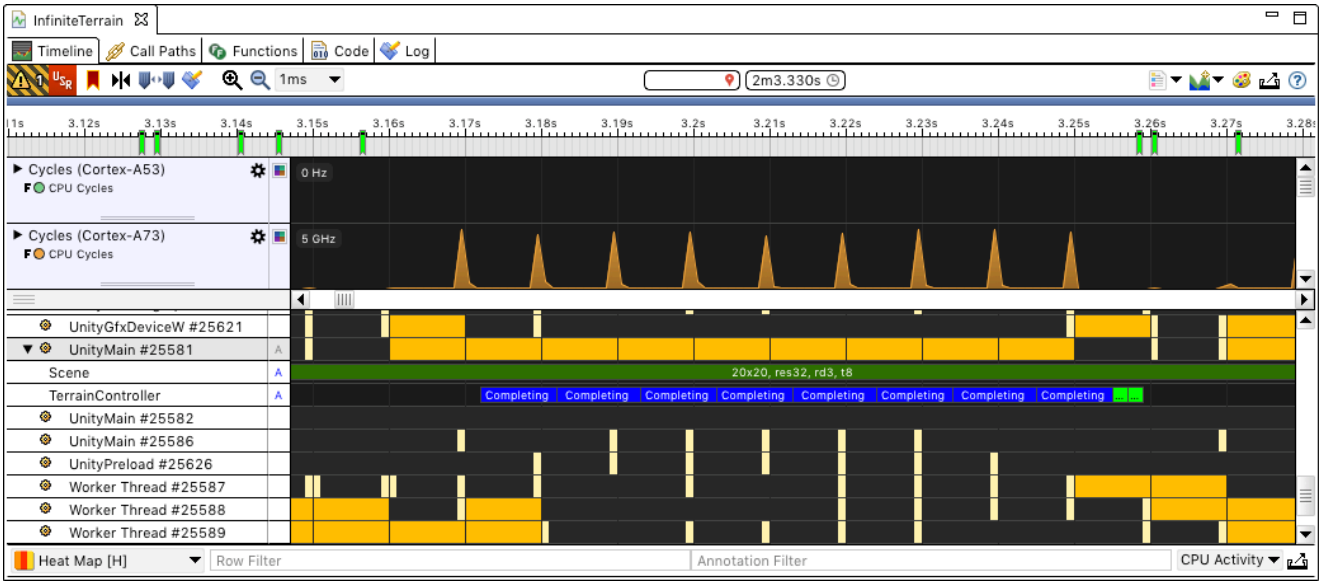


図 3-8 [UnityMain] スレッドの拡大表示

ゲーム内に配置したアノテーションは、Streamline チャネルの [Scene] と [TerrainController] を制御します。[Scene] チャネルは、実行中のシーンを示します。上の図では、実行中のバージョンは 20x20 で、3 つおよび 8 つのスレッドのレンダリング距離で 32x32 シーンが並列で実行されています。

[TerrainController] チャネルは、メイン Unity スレッドでコードの重要部分が実行されているタイミングを示すために使用されます。青いブロックでは、地形ジョブの完了時に実行されるコードがマークアップされます。緑色のブロックでは、生成が現在スケジュールされている新しい地形がマークアップされます。

メイン スレッド アクティビティはすべて、ジョブの完了時に実行する必要のある作業に基づいており、最終メッシュを生成してシーンに挿入する必要があります。

特定のスレッドに集中する際は、解析を特定の期間に制限することができます。Streamline のキャリパにより、解析のために特定の時間領域をマークアップできます。

以下の図は、キャリパのスクリーンショットを示しています。青いバーと矢印はキャリパを表し、タイムラインの上に表示されます。Terrain ジョブの完了に関連するアクティビティが集中する期間が、このシーンに対して選択されています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

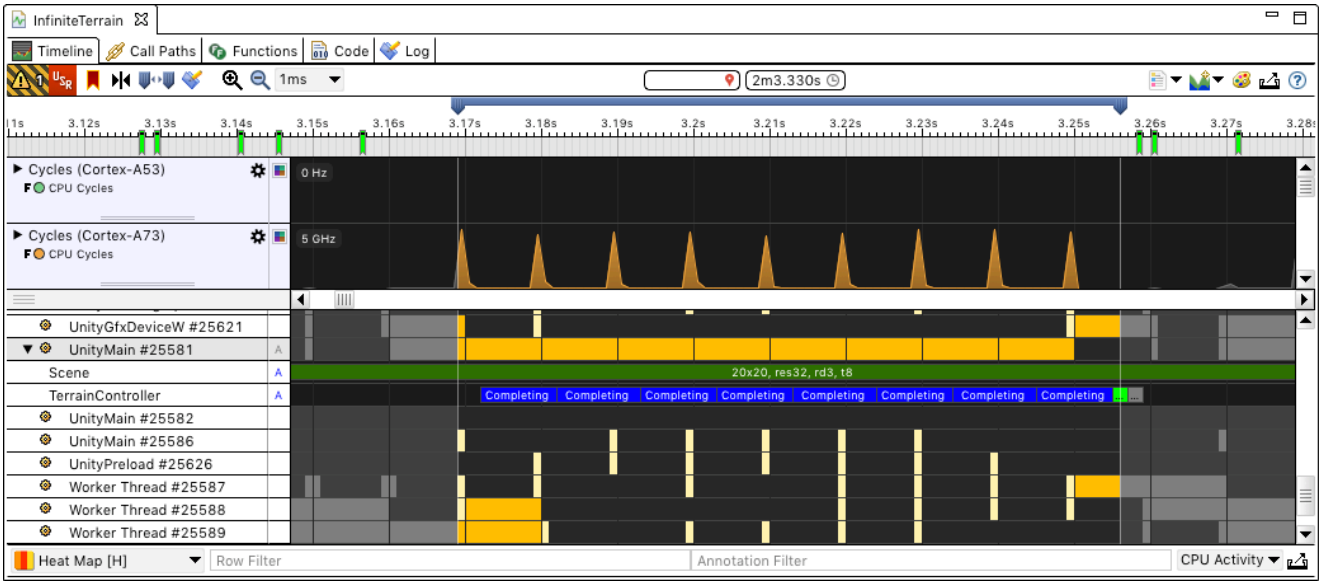


図 3-9 Streamline のキャリパ

[Call Paths] ビューを見ると、キャリパで選択された時間領域の中で時間がどこで消費されたかがわかります。*Unity の IL2CPP スクリプト バックエンド*が使用されているため、実行時に Mono を使用する場合と比べてはるかに多くの情報が得られます。

以下のスクリーンショットは、Unity 内での IL2CPP スクリプト バックエンドの詳細ビューです。

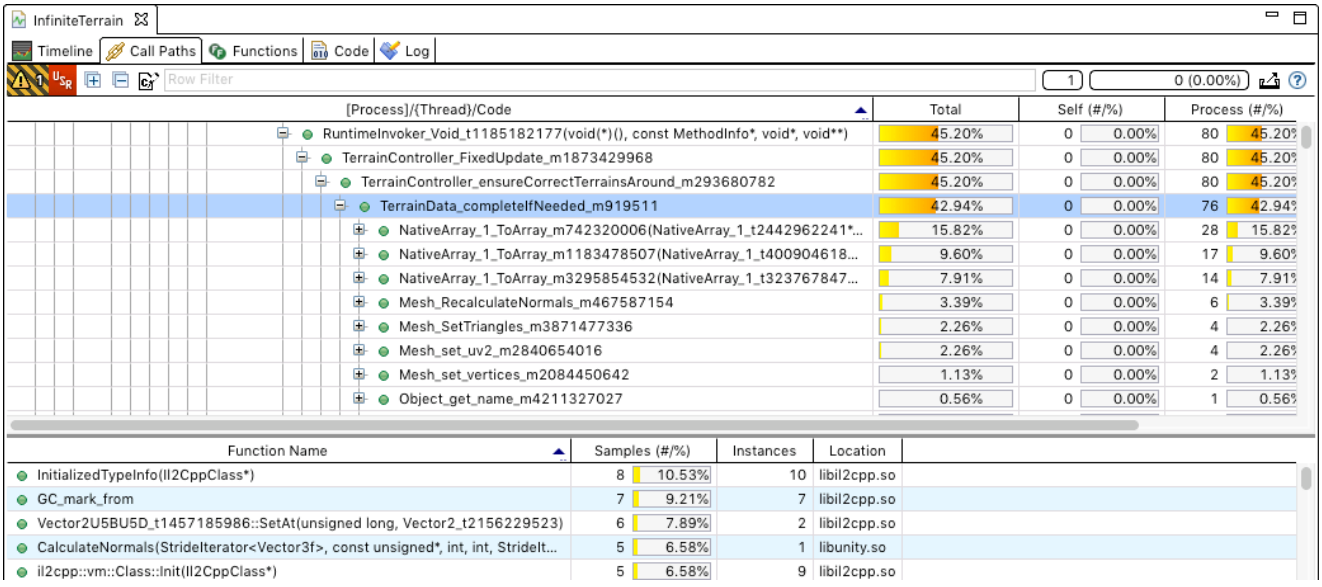


図 3-10 Unity の IL2CPP スクリプト バックエンド

3.2.4 ワーカー スレッド

ゲーム自体からのアノテーションを使用してさらに高レベルのコンテキストを提供する場合は、Streamline の他の機能を利用しましょう。

フィルタ処理後は、最大 8 つの並列ジョブの仕様に従い、8 つの [Worker Thread] スレッドが表示されます。次のスクリーンショットでは、Cortex-A53 クラスタが展開されており、個々のコアの使用率が示されています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

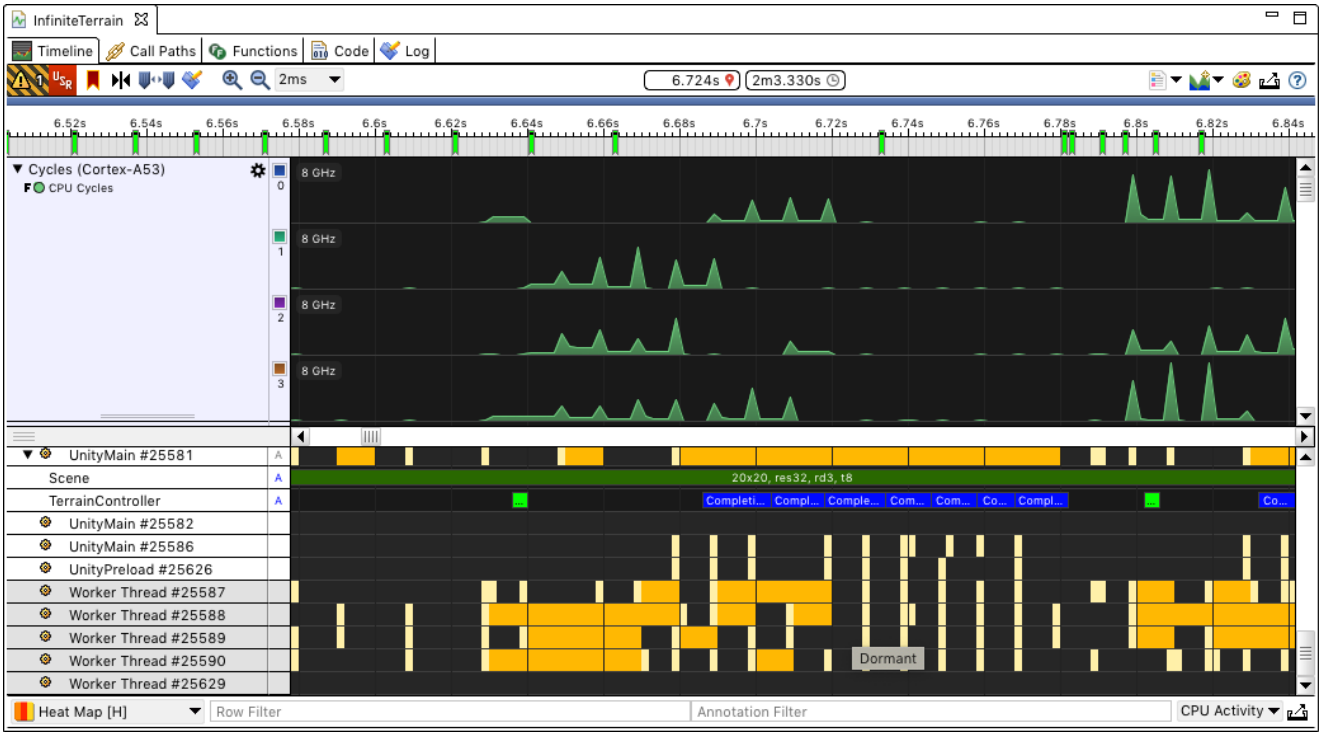


図 3-11 展開された Cortex-A53 クラスタ

[TerrainController] チャネルの緑のブロックは、スケジュールされている新しい地形を示しています。この後には、コア全体における集中的なアクティビティがいくつか示されています。次に、メイン スレッド内のこれらの地形を生成時に処理するために、[TerrainController] 内に青いアクティビティがいくつかあります。これらのグラフにはメイン スレッドのアクティビティは表示されません。これは、[UnityMain] スレッドが選択されていないためです。

これを 2 番目のシーンと比較してみます。地形タイルの複雑度は同じですが、一度に 1 つのみのスケジュールが可能になっています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

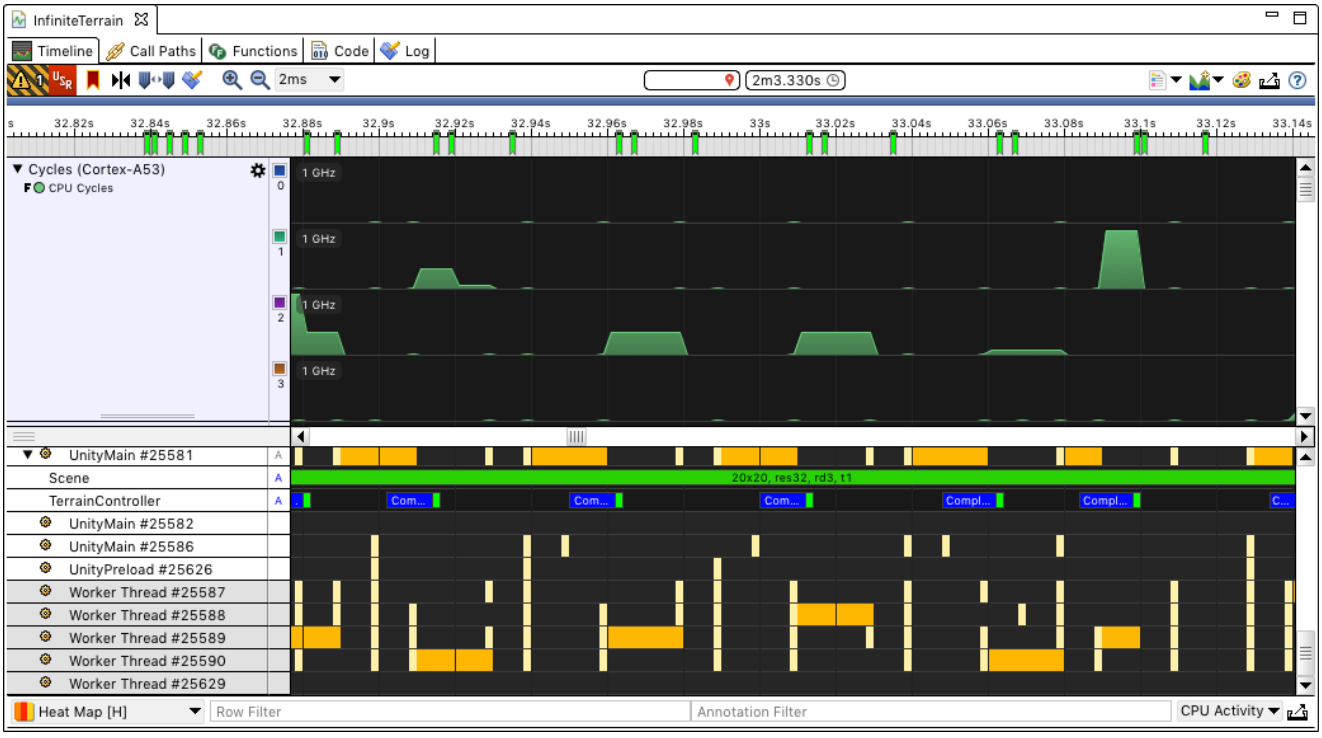


図 3-12 2 番目のシーンとの比較

ここでは次のことに注目する必要があります。

1. 長い時間、コアのほとんどがアイドル状態またはアイドルに近い状態になっているため、CPU アクティビティはあまり活発ではありません。
2. フレーム レートは完璧ではありませんが、スムーズになっています。一度に完了するフレームが 1 つのみのため、メイン スレッドにおけるアクティビティの大きなバーストは削減されています。

また、さらに小さなタイルを使用している 3 番目のシーンとプロファイルを比較してみます。

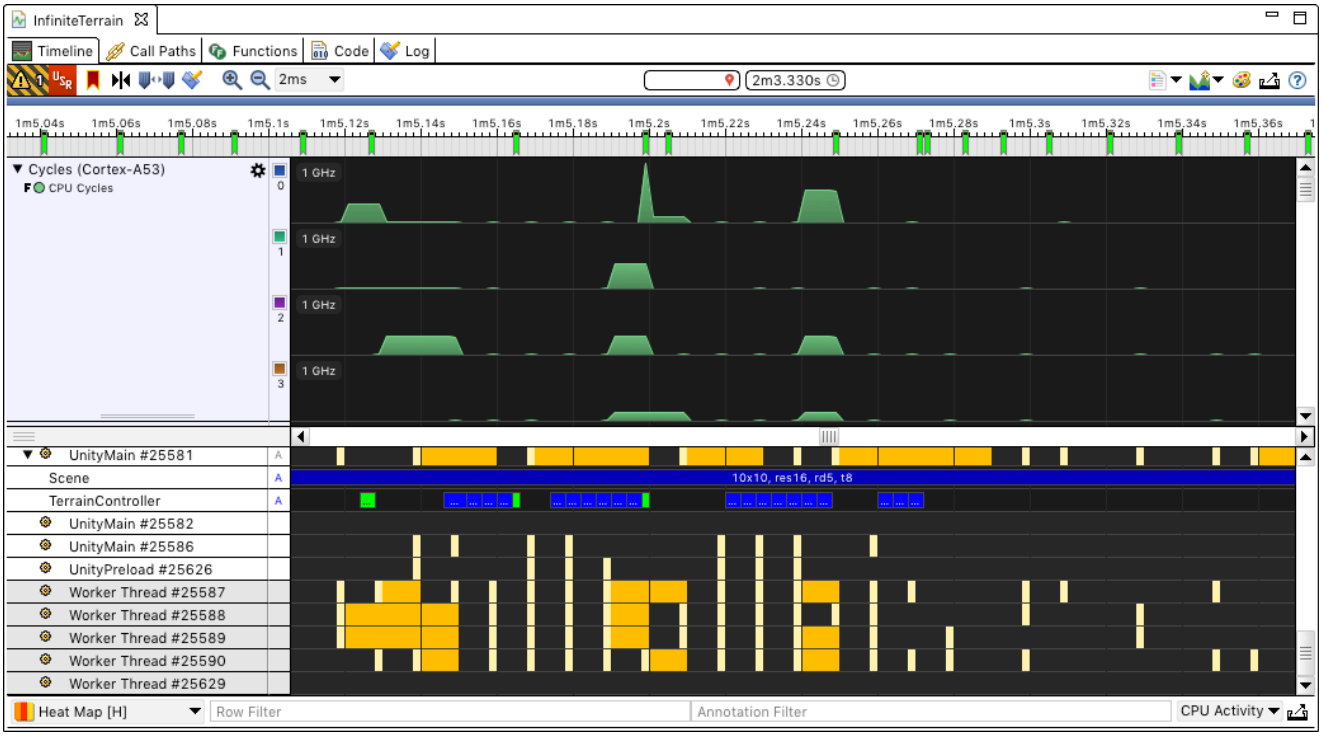
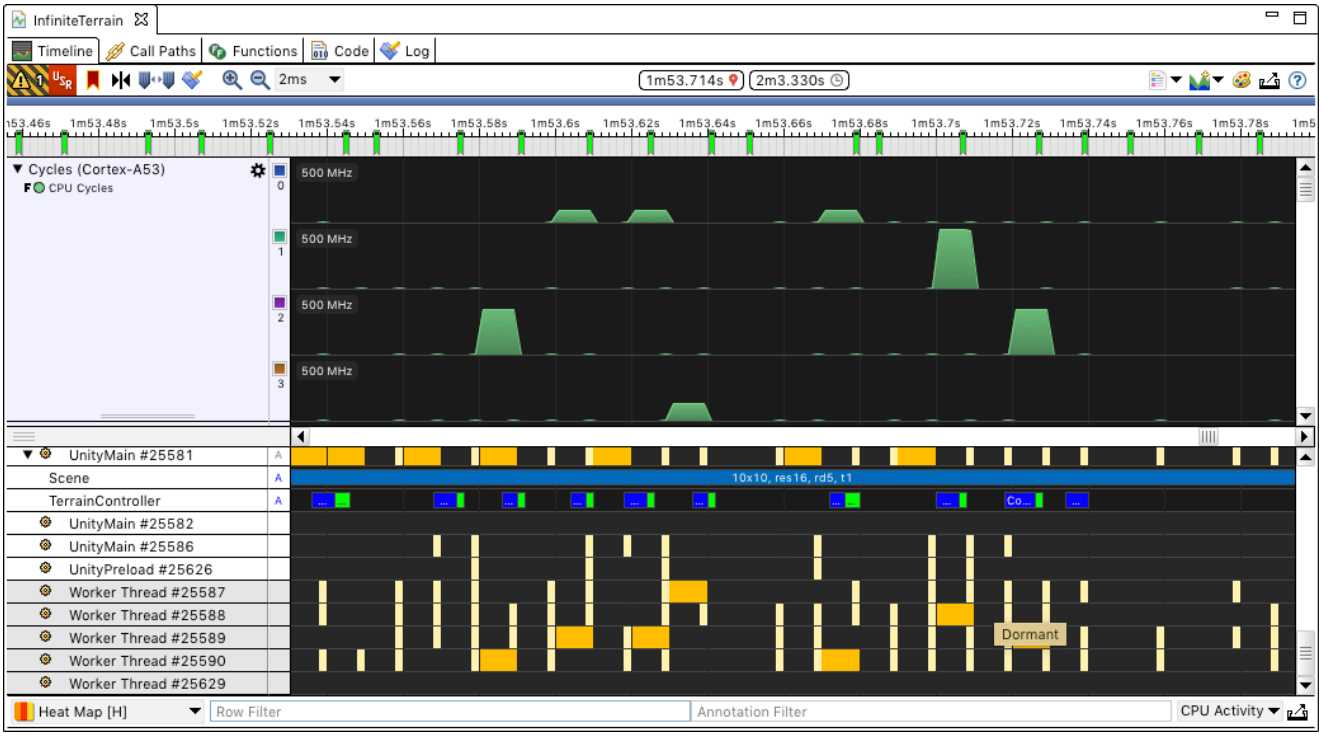


図 3-13 3 番目のシーン

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

CPU アクティビティは低下しており、メイン スレッドにおける青い完了ワークのブロックは短くなっています。これにより、最初のシーンと比較してフレーム レートがスムーズになっていますが、全体的にジョブが増加しているため、カメラが地形上を移動する速度に地形の生成が追いついていることを確認してください。

4 番目のシーンには小さなタイルがあり、一度に 1 つのみの地形が生成されています。したがって、全体的にフレーム レートは最もスムーズですが、地形の生成がカメラの動きに対応するのに十分なレートで行われていることを確認してください。カメラが 4 番目のシーン上を高速で移動している場合、地形の生成が必ず間に合うとは言えません。



Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 3-14 4 番目のシーン

ワーカー スレッドが地形の生成をどのように行っているかをさらに詳しく把握するには、カスタム アクティビティ マップを使用します。各カスタム アクティビティ マップは、左下のメニューにオプションとして表示されます。

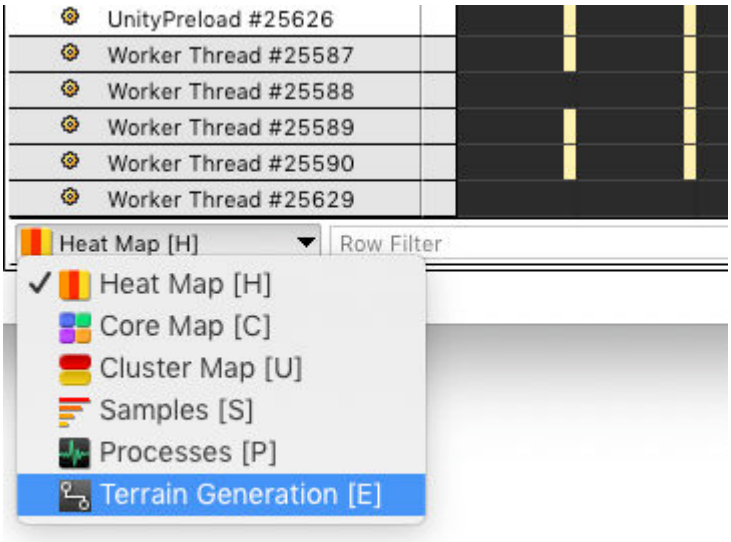


図 3-15 カスタム アクティビティ マップ

[Terrain Generation] ビューが選択されているときには、各地形生成アクティビティに対応した色が付けられたボックスが、開始時点と終了時点を示します。マウスのカーソルを上に置くと、その地形タイルのワールド座標、開始時点、完了までの時間が示されます。スクリーンショットは、Mali GPU で行われた演算作業と、地形が埋められていく過程での GPU アクティビティの着実な増加がグラフで示されています。このスクリーンショットは、8 つの大きなタイルが同時に生成されている最初のシーンの開始時点

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

に焦点を当てて撮られたものです。メイン スレッドが新しいジオメトリを準備している間の一時的な停止によって、GPU が長時間アイドル状態になっています。

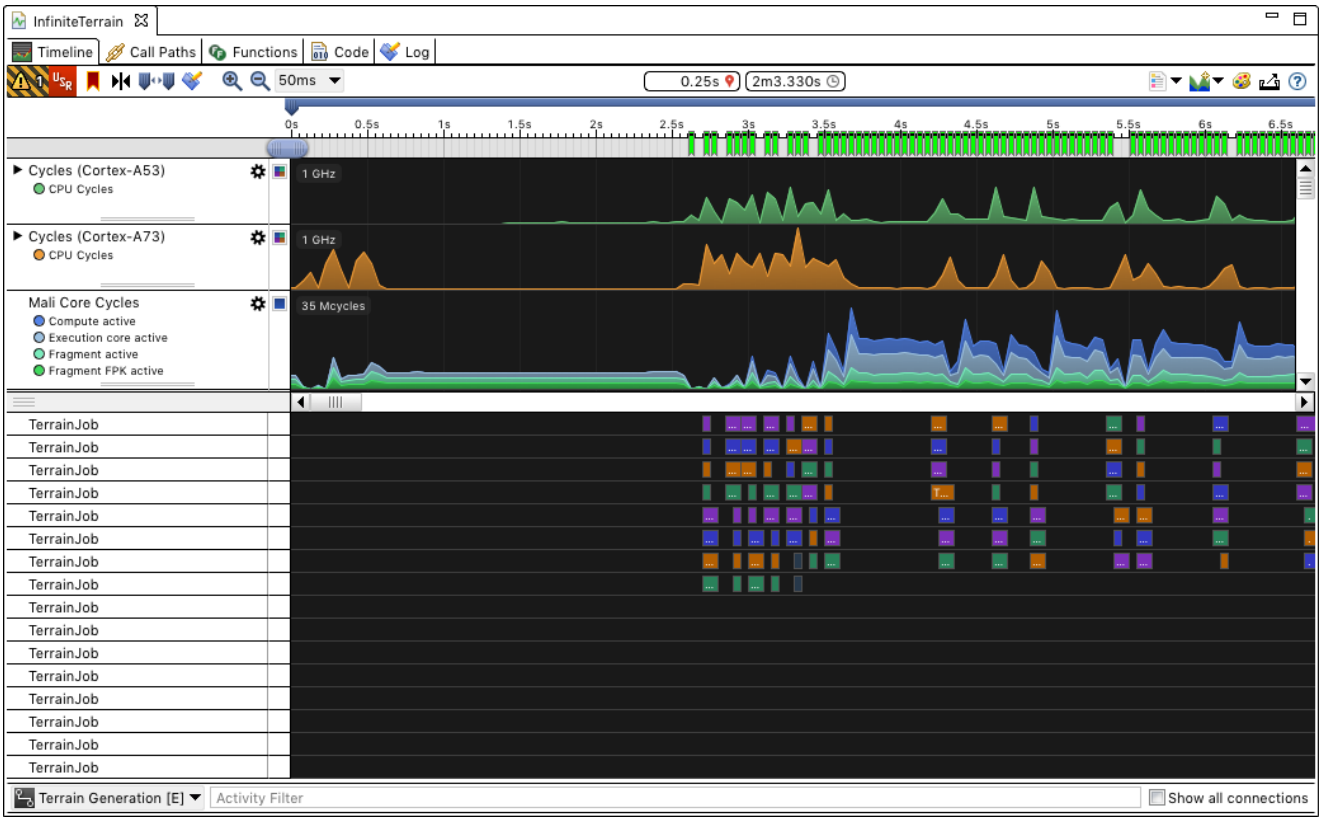


図 3-16 演算作業のグラフ化

4 番目のシーンでは、小さなタイルが連続して生成されるため、GPU アクティビティがよりスムーズに増加しています。一度に実行される地形ジョブは 1 つだけであり、タイルのサイズが小さいため、各ジョブの長さは短くなります。

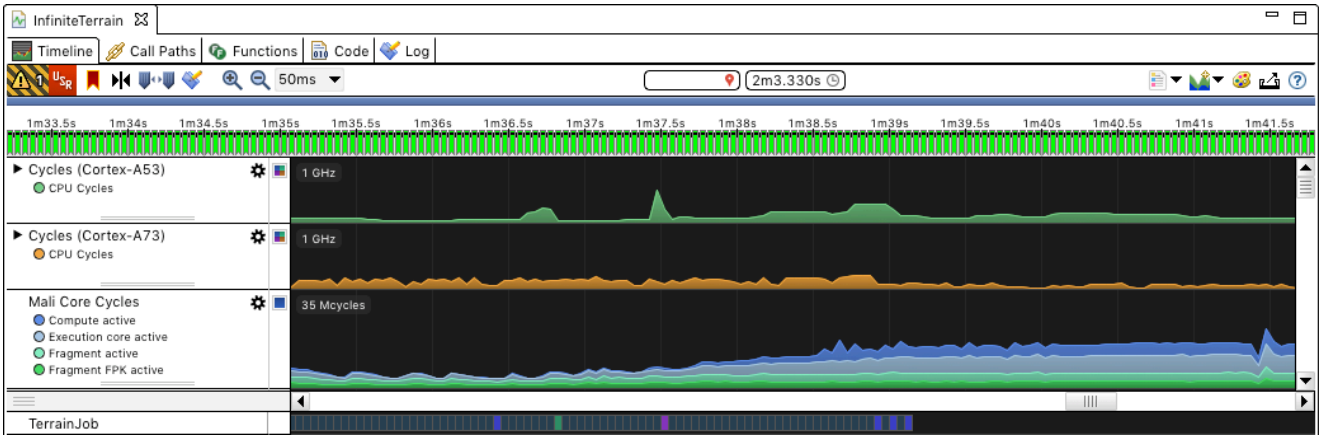


図 3-17 4 番目のシーンの CPU サイクル

3.2.5 Streamline アノテーション

Streamline アノテーションとは何か、その仕組み、および Unity での使用方法を理解することは重要です。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

Streamline アノテーションについて

Android アプリケーションを分析する際、アプリケーションと同じユーザーとして実行している gator という別のプロセスが同じデバイス上で実行されます。gator は、Mali GPU や ARM Cortex -A CPU などのさまざまなハードウェア ソースからプロファイリング情報を収集し、集計されたメトリックのストリームをコンピュータに送信します。アプリケーション自体は、Streamline アノテーションというメカニズムを使用してそのストリームに独自のマーカーとメトリックを挿入します。

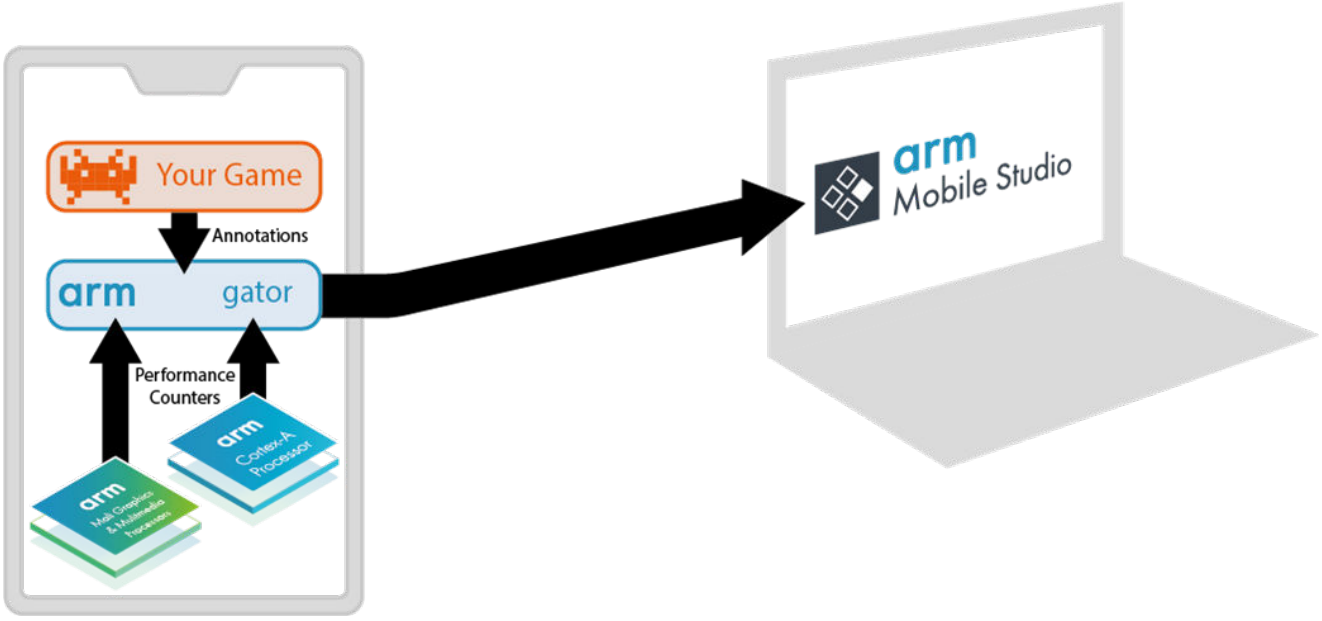


図 3-18 Streamline の仕組み

Unity での Streamline アノテーションの使用方法

Streamline アノテーションでは特定のプロトコルを使用し、オープンソース C 実装が ARM Mobile Studio に付属しています。Unity コンテンツ内から Streamline アノテーションを簡単に生成できるようにするには、C 実装の周囲に C# ラッパーがいくつか必要です。この説明で使用するラッパーは、必要な C 実装とともに、*https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/ArmMobileStudio.unitypackage?raw=true* から Unity アセット パッケージとして入手できます。

このパッケージをダウンロードして、付属の説明書に従ってカスタム アセット パッケージとしてプロジェクトにインポートしてください。これらの手順を表示するには、*https://docs.unity3d.com/Manual/AssetPackages.html* を参照してください。

このパッケージでは、プロジェクトで Streamline アノテーションを簡単に使用するための新しいメソッドが `ARM` 名前空間に追加されています。API ドキュメントは、このパッケージ内の `README.md` ファイル (*https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/InfiniteTerrain/Assets/Arm%20Mobile%20Studio/README.md*) にあります。

3.2.6 Unity プロジェクトの効果的なセットアップ

Streamline を使用して、プロジェクトの設定、マーカーの追加、チャネルの追加、カスタム アクティビティ マップの使用、プロファイルの収集を行うことができます。

迅速かつ簡単に Android ビルドを分析できるようにするには、一部の Android プレーヤ設定を指定します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

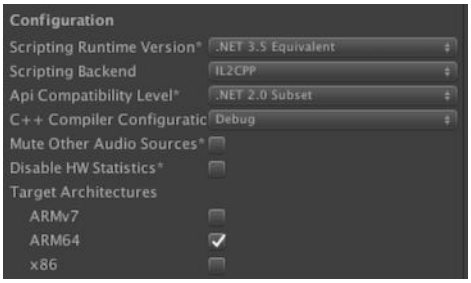


図 3-19 コンフィギュレーション設定

スクリプティング バックエンドとして IL2CPP が使用されていることを確認し、C ++ コンパイラ コンフィギュレーションをデバッグに設定します。これにより、スクリプトがネイティブ コードにコンパイルされてパフォーマンスが向上するだけでなく、Streamline がデバッグ情報を確認して、パフォーマンス データをコール パス ビューの関数にマップできるようになります。

ターゲットアーキテクチャを ARM64 に設定します (デフォルトは ARMv7 です)。最近のほとんどのモバイル デバイスは 64 ビットであるため、結果的にコード生成の品質が向上します。

マーカーの追加

マーカーは、最も使いやすいアノテーションです。指定したメソッドが、文字列と色を受け取ります。たとえば、フレームごとのマーカーを緑色で出力するには、GameObjects のいずれかで次のコードが使用されます (`Update()` メソッドはフレームごとに 1 回自動的に呼び出されます)。

```
void Update ()
{
  Arm.Annotations.marker("Frame " + Time.frameCount, Color.green);
}
```

チャネルの追加

チャネルも簡単に使用できます。まず、チャネルを作成して名前を指定します。次に、Channel オブジェクトのメソッドを使用してそのチャネルにアノテーションを送り込みます。次にサンプルを示します。

```
channel = new Arm.Annotations.Channel("Scene");

channel.annotate(sceneDescription, color);
```

チャネル内のアノテーションは一定の期間に及びます。アノテーションを終了してから次のアノテーションを開始するには、`end()` メソッドを使用します。たとえば、メイン スレッドで Terrain ジョブの完了を行う TerrainController の一部は、次のようにラップされています。

```
// Begin annotation
channel.annotate("Completing", Color.blue);

Mesh mesh = obj.GetComponent<MeshFilter>().mesh;

mesh.vertices = job.vertices.ToArray();
mesh.uv = job.uv.ToArray();
mesh.uv2 = job.uv2.ToArray();

mesh.SetTriangles(job.grassTriangles.ToArray(), 0);
mesh.SetTriangles(job.sandTriangles.ToArray(), 1);
mesh.SetTriangles(job.waterTriangles.ToArray(), 2);
mesh.RecalculateNormals();

// End annotation
channel.end();
```

カスタム アクティビティ マップの使用

カスタム アクティビティ マップ (CAM) は、チャネルの上にある別のレイヤーと考えることができます。まず CAM に名前を付けてから、その中にトラックを作成します。これらのトラックにアノテーションを追加する方法は、チャネルにアノテーションを追加する方法と似ています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

このサンプルでは、地形生成 CAM は次のように作成されています。

```
terrainCAM = new Arm.Annotations.CustomActivityMap("地形生成");

terrainTracks = new Arm.Annotations.CustomActivityMap.Track[16];

for (int i = 0; i < 16; i++)
{
    terrainTracks[i] = terrainCAM.createTrack("TerrainJob " + i);
}
```

ただし、この場合は厄介な問題が 1 つあります。Unity Job System でジョブが実行中である場合、そのジョブはゲームの他のオブジェクト モデルとほとんどやり取りできません。ただし、これはスレッドセーフな状態を確保するうえで役立ちます。そのため、開始時刻と終了時刻をジョブに保存してから、メインスレッドによってジョブがクリーンアップされたときに、ジョブのアクティビティを CAM に登録します。

C# のラッパーには、Streamline のアノテーションで必要とされる形式で現在時刻を返す関数が提供されており、その関数をジョブ内から安全に呼び出すことができます。

```
UInt64 startTime = Arm.Annotations.getTime();
```

メイン スレッドに戻ったときに、使用するトラック (トラックをプールで管理してオーバーラップを防ぐと、視覚的にわかりやすくなります) を選択し、ジョブをそのトラックに登録します。ここで、`job.timings` は、ジョブの開始時刻と終了時刻が格納されている 2 要素の配列です。

```
track.registerJob(obj.name, Color.grey, job.timings[0], job.timings[1]);
```

Streamline での基本的なプロファイルの収集

Streamline でプロファイルの収集を開始する手順:

まず、Windows、Mac、または Linux の無償版の ARM Mobile Studio Starter Edition を *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/downloads?_ga=2.211441174.2077719384.1568032571-433518899.1564491901* からダウンロードしてインストールします。

以下の点に留意してください。

- 実行中の gator がモバイル デバイス上のアプリケーションにアクセスできるようにする必要があります。
- デバイスからデータを取得してツールに渡す手段があることを確認してください。

幅広いデバイスで使用できる堅実かつ最もシンプルな方法は、以下の情報を把握しておくことです。

- アプリケーションのサイズ (32 ビットまたは 64 ビット)。前述の手順に従って、`ARM64` オプションを指定して Unity ゲームをビルドした場合は 64 ビットです。
- アプリケーションのパッケージ名 (Unity の Android プレーヤ設定で指定した名前)。このサンプル アプリケーションの場合、パッケージ名は `com.Arm.InfiniteTerrain` です。
- gator バイナリの入手方法。ARM Mobile Studio のインストール先の `streamline/bin/arm` (32 ビット) または `streamline/bin/arm64` (64 ビット) フォルダにあります。

これらの情報を把握したら、以下の手順に従って分析を実行します。

- アプリケーションがデバイスにインストールされていることを確認します。
- Android の `adb` ツールを使用して、モバイル デバイスから、ARM Mobile Studio を実行しているシステム上のローカル ポートにネットワーク ポートを転送します。
- `adb` を使用して gator (お使いのアプリケーションに応じて 32 ビット版または 64 ビット版のいずれか) をデバイスにプッシュし、アプリケーションと同じパッケージ名を使用して実行を開始します。
- Streamline を起動し、`adb` を使用してトラフィックの転送先に指定したローカル ポートに接続します。収集するパフォーマンス カウンタを選択します。
- モバイル デバイス上でアプリケーションが起動されると、Streamline が分析データを受信し、収集を開始します。

gator が実行中であれば、最新バージョンのアプリケーションをインストールし、Streamline を起動および停止して、gator を再起動することなくさらに分析を実行できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

このプロセスを簡単にするには、gatorme スクリプトを *https://github.com/ARM-software/Tool-Solutions/tree/master/mobile-application-profiling/mobile-studio-scripts* からダウンロードします。このスクリプトを使用して、gator と adb を設定して実行します。これには、実行する gator バイナリへのパス、アプリケーションのパッケージ名、およびデバイスで使用されている Mali GPU が必要です。この情報を提供すると、デバイスのプローブを行っても gator に対して使用している GPU が伝達されない場合に役立ちます。このスクリプトでは、このメソッドが幅広いモバイル デバイスでうまく動作するようにし、プロファイリングの終了時に gator が適切にシャットダウンされるようにするための手順もいくつか実行されます。

詳細は gatorme のドキュメントで説明されていますが、ここではサンプルとして、APK がデバイスにインストールされている場合に InfiniteTerrain のコンテンツをプロファイリングする方法を説明します。

まず、コマンドラインで gatorme を次のように実行します。

```
$ ./gatorme.sh com.Arm.InfiniteTerrain G71 ./mobilestudio-macosx/streamline/bin/arm64/gatord
```

Streamline を起動して、キャプチャの準備をします。Streamline を起動するには、次の設定に従ってください。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

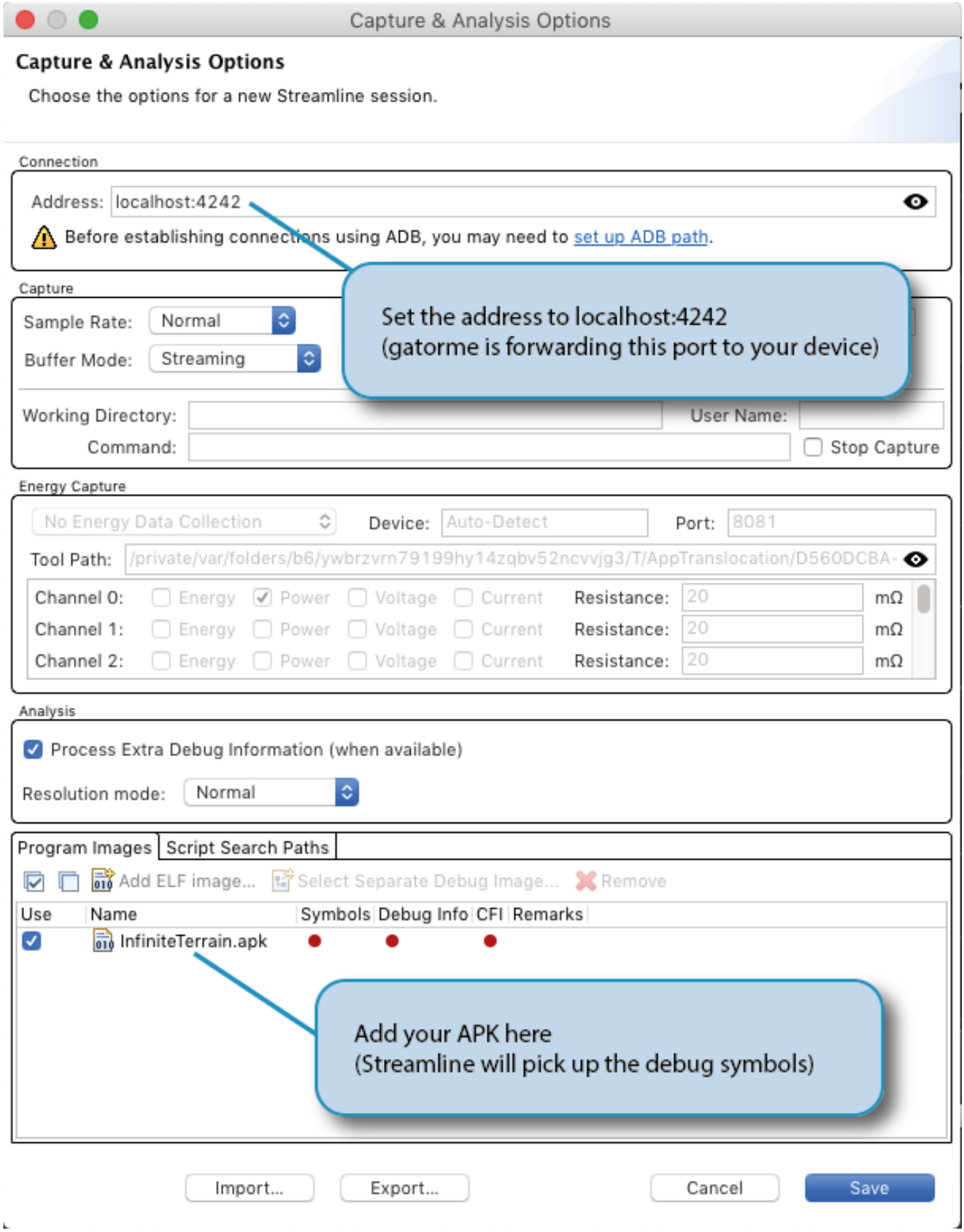


図 3-20 Streamline を起動するための設定

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

設定が完了したら、展開/分析/修正の手順を繰り返すのは簡単です。`gatorme` は実行したままでアプリケーションをシャットダウンし、Unity で [Build And Run] を指示して Streamline の情報をさらにキャプチャすることができます。

InfiniteTerrain サンプルのソースコードは、Apache 2.0 ライセンスに基づいて GitHub からダウンロードできます。ソースコードをダウンロードするには、*https://github.com/ARM-software/Tool-Solutions/tree/master/mobile-application-profiling/mobile-studio-with-unity* にアクセスしてください。

ソースコードとグラフィックス アセットに加え、*https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/ArmMobileStudio.unitypackage?raw=true* からダウンロードできる `ArmMobileStudio.unitypackage` という Unity カスタム アセット パッケージもあります。

Unity では、このパッケージを独自のプロジェクトにインポートして、Streamline アノテーションをパッケージに追加する手順を提供しています。これらの手順を表示するには、*https://docs.unity3d.com/Manual/AssetPackages.html* を参照してください。

ビルド済みの `InfiniteTerrain.apk` をダウンロードするには、*https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/InfiniteTerrain.apk?raw=true* にアクセスします。これは 64 ビット版の Android 開発用ビルドであり、デバイスに展開して、すぐにコンテンツの分析を開始できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 4 章
最適化リスト

本章では、Unity アプリケーションのいくつかの最適化について説明します。

以下のセクションから構成されています。

- *4.1 アプリケーション プロセッサの最適化*(4-43 ページ).
- *4.2 GPU の最適化*(4-49 ページ).
- *4.3 アセットの最適化*(4-70 ページ).
- *4.4 Mali™ オフライン シェーダ コンパイラによる最適化*(4-72 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

4.1 アプリケーション プロセッサの最適化

以下の一覧に、アプリケーション プロセッサの最適化を示します。

Invoke() に代わるコルーチンの使用

`Monobehaviour.Invoke()` メソッドは、時間遅延を伴うクラス内のメソッドを呼び出すための迅速かつ便利な方法ですが、次の制限があります。

- C# のリフレクションを使用して呼び出すメソッドを見つけることができますが、これはメソッドを直接呼び出すよりも時間がかかることがあります。
- メソッド シグネチャはコンパイル時にチェックされません。
- 追加のパラメータを指定することはできません。

次のコードは `Invoke()` 関数を示します。

```
public void Function()
{
    [...]
}
Invoke("Function", 3.0f);
```

代わりにコルーチンを使用することもできます。コルーチンは `IEnumerator` 型の関数で、特別な `yield return` ステートメントを使って Unity に制御を戻すことができます。後でこの関数を呼び出して、以前に中断した箇所から再開することができます。
`MonoBehaviour.StartCoroutine()` メソッドでコルーチンを呼び出すことができます。

```
public IEnumerator Function(float delay)
{
    yield return new WaitForSeconds(delay);
    [...]
}
StartCoroutine(Function(3.0f));
```

`Monobehaviour.Invoke()` メソッドを使用せずに、コルーチンを使用するように変更すると、アニメーションの状態を扱う関数に渡されるパラメータの柔軟性が高まります。

緩やかな更新にコルーチンを使用する

ゲームで特定の時間間隔ごとにアクションが必要な場合には、`MonoBehaviour.Update()` コールバックでフレームごとにアクションを実行するのではなく、`MonoBehaviour.Start()` コールバックでコルーチンを起動してみてください。たとえば、次のようにします。

```
void Update()
{
   // フレームごとにアクションを実行する
}

IEnumerator Start()
{
   while(true)
   {
      // 1/4 秒ごとにアクションを実行する
      yield return new WaitForSeconds(0.25f);
   }
}
```

注

このテクニックの別の使い方は、不規則な間隔で敵を出すことです。敵を出すコルーチン内で無限ループを使って、乱数を生成します。その乱数を `WaitForSeconds()` 関数に渡します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

タグへの文字列のハードコードを避ける

タグの値をハードコードすると、ゲームのスケーラビリティと堅牢性が制限されるため、避けてください。たとえば、タグ名の場合、名前を文字列で直接参照すると、それらの名前を簡単に変更することはできず、スペルミスにさらされる可能性があります。たとえば、

```
if(gameObject.CompareTag("Player"))
{
    [...]
}
```

Public の定数文字列を公開するタグ用の特別なクラスを実装することで、これを改善できます。たとえば、

```
public class Tags
{
    public const string Player = "Player";
    [...]
}

if(gameObject.CompareTag(Tags.Player))
{
    [...]
}
```

注

Public の定数文字列を持つ Tags クラスを使用して、一貫したスケーラブルな方法で新しいタグを追加できます。

固定時間ステップを変更して物理計算の数を減らす

固定時間ステップを変更することで、物理計算の計算負荷を軽減できます。通常、多くの物理計算は一定の時間ステップで実行されます。このステップの長さは増減可能です。
時間ステップを増やすと、アプリケーション プロセッサの負荷は減少しますが、物理計算の精度は低下します。
メイン メニューからタイム マネージャにアクセスできます。[Edit] > [Project Settings] > [Time] の順に選択します。
以下の図に、タイム マネージャを示します。

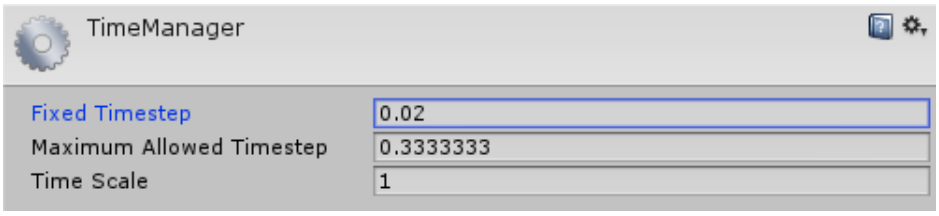


図 4-1 固定 timestep 設定

空のコールバックを削除する

`Awake()`、`Start()`、`Update()` など、関数の空の定義がコードに含まれている場合は、それらを削除します。エンジンは定義が空であってもアクセスしようとするため、関連オーバーヘッドがあります。たとえば、

```
// 次の空の定義を削除します

void Awake()
{

}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

すべてのフレームで GameObject.Find() を使用しない

`GameObject.Find()` は、シーンのすべてのオブジェクトを反復処理する関数です。コードの正しくない部分で使用された場合、メイン スレッドのサイズが大幅に増加する可能性があります。たとえば、

```
void Update()
{
    GameObject playerGO = GameObject.Find("Player");
    playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

より良い方法は、起動時に `GameObject.Find()` を呼び出して、`Start()` や `Awake()` 関数の結果をキャッシュすることです。

```
private GameObject _playerGO = null ;

void Start()
{
    _playerGO = GameObject.Find("Player");
}

void Update()
{
    _playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

別の方法は、`GameObject.FindWithTag()` を使用することです。

```
void Update()
{
    GameObject playerGO = GameObject.FindWithTag("Player");
    playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

注

シーンの読み込みの完了後に直ちにすべてのオブジェクト取得を実行する、`LocatorManager` という専用クラスを使用します。他のクラスはこれをサービスとして使用できます。オブジェクトの複数回の取得は行われません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

StringBuilder クラスを使用して文字列を連結する

複雑な文字列を連結する場合は、`System.Text.StringBuilder` クラスを使用します。これは `string.Format()` メソッドよりも高速で、プラス演算子を使用した連結よりも少ないメモリ使用ですみます。

```
// プラス演算子による連結
string str = "foo" + "bar";

// String.Format() メソッド
string str = string.Format("{1}{2}", "foo", "bar");
```

`System.Text.StringBuilder` クラス:

```
// StringBuilder クラス
using System.Text;

StringBuilder strBld = new StringBuilder();
strBld.Append("foo");
strBld.Append("bar");
string str = strBld.toString();
```

以下の図は、文字列の連結を示しています。

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ StringConcatenationTest.Start()	99.8%	0.0%	1	1.12 GB	2488.86	0.33
▼ StringConcatenationTest.StringFormatMethod()	51.0%	0.2%	1	0.56 GB	1273.01	6.78
▶ String.Format()	50.8%	0.3%	10000	0.56 GB	1266.22	7.51
▼ StringConcatenationTest.PlusConcatenation()	48.4%	0.2%	1	0.56 GB	1208.21	6.53
▶ String.Concat()	48.2%	0.5%	10000	0.56 GB	1201.68	13.15
▼ StringConcatenationTest.StringBuilderClass()	0.2%	0.2%	1	256.2 KB	7.31	6.97
▶ StringBuilder.Append()	0.0%	0.0%	10000	256.1 KB	0.32	0.12
▶ StringBuilder..ctor()	0.0%	0.0%	1	0 B	0.00	0.00
▶ StringBuilder.ToString()	0.0%	0.0%	1	0 B	0.01	0.00

図 4-2 文字列の連結

タグ プロパティの代わりに CompareTag () メソッドを使用する

`GameObject.tag` プロパティの代わりに `GameObject.CompareTag()` メソッドを使用します。`CompareTag()` メソッドは高速で、余分なメモリを割り当てません。

```
GameObject mainCamera = GameObject.Find("Main Camera");

// Gameobject.tag プロパティ
if(mainCamera.tag == "MainCamera")
{
    // アクションを実行する
}

// Gameobject.CompareTag() メソッド
if(mainCamera.CompareTag("MainCamera"))
{
    // アクションを実行する
}
```

次の図は、`CompareTag()` の使用を示します。

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ TagComparisonTest.Start()	97.2%	0.1%	1	3.6 MB	127.48	0.26
▼ TagComparisonTest.TagProperty()	68.0%	59.6%	1	3.6 MB	89.27	78.14
▼ GameObject.get_tag()	7.9%	1.0%	100000	3.6 MB	10.44	1.32
GC.Collect	6.9%	6.9%	4	0 B	9.12	9.12
▼ String.op_Equality()	0.5%	0.3%	100000	0 B	0.69	0.50
String.Equals()	0.1%	0.1%	100000	0 B	0.18	0.18
▶ TagComparisonTest.CompareTagMethod()	28.9%	28.4%	1	0 B	37.94	37.27
GameObject.Find()	0.0%	0.0%	1	0 B	0.00	0.00

図 4-3 比較タグ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

オブジェクト プールを使用する

実行時に作成して破棄される同じ種類の多数のオブジェクトがあるゲームでは、デザイン パターンの*オブジェクト プール*を使用できます。この設計パターンにより、多数のオブジェクトの動的な割り振りと解放に伴う、パフォーマンスの低下を回避できます。
必要なオブジェクトの総数がわかっている場合は、すぐにそれらをすべて作成し、直ちに必要ではないオブジェクトを無効にすることができます。新しいオブジェクトが必要な場合には、プールの中で使用されていない最も古いものを検索して有効化します。
オブジェクトが不要になったときには、プールに戻すことができます。これは、オブジェクトをデフォルトの開始状態にリセットし、無効にすることを意味します。
この手法は、敵、投射物、パーティクルなどのオブジェクトで使用できます。必要なオブジェクトの正確な数がわからない場合は、テストを行って使用数を調べ、見つかった数より少し大きいプールを作成します。

注

敵と爆弾にオブジェクト プールを使用します。これにより、それらのオブジェクトの割り当てはゲームの読み込み時に制限されます。

コンポーネント取得のキャッシュ

`GameObject.GetComponent<Type>()` が返すコンポーネント インスタンスをキャッシュします。関与する関数呼び出しは、相当コストが高いものです。
`GameObject.camera`、`GameObject.renderer`、`GameObject.transform` などのプロパティは、対応する `GameObject.GetComponent<Camera>()`、`GameObject.GetComponent<Renderer>()`、`GameObject.GetComponent<Transform>()` へのショートカットです。

```
private Transform _transform = null;

void Start()
{
    _transform = GameObject.GetComponent<Transform>();
}

void Update()
{
    _transform.Translate(Vector3.forward * Time.deltaTime);
}
```

`Transform.position` の戻り値をキャッシュすることを検討します。C# getter プロパティであっても、グローバルでの位置を計算するために、変換階層上の繰り返しに関連するオーバーヘッドがあります。

注

Unity 5 以降では、変換コンポーネントは自動的にキャッシュされます。

OnBecameVisible() と OnBecameInvisible() コールバックを使用する

`MonoBehaviour.OnBecameVisible()` や `MonoBehaviour.OnBecameInvisible()` などのコールバックは、関連するゲーム オブジェクトが画面上で表示または非表示になった場合に、スクリプトに通知します。
これらの呼び出しを使用すると、たとえばゲーム オブジェクトが画面に表示されないときに、計算量の多いコード ルーチンやエフェクトを無効にすることができます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ベクトルの大きさの比較に sqrMagnitude を使用する

アプリケーションでベクトルの magnitude の比較が必要な場合は、`Vector3.Distance()` や `Vector3.magnitude` ではなく、`Vector3.sqrMagnitude` を使用します。
`Vector3.sqrMagnitude` は、平方根を計算せずに 2 乗されたコンポーネントを合計しますが、これは比較に役立ちます。他の呼び出しでは、計算コストの高い、平方根を使用しています。
次のコードは、空間内の 2 つの位置を比較する際に使用される、3 つの異なる手法を示しています。

```
// Vector3.sqrMagnitude プロパティ
if ((_transform.position - targetPos).sqrMagnitude < maxDistance * maxDistance)
{
    // アクションを実行する
}

// Vector3.Distance() メソッド
if (Vector3.Distance(transform.position, targetPos) < maxDistance)
{
    // アクションを実行する
}

// Vector3.magnitude プロパティ
if ((_transform.position - targetPos).magnitude < maxDistance)
{
    // アクションを実行する
}
```

組み込み配列を使用する

あらかじめ配列のサイズがわかっている場合には、組み込みの配列を使用します。
`ArrayList` クラスと `List` クラスは、挿入する要素が増えるほどサイズが大きくなるため、柔軟性は高くなりますが、組み込み配列よりも処理速度は遅くなります。

平面を衝突ターゲットとして使用する

シーンで床や壁などの平面オブジェクトとパーティクルの衝突のみが必要な場合は、パーティクル システムの衝突モードを `Planes` に変更します。平面を使用するように設定を変更すると、必要な計算量が少なくなります。このモードでは、Unity に空の GameObjects のリストを指定して、コライダー平面として動作させることができます。
以下の図に、衝突の設定を示します。

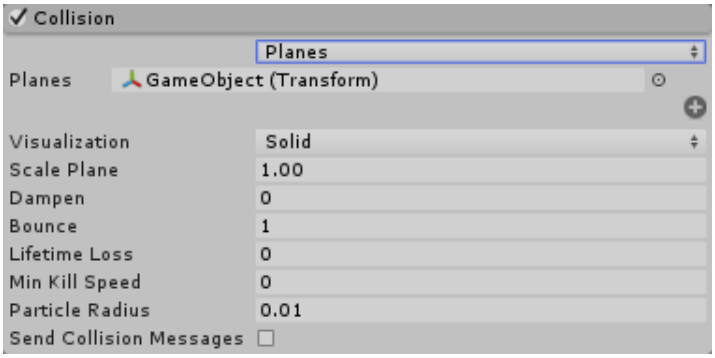


図 4-4 競合設定

メッシュ コライダーではなく複合プリミティブ コライダーを使用する

メッシュ コライダーは、オブジェクトの実際のジオメトリに基づいています。これは衝突検出に関して非常に正確ですが、計算コストは高くなります。
元のメッシュの形状を模した複合コライダーに、ボックス、カプセル、または球などの形状を組み合わせることができます。これによって、計算上のオーバーヘッドを大幅に削減しながら、同様の結果を得ることができます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

4.2 GPU の最適化

このセクションでは、GPU の最適化について説明します。

以下のサブセクションから構成されています。

- *4.2.1 その他の GPU 最適化*(4-49 ページ).
- *4.2.2 ライトマップとライト プローブ*(4-50 ページ).
- *4.2.3 ASTC テクスチャ圧縮*(4-59 ページ).
- *4.2.4 ミップマッピング*(4-63 ページ).
- *4.2.5 スカイボックス*(4-64 ページ).
- *4.2.6 影*(4-64 ページ).
- *4.2.7 オクルージョン カリング*(4-65 ページ).
- *4.2.8 OnBecameVisible() と OnBecomeInvisible() コールバックの使用*(4-66 ページ).
- *4.2.9 レンダリング順序を指定する*(4-67 ページ).
- *4.2.10 デプス プリパスを使用する*(4-68 ページ).

4.2.1 その他の GPU 最適化

以下の一覧にその他の GPU 最適化を示します。

静的バッチの使用

静的バッチは、ドローコールの回数を減らす一般的な最適化手法であり、これによってアプリケーション プロセッサの使用率を低下できます。
動的バッチは Unity によって透過的に実行されますが、計算オーバーヘッドが大きくなりすぎるため、多数の頂点で構成されるオブジェクトには適用できません。
静的バッチは、多数の頂点で構成されるオブジェクトでも機能しますが、バッチ処理されるオブジェクトは、レンダリング中に移動、回転、または拡大縮小を行うことはできません。
Unity が静的バッチのためにオブジェクトをグループ化できるようにするには、それらを Inspector で静的としてマークします。
以下の図に、静的バッチの設定を示します。

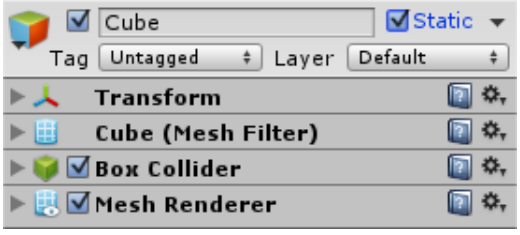


図 4-5 静的バッチ設定

4 倍の MSAA の使用

ARM Mali GPU は、非常に低い計算オーバーヘッドで、4 倍のマルチサンプル アンチエイリアシングを実行できます。
Unity の `[Quality Settings]` で 4 倍の MSAA を有効化します。
以下の図に、MSAA の設定を示します。

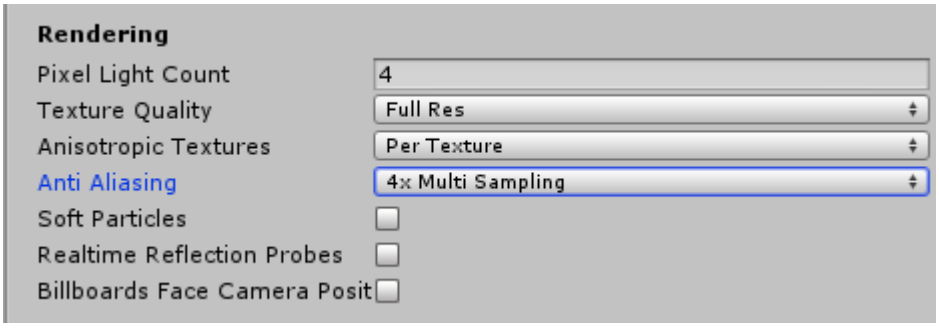


図 4-6 MSAA 設定

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

詳細レベルを使用する

詳細レベル (LOD) は、Unity エンジンがカメラからの距離に応じて、同じオブジェクトに対して異なるメッシュをレンダリングする手法です。
オブジェクトがカメラの近くにあるとき、ジオメトリはより詳細になります。オブジェクトがカメラから遠ざかるにつれて詳細レベルが下がり、最も遠い距離では平面のビルボードを使用できます。使用するメッシュおよび関連する距離範囲を管理するには、LOD グループを適切に設定する必要があります。
LOD グループの設定にアクセスするには、[Add Component] > [Rendering] > [LOD Group] の順に選択します。
Unity 5 では、各 LOD レベルにフェード モードを設定して、連続した LOD にブレンドすることができます。これにより、レベル間の遷移が滑らかになります。Unity は、オブジェクトの画面サイズに応じてブレンド係数を計算し、シェーダに渡してブレンドします。ジオメトリ ブレンディングをシェーダに実装する必要があります。
次の図は、LOPD グループの設定を示しています。

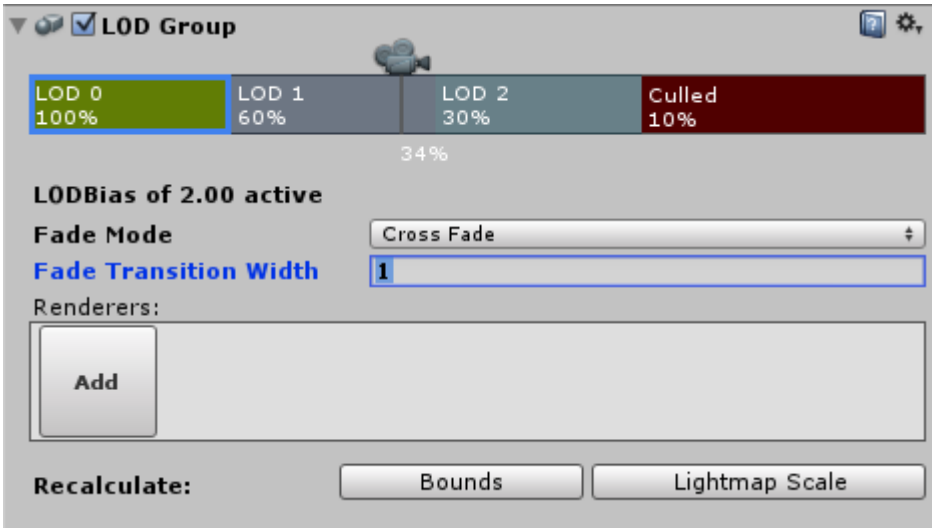


図 4-7 LOD グループ設定

カスタム シェーダでの数学関数の回避

カスタム シェーダを作成するときは、次のような、コストの高い組み込みの数学関数を使わないようにします。

- `pow()`
- `exp()`
- `log()`
- `cos()`
- `sin()`
- `tan()`

4.2.2 ライトマップとライト プローブ

ランタイム ライティングの計算はコストが高くなります。ライトマッピングと呼ばれる、計算要件を軽減するためによく使われる手法では、ライティング計算が事前に計算され、ライトマップと呼ばれるテクスチャにベイク処理されます。

これにより、完全な動的ライティング環境による柔軟性は失われますが、パフォーマンスに影響を与えることなく非常に高品質な画像を生成できます。

結果のライティングを静的ライトマップにベイクする方法:

- ライティングを受けるジオメトリを [Static] に設定します。
- ライトの [Baking] オプションを [Realtime] ではなく [Baked] に設定します。
- Lightmapping ウィンドウの [Scene] タブで [Baked GI] オプションを選択します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

結果のライトマップを表示する方法:

- ジオメトリを選択します。
- [Window] > [Lighting] の順に選択して、Lighting ウィンドウを開きます。
- [Object] ボタンを押します。
- プレビュー オプションで [Baked Intensity lightmap] を選択します。

[Continuous Baking] オプションが選択されている場合、Unity は数秒でライトマップをベイクしてエディタでシーンを更新します。

ライトマップが正しく設定されていることを確認する簡単な方法は、エディタでゲームを実行して、ライトを無効にすることです。それでもライティングが存在する場合には、ライトマップが正しく作成されて使用されています。

次の図は、[Lighting] タブの強度ライトマップを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

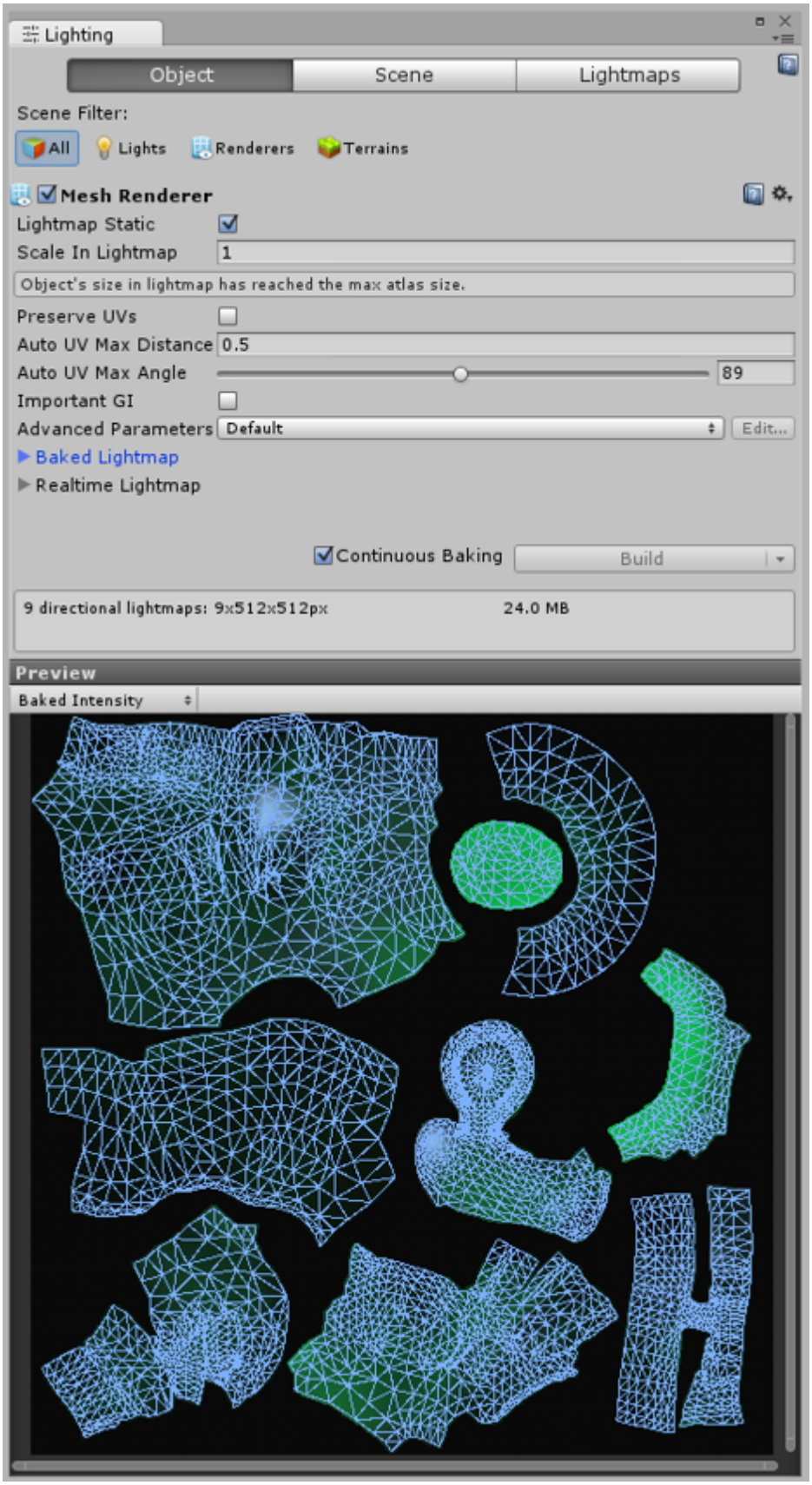


図 4-8 強度ライトマップ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

次の図は、洞くつの奥の緑色のライトからのライティングを表示するエディタを示しています。このライティングは、静的なライトマップで生成されています。

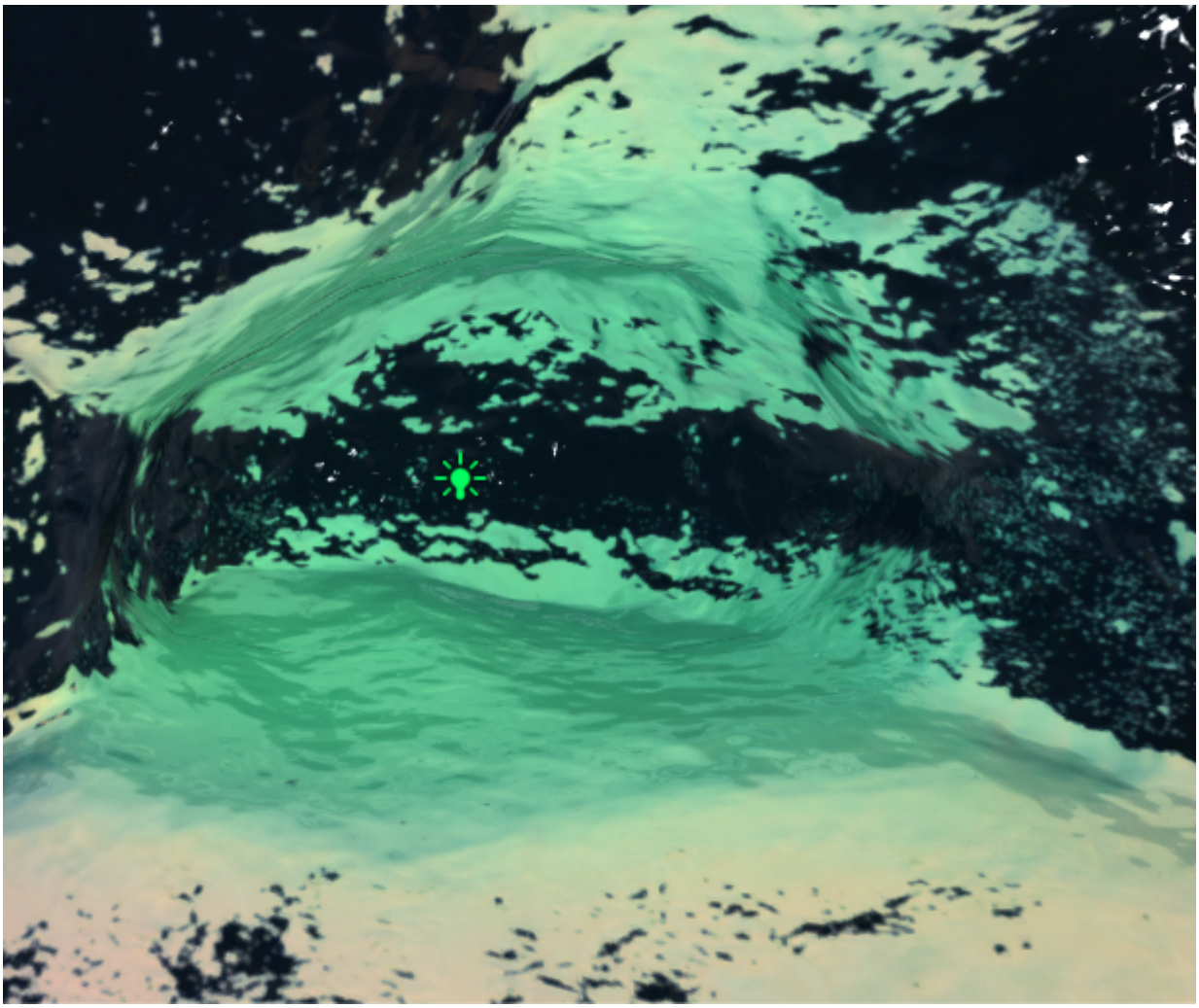


図 4-9 ライトを追加して静的なライトマップをベイクする

次の図は、Ice Cave デモの静的ライトマップの結果を示しています。

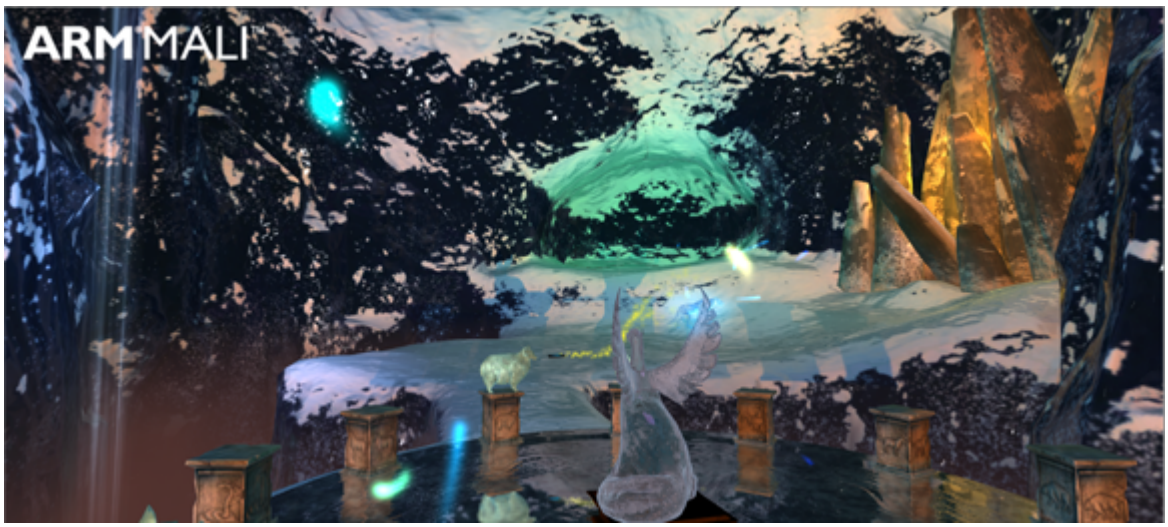


図 4-10 ライトマップを使った洞くつ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ライトマッピングの設定

ライトマッピングのためのオブジェクトを準備するには、次のものが必要です。

- ライトマップ UV を使ったシーンでのモデル。
- モデルは**ライトマップ静的**としてマークする必要があります。
- モデルの範囲内にライトがある必要があります。
- ライトの [Baking type] は `[Baked]` に設定されている必要があります。

——— 注 ———

シーン内の静的オブジェクトだけがライトマップされます。これらは完璧ではない可能性があるので、ゲームに最適なものを試してみてください。

静的としてマークされていないオブジェクトは、ライトマップに配置されません。レンダラーを選択すると、いくつかの設定が提供され、ライトマップが静的であるかどうかを設定できます。

———

エディタ ウィンドウのメイン メニューから [Lighting] ウィンドウを開き、[Window] と [Lightmapping] を選択します。次の 3 つのボタンがあります。

- **Object**
- **Scene**
- **Lightmaps**

次の図は、ライトマップのオプションを示しています。

図 4-11 ライトマップのオプション

Object

[Object] ボタンをクリックすると、階層で選択したオブジェクトのライトマップに関連する設定を変更できます。これにより、ライトマッピング プロセスに影響する、オブジェクト設定を変更できます。ライトを選択し、いくつかのオプションを変更することができます。

- [Baked Only] は、ベイク時にライトを有効にし、実行時にそれを無効にします。
- [Baked If Baked GI] を選択すると、ライトはベイクされます。
- [Realtime] このライトは、事前計算されたリアルタイム GI と GI なしの両方で機能します。
- [Realtime Only] は、ベイク時にライトを無効にし、実行時に有効にします。
- [Mixed] ライトはベイクされますが、実行時にも存在し、静的でないオブジェクトに直接ライティングを与えます。

大半のライトを [Baked] に設定すると、実行時の計算数が比較的少なくなります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

Scene

[Scene] タブには、シーン全体に適用される設定が含まれています。このタブでは、事前計算されたリアルタイム GI および Baked GI 機能を有効または無効にすることができます。

[Environment Lighting] セクションには、Skybox、Ambient Source のタイプ、Ambient Intensity など、環境照明に影響を及ぼすいくつかの要因を定義できる、さまざまなオプションがあります。

- [Reflection Bounces] オプションは、パフォーマンスの観点から最も重要です。[Reflection Bounces] は、反射オブジェクト間の相互反射の数、つまりオブジェクトを見るプローブのベイクの回数を定義します。リフレクション プローブが実行時に更新されると、このオプションはパフォーマンスに大きな悪影響を及ぼす場合があります。反射物がプローブにはっきりと見える場合のみ、バウンス数を 1 より大きく設定してください。
- [Precomputed Realtime GI] タブの [CPU Usage] オプションは、実行時に GI を評価するために使われるプロセッサ時間の量を定義します。[CPU Usage] の値が高いほど、ライティングからの反応は速くなりますが、フレーム レートに影響する可能性があります。マルチプロセッサ システムでは、パフォーマンスへの影響は低くなります。
- [Baked GI] タブには、ライトマップ テクスチャを圧縮するように設定できるオプションがあります。ライトマップ テクスチャを圧縮すると、実行時に必要な記憶領域と帯域幅が少なくなりますが、圧縮プロセスによってアーティファクトがテクスチャに追加される可能性があります。
- [General GI] タブでは [Directional Mode] オプションに注意します。デュアル ライトマップで遅延照明を使用できない場合は、別の方法として、指向性ライトマップを使用できます。これにより、リアルタイム ライトなしで通常のマッピングと反射照明を使用することができます。通常のマッピングを維持する必要があるが、デュアル ライトマップを使用できない場合は、指向性ライトマップを使用します。これは通常、モバイル デバイスで当てはまります。[Directional Mode] を [Directional] に設定すると、入射光の優勢な方向を保存するための、追加のライトマップが作成されます。その結果、このモードでは約 2 倍の記憶スペースが必要になります。
- [Directional Specular] モードでは、鏡面反射および法線マップ用の追加データが格納されます。この場合、ストレージ要件は 4 倍になります。
- [Lightmaps] タブでは、シーンに使用されるライトマップのアセット ファイルの設定および検索が可能です。ライトマップのスナップショット ボックスにアクセスするには、[Continuous Baking] オプションをオフにする必要があります。

次の図は、[Lighting] タブのライトマップを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

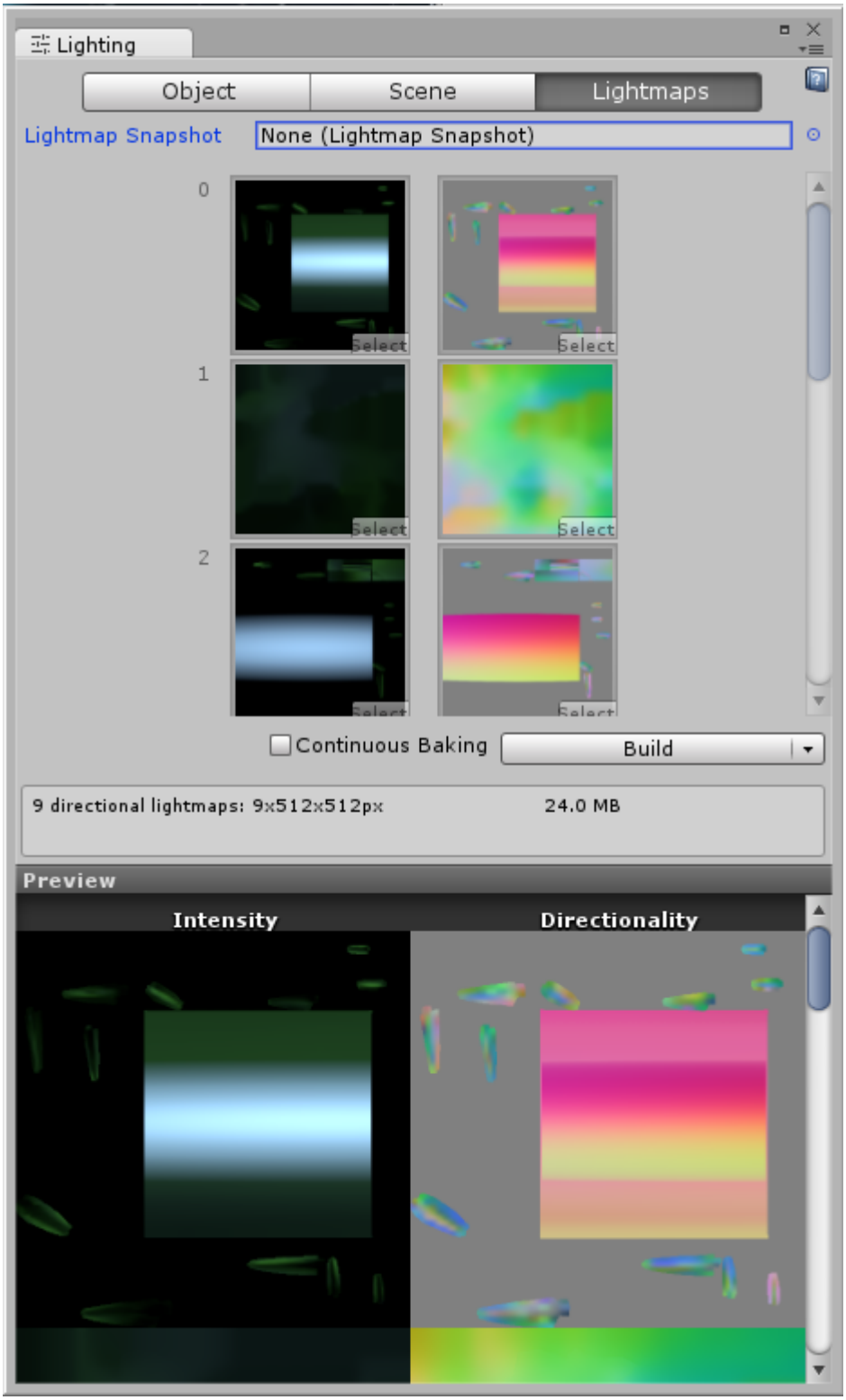


図 4-12 [Lighting] タブのライトマップ

指向性ライトマップを使用する

デュアル ライトマップで遅延照明を使用できない場合は、別の方法として、指向性ライトマップを使用できます。これにより、リアルタイム ライトなしで通常のマッピングと反射照明を使用することができます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

通常のマッピングを維持する必要があるが、デュアル ライトマップを使用できない場合は、指向性ライトマップを使用します。これは通常、モバイル デバイスで当てはまります。

——— 注 ———

この手法は、方向情報を格納するための第 2 のライトマップ セットを計算するため、より多くのビデオ メモリを必要とします。

ゲーム内の動的オブジェクトにライト プローブを使用する

ライト プローブを使用すると、ライトマッピングされたシーンに動的なライティングを追加することができます。

次の図は、ライト プローブの設定を示しています。

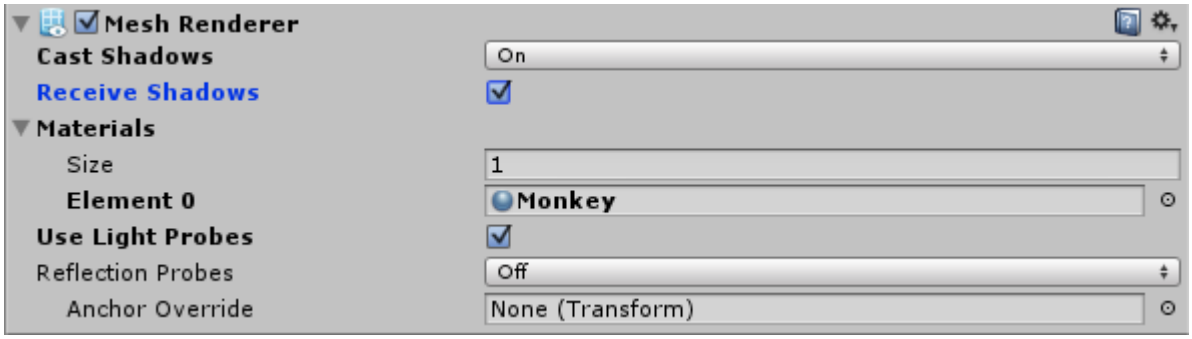


図 4-13 ライト プローブ設定

ライト プローブは、エリア内のライトのサンプルまたはプローブを取得します。プローブがボリュームまたはセルを形成する場合、ライティングはセル内の位置に応じてそれらのプローブ間で補間されます。

次の図は、ライト プローブを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

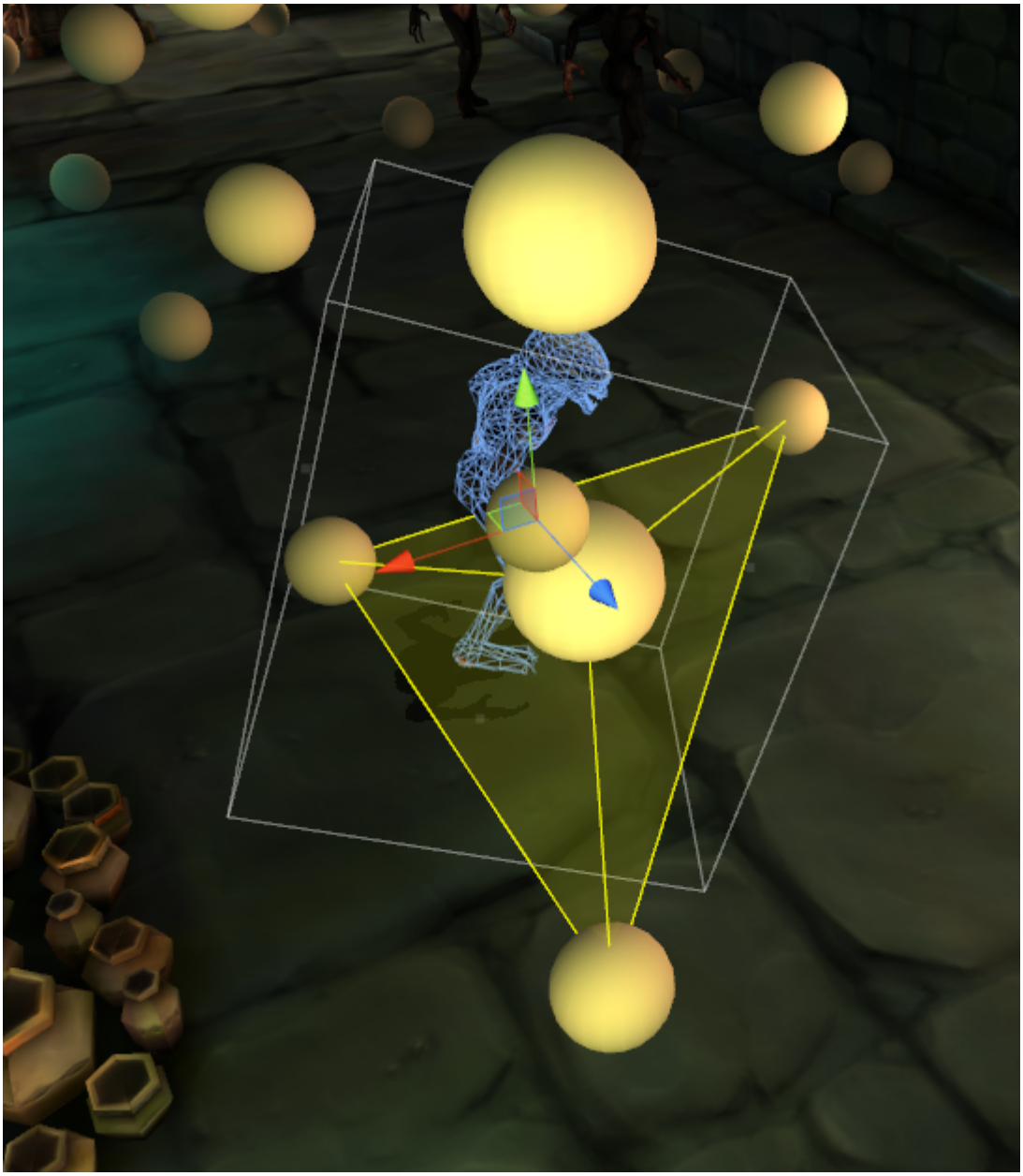


図 4-14 ライト プローブ

より多くのプローブが存在するほど、より正確なライティングが得られます。プローブ間の補間があるので、通常は多くのライト プローブを必要としません。光のカラーや強度が大きく変化するエリアでは、より多くのライト プローブが必要になります。

最も近いプローブによって取得されたサンプル間を補間することによって、任意の位置のライティングを近似できます。

ライト プローブを配置し、`[Use Light Probes]` オプションを使用して、影響を受けるメッシュをマークします。

次の図は、複数のライト プローブを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

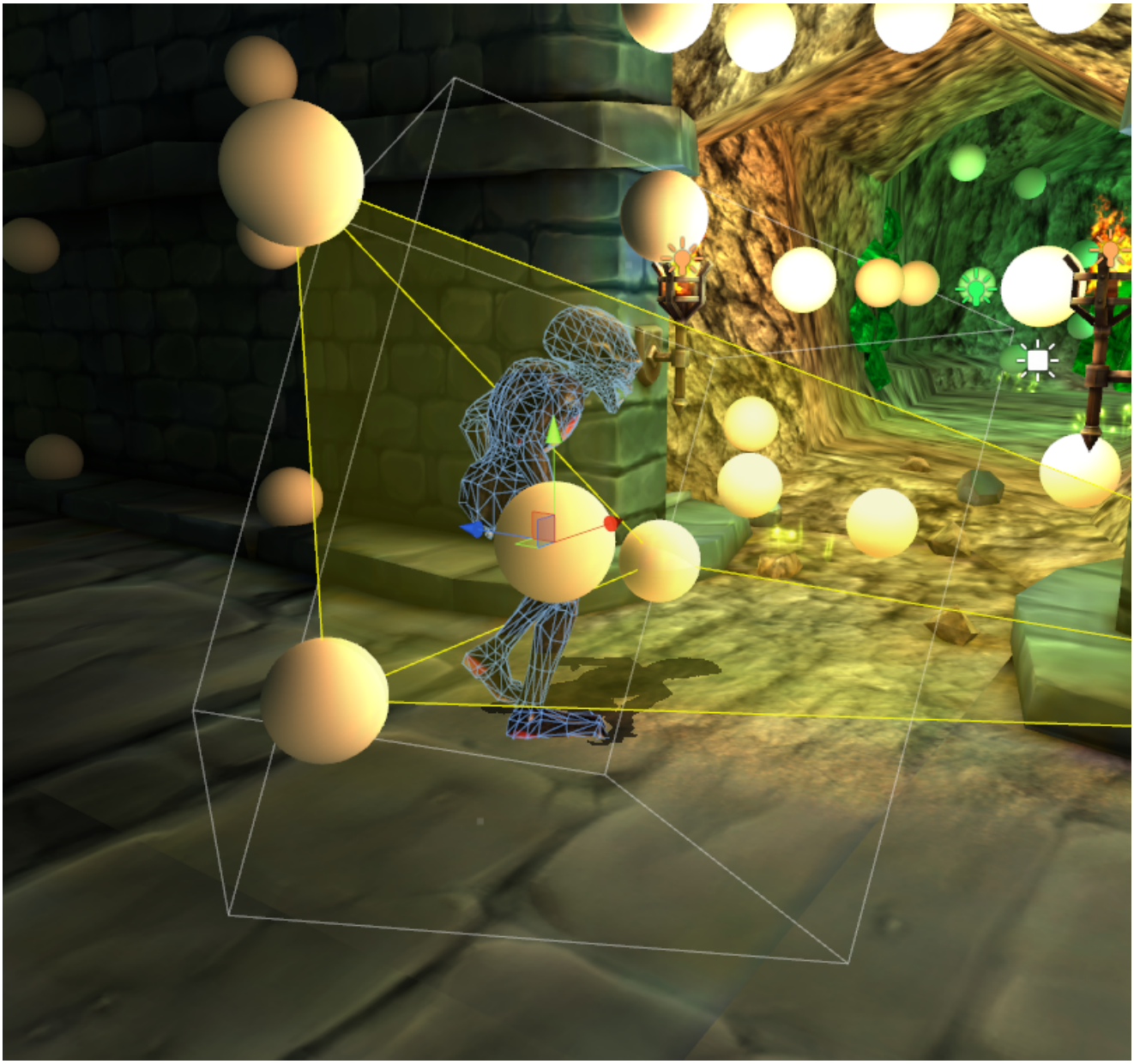


図 4-15 複数のライト プローブ

4.2.3 ASTC テクスチャ圧縮

ASTC テクスチャ圧縮は、OpenGL および OpenGL ES グラフィックス API の正式な拡張です。ASTC はアプリケーションが必要とするメモリを削減し、GPU が必要とするメモリ帯域幅を削減することができます。

ASTC は高品質で低ビットレートのテクスチャ圧縮を提供し、多くの制御オプションを備えています。これには以下の機能があります。

- ビットレートは、*ピクセル当たり* 8 ビット (bpp) から 1 bpp 未満に及びます。これにより、ファイル サイズと品質のトレードオフを微調整することができます。
- 1 ー 4 色のチャネルをサポートします。
- *低ダイナミック レンジ* (LDR) と *高ダイナミック レンジ* (HDR) の両方の画像をサポートします。
- 2D および 3D 画像をサポートします。
- さまざまな機能の組み合わせの選択をサポートします。

次の図は、ASTC 設定ウィンドウを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 4-16 ASTC 設定

ASTC 設定ウィンドウでは、いくつかのブロック サイズ オプションを使用できます。これらのオプションから選択して、アセットに最適なブロック サイズを選択することができます。ブロック サイズが大きいほど、圧縮率が高くなります。たとえば、カメラから遠く離れたオブジェクトなど、詳細には表示されないテクスチャには、大きなブロック サイズを選択します。カメラに近いテクスチャなど、より詳細に表示されるテクスチャには、小さなブロック サイズを選択します。

注

- デバイスが ASTC をサポートする場合は、それを使用して 3D コンテンツのテクスチャを圧縮します。デバイスが ASTC を サポートしない場合は、ETC2 を試してみてください。
- 3D コンテンツで使用されるテクスチャと、GUI エレメントで使用されるテクスチャを区別する必要があります。場合によっては、望ましくないアーティファクトを避けるために、GUI テクスチャは非圧縮とする方がよい可能性があります。

次の図は、さまざまなテクスチャ圧縮形式で使用できるブロック サイズを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

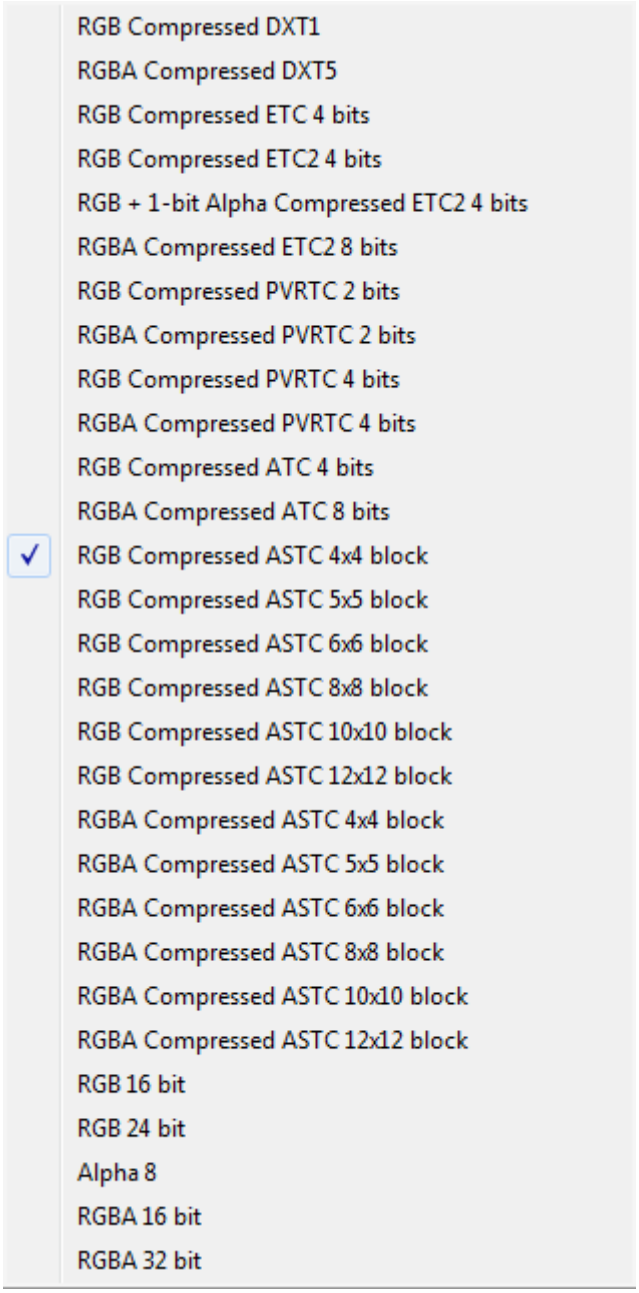


図 4-17 テクスチャ圧縮のブロック サイズ

ASTC テクスチャの正しいフォーマットの選択

ASTC テクスチャを圧縮する場合、いくつかのオプションを選択できます。

テクスチャ圧縮アルゴリズムには、通常は RGB と RGBA という、異なるチャネル フォーマットがあります。ASTC はいくつかの他のフォーマットをサポートしていますが、これらのフォーマットは Unity では公開されていません。各テクスチャは通常、標準テクスチャ、法線マッピング、鏡面反射、HDR、アルファ、およびルック アップ テクスチャなど、異なる目的に使用されます。これらのテクスチャ タイプはすべて、最良の結果を得るために、異なる圧縮フォーマットが必要です。

以下の図に、テクスチャの設定を示します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

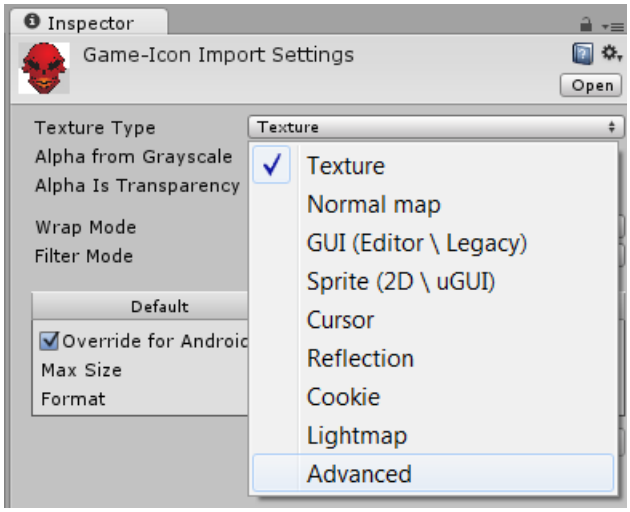


図 4-18 テクスチャ設定

[Build Settings] の 1 つの形式を使ってすべてのテクスチャを圧縮しないでください。テクスチャ圧縮は [Don't Override] のままとします。

プロジェクト階層内のテクスチャを見つけて、[Inspector] で表示します。Unity は通常、テクスチャを [Texture] タイプとしてインポートします。このタイプは圧縮に限定されたオプションのみを提供します。より多くのオプションを表示するには、タイプを [Advanced] に設定します。

次の図は、透過性のある GUI テクスチャの設定を示しています。このテクスチャは GUI 用なので、[sRGB] および [MipMaps] は無効になっています。透明度を含めるには、アルファ チャネルが必要です。これを有効にするには、[Alpha Is Transparency] のチェックボックスをオンにし、[Override for Android] のチェックボックスをオンにします。

以下の図に、高度なテクスチャの設定を示します。

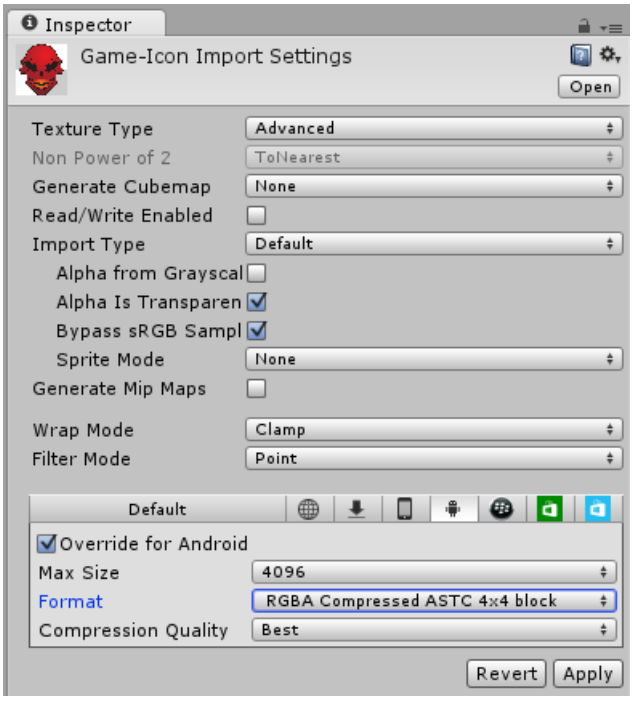


図 4-19 テクスチャの詳細設定

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

フォーマットとブロック サイズを選択するオプションがあります。RGBA にはアルファ チャネルが含まれ、4x4 は選択できる最小のブロック サイズです。[Max texture size] を最大に設定し、[Compression Quality] を設定します。この設定は、正確な圧縮を探すために費やされる時間を定義します。

すべてのテクスチャに特定の設定を選択すると、プロジェクトのビジュアル品質が向上し、圧縮時に不必要なテクスチャ データを避けることができます。

次の表は、チャネルあたり RGBA 8 ビット、ピクセルの解像度 1024×1024、サイズ 4 MB の テクスチャについて、Unity で利用可能な ASTC ブロック サイズでの圧縮率を示しています。

表 4-1 Unity で使用可能な ASTC ブロック サイズでの圧縮率

ASTC ブロック サイズ	サイズ	圧縮率
4x4	1 MB	4.00
5x5	655 KB	6.25
6x6	455 KB	9.00
8x8	256 KB	16.00
10x10	164 KB	24.97
12x12	144 KB	35.93

4.2.4 ミップマッピング

ミップマッピングは、ビジュアルの品質とゲームのパフォーマンスを向上させることができる、テクスチャに関連する手法です。

ミップマップは、異なるサイズのテクスチャを事前計算したものです。生成された各テクスチャはレベルと呼ばれ、前のものの半分の幅と半分の高さです。Unity は、元の最初のサイズのレベルから、1x1 ピクセル バージョンまで、レベルの完全なセットを自動的に生成することができます。

ミップマップを生成するには、次の手順を実行します。

1. [Project] ウィンドウでテクスチャを選択します。
2. [Texture Type] を [Advanced] に変更します。
3. [Inspector] の [Generate Mip Maps] オプションを有効化します。

以下の図に、ミップマップの設定を示します。

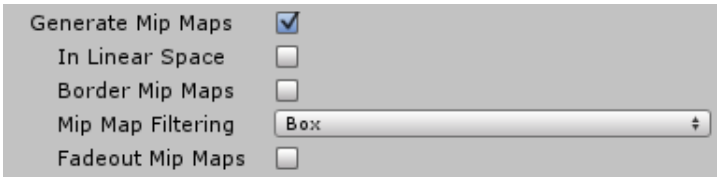


図 4-20 ミップマップ設定

テクスチャにミップマップ レベルがない場合で、テクスチャ サーフェスで覆われたピクセルのエリアがテクスチャのサイズよりも小さい場合には、GPU はテクスチャの小さいエリアに合わせてテクスチャをスケール ダウンします。しかし、ピクセル カラーを補間するフィルタを用いても、このプロセスではある程度の精度が失われます。

テクスチャにミップマップ レベルがある場合、GPU はオブジェクト サイズに最も近いレベルからピクセル データをフェッチして、テクスチャをレンダリングします。これにより、レベルをオフラインでスケールして品質を向上させ、GPU は適切なレベルのテクスチャ データのみをフェッチするため、イメージ品質を向上でき、帯域幅を低減できます。ミップマップのデメリットは、テクスチャ データを保管するために、メモリを 33% 余分に必要とすることです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ミップマップと GUI テクスチャ

通常、2D UI で使用されるテクスチャには、ミップマップは必要ありません。UI テクスチャは通常、スケーリングなしで画面にレンダリングされるため、ミップマップ チェーンの最初のレベルのみが使用されます。

この設定を変更するには、[Project] ウィンドウでテクスチャを選択し、[Inspector] に移動し、[Texture Type] を確認します。タイプを [Editor GUI and Legacy GUI] に設定するか、またはタイプを [Advanced] に設定して [Generate Mip Maps] オプションを無効にします。

4.2.5 スカイボックス

スカイボックスはゲームや他のアプリケーションでよく使用されますが、それらを実装する方法はいくつかあります。

スカイボックスを描画するには、単一のキューブマップを使用して、カメラの背景をレンダリングします。

これには、1 つのキューブマップ テクスチャと 1 回のドローコールが必要です。これは他の方法と比較して、メモリ、メモリ帯域幅、ドローコールの使用が少なくてすみます。

この方法でスカイボックスを設定するには:

1. カメラを選択します。
2. [Clear Flags] が [Skybox] に設定されていることを確認します。
3. スカイボックスのコンポーネントを選択または追加します。

スカイボックスのコンポーネントには、マテリアルのスポットが 1 つあります。これは各フレームの開始時に、カメラの背景を描画するために、Unity が使用するマテリアルです。

使用するマテリアルは、必要なすべての情報を含むマテリアルです。[Mobile] > [Skybox] シェーダを使用してマテリアルを作成し、スカイボックスの 6 つの画像にマテリアルを指定します。マテリアル プレビューに画像が表示されます。

マテリアルが完成したら、それをスカイボックス コンポーネントにドラッグします。スカイボックスは、バックグラウンドで正しくレンダリングされ、はっきりした継ぎ目や不必要なドローコールはありません。

4.2.6 影

影はシーンに遠近感と臨場感を加えるために役立ちます。影がない場合、オブジェクトの深さを伝えるのが難しい場合があります。特に周囲のオブジェクトに似ている場合は難しくなります。

影のアルゴリズムは非常に複雑となる場合があります。特に高解像度の影の正確なレンダリングは複雑となります。ゲームの影に適切な複雑性と解像度を選択するようにします。

Unity は、リアルタイムでの影の計算のための、変換フィードバックをサポートしています。

注

Ice Cave デモでは、カスタム シャドウを実装しています。ローカル キューブマップに基づく影と、実行時にレンダリングされる影を結合しています。

Unity は、[Edit] > [Project Settings] > [Quality]で、影のためのいくつかのオプションを提供しており、これはゲームのパフォーマンスに影響を与える可能性があります。

Hard/Hard and Soft Shadows
: Soft Shadows はよりリアルに見えますが、計算に時間がかかります。

Shadow Distance
: Shadow Distance オプションは、影を写すがカメラからの距離を定義します。シャドウ距離を大きくすると、表示される影の数が多くなり、計算負荷が増加します。またシャドウ距離を大きくすると、シャドウ マップで影が使用できるテクセル数が増加するため、影の解像度が増加します。

Hard Shadow は、影の距離が小さく、解像度が高い場合に使用できます。カメラからの距離が近いところに、あまり複雑ではない、妥当な品質の影を生成します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ライトマッピングされたオブジェクトは、リアルタイムの影を生成しません。そのため、シーンに静的な影をベイクする数を増やすと、GPU が実行するリアルタイム計算は少なくなります。

次の図は、影の付いたエイリアンのキャラクタを示しています。

図 4-21 影の付いたエイリアン

リアルタイムの影を慎重に使用する

リアルタイムの影は、シーンの臨場感を飛躍的に向上させることができますが、計算コストが高くなります。

モバイル デバイスでは、リアルタイムのみの影を含むライトの数を制限し、代わりにライトマッピングを使用してみてください。

シーン内のオブジェクトのメッシュ レンダラー コンポーネントについて検討します。影の投影または受け取りにそれらを使用しない場合は、それに応じて [Cast Shadows] および [Receive Shadows] のオプションを無効にします。これにより、影をレンダリングする計算コストが削減されます。

[Quality Settings] セクションには、さらに次のような影の設定があります。

- [Shadow Resolution] では、品質と処理時間のバランスを選択できます。
- [Shadow Distance] を使用すると、影の生成をカメラに近いオブジェクトに限定できます。
- [Shadow Cascades] を使用すると、品質と処理時間のバランスを選択できます。これは 0、2、または 4 に設定できます。カスケード シャドウ マップは指向性ライトに使用すると、非常に良好なシャドウクオリティを得られます。特に表示距離が長い場合に有効です。カスケード数が多いほど品質は向上しますが、処理オーバーヘッドが増加します。

4.2.7 オクルージョン カリング

オクルージョン カリングは、オブジェクトがカメラの視野から隠れているときに、オブジェクトのレンダリングを無効にするプロセスです。これにより、レンダリングするオブジェクトの数を減らして、GPU の処理時間を節約できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

オブジェクトがカメラの錐台から完全に外れると、Unity は自動的に錐台のカリングを実行しますが、アプリケーションのスタイルによっては、表示されずレンダリングの必要のないオブジェクトが残る場合があります。

Unity には、Umbra と呼ばれるオクルージョン カリング システムが含まれています。Umbra の詳細については、Unity ドキュメントのオクルージョン カリングを参照してください。

オクルージョン カリングに使用する設定は、ゲームのスタイルによって異なります。シーンで誤った設定によりオクルージョン カリングを使用するとパフォーマンスが低下する可能性があるため、設定を慎重に選択する必要があります。

4.2.8 OnBecameVisible() と OnBecomeInvisible() コールバックの使用

`MonoBehaviour.OnBecameVisible()` および `MonoBehaviour.OnBecameInvisible()` コールバックを使用すると、Unity は 関連するゲーム オブジェクトがカメラの錐台の中に出入りするときに、スクリプトに通知します。アプリケーションはそれを使って適切な動作を行うようにすることができます。

`OnBecameVisible()` および `OnBecameInvisible()` を使用してレンダリング プロセスを最適化できます。たとえば、第 2 のカメラとレンダリング ターゲットを使用して、水たまりへの反射をレンダリングできます。

これにはジオメトリをレンダリングし、最終的な画面にレンダリングする前に、画面の外のテクスチャを組み合わせることが含まれます。この手法は比較的コストが高いため、必要なときにのみ使用するようにします。反射のレンダリングは、表示されているときのみ行う必要があります。それは次の場合です。

- 反射面が、カメラ錐台内にある。
- 表面の前に不透明なものがない。

これらの条件は、反射面からの `OnBecameVisible()` と `OnBecameInvisible()` コールバックを使用してチェックされます。

```
void OnBecomeVisible()
{
    enabled = true;
}

void OnBecomeInvisible()
{
    enabled = false;
}
```

これらのチェックが行われていても、反射が画面外にレンダリングされる場合があります (画面上では見えませんが)。これを避けるには、別の条件を追加することができます。

たとえば、カメラが反射面の室内にある必要があるようにします。

```
void OnBecomeVisible()
{
    if (inside == false)
    {
        return;
    }
    enabled = true;
}

void OnBecomeInvisible()
{
    if (inside == false)
    {
        return;
    }
    enabled = false;
}

void OnTriggerEnter()
{
    inside = true;
}

void OnTriggerExit()
{
    inside = false;
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

これらの条件により、ゲームの特定のエリアへの反射のレンダリングを制限します。これは、ゲームで計算の少ない他のエリアにエフェクトを追加できることを意味します。

4.2.9 レンダリング順序を指定する

シーンにおけるオブジェクトのレンダリング順序は、パフォーマンスにとって非常に重要です。

オブジェクトがランダムな順序でレンダリングされた場合、あるオブジェクトがレンダリングされた後で、その前にある別のオブジェクトによって閉塞される場合があります。これは、閉塞されたオブジェクトをレンダリングするために必要なすべての計算が無駄になることを意味します。

閉塞されたオブジェクトのための無駄な計算を減らすための、さまざまなソフトウェアやハードウェアの手法がありますが、シーンでプレーヤーがどのように動くかを知っているゲームの作成者は、無駄な計算を減らすためのプロセスを進めることができます。

Mali -T600 シリーズ以降の ARM Mali GPU で利用可能な、無駄な計算を減らすためのハードウェア手法の 1 つが Early-Z です。Early-Z は作成者にとって完全に透過的なシステムで、フラグメント シェーダが実際に処理される前に Z テストを実行します。GPU が Early-Z 最適化を有効化できない場合には、フラグメント シェーダの後にデプス テストが実行されます。これは計算コストが高く、フラグメントが閉塞されると、計算が無駄になる場合があります。Early-Z システムでは、処理中のピクセルのデプスが、より近いピクセルによって占有されていないことをチェックします。占有されている場合、フラグメント シェーダは実行されません。このシステムは性能上の利点をもたらしますが、場合によっては自動的に無効になります。たとえば、フラグメント シェーダが `gl_FragDepth` 変数に書き込んでデプスを変更する場合や、フラグメント シェーダが `discard` を呼び出す場合や、透過オブジェクトなどのオブジェクトに対してブレンドまたはアルファ テストが有効になっている場合などです。このシステムが最大の効率を発揮するようにするには、不透明なオブジェクトが前面から背面にレンダリングされるようにします。これにより、不透明なオブジェクトのみを持つシーンで、オーバードローの要因を減らすのに役立ちます。

各フレームで前面から背面への順序でレンダリングを行うことはコストが高くなる可能性があります。また透過オブジェクトを同じパスでレンダリングすると、結果が正しくない場合があります。ARM Mali T620 以降の GPU は、*Pixel Forward Kill* (PFK) と呼ばれるメカニズムを提供します。 Mali GPU はパイプライン化されているため、同じピクセルに対して複数のスレッドを同時に実行できます。スレッドがその実行を完了すると、現在のスレッドがそのピクセルの他のすべてのスレッドをカバーする場合、PFK システムはそれらを停止します。これにより、無駄な計算量の削減できます。

Unity は、シェーダ内またはマテリアル内の Queue オプションを提供します。これにより、レンダリングの順序を指定できます。これはシェーダで設定できるため、そのシェーダを使用するマテリアルを持つオブジェクトは一緒にレンダリングされます。このレンダリング グループ内では、透明性などのいくつかの場合を除いて、レンダリングの順番はランダムです。

デフォルトでは、Unity は次の順序で最初から最後までレンダリングされる、いくつかの標準グループを提供します。

表 4-2 レンダリング順序を指定するためのキューの値

名前	値	備考
Background	1000	-
Geometry	2000	デフォルト。不透明なジオメトリに使用されます。
AlphaTest	3000	すべての不透明なオブジェクトの後に描画されます。たとえば、木の葉などです。
Transparent	4000	このグループは、正しい結果を得るために、後ろから前への順序でレンダリングされます。
Overlay	5000	ユーザ インタフェース、レンズ フレア、ダーティ レンズ などのオーバーレイ エフェクトです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

文字列名の代わりに整数値を使用できます。これらは利用可能な唯一の値ではありません。表示されている値の間の整数値を使用して、他のキューを指定できます。数値が高いほど、後でレンダリングされます。

たとえば、次のいずれかの手順を使用して、Geometry キューの後で AlphaTest キューの前に、シェーダをレンダリングできます。

```
Tags { "Queue" = "Geometry+1" }
```

```
Tags { "Queue" = "2001" }
```

レンダリング順序を使用してパフォーマンスを向上させる

Ice Cave デモでは、洞くつは画面の大部分をカバーしており、そのシェーダはコストが高いものです。可能な場合はその一部のレンダリングを避けることにより、パフォーマンスが向上します。

Unity フレーム デバッガや、Graphics Analyzer などのその他のツールを使用して、フレームバッファの構成を調べた後、レンダリング順序の最適化が行われました。これにより、レンダリングの順序を確認できます。

Unity **フレーム デバッガ**を開くには、[Window] > [Frame Debugger]のメニュー オプションを選択します。エディタ モードでは正しいように見えるが、実行すると正しく動作しないものもあるため、これは便利です。たとえば、実行時のみの設定がある場合や、別のカメラのテクスチャにレンダリングしている場合などです。デモをプレイ モードで開始してカメラを配置した後、フレーム デバッガを有効にして、Unity によって実行される描画のシーケンスを取得できます。

Ice Cave デモでは、ドローコールを下にスクロールすると、洞くつが最初にレンダリングされたことが示されます。それから、既にレンダリングされた洞くつの一部を閉塞する、さまざまなオブジェクトがシーンにレンダリングされます。もう 1 つの例は、いくつかの場面で洞くつに閉塞される、反射するクリスタルです。このような場合には、レンダリング順序を高く設定すると、閉塞されたクリスタルにはフラグメント シェーダが実行されないため、計算量が減少します。

4.2.10 デプス プリパスを使用する

オーバードローを避けるためにオブジェクトのレンダリング順序を設定すると便利ですが、各オブジェクトのレンダリング順序を指定することは必ずしも可能ではありません。

たとえば、計算コストの高いシェーダを持つ一連のオブジェクトがあり、カメラがその周囲を自由に回転できる場合、後ろにあったオブジェクトの一部が前に移動する場合があります。この場合、これらのオブジェクトに対して静的なレンダリング順序が設定されていると、閉塞されていても最後に描画される可能性があります。これは、オブジェクトがそれ自身の一部を覆う場合にも発生します。

このような場合は、デプス プリパスを使用して、オーバードローを減らすことができます。デプス プリパスは、フレームバッファにカラーを書き込まずに、ジオメトリをレンダリングします。これにより、各ピクセルのデプス バッファが、最も近い可視オブジェクトのデプスで初期化されます。このプリパスの後、ジオメトリは通常どおりにレンダリングされますが、Early-Z 手法を使用すると、最終的なシーンに影響するオブジェクトだけが実際にレンダリングされます。頂点シェーダはオブジェクトごとに 2 回、デプス バッファを満たすために 1 回、実際のレンダリングのためにもう 1 回計算されるため、この手法では余分な頂点シェーダの計算が必要です。この手法は、ゲームでフラグメントがボトルネックになっていて、頂点シェーダには余裕がある場合に、非常に便利です。

Unity で、カスタム シェーダを持つオブジェクトのレンダリング プレパスを実行するには、シェーダに余分なパスを追加します。

```
// デプス バッファのみにレンダリングする余分なパス
 Pass {
    ZWrite On
    ColorMask 0
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

このパスを追加した後では、フレーム デバッガはオブジェクトが 2 回レンダリングされたことを示します。初めてレンダリングされるときには、カラー バッファに変更はありません。

注

フレーム デバッガの左上メニューで選択して、デプス バッファを表示できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

4.3 アセットの最適化

以下の一覧にアセットの最適化を示します。

静的テクスチャの読み出し/書き込みを無効にする

テクスチャを動的に変更しない場合は、**インスペクタ**の [Read/Write Enabled] オプションが無効になっていることを確認してください。

メッシュを結合してドローコールを減らす

レンダリングに必要なドローコールの数を減らすためには、`Mesh.CombineMeshes()` メソッドを使って、複数のメッシュを 1 つに結合できます。すべてのメッシュが同じマテリアルを共有する場合は、`mergeSubMeshes` 引数を `true` に設定して、結合グループ内の各メッシュから 1 つのサブメッシュを生成します。
複数のメッシュを 1 つの大きなメッシュに結合すると、次のことが可能になります。

- より効果的なオクルーダーを作成できます。
- タイルベースのアセットを単一の大きなシームレスなアセットに変換できます。

メッシュ結合スクリプトは、パフォーマンスの最適化に役立つ場合もありますが、シーンの構成によって異なります。大きなメッシュは小さなメッシュよりも長い時間視界にとどまる傾向があります。適切なサイズが得られるように試行してください。

この手法を適用する 1 つの方法は、階層に空のゲーム オブジェクトを 1 つ作成して、それを結合するすべてのメッシュの親とし、スクリプトにアタッチすることです。

メッシュ結合スクリプトの詳細については、以下の Unity ドキュメントを参照してください。

http://unity3d.com

アニメーションのない FBX メッシュ モデルのアニメーション データをインポートしない

アニメーション データが含まれていない FBX メッシュをインポートする場合、インポート設定の [Rig] タブで [Animation Type] を [None] に設定できます。これが設定されている場合、メッシュを階層に配置するときに、Unity は使用しないアニメータ コンポーネントを生成しません。

読み出し/書き込みメッシュを避ける

実行時にモデルが変更された場合、Unity はメッシュ データのオリジナルを保持しながら、第 2 のコピーをメモリ内に持ち、それを変更します。
実行時にモデルが変更されていない場合や、スケーリングされている場合でも、インポート設定の [Model] タブから [Read/Write Enabled] オプションを無効にできます。これにより第 2 のコピーは不要となり、メモリが節約できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

テクスチャ アトラスの使用

テクスチャ アトラスを使用すると、一連のオブジェクトに必要なドローコールの数を減らすことができます。

テクスチャ アトラスは、1 つの大きなテクスチャにまとめられた、テクスチャのグループです。複数のオブジェクトは、このテクスチャを適切な座標セットで再利用できます。これは Unity が同じマテリアルを共有するオブジェクトに適用する、自動バッチ処理に役立ちます。

オブジェクトの UV テクスチャ座標を設定するときは、そのマテリアルの `mainTextureScale` プロパティおよび `mainTextureOffset` プロパティを変更しないでください。変更した場合、バッチ処理で機能しない、新たな一意のマテリアルが作成されます。`MeshFilter` コンポーネントを介してメッシュ データにアクセスし、`Mesh.uv` プロパティを使用して、頂点ごとの座標を変更します。

以下の図にテクスチャ アトラスを示します。

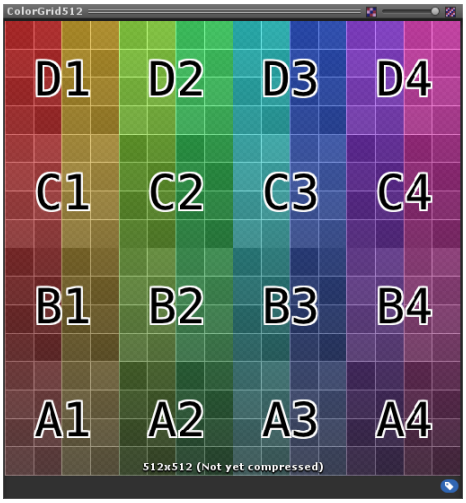


図 4–22 テクスチャ アトラス

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

4.4 Mali™ オフライン シェーダ コンパイラによる最適化

Mali オフライン シェーダ コンパイラは、頂点シェーダ、フラグメント シェーダ、コンピュート シェーダをバイナリ形式にコンパイルするためのツールです。コンパイラをプロファイリング ツールとして使用することもできます。

以下のサブセクションから構成されています。

- *4.4.1 Mali™ オフライン シェーダ コンパイラについて*(4-72 ページ).
- *4.4.2 Unity シェーダの測定*(4-72 ページ).
- *4.4.3 統計情報の分析*(4-73 ページ).
- *4.4.4 Mali™ GPU パイプラインの最適化*(4-73 ページ).
- *4.4.5 パイプライン サイクルを減らすためのその他の手法*(4-74 ページ).

4.4.1 Mali™ オフライン シェーダ コンパイラについて

アプリケーションのシェーダは GPU 上で実行されます。このため、GPU でシェーダの最終的な結果(頂点の位置やピクセルのカラーなど) を計算する時間がかかります。

Mali オフライン シェーダ コンパイラは、Mali GPU の各パイプラインで実行するためにシェーダが必要とするサイクル数に関する情報を提供します。

生成されるサイクル値は、特定の GPU に合わせて調整されます。コマンドラインでオプションとして GPU を選択します。アプリケーションでターゲットにするデバイスの範囲に対応する GPU を選択します。これにより、ツールから得られる統計が現実的で、通常のユースケース シナリオに合ったものになります。

4.4.2 Unity シェーダの測定

Unity シェーダは、プログラミング言語 *C for Graphics* (Cg) で記述します。Cg はプログラミング言語 C に基づき GPU プログラミングに適合するための変更が加えられています。

Unity は、ビルド処理中に Cg を OpenGL、OpenGL ES、または DirectX に変換します。

OpenGL ES のシェーダ コードを取得するには:

1. Unity で分析を行うシェーダを選択します。
2. ビルドを行うカスタム プラットフォームとして、OpenGLES30 または OpenGLES20 を選択します。
3. [Compile and show] ボタンをクリックします。

結果は開発環境に表示されます。

注

- Mali オフライン シェーダ コンパイラは、OpenGL ES シェーダのみをサポートしています。
- ビルド プラットフォームが Android に設定されている場合、Unity はデフォルトで OpenGLES30 シェーダをビルドします。

頂点セッションとフラグメント セッションは通常、`#ifdef VERTEX` または `#ifdef FRAGMENT` によって区切られます。`#pragma multi_compile <FEATURE_OFF> <FEATURE_ON>` などのオプションを使用すると、複数のシェーダ バリアントがファイルに組み込まれます。

通常、複数の `VERTEX` セクションと `FRAGMENT` セクションがあります。Unity がアプリケーションを起動すると、各バリアントが個別にコンパイルされます。各機能を有効にすると、関連するバリアントが選択されます。

コードは OpenGL ES に変換されているので、頂点とフラグメントのシェーダ コードを 2 つの別々のファイルにコピーして、Mali オフライン シェーダ コンパイラでコンパイルすることができます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

次のいずれかのオプションを使って、シェーダをコンパイルします。

- 頂点シェーダに -v
- フラグメント シェーダに -f

次の図は、Mali オフライン シェーダ コンパイラの出力を示しています。

```
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.


8 work registers used, 13 uniform registers used, spilling not used.

                A       L/S     T       Total   Bound
Cycles:         43      11      13      67      A
Shortest Path:  16      11      13      40      A
Longest Path:   16      11      13      40      A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

図 4-23 Mali オフライン シェーダ コンパイラの出力

4.4.3 統計情報の分析

Mali オフライン シェーダ コンパイラによって生成された統計は、シェーダが必要とする頂点またはピクセルあたりのサイクル数の測定を提供します。

結果は 3 行に分かれています。

- 合計。
- 最短パス。
- 最長パス。

最短および最長のパスは、コード内でエフェクトが分岐を取るかどうかを調べることによって測定されます。これにより実行サイクルの最小数および最大数の推定値を提供します。

算術演算の場合、最初の行の測定値は算術演算パイプラインの数で除算されます。これは、Mali GPU によって、1、2、または 3 です。

2 行目と 3 行目は、ロード/ストアとテクスチャ パイプライン用です。これらはキャッシュ ミスを考慮しないため、より現実的な見積もりのためには、これらの数値に 1.5 を乗じます。

4.4.4 Mali™ GPU パイプラインの最適化

Mali オフライン シェーダ コンパイラは、各パイプラインで使用されるサイクル数を提供します。このシェーダは、サイクル数が最も多いパイプラインで最も遅くなります。最も遅いパイプラインをターゲットとする最適化を使用して、シェーダを最適化します。

Mali GPU には、次の 3 種類の処理パイプラインがあります。

- 算術演算パイプライン。
- ロード/ストア パイプライン。
- テクスチャ パイプライン。

パイプラインはすべて並列に実行されます。シェーダは通常、3 種類のパイプラインをすべて使用します。

算術演算パイプライン

すべての算術演算は、算術演算パイプラインのサイクルを消費します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

算術演算パイプラインの使用を減らすいつくかの方法は次のとおりです。

- 次のような複雑な算術演算は避けるようにします。
  — 逆行列関数。
  — モジュロ演算子。
  — 除算。
  — 行列式。
  — サイン (正弦)。
  — コサイン (余弦)。
- 整数オペランドの場合、シフト演算による除算、モジュロ、乗算などの演算を使用します。
- 直交行列の逆行列の代わりに転置を使用します。
- 転置の計算を避けるため、行列の 1 つが転置されている場合は、行列とベクトルまたは行列と行列の乗算のオペランドの順序を切り替えます。たとえば、

```
Transpose(A)*Vector == Vector * A.
```

他のパイプラインに負荷を移動して、算術演算パイプの負荷を減らすこともできます。

- 行列を計算する代わりに、ユニフォームとして渡します。これはロード/ストア パイプラインを使用します。
- テクスチャを使用して、サインやコサインなどの関数を表す、事前計算された値のセットを格納します。これにより負荷をテクスチャ パイプラインに移動できます。

ロード/ストア パイプライン

ロード/ストア パイプラインは、ユニフォーム バッファ オブジェクトやシェーダ ストレージ バッファ オブジェクトなどの、シェーダ内のユニフォームの読み取り、可変変数の書き込み、バッファへのアクセスに使用されます。

アプリケーションでロード/ストア パイプラインがボトルネックになっている場合は、次の方法を試してみてください。

- バッファ オブジェクトの代わりにテクスチャを使用して、シェーダ内のデータを読み込みます。
- 算術演算を使用してデータを計算します。
- ユニフォームと可変変数を圧縮するか、または減らします。

テクスチャ パイプライン

テクスチャ アクセスは、テクスチャ パイプラインのサイクルを使用し、メモリ帯域幅を使用します。大きなテクスチャを使用すると、キャッシュ ミスが発生する可能性が高く、データの待機中に複数のスレッドが停止する可能性があるため、問題が生じる場合があります。

テクスチャ パイプラインのパフォーマンスを向上させるには、以下を試してみます。

ミップマップの使用
: ミップマップは、テクスチャ座標の変化に基づいて使用するテクスチャの最適な解像度を選択するため、キャッシュ ヒット率を向上させます。

テクスチャ圧縮の使用
: これもメモリ帯域幅を削減し、キャッシュ ヒット率を向上させるために適しています。圧縮された各ブロックには複数のテクセルが含まれているため、アクセスするとキャッシュ可能性が向上します。

三線形または異方性フィルタリングを避ける
: 三線形および異方性フィルタリングは、テクセルをフェッチするために必要な操作の数を増加させます。必須でない限り、これらの手法を使用しないようにします。

4.4.5 パイプライン サイクルを減らすためのその他の手法

各パイプラインで使用されるサイクルを減らすために使用できる、いくつかの追加の手法があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

レジスタあふれを避ける

Mali オフライン シェーダ コンパイラは、シェーダによりレジスタあふれが生じるかどうかを示します。レジスタあふれは通常、レジスタ セットに収まりきらない多数の変数によって、スレッド内で発生します。

レジスタあふれは通常、以下のものが多数、スレッドで生じることによって発生します。

- 入力ユニフォーム。
- 可変変数。
- 一時変数。

変数が高精度である場合にも、レジスタあふれが発生することがあります。

レジスタあふれによって強制的に、Mali GPU はメモリからいくつかのユニフォームを読み取ります。これによりロード/ストア ユニットの負荷が増加し、パフォーマンスが低下します。この問題を解決するには、シェーダに提供されるユニフォームの数と精度を下げてみます。

たとえば、Ice Cave デモでは、シェーダの中にはレジスタあふれに苦労したものがあります。

```
8 work registers used, 16 uniform registers used, spilling used.
```

図 4-24 レジスタあふれが生じたシェーダ。

許可されたユニフォームの数を減らすことでこの問題は解決され、その結果、パフォーマンスが向上します。たとえば、次のようになります。

```
8 work registers used, 13 uniform registers used, spilling not used.
```

図 4-25 レジスタあふれのないシェーダ。

可変変数とユニフォームの精度を下げる

カスタム シェーダを作成する場合、32 ビット浮動小数点数または 16 ビット半精度浮動小数点を使用して、ユニフォームと可変変数の浮動小数点の精度を指定できます。この精度は、変数が表すことができる値の最小値と最大値、および粒度を決定します。

半精度浮動小数点の使用には、いくつかの利点があります。

- 使用帯域幅が削減されます。
- シェーダ コンパイラによるコードの最適化によって、より並列化を活用できるため、算術演算パイプラインでの使用サイクルが減少します。
- 必要なユニフォーム レジスタの数が減少するため、レジスタあふれのリスクが低減されます。

次のコードでは、Ice Cave デモの単純なフラグメント シェーダのバリアントのサンプルを示します。このシェーダは Mali オフライン シェーダ コンパイラで 2 回コンパイルされています。

第 1 のサンプル コードは、浮動小数点でコンパイルされています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
$ malisc -f -V Compiled-CaveMaliStandardFloat.frag
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.


8 work registers used, 13 uniform registers used, spilling used.

                A       L/S     T       Total   Bound
Cycles:         47      15      10      72      A
Shortest Path:  15      13      9       37      A
Longest Path:   16      15      10      41      A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

図 4-26 浮動小数点でコンパイルされたシェーダ

第 2 のサンプル コードは、半精度浮動小数点でコンパイルされています。

```
$ malisc -f -V Compiled-CaveMaliStandardHalf.frag
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.


7 work registers used, 7 uniform registers used, spilling not used.

                A       L/S     T       Total   Bound
Cycles:         47      11      10      68      A
Shortest Path:  15      9       9       33      A
Longest Path:   15      11      10      36      A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

図 4-27 半精度浮動小数点でコンパイルされたシェーダ

半精度浮動小数点の方では、ロード/ストア命令の数が減ります。使用されるワーク レジスタとユニフォーム レジスタの数が減り、レジスタあふれはありません。

半精度浮動小数点で生成されたコードは、浮動小数点で生成されたコードよりも小さくなります。これにより、Mali GPU のキャッシュ ヒット率が向上し、パフォーマンスが向上します。

静的オブジェクトにワールド空間の法線マップを使用する

接空間の法線マップを使用すると、ジオメトリの詳細を増やすことなく、モデルの詳細を増やすことができます。メッシュの各三角形への局所性のため、アニメーション化されたオブジェクト上の接空間の法線マップを、変更することなく使用することができます。

残念なことに、これらで正しい結果を得るためには、シェーダでより多くの算術演算を実行する必要があります。静的オブジェクトの場合、これらの計算は通常は不要です。

代わりに、ローカル空間の法線マップまたはワールド空間の法線マップを使用することもできます。ローカル空間の法線マップを使用すると、シェーダで実行される計算の数は減りますが、モデル上の変換をサンプリングされた法線に適用する必要があります。ワールド空間の法線マップは変換を必要としませんが、これらは静的であり、オブジェクトは移動できません。Ice Cave デモでは、洞くつやその他の高

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

品質のオブジェクトは静的であり、ワールド空間の法線マップを使用すると、シェーダが必要とする ALU 演算の数が大幅に減ります。多くの一般的な 3D モデリング ツールでは、ワールド空間の法線マップを作成することも、オフライン プロセスでコードによって生成することもできます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 5 章
リアルタイム 3D アートのベスト プラクティス: ジオメトリ

この章では、3D アセットに対する主なジオメトリの最適化について説明します。ジオメトリ最適化は、ゲームの効率を高めるだけでなく、モバイル プラットフォームでのゲームのパフォーマンスを向上させるという総合的な目標を達成するうえで役立ちます。

注意

このコンテンツの最新バージョンは、Arm の開発者 Web サイト(*https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/real-time-3d-art-best-practices-geometry*)にあります。

以下のセクションから構成されています。

- *5.1 ジオメトリとは*(5-79 ページ).
- *5.2 三角形とポリゴンの使用*(5-80 ページ).
- *5.3 詳細レベル*(5-86 ページ).
- *5.4 ジオメトリに関するその他のベスト プラクティス*(5-91 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

5.1 ジオメトリとは

ジオメトリ、つまりポリゴン メッシュは、3D オブジェクトのシェイプを構成する頂点、エッジ、面の集まりです。この 3D オブジェクトは、車、武器、環境、キャラクタ、あるいはビデオ ゲームで使用するあらゆるタイプのアセットである場合があります。

3D オブジェクトのジオメトリを構成する 3 つの要素は、次のとおりです。

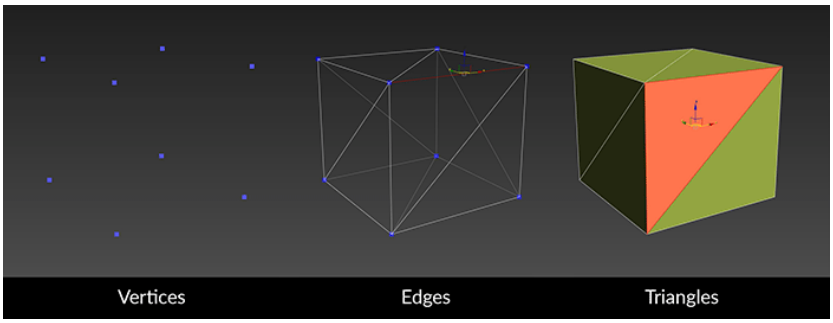


図 5-1 3D オブジェクトを構成する 3 つの要素

頂点

頂点は、3D オブジェクトのサーフェスを構成する点です。

エッジ

エッジは、2 つの頂点を直線で結ぶと得られます。

三角形

三角形は、3 本のエッジでつながった 3 つの頂点で構成されます。ポリゴンまたは面と呼ばれることもあります。Max、Maya、または Blender などの 3D ソフトウェア内では、クワッドと呼ばれる四角ポリゴンをよく使用します。これらの 3D プログラムでは、クワッドの方が簡単に変更や作業を行うことができます。画面にレンダリングすると、これらのポリゴンはすべて三角形として表示されます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

5.2 三角形とポリゴンの使用

ゲームのパフォーマンスを最適化するには、画面上の三角形の数を常に把握しておくことが重要です。

3D オブジェクト (モデル) の意図した品質との適切なバランスを取りつつ、一貫したパフォーマンスを実現するには、使用する三角形の数を最小限に抑えることが重要です。

以下のヒントを試してみてください。

- 使用する三角形の数が少ないほど、パフォーマンスが向上します。
  - モバイル プラットフォーム向けのコンテンツを作成する際は、三角形の数に注意を払うことが不可欠です。
  - 三角形の数を少なくすると、GPU が処理する必要のある頂点が少なくなります。
  - 頂点の処理では計算量が多くなります。そのため、処理される頂点の数が少ないほど、全体のパフォーマンスが向上します。
- 使用する三角形の数を少なくすると、より多くのデバイスでゲームをリリースでき、強力な GPU を備えた最新のデバイスだけに限られなくなります。

次の画像は、2 つの 3D オブジェクト間の比較を示しています。1 つは 584 個の三角形を使用したもの、もう 1 つは 704 個の三角形を使用したものです。シェーディング モードでは、両方のオブジェクトの外観が同じです。そのため、シルエットに寄与していないエッジは削除します。

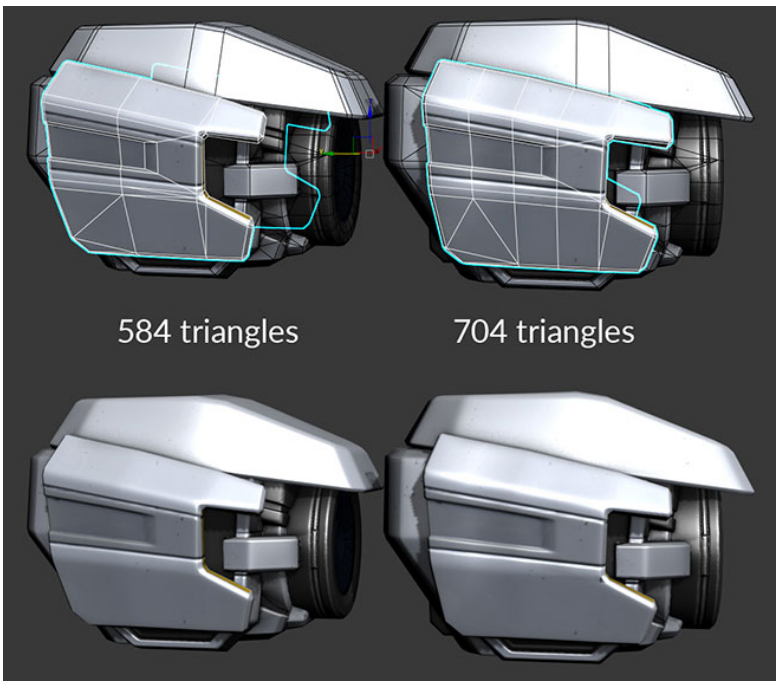


図 5-2 使用する三角形の数を最小限に抑えて意図した結果を得る

モバイル デバイスでは、各 3D オブジェクト (メッシュ) が使用できる頂点の最大数は 65,535 個です。そのため、頂点の数をこの値よりも少なく保つ必要があります。Mali-400 を使用する Android デバイスなど、一部の旧式の GPU は、この値を超えた処理に対応しておらず、頂点の数がこの値を上回る 3D オブジェクトはレンダリングされません。

ターゲット デバイス上、またはできるだけ多くのターゲット デバイス上でゲームを表示またはテストすることが重要です。コンピュータ画面だけでゲームをテストしても、最適化に必要な情報は得られません。

モバイル デバイスの画面は、平均的なコンピュータ モニタよりも小さいことに注意してください。そのため、多数の三角形を使用して作成されたディテールは、モバイル デバイスでは見えない場合があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

前景にある (つまりカメラに近い) 3D オブジェクトには多くの三角形を使用し、背景にある遠くの 3D オブジェクトは三角形を少なくします。この手法は、静的カメラの *視点* (POV) を使用しているゲームの場合に効果的です。

次の画像は、3D モデルが前景に使用されている一方で、低品質の 3D モデルが 2D 背景にベーキングされているサンプルを示しています。

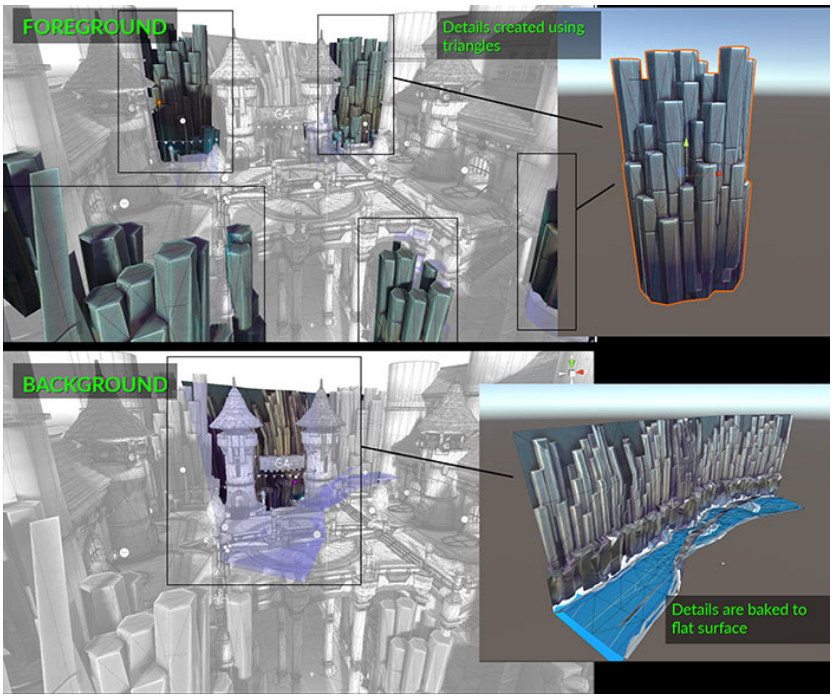


図 5-3 前景および背景オブジェクトでの三角形の使用のサンプル

3D オブジェクトで使用できる三角形の最大数が決まっているわけではありませんが、画面上に同時に表示される 3D オブジェクトが多いほど、オブジェクトごとに使用できる三角形は少なくなります。一方、画面上に表示する 3D オブジェクトが少ないと、より多くの三角形を使用できます。

ターゲット デバイスも重要です。最新の Samsung Galaxy S シリーズなどの最新スマートフォンでは、旧式のモバイル デバイスよりも複雑なジオメトリを処理できます。

次のサンプルは、2 つのデモのキャラクタを示しています。Circuit VR デモのロボット キャラクタは 1 つのみのため、ポリゴン数の多いモデルを使用できます。一方、Armies デモでは 1 つのフレームに数百人の兵士が表示されるため、ポリゴン数を少なくする必要があります。

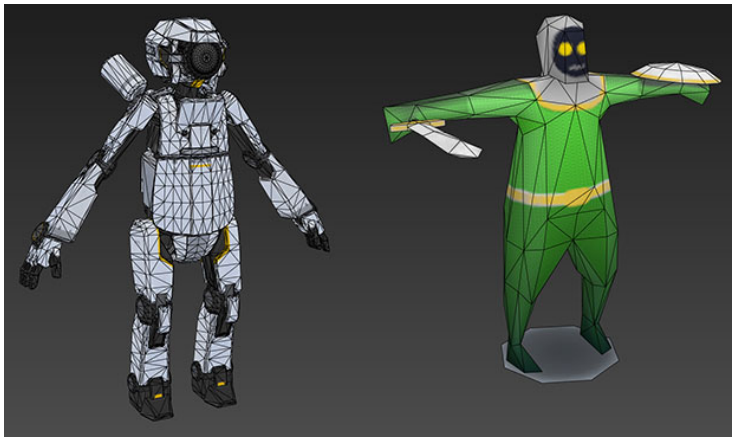


図 5-4 三角形の数の違いのサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

Armies デモ (Unity でビルドされた 64 ビットのモバイル デバイス テクノロジー デモ) では、アニメーション化された多数のキャラクタに対してカメラが静的です。各フレームは合計約 210,000 個の三角形をレンダリングします。三角形の数がこのレベルであるため、デモは約 30 *フレーム/秒* (FPS) でスムーズに実行できます。

注

使用する三角形の数は、制作中のゲームのタイプとターゲット デバイスの仕様の両方によって決まります。

次のサンプルは、このテクノロジー デモで使用された三角形の総数を示しています。

図 5-5 サンプル シーンで使用されている三角形の総数

このシーンで最大のオブジェクトであるキャノン タワーは、画面の大部分を占める約 3,000 個の三角形でできています。

キャラクタに使用されている三角形は約 360 個だけです。これは、キャラクタの数が非常に多く、また、遠くからしか見ることがないためです。つまり、カメラの視点からは、キャラクタは画面上でスムーズに見えます。

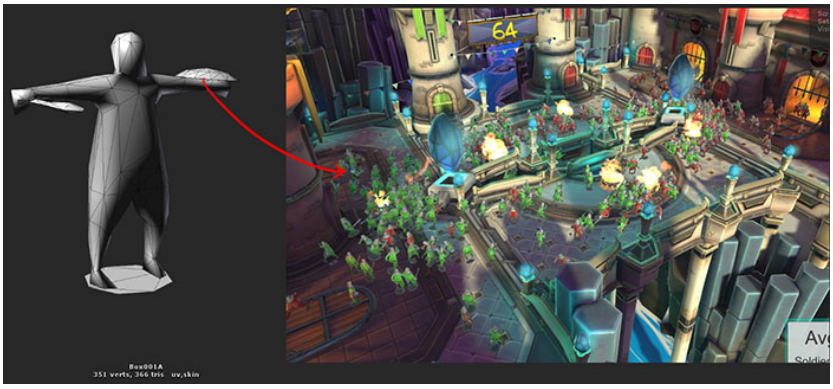


図 5-6 兵士に使われている三角形は少数

以下のサブセクションから構成されています。

- *5.2.1 重要な領域への三角形の分布*(5-83 ページ).
- *5.2.2 微小三角形が好ましくない理由*(5-84 ページ).
- *5.2.3 細長い三角形が好ましくない理由*(5-85 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

5.2.1 重要な領域への三角形の分布

ポリゴンと頂点は、モバイル プラットフォームではどちらも計算コストが高くなります。ゲームのビジュアル品質に寄与する領域にそれらを配置することで、処理予算を無駄にせずに済みます。

ほとんどのデバイスでは画面サイズが小さく、3D オブジェクトがゲーム レベルにあるため、3D オブジェクト上の多くの小さな三角形のディテールがゲーム中に見えることはありません。代わりに、詳細ではなく、オブジェクトの輪郭に影響する大きなシェイプとパーツにフォーカスが絞られます。

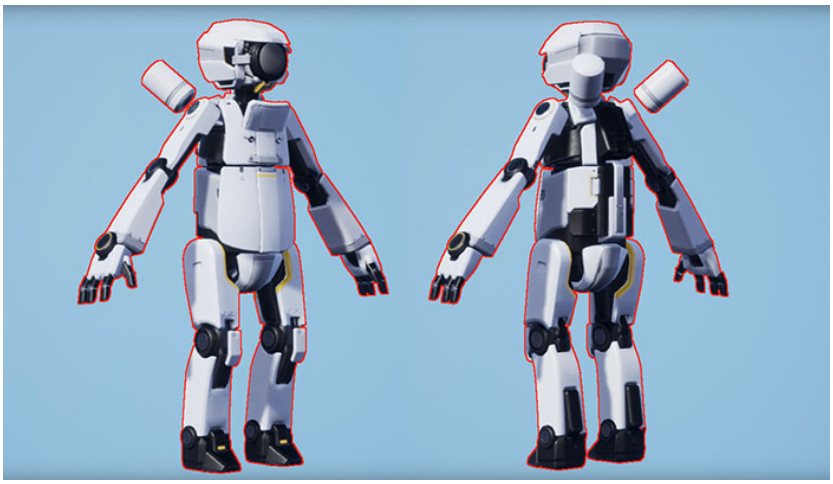


図 5-7 大きなシェイプと輪郭へのフォーカス

画面にあまり表示されない領域には、使用する三角形の数を減らします。この例としては、車の底や衣装ダンスの背部などがあります。

細かいディテールのモデリングには、高密度の三角形メッシュの使用は避けてください。代わりに、細かいディテールにはテクスチャおよび法線マップを使用します。

注

法線マップは、各ピクセルでサーフェス方向をストアするテクスチャ マップです。

次の画像は、法線マップがある場合とない場合の同じメッシュを示したものです。

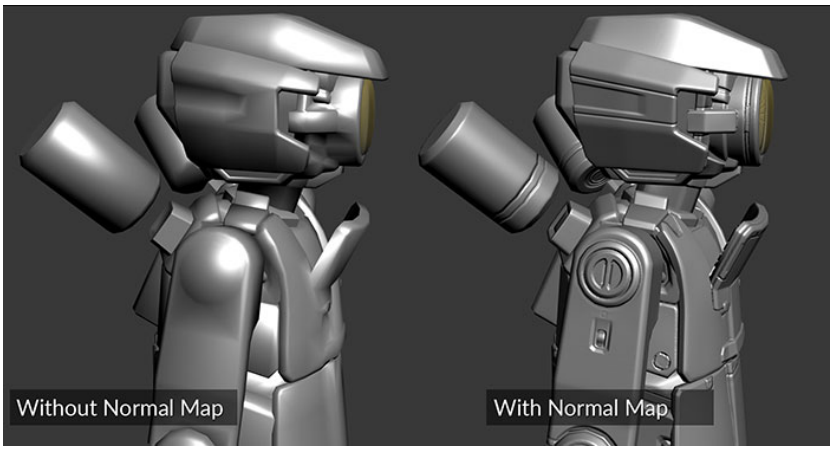


図 5-8 細かいディテールに法線マップとテクスチャを使用している

カメラ POV からは見えることのない裏面や底面などのオブジェクト パーツの削除を検討してください。

ただし、この操作はシーンの再利用可能性を制限することがあるため、注意して行う必要があります。たとえば、テーブル メッシュの底部はエンド ユーザーから見えることはないと考え、それを削除するとします。底部を削除することは、そのモデルを逆さまに配置したり、別のことに再使用したりできなくなることを意味します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

5.2.2 微小三角形が好ましくない理由

微小三角形とは、オブジェクトやシーンの最終的な外観にさほど寄与しない小さな三角形のことです。

ポリゴン数が多い 3D オブジェクトをカメラから遠ざけると、微小三角形の問題が発生します。微小三角形は、デバイス上のサイズが 1 ー 10 ピクセルの三角形とされる場合がほとんどです。

GPU は微小三角形をすべて処理しなければならないため、このような三角形の使用は好ましくありません。一方で、微小三角形は小さすぎて見えないため、最終的な画像にはあまり影響しません。覚えておく必要があるのは、頂点の処理に計算コストがかかることです。画面上で一度に多数の微小三角形を使用すると、処理する頂点の数が多くなります。

微小三角形の発生原因となるアプローチは次の 2 つです。

- あまりにも小さく、多数の三角形で構成されるディテール。
- カメラから遠くにある三角形を多数含んだオブジェクト。

次の画像は、3D オブジェクトがカメラの近くにある場合と遠くにある場合に使用される三角形の数を示しています。左側の画像では使用する三角形の数が少なく、右側のサンプルでは同等のビジュアル ディテールが法線マップを使用して表示されています。

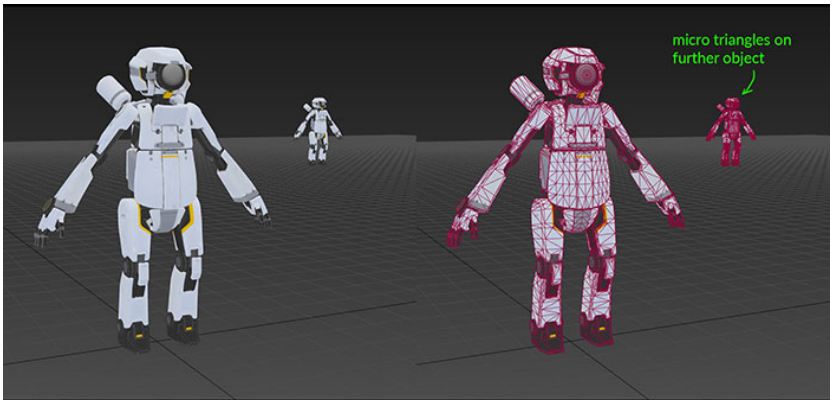


図 5-9 カメラから遠く離れたオブジェクト上の微小三角形

次の画像では、強調表示された領域のほとんどの三角形が小さすぎるため、モバイル デバイスでは表示できません。そのため、これらの三角形は最終的な外観にあまり影響しません。

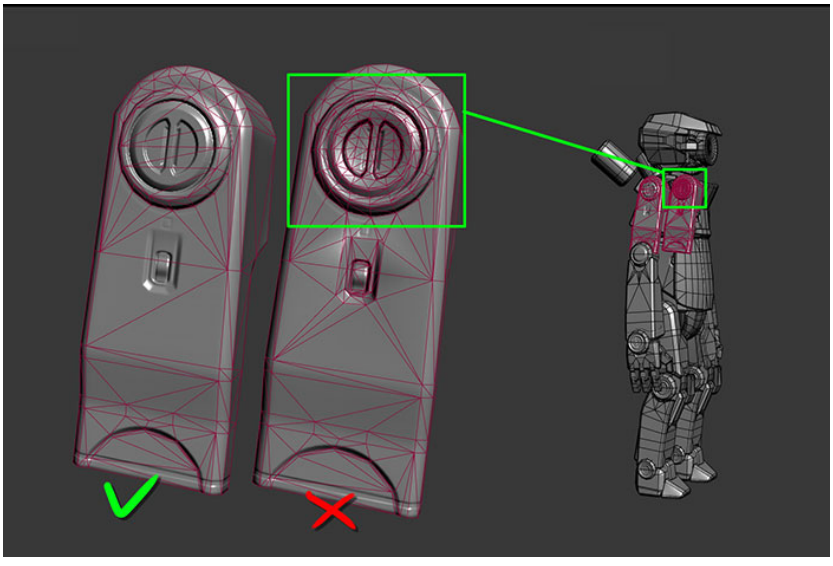


図 5-10 近くから見た微小三角形

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

この問題を最小限に抑える方法

この問題を軽減するために実行できる手順をいくつかご紹介します。

- カメラとの距離が変化するオブジェクトには、*詳細レベル* (LOD) を使用します。正しい LOD を使用すると、遠くにあるオブジェクトを、三角形の数を少なくすることでシンプルにすることができます。
- 背景オブジェクトに使用する三角形を少なくします。
- 細かいディテールをモードで作成するためにポリゴンを使用しないでください。代わりに、テクスチャと法線マップを組み合わせて使用します。
- 画面に表示するには小さすぎるか、最終画像に大きく影響しない頂点または三角形はマージします。
- 三角形の面積を 10 ピクセルより大きく保つようにします。

微小三角形の使用を最小限に抑えることが重要な理由

微小三角形の使用を最小限に抑えることが重要な理由はいくつかあります。理由は次のとおりです。

- GPU は、三角形と頂点をすべて処理する必要があります。これには、画面上の最終シーンに付加価値のない三角形と頂点も含まれます。
- より多くのデータを GPU に送信して処理する必要があるため、メモリ帯域幅に悪影響が及びます。
- 必要な処理量は、モバイル デバイスのバッテリ寿命に直接影響します。そのため、GPU で処理する必要のあるデータが少ないほど、バッテリ寿命は長くなります。

5.2.3 細長い三角形が好ましくない理由

細長い三角形とは、最終画像でレンダリングされたときに 10 ピクセルよりも小さく、画面に沿って広がる頂点のことです。細長い三角形は、GPU での処理負担が通常の三角形よりも大きいため好ましくありません。

次のスクリーンショットは、細長い三角形が使用されているサンプルを示しています。このスクリーンショットでは、遠くから見た柱のベベルが強調表示されています。ただし、近くで見るとベベルは問題になりません。

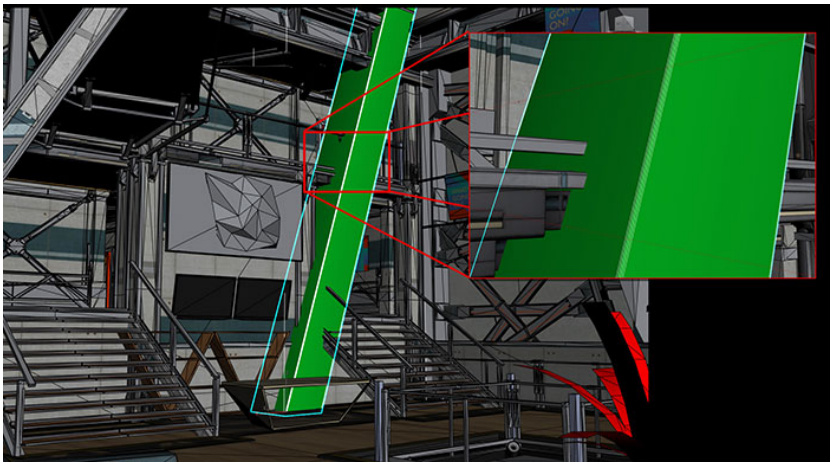


図 5-11 細長い三角形のサンプル

この問題を最小限に抑える方法

この問題を軽減するために実行できる手順をいくつかご紹介します。

- 可能な場合は、見てわかる細長い三角形をすべてのオブジェクトから削除します。これができない場合もありますが、最善策は、細長い三角形を完全に削除することです。
- 光沢のあるマテリアルはちらつきの原因となるため、細長い三角形のオブジェクトには使用しないでください。
- *詳細レベル* (LOD) を使用して、画面上の遠くにある細長い三角形を削除します。
- 具体的には、三角形の辺の長さをなるべく等しくすることが推奨されます。三角形の内側の領域がエッジよりも広いことを確認します。
- この問題に関するより専門的な説明は *http://www.humus.name/index.php?page=News&ID=228* に記載されています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

5.3 詳細レベル

詳細レベル (LOD) とは、オブジェクトがビューアから遠くになるにつれて、メッシュの複雑度を低下させる技法です。

LOD により、処理が必要な頂点の数が減少します。LOD では、微小三角形の問題も回避され、シーンで遠くに配置されたオブジェクトの外観が良くなります。したがって ARM では、カメラからの距離が大きく変化した 3D オブジェクトごとに LOD を最適化することをお勧めします。

次の画像は、LOD 管理を利用して、モデルがカメラから離れる際に適切な詳細レベルを保ちながら、3D モデルの複雑度を軽減する方法を示しています。

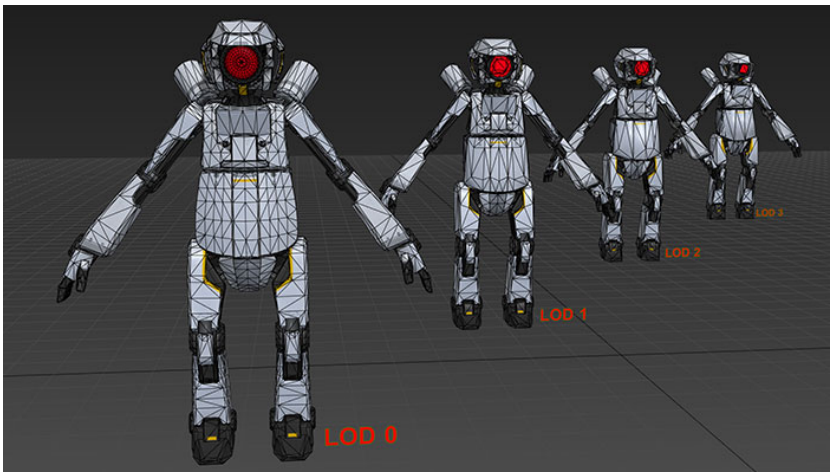


図 5-12 4 レベルの LOD のサンプル

LOD の使用を最適化する方法を考える際のその他のヒントを次に示します。

- LOD で三角形の数を減らす際、三角形がオブジェクトの輪郭にどのように影響するかを考慮してください。
- LOD はシェーダの複雑度にも適用できます。遠くにある 3D オブジェクトでは、シェーダとマテリアルを最適化することができます。たとえば、使用するテクスチャの数を減らします。
- より平坦な領域ではポリゴンをさらに減らします。
- テクスチャには LOD としてミップマップします。

LOD を使用すべきでない状況

あらゆる状況で LOD を使用することが望ましいわけではありません。たとえば、カメラ ビューとオブジェクトの両方が静的である場合や、オブジェクトで使用しているポリゴンの数が既に少ない場合は、LOD を使用しないでください。

LOD には、メモリ オーバーヘッドと大きなファイル サイズが伴います。全ての LOD のメッシュ データはリアルタイムで利用できるように、メモリに保存する必要があります。

次の画像は、シーンが静的であるために LOD が使用されていないシーンのサンプルを示しています。代わりに、プレイヤーから見えないポリゴンを削除するなど、他の最適化方法を使用できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 5-13 LOD を使用しない静的シーンのサンプル

LOD を使用する理由

オブジェクトがカメラから離れるにつれて、オブジェクトの詳細は見えなくなっていきます。20 メートルの距離からは、三角形が 200 個のオブジェクトと三角形が 2000 個のバージョンのオブジェクトの違いはほとんどわかりません。したがって、シーンに何の価値も与えない余分な三角形を多く使用しても意味がありません。

他の主な特長には以下が挙げられます。

- 処理が必要な三角形の数が少ないため、パフォーマンスが向上します。
- LOD により微小三角形が原因となる問題を軽減することができます。

次の画像は、ポリゴンの数が異なっていても、遠くのオブジェクトの見た目が変わらない例を示しています。

図 5-14 遠くにあるモデル LOD のサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

各 LOD に対して適切な数の三角形を使用してください。

各 LOD の三角形の数を決定する際の重要なポイントは、次のとおりです。

- たいていの場合、各 LOD レベル間で三角形の数を 50% ずつ減らすと効果があります。
- 低い LOD に対して密度が高くなる数の三角形を使用しないでください。このような三角形が見えるのは、オブジェクトが遠い場合のみです。
- 表示するカメラから適切な距離にある LOD の見え方を確認してください。LOD が低いと近くからの外観は良くありませんが、近距離から見ることは意図されていないため問題ありません。

次の画像は、使用したポリゴン数がレベルごとに 50% 少なくなった場合にオブジェクトがどのように見えるかを示しています。

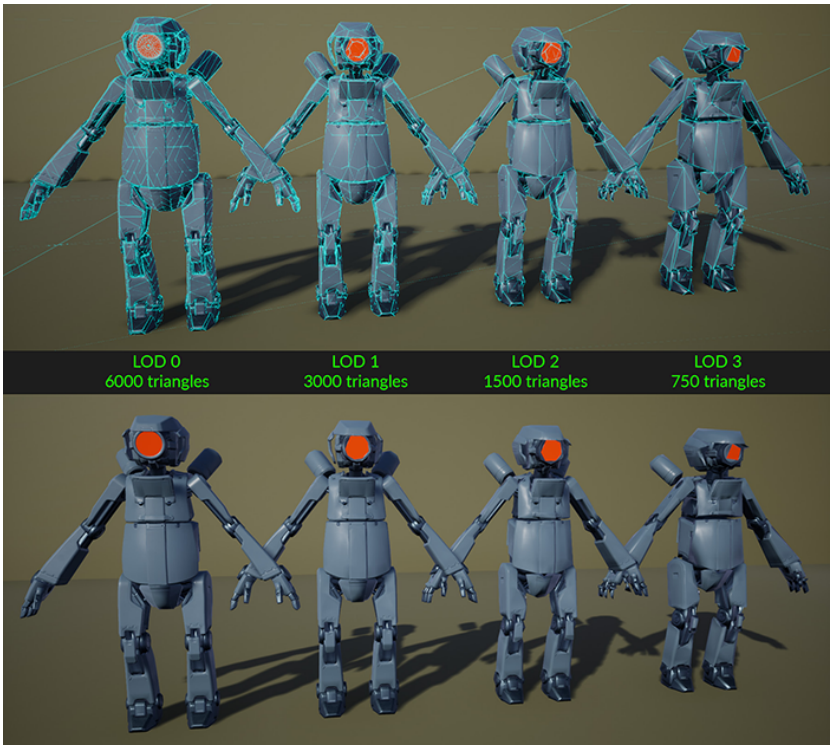


図 5–15 ポリゴン数の LOD のサンプル

次の画像は、ポリゴン数の少ない遠くにあるオブジェクトのサンプルを示しています。画面に近すぎる距離から見た場合に LOD の外観が悪くなることも強調されています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

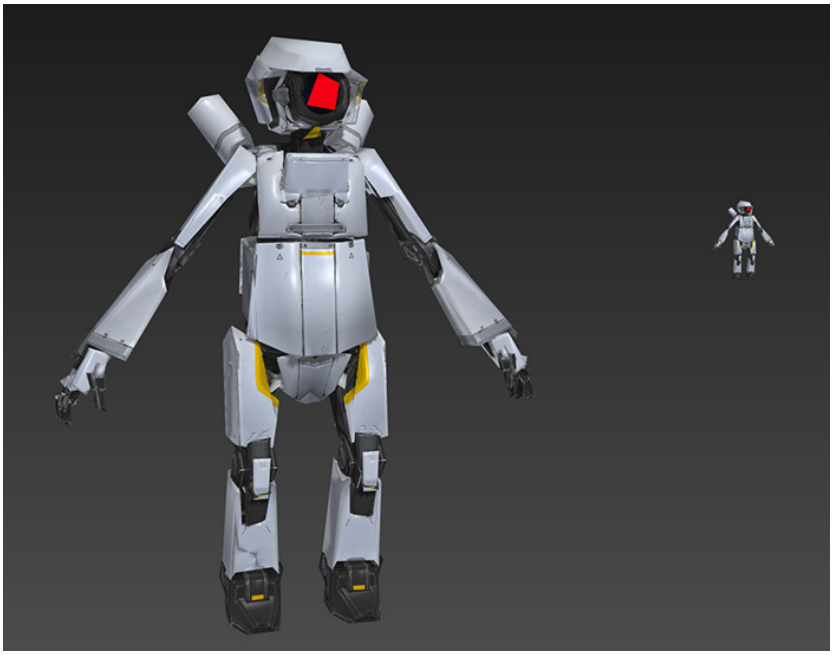


図 5-16 LOD の距離のサンプル

LOD の三角形の数は重要です。それは、LOD の低いオブジェクトのポリゴン数を十分減らさないと、ゲームのパフォーマンスに悪影響が及ぶからです。これは、CPU が必要以上の数の頂点を処理しているためです。

LOD の低いオブジェクトのポリゴン数を減らしすぎると、リアルタイムでアイテムの詳細が現れたり消えたりしているように見えます。この効果はユーザーから見えるため、ゲームに集中できなくなる可能性があります。

オブジェクトに適用する LOD レベルの妥当な数を教えてください。

オブジェクトに適用すべき LOD 数として確かな数値はありません。オブジェクトのサイズと重要度によって異なります。たとえば、アクション ゲームのキャラクタやレーシング ゲームの車には、木などの小さな背景オブジェクトよりも高い LOD レベルを使用すると効果がある可能性があります。

使用する LOD レベルの数が少なすぎると、次のような影響があります。

- LOD レベル間でのポリゴンの削減が十分でない場合、パフォーマンス向上率は低下します。
- ポリゴン数削減のための数値の変化が大きすぎると、LOD スイッチで発生するポッピングがより顕著になります。

使用する LOD レベルの数が多すぎると、次のような影響があります。

- CPU のワークロードの増大。表示する LOD を決定するために必要な処理が多くなるためです。
- メモリ使用量の増大。これらの余分なメッシュをどこかに保存しなければならず、ファイル サイズが大きくなるためです。
- しかし、最大の代償は、これらの LOD モデルの作成と検証に要する時間です。アーティストがそれらを手動で作成する場合は、特にそれが言えます。

LOD メッシュの作成方法

3D ソフトウェアで LOD メッシュを手動で作成する場合は、エッジ ループを削除するか、3D オブジェクト内の頂点の数を減らします。これによりアーティストの制御能力は上がりますが、時間がかかる可能性があります。

LOD メッシュを自動的に作成する場合は、ビルトインモディファイア、または個別の LOD 生成ソフトウェアを使用します。3DS Max のビルトインモディファイアの例として、[ProOptimizer] 機能、または Maya 内の [Generate LOD Meshes] 機能を使用します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

Unity での LOD の実装

Unity での LOD の実装方法に関するガイダンスについては、次のリンクをクリックしてください。
Implementing Level of Detail in Unity(11-264 ページ)

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

5.4 ジオメトリに関するその他のベスト プラクティス

ゲームのパフォーマンスをさらに最適化するために使用できる技法や方法が他にもいくつかあります。

スムージング グループ

スムージング グループまたはカスタム頂点法線を使用すると、エッジの硬度を定義し、モデルの外観を変更することができます。スムージング グループにより、アート ディレクションで微少のポリゴン数が意図的に使用される場合により良い影付を作成することができます。

スムージング グループは、モデルの UV アイランドだけでなく、ベーキングを行う際に正規マップの品質にも影響を与えるため、十分に注意する必要があります。これについては、*テクスチャリングのベスト プラクティスの章*(6-94 ページ) で詳しく説明しています。

スムージングが 3D モデルで実装されている場合は、3D ソフトウェアからエクスポートする必要があり、エンジンにインポートされます。

次の画像は、オブジェクトに適用した場合のスムージングの動作のサンプルを示しています。

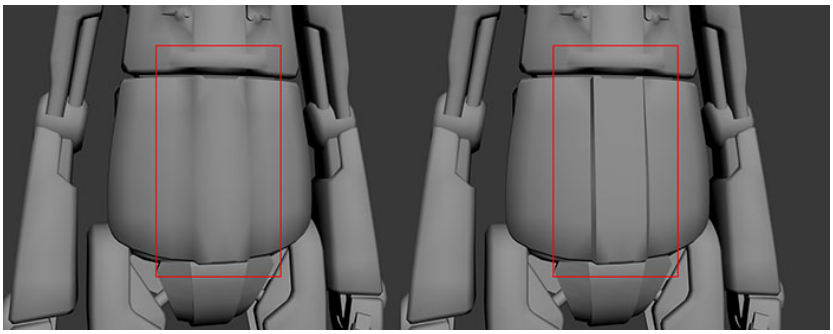


図 5-17 スムーズ グルーピング アプリケーションのサンプル

メッシュ トポロジ

メッシュ トポロジを使用する際は、以下のヒントを覚えておいてください。

- 3D アセットを作成する場合は、適度に整ったクリーンなトポロジを使用してください。
- クリーンなトポロジは、キャラクタや、変形またはアニメーション化するその他のオブジェクトに欠かせません。
- 3D モデルで完璧なトポロジを得ることにこだわりすぎないでください。すべてのオブジェクトで完璧なエッジ フローが必要というわけではありませんが、モデルは整った状態に保ってください。
  - プレイヤーまたはエンドユーザーには、3D モデルのワイヤフレームは見えません。
  - メッシュに適用されるテクスチャとマテリアルの方が、3D モデルの外観に対して大きな影響力があります。

次の画像は、単純なジオメトリとトポロジを使用する岩壁メッシュのワイヤフレームのサンプルを示しています。岩壁の外観は、マテリアルが適用されるとはるかに良くなります。したがって、トポロジの問題はすでに解消されています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

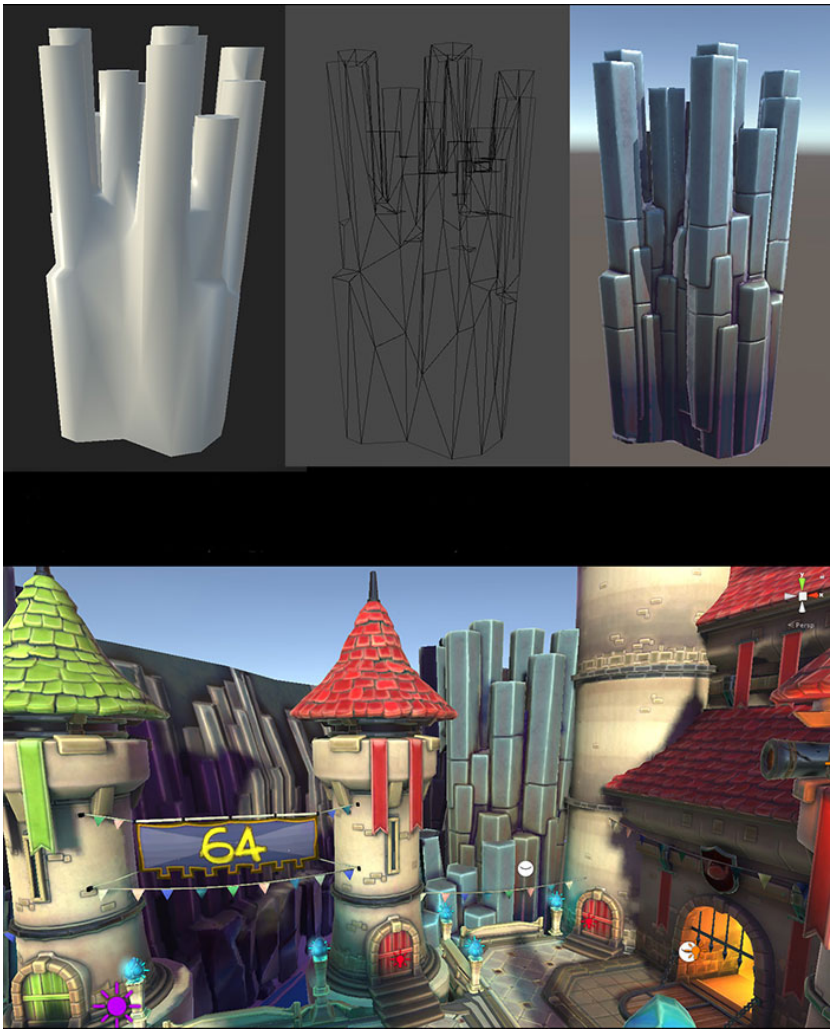


図 5-18 メッシュ トポロジのサンプル

シェイプ誇張

シェイプ誇張は、特定のパーツやシェイプを、見やすいように普通よりも大きく表示する技法です。ただし、ゲームに対してこれが正しいアプローチであるかどうかは、制作中のゲームのタイプとスタイルによって決まります。

以下はその一例です。

- モバイル デバイスの画面は小さく、小さい特定のシェイプのキャプチャが難しいことがあります。シェイプを誇張することで、この問題を解決することができます。たとえば、小さな画面でも見やすいように、キャラクタの手を大きくします。

次の画像は、手のサイズの誇張がどのように見えるかを示しています。手、剣、および全体的な体形がすべて異なる方法で誇張されています。これは、使用したポリゴン数が少ないことを考慮しながら、可視性を向上させるために行われました。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 5-19 シェイプ誇張のサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 6 章
リアルタイム 3D アートのベスト プラクティス: テクスチャリング

この章では、ゲームをスムーズに実行し、外観を向上させるために役立ついくつかのテクスチャ最適化について説明します。

注意

このコンテンツの最新バージョンは、Arm の開発者 Web サイト(*https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/real-time-3d-art-best-practices-texturing*)にあります。

以下のセクションから構成されています。

- *6.1 テクスチャ アトラス、フィルタリング、ミップマップ テクスチャ アトラス*(6-95 ページ).
- *6.2 テクスチャ フィルタリング*(6-96 ページ).
- *6.3 ミップマッピング*(6-100 ページ).
- *6.4 テクスチャ サイズ、色空間、および圧縮*(6-101 ページ).
- *6.5 UV アンラップ、ビジュアル インパクト、テクスチャ チャネル パッキング*(6-104 ページ).
- *6.6 アルファ チャネルおよび法線マップのベスト プラクティス*(6-108 ページ).
- *6.7 法線マップ ベーキングのベスト プラクティス*(6-110 ページ).
- *6.8 テクスチャ設定の編集*(6-114 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.1 テクスチャ アトラス、フィルタリング、ミップマップ テクスチャ アトラス

テクスチャ アトラスとは、まとめてパックされている複数の小さな画像からのデータを含んだ画像のことです。テクスチャ アトラスでは、テクスチャごとに別々のメッシュを使用するのではなく、大きなテクスチャを複数のメッシュで共有します。

テクスチャ アトラスは、アセットを作成する前に作成できます。つまり、アセットはテクスチャ アトラスに従って UV アンラップされます。これには、テクスチャ作成時の早期計画が必要となります。

テクスチャ アトラスは、アセットが完成した後に、ペイント ソフトウェアでテクスチャをマージして作成することもできます。ただし、この場合は、テクスチャに従って UV アイランドを再配置することも必要です。

注

UV アイランドとは、テクスチャ マップ内で互いに接続されたポリゴンのグループのことです。

次の画像では、どの 3D オブジェクトが 1 つのテクスチャ セットを使用しているかが強調表示されています。

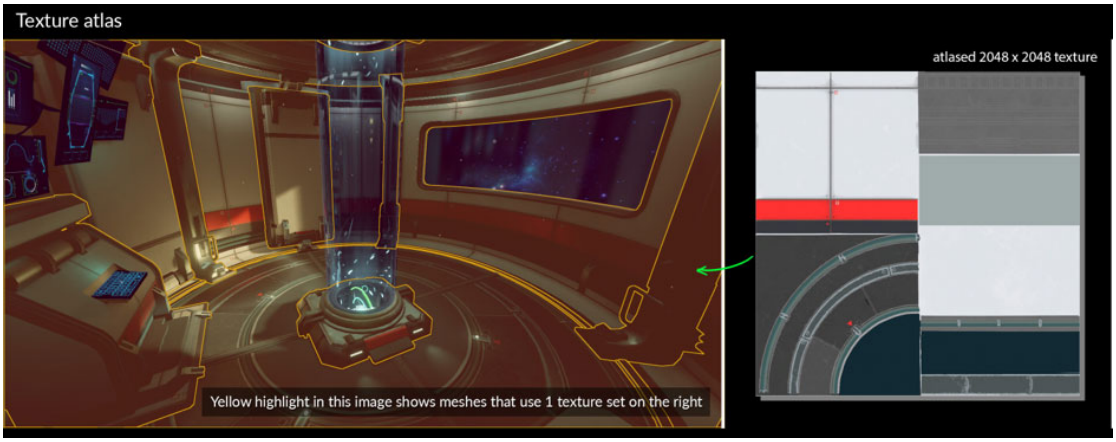


図 6–1 テクスチャ アトラスのサンプル

テクスチャ アトラスを使用すべき理由

テクスチャ アトラスを使用すると、テクスチャ アトラスおよび同じマテリアルを共有している複数の静的オブジェクトをバッチ処理できます。バッチ処理によってドロー コールの数が減り、ドロー コールが少なくなることで、ゲームが CPU バウンドの場合のパフォーマンスが向上します。

Unity ゲーム エンジンには、オブジェクトが静的とマークされている場合にバッチ処理を行う機能があります。この処理は、オブジェクトを手動でマージしなくても行われます。詳細については、*Unity の Web サイト*を参照してください。

テクスチャ アトラスでは、テクスチャがまとめてパックされるため、ゲーム内のテクスチャも少なくする必要があります。このように圧縮することにより、テクスチャのメモリ コストを抑えることができます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.2 テクスチャ フィルタリング

テクスチャ フィルタリングは、シーン内のテクスチャの画質向上のために使用されます。一般に、テクスチャ フィルタリングを行わない場合、エイリアシングなどのアーティファクトの外観はさらに悪くなります。Unity では、テクスチャ フィルタリングに使用できるオプションが複数あります。

テクスチャ フィルタリングにより、テクスチャの外観が向上し、ブロック ノイズが減少します。通常は、これによりゲームの外観が良くなります。

しかし、画質が良いということは必要な処理も多くなることを意味するため、テクスチャ フィルタリングがパフォーマンスの低下につながることもあります。テクスチャ フィルタリングは、GPU のエネルギー消費の半分を占める可能性があります。そのため、より単純なテクスチャ フィルタリングを選択すれば、アプリケーションのエネルギー需要を削減することができます。

最近傍またはポイント フィルタリング

最近傍フィルタリングでは、近くで見たときにテクスチャにはブロック ノイズがあります。これは最も単純で安価なテクスチャ フィルタリングです。

バイリニア フィルタリング

バイリニア フィルタリングでは、近くのテクスチャはぼやけています。4 つの最近傍テクセルがサンプリングされ、その平均色がメイン ピクセルの色になります。最近傍フィルタリングとは異なり、バイリニア フィルタリングではピクセルの階調がスムーズになるため、ピクセルのブロック ノイズが低減されます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

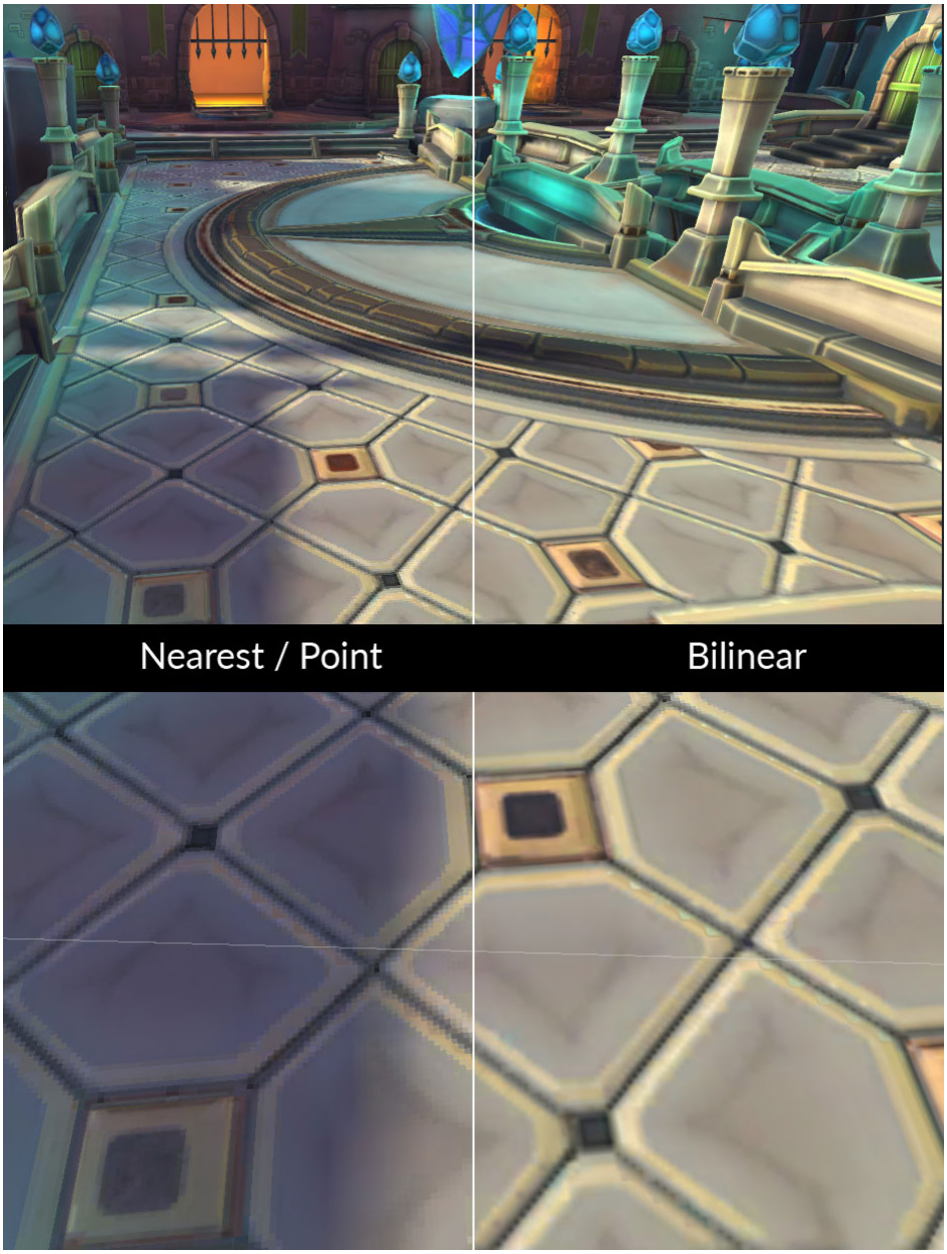


図 6-2 最近傍およびバイリニア テクスチャ フィルタリングのサンプル

トライリニア フィルタリング

トライリニア フィルタリングは、バイリニア フィルタリングと似ていますが、ミップマップ レベル間の組み合わせが追加されています。トライリニア フィルタリングでは、ミップマップ間での遷移を円滑にすることで、目立つ変化を取り除きます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

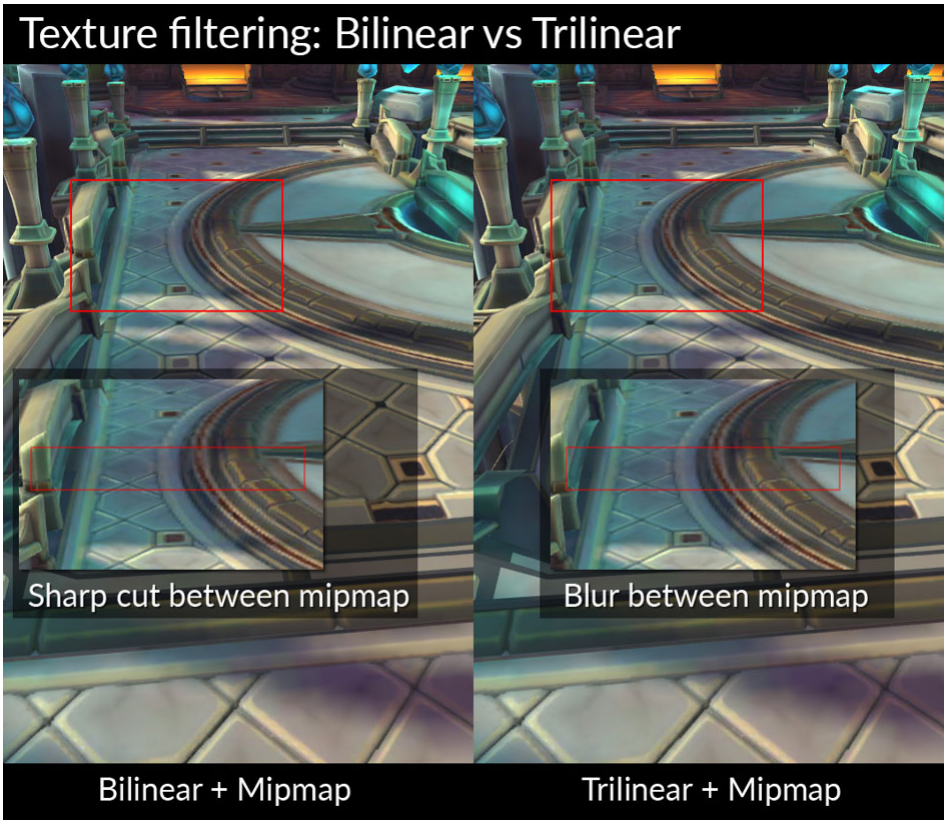


図 6-3 バイリニア テクスチャ フィルタリングとトライリニア テクスチャ フィルタリングの比較サンプル

注

バイリニアおよびトライリニア フィルタリングでは、さらに多くのピクセルのサンプリングが必要なため、より多くの計算が必要になります。

異方性フィルタリング

異方性フィルタリングにより、ある角度から見たテクスチャの外観が向上します。たとえば、地表レベルのテクスチャに適しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 6-4 異方性テクスチャ フィルタリングのサンプル

テクスチャ フィルタリングのベスト プラクティス

ARM では、テクスチャ フィルタリングに関する次のヒントを試してみることをお勧めしています。

- バイリニア フィルタリングは、パフォーマンスとビジュアル品質のバランスをとるために使用します。
- トライリニア フィルタリングでは、バイリニア フィルタリングよりも大きいメモリ帯域幅が必要になるため、選択的に使用します。
- トライリニア フィルタリングおよび 1 倍の異方性フィルタリングではなく、外観もパフォーマンスもより優れたバイリニア フィルタリングおよび 2 倍の異方性フィルタリングを使用します。
- 異方性レベルは低く保ち、2 よりも上のレベルは、重要なゲーム アセットのみに使用します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.3 ミップマッピング

ミップマッピングとは、元のテクスチャを低解像度で保存するコピーのことです。したがって、ミップマッピングはテクスチャの*詳細レベル* (LOD) であると考えることができます。

フラグメントが占めるテクスチャ領域の大きさに基づき、サンプリングに使用する適切なレベルが選択されます。オブジェクトがカメラから遠い場合は、低解像度のテクスチャが適用されます。オブジェクトがカメラに近い場合は、高解像度のテクスチャが適用されます。

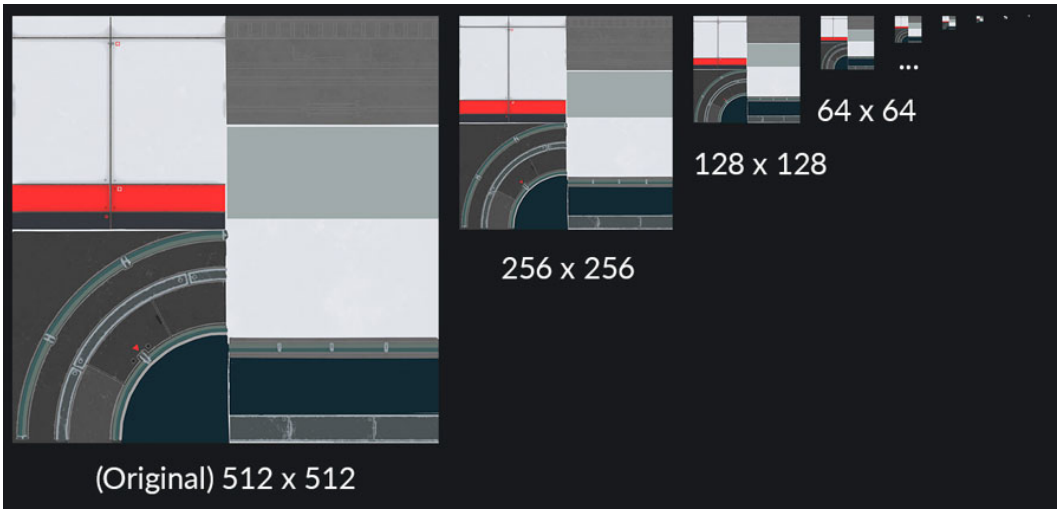


図 6–5 ミップマップ チェインのサンプル

ベスト プラクティス

ミップマッピングを使用することで性能と品質が向上するため、必ず使用してください。ミップマッピングを行うと、GPU はカメラから遠いオブジェクトについてはフル解像度のテクスチャをレンダリングする必要がないため、GPU の性能が向上します。

さらに、ミップマッピングによりテクスチャ エイリアシングが減少することで、最終画質が向上します。テクスチャ エイリアシングは、カメラから遠い領域でちらつきが発生する原因となります。

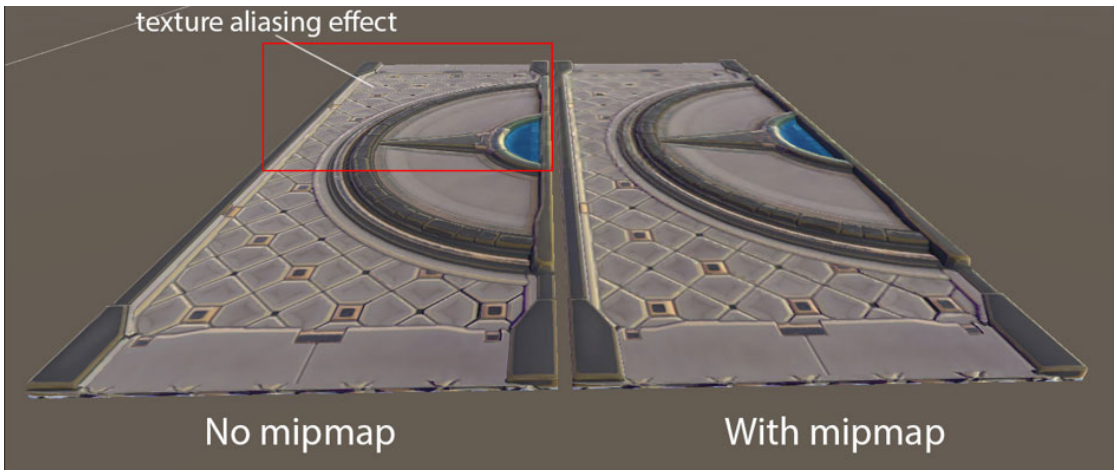


図 6–6 ミップマップの比較サンプル

Unity では、必要に応じて、インポート時にミップマップが自動的に作成され、2 のべき乗でないテクスチャは再スケーリングされます。詳細については、*Unity の Web サイトを参照してください*。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.4 テクスチャ サイズ、色空間、および圧縮

テクスチャは、必要な品質を満たすのに十分な大きさである必要がありますが、大きすぎないように注意してください。また、多くのメッシュ間で共有される複数のテクスチャを含む大きなテクスチャ アトラスを使用すると良いでしょう。

テクスチャのサイズ

テクスチャはそれぞれサイズを変えることができます。ディテールが必要ない一部のテクスチャのサイズを縮小すると、帯域幅レベルの削減に役立ちます。たとえば、拡散テクスチャは 1024x1024 に設定し、ラフネス マップまたはメタリック マップは 512x512 に設定できます。

テクスチャのサイズを用途に合わせて縮小した後、ビジュアル品質が低下しているかどうかを確認します。

テクスチャの色空間

Adobe Photoshop や Substance Painter などのほとんどのテクスチャリング ソフトウェアは、sRGB 色空間を使用すると適切に動作し、エクスポートできます。

以下を試してみてください。

- sRGB 色空間で拡散テクスチャを使用します。
- 色として処理されないテクスチャは、sRGB 色空間に含めてはなりません。例としてはメタリック マップ、ラフネス マップ、法線マップなどがありますが、これらに限定されません。
  - その理由は、マップが色ではなくデータとして使用されるためです。
  - これらのマップで sRGB を使用すると、外観、ビジュアル、またはマテリアルが不適切になります。

——— 注 ———

テクスチャの [Inspector] ウィンドウの [sRGB (Color Texture)] 設定は、ラフネス マップ、鏡面反射マップ、法線マップなどに対してはオンにしないでください。

次のスクリーンショットは、sRGB がこのようなテクスチャに不適切に適用された場合にどうなるかを示しています。

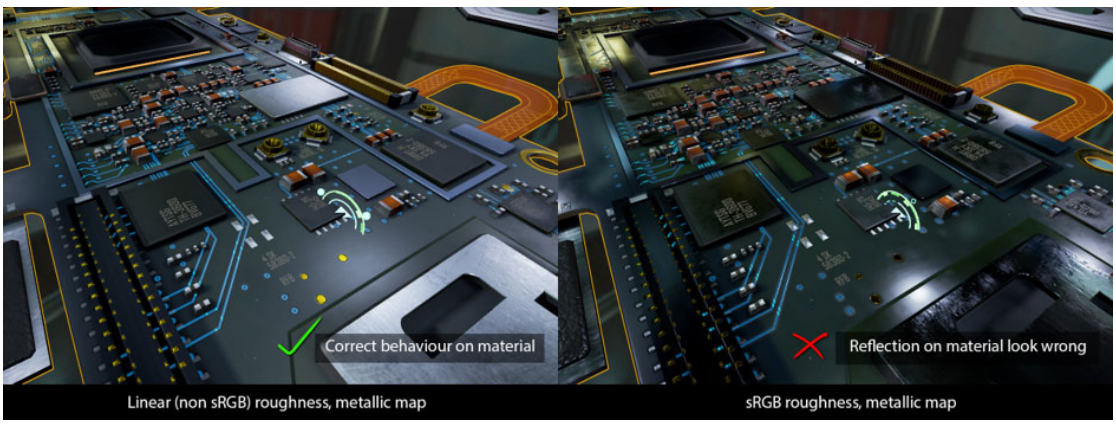


図 6-7 テクスチャに対する sRGB の適用比較サンプル

テクスチャ圧縮

テクスチャ圧縮は、テクスチャのデータ サイズを縮小する一方で、ビジュアル品質の低下を最小限に抑えるために適用される画像圧縮機能です。開発中は、TGA や PNG などの一般的な形式を使用してテクスチャをエクスポートします。これらの形式は使いやすく、主要なイメージ ソフトウェア プログラムによってサポートされています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

これらの形式は、最終的なレンダリングに使用しないでください。専用のイメージ形式と比較して、アクセスとサンプリングの両方について低速になるためです。Android には、*Adaptive Scalable Texture Compression* (ASTC)、*Ericsson Texture Compression* (ETC) 1、ETC2 などのいくつかのオプションが用意されています。

テクスチャ圧縮のベスト プラクティス

ARM が生み出した ASTC テクノロジを使用することを推奨します。ASTC を使用すべき理由を説明します。

- ASTC は、ETC と同じメモリ サイズでありながら、より優れた品質をもたらします。
- 言い方を変えると、ASTC は ETC よりも小さなメモリ サイズで同じ品質をもたらします。
- ASTC は ETC よりもエンコードに時間がかかり、ゲーム パッケージング プロセスにより長い時間がかかる場合があります。このことが問題になる場合は、ゲームの最終パッケージングで ASTC を使用すると良いでしょう。
- ASTC ではブロック サイズを設定できるため、品質をさらに細かく制御できます。ブロック サイズに対して最適なデフォルト値というものは存在しませんが、開始点としては 5x5 または 6x6 に設定するのが適しています。

ゲームをデバイスにすばやく展開する必要がある場合は、ETC を使用して開発した方が作業が速くなる場合があります。高速の圧縮設定で ASTC を使用して、展開時間の延長を回避することもできます。エンコーディング時は、品質とサイズの両方について速度をトレードオフできるオプションが用意されています。最終的なビルドにとって、ASTC はビジュアル品質とファイル サイズのバランスが取れた最適なオプションです。

ゲームをパッケージングする際には、ゲーム エンジンによってテクスチャ圧縮が処理されます。しかし、どれを使用するかは選択できます。使用する形式を選択する必要があるため、この手順を省略することは通常はあり得ません。

次のスクリーンショットは、Unity での Android パッケージのビルド時にどこで ASTC を選択するべきかを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

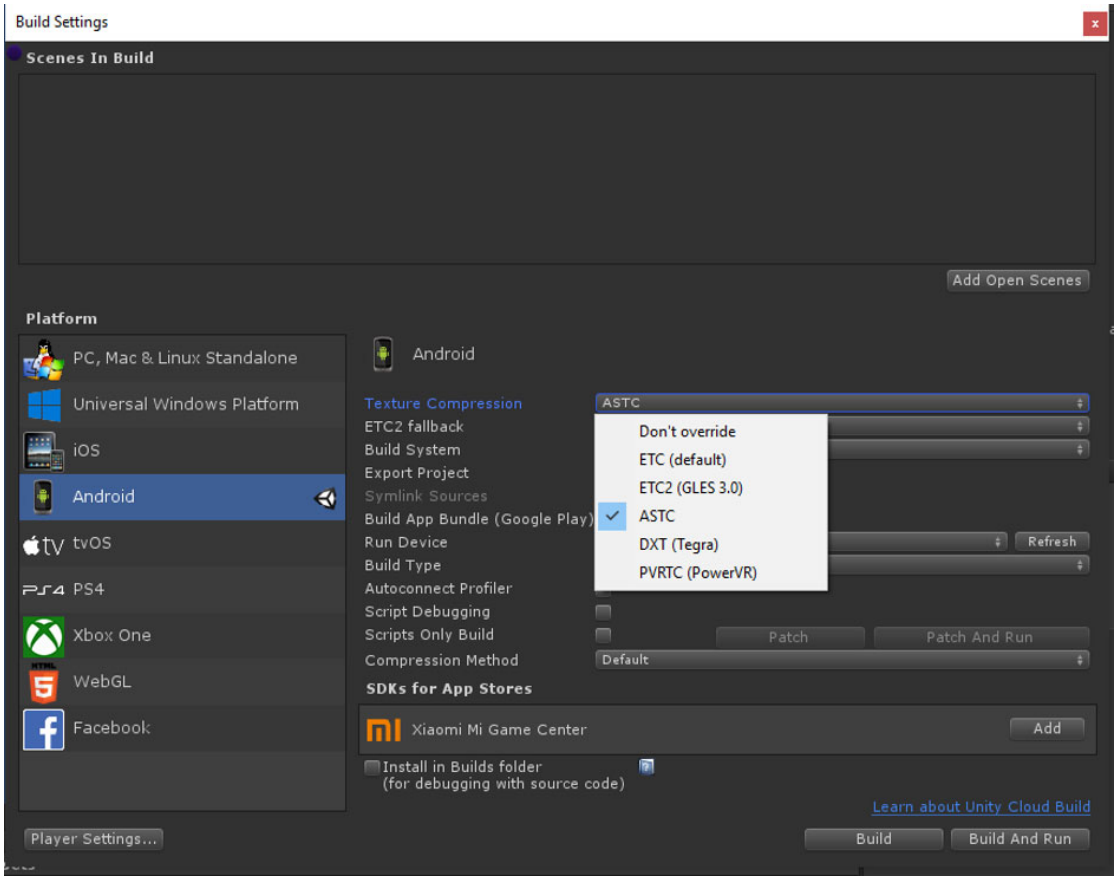


図 6-8 Unity での Android 向け ASTC テクスチャ圧縮の有効化

次のイメージは、ETC 圧縮と ASTC 圧縮間の品質および対応するファイル サイズを比較したものです。

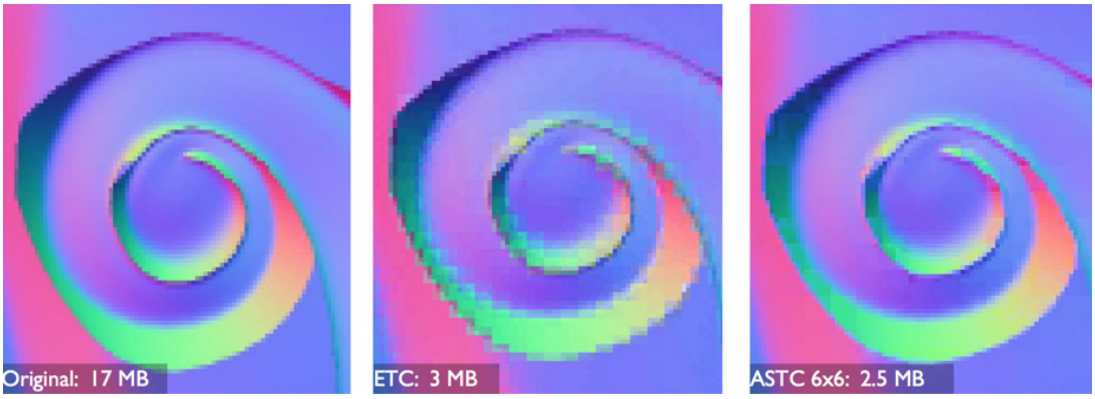


図 6-9 ETC および ASTC のテクスチャ圧縮の比較

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.5 UV アンラップ、ビジュアル インパクト、テクスチャ チャネル パッキング

UV マップは、2D テクスチャを 3D モデルのサーフェスに投影します。UV アンラッピングは、UV マップを作成するプロセスです。

UV アンラッピング

UV アイランドをできる限り直線状に保つことをお勧めします。

注

UV アイランドとは、テクスチャ マップ内で互いに接続されたポリゴンのグループのことです。

その理由は次のとおりです。

- UV アイランドのパッキングが容易になり、無駄なスペースが少なくなります。
- UV を直線状にすると、テクスチャで発生する階段効果を減らすことができます。
- モバイル プラットフォームでは、通常、テクスチャ サイズがゲーム コンソールや PC よりも小さいため、テクスチャ スペースが限られています。UV パッキングを正しく行うことにより、テクスチャから最適な解像度を得ることができます。
- UV を直線状に保ってテクスチャ全体の品質を向上させることにより、UV をわずかに歪ませると良いかもしれません。

UV の継ぎ目は目立たない場所に配置します。これはビジュアル品質のためです。テクスチャの継ぎ目はモデル上の外観を損なう可能性があります。したがって、エッジがシャープで UV アイランド間に小さなスペースがある UV アイランドは分割します。これにより、後でベーキング処理を行って、より質の高い法線マップを作成できます。

次の画像は、UV アンラッピングを使用してテクスチャ スペースを最大化する方法のサンプルを示しています。

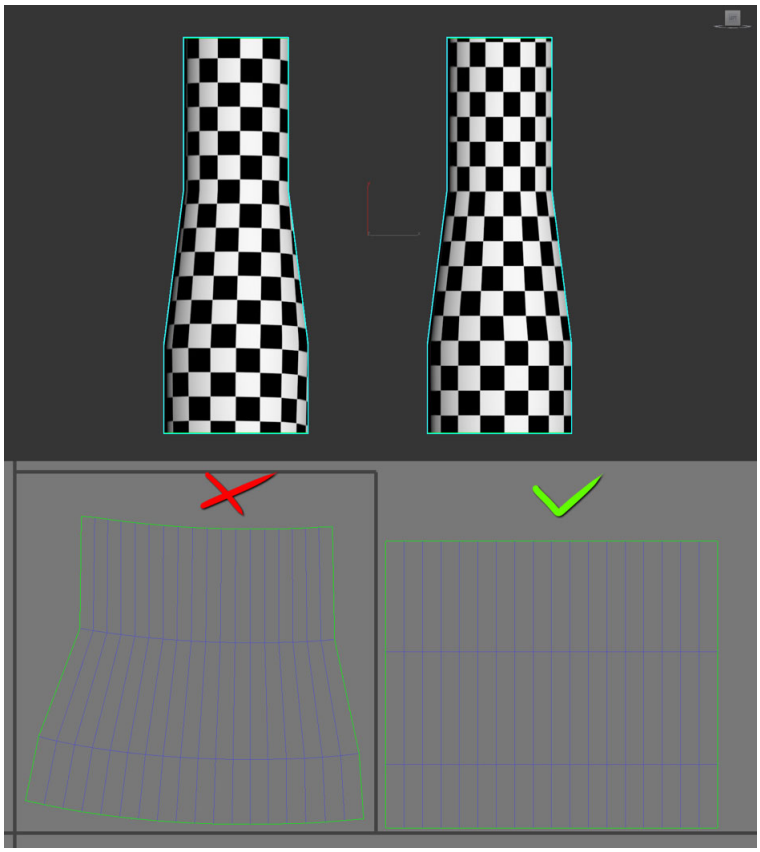


図 6–10 直線状の UV アンラッピングを使用してテクスチャ スペースを最大化する方法のサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ビジュアル インパクト

見てわかるディテールのみを作成しましょう。スマートフォンの画面は小さいため、細かいディテールは表示されません。テクスチャを作成するときは、この点を考慮してください。たとえば、部屋の隅にかろうじて見える椅子のために、多くのディテールを含んだ 4K テクスチャは不要です。

次のスクリーンショットは、必要な量のディテールだけを含んだ兵士の小さなテクスチャのサンプルを示しています。

図 6-11 適切な量のディテールを含んだ小さなテクスチャのサンプル

場合によっては、シェイプの読み取りを容易にするために、エッジとシェーディングを誇張して強調表示する必要があります。一般に、モバイル プラットフォームではさらに小さなテクスチャが使用されます。そのため、小さなテクスチャ内で必要なすべてのディテールをキャプチャすることが難しい可能性があります。

使用するテクスチャを少なくして、余分なディテールを 1 つのテクスチャにベーキングします。これが重要な理由は次のとおりです。

- スマートフォンの画面は小さいため、ディテールを拡散テクスチャ自体にベーキングして確実に見えるようにすることをお勧めします。
- アンビエント オクルージョンや小さな鏡面反射ハイライトなどの要素をベーキングしてから、拡散テクスチャに追加できます。

このアプローチでは、鏡面反射とアンビエント オクルージョンを得るために、シェーダとエンジンの機能に過度に依存する必要がありません。

次のスクリーンショットは、ディテールをテクスチャにベーキングする方法のサンプルを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 6-12 テクスチャにベーキングされたディテールのサンプル

可能な場合は、シェーダで色つけできるグレースケール テクスチャを使用します。これにより、色つけを実行するカスタム シェーダを作成する必要がなくなり、テクスチャ メモリが節約されます。

この方法ですべてのオブジェクトが適切に表示されるわけではないため、この方法が適しているかどうかを慎重に判断してください。この方法は、均一または類似の色を持つオブジェクトに適しています。

RGB マスクを使用してから、マスクの色範囲に基づくテクスチャを適用する方法もあります。

次の画像は、色つけされた柱にグレースケール テクスチャを適用するサンプルを示しています。

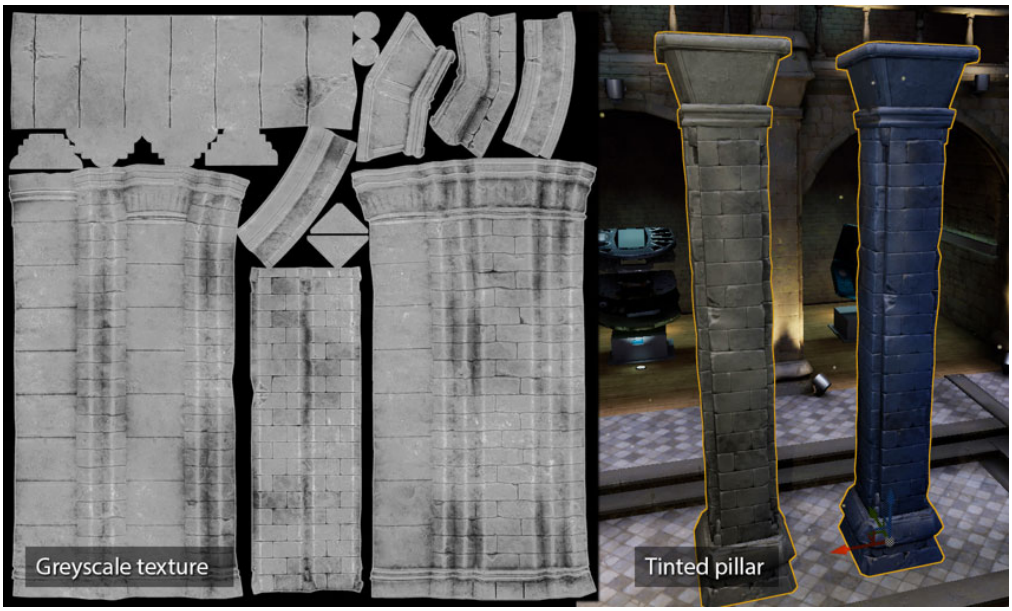


図 6-13 グレースケール テクスチャのサンプル

テクスチャ チャネル パッキング

複数のテクスチャを 1 つにパッキングするには、テクスチャ チャネルを使用します。以下を行うこともお勧めします。

- パッキングは 3 つのマップを 1 つのテクスチャにまとめることができるため、テクスチャ メモリの節約に役立ちます。これは、テクスチャ サンプラーが少なくなることを意味します。
- このテクスチャ パッキング手法は、一般的にラフネス (または滑らかさ) とメタリックを 1 つのテクスチャにパッキングするために使用されますが、任意のテクスチャ マスクに適用できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

重要度の高いマスクの保存には、緑のチャネルを使用します。通常、緑のチャネルの方がビット数は多くなります。これは、人の目は緑に敏感で、青に敏感ではないためです。以下を行うこともお勧めします。

- ラフネス (または滑らかさ) マップは、通常、メタリックよりもディテールが多いため、緑のチャネルに適しています。
- これらのテクスチャは、sRGB 色空間ではなく、リニアまたは RGB に設定します。

次の画像は、テクスチャ パッキングのサンプルを示しています。

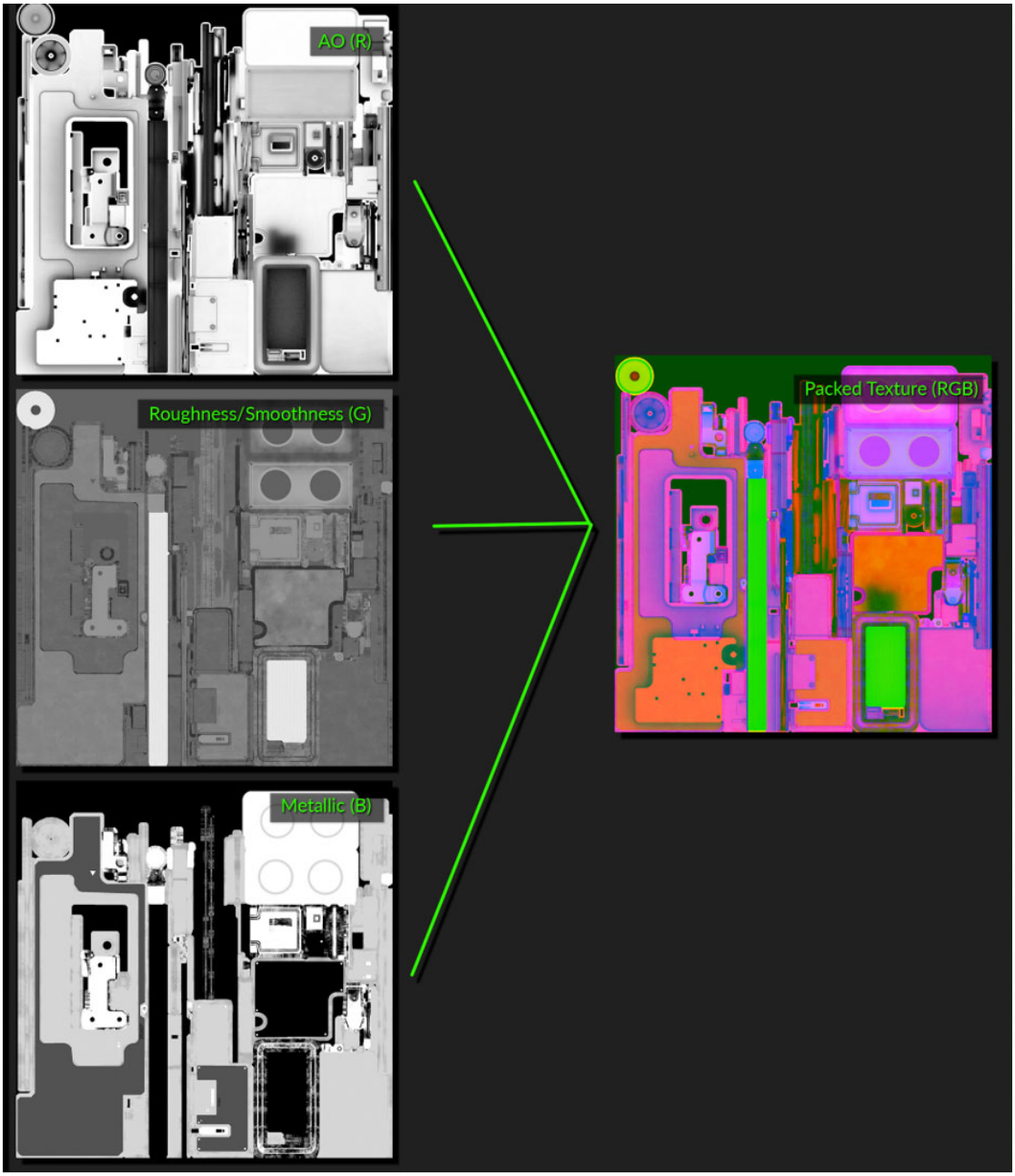


図 6-14 テクスチャ チャネル パッキングのサンプル

色覚錐体の詳細については、*こちらをご覧ください*。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.6 アルファ チャネルおよび法線マップのベスト プラクティス

ゲームでアルファ チャネルまたは法線マップを使用する場合に推奨されるその他のベスト プラクティスをご紹介します。

アルファ チャネルの使用

テクスチャにアルファ チャネルを追加する際は、選択的に行ってください。透過性を追加すると、32 ビット形式に変換されるため、テクスチャはファイル サイズよりも大きくなることが多く、全体的な帯域幅の使用量が増大します。

アルファ チャネルを保存するもう 1 つの方法では、ラフネス テクスチャまたはメタリック テクスチャで追加のチャネルを使用します。Unity では、このテクスチャは 3 つのうち 2 つのチャネルのラフネス (G) およびメタリック (B) を使用することがあり、(R) チャネルは自由に利用できます。

アルファ マスクの保存に空きチャネルを使用することで、そのファイル サイズで 16 ビットのディフューズ テクスチャを維持できます。通常、アンビエント オクルージョン マップはディフューズ マップにベーキングすることができます。

次の画像は、赤いチャネルに不透明度マップを保存する方法のサンプルを示しています。

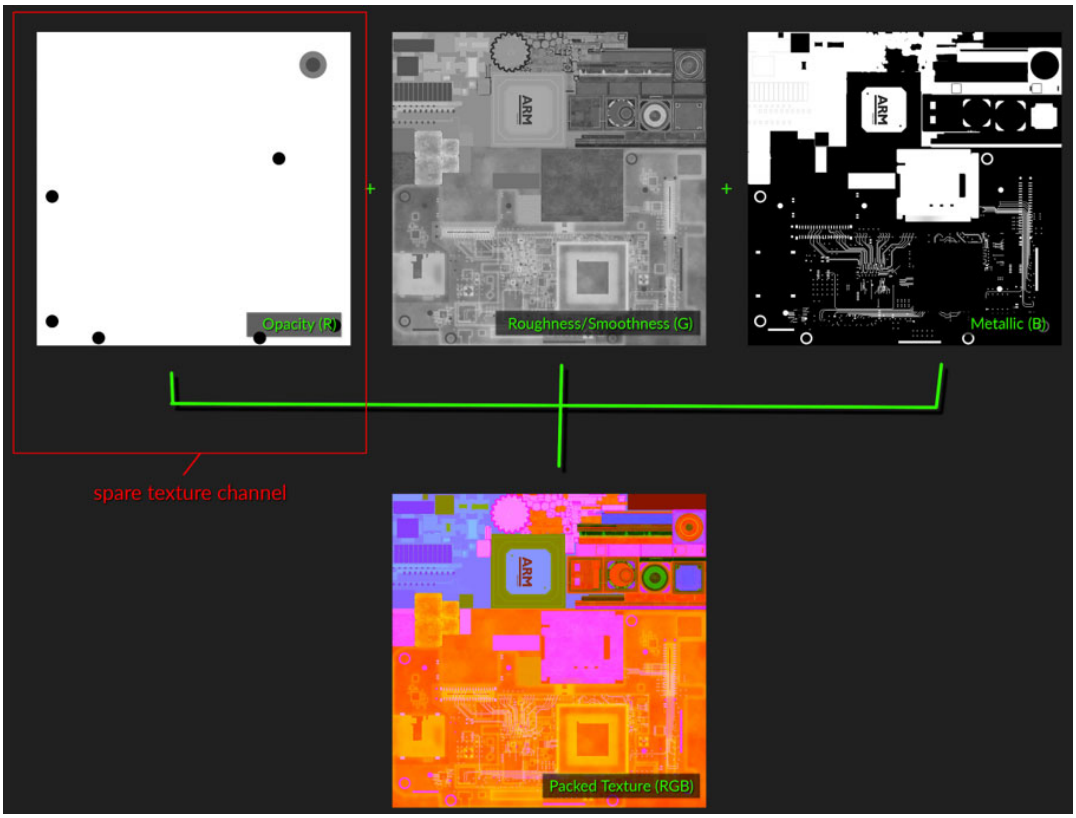


図 6-15 赤いチャネルへの不透明度マップの保存

法線マップのベスト プラクティス

法線マップは、3D オブジェクトの詳細を見えるようにする場合に効果的な方法です。この方法は、モデルに多くの三角形を要するしわ、ボルト、その他のディテールなど、細かいディテールを追加するのに最適です。法線マッピングの使用は、ゲームのタイプやアート ディレクションによって異なります。

ほとんどの内部プロジェクトでは、パフォーマンスに著しい低下のない法線マッピングを使用しています。ほとんどのデモではハイエンドのデバイスをターゲットとしていますが、ローエンドのデバイスでは結果が異なる可能性があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

法線マッピングの使用は、少ないとは言えある程度のコストがかかります。以下のポイントを覚えておいてください。

- 法線マップは追加のテクスチャです。つまり、テクスチャ フェッチが増え、より多くの帯域幅が使用される結果となることを意味します。
- 法線マップは、ローエンド デバイスをターゲットとする場合は控え目に使用してください。

ただし、法線マップを使用した結果として、ジオメトリの三角形が少なくなる場合は、性能が向上する可能性があります。

次の画像は、細かいディテールに法線マップとテクスチャを使用する方法のサンプルを示しています。

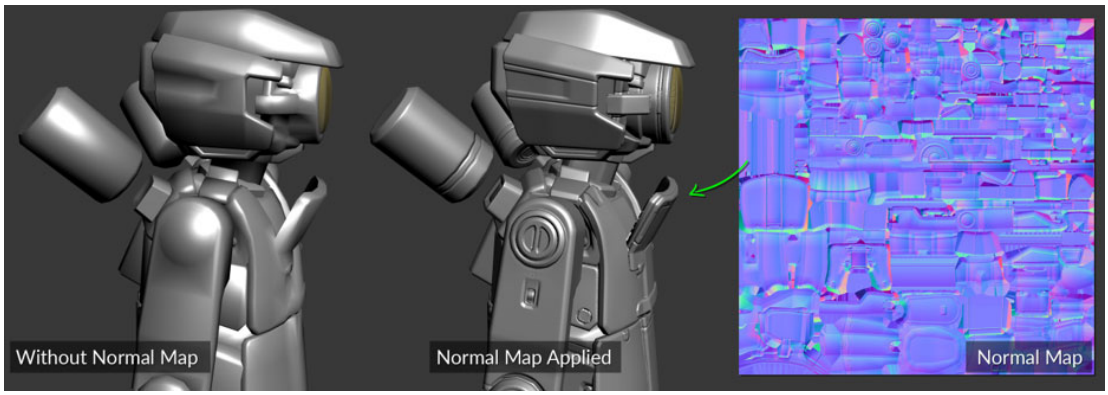


図 6-16 法線マップとテクスチャのサンプル

法線マップ ベーキングのベスト プラクティス

ケージを使用すると、ベーキングするサーフェスのタイプにかかわらず、常に高品質の法線マップを取得できます。ほとんどの法線マッピング ソフトウェアでは、ケージが自動的に作成されます。必要であれば、低ポリゴン モデルのコピーから法線マップを作成して、スケールを多少大きくすることができます。

プログラムではケージを使用して、ベーキング時に法線の計算に使用する方向を変更します。これにより、分割法線エッジおよびハード エッジではるかに良い結果が得られます。

ケージとはポリゴン数の少ないモデルの大型バージョンです。したがって、モデルは押し出されたように見えます。ベーキングが正しく機能するには、ポリゴン数の多いモデルも物理的に含める必要があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.7 法線マップ ベーキングのベスト プラクティス

ケージを使用すると、ベーキングするサーフェスにかかわらず、高品質の法線マップを取得できます。ほとんどの法線マッピング ソフトウェアでは、ケージを自動的に作成できますが、低ポリゴン モデルをコピーしてからスケールを少し大きくしてケージを作成することもできます。

ケージは、ベーキング時に法線の計算に使用する方向をプログラムで変更するために使用します。これにより、次の画像に示すように、分割法線エッジとハード エッジではるかに優れた結果が得られます。

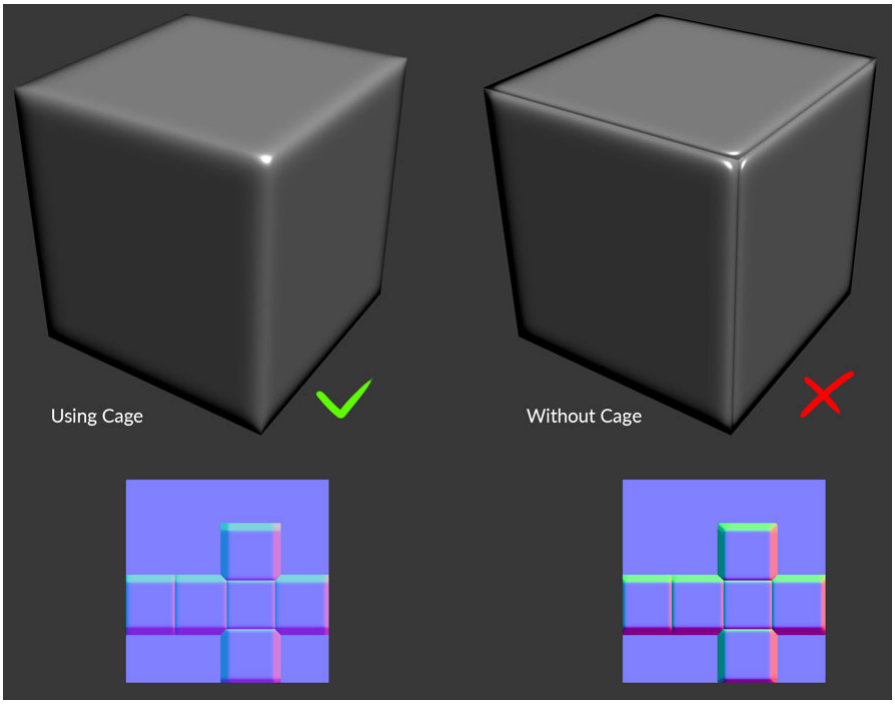


図 6-17 ケージの使用の比較

ケージとは基本的に、ポリゴン数の少ないモデルの大型バージョンまたは押し出しバージョンを指します。ベーキングが正しく機能するよう、ポリゴン数の多いモデルを含めます。

メッシュ ケージは、法線マップのベーキング時に使用されるレイ キャスト距離を制限するために使用されます。次の画像に示されているように、ケージは法線マップ上の分割法線の継ぎ目の問題も解決できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

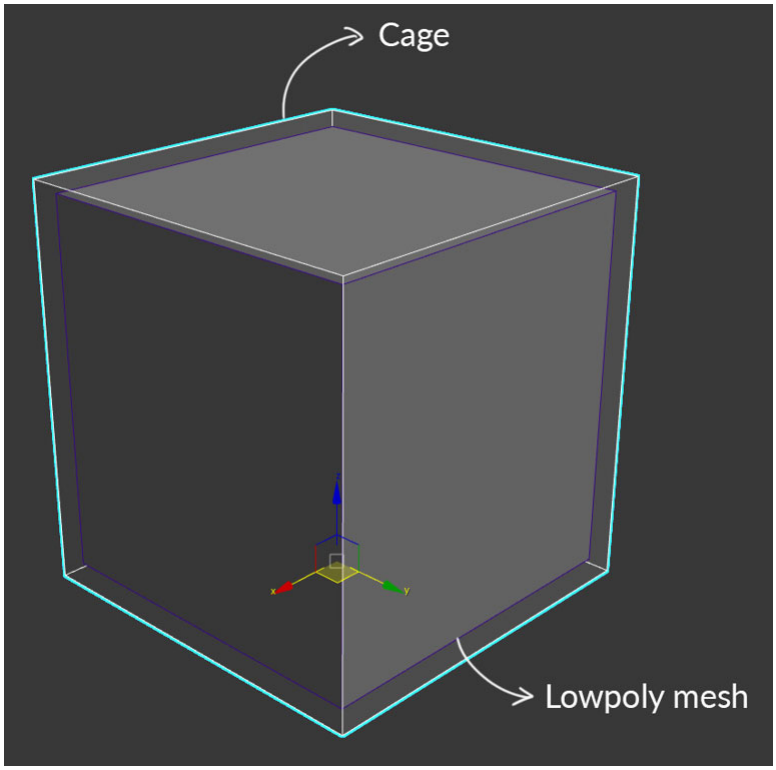


図 6-18 メッシュ ケージのサンプル

ベーキング ソフトウェアでサポートされている場合は、メッシュ名を照合してベーキングします。これにより、間違った法線マップ投影が作成される問題が軽減されます。オブジェクト同士が近すぎると、間違った面に法線マップが意図せず投影される可能性があります。この方法に従うことで、名前が一致する正しいサーフェスでのみベーキングが実行されるようになります。

メッシュ名の照合の詳細については、*substance3d の Web サイト*に加え、*Marmoset Toolbag のチュートリアル*を参照してください。

ベーキングの際にメッシュ名を照合できない場合は、メッシュを分解します。メッシュを分解すると、パーツが互いに分離され、不要なサーフェスに法線マップが投影されなくなります。これにより、不正確な法線マップの投影の問題も解消できます。

次の画像は、分解されたメッシュのサンプルを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

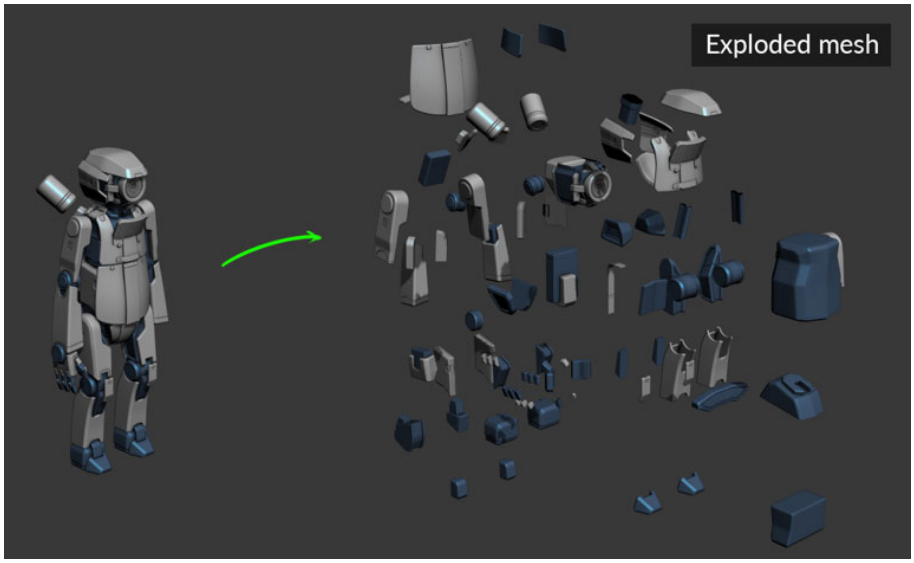


図 6-19 法線ベーキング用に分解されたメッシュのサンプル

このソリューションでは、アンビエント オクルージョンを個別にベーキングすることが必要になる場合があります。ハード エッジ上に連続した UV があると継ぎ目が目立つため、ハード エッジ上の UV を分割します。原則は、角度を 90 度未満に保つか、UV を別のスムージング グループとして設定することです。三角形上のさまざまなスムージング グループと UV の継ぎ目を一致させます。

次の画像は、ハード エッジ上での分割 UV の見え方のサンプルを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

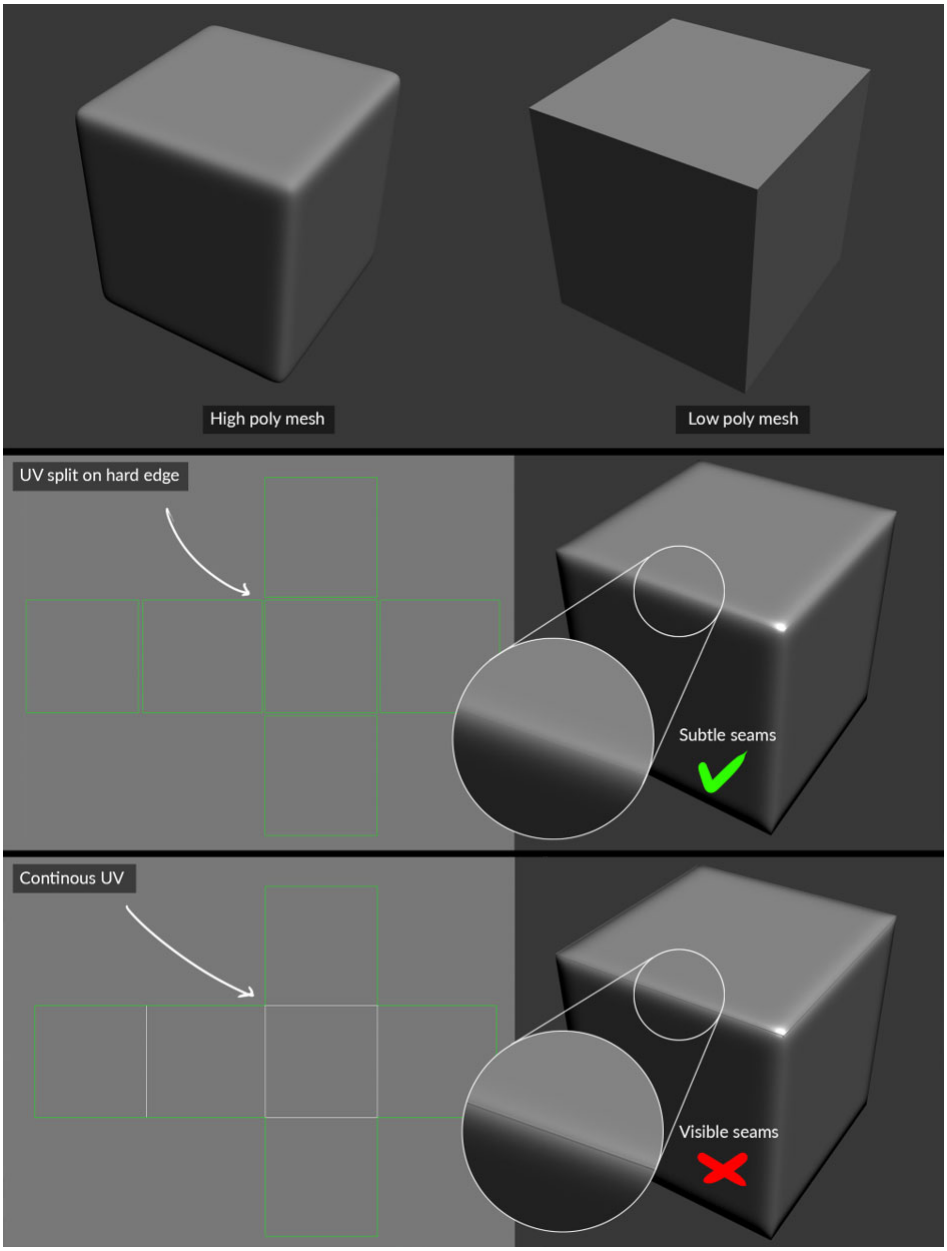


図 6-20 ハード エッジ上の分割 UV のサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

6.8 テクスチャ設定の編集

Unity ではテクスチャ設定を簡単に編集できるようになりました。編集するテクスチャを選択して、インスペクタ ウィンドウにテクスチャ設定を表示します。

テクスチャ設定を編集する場合のヒントをご紹介します。

- [Texture Type] では、エンジン内でテクスチャをさまざまな方法で使用できるよう、テクスチャのタイプを制御できます。
- [Texture Shape] では、一部のテクスチャ タイプに対して 2D ではなくキューブ マップを選択できます。
- [Texture Type] で選択したテクスチャ タイプに応じて、次の 2 つのセクションのいずれかが表示されます。
  - [Texture Settings] には、テクスチャを必要に応じて機能させるための特定のコントロールがあります。
  - [Advanced Settings には、[sRGB]、[Alpha Source]、[Alpha Is Transparent]、[Read/Write Enabled]、[Generate Mip Maps] が表示されます。
- [Wrap Mode] は、テクスチャによる UV のラッピング方法を制御します。使用可能なオプションは次のとおりです。
  - [Repeat] はテクスチャを並べて表示し、パターンを繰り返すために使用されます。
  - [Clamp] は、テクスチャをエッジの最後のピクセルに固定します。
  - [Mirror] は [Repeat] と似ていますが、テクスチャの繰り返しを 1 回おきにミラーリングします。
  - [Mirror Once] は、テクスチャをエッジ ピクセルに固定する前に、テクスチャを 1 回ミラーリングします。
- [Filter Mode] は、テクスチャで使用されるテクスチャ フィルタを制御します。
- テクスチャ圧縮ボックスでは、[Max Size]、[Resize Algorithim]、[Format]、[Compression]、[Use Crunch Compression] を制御できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 7 章
リアルタイム 3D アートのベスト プラクティス: マテリアルとシェーダ

この章では、ゲームのより効率的な実行とより良い外観を実現するための、複数の異なるマテリアルとシェーダの最適化について説明します。

注意

このコンテンツの最新バージョンは、Arm の開発者 Web サイト(*https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/real-time-3d-art-best-practices-materials-and-shaders*)にあります。

以下のセクションから構成されています。

- *7.1 シェーダとマテリアルの概要*(7-116 ページ).
- *7.2 モバイル プラットフォーム用に最適化されたシェーダの使用*(7-118 ページ).
- *7.3 テクスチャの最適化*(7-119 ページ).
- *7.4 ライティングなしシェーダとライティングありシェーダの比較*(7-120 ページ).
- *7.5 透明度を適用する場合の注意点*(7-122 ページ).
- *7.6 透明度の実装方法のプロファイリングと比較*(7-124 ページ).
- *7.7 補足資料とシェーダのベスト プラクティス*(7-126 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

7.1 シェーダとマテリアルの概要

マテリアルとシェーダは 3D オブジェクトの外観を決定するため、それぞれの機能と最適化する方法を知っておくことが重要です。

シェーダとは、画面上にオブジェクトを描画する方法と、そのオブジェクトに対して行う必要があるすべての計算を GPU に指示する小さなプログラムのことです。シェーダは、マテリアルにアタッチされている場合にのみ使用できます。

シェーダの記述に使用される一般的なスクリプト言語は、*高級シェーディング言語* (HLSL) と *OpenGL シェーディング言語* (GLSL) の 2 つです。

マテリアルは、オブジェクトまたはメッシュに適用して、そのオブジェクトのビジュアル外観を定義します。マテリアルは、シェーダから利用できるパラメータに対する特定の値を設定するために使用されます。たとえば、色、テクスチャ、数値などです。

次の 2 つのスクリーンショットは、シェーダを作成してさまざまなマテリアルで使用する 2 通りの方法を示しています。

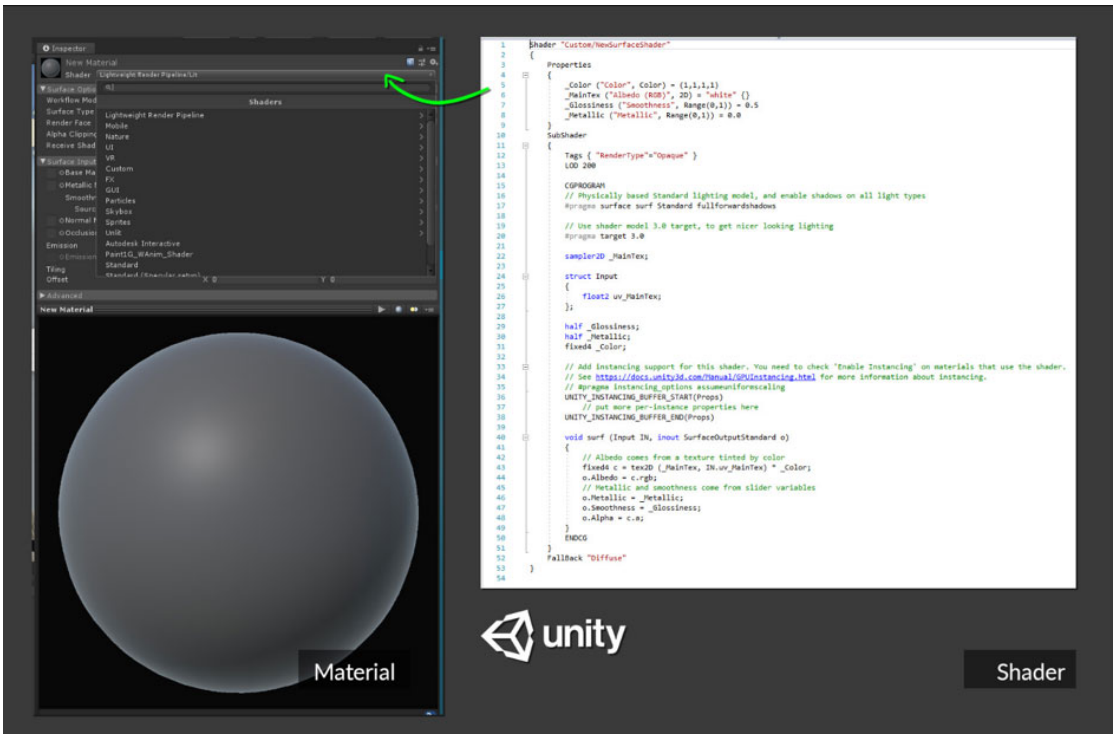


図 7-1 シェーダとマテリアル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

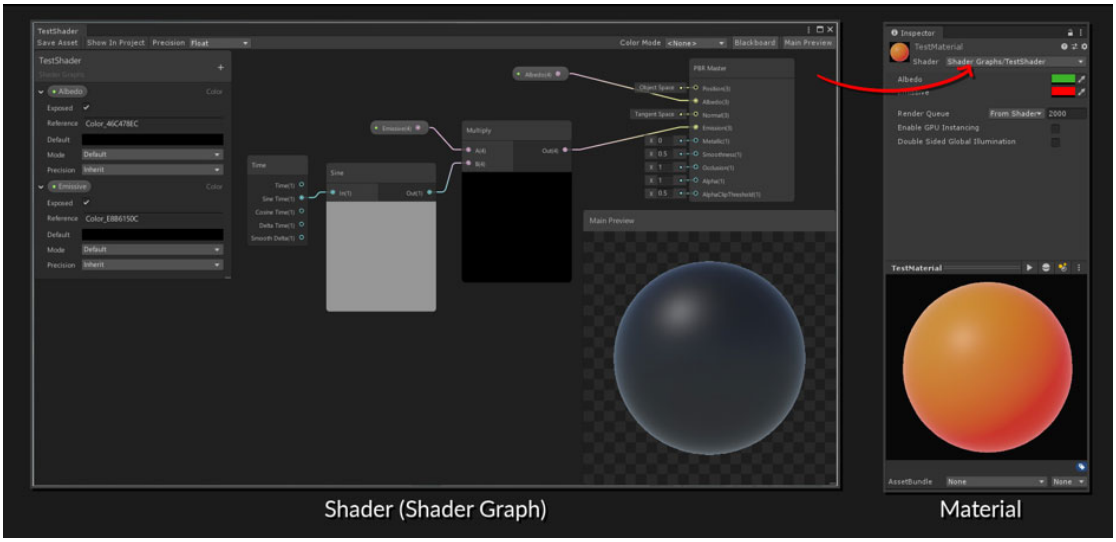


図 7-2 シェーダ グラフとマテリアルのメニュー

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

7.2 モバイル プラットフォーム用に最適化されたシェーダの使用

Unity には、モバイル プラットフォーム用に最適化されているシェーダのコレクションが用意されています。これらのシェーダは Unity 内のモバイル カテゴリに含まれています。

最適化されているこれらのシェーダには簡素化された機能が含まれており、モバイルベースのプラットフォームで優れたパフォーマンスを発揮できるようにします。ただし、モバイル以外の標準のシェーダと比較すると、利用できる機能は限られています。たとえば、色つけはできません。

次のスクリーンショットは、モバイル専用シェーダがある場所を示しています。

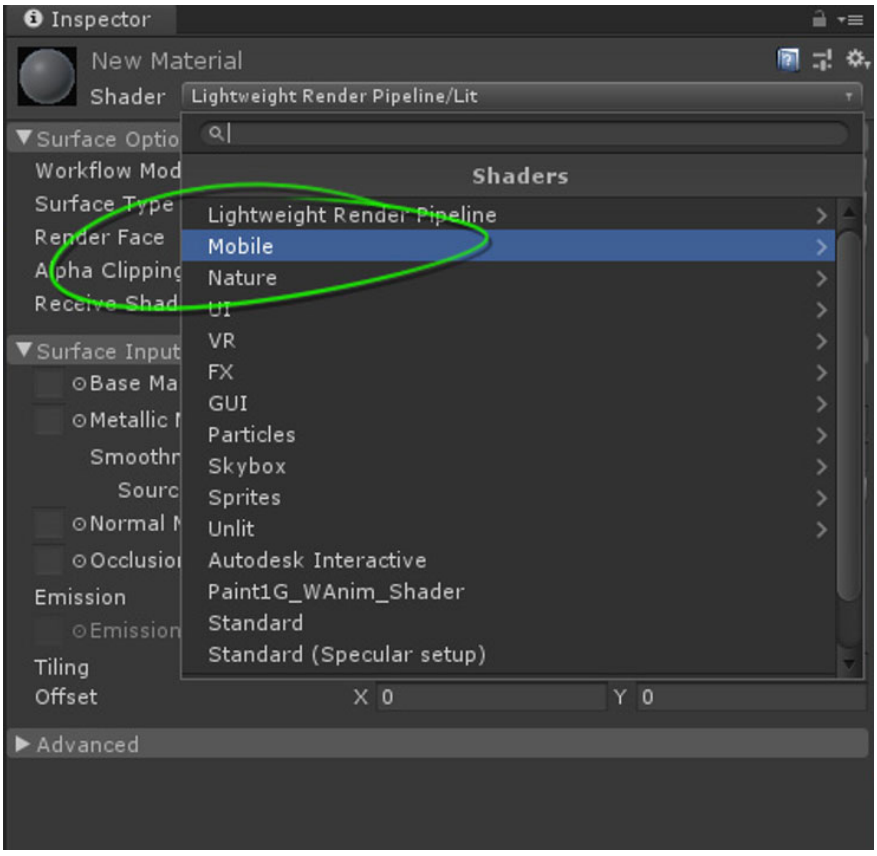


図 7-3 Unity モバイル シェーダの場所

必要な機能のみを使用してください。Unity では、マテリアルで選択され設定されているシェーダに基づいて、最適化されたランタイム バージョンを作成します。このようにしてシェーダの複雑度を軽減することで、モバイル プラットフォームでのパフォーマンスが向上します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

7.3 テクスチャの最適化

モバイル プラットフォームでは、使用するテクスチャをできるだけ少なくしてください。使用するテクスチャが多いと、テクスチャ フェッチが増えるためです。テクスチャ フェッチが増えると、使用される帯域幅も多くなり、デバイスのバッテリ寿命に影響します。

より多くのテクスチャがメモリに保存されるため、アプリケーション サイズも増大します。ラフネス テクスチャとメタリック テクスチャに個々のテクスチャを使用する代わりに、これらを 1 つのテクスチャのチャネルにまとめることができます。この技法はテクスチャ パッキングと呼ばれ、使用するテクスチャの数を削減するうえで役立ちます。

次の画像は、3 つの個別のテクスチャを 1 つのチャネルにまとめて、帯域幅を節約する方法のサンプルを示しています。

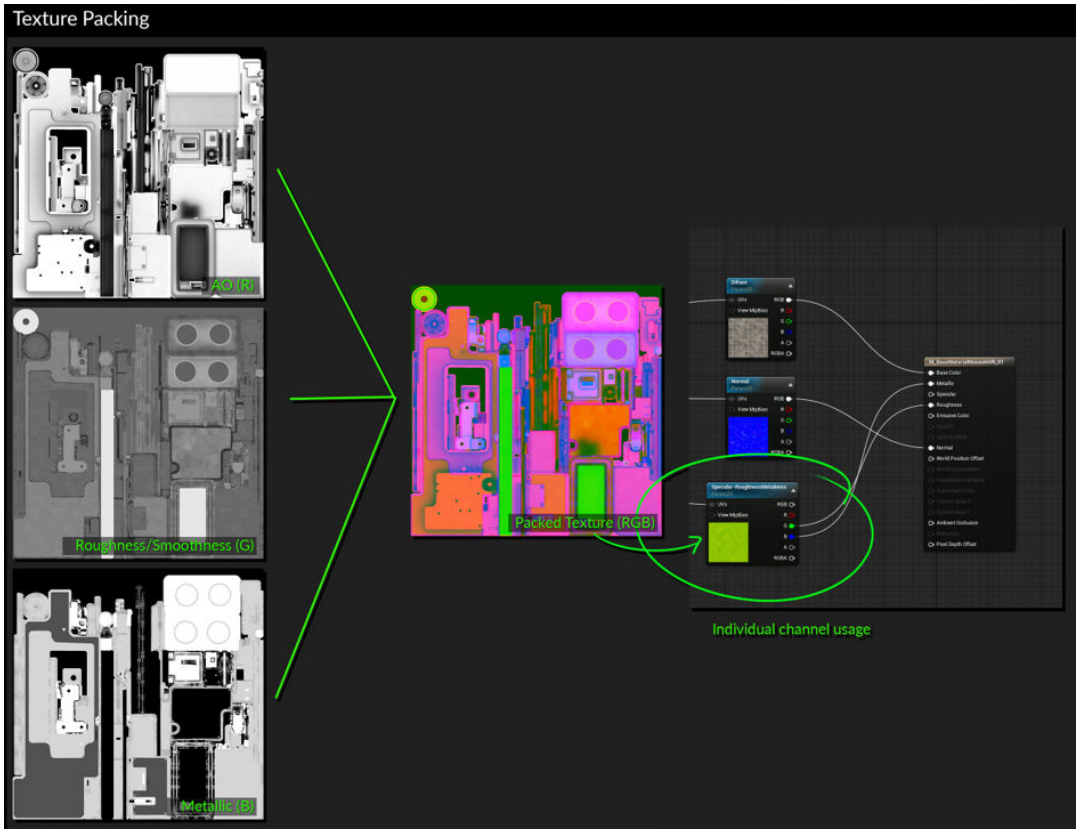


図 7-4 Unity でのテクスチャ パッキング

メタリック、ラフネス、またはスムーズネスなどの一部のパラメータには、テクスチャの代わりに数値を使用することもできます。できればこれを行って、ビジュアル品質に影響があるかどうかを観察してください。数値を使用すると、使用するテクスチャの数がさらに減少します。

注

Unity では、シェーダ内で値を追加する必要があります。

ライトはマテリアルに影響しないため、ライティングなしのシェーダを使用した場合も、使用するテクスチャの数が減少します。これは、ラフネス テクスチャまたはメタリック テクスチャは必要ないことを意味します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

7.4 ライティングなしシェーダとライティングありシェーダの比較

シェーダを作成するときに、マテリアルが光にどのように反応するかを指定できます。モバイル ゲームで使用される大半のシェーダは、ライティングありまたはライティングなしに分類されます。

ライティングなし

ライティングなしシェーダは、最速かつ最小コストのシェーディング モデルです。ローエンド デバイスをターゲットにしている場合は、このタイプを使用します。考慮すべき重要なポイントは次のとおりです。

- ライティングは、ライティングなしのシェーディング モデルには影響せず、放射のみを色として出力します。
- そのため、鏡面反射などの多くの計算が必要ありません。その結果、レンダリングが低コストまたは高速になります。
- 漫画のように定型化されたアート ディレクションは、ライティングなしシェーディングに適しています。そのため、このアート スタイルは、モバイル プラットフォーム用のゲームを制作する場合に検討する価値があります。

ライティングあり

ライティングありシェーダは、ライティングなしシェーダよりも多くの処理能力を使用します。ただし、以下のような例外もあります。

- ライトはライティングありシェーダに影響を与え、サーフェスに鏡面性を持たせることができます。
- おそらく、今日のモバイル ゲームで最も使用されているシェーディング モデルです。

次の画像は、ライティングありオブジェクトとライティングなしオブジェクトの比較を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

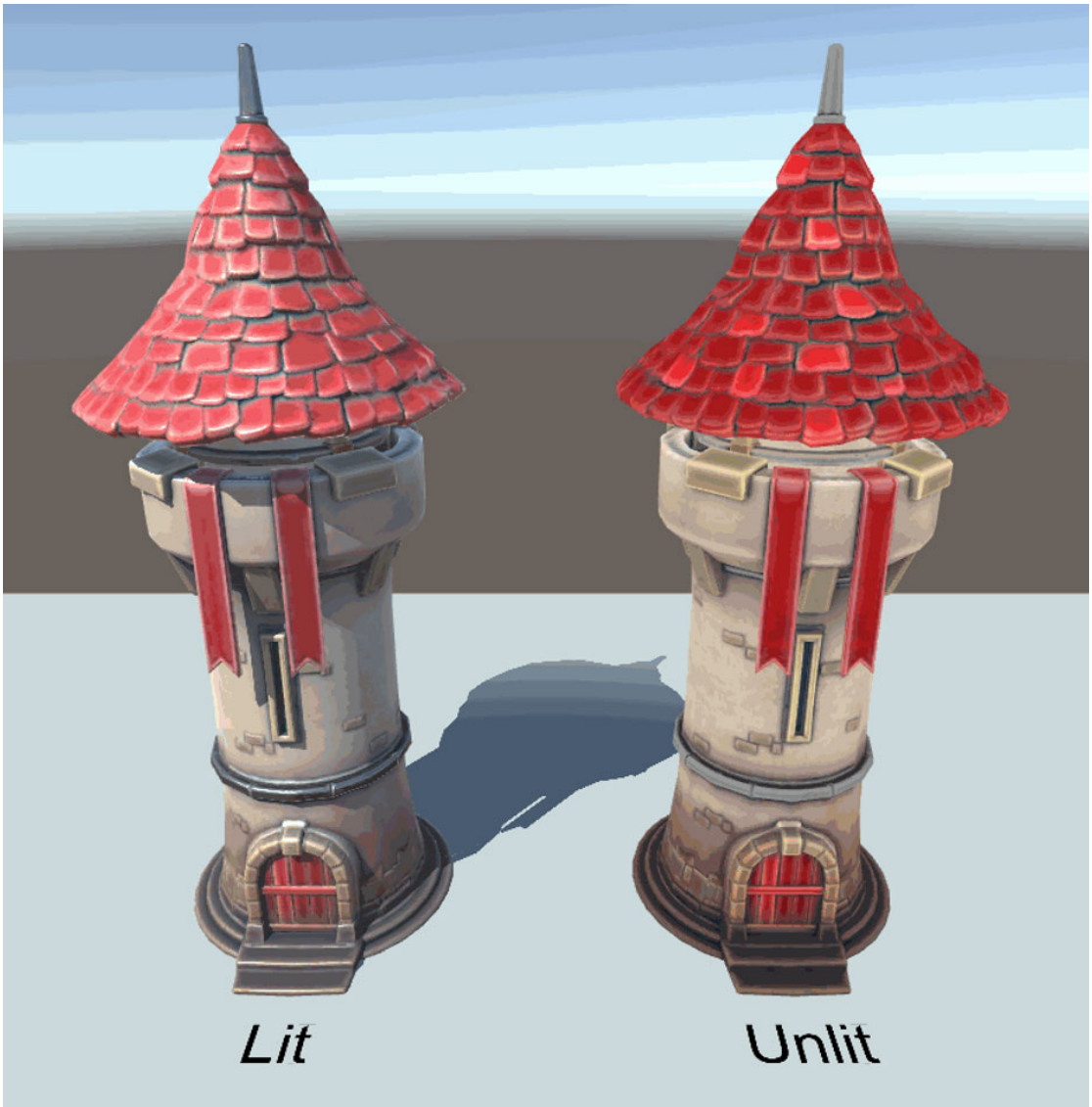


図 7-5 ライティングありシェーダとライティングなしシェーダの比較

同じタワー メッシュとテクスチャ セットが適用されていますが、使用されているシェーダが異なります。ライトはライティングなしシェーダに影響を与えないため、必要な計算は少なくなります。その結果、特に性能の低いデバイスではゲームのパフォーマンスが向上します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

7.5 透明度を適用する場合の注意点

可能な限り不透明なマテリアルを使用してください。モバイル プラットフォームでは、必要な場合を除き、透明度の適用は避けることをお勧めします。

透明なオブジェクトをレンダリングすると、不透明なオブジェクトをレンダリングする場合よりも多くの GPU リソースが使用されます。透明オブジェクトを多数使用すると、モバイル プラットフォームのパフォーマンスに影響します。このことは、透明オブジェクトが複数回重ねてレンダリングされる場合に特に当てはまります。

同じピクセルが画面上に複数回描画されることを、オーバードローと呼びます。オーバードローが問題となるのは、透明なレイヤーが多いほど、レンダリングのコストが増加するためです。モバイル プラットフォームの場合、オーバードローはパフォーマンスに深刻な影響を与える可能性があります。

次のスクリーンショットからわかるように、青色のライトは透明効果を出すのに効果的かもしれませんが、不透明なマテリアルでも優れた外観を得ることができます。

図 7-6 不透明なマテリアルの使用

そのため、レベルをビルドするときは、オーバードローを最小限に抑えることが推奨されます。Unity には、シーン内のオーバードローの量を確認できるモードがあります。このモードは、次のスクリーンショットで取り上げています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

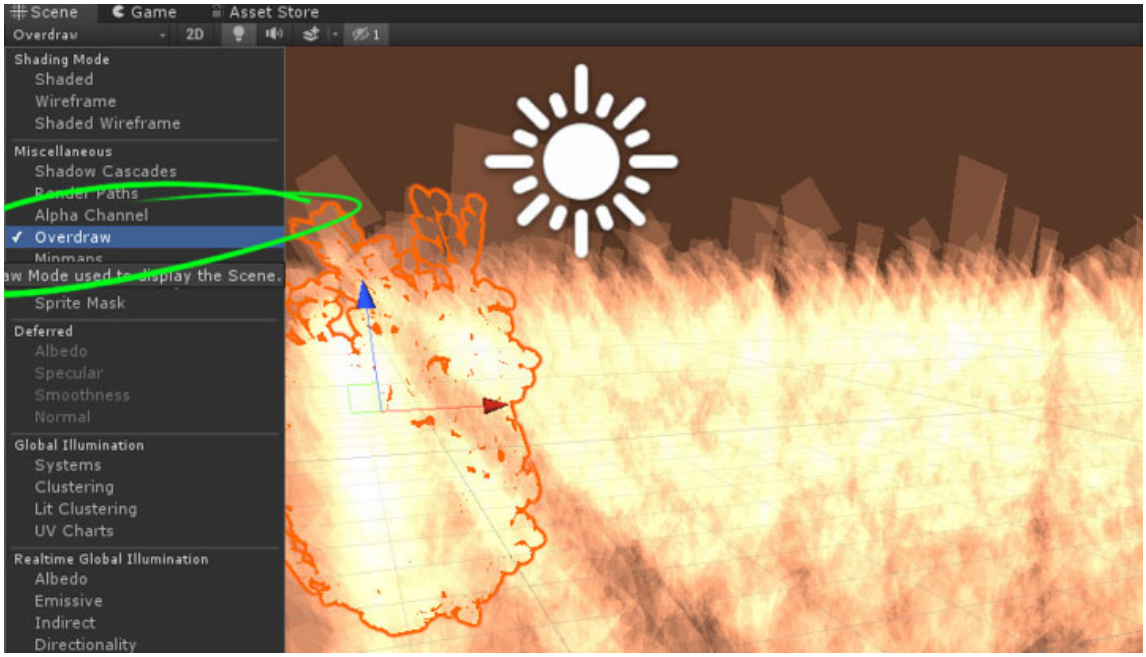


図 7-7 Unity でのオーバードローの可視化

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

7.6 透明度の実装方法のプロファイリングと比較

シェーダで透明度を実装する方法は多数あり、最も一般的に使用されるのは、アルファ テストとアルファ ブレンドです。ただし、どちらの方法が最高のパフォーマンスを発揮するかは、ユース ケースに応じて異なります。2 つのパフォーマンスの違いを常にプロファイリングすることをお勧めします。

モバイル プラットフォームでは、ARM ツールの *Streamline* を使用して、デバイスのパフォーマンス データを収集できます。

アルファ テストまたはカットアウト
: アルファ テストを実装すると、オブジェクト マテリアルは 100% 不透明になるか 100% 透明になります。マスクのカットアウトのしきい値を設定できます。Unity では、この透明度のタイプを**カットアウト**と呼びます。

アルファ ブレンド
: 視覚的には、アルファ ブレンドを使用すると、マテリアルにさまざまな透明度を適用し、オブジェクトを不透明または完全に透明にするのではなく、一部を透明にすることができます。Unity では、このブレンド モードを**透明度**と呼びます。

アルファ ブレンドでは一部を透明にできますが、アルファ テストではカットアウトがシャープになります。アルファ ブレンドとアルファ テストの比較は、次の画像で確認できます。

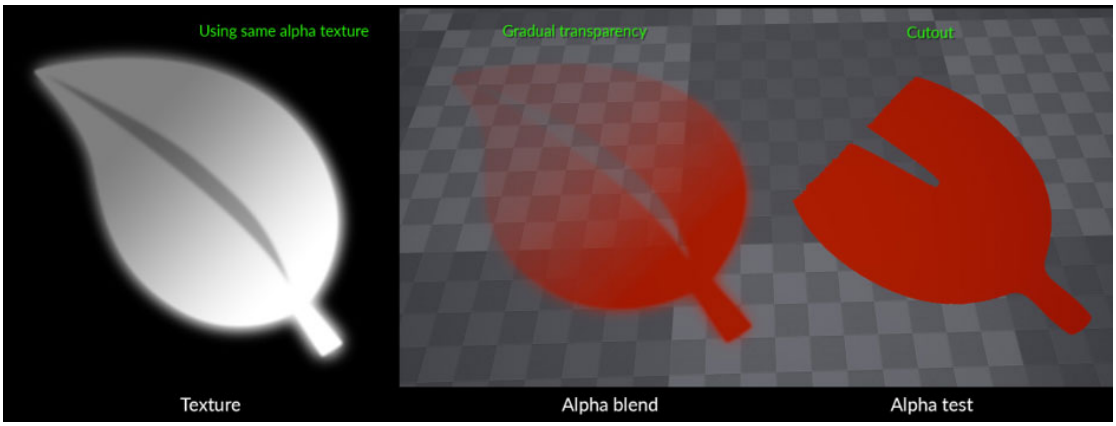


図 7-8 アルファ ブレンドとアルファ テストの比較

アルファ テスト

ゲームでアルファ テストを使用する場合に覚えておくべきヒントをご紹介します。

- 必要な場合を除き、アルファ テストまたはカットアウトは避けるのが最善です。
- アルファ テストでは、マテリアルは 100% 不透明または 100% 透明になります。
- アルファ テストでは、GPU 内の一部の最適化機能が無効になるため、ターゲット モバイル プラットフォームでテストすることをお勧めします。また、パフォーマンスの違いについてプロファイリングを行い、アルファ ブレンドと比較することが重要です。

アルファ ブレンド

ゲームでアルファ ブレンドを使用する場合に覚えておくべきヒントをご紹介します。

- モバイル プラットフォームの場合、Unity では、*アルファ テストよりもアルファ ブレンドを使用することを推奨*しています。実際には、アルファ テストとアルファ ブレンドのパフォーマンスはコンテンツに依存し、評価が必要になるため、プロファイリングしてパフォーマンスを比較する必要があります。
- 一般に、モバイル プラットフォームでは、アルファ ブレンドで透明度を使用しないでください。
- アルファ ブレンドが必要な場合は、ブレンド領域の範囲を小さくしてください。

樹木の葉にアルファ テストの透明度を使用

樹木の葉の静的ビューでは、ソフト エッジによってアルファ ブレンドの方が外観が向上します。次の画像は、これをアルファ テストのシャープ カットと比較したものです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

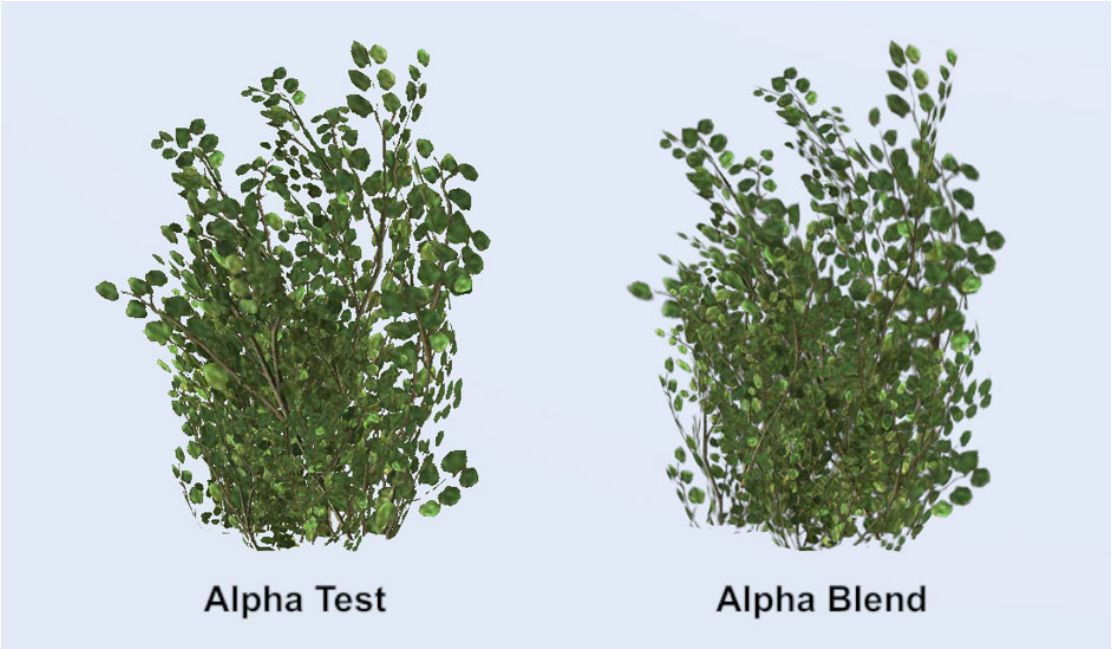


図 7–9 アルファ テストとアルファ ブレンドの樹木の葉のサンプル

ただし、シーンが動いている場合、アルファ ブレンドは適切に表示されず、葉は正しい順序でレンダリングされません。アルファ テストは、透明度の適用と葉のレンダリング順序がはるかに優れています。ただし、アルファ ブレンドと比較すると、エッジは粗くなるかエイリアシングされます。

通常、アルファ テストのビジュアル品質は許容範囲内です。これは、動いているシーンでは、エイリアシングされたエッジが目立たないためです。ただし、葉や枝でアルファ ブレンドの使用を切り替えると、錯視が損なわれます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

7.7 補足資料とシェーダのベスト プラクティス

ARM では、ゲームの開発中は以下の追加のベスト プラクティスを念頭に置いておくことを推奨します。

オーバードローが避けられない場合は、シェーダをできる限りシンプルにします。以下の原則に従ってください。

- ライティングがないなど、できる限りシンプルなシェーダを使用し、不要な機能の使用を避けます。
- Unity にはパーティクル専用に設計されたシェーダがあります。これは、最初の段階の実装に適しています。
- オーバードローを最小限に抑えます。そのためにはパーティクルの数、つまりサイズを削減します。

プロファイル シェーダの複雑性

テクスチャ サンプラーや透過性などの機能を追加すると、シェーダがさらに複雑になり、レンダリングに影響が及ぶ可能性があります。このため、シェーダのプロファイリングは頻繁に行うことをお勧めします。

ARM では、それを行うための、*Mali オフライン シェーダ コンパイラ*や *Streamline* などのツールを提供しています。ただし、これらのツールを活用するには、GPU がどのように機能するかを詳しく理解するために、グラフィックに関する高度な知識が必要です。シェーダの編集を担当している場合は、この分野についてさらに深く学ぶことをお勧めします。

バーテックス シェーダでできるだけ多くの演算を行う

プロジェクトでは、バーテックス シェーダとピクセル、またはフラグメント シェーダを組み合わせて、特定の外観を生み出すことが一般的です。バーテックス シェーダは各バーテックスに適用され、ピクセル シェーダは各ピクセルに適用されます。

通常、画面には頂点よりも多くのピクセルがレンダリングされます。つまり、ピクセル シェーダはバーテックス シェーダよりも頻繁に実行されます。可能な限り、ピクセル シェーダからバーテックス シェーダに計算を移行することをお勧めします。

通常、演算をバーテックス シェーダに移行することは、可変変数を経由して、処理データをピクセル シェーダに移行することを意味します。これは一般的に優れたアイデアですが、タイラーに注意して、それがボトルネックにならないようにする必要があります。最適化を行った後は、通常通りにさらにプロファイリングを行う必要があります。

可能な限り複雑な数学演算の使用を避ける

シェーダ内では数学演算を使用して、外観と動作をカスタマイズします。これらの数学演算の例には、乗算、加算、累乗、床、対数、平方根などがあります。

これらの数学演算のパフォーマンス コストはそれぞれ異なります。したがって、使用には注意を払わなければなりません。より複雑な演算には、sin、pow、cos、除算、ノイズなどがあります。

加算や乗算などの基本演算は処理が速いため、時間のかかる数学演算の数はできるだけ少なくします。使用する複雑な数学演算量は、GLES 2.0 を使用しているデバイスなどの古いデバイスでは少なくする必要があります。

常にパフォーマンス プロファイリングを行う

アプリケーション内のどこでボトルネックが発生しているかを把握するには、パフォーマンスのプロファイリングを行います。プロファイリングは、使用する最適化の実施前後の効果を比較する場合にも推奨されます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 8 章
リアルタイム 3D アートのベストプラクティス: ライティング

この章では、ゲームの見栄えを良くして、スムーズに実行するのに役立ついくつかのライティングの最適化について説明します。

注意

このコンテンツの最新バージョンは、Arm の開発者 Web サイト(*https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/real-time-3d-art-best-practices-lighting*)にあります。

以下のセクションから構成されています。

- *8.1 ライティングの概要*(8-128 ページ).
- *8.2 レンダーパイプライン*(8-129 ページ).
- *8.3 ライトモード*(8-130 ページ).
- *8.4 静的なライトの使用(可能な場合)*(8-131 ページ).
- *8.5 可能な限りベイクする*(8-132 ページ).
- *8.6 ライトマップの最適化*(8-134 ページ).
- *8.7 可能な限りフェイクする*(8-138 ページ).
- *8.8 ライトプローブ*(8-139 ページ).
- *8.9 メッシュレンダラーの設定*(8-141 ページ).
- *8.10 リアルタイムライトとライトのタイプ*(8-142 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.1 ライティングの概要

ライティングは、ゲームの最も重要な要素の 1 つです。なぜなら、ライティングによってムードを盛り上げ、ゲームプレイをリードして、スレットやオブジェクトを識別できるためです。

ライティングは、ゲームのビジュアルを良くすることもあれば悪くすることもあります。たとえば、単純なモデルでも優れたライティング技術を使えばゲームでの見た目が良くなり、ディテールの細かいモデルでもライティング技術が良くないと見た目が悪くなることがあります。

Unity ゲームエンジンを使用すると、ライティングの操作が簡単でわかりやすくなります。モバイルゲームのパフォーマンスはライティングの決定に左右されるため、ライティングを効果的に使用する必要があります。

このセクションでは、以下について説明します。

- 静的なライトと動的なライトの違い。
- ライトの最適化。
- ライティングをフェイクする方法。
- リアルタイムライトとライトのタイプ。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.2 レンダーパイプライン

Unity の組み込みレンダーパイプラインでは、フォワードレンダリングとディファードシェーディングという 2 つのレンダーパスを選択できます。

フォワードレンダリングでは、リアルタイムライトは非常にコストがかかりますが、このコストを埋め合わせるために、1 ピクセルあたりにレンダリングするライトの数を選択することができます。

ディファードシェーディングは GPU サポートを必要としますが、対応するハードウェア上では、多くのリアルタイムライトをレンダリングし、高いレベルのライティングの忠実度を必要とします。

ディファードシェーディングは、ライトを多用する PC やコンソール向けのゲームには大変魅力的ですが、モバイル GPU ではあまり高いパフォーマンスは発揮されません。その原因は、量産向けデバイスの帯域幅が狭いことです。モバイル向けタイトルを制作するときは、作業ができるだけ多くのデバイスでスムーズに動作することが重要です。そのため、ユーザーをサポートするために、Unity はユニバーサル・レンダー・パイプライン(URP)を提供しています。

これは、ビルド済みスクリプタブル・レンダー・パイプラインの一種であり、パフォーマンス重視のチューニングが施されています。URP は、アプリが最大限のパフォーマンスを発揮できるように支援する機能を多数備えています。詳細については、こちらをご覧ください。これらの最適化のために、ライティングプロジェクトでは URP を使用することをお勧めします。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.3 ライトモード

ライトにはさまざまなモードがあります。これらのモードは、ライトの移動とシーン内での使用方法に関連しています。モードはパフォーマンスの点で異なるため、ライトを実装する際にはライトモードを考慮することが重要です。

ベイク、混合、リアルタイムの 3 つの異なるライティングモードを使用する場合のメリットとデメリットを見ていきます。

ベイク

ベイクライトモードは、静的なライティングを提供します。オブジェクトは実行時にライティングを変更しません。ライトのベーキングは、ゲームを実行する前に、ライティングデータをテクスチャマップに保存するプロセスです。

ベイクライトモードの主な特徴は以下のとおりです。

- 実行時にライトを変更することはできません。光と影はライトマップにベイクされます。この処理は、Unity でライティングを作成する際に実行され、実行後のパフォーマンスには影響しません。
- 影は静的で、ゲームプレイ中に動的オブジェクトまたは動くオブジェクトでは奇妙に映る場合があります。
- ベイクライトモードは、このガイドで説明する中で計算コストが最もかからない方法です。

混合

混合ライトモードは、動くオブジェクトに固定ライトを提供します。これは、他の 2 つの方法を混合したものと考えることができます。

混合ライトモードの主な特徴は以下のとおりです。

- 動的な直接光のライティングとシャドウ。
- 静的オブジェクトのライトマップ計算にライトを含めることができます。
- ライトは、動的オブジェクトに影響を与えます。オブジェクトの影の生成も含まれます。
- 強度は実行時に変更でき、直接光のみが更新されます。
- 混合ライトモードは、コストのかかる計算方法です。

リアルタイム

リアルタイムライトモードは、動的なライトまたは可動ライトを提供します。ライティングを操作するうえで最もコストのかかる複雑な方法です。

リアルタイムライトモードの主な特徴は以下のとおりです。

- 動的なライトとシャドウ。プロパティはライトマップにベイクされるのではなく、実行時に変更できます。
- リアルタイムライトモードは、このガイドで説明する中で計算コストが最もかかる方法です。

Unity ライティングパイプラインの詳細については、Unity のドキュメントを参照してください。*関連情報*にリンクがあります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.4 静的なライトの使用(可能な場合)

モバイルデバイスで作業する場合は、静的なライトを使用することが重要です。デバイス上で実行する方がコストがかからず、ゲームのエンドユーザーのエクスペリエンスが向上します。

動的なライティングまたはリアルタイムライティングは、フレームごとに計算され更新されます。これは、オブジェクトを移動し、対話性を高めて、シーンを情景的なものにするのに効果的な方法です。

対照的に、静的なライト情報はベイクできます。Unity は、Unity エディタでベイクされたライトの計算を実行し、その結果をライティングデータとして保存します。このプロセスをベーキングと呼びます。

計算が完了すると、実行時の値はそのシーンに必要な値だけになります。静的なライトのコストは常に、動的なライトよりもはるかに少なくなります。つまり、リソースが限られているモバイルゲームでは、静的なライトを最初に実装する必要があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.5 可能な限りベイクする

Unity を使用してモバイルプラットフォームでライティングするための最初のアプローチは、可能な限りベーキングすることです。

ライトマップベーキングは、イルミネーションやシャドウなどのライトのエフェクトを計算し、この情報をライトマップと呼ばれる別のテクスチャに保存するプロセスです。ライトマップを使用すると、オブジェクトの外観を強化できます。このベーキングプロセスを実行すると、オフライン計算を 1 回実行するだけで済みます。実行時に余計なパフォーマンスコストが発生することはありません。

プリベイクされたライティングは、シーンの動的な要素や動きのある要素に対応していませんが、すべての静的要素のグローバルイルミネーションが含まれています。これは、各静的要素が、オブジェクトに反射した間接光と静的ライティングの直接光を取得することを意味します。以下の画像は、完全にベイクされたシーンのサンプルを示しています。

図 8-1 ベイクされたライティング設定(Armies の技術デモ)

Unity を使用すると、ライトを簡単にベイクできます。ライトをベイクする前に設定する必要がある主な手順が 2 つあります。

1. [Windows] > [Rendering] > [Lighting Settings]をクリックし、ベイクするライトを[Mixed]または[Baked]に設定します。
   a. モバイル向けタイトルの場合は、可能な限り混合するのではなく、ベイクされたライトを使用してください。ベイクはすべてのオプションの中で最もコストがかからないからです。
2. ベイクされたライトを受けるオブジェクトを静的とマークする:
   a. 静的とマークされるオブジェクトには複数の最適化が考えられますが、通常は設定ですべて選択しています。オブジェクトが静的としてマークされているため、Unity はライトのベーキングに含めることを認識します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

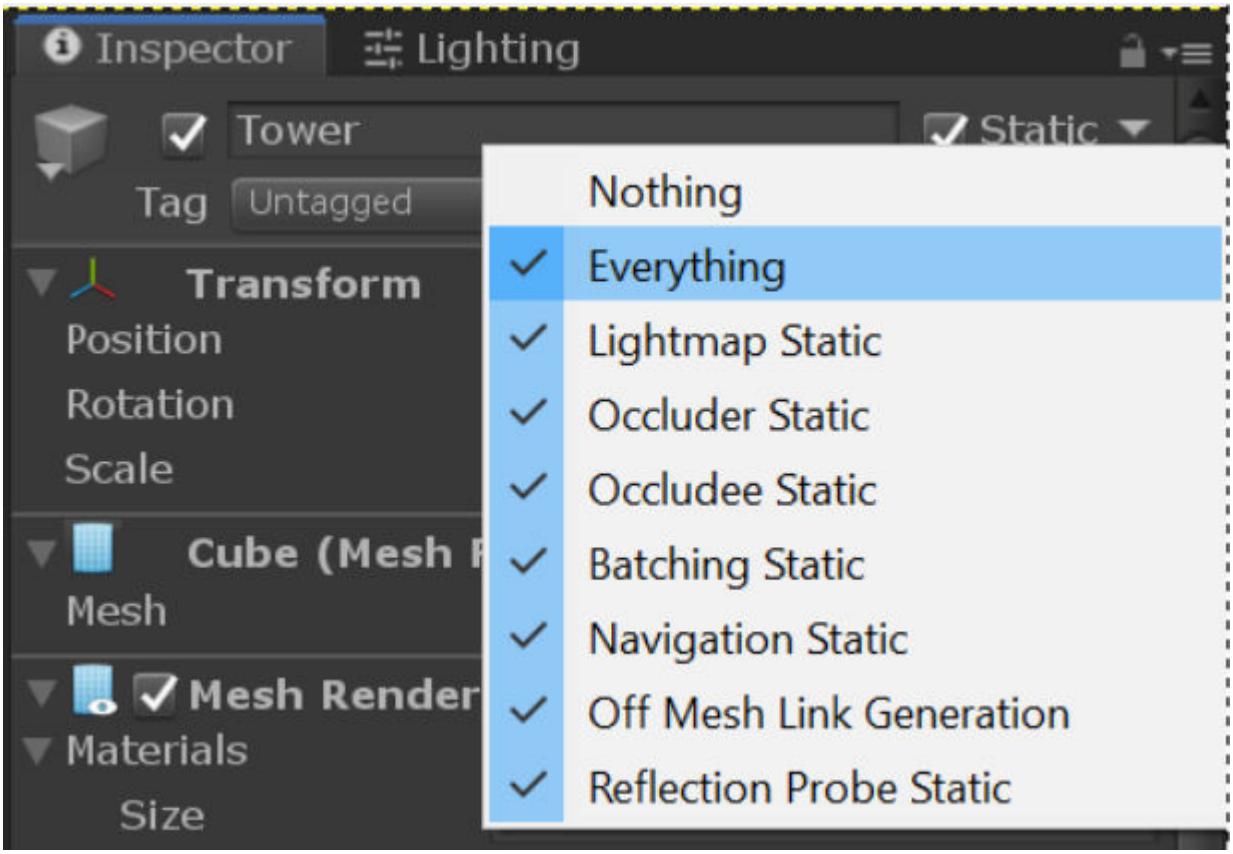


図 8-2 ベイクされたライトを静的としてマーク

注

[Batching Static]を有効にすると、マークしたオブジェクトを移動またはアニメーション化できなくなります。これがもう 1 つの最適化であり、可能な限りオンにしておく必要があります。

ライトをベイクする場合は、ベイクを開始したときにアクティブであったシーンに基づいてデータが保存されることに注意してください。ベイクしたシーンと同じ名前のフォルダが生成されます。このフォルダに、ライティングデータのすべてのコンポーネントが保存されます。

一度にロードされる複数のシーンをプロジェクトで使用する場合は、各シーンのライトをベイクする必要があります。シーンを調整する場合は、ライトをベイクし直す必要があります。

次の画像は、フォルダが作成される場所を示しています。

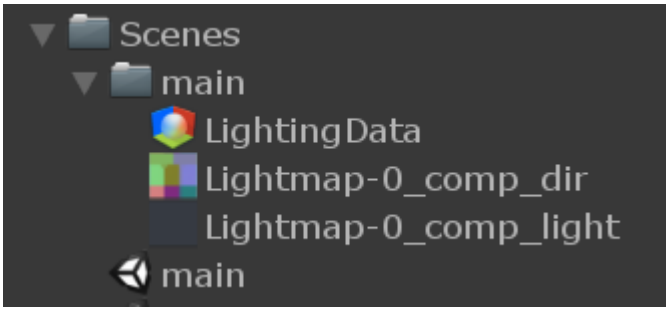


図 8-3 ベイクされたライトのフォルダ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.6 ライトマップの最適化

ライトをベイクするよう構成したら、ベイクされたマップが最適化されていることを確認する必要もあります。

ライトマップのサイズは、ベイクの設定によって異なります。モバイルプラットフォームではメモリ使用量を最小限に抑えなければならないため、ライトマップのサイズを監視する必要があります。次のサンプル画像では、7 つの 1024x1024 ピクセルライトマップが表示されています。

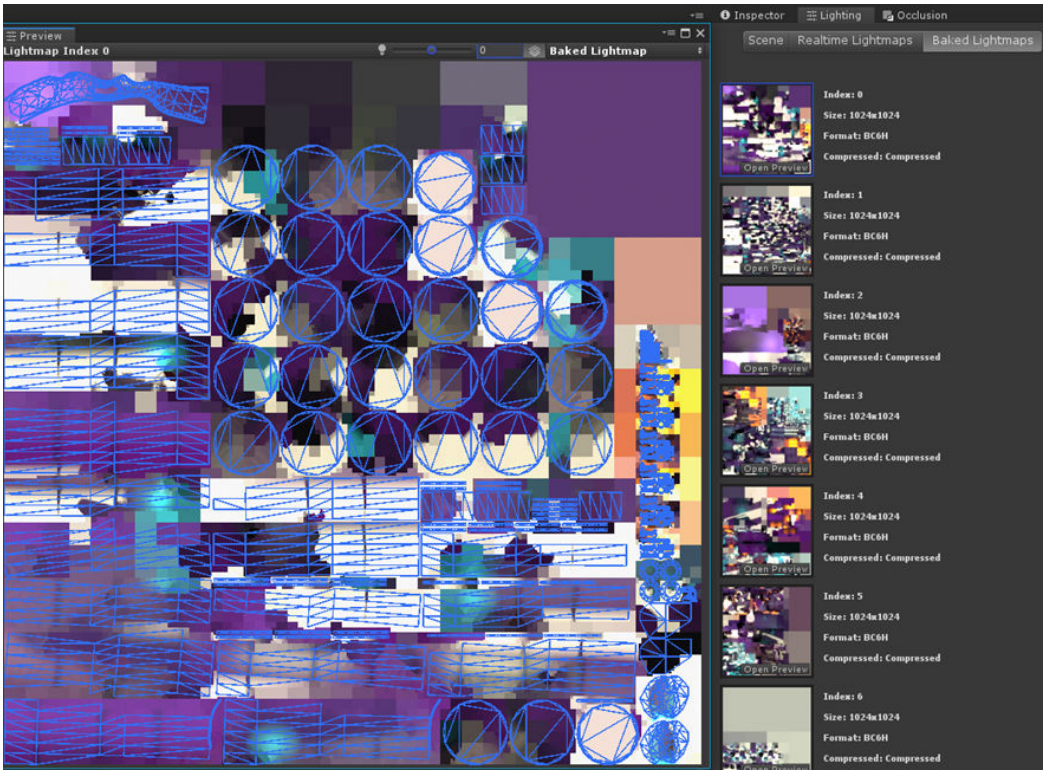


図 8-4 ベイクされたライトマップインデックスのサンプル

マップのプレビューでは、マップ上に配置されたメッシュが表示され、選択したメッシュが強調表示されています。

[Lightmapping]の一部の設定と、実際のマップのサイズによって、使用されるスペースの量が決まります。

次の各セクションでは、重要な設定について説明します。

ライトマッパー

Unity のライトマッパーには、シーン内のライトをベイクする方法が 3 つあります。

- Enlighten
- Progressive CPU
- Progressive GPU

Unity では、新しい作業にはプログレッシブオプションのいずれかを使用する必要があります。プログレッシブ・ライトマッパーはライトマップを段階的に作成するため、時間を大幅に節約できます。[Prioritize View]オプションを選択すると、シーンビューにある領域の優先順位が付けられます。ビューに優先順位を付けると、シーンのライティングを設定する反復時間が短縮されます。

CPU と GPU のプログレッシブ・ライトマッパーの大きな違いは、ライトマップが CPU で生成されるか、GPU で生成されるかです。2 つのオプションで得られる結果は同じですが、強力な GPU を使用すると、そのオプションははるかに高速になります。GPU オプションのその他の要件と設定手順については、こちらをご覧ください。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

テクセル

テクセル(テクスチャ要素)は、テクスチャマップ内の個々のピクセルを指します。テクセルは、たとえば、ライトがライトマップ内のオブジェクトに当たる各ポイントにライティング情報を保存するために使用されます。使用されているテクセルの数を数えることで、ライトでのベイクに必要な作業量を測定できます。

テクセルとは何か、また、テクセルの制御方法によって、ライティングの品質、ベイクの計算時間、ディスクストレージのコスト、およびライトマップの VRAM コストが変化する可能性があることを理解することが重要です。

必要なライトマップデータの量に最大の影響を与えるには、ベイクの各ユニットのテクセル数を調整する必要があります。これは、[Lightmapping Settings]で行うことができます。サンプル画像に示すように、これらの設定によって、各オブジェクトがベイクで使用するテクセルの数など、ライトマップを制御できます。

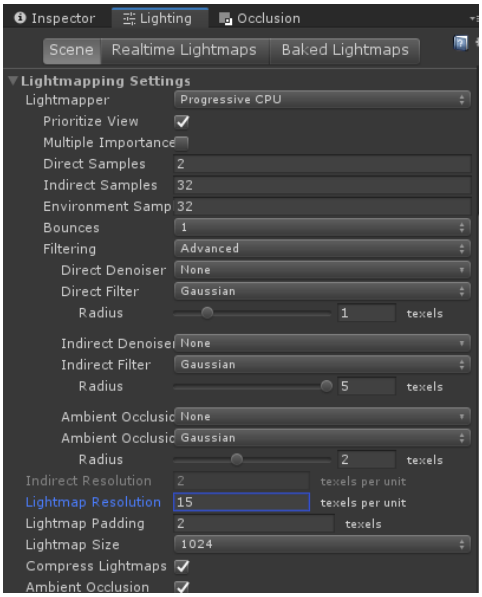


図 8-5 Lightmapping Settngs

[Lightmapping Settings]には、[Lightmap Resolution]と呼ばれるオプションが含まれています。このオプションは、ライトマップの各ユニットに使用されるテクセル数を設定します。

次の方法で、テクセルがシーン内でどのように配置されているかを確認できます。

- [Scene]ビューで[draw mode]ドロップダウンをクリックします。
- [Lightmap Indices]を見つけてクリックします。

ベイクされたオブジェクトは市松模様のオーバーレイで覆われるようになります。これが、ライトをベイクしたときにテクセルを分散させる方法です。

次のスクリーンショットは、[Lightmap Resolution]の設定が異なるキューブのサンプルを示しています。左側の画像の設定は 1、中央の画像の設定は 2、右側の画像の設定は 5 です。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

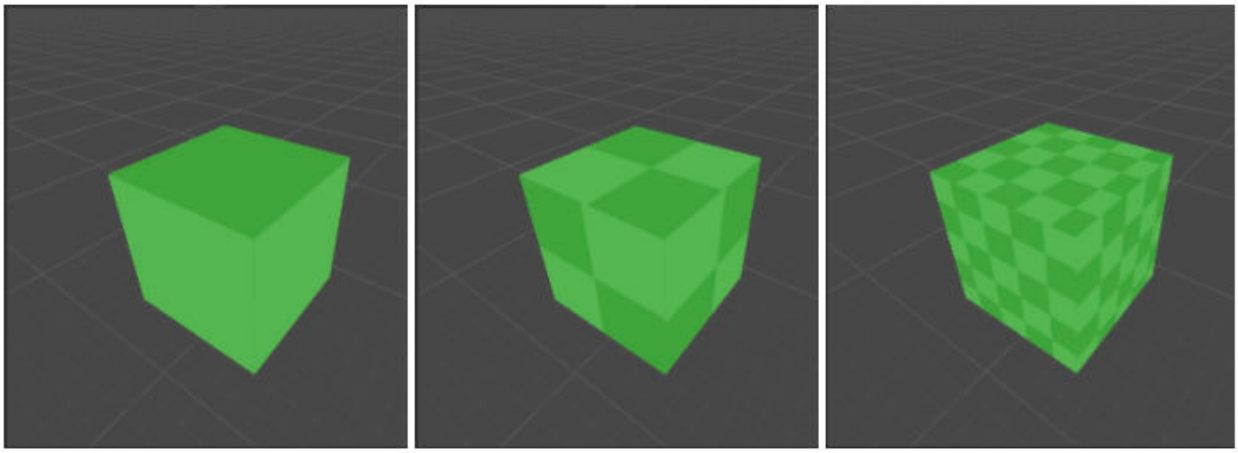


図 8-6 ライトマップの解像度の設定

解像度が高いほど、必要な作業量が急速に増えることがわかります。まずは[Lightmap Resolution]を 5 ～10 の低い値に設定して、シーンのニーズに応じて増減することをお勧めします。ライトマップの解像度を上げると、反復ごとにサイズが大幅に大きくなります。

たとえば、サンプルデモで[Lightmap Resolution]を 15 から 12 に下げると、次の画像に示されているように、必要なライトマップ数が 7 から 4 に減ります。

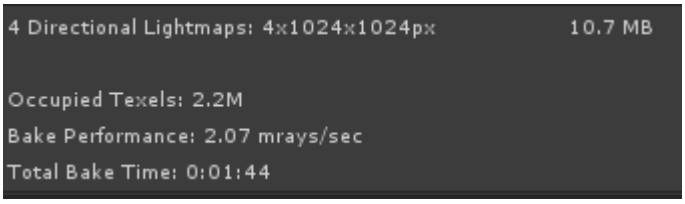


図 8-7 ライトマップの解像度のサンプル

テクセル使用量

[Lightmapping Settings]では、シーンがユニットごとに使用するテクセルの数を設定しますが、その数のテクセルを使用したくないオブジェクトもあります。

Unity では、各オブジェクトが使用できるテクセルの数を制御できます。オブジェクトの[Inspector] > [Mesh Renderer]には、[Scale In Lightmap]と呼ばれるパラメータが含まれています。[Scale In Lightmap]を調整して、このオブジェクトがライトマップで使用するテクセルの数を変更できます。

次のスクリーンショットの左側のオブジェクトは、[Lightmap Resolution]が 5 に設定されているため、ベーキングユニットごとに 5 つのライティング情報のテクセルを取得する平均的なオブジェクトです。右側は、[Scale In Lightmap]が 0.5 に設定されているボックスです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

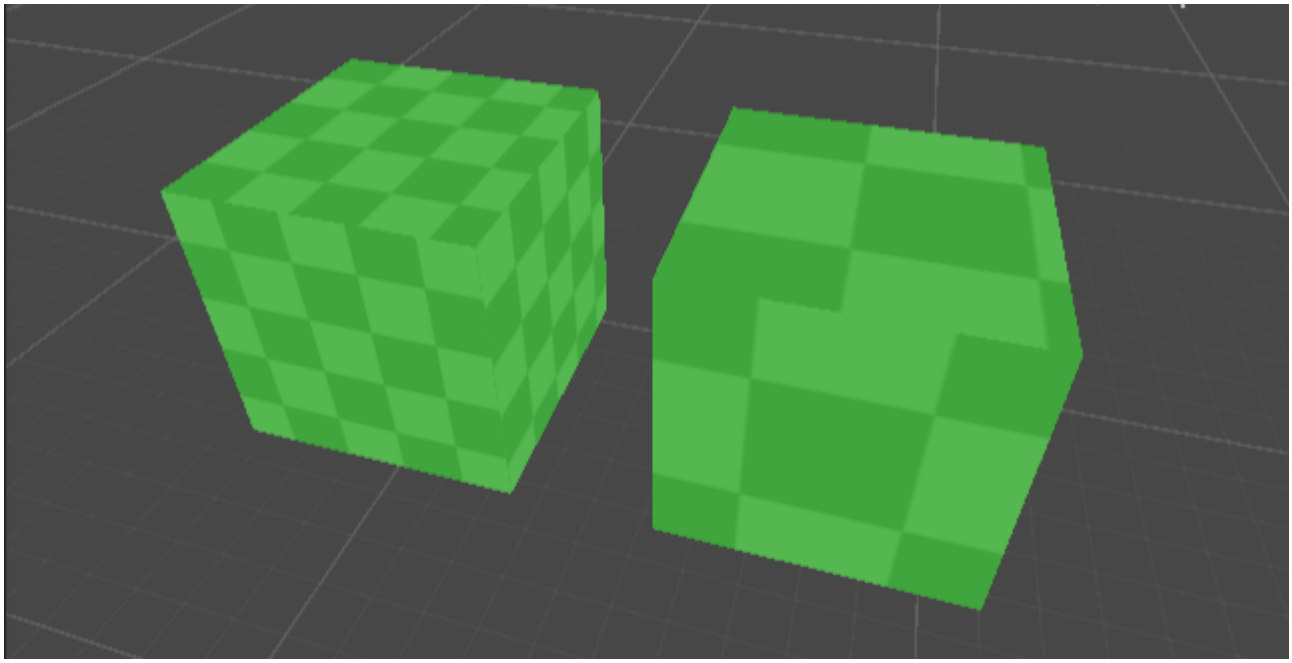


図 8–8 ライトマップのサンプル

右側のボックスがライトマップで使用するスペースは、左側のボックスよりもはるかに少なくなります。次のスクリーンショットでは、ライトマップの設定を確認できます。

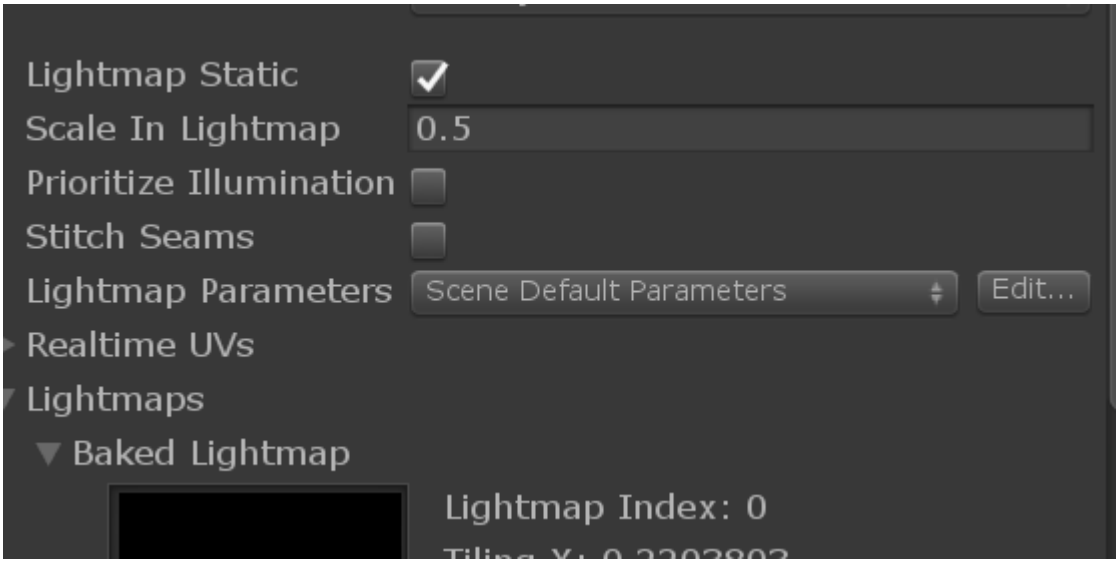


図 8–9 ライトマップの設定のサンプル

次の要素ではテクセルを使わないようにしてください。

- プレイヤーには見えないサーフェスとオブジェクト。画面に表示されないディテールに対して、大きなライトマップでメモリを消費するのを回避できます。
- ライトの変化がほとんどないサーフェス。たとえば、影の中になるオブジェクトや 1 つのライトだけが触れるオブジェクトなどです。
- 小さなオブジェクトや薄いオブジェクト。小さなオブジェクトや薄いオブジェクトが受けるライティングの量は、シーンの最終的なレンダリングにはあまり影響しません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.7 可能な限りフェイクする

リアルな影は計算コストが高くなります。動的なライトに頼らずに、フェイクシャドウを実装してダイナミックオブジェクトに影を付けることをお勧めします。

リアルタイムシャドウは、多くの場合、シャドウマップと呼ばれる手法を使用して生成されます。シャドウマップにレンダリングされるシーンのジオメトリのコストは、シャドウマップに描画される頂点の数に匹敵します。そのため、影をキャストするジオメトリと、リアルタイムに影をキャストするライトの数を制限することが重要です。

以下は、フェイクシャドウの実装方法の一例です。

- キャラクタの下に 3D メッシュ、平面、またはクワッドを配置して、ブラーテクスチャを適用します。
- Unity の組み込みレンダーパイプライン機能を使用し、*プロジェクタ*を使用して動的なブロブシャドウを適用します。この方法は、キャラクタの下にクワッドを使用するよりもコストがかかり、モバイルに推奨されるユニバーサル・レンダー・パイプラインでは使用できません。
- カスタムシェーダを作成して、より洗練されたブロブシャドウを作成します。

以下の Armies のサンプルデモのスクリーンショットでは、シャドウメッシュを使用したシャドウの実装が示されています。

図 8-10 シャドウの実装例

ライティング情報をテクスチャに直接ペイントします。ライトシェーディングの一部をテクスチャにペイントすることで、余分なライトが必要とする余分な計算を減らします。テクスチャに直接ペイントすると、シーンで必要なテクスチャメモリが少なくなるため、シーンのライトをベイクする際のメモリも節約されます。

シェーダまたはマテリアルを使用して、ライティングをシミュレートします。カスタムマテリアルを使用して、ライトエフェクトをシミュレートできます。たとえば、ゲームレベルでは、可視性と見た目を向上させるために、キャラクタにリムライティングを持たせたい場合があります。ライトを使用してこのエフェクトを作成する代わりに、シェーダエフェクトを使用してライトが当たっているように見せます。

シェーダには、ゲームに追加できる便利なエフェクトが数多くあります。詳細については、*シェーダとマテリアルのベストプラクティス・ガイド*を参照してください。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.8 ライトプローブ

ライトプローブの主な用途は 2 つあります。1 つ目は、シーン内の動くオブジェクトに高品質のライティング(間接反射光を含む)を提供することです。

2 つ目は、静的なシーンで Unity の*詳細レベル*(LoD)システムを使用している場合に、そのシーンのライティング情報を提供することです。

ベイクされたライティングで動的オブジェクトを使用する場合、ライトプローブは通常、ライトマップの影響を受けません。これにより、奇妙に映ったり、シーンに溶け込んでいないように見えたりすることがあります。

これを解決するのがライトプローブです。ライトプローブは、オフラインで計算できるライティングデータを保存するという点で、ライトマップと同じ利点を多く持っています。繰り返しになりますが、これにより計算コストのほとんどはランタイムではなく編集時に移されます。

ライトマップが、シーン内の表面の特定のテクセルで受けたライティングをコード化するのに対し、ライトプローブは、シーン内の空のスペースを通過する光を保存します。すると、このデータを使用して動的オブジェクトを照らすことができます。これは、シーン全体を通じてライトマップされたオブジェクトとの視覚的な差をなくす上で役立ちます。

ライトプローブは、静的シーンの光と影のみを保存して表示します。これは、ライトプローブもプリベイクされているためです。動的オブジェクト、リアルタイムライト、またはセルフシャドウイングからの影と光の解決策ではありませんが、ライトプローブは、シーンのほとんどのライティングを提供できます。

ライトプローブの主な用途は 2 つあります。

- シーン内の動くオブジェクトを照らすこと。ライトプローブは、ベイクされた光を利用するため、オブジェクトはシーンと同じライティングになります。ライトプローブを使用した動的オブジェクトのライティングは、リアルタイムライトを使用するよりもコストがかかりません。
- LoD システムを使用する静的にマークされたオブジェクトにライティングデータを提供すること。

ライトプローブの詳細については、*関連情報*を参照してください。

次のスクリーンショットは、ライトプローブの使用例を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

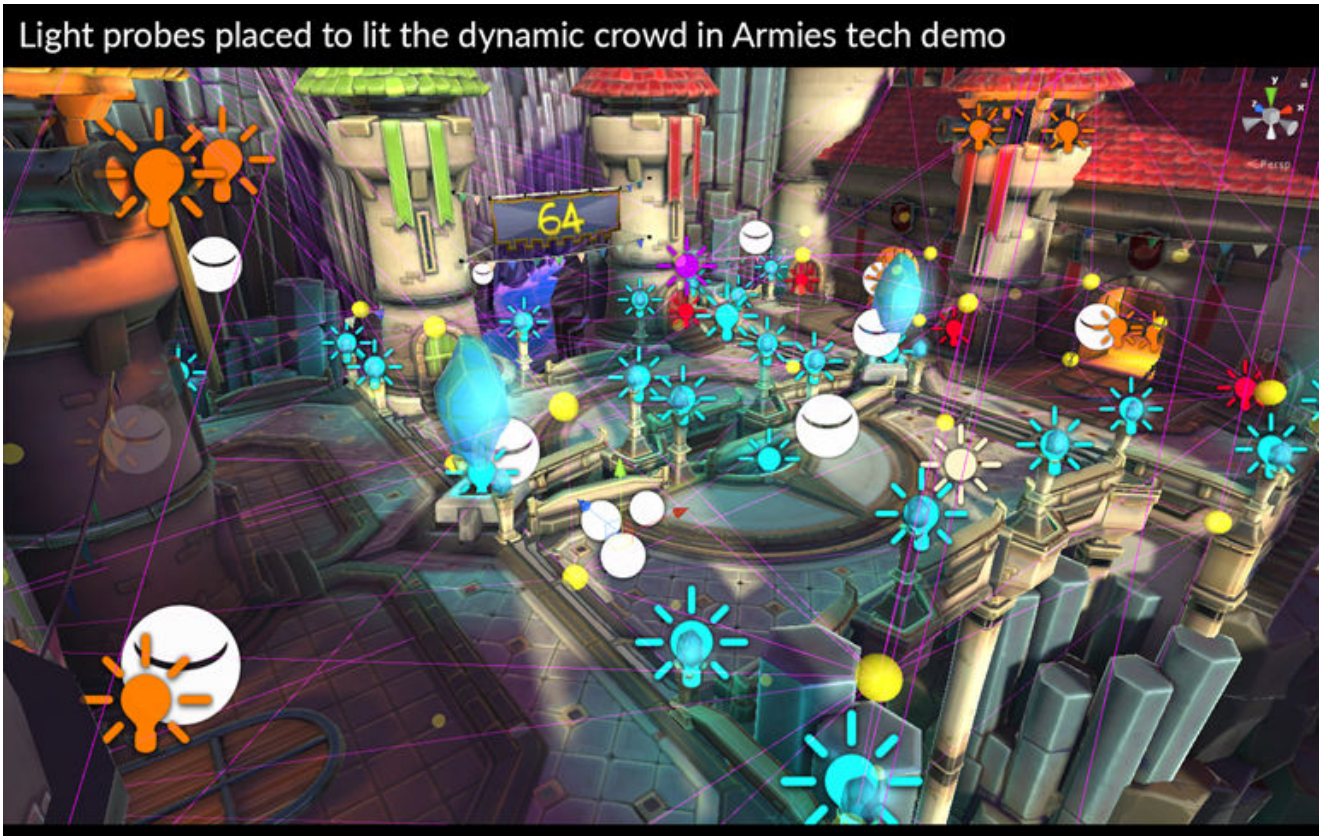


図 8-11 ライトプローブのサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.9 メッシュレンダラーの設定

シーンで使用するライティングに関係なく、メッシュレンダラーの設定が正しいことを確認することが重要です。

使用しないものはすべてオフにすることをお勧めします。[Cast Shadows]のような設定は、オブジェクトに光が当たっていなくても、シーンのレンダリングに余分なコストがかかる可能性があります。以下のスクリーンショットは、シーンの[Mesh Renderer]設定のサンプルを示しています。

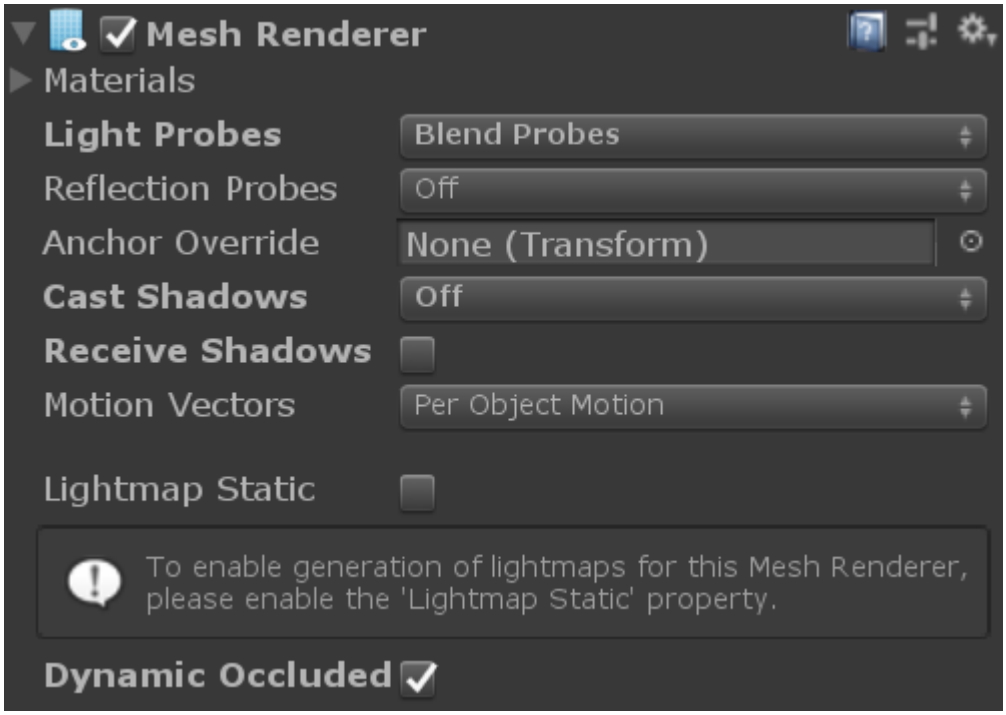


図 8-12 [Mesh Renderer]設定のサンプル

オプションは、最も近いプローブからのライティング情報がプローブ間を移動するときにスムーズにブレンドされることで、反射ではなくライティングを取得するように設定されています。プロブ解析を使用しているため、[Cast Shadows]はオフになっています。シーンがベイクされ、リアルタイムシャドウをキャストしているものがないため、[Receive Shadows]もオフになっています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

8.10 リアルタイムライトとライトのタイプ

ベイクされたライティング、ライトプローブ、およびマテリアルのエフェクトを使用して、すべてのライティングに対応するようにしてください。リアルタイムライトの使用が必要になる場合があります。その場合、使用するリアルタイムライトのタイプを検討する必要があります。

リアルタイムライトのタイプごとに、計算するコストが異なります。

指向性

指向性ライトは、方向が均一で減衰することがないため、実現できるものの中で最もコストのかからないリアルタイムライティングです。指向性ライトはシーン全体を照らすことができるため、通常、必要なのは 1 つだけです。つまり、フォワードレンダリングでは、Unity は常に一方向のライトをレンダリングします。これは、シーンに指向性ライトがない場合でも当てはまります。

ポイント

ポイントライトは、空間のあるポイントに位置し、全方向に均等に光を放ちます。

スポット

スポットライトは、球形のポイントライトよりも多くのオブジェクトをカリングするため、二番目にコストのかからないリアルタイムライティングです。このライトから最大限のパフォーマンスを得るには、円錐の幅を狭くし、選択したオブジェクトのみに当たるようにします。

全方向のライティングは便利ですが、コストも非常に高くなります。指向性ライトは、場所に関係なく、計算コストは比較的低くなります。スポットライトは、狭い領域に対してのみコストが高くなるように限定できます。ポイントライトは広い領域ではコストが高くなります。また、影の計算はライティングで最もコストのかかる部分でもあるため、全方向にライトを投影するとコストが高くなります。

動的なライトはレンダリングにコストがかかるため、モバイルゲームでは避けるのがベストです。使用するデバイスとグラフィックス API によっては、使用に制限がある場合があります。たとえば、OpenGL ES 2.0 を使用する Unity のユニバーサル・レンダー・パイプラインのフォワードレンダラーでは、1 つのオブジェクトに対してライトは 4 つに制限されています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 9 章
高度なグラフィックス手法

この章では、使用可能ないくつかの高度なグラフィックス手法を説明します。

以下のセクションから構成されています。

- *9.1 カスタム シェーダ*(9-144 ページ).
- *9.2 ローカル キューブマップによる反射の実装*(9-157 ページ).
- *9.3 反射の結合*(9-174 ページ).
- *9.4 ローカル キューブマップに基づく動的なソフト シャドウ*(9-180 ページ).
- *9.5 ローカル キューブマップに基づく屈折*(9-188 ページ).
- *9.6 Ice Cave デモの鏡面反射エフェクト*(9-194 ページ).
- *9.7 Early-Z を使用する*(9-197 ページ).
- *9.8 ダーティ レンズ エフェクト*(9-198 ページ).
- *9.9 ライト シャフト*(9-201 ページ).
- *9.10 フォグ エフェクト*(9-205 ページ).
- *9.11 ブルーム*(9-212 ページ).
- *9.12 氷壁のエフェクト*(9-219 ページ).
- *9.13 手続き型スカイボックス*(9-225 ページ).
- *9.14 ホタル*(9-233 ページ).
- *9.15 接空間からワールド空間への法線変換ツール*(9-237 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.1 カスタム シェーダ

このセクションでは、カスタム シェーダについて説明します。

以下のサブセクションから構成されています。

- *9.1.1 カスタム シェーダについて*(9-144 ページ).
- *9.1.2 シェーダ構造体*(9-145 ページ).
- *9.1.3 コンパイル ディレクティブ*(9-147 ページ).
- *9.1.4 インクルード*(9-147 ページ).
- *9.1.5 OpenGL ES 3.0 グラフィックス パイプライン*(9-148 ページ).
- *9.1.6 頂点シェーダ*(9-149 ページ).
- *9.1.7 頂点シェーダの入力*(9-150 ページ).
- *9.1.8 頂点シェーダの出力と可変変数*(9-150 ページ).
- *9.1.9 フラグメント シェーダ*(9-152 ページ).
- *9.1.10 シェーダにデータを提供する*(9-152 ページ).
- *9.1.11 Unity でのシェーダのデバッグ*(9-154 ページ).

9.1.1 カスタム シェーダについて

Unity 5 以降では、マテリアルと光の相互作用をシミュレートする PBS (*Physically Based Shading*) モデルが含まれています。これは高い臨場感を提供し、異なるライティング条件下でも一貫した外観を実現できます。

PBS は、標準のシェーダで簡単に使用できます。独自のマテリアルを作成すると、自動的に標準シェーダが割り当てられます。

標準シェーダには簡単にアクセスできます。独自のマテリアルを作成すると、標準シェーダが割り当てられます。初心者には非常に便利な他の組み込みシェーダがいくつかあります。[Inspector] の [Shader] ドロップダウン メニューをクリックすると、使用可能なすべての組み込みシェーダが、ファミリに分割されて表示されます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

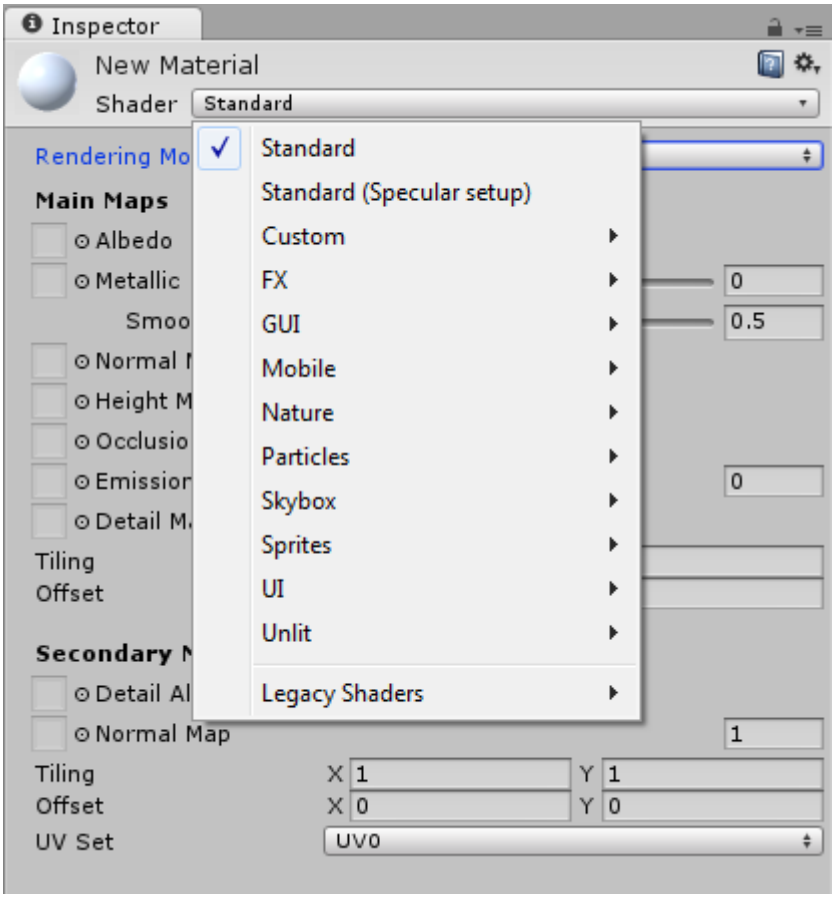


図 9-1 Unity 組み込みのシェーダ

組み込みシェーダのソースコードは、Unity ダウンロード アーカイブ *http://unity3d.com/* で入手できます。これには 120 以上のシェーダが含まれています。これらのシェーダのコードの読解により、多くのことを学べます。

これらに加え、既存のシェーダの使用では実現できない多くのエフェクトがあります。たとえば、ローカル キューブマップに基づいて反射を実装するシェーダがあります。詳細については、*9.2 ローカル キューブマップによる反射の実装*(9-157 ページ)を参照して下さい。

Unity では通常、シェーダを作成する 2 つの方法があります。

サーフェス シェーダ
: これらはシェーダが光と影の影響を受ける場合によく使用されます。Unity はライティング モデルに関連する処理を行い、よりコンパクトなシェーダを作成できます。

頂点シェーダとフラグメント シェーダ
: これらは最も柔軟なシェーダですが、すべてを実装する必要があります。Unity ShaderLab は頂点シェーダとフラグメント シェーダ以上の機能を備えていますが、それらはすべてのシェーディングが行われるグラフィックス パイプラインのメインのプログラム可能な部分にあります。そのため、カスタムの頂点シェーダとフラグメント シェーダの作成方法を理解することが重要です。

9.1.2 シェーダ構造体

次のコードは、頂点シェーダやフラグメント シェーダで必要なほとんどの要素を含む、非常に単純な頂点シェーダとフラグメント シェーダを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

このシェーダのサンプルは、Cg で書かれています。Unity は、シェーダ スニペットに HLSL 言語もサポートしています。

```
Shader "Custom/ctTextured"
{
    Properties
    {
        _AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma target 3.0
            #pragma glsl
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            // ユーザ指定のユニフォーム
            uniform float4 _AmbientColor;
            uniform sampler2D _MainTex;

            struct vertexInput
            {
                float4 vertex : POSITION;
                float4 texCoord : TEXCOORD0;
            };
            struct vertexOutput
            {
                float4 pos : SV_POSITION;
                float4 tex : TEXCOORD0;
            };

            // 頂点シェーダ。
            vertexOutput vert(vertexInput input)
            {
                vertexOutput output;

                output.tex = input.texCoord;
                output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                return output;
            }

            // フラグメント シェーダ。
            float4 frag(vertexOutput input) : COLOR
            {
                float4 texColor = tex2D(_MainTex, float2(input.tex));
                return _AmbientColor + texColor;
            }

            ENDCG
        }
    }
    Fallback "Diffuse"
}
```

最初のキーワードは `Shader` で、その後にシェーダの *パス/名前* が続きます。パスは、マテリアルを設定するときに、ドロップダウン メニューにシェーダが表示されるカテゴリを定義します。このサンプルのシェーダは、ドロップダウン メニューの Custom シェーダのカテゴリの下に表示されます。

`Properties{}` ブロックには、インスペクタに表示されるシェーダ パラメータ、および対話できるパラメータがリストされています。

Unity の各シェーダは、サブシェーダのリストで構成されています。Unity がメッシュをレンダリングすると、使用するシェーダが検索され、グラフィックス カードで実行可能な最初のサブシェーダ が選択されます。これによって、シェーダは異なるシェーダモデルをサポートする異なるグラフィックス カード上で、正しく実行されます。GPU のハードウェアと API が常に進化しているため、この機能は重要です。たとえば、メインのシェーダを Mali Midgard GPU をターゲットにして作成し、OpenGL ES 3.0 の最新機能を使用することができます。別のサブシェーダでは、OpenGL ES 2.0 以前をサポートするグラフィックス カードの代替シェーダを作成できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

Pass ブロックでは、オブジェクトのジオメトリを 1 回レンダリングします。シェーダには 1 つまたは複数の Pass を含めることができます。古いハードウェアで複数の Pass を使用することも、特殊なエフェクトを得ることもできます。

Unity がシェーダの本体内で、ジオメトリを正しくレンダリングできるサブシェーダを見つけられない場合は、Fallback ステートメントの後に定義された別のシェーダにロールバックします。このサンプルでは、これは Diffuse 組み込みシェーダです。

Cg プログラム スニペットは CGPROGRAM と ENDCG の間に書かれています。

9.1.3 コンパイル ディレクティブ

コンパイル ディレクティブを #pragma ステートメントとして渡します。コンパイル ディレクティブは、コンパイルされるシェーダ関数を示します。

各コンパイル ディレクティブには、少なくとも頂点シェーダとフラグメント シェーダをコンパイルするためのディレクティブ #pragma vertex name, #pragma fragment name が含まれている必要があります。

デフォルトでは、Unity はシェーダをシェーダ モデル 2.0 にコンパイルします。#pragma target ディレクティブにより、シェーダを他の機能レベルにコンパイルすることができます。シェーダが大きくなると、次のタイプのエラーが発生します。

```
Shader error in 'Custom/MyShader': Arithmetic instruction limit of 64 exceeded; 83
arithmetic instructions needed to compile program;
```

この場合には、#pragma target 3.0 ステートメントを追加して、シェーダ モデル 2.0 からシェーダ モデル 3.0 に変更する必要があります。シェーダ モデル 3.0 では命令の制限がより高くなっています。

頂点シェーダからフラグメント シェーダにいくつかの可変変数を渡すと、次のエラーが発生する可能性があります。

```
Shader error in 'Custom/MyShader': Too many interpolators used (maybe you want
#pragma glsl?) at line 75.
```

このような場合は、コンパイル ディレクティブ #pragma glsl を追加します。このディレクティブは、Cg または HLSL コードを GLSL に変換します。

#pragma only_renderers ディレクティブ。

Unity は、gles、gles3、opengl、d3d11、d3d11_9x、xbox360、ps3、flash などの、いくつかのレンダリング プラットフォームをサポートしています。デフォルトでは、シェーダはすべてのプラットフォームにコンパイルされます。そうしない場合には、明示的に #pragma only_renderers を使い、必要なレンダリング API を空白で区切って指定して、数を制限します。

モバイル デバイスをターゲットにしている場合は、シェーダのコンパイルを gles と gles3 に制限します。Unity エディタで使用される opengl と d3d11 レンダラーも追加する必要があります。

```
#pragma only_renderers gles gles3 [opengl, d3d11]
```

9.1.4 インクルード

シェーダにインクルード ファイルを追加すると、Unity の定義済みの変数とヘルパ関数を利用することができます。

利用できるインクルードは、C:\Program Files \Unity\Editor\Data\CGIncludes にあります。例えば、インクルード UnityCG.cginc には、役立つヘルパ関数と多くの標準シェーダで使用されるマクロがいくつかあります。これを使用するには、インクルードをシェーダ内で宣言します。

シェーダでは、いくつかの Unity 組み込み変数を利用できます。それらは UnityShaderVariables.cginc インクルードにあります。このファイルはシェーダに含める必要はありません。Unity が自動的に行います。いくつかの有用な変換行列と magnitude をシェーダで直接利用できます。作業の重複を避けるために、これらのすべてを知ることが重要です。たとえば、シェーダ、カメラの位置、投影パラメータ、光パラメータなどに行列を渡す方法を検討する前に、インクルードがこれを提供しているかどうかを確認します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

パフォーマンスを向上させるには、すべての頂点で頂点シェーダで実行するよりも、CPU で操作を実行してその結果を GPU に渡す方が望ましい場合があります。たとえば、これは行列のユニフォームの乗算の場合です。このため、Unity はいくつかの複合行列を、組み込みのユニフォームとして利用できるようにしています。重要な Unity シェーダの組み込み値の一部を次の表に示します。

表 9-1 重要な Unity シェーダの組み込み値

組み込みユニフォーム	説明
UNITY_MATRIX_V	現在のビュー行列
UNITY_MATRIX_P	現在の投影行列
Object2World	現在のモデル行列
_World2Object	現在のワールド行列の逆行列
UNITY_MATRIX_VP	現在のビュー * 投影行列
UNITY_MATRIX_MV	現在のモデル * ビュー行列
UNITY_MATRIX_MVP	現在のモデル * ビュー * 投影行列
UNITY_MATRIX_IT_MV	現在のモデルの逆行列の転置 * ビュー行列
_WorldSpaceCameraPos	ワールド空間でのカメラ位置
_ProjectionParams	近平面と遠平面とベクトルのコンポーネントとしての、1/farPlane
_Time	現在の時間とベクトルの分数 (t/20, t, t*2, t*3)

9.1.5 OpenGL ES 3.0 グラフィックス パイプライン

プログラム可能な頂点シェーダとフラグメント シェーダが、グラフィックス パイプラインのどこにあるかを知ることは重要です。

次の図は、OpenGL ES 3.0 グラフィックス パイプライン フローの概要を示しています。OpenGL ES 3.0 は、エンベデッド グラフィックスの進化の主要なステップであり、OpenGL 3.3 仕様に基づいています。

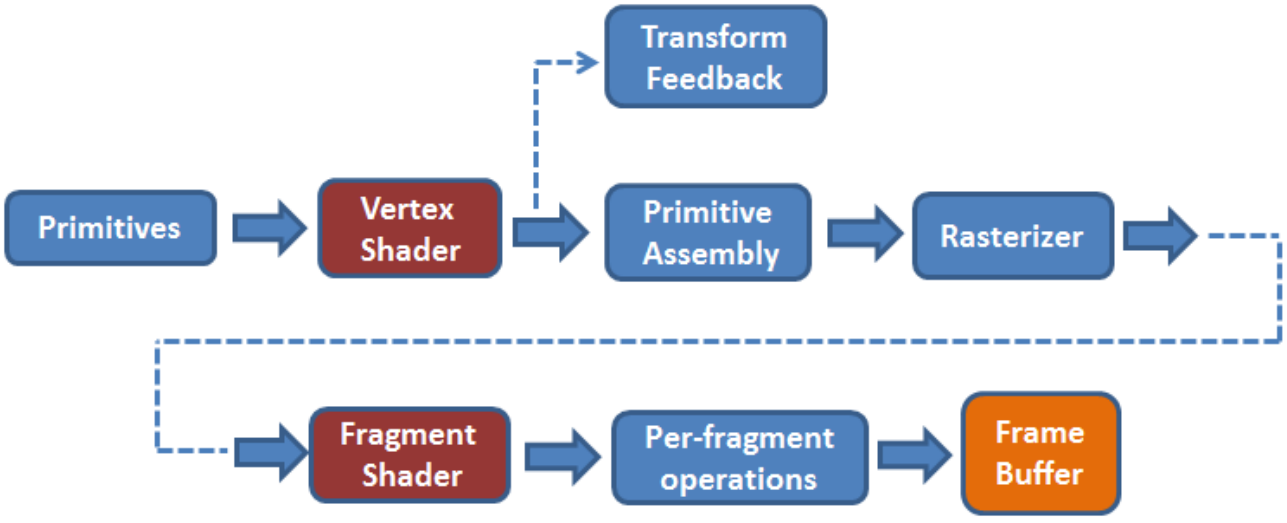


図 9-2 OpenGL ES 3.0 プログラマブル パイプライン

プリミティブ

プリミティブ ステージでは、パイプラインは、頂点、点、線、ポリゴンによって記述されたジオメトリ プリミティブ上で動作します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

頂点シェーダ
頂点シェーダは、頂点を操作する、汎用のプログラム可能なメソッドを実装しています。頂点シェーダは、頂点の変換と頂点の照射を行います。

プリミティブ アセンブリ
プリミティブ アセンブリでは、頂点はジオメトリ プリミティブにアセンブルされます。結果のプリミティブは、クリッピング ボリュームにクリップされ、ラスタライザに送信されます。

ラスタライズ
頂点シェーダからの出力値は、生成されたフラグメントごとに計算されます。このプロセスは補間として知られています。ラスタライズ中に、プリミティブは 2 次元のフラグメントのセットに変換され、フラグメント シェーダに送信されます。

変換フィードバック
変換フィードバックは、頂点シェーダが出力し、その後頂点シェーダに戻される、出力バッファへの選択的書き込みを可能にします。この機能は Unity によって公開されていませんが、内部的にはたとえばキャラクタのスキニングを最適化するために使用されています。

フラグメント シェーダ
フラグメント シェーダは、フラグメントを次のステージに送る前に操作するための、汎用のプログラム可能なメソッドを実装します。

フラグメント単位の操作
フラグメント単位の操作では、各フラグメントにいくつかの機能とテストが適用されます。ピクセル オーナーシップ テスト、シザー テスト、ステンシルとデプス テスト、ブレンディングとディザリングです。このフラグメント単位の段階の結果として、フラグメントは破棄されるか、フラグメント カラー、デプスまたはステンシル値がスクリーン座標でフレームバッファに書き込まれます。

9.1.6 頂点シェーダ

頂点シェーダのサンプルは、ジオメトリの頂点ごとに 1 回実行されます。頂点シェーダの目的は、オブジェクトのローカル座標で与えられた各頂点の 3D 位置を、スクリーン空間の投影された 2D 位置に変換し、Z バッファのデプス値を計算することです。

頂点シェーダのサンプル コードは *9.1.2 シェーダ構造体*(9-145 ページ) にあります。

頂点シェーダの出力は通常は、変換された位置となります。頂点シェーダが値を返さない場合、コンソールは次のエラーを表示します。

```
Shader error in 'Custom/ctTextured': '' : function does not return a value: vert at
line 36
```

このサンプルでは、頂点シェーダは、ローカル空間の頂点座標とテクスチャ座標を、入力として受け取ります。頂点座標は、Unity のビルトイン値である Model View Projection 行列の `UNITY_MATRIX_MVP` を使用して、ローカル空間からスクリーン空間に変換されます。

```
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
```

テクスチャ座標は可変変数としてフラグメント シェーダに渡されますが、変換されていないわけではありません。

法線は、別の方法でオブジェクト空間からワールド空間に変換されます。不均等なスケーリング操作の後でも、法線が三角形に対して依然として垂直であることを保証するためには、変換行列の逆行列の転置を乗じる必要があります。転置演算を適用するには、乗算の要素の順序を反転します。ローカルからワールド行列への逆行列は、ビルトインの `World2Object` Unity 行列です。これは 4×4 行列であるため、3 成分の法線入力ベクトルから 4 成分のベクトルを構築する必要があります。

```
float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);
```

4 つのコンポーネント ベクトルを構築するときは、第 4 のコンポーネントとして 0 を追加します。これは 4 次元空間においてベクトル変換を正しく処理するために必要であり、座標については第 4 成分が 1 つのユニットでなければなりません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

法線が既にワールド座標で提供されている場合は、法線の変換処理をスキップできます。これにより、頂点シェーダの作業が節約できます。オブジェクト メッシュが Unity のビルトイン シェーダによって処理される可能性がある場合は、このヒントを避けてください。その場合は通常は、法線がオブジェクト座標によるためです。

ほとんどのグラフィックス エフェクトはフラグメント シェーダで実装されていますが、頂点シェーダでいくつかのエフェクトを実行することもできます。頂点 Displacement マッピングは Displacement マッピングとも呼ばれ、テクスチャを使用して表面のディテールを追加して、ポリゴン メッシュを変形することを可能にする、よく知られている手法です。たとえば、高さマップを使用した地形の生成などを行えます。Displacement マップとも呼ばれるこのテクスチャに頂点シェーダでアクセスするには、プラグマ ディレクティブ `#pragma target 3.0` を追加する必要があります。これは、シェーダ モデル 3.0 のみで使用できるためです。シェーダ モデル 3.0 によれば、少なくとも 4 つのテクスチャ ユニットが頂点シェーダ内でアクセス可能でなければなりません。エディタで OpenGL レンダラーを強制的に使用する場合は、`#pragma glsl` ディレクティブも追加する必要があります。このディレクティブを宣言していない場合、生成されたエラーメッセージには次のような意味があります。

```
Shader error in 'Custom/ctTextured': function "tex2D" not supported in this profile
(maybe you want #pragma glsl?) at line 57
```

頂点シェーダでは、「手続きアニメーション」の手法を使用して、頂点をアニメーション化することもできます。シェーダで時間変数を使用すると、時間の関数として頂点座標を変更できます。メッシュ スキニングは、頂点シェーダで実装されている、もう 1 つの機能です。Unity はこれを使用して、キャラクタ スケルトンに関連付けられたメッシュの頂点をアニメーション化します。

9.1.7 頂点シェーダの入力

頂点シェーダの入力と出力は、構造体によって定義されます。このサンプルの入力構造体では、頂点属性の位置とテクスチャ座標だけを宣言します。

さらに多くの属性を入力として定義することができます。たとえば、次のセマンティクスを使用して、テクスチャ座標の 2 番目のセット、オブジェクト座標の法線、カラー、接線などを定義できます。

```
struct vertexInput
{
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    fixed4 color : COLOR;
};
```

セマンティックは、シェーダの入力または出力に付加された文字列で、パラメータの使用に関する情報を提供します。シェーダ ステージ間で渡されるすべての変数にセマンティックを指定する必要があります。

`float3 tangent2 : TANGENTIAL` などの正しくないセマンティクスを使用すると、次のタイプのエラーが発生します。

```
Shader error in 'Custom/ctTextured': unknown semantics "TANGENTIAL" specified for
"tangent2" at line 32
```

パフォーマンスのため、入力構造内で厳密に必須なパラメータのみを指定します。Unity には、最も一般的な入力パラメータの組み合わせである、`appdata_base`、`appdata_tan`、および `appdata_full` のための、いくつかの事前定義の入力構造体があります。これらは `UnityCG.cginc` インクルード ファイルに記述されています。先述の頂点の入力構造体のサンプルは、`appdata_full` に対応します。この場合、構造体を宣言する必要はなく、インクルード ファイルの宣言のみを行います。

9.1.8 頂点シェーダの出力と可変変数

頂点シェーダの出力は、出力構造体で定義され、これには必ず頂点変換座標が含まれます。次のサンプルでは、出力構造体は非常に単純ですが、他の magnitude を追加することもできます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

次のコードは、Unity がサポートするセマンティクスの一覧を示しています。

```
struct vertexOutput
{
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
    float4 texSpecular : TEXCOORD1;
    float3 vertexInWorld : TEXCOORD2;
    float3 viewDirInWorld : TEXCOORD3;
    float3 normalInWorld : TEXCOORD4;
    float3 vertexToLightInWorld : TEXCOORD5;
    float4 vertexInScreenCoords :  TEXCOORD6;
    float4 shadowsVertexInScreenCoords :  TEXCOORD7;
};
```

変換された頂点座標はセマンティック SV_POSITION で定義されます。セマンティック TEXCOORDn を呼び出す異なる空間の 2 つのテクスチャ、いくつかのベクトル、座標も、フラグメント シェーダに渡されます。

TEXCOORD0 は通常は UV のために予約されていて、TEXCOORD1 は通常はライトマップ UV のために予約されていますが、技術的には TEXCOORD0 から TEXCOORD7 までのいずれも、フラグメント シェーダに送ることができます。各インターポレータはセマンティックですが、最大 4 つの浮動小数点しか処理できないことに注意することが重要です。行列などのより大きな変数を複数のインターポレータに入れることができます。これは、float4x4 myMatrix : TEXCOORD2 という可変変数として渡される行列を定義した場合、Unity は TEXCOORD2 から TEXCOORD5 までのインターポレータを使用することを意味します。

デフォルトでは、頂点シェーダからフラグメント シェーダに送信されるものはすべて、線形補間されます。頂点 V1、V2、V3 で定義される三角形の各ピクセルについて、頂点とフラグメント シェーダの間のグラフィックス パイプラインに位置するラスタライザは、重心座標 λ1、λ2、λ3 を使用して頂点座標の線形補間としてピクセル座標を計算します。

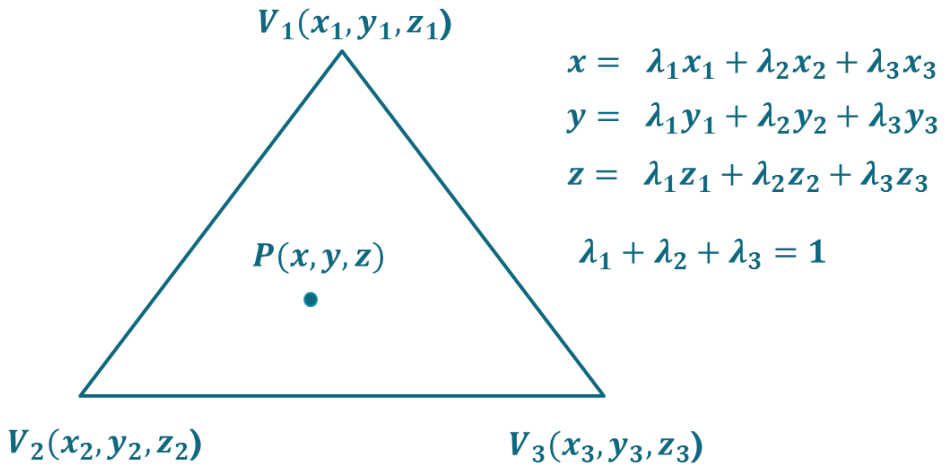


図 9-3 重心座標を使った線形補間

次の図は、頂点のカラーが赤、緑、青の三角形内の色補間の結果を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

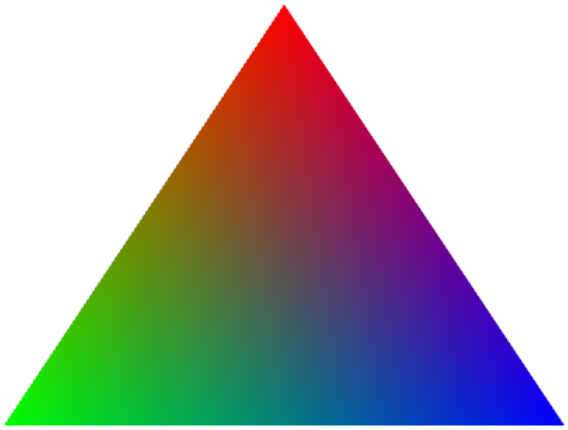


図 9-4 カラー補間

同じ補間が、頂点からフラグメント シェーダに渡された、任意の可変値に適用されます。これは、ハードウェア線形インターポレータがあるため、非常に強力なツールです。たとえば、ある平面で、中心 C への距離の関数としてカラーを適用する場合は、中心 C の座標を頂点シェーダに渡し、頂点から C までの二乗距離を計算し、その magnitude をフラグメント シェーダに渡します。距離の値は、すべての三角形の各ピクセルで自動的に補間されます。

値は線形補間されるので、頂点ごとの計算を実行して、それらをフラグメント シェーダで再利用することができます。つまり、フラグメント シェーダで線形補間できる値を、頂点シェーダで計算することができます。頂点シェーダはフラグメント シェーダよりもはるかに小さいデータセットで動作するため、これによって大幅なパフォーマンスの向上をもたらす場合があります。

多くのゲームの成功にとってパフォーマンスやメモリ帯域幅の消費が重要なモバイルでは、特に可変変数の使用に注意する必要があります。より多くの可変変数が存在するほど、より多くの頂点アクセスが発生し、フラグメント シェーダの帯域幅の読み取りが発生します。可変変数を使用する場合には、妥当なバランスを目指すようにします。

9.1.9 フラグメント シェーダ

フラグメント シェーダは、プリミティブ ラスタライズ後のグラフィックス パイプライン ステージです。

プリミティブによってカバーされる画素の各サンプルについて、フラグメントが生成されます。フラグメント シェーダのコードは、生成されたフラグメントごとに実行されます。頂点よりも多くのフラグメントがあるので、フラグメント シェーダで実行される操作の数に注意する必要があります。

フラグメント シェーダでは、ウィンドウ空間内のフラグメント座標にアクセスできます。これには頂点シェーダからのすべての補間された頂点出力値が含まれます。

9.1.2 シェーダ構造体(9-145 ページ) のシェーダのサンプルでは、フラグメント シェーダが頂点シェーダから補間されたテクスチャ座標を受け取り、テクスチャ ルックアップを実行してこれらの座標のカラーを取得します。このカラーをアンビエント カラーと組み合わせて、最終的な出力カラーを生成します。フラグメント シェーダの宣言 `float4 frag(vertexOutput input) : COLOR` では、明らかにフラグメント カラーの生成が予想されます。フラグメントシェーダでは、必要な効果を得るための操作が行われます。つまりこれは、フラグメントに正しいカラーを割り当てることを意味します。

9.1.10 シェーダにデータを提供する

Pass ブロックでユニフォームとして宣言されたデータは、頂点シェーダとフラグメント シェーダの両方で使用できます。

ユニフォームは、シェーダ内で変更することができないため、グローバル定数変数の一種と考えることができます。

次の方法で、このユニフォームをシェーダに提供することができます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

- Properties ブロックを使用する。
- スクリプトからプログラムで行う。

Properties ブロックでは、Inspector でユニフォームをインタラクティブに定義できます。Properties ブロックで宣言された変数は、マテリアル インスペクタに変数名と共にリストされて表示されます。

次のコードは、ctSphereMat マテリアルに関連付けられたシェーダのサンプルの Properties ブロックを示しています。

```
Properties
{
    _AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)
    _MainTex ("Base (RGB)", 2D) = "white" {}
}
```

Ambient Color と Base (RGB) という名前の Properties ブロックで宣言された変数 _AmbientColor と _MainTex はそれぞれ Material Inspector にそれらの名前で表示されます。

次の図は、Material Inspector のプロパティを示しています。

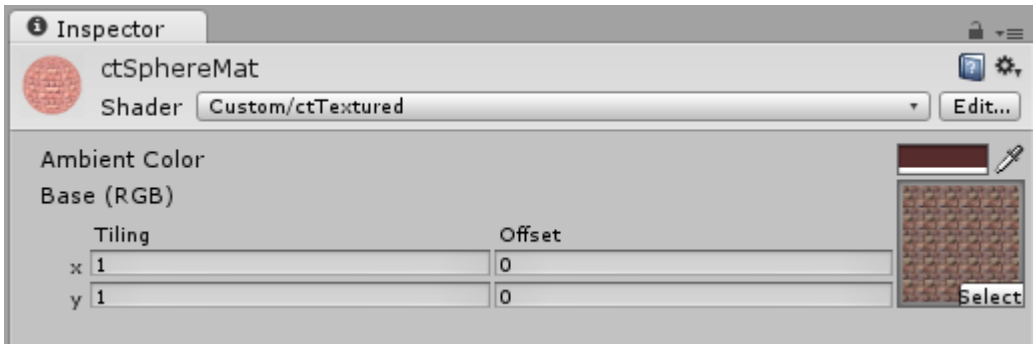


図 9-5 マテリアル インスペクタのプロパティ

Properties ブロックを使用してシェーダにデータを渡すことは、データをインタラクティブに変更して実行時の効果を確認できるため、シェーダの開発段階にあるときに特に便利です。

Properties ブロックには、次のタイプの変数を配置できます。

- 浮動小数点数。
- カラー。
- テクスチャ 2D。
- キューブマップ。
- 四角形。
- ベクトル。

Properties ブロックは、たとえば、以前の計算でデータが必要な場合や、特定の時点でデータを渡す必要がある場合などには、データを渡すのに便利な方法ではありません。

シェーダにデータを渡す別の方法は、スクリプトからプログラムで行う方法です。

マテリアル クラスは、マテリアルに関連付けられたデータをシェーダに渡すために使用できる、いくつかのメソッドを公開しています。次の表に、最もよく使われるメソッドのリストを示します。

表 9-2 マテリアルに関連付けられたデータをシェーダに渡すための一般的なメソッド

メソッド
SetColor (propertyName: string, color: Color);
SetFloat (propertyName: string, value: float);
SetInt (propertyName: string, value: int);
SetMatrix (propertyName: string, matrix: Matrix4x4);

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

表 9-2 マテリアルに関連付けられたデータをシェーダに渡すための一般的なメソッド（続き）

メソッド
`SetVector (propertyName: string, vector: Vector4);`
`SetTexture (propertyName: string, texture: Texture);`

次のコードでは、メイン カメラがシーンをレンダリングする直前に、セカンダリ カメラ `shwCam` が、メイン カメラのレンダリング パスと組み合わされるテクスチャに、影をレンダリングします。

影テクスチャの投影処理のため、都合のよい方法で頂点を変換する必要があります。シャドウ カメラ投影行列 (`shwCam.projectionMatrix`)、ワールドからローカルへの変換行列 (`shwCam.transform.worldToLocalMatrix`)、レンダリングされた影テクスチャ (`m_ShadowsTexture`) は、シェーダに渡されます。

これらの値はシェーダで `_ShwCamProjMat`、`_ShwCamViewMat`、`m_ShadowsTexture` という名前のユニフォームとして使用できます。

次のコードは、リスト `shwMats` に含まれるマテリアルを使用して、マトリックスとテクスチャをシェーダに送る方法を示しています。

```
// オブジェクトがレンダリングされる前に呼び出される。

public void OnWillRenderObject()
{
    // 別のチェックを実行する。
    ...
    CreateShadowsTexture();
    // シャドウ カメラ shwCam を設定する。
    ...
    // マトリックスをシェーダに渡す
    for(int i = 0; i < shwMats.Count; i++)
    {
        shwMats[i].SetMatrix("_ShwCamProjMat", shwCam.projectionMatrix);
        shwMats[i].SetMatrix("_ShwCamViewMat", shwCam.transform.worldToLocalMatrix);
    }
    // 影テクスチャをレンダリングする
    shwCam.Render();
    for(int i = 0; i < shwMats.Count; i++)
    {
        shwMats[i].SetTexture( "_ShadowsTex", m_ShadowsTexture );
    }
    s_RenderingShadows = false;
}
```

9.1.11 Unity でのシェーダのデバッグ

Unity では、従来のコードと同じような方法でシェーダをデバッグすることはできません。ただし、フラグメント シェーダの出力を使用して、デバッグする値を視覚化することができます。その後、作成されたイメージを解釈する必要があります。

次の図は、*9.2 ローカル キューブマップによる反射の実装*(9-157 ページ)のフロアの反射面に適用されたシェーダ `ctReflLocalCubemap.shader` の出力を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9-6 反射のあるチェス ルーム

次のフラグメント シェーダでは、出力カラーは、正規化された局所補正反射ベクトルに置換されました。

```
return float4(normalize(localCorrReflDirWS), 1.0);
```

反射画像の代わりに、反射ベクトルの成分をカラーとして正規化し、視覚化します。

床の赤みがかった部分は、反射ベクトルが強い X 成分を持つことを示しています。つまり、ほとんどが X 軸に向いています。赤みがかった部分は、その方向、つまり窓のある壁からの反射を示します。

青みがかった部分は、Z 軸に向けられた反射ベクトルが優位であること、つまり右側の壁からの反射を示します。

黒い部分では、ベクトルは主に -Z 方向に向いていますが、負の成分は 0 にクランプされるため、カラーは正の成分しか持てません。

次の図は、フラグメントの出力カラーを、正規化されたローカル反射ベクトルで置換した結果を示しています。

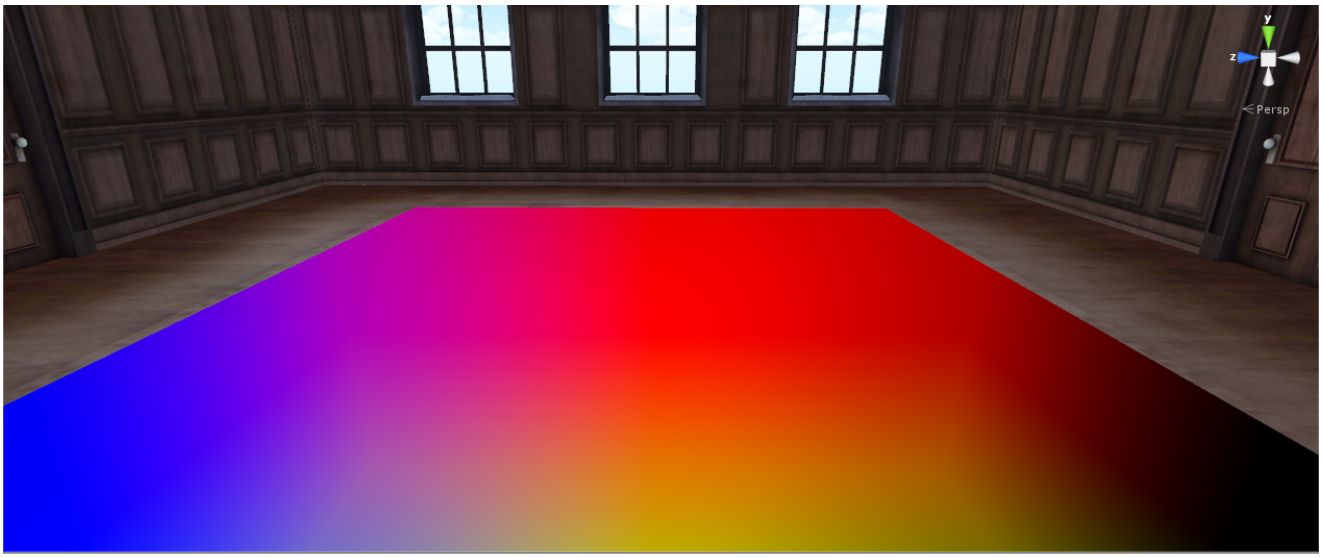


図 9-7 複数のカラーを使ったシェーダのデバッグ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

最初はデバッグ時にカラーの意味を解釈するのが難しいかもしれません。単一のカラーの成分に焦点を当ててみてください。たとえば、正規化された局所補正反射ベクトルの Y 成分のみを返すことができます。

```
float3 normLocalCorrReflDirWS = normalize(localCorrReflDirWS);
return float4(0, normLocalCorrReflDirWS.y, 0, 1);
```

この場合、出力は主にカメラの上方の屋根からの反射のみです。つまり、部屋の一部が Y 軸に向いています。部屋の壁からの反射は、X、Z、および −Z 方向から来ているため、黒でレンダリングされます。

次の図は、単一のカラーを使ったシェーダ デバッグを示しています。

図 9-8 単一のカラーを使ったシェーダのデバッグ

カラーを使ってデバッグしている magnitude が 0 と 1 の間であることを確認してください。他の値は自動的にクランプされます。負の値には 0 が割り当てられ、1 より大きい値には 1 が割り当てられます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.2 ローカル キューブマップによる反射の実装

ローカル キューブマップに基づく反射は、モバイル デバイス上で反射をレンダリングするのに便利な手法です。

Unity バージョン 5 以降は、ローカル キューブマップに基づく反射をリフレクション プローブとして実装しています。これらは、カスタム シェーダで実行時にレンダリングされる反射など、他の種類の反射と組み合わせることができます。

以下のサブセクションから構成されています。

- *9.2.1 反射の実装の歴史*(9-157 ページ).
- *9.2.2 ローカル キューブマップで正しい反射を生成する*(9-159 ページ).
- *9.2.3 シェーダの実装*(9-161 ページ).
- *9.2.4 キューブマップのフィルタリング*(9-163 ページ).
- *9.2.5 光線とボックスの交差アルゴリズム*(9-168 ページ).
- *9.2.6 キューブマップを生成するためのエディタ スクリプトのソースコード*(9-171 ページ).

9.2.1 反射の実装の歴史

グラフィックス開発者は、常に計算コストの低い方法で反射を実装しようとしてきました。

最初の解決法の 1 つは、球面マッピングです。この手法では、コストの高いレイトレーシングやライティング計算を使用せずに、オブジェクトの反射やライティングをシミュレートできます。

次の図は、球面上の環境マップを示しています。

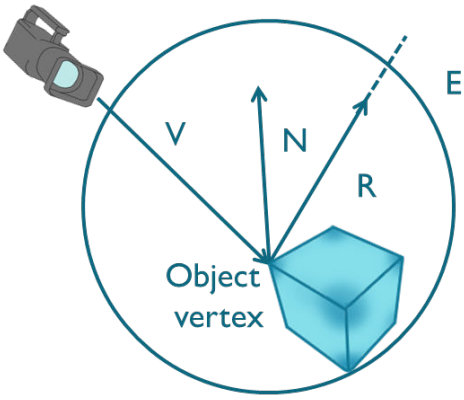


図 9-9 球面上の環境マップ

次の図は、球面を 2 次元にマッピングする式を示しています。

The spherical surface is mapped into 2D:

$$u = \frac{R_x}{m} + \frac{1}{2}$$

$$v = \frac{R_y}{m} + \frac{1}{2}$$

$$m = 2 \cdot \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

図 9-10 球表面の 2D マッピング式

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

このアプローチにはいくつかのデメリットがあります。主な問題は、画像を球にマッピングするときに発生する歪みです。1999 年には、ハードウェア アクセラレーションを使ったキューブマップの使用が可能になりました。キューブマップにより、球面マッピングに関連する、画像の歪み、視点の依存性、および計算の非効率性の問題が解消されました。

次の図は、展開されたキューブを示しています。

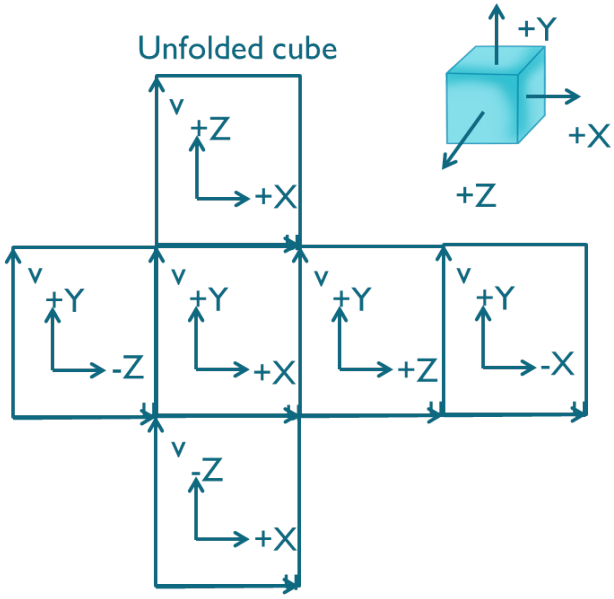


図 9-11 展開されたキューブ

キューブマッピングでは、キューブの 6 つの面をマップシェイプとして使用します。環境はキューブの各面に投影され、6 つの正方形のテクスチャとして格納されるか、または 1 つのテクスチャの 6 つの領域に展開されます。キューブマップは、90 度の錐台の視野を持ち、それぞれが立方体の 1 つの面を表す、異なる 6 つの方位のカメラを持つ、ある位置から、シーンをレンダリングすることによって生成されます。ソース画像は直接サンプリングされます。中間環境マップにリサンプリングすることにより、歪みは発生しません。

次の図は、無限反射を示しています。

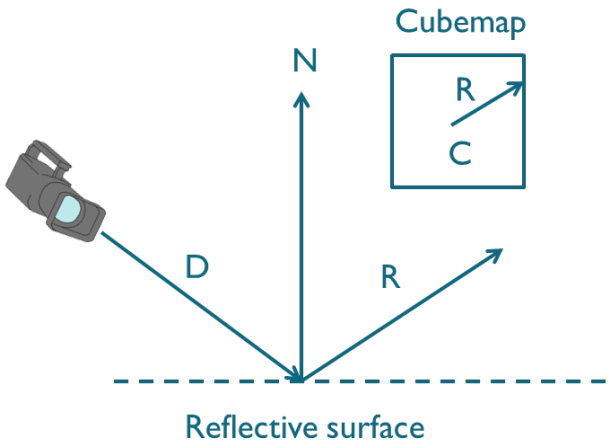


図 9-12 無限の反射

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

キューブマップに基づいて反射を実装するには、反射ベクトル R を評価し、それを使ってキューブマップ _Cubemap から、使用可能なテクスチャ ルックアップ関数 texCUBE() を使用してテクセルを取得します。

```
float4 color = texCUBE(_Cubemap, R);
```

法線 N とビュー ベクトル D は、頂点シェーダからフラグメント シェーダに渡されます。フラグメント シェーダはキューブマップからテクスチャのカラーを取得します。

```
float3 R = reflect(D, N);
float4 color = texCUBE(_Cubemap, R);
```

この方法では、キューブマップの位置が関係ない、離れた環境からの反射を正しく再現することのみが可能です。この簡単で効果的な手法は、主に屋外のライティング、たとえば空の反射を追加するために使用されます。

次の図は、正しくない反射を示しています。

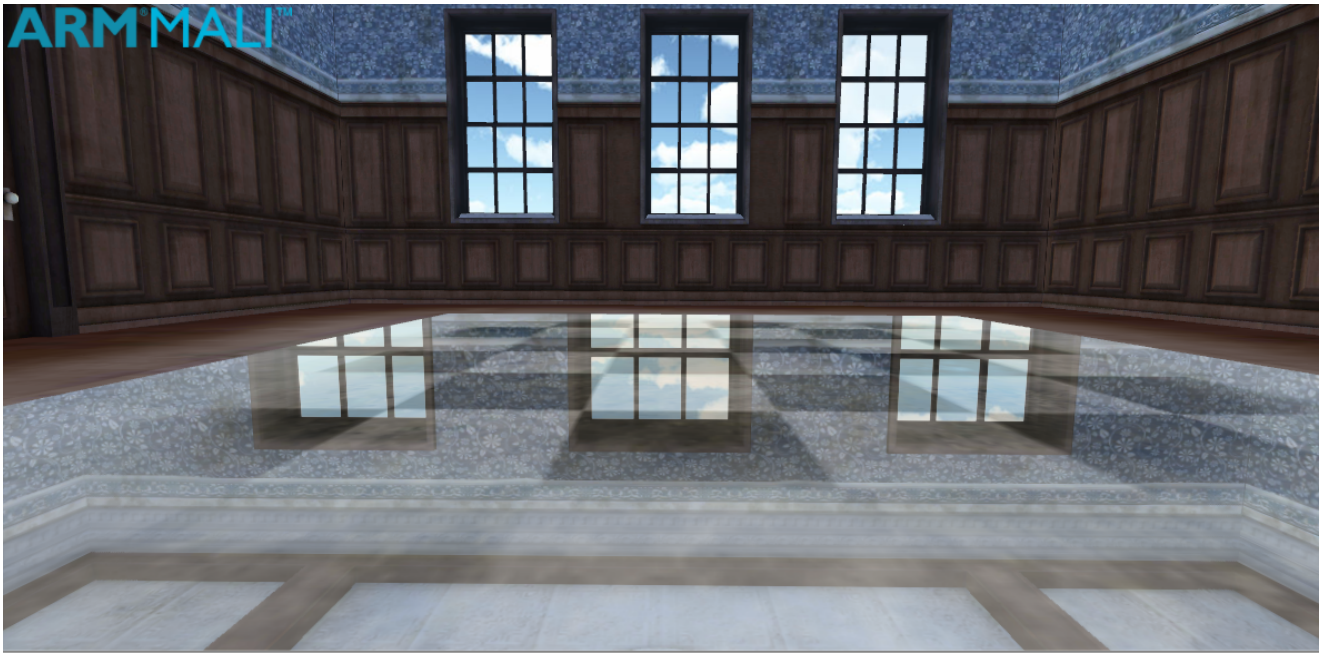


図 9-13 正しくない反射

この手法をローカル環境で使用すると、正しくない反射が発生します。反射が正しくないのは、float4 color = texCUBE(_Cubemap, R); の式で、ローカル ジオメトリへのバインディングがないためです。たとえば、反射のある床を横切って歩きながらその床を同じ角度から見ると、常に同じ反射が見えます。反射ベクトルは常に同じであり、式は常に同じ結果を生成します。これは、ビュー ベクトルの方向が変わらないためです。実際には、反射は視野角と見る位置の両方に依存します。

9.2.2 ローカル キューブマップで正しい反射を生成する

この問題の解決法は、反射を計算するためのプロシージャで、ローカル ジオメトリにバインドすることです。

この解決法は、「*GPU のメリット: リアルタイム グラフィックスのためのプログラミング技術とヒントとコツ* 著: Randima Fernando (シリーズ エディタ)」に記載されています。

次の図は、境界球を使用した局所補正を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

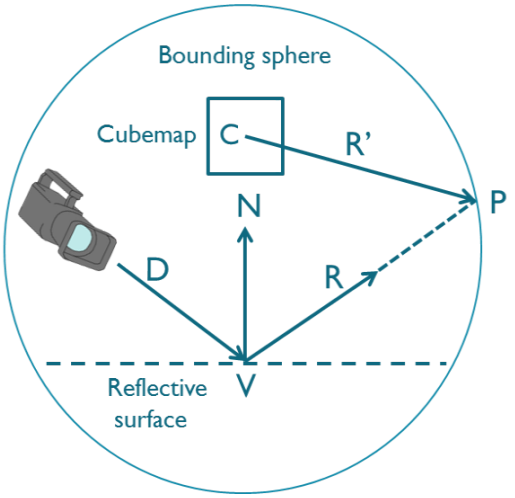


図 9-14 境界球面を使った局所補正

境界球は、反射されるシーンを区切る、プロキシ ボリュームとして使用されます。反射ベクトル R を使用してキューブマップからテクスチャを取得する代わりに、新しいベクトル R' が使用されます。この新しいベクトルを作成するには、光線の境界球で、ローカル ポイント V から、反射ベクトル R の方向に、交点 P を見つけます。キューブマップが生成された場所であるキューブマップ C の中心から、交点 P に、新しいベクトル R' を作成します。このベクトルを使用して、キューブマップからテクスチャを取得します。

```
float3 R = reflect(D, N);
Find intersection point P
Find vector R' = CP
float4 col = texCUBE(_Cubemap, R');
```

この手法は、ほぼ球形のオブジェクトの表面においては良好な結果をもたらしますが、平面の反射表面における反射は変形されます。この手法のもう 1 つのデメリットは、境界球との交点を計算するアルゴリズムが 2 次方程式を解くことであり、これは比較的複雑です。

2010 年に、ある開発者が *http://www.gamedev.net* のフォーラムで、より良い解決法を提案しました。この方法では、先述の境界球をボックスで置換し、先述の方法での変形と複雑性の問題を解決しています。詳細については、*http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/?&p=4637262* を参照してください。

次の図は、境界ボックスを使用した局所補正を示しています。

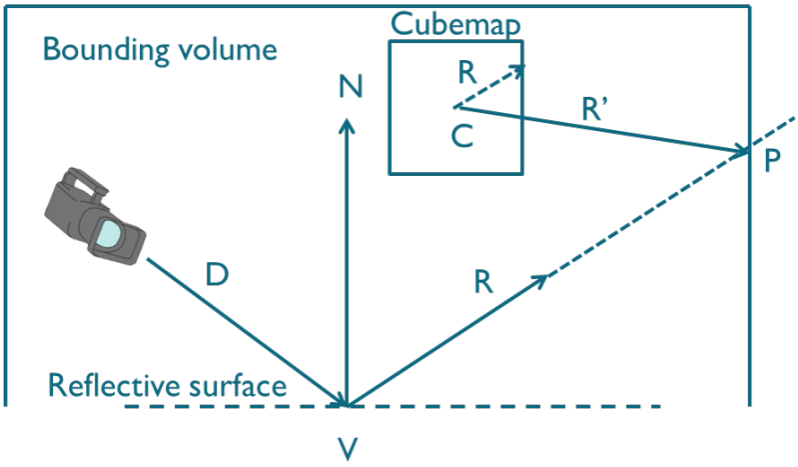


図 9-15 境界ボックスを使った局所補正

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

2012 年の Sebastien Lagarde による最近の研究では、この新しいアプローチを使い、複数のキューブマップを使用して、より複雑な周囲の鏡面反射のライティングをシミュレートし、アルゴリズムを使って、各キューブマップの影響を評価し、GPU で効率的にブレンドしています。*http://seblagarde.wordpress.com* を参照してください。

表 9-3 無限キューブマップとローカル キューブマップの相違点

無限キューブマップ	ローカル キューブマップ
• 離れた環境からのライティングを表現するために、主に屋外で使用されます。 • キューブマップの位置は関係ありません。	• 有限のローカル環境からのライティングを表現します。 • キューブマップの位置は意味があります。 • これらのキューブマップからのライティングは、キューブマップが作成された場所でのみ正しく表示されます。 • キューブマップは本来無限であるため、それをローカル環境に適応させるには、局所補正を適用する必要があります。

次の図は、ローカル キューブマップで生成された正しい反射のあるシーンを示しています。

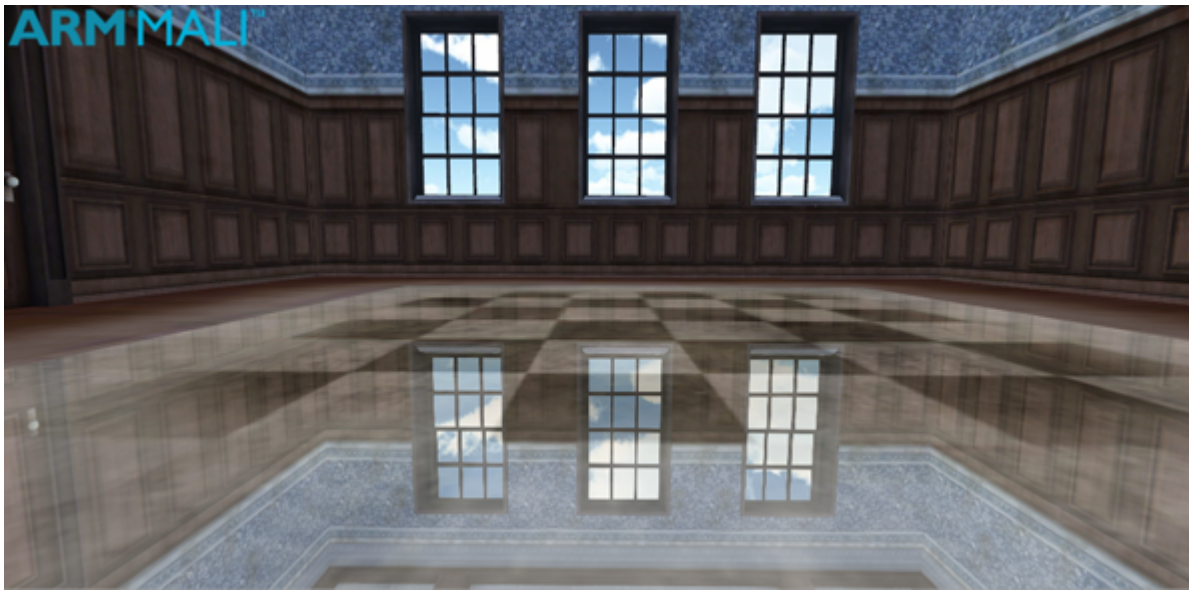


図 9-16 正しい反射

9.2.3 シェーダの実装

このセクションでは、ローカル キューブマップを使用して反射を実装するシェーダについて説明します。

頂点シェーダは、フラグメント シェーダに補間値として渡される、3 つの magnitude を計算します。

- 頂点の位置。
- ビューの方向。
- 法線。

これらの値はワールド座標系です。

次のコードは Unity での、ローカル キューブマップを使用した反射のシェーダの実装です。

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;
    // 頂点座標をローカルからワールドに変換する。
    float4 vertexWorld = mul(_Object2World, input.vertex);
    // 法線をワールド座標に変換する。
    float4 normalWorld = mul(float4(input.normal,0.0), _World2Object);
    // 最終的な頂点の出力位置。 output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    // ------------ 局所補正 ------------
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
    output.vertexInWorld = vertexWorld.xyz;
    output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
    output.normalInWorld = normalWorld.xyz;
    return output;
}
```

ボリューム ボックス内の交点と反射ベクトルは、フラグメント シェーダで計算されます。新しい局所補正反射ベクトルを作成し、これを使用して、ローカル キューブマップから反射テクスチャを取得します。次に、テクスチャと反射を組み合わせて、出力カラーを生成します。

```
float4 frag(vertexOutput input) : COLOR
{
    float4 reflColor = float4(1, 1, 0, 0);
    // WS で反射ベクトルを見つける。
    float3 viewDirWS = normalize(input.viewDirInWorld);
    float3 normalWS = normalize(input.normalInWorld);
    float3 reflDirWS = reflect(viewDirWS, normalWS);
    // ワールド座標系を使う。
    float3 localPosWS = input.vertexInWorld;
    float3 intersectMaxPointPlanes = (_BBoxMax - localPosWS) / reflDirWS;
    float3 intersectMinPointPlanes = (_BBoxMin - localPosWS) / reflDirWS;
    // 光線の前方の交差のみを探す。
    float3 largestParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);
    // 光線パラメータの最小値が交差を示す。
    float distToIntersect = min(min(largestParams.x, largestParams.y), largestParams.z);
    // 交点の位置を求める。
    float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;
    // 局所補正された反射ベクトルを取得する。
    Float3 localCorrReflDirWS = intersectPositionWS - _EnviCubeMapPos;
    // 正しいベクトルを持つ環境反射テクスチャをルックアップする。
    reflColor = texCUBE(_Cube, localCorrReflDirWS);
    // テクスチャ カラーをルックアップする。
    float4 texColor = tex2D(_MainTex, float2(input.tex));
    return _AmbientColor + texColor * _ReflAmount * reflColor;
}
```

前のフラグメント シェーダのコードでは、magnitude `_BBoxMax` と `_BBoxMin` が境界ボリュームの最大点と最小点です。変数 `_EnviCubeMapPos` は、キューブマップが作成された位置です。これらの値を次のスクリプトからシェーダに渡します。

```
[ExecuteInEditMode]
public class InfoToReflMaterial : MonoBehaviour
{
    // ローカル反射の計算に使用されるプロキシ ボリューム。
    public GameObject boundingBox;

    void Start()
    {
        Vector3 bboxLenght = boundingBox.transform.localScale;
        Vector3 centerBBox = boundingBox.transform.position;

        // ワールド座標での BBox の最小点と最大点
        Vector3 BMin = centerBBox - bboxLenght/2;
        Vector3 BMax = centerBBox + bboxLenght/2;

        // 値をマテリアルに渡す。
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMin", BMin);
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMax", BMax);
        gameObject.renderer.sharedMaterial.SetVector("_EnviCubeMapPos", centerBBox);
    }
}
```

`_AmbientColor`、`_ReflAmount`、メインテクスチャ、およびキューブマップ テクスチャの値をプロパティ ブロックのユニフォームとしてシェーダに渡します。

```
Shader "Custom/ctReflLocalCubemap"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" { }
        _Cube("Reflection Map", Cube) = "" {}
        _AmbientColor("Ambient Color", Color) = (1, 1, 1, 1)
        _ReflAmount("Reflection Amount", Float) = 0.5
    }
    SubShader
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
    {
        Pass
        {
            CGPROGRAM
            #pragma glsl
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            // ユーザ指定のユニフォーム
            uniform sampler2D _MainTex;
            uniform samplerCUBE _Cube;
            uniform float4 _AmbientColor;
            uniform float _ReflAmount;
            uniform float _ToggleLocalCorrection;
            // ----スクリプト InfoRoReflmaterial.cs から渡される--------
            uniform float3 _BBoxMin;
            uniform float3 _BBoxMax;
            uniform float3 _EnviCubeMapPos;

            struct vertexInput
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD0;
            };
            struct vertexOutput
            {
                float4 pos : SV_POSITION;
                float4 tex : TEXCOORD0;
                float3 vertexInWorld : TEXCOORD1;
                float3 viewDirInWorld : TEXCOORD2;
                float3 normalInWorld : TEXCOORD3;
            };

        Vertex shader    { }
        Fragment shader    { }

        ENDCG
        }
    }
}
```

境界ボリューム内の交点を計算するアルゴリズムは、ローカル位置またはフラグメントからの反射光線のパラメトリック表現の使用に基づいています。光線ボックス交差アルゴリズムの説明については、*9.2.5 光線とボックスの交差アルゴリズム*(9-168 ページ) を参照してください。

9.2.4 キューブマップのフィルタリング

ローカル キューブマップを使用して反射を実装する利点の 1 つは、キューブマップが静的であるということです。つまり、実行時ではなく開発時に生成されます。このため、エフェクトを実現するために、任意のフィルタリングをキューブマップ画像に適用することができます。

AMD によるツールである CubeMapGen は、キューブマップにフィルタリングを適用します。CubeMapGen は AMD の開発者 Web サイト *http://developer.amd.com* から入手できます。

キューブマップ画像を Unity から CubeMapGen にエクスポートするには、各キューブマップの画像を個別に保存する必要があります。画像を保存するツールのソースコードについては、*9.2.6 キューブマップを生成するためのエディタ スクリプトのソースコード*(9-171 ページ) を参照してください。このツールはキューブマップを作成することができ、オプションで各キューブマップの画像を別々に保存することができます。

このツールのスクリプトは、Asset ディレクトリの `Editor` というフォルダに配置する必要があります。

キューブマップ エディタ ツールを使用するには、

1. キューブマップを作成します。
2. GameObject メニューから Bake CubeMap ツールを起動します。
3. キューブマップとカメラのレンダリング位置を指定します。
4. オプションとして、キューブマップにフィルタリングを適用する場合は、個々の画像を保存します。

次の図は、Bake CubeMap ツールのインタフェースを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

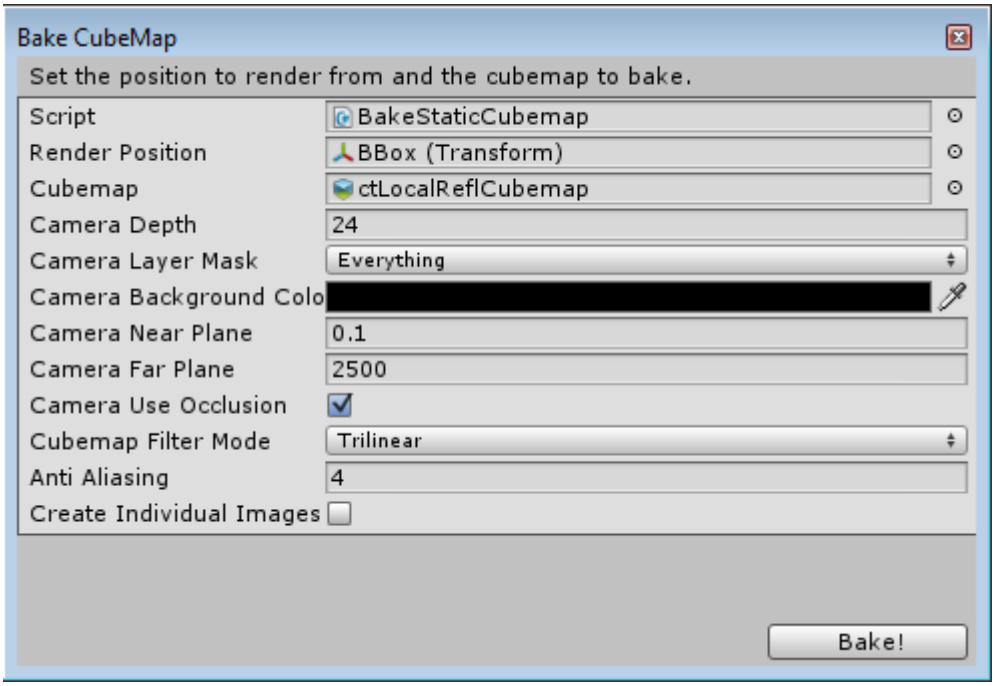


図 9-17 ベーキング キューブマップ ツールのインタフェース

CubeMapGen を使用して、キューブマップの各画像を個別に読み込むことができます。

[Select Cube Face] ドロップダウン メニューから読み込む面を選択し、[Load Cubemap Face] ボタンを押します。すべての面を読み込んだら、キューブマップを回転して、正しいことを確認することができます。

CubeMapGen には、[Filter Type]ドロップダウン メニューに、さまざまなフィルタリング オプションがあります。必要なフィルタ設定を選択し、[Filter Cubemap] を押して、フィルタを適用します。フィルタリングには、キューブマップのサイズに応じて、数分かかることがあります。元に戻すオプションはありません。フィルタリングを適用する前に、キューブマップを単一の画像として保存してください。フィルタリングの結果が期待に沿わない場合は、キューブマップをリロードして、パラメータを調整してみてください。

キューブマップ画像を CubeMapGen にインポートするには、次の手順を実行します。

1. キューブマップをベイクする際に個々の画像を保存するための、チェックボックスをオンにします。
2. 次の表に示す関係に従って、CubeMapGen ツールを起動し、キューブマップ画像を読み込みます。
3. キューブマップを単一の DDS またはキューブ クロス イメージとして保存します。元に戻すことはできません。フィルタを試している場合は、キューブマップをリロードすることができます。
4. 必要に応じて、十分な結果が得られるまで、キューブマップにフィルタを適用します。
5. キューブマップを個々の画像として保存します。

次の表は、CubeMapGen と Unity の間で同等なキューブマップのフェイス インデックスを示しています。

表 9-4 CubeMapGen と Unity の間で同等なキューブマップのフェイス インデックス

AMD CubeMapGen	Unity
X+	-X
X-	+X
Y+	+Y
Y-	-Y

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

表 9-4 CubeMapGen と Unity の間で同等なキューブマップのフェイス インデックス（続き）

AMD CubeMapGen	Unity
Z+	+Z
Z-	-Z

次の図は、6 つのキューブマップ画像を読み込んだ後の CubeMapGen を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9-18 CubeMapGen

次の図は、CubeMapGen でガウス フィルタを適用して、*霜*のようなエフェクトを実行した結果を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

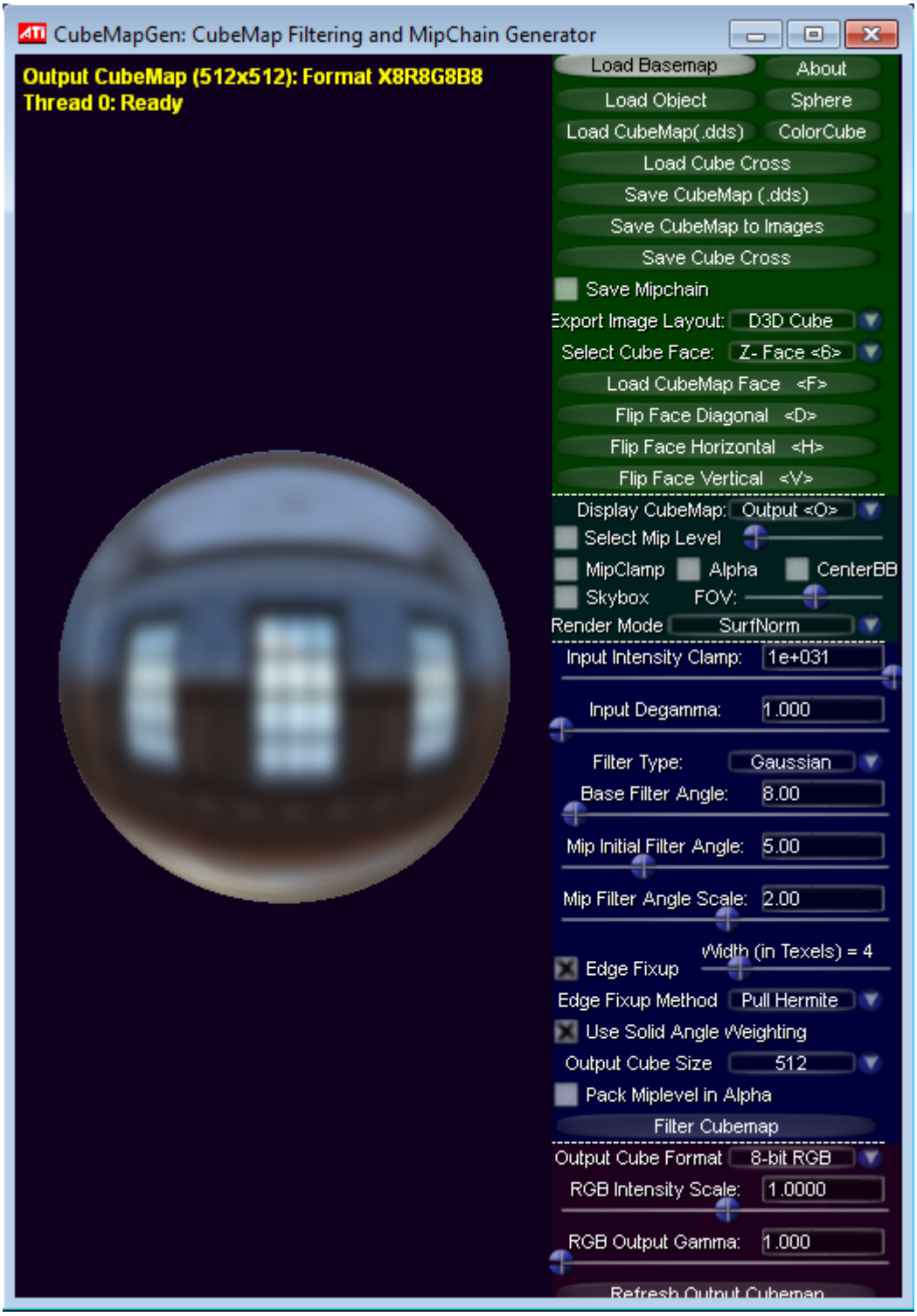


図 9–19 霜のようなエフェクトを示す CubeMapGen

次の表は、霜のようなエフェクトを得るためにガウス フィルタで使用されるフィルタ パラメータを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

表 9-5 反射に霜のようなエフェクトを生成するために CubeMapGen で使用するパラメータ。

フィルタ設定	値
Type	Gaussian
Base Filter Angle	8
Mip Initial Filter Angle	5
Mip Filter Angle Scale	2.0
Edge Fixup	Checked
Edge Fixup Width	4

次の図は、霜のエフェクトを使ったキューブマップで生成された反射を示しています。

図 9-20 霜のようなエフェクトを使った反射

次の図は、CubeMapGen ツールを使用して、Unity キューブマップにフィルタリングを適用する作業フローをまとめたものです。

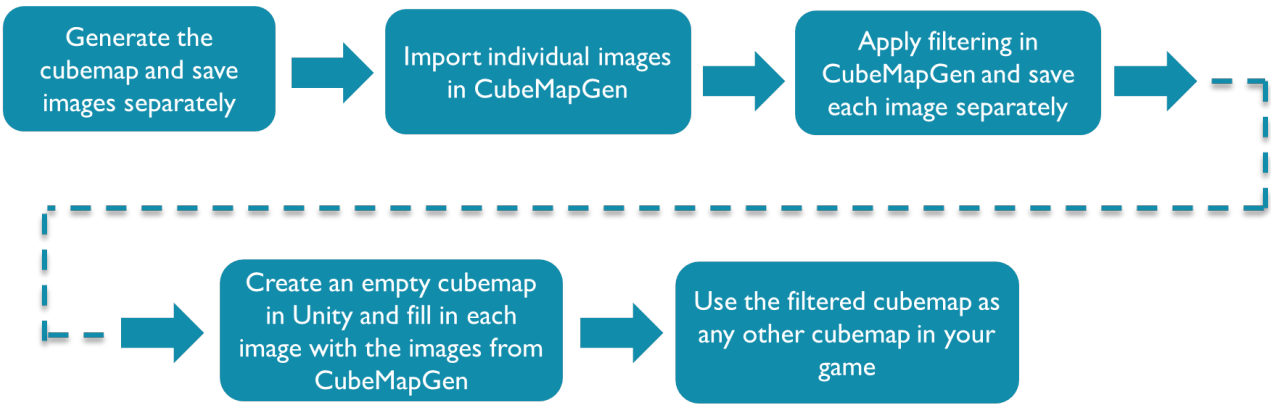


図 9-21 キューブマップのフィルタリング ワークフロー

9.2.5 光線とボックスの交差アルゴリズム

このセクションでは、光線とボックスの交差アルゴリズムについて説明します。

次の図は、線の式を含むグラフを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

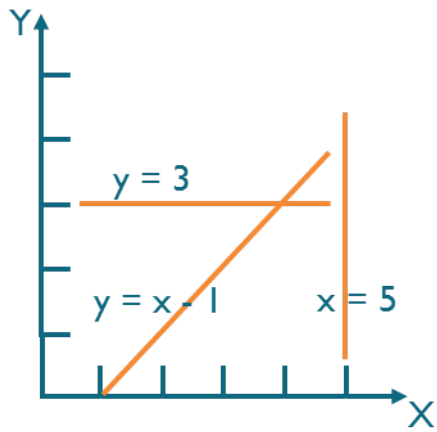


図 9-22 線の式を使ったグラフ

線の式

```
y = mx + b
```

この式のベクトル形式は次のとおりです。

```
r = O + t*D
```

各項目は次のとおりです。

`O` は原点です

`D` は方向ベクトルです

`t` はパラメータです

次の図は、軸にアラインされた境界ボックスを示しています。

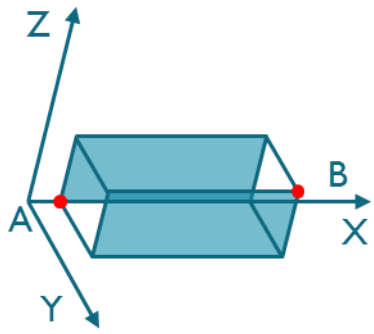


図 9-23 軸にアラインされた境界ボックス

軸にアラインされた境界ボックス AABB は、その最小点 A と最大点 B で定義できます。

AABB は、座標軸に平行な線の集合を定義します。線の各成分は、次の式で定義することができます。

$$x = A_x;\ y = A_y;\ z = A_z x = B_x;\ y = B_y;\ z = B_z$$

ある光線がそれらの線の 1 つと交差する場所を見つけるには、両方の式を等しくします。たとえば:

$$O_x + t_x * D_x = A_x$$

解は次のように記述できます。

$$t_{Ax} = (A_x - O_x) / D_x$$

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

同様に、両方の交点のすべての成分の解を求めることができます。

```
tAx = (Ax - Ox) / Dx
tAy = (Ay - Oy) / Dy
tAz = (Az - Oz) / Dz
tBx = (Bx - Ox) / Dx
tBy = (By - Oy) / Dy
tBz = (Bz - Oz) / Dz
```

ベクトル形式では、次のとおりです。

```
tA = (A - O) / D
tB = (B - O) / D
```

これによって、線がキューブの面によって定義された面と交差する場所を検出できますが、交点がキューブ上にあることは保証されません。

次の図は、光線とボックスの交差の 2D 表現を示しています。

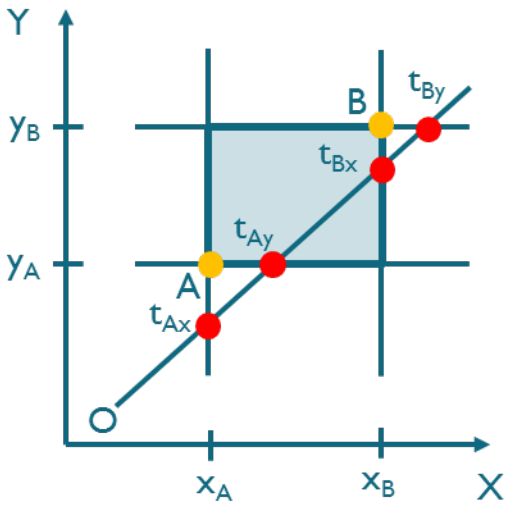


図 9-24 光線とボックスの交差部の 2D 表現

ボックスとの交差の実際の解を知るには、最小平面の交点の t パラメータの値を大きくする必要があります。

```
tmin = (tAx> tAy) ? tAx: tAy
```

最大平面の交点の t パラメータの値を小さくする必要があります。

```
tmin = (tAx> tAy) ? tAx: tAy
```

交点がない場合も考慮する必要があります。

次の図は、光線とボックスの交差がない場合を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

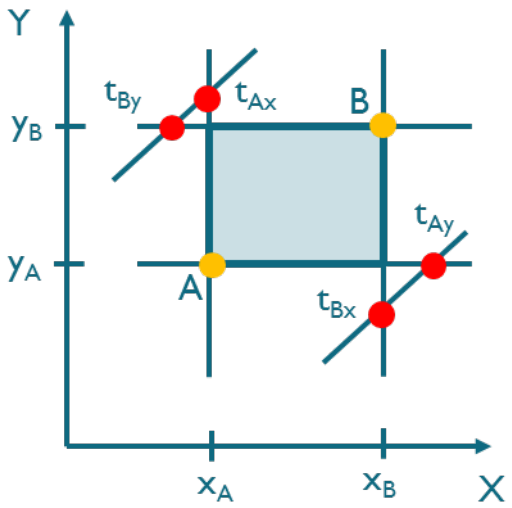


図 9-25 交差部のない光線とボックス

反射面が BBox で囲まれていること、つまり反射光線の原点が BBox の内側にあることが保証されている場合には、常にボックスとの交点が 2 つあるため、異なるケースの取り扱いが簡単になります。

次の図は、BBox 内の光線とボックスの交差を示しています。

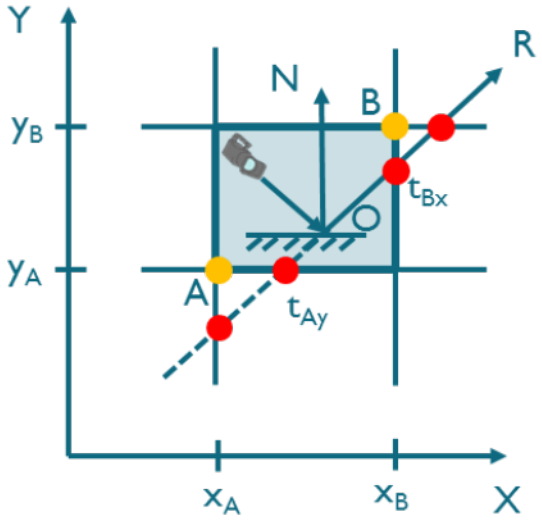


図 9-26 BBox の光線とボックスの交差部

9.2.6 キューブマップを生成するためのエディタ スクリプトのソースコード

このセクションでは、キューブマップを生成するためのエディタ スクリプトのソースコードを提供します。

```
/*
* This confidential and proprietary software may be used only as
* authorised by a licensing agreement from ARM Limited* (C) COPYRIGHT 2014 ARM Limited* ALL
RIGHTS RESERVED
* The entire notice above must be reproduced on all authorised
* copies and copies may only be made to the extent permitted
* by a licensing agreement from ARM Limited.
*/

using UnityEngine;
using UnityEditor;
using System.IO;

/**
* This script must be placed in the Editor folder.
* The script renders the scene into a cubemap and optionally
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
* saves each cubemap image individually.
* The script is available in the Editor mode from the
 * Game Object menu as "Bake Cubemap" option.
* Be sure the camera far plane is enough to render the scene.
*/

public class BakeStaticCubemap : ScriptableWizard
{

    public Transform renderPosition;
    public Cubemap cubemap;
    // Camera settings.
    public int cameraDepth = 24;
    public LayerMask cameraLayerMask = -1;
    public Color cameraBackgroundColor;
    public float cameraNearPlane = 0.1f;
    public float cameraFarPlane = 2500.0f;
    public bool cameraUseOcclusion = true;
    // Cubemap settings.
    public FilterMode cubemapFilterMode = FilterMode.Trilinear;
    // Quality settings.
    public int antiAliasing = 4;

    public bool createIndividualImages = false;

    // The folder where individual cubemap images will be saved
    static string imageDirectory = "Assets/CubemapImages";
    static string[] cubemapImage
         = new string[]{"front+Z", "right+X", "back-Z", "left-X", "top+Y", "bottom-Y"};
           static Vector3[] eulerAngles = new Vector3[]{new Vector3(0.0f,0.0f,0.0f),
           new Vector3(0.0f,-90.0f,0.0f), new Vector3(0.0f,180.0f,0.0f),
           new Vector3(0.0f,90.0f,0.0f), new Vector3(-90.0f,0.0f,0.0f),
           new Vector3(90.0f,0.0f,0.0f)};

    void OnWizardUpdate()
    {
        helpString = "Set the position to render from and the cubemap to bake.";
        if(renderPosition != null && cubemap != null)
        {
            isValid = true;
        }
        else
        {
            isValid = false;
        }
    }

    void OnWizardCreate ()
    {

        // Create temporary camera for rendering.
        GameObject go = new GameObject( "CubemapCam", typeof(Camera) );
        // Camera settings.
        go.camera.depth = cameraDepth;
        go.camera.backgroundColor = cameraBackgroundColor;
        go.camera.cullingMask = cameraLayerMask;
        go.camera.nearClipPlane = cameraNearPlane;
        go.camera.farClipPlane = cameraFarPlane;
        go.camera.useOcclusionCulling = cameraUseOcclusion;
        // Cubemap settings
        cubemap.filterMode = cubemapFilterMode;
        // Set antialiasing
        QualitySettings.antiAliasing = antiAliasing;

        // Place the camera on the render position.
        go.transform.position = renderPosition.position;
        go.transform.rotation = Quaternion.identity;

        // Bake the cubemap
        go.camera.RenderToCubemap(cubemap);

        // Rendering individual images
        if(createIndividualImages)
        {
            if (!Directory.Exists(imageDirectory))
            {
                Directory.CreateDirectory(imageDirectory);
            }

            RenderIndividualCubemapImages(go);
        }


        // Destroy the camera after rendering.
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
        DestroyImmediate(go);
    }

    void RenderIndividualCubemapImages(GameObject go)
    {
        go.camera.backgroundColor = Color.black;
        go.camera.clearFlags = CameraClearFlags.Skybox;
        go.camera.fieldOfView = 90;
        go.camera.aspect = 1.0f;

        go.transform.rotation = Quaternion.identity;

        //Render individual images
        for (int camOrientation = 0; camOrientation < eulerAngles.Length ; camOrientation++)
        {
            string imageName = Path.Combine(imageDirectory, cubemap.name + "_"
                    + cubemapImage[camOrientation] + ".png");
            go.camera.transform.eulerAngles = eulerAngles[camOrientation];
            RenderTexture renderTex = new RenderTexture(cubemap.height,
                    cubemap.height, cameraDepth);
            go.camera.targetTexture = renderTex;
            go.camera.Render();
            RenderTexture.active = renderTex;

            Texture2D img = new Texture2D(cubemap.height, cubemap.height,
                    TextureFormat.RGB24, false);
            img.ReadPixels(new Rect(0, 0, cubemap.height, cubemap.height), 0, 0);

            RenderTexture.active = null;
            GameObject.DestroyImmediate(renderTex);

            byte[] imgBytes = img.EncodeToPNG();
            File.WriteAllBytes(imageName, imgBytes);

            AssetDatabase.ImportAsset(imageName, ImportAssetOptions.ForceUpdate);
        }

        AssetDatabase.Refresh();
    }

    [MenuItem("GameObject/Bake Cubemap")]
    static void RenderCubemap ()
    {
        ScriptableWizard.DisplayWizard("Bake CubeMap", typeof(BakeStaticCubemap),"Bake!");
    }
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.3 反射の結合

このセクションでは、反射の結合について説明します。

以下のサブセクションから構成されています。

- *9.3.1 反射の結合について*(9-174 ページ).
- *9.3.2 反射シェーダの実装を結合する*(9-176 ページ).
- *9.3.3 離れた環境からの反射を組み合わせる*(9-178 ページ).

9.3.1 反射の結合について

ローカル キューブマップの手法に基づく反射を使うと、静的なローカル キューブマップに基づく、高品質かつ効率的な反射のレンダリングができます。しかし、オブジェクトが動的である場合には、静的なローカル キューブマップは有効ではなくなり、この手法は機能しません。

静的な反射と、動的に生成された反射を組み合わせることで、これを解決できます。

次の図は、静的および動的なジオメトリからの反射の組み合わせを示しています。

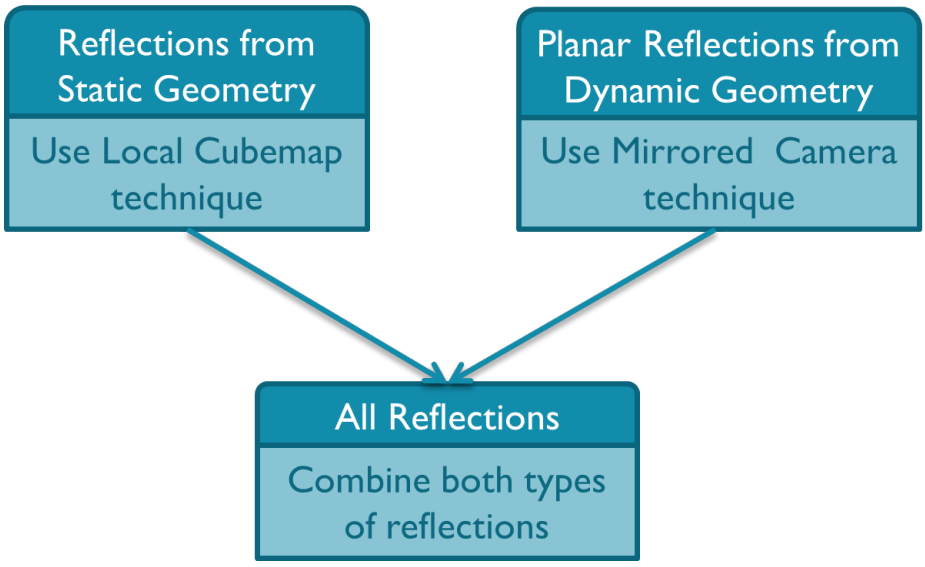


図 9-27 静的および動的ジオメトリからの反射の結合

反射面が平面の場合は、ミラーリングされたカメラで、動的な反射を生成できます。

ミラーリングされたカメラを作成するには、実行時に反射をレンダリングする、メイン カメラの位置と向きを計算します。

メイン カメラの位置と向きを、反射平面と相対的に、ミラーリングします。

次の図は、ミラーリングされたカメラの手法を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

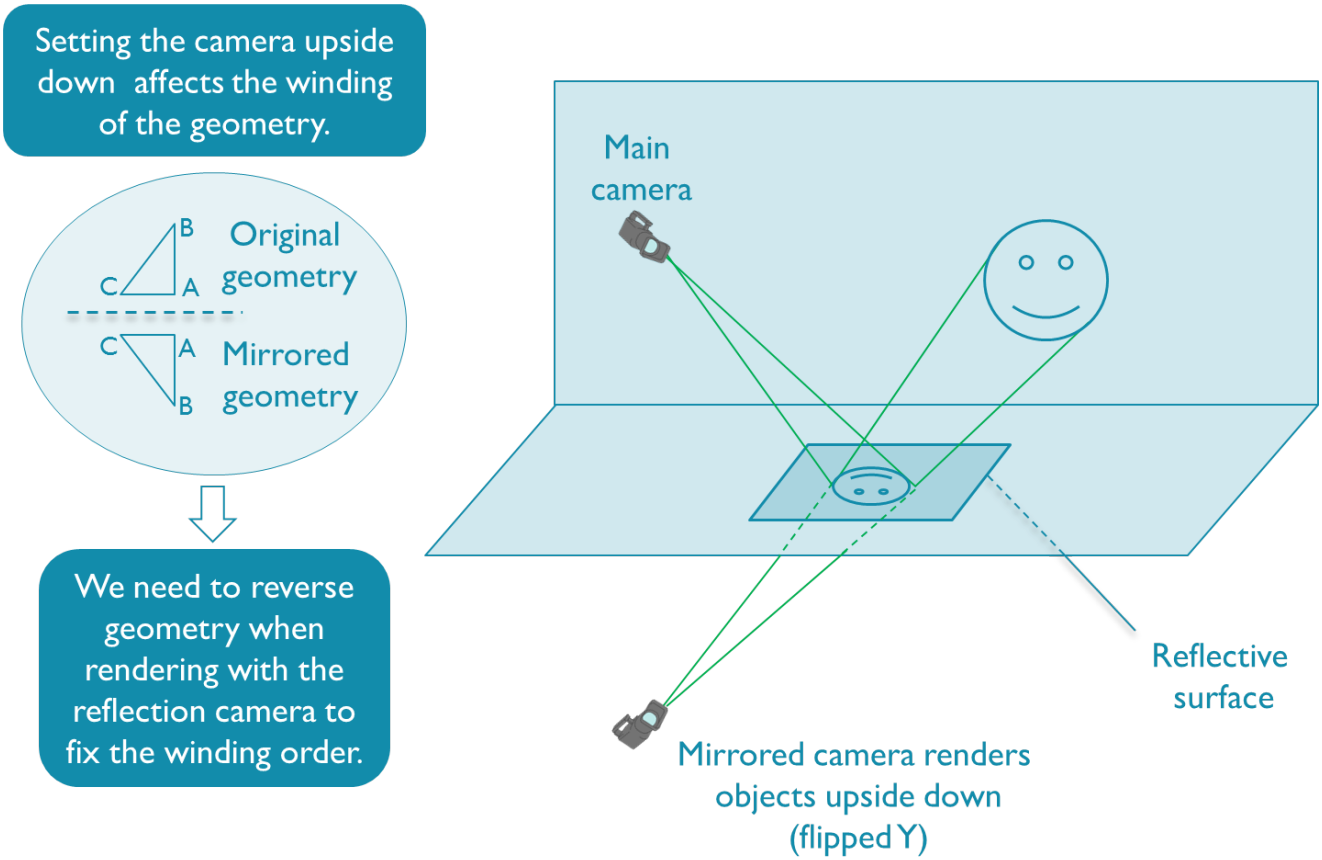


図 9-28 平面反射をレンダリングするための、ミラーリングされたカメラの手法

ミラーリング処理では、新しい反射カメラの軸は反対の向きとなります。物理的なミラーと同様に、左右からの反射が反転します。これは反射カメラが、逆の屈曲でジオメトリをレンダリングすることを意味します。

ジオメトリを正しくレンダリングするには、反射をレンダリングする前に、ジオメトリの屈曲を反転する必要があります。反射のレンダリングが終了したら、元の屈曲を復元します。

次の図は、ミラーリングされたカメラを設定し、反射をレンダリングするために必要な、一連の手順を示しています。

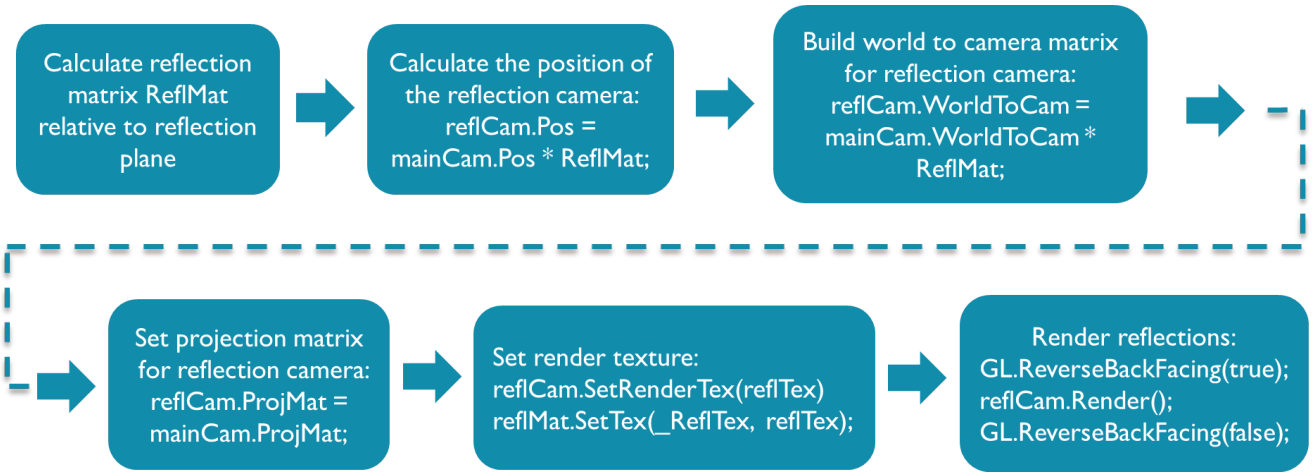


図 9-29 ミラーリングされたカメラを設定し、反射をレンダリングするための主なステップ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

ミラー反射の変換行列を作成します。この行列を使用して、反射カメラの位置と、ワールド ツー カメラ変換行列を計算します。

次の図は、ミラー反射変換行列を示しています。

$$R = \begin{bmatrix} 1-2n_x^2 & -2n_xn_y & -2n_xn_z & -2n_zn_w \\ -2n_xn_y & 1-2n_y^2 & -2n_yn_z & -2n_yn_w \\ -2n_xn_z & -2n_yn_z & 1-2n_z^2 & -2n_zn_w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} n_x &= planeNormal.x \\ n_y &= planeNormal.y \\ n_z &= planeNormal.z \\ n_w &= -dot(planeNormal, planePos) \end{aligned}$$

図 9-30 ミラー反射変換行列

反射行列変換をメイン カメラの位置およびワールド ツー カメラ行列に適用します。これにより、反射カメラの位置と、ワールド ツー カメラ行列が提供されます。

反射カメラの投影行列は、メイン カメラの投影行列と同じでなければなりません。

反射カメラは、反射をテクスチャにレンダリングします。

適切な結果を得るには、レンダリングの前に、このテクスチャを適切に設定する必要があります。

- ミップマップを使用します。
- フィルタリング モードを三線形に設定します。
- マルチサンプリングを使用します。

テクスチャ サイズが反射面の面積に比例することを確認します。テクスチャが大きくなるほど、反射のピクセル化は少なくなります。

ミラーリングされたカメラのスクリプトのサンプルは、*http://wiki.unity3d.com/index.php/File:MirrorReflection.png* を参照してください。

9.3.2 反射シェーダの実装を結合する

静的な環境の反射と、動的な平面反射を、シェーダで組み合わせることができます。

シェーダで反射を組み合わせるには、*9.2.3 シェーダの実装*(9-161 ページ) で提供されているシェーダのコードを変更する必要があります。

シェーダは、実行時に反射カメラでレンダリングされた、平面反射を組み込む必要があります。これを行うには、反射カメラからのテクスチャ `_ReflectionTex` をフラグメント シェーダにユニフォームとして渡し、`lerp()` 関数を使用して、平面反射結果と結合します。

局所補正に関連するデータに加えて、頂点シェーダは、組み込み関数 `ComputeScreenPos()` を使用して、頂点のスクリーン座標も計算します。これらの座標をフラグメント シェーダに渡します。

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;

    // 頂点座標をローカルからワールドに変換する。
    float4 vertexWorld = mul(_Object2World, input.vertex);
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
    // 法線をワールド座標に変換する。
    float4 normalWorld = mul(float4(input.normal,0.0), _World2Object);

    // 最終的な頂点出力位置。
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);

    // ----------- 局所補正 -------------
    output.vertexInWorld = vertexWorld.xyz;
    output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
    output.normalInWorld = normalWorld.xyz;

    // ----------- 平面反射 -------------
    output.vertexInScreenCoords = ComputeScreenPos(output.pos);
    return output;
}
```

平面反射はテクスチャにレンダリングされるので、フラグメント シェーダはフラグメントのスクリーン座標にアクセスできます。これを有効にするには、頂点のスクリーン座標をフラグメント シェーダに可変変数として渡します。

フラグメント シェーダで、

- 反射ベクトルに局所補正を適用します。
- ローカル キューブマップから環境の反射 `staticReflColor` のカラーを取得します。

次のコードは、ローカル キューブマップ手法を使った静的環境の反射と、ミラーリングされたカメラの手法を使用して実行時にレンダリングされる動的な平面の反射を、組み合わせる方法を示しています。

```
float4 frag(vertexOutput input) : COLOR
{
    float4 staticReflColor = float4(1, 1, 1, 1);

    // WS で反射ベクトルを見つける。
    float3 viewDirWS = normalize(input.viewDirInWorld);
    float3 normalWS = normalize(input.normalInWorld);
    float3 reflDirWS = reflect(viewDirWS, normalWS);

    // ワールド座標系を使う。
    float3 localPosWS = input.vertexInWorld;
    float3 intersectMaxPointPlanes = (_BBoxMax - localPosWS) / reflDirWS;
    float3 intersectMinPointPlanes = (_BBoxMin - localPosWS) / reflDirWS;

    // 光線の前方の交差のみを探す。
    float3 largestParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);

    // 光線パラメータの最小値が交差を示す。
    float distToIntersect = min(min(largestParams.x, largestParams.y), largestParams.z);

    // 交点の位置を求める。
    float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;

    // 局所補正された反射ベクトルを取得する。
    float3 localCorrReflDirWS = intersectPositionWS - _EnviCubeMapPos;

    // 正しいベクトルを持つ環境反射テクスチャをルックアップする。
    float4 staticReflColor = texCUBE(_Cube, localCorrReflDirWS);

    // 平面のランタイム テクスチャをルックアップする
    float4 dynReflColor = tex2Dproj(_ReflectionTex,
    UNITY_PROJ_COORD(input.vertexInScreenCoords);

    // 反射カメラの背景色でブレンドを元に戻す
    dynReflColor.rgb /= (dynReflColor.a < 0.00392)?1:dynReflColor.a;

    // 静的環境の反射と動的な平面の反射を組み合わせる
    float4 combinedRefl = lerp(staticReflColor.rgb, dynReflColor.rgb, dynReflColor.a);

    // テクスチャ カラーをルックアップする。
    float4 texColor = tex2D(_MainTex, float2(input.tex));
    return _AmbientColor + texColor * _ReflAmount * combinedRefl;
}
```

平面ランタイム反射テクスチャ `_ReflectionTex` からテクスチャ カラー `dynReflColor` を抽出します。

`_ReflectionTex` をシェーダのユニフォームとして宣言します。

プロパティ ブロックで `_ReflectionTex` を宣言します。これにより、実行時にどのように見えるかを確認することができ、ゲームの開発中のデバッグをサポートします。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

テクスチャ ルックアップには、テクスチャ座標を投影します。つまり、テクスチャ座標を座標ベクトルの最後の成分で除算します。Unity の組み込み関数 `UNITY_PROJ_COORD()` を使用できます。

`lerp()` 関数を使用して、静的環境の反射と動的な平面の反射を組み合わせます。次のものを結合します。

- 反射のカラー。
- 反射面のテクスチャ カラー。
- アンビエント カラーの成分。

9.3.3 離れた環境からの反射を組み合わせる

静的オブジェクトや動的オブジェクトからの反射をレンダリングする場合は、離れた環境からの反射も考慮する必要があります。たとえば、ローカル環境の窓から見える空からの反射などです。

この場合、3 種類の反射を組み合わせる必要があります。

- ローカル キューブマップ手法を用いた静的環境からの反射。
- ミラーリングされたカメラの手法を使用した動的オブジェクトからの平面反射。
- 標準のキューブマップ手法を使ったスカイボックスからの反射。反射ベクトルは、キューブマップからテクスチャを取得する前に修正する必要はありません。

スカイボックスからの反射を組み込むには、反射ベクトル `reflDirWS` を使用して、スカイボックスのキューブマップからテクセルを取得します。スカイボックスのキューブマップ テクスチャをユニフォームとしてシェーダに渡します。

注

局所補正を適用しないでください。

スカイボックスがウィンドウからのみ表示されるようにするには、反射の静的キューブマップをベーキングするときに、アルファ チャネルでシーンの透明度をレンダリングします。

値 1 を不透明なジオメトリに割り当て、ジオメトリが存在しないか、またはジオメトリが完全に透明である場合は、値 0 を割り当てます。たとえば、アルファ チャネルでウィンドウに対応するピクセルを 0 でレンダリングします。

スカイボックスのキューブマップ `_Skybox` をユニフォームとしてシェーダに渡します。

9.3.2 反射シェーダの実装を結合する(9-176 ページ) のフラグメント シェーダ コードで、次のコメントを探します。

// 平面のランタイム テクスチャをルックアップする

このコメントの前に、次の行を挿入します。

```
float4 skyboxReflColor = texCUBE(_Skybox, reflDirWS);
staticReflColor = lerp(skyboxReflColor.rgb, staticReflColor.rgb, staticReflColor.a);
```

このコードは静的反射とスカイボックスからの反射を組み合わせます。

次の図は、さまざまな種類の反射を組み合わせたものです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9–31 さまざまな種類の反射の組み合わせ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.4 ローカル キューブマップに基づく動的なソフト シャドウ

この手法では、ローカル キューブマップを使用して、静的環境の透明度を表すテクスチャを保持します。これは、高品質のソフト シャドウを生成する、非常に効率的な手法です。

以下のサブセクションから構成されています。

- *9.4.1 ローカル キューブマップに基づく動的なソフト シャドウについて*(9-180 ページ).
- *9.4.2 シャドウ キューブマップの生成*(9-180 ページ).
- *9.4.3 影のレンダリング*(9-180 ページ).
- *9.4.4 キューブマップ シャドウとシャドウ マップの組み合わせ*(9-184 ページ).
- *9.4.5 キューブマップ シャドウ手法の結果*(9-185 ページ).

9.4.1 ローカル キューブマップに基づく動的なソフト シャドウについて

シーンには、移動するオブジェクトや、部屋などの静的な環境が含まれます。この手法を使用すると、フレームごとに静的なジオメトリをシャドウ マップにレンダリングする必要はありません。これにより、テクスチャを使用して影を表すことができます。

キューブマップは多くの場合、Ice Cave デモの洞くつのような不規則な形状を含む、多くの種類の静的なローカル環境の、適切な近似です。アルファ チャネルは、部屋に入る光の量を表すこともできます。

移動するオブジェクトは通常、部屋を除くすべてのものです。次のようなオブジェクトです。

- 太陽。
- カメラ。
- 動的なオブジェクト。

部屋全体をキューブ テクスチャで表現すると、フラグメント シェーダ内の環境の任意のテクセルにアクセスできます。これはたとえば、太陽を任意の位置に配置し、キューブマップから取得した値に基づいて、フラグメントに届く光の量を計算することができます。

アルファ チャネルまたは透明度は、ローカル環境に入る光の量を表します。シーンで、キューブマップ テクスチャをフラグメント シェーダにアタッチします。シェーダにより、静的オブジェクトと動的オブジェクトがレンダリングされ、影が追加されます。

9.4.2 シャドウ キューブマップの生成

まず、環境外の光源から影を適用するローカル環境から始めます。たとえば、部屋、洞くつ、おりなどです。

この手法は、ローカル キューブマップに基づく反射に似ています。詳細については、*9.2 ローカル キューブマップによる反射の実装*(9-157 ページ)を参照して下さい。

反射キューブマップと同じ方法でシャドウ キューブマップを作成しますが、アルファ チャネルも追加する必要があります。アルファ チャネルまたは透明度は、ローカル環境に入る光の量を表します。

キューブマップの 6 つの面をレンダリングする位置を求めます。多くの場合、これはローカル環境の境界ボックスの中央にあります。キューブマップを生成するには、この位置が必要となります。さらにこの位置をシェーダに渡し、局所補正ベクトルを計算して、キューブマップから正しいテクセルをフェッチする必要があります。

キューブマップの中心を配置する場所を決めたら、すべての面をキューブマップ テクスチャにレンダリングし、ローカル環境の透明度またはアルファを記録できます。エリアが透明であればあるほど、より多くの光が環境に入ります。ジオメトリがない場合は、完全に透明です。必要に応じて、RGB チャネルを使用して、ステンドグラス、反射、屈折などの、カラーの付いた影の環境色を保存できます。

9.4.3 影のレンダリング

頂点またはフラグメントからライトまでのワールド空間のベクトル P_iL を作成し、このベクトルを使用してキューブマップ シャドウをフェッチします。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

各テクセルをフェッチする前に、P_iL ベクトルに局所補正を適用する必要があります。ARM では、より正確な影を得るために、フラグメント シェーダで局所補正を行うことを推奨しています。

局所補正を計算するには、フラグメントからライトへのベクトルと環境の境界ボックスとの交点を計算する必要があります。この交点を使用して、キューブマップの原点位置 C から交点 P までの、別のベクトルを作成します。これにより、テクセルをフェッチするために使用する最終的なベクトル CP が得られます。

局所補正を計算するには、次の入力パラメータが必要です。

- _EnviCubeMapPos キューブマップの原点位置です。
- _BBoxMax 環境の境界ボックスの最大点です。
- _BBoxMin 環境の境界ボックスの最小点です。
- Pi ワールド空間におけるフラグメントの位置です。
- PiL ワールド空間における正規化されたフラグメントからライトへのベクトルです。

出力値 CP を計算します。これは、シャドウ キューブマップからテクセルをフェッチするために使用する、フラグメントからライトへの補正されたベクトルです。

次の図は、フラグメントからライトへのベクトルの局所補正を示しています。

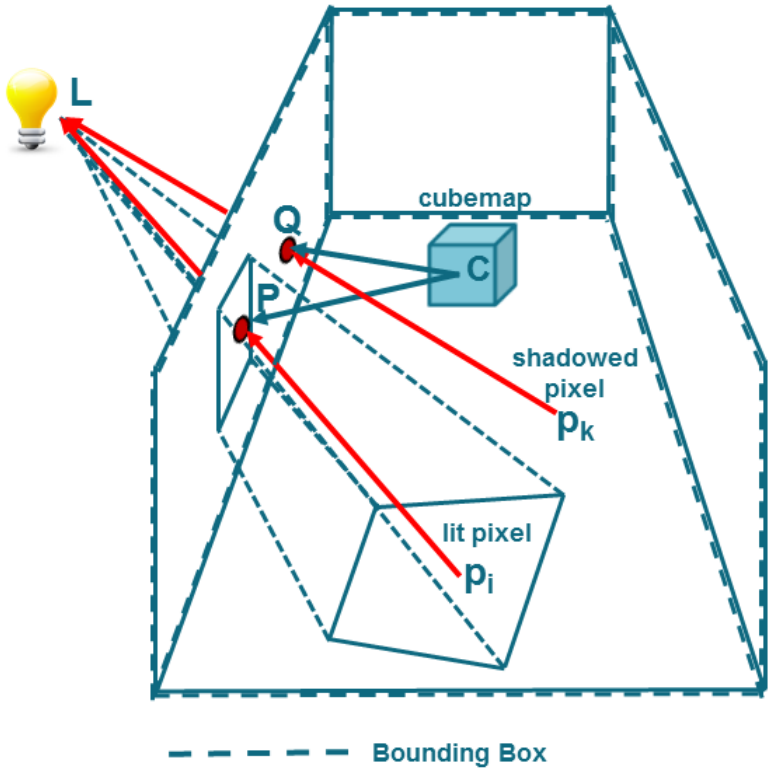


図 9-32 フラグメントからライトへのベクトルの局所補正

次のサンプル コードは、正しい CP ベクトルを計算する方法を示しています。

```
// ワールド座標系を使う。
vec3 intersectMaxPointPlanes = (_BBoxMax - Pi) / PiL;
vec3 intersectMinPointPlanes = (_BBoxMin - Pi) / PiL;

// 光線の前方の交差のみを探す。
vec3 largestRayParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);

// 光線パラメータの最小値が交差を示す。
float dist = min(min(largestRayParams.x, largestRayParams.y), largestRayParams.z);

// 交点の位置を求める。
vec3 intersectPositionWS = Pi + PiL * dist;
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
// 局所補正されたベクトルを取得する。
CP = intersectPositionWS - _EnviCubeMapPos;
```

キューブマップからテクセルをフェッチするには、CP ベクトルを使用します。テクセルのアルファ チャネルは、フラグメントに適用する必要があるライトまたは影の量に関する情報を提供します。

```
float shadow = texCUBE(cubemap, CP).a;
```

次の図は、ハード シャドウのあるチェス ルームを示しています。

図 9-33 ハード シャドウのあるチェス ルーム

この手法では、シーンに作業シャドウが生成されますが、シャドウの品質をさらに 2 つのステップで向上させることができます。

- シャドウの背面
- 滑らかさ

シャドウの背面

キューブマップ シャドウの手法では、影の適用にデプス情報を使用しません。これは、一部の面では、影になる場合には、正しく照らされないことを意味します。

この問題は、サーフェスが光の反対方向を向いている場合にのみ発生します。この問題を解決するには、法線ベクトルとフラグメントからライトへのベクトル P_iL の間の角度を調べます。この角度が -90 ー 90 度の範囲から外れる場合、サーフェスは影になります。

次のコード スニペットはこのチェックを行います。

```
if (dot(PiL,N) < 0)
shadow = 0.0;
```

前のコードにより、各三角形は明るい部分から陰へのハード スイッチとなります。スムーズな遷移を行うには、次の式を使用します。

```
shadow *= max(dot(PiL, N), 0.0);
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

各項目は次のとおりです。

- `shadow` は、シャドウ キューブマップからフェッチされたアルファ値です。
- P_iL は、ワールド空間における、フラグメントからライトへの正規化ベクトルです。
- `N` は、ワールド空間におけるサーフェスの法線ベクトルです。

次の図は、影の背面のあるチェス ルームを示しています。

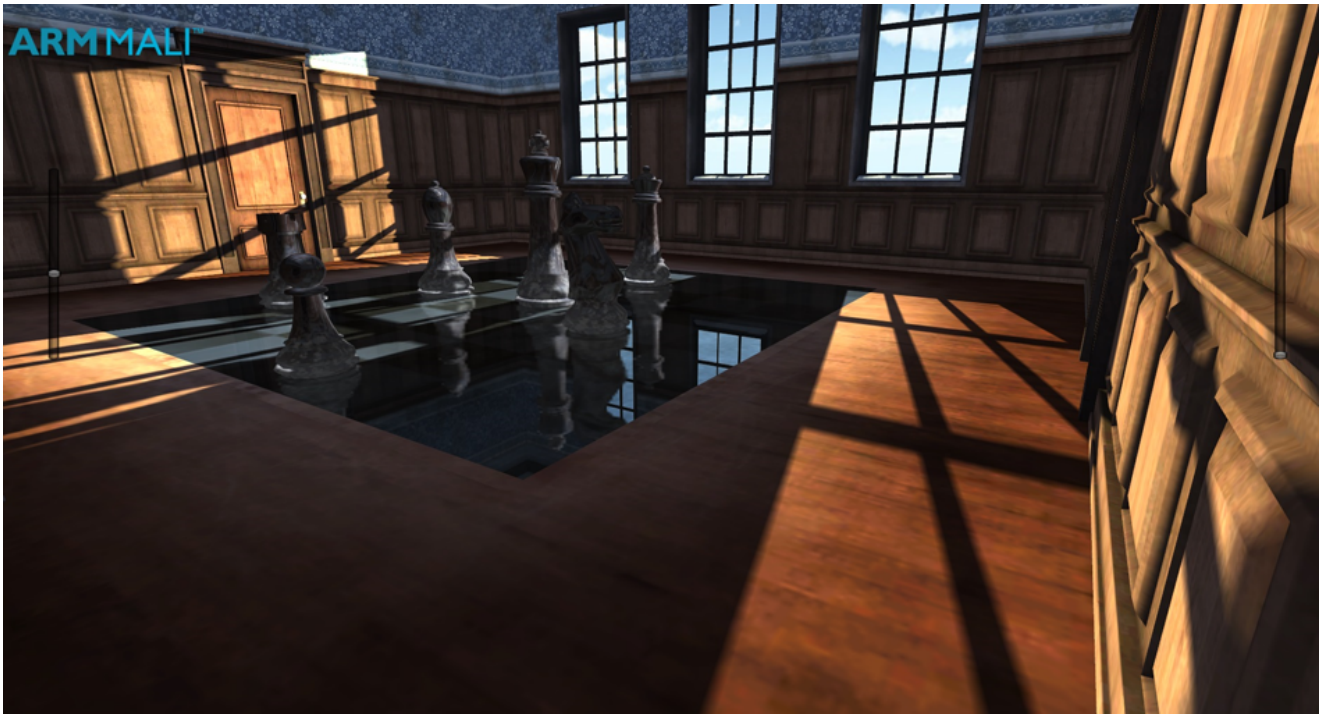


図 9-34 影の背面のあるチェス ルーム

滑らかさ

この影の手法は、シーンに臨場感のあるソフト シャドウを提供できます。

1. ミップマップを生成し、キューブマップ テクスチャに対して三線形フィルタリングを設定します。
2. フラグメントから交点へのベクトルの長さを測定します。
3. 長さに係数を乗じます。

係数は、環境内のミップマップ レベルの数に対する最大距離の正規化です。境界ボリュームとミップマップ レベルに対して、これを自動的に計算することができます。シーンに合わせて、係数をカスタマイズする必要があります。これにより、環境に合わせて設定を微調整し、画質を向上させることができます。たとえば、Ice Cave プロジェクトで使用している係数は `0.08` です。

局所補正に行った計算結果を再利用することができます。局所補正のコード スニペットの `dist` を、フラグメントの位置から、フラグメントからライトへのベクトルと境界ボックスとの交点までの、線分の長さとして再利用します。

```
float texLod = dist;
```

`texLod` に距離係数を乗じます。

```
texLod *= distanceCoefficient;
```

柔らかさを実装するには、Cg の関数 `texCUBElod()` または GLSL の関数 `textureLod()` を使用して、テクスチャの正しいミップマップ レベルを取得します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

vec4 を作成します。ここで、XYZ は方向ベクトルを表し、W 成分は LOD を表します。

```
CP.w = texLod;
shadow = texCUBElod(cubemap, CP).a;
```

この手法は、シーンに高品質で滑らかな影を提供します。

次の図は、滑らかな影のあるチェス ルームを示しています。

図 9–35 滑らかな影

9.4.4 キューブマップ シャドウとシャドウ マップの組み合わせ

動的なコンテンツで影を完成させるには、キューブマップ シャドウと従来のシャドウ マップ手法を組み合わせる必要があります。これには多くの作業が必要となりますが、動的オブジェクトをシャドウ マップにレンダリングするだけで済むため、その価値があります。

次の図は、滑らかな影のみのあるチェス ルームを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9–36 滑らかな影

次の図は、滑らかな影と動的な影を組み合わせたチェス ルームを示しています。

図 9–37 動的な影と組み合わせた滑らかな影

9.4.5 キューブマップ シャドウ手法の結果

従来の手法では、レンダリング シャドウは、影を投影するそれぞれの光源の視点から、シーン全体をレンダリングする必要があるため、コストがかなり高くなる場合があります。ここで説明するキューブマップ シャドウの手法では、ほとんどがプリベイクされるため、パフォーマンスが向上します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

また、この手法は出力解像度に非依存です。1080p、720p、および他の解像度で、同じビジュアル品質を実現します。

ソフトネス フィルタリングはハードウェアで計算されるので、滑らかさの計算コストはほとんどありません。この手法は、影が滑らかなほど、より効率的になります。それは、ミップマップ レベルが小さいほど、従来のシャドウ マップ手法と比べて、データが少なくなるためです。従来の手法では、影を視覚的に魅力的なものにするために、大きなカーネルが必要となります。そのためには高いメモリ帯域幅が必要となり、パフォーマンスが低下します。

キューブマップ シャドウの手法では、想像以上に優れた品質が得られます。臨場感のある柔らかさを備え、エッジに揺らぎのない、安定した影を得ることができます。エッジの揺らぎは、従来のシャドウ マップ手法の使用時に、ラスタライズやエイリアシングのために発生する場合があります。アンチエイリアシング アルゴリズムでは、この問題を完全に解決することはできません。

キューブマップ シャドウ手法では、エッジの揺らぎの問題は生じません。レンダリング ターゲットで使用する解像度よりもはるかに低い解像度を使用しても、エッジは安定しています。出力よりも 4 倍低い解像度を使用することができ、アーティファクトや望ましくない揺らぎはありません。また 4 倍低い解像度を使用すると、メモリ帯域幅を節約でき、パフォーマンスが向上します。

この手法は、OpenGL ES 2.0 以上のシェーダをサポートする、すべての市販のデバイスで使用できます。ローカル キューブマップ手法による反射を使用する場所とタイミングがわかっている場合は、この手法を簡単に実装できます。

注

この手法は、シーンのすべてのものに使用することはできません。たとえば、動的オブジェクトはキューブマップから影を受け取りますが、キューブマップのテクスチャにプリベイクすることはできません。動的オブジェクトでは、シャドウ マップを使用して影を生成し、キューブマップの影の手法とブレンドします。

次の図は、影のある Ice Cave を示しています。

図 9-38 影のある Ice Cave

次の図は、滑らかな影のある Ice Cave を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9-39 滑らかな影のある Ice Cave

次の図は、滑らかな影のある Ice Cave を示しています。

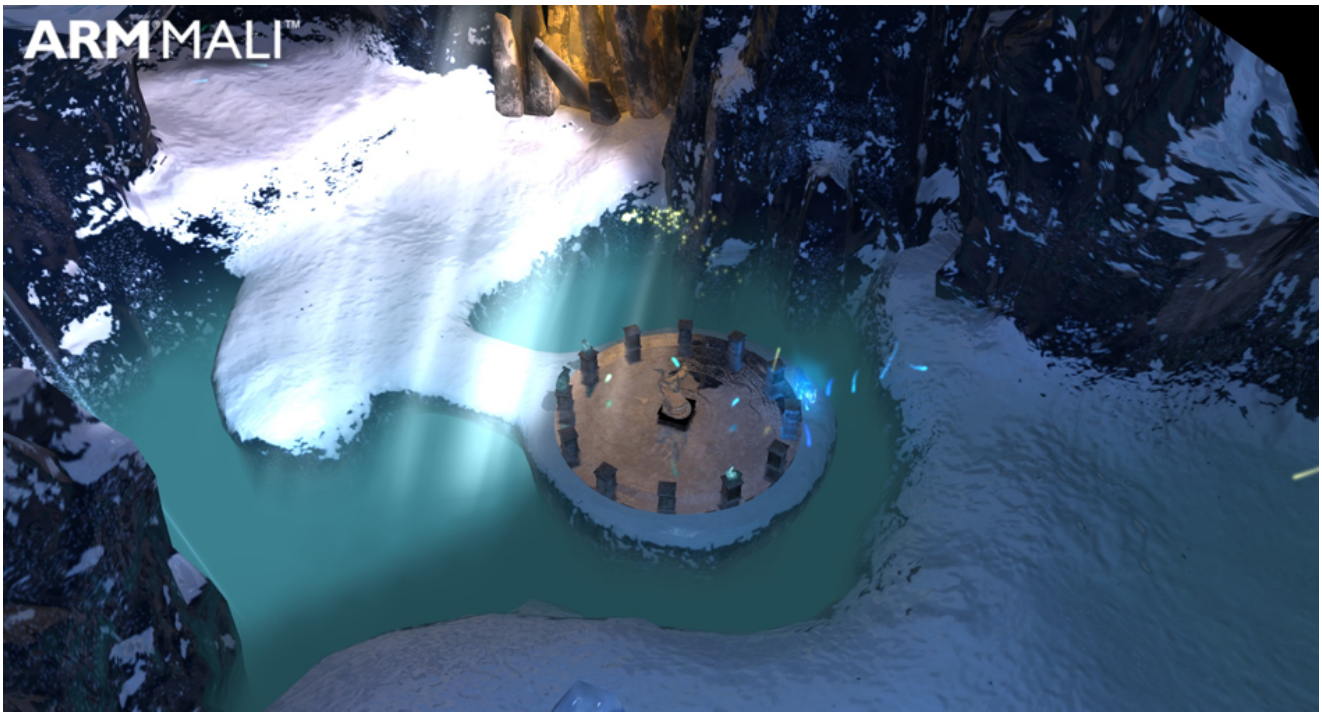


図 9-40 滑らかな影のある Ice Cave

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.5 ローカル キューブマップに基づく屈折

ローカル キューブマップを使用して、高品質の屈折を実装することができます。実行時にこれらを反射と組み合わせることができます。

以下のサブセクションから構成されています。

- *9.5.1 屈折について*(9-188 ページ).
- *9.5.2 屈折の実装*(9-188 ページ).
- *9.5.3 ローカル キューブマップに基づく屈折について*(9-189 ページ).
- *9.5.4 キューブマップの準備*(9-189 ページ).
- *9.5.5 シェーダの実装*(9-191 ページ).

9.5.1 屈折について

ゲーム開発者は常に、ゲームで見た目に印象的なエフェクトを実現する、効率的な方法を求めています。モバイル プラットフォームをターゲットにする場合には、パフォーマンスの最大化とリソースのバランスを慎重に取る必要があるため、特に重要です。

屈折は、光波が通過する媒体の変化により、その方向が変化する現象です。半透明のジオメトリにおける高いリアリズムを求める場合、屈折は考慮すべき重要なエフェクトです。

光がマテリアルに進入するときにどれほど曲がるか、つまり屈折するかは屈折率で決まります。屈折の定義は、光が屈折率 n_1 の媒体から屈折率 n_2 の媒体に通り抜けるときに曲がることです。

スネルの法則を使って、入射角および屈折角の正弦と屈折係数の正弦の関係を計算します。

以下の図に、スネルの法則と屈折の関係を示します。

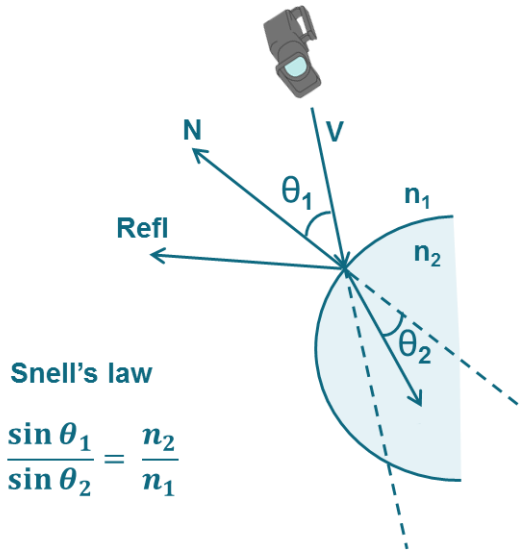


図 9-41 ある媒体から別の媒体へ通過する光の屈折

9.5.2 屈折の実装

開発者は、反射のレンダリングを始めたときから、屈折のレンダリングを試みてきました。これらのプロセスは、半透明の表面で同時に発生するためです。反射のレンダリングを行うにはいくつかの技法がありますが、屈折の技法は多くはありません。

屈折を実行時に実装する既存の方法は、屈折の特定の種類によって異なります。ほとんどの技法は、実行時に屈折オブジェクトの背後のシーンをテクスチャにレンダリングし、第 2 パスでテクスチャの変形を適用して、屈折の外観を実現します。テクスチャの変形に応じて、このアプローチを使用して、水、熱曇り、ガラス、その他のエフェクトなどの、屈折エフェクトをレンダリングできます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

これらの技法の中には良い結果が得られるものもありますが、テクスチャの変形は物理的なものではないため、結果が常に正しいとは限りません。たとえば、屈折カメラの視点からテクスチャをレンダリングすると、カメラには直接映らないが、物理的な屈折のため、実際には見えるエリアがある可能性があります。

テクスチャを使ったレンダリングの技法を使う場合の主な制限事項は品質です。カメラを動かすと多くの場合、ピクセルの揺らぎやピクセルの不安定性が目につきます。

9.5.3 ローカル キューブマップに基づく屈折について

ローカル キューブマップは反射のレンダリングのための優れた手法です。開発者はこれまでも、静的キューブマップを使用して、反射と屈折の実装を行ってきています。

しかし、静的キューブマップを使用してローカル環境で反射または屈折を実装する場合は、局所補正を適用しないと結果が正しくありません。

ここに記載された手法では、正しい結果となるように局所補正が適用されています。この手法は高度に最適化されています。ランタイム リソースの制限により、注意深いバランスが必要となる、モバイル デバイスでは特に便利です。

9.5.4 キューブマップの準備

屈折の実装で使用するキューブマップを準備する必要があります。

キューブマップを準備するには、次の手順を実行します。

1. 屈折ジオメトリの中心にカメラを置きます。
2. 屈折オブジェクトを隠し、周囲の静的環境を 6 方向のキューブマップにレンダリングします。このキューブマップを使用して、屈折と反射の両方を実装することができます。
3. 屈折オブジェクトの周囲の環境を静的キューブマップにベイクします。
4. 屈折ベクトルの方向を決定し、それがローカル環境の境界ボックスと交差する場所を見つけます。
5. *9.4 ローカル キューブマップに基づく動的なソフト シャドウ*(9-180 ページ) と同じ方法で局所補正を適用します。
6. キューブマップが生成された位置から交点までの新しいベクトルを作成します。この最終的なベクトルを使用して、キューブマップからテクセルを取り出し、屈折オブジェクトの背後にあるものをレンダリングします。

屈折ベクトル R_{rf} を使用してキューブマップからテクセルをフェッチする代わりに、屈折ベクトルと境界ボックスの交点 P を見つけ、キューブマップ C の中心から交点 P までの新しいベクトル R'_{rf} を作成します。この新しいベクトルを使用して、キューブマップからテクスチャ カラーをフェッチします。

```
float eta=n2/n1;
```

float3R_{rf} = refract(D,N,eta);

交点 P を見つける

ベクトルを見つける R'$_{rf}$ = CP;

Float4 col = texCube(Cubemap, R'$_{rf}$);

次の図は、キューブマップと屈折ベクトルを持つシーンを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

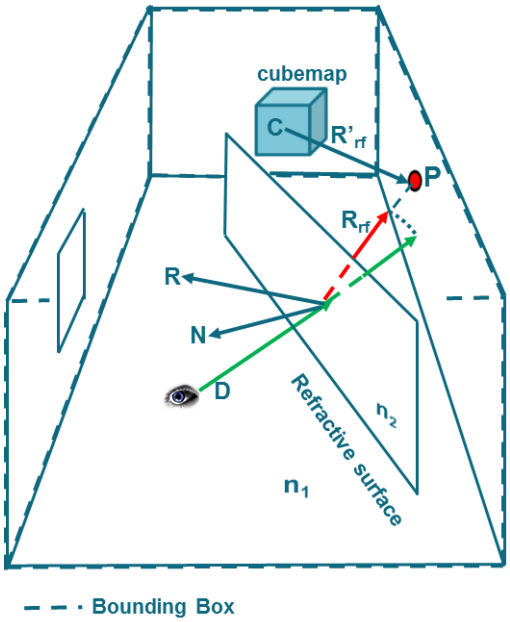


図 9-42 屈折ベクトルの局所補正

スネルの法則を用いて屈折ベクトルの方向を計算しているため、この手法で生成される屈折は物理法則に正確に基づいています。

スネルの法則に厳密に従って、屈折ベクトル `R` を見つけるためにシェーダで使用できる、組み込み関数もあります。

```
R = refract( I, N, eta);
```

各項目は次のとおりです。

- `I` は正規化されたビューまたはインシデント ベクトルです。
- `N` は正規化された法線ベクトルです。
- `eta` は、屈折率の比 n_1/n_2 です。

次の図は、ローカル キューブマップに基づいて屈折を実装する、シェーダのフローを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

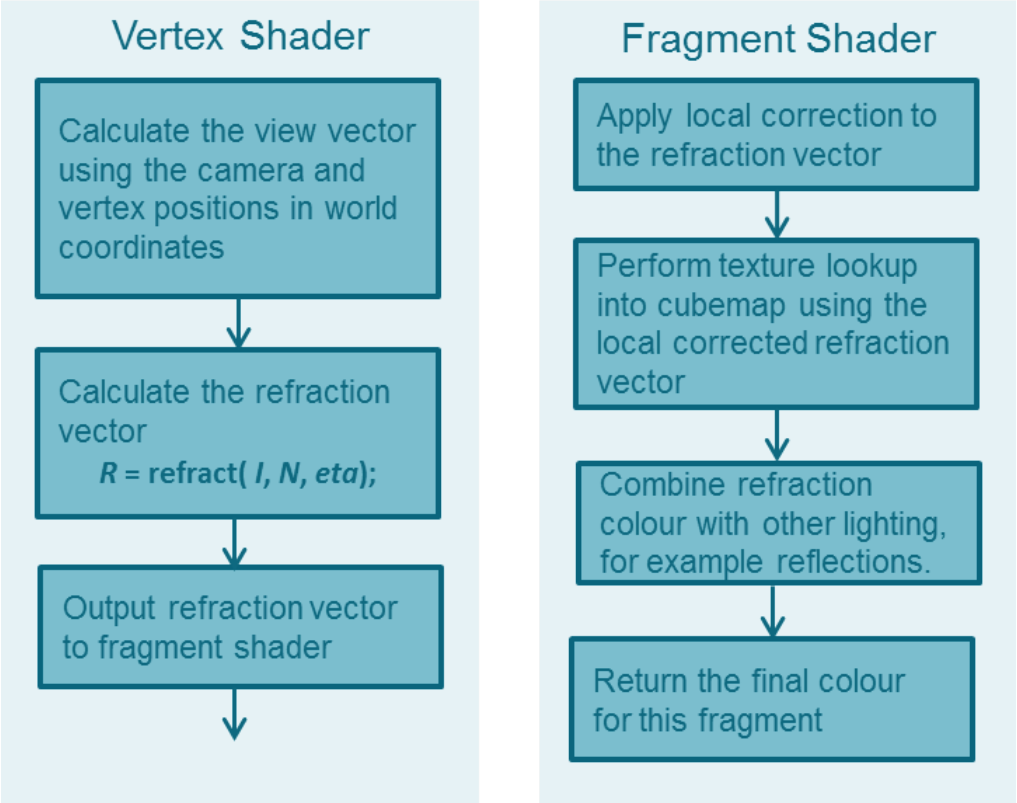


図 9-43 ローカル キューブマップに基づく屈折のシェーダの実装

9.5.5 シェーダの実装

局所補正された屈折の方向に対応するテクセルをフェッチするとき、屈折のカラーを他のライティングと組み合わせることができます。たとえば、屈折と同時に生じる反射を考慮できます。

屈折のカラーを他のライティングと組み合わせるには、追加のビュー ベクトルをフラグメント シェーダに渡し、それに局所補正を適用する必要があります。結果を使用して、同じキューブマップから反射のカラーを取得します。

次のコード スニペットは、反射と屈折を組み合わせて、最終的な出力カラーを生成する方法を示しています。

```
// ----------------- 環境の反射 ----------------
float3 newReflDirWS = LocalCorrect(input.reflDirWS, _BBoxMin, _BBoxMax, input.posWorld, _EnviCubeMapPos);
float4 staticReflColor = texCUBE(_EnviCubeMap, newReflDirWS);
// ----------------- 環境の屈折 ----------------
float3 newRefractDirWS = LocalCorrect(RefractDirWS, _BBoxMin, _BBoxMax, input.posWorld, _EnviCubeMapPos);
float4 staticRefractColor = texCUBE(_EnviCubeMap, newRefractDirWS);
// ----------------- 反射と屈折の組み合わせ ----------------
float4 combinedReflRefract = lerp(staticReflColor, staticRefractColor, _ReflAmount);

float4 finalColor = _AmbientColor + combinedReflRefract;
```

`_ReflAmount` 係数は、フラグメント シェーダにユニフォームとして渡されます。この係数を使用して、反射と屈折の影響のバランスを調整します。`_ReflAmount` を手動で調整して、必要な視覚的効果を得ることができます。

`LocalCorrect` 関数の実装については、反射のブログ *http://community.arm.com/groups/arm-mali-graphics/blog/2014/08/07/reflections-based-on-local-cubemaps* を参照してください。

屈折ジオメトリが中空の物体である場合、屈折と反射はサーフェスの前面および背面の両方で生じます。

次の図は、ローカル キューブマップに基づくビショップ ガラスの屈折を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

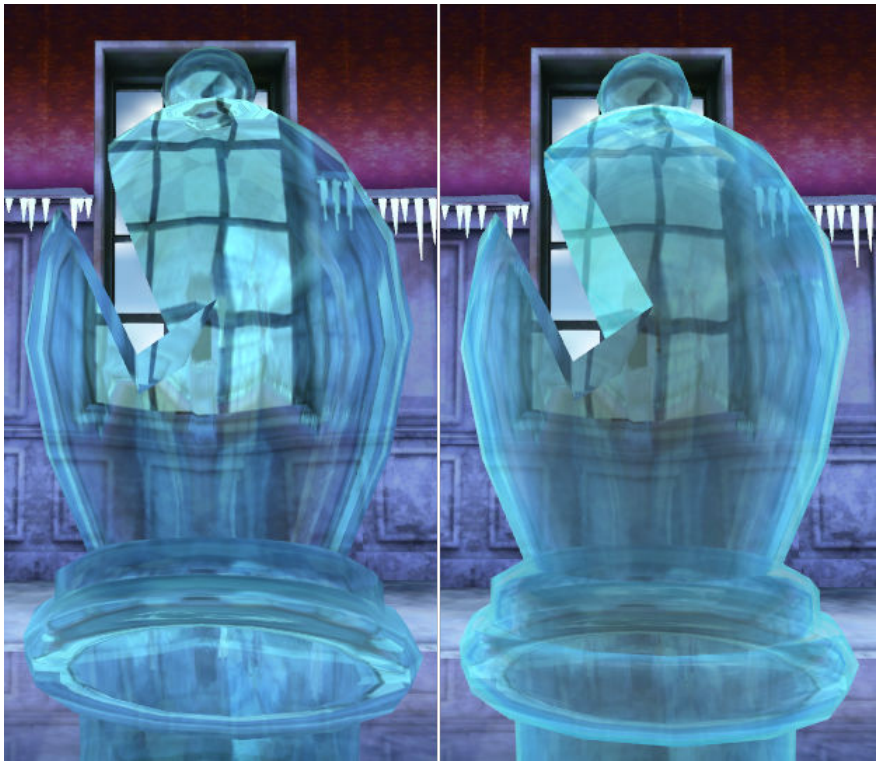


図 9-44 ローカル キューブマップに基づくビショップ ガラスの屈折

左の画像は、ローカルの屈折と反射のある背面のみをレンダリングする、第 1 パスを示しています。

右の画像は、第 2 パスがローカルの屈折と反射のある前面のみをレンダリングし、第 1 パスとアルファブレンドすることを示しています。

- 第 1 パスでは、不透明なジオメトリをレンダリングするのと同じ方法で、半透明オブジェクトをレンダリングします。最後にオブジェクトを前面のカリングを使ってレンダリングします。つまり、背面のみをレンダリングします。他のオブジェクトを閉塞しないようにするため、デプス バッファに書き込まないようにします。

  背面のカラーは、オブジェクト自体の反射、屈折、拡散のカラーから計算されたカラーを混合して得られます。
- 第 2 パスでは、背面のカリングで前面をレンダリングします。これをレンダリング キューで最後に行います。デプスの書き込みがオフであることを確認します。屈折と反射のテクスチャを拡散カラーと混合して前面のカラーを取得します。第 2 パスでの屈折により、最終的なレンダリングに臨場感を与えます。背面の屈折がエフェクトのために十分であれば、このステップをスキップできます。
- 最後のパスでは、結果のカラーを第 1 パスとアルファ ブレンドします。

次の図は、Ice Cave デモの半透明のフェニックスに、ローカル キューブマップに基づいて屈折を実装した結果を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

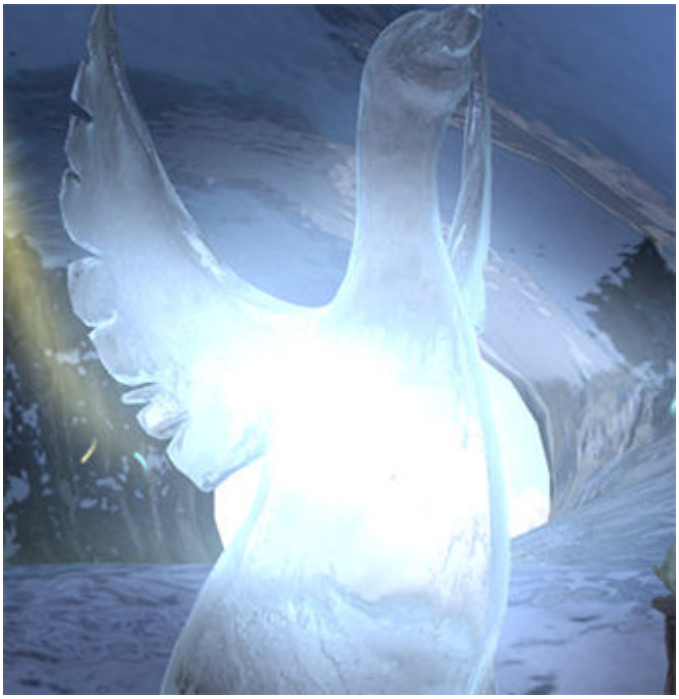


図 9–45 半透明のフェニックスの屈折

次の図は、半透明のフェニックスの羽を示しています。

図 9–46 半透明のフェニックスの羽

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.6 Ice Cave デモの鏡面反射エフェクト

Ice Cave デモの鏡面反射エフェクトは、Blinn 手法を使用して実装されています。これは良い結果を生成できる非常に効率的な手法です。

次のコードは、Blinn 手法で鏡面反射エフェクトを実装する方法を示しています。

```
// 影を考慮しない鏡面反射エフェクトの強度を返す
float SpecularBlinn(float3 vert2Light, float3 viewDir, float3 normalVec, float4 power)
{
    float3 floatDir = normalize(vert2Light - viewDir);
    float specAngle = max(dot(floatDir, normalVec), 0.0);
    return pow(specAngle, power);
}
```

Blinn 手法のデメリットは、特定の状況では正しくない結果を生成する場合があることです。たとえば、鏡面反射エフェクトが影の領域に現れることがあります。

次の図は、影の領域で鏡面反射エフェクトのないサンプルを示しています。

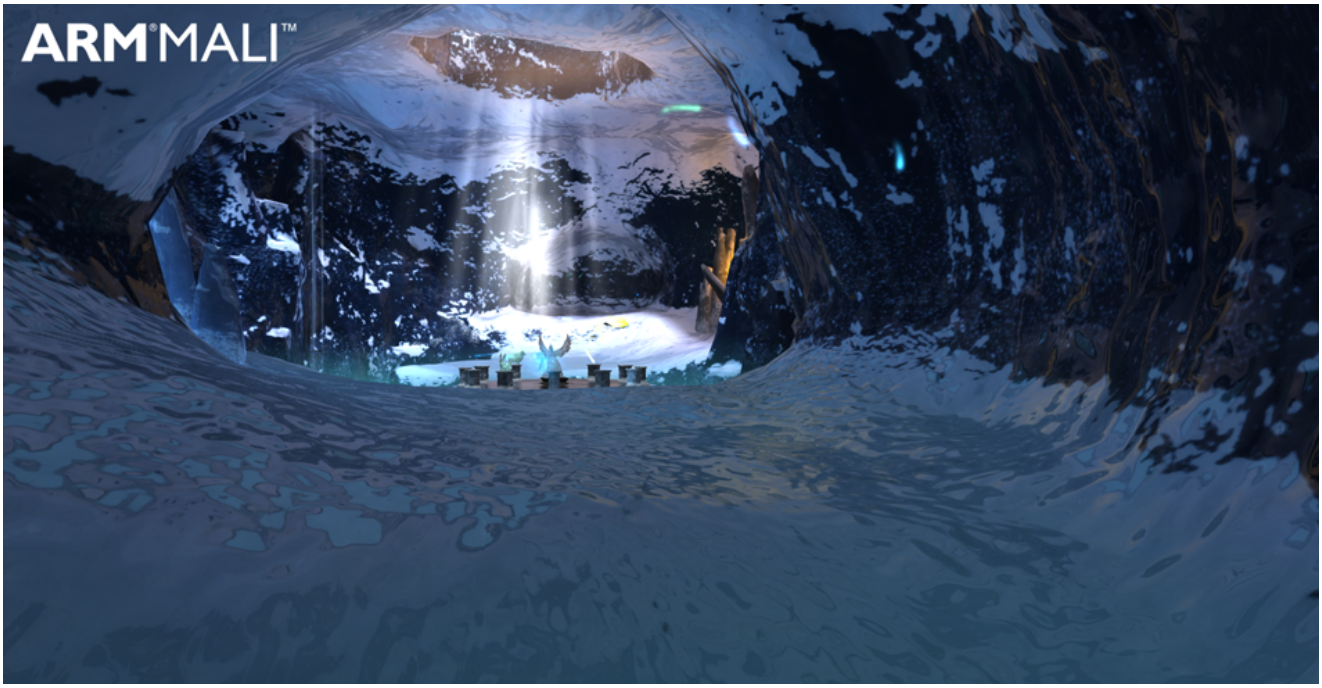


図 9–47 影の領域

次の図は、影の領域で鏡面反射エフェクトのあるサンプルを示しています。これは正しくありません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

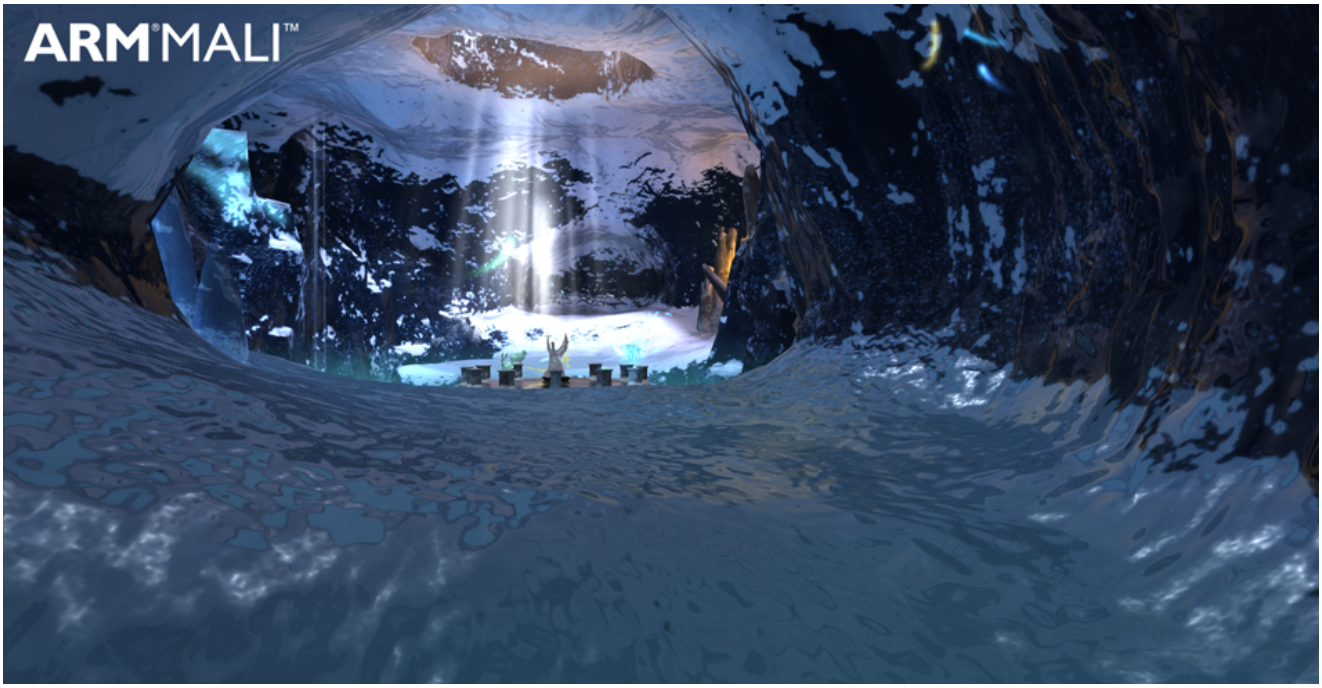


図 9-48 正しくない鏡面反射エフェクトのある影の領域

次の図は、照らされた領域で鏡面反射エフェクトのあるサンプルを示しています。これは正しいものです。

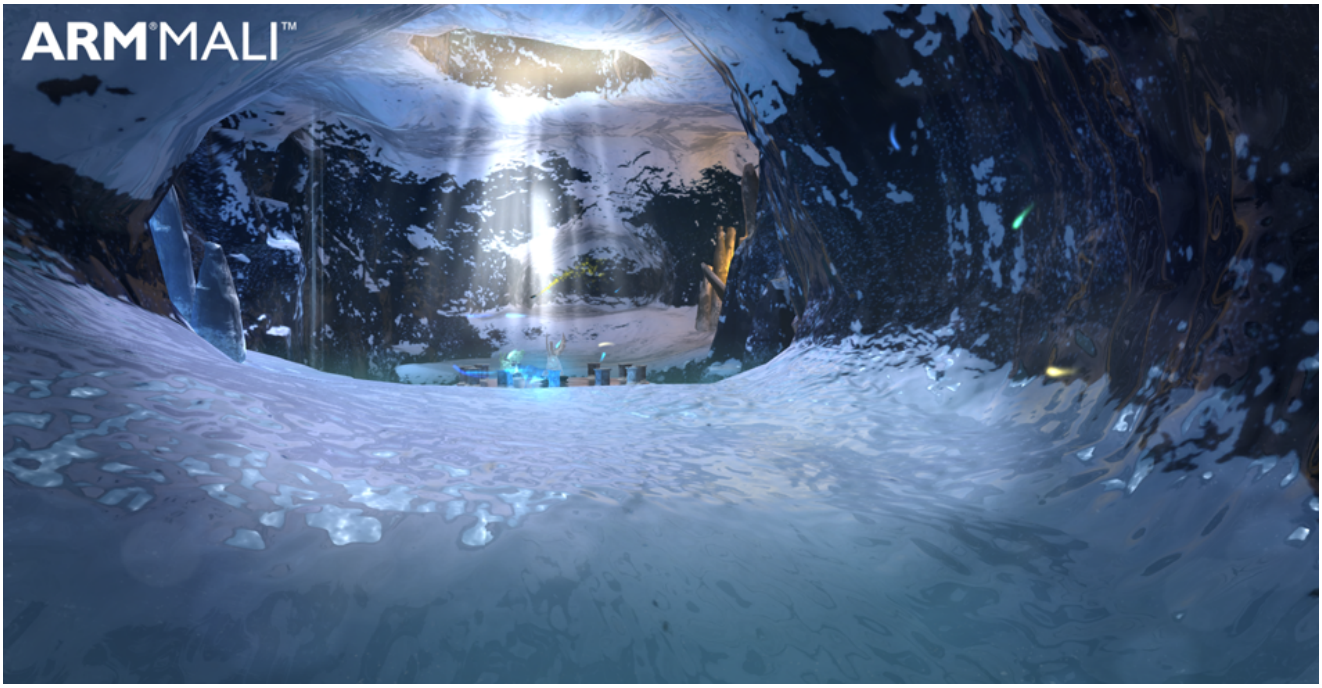


図 9-49 正しい鏡面反射エフェクトで照らされた領域

鏡面反射サーフェスに届く光に応じて、影は鏡面反射エフェクトの強度を強くまたは弱くする必要があります。鏡面反射エフェクトの強度を補正するために必要なすべての情報は、既に Ice Cave デモに含まれているため、修正は比較的容易です。反射と影のエフェクトに使用される、環境のキューブマップテクスチャには、2 種類の情報が含まれています。RGB チャネルには、反射に使用される環境のカラーが含まれています。アルファ チャネルには不透明度が含まれ、これは影に使用されます。アルファ チャネルは環境に光を差し込む洞くつの穴を表しているため、アルファ チャネルを使用して鏡面反射の

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

強度を決定することができます。このため、アルファ チャネルは、照らされているサーフェスにのみに鏡面反射エフェクトが適用されるようにするために使用されます。

これを行うには、フラグメント シェーダで補正された反射ベクトルを計算して、反射エフェクトを作成します。補正された反射ベクトルの作成の詳細については、*9.2 ローカル キューブマップによる反射の実装*(9-157 ページ) を参照してください。このベクトルを使用して、キューブマップ テクスチャから RGBA テクセルをフェッチします。

```
// 局所補正された静的反射
const half4 reflColor = SampleCubemapWithLocalCorrection(
        ReflDirectionWS,
        _ReflBBoxMinWorld,
        _ReflBBoxMaxWorld,
        input.vertexInWorld,
        _ReflCubePosWorld,
        _ReflCube);
```

`SampleCubemapWithLocalCorrection()` 関数の詳細については、*9.2 ローカル キューブマップによる反射の実装*(9-157 ページ) を参照してください。

`reflColor` は RGBA 形式で、RGB コンポーネントには反射のためのカラー データが含まれ、アルファ チャネルには鏡面反射エフェクトの強度が含まれます。Ice Cave デモでは、アルファ チャネルに Blinn 手法で計算された鏡面反射のカラーが乗算されます。

```
half3 specular = _SpecularColor.rgb *
        SpecularBlinn(
            input.vertexToLight01InWorld,
            viewDirInWorld,
            normalInWorld,
            _SpecularPower) *
        reflColor.a;
```

`specular` 値は、最終的な鏡面反射のカラーを表します。これをライティング モデルに追加できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.7 Early-Z を使用する

Mali GPU では、Early-Z アルゴリズムを実行することができます。Early-Z は、フラグメントのオーバードローをなくし、パフォーマンスを向上させます。

Mali GPU は通常、ほとんどのコンテンツで Early-Z アルゴリズムを実行しますが、正確さを保持するために、実行されない場合もあります。これは、Unity エンジンとコンパイラによって生成されたコードの両方に依存するため、Unity 内から制御するのは困難ですが、確認できるいくつかの兆候があります。

シェーダをモバイル向けにコンパイルして、コードを確認します。シェーダが次のカテゴリのいずれかに該当しないことを確認します。

シェーダに副作用がある

シェーダ スレッドが実行中にグローバルなステータスを変更するため、シェーダを 2 回実行すると、異なる結果が生じる場合があります。これは通常、シェーダ ストレージ バッファ オブジェクトやイメージなど、共有の読み書きメモリ バッファに、シェーダが書き込むことを意味します。たとえば、パフォーマンスを測定するために、カウンタをインクリメントするシェーダを作成する場合、このシェーダには副作用があります。

以下のものは、副作用とは分類されません。

- 読み出し専用のメモリアクセス。
- 書き込み専用バッファへの書き込み。
- 純粋なローカル メモリアクセス。

シェーダが discard() を呼び出す

フラグメント シェーダがその実行中に `discard()` を呼び出すことができる場合、Mali GPU は Early-Z を有効化できません。これは、フラグメント シェーダは現在のフラグメントを破棄することはできますが、デプスの値が Early-Z テストによって以前に変更されていたため、これを元に戻すことができないためです。

Alpha-to-coverage が有効化されている

Alpha-to-coverage が有効化されている場合、フラグメント シェーダはデータを計算し、後でそれにアクセスして、アルファを定義します。

たとえば、樹木の葉をレンダリングするとき、それらは通常、平面として表され、テクスチャは葉のどの領域が透明か不透明かを定義します。Early-Z が有効化されている場合、シーンの一部が平面の透明部分で閉塞されることがあるため、結果が正しくなりません。

デプス ソースが固定されていない関数

デプス テストに使用されるデプス値は、頂点シェーダから得られるものではありません。フラグメント シェーダが `gl_FragDepth` に書き込む場合、Mali GPU は Early-Z テストを実行できません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.8 ダーティ レンズ エフェクト

ダーティ レンズ エフェクトを使って、ドラマのような感覚を演出できます。多くの場合、レンズのフレア エフェクトと一緒に使用されます。

モバイル デバイスに適した、非常に軽量で簡単な方法で、ダーティ レンズ エフェクトを実装できます。

Ice Cave デモでは、ダーティ レンズ エフェクトが最小限のシェーダで実装され、シーンの上にフルスクリーンで可変の強度のクワッドをレンダリングします。クワッドの強度は、スクリプトによって渡されます。

シェーダは、すべての透明なジオメトリがレンダリングされた後で、最後に追加アルファ ブレンディングを使用して、クワッドをレンダリングします。

以下のサブセクションから構成されています。

- *9.8.1 シェーダの実装*(9-198 ページ).
- *9.8.2 スクリプトの実装*(9-200 ページ).
- *9.8.3 ダーティ レンズ シェーダのコード*(9-200 ページ).

9.8.1 シェーダの実装

このセクションでは、ダーティ レンズ シェーダについて説明します。

ダーティ レンズ シェーダの完全なソースコードについては、*9.8.3 ダーティ レンズ シェーダのコード* (9-200 ページ) を参照してください。

次のサブシェーダ タグは、フルスクリーン クワッドに対して、すべての不透明なジオメトリおよび他の 9 つの透明なオブジェクトの後でレンダリングされるように指示します。

```
Tags {"Queue" = "Transparent+10"}
```

シェーダは、次のコマンドを使用してデプス バッファの書き込みを無効にします。これにより、クワッドがその背後にあるジオメトリを閉塞するのを防ぎます。

```
ZWrite Off
```

ブレンディング段階で、フラグメント シェーダの出力は、既にフレーム バッファにあるピクセル カラーとブレンドされます。

シェーダは、追加のブレンド タイプ `Blend One One` を指定します。このブレンド タイプでは、ソース ファクタおよびデスティネーション ファクタは両方とも `float4 (1.0, 1.0, 1.0, 1.0)` です。

このタイプのブレンドは、パーティクル システムが、透明で発光する、火のようなエフェクトを表す場合によく使用されます。

シェーダはカリングと ZTest を無効にして、ダーティ レンズ エフェクトが常にレンダリングされるようにします。

クワッドの頂点はビューポート座標で定義されているため、頂点シェーダでは頂点変換は行われません。

フラグメント シェーダはテクスチャからテクセルのみをフェッチし、エフェクトの強度に比例する係数を適用します。

次の図は、ダーティ レンズ エフェクトのフルスクリーン クワッド テクスチャとして使用される画像を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9-50 Ice Cave デモのダーティ レンズ エフェクトで使用されているテクスチャ

次の図は、Ice Cave デモで、カメラが洞くつの入り口からの太陽光に向いているときに、ダーティ レンズ エフェクトが表示される様子を示しています。

図 9-51 Ice Cave デモで実装されたダーティ レンズ エフェクト

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.8.2 スクリプトの実装

簡単なスクリプトにより、フルスクリーン クワッドを作成して、フラグメント シェーダに渡される強度係数を計算します。

次の関数は、Start 関数にクワッドのメッシュを作成します。

```
void CreateQuadMesh()
{
    Mesh mesh = GetComponent<MeshFilter>().mesh;
    mesh.Clear();
    mesh.vertices = new Vector3[] {new Vector3(-1, -1, 0), new Vector3(1, -1, 0),
                new Vector3(1, 1, 0), new Vector3(-1, 1, 0)};
    mesh.uv = new Vector2[] {new Vector2(0, 0), new Vector2(1, 0),
            new Vector2(1, 1), new Vector2(0, 1)};
    mesh.triangles = new int[] {0, 2, 1, 0, 3, 2};
    mesh.RecalculateNormals();

    //bounds を増加させ、錐台が切り取られないようにする。
    bigBounds.SetMinMax(new Vector3(-100, -100, -100), new Vector3(100, 100, 100));
    mesh.bounds = bigBounds;
}
```

メッシュが作成されると、境界線はインクリメントされ、錐台が切り取られることはありません。シーンのサイズに応じて、境界のサイズを設定します。

スクリプトにより、強度係数を計算し、それをフラグメント シェーダに渡します。これは、カメラから太陽へのベクトルと、カメラの前方ベクトルの相対的な向きに基づいています。エフェクトの最大強度は、カメラが太陽を直視しているときに発生します。

次のコードは、強度係数の計算を示します。

```
Vector3 cameraSunVec = sun.transform.position - Camera.main.transform.position;
cameraSunVec.Normalize();
float dotProd = Vector3.Dot(Camera.main.transform.forward, cameraSunVec);
float intensityFactor = Mathf.Clamp(dotProd, 0.0f, 1.0f);
```

9.8.3 ダーティ レンズ シェーダのコード

これは DirtyLensEffect.shader のコードです。

```
half3 normalInWorld = half3(0.0,0.0,0.0);
half3 bumpNormal = UnpackNormal(tex2D(_BumpMapGlobal, input.tc));
half3x3 local2WorldTranspose = half3x3(input.tangentWorld,
                                input.bitangentWorld,
                                input.normalInWorld);
normalInWorld = normalize(mul(bumpNormal, local2WorldTranspose));
normalInWorld = normalInWorld*0.5 + 0.5;
return  half4(normalInWorld,1.0);
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.9 ライト シャフト

ライト シャフトは、薄明光線、大気散乱、またはシャドウイングの効果をシミュレートします。それらは、シーンに深みと臨場感を加えるために使用されます。

以下のサブセクションから構成されています。

- *9.9.1 ライト シャフトについて*(9-201 ページ).
- *9.9.2 コーン エッジのフェード アウト*(9-203 ページ).

9.9.1 ライト シャフトについて

Ice Cave デモでは、ライト シャフトは、洞くつの上部の開口部から洞くつに入る太陽光線の散乱をシミュレートします。

次の図は、Ice Cave デモのライト シャフトを示しています。

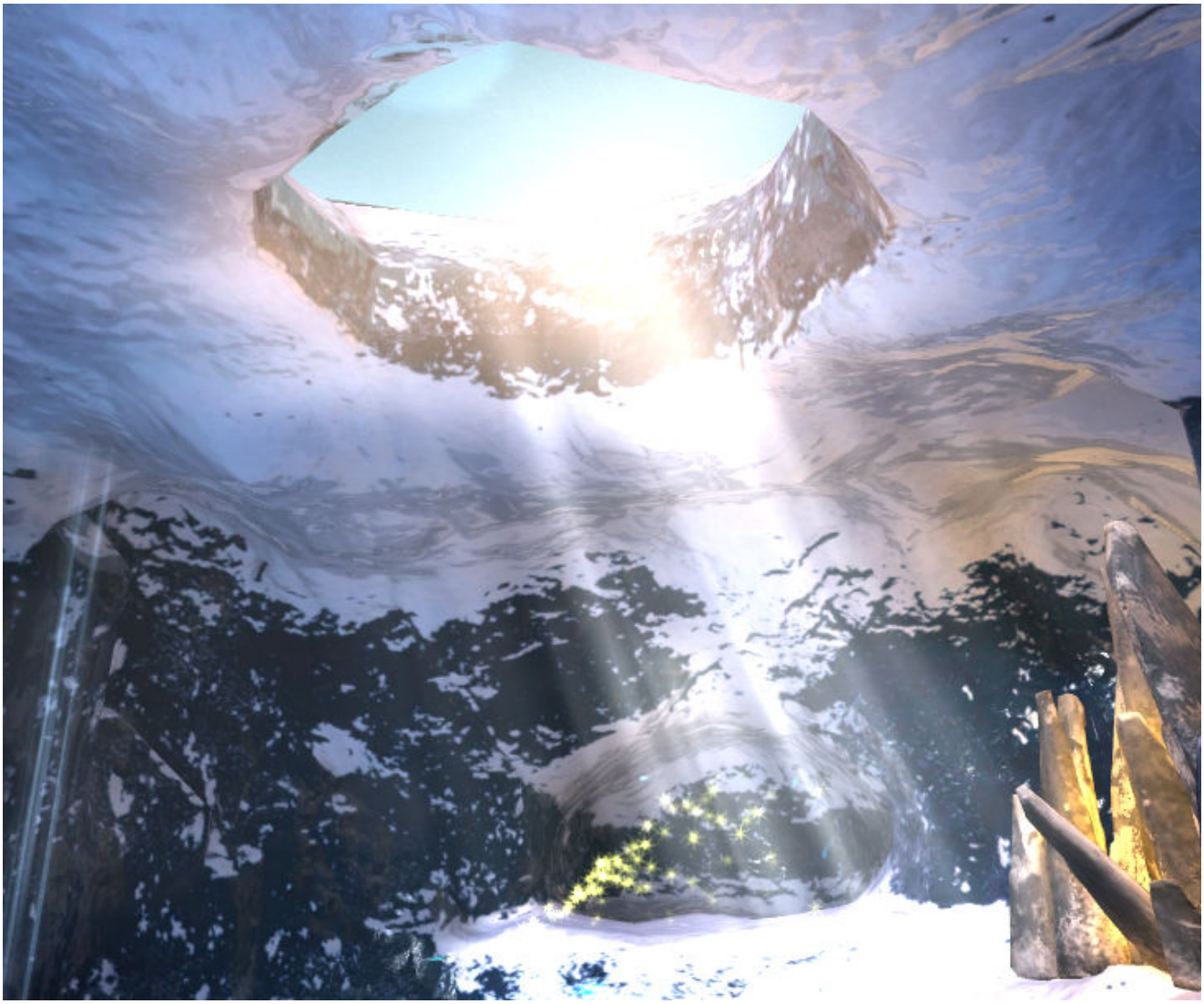


図 9-52 Ice Cave デモのライト シャフト

ライト シャフトは円錐台をベースにしています。円錐台は、その上部が常に固定されながら、光線の向きに従います。

次の図は、円錐台のジオメトリを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

- a は、ライト シャフトの基本的な形状が上下 2 つの断面を持つ円柱であることを示しています。
- b は、下側の断面が、定義した角度 θ に基づいて、円錐台のジオメトリとなるように拡張されていることを示しています。
- c は、上側の断面が固定され、下側の断面が太陽の光線の方向に沿って水平方向にシフトされることを示しています。

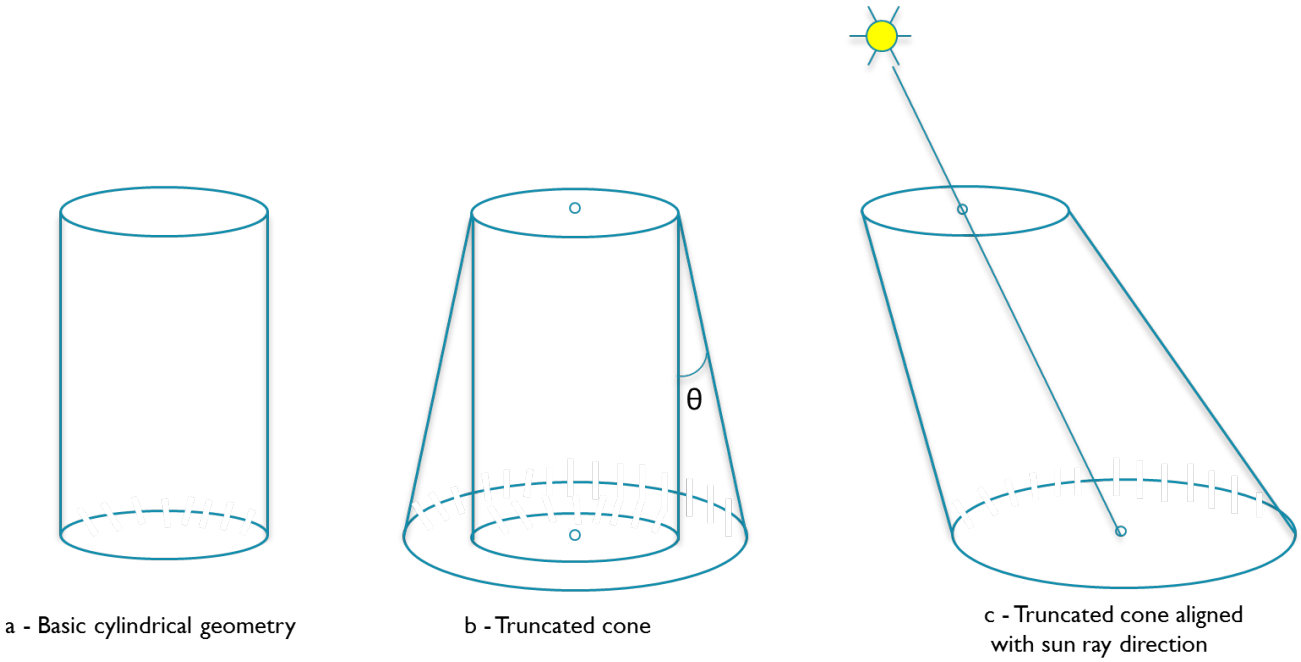


図 9-53 ライト シャフトのジオメトリ

スクリプトは、太陽の位置を使用して、次の値を計算します。

- 入力として提供された角度 θ の値に基づく、円錐台の下側の断面の拡張の大きさ。
- 断面のシフトの方向と大きさ。

頂点シェーダは、このデータに基づいて変換を適用します。変換は、ライト シャフトのローカル座標で、元の円柱ジオメトリの頂点に適用されます。

ライト シャフトをレンダリングするときは、ジオメトリを明示するハード エッジを描画しないようにします。そのためには、上下の断面を滑らかにフェード アウトするテクスチャ マスクを使用します。

次の図は、ライト シャフトのテクスチャを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9–54 ライト シャフトのテクスチャ。左はマスク テクスチャ。右はビーム テクスチャ

9.9.2 コーン エッジのフェード アウト

断面に平行な平面では、カメラから頂点への相対的な向きに応じて、光軸の強度をフェードアウトします。

頂点シェーダで、カメラの位置を断面に投影します。

断面の中心から投影への新しいベクトルを作成し、それを正規化します。

このベクトルのドット積を頂点法線を使って計算し、その結果をべき乗指数にします。

結果を可変変数としてフラグメント シェーダに渡します。これは光軸の強度を調整するために使用されます。

頂点シェーダは、これらの計算を*ローカル座標系* (LCS) で実行します。

次のコードは、頂点シェーダを示しています。

```
// カメラの位置を断面に投影する
float3 axisY = float3(0, 1, 0);
float dotWithYAxis = dot(camPosInLCS, axisY);
float3 projOnCrossSection = camPosInLCS - (axisY * dotWithYAxis);
projOnCrossSection = normalize(projOnCrossSection);

// ドット積により断面のエッジのジオメトリをフェードする
float dotProd = abs(dot(projOnCrossSection, input.normal));
output.overallIntensity = pow(dotProd, _FadingEdgePower) * _CurrLightShaftIntensity;
```

係数 `_FadingEdgePower` を使用すると、光軸端のフェーディングを微調整できます。

このスクリプトは係数 `_CurrLightShaftIntensity` を渡します。これにより、カメラが近づくと光軸がフェード アウトします。

次のコードは、スクリプトからテクスチャをゆっくりとスクロール ダウンして光軸に追加できる、最終的な仕上げを示しています。

```
void Update()
{
    float localOffset = (Time.time * speed ) + offset;
    localOffset = localOffset % 1.0f;
    GetComponent<Renderer>().material.SetTextureOffset("_MainTex", new Vector2(0,
localOffset));
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

フラグメント シェーダは、ビームとマスクのテクスチャをフェッチし、それらを強度係数と結合します。

```
float4 frag(vertexOutput input) : COLOR
{
    float4 textureColor = tex2D(_MainTex, input.tex.xy);
    float textureMask = tex2D(_MaskTex, input.tex.zw).a;
    textureColor *=  input.overallIntensity * textureMask;
    textureColor.rgb = clamp(textureColor.a, 0, 1);
    return textureColor;
}
```

光軸ジオメトリは、すべての不透明なジオメトリの後に、透過キューにレンダリングされます。

フレーム バッファ内の対応するピクセルとフラグメント カラーを結合するために、追加のブレンディングを使用します。

またシェーダは、他のオブジェクトを閉塞しないように、カリングとデプス バッファへの書き込みを無効にします。

パスの設定は次のとおりです。

```
Blend One One
Cull Off
ZWrite Off
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.10 フォグ エフェクト

フォグ エフェクトはシーンに雰囲気を加えます。フォグを生成するためには高度な実装を必要としません。簡単なフォグ エフェクトでも効果的となります。

以下のサブセクションから構成されています。

- *9.10.1 フォグ エフェクトついて*(9-205 ページ).
- *9.10.2 手続き型リニア フォグ*(9-205 ページ).
- *9.10.3 高さのあるリニア フォグ*(9-206 ページ).
- *9.10.4 不均一なフォグ*(9-207 ページ).
- *9.10.5 フォグのプリベーキング*(9-207 ページ).
- *9.10.6 パーティクルを使った 3 次元のフォグ*(9-207 ページ).

9.10.1 フォグ エフェクトついて

現実の世界では、距離が遠くなるほど色が薄くなります。この効果を見るには霧がかかっている必要はなく、晴れた日にも見ることができます。特に山を見るときにはそうなります。

これは現実の世界ではとても一般的であるため、このエフェクトをゲームに加えると臨場感が加わります。

このセクションでは、フォグ エフェクトの 2 つのバージョンについて説明します。

- 手続き型リニア フォグ。
- パーティクル ベースのフォグ。

これらのエフェクトは同時に適用することができます。Ice Cave デモでは両方の手法が使用されています。

9.10.2 手続き型リニア フォグ

オブジェクトが遠くになるほど、定義されたフォグのカラーへとカラーが薄くなるようにします。これをフラグメント シェーダで実現するには、フラグメント カラーとフォグ カラーの間で単純な線形補間を使用できます。

次のサンプル コードは、カメラまでの距離に基づいて、頂点シェーダのフォグのカラーを計算する方法を示しています。このカラーは、可変変数としてフラグメント シェーダに渡されます。

```
output.fogColor = _FogColor * clamp(vertexDistance * _FogDistanceScale, 0.0, 1.0);
```

ここでは下記のとおりです。

- `vertexDistance` は頂点からカメラまでの距離です。
- `_FogDistanceScale` はシェーダにユニフォームとして渡される係数です。
- `_FogColor` は定義可能でユニフォームとして渡す、ベースのフォグ カラーです。

フラグメント シェーダで、補間された `input.fogColor` が フラグメント カラー `output.Color` と結合されます。

```
outputColor = lerp(outputColor, input.fogColor.rgb, input.fogColor.a);
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

以下の図に、シーンで得られる結果を示します。

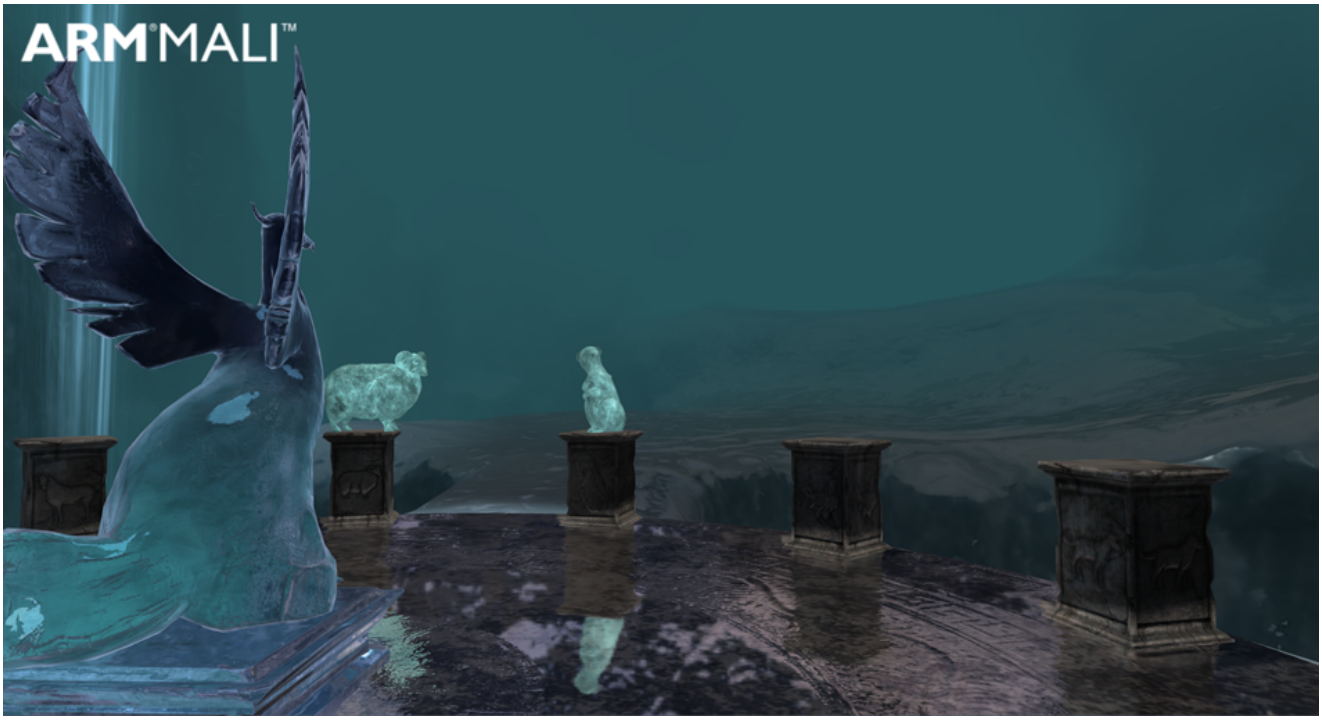


図 9-55 距離に基づくリニア フォグ

注

シェーダでフォグを計算するということは、フォグ エフェクトを生成するために追加の後処理を行う必要がないことを意味します。

できるだけ多くのエフェクトを 1 つのシェーダにマージしてみてください。ただし、キャッシュを超えるとパフォーマンスが低下する可能性があるため、パフォーマンスを確認するようにしてください。場合によっては、シェーダを 2 つ以上のパスに分割する必要があります。

フォグ カラーの計算は、頂点シェーダまたはフラグメント シェーダで行うことができます。フラグメント シェーダでの計算の方がより正確ですが、すべてのフラグメントに対して計算されるため、より多くの処理能力が必要となります。

頂点シェーダの計算は低精度ですが、頂点ごとに 1 回のみ計算されるため、パフォーマンスは高くなります。

9.10.3 高さのあるリニア フォグ

フォグは、シーン全体に均一に適用されます。高さに応じて密度を変えることで、よりリアルにすることができます。

低いところではフォグをより濃くし、高いところではより薄くします。高さレベルを公開して、手動で調整することもできます。

次の図は、距離と高さに基づくリニア フォグを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9-56 距離と高さに基づくリニア フォグ

9.10.4 不均一なフォグ

フォグは均一である必要はありません。ノイズを加えることで、フォグをより見た目に効果的にすることができます。

ノイズ テクスチャを適用することで、不均一なフォグを作成することができます。

より複雑なエフェクトを作成するには、複数のノイズ テクスチャを適用して、異なる速度でスライドさせることもできます。たとえば、カメラから遠くにあるノイズ テクスチャを、近くにあるテクスチャよりもゆっくりとスライドさせることができます。

1 つのシェーダに 1 回のパスで複数のテクスチャを適用することができます。距離に応じてノイズ テクスチャをブレンドできます。

9.10.5 フォグのプリベーキング

フォグにカメラが近づくことがない場合には、フォグをテクスチャにプリベーキングすることができます。これにより、必要な処理能力を低減できます。

9.10.6 パーティクルを使った 3 次元のフォグ

パーティクルを使って 3 次元のフォグをシミュレートできます。これにより高品質な結果を得ることができます。

パーティクルの数を最小限にします。フラグメントごとにより多くのシェーダが実行されるため、パーティクルはオーバードローを増加させます。パーティクルの数を増やさず、より大きなパーティクルを使用するようにします。

Ice Cave デモでは、同時に最大 15 個のパーティクルを使用しています。

ビルボード レンダリングに代わるジオメトリ

パーティクル システムのレンダリング モードを、ビルボードの代わりにメッシュに設定します。これは、3 次元のエフェクトのためには、個々のパーティクルがランダムに回転する必要があるためです。

次の図は、テクスチャなしのパーティクルのジオメトリを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

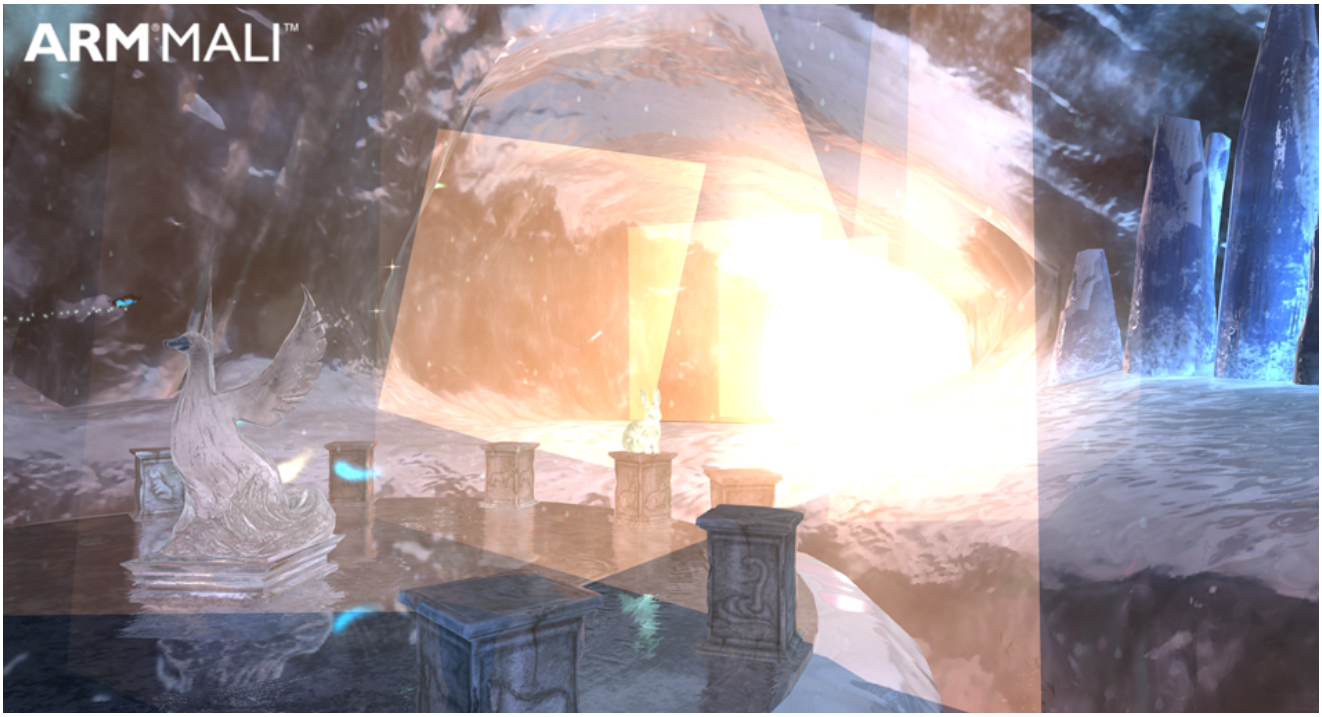


図 9-57 テクスチャなしのパーティクル

次の図は、テクスチャありのパーティクルのジオメトリを示しています。

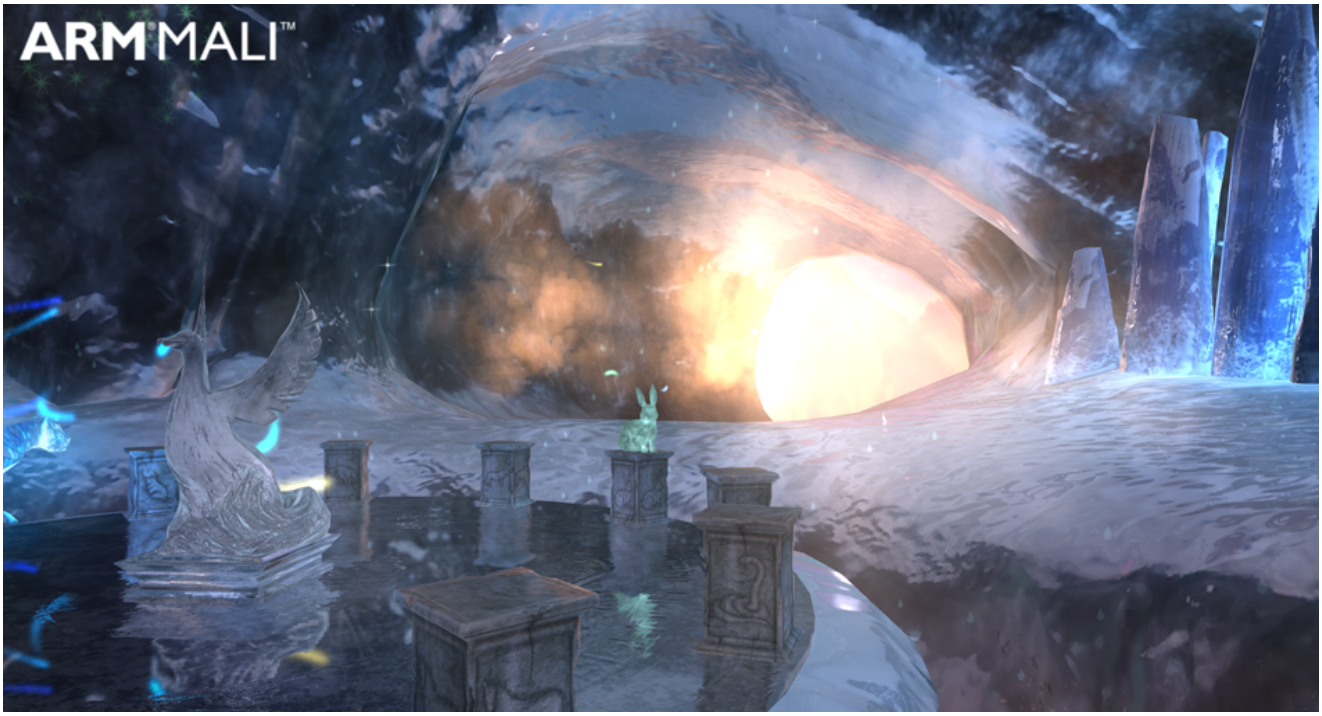


図 9-58 テクスチャありのパーティクル

次の図は、各パーティクルに適用されるテクスチャを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

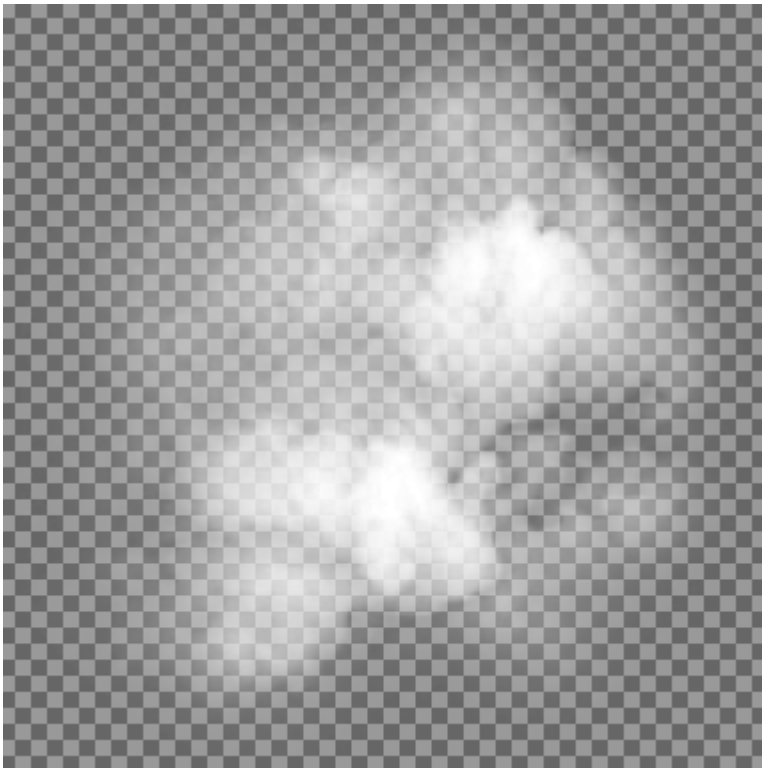


図 9-59 個々のパーティクルのテクスチャ

アングル フェード エフェクト

カメラの位置と比較した向きにしたがって、個々のパーティクルをフェード インまたはフェード アウトします。パーティクルをフェード インしたりフェード アウトしたりしないと、パーティクルの鋭いエッジが見えます。

次のサンプル コードは、フェーディングを実行する頂点シェーダを示しています。

```
 half4 vertexInWorld = mul(_Object2World, input.vertex);
half3 normalInWorld = (mul(half4(input.normal, 0.0), _World2Object).xyz);
const half3 viewDirInWorld = normalize(vertexInWorld - _WorldSpaceCameraPos);
output.visibility = abs(dot(-normalInWorld, viewDirInWorld));
output.visibility *= output.visibility;  // 2 乗の代わりに
```

可変パラメータ `output.visibility` は、パーティクル ポリゴンにわたって補間されます。フラグメント シェーダでこの値を読み取り、一定量の透明度を適用します。

次のコードは、この方法を示しています。

```
    half4 diffuseTex = _Color * tex2D(_MainTex, half2(input.texCoord));
    diffuseTex *= input.visibility;
    return diffuseTex;
```

パーティクルのレンダリング

パーティクルをレンダリングするには、次の手順を使用します。

1. パーティクルをフレーム内の最後のプリミティブとしてレンダリングします。

   ```
   Tags { "Queue" = "Transparent+10" }
   ```

   Ice Cave デモでは、パーティクルの前に 9 個の透明なオブジェクトがレンダリングされているため、このサンプルでは +10 となっています。
2. 適切なブレンド モードを設定します。

   シェーダ パスの始めに、次の行を追加します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
Blend SrcAlpha One
```

3. z バッファへの書き込みを無効にします。

   次の行を追加します。

```
ZWrite Off
```

パーティクル システムの設定

次の図は、Ice Cave デモのパーティクル システムで使用されている設定を示しています。

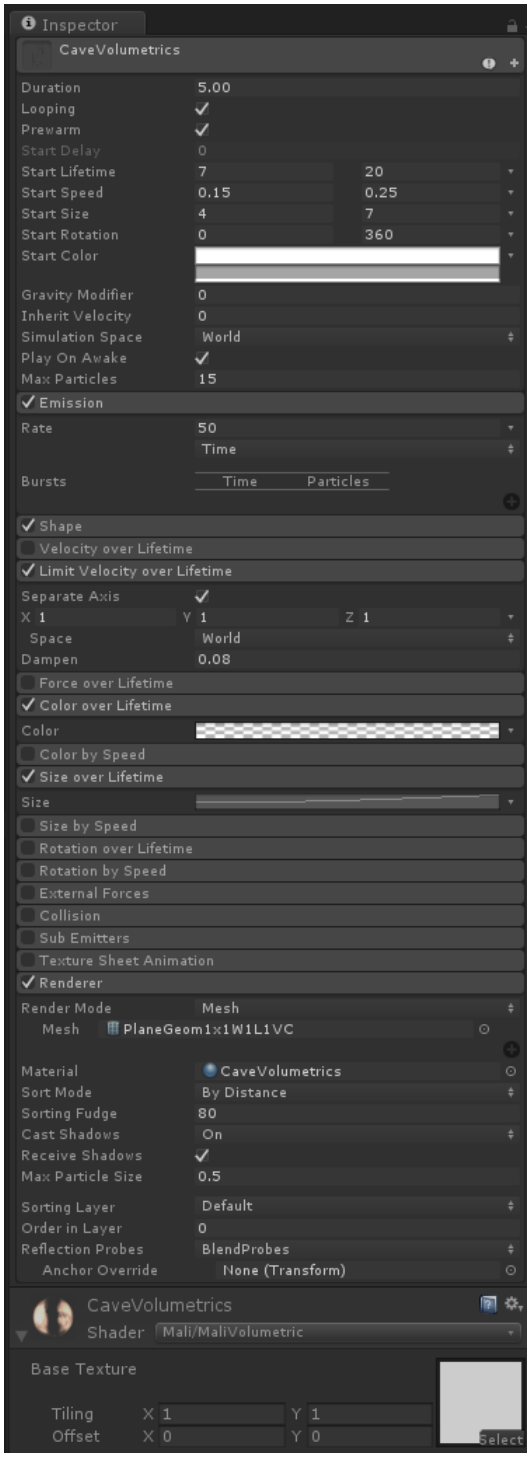


図 9-60 Ice Cave デモのパーティクル システム パラメータ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

次の図は、パーティクルが生成されるエリアを示しています。これは画像のボックスで示されます。このボックスは、Unity Particle System の Shape によって定義されます。

図 9-61 パーティクルが生成されるパーティクル ボックスの定義

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.11 ブルーム

ブルームは、明るい環境で写真を撮るときに実際のカメラで起こるエフェクトを再現するために使用されます。ブルーム エフェクトは、明るい領域の境界線から延びる光のふちをシミュレートし、カメラを圧倒する明るい光の錯覚を作り出します。

次の図は、Ice Cave デモの洞くつの入り口のブルームを示します。

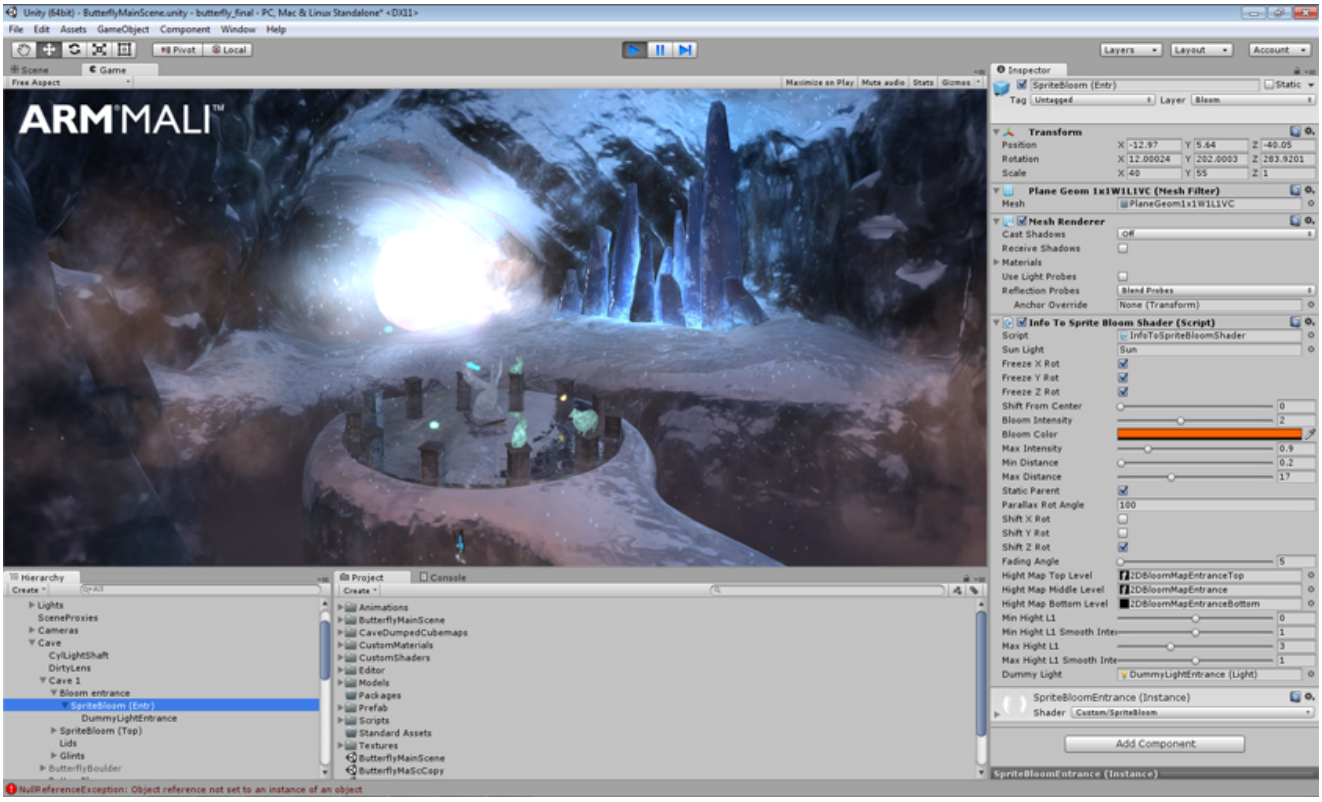


図 9-62 Ice Cave デモで実装された洞くつの入り口のブルーム

ブルーム エフェクトは、実際のレンズでは完全に焦点を合わせることができないことにより生じます。光がカメラの円形絞りを通過するときに、光の一部が回折して、画像のまわりに明るい円盤ができます。このエフェクトは、通常は多くの状況では顕著ではありませんが、非常に強いライティングの下で生じます。

以下のサブセクションから構成されています。

- *9.11.1 ブルームの実装*(9-212 ページ).
- *9.11.2 ブルーム調整係数スクリプトの作成*(9-213 ページ).
- *9.11.3 頂点シェーダ*(9-213 ページ).
- *9.11.4 フラグメント シェーダ*(9-214 ページ).
- *9.11.5 ブルーム エフェクトの修正*(9-214 ページ).
- *9.11.6 オクルージョン マップ間の補間*(9-217 ページ).

9.11.1 ブルームの実装

ブルーム エフェクトは通常、後処理エフェクトとして実装されます。このようなエフェクトを生成すると、大量の処理能力を使用する場合があるため、モバイル ゲームでの使用には適していません。これは、Ice Cave デモのような、多くの複雑なエフェクトを使用するゲームで特に当てはまります。

別のアプローチでは、単純な平面を使用します。カメラと光源の間に平面を配置します。平面は、カメラが移動する予定の場面の部分を指し示す法線を向いている必要があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

この方法によってブルーム エフェクトを作成するには、ブルーム エフェクトを発生させる場所に平面を配置します。ビュー ベクトルと、平面法線と光源のアラインメントに基づいた係数を使用して、エフェクトの強度を調整します。Ice Cave デモではこの方法が実装されています。

次の図は、平面を使ったブルーム エフェクトの実装を示しています。

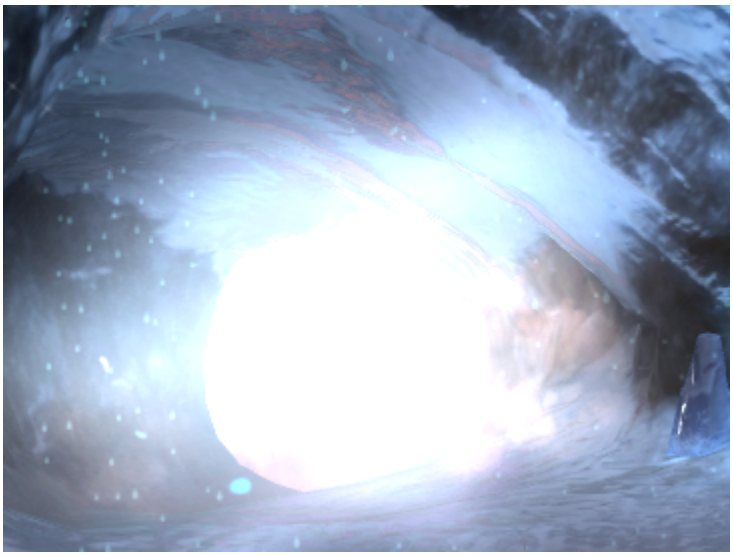


図 9-63 ブルームを実装した洞くつの入り口の平面

9.11.2 ブルーム調整係数スクリプトの作成

アラインメント係数はスクリプトを使って計算できます。この係数により、カメラからエフェクトを見る角度に応じて、ブルーム エフェクトが変化します。

たとえば、次のスクリプトは、接している平面のアラインメント係数を計算します。

```
// ライト面
normal-camera alignmentVector3 planeToCamVec = Camera.main.transform.position
   - gameObject.transform.position;
planeToCamVec.Normalize();
Vector3 sunLightToPlainVec = origPlainPos - sunLight.transform.position;
sunLightToPlainVec.Normalize();
float sunLightPlainCameraAlignment = Vector3.Dot(plainToCamVec, sunLightToPlainVec);
sunLightPlainCameraAlignment = Mathf.Clamp (sunLightPlainCameraAlignment, 0.0f, 1.0f);
```

アラインメント係数 `sunLightPlaneCameraAlignment` はシェーダに渡され、レンダリングされるカラーの強度を調整します。

9.11.3 頂点シェーダ

頂点シェーダは `sunLightPlainCameraAlignment` 係数を受け取り、それを使ってレンダリングされるブルームのカラーの強度を調整します。

頂点シェーダは、MVP 行列を適用し、テクスチャ座標をフラグメント シェーダに渡して、頂点座標を出力します。

次のコードは、この方法を示しています。

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.11.4 フラグメント シェーダ

フラグメント シェーダは、テクスチャ カラーを取り出し、CurrBloomColor ティント カラーを使用してインクリメントします。これは、ユニフォーム変数として渡されます。アライメント係数は、使用される前にカラーを調整します。

次のコードは、この処理が実行される方法を示しています。

```
half4 frag(vertexOutput input) : COLOR
{
    half4 textureColor = tex2D(_MainTex, input.tex.xy);
    textureColor += textureColor * _CurrBloomColor;
    return textureColor * _AligmentFactor;
}
```

ブルーム平面は、すべての不透明なジオメトリの後に、透過キューにレンダリングされます。シェーダで、次のキュー タグを使用してレンダリング順序を設定します。

```
Tags { "Queue" = "Transparent"  + 1}
```

ブルーム平面は、フレーム バッファに既に格納されている対応するピクセルとフラグメント カラーを結合するために、追加の 1 対 1 ブレンディングを使用して適用されます。またシェーダは ZWrite Off 命令を使用してデプス バッファへの書き込みを無効にして、既存のオブジェクトが閉塞されないようにします。

次の図は、Ice Cave デモがブルーム エフェクトに使用するテクスチャと、その結果を示しています。

図 9–64 不透明な柱脚の背後のオクルージョン ゾーンに入るカメラを示すシーケンス

9.11.5 ブルーム エフェクトの修正

平面を使用したブルーム エフェクトの生成は、シンプルでモバイル デバイスに適しています。ただし、ブルームを生成する光源とカメラの間に不透明なオブジェクトが配置されている場合、期待される結果が得られません。これはオクルージョン マップを使って修正できます。

不透明なオブジェクトの背後にあるブルーム エフェクトが適切に表示されないのは、エフェクトが透過キューにレンダリングされるためです。このためブレンディングは、ブルーム平面の前のオブジェクトの上で行われます。

次の図は、修正なしで生成された不適切なブルーム エフェクトのサンプルを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9-65 不透明な柱脚に表示された、不適切なブルーム エフェクト

これらのエラーが発生しないようにするには、アライメント係数を計算するスクリプトを変更して、追加のオクルージョン マップを処理するようにします。このオクルージョン マップは、カメラが通常ブルーム エフェクトを不適切に適用するエリアを示しています。これらの競合エリアには、オクルージョン マップの値 0 が割り当てられます。この係数はアラインメント係数と組み合わせて機能し、これらのエリアでの強度を 0 に設定します。この変更により、ブルームを 0 に設定して、閉塞するオブジェクトをレンダリングしないようにします。次のコードはこの調整を実装します。

```
IntensityFactor = alignmentFactor * occlusionMapFactor
```

Ice Cave デモのカメラは 3 次元で自由に動くことができるため、3 つのグレースケール マップが使用され、それぞれが異なる高さをカバーしています。黒は 0 を意味するため、オクルージョン マップ係数にアライメント係数を掛けた最終的な強度は 0 となり、ブルームを閉塞します。白は 1 を意味するため、強度はアラインメント係数と等しくなり、ブルームは閉塞されません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

次の図は、地表面での `occlusionMapFactor` を示しています。

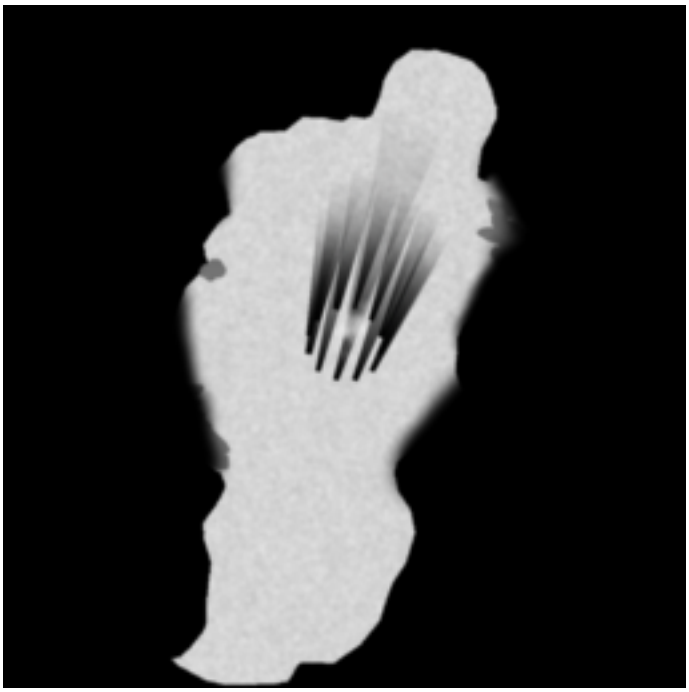


図 9-66 地表面でのブルーム オクルージョン マップ

注

マップ中央の黒い放射状のエリアは、洞くつの入り口からのブルーム効果を閉塞する不透明な柱脚の背後のゾーンです。

地表面からの高さが H_{max} のところには、閉塞オブジェクトは存在しません。これは、地表面より上のブルーム オクルージョン マップが白であることを意味します。次の図は、地表面より上での `occlusionMapFactor` を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

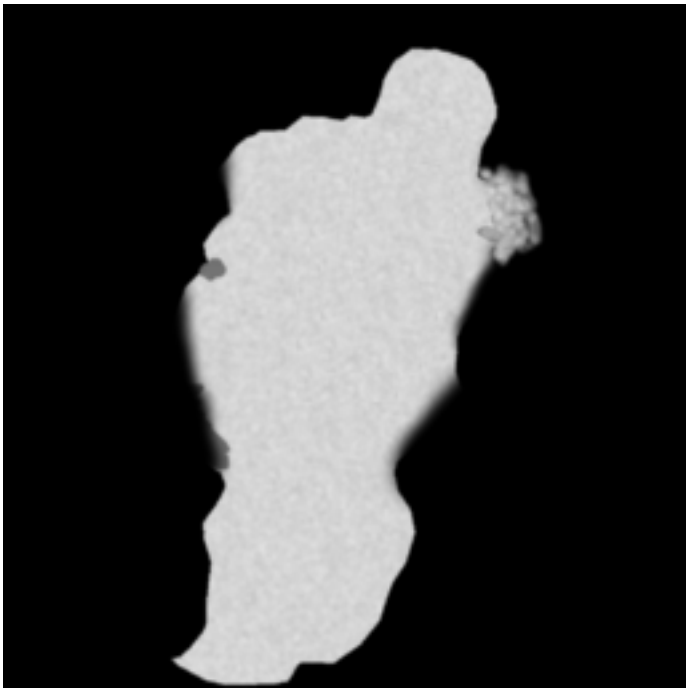


図 9–67 地表面の上でのブルーム オクルージョン マップ

高さ H_{min} より下の、地表面の下では、ブルーム エフェクトは完全に閉塞されています。これは、地表面より下のブルーム オクルージョン マップが完全に黒であることを意味します。白い輪郭は、マップが見えるように追加されています。次の図は、地表面より下での `occlusionMapFactor` を示しています。

図 9–68 地表面の下でのブルーム オクルージョン マップ

9.11.6 オクルージョン マップ間の補間

Ice Cave デモでは、スクリプトを使って洞くつの 2D マップ上のカメラの位置の XZ 投影をフレームごとに計算し、結果を 0 と 1 の間で正規化しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

カメラの高さが H_{min} より低いか、または H_{max} より高い場合は、正規化された座標を使用して、単一のマップからカラー値を取得します。カメラの高さが H_{min} と H_{max} の間にある場合は、両方のマップからカラーを取得して、補間します。

この補間により、異なる高さのエフェクトの間でスムーズなトランジションが作成されます。

Ice Cave デモでは、`GetPixelBilinear()` 関数を使用して、単一または複数のマップからカラーを取得しています。この関数は、次のコードを使用して、フィルタ カラー値を返します。

```
float groundOcclusionFactor = groundOcclusionMap.GetPixelBilinear(camPosXZNormalized.x,
camPosXZNormalized.y)).r;
```

ブルーム オクルージョン マップを使用すると、不透明な閉塞オブジェクト上にブルームがブレンドされるのを防ぐことができます。以下の図に、その結果のエフェクトを示します。カメラが出入りするときには、閉塞の黒が不透明な柱脚の背後にあり、不適切なブルームを防ぎます。

次の図は、不透明な柱脚の背後のオクルージョン ゾーンに入るカメラを示すシーケンスを示します。

図 9-69 不透明な柱脚の背後のオクルージョン ゾーンに入る

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.12 氷壁のエフェクト

Ice Cave デモでは、洞くつの氷壁で微妙な反射のエフェクトを使用しています。このエフェクトはわずかなものですが、シーンに臨場感と雰囲気を加えています。

以下のサブセクションから構成されています。

- *9.12.1 氷壁のエフェクトについて*(9-219 ページ).
- *9.12.2 反射に影響を及ぼす法線マップの修正と結合*(9-220 ページ).
- *9.12.3 異なる法線マップから反射を作成する*(9-222 ページ).
- *9.12.4 反射に局所補正を適用する*(9-223 ページ).

9.12.1 氷壁のエフェクトについて

氷は、表面の小さな細部に応じて、光がさまざまな方法で散乱するため、複製が難しいマテリアルとなります。氷の反射は、完全にクリアな場合もあり、完全に歪んでいる場合もあり、その中間の場合もあります。Ice Cave デモではこのエフェクトを使い、よりリアルな視差のエフェクトを加えています。

次の図は、このエフェクトを使った Ice Cave デモを示しています。

図 9-70 Ice Cave デモ

次の図は、このエフェクトの拡大図を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

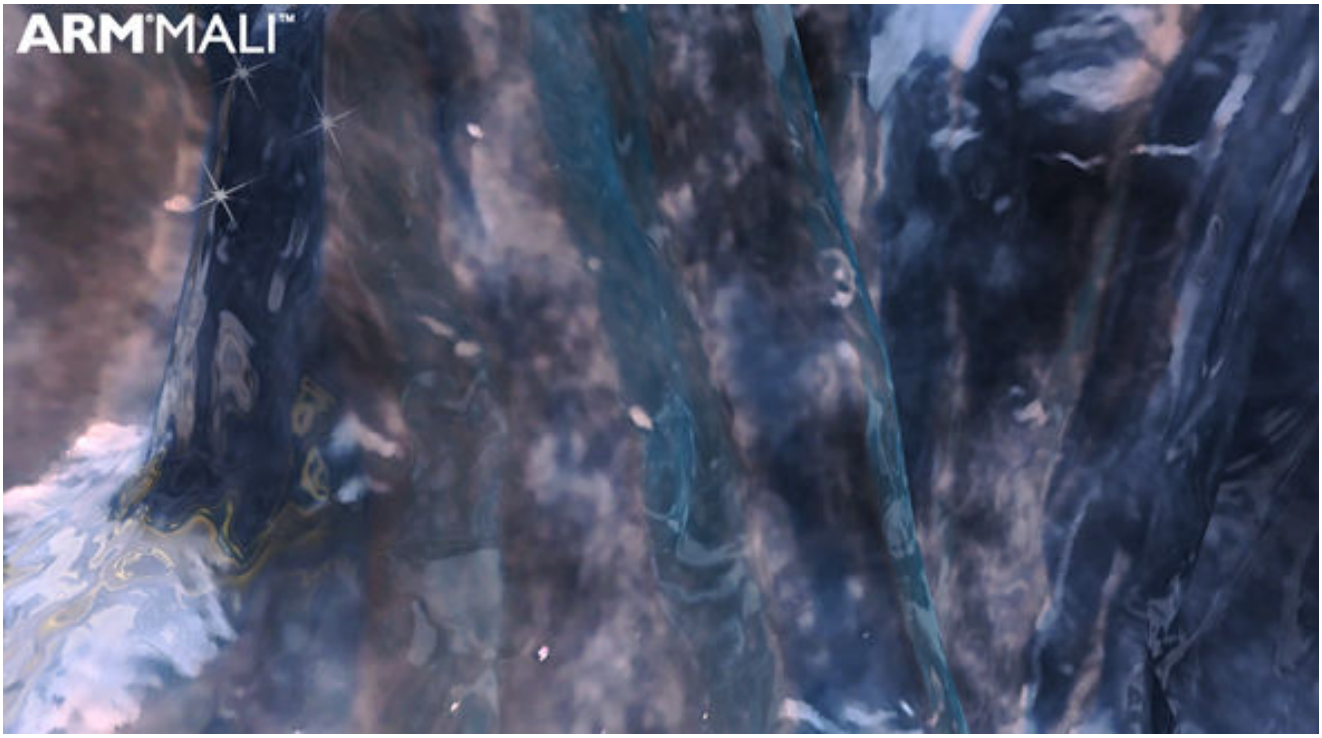


図 9-71 反射している氷壁の拡大図

9.12.2 反射に影響を及ぼす法線マップの修正と結合

Ice Cave デモの反射エフェクトは、接空間の法線マップと、計算されたグレースケールのフェイク法線マップを使用しています。これらのマップの両方をいくつかの修飾子と組み合わせて、このデモのエフェクトが作成されています。

グレースケールのフェイク法線マップは、接空間の法線マップのグレースケールです。Ice Cave デモでは、グレースケール マップのほとんどの値が 0.3 ー 0.8 の範囲にあります。次の図は、Ice Cave デモで使用されている、接空間の法線マップと、グレースケールのフェイク法線マップを示しています。

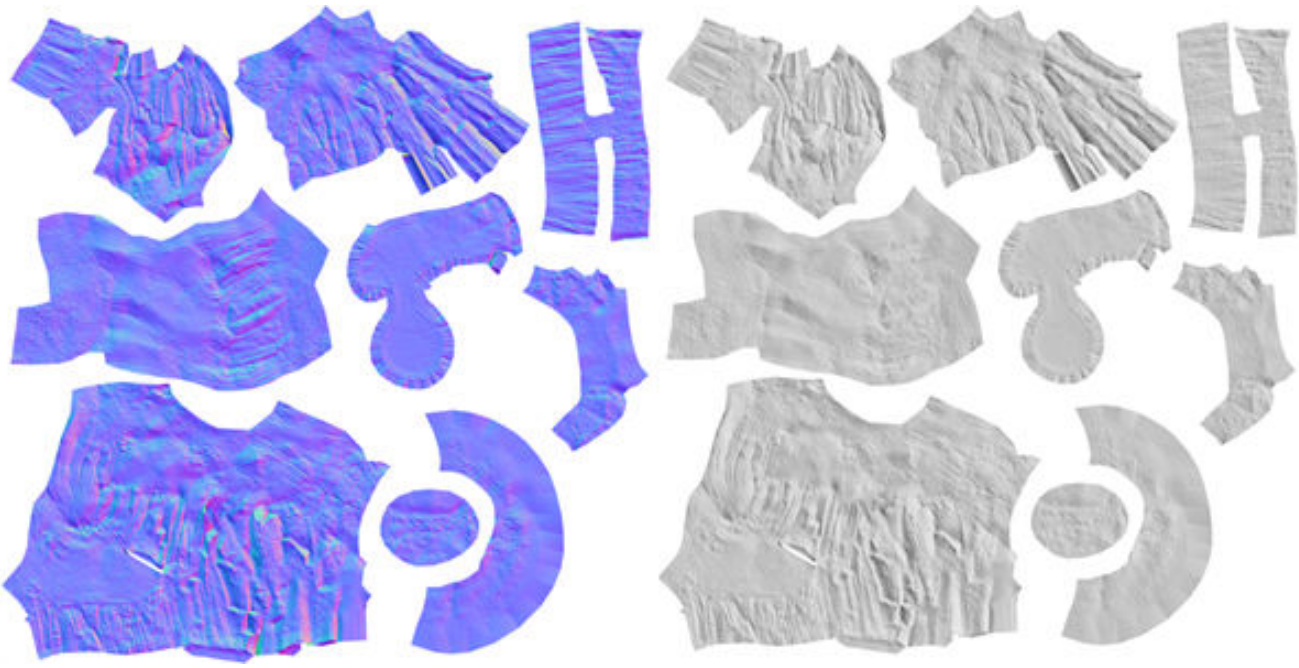


図 9-72 接空間の法線マップと、グレースケールのフェイク法線マップ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

接空間の法線マップを使う理由

Ice Cave デモで接空間の法線マップを使用しているのは、結果として得られるグレースケールの値の範囲が小さいためです。これは、接空間の法線マップが、このレンダリング プロセスの後の段階でうまく機能することを意味します。

または、オブジェクト空間の法線マップを使用することもできます。これらのマップは、接空間の法線マップと同等の詳細を示しますが、ライトが当たるエリアも示します。そのため、結果として得られるオブジェクト空間の法線マップのグレースケールの値の範囲が大きくなり、レンダリング プロセスの後の段階でうまく機能しません。次の図に、このエフェクトを示します。

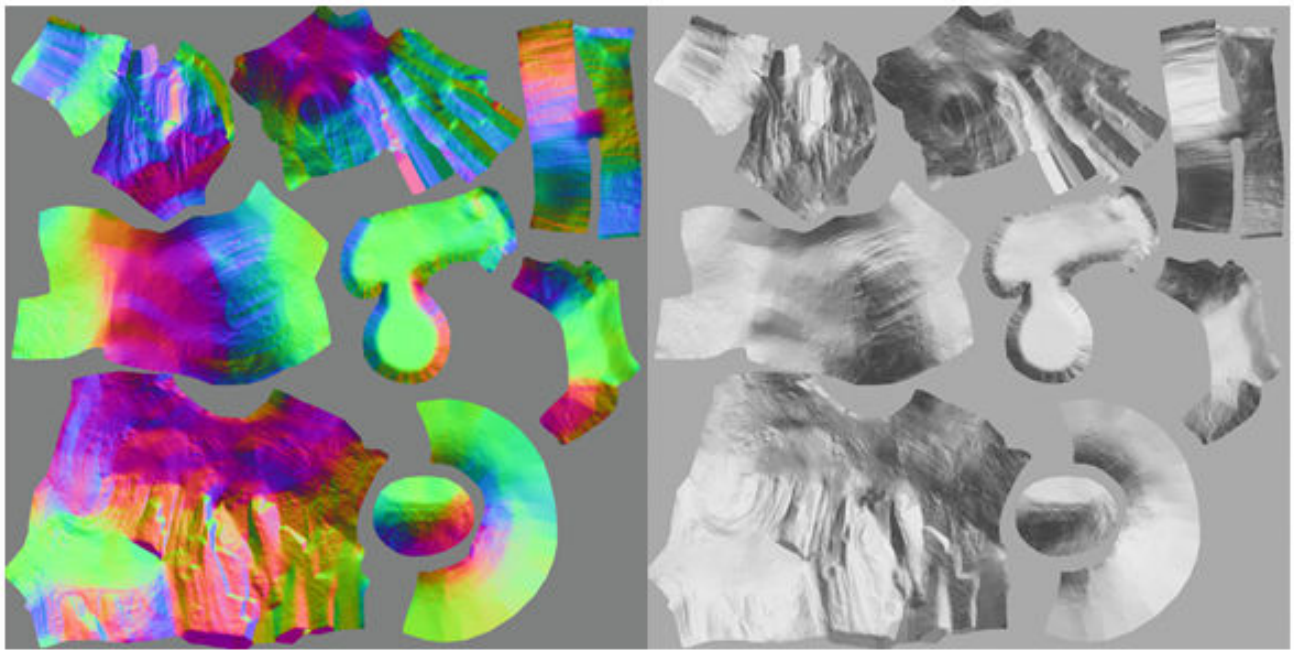


図 9–73 オブジェクト空間の法線マップと、グレースケール エフェクト

法線マップの一部に透明度を適用する

グレースケールのフェイク法線マップは、雪のないエリアにのみに適用されます。グレイスケールのフェイク法線マップが雪のあるエリアに適用されないようにするため、同じ表面の拡散テクスチャ マップを使用して修正しています。この修正により、テクスチャ マップで雪が表示されるエリアの、グレースケールのフェイク法線マップのアルファ成分を増加させています。

次の図は、Ice Cave デモの静的表面の拡散テクスチャ マップと、それらをグレースケールのフェイク法線マップに適用したときの結果を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

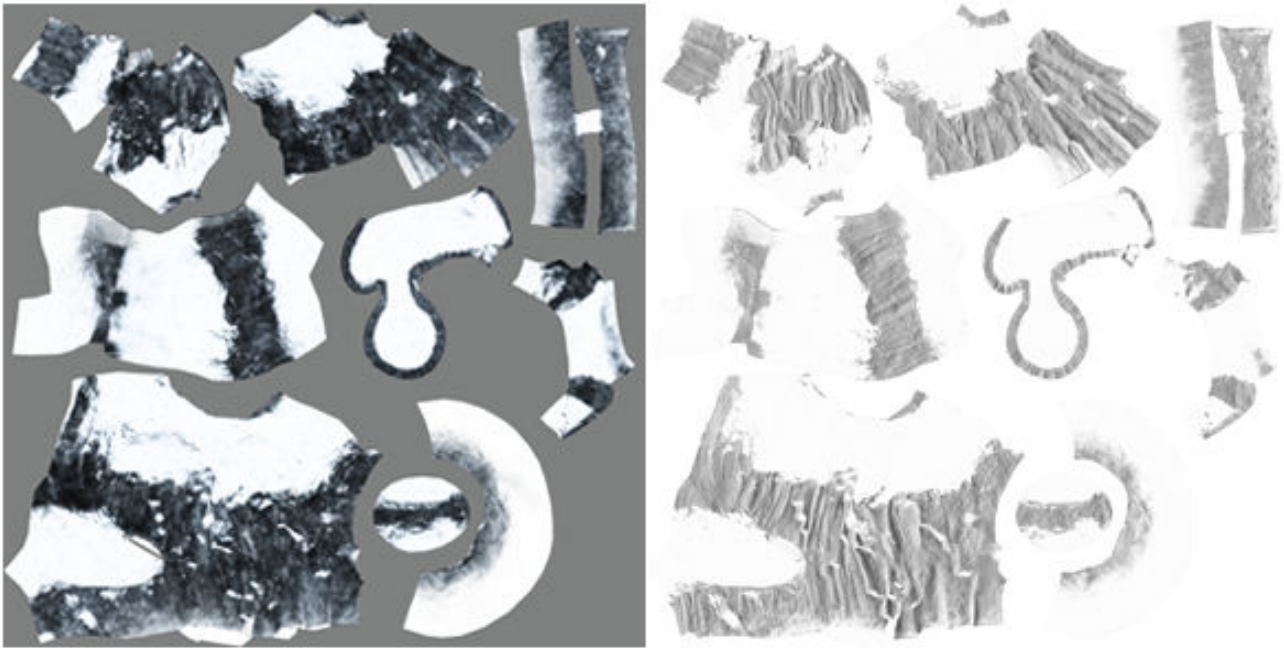


図 9-74 拡散テクスチャと、透過エリアのある最終的なフェイク法線マップ

9.12.3 異なる法線マップから反射を作成する

透明度を調整したグレイスケールのフェイク法線マップと、真の法線マップを組み合わせることにより、Ice Cave デモで示されている反射エフェクトが作成されます。

透明度を調整したグレースケールのフェイク法線マップ bumpFake と、真の法線マップ bumpNorm は比例的に結合されます。この結合には、次の関数を使います。

```
half4 bumpNormalFake = lerp(bumpNorm, bumpFake, amountFakeNormalMap);
```

このコードは、洞くつのより暗い部分では、反射は主にグレイスケールのフェイク法線によるものであることを意味しています。洞くつの雪のある部分では、エフェクトはオブジェクト空間の法線によるものです。

グレースケールのフェイク法線マップを使用してエフェクトを適用するには、グレースケールを法線ベクトルに変換する必要があります。このプロセスを開始するには、法線ベクトルの 3 つの成分をグレースケール値に等しく設定します。Ice Cave デモでは、ベクトルの成分が (0.3, 0.3, 0.3) と (0.8. 0.8, 0.8) の間にあることを意味します。すべての成分が同じに設定されているので、すべての法線ベクトルは同じ方向を指します。

シェーダは、法線の成分に変換を適用します。0 ー 1 の範囲の値を -1 ー 1 の範囲の値に変換するために通常使用される変換を使用します。この変換を行う式は、結果 = 2 * 値 - 1 です。この式は法線ベクトルを変更して、元の方向を指すか、または反対の方向を指すようにします。たとえば、元の成分が (0.3, 0.3, 0.3) の場合、結果の法線は (-0.4, -0.4, -0.4) となります。元の成分が (0.8, 0.8, 0.8) の場合、結果の法線は (0.6, 0.6, 0.6) となります。

-1 から 1 の範囲に変換された後で、ベクトルは reflect() 関数に渡されます。この関数は正規化された法線ベクトルを使用して動作するように設計されていますが、この場合は、正規化されていない法線が関数に渡されます。次のコードは、シェーダのビルトイン関数 reflect() の動作を示しています。

```
R = reflect(I, N) = I - 2 * dot(I, N) * N
```

1 より小さい長さの正規化されていない入力法線でこの関数を使用すると、反射ベクトルは、反射法則による通常の想定から大きくずれます。

法線ベクトルの成分の値が 0.5 を下回ると、反射ベクトルは反対方向に切り替わります。すると、キューブマップの別の部分が読み込まれます。キューブマップの別の部分に切り替わることにより、キューブマップの岩の部分の反射の隣にキューブマップの白い部分を反映する、不均一なスポットのエフェクトが

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

生成されます。グレースケールのフェイク法線マップで正と負の法線の切り替えが発生するエリアは、反射ベクトル上で最も変形した角度を生成するエリアでもあるため、効果的な渦エフェクトが作成されます。次の図に、このエフェクトを示します。

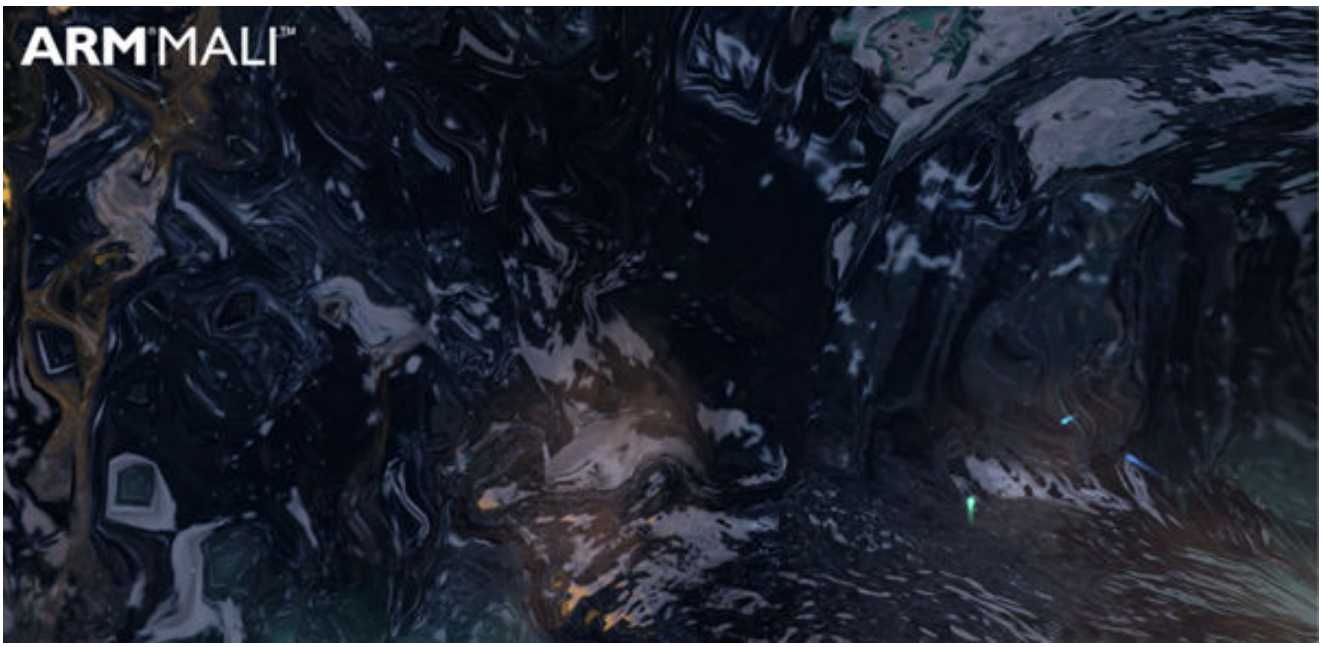


図 9–75 氷の反射の渦エフェクト

シェーダが、非正規化クランプ段階なしで、フェイク法線の値を使用するようにプログラムされている場合、結果として斜めの帯が表示されます。これは大きな違いであり、クランプ段階が重要であることを示しています。以下の図に、クランプ段階なしの場合の結果のエフェクトを示します。

図 9–76 クランプ段階なしの氷の反射

9.12.4 反射に局所補正を適用する

反射ベクトルに局所補正を適用することで、エフェクトの臨場感が向上します。この補正がなければ、現実の反射のように、カメラの位置によって反射が変化しません。これは、カメラの横方向の動きに対して特に当てはまります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

関連情報

9.2.2 ローカル キューブマップで正しい反射を生成する(9-159 ページ)

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.13 手続き型スカイボックス

Ice Cave デモでは、時刻システムを使用して、ローカル キューブマップで実現できる動的シャドウイング エフェクトを表示します。

以下のサブセクションから構成されています。

- *9.13.1 手続き型スカイボックスについて*(9-225 ページ).
- *9.13.2 時刻の管理*(9-226 ページ).
- *9.13.3 太陽のレンダリング*(9-227 ページ).
- *9.13.4 山の背後でフェードする太陽*(9-228 ページ).
- *9.13.5 表面下の散乱*(9-230 ページ).

9.13.1 手続き型スカイボックスについて

動的な時刻エフェクトを得るためには、以下の要素を組み合わせます。

- 手続き型で生成された太陽。
- 昼夜のサイクルを表す、フェードするスカイボックスの一連の背景キューブマップ。
- 雲のスカイボックス キューブマップ。

また、手続き型の太陽および分離された雲のテクスチャにより、低い計算コストで、表面下の散乱エフェクトを生成できます。

次の図は、スカイボックスのビューを示しています。

図 9-77 Ice cave デモでの、表面下の散乱のある太陽のレンダリング

Ice Cave デモでは、ビューの方向を使用して、キューブマップからサンプリングを行います。これにより、デモでスカイボックスの半球をレンダリングしないようにしています。代わりに、洞くつの穴の近くの平面だけを使用しています。

これによって、他のジオメトリによって大部分が閉塞された半球のレンダリングを行う場合と比べて、パフォーマンスを向上させています。

次の図は、平面を使用してレンダリングされたスカイボックスを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

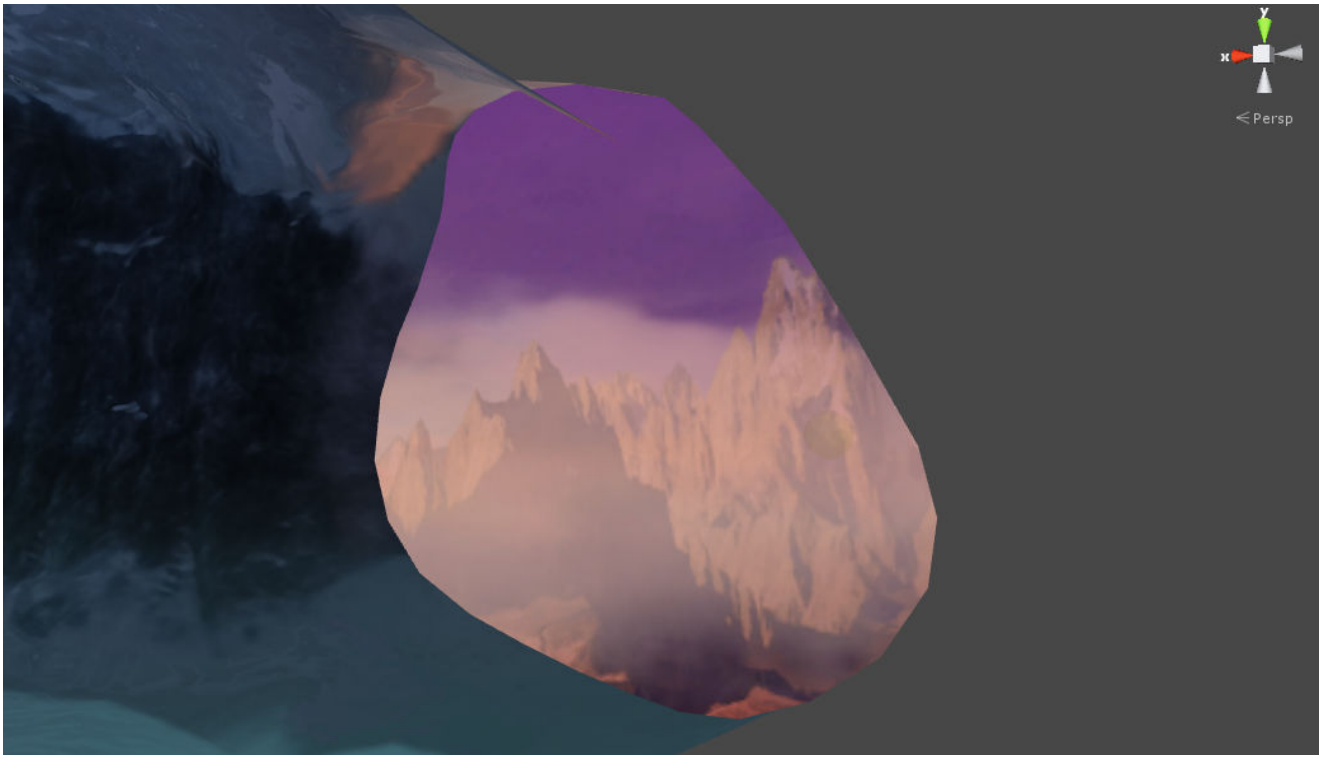


図 9–78 平面を使用してレンダリングされたスカイボックス

9.13.2 時刻の管理

このエフェクトは、C# スクリプトを使用して、時刻の数学と昼夜のアニメーションを管理します。シェーダは太陽とスカイマップを結合します。

スクリプトには、次の値を指定する必要があります。

- フェーディング スカイボックスの背景フェーズの数。
- 昼夜サイクルの最大期間。

各フレームについて、スクリプトは一緒にブレンドするスカイボックス キューブマップを選択します。選択されたスカイボックスは、シェーダのテクスチャとして設定され、レンダリング中にテクスチャをブレンドします。

テクスチャを設定するため、Ice Cave デモでは Unity のシェーダのグローバル機能を使用しています。これにより、アプリケーション内のすべてのシェーダで使用できるテクスチャを 1 か所で設定できます。次のコードはこれを示しています。

```
Shader.SetGlobalTexture (ShaderCubemap1, _phasesCubemaps [idx1]);
Shader.SetGlobalTexture (ShaderCubemap2, _phasesCubemaps [idx2]);
Shader.SetGlobalFloat (ShaderAlpha, blendAlpha);
Shader.SetGlobalVector (ShaderSunPosition, normalizedSunPosition);
Shader.SetGlobalVector (ShaderSunParameters, _sunParameters);
Shader.SetGlobalVector (ShaderSunColor, _sunColor);
Shader.SetGlobalTexture (ShaderCloudsCubemap, _CloudsCubemap);
```

注

使用されるサンプラーの名前は、シェーダによってローカルに定義されたものと競合してはなりません。

スクリプト コードでは次のものが設定されています。

- 補間される 2 つのキューブマップ。`idx1` と `idx2` の値は、経過時間に基づいて計算されます。
- シェーダで 2 つのキューブマップをブレンドするために使用される `blendAlpha` 係数。
- 太陽の球をレンダリングするために使用される、正規化された太陽の位置。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

- 太陽のためのいくつかのパラメータ。
- 太陽のカラー。
- 雲のキューブマップ。ShaderCubemap1 と ShaderCubemap2 は一意のサンプラー名を含む 2 つの文字列です。この場合は、_SkyboxCubemap1 と _SkyboxCubemap2 です。

シェーダでこれらのテクスチャにアクセスするには、次のコードを使用してテクスチャを宣言する必要があります。

```
samplerCUBE _SkyboxCubemap1;
samplerCUBE _SkyboxCubemap2;
```

スクリプトは、指定したリストに基づいて、太陽のカラーとアンビエント カラーを選択します。これらは、各フェーズごとに補間されます。

太陽のカラーをシェーダに渡して、太陽が正しいカラーでレンダリングされるようにします。

アンビエント カラーは、Unity 変数 RenderSettings.ambientSkyColor を動的に設定するために使用されます。

```
RenderSettings.ambientSkyColor = _ambientColor;
```

この変数を設定することにより、すべてのマテリアルは正しいアンビエント カラーを受け取ることができます。Ice Cave デモでは、このエフェクトによって、シーンの全体的な色が 1 日のフェーズに基づいて段階的に変化します。

9.13.3 太陽のレンダリング

太陽をレンダリングするには、フラグメント シェーダでスカイボックスの各ピクセルについて、太陽の円周にあるかどうかをチェックする必要があります。

これを行うには、シェーダは、レンダリングされるピクセルのワールド座標における、正規化された太陽の位置ベクトルと正規化されたビューの方向ベクトルとのドット積を計算します。

次の図は、太陽のレンダリングを示しています。正規化されたビュー ベクトルと太陽の位置のドット積を使用して、フラグメントが P2 と同じ空としてレンダリングされるか、P1 として太陽としてレンダリングされるかが決定されます。

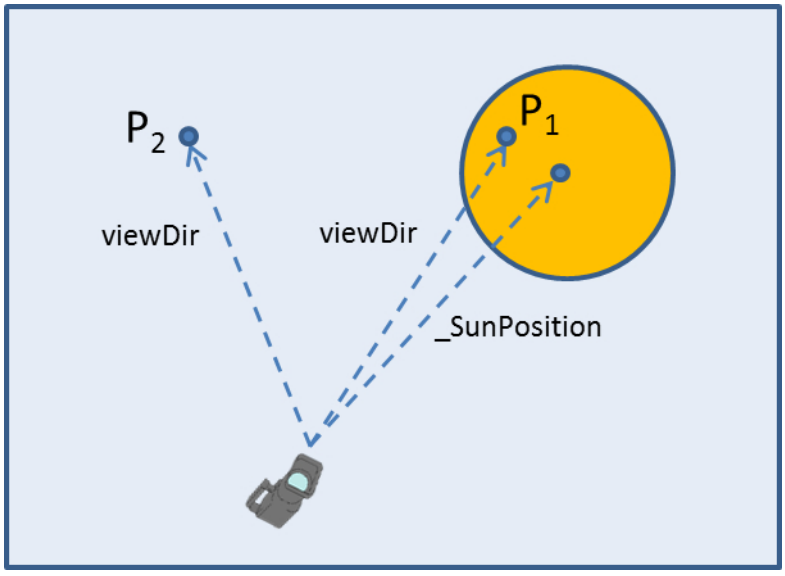


図 9-79 太陽のレンダリング

正規化された太陽の位置ベクトルは、C# スクリプトによって、シェーダに渡されます。

- 結果が特定のしきい値より大きい場合、ピクセルは太陽のカラーで色付けされます。
- 結果がしきい値よりも小さい場合、ピクセルは空のカラーで色付けされます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

さらにドット積は、太陽の縁に向かって、フェーディング エフェクトを生成します。

```
half _sunContribution = dot(viewDir,_SunPosition);
```

次の図は、晴れた太陽のビューを示しています。

図 9–80 空が晴れた状態での太陽の手続き型レンダリング

9.13.4 山の背後でフェードする太陽

太陽が空の低い場所にある場合は問題があります。実際には、山の背後に消えてしまいます。

このエフェクトを作成するには、キューブマップのアルファ チャネルを使い、テクセルが空を表す場合は値 `0` を格納し、テクセルが山を表す場合は `1` を格納します。

太陽をレンダリングしている間に、テクスチャがサンプリングされ、アルファを使って太陽を山の背後にフェードさせます。このサンプリングには、実質的にコストがかかりません。テクスチャは山をレンダリングするために、既にサンプリングされているためです。

またアルファを縁の近くで徐々にフェードさせたり、雪のある山でフェードさせたりすることもできます。これにより、わずかな計算コストのみで、雪で照り返す太陽のエフェクトを作成できます。

同様の手法を用いて、雲による表面下の散乱エフェクトを、低い計算コストで生成できます。

元のフェーズのキューブマップは、別の 2 つのグループに分割されます。

- 1 つのキューブマップのセットには、空と山が含まれています。アルファ値は、空には 0 に設定され、山には 1 に設定されます。
- もう 1 つのキューブマップのセットには雲が含まれています。アルファ値は、雲がないエリアでは 0 に設定され、雲が濃いエリアにかけて徐々に増加して 1 となります。

次の図は、山のテクスチャを示しています。山のアルファ値は 1、空は値 0 となります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9-81 山のテクスチャ

雲のスカイボックスは、雲のないエリアでアルファ値 0 で、雲のエリアにかけて徐々に 1 となります。アルファ チャネルはスムーズにフェードし、雲が人為的に空に置かれたように見えないようにします。次の図は、アルファのある雲のテクスチャを示しています。

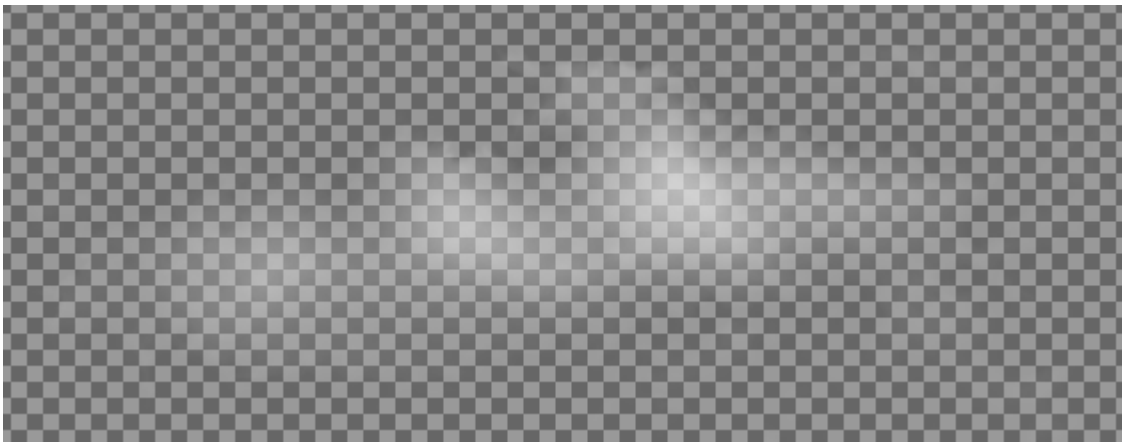


図 9-82 雲のテクスチャ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

シェーダは次の処理を行います。

1. 現在の時刻のフェーズを表す、2 つのスカイボックスをサンプリングします。
2. C# スクリプトで計算されたブレンド係数に基づいて、2 つの色をブレンドします。
3. 雲のスカイボックスをサンプリングします。
4. 雲のアルファを使用して、ステップ 2 のカラーと雲のカラーをブレンドします。
5. 雲のアルファとスカイボックスのアルファを一緒に追加します。
6. 現在のピクセルへの太陽のカラーの影響を計算する関数を呼び出します。
7. ステップ 6 の結果を、以前にブレンドされたスカイボックスと雲のカラーと一緒に追加します。

注

空、山、雲を 1 つのスカイボックスに入れることで、この手順を最適化できます。これらは Ice Cave デモでは別々になっているので、スカイボックスと雲を簡単に個別に変更できます。

9.13.5 表面下の散乱

アルファ情報を使用する、表面下の散乱エフェクトを追加して、太陽の半径を広げることができます。

実際には、太陽が晴れた空にあるときには、その大きさは比較的小さく見えます。これは、目に届く光が雲によって偏向または散乱することがないためです。大気による拡散は非常にわずかで、太陽の光線を直接見ています。

太陽の一部が雲に遮られ、完全には隠れていない場合、光の一部が雲の周りに散らばります。この光は、太陽からある程度離れた方向から目に届く場合があります。これにより、太陽が実際よりも大きく見える場合があります。

Ice Cave デモでこのエフェクトを得るには、次のコードを使用します。

```
half4 sampleSun(half3 viewDir, half alpha);
{
  half _sunContribution = dot(viewDir,_SunPosition);
  half _sunDistanceFade = smoothstep(_SunParameters.z -(0.025*alpha), 1.0, _sunContribution);
  half _sunOcclusionFade = clamp( 0.9-alpha, 0.0, 1.0);

   half3 _sunColorResult = _sunDistanceFade * _SunColor * _sunOcclusionFade;
   return half4( _sunColorResult.xyz, 1.0 );
}
```

雲のアルファとスカイボックスのアルファが一緒に追加された場合、関数のパラメータは `viewDirection` および計算されたアルファです。

太陽の位置とビューの方向のドット積を使って、太陽の中心から現在のピクセルの距離を表す倍率を計算します。

`_sunDistanceFade` の計算では、`smoothstep()` 関数を使用して、太陽の中心からエッジの近くの空まで、徐々にフェードします。またこれにより、表面下の散乱エフェクトをシミュレートして、雲の近くで太陽の半径を増加させます。

この関数は、アルファに基づいた可変ドメインを持ちます。晴れた空の場合、アルファは `0` で、範囲は `_SunParameters.z` と `1.0` の間です。この場合、`_SunParameters.z` は C# スクリプトで `0.995` に初期化されます。これは `cos(5 degrees) = 0.995` となり、直径 5 度の太陽に対応します。

処理中のピクセルに雲が含まれている場合、太陽の半径を 13 度に増加させ、雲に近づくと細長い散乱効果が得られるようにします。

`_sunOcclusionFade` 係数は、山と雲から受けた閉塞に基づいて、太陽の影響をフェード ダウンするために使用します。

次の図は、雲に遮られていない太陽を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9–83 雲に遮られていない太陽

次の図は、雲に遮られている太陽を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 9–84 雲に遮られている太陽

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.14 ホタル

ホタルは光を放つ飛翔虫で、Ice Cave デモでは躍動感を加えるために使用されています。

以下のサブセクションから構成されています。

- *9.14.1 ホタルについて*(9-233 ページ).
- *9.14.2 ホタル ジェネレータ プレハブ*(9-235 ページ).

9.14.1 ホタルについて

ホタルは以下のコンポーネントで構成されています。

- 実行時にインスタンス化されるプレハブ オブジェクト。
- ホタルが飛ぶことができるエリアを制限するために使用するボックス コライダー。

これらの 2 つのコンポーネントは、ホタルの動きを管理し、ホタルが動くパスを定義する C# スクリプトによって結合されています。

次の図はホタルを示しています。

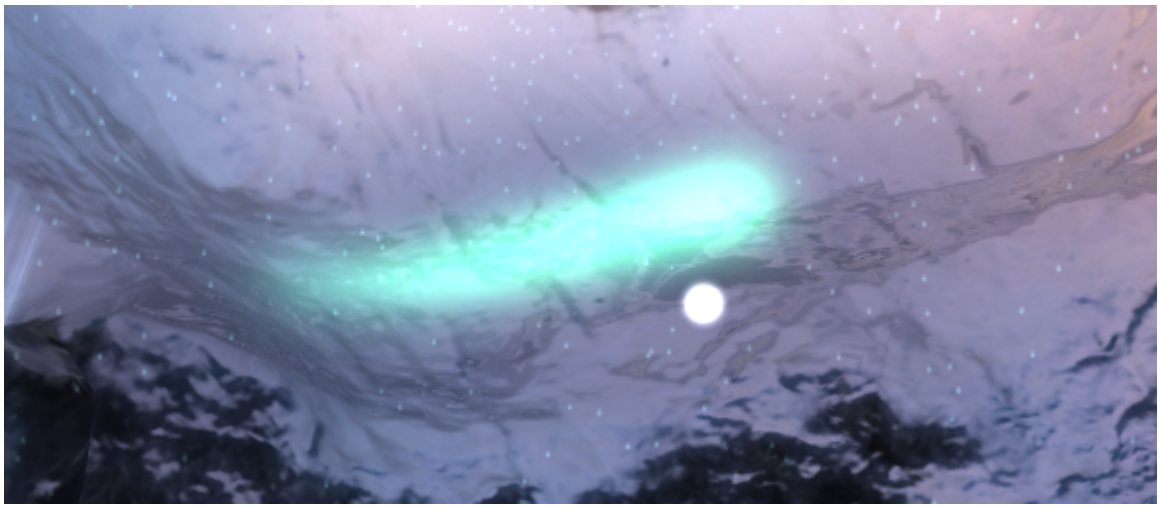


図 9-85 ホタル

ホタルのプレハブは、Unity 標準のパーティクル システムを使用して、ホタルのトレイルを生成します。

次の図は、ホタルが使用する Unity のパーティクル システムの設定を示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

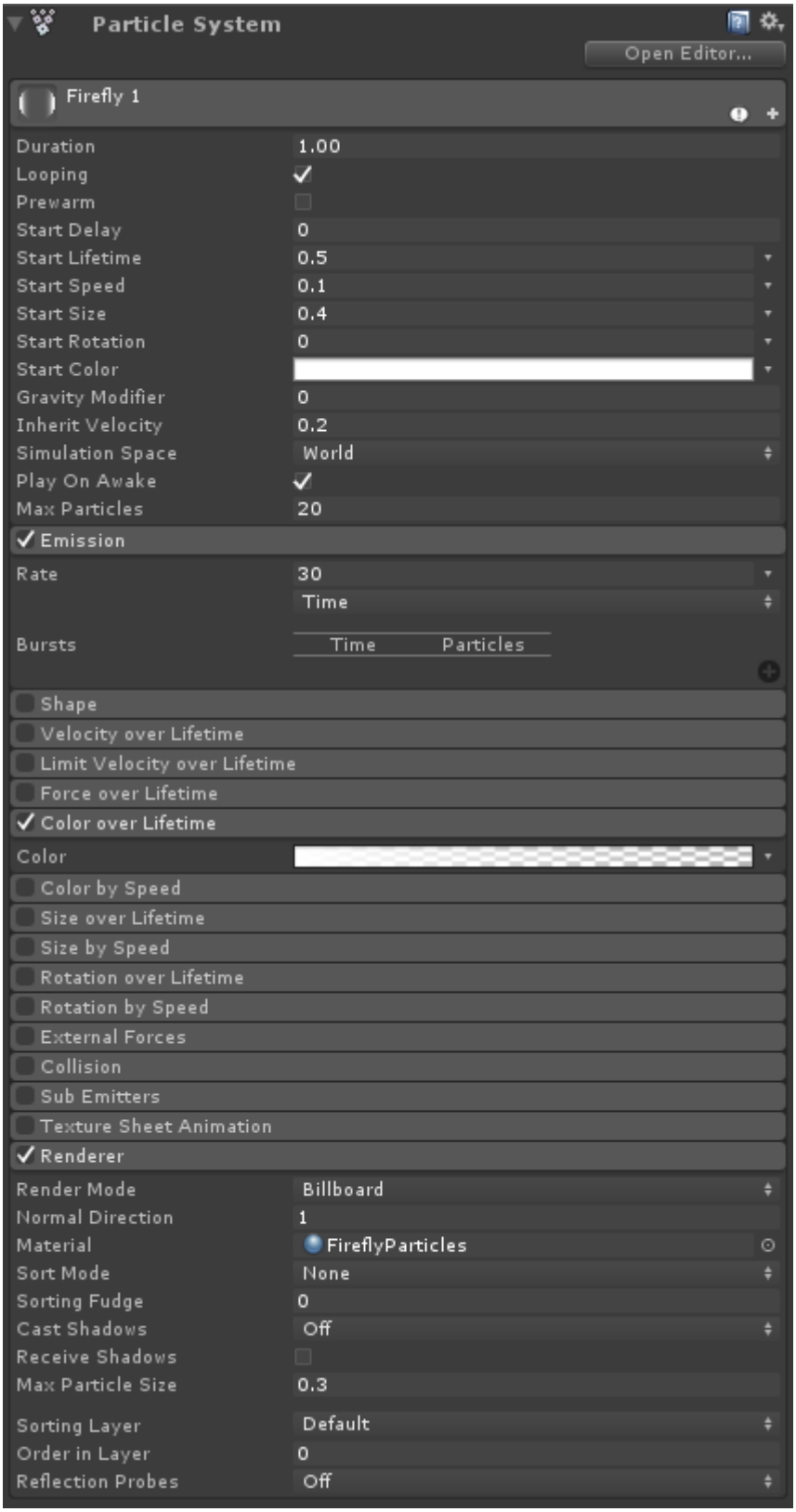


図 9-86 Unity パーティクル システムの設定

実行するエフェクトに応じて、Unity トレイル レンダラーを使用して、より連続的な外観を実現できます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

トレイル レンダラーは、各トレイルごとに多数の三角形を生成します。トレイル レンダラーの設定で [Min Vertex Distance] を変更すると三角形の数を変更できますが、値を大きくすると、ソースを速く動かした場合にトレイルの動きがぎくしゃくすることがあります。

[Min Vertex Distance] オプションは、トレイルを形成する頂点間の最小距離を定義します。直線のトレイルの場合は大きな数値でも良いかもしれませんが、曲がったトレイルの場合は滑らかに見えません。

生成されたトレイルは常にカメラに面しています。そのため、ソースの不意な動きによってトレイルが重なり合う場合があります。これにより、トレイルを形成する三角形のブレンドによって生成されるアーティファクトが発生します。

次の図は、重複するトレイルに起因するアーティファクトを示しています。

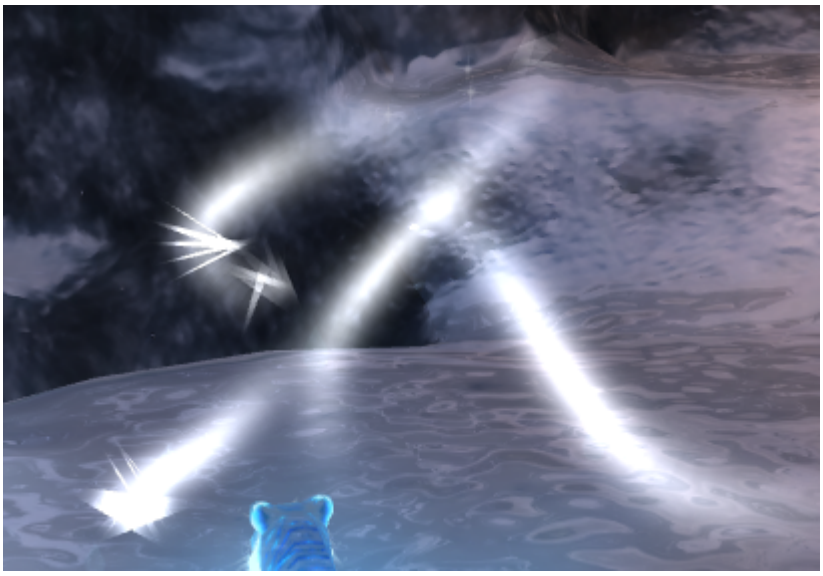


図 9-87 重複するトレイルによるアーティファクト

プレハブに追加される最後の要素は、動いているシーンで光を投射するポイント ライトです。

9.14.2 ホタル ジェネレータ プレハブ

ホタル ジェネレータ プレハブは、ホタルの作成を管理し、各フレームを更新します。これは、更新用の C# スクリプトと、各ホタルが移動できるボリュームを囲むボックス コライダーで構成されています。

このスクリプトは、パラメータとして生成されるホタルの数を取得し、ホタル オブジェクトをインスタンス化するためのプレハブを取得します。ホタルの動きは境界ボックス内でランダムなため、ホタルが動いている方向から特定の角度への変化が制限されます。これにより、ホタルの方向が突然変わらないようにします。

ランダムな動きを生成するために、区分的 3 次元エルミート補間を用いて制御点を生成します。エルミート補間は、異なるパスが一緒に接続されている場合でも正しく動作する、滑らかな連続関数を提供します。終点の一次導関数も連続であるため、急激な速度や方向の変化はありません。

この補間は、開始点と終了点のそれぞれに 1 つの制御点と、各制御点に 2 つの接線を必要とします。これらはランダムに生成されるため、スクリプトは 3 つの制御点と 2 つの接線を格納できます。第 1 および第 2 の制御点の位置を使用して第 1 の制御点の接線を定義し、第 2 および第 3 の制御点を使用して第 2 の制御点の接線を定義します。

読み込み時に、スクリプトは各ホタルに対して次のものを生成します。

- 初期の位置。
- Unity 関数 `Random.onUnitSphere()` を使った、初期の方向。

次のコードは、制御点の初期化方法を示しています。

```
_fireflySpline[i*_controlPoints] = initialPosition;
Vector3 randDirection = Random.onUnitSphere;
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
_fireflySpline[i*_controlPoints+1] = initialPosition + randDirection;
_fireflySpline[i*_controlPoints+2] = initialPosition + randDirection * 2.0f;
```

初期の制御点は直線上にあります。これらの制御点から接線が生成されます。

```
// 第 1 の制御点の接線は、初期の方向ベクトルと同じ方向です。
_fireflyTangents[i*_controlPoints] = randDirection;

// このコードは、制御点の位置から接線を計算します。これは初期化時に randDirection に設定できるため、
// ここでは参照用に示しています。
_fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
_fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1] -
_fireflySpline[i*_controlPoints])/2;
```

ホタルの初期化を完了するには、ホタルのカラーと現在のパス間隔の長さを設定する必要があります。

各フレームで、スクリプトは次のコードを使用してエルミート補間を計算し、各ホタルの位置を更新します。

```
// t は、ホタルを配置する曲線上の場所を定義するパラメータです。
// ホタルがパスに沿って移動した時間の、合計時間に対する比率を表します。
float t = _fireflyLifetime[i].y / _fireflyLifetime[i].x;

// エルミート補間パラメータ
Vector3 A = _fireflySpline[i*_controlPoints];
Vector3 B = _fireflySpline[i*_controlPoints+1];
float h00 = 2*Mathf.Pow(t,3) - 3*Mathf.Pow(t,2) + 1;
float h10 = Mathf.Pow(t,3) - 2*Mathf.Pow(t,2) + t;
float h01 = -2*Mathf.Pow(t,3) + 3*Mathf.Pow(t,2);
float h11 = Mathf.Pow(t,3) - Mathf.Pow(t,2);
// 更新されたホタルの位置
_fireflyObjects[i].transform.position = h00 * A + h10 * _fireflyTangents[i*_controlPoints]
+ h01 * B + h11 * _fireflyTangents[i*_controlPoints+1];
```

ホタルがランダムに生成されたパス全体を通過した場合、スクリプトは現在のパスの終わりから始まる新しいランダムなパスを作成します。

```
// t > 1.0 は現在のパスの終わりを示します。
if( t >= 1.0 )
{

    // 新しい位置を更新
    // 第 2 の点を第 1 の点にシフトし、接線も同様にシフトします。
    _fireflySpline[i*_controlPoints] = _fireflySpline[i*_controlPoints+1];
    _fireflyTangents[i*_controlPoints] = _fireflyTangents[i*_controlPoints+1];

    // 第 3 の点を第 2 の点にシフトします。この点には接線はありません。
    _fireflySpline[i*_controlPoints+1] = _fireflySpline[i*_controlPoints+2];

    // 現在の飛行方向から一定の角度内で新しいランダムな制御点を取得します。
    _fireflySpline[i*_controlPoints+2] = GetNewRandomControlPoint();

    // 中心点の接線を計算します。
    _fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
    _fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1] -
    _fireflySpline[i*_controlPoints])/2;

    // パスのこの部分を通過するのにかかる時間を設定します。
    _fireflyLifetime[i].x = _fireflyMinLifetime;

    // パスに沿った移動時間を計るためのタイマ。
    _fireflyLifetime[i].y = 0.0f;
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

9.15 接空間からワールド空間への法線変換ツール

接空間からワールド空間への法線変換ツールは、C# スクリプトとシェーダで構成されています。このツールは Unity エディタ内からオフラインで実行され、ゲームのランタイム パフォーマンスには影響しません。

以下のサブセクションから構成されています。

- *9.15.1 接空間からワールド空間への法線変換ツールについて*(9-237 ページ).
- *9.15.2 C# スクリプト*(9-237 ページ).
- *9.15.3 WorldSpaceNormalCreator シェーダ*(9-240 ページ).
- *9.15.4 WorldSpaceNormalsCreators C# スクリプト*(9-241 ページ).
- *9.15.5 WorldSpaceNormalCreator シェーダ コード*(9-243 ページ).

9.15.1 接空間からワールド空間への法線変換ツールについて

Ice Cave デモをプロファイリングすると、算術演算パイプラインにボトルネックがあることがわかりました。負荷を軽減するために、Ice Cave デモでは、接空間の法線マップの代わりに、静的ジオメトリのワールド空間の法線マップを使用しています。

接空間の法線マップは、アニメーション オブジェクトや動的オブジェクトには便利ですが、サンプリングされた法線の向きを正しく配置するには、追加の計算が必要です。

Ice Cave デモのジオメトリのほとんどは静的なので、法線マップはワールド空間の法線マップに変換されました。これにより、テクスチャのためにサンプリングされた法線は、ワールド空間で正しい向きで配置されます。Ice Cave デモのライティングはカスタム シェーダで計算されるのに対し、Unity 標準のシェーダは接空間の法線マップを使用するため、この変更が可能となります。

変換ツールは、次のもので構成されています。

- エディタに新しいオプションを追加する C# スクリプト。
- 変換を実行するシェーダ。

このツールは Unity エディタ内からオフラインで実行され、ゲームのランタイム パフォーマンスには影響しません。

9.15.2 C# スクリプト

C# スクリプトは Unity の `Assets/Editor` ディレクトリに配置する必要があります。これにより、スクリプトは Unity エディタ の GameObject メニューに新しいオプションを追加することができます。このディレクトリが存在しない場合には、作成します。

次のコードは Unity エディタで新しいオプションを追加する方法を示しています。

```
[MenuItem("GameObject/World Space Normals Creator")]

static void CreateWorldSpaceNormals ()
{
    ScriptableWizard.DisplayWizard("Create World Space Normal",
    typeof(WorldSpaceNormalsCreator),"Create");
}
```

次の図は、スクリプトによって追加された GameObject メニュー オプションを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

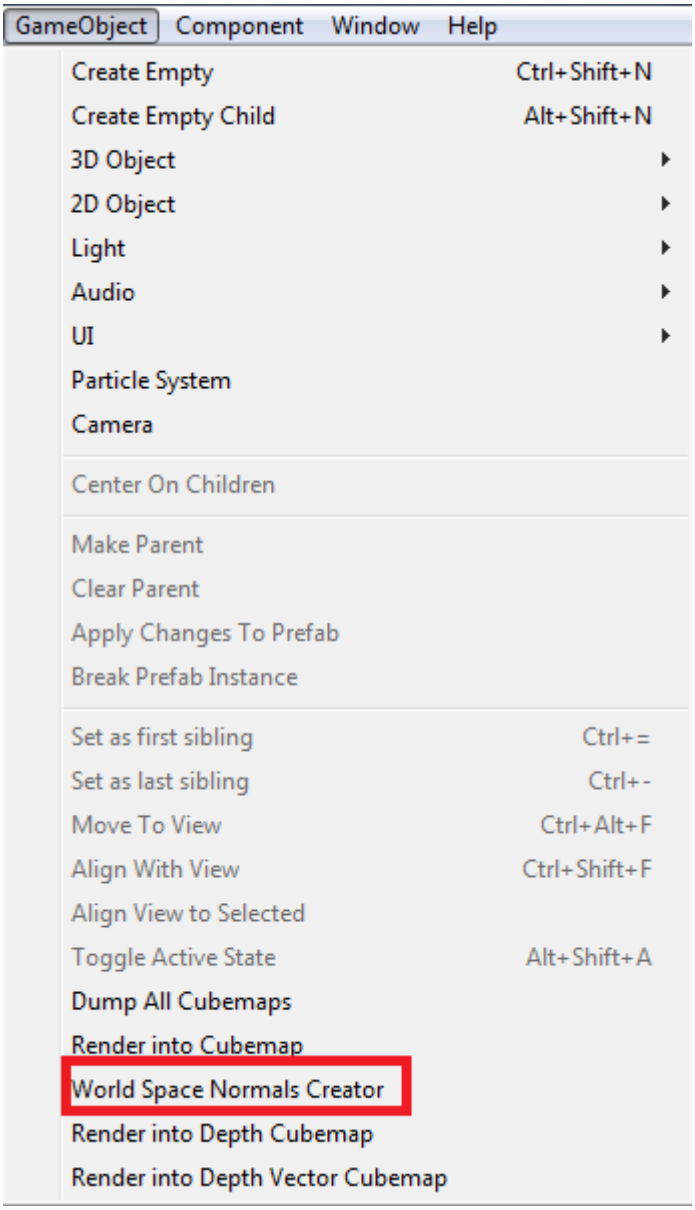


図 9-88 スクリプトによって追加された GameObject メニュー オプション

C# スクリプトで定義されたクラスは、Unity `ScriptableWizard` クラスから派生し、そのメンバの一部にアクセスできます。このクラスから派生して、エディタ ウィザードを作成します。エディタ ウィザードは通常、メニュー項目を使用して開きます。

`OnWizardUpdate` のコードで、`helpString` 変数は、ウィザードが作成するウィンドウに表示されるヘルプ メッセージを保持します。

`isValid` メンバは、すべての正しいパラメータが選択されて [Create] ボタンが使用可能となるときを定義するために使用されます。この場合、`_currentObj` メンバが有効なオブジェクトを指していることを確認するために、そのメンバーがチェックされます。

ウィザード ウィンドウのフィールドは、クラスのパブリック メンバです。この場合、`_currentObj` のみがパブリックであるため、ウィザード ウィンドウには 1 つのフィールドしかありません。

次の図は、カスタムのウィザード ウィンドウを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

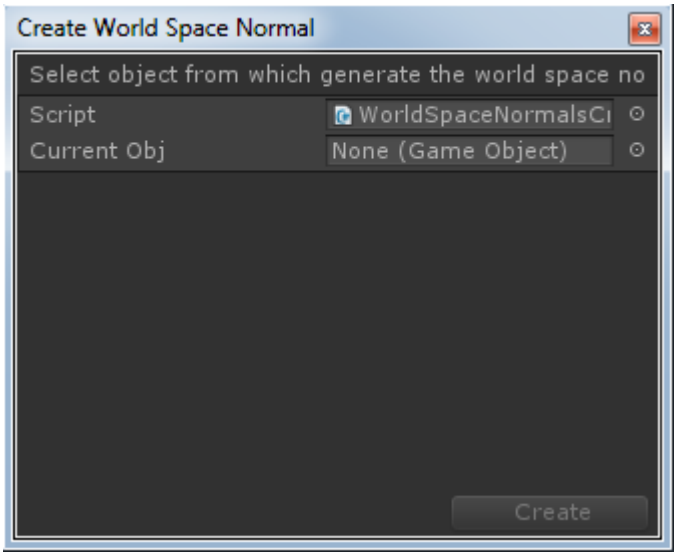


図 9-89 カスタムのウィザード ウィンドウ

オブジェクトが選択され、[Create] ボタンがクリックされると、`OnWizardCreate()` 関数が呼び出されます。

`OnWizardCreate()` 関数は、変換の主な作業を実行します。

法線を変換するため、ツールは新しいワールド空間を `RenderTexture` に垂直にレンダリングする、一時的なカメラを作成します。これを行うため、カメラは正射投影モードに設定され、オブジェクトのレイヤは未使用レベルに変更されます。これにより、オブジェクトが既にシーンの一部であっても、オブジェクトをそのままレンダリングすることができます。

次のコードは、カメラの設定方法を示しています。

```
// アンチエイリアシングを設定する
QualitySettings.antiAliasing = 4;
Shader wns = Shader.Find ("Custom/WorldSpaceNormalCreator");
GameObject go = new GameObject( "WorldSpaceNormalsCam", typeof(Camera) );
_renderCamera = go.GetComponent<Camera> ();
_renderCamera.orthographic = true;
_renderCamera.nearClipPlane = 0.0f;
_renderCamera.farClipPlane = 10f;
_renderCamera.orthographicSize = 1.0f;
int prevObjLayer = _currentObj.layer;
_currentObj.layer = 30; //0x40000000
```

このスクリプトは、変換を実行する置換シェーダを設定します。

```
_renderCamera.SetReplacementShader (wns,null);
_renderCamera.useOcclusionCulling = false;
```

カメラはオブジェクトに向けられます。これにより、錐台のカリングの間は、削除されるオブジェクトがレンダリングされなくなります。

```
_renderCamera.transform.rotation = Quaternion.LookRotation (_currentObj.transform.position -
_renderCamera.transform.position);
```

オブジェクトに割り当てられた各マテリアルについて、スクリプトは `_BumpMap` を探します。このテクスチャは、シェーダのグローバル関数を使用して、置換シェーダのソース テクスチャとして設定されます。

負の方向を指す法線を表現する必要があるため、クリア カラーは `(0.5,0.5,0.5)` に設定されます。

```
foreach (Material m in materials)
{
    Texture t = m.GetTexture("_BumpMap");
    if( t == null )
    {
       Debug.LogError("the material has no texture assigned named Bump Map");
       continue;
    }
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
    Shader.SetGlobalTexture ("_BumpMapGlobal", t);
    RenderTexture rt = new RenderTexture(t.width,t.height,1);
    _renderCamera.targetTexture = rt;
    _renderCamera.pixelRect = new Rect(0,0,t.width,t.height);
    _renderCamera.backgroundColor = new Color( 0.5f, 0.5f, 0.5f);
    _renderCamera.clearFlags = CameraClearFlags.Color;
    _renderCamera.cullingMask = 0x40000000;
    _renderCamera.Render();
    Shader.SetGlobalTexture ("_BumpMapGlobal", null);
```

カメラがシーンをレンダリングした後、ピクセルは読み戻され、PNG イメージとして保存されます。

```
    Texture2D outTex = new Texture2D(t.width,t.height);
    RenderTexture.active = rt;
    outTex.ReadPixels(new Rect(0,0,t.width,t.height), 0, 0);
    outTex.Apply();
    RenderTexture.active = null;
    byte[] _pixels = outTex.EncodeToPNG();
    System.IO.File.WriteAllBytes("Assets/Textures/GeneratedWorldSpaceNormals/"+t.name
    +"_WorldSpace.png",_pixels);
    }
```

カメラのカリング マスクは、16 進形式で表されたバイナリ マスクを使用して、レンダリングするレイヤを指定します。

この場合、レイヤ 30 が使用されています。

```
_currentObj.layer = 30;
```

その 30 番目のビットが 1 に設定されているため、16 進数は 0x40000000 です。

9.15.3 WorldSpaceNormalCreator シェーダ

変換を実装するシェーダのコードはかなり簡単です。実際の頂点位置を使用する代わりに、頂点のテクスチャ座標をその位置として使用します。これにより、テクスチャリングするときと同じように、オブジェクトは 2D 平面に投影されます。

OpenGL パイプラインを正しく動作させるために、UV 座標は標準の `[0,1]` の範囲から `[-1,1]` の範囲に移動し、Y 座標を反転させます。Z 座標は使用されないため、`0` または Near Clip Plane と Far Clip Plane の間の任意の値に設定できます。

```
output.pos = half4(input.tex.x*2.0 - 1.0,((1.0-input.tex.y)*2.0 - 1.0), 0.0, 1.0);
output.tc = input.tex;
```

法線、接線、従法線は、頂点シェーダで計算され、フラグメント シェーダに渡されて、変換が実行されます。

```
output.normalInWorld  = normalize(mul(half4(input.normal, 0.0), _World2Object).xyz);
output.tangentWorld   = normalize(mul(_Object2World, half4(input.tangent.xyz, 0.0)).xyz);
output.bitangentWorld = normalize(cross(output.normalInWorld, output.tangentWorld)
                      * input.tangent.w);
```

フラグメント シェーダ:

1. 法線を接空間からワールド空間に変換します。
2. 法線を `[0,1]` の範囲にスケールします。
3. 新しいテクスチャに法線を出力します。

次のコードはこれを示しています。

```
half3 normalInWorld = half3(0.0,0.0,0.0);
half3 bumpNormal = UnpackNormal(tex2D(_BumpMapGlobal, input.tc));
half3x3 local2WorldTranspose = half3x3( input.tangentWorld,
                                        input.bitangentWorld,
                                        input.normalInWorld);
normalInWorld = normalize(mul(bumpNormal, local2WorldTranspose));
normalInWorld = normalInWorld*0.5 + 0.5;
return  half4(normalInWorld,1.0);
```

次の画像は、処理前の接空間の法線マップを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

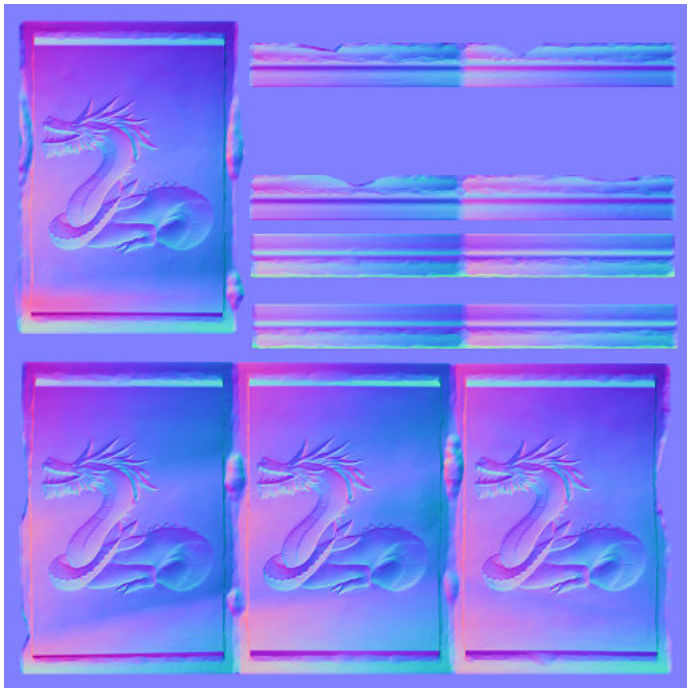


図 9-90 元の接空間の法線マップ

次の画像は、ツールによって生成された、ワールド空間の法線マップを示しています。

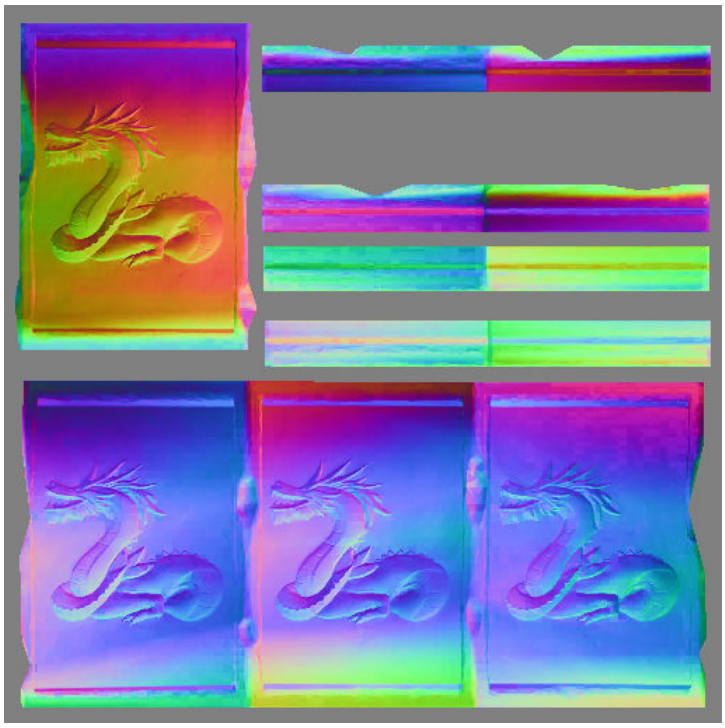


図 9-91 生成されたワールド空間の法線マップ

9.15.4 WorldSpaceNormalsCreators C# スクリプト

これは WorldSpaceNormalsCreators C# スクリプトのコードです。

```
using UnityEngine;
using UnityEditor;
using System.Collections;

public class WorldSpaceNormalsCreator : ScriptableWizard
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
{
    public GameObject _currentObj;

    private Camera _renderCamera;

    void OnWizardUpdate()
    {
        helpString = "Select object from which generate the world space normals";
        if(_currentObj != null)
        {
            isValid = true;
        }
        else
        {
            isValid = false;
        }
    }

    void OnWizardCreate ()
    {
        // アンチエイリアシングを設定する
        QualitySettings.antiAliasing = 4;
        Shader wns = Shader.Find ("Custom/WorldSpaceNormalCreator");
        GameObject go = new GameObject( "WorldSpaceNormalsCam", typeof(Camera) );

        // 新しいカメラを設定して正射投影を実行する
        _renderCamera = go.GetComponent<Camera> ();
        _renderCamera.orthographic = true;
        _renderCamera.nearClipPlane = 0.0f;
        _renderCamera.farClipPlane = 10f;
        _renderCamera.orthographicSize = 1.0f;

        // 現在のオブジェクト レイヤを保存し、それを未使用のレイヤに設定する
        int prevObjLayer = _currentObj.layer;
        _currentObj.layer = 30; //0x40000000

        // カメラに置換シェーダを設定する
        _renderCamera.SetReplacementShader (wns,null);
        _renderCamera.useOcclusionCulling = false;

        // カメラを回転させてオブジェクトを見るようにし、錐台のカリングを避ける
        _renderCamera.transform.rotation = Quaternion.LookRotation
          (_currentObj.transform.position - _renderCamera.transform.position);

        MeshRenderer mr = _currentObj.GetComponent<MeshRenderer> ();
        Material[] materials = mr.sharedMaterials;

        foreach (Material m in materials)
        {
            Texture t = m.GetTexture("_BumpMap");
            if( t == null )
            {
                Debug.LogError("the material has no texture assigned named Bump Map");
                continue;
            }

            // ワールド空間の法線マップをテクスチャにレンダリングする
            Shader.SetGlobalTexture ("_BumpMapGlobal", t);
            RenderTexture rt = new RenderTexture(t.width,t.height,1);
            _renderCamera.targetTexture = rt;
            _renderCamera.pixelRect = new Rect(0,0,t.width,t.height);
            _renderCamera.backgroundColor = new Color( 0.5f, 0.5f, 0.5f);
            _renderCamera.clearFlags = CameraClearFlags.Color;
            _renderCamera.cullingMask = 0x40000000;
            _renderCamera.Render();
            Shader.SetGlobalTexture ("_BumpMapGlobal", null);

            Texture2D outTex = new Texture2D(t.width,t.height);
            RenderTexture.active = rt;
            outTex.ReadPixels(new Rect(0,0,t.width,t.height), 0, 0);
            outTex.Apply();
            RenderTexture.active = null;

            // PNG に保存する
            byte[] _pixels = outTex.EncodeToPNG();
            System.IO.File.WriteAllBytes("Assets/Textures/GeneratedWorldSpaceNormals/"
            +t.name+"_WorldSpace.png",_pixels);
        }
        _currentObj.layer = prevObjLayer;
        DestroyImmediate(go);
    }

    [MenuItem("GameObject/World Space Normals Creator")]
    static void CreateWorldSpaceNormals ()
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
    {
        ScriptableWizard.DisplayWizard("Create World Space Normal",
        typeof(WorldSpaceNormalsCreator),"Create");
    }

}
```

9.15.5 WorldSpaceNormalCreator シェーダ コード

以下に示したのは、WorldSpaceNormalCreator シェーダのコードです。

```
Shader "Custom/WorldSpaceNormalCreator" {
    Properties {
    }
    SubShader {

        Cull off

        Pass
        {
            CGPROGRAM
            #pragma target 3.0
            #pragma glsl
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            uniform sampler2D _BumpMapGlobal;

            struct vin
            {
                half4 tex : TEXCOORD0;
                half3 normal   : NORMAL;
                half4 tangent  : TANGENT;
            };

            struct vout
            {
            half4 pos : POSITION;
            half2 tc : TEXCOORD0;
            half3 normalInWorld : TEXCOORD1;
            half3 tangentWorld : TEXCOORD2;
            half3 bitangentWorld : TEXCOORD3;
            };

            vout vert (vin input )
            {
            vout output;
            output.pos = half4(input.tex.x*2.0 - 1.0,((1.0-input.tex.y)*2.0 - 1.0),
                       0.0, 1.0);
            output.tc  = input.tex;
            output.normalInWorld  = normalize(mul(half4(input.normal, 0.0),
                                  _World2Object).xyz);
            output.tangentWorld   = normalize(mul(_Object2World,
                                half4(input.tangent.xyz, 0.0)).xyz);
            output.bitangentWorld = normalize(cross(output.normalInWorld,
                                output.tangentWorld) * input.tangent.w);

            return output;
            }

            float4 frag( vout input ) : COLOR
            {
            half3 normalInWorld = half3(0.0,0.0,0.0);
            half3 bumpNormal = UnpackNormal(tex2D(_BumpMapGlobal, input.tc));
            half3x3 local2WorldTranspose = half3x3(
                       input.tangentWorld,
                       input.bitangentWorld,
                       input.normalInWorld);
            normalInWorld = normalize(mul(bumpNormal, local2WorldTranspose));
            normalInWorld = normalInWorld*0.5 + 0.5;

            return  half4(normalInWorld,1.0);
            }
            ENDCG
        }
    }
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 10 章
バーチャル リアリティ

この章では、バーチャル リアリティ ハードウェア上で実行されるアプリケーションまたはゲームの適用プロセスと、バーチャル リアリティにおける反射の実装のいくつかの違いについて説明します。

以下のセクションから構成されています。

- *10.1 Unity のバーチャル リアリティ ハードウェア サポート*(10-245 ページ).
- *10.2 Unity VR への移植プロセス*(10-246 ページ).
- *10.3 VR への移植時に考慮すべき事項*(10-249 ページ).
- *10.4 VR での反射*(10-251 ページ).
- *10.5 結果*(10-256 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

10.1 Unity のバーチャル リアリティ ハードウェア サポート

Unity はいくつかの種類の*バーチャル リアリティ*(VR) ハードウェアをサポートしています。いくつかのデバイスでは Unity がネイティブ サポートしています。それ以外のデバイスではプラグインによりサポートされます。

Unity VR をネイティブ サポートしているデバイスでは、プラグインによりサポートされるデバイスよりも優れたパフォーマンスが得られます。Unity VR をネイティブ サポートしているデバイスでは Unity の内部最適化が実装されているためです。

次のデバイスでは、Unity VR をネイティブ サポートしています。

- Oculus Rift。
- Samsung Gear VR。
- PlayStation VR。
- Microsoft Hololens。

次のデバイスでは、プラグインにより Unity VR をサポートしています。

- Google Cardboard。
- Moverio。
- HTC Vive および SteamVR プラットフォームがサポートする他のデバイス。

これ以外に、Unity 用の *Open Source Virtual Reality* (OSVR) プラグインを使ってサポートされるデバイスもあります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

10.2 Unity VR への移植プロセス

アプリケーションやゲームをネイティブの Unity VR に移植するには、複数の段階のプロセスが必要です。

アプリケーションを Unity VR に移植するために必要な段階は次のとおりです。

1. Unity バージョン 5.1 以降をインストールします。Unity バージョン 5.1 以降では、VR をネイティブサポートしています。
2. 必要に応じて、適切な Web サイトからデバイスの署名ファイルを入手し、デバイス上の必要なフォルダに配置します。

   Ice Cave デモを実行している Samsung Gear VR の場合、Oculus の開発者 Web サイト *https://developer.oculus.com/osig* を参照します。Samsung デバイス用の署名ファイルは `Plugins/Android/assets` フォルダに配置する必要があります。
3. Unity で [Open File] > [Build Settings] > [Player Settings] の順に選択します。[Player Settings] ウィンドウが表示されます。[Other Settings] セクションの [Virtual Reality Supported] オプションを有効化します。

   以下の図は、このウィンドウのスクリーンショットを示しています。

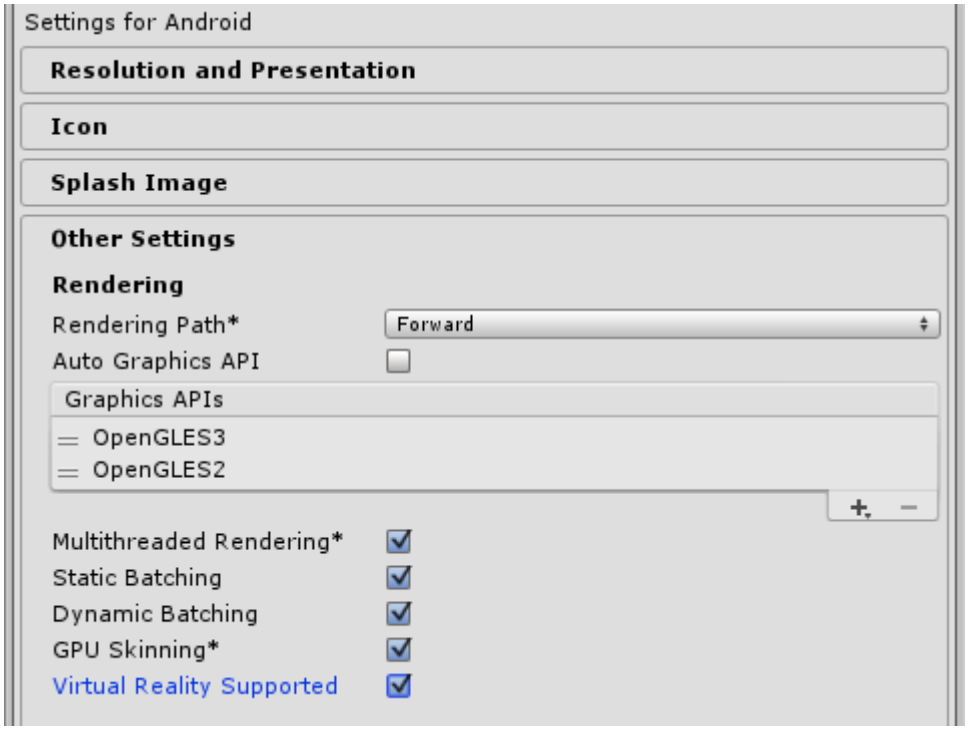


図 10-1 [Player Settings] ウィンドウ

4. カメラのペアレントを設定します。すべてのカメラ コントロールは、カメラの位置と向きをこのカメラのペアレントに設定する必要があります。
5. 必要に応じて、カメラ コントロールを VR ヘッドセットのタッチ パッドに関連付けます。
6. Android デバイスでは、[開発者向けオプション] メニューから [USB デバッグ] を有効化する必要があります。

   次の図は [開発者向けオプション] メニューを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

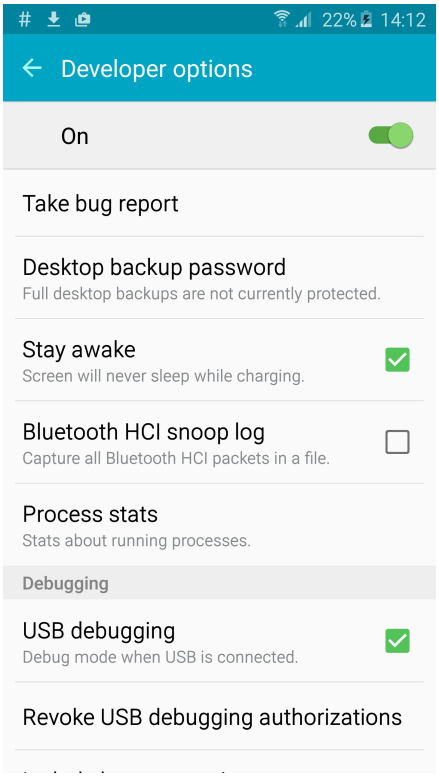


図 10–2 ［開発者向けオプション］メニュー

7. アプリケーションをビルドし、デバイスにインストールします。
8. アプリケーションを起動します。

Samsung デバイスでアプリケーションを起動すると、デバイスをヘッドセットに挿入するように指示されます。デバイスで VR の準備が整っていない場合は、ネットワークに接続して Samsung VR ソフトウェアをダウンロードするように指示されます。

以下のサブセクションから構成されています。

- *10.2.1 Samsung Gear VR の開発者モードの有効化*（10-247 ページ）.

10.2.1 Samsung Gear VR の開発者モードの有効化

開発者モードは、デバイスを VR ヘッドセットに挿入することなく、実行中の VR アプリケーションを視覚化するのに役立ちます。

Samsung Gear VR の開発者モードは、署名済みの VR アプリケーションがインストールされている場合に有効化できます。署名済みの VR アプリケーションをインストールしていない場合には、署名済み VR アプリケーションをインストールして、このモードを有効化します。

開発者モードを有効化するには、以下の手順に従います。

1. Samsung のデバイスで、**［設定］** > **［アプリケーション マネージャー］** > **［Gear VR サービス］** の順に選択します。
2. **［ストレージの管理］** を選択します。
3. ［VR サービスのバージョン］を 6 回タップします。
4. スキャン プロセスの完了を待ちます。開発者モードの切り替えボタンが表示されます。

注

開発者モードを使用すると、使用していないときにヘッドセットをオフにするすべての設定が無効になるため、スマートフォンのバッテリ寿命が短くなります。

次の図は、Samsung Gear VR での Ice Cave の開発者モード表示のスクリーンショットのサンプルです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

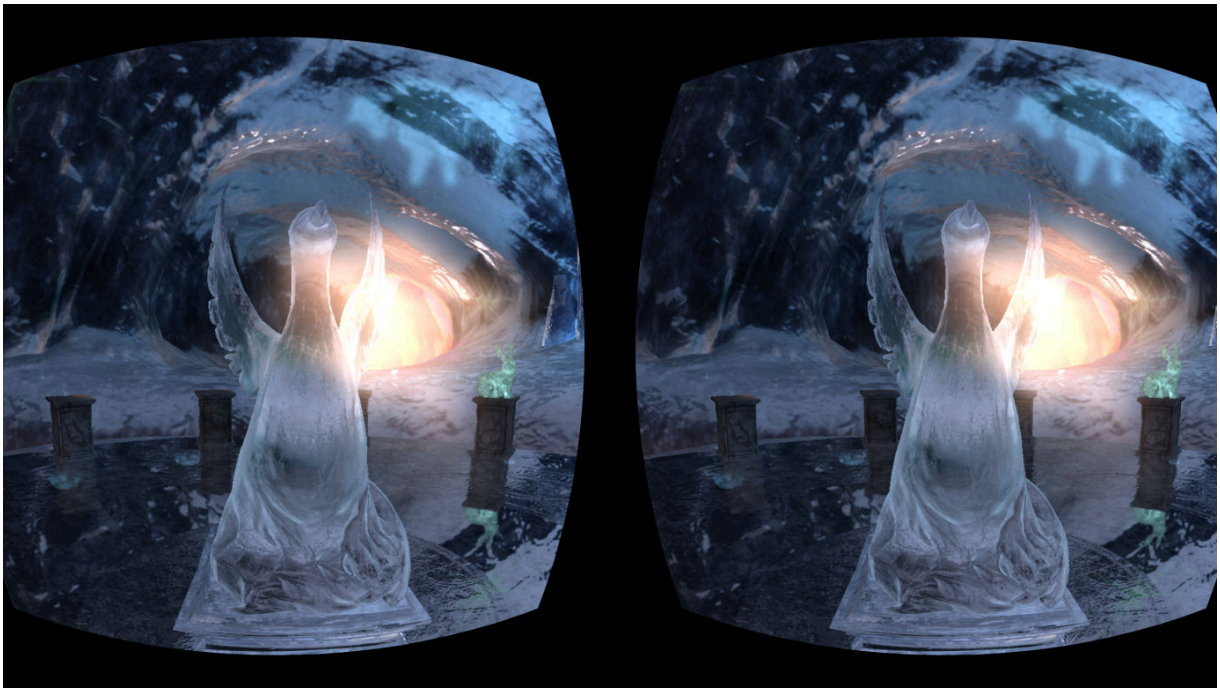


図 10–3 Samsung Gear VR の開発者モードで実行されている VR アプリケーションのスクリーンショットのサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

10.3 VR への移植時に考慮すべき事項

VR は、他の種類のアプリケーションとは非常に異なるユーザ エクスペリエンスを提供します。このため、VR 以外のアプリケーションやゲームでは機能することが、VR では機能しない場合があります。

何人かのユーザでアプリケーションをテストし、快適性のレベルを判断して、コードを調整して快適なエクスペリエンスを見つけます。

VR 以外のゲームで快適なカメラ アニメーションは、VR のゲームでは不快となる場合があります。たとえば、VR なしの Ice Cave デモには、カメラのアニメーション モードがあります。ユーザによってはこれを不快に感じ、特にカメラが後方に移動したときに、乗り物酔いのように感じる場合があります。このモードを削除することにより、不快なエクスペリエンスを避けることができます。

VR 以外のアプリケーションでは、スマートフォンのタッチ スクリーンを使用したり、スマートフォンを傾けることによって制御を行うことができます。これらの制御メカニズムは、VR アプリケーションでは可能でない場合があります。たとえば、Ice Cave デモのオリジナル版では、2 つの仮想ジョイスティックを使用してカメラを制御していますが、VR デバイスではタッチスクリーンにアクセスできないため、この方法は使えません。Ice Cave VR デモは、ヘッドセットの側面にタッチパッドを備えた Samsung Gear VR で動作するように設計されています。タッチ スクリーンの代わりにタッチパッドを使用することにより、この問題を解決しています。

次の図は、Samsung Gear VR ヘッドセットのタッチパッドを示しています。

図 10-4 タッチパッド付きの Samsung Gear VR ヘッドセット

非 VR のアプリケーションでは機能する視覚効果が、VR アプリケーションでは正しく見えない場合があります。たとえば、非 VR の Ice Cave デモでは、太陽の位置とカメラの向きに基づいて強度を変更するダーティ レンズ エフェクトを使用しています。VR でこのエフェクトをテストしたユーザにより、これは正しく表示されないことがわかったため、削除されました。

以下のサブセクションから構成されています。

- *10.3.1 カメラの外部デバイス制御*(10-249 ページ).

10.3.1 カメラの外部デバイス制御

VR は、通常はスマートフォンやモバイル デバイスには関連しない制御方法を活用できます。それらの方法の中には、アプリケーションのターゲットによっては検討する価値があるものがあります。

Bluetooth を使用して VR デバイスに接続できるコントローラを活用できます。これを実装するために、Ice Cave デモでは、Android の Bluetooth イベントを解釈するように Unity の機能を拡張する、カスタム プラグインを使用しています。これらのイベントは、カメラの動きをトリガします。

次の図は、Ice Cave デモで動作する Bluetooth コントローラを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 10-5 Ice Cave デモで動作する Bluetooth コントローラ

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

10.4 VR での反射

反射は、VR アプリケーションにとって重要です。現実の世界には多くの反射が含まれているため、ゲームやアプリケーションで反射がない場合には、人々の気に留まります。

VR での反射には、従来のゲームと同じ技法を活用できますが、ユーザが見るステレオ視覚出力で動作するように修正する必要があります。

反射を適切に実装すると、アプリケーションやゲームがよりリアルになり、没入感を感じさせることができます。ただし、VR で反射を実装する際には、考慮する必要がある問題がいくつかあります。

以下のサブセクションから構成されています。

- *10.4.1 ローカル キューブマップを使用した反射*(10-251 ページ).
- *10.4.2 さまざまな種類の反射の組み合わせ*(10-251 ページ).
- *10.4.3 ステレオ反射*(10-251 ページ).

10.4.1 ローカル キューブマップを使用した反射

ローカル キューブマップを使用して反射を作成できます。この方法では、フレームごとに反射テクスチャを作成するのではなく、プリレンダリングされたキューブマップから反射テクスチャをフェッチします。この方法は、キューブマップが生成された場所とシーン境界ボックスに基づいて反射ベクトルに局所補正を適用し、その情報を使用して適切なテクスチャを取得します。

ローカル キューブマップを使用して生成した反射では、各フレームの実行時に反射を生成する場合に発生する場合があるピクセルの不安定性やピクセルの揺らぎが生じることはありません。

ローカル キューブマップを使用した反射の詳細については、*9.2 ローカル キューブマップによる反射の実装*(9-157 ページ) を参照してください。

10.4.2 さまざまな種類の反射の組み合わせ

最高の性能と効果を得るには、さまざまな反射生成手法が必要です。必要な手法は、反射面の形状や、反射面と反射オブジェクトが静的であるか動的であるかによって異なります。さまざまな反射生成手法の結果を組み合わせて、ユーザが見る結果を生成する必要があります。

さまざまな種類の反射の組み合わせについての情報は、*9.3 反射の結合*(9-174 ページ)を参照してください。

10.4.3 ステレオ反射

VR 以外のゲームでは、カメラの視点は 1 つだけです。VR では、それぞれの目に 1 つのカメラの視点があります。これは、反射の計算をそれぞれの目に個別に行う必要があることを意味します。

両方の目に同じ反射が表示されている場合には、ユーザはすぐに反射に深さがないことに気付きます。これはユーザの期待に反し、没入感を損ない、VR 体験の質に悪影響を及ぼします。

この問題を解決するには、2 つの反射の計算を行い、ユーザがゲーム内を見回している間、各目の位置を適切に調整して表示する必要があります。

Ice Cave デモでは、これらの反射を実装するために、動的オブジェクトからの平面反射のための 2 つの反射テクスチャと、静的オブジェクト反射のために 1 つのローカル キューブマップからテクスチャをフェッチする 2 つの異なる局所補正された反射ベクトルを使用しています。

反射は、動的オブジェクトまたは静的オブジェクトとなります。それぞれの種類の反射が VR で機能するには、それぞれで一連の変更が必要となります。

Unity VR でのステレオ平面反射の実装

VR ゲームにステレオ反射を実装するには、非 VR のゲームのコードに、いくつかの調整を加える必要があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

まず最初に、Unity でバーチャル リアリティのサポートが有効化されていることを確認します。これには、[Build Settings] > [Player Settings] > [Other Settings] の順に選択し、[Virtual Reality Supported] のチェックボックスをオンにします。

動的なステレオ平面反射

動的な反射では、両目に適切な結果を生成するためには、変更が必要となります。

2 つのカメラを作成し、レンダリングするカメラのそれぞれにターゲット テクスチャを作成する必要があります。両方のカメラを無効化して、レンダリングがプログラムで実行されるようにします。それから、次のスクリプトを両方にアタッチします。

```
void OnPreRender(){
        SetUpReflectionCamera();
        // 屈曲を反転させる
        GL.invertCulling = true;
}
void OnPostRender(){
        // 屈曲を復元する
        GL.invertCulling = false;
}
```

このスクリプトは、メイン カメラの位置と向きを使用して、反射カメラを配置します。これを行うには、左と右の反射カメラがレンダリングを行う直前に `SetUpReflectionCamera()` 関数を呼び出します。次のコードは、この関数の実装方法を示しています。

```
public  GameObject reflCam;
public  float clipPlaneOffset ;
…
private void SetUpReflectionCamera(){
        // 反射面を見つける: ワールド空間における位置と法線
        Vector3 pos = gameObject.transform.position;

        // Y 軸方向の反射平面法線
        Vector3 normal = Vector3.up;
        float d = -Vector3.Dot(normal, pos) - clipPlaneOffset;
        Vector4 reflPlane = new Vector4(normal.x, normal.y, normal.z, d);
        Matrix4x4 reflection = Matrix4x4.zero;
        CalculateReflectionMatrix(ref reflection, reflPlane);

        // メイン カメラの位置と向きを考慮した反射カメラの更新
        // ビュー行列の設定
        Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;
        reflCam.GetComponent<Camera>().worldToCameraMatrix = m;

        // 投影行列の設定
        reflCam.GetComponent<Camera>().projectionMatrix = Camera.main.projectionMatrix;
}
```

この関数は、反射カメラのビュー行列と投影行列を計算します。これは メイン カメラ `worldToCameraMatrix` のビュー行列に適用する反射変換を決定します。

ぞれぞれの目のカメラの位置を設定するには、下記のコードを次の行の後に追加します `Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;`:

左目用

```
m[12] += stereoSeparation;
```

右目用

```
m[12] -= stereoSeparation;
```

シフト値 `stereoSeparation` は 0.011 です。`stereoSeparation` 値は、目の分離値の半分です。

メイン カメラに別のスクリプトをアタッチして、左右の反射カメラのレンダリングを制御します。次のコードは、このスクリプトの Ice Cave での実装を示しています。

```
public class RenderStereoReflections : MonoBehaviour
{
    public GameObject reflectiveObj;
    public GameObject leftReflCamera;
    public GameObject rightReflCamera;
    int eyeIndex = 0;
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
    void OnPreRender(){
        if (eyeIndex == 0){
            // 左カメラのレンダリング
            leftReflCamera.GetComponent<Camera>().Render();
            reflectiveObj.GetComponent<Renderer>().material.SetTexture(
                "_DynReflTex", leftReflCamera.GetComponent<Camera>().targetTexture);
        }
        else{
            // 右カメラのレンダリング
            rightReflCamera.GetComponent<Camera>().Render();
            reflectiveObj.GetComponent<Renderer>().material.SetTexture(
                "_DynReflTex", rightReflCamera.GetComponent<Camera>().targetTexture);
        }
        eyeIndex = 1 - eyeIndex;
    }
}
```

このスクリプトは、メイン カメラの `OnPreRender()` コールバック関数で、左右の反射カメラのレンダリングを処理します。このスクリプトは、左目用に 1 回、次に右目用に 1 回呼び出されます。`eyeIndex` 変数は、各反射カメラの正しいレンダリング順を割り当て、正しい反射をメイン カメラのそれぞれの目に適用します。コールバック関数が最初に呼び出されるときには、それは左目用であると想定します。これが、Unity が `OnPreRender()` メソッドを呼び出す順序です。

それぞれの目に異なるテクスチャが使用されていることを確認する

スクリプトによって、それぞれの目に対して異なるレンダリング テクスチャが正しく生成されていることを確認することが重要です。

それぞれの目に対して正しいテクスチャが表示されているかどうかをテストするには、次のようにします。

手順

1. スクリプトを変更して、シェーダに `eyeIndex` 値をユニフォームとして渡すようにします。
2. 反射テクスチャに 2 色を使用し、それらをそれぞれの `eyeIndex` 値に使用します。

スクリプトが正常に動作している場合、2 つの異なる安定した反射が表示される、次のスクリーンショットのような出力になります。

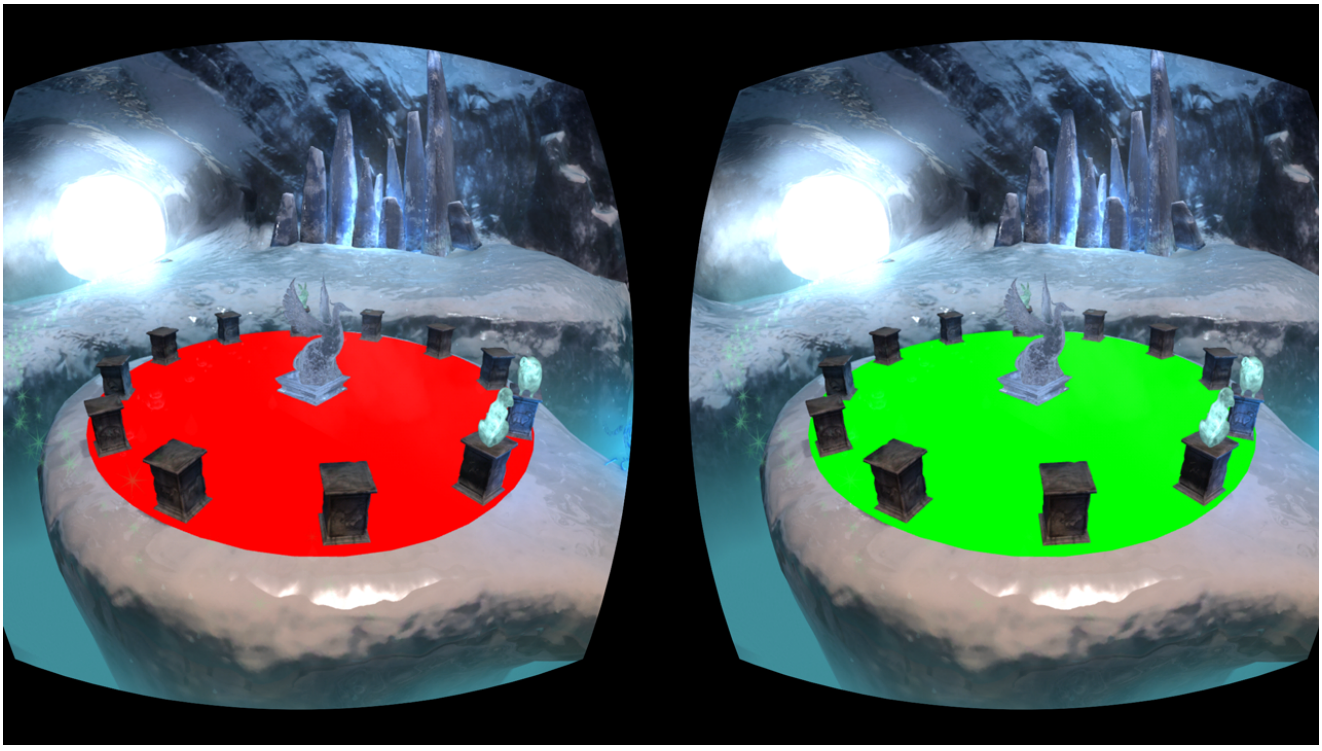


図 10-6 正しい反射テクスチャ出力のチェックのサンプル

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

静的なステレオ反射

キューブマップを使用すると、静的オブジェクトからステレオ反射を効率的に作成できます。唯一の違いは、キューブマップからテクセルをフェッチするために、それぞれの目に 1 つずつ、合わせて 2 つの反射ベクトルを使用する必要があることです。

Unity は、シェーダ内でワールド座標系によるカメラ位置にアクセスするための、次のビルトイン値を提供します。

`_WorldSpaceCameraPos`.

しかし VR では、左右のカメラの位置が必要となります。`_WorldSpaceCameraPos` は、左右のカメラの位置を提供することはできません。したがって、スクリプトを使用して左右のカメラの位置を計算し、結果をシェーダに単一のユニフォームとして渡す必要があります。

シェーダで、カメラの位置の情報を渡すことができる、新しいユニフォームを宣言します。

```
uniform float3 _StereoCamPosWorld;
```

左右のカメラの位置を計算する最も良い場所は、メイン カメラにアタッチされているスクリプトの中です。それはメイン カメラのビュー行列に簡単にアクセスできるためです。次のコードは、`eyeIndex = 0` の場合にこれを行う方法を示しています。

このコードは、メイン カメラのビュー行列を変更して、ローカル座標で左目の位置を設定します。ワールド座標系での左目の位置が必要なので、逆行列を使用します。左目のカメラ位置は、ユニフォームの `_StereoCamPosWorld` を通してシェーダに渡されます。

```
Matrix4x4 mWorldToCamera = gameObject.GetComponent<Camera>().worldToCameraMatrix;
mWorldToCamera[12] += stereoSeparation;
Matrix4x4 mCameraToWorld = mWorldToCamera.inverse;
Vector3 mainStereoCamPos = new Vector3(mCameraToWorld[12], mCameraToWorld[13],
          mCameraToWorld[14]);
reflectiveObj.GetComponent<Renderer>().material.SetVector("_StereoCamPosWorld",
          new Vector3 (mainStereoCamPos.x, mainStereoCamPos.y, mainStereoCamPos.z));
```

このコードは右目用では、ステレオ分離を `mWorldToCamera[12]` に加算するのでなく減算する点のみが異り、他は同じです。

頂点シェーダでは、次の行を見つける必要があります。これは、ビュー ベクトルの計算を行います。

```
output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
```

この行を、ワールド座標で新しい左と右の目のカメラの位置を使用する次の行に置き換えます。

```
output.viewDirInWorld = vertexWorld.xyz - _ StereoCamPosWorld;
```

ステレオ反射が実装されているときは、アプリケーションがエディタモードで実行されているときに表示されます。これは、反射テクスチャが左目から右目に繰り返し変化するときに点滅するためです。VR デバイスではそれぞれの目に異なるテクスチャが使用されているため、この点滅は見えません。

ステレオ反射の最適化

それ以上の最適化がない場合、ステレオ反射の実装は常に実行されます。これは、反射が見えない場合には、その処理時間が無駄になることを意味します。

反射の処理が実行される前に、反射面が見えるかどうかを確認するコードを挿入します。これを行うには、次のサンプル コードのようなコードを反射オブジェクトにアタッチします。

```
public class IsReflectiveObjectVisible : MonoBehaviour
{
        public bool reflObjIsVisible;

        void Start(){
                reflObjIsVisible = false;
        }

        void OnBecameVisible(){
                reflObjIsVisible = true;
        }
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
        void OnBecameInvisible(){
                reflObjIsVisible = false;
        }
}
```

このクラスを定義した後に、メイン カメラにアタッチされたスクリプトで次の `if` ステートメントを使用して、反射オブジェクトが見える場合のみステレオ反射の計算を実行するようにします。

```
void OnPreRender(){
        if (reflectiveObjetc.GetComponent<IsReflectiveObjectVisible>().reflObjIsVisible){
        …
        }
}
```

コードの残りはこの `if` ステートメントの中に含めます。この `if` ステートメントは、`IsReflectiveObjectVisible` クラスを使用して反射オブジェクトが見えるかどうかをチェックします。反射が見えない場合には、その計算は行われません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

10.5 結果

これらによって、没入感にあふれたステレオ反射を実装する VR 版のゲームを作成でき、VR ユーザの全体的な体験を向上できます。ステレオ反射が実装されていない場合には、ユーザは反射に深さがないことに気づきます。ステレオ反射に変更することにより、ユーザ エクスペリエンスを大幅に向上できます。

次の図は、開発者モードで実行されている Ice Cave デモのスクリーンショットのサンプルです。ステレオ反射が実装されていることを示しています。

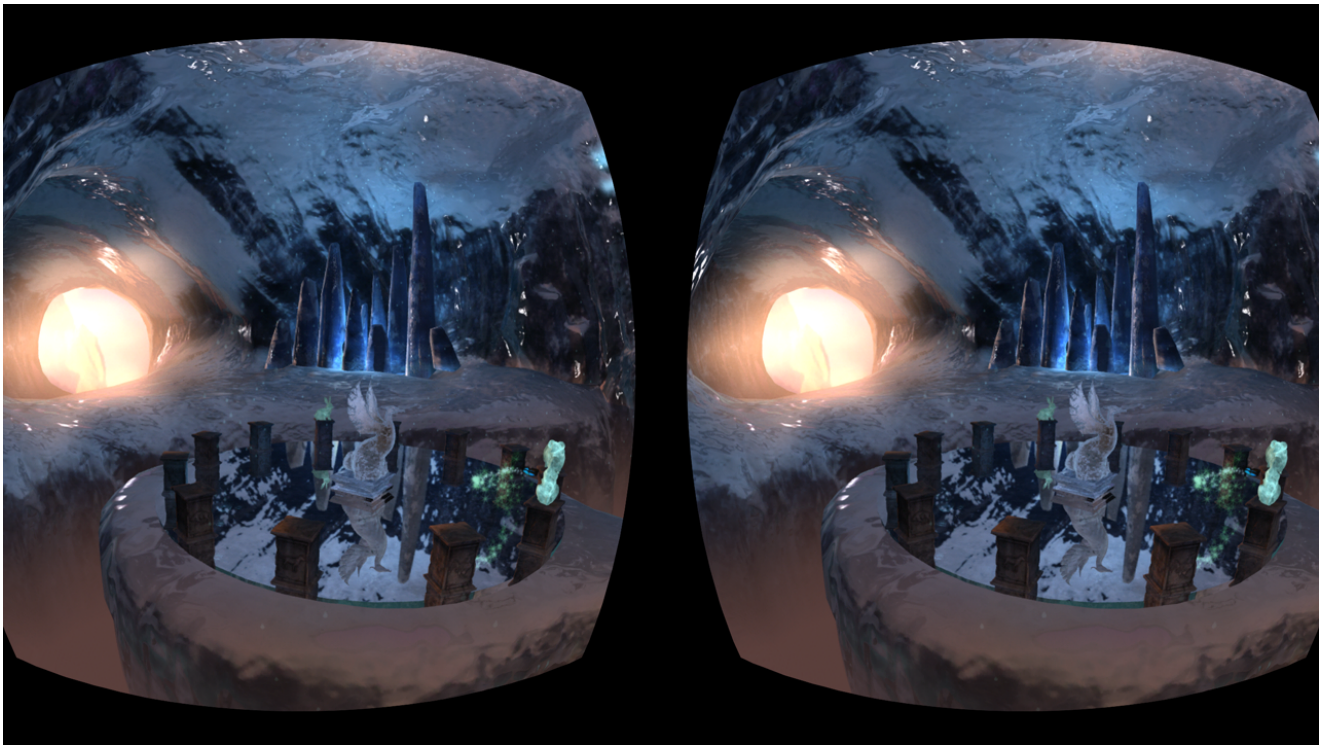


図 10-7 Ice Cave デモのスクリーンショット

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 11 章
高度な VR グラフィックス手法

この章では、バーチャル リアリティ アプリケーションのグラフィック性能を向上させるために使用できるさまざまな技法について説明します。

注意

このコンテンツの最新バージョンは、Arm の開発者 Web サイト(*https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/advanced-vr-graphics-techniques*)にあります。

以下のセクションから構成されています。

- *11.1 エイリアシング*(11-258 ページ).
- *11.2 マルチサンプル アンチエイリアシング*(11-260 ページ).
- *11.3 ミップマッピング*(11-261 ページ).
- *11.4 詳細レベル*(11-263 ページ).
- *11.5 色空間*(11-266 ページ).
- *11.6 テクスチャ フィルタリング*(11-267 ページ).
- *11.7 アルファ合成*(11-270 ページ).
- *11.8 レベル設計*(11-274 ページ).
- *11.9 バンディング*(11-276 ページ).
- *11.10 バンプ マッピング*(11-278 ページ).
- *11.11 影*(11-280 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.1 エイリアシング

エイリアシングは、オブジェクトを忠実に再作成するためのサンプルが十分にない状態で情報を取得しようとしたときに発生します。ほとんどの実用的な用途では、これは無視できない信号処理の側面です。

オーディオとビデオでは、サウンドまたは色の未加工データをサンプリングして個々のデジタル信号に変換するときに、エイリアシングが発生します。エイリアシングは高いオーディオ周波数や、ビデオの細かいディテールで顕著に現れます。

エイリアシングは、長い間隔を空けて未加工データをサンプリングしたときに、コンテキスト内の最高周波数の半分未満のレートで発生します。グラフィックスでは、表示を形成するピクセルの離散グリッドに逆らって移動するカメラからのオブジェクトを表示するラスタライズ プロセスで最も顕著になります。

さらに、グラフィックスのラスタライズ プロセスでは、表示を形成するピクセルの離散グリッドに逆らって移動するカメラによって捉えられたオブジェクトを表示することによる直接的な結果としてエイリアシングが発生します。

VR 環境には、特に顕著に現れる 2 つのエイリアシング バリアントがあります。ジオメトリ エイリアシングと鏡面反射エイリアシングです。

ジオメトリ エイリアシングは、比較的低周波数のピクセル サンプルに表示されているシーンの色空間に高周波数の入力信号が含まれている場合に最も顕著になります。たとえば、2 つの対比色間で急速に移行する場合などです。

このエイリアシングは、カメラの移動時にゆっくり進んでいるように見える直線で特に顕著になります。このジオメトリ エイリアシングのサンプルは、次の画像のシェイプの赤いエッジに見られます。

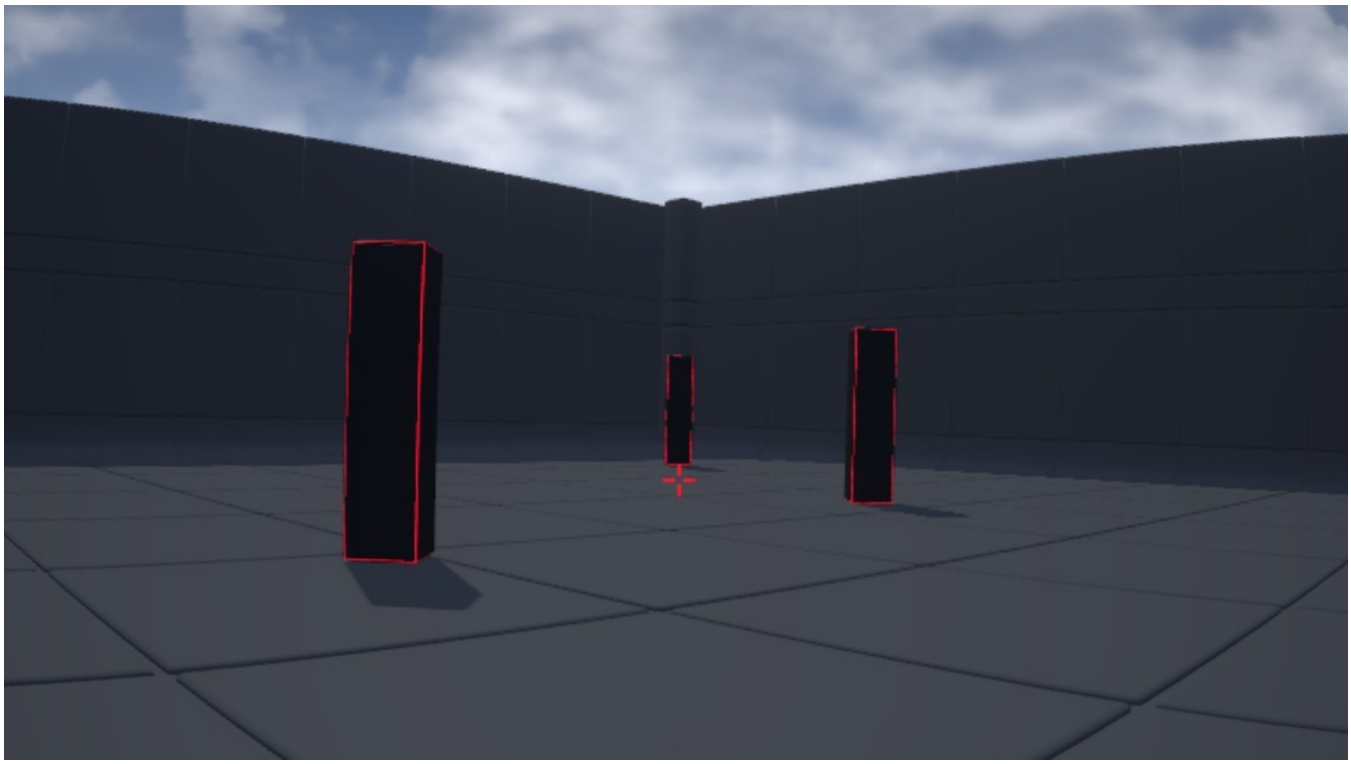


図 11-1 ジオメトリ エイリアシングのサンプル

ジオメトリ エイリアシングは、ポリゴン数の多いメッシュが低解像度のディスプレイにレンダリングされ、少ないピクセル数の幅を持つポリゴンの量によって過度にディテールが示されたときに発生することもあります。ポリゴンが多いことは、多くのエッジが存在することを意味しますが、それらのエッジをレンダリングするためのピクセル数が少ない場合に、さらなるエイリアシングが発生します。

また、高密度の三角形は、GPU でのラスタライズには非効率であり、最終的な画像に有意な違いをもたらさない、小さすぎるジオメトリに GPU が多くのサイクルを費やすことになります。また、ジオメトリのラス

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

タライズに GPU が過剰なサイクルを費やすことで、フラグメント シェーダの実行に使用できるサイクルがほとんどなくなります。これは、GPU ではフラグメント シェードの実行時よりも、ジオメトリのラスタライズに多くのサイクルを消費しなければならないためです。

鏡面反射エイリアシングは、カメラまたはオブジェクトが移動する際に、現れたり消えたりするシャープなハイライトを持つオブジェクトから発生します。これは、鏡面反射効果が 1 つのフレームに存在して次のフレームには存在しない結果、フレーム間で揺らいでいるように見えるピクセルとして現れます。この揺らぎ効果は、ユーザーにとって厄介なものです。これによってユーザーは気が散り、没入感が損なわれ、ユーザー エクスペリエンスの質が低下します。以下の画像の壁にある隆線は、鏡面反射エイリアシングを示しています。

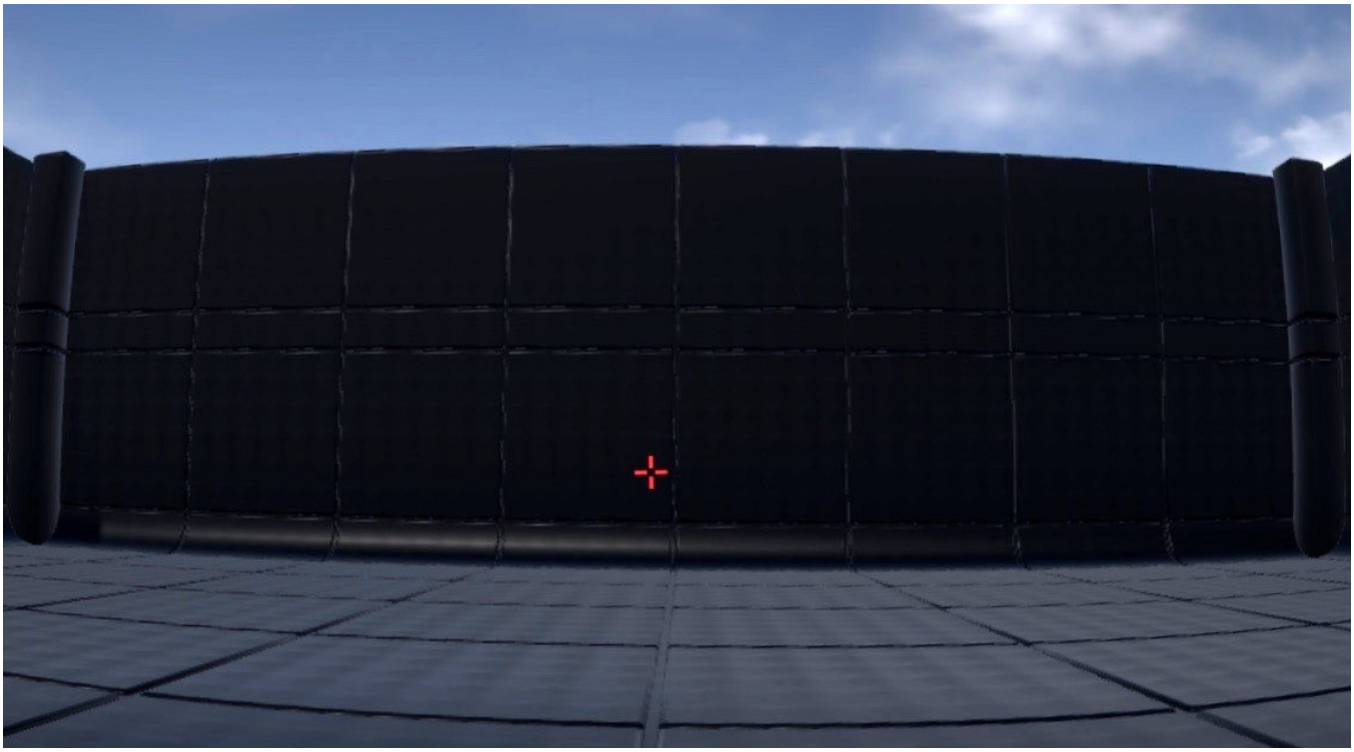


図 11-2 鏡面反射エイリアシングのサンプル

スーパーサンプリングなどの、一般的なアンチエイリアシング技術は計算が複雑であるため、モバイルでは使用できません。*マルチサンプリング アンチエイリアシング*（MSAA）は効果的かつ効率的ですが、ジオメトリ エイリアシングにしか適用されないため、鏡面反射エイリアシングなどの、シェーダ内で発生するアンダーサンプリング アーティファクトには効果がありません。

各種のエイリアシングを軽減するにはさまざまな技術が必要になります。コンピュータ グラフィックス、特に VR では多くの種類のエイリアシングが発生するためです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.2 マルチサンプル アンチエイリアシング

マルチサンプル アンチエイリアシング(MSAA) は、スーパーサンプリングをさらに効率化したアンチエイリアシング技法です。スーパーサンプリングでは、画像は高い内部解像度でレンダリングしてから、その高解像度で各ピクセルに対してフラグメント シェードを実行することにより、ディスプレイ解像度にダウンスケールします。

通常、MSAA は頂点シェードを実行しますが、その後、各ピクセルがサブサンプルに分割され、サブサンプルのビット単位カバレッジ マスクを使用してテストされます。いずれかのサブサンプルがこのカバレッジ テストに合格すると、フラグメント シェードが実行され、カバレッジ テストに合格した各サブサンプルにフラグメント シェーダの結果が保存されます。

MSAA では、ジオメトリック エイリアシングを軽減できるのは 2 つの三角形の交点のみですが、ピクセルごとにフラグメント シェーダを 1 回だけ実行することにより、スーパーサンプリングよりも効率がかなり高くなります。

VR アプリケーションでは、三角形のエッジに沿った「ギザギザ」の減少につながる 4 倍の MSAA がもたらす品質面での向上は、コスト面をはるかに上回るため、できる限り使用してください。

Mali GPU は、4 倍の MSAA を使用した場合のフル フラグメント スループット用に設計されているため、三角形のエッジに沿って生成された追加のフラグメントによる性能への影響は最低限で済みます。

以下のサブセクションから構成されています。

- *11.2.1 Unity でのマルチサンプル アンチエイリアシングの実装*(11-260 ページ).

11.2.1 Unity でのマルチサンプル アンチエイリアシングの実装

Unity では、URP を使用している場合、*マルチサンプル アンチエイリアシング*(MSAA) は *[Universal Render Pipeline]*(URP) 設定でセットアップします。それ以外の場合は、[Project Quality Settings] パネルでセットアップします。

URP パネルを使用して MSAA を有効にするには、次の手順に従います。

1. [Assets] ウィンドウに移動します。
2. [Inspector] ウィンドウの [Quality] ドロップダウンをクリックします。
3. ここでドロップダウン メニューから必要な MSAA レベルを選択します。

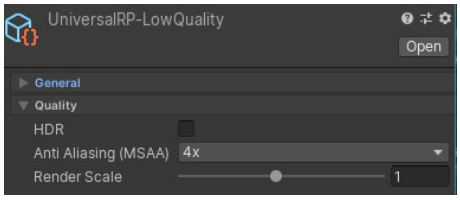


図 11-3 [URP MSAA settings] パネル

[Project Quality Settings] パネルを使用して MSAA を有効にするには、次の手順に従います。

1. [Edit]、[Project Settings] の順に選択します。
2. [Quality] を選択すると、[Project Quality Settings] パネルが開きます。
3. [Anti Aliasing] ドロップダウンを選択し、適切な設定を選びます。できる限り、[4x Multi Sampling] を選択してください。

注

すべての品質レベルに対して、必ずアンチエイリアシング値を設定してください。

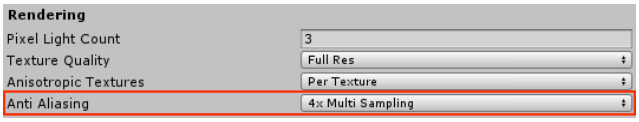


図 11-4 MSAA 設定

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.3 ミップマッピング

ミップマッピングは、高解像度のテクスチャをダウンスケールおよびフィルタ処理し、その後の各ミップ レベルを以前のレベルの面積の 4 分の 1 にする技術です。つまり、テクスチャおよび生成されるミップすべてに必要なサイズは、元のテクスチャ サイズの 1.5 倍以下になります。

ミップマップは、アーティストが手動で作成することも、コンピュータで生成することもでき、GPU にアップロードされます。その後、実行されているサンプリングに最適なミップが GPU によって選択されます。小さなミップ レベルからサンプリングを行うことで、テクスチャのエイリアシングを最小限に抑え、サーフェスにおけるテクスチャの定義を維持し、遠くのサーフェスでモアレ パターンが形成されるのを防ぐことができます。

図 11-5 1x1 までの 9 つのミップ レベルを示した 512x512 テクスチャ

単一のテクスチャ アトラス内に複数の連続するミップマップ レベルを格納する際は注意が必要です。フォービエイテッド レンダリングを使用すると、ビジュアルの問題が発生する場合があるためです。

フォービエイテッド レンダリングではアイトラッキングを使用して、ユーザーが現在見ている場所を高品質のグラフィックスで表示します。

1 つのタイルがネイティブ解像度でレンダリングされ、その近隣のタイルが 4 分の 1 の解像度でレンダリングされた場合、問題が発生します。これは、近隣のタイルが 2 レベル低いミップマップからテクスチャをサンプリングすることを意味します。

結果として生じる 1 つの問題は、サンプリング中に発生します。たとえば、テクスチャのフィルタリング中にテクセルがその近隣のテクセルによってぼけて表示され、テクスチャ アトラス内の近隣のレベルの色が誤ってにじむ場合があります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

その結果、2 つの異なるフォービエイテッド領域内の、それぞれ異なるタイルにある 2 つの近隣のピクセルの色は異なります。テクスチャのエッジを延長してアトラスへのエントリ間に分離ギャップを形成すると、この問題を解決できます。

以下のサブセクションから構成されています。

- *11.3.1 Unity でのミップマッピングの実装*(11-262 ページ).

11.3.1 Unity でのミップマッピングの実装

[Inspector] ウィンドウでは、ミップマップの自動生成がテクスチャに対して有効になっています。

ミップマップの生成を有効にするには、次の手順を実行します。

1. [Project] ウィンドウの [Assets] セクションでテクスチャを選択して、[Texture Inspector] ウィンドウを開きます。
2. [Generate Mip Maps] オプションを有効にします。

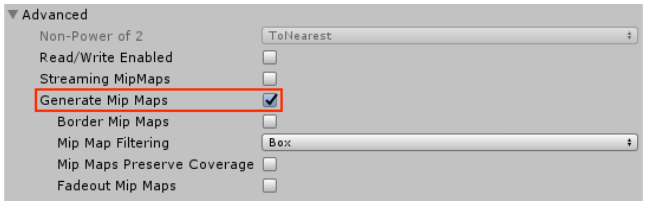


図 11-6 ミップマップの生成設定

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.4 詳細レベル

詳細レベル (LOD) は、オブジェクトとビューア間の距離が離れるにつれて、オブジェクトのディテールと複雑度を減少させる技法です。

ビューアがオブジェクトから近いときに LOD を使用することで、オブジェクトのジオメトリの詳細レベルが高くなります。したがって、オブジェクトはディテールまで正確に表示されます。オブジェクトが画面から離れると、詳細レベルの低いモデルに自動的に切り替わり、プレイヤーに対してディテールの少ないモデルが表示されます。

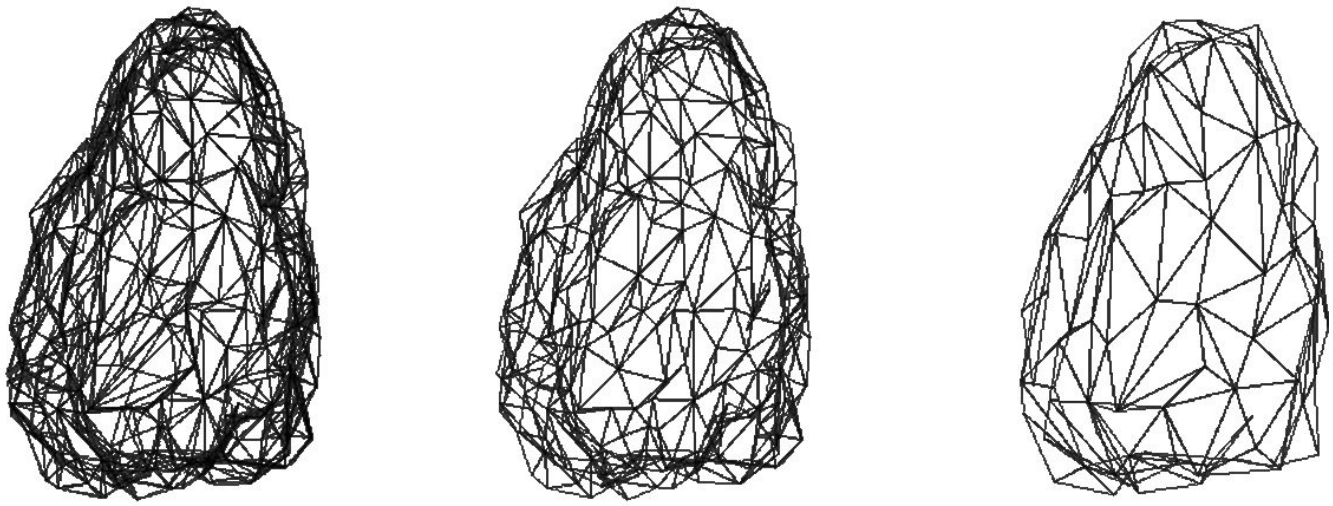


図 11-7 単一のメッシュに 3 つの LOD

モデルを詳細レベルの低いモデルに切り替えることには、主に次の 2 つの利点があります。

- ジオメトリ エイリアシングが発生する可能性が低くなるため、アーティファクトのレベルが低減されます。
- シェーディングを必要とする頂点の数が減少することで、パフォーマンスの向上につながります。

極小のジオメトリによる不要なオーバーサンプリングの発生を防ぐために、LOD は LOD レベル間で顕著な差をつけて作成する必要があります。

LOD の詳細については、*ジオメトリのベストプラクティスの章*(5-86 ページ) の LOD セクションを参照してください。

LOD の外観を示すビジュアルのサンプル

次の画像とビデオは、カメラに対してオブジェクトが近づいたり離れたりした際の LOD の仕組みを示したサンプルです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

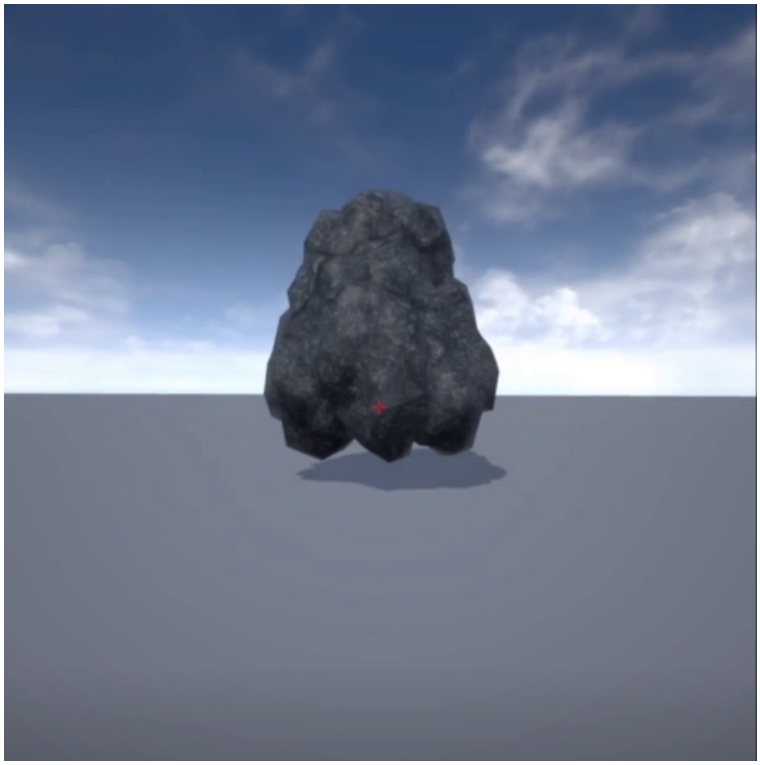


図 11-8 最終 LOD レンダリングのサンプル

次の Web アドレスは、カメラがオブジェクトに接近するのに伴い、3 つの異なる LOD を通るメッシュのサンプルを示すビデオにリンクしています。このビデオは、遠くからは見ることができず、カメラがオブジェクトに接近すると見える細かいディテールを示しています。ただし、この近くで見たときの細かいディテールはオブジェクトのリアリズムを向上させるのに役立ちます。

https://developer.arm.com/graphics/videos/level-of-detail-example

以下のサブセクションから構成されています。

- *11.4.1 Unity での詳細レベルの実装*(11-264 ページ).

11.4.1 Unity での詳細レベルの実装

Unity での*詳細レベル* (LOD) の実装は、[Component] メニュー オプションの使用に基づいています。

モデルの LOD を有効にするには、次の手順に従います。

1. 必要なモデルを選択します。
2. [Object Inspector Panel] の [Component] に移動します。
3. [Rendering] を選択します。
4. 次に [LOD Group] を選択します。

図 11-9 オブジェクトに対する LOD グループの有効化

[LOD Group] を選択すると、[LOD Group] ウィンドウが [Object Inspector Panel] に表示されます。さまざまな LOD レベルを選択して [Renderers] パネルを表示できます。このパネルでは、必要な複雑度のメッシュを選択できます。アプリケーションを LOD 間で移動するポイントに応じて、使用可能な LOD レベルのサイズを変更できます。

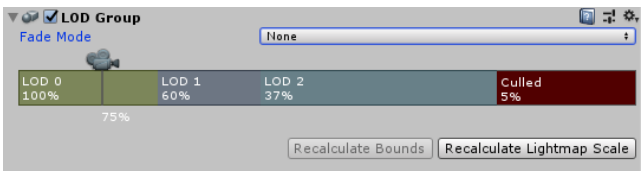


図 11-10 LOD 設定

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

必要な場合は、[Fade Mode] を [Cross Fade] または [SpeedTree] のいずれかに設定できます。[Fade Mode] では、オブジェクトのアプリケーションに応じて、シェーダ内でアクセスできるブレンド係数が表示されます。ブレンド係数により、LOD 間のスムーズなブレンドの実装が可能になります。

Unity はシェーダの LOD もサポートしているため、`Shader.globalMaximumLOD` または `Shader.maximumLOD` の現在の値に応じて、異なるシェーダまたはシェーダ パスを適用できます。

シェーダの LOD を活用すると、シェーダがもたらすメリットがハードウェアにとってコストがかかりすぎる場合に、より効率的なシェーダを適用できるようになります。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.5 色空間

色空間は、各色に割り当てられる数値を規定します。色空間は、空間内で各色が割り当てられている領域も定義するため、その色内で可能な変化量を定義します。

当初、レンダリングはガンマ色空間で実行されていました。つまり、モニタはガンマ色空間の画像を表示するように設計されているため、最終画像をモニタに表示するには、その前にガンマ補正が必要でした。

物理ベース レンダリング(PBR) の到来とともに、線形色空間でのレンダリングへのシフトがありました。これにより、複数の光源の値をシェーダ内で簡単かつ正確に累積できるようになりました。ガンマ色空間では、物理ベース レンダリングを追加しても、ガンマ色空間でのレンダリングに本来備わっている曲線のために、物理的に正確にはなりません。

線形色空間でのレンダリングでは鏡面反射エイリアシングを軽減できるため、さらなる利点がもたらされます。ガンマ色空間でのレンダリングでシーン内の輝度を上げると、オブジェクトが急激に白くなり、鏡面反射エイリアシングが発生するような場合に、利点が得られます。線形色空間では、オブジェクトの輝度は直線的に増し、オブジェクトが急激に白くなることが防止されるため、鏡面反射エイリアシングのリスクが軽減されます。

以下のサブセクションから構成されています。

- *11.5.1 Unity での色空間の選択方法*(11-266 ページ).

11.5.1 Unity での色空間の選択方法

Unity では、[Project Player Settings] で色空間を選択します。

線形色空間を有効にするには、次の手順に従います。

1. [Edit]、[Project Settings] の順に選択します。
2. [Player] を選択して [Project Player Settings] パネルを開きます。
3. [Player Settings] の [Other Settings] オプションにある、[Color Space] ドロップダウン オプションを開きます。
4. [Rendering] セクションで、ドロップダウン メニューから [Linear] オプションを選択します。

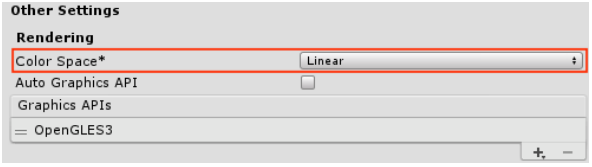


図 11-11 色空間の設定

——— 注 ———

アプリケーションを線形色空間に設定する場合は、グラフィックス API を OpenGLES 3.0 に設定する必要があります。正しい API を選択するには、[Auto Graphics API] オプションの選択を解除して、[OpenGLES2] を名前付きグラフィックス API のリストから削除します。[Identification]セクション内の [Minimum API Level] 設定も、少なくとも Android 4.3「Jelly Bean」、API レベル 18 に設定する必要があります。

———

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.6 テクスチャ フィルタリング

テクスチャ フィルタリングはテクスチャからのサンプリング時に発生するエイリアシングを削減するための技法です。

エイリアシングは、画面にレンダリングしているピクセルが、レンダリングしているオブジェクトにマッピングされているテクスチャ内のピクセルのグリッドに正確に収まらない場所で発生します。代わりに、テクスチャのマッピング先のオブジェクトが、ビューアに対して任意の距離および向きにあるときにある程度相殺されます。

テクスチャのピクセル、つまりテクセルを画面のピクセルにマッピングするときに発生し得る、問題となる状況が 2 つあります。1 つは、テクセルが画面のピクセルよりも大きくなることです。この場合は、画面のピクセルに収まるようにテクセルを縮小する必要があります。もう 1 つは、テクセルが画面のピクセルよりも小さくなる場合に生じ得る状況です。このような場合は、複数のテクセルを組み合わせて画面のピクセルに合わせる必要があります。

テクスチャ フィルタリングは、ミップマッピングに大きく依存します。これは、拡大中に、フェッチするテクセルの数が 4 を超えることがないためです。ただし、拡大中にテクスチャ オブジェクトが遠ざかった際に、テクスチャ全体が 1 つのピクセル内に収まることがあります。この場合は、各テクセルをフェッチした後にマージして正確な結果を得る必要があります。これを GPU に対して実装するにはコストがかかりすぎるためです。

代わりに、4 つのサンプルが選択され、予測不能なデータのアンダーサンプリングが発生します。ミップマッピングは、異なるサイズでテクスチャを事前にフィルタ処理し、オブジェクトが遠ざかる際により小さなテクスチャ サイズが代わりに適用されるようにすることで対処します。

幅広く実装されているフィルタリング方法には、バイリニア、トライリニア、および異方性フィルタリングがあります。これらの違いは、サンプリングされるテクセルの数、これらのテクセルの組み合わせ方法、およびフィルタリング プロセスにミップマッピングが利用されるかどうかです。

各フィルタリング方法のパフォーマンス コストはそれぞれ異なります。このため、使用するフィルタリングの種類の選択は、フィルタリング方法のパフォーマンス コストとそれからもたらされるビジュアルのメリットを比較して、状況に応じて検討する必要があります。

たとえば、トライリニア フィルタリングはバイリニア フィルタリングの 2 倍のコストがかかりますが、特にテクスチャが遠くのオブジェクトに適用される場合、明確なビジュアルのメリットは必ずしも得られません。それよりも、2 倍の異方性フィルタリングの使用が推奨されます。このフィルタリングの方が、画質とパフォーマンスが向上するためです。

異方性フィルタリングを使用する場合は、サンプルの最大数を設定する必要があります。ただし、モバイルではパフォーマンスに影響が及ぶため、多数のサンプルを使用する場合、特に 8 以上のサンプルを使用する場合は注意が必要です。この技術は、遠くの地面などの傾斜した表面のテクスチャに最も適しています。

以下の一連の画像は、各種テクスチャ フィルタリングの違いを示したものです。2 倍のサンプル異方性フィルタリングが有効になっている場合のバイリニア フィルタリングとトライリニア フィルタリングの違いがほとんどないことに注目してください。それにもかかわらず、トライリニア フィルタリングはより多くのリソースを消費します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

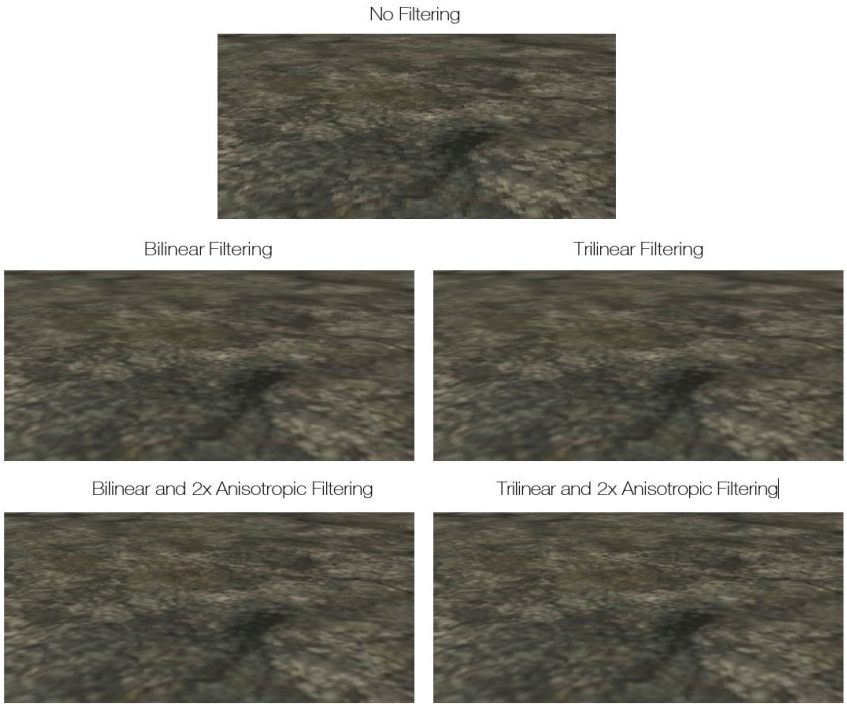


図 11-12 各種テクスチャ フィルタリング方法の比較

テクスチャフィルタリングの詳細については、*テクスチャリングのベストプラクティスの章*(6-96 ページ)を参照してください。

以下のサブセクションから構成されています。

- *11.6.1 Unity でのテクスチャ フィルタリングの実装*(11-268 ページ).

11.6.1 Unity でのテクスチャ フィルタリングの実装

Unity では、テクスチャ フィルタリングは [Project Quality Settings] で設定します。

Unity でテクスチャのテクスチャ フィルタリング設定を指定するには、次の手順に従います。

1. [Edit] に移動します。
2. 次に [Project Settings] を選択します。
3. [Quality] を選択して [Project Quality Settings] パネルを開きます。
4. このウィンドウで [Anisotropic Textures] ドロップダウンを選択します。
5. 最後に、[Per Texture] オプションを選択します。

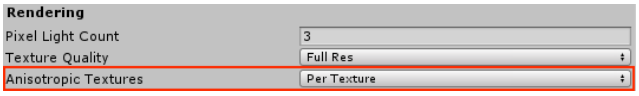


図 11-13 テクスチャ フィルタリング設定をテクスチャごとに設定

[Project] ウィンドウの [Assets] セクション内で各テクスチャを選択して、[Texture Inspector] ウィンドウを開きます。ウィンドウが開いたら、そのテクスチャの [Filter Mode] と [Aniso Level] の両方を設定します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

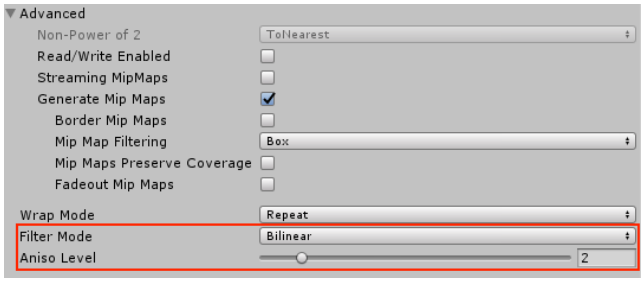


図 11-14 テクスチャのテクスチャ フィルタ設定の選択

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.7 アルファ合成

アルファ合成は、画像を背景画像と組み合わせて、透明に見える合成画像を生成する技法です。

アルファ テストは、アルファ合成を広く実装した形式です。ただし、アルファ チャネルがビット単位であるため、アルファ テストでは、オブジェクトのエッジに極端なエイリアシング効果が発生してエッジ間のブレンディングがないことがあります。

この場合、シェーダはピクセルごとに 1 回しか実行されないため、マルチサンプリングは影響しません。各サブサンプルは同じアルファ値を返し、エッジはエイリアスされたままになります。

アルファ ブレンディングは代替ソリューションです。しかし、ポリゴン ソートを実行しないとブレンディングは失敗し、オブジェクトは正しくレンダリングされません。ただし、ソートの有効化はコストのかかるプロセスであり、パフォーマンスは低下します。

Alpha to Coverage (ATOC) は別のアルファ合成方法で、エイリアシングの削減に役立ちます。ATOC は、フラグメント シェーダのアルファ合成出力をカバレッジ マスクに変換し、それをマルチサンプリング マスクと組み合わせます。その後、ATOC は AND 演算子を使用し、その演算に合格したピクセルのみをレンダリングします。

アルファ テストとアルファ カバレッジの動作を示すビジュアル サンプル

次の画像とその下のリンクはどちらも基本的なアルファ テスト実装 (左) と Alpha to Coverage 実装 (右) の違いを示しています。

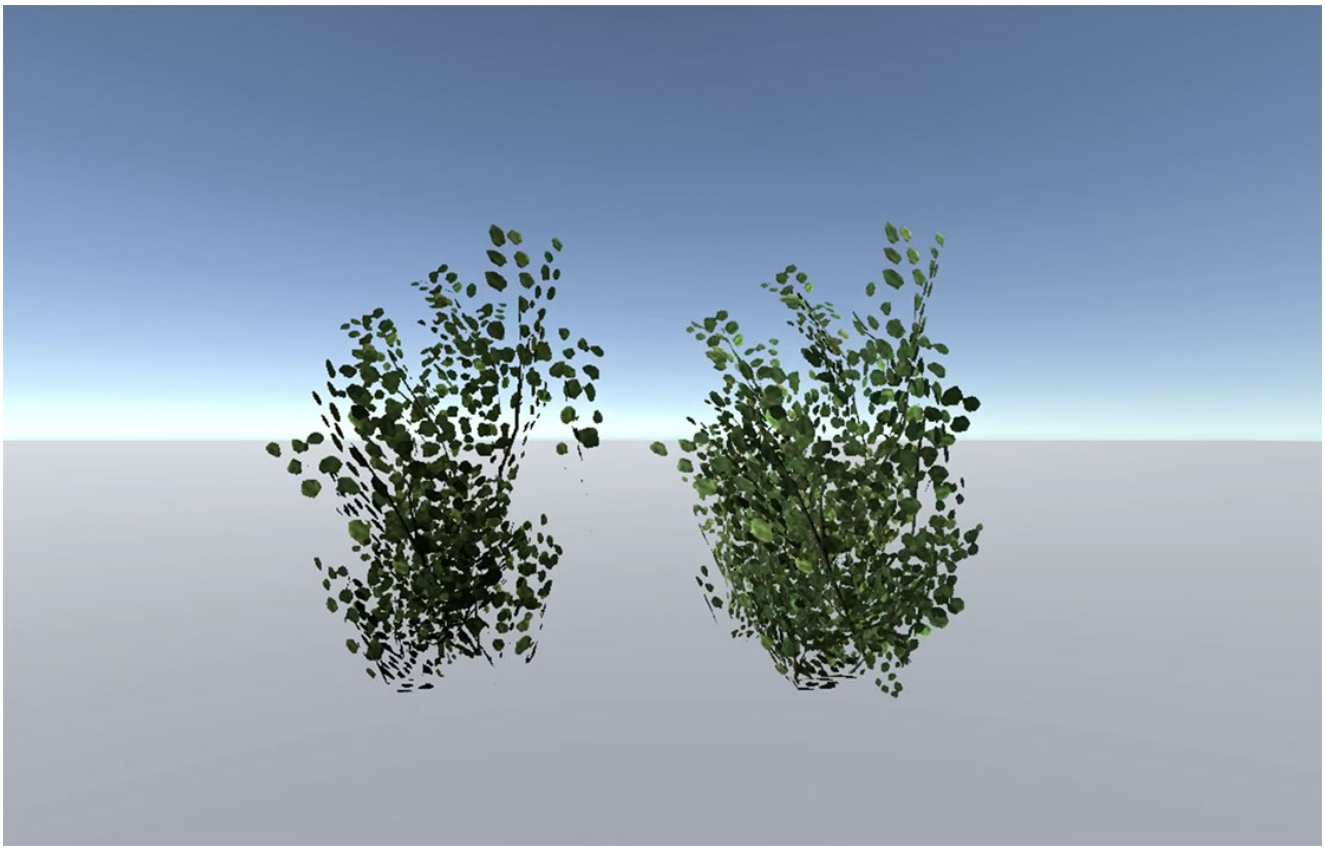


図 11-15 アルファ合成のサンプル

https://developer.arm.com/graphics/videos/alpha-compositing-example

Alpha to Coverage 実装では、エイリアシングがはるかに少なく、ちらつきも減少しています。

以下のサブセクションから構成されています。

- *11.7.1 Unity でのアルファ合成の実装*(11-271 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.7.1 Unity でのアルファ合成の実装

Alpha-to-Coverage は Unity シェーダ内で実装されます。

ミップ マップ カバレッジ保持オプション – Unity 2017 以降

ミップ マップ カバレッジ保持オプションにより、以前のバージョンで必要とされるミップ マップ レベルの計算が不要になります。これは、[Texture Inspector] で有効にすることができます。

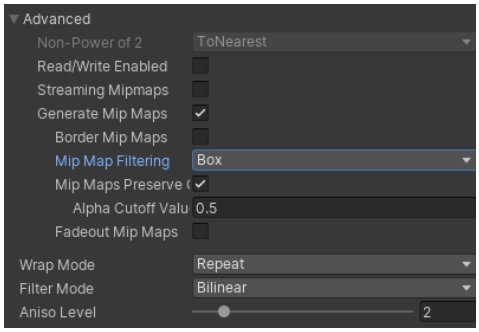


図 11-16 Unity 2017 以降のミップ マップ カバレッジ保持オプション

2017 以降の Unity で ATOC を有効にするには、[Project] ウィンドウで新しいシェーダを作成してから、以下のシェーダ コードを挿入する必要があります。

注

このシェーダをマテリアルに適用し、そのマテリアルを、アルファ合成が必要なオブジェクトに設定します。

```
AlphaToMask On
                struct frag_in
                {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                half3 worldNormal : NORMAL;
                };

                fixed4 frag(frag_in i, fixed facing : VFACE) : SV_Target
                {
                /* Sample diffuse texture */
                fixed4 col = tex2D(_MainTex, i.uv);

                /* Sharpen texture alpha to the width of a pixel */
                col.a = (col.a - 0.5) / max(fwidth(col.a), 0.0001) + 0.5;
                clip(col.a - 0.5);

                /* Lighting calculations */
                half3 worldNormal = normalize(i.worldNormal * facing);
                fixed ndotl = saturate(dot(worldNormal,
normalize(_WorldSpaceLightPos0.xyz)));
                col.rgb *= ndotl * _LightColor0;

                return col;
                }
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

2017 より前のバージョンの Unity を使用している場合

2017 より前のバージョンの Unity で ATOC を有効にするには、ミップマップ レベルの計算をシェーダ コードに追加する必要もあります。2017 以降のバージョンの Unity では、最初に [Project] ウィンドウで新しいシェーダを作成する必要があります。その後で、以下のシェーダ コードを挿入します。

注

このシェーダをマテリアルに適用し、そのマテリアルを、アルファ合成が必要なオブジェクトに設定します。

```
Shader "Custom/Alpha To Coverage"
{
    Properties
    {
        MainTex("Texture", 2D) = "white" {}
    }
        SubShader
    {
        Tags { "Queue" = "AlphaTest" "RenderType" = "TransparentCutout" } Cull Off

        Pass
        {
            Tags { "LightMode" = "ForwardBase" }

            CGPROGRAM
            #pragma vertex vert #pragma fragment frag

            #include "UnityCG.cginc"
            #include "Lighting.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
                half3 normal : NORMAL;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                half3 worldNormal : NORMAL;
            };

            sampler2D _MainTex; float4 _MainTex_ST;
            float4 _MainTex_TexelSize;

            v2f vert(appdata v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                o.worldNormal = UnityObjectToWorldNormal(v.normal);
                return o;
            }

            fixed4 frag(v2f i, fixed facing : VFACE) : SV_Target
            {

                fixed4 col = tex2D(_MainTex, i.uv);

                float2 texture_coord = i.uv * _MainTex_TexelSize.zw; float2 dx =
ddx(texture_coord);
                float2 dy = ddy(texture_coord);
                float MipLevel = max(0.0, 0.5 * log2(max(dot(dx, dx), dot(dy, dy))));

                col.a *= 1 + max(0, MipLevel) * 0.25; clip(col.a - 0.5);

                half3 worldNormal = normalize(i.worldNormal * facing);

                fixed ndotl = saturate(dot(worldNormal,
normalize(_WorldSpaceLightPos0.xyz)));
                fixed3 lighting = ndotl * _LightColor0; lighting +=
ShadeSH9(half4(worldNormal, 1.0));
                col.rgb *= lighting; return col;
             }
             ENDCG
          }
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

```
    }
}
```


Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.8 レベル設計

エイリアシングを最小限に抑えるためにあらゆる技術的オプションを利用できるにもかかわらず、エイリアシング アーティファクトを削減するうえで、慎重なレベル設計は依然として必要です。レベル設計の選択を誤ると、すべての技術的ソリューションが冗長となる可能性があるため、レベル設計において賢明な意思決定を行うことはエイリアシングの低減に重要な役割を果たします。

シーンのジオメトリを作成する際、メッシュのエッジをモデリングする際にエッジが鋭くなったり突然現れたりしないように注意する必要があります。たとえば、各段にフラット エッジのある階段では、ビューアがヘッドを回転させるとエイリアシングが発生する可能性があります。

ただし、遠くから階段を見た場合、個々の段自体がエイリアシングの原因となることもあります。遠距離では各段が薄いオブジェクトになり、ビューアが動くとちらつく可能性があります。

次の画像は、階段を使った場合と斜面を使った場合の異なるレベルのエイリアシングのサンプルを示しています。

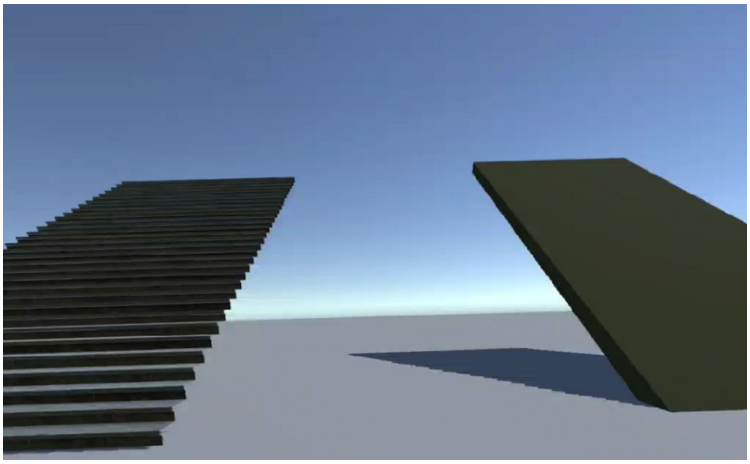


図 11-17 階段と斜面でのエイリアシングの比較

階段と斜面でのエイリアシングの比較ビデオ

次のビデオは、ランプではなく、階段を使った場合と斜面を使った場合の異なるレベルのエイリアシングのサンプルを示しています。

https://developer.arm.com/graphics/videos/ramp-vs-stairs-aliasing-example

可能な限り、ハード エッジやベベルのあるシェイプの代わりに、スムーズな丸いシェイプを使用してください。ワイヤやケーブルなどの細いオブジェクトは、遠くから見た場合に多くのエイリアシングを発生させる傾向があります。これは、細い/薄いオブジェクトは線としてレンダリングされ、エイリアシングが発生する原因となるからです。

メタリック マテリアルの使用については、慎重に考慮する必要があります。メタリック オブジェクトは、光を照らした場合に鏡面反射効果があるため、使用を最小限にとどめる必要があります。この鏡面反射効果により、ビューアの動きに伴ってちらつきが生じるため、エイリアシングが発生します。したがって、マット マテリアルを使用することをお勧めします。

次の画像は、メタリック マテリアルとマット マテリアルによって発生するエイリアシングの比較を示しています。上の画像はメタリック マテリアル、下の画像は上の画像と同等のマット マテリアルを示しています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 11-18 メタリック マテリアルとマット マテリアルの比較

明るく光るライトは、照らされたオブジェクトに鏡面反射効果を与えるため、シーンの照明は注意して実装してください。照明によって、非常に顕著な鏡面反射エイリアシングが発生する可能性があります。鏡面反射効果がフレーム間で出現したり消えたりするため、視線がピクセルの「ちらつき」に引かれます。

コストの高いリアルタイムのライティング計算を避けて、バンディングなどのライティング アーティファクトの発生を削減するために、シーンの構築時にライティングをプリベークすることができます。そのためには、移動するオブジェクトを減らし、ライブのデイ サイクルやナイト サイクルをなくす必要があるプリベークに適するようにシーンを設計しなければなりません。

ライト プローブを利用すると、必要なプリベークの量を最小限に抑えることができ、6 自由度の VR で特に有効です。

反射は、一般的なユース ケースよりも VR の方が注意を要します。VR の場合、画面空間の反射などの技法は計算量が多すぎます。したがって、反射には別の手法を採用する必要があります。反射は実行時にあらかじめレンダリングすることができるため、高品質の反射を必要とするオブジェクトに使用できる技法には、リフレクション プローブおよびキューブ マップがあります。

エイリアシングを隠すには、フォグやスモークなどのパーティクル効果を展開することができます。短いレンダリング距離を隠すために使用されてきたこれらの技法は、遠距離のオブジェクトが原因のエイリアシングを覆うためにも使用できます。

次の図では、右の画像はパーティクル効果を追加した後でエイリアシングがどのように軽減されているかを示しています。

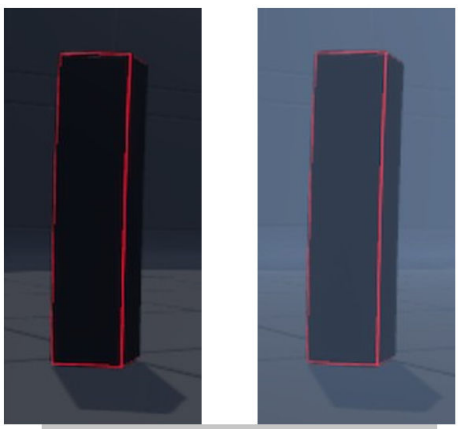


図 11-19 パーティクル効果適用前後の比較

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.9 バンディング

バンディングは、ピクセル当たりの指定のビット数内で、必要な色を正確に表現することができない場合に発生します。VR 内では、各バンド間での不意の変化によって示される異なる色のバンドとしてバンディングが発生します。

バンドは VR 内ではより顕著に表れます。これは、ユーザーがシーンに没頭しているためです。このため、ユーザーの目はシーンの中の光レベルに適応します。目が慣れてくると、バンドはさらに顕著になります。バンドが多く存在する場合、絶えず変化する光レベルに目が適応しようとするため、目が疲れる可能性があります。

軽減方法 – ディザリング

ディザリングは、バンディング マテリアルまたはビューアにノイズを入れるプロセスです。このプロセスによってそれぞれの色のバンドが分解されて乱されるため、境界がそれほど目立たなくなります。

使用できるディザリングにはさまざまなバリエーションがあり、それぞれ異なるノイズの形式を適用するか、異なるアプローチを使用して、適用されるノイズを集めます。たとえば、リアルタイムでノイズを生成するものもあれば、ランダム ノイズを含むように作成されたテクスチャからノイズをサンプリングするものもあります。

軽減方法 – トーンマッピング

トーンマッピングは、*高ダイナミック レンジ*(HDR) の色を変換し、HDR に対応していないスクリーンでの表示に適した*低ダイナミック レンジ*(LDR) 内に収める一種のカラー グレーディングです。トーンマッピングでは*ルック アップ テーブル* (LUT) を使用して、各色が新しい色調の適切な対応色にマッピングされるようにします。

バンディングの原因になっている低い光レベルやテクスチャの急な勾配は、トーンマッピングをシーンに適用することで軽減できます。

以下のサブセクションから構成されています。

- *11.9.1 Unity でのディザリングの実装*(11-276 ページ).
- *11.9.2 Unity でのトーンマッピングの実装*(11-277 ページ).

11.9.1 Unity でのディザリングの実装

Unity でのディザリングは、通常は後処理パッケージを経由して行われます。このパッケージは追加とセットアップが必要です。または、[Universal Render Pipeline] のカメラに組み込まれているエフェクトを使用することもできます。

ディザリングを有効にするには、以下の手順に従います。

1. [Window] の [Package Manager] で、[Post Processing] を検索してパッケージをインストールします。必ずバージョン 2.x を使用してください。
2. [Project Window] を右クリックし、[Project Window] で [Post Processing Profile] を作成します。
3. [Create] で、[Post-processing Profile] を選択します。
4. カメラの [Inspector] 内から、ディザリングが必要なシーン内の各カメラに [Post Process Volume] および [Post Process Layer] コンポーネントを追加します。
5. 最後に、[Post Processing Volume] 内から、作成された [Post Processing Profile] を [Profile] として設定します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

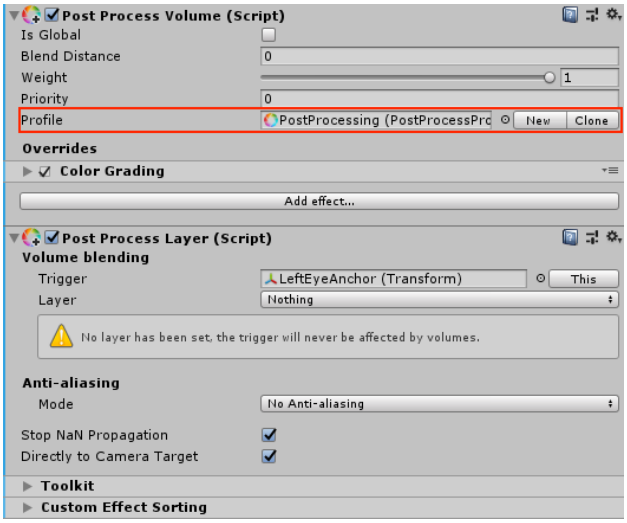


図 11-20 後処理プロファイルの設定

以下のスクリーンショットに示すように、Unity の [Universal Render Pipeline] では、カメラのディザリングを有効にすることができます。

1. カメラの [Inspector] ウィンドウのレンダリング セクションで、[Post-Processing] ボックスをオンにします。
2. 次に、その下の [Dithering] ボックスをオンにします。

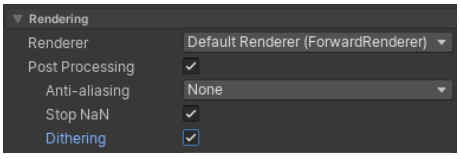


図 11-21 カメラのディザリングを有効にする方法

11.9.2 Unity でのトーンマッピングの実装

Unity でのトーンマッピングには、後処理パッケージのインストールとセットアップが必要です。

トーンマッピングを有効にするには、次の手順を実行します。

1. [Window] の [Package Manager] で、[Post Processing] を検索してパッケージをインストールします。必ずバージョン 2.x を使用してください。
2. 次に、[Project Window] を右クリックし、[Project Window] で [Post Processing Profile] を作成します。
3. [Create] で、[Post-processing Profile] を選択します。
4. 次に、この [Post-processing Profile] を選択します。
5. [Inspector] で、[Add effect…] を選択します。
6. [Unity] で、[Unity Color Grading] を選択し、次に [Mode] オプションを [High Definition Range] に設定します。
7. トーンマッピングの [Mode] の横にあるボックスをオンにし、コンボボックスで [ACES] を選択します。
8. カメラ セクションの [Inspector] 内から、トーンマッピングが必要なシーン内の各カメラに [Post Process Volume] および [Post Process Layer] コンポーネントを追加します。
9. 最後に、[Post Process Volume] 内から、事前に作成された [Post Processing Profile] を [Profile] として設定します。

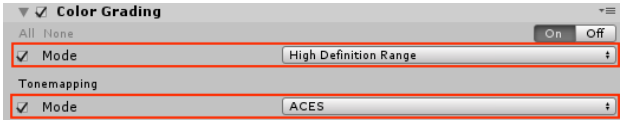


図 11-22 トーン マッピングの設定

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.10 バンプ マッピング

バンプ マッピングはメッシュの頂点の数を減らすために使用する技法で、バンプなど、オブジェクト表面での細かいディテールのシミュレーションを伴います。必要なシミュレーションは、ライティング計算に使用する前にオブジェクトの法線を操作することで実行します。

法線マッピングはほとんどの使用に適した技法ですが、法線マッピングされたテクスチャの視野角をプレイヤーが簡単に変更できてしまうため、VR の場合は効果的ではありません。したがって、法線マッピング技法では視点の変化が考慮されないため、奥行きの錯覚が壊れます。

また、法線は 1 つの視点のみから生成されるため、法線マッピングでは、VR ヘッドセットで使用される立体視レンズの使用は考慮できません。したがって、それぞれの目は人の目ではまちがって見える同じ法線を受け取ります。

別のバンプ マッピング方法を比較したビデオ

次のビデオでは、法線マッピングと視差オクルージョン マッピングのサンプルを示しています。

https://developer.arm.com/graphics/videos/bump-mapping-example

軽減方法 – 法線マッピング

法線マッピングは、バンプ マッピングの最も一般的な実装であり、メッシュの高ポリゴン バージョンと低ポリゴン バージョンの両方の作成を伴います。その後、法線マップが高ポリゴン バージョンからエクスポートすることによって作成され、細かいディテールの法線は法線マップ テクスチャに保存されます。

レンダリングの際、法線マップおよび法線からのフラグメント シェーダ サンプルは、サンプル値から生成されます。生成されたこれらの法線は、その後、低ポリゴン バージョンのサーフェス法線と組み合わされ、ライティングの計算時に使用されます。ライティングによって細かいサーフェスのディテールが明らかになります。ディテールの個々の頂点をレンダリングする必要はありません。

通常、法線マッピングは VR ではあまり効果的ではありませんが、フラット マテリアルよりは効果的です。特に、シーンのライティングを行う際、法線マップの配置を慎重に考慮する場合に効果的です。

軽減方法 – 視差オクルージョン マッピング

視差オクルージョン マッピングは、法線マッピングに似た技法です。ただし、このマッピングでは、テクスチャ座標の位置変更時にビューアの角度がサーフェス法線に対して考慮されます。

したがって、視野角が狭いほど、高い度合いでテクスチャ座標の位置が変更されます。これにより、奥行きの錯覚が維持されます。

視差オクルージョン マッピングは、計算量の多いプロセスです。そのため、ビューアが接近できる小さいマテリアルのみにこの技法を使用してください。遠くにあるテクスチャでは、視野角を大きく変更できないため、視差オクルージョン マッピングを使用する利点はほとんどありません。

以下のサブセクションから構成されています。

- *11.10.1 Unity での法線マッピングの実装*(11-278 ページ).
- *11.10.2 Unity での視差オクルージョンの実装*(11-279 ページ).

11.10.1 Unity での法線マッピングの実装

法線マッピングは、Unity 内に組み込まれている多くのシェーダで使用できます。

法線マッピングをマテリアルに追加するには、以下の手順に従います。

1. マテリアルを [Project Window] で選択します。
2. [Material Inspector Panel] を開き、法線マッピングのサポートを含むシェーダを選択します。これらのシェーダには、[Standard]、[Universal Render Pipeline]、モバイル用の [Bumped Diffuse] などがあります。
3. 最後に、必要な [Normal Map] テクスチャを設定します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

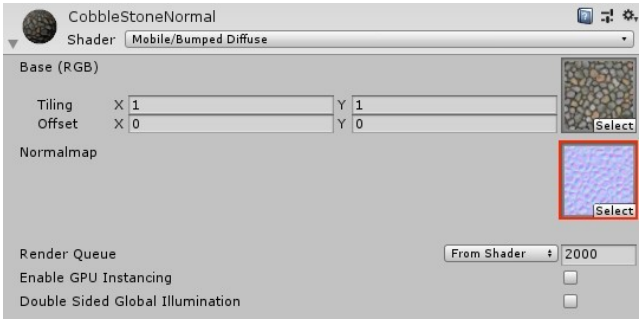


図 11-23 法線マップ テクスチャの追加

11.10.2 Unity での視差オクルージョンの実装

視差オクルージョン マッピングは、Unity の組み込みシェーダを使用して行うことができます。

視差オクルージョン マッピングをマテリアルに追加するには、次の手順に従います。

1. マテリアルを [Project Window] で選択します。
2. [Material Inspector Panel] を開いて、視差拡散マッピングをサポートしているシェーダを選択します。たとえば、[Standard] シェーダを選択します。
3. 最後に、必須のテクスチャである [Albedo]、[Normal Map]、および [Height Map] を設定します。

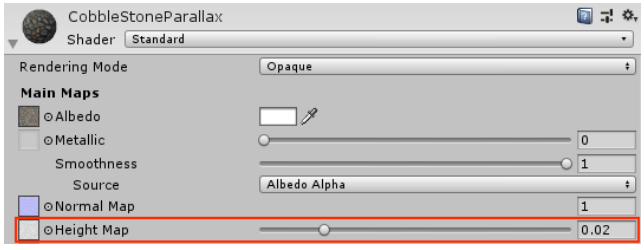


図 11-24 視差オクルージョン マッピングをマテリアルに追加

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

11.11 影

シャドウ マッピングの従来の手法は、モバイル デバイスでは計算負荷が高くなります。シャドウ マップに追加のフレーム バッファとレンダリング パスが必要になるため、パフォーマンスが大幅に低下します。

したがって、モバイル デバイスのシャドウ バッファは解像度が低いことが多く、フィルタリングを使用しません。また、シャドウはしばしば大量のエイリアシングを引き起こすため、シャドウ アクネやハード シャドウなどのアーティファクトによってシミュレーションのリアリズムが損なわれます。

次のスクリーンショットは、左側は典型的なシャドウを、右側は丸影の比較を示しています。

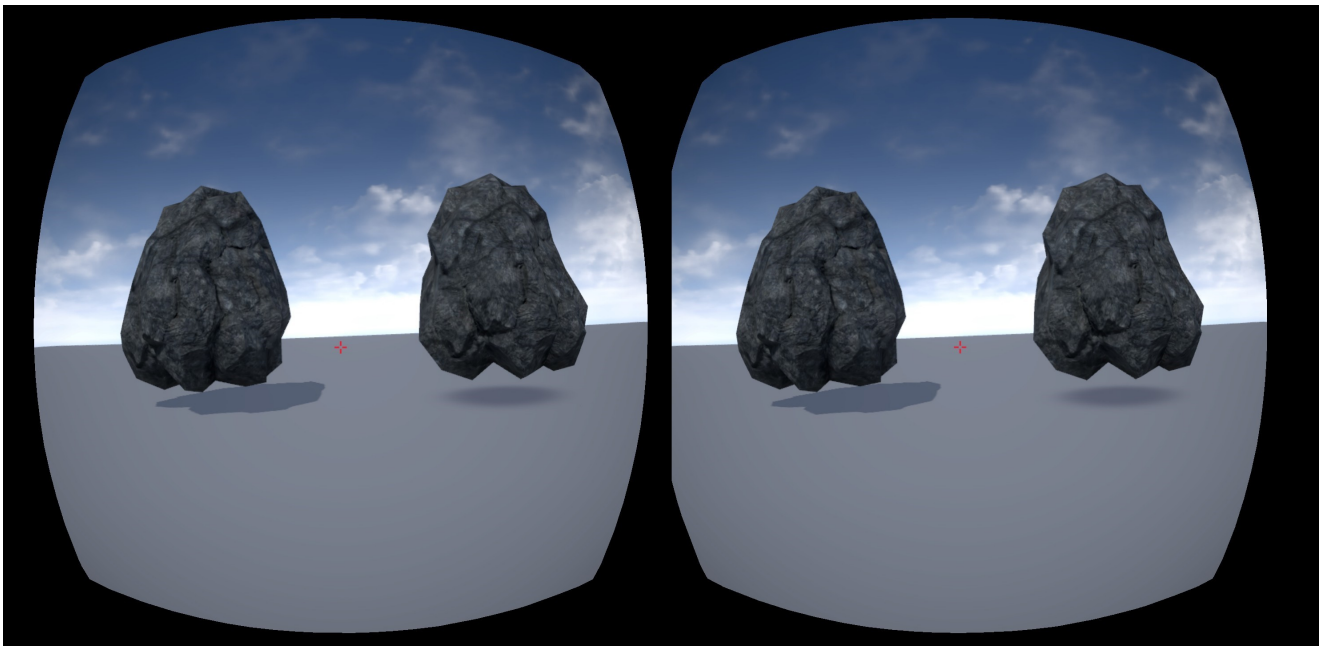


図 11-25 シャドウ マッピングと丸影の比較

検討する価値のあるもう 1 つの手法は、ベーキングされたシャドウを使用することです。シャドウ マッピングとは異なり、シャドウ ベーキングでは追加のレンダリング パスは必要ありません。代わりに、高品質のシャドウが、影を落とすテクスチャにベーキングされます。

軽減方法 – 丸影

VR では、シャドウのレンダリングを可能な限り避けることが推奨されます。ただし、プレイヤーの下などにシャドウが必要な場合は、オブジェクトの下に丸影をレンダリングできます。丸影は、一般的なシャドウ マッピング手法を使用する場合よりも、パフォーマンスを向上させると同時に、エイリアシングが大幅に少なくなります。

以下のサブセクションから構成されています。

- *11.11.1 Unity での丸影の実装*(11-280 ページ).

11.11.1 Unity での丸影の実装

丸影は、Unity のシーン コンポーネントとして無料で使用でき、**アセット ストア**から入手できます。

丸影を実装するには、次の手順に従います。

1. Unity アセット ストアから **Unity 標準アセット**をインポートします。
2. [Assets] > [StandardAssets] > [Effects Projects] > [Prefabs] 内で [BlobShadowProject] プレファブに移動します。
3. [BlobShadowProject] プレファブを丸影を必要とするオブジェクトのチャイルドとして [Hierarchy] にドラッグします。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

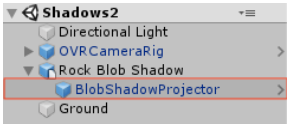


図 11-26 チャイルド オブジェクトの設定

完了すると丸影が表示されます。丸影の作成を完了するには、次の手順に従います。

1. [prefab] オプションを編集します。影を落とす必要のあるオブジェクトに対する丸影の場所を必ず含めてください。
2. 影を落とすオブジェクトを選択してから、[Cast Shadows] オプションを [Off] に設定します。

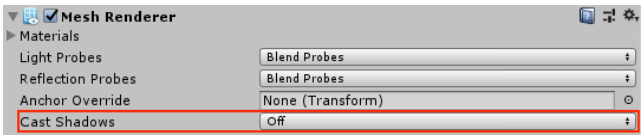


図 11-27 オブジェクトに対する [Cast Shadows] の無効化

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

第 12 章
Vulkan

本章では、Vulkan とそれを有効にする方法について説明します。

Vulkan は、クロノス・グループが発表したクロスプラットフォームのグラフィックスおよびコンピュート API であり、OpenGL および OpenGL ES と比較して多くのメリットがあります。メリットとしては、以下が挙げられます。

- モバイル、デスクトップ、コンソール、サーバ、および組み込みシステムに対して、1 つの統合された API フレームワークを提供する。
- 多様な機能を搭載したハードウェアをサポートする。
- ドライバのオーバーヘッドが最小限に抑えられているため、ARM Mali GPU ハードウェアで高性能を発揮できる。
- アプリケーションが、GPU 全体でさらにローレベルのアクセスを獲得して、リソースをコンピュートする。
- アプリケーション プロセッサのボトルネックが低減される。
- マルチスレッディングとマルチプロセッシングがサポートされる。
- 複数のアプリケーション プロセッサを効率的に使用できる。
- シェーダに SPIR-V 中間言語を使用することで、ランタイム カーネルのコンパイル時間が短縮される。
- シェーダのソース コードの出荷に対する要件がない。
- アプリケーション プロセッサのオーバーヘッドが低減され、ドライバが単純化され、オンチップ メモリがより多く使用されるため、エネルギー消費量が低減する。

以下のセクションから構成されています。

- *12.1 Vulkan について*(12-283 ページ).
- *12.2 Unity における Vulkan について*(12-285 ページ).
- *12.3 Unity における Vulkan の有効化*(12-286 ページ).
- *12.4 Vulkan のケース スタディ*(12-287 ページ).

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

12.1 Vulkan について

Vulkan は、クロノス・グループが開発したクロスプラットフォームのグラフィックスおよびコンピュート API です。

Vulkan は古い OpenGL および OpenGL ES 規格と比較して多くの利点があります。

統合されており、可搬性がある

Vulkan はモバイル、デスクトップ、コンソール、および組み込みシステムに対して、1 つの統合された API フレームワークを提供します。
広範囲に及ぶインプリメンテーションで可搬性があり、多様なアプリケーションに役立ちます。

ドライバがシンプル

Vulkan ではよりシンプルなドライバを使って、ドライバ オーバーヘッドを最小限に抑えます。遅延が低下し効率性が向上しているため、Vulkan を使用しているアプリケーションは、OpenGL ES 3.1 と比較してパフォーマンスが向上します。
ドライバがよりシンプルであるため、アプリケーション プロセッサのボトルネックが低減されます。
アプリケーションでリソース管理を行い、GPU をローレベルで直接コントロールします。

マルチスレッディングとマルチプロセッシング

Vulkan は、複数のアプリケーション プロセッサ間でのマルチスレッディングをサポートします。この機能により、複数のアプリケーション プロセッサを効率的に使用して、処理負荷と消費電力を低減することができます。
アプリケーションでスレッド管理と同期化をコントロールします。

コマンド バッファ

マルチスレッディングを使用して、コマンド バッファのためにコマンド作成を並行して行うことができます。また、個別のサブミッション スレッドを使用して、コマンド バッファをコマンド キューに入れることもできます。
グラフィックス、コンピュート、および DMA コマンド バッファを追加できます。
さまざまなグラフィックス、DMA、およびコンピュート キューがあるため、ジョブ ディスパッチの柔軟性が向上します。
マルチスレッド化されたコマンド作成により、コードを複数のアプリケーション プロセッサ コア間で実行して、パフォーマンスを向上させることができます。複数のアプリケーション プロセッサを比較的低いクロック レートで使用することで、クロック数が高いプロセッサを 1 つ使用する場合よりも消費電力が低減されます。

SPIR-V

Vulkan は、共通言語のフロント エンドを使用できるようにする SPIR-V 中間言語を使用します。
SPIR-V は、並列コンピュートおよびグラフィックスを実現するマルチ API 中間言語です。それには、フロー制御、グラフィックス、および並列コンピュート コンストラクトが内蔵されています。
Vulkan シェーダと OpenCL カーネル ソース言語にネイティブ表現を提供します。
複数のプラットフォームで同じ 1 つの SPIR-V フロントエンド コンパイラを使って、プレコンパイルド シェーダを生成できます。
SPIR-V を使用するということは、Vulkan ドライバにはフロントエンド コンパイラがないことを意味します。そのため、ドライバはよりシンプルになり、シェーダのコンパイル処理はさらに高速になります。
中間言語を使用するということは、アプリケーションにシェーダのソース コードを添付して出荷する必要がないことを意味します。また、将来的にさまざまなシェーディング言語を使用するための柔軟性も得られます。

ローダブル レイヤ

Vulkan では、テストとデバッグを行うために、開発段階のソフトウェア レイヤをロードできます。追加したソフトウェア レイヤは製造環境で削除できるため、出荷製品にはテストのオーバーヘッドは存在しません。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

マルチパス レンダリング

マルチパス レンダリングは、レンダー パスですべてのものをあらかじめ宣言しておく手法です。サブ パスごとに異なる出力を指定し、その出力を連鎖させることができます。

マルチパスでは、1 つのサブパスのピクセルが同じピクセル位置にある直前のサブパスの結果にアクセスするときに、ドライバで最適化を行うことができます。こうして、データを高速のオンチップ メモリに封じ込めて、帯域幅と電力を節約することができます。このプロセスは、Mali GPU などのタイルベースの GPU では効率性がさらに向上します。

マルチパス レンダリングの使用例は、以下のとおりです。

- ディファード レンダリング
- ソフト パーティクル
- トーン マッピング

以下の図は、Vulkan の構造を示したものです。

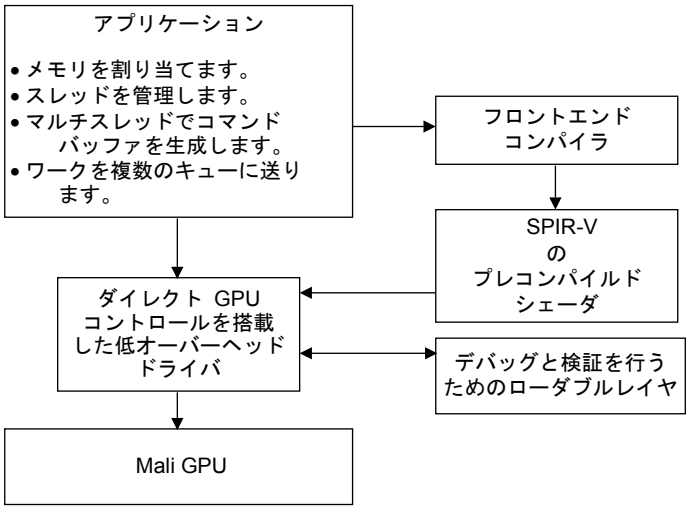


図 12-1 Vulkan の構造

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

12.2 Unity における Vulkan について

Unity で Vulkan を使用することのメリット。

Vulkan のメリットの多くは、OpenGL や OpenGL ES と比較して Vulkan のドライバがシンプルであるという事実に基づきます。

しかし、管理作業の多くをアプリケーション内で行わなければならず、このローレベル アクセスが原因で、アプリケーションを書く作業がさらに複雑になってしまうというマイナス面もあります。

Unity はユーザに代わってこの作業の大半を担うため、アプリケーションは Vulkan の強化されたパフォーマンスの恩恵を自動的に受けることができます。

Unity では、ユーザーは、Unity がアプリケーションビルドの際に使用するグラフィックス API のリストに、Vulkan を追加するだけで良いのです。

Vulkan は、OpenGL ES より効率的であり、消費電力を低減し、バッテリー寿命を延ばすので、利用できるのであればより良い選択肢となります。以下の場合は、Vulkan を使用することを検討してください。

- アプリケーションにとって最大のパフォーマンスを希望する。
- アプリケーションがアプリケーション プロセッサにバインドされている。
- 問題の原因が OpenGL または OpenGL ES ドライバにある可能性がある。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

12.3 Unity における Vulkan の有効化

ここでは、Unity 5.6 で Vulkan を有効にする手順について説明します。

Vulkan を有効にするには、以下の手順に従います。

1. [File] > [Build Settings …]を選択します。
2. 表示されたダイアログ ボックスで、[Player Settings …] ボタンを押します。
3. [Player Settings] ダイアログ ボックスで、[Other Settings] セクションを見つけます。
4. [Auto Graphics API] チェック ボックスがオフになっていることを確認します。オフになっていると、API を手動で選択できます。
5. Vulkan を API として追加するには、[+] を押して新しい API をリストに追加し、Vulkan を追加します。Vulkan は最後のオプションとして追加されます。
6. Vulkan をプライマリ API にするには、[Vulkan] を選択し、リストの一番上に移動します。

これで、Vulkan がプレーヤーのプライマリ API になりました。これを示したのが以下の画像です。

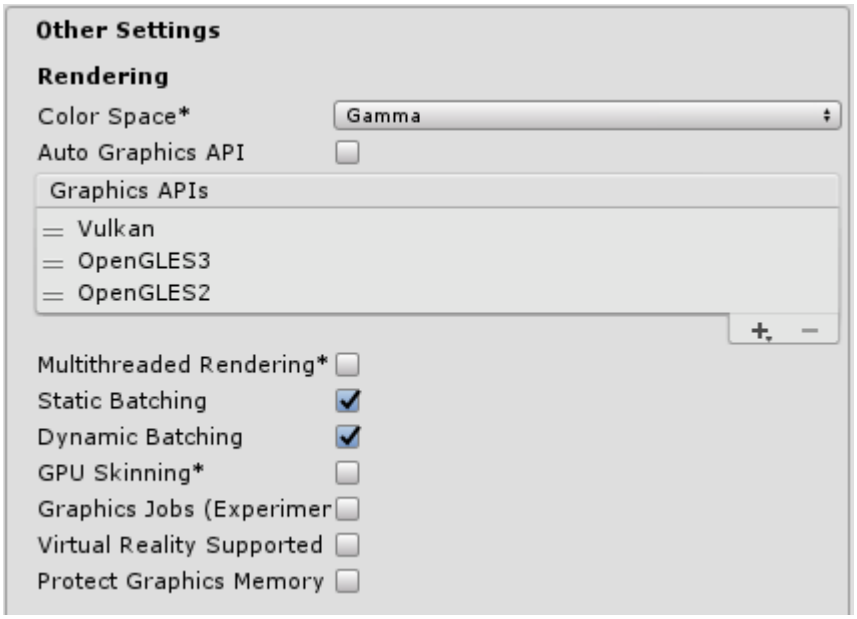


図 12-2 Unity API の選択

他の 2 つを削除して、Vulkan を唯一の API として残すことができます。

これを行うには、API を選択し、[–] を押して削除します。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

12.4 Vulkan のケース スタディ

ゲーム デベロッパーの Infinite Dreams は、Vulkan を使用してどのようにパフォーマンスを向上させたのか。

モバイル ゲーム デベロッパーの Infinite Dreams は、人気のモバイル ゲーム Sky Force Reloaded を開発しています。Sky Force Reloaded の特徴は、豊かなグラフィカル環境内で繰り広げられる激しいアクションであり、これにはアプリケーション プロセッサと GPU の両方が非常に要求されます。Sky Force Reloaded は、当初は OpenGL ES を搭載した Unity を使用してビルドされました。

図 12-3 Sky Force Reloaded

Sky Force Reloaded の最適化

ゲームの初期開発で、必要とされる豊かなグラフィカル体験が生み出されました。次の段階は、最適化でした。

多くのものを画面に同時に表示する点で、ゲームはグラフィックス的に複雑であるため、最適化すべき最大の分野はフィル レートだと開発チームは考えました。大抵の場合、フィル レート関連の問題はフレームバッファの解像度を下げることで解決できます。

ところが、ゲームを低解像度でレンダリングしてみても、パフォーマンス問題は解決しませんでした。ハイエンド デバイス上でも、60 *フレーム/秒* (FPS) を常時維持することはできませんでした。チームは、ゲームで膨大な数のドロー コールが行われていることに気付きました。ときには、1 フレームあたり 1,000 ドロー コールに達することもありました。

どのドロー コールにもコンピューテーション オーバーヘッドがあるので、膨大な数のドロー コールを行うことは、計算的に高コストになります。これにより、OpenGL ES ドライバが、データを GPU 向けに準備するためにアプリケーション プロセッサを長時間ビジー状態にします。その結果、ハイエンド モバイル デバイスであってもスロー ダウンしてしまいます。

最適化のために、チームはドロー コール数を最小限にする、またはドロー コールの改変によって、ゲーム エンジンがドロー コールを一括処理できるようにしました。とは言え、ゲームの質を落とすことなく、いつでもこれを行うことができるとは限りません。

Vulkan のテスト

チームは Vulkan のことを聞き、ゲームのパフォーマンスを向上できないか試してみることにしました。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

Unity では、Vulkan はレンダリング API の 1 つに過ぎず、Unity がすべての大変な作業を担います。チームは Vulkan を Unity で有効にするだけで、Vulkan を使用することができました。

OpenGL ES と Vulkan の違いをテストするために、チームはゲームを分析しました。チームはゲームで最も遅い部分の 1 つを見つけました。そこでは OpenGL は 60 FPS を達成できませんでした。OpenGL ES と Vulkan のパフォーマンスの違いを測定するために、チームはゲームのこの部分に基づいて合成ベンチマークを作成しました。

ベンチマークでは、API がレンダリングする反復可能シーンを 1 つ使用して、両方の API に対して一貫性のあるテストを行います。

以下に示すグラフを見ると、Vulkan はほとんどの時間 60 FPS を維持できるのに対して、OpenGL ES が悪戦苦闘していることがわかります。OpenGL ES と比較したパフォーマンスの全体の向上率は 15% です。チームはこれが非常に良い結果だとは思いましたが、この時点ではまだ Vulkan のことをグラフィックス API に過ぎないと考えていました。

以下の画像は、OpenGL ES と Vulkan を比較したベンチマークの結果を示したものです。

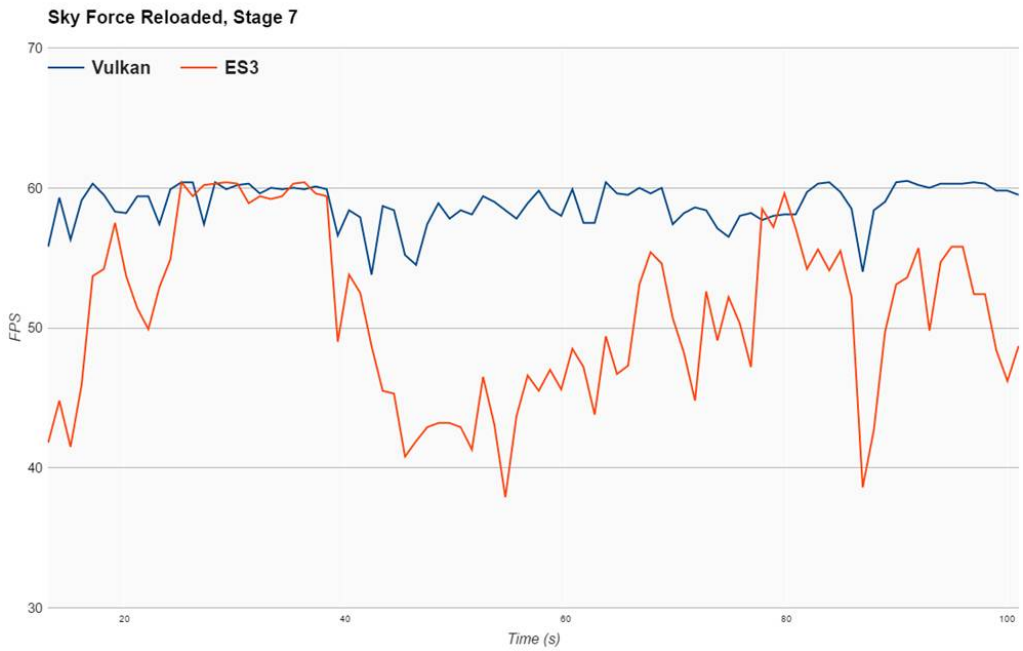


図 12-4 OpenGL ES と Vulkan を比較した Sky Force Reloaded のベンチマーク

Vulkan のその他のパフォーマンスのテスト

Vulkan がほとんどの時間を 60 FPS で実行できることから、チームは、Vulkan を使えばパフォーマンスはあとどれほど向上するだろうかと考えるようになりました。

開発者たちは、Vulkan でも 60 FPS を達成できなくなるまで、ベンチマーク レベルにオブジェクトを追加し始めました。OpenGL ES と Vulkan のギャップはさらに大きくなりました。Vulkan は OpenGL ES より平均で 32% 高速でした。

以下のグラフは、OpenGL ES と Vulkan を比較したベンチマークの結果を示したものです。シーンにオブジェクトが追加されています。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

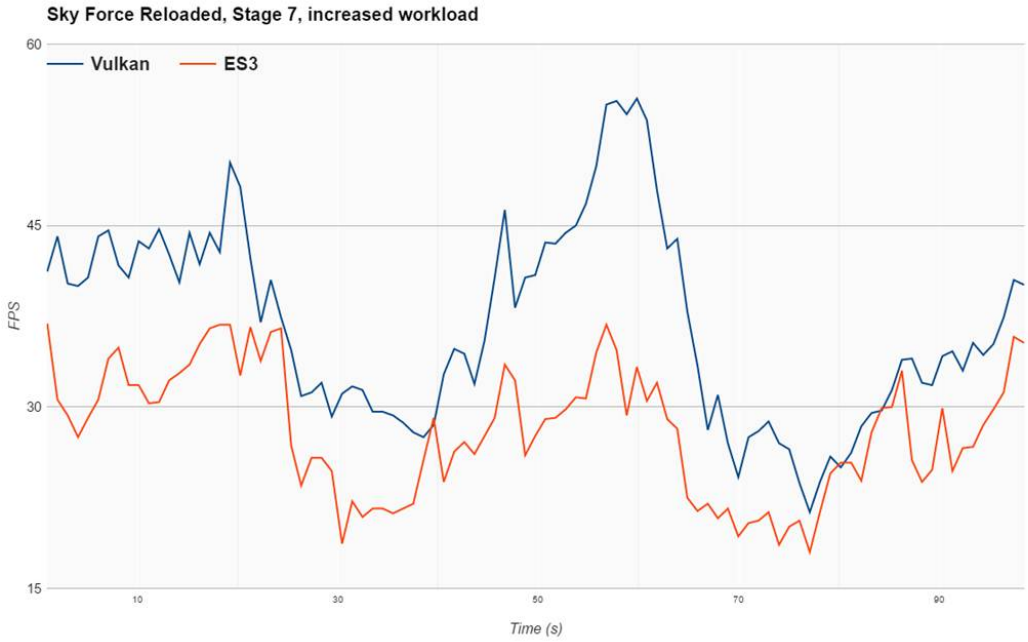


図 12-5 オブジェクトが追加された Sky Force Reloaded のベンチマーク

パフォーマンスにいくらか余裕があることは明らかだったため、チームはこの余剰パワーをゲームでどのように活用できるだろうかと考えました。チームはグラフィックス、パーティクル、オブジェクト、アニメーションをさらに追加しました。チームはゲームの見映えをさらに良くしながらも、60 FPS を維持できることを知りました。Vulkan のおかげで、事実上すべて無料で達成できたのです。

Vulkan ビデオの比較

比較ビデオは、*https://www.youtube.com/watch?v=VCSkp-QZ37M* でご覧いただけます。

このビデオでは、Vulkan バージョンと OpenGL ES バージョンのパフォーマンスを横に並べて比較しています。左側は、OpenGL ES を使用したゲームです。右側は、Vulkan を使用したゲームです。どちらのバージョンも、60 FPS で実行しています。

このフレーム レートで、Vulkan は OpenGL ES バージョンと比較して 6 倍の数の星と 2 倍の数の弾丸をレンダリングできます。OpenGL ES で同じ個数のオブジェクトをレンダリングすると、動きの激しいシーンでフレーム レートが大幅に低下してしまいます。Vulkan なら、OpenGL ES と比較して、同じフレーム レートで同じシーンにより多くのジオメトリを追加することができます。

以下の画像は、比較ビデオからの 1 フレームを示したものです。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

図 12-6 OpenGL ES と Vulkan を比較した Sky Force Reloaded のフレーム

Vulkan の消費電力

Sky Force Reloaded は、非常にアプリケーション プロセッサおよび GPU を集中的に使うゲームです。プレーヤーの中には、このゲームをプレイするとバッテリーの減り具合があまりにも早過ぎるとこぼす人もいます。チームはコンソール ゲーム同様のゲーム品質を損ないたくなかったため、Vulkan を役立てることはできないかチェックしました。チームは Vulkan でテストして、消費電力を低下させることで、バッテリー寿命が延びないか確認しました。Unity で Vulkan を有効にしたところ、消費電力が 10 ～ 12% 低下し、プレイ可能時間が 10 ～ 12% 延びました。

比較ビデオは、*https://www.youtube.com/watch?v=WI7nXq8oozw* でご覧いただけます。

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

付録 A
リビジョン

この付録では、本書の改訂版に加えられた変更について説明しています。

以下のセクションから構成されています。

- *A.1 リビジョン*（付録-A-292 ページ）.

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

A.1 リビジョン

この付録では、本書のこの版での追加点について説明しています。

表 A-1 バージョン 3.0 での追加点

追加点	場所	影響
カスタム シェーダでの Enlighten の使用	-	バージョン 3.0
反射の結合	*9.3 反射の結合*(9-174 ページ)	バージョン 3.0
Early-Z を使用する	*9.7 Early-Z を使用する*(9-197 ページ)	バージョン 3.0
ダーティ レンズ エフェクト	*9.8 ダーティ レンズ エフェクト*(9-198 ページ)	バージョン 3.0
ライト シャフト	*9.9 ライト シャフト*(9-201 ページ)	バージョン 3.0
フォグ エフェクト	*9.10 フォグ エフェクト*(9-205 ページ)	バージョン 3.0
ブルーム	*9.11 ブルーム*(9-212 ページ)	バージョン 3.0
氷壁のエフェクト	*9.12 氷壁のエフェクト*(9-219 ページ)	バージョン 3.0
手続き型スカイボックス	*9.13 手続き型スカイボックス*(9-225 ページ)	バージョン 3.0
ホタル	*9.14 ホタル*(9-233 ページ)	バージョン 3.0
接空間からワールド空間への法線変換ツール	*9.15 接空間からワールド空間への法線変換ツール*(9-237 ページ)	バージョン 3.0

表 A-2 バージョン 3.0_01 での変更点

変更点	場所	影響
「Early-Z を使用する」からリスト項目を削除	*9.7 Early-Z を使用する*(9-197 ページ)	バージョン 3.0

表 A-3 バージョン 3.1 での変更点

変更点	場所	影響
バーチャル リアリティに関する章を追加	*章 10 バーチャル リアリティ*(10-244 ページ)	バージョン 3.1

表 A-4 バージョン 3.2 での変更点

変更内容	位置	影響
Enlighten の章の情報と構成を更新	-	バージョン 3.2 の初回リリース

表 A-5 バージョン 3.3 での変更点

変更内容	位置	影響
Vulkan に関する章を追加	*章 12 Vulkan*(12-282 ページ)	バージョン 3.3 の初回リリース
Mali Graphics Debugger に関する章を追加	Removed in version 4.2.1	バージョン 3.3 の初回リリース

表 A-6 バージョン 3.3_01 での変更点

変更内容	位置	影響
セクション 8.3 のスクリーンショットを更新	*12.3 Unity における Vulkan の有効化*(12-286 ページ)	バージョン 3.3 の 2 回目のリリース

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い

表 A-7 バージョン 4.0 での変更点

変更点	ロケーション	影響
ARM Mobile Studio の章に追加。	Removed in version 4.2.1	バージョン 4.0 初版
Mali グラフィックス デバッガの全インスタンスを Graphics Analyzer に変更。	ドキュメント全体	バージョン 4.0 初版
Graphics Analyzer のコンテンツを ARM Mobile Studio の章内に移動。	Removed in version 4.2.1	バージョン 4.0 初版
ARM ブランド画像を更新。	ドキュメント全体	バージョン 4.0 初版
メイン テキスト内の Enlighten の参照をすべて削除。	ドキュメント全体	バージョン 4.0 初版

表 A-8 バージョン 4.1 での変更点

変更点	ロケーション	影響
ジオメトリのベストプラクティスの章を追加。	*章 5 リアルタイム 3D アートのベスト プラクティス: ジオメトリ*(5-78 ページ)	バージョン 4.1 初版
テクスチャリングのベストプラクティスの章を追加。	*章 6 リアルタイム 3D アートのベスト プラクティス: テクスチャリング*(6-94 ページ)	バージョン 4.1 初版
マテリアルとシェーダのベストプラクティスの章を追加。	*章 7 リアルタイム 3D アートのベスト プラクティス: マテリアルとシェーダ* (7-115 ページ)	バージョン 4.1 初版
高度な VR グラフィックス技術の章を追加。	*章 11 高度な VR グラフィックス手法* (11-257 ページ)	バージョン 4.1 初版

表 A-9 バージョン 4.2 での変更点

変更点	ロケーション	影響
ライティングのベストプラクティスの章を追加。	*章 8 リアルタイム 3D アートのベストプラクティス: ライティング*(8-127 ページ)	バージョン 4.2 初版
パフォーマンス解析の章を更新。	*章 3 パフォーマンス解析*(3-22 ページ)	バージョン 4.2 初版
Arm mobile studio の章を削除。	バージョン 4.2.1 で削除	バージョン 4.2 第 2 版

Copyright © 2014–2017, 2019–2021 Arm. All rights reserved.
非機密扱い