# Unity 개발자를 위한 ARM® 설명서

버전 4.2

## 모바일 게임 그래픽 최적화

arm


100140_0402_00_ko

# Unity 개발자를 위한 ARM® 설명서

## 모바일 게임 그래픽 최적화



릴리스 정보

문서 사용 기록

| 발행 | 날짜 | 기밀 상태 | 변경 |
|---|---|---|---|
| 0100-00 | 05 9월 2014 | 공개 문서 | 버전 1.0 최초 릴리스 |
| 0200-00 | 23 6월 2015 | 공개 문서 | 버전 2.0 최초 릴리스 |
| 0201-00 | 28 7월 2015 | 공개 문서 | 버전 2.1 최초 릴리스 |
| 0300-00 | 18 9월 2015 | 공개 문서 | 버전 3.0 최초 릴리스 |
| 0300-01 | 05 11월 2015 | 공개 문서 | 버전 3.0 2차 릴리스 |
| 0301-00 | 07 4월 2016 | 공개 문서 | 버전 3.1 최초 릴리스 |
| 0301-01 | 25 4월 2016 | 공개 문서 | 버전 3.1 2차 릴리스 |
| 0302-00 | 31 5월 2016 | 공개 문서 | 버전 3.2의 최초 릴리스 |
| 0303-00 | 17 3월 2017 | 공개 문서 | 버전 3.3의 최초 릴리스 |
| 0303-01 | 01 6월 2017 | 공개 문서 | 버전 3.3 2차 릴리스 |
| 0400-00 | 19 9월 2019 | 공개 문서 | 버전 4.0 최초 릴리스 |
| 0401-00 | 28 2월 2020 | 공개 문서 | 버전 4.1 최초 릴리스 |
| 0402-00 | 14 5월 2020 | 공개 문서 | 버전 4.2 최초 릴리스 |

*관련 참조*
*Non-Confidential Proprietary Notice* 페이지의 2
*기밀 상태* 페이지의 4
*제품 상태* 페이지의 4
*웹 주소* 페이지의 4

**Non-Confidential Proprietary Notice**

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*.



Arm Limited. 잉글랜드 등록 법인 02557590호.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

## 공개 소유권 고지 사항

이 문서는 저작권 및 기타 관련 권한에 의해 보호되며 이 문서에 포함된 정보의 실행 및 구현은 하나 이상의 특허 또는 출원 중인 특허 신청에 의해 보호될 수 있습니다. 이 문서의 어떠한 부분도 Arm의 명시적인 사전 서면 승인 없이는 어떠한 수단에 의해서든 어떠한 형태로도 복제할 수 없습니다. 구체적인 진술이 있는 경우가 아닌 한 이 문서로 금반언 또는 다른 방식에 의해 지적 재산권에 대한 명시적 또는 암시적 라이센스가 부여되지 않습니다.

귀하는 이 문서의 정보를 Arm의 사전 서면 동의 없이 구현 시 타사 특허를 침해하는지 확인할 목적으로 사용하거나 다른 사람이 사용하도록 허용하지 않는 것에 동의해야만 정보에 액세스할 수 있습니다.

이 문서는 "있는 그대로" 제공됩니다. Arm은 암시적 상품성 보증, 만족스러운 품질, 문서와 관련해 특정 목적 적합성이 포함되나 그에 국한되지 않는 명시적, 암시적 또는 법적 진술, 보증을 하지 않습니다. 의문을 방지할 수 있도록, Arm은 타사 특허, 저작권, 사업 비밀 또는 기타 권한의 범위 및 내용을 식별 또는 파악하기 위한 분석과 관련하여 어떠한 진술도 하지 않으며 그러한 분석에 착수하지도 않았습니다.

이 문서에는 기술적 부정확성 또는 오타가 포함되어 있을 수 있습니다.

법률에 의해 금지되지 않는 범위 내에서, Arm은 Arm이 해당 손해 발생 가능성에 대해 고지를 받았더라도 원인 및 책임 이론에 관계없이, 이 문서 사용에 의해 발생하는 직접적, 간접적, 특별적, 우발적, 징벌적 또는 결과적 손해가 포함되나 그에 국한되지 않는 손해에 대해 책임을 지지 않습니다.

이 문서는 상업적 항목으로만 구성됩니다. 귀하는 이 문서를 사용, 중복 또는 공개 시 이 문서 또는 문서의 어떤 부분을 관련 수출법을 위반하여 수출되지 않도록 보장하기 위해 관련 수출법 및 규정을 완전하게 준수할 책임이 있습니다. Arm의 고객을 가리키는 "파트너"라는 단어를 사용하더라도 다른 회사와의 파트너 관계를 형성하거나 이를 가리키려는 의도가 아닙니다. Arm은 언제든지 사전 통지 없이 이 문서를 변경할 수 있습니다.

이 약관에 포함된 조항 중 Arm과 이 문서가 포함되는 클릭 또는 서명을 통해 체결된 서면 계약의 조항과 충돌하는 것이 있는 경우 클릭 또는 서명을 통해 체결된 서면 계약이 이 약관의 충돌 조항에 우선하며 이를 대체합니다. 이 문서는 편의를 위해 다른 언어로 번역될 수 있으며 귀하는 이 문서의 영문본과 번역이 충돌하는 경우 영문본 계약의 약관이 우선 적용됨에 동의합니다.

Arm의 기업 로고 및 ® 또는 ™으로 표시된 단어는 미국 및/또는 다른 지역에서 Arm Limited(또는 자회사)의 등록 상표 또는 상표입니다. All rights reserved. 그 외 이 문서에 언급된 브랜드 및 이름은 해당 소유자의 상표일 수 있습니다. *http://www.arm.com/company/policies/trademarks*에 있는 Arm의 상표 사용 지침을 따르십시오.



Arm Limited. 잉글랜드 등록 법인 제02557590호.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

*관련 참조*
*릴리스 정보* 페이지의 2

*기밀 상태* 페이지의 4
*제품 상태* 페이지의 4
*웹 주소* 페이지의 4

### 기밀 상태

이 설명서는 기밀 문서가 아닙니다. 이 설명서의 사용, 복사 및 공개에 대한 권한은 ARM과 ARM으로부터 이 설명서를 제공받은 당사자가 동의한 계약 조건에 따라 라이센스 제한의 적용을 받을 수 있습니다.

액세스 무제한은 ARM의 내부 분류입니다.

*관련 참조*
*릴리스 정보* 페이지의 2
*Non-Confidential Proprietary Notice* 페이지의 2
*제품 상태* 페이지의 4
*웹 주소* 페이지의 4

### 제품 상태

이 설명서의 정보는 개발이 완료된 제품에 대한 최종 정보입니다.

*관련 참조*
*릴리스 정보* 페이지의 2
*Non-Confidential Proprietary Notice* 페이지의 2
*기밀 상태* 페이지의 4
*웹 주소* 페이지의 4

### 웹 주소

*developer.arm.com*

*관련 참조*
*릴리스 정보* 페이지의 2
*Non-Confidential Proprietary Notice* 페이지의 2
*기밀 상태* 페이지의 4
*제품 상태* 페이지의 4

# 컨텐츠
# Unity 개발자를 위한 ARM® 설명서 모바일 게임 그래픽 최적화

# 서문

이 서문에서는 다음을 소개합니다. *Unity 개발자를 위한 ARM® 설명서 모바일 게임 그래픽 최적화*.

여기에는 다음 내용이 포함됩니다.
- *설명서 개요* 페이지의 9.
- *사용자 의견* 페이지의 11.

## 설명서 개요

이 설명서는 모바일 플랫폼에서, 특히 Mali™ GPU를 탑재한 경우 Unity를 최대한 활용할 수 있는 애플리케이션 및 콘텐츠를 제작할 수 있도록 구성되었습니다.

### 제품 버전 상태

r*m*p*n* 식별자(예: r1p2)는 이 설명서에서 설명하는 제품의 버전 상태를 표시합니다. 여기서,

r*m* 제품의 주 버전을 식별합니다. 예: r1.

p*n* 제품의 부 버전 또는 수정 상태를 식별합니다. 예: p2.

### 대상 독자

### 설명서 사용

이 설명서는 다음 장으로 구성되어 있습니다.

*제 1 장 : 소개*
이 장에서는 ARM 모바일 게이밍 그래픽 최적화용 Unity 개발자 설명서를 소개합니다.

*제 2 장 : 응용 프로그램 최적화*
이 장에서는 Unity에서 응용 프로그램을 최적화하는 방법을 설명합니다.

*제 3 장 : 성능 분석*
이 장에서는 응용 프로그램 프로파일링에 대해 설명합니다.

*제 4 장 : 최적화 목록*
이 장에서는 Unity 응용 프로그램에 대한 여러 최적화를 소개합니다.

*제 5 장 : 실시간 3D 아트 모범 사례: 지오메트리*
이 장은 3D 애셋의 주요 지오메트리 최적화를 중심으로 설명합니다. 지오메트리 최적화는 효율적인 게임을 만들고 모바일 플랫폼에서 게임 성능 목표를 달성하는 데 도움이 됩니다.

*제 6 장 : 실시간 3D 아트 모범 사례: 텍스처링*
이 장은 게임의 시각 품질을 높이고 부드럽게 실행되도록 돕는 다양한 텍스처 최적화를 다룹니다.

*제 7 장 : 실시간 3D 아트 모범 사례: 머티리얼과 셰이더*
이 장에서는 게임의 시각 품질을 높이고 효율적으로 실행되도록 하는 다양한 머티리얼 및 셰이더 최적화를 다룹니다.

*제 8 장 : 실시간 3D 아트 모범 사례: 조명*
이 장은 게임이 부드럽게 실행되며 시각적으로 개선되도록 돕는 여러 조명 최적화를 다룹니다.

*제 9 장 : 고급 그래픽 기법*
이 장에서는 몇 가지 고급 그래픽 기법을 소개합니다.

*제 10 장 : 가상현실*
이 장에서는 응용 프로그램 또는 게임을 가상현실 하드웨어에서 실행되도록 조정하는 프로세스와 가상현실에서 반사를 구현할 때 몇몇 차이점에 대해 설명합니다.

*제 11 장 : 고급 VR 그래픽 기법*
이 장은 가상 현실 응용 프로그램의 그래픽 성능을 향상시키는 데 사용할 수 있는 다양한 기술을 설명합니다.

*제 12 장 : **Vulkan***
이 장에서는 Vulkan과 Vulkan을 활성화하는 방법을 설명합니다.

*제 13 장 : **ARM Mobile Studio***
이 장은 Arm Mobile Studio 도구를 다룹니다.

*부록 A: 버전*
이 부록에서는 이 설명서의 버전 간 변경 사항을 설명합니다.

### 용어집

ARM® 용어집은 ARM 설명서에서 사용되는 용어의 목록으로서 해당 용어의 정의를 포함합니다. ARM 용어집에는 ARM에서 사용하는 의미가 일반적으로 사용되는 의미와 다르지 않은 한 업계 표준인 용어는 포함되지 않습니다.

자세한 내용은 *ARM® 용어집* 섹션을 참조하십시오.

### 인쇄 규칙

*기울임꼴*
특수한 용어를 소개하거나 상호 참조와 인용을 나타냅니다.

굵은 글꼴
메뉴 이름과 같은 인터페이스 요소를 강조 표시합니다. 신호 이름을 나타냅니다. 해당되는 경우 설명 목록에 있는 용어에도 사용됩니다.

고정 폭
명령, 파일 및 프로그램 이름, 소스 코드와 같이, 키보드로 입력할 수 있는 텍스트를 나타냅니다.

고정 폭 글꼴
명령 또는 옵션에 허용되는 약어를 나타냅니다. 전체 명령 또는 옵션 이름 대신 밑줄 표시된 텍스트를 입력할 수 있습니다.

*고정 폭 기울임꼴*
인수를 특정한 값으로 바꾸어야 하는 고정 폭 텍스트에 대한 인수를 나타냅니다.

고정 폭 굵은 글꼴
예시 코드 외부에서 사용 시 언어 키워드를 나타냅니다.

<and>
대체 가능한 용어가 코드 또는 코드 조각에 나타나는 어셈블러 구문을 위해 이러한 용어를 둘러쌉니다. 예:

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

작은 대문자
본문에서 *ARM® 용어집*에 정의된 특정한 기술적 의미를 지니는 일부 용어에 사용됩니다. 예를 들어 IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, UNPREDICTABLE 등입니다.

## 추가 정보

ARM 게시물

기타 게시물

# 사용자 의견

## 이 제품에 대한 사용자 의견

이 제품에 대한 의견 또는 제안이 있으시면 해당 공급업체에 연락하여 다음 정보를 제공하시기 바랍니다.

- 제품 이름.
- 제품 버전.
- 가능한 한 많은 정보를 포함한 설명. 해당할 경우 증상 및 진단 절차를 포함합니다.

## 내용에 대한 사용자 의견

내용과 관련된 의견이 있으시면 다음 사항을 기재하여 *errata@arm.com* 으로 전자 메일을 보내 주시기 바랍니다.

- 제목 *Unity 개발자를 위한 ARM 설명서 모바일 게임 그래픽 최적화*.
- 번호 100140_0402_00_ko.
- 해당할 경우 의견에 해당하는 페이지 번호
- 의견에 대한 간략한 설명

ARM추가 및 향상되었으면 하는 기능에 대한 일반적인 제안도 환영합니다.

———— 참고 ————

ARMAdobe Acrobat 및 Acrobat Reader에서만 PDF를 테스트했으며 다른 PDF 리더에서 이 설명서를 사용할 경우 문서 품질을 보장할 수 없습니다.

————————

# 제 1 장
# 소개

이 장에서는 ARM 모바일 게이밍 그래픽 최적화용 Unity 개발자 설명서를 소개합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 1.1 Unity 개요

Unity는 2D 게임, 3D 게임 및 기타 응용 프로그램을 작성 및 배포할 수 있게 해주는 소프트웨어 플랫폼입니다.

이 설명서는 모바일 플랫폼에서, 특히 Mali™ GPU를 탑재한 경우 Unity를 최대한 활용할 수 있는 애플리케이션 및 콘텐츠를 제작할 수 있도록 구성되었습니다. 응용 프로그램의 성능을 개선하는 데 사용할 수 있는 기법과 모범 사례에 대해서도 설명합니다.

——— 참고 ———

달리 명시하지 않은 한, 여기에서 설명하는 기법은 다른 플랫폼에도 적용할 수 있습니다.

———————————

## 1.2 ARM® Mali™ GPU 정보

ARM Mali GPU는 모바일 또는 임베디드 기기용으로 설계되었습니다. ARM Mali GPU는 다음 제품군으로 나뉩니다.

**Bifrost GPU**

Bifrost GPU에는 정점, 조각, 형상, 공간 분할 및 컴퓨팅 처리를 수행하는 통합 셰이더 코어가 있습니다. 이 GPU는 Vulkan, OpenGL ES 1.1부터 OpenGL ES 3.2, OpenCL 1.2 Full Profile을 사용하는 그래픽 및 컴퓨팅 응용 프로그램에 사용됩니다.

**Midgard GPU**

Midgard 계열의 GPU는 정점, 조각, 지오메트리, 모자이크, 컴퓨팅 처리를 수행하는 통합 쉐이더 코어를 포함합니다. 이들은 Vulkan, OpenGL ES 1.1 ~ OpenGL ES 3.2 및 OpenCL 1.2 Full Profile을 사용하는 그래픽 및 컴퓨팅 응용 프로그램에 사용됩니다.

**Utgard GPU**

Utgard GPU에는 정점 프로세서와 하나 이상의 조각 프로세서가 있습니다. 이들은 OpenGL ES 1.1 및 2.0을 사용하는 그래픽 전용 응용 프로그램에 사용됩니다.

## 1.3 최적화 정보

그래픽은 사물을 제대로 보이게 만드는 과정입니다. 최적화는 최소의 컴퓨팅 노력으로 사물을 제대로 보이게 만드는 과정입니다. 컴퓨팅 파워와 메모리 대역을 제한하여 전력 소비를 낮게 유지하는 모바일 디바이스의 경우에는 최적화가 특히 중요합니다.

## 1.4 얼음 동굴 데모 개요

얼음 동굴 데모는 모바일 장치를 위한 고품질 시각적 콘텐츠를 생성하기 위해 최적화 테크닉을 다수 사용해 ARM로 생성한 데모 응용 프로그램입니다.

이 설명서에서는 얼음 동굴 데모에서 사용된 그래픽 기법과 프로젝트 개발 중 발생한 문제에 대한 해결책을 설명합니다.

얼음 동굴 데모는 ARM Cortex® -A57 MP4 프로세서와 ARM Mali -T760 MP8 GPU를 사용한 모바일 장치용으로 개발되어 테스트되었습니다.

# 제 2 장
# 응용 프로그램 최적화

이 장에서는 Unity에서 응용 프로그램을 최적화하는 방법을 설명합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 2.1 최적화 프로세스

최적화는 응용 프로그램을 보다 효율적으로 만드는 프로세스입니다. 그래픽 응용 프로그램의 경우에는 일반적으로 응용 프로그램의 속도를 개선하도록 수정하는 것을 의미합니다.

예를 들어 프레임 속도가 낮은 게임은 뭔가 덜거덕거리는 것처럼 보일 수 있습니다. 이는 사용자에게 나쁜 인상을 주며 게임을 플레이하기 어렵게 만들 수 있습니다. 최적화를 사용하여 게임의 프레임 속도를 개선해 더 우수하고 원활한 게이밍 경험을 제공할 수 있습니다.

코드를 최적화하려면 최적화 프로세스를 사용합니다. 최적화는 성능 문제를 발견하고 제거하도록 안내하는 반복적 프로세스입니다.

최적화 프로세스는 다음 단계들로 이루어집니다.

1. 프로파일러를 사용하여 응용 프로그램을 측정합니다.
2. 데이터를 분석하여 병목을 확인합니다.
3. 적용할 최적화를 결정합니다.
4. 최적화가 효과를 발휘했는지 확인합니다.
5. 성능이 만족스럽지 않으면 1단계로 돌아가 프로세스를 반복합니다.

다음은 최적화 프로세스의 예입니다.

1. 게임이 필요한 성능을 내지 못할 경우 프로파일러를 사용하여 측정할 수 있습니다.
2. 프로파일러를 사용하여 측정 결과를 분석함으로써 성능 문제의 원인을 파악해 격리할 수 있습니다.
3. 예를 들어 너무 많은 정점을 렌더링하는 것이 게임의 문제입니다.
4. 메쉬의 정점 수를 줄입니다.
5. 게임을 다시 실행하여 최적화가 효과를 발휘했는지 확인합니다.

이렇게 해도 게임이 기대한 대로 동작하지 않을 경우 프로세스를 다시 시작하여 응용 프로그램을 다시 프로파일링하고 문제의 다른 원인을 찾습니다.

이 프로세스를 여러 번 반복해야 할 수 있습니다. 최적화는 여러 영역에서 성능 문제가 발견될 수 있는 반복적 프로세스이기 때문입니다.

## 2.2 Unity 품질 설정

응용 프로그램에 올바른 설정을 선택하려면 Unity 품질 설정에 대해 아는 것이 도움이 됩니다.

Unity는 게임의 이미지 품질을 변경할 수 있는 여러 옵션을 제공합니다. 이들 옵션 중 일부는 연산 비용이 높고 게임 성능에 부정적인 영향을 미칠 수 있습니다.

다음 그림은 검사기의 품질 설정입니다.

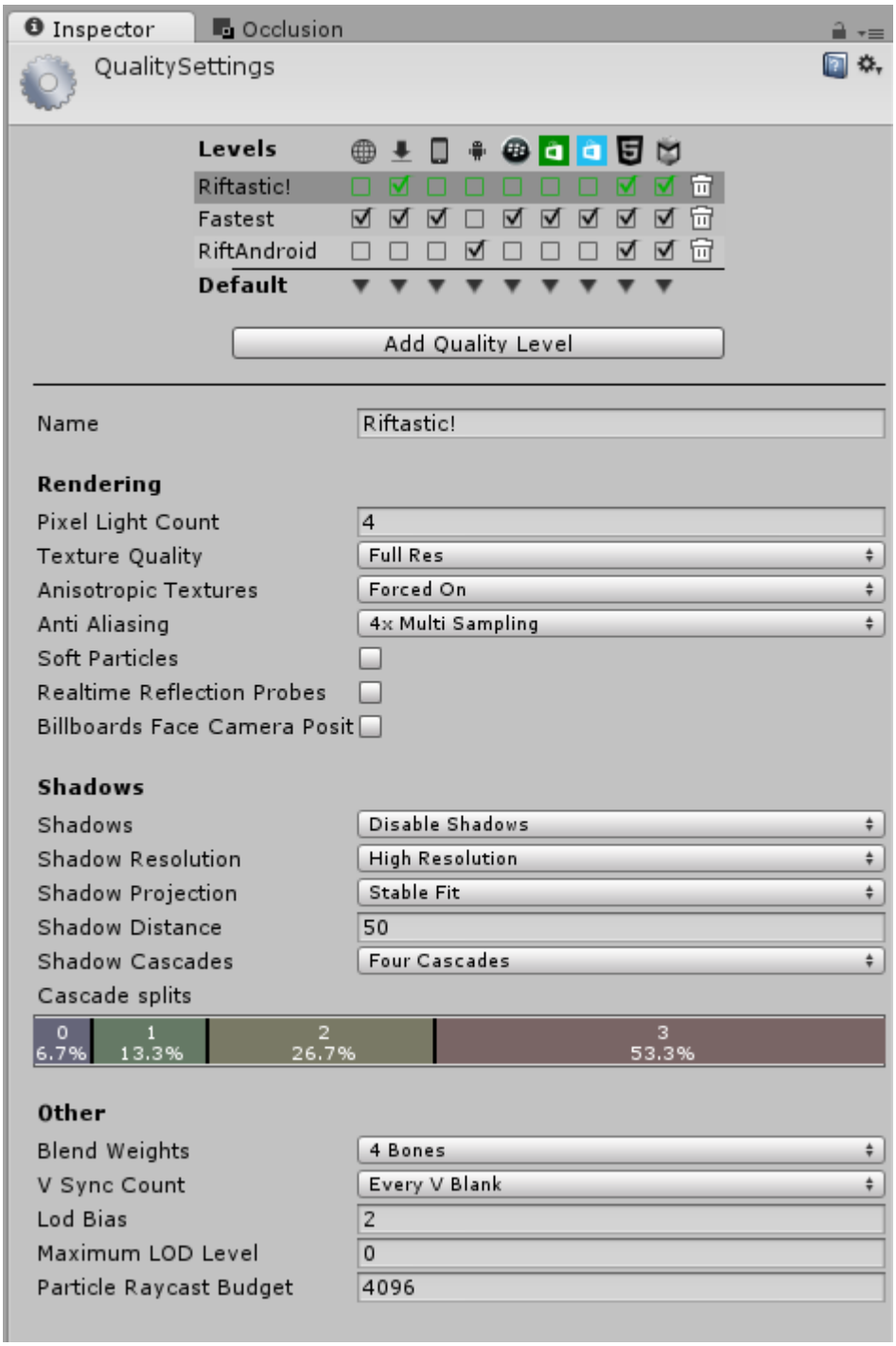


그림 2-1 품질 설정

약간의 성능 절충으로 게임의 이미지 품질을 향상할 수 있는 다수의 옵션이 존재합니다. 예를 들어 게임의 프레임 레이트가 낮을 경우 GPU가 복잡한 그래픽 효과를 실행할 때 너무 많은 정보를 처리할 수 있습니다. 그래픽 품질에 비교적 적은 영향을 미치도록 그림자, 조명과 같은 그래픽 효과를 덜 복잡한 버전으로 실행할 수 있습니다. 단순한 효과는 GPU 부하를 크게 낮춰 프레임 레이트를 높일 수 있습니다.

조명의 기본 설정은 모바일 디바이스에는 너무 복잡한 경우가 가끔 있으므로, 모바일 플랫폼용으로 제작된 일부 게임은 복잡한 기법을 피하거나 게임 고유의 기법을 사용합니다. 여기에는 라이트맵에 미리 조명을 베이크하거나 그림자를 캐스트하는 대신 텍스처를 투영하는 것과 같은 기법이 포함될 수 있습니다.

Project Settings > Quality에는 게임 성능에 큰 영향을 미치는 여러 옵션이 있습니다.

**Pixel Light Count**

Pixel Light Count는 특정 픽셀에 영향을 미칠 수 있는 라이트의 수입니다. 픽셀 라이트 카운트가 크면 많은 계산이 필요합니다. 대부분의 게임은 이미지 품질에 최소한의 영향을 미치는 동적 및 실시간 라이트를 매우 적게 사용할 수 있습니다. 조명이 성능 문제를 일으키는 경우 게임에서 라이트맵과 투영된 텍스처와 같은 기법을 사용할 것을 고려해 보십시오.

**Texture Quality**

Texture Quality는 GPU에 부하를 줄 수 있지만 일반적으로 성능 문제를 초래하지는 않습니다. 텍스처 품질을 낮추면 게임의 이미지 품질에는 부정적인 영향을 줍니다. 그러므로 불가피한 경우에만 이 품질을 낮추십시오. 얼음 동굴 데모에서 Texture Quality는 최대 해상도로 설정되어 있습니다.
텍스처가 성능 문제를 유발하는 경우 밉매핑을 사용해 보십시오. 밉매핑은 이미지 품질에 영향을 주지 않으면서 연산 및 대역폭 요구 사항을 낮춥니다.

**AntiAliasing**

AntiAliasing은 삼각형 에지 주위의 픽셀을 블렌딩하는 에지 스무딩 기법입니다. 이 기법은 게임의 이미지 품질을 현저히 개선해줍니다. 안티앨리어싱 방법은 여러 가지가 있지만, 이 경우에는 *멀티 샘플 안티앨리어싱*(MSAA)이 토글됩니다. 4x MSAA는 Mali GPU 부하가 매우 낮은 연산이므로 가능하다면 항상 이 옵션을 사용하십시오.

**Soft Particles**

Soft Particles는 깊이 텍스처로 렌더링 또는 지연 모드에서의 렌더링을 요구합니다. 이는 GPU 부하를 증가시키지만 파티클에서 사실적 비주얼을 구현한다는 데 가치가 있습니다. 모바일 플랫폼에서는 깊이 텍스처로 또는 깊이 텍스처로부터 렌더링은 소중한 대역폭을 소모하며, 지연 경로를 사용하는 렌더링은 MSAA에 액세스할 수 없음을 의미합니다. 게임에서 소프트 파티클을 사용할 이유가 충분한지 판단해야 합니다.

**Anisotropic Textures**

Anisotropic Textures는 높은 경사도에서 그려진 텍스처에서 왜곡을 제거하는 기법입니다. 이 기법은 이미지 품질을 개선하지만 연산 비용이 높은 기법입니다. 왜곡이 특히 두드러지지 않은 한 이 기법은 사용하지 마십시오.

**Shadows**

Shadows는 고품질인 경우 연산 비용이 높을 수 있습니다. 그림자가 성능 문제를 유발할 경우 단순한 그림자를 사용하거나 그림자를 끄십시오. 게임에서 그림자가 중요할 경우 투영된 텍스처와 같은 단순한 동적 그림자 처리 기법을 사용할 것을 고려해 보십시오.

**Realtime Reflection Probes**

Realtime Reflection Probes 옵션은 런타임 성능에 상당히 부정적인 영향을 미칠 수 있습니다.

반사 프로브가 렌더링될 때 큐브맵의 각 면이 프로브의 원점에 위치한 카메라에 의해 개별적으로 렌더링됩니다. 상호반사가 고려되는 경우 이 프로세스는 모든 반사 바운스 레벨에서 이루어집니다. 광택 반사인 경우 블러링 프로세스를 적용하기 위해 큐브맵 밉맵도 사용됩니다.

다음 요소가 큐브맵의 렌더링에 영향을 미칩니다.

큐브맵 해상도
: 큐브맵 해상도가 높을수록 렌더링 시간이 증가합니다. 필요한 품질에 맞춰 최대한 낮은 해상도의 큐브맵을 사용하십시오.

컬링 마스크
: 반사에서 관련이 없는 지오메트리가 렌더링되지 않도록 큐브맵을 렌더링할 때 컬링 마스크를 사용하십시오.

큐브맵 업데이트
: Refresh Mode 옵션은 큐브맵의 업데이트 주기를 정의합니다.
  - Every Frame 옵션은 프레임마다 큐브맵을 렌더링합니다. 이 옵션은 연산 비용이 가장 높으므로 필요하지 않으면 사용하지 마십시오.
  - On Awake 옵션은 장면이 시작될 때 런타임에서 한 번 큐브맵을 렌더링합니다.
  - Via Scripting 옵션은 사용자에게 큐브맵 업데이트 시점을 제어할 수 있게 해줍니다. 이 옵션을 사용하면 업데이트 실행 조건을 지정하여 런타임 리소스의 사용을 제한할 수 있습니다.

# 제 3 장
# 성능 분석

이 장에서는 응용 프로그램 프로파일링에 대해 설명합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 3.1 성능 분석 정보

모바일 장치에서 게임 성능을 테스트하기 위해 프로파일러나 그래픽 디버거 같은 평가 도구를 사용할 수 있습니다. 이런 도구는 게임에서 장치 리소스를 정확히 어떻게 사용하는지 자세한 내용을 보여줍니다.

*Unity 내장 프로파일러* 및 *Arm Mobile Studio의 Streamline* 등의 프로파일러는 연결된 장치에서 실행되는 게임의 장면을 분석해 응용 프로그램이 가장 많은 시간을 소요하는 장소를 보여줍니다. 이런 도구는 데이터를 그래프로 표시하여 장치가 CPU와 GPU 작업량을 처리하는 방식을 시각화합니다. 이를 통해 게임의 문제 영역을 식별하고 성능 최적화 기회를 발견할 수 있습니다.

*Unity의 내장 프레임 디버거*와 *Arm Mobile Studio의 Graphics Analyzer* 같은 그래픽 디버거는 게임에서 발생하는 OpenGL ES 또는 Vulkan API 호출을 드로우 콜별로 확인하여 렌더링 문제 파악, 장면 LoD 분석, 최적화 가능한 높은 비용의 쉐이더 탐색을 할 수 있습니다.

## 3.2 Unity 게임 사례 프로파일링

스트림라인은 *ARM Mobile Studio*(AMS) 최적화 및 디버깅 소프트웨어 도구 스위트의 일부로 제공되는 다목적 성능 분석 도구입니다.

스트림라인은 Android 장치의 여러 소스에서 샘플 및 이벤트에 기반한 성능 데이터를 수집합니다. 타임라인 보기 같은 다양한 보기로 종합적인 결과를 표시합니다. 타임라인 보기 상단에는 수집된 시스템 성능 카운터를 표시하며, 하단에는 동일한 타임라인에 대한 다양한 정보 유형을 보여줍니다.

다음 스크린샷은 하단에 히트맵을 표시하도록 선택된 타임라인 보기 스크린샷입니다.

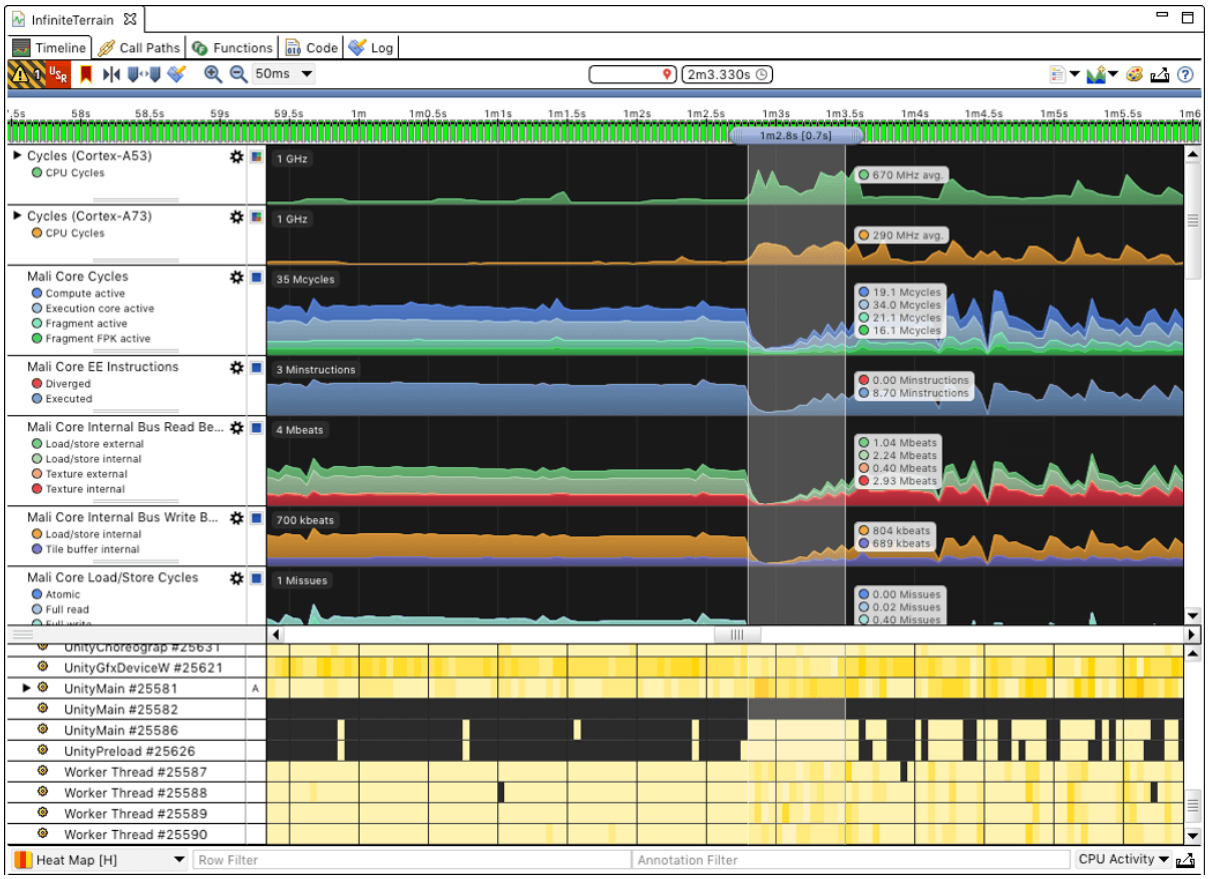


그림 3-1 스트림라인 타임라인 히트맵 보기

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 3.2.1 예시 장면 분석

다음 분석은 카메라가 회전하고 돌아갈 때 신규 지형이 즉시 생성되는 경우 절차적으로 생성된 지형으로 구성됩니다.

장면이 확대되면 카메라와 매우 먼 타일은 삭제됩니다. 장면의 복잡성을 유지하는 타일 삭제는 시간이 지나도 거의 비슷하게 유지됩니다. 카메라가 천천히 움직이는 경우 새 지형 생성 속도가 느려집니다. 카메라 속도가 빨라지면 지형 생성 속도도 즉시 더 빠른 속도로 증가합니다.

다음은 각 타일 가장자리가 어두운 색으로 렌더링되며 각 타일 크기를 표시하는 스크린샷입니다.

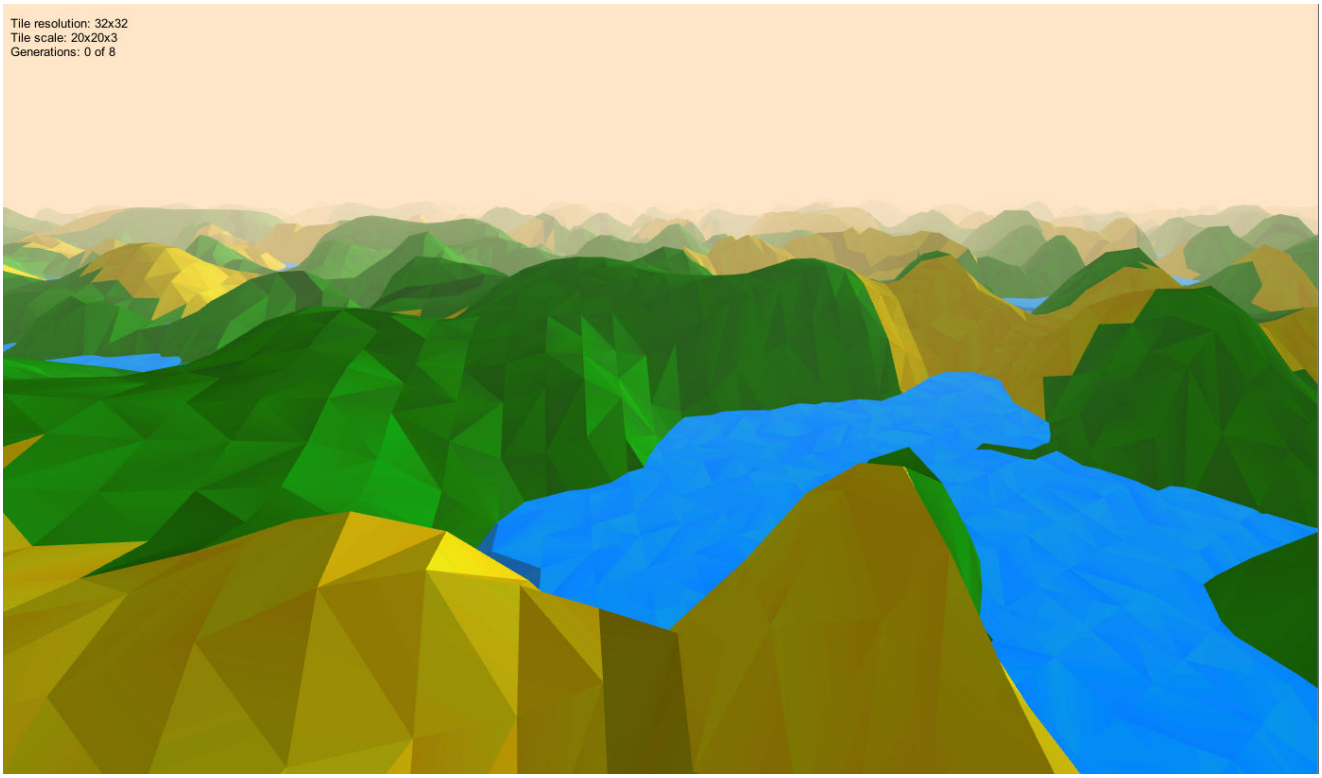


그림 3-2 렌더링된 타일

즉시 지형 타일을 생성하는 것은 컴퓨팅 비용이 높습니다. 컴퓨팅 비용을 최소화하기 위해 *Unity Job Scheduler*(UJS)는 주요 Unity 스레드가 완료되지 않는 백그라운드 스레드를 호출합니다.

이렇게 함으로써 사용자에게 새 지형이 생성될 때마다 끊기지 않고 일정한 프레임 속도 경험을 보장합니다.

예시 데모는 시각적으로 동일하게 보이지만 타일이 다르게 생성되는 4개의 다른 장면을 볼 수 있도록 구성되었습니다.

다음 다이어그램에 표시된 것처럼 지형은 고정된 크기의 지형 블록 여러 개로 구성됩니다. 각 블록은 여러 개의 고정 해상도 메시와 녹색 지형 1개, 노란색 지형 1개, 물 지형 1개로 구성됩니다. 렌더링 거리는 플레이어 주변에 생성되는 타일 수를 제어합니다.

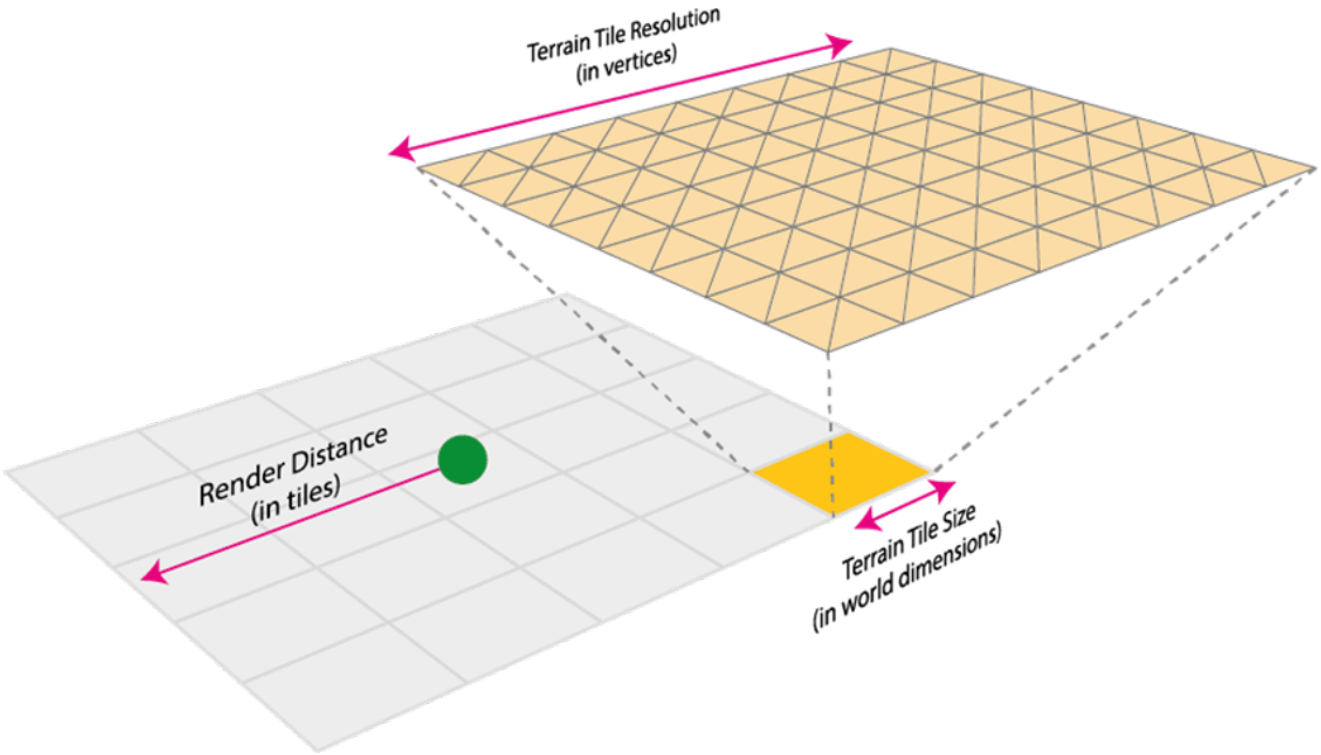


그림 3-3 지형 타일 해상도

4개 장면은 다음과 같이 구성됩니다.

표 3-1 예시 장면 구성 설정

| 장면 | 렌더링 거리 | 지형 타일 크기 | 지형 해상도 | 병렬 지형 생성 수 |
|---|---|---|---|---|
| 1 | 3 | 20x20 | 32x32 | 8 |
| 2 | 3 | 20x20 | 32x32 | 1 |
| 3 | 6 | 10x10 | 16x16 | 8 |
| 4 | 6 | 10x10 | 16x16 | 1 |

2017년 화웨이 P10에서 프로파일링을 실행했습니다. 화웨이 P10은 HiSilicon Kirin 960 칩을 사용했으며 스펙은 다음과 같습니다.

- 4개의 고성능 ARM Cortex -A73 CPU 코어.
- 4개의 고효율 ARM Cortex -A53 CPU 코어.
- ARM Mali -G71 MP8 GPU.

### 3.2.2 Unity에서 프로파일링

Unity의 프로파일러를 설명한 것입니다. 그러나 배터리 사용 등 핵심 데이터가 누락되었습니다. 따라서 Unity의 프로파일러를 사용해 더 자세한 분석을 위한 스트림라인으로 정보를 전달할 수 있습니다.

Unity의 프로파일러는 작업이 예약되면 이를 표시합니다. 그러나 프로파일러는 가용 CPU나 GPU 리소스 세부 사항이나 리소스가 어떻게 사용되는지는 표시하지 않습니다.

대상 응용 프로그램이 60*초당 프레임 수*(FPS)를 기록할 수 있지만 다른 영역을 모니터링하는 것도 중요합니다. 이때 영역에는 CPU 코어 활용 및 GPU 성능 정보도 포함합니다. 스트림라인은 핵심 영역을 모니터링하는 데도 사용될 수 있습니다.

다음 그림은 Unity에 내장된 프로파일러에서 이용 가능한 정보 스크린샷입니다.

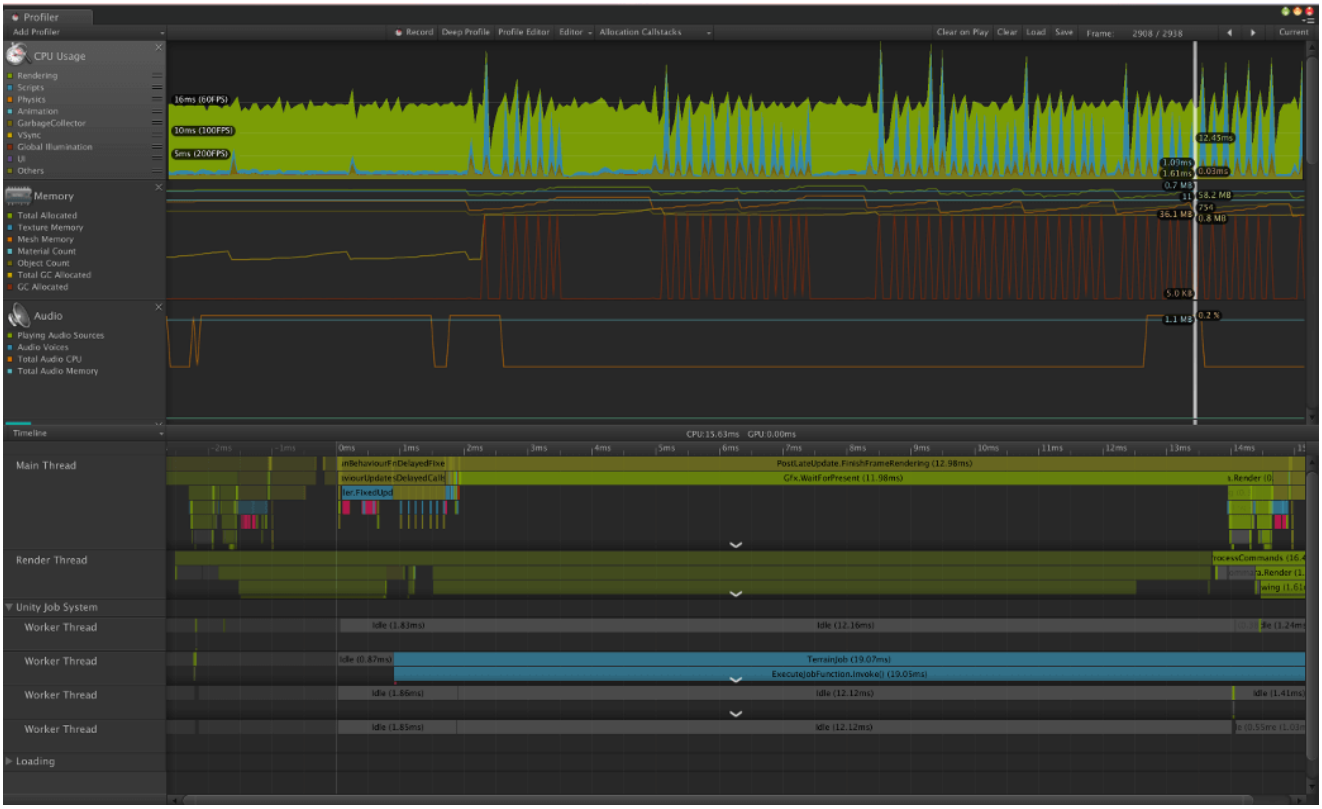


그림 3-4 Unity 프로파일러

### 3.2.3 스트림라인을 사용한 Unity 프로파일링 기능

스트림라인은 응용 프로그램 성능을 스레드 단위 수준까지 최적화하는 데 도움이 되는 여러 핵심 프로파일링 기능을 제공합니다.

이 예시 장면은 스트림라인 내에서 3개 유형의 각각 다른 주석을 사용해 수정되었습니다. 사용된 주석은 다음과 같습니다.

1. 마커 – 마커는 타임라인 보기 상단에 표시되는 라벨이 있는 단일 시점입니다.
2. 채널 – 채널은 각 스레드와 개별 열 정보를 제공합니다.
3. *사용자 작업 맵* (CAM) – CAM은 복잡한 종속성이 있을 수 있는 스레드 분산 활동을 표시할 수 있습니다. 각 CAM은 *사용자 인터페이스*(UI)의 하단에 각각 표시됩니다.

다음 스크린샷은 타임라인 보기에서 비정상적 프레임 속도가 어떻게 보이는지 나타내는 예입니다.

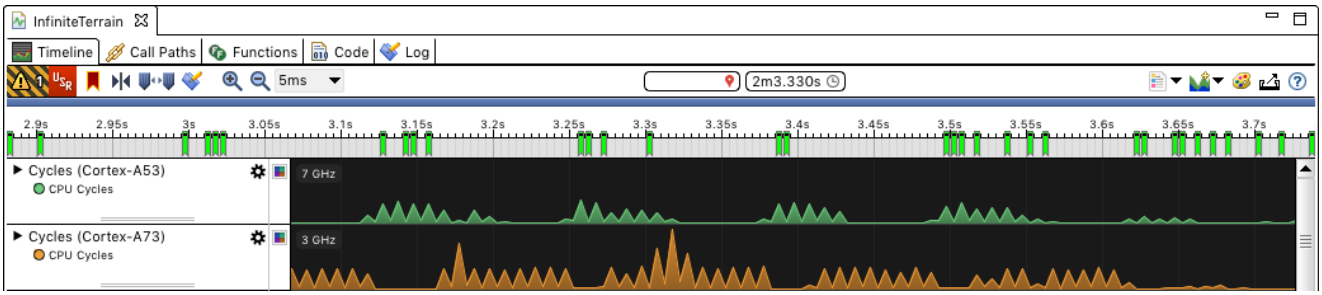


그림 3-5 프레임 속도

프레임 속도가 의도처럼 매끄럽지 않으며 모든 CPU 코어에 집중 작업이 상당히 발생하는 것을 볼 수 있습니다. 프레임 속도가 느려졌다가 최대로 상승하며 중간에 멈추기도 합니다. 그러나 이 결과는 지형 생성시 예상되는 결과와 일치합니다. 문제를 자세히 살펴보면 좋습니다.

스트림라인의 타임라인 보기는 2개 부분으로 나뉘어져 있습니다. 상단 보기는 메트릭 그래프를 표시하며 하단에는 다양한 프로필을 표시합니다. 히트 맵 같은 프로필은 이 작업이 시스템에 어떻게 분배되는지 표시합니다. 상단 타임라인은 특정 처리나 스레드에 영향을 주는 것만 표시하도록 필터를 적용할 수 있습니다.

먼저 UnityMain 스레드를 선택하고 모든 Worker Thread 스레드를 선택하면 히트 맵을 살펴볼 수 있습니다. CPU 작업이 메인 Unity 스레드와 작업 스케줄러의 스레드에 어떻게 나뉘어져 있는지 볼 수 있습니다.

다음 그림은 UnityMain 스레드의 CPU 프로필에 대한 스크린샷입니다. Cortex -A73 CPU에 대량 작업이 있습니다.


그림 3-6 Unity Main 스레드의 CPU 프로필

다음 그림은 모든 Worker Thread 스레드의 CPU 프로필에 대한 스크린샷입니다. 이 스레드는 Cortex -A73 과 Cortex -A53 CPU의 소량 작업을 보여줍니다.

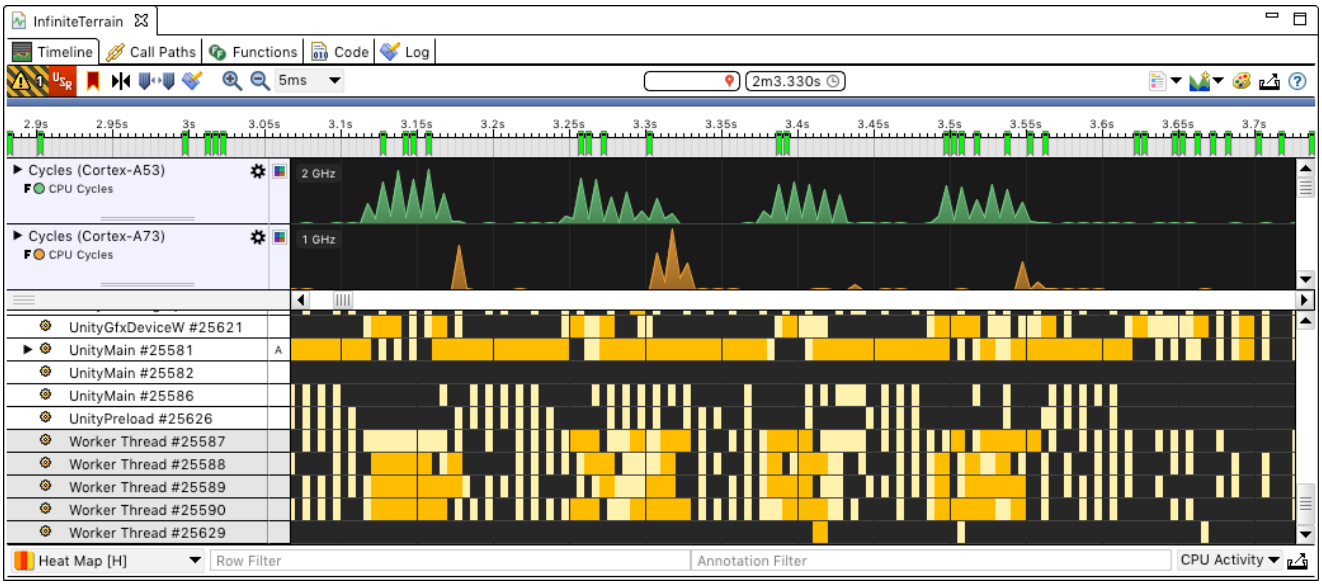


그림 3-7 Worker Thread 스레드의 CPU 프로필

다음 그림은 UnityMain 스레드의 CPU 확대 보기 스크린샷입니다. 스크린샷 왼쪽 UnityMain 숫자 옆에 A 아이콘이 표시됩니다. A 아이콘은 *스트림라인 주석 채널*(SAC)이 있음을 나타냅니다.

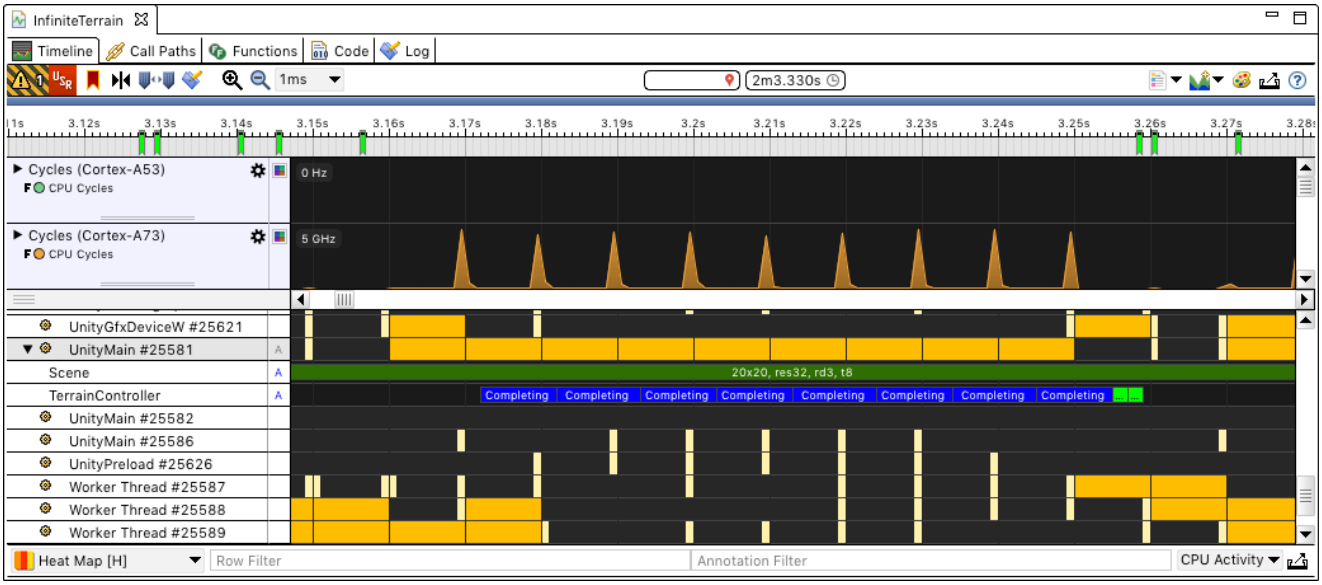


그림 3-8 UnityMain 스레드 확대

주석은 게임 컨트롤의 스트림라인 채널 장면과 TerrainController에 표시됩니다. 장면 채널은 처리 중인 장면을 보여줍니다. 이전 그림은 실행 중인 버전이 20x20, 32x32 장면이며, 렌더링 거리 3과 8 스레드가 동시에 실행 중임을 보여줍니다.

TerrainController 채널은 중요 코드 부분이 Unity 메인 스레드에서 실행되고 있음을 표시하는 데 사용됩니다. 파란색 블록은 Terrain 작업이 완료되면 실행되는 코드를 나타냅니다. 초록색 블록은 새 지형이 생성 예약되었을 때 표시됩니다.

모든 주요 스레드 작업은 작업이 완료되거나, 최종 메시를 생성하여 장면에 삽입되어야 하는 작업을 기준으로 합니다.

특정 스레드에 집중하면 특정 시간의 분석에 제약이 생길 수 있습니다. 스트림라인의 캘리퍼로 분석을 위한 특정 시간 영역을 표시할 수 있습니다.

다음 그림은 이 캘리퍼의 스크린샷입니다. 파란색 바와 화살표가 캘리퍼를 나타내며 타임라인 위에 위치합니다. 이 장면에서는 지형 완성과 관련된 작업에 집중한 기간이 선택되었습니다.

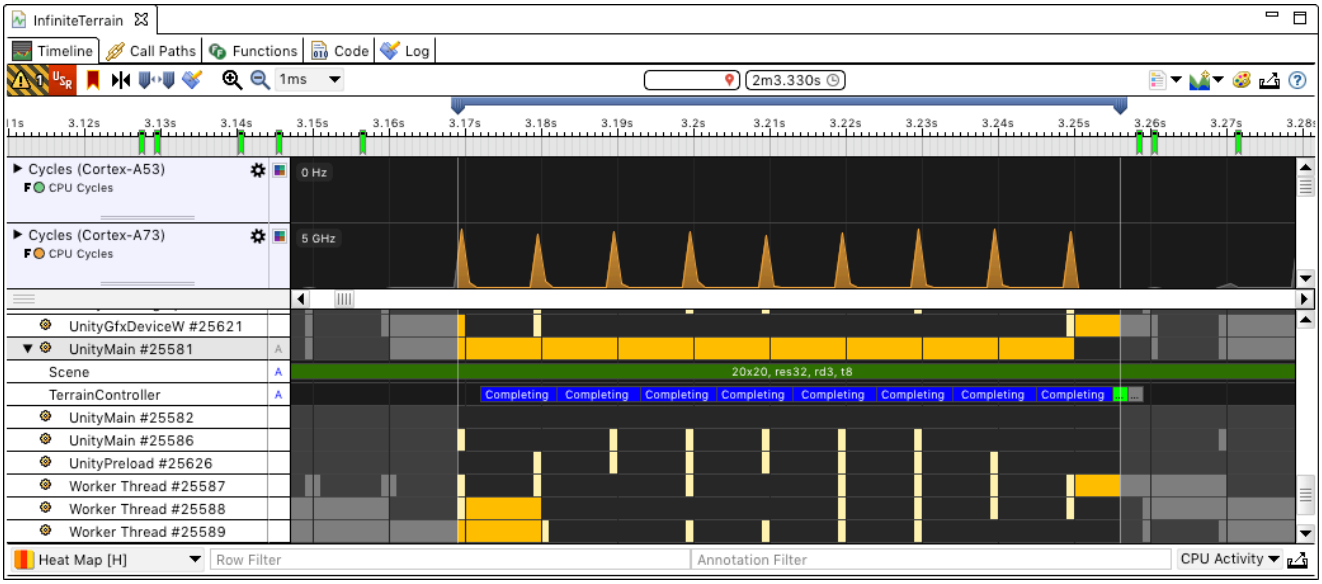


그림 3-9 스트림라인 캘리퍼

경로 호출 보기에서 캘리퍼로 선택된 시간 영역 동안 시간이 어떻게 사용되었는지 알 수 있습니다. *Unity용 IL2CPP 스크립팅 백엔드*를 사용하기 때문에 런타임에서 Mono를 사용하는 것보다 더 많은 정보를 얻을 수 있습니다.

다음 그림은 Unity 내에서 IL2CPP 스크립팅 백엔드를 세부 보기한 스크린샷입니다.

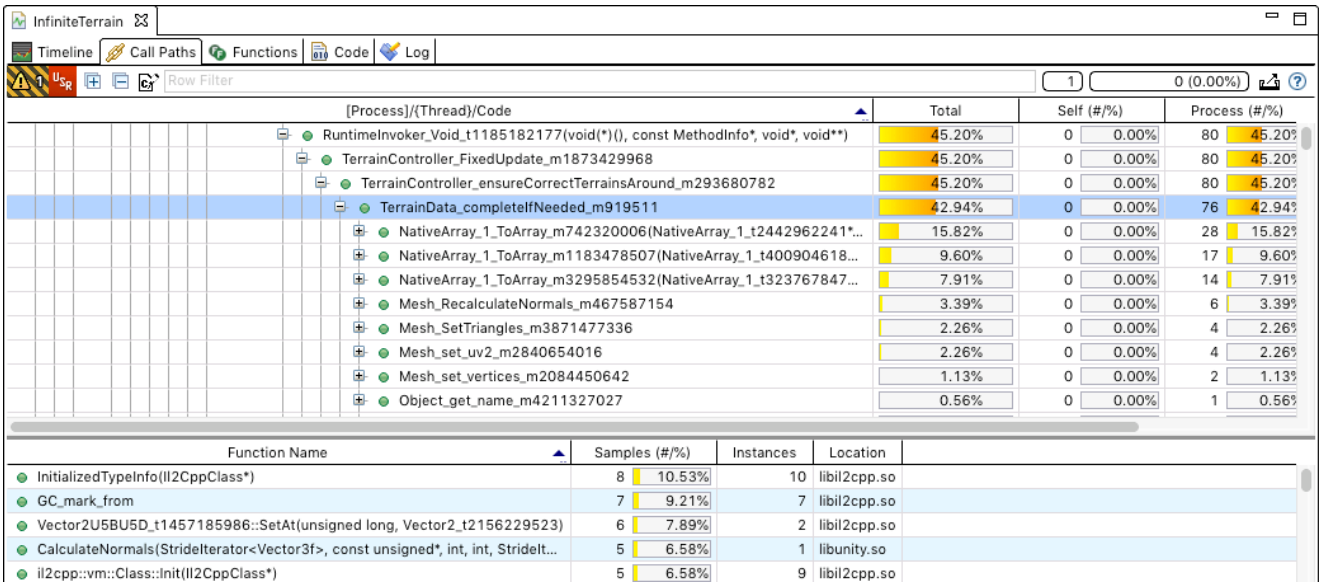


그림 3-10 Unity용 IL2CPP 스크립팅 백엔드

### 3.2.4 워커 스레드

게임 자체 주석이 더 수준 높은 콘텍스트를 제공하는 경우 스트림라인의 더 많은 기능이 잠금 해제될 수 있습니다.

필터링 후 최대 8개의 병렬 작업 지정에 따라 8개의 워커 스레드 스레드가 표시됩니다. 이 스크린샷에서 Cortex-A53 클러스터는 개별 코어 활용을 보기 위해 확대되었습니다.

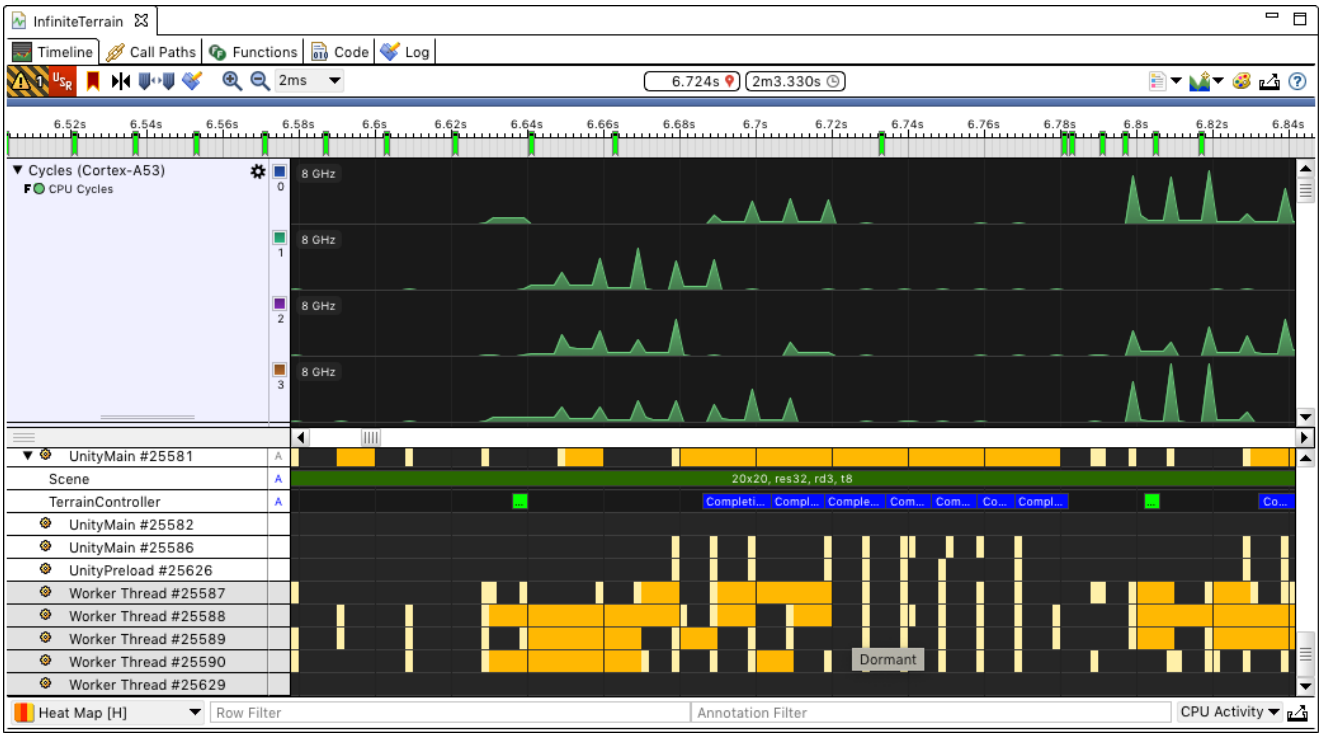


그림 3-11 확대된 Cortex-A53 클러스터

TerrainController 채널의 초록색 블록은 새 지형이 생성 예약되었을 때 표시됩니다. 이어서 모든 코어에 활발한 활동이 발생합니다. TerrainController의 파란색 활동은 생성되었을 때 메인 스레드에서 지형을 처리합니다. UnityMain 스레드가 선택되지 않았으므로 이 그래프에서는 메인 스레드 활동이 표시되지 않습니다.

두 번째 장면과 비교하면 지형 타일의 복잡성은 동일하지만 한 번에 하나만 예약이 가능합니다.

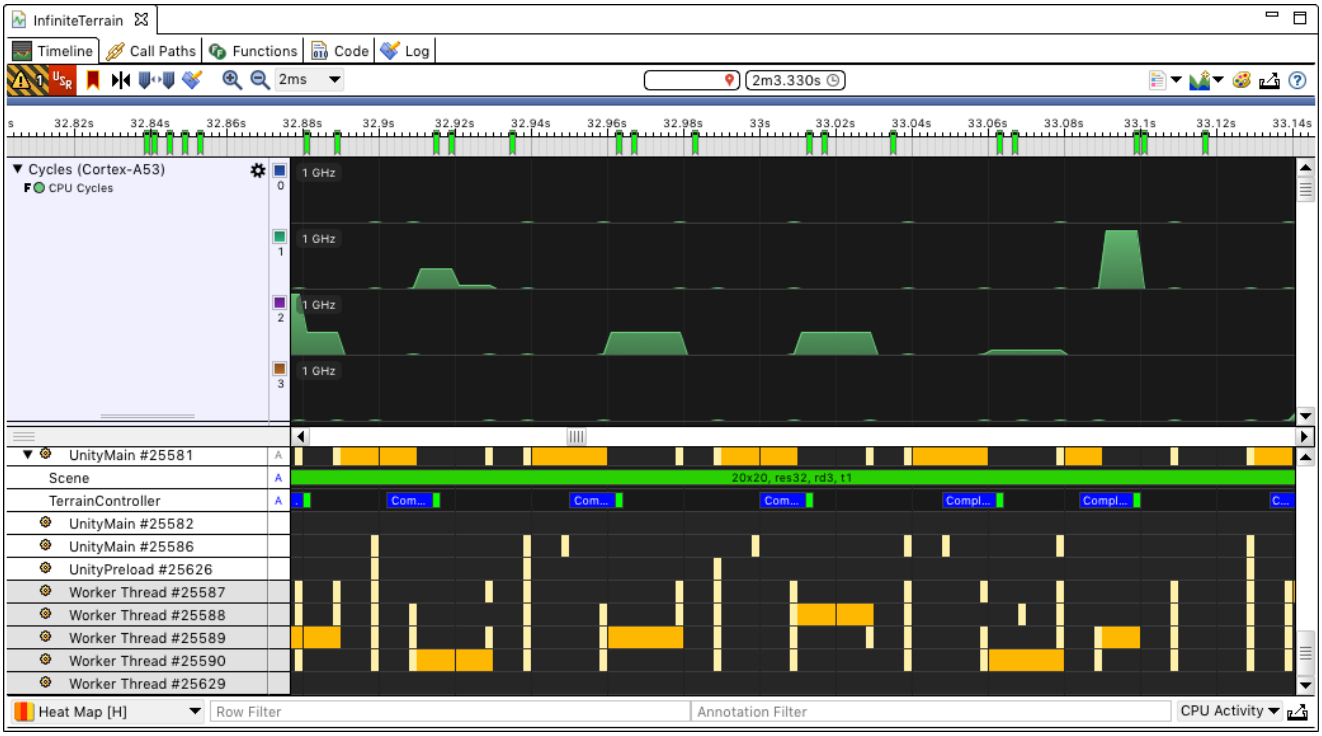


그림 3-12 두 번째 장면과 비교

여기서 중요한 점은 다음과 같습니다.

1. 대부분의 시간 동안 대부분의 코어가 유휴 또는 대부분 유휴에 가깝기 때문에 CPU 활동은 덜 활발합니다.
2. 프레임 속도는 완벽하지 않아도 더 부드럽습니다. 한 번에 하나의 프레임만 완료되므로 메인 스레드의 대량 활동이 억제되었습니다.

또한 더 작은 타일을 사용하는 세 번째 장면의 프로파일과 비교하면 다음과 같습니다.

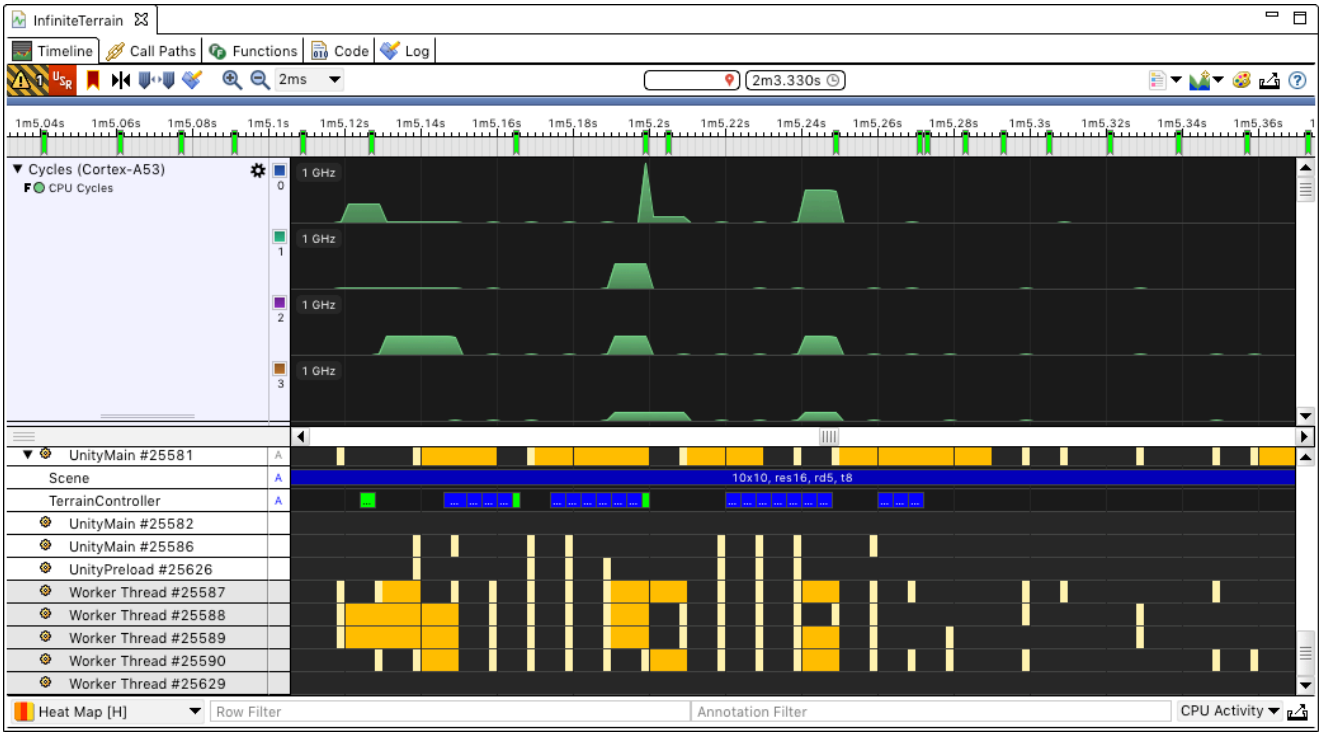


그림 3-13 세 번째 장면

CPU 활동은 덜 활발하며 메인 스레드에서 완료된 작업을 표시하는 파란색 블록의 길이가 짧습니다. 첫 번째 장면보다 프레임 속도는 매끄럽지만 전체 작업은 더 많기 때문에 카메라가 지형을 따라 이동할 때 속도를 유지하며 지형이 생성되는지 확인합니다.

네 번째 장면은 타일이 작고 한 번에 지형 하나만 생성합니다. 따라서 전체 프레임 속도는 매끄럽지만 카메라 속도를 따라가기 충분한 속도로 지형이 생성되는지 확인해야 합니다. 네 번째 장면에서 카메라가 빠르게 통과하는 일은 흔한 경우가 아닙니다.

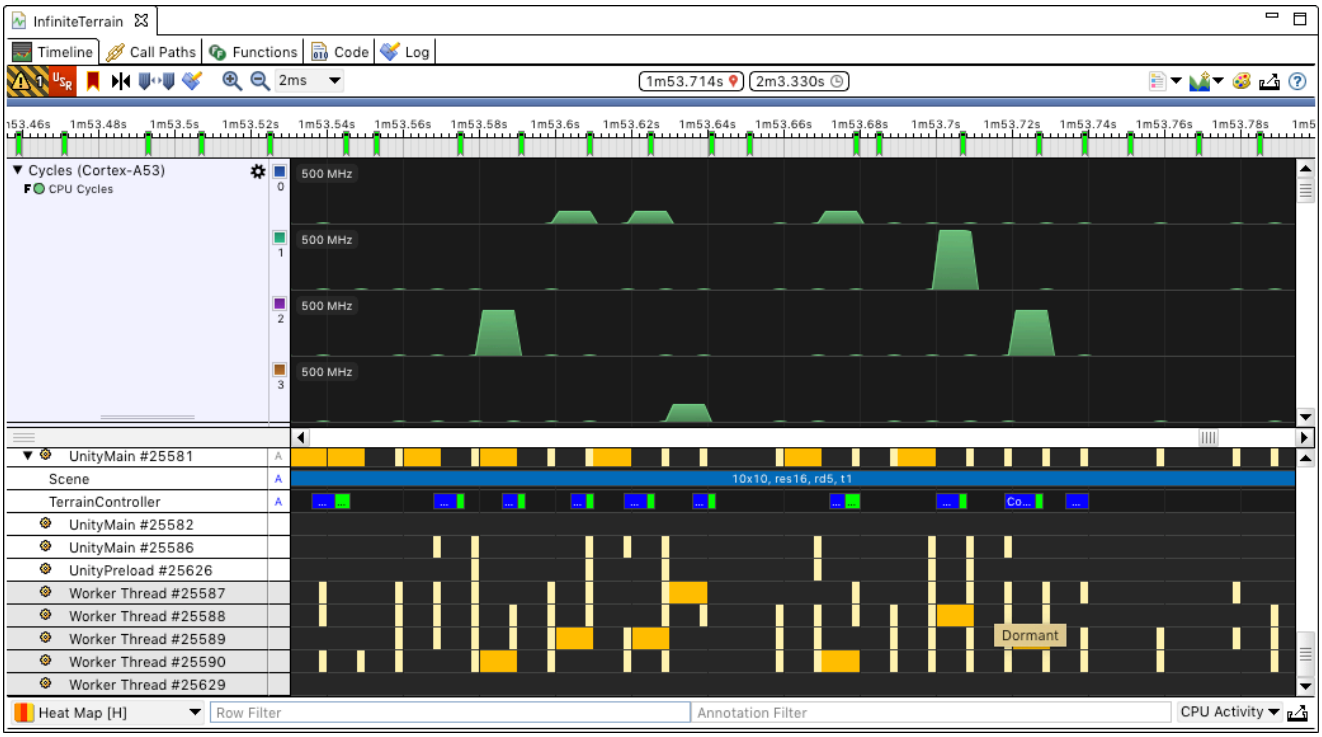


그림 3-14 네 번째 장면

지형 생성에 워커 스레드가 어떻게 작동하는지 더 자세한 내용을 알고 싶은 경우, 맞춤 활동 맵을 사용합니다. 각 맞춤 활동 맵은 하단 왼쪽 메뉴의 옵션으로 표시됩니다.

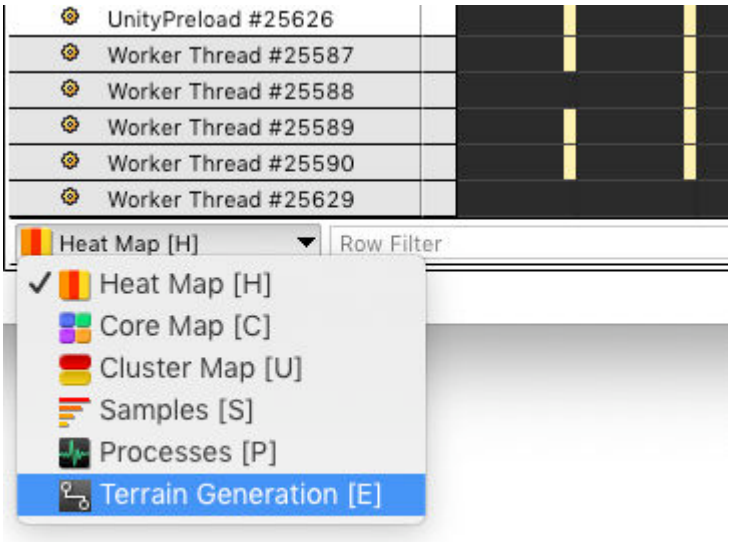


그림 3-15 맞춤 활동 맵

지형 생성 보기가 선택되면 각 지형 생성 작업의 색상이 칠해진 상자가 시작과 중지를 표시합니다. 마우스를 올려놓으면 지형 타일의 세계 좌표와 시작된 시점, 완료 시간을 표시합니다. 스크린샷은 Mali GPU에서 발생한 컴퓨팅 작업과 지형이 생성되며 GPU 활동이 꾸준히 증가하는 것을 보여주는 그래프입니다. 이 스크린샷은 최대 8개 타일이 동시에 생성되는 첫 번째 장면이 시작될 때를 중심으로 촬영된 것입니다. 메인 스레드가 새 지오메트리를 준비하는 동안 GPU가 장시간 유휴로 있어야 하기 때문에 중지가 발생합니다.

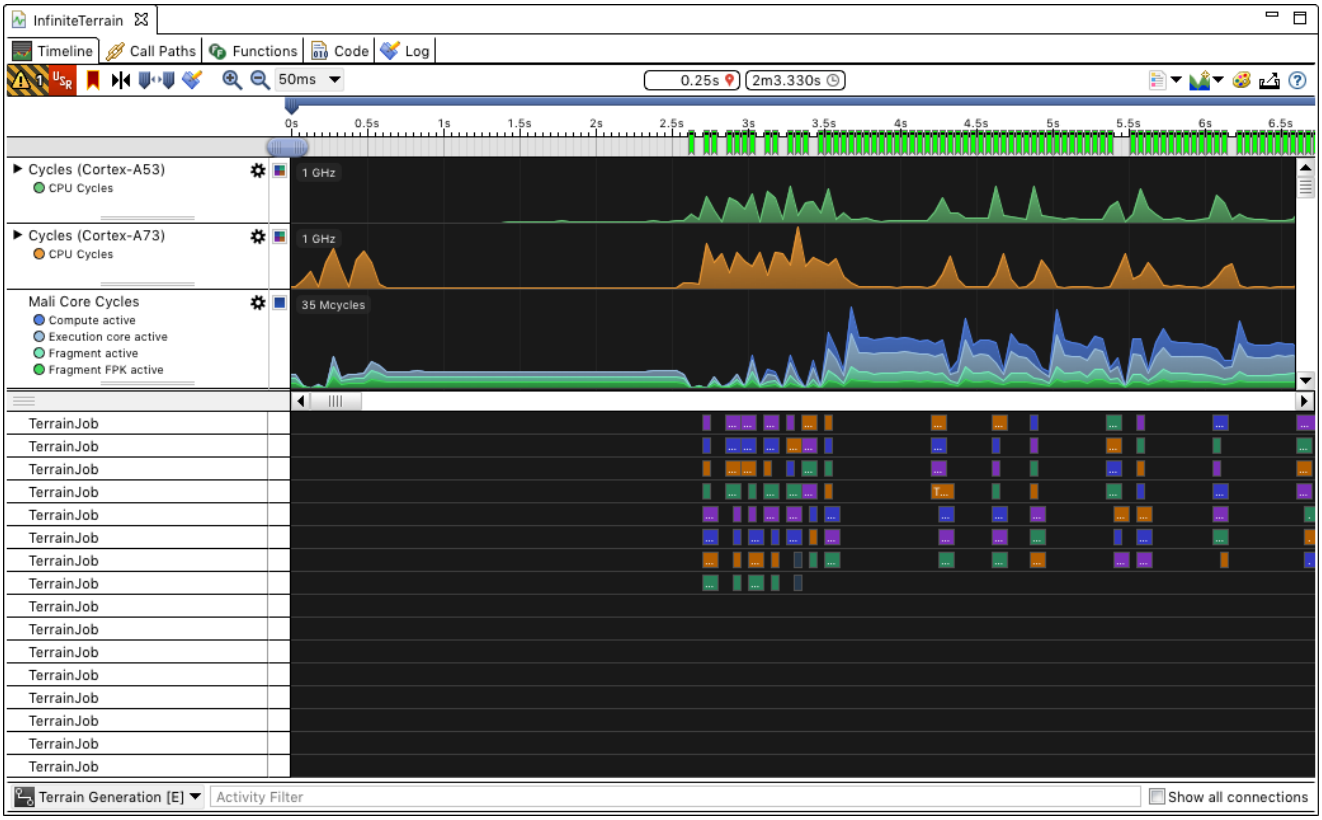


그림 3-16 그래픽 컴퓨팅 작업

네 번째 장면에서 작은 타일이 차례로 생기면서 GPU 활동이 천천히 상승합니다. 한 번에 하나의 지형 작업만 발생하며 타일 크기가 작아 각 작업 시간도 짧습니다.

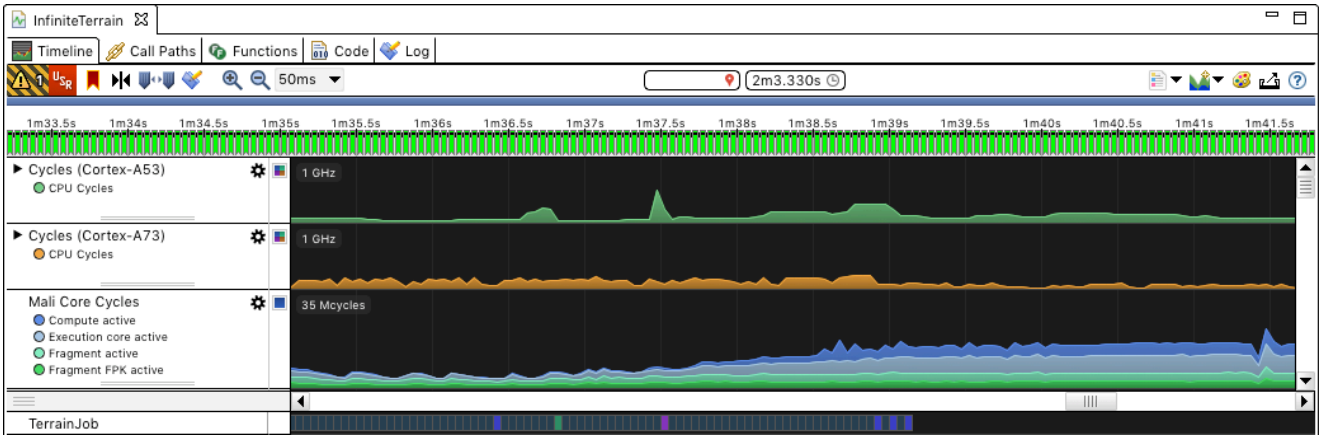


그림 3-17 네 번째 장면 CPU 주기

### 3.2.5 스트림라인 주석

스트림라인 주석이 무엇이며 이것이 어떻게 작동하고 Unity에서 어떻게 사용되는지 이해하는 것이 중요합니다.

#### 스트림라인 주석 설명

Android 응용 프로그램을 분석할 때 애플리케이션과 동일한 사용자가 실행하는 게이터라는 별도의 프로세스가 장치를 실행합니다. 게이터는 Mali GPU나 ARM Cortex -A CPU 등 다양한 하드웨어 소스에서 프로파일링 정보를 수집하여 메트릭스의 종합적인 스트림을 컴퓨터

에 다시 전송합니다. 응용 프로그램 역시 스트림라인 주석이라는 메커니즘을 사용해 고유의 제작자와 메트릭스를 스트림에 삽입합니다.

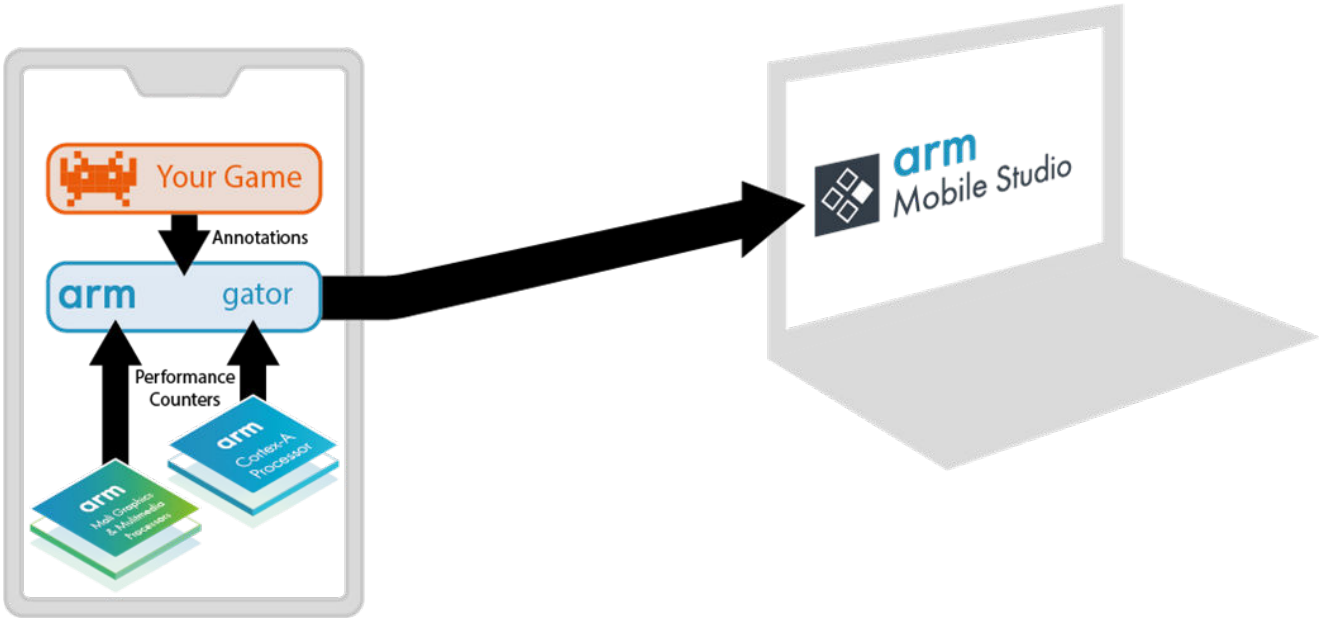


그림 3-18 스트림라인 작동 방식

Unity에서 스트림라인 주석 사용 방법

스트림라인 주석은 특정 프로토콜을 사용하며 오픈 소스 C 구현은 ARM 모바일 스튜디오의 일부로 제공됩니다. Unity 콘텐츠에서 스트림라인 주석을 쉽게 생성하려면 C 구현 주변에 일부 C# 래퍼가 필요합니다. 이 설명에서 사용된 래퍼와 필요한 C 구현은 *https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/ArmMobileStudio.unitypackage?raw=true* 에서 Unity 애셋 패키지로 이용할 수 있습니다.

다운로드 후 Unity에서 제공된 설명에 따라 프로젝트에 사용자 애셋 패키지로 가져옵니다. 관련 설명은 *https://docs.unity3d.com/Manual/AssetPackages.html*을 참조하십시오.

패키지는 `ARM` 명칭의 신규 메서드로 추가되어 프로젝트에서 쉽게 스트림라인 주석을 사용할 수 있습니다. API 설명서는 이 패키지*https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/InfiniteTerrain/Assets/Arm%20Mobile%20Studio/README.md* 내의 `README.md` 파일을 참조하십시오.

### 3.2.6 효율적인 Unity 프로젝트 설정

스트림라인을 사용하면 프로젝트를 구성하고, 마커와 채널을 추가하고, 맞춤 활동 맵을 사용하고, 프로파일을 수집할 수 있습니다.

특정 Android 플레이어 설정을 구성하면 빠르고 간편하게 Android 빌드를 분석할 수 있습니다.

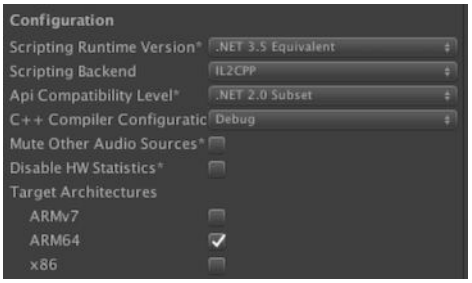


그림 3-19 구성 설정

IL2CPP가 스크립팅 백엔드로 사용되며 C++ 컴파일러 구성이 디버그로 설정되었는지 확인합니다. 이렇게 하면 스크립트를 네이티브 코드로 컴파일해 성능을 개선할 수 있고 스트림라인이 디버그 정보를 확인해 호출 경로 보기의 함수에 성능 데이터를 다시 매핑할 수 있습니다.

타겟 아키텍쳐를 ARM64(ARMv7이 기본)로 설정합니다. 현재 대부분의 모바일 장치는 64비트이며 고품질 코드 생성을 할 수 있습니다.

### 마커 추가

마커는 가장 사용이 편리한 주석입니다. 문자열과 색상을 사용합니다. 예를 들어 프레임마다 녹색 마커를 표시하려면 GameObject 중 하나에 다음 코드를 사용해야 합니다(`Update()` 메서드는 자동으로 프레임별로 호출됩니다).

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

### 채널 추가

채널을 사용하는 것도 쉽습니다. 먼저 채널을 생성하고 이름을 지정합니다. 채널 오브젝트의 메서드를 사용해 채널에 로그 주석을 다음과 같이 입력합니다.

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

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

채널의 주석은 기간 범위가 있습니다. 다음 주석을 시작하기 전 주석을 종료하고 싶으면 `end()` 메서드를 사용합니다. 예를 들어 메인 스레드에서 지형을 완성하는 TerrainController는 다음과 같이 래핑됩니다.

```
// 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();
```

### 맞춤 활동 맵 사용

*맞춤 활동 맵*(CAM)은 채널 맨 위에 추가되는 레이어입니다. 트랙을 만들기 전 CAM에 이름을 지정합니다. 트랙에 주석을 추가하는 것은 채널에 주석을 추가하는 것과 유사합니다.

예에서 지형 생성 CAM은 다음과 같이 생성되었습니다.

```
terrainCAM = new Arm.Annotations.CustomActivityMap("지형 생성");

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

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

그러나 이 경우 하나만 완료됩니다. Unity 작업 시스템에 작업이 실행 중인 경우 게임 내 다른 오브젝트 모델과 상호 교류하지 않으므로 스레드 안전을 유지할 수 있습니다. 따라서 메인 스레드에서 작업을 정리하면 작업의 시작과 종료 시간을 저장하고 CAM에 작업 활동을 등록합니다.

C# 래퍼는 작업 내에서 안전하게 호출되고 스트림라인 주석에 필요한 형식으로 현재 시간을 반환하는 함수를 제공합니다.

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

메인 스레드로 돌아오면 사용할 트랙을 선택하고(풀에서 겹침이 없도록 관리하면 시각적으로 도움이 됩니다) 트랙에 작업을 등록합니다. `job.timings`는 작업 시작 시간과 종료 시간을 포함하는 2개의 엔트리 배열로 채워져 있습니다.

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

### 스트림라인에서 기본 프로필 수집

스트림라인에서 프로필을 수집하는 단계는 다음과 같습니다.

먼저 *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/downloads?_ga=2.211441174.2077719384.1568032571-433518899.1564491901*에서 Windows, Mac 또는 Linux 버전의 무료 ARM Mobile Studio Starter Edition을 다운로드하여 설치합니다.

중요 사항:

- 모바일 장치에서 응용 프로그램에 액세스를 제공하는 게이터가 실행되어야 합니다.
- 데이터를 장치에서 가져와 툴에 보내는 방법이 확보되어야 합니다.

여러 장치에서 활용할 수 있는 간단한 방법을 사용하려면 알려진 다음 방법을 사용합니다.

- 응용 프로그램의 크기(32비트 또는 64비트). 이전 안내를 따라 `ARM64` 옵션으로 Unity 게임을 빌드했다면 64비트입니다.
- 응용 프로그램의 패키지 이름(Unity의 Android 플레이어 설정에서 지정한 이름). ARM 응용 프로그램의 경우 `com.Arm.InfiniteTerrain`입니다.
- 게이터 바이너리 받기. 설치된 ARM Mobile Studio의 `streamline/bin/arm`(32비트) 또는 `streamline/bin/arm64`(64비트) 폴더에서 확인합니다.

이 정보를 확인한 뒤 분석하는 단계는 다음과 같습니다.

- 응용 프로그램이 장치에 설치되었는지 확인합니다.
- Android `adb` 도구를 사용해 모바일 장치의 네트워크 포트를 ARM Mobile Studio에서 실행 중인 시스템의 로컬 포트로 포워딩합니다.
- `adb`를 사용해 게이터(응용 프로그램에 따라 32비트 또는 64비트)를 디바이스로 푸시하고 응용 프로그램과 동일한 패키지 이름으로 실행하기 시작합니다.
- 스트림라인을 시작하고 이를 로컬 포트에 연결해 `adb`로 트래픽이 포워딩되도록 합니다. 성능 카운터를 여기에 수집을 선택합니다.
- 응용 프로그램이 모바일 장치에서 실행되면 스트림라인은 분석 데이터가 들어오는 대로 수집을 시작합니다.

게이터가 실행되면 새로운 응용 프로그램 버전을 설치하고 스트림라인을 시작, 정지하며 게이터를 재시작하지 않고 분석을 계속합니다.

이 프로세스를 더 편리하게 하려면 *https://github.com/ARM-software/Tool-Solutions/tree/master/mobile-application-profiling/mobile-studio-scripts*에서 `gatorme` 스크립트를 다운로드합니다. 이 스크립트를 사용해 게이터와 `adb`를 구성하고 실행합니

다. 실행하려는 게이터 바이너리 경로와 응용 프로그램의 패키지 이름, 디바이스에서 사용되는 Mali GPU가 필요합니다. 장치 조사에서 게이터에 사용 GPU가 파악되지 않는 경우 이 정보를 제공하면 좋습니다. 또한 이 메서드가 가장 다양한 범위의 모바일 장치에서 사용되도록 하려면 여러 단계가 필요하며 프로파일링이 끝나면 게이터를 적절하게 종료시켜야 합니다.

gatorme 문서에서 자세한 내용을 설명하지만 이것은 장치에 APK가 설치된 경우 InfiniteTerrain 콘텐츠를 프로파일링하는 방법을 보여주는 작업 예시입니다.

먼저 명령 행에서 gatorme를 실행합니다.

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

이제 스트림라인을 실행하면 정보 수집 준비가 끝납니다. 이 설정에 따라 스트림라인을 시작합니다.

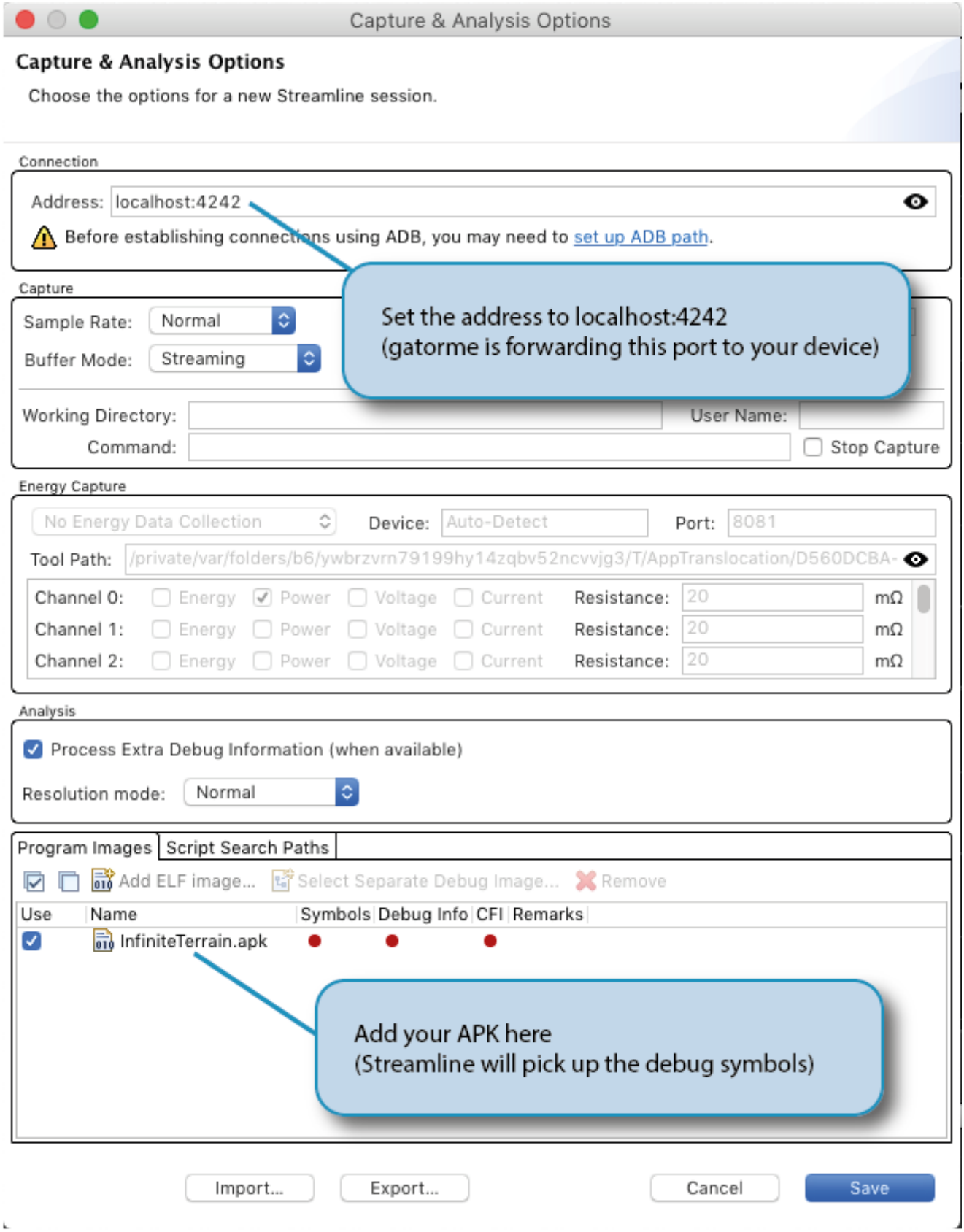


그림 3-20 스트림라인 시작 설정

설정이 끝나면 gatorme 실행 상태에서 응용 프로그램을 종료하고, Unity에서 직접 빌드하고 실행한 뒤 스트림라인 정보를 더 수집하면 되므로 배포/분석/수정 단계 반복이 간편합니다.

GitHub에서 Apache 2.0 라이선스로 InfiniteTerrain 예시의 소스 코드를 다운로드할 수 있습니다. 소스 코드를 다운로드하려면 *https://github.com/ARM-software/Tool-Solutions/tree/master/mobile-application-profiling/mobile-studio-with-unity*로 이동하십시오.

소스 코드와 그래픽 애셋 외에도 *https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/ArmMobileStudio.unitypackage?raw=true*에서 다운로드할 수 있는 Unity 맞춤 애셋 패키지 ArmMobileStudio.unitypackage가 있습니다.

Unity는 이 패키지를 프로젝트에 가져오고 패키지에 스트림라인 주석을 추가하는 방법 안내를 제공합니다. 관련 설명은 *https://docs.unity3d.com/Manual/AssetPackages.html*을 참조하십시오.

사전 빌드 버전인 InfiniteTerrain.apk를 다운로드받으려면 *https://github.com/ARM-software/Tool-Solutions/blob/master/mobile-application-profiling/mobile-studio-with-unity/InfiniteTerrain.apk?raw=true*로 이동하십시오. 이것은 64비트 Android 개발 빌드로 장치에 배포해 콘텐츠를 빠르게 분석할 수 있습니다.

# 제 4 장
# 최적화 목록

이 장에서는 Unity 응용 프로그램에 대한 여러 최적화를 소개합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 4.1 응용 프로그램 프로세서 최적화

다음 목록은 응용 프로그램 프로세서 최적화에 대한 설명입니다.

Invoke() 대신 코루틴을 사용
Monobehaviour.Invoke() 메서드는 시간 지연을 갖는 클래스에서 메서드를 호출하는 빠르고 편리한 방법이지만 다음과 같은 제약이 있습니다.

- 이 메서드는 C#의 리플렉션을 사용하여 호출할 메서드를 찾기 때문에 메서드를 직접 호출하는 경우보다 느립니다.
- 메서드 서명에 대한 컴파일 시간 확인이 없습니다.
- 추가 매개 변수를 제공할 수 없습니다.

다음 코드는 Invoke() 함수입니다.

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

한 가지 대안은 코루틴을 사용하는 것입니다. 코루틴은 Unity에 특수 yield return 문을 포함하는 제어를 반환하는 IEnumerator 형식의 함수입니다. 이 함수는 나중에 다시 호출하여 이전에 중단한 위치에서 다시 시작할 수 있습니다.
코루틴은 MonoBehaviour.StartCoroutine() 메서드를 사용하여 호출할 수 있습니다.

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

Monobehaviour.Invoke() 메서드에서 코루틴 사용으로 전환하면 애니메이션 상태를 처리하는 함수로 전달되는 매개 변수에 대한 유연성이 증가합니다.

낮은 레벨 업데이트에 코루틴 사용
게임이 모든 특정 시간 간격에서 동작을 필요로 하는 경우 MonoBehaviour.Update() 콜백에서 프레임미다 동작을 실행하는 대신 MonoBehaviour.Start() 콜백에서 코루틴을 실행해 보십시오. 예를 들면 다음과 같습니다.

```
void Update()
{
    // Perform an action every frame
}

IEnumerator Start()
{
    while(true)
    {
        // Do something every quarter of second
        yield return new WaitForSeconds(0.25f);
    }
}
```

———— 참고 ————

이 기법의 다른 용도는 불규칙한 간격으로 적을 만들어내는 것입니다. 적을 만들어내고 난수를 생성하는 코루틴 내에서 무한 루프를 사용합니다. 난수를 WaitForSeconds() 함수로 전달합니다.

————————————

태그에 하드 코딩된 문자열을 사용하지 않음

태그에 하드 코딩된 값을 사용하지 마십시오. 이러한 값은 게임의 확장성과 견고성을 제한하기 때문입니다. 예를 들어 태그 이름을 사용하는 경우 이름을 문자열로 직접 참조하면 이름을 용이하게 수정할 수 없고 스펠링 오류의 위험이 있습니다. 예를 들면 다음과 같습니다.

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

태그에 대해 공개 상수 문자열을 표시하는 특수 클래스를 구현하여 이를 개선할 수 있습니다. 예를 들면 다음과 같습니다.

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

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

———— 참고 ————

공개 상수 문자열을 포함하는 태그 클래스를 사용하여 일관적이고 확장 가능한 방식으로 새 태그를 추가하는 데 도움을 줄 수 있습니다.

————————

고정 타임스텝을 변경하여 물리 계산 횟수를 축소

고정 타임스텝을 변경하여 물리 계산의 컴퓨팅 부하를 줄일 수 있습니다. 일반적으로 대부분의 물리 계산은 고정 타임스텝으로 이루어지며, 이 스텝의 길이를 늘리거나 줄일 수 있습니다.

타임스텝을 늘리면 응용 프로그램 프로세서에 대한 부하가 줄어들지만 물리 계산의 정확도는 감소합니다.

다음과 같이 메인 메뉴에서 시간 관리자에 액세스할 수 있습니다. Edit > Project Settings > Time.

다음 그림은 시간 관리자입니다.

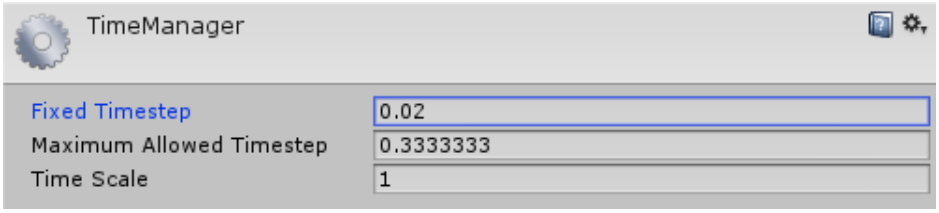


그림 4-1 고정 타임스텝 설정

빈 콜백을 제거

코드에 Awake(), Start() 또는 Update()와 같은 함수에 대한 빈 정의가 포함된 경우 이들을 제거하십시오. 이러한 정의가 비어 있더라도 엔진은 계속 액세스를 시도하기 때문에 이들과 연결되는 오버헤드가 존재합니다. 예를 들면 다음과 같습니다.

```
// Remove the following empty definition

void Awake()
{

}
```

모든 프레임에서 GameObject.Find()를 사용하지 마십시오.

GameObject.Find()는 장면의 모든 객체에 걸쳐 반복되는 함수입니다. 이 함수가 코드의 잘못된 부분에서 사용될 경우 메인 스레드 크기가 현저히 증가하게 될 수 있습니다. 예를 들면 다음과 같습니다.

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

보다 나은 기법은 시작 시 GameObject.Find()를 호출하고 예를 들어 Start() 또는 Awake() 함수에 결과를 캐싱하는 것입니다.

```
private GameObject _playerGO = null ;

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

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

또 하나의 대안은 GameObject.FindWithTag()를 사용하는 것입니다.

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

——— 참고 ———

장면이 로드되는 즉시 모든 객체를 검색하는 LocatorManager라는 전용 클래스를 사용하십시오. 객체가 여러 번 검색되지 않도록 다른 클래스가 이 클래스를 서비스로 사용할 수 있습니다.

———————————

### StringBuilder 클래스를 사용하여 문자열을 연결

복잡한 문자열을 연결할 때 System.Text.StringBuilder 클래스를 사용하십시오. 이 클래스는 string.Format() 메서드보다 속도가 빠르며, 더하기 연산자를 사용하는 연결보다 메모리를 적게 사용합니다.

```
// Concatenation with the plus operator
string str = "foo" + "bar";

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

System.Text.StringBuilder 클래스:

```
// StringBuilder class
using System.Text;

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

다음 그림은 문자열 연결을 보여줍니다.

| 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 |

그림 4-2 문자열 연결

### 태그 속성 대신 CompareTag() 메서드를 사용

GameObject.tag 속성 대신 GameObject.CompareTag() 메서드를 사용하십시오. CompareTag() 메서드는 속도가 더 빠르고 추가 메모리를 할당하지 않습니다.

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

// Gameobject.tag property
if(mainCamera.tag == "MainCamera")
{
    // Perform an action
}

// Gameobject.CompareTag() method
if(mainCamera.CompareTag("MainCamera"))
{
    // Perform an action
}
```

다음 그림은 CompareTag()의 용례입니다.

| 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 |

그림 4-3 비교 태그

객체 풀을 사용

게임에 런타임 시 생성 및 파괴되는 동일한 유형의 객체가 많이 있을 경우 디자인 패턴 *object pool*을 사용할 수 있습니다. 이 디자인 패턴은 많은 객체를 동적으로 할당 및 해제하는 성능 페널티를 회피합니다.
필요한 객체의 총 수를 알고 있는 경우 즉각적으로 모든 객체를 생성하고 즉시 필요하지 않은 객체를 비활성화할 수 있습니다. 새로운 객체가 필요한 경우 풀에서 사용되지 않은 첫 번째 객체를 검색하여 활성화할 수 있습니다.
객체가 더 이상 필요하지 않으면 풀에 반환할 수 있습니다. 즉, 객체를 기본 시작 상태로 재설정하고 비활성화할 수 있습니다.
이 기법은 적, 발사체, 파티클과 같은 객체에 사용할 수 있습니다. 필요한 객체 수를 정확히 알지 못하는 경우 테스트를 통해 얼마나 많은 객체가 사용되는지 확인하여 이보다 약간 큰 풀을 생성합니다.

———— 참고 ————

적과 폭탄에 대해 객체 풀을 사용하십시오. 이렇게 하면 게임 로드 단계에서 이들 객체의 할당이 제한됩니다.

————————

구성요소 검색 결과를 캐시

`GameObject.GetComponent<Type>()`이 반환하는 구성요소 인스턴스를 캐시하십시오. 이와 관련된 함수 호출은 상당히 연산 비용이 높습니다.
`GameObject.camera`, `GameObject.renderer` 또는 `GameObject.transform`과 같은 속성은 대응되는 `GameObject.GetComponent<Camera>()`, `GameObject.GetComponent<Renderer>()` 및 `GameObject.GetComponent<Transform>()`에 대한 바로 가기입니다.

```
private Transform _transform = null;

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

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

`Transform.position`의 반환 값을 캐시하는 것을 고려해 보십시오. 이것은 C# getter 속성이지만 전역 위치를 계산하기 위해 변환 계층에 대한 이터레이션과 연결되는 오버헤드가 존재합니다.

———— 참고 ————

Unity 5 이상에서는 변환 구성요소가 자동으로 캐시됩니다.

————————

OnBecameVisible() 및 OnBecameInvisible() 콜백 사용

`MonoBehaviour.OnBecameVisible()` 및 `MonoBehaviour.OnBecameInvisible()`과 같은 콜백은 연결된 게임 객체가 화면에 표시되는지 여부를 스크립트에 통보합니다.
이러한 호출을 사용하여 예를 들어 게임 객체가 화면에서 렌더링되지 않을 때 많은 연산 집약적 코드 루틴 또는 효과를 비활성화할 수 있습니다.

벡터 크기를 비교할 때 sqrMagnitude를 사용

응용 프로그램에서 벡터 크기 비교가 필요할 경우 Vector3.Distance() 또는 Vector3.magnitude 대신 Vector3.sqrMagnitude를 사용하십시오.

Vector3.sqrMagnitude는 제곱된 구성요소를 제곱근은 계산하지 않고 합산하지만 이것이 비교에는 유용합니다. 다른 호출은 연산 비용이 높은 제곱근 계산을 사용합니다.

다음 코드는 공간에서 두 위치를 비교하는 데 사용되는 3가지 기법을 보여줍니다.

```
// Vector3.sqrMagnitude property
if ((_transform.position - targetPos).sqrMagnitude < maxDistance * maxDistance)
{
    // Perform an action
}

// Vector3.Distance() method
if (Vector3.Distance(transform.position, targetPos) < maxDistance)
{
    // Perform an action
}

// Vector3.magnitude property
if ((_transform.position - targetPos).magnitude < maxDistance)
{
    // Perform an action
}
```

내장 배열을 사용

배열의 크기를 미리 알고 있다면 내장 배열을 사용하십시오.

ArrayList 및 List 클래스는 더 많은 요소를 삽입할수록 크기가 증가하므로 유연성은 더 뛰어나지만 내장 배열보다 속도가 느립니다.

충돌 타겟으로 평면을 사용

장면이 바닥 또는 벽과 같은 평면 객체와의 파티클 충돌만을 요구하는 경우 파티클 시스템 충돌 모드를 Planes로 변경하십시오. 평면을 사용하도록 설정을 변경하면 필요한 계산이 줄어듭니다. 이 모드에서 콜라이더 평면으로 동작할 빈 GameObjects의 목록을 Unity에 제공할 수 있습니다.

다음 그림은 충돌 설정입니다.

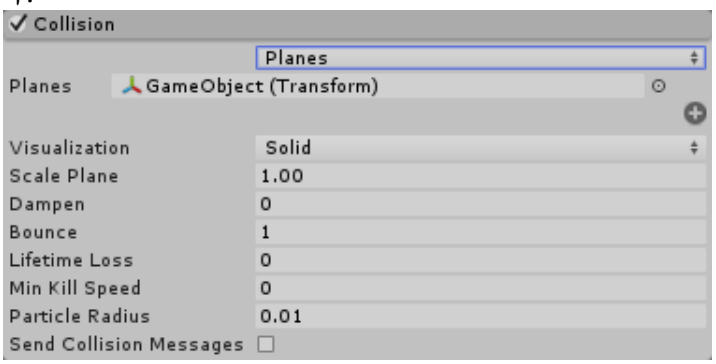


그림 4-4 충돌 설정

메시 콜라이더 대신 복합 기본 콜라이더를 사용

메시 콜라이더는 객체의 실제 지오메트리를 기반으로 합니다. 이 콜라이더는 매우 정확하게 충돌을 탐지하지만 연산 비용이 높습니다.

상자, 캡슐 또는 구체와 같은 형태를 원래 메시의 형태를 모방한 복합 콜라이더로 결합할 수 있습니다. 그러면 훨씬 적은 컴퓨팅 오버헤드로도 유사한 결과를 얻을 수 있습니다.

## 4.2 GPU 최적화

이 단원에서는 GPU 최적화를 소개합니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 4.2.1 기타 GPU 최적화

다음 목록은 기타 GPU 최적화에 대한 설명입니다.

정적 배칭을 사용
: 정적 배칭은 드로우 콜을 줄이고 결과적으로 응용 프로그램 프로세서 사용을 낮추는 일반적인 최적화 기법입니다.
동적 배칭은 Unity에 의해 투명하게 실행되지만 많은 수의 정점으로 구성된 객체에는 적용할 수 없습니다. 컴퓨팅 오버헤드가 너무 커지기 때문입니다.
정적 배칭은 많은 수의 정점으로 구성된 객체에 사용할 수 있지만, 배칭된 객체가 렌더링 도중 이동, 회전 또는 확대/축소될 수 없습니다.
Unity가 정적 배칭을 위해 객체를 그룹화할 수 있도록 검사기에서 해당 객체를 정적으로 표시합니다.
다음 그림은 정적 배칭 설정입니다.

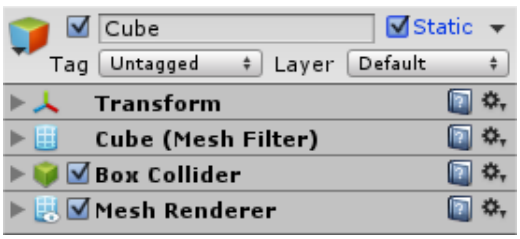


그림 4-5 정적 배칭 설정

4x MSAA를 사용
: ARM Mali GPU는 매우 적은 컴퓨팅 오버헤드로 4x 멀티 샘플 안티앨리어싱을 실행할 수 있습니다.
Unity 품질 설정에서 4x MSAA를 활성화할 수 있습니다.
다음 그림은 MSAA 설정입니다.

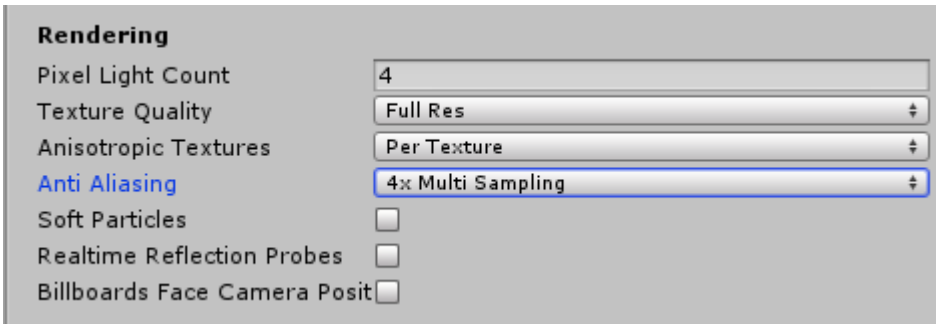


그림 4-6 MSAA 설정

디테일 레벨을 사용

*디테일 레벨*(LOD)은 Unity 엔진이 카메라와의 거리에 따라 동일한 객체에 대해 다른 메시를 렌더링하는 기법입니다.

객체가 카메라에 근접할수록 지오메트리가 높은 디테일 레벨로 렌더링됩니다. 객체가 카메라에서 멀어지면 디테일 레벨이 감소하고 가장 원거리에서는 평면 빌보드를 사용할 수 있습니다.

사용할 메시 및 연결된 거리 범위를 관리하려면 LOD 그룹을 올바로 설정해야 합니다. LOD 그룹 설정에 액세스하려면 Add Component > Rendering > LOD Group을 선택합니다.

Unity 5에서는 연속 LOD로 블렌딩하기 위해 각 LOD 레벨에 대해 `Fade Mode`를 설정합니다. 그러면 그룹 간 전환이 매끄러워집니다. Unity는 객체의 화면 크기에 따라 블렌딩 인수를 계산하여 블렌딩을 수행하는 쉐이더로 전달합니다. 쉐이더에서 지오메트리 블렌딩을 구현해야 합니다.

다음 그림은 LOD 그룹 설정입니다.

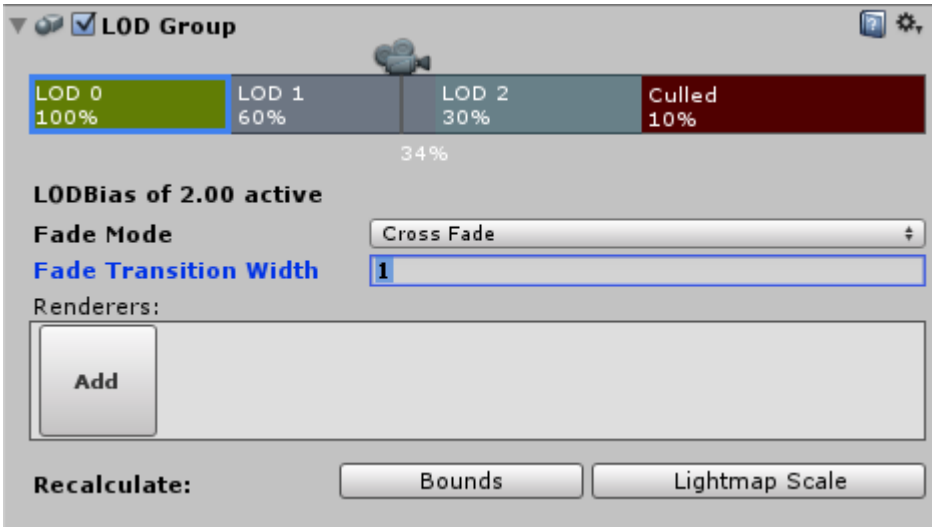


그림 4-7 LOD 그룹 설정

사용자 지정 쉐이더에서 수학 함수를 사용하지 않음

사용자 지정 쉐이더를 작성할 때 다음과 같이 연산 비용이 높은 내장 수학 함수는 사용하지 않도록 노력하십시오.

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

### 4.2.2 라이트맵 및 라이트 프로브

런타임 조명 계산은 연산 비용이 높습니다. 연산 요건을 줄이기 위한, 라이트매핑이라고 하는 인기 있는 기법은 조명 계산을 사전 연산하여 라이트맵이라고 하는 텍스처로 베이크합니다.

이는 완전히 동적으로 조명되는 환경이라는 유연성은 상실하지만 성능에 영향을 미치지 않으면서 매우 고품질의 이미지를 얻을 수 있음을 의미합니다.

정적 라이트맵에서 결과 조명을 베이크하는 방법

- 조명을 받는 지오메트리를 static으로 설정합니다.
- 라이트에서 Baking 옵션을 Realtime 대신 Baked로 설정합니다.
- Lightmapping 창의 Scene 탭에서 Baked GI 옵션을 선택합니다.

결과 라이트맵을 확인하는 방법
- 지오메트리를 선택합니다.
- Window > Lighting을 선택하여 Lighting 창을 엽니다.
- Object 버튼을 누릅니다.
- 미리 보기 옵션에서 Baked Intensity lightmap을 선택합니다.

Continuous Baking 옵션을 선택한 경우 Unity가 라이트맵을 베이크하고 몇 초 후 편집기에서 장면을 업데이트합니다.

라이트맵이 올바로 설정되었는지 빠르게 확인할 수 있는 방법 한 가지는 편집기에서 게임을 실행하고 라이트를 비활성화하는 것입니다. 조명이 계속 유지된다면 라이트맵이 올바로 생성되어 사용되고 있는 것입니다.

다음 그림은 Lighting 탭의 강도 라이트맵을 보여줍니다.

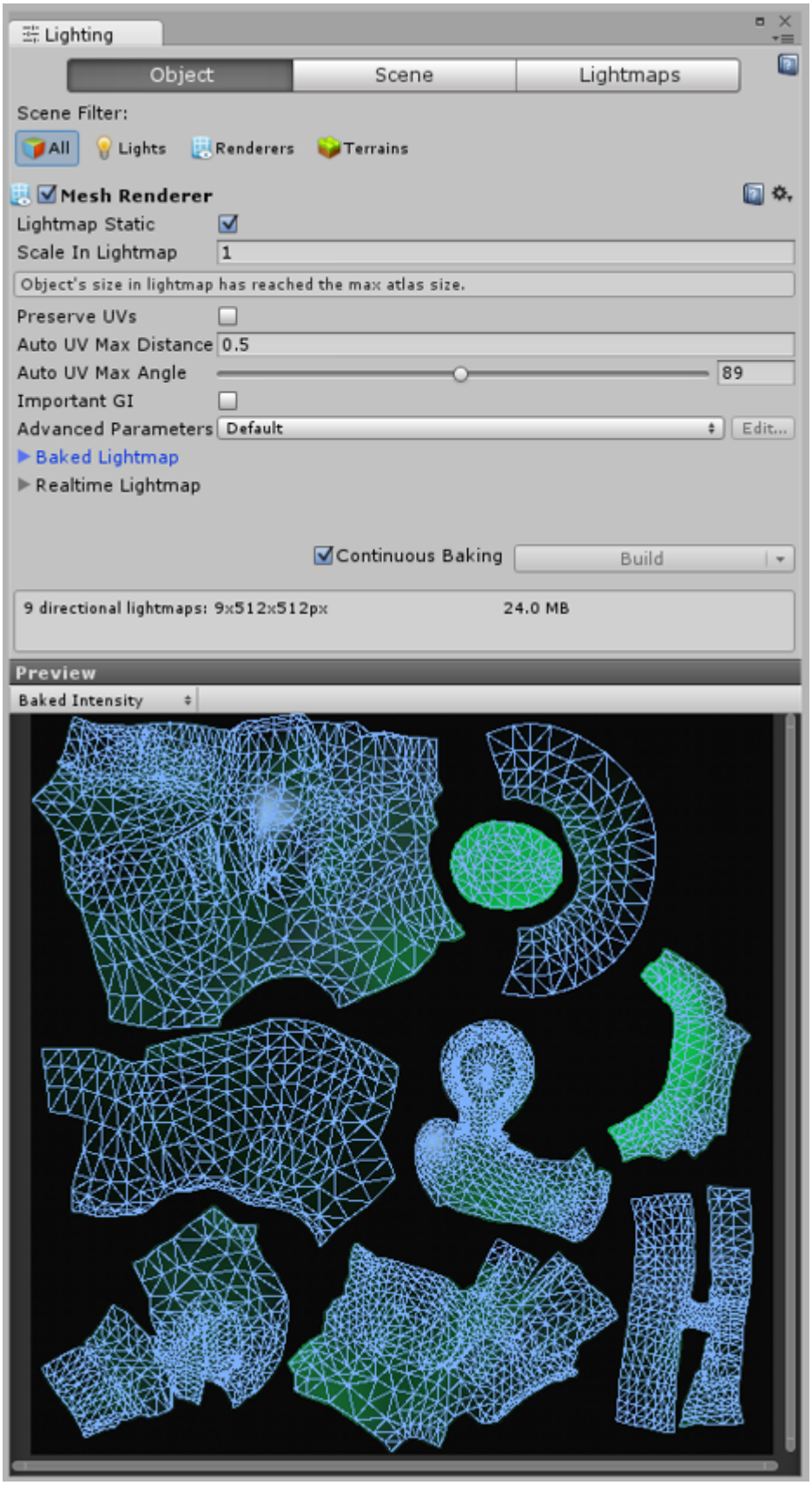


그림 4-8 강도 라이트맵

다음 그림은 동굴 끝에서 녹색 라이트로부터의 조명이 표시되고 있는 편집기를 보여줍니다. 이 조명은 정적 라이트맵을 사용하여 생성됩니다.

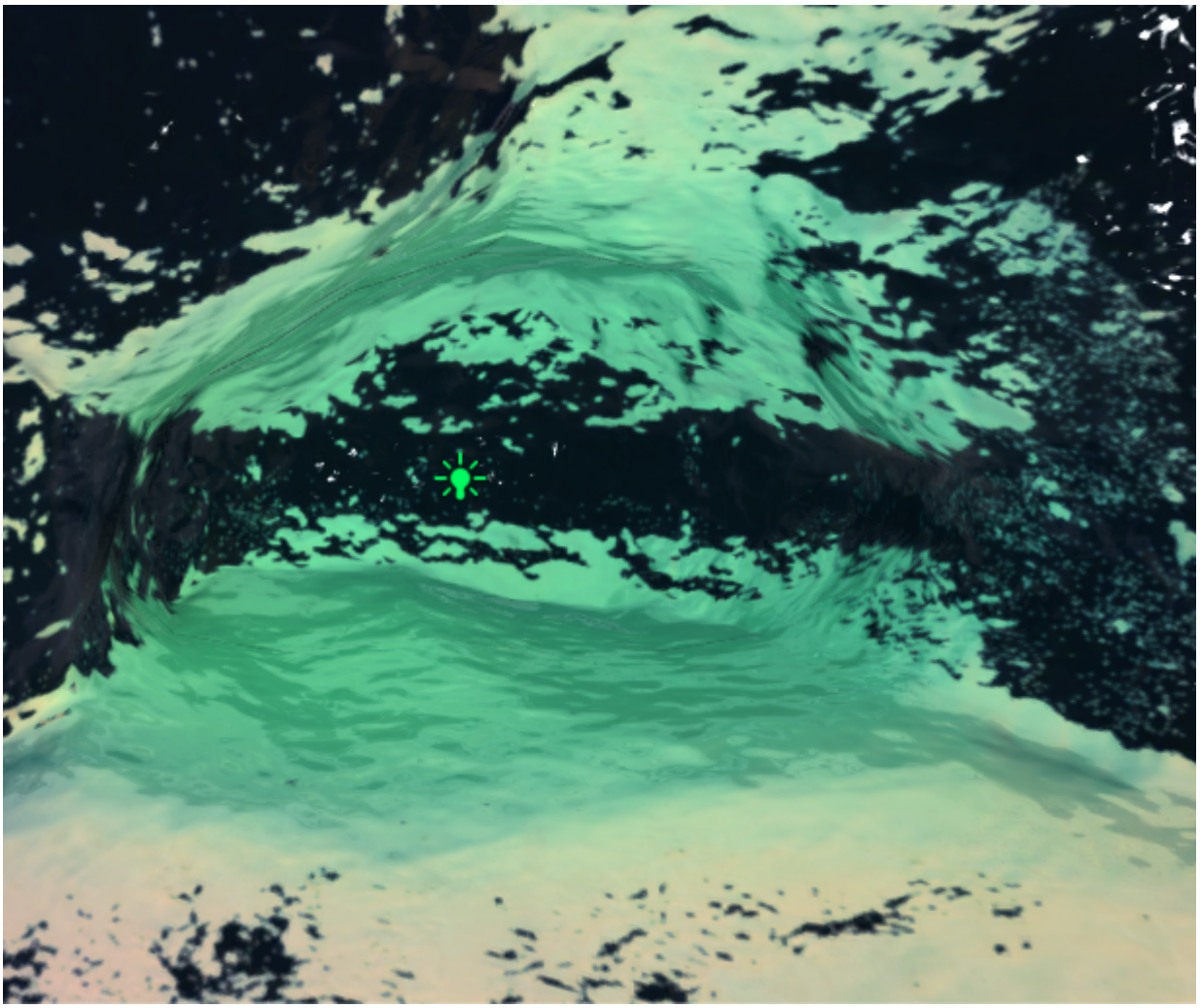

그림 4-9 정적 라이트맵을 베이크할 라이트 추가

다음 그림은 얼음 동굴 데모에서 정적 라이트맵의 결과를 보여줍니다.

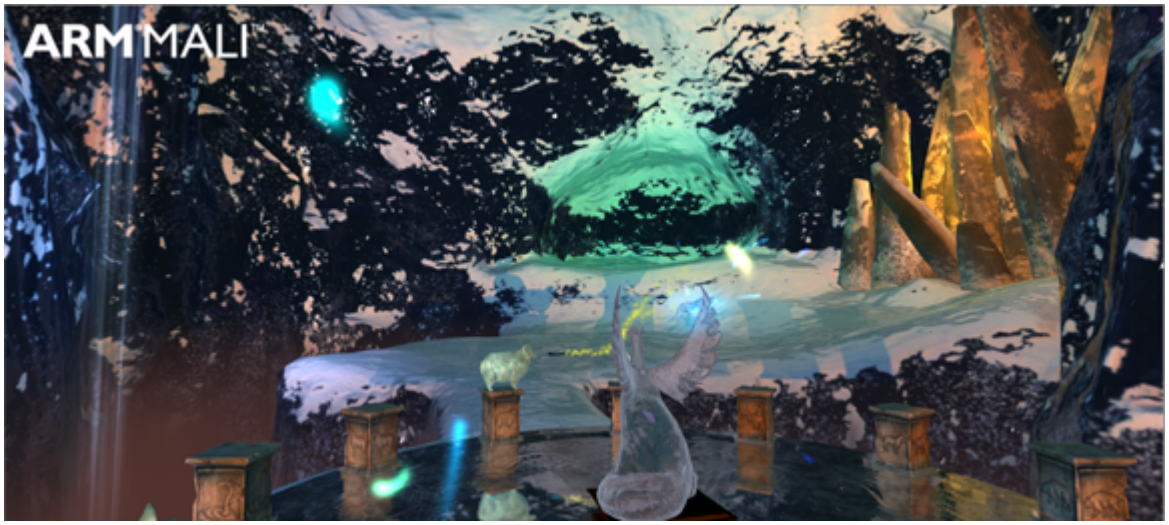


그림 4-10 라이트매핑된 동굴

### 라이트매핑 설정

라이트매핑을 위해 객체를 준비하려면 다음이 필요합니다.

- 장면에 라이트맵 UV를 포함하는 모델이 있어야 합니다.
- 모델이 lightmap static로 표시되어야 합니다.
- 모델의 범위 안에 라이트가 있어야 합니다.
- 라이트의 Baking type이 `Baked`로 설정되어야 합니다.

———— 참고 ————

장면의 정적 객체만 라이트매핑됩니다. 이들은 완전하지 않으므로 실험을 통해 게임에 최상으로 작용하는 라이트매핑을 확인해야 합니다.

정적으로 표시되지 않은 객체는 라이트맵에 배치되지 않습니다. 특정 렌더러를 선택하면 여러 설정이 제공되어 해당 라이트맵이 정적인지 여부를 설정할 수 있습니다.

————————

편집기 창의 메인 메뉴에서 Lighting 창을 열고 Window 및 Lightmapping을 선택합니다. 3개의 버튼이 있습니다.

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

다음 그림은 라이트맵 옵션입니다.


그림 4-11 라이트맵 옵션

**Object**

Object 버튼을 클릭하면 계층에서 선택한 객체에 대한 라이트매핑과 관련된 설정을 변경할 수 있습니다. 그러면 라이트매핑 프로세스에 영향을 미치는 객체 설정을 변경할 수 있습니다. 라이트를 선택하면 다음과 같은 옵션을 변경할 수 있습니다.

- Baked Only 라이트를 베이킹 시 활성화하고 런타임 시 비활성화합니다.
- Baked Baked GI를 선택하면 라이트가 베이크됩니다.
- Realtime 라이트가 미리 계산된 실시간 GI에대해 그리고 GI가 없을 때 모두 작용합니다.
- Realtime Only 라이트를 베이킹 시 비활성화하고 런타임 시 활성화합니다.
- Mixed 라이트가 베이크되지만 런타임 시에도 존재하여 비정적 객체에 직접 조명을 제공합니다.

라이트의 대부분을 Baked로 설정하면 런타임 시 계산 횟수가 비교적 적습니다.

**Scene.**

Scene 탭에는 전체 장면에 적용되는 설정이 포함되어 있습니다. 이 탭에서 Precomputed Realtime GI 및 Baked GI 기능을 설정할 수 있습니다.

Environment Lighting 섹션에는 스카이박스, 주변 광원 유형, 주변 강도와 같이 환경 조명에 영향을 미치는 여러 요소를 정의할 수 있는 옵션들이 있습니다.

- Reflection Bounces 옵션은 성능 관점에서 가장 중요합니다. Reflection Bounces는 반사성 객체 사이의 상호반사의 수, 즉 객체를 보는 프로브의 베이크 시간을 정의합니다. 이 옵션은 런타임 시 반사 프로브가 업데이트될 경우 성능에 상당히 부정적인 영향을 미칠 수 있습니다. 반사성 객체가 프로브에서 확실히 보이는 경우에만 바운스 수를 1보다 높은 수로 설정하십시오.
- Precomputed Realtime GI 탭에서 CPU Usage 옵션은 런타임 시 GI를 평가하기 위해 소비되는 프로세서 시간을 정의합니다. CPU Usage 값이 높을수록 조명 반응이 빨라지지만 프레임 레이트에 악영향을 미칠 수 있습니다. 멀티 프로세서 시스템에서는 성능에 대한 영향이 낮습니다.
- Baked GI 탭에는 압축할 라이트맵 텍스처를 설정할 수 있는 옵션이 포함되어 있습니다. 라이트맵 텍스처를 압축하면 런타임 시 필요한 저장 공간 및 대역폭이 감소하지만 압축 프로세스가 텍스처에 아티팩트를 추가할 수 있습니다.
- General GI 탭에서 Directional Mode 옵션을 설정할 때 주의해야 합니다. 이중 라이트맵을 포함하는 지연 조명을 사용할 수 없는 경우 또 다른 기법은 방향성 라이트맵을 사용하는 것입니다. 방향성 라이트맵은 실시간 라이트 없이 노말 매핑 및 반사성 조명을 사용할 수 있게 해줍니다. 노말 매핑을 보존해야 하지만 이중 라이트맵은 사용할 수 없는 경우 방향성 라이트맵을 사용하십시오. 이 경우는 일반적으로 모바일 디바이스에서 발생합니다. Directional Mode가 Directional으로 설정되면 입사광의 우세한 방향을 저장하기 위해 추가 라이트맵이 생성됩니다. 결과적으로 이 모드는 두 배의 저장 공간을 필요로 합니다.
- Directional Specular 모드에서는 반사성 반사 및 노말 맵에 대해 추가 데이터가 저장됩니다. 이 경우 필요한 저장 공간이 4배로 증가합니다.
- Lightmaps 탭에서는 장면에 사용되는 라이트맵 에셋 파일을 설정하고 찾을 수 있습니다. 라이트맵 스냅샷 상자에 액세스하려면 Continuous Baking 옵션을 해제해야 합니다.

다음 그림은 Lighting 탭의 라이트맵을 보여줍니다.

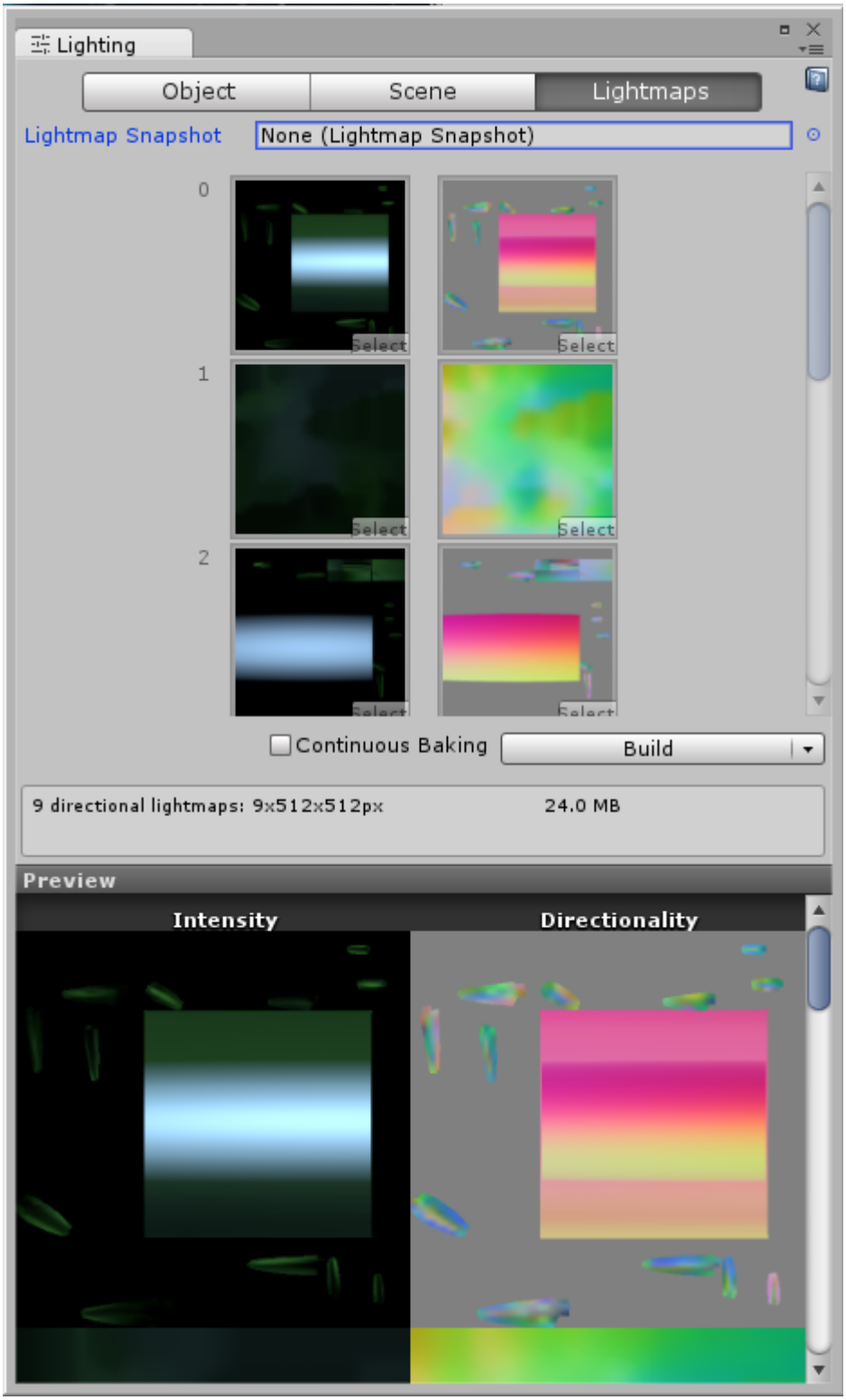


그림 4-12 Lighting 탭 내 라이트맵

### 방향성 라이트맵을 사용

이중 라이트맵을 포함하는 지연 조명을 사용할 수 없는 경우 또 다른 기법은 방향성 라이트맵을 사용하는 것입니다. 방향성 라이트맵은 실시간 라이트 없이 노말 매핑 및 반사성 조명을 사용할 수 있게 해줍니다.

노말 매핑을 보존해야 하지만 이중 라이트맵은 사용할 수 없는 경우 방향성 라이트맵을 사용하십시오. 이 경우는 일반적으로 모바일 디바이스에서 발생합니다.

———— 참고 ————

이 기법은 두 번째 라이트맵 세트를 계산하여 방향 정보를 저장하므로 더 많은 비디오 메모리를 필요로 합니다.

————————————

### 게임 내 동적 객체에 라이트 프로브를 사용

라이트 프로브는 라이트매핑된 장면에 동적 조명을 일부 추가할 수 있게 해줍니다.

다음 그림은 라이트 프로브 설정입니다.

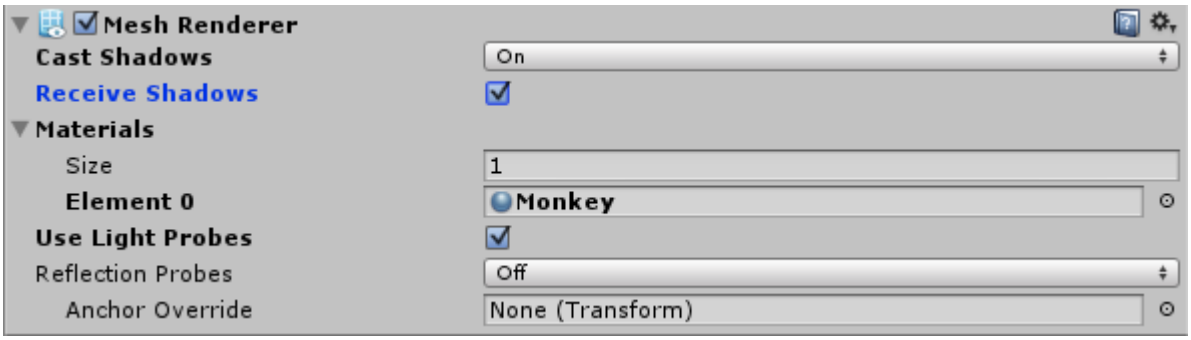


그림 4-13 라이트 프로브 설정

라이트 프로브는 특정 영역에서 조명의 샘플, 즉 프로브를 취합니다. 프로브가 볼륨, 즉 셀을 형성하는 경우 셀 내부에서의 프로브 위치에 따라 조명이 각 프로브 사이에서 보간됩니다.

다음 그림은 라이트 프로브를 보여줍니다.

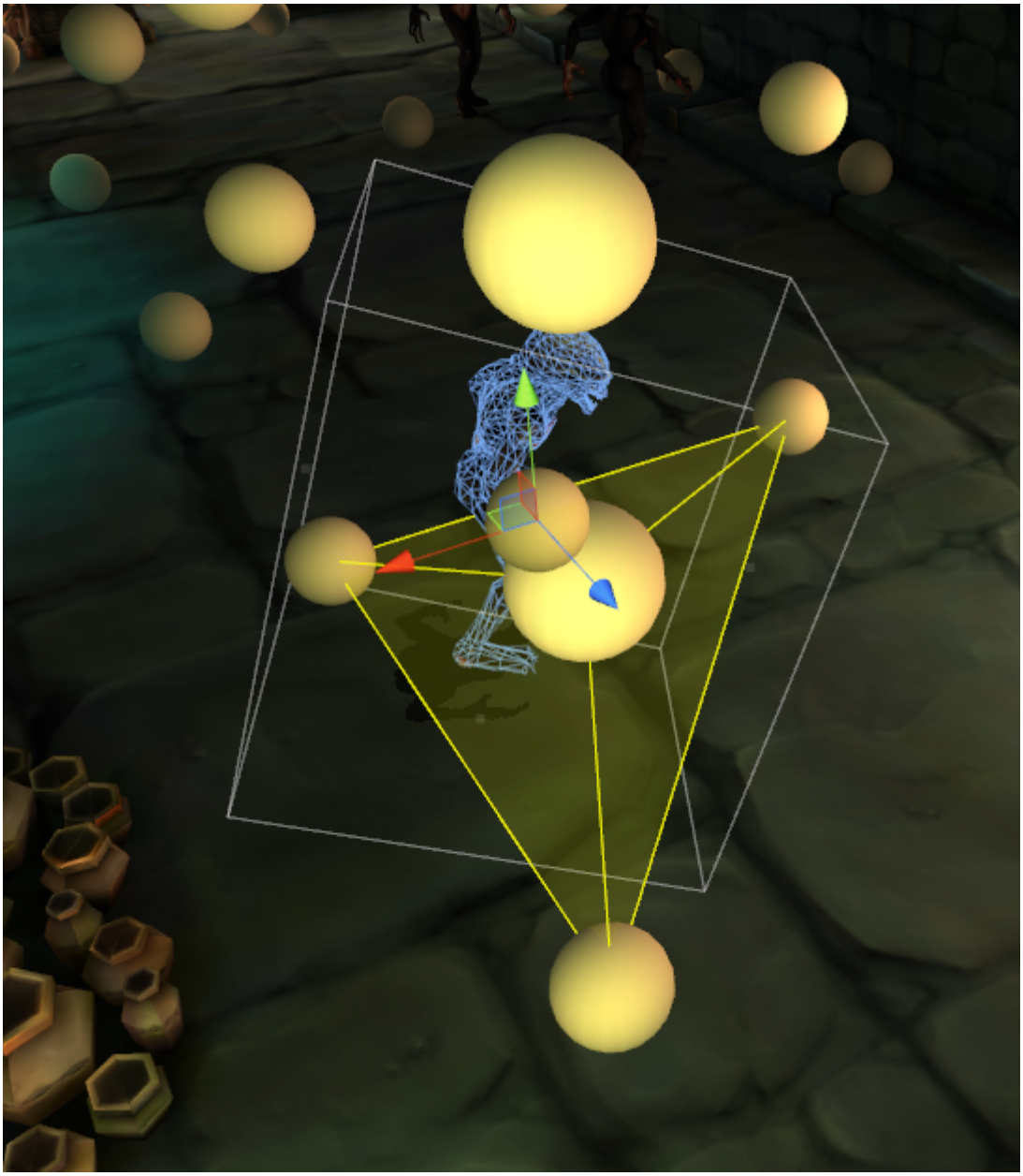

그림 4-14 라이트 프로브

프로브가 많을수록 조명이 정확해집니다. 일반적으로 많은 라이트 프로브가 필요하지는 않습니다. 프로브 사이에서 보간이 이루어지기 때문입니다. 조명 색상 또는 강도가 크게 변화하는 영역에 더 많은 라이트 프로브가 필요합니다.

그러면 임의의 위치에서의 조명을 가장 가까운 프로브에서 취한 샘플 사이를 보간하여 근사치로 계산할 수 있습니다.

라이트 프로브는 신중하게 배치해야 하며 `Use Light Probes` 옵션을 사용하여 영향을 받을 메시를 표시합니다.

다음 그림은 다중 라이트 프로브를 보여줍니다.

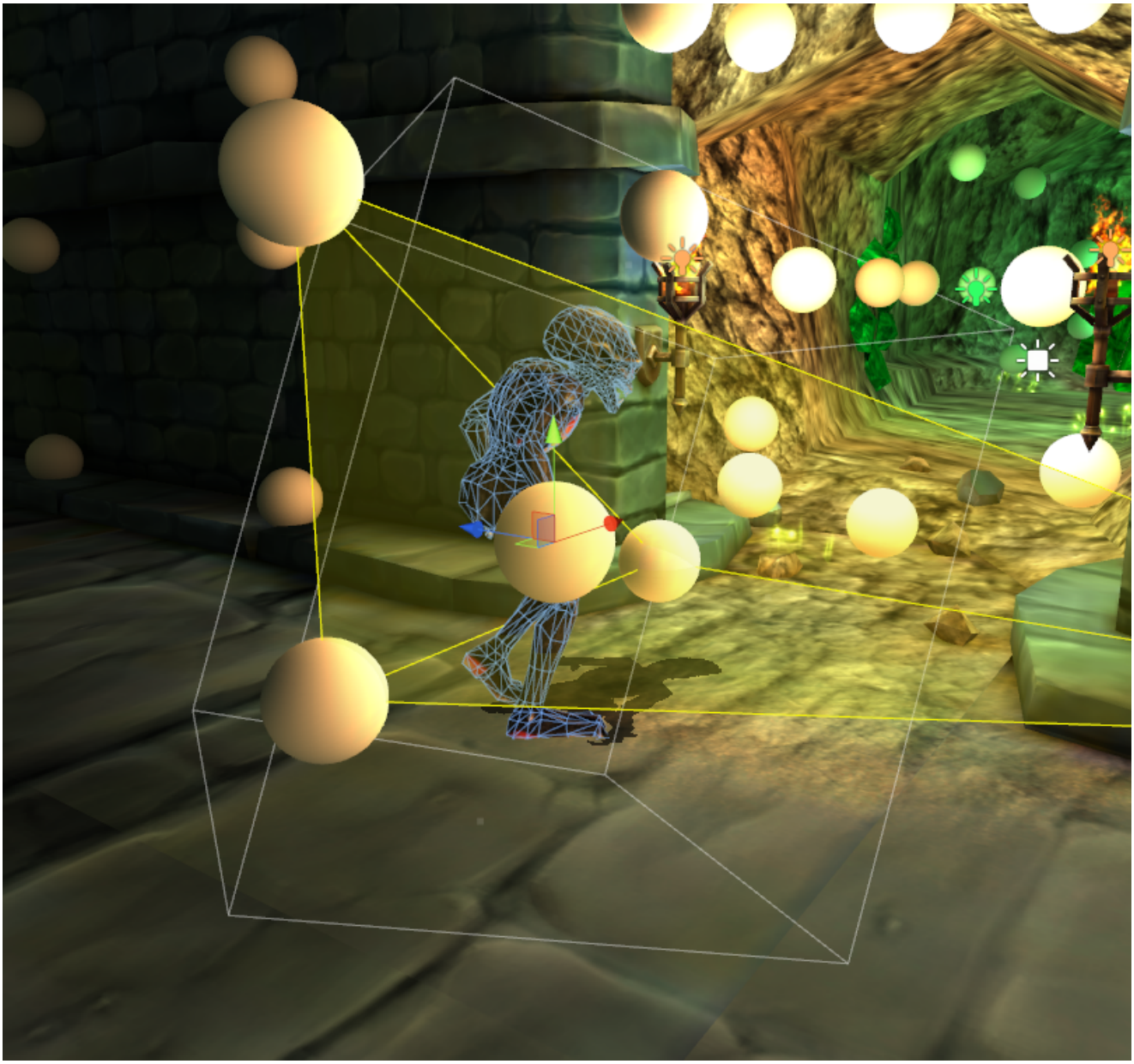

그림 4-15 다중 라이트 프로브

### 4.2.3 ASTC 텍스처 압축

ASTC 텍스처 압축은 OpenGL 및 OpenGL ES 그래픽 API의 공식 확장자입니다. ASTC는 응용 프로그램에 필요한 메모리를 줄일 수 있고 GPU에 필요한 메모리 대역폭을 좁힐 수 있습니다.

ASTC는 고품질의 낮은 비트레이트로 텍스처 압축을 제공하며 제어 옵션이 다양합니다. 여기에는 다음 기능이 포함됩니다.

- 8bpp(*픽셀당 비트*)에서 1bpp 미만까지의 비트레이트 범위. 그러므로 파일 크기와 품질 간 절충을 미세하게 조정할 수 있습니다.
- 1~4개 색상 채널을 지원.
- *저 다이내믹 레인지*(LDR) 및 *고 다이내믹 레인지*(HDR) 이미지를 모두 지원.
- 2D 및 3D 이미지 지원.
- 기능을 다양한 조합으로 선택 가능.

다음 그림은 ASTC 설정 창입니다.


그림 4-16 ASTC 설정

ASTC 설정 창에서는 여러 가지의 블록 크기 옵션을 제공합니다. 이러한 옵션 중에서 선택하고 에셋에 가장 적합한 블록 크기를 선택할 수 있습니다. 큰 블록 크기일수록 높은 압축을 제공합니다. 세밀한 디테일로 표시되지 않는 텍스처에 대해서는 큰 블록 크기를 선택합니다(예: 카메라와 멀리 떨어져 있는 객체). 보다 디테일하게 텍스처에 대해서는 더 작은 블록 크기를 선택합니다(예: 카메라와 가까이 있는 객체).

참고

- 디바이스가 ASTC를 지원할 경우 이를 사용하여 3D 콘텐츠에서 텍스처를 압축하십시오. 디바이스가 ASTC를 지원하지 않을 경우, ETC2를 사용해 보십시오.
- 3D 콘텐츠에서 사용되는 텍스처를 GUI 요소에 사용되는 텍스처와 구별해야 합니다. 때로는 GUI 텍스처를 비압축 상태로 유지하는 것이 불필요한 아티팩트를 방지하는 데 가장 좋을 수 있습니다.

다음 그림은 텍스처 압축 형식별로 사용 가능한 블록 크기입니다.

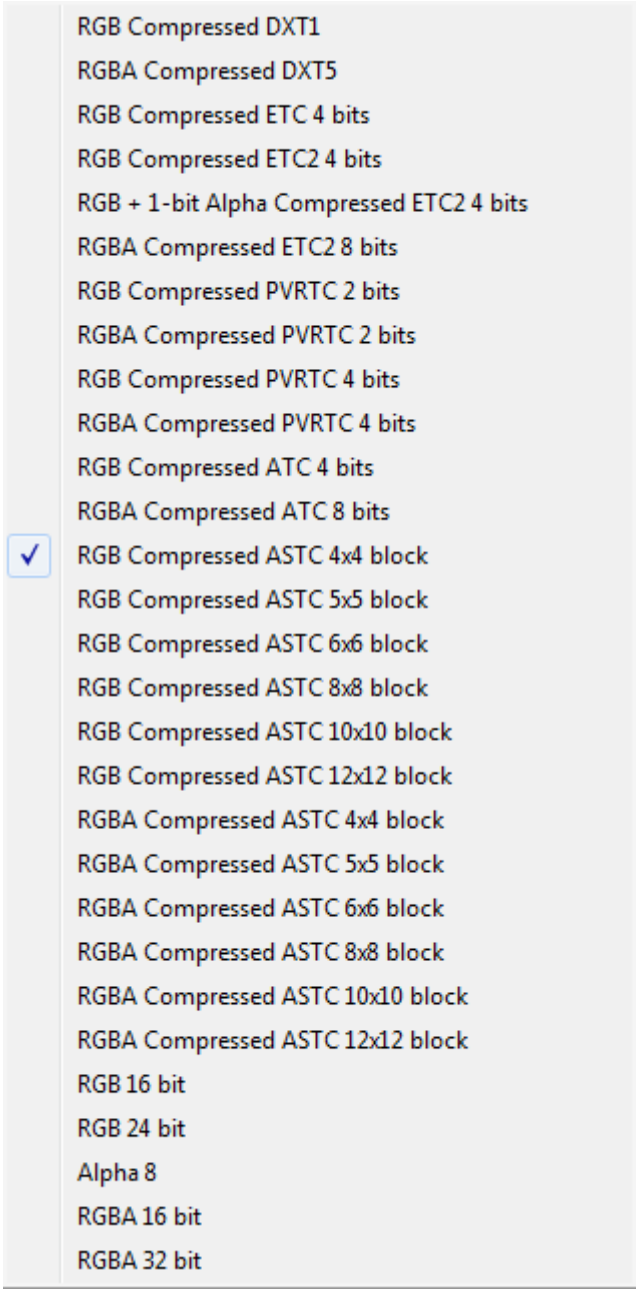


그림 4-17 텍스처 압축 블록 크기

### ASTC 텍스처에 대해 올바른 형식을 선택

ASTC 텍스처를 압축할 때 몇 가지 옵션 중에서 선택할 수 있습니다.

텍스처 압축 알고리즘은 여러 채널 형식을 사용하는데, RGB 및 RGBA가 일반적입니다. ASTC는 다른 형식을 다수 지원하지만 이들 형식은 Unity 내에서 노출되지 않습니다. 각 텍스처는 일반적으로 표준 텍스처링, 노말 매핑, 스펙큘러, HDR, 알파 및 록업 텍스처와 같은 다른 용도로 사용됩니다. 이들 텍스처는 다른 압축 형식을 사용해야 모두 최상의 결과를 얻을 수 있습니다.

다음 그림은 텍스처 설정입니다.

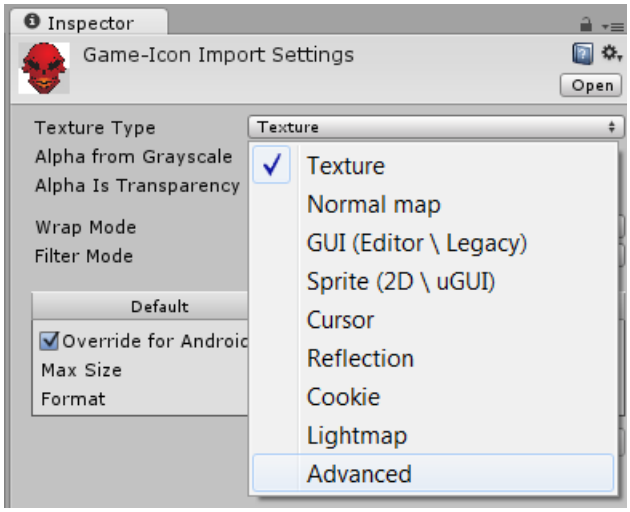


그림 4-18 텍스처 설정

모든 텍스처를 Build Settings 내 형식 한 가지만 사용하여 압축하지 마십시오. 텍스처 압축을 Don’t Override로 유지하십시오.

프로젝트 계층 안에서 텍스처를 찾아 검사기로 불러옵니다. 일반적으로 Unity는 텍스처를 Texture 유형으로 가져옵니다. 이 유형은 압축 옵션을 일부만 제공합니다. 더 많은 옵션이 표시되도록 유형을 Advanced로 설정합니다.

다음 다이어그램은 약간의 투명도를 갖는 GUI 텍스처에 대한 설정입니다. 이 텍스처는 GUI 용이므로 sRGB 및 MipMaps는 해제됩니다. 투명도를 포함시키려면 알파 채널이 필요합니다. 이를 활성화려면 Alpha Is Transparency 확인란을 선택하고 Override for Android 상자를 선택합니다.

다음 그림은 고급 텍스처 설정입니다.

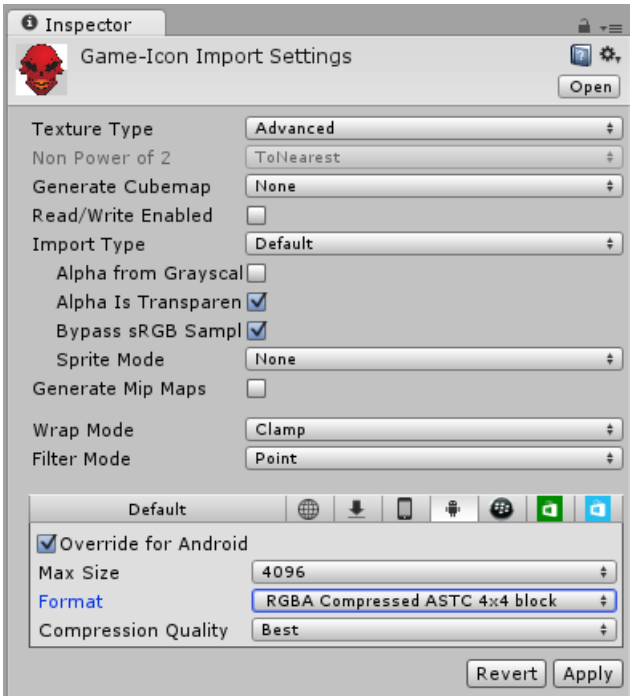


그림 4-19 고급 텍스처 설정

형식 및 블록 크기를 선택할 수 있는 옵션이 있습니다. RGBA는 알파 채널을 포함하며, 4x4는 선택 가능한 최소 블록 크기입니다. Max texture size를 최대로 설정하고 Compression

Quality를 설정하십시오. 이 설정은 정확한 압축을 찾는 데 시간을 얼마나 사용할지 정의합니다.

모든 텍스처에 고유한 설정을 선택하면 프로젝트의 시각적 품질이 향상되고 압축 시 불필요한 텍스처 데이터가 방지됩니다.

다음 표에는 4MB 크기에서 1024x1024 픽셀 해상도를 갖는 RGBA 8bpc 텍스처를 기준으로 Unity에서 사용 가능한 ASTC 블록 크기에 대한 압축비가 나와 있습니다.

표 4-1 Unity에서 사용 가능한 ASTC 블록 크기에 대한 압축비

| ASTC 블록 크기 | 크기 | 압축비 |
|---|---|---|
| 4x4 | 1MB | 4.00 |
| 5x5 | 655KB | 6.25 |
| 6x6 | 455KB | 9.00 |
| 8x8 | 256KB | 16.00 |
| 10x10 | 164KB | 24.97 |
| 12x12 | 144KB | 35.93 |

### 4.2.4 밉매핑

밉매핑은 게임의 시각적 품질 및 성능을 모두 향상할 수 있는 텍스처와 관련된 기법입니다.

밉맵은 다양한 크기로 미리 계산된 텍스처 버전입니다. 생성된 각 텍스처를 레벨이라고 부르며, 각 레벨은 너비 및 높이가 이전 레벨의 절반입니다. Unity는 원래 크기의 1차 레벨에서 1x1 픽셀 버전까지 전체 레벨 세트를 자동으로 생성할 수 있습니다.

밉맵을 생성하려면 다음을 수행합니다.
1. Project window에서 텍스처를 선택합니다.
2. Texture Type을 Advanced로 변경합니다.
3. 검사기에서 Generate Mip Maps 옵션을 선택합니다.

다음 그림은 밉맵 설정입니다.

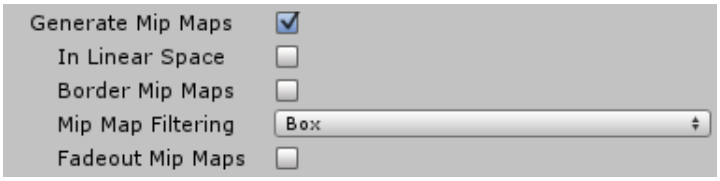


그림 4-20 MIP 맵 설정

텍스처에 밉맵 레벨이 없는 경우, 텍스처 처리된 표면이 커버하는 픽셀 내 영역이 텍스처의 크기보다 작으면 GPU가 더 작은 영역에 맞춰 텍스처를 축소합니다. 하지만 이 과정에서 필터를 사용하여 픽셀 색상을 보간하더라도 정확도가 어느 정도 감소합니다.

텍스처에 밉맵 레벨이 있는 경우 GPU는 객체 크기와 가장 비슷한 레벨로부터 데이터를 가져와 텍스처를 렌더링합니다. 그러면 이미지 품질이 향상되는 동시에 대역폭이 감소하는데, 품질 향상을 위해 레벨 스케일링이 오프라인에서 이루어지고 GPU가 올바른 레벨의 텍스처 데이터만 가져오기 때문입니다. 밉매핑의 단점은 텍스처 데이터를 저장하기 위한 메모리가 33% 더 필요하다는 것입니다.

#### 밉맵 및 GUI 텍스처

일반적으로 2D UI에 사용되는 텍스처에는 밉매핑이 필요하지 않습니다. UI 텍스처는 대개 크기 조정 없이 화면에서 렌더링되기 때문에 밉맵 체인에서 첫 번째 레벨만 사용합니다.

이 설정을 변경하려면 Project 창에서 텍스처를 선택하고 검사기로 이동하여 Texture Type을 확인합니다. 유형을 Editor GUI and Legacy GUI로 설정하거나, 유형을 Advanced로 설정하고 Generate Mip Maps 옵션을 해제합니다.

### 4.2.5 스카이박스

스카이박스는 게임 및 기타 다른 응용 프로그램에서 자주 사용되는데, 스카이박스를 구현하는 방법은 여러 가지가 있습니다.

단일 큐브맵을 사용하여 카메라의 배경을 렌더링하는 방식으로 스카이박스를 그릴 수 있습니다.

이를 위해서는 큐브맵 텍스처와 드로우 콜이 하나씩 필요합니다. 이 방법은 다른 방법에 비해 메모리, 메모리 대역폭, 드로우 콜을 덜 사용합니다.

이 방법을 사용하여 스카이박스를 설정하는 방법

1. 카메라를 선택합니다.
2. Clear Flags가 Skybox로 설정되어 있는지 확인합니다.
3. 스카이박스 구성요소를 선택하거나 추가합니다.

스카이박스 구성요소는 한 자료에 하나의 스폿을 가집니다. 이것은 Unity가 각 프레임이 시작할 때 카메라의 배경을 그리기 위해 사용하는 자료입니다.

사용할 자료는 필요한 정보를 모두 포함하는 자료입니다. Mobile > Skybox 쉐이더를 사용하여 자료를 생성하고 스카이 상자의 6개 이미지를 자료로 채웁니다. 자료 미리 보기에 이미지가 표시됩니다.

완료된 자료를 스카이박스 구성요소로 끌어 넣습니다. 스카이박스가 확연한 심 또는 불필요한 드로우 콜 없이 배경에서 제대로 렌더링됩니다.

### 4.2.6 Shadows

그림자는 장면에 원근감과 사실성을 더하는 데 유용합니다. 그림자가 없으면 객체의 입체감을 살리기 어려울 수가 있습니다. 주변 객체들이 비슷할 경우에는 더욱 그렇습니다.

그림자 알고리즘은 매우 복잡할 수 있습니다. 그림자를 정확하고 고해상도로 렌더링하는 경우에는 특히 그렇습니다. 게임의 그림자에 적절한 레벨의 복잡도 및 해상도를 선택하도록 하십시오.

Unity는 실시간 그림자를 계산하기 위한 변환 피드백을 지원합니다.

———— 참고 ————

얼음 동굴 데모는 사용자 지정 그림자를 구현합니다. 로컬 큐브맵에 기반한 그림자는 런타임 시 렌더링되는 그림자와 결합됩니다.

————————————

Unity는 Edit > Project Settings > Quality에서 게임의 성능에 영향을 미칠 수 있는 다수의 그림자 옵션을 제공합니다.

**Hard/Hard and Soft Shadows**

Soft Shadows는 보다 사실적이지만 계산이 더 오래 걸립니다.

**Shadow Distance**

Shadow Distance 옵션은 그림자가 표시되는 카메라와의 거리를 정의합니다. 그림자 거리를 늘리면 표시되는 그림자 수가 증가하고, 따라서 연산 부하도 증가합니다. 그림자 거리를 늘릴 경우 그림자 맵에서 그림자에 사용 가능한 텍셀 수도 증가하여 수동적으로 그림자 해상도가 높아집니다.

Hard Shadows를 낮은 그림자 거리와 높은 해상도로 사용할 수 있습니다. 그러면 카메라와 양호한 거리에서 너무 복잡하지 않은 적당한 품질의 그림자가 생성됩니다.

라이트매핑된 객체는 실시간 그림자를 생성하지 않습니다. 따라서 장면에 정적인 그림자를 많이 베이크할수록 GPU가 수행하는 실시간 계산이 줄어듭니다.

다음 그림은 그림자가 있는 외계인 캐릭터입니다.

그림 4-21 그림자가 있는 외계인

#### 실시간 그림자 사용을 절제

실시간 그림자는 장면의 사실성을 극적으로 높여주지만 연산 비용이 높습니다.

모바일 디바이스에서는 실시간 그림자를 포함하는 라이트의 수를 제한하고 그 대신 라이트매핑을 사용하도록 노력하십시오.

장면 내 객체의 메시 렌더러 구성요소를 고려해 보십시오. 그림자를 캐스팅하거나 받기 위해 사용할 용도가 아니라면 Cast Shadows 및 Receive Shadows 옵션을 적절히 해제합니다. 그러면 그림자 렌더링의 연산 비용이 절감됩니다.

Quality Settings 섹션에 그림자에 대한 더 많은 설정이 있습니다.

- Shadow Resolution 품질과 처리 시간 사이의 균형을 선택할 수 있습니다.
- Shadow Distance 그림자 생성을 카메라와 가까운 객체로 제한할 수 있습니다.
- Shadow Cascades 품질과 처리 시간 사이의 균형을 선택할 수 있습니다. 이 값을 0, 2 또는 4로 설정할 수 있습니다. 캐스케이드 그림자 맵은 특히 멀리 보는 거리에서 매우 뛰어난 그림자 품질을 얻기 위해 방향성 라이트에 사용됩니다. 캐스케이드 수가 많을수록 품질이 향상되지만 처리 오버헤드도 증가합니다.

### 4.2.7 오클루전 컬링

오클루전 컬링은 객체가 카메라 시야에서 가려진 상태에서는 객체의 렌더링을 비활성화하는 프로세스입니다. 이 프로세스는 렌더링할 객체를 줄여 GPU 처리 시간을 절약해줍니다.

Unity는 객체가 카메라 프루스텀을 완전히 벗어날 때 자동으로 프루스텀 컬링을 수행하지만, 응용 프로그램의 스타일에 따라서는 화면에 보이지 않아 렌더링할 필요가 없는 다른 객체가 또 있을 수 있습니다.

Unity는 Umbra라고 하는 오클루전 컬링 시스템을 포함하고 있습니다. Umbra에 관한 자세한 내용은 Unity 설명서에서 오클루전 컬링을 참조하십시오.

오클루전 컬링에 사용되는 설정은 게임의 스타일에 따라 달라집니다. 잘못된 설정으로 장면에서 오클루전 컬링을 사용하면 성능이 저하될 수 있으므로 신중하게 설정을 선택해야 합니다.

### 4.2.8 OnBecameVisible() 및 OnBecomeInvisible() 콜백 사용

`MonoBehaviour.OnBecameVisible()` 및 `MonoBehaviour.OnBecameInvisible()` 콜백을 사용할 경우, Unity는 연결된 게임 객체가 카메라 프루스툼에 들어가거나 나올 때 스크립트에 통보합니다. 그러면 응용 프로그램이 그에 따라 동작할 수 있습니다.

`OnBecameVisible()` 및 `OnBecameInvisible()`을 사용하여 렌더링 프로세스를 최적화할 수 있습니다. 예를 들어 두 번째 카메라와 렌더 타겟을 사용하여 풀에서 반사를 렌더링하는 식입니다.

여기에는 최종 화면 표면으로 렌더링하기 전에 지오메트리를 렌더링하고 화면 밖의 텍스처를 결합하는 프로세스가 포함됩니다. 이 기법은 비교적 연산 비용이 높으므로 필요할 때에만 사용됩니다. 반사를 표시되는 경우에만 렌더링할 필요가 있습니다. 즉, 다음 조건을 충족하는 경우입니다.

- 반사 표면이 카메라 프루스툼 안에 위치.
- 표면 앞에 불투명한 것이 없음.

이러한 조건은 반사 표면으로부터`OnBecameVisible()` 및 `OnBecameInvisible()` 콜백을 사용하여 확인됩니다.

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

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

이러한 확인이 구현되어 있더라도 화면에 보이지 않는 반사가 화면 밖에서 렌더링되는 경우가 생깁니다. 이를 방지하기 위해 다른 조건을 추가할 수 있습니다.

예를 들어 카메라가 반사 표면의 실내 안에 있어야 합니다.

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

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

void OnTriggerEnter()
{
    inside = true;
}

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

이러한 조건은 반사 렌더링을 게임의 특정 영역으로 제한합니다. 이는 게임의 덜 연산 집약적인 다른 영역에 효과를 추가할 수 있음을 의미합니다.

### 4.2.9 렌더링 순서를 지정

장면에서 객체 렌더링 순서는 성능에 매우 중요합니다.

객체가 임의의 순서로 렌더링될 경우 객체가 렌더링된 후 그 앞의 다른 객체에 의해 오클루드될 수 있습니다. 즉 오클루드된 객체를 렌더링하는 데 사용된 연산은 낭비된 것입니다.

오클루드된 객체로 인한 연산 낭비를 줄이기 위한 소프트웨어 및 하드웨어 기법이 다양하게 존재합니다. 하지만 플레이어가 어떻게 장면을 탐색하는지에 대한 지식이 있는 사용자가 이 프로세스를 유도할 수 있습니다.

Mali -T600 시리즈 이후의 ARM Mali GPU에서 사용 가능한, 연산 낭비를 줄이기 위한 하드웨어 기법 중 하나가 Early-Z입니다. Early-Z는 조각 셰이더가 실제로 처리되기 전에 Z 테스트를 수행하는, 사용자 관점에서 완전히 투명한 시스템입니다. GPU가 Early-Z 최적화를 활성화할 수 없는 경우 깊이 테스트가 조각 셰이더 이후에 실행됩니다. 그러면 연산 비용이 높아질 수 있으며 해당 조각이 오클루드될 경우 연산이 낭비될 수 있습니다. Early-Z 시스템은 처리 중인 픽셀의 깊이가 인접한 픽셀에 의해 이미 오클루드되지 않았는지 확인합니다. 오클루드된 경우 조각 셰이더를 실행하지 않습니다. 이 시스템은 성능상 장점이 있지만, 예를 들어 조각 셰이더가 `gl_FragDepth` 변수를 통해 깊이를 수정하는 경우, 조각 셰이더가 `discard`를 호출하는 경우, 또는 블렌딩 또는 투명 객체와 같은 객체에 대해 알파 테스트가 활성화된 경우 등 일부의 경우에는 자동으로 비활성화됩니다. 이 시스템이 최대의 효율을 발휘하도록 하려면 불투명 객체가 앞에서 뒤로 렌더링되도록 하십시오. 그러면 불투명 객체만 있는 장면에서 오버드로 요소가 감소합니다.

각 프레임의 렌더링 순서를 앞에서 뒤로 설정하는 것은 컴퓨팅 비용이 클 수 있으며, 동일한 패스에서 투명 객체를 렌더링할 경우 부정확할 수도 있습니다. T620 이후의 ARM Mali GPU는 *Pixel Forward Kill*(PFK)이라는 메커니즘을 제공합니다. Mali GPU는 한 픽셀에서 여러 스레드가 동시에 실행되도록 파이프라인되어 있습니다. 한 스레드가 실행을 완료하면 현재 스레드가 이들을 커버할 경우 PFK 시스템이 그 픽셀에서 실행되는 다른 모든 스레드를 중단시킵니다. 그 결과 연산 낭비가 줄어듭니다.

Unity는 쉐이더 또는 자료에서 렌더링 순서를 지정할 수 있는 Queue 옵션을 제공합니다. 이 옵션은 특정 쉐이더를 사용하는 자료를 갖는 객체가 함께 렌더링되도록 해당 쉐이더에서 설정할 수 있습니다. 이 렌더링 그룹 내부에서는 투명도와 같은 일부 경우를 제외하고 렌더링 순서가 무작위적입니다.

기본적으로 Unity는 처음에서 끝까지 다음 순서로 렌더링되는 몇몇 표준 그룹을 제공합니다.

표 4-2 렌더링 순서를 지정하기 위한 큐 값

| 이름 | 값 | 메모 |
|---|---|---|
| Background | 1000 | - |
| Geometry | 2000 | 불투명 지오메트리에 사용되는 기본값. |
| AlphaTest | 3000 | 이 그룹은 모든 불투명 객체 이후에 드로잉됩니다. 예: 나뭇잎. |
| Transparent | 4000 | 이 그룹도 올바른 결과를 생성하기 위해 뒤에서 앞의 순서로 렌더링됩니다. |
| Overlay | 5000 | Overlay는 사용자 인터페이스, 렌즈 플레어, 더티 렌즈와 같은 효과에 영향을 미칩니다. |

문자열 이름 대신 정수 값을 사용할 수 있습니다. 이러한 값만 사용할 수 있는 것은 아닙니다. 표시된 정수 값 사이의 값을 사용하여 다른 큐를 지정할 수 있습니다. 값이 클수록 나중에 렌더링됩니다.

예를 들어 다음 명령어 중 하나를 사용하여 특정 쉐이더를 Geometry 큐보다는 늦지만 AlphaTest 큐보다 먼저 렌더링할 수 있습니다.

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

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

#### 렌더링 순서를 사용하여 성능 향상

얼음 동굴 데모에서 동굴은 화면의 많은 부분을 차지하며 해당 쉐이더는 연산 비용이 높습니다. 가능한 한 렌더링 부분을 줄이면 성능을 개선할 수 있습니다.

렌더링 순서 최적화는 Unity 프레임 디버거를 비롯해 Graphics Analyzer 같은 다른 도구를 사용하여 프레임 버퍼의 구성을 확인한 후에 포함되었습니다. 이러한 도구를 통해 렌더링 순서를 확인할 수 있습니다.

Unity 프레임 디버거를 열려면 Window > Frame Debugger 메뉴 옵션을 선택합니다. 편집기 모드에서는 올바른 것으로 보이지만 실행 시에는 제대로 작동하지 않는 것들이 있을 수 있으므로 이 도구가 유용합니다. 예를 들어 런타임 전용 설정을 사용하거나 텍스처를 다른 카메라로 렌더링하는 경우 등입니다. 플레이 모드에서 데모를 시작하고 카메라를 배치한 후 프레임 디버거를 활성화하고 Unity가 실행하는 드로잉의 시퀀스를 가져옵니다.

얼음 동굴 데모에서는 드로우 콜을 아래로 스크롤하면 먼저 동굴이 렌더링됩니다. 그런 다음 객체가 장면으로 렌더링되어 동굴의 이미 렌더링된 부분을 오클루드합니다. 다른 예로는 일부 장면에서 동굴에 의해 오클루드되는 반짝이는 수정이 있습니다. 이러한 경우 렌더링 순서를 더 높게 설정하면 오클루드되는 수정에 대해 조각 쉐이더가 실행되지 않으므로 연산이 감소합니다.

### 4.2.10 깊이 프리패스를 사용

객체 렌더링 순서를 설정하여 오버드로우를 방지하는 것은 유용하지만 각 객체에 대해 렌더링 순서를 지정하는 것이 항상 가능한 것은 아닙니다.

예를 들어 연산 비용이 높은 쉐이더를 포함하는 객체 세트가 있고 카메라가 이들 객체 주위를 자유로이 회전할 수 있는 경우 뒤쪽에 위치하던 일부 객체가 앞쪽으로 이동할 수 있습니다. 이 경우, 이들 객체에 정적 렌더링 순서가 설정된 경우 일부 객체는 오클루드되었음에도 마지막에 드로우될 수도 있습니다. 객체가 자신의 일부를 가릴 수 있는 경우에도 이러한 상황이 발생할 수 있습니다.

이러한 경우 깊이 프리패스를 사용하여 오버드로우를 줄일 수 있습니다. 깊이 프리패스는 프레임 버퍼에 색상을 기록하지 않고 지오메트리를 렌더링합니다. 이 프로세스는 가장 가까운 가시 객체의 깊이를 사용하여 각 픽셀에 대해 깊이 버퍼를 시작합니다. 이 프리패스 후 지오메트리가 정상적으로 렌더링되지만 Early-Z 기법을 사용하면 최종 장면에 기여하는 객체만 실제로 렌더링됩니다. 이 기법에서는 추가 정점 쉐이더 연산이 필요합니다. 정점 쉐이더가 각 객체에 대해 깊이 버퍼를 채우기 위해 한 번, 실제 렌더링을 위해 한 번, 모두 두 번 연산되기 때문입니다. 게임이 조각에 바운드되고 정점 쉐이더에 여분의 용량이 있는 경우 이 기법은 매우 유용합니다.

Unity에서 사용자 지정 쉐이더를 사용하는 객체에 대해 렌더링 프리패스를 수행하려면 쉐이더에 추가 패스를 추가하십시오.

```
// extra pass that renders to depth buffer only
 Pass {
     ZWrite On
     ColorMask 0
 }
```

이 패스를 추가한 후 프레임 디버거는 객체가 두 번 렌더링됨을 표시합니다. 첫 번째 렌더링될 때는 색상 버퍼에 변화가 없습니다.

———— 참고 ————

프레임 디버거의 좌측 상단 메뉴에서 선택하면 깊이 버퍼를 볼 수 있습니다.

————————

## 4.3 에셋 최적화

다음 목록은 에셋 최적화에 대한 설명입니다.

정적 텍스처 읽기/쓰기를 비활성화
: 동적으로 텍스처를 수정하지 않을 경우 Inspector에서 Read/Write Enabled 옵션이 비활성화되어 있는지 확인하십시오.

메시를 결합하여 드로우 콜을 감소
: 렌더링에 필요한 드로우 콜 수를 줄이려면 `Mesh.CombineMeshes()` 메서드를 사용하여 여러 메시를 하나로 결합할 수 있습니다. 메시가 동일한 자료를 공유할 경우 콤바인 그룹의 각 메시에서 단일 서브 메시가 생성되도록 `mergeSubMeshes` 인수를 `true`로 설정합니다.
여러 메시를 더 큰 규모의 단일 메시로 결합할 경우,
- 보다 효과적인 오클루더를 만들 수 있습니다.
- 타일 기반 에셋을 하나의 원활한 대규모 솔리드 에셋으로 변환할 수 있습니다.

메시 콤바인 스크립트는 성능 최적화에 유용할 수 있지만 정면의 구성에 의존합니다. 대규모 메시는 더 작은 규모의 메시에 비해 뷰에 오래 머무는 경향이 있으므로 실험을 통해 올바른 크기를 결정합니다.

이 기법을 적용하기 위한 한 방법은 계층에서 빈 게임 객체를 하나 생성한 다음, 결합할 모든 메시의 상위로 만들어 스크립트에 연결하는 것입니다.

메시 콤바인 스크립트에 관한 자세한 내용은 Unity 설명서를 참조하십시오.

*http://unity3d.com*.

애니메이션화되지 않는 FBX 메시 모델에서 애니메이션 데이터를 가져오지 않음
: 애니메이션 데이터를 포함하지 않은 FBX 메시를 가져올 때 가져오기 설정의 Rig 탭에서 Animation Type을 None으로 설정할 수 있습니다. 이와 같이 설정할 경우 메시를 계층에 배치할 때 Unity가 사용되지 않는 애니메이터 구성요소를 생성하지 않습니다.

읽기/쓰기 메시를 사용하지 않음
: 런타임 시 모델이 수정되는 경우 Unity가 원본은 보존한 상태에서 수정할 메시 데이터의 두 번째 사본을 메모리에 유지합니다.
런타임 시 모델이 수정되지 않는 경우 가져오기 설정의 Model 탭에서 Read/Write Enabled 옵션을 비활성화합니다. 두 번째 사본이 더 이상 필요하지 않으므로 메모리가 절약됩니다.

### 텍스처 아틀라스 사용

텍스처 아틀라스를 사용하여 객체 세트에 필요한 드로우 콜 수를 줄일 수 있습니다. 텍스처 아틀라스는 일단의 텍스처가 큰 규모의 단일 텍스처로 결합된 것입니다. 여러 객체가 적절한 좌표 세트를 사용하여 이 텍스처를 재사용할 수 있습니다. 그러면 Unity가 동일한 자료를 공유하는 객체에 적용하는 자동 배칭에 도움이 됩니다.

객체의 UV 텍스처 좌표를 설정할 때 해당 자료의 `mainTextureScale` 및 `mainTextureOffset` 속성을 변경하지 않도록 하십시오. 그러면 배칭에서 동작하지 않는 고유한 자료가 새로 생성됩니다. 그 대신, `MeshFilter` 구성요소를 통해 메시 데이터에 액세스하고, `Mesh.uv` 속성을 사용하여 정점별 좌표를 변경합니다.

다음 그림은 텍스처 아틀라스입니다.

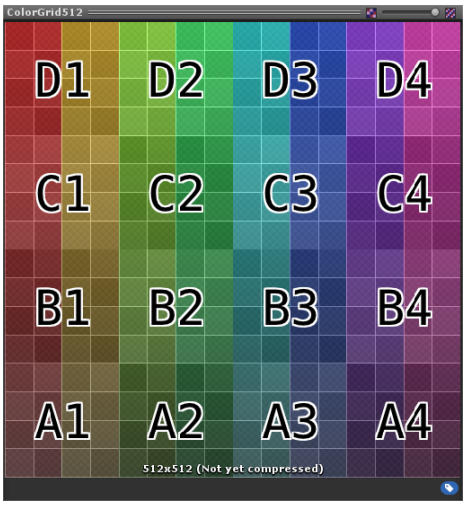


그림 4-22 텍스처 아틀라스

## 4.4 Mali™ 오프라인 쉐이더 컴파일러를 사용한 최적화

Mali 오프라인 쉐이더 컴파일러는 정점, 조각 및 연산 쉐이더를 이진 형식으로 컴파일하기 위한 도구입니다. 이 컴파일러를 프로파일링 도구로 사용할 수도 있습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 4.4.1 Mali™ 오프라인 쉐이더 컴파일러 개요

응용 프로그램 내 쉐이더는 GPU에서 실행됩니다. 이를 위해 GPU가 정점 위치, 픽셀 색상 등 쉐이더의 최종 결과를 연산하는 데 시간을 소비해야 합니다.

Mali 오프라인 쉐이더 컴파일러는 Mali GPU의 각 파이프라인에서 쉐이더가 실행해야 하는 사이클 수에 대한 정보를 제공합니다.

생성된 사이클 값은 특정 GPU에 맞게 조정됩니다. 명령 행에 대한 옵션으로 GPU를 선택합니다. 응용 프로그램이 타겟으로 하는 디바이스 범위에 해당하는 GPU를 선택해야 합니다. 그래야 도구에서 제공하는 통계가 실제적이고, 일반적인 사용 사례 시나리오에 부합합니다.

### 4.4.2 Unity 쉐이더 측정

Unity 쉐이더는 프로그래밍 언어 *C for Graphics*(Cg)로 작성합니다. Cg는 C 프로그래밍 언어를 기반으로 GPU 프로그래밍에 보다 적합하게 수정한 언어입니다.

Unity는 빌드 프로세스 도중 Cg를 OpenGL, OpenGL ES 또는 DirectX로 변환합니다.

OpenGL ES 쉐이더 코드를 검색하는 방법

1. Unity에서 분석할 쉐이더를 선택합니다.
2. 응용 프로그램을 빌드할 사용자 지정 플랫폼으로 OpenGLES30 또는 OpenGLES20을 선택합니다.
3. Compile and show 버튼을 클릭합니다.

개발 환경에 결과가 표시됩니다.

———— 참고 ————

- Mali 오프라인 쉐이더 컴파일러는 OpenGL ES 쉐이더만 지원합니다.
- 빌드 플랫폼이 Android로 설정된 경우 Unity가 기본적으로 OpenGLES30 쉐이더를 빌드합니다.

————————————

정점 및 조각 세션은 일반적으로 `#ifdef VERTEX` 또는 `#ifdef FRAGMENT`에 의해 구분됩니다. `#pragma multi_compile <FEATURE_OFF> <FEATURE_ON>`과 같은 옵션을 사용할 경우 파일에 여러 쉐이더 변형이 빌드됩니다.

일반적으로 여러 `VERTEX` 및 `FRAGMENT` 섹션이 있습니다. 각 변형은 Unity가 응용 프로그램을 시작할 때 개별적으로 컴파일됩니다. 특정 기능을 활성화하면 관련 변형이 선택됩니다.

코드가 OpenGL ES로 변환되었기 때문에 정점 및 조각 쉐이더 코드를 2개의 파일로 복사하여 Mali 오프라인 쉐이더 컴파일러를 사용하여 각각 컴파일할 수 있습니다.

다음 옵션 중 하나를 사용하여 쉐이더를 컴파일합니다.

- 정점 쉐이더의 경우 `-v`.
- 조각 쉐이더의 경우 `-f`.

다음 그림은 Mali 오프라인 쉐이더 컴파일러의 출력입니다.

```
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.
```

그림 4-23 Mali 오프라인 쉐이더 컴파일러의 출력

### 4.4.3 통계 분석

Mali 오프라인 쉐이더 컴파일러가 생산하는 통계는 쉐이더가 정점 또는 조각당 몇 번의 사이클을 필요로 하는지에 대한 측정치입니다.

결과는 세 행으로 구분됩니다.

- 총계.
- 최단 경로.
- 최장 경로.

최단 및 최장 경로는 코드에서 분기를 취한 또는 취하지 않은 결과를 보고 측정됩니다. 이는 가장 작은 또는 큰 실행 사이클 수에 대한 추정치를 제공합니다.

산술의 경우, 첫 번째 행의 측정치를 산술 파이프라인 수로 나눕니다. 이 수는 Mali GPU에 따라 1, 2 또는 3입니다.

두 번째 및 세 번째 행은 로드/저장 및 텍스처 파이프라인을 위한 것입니다. 여기에는 캐시 미스가 고려되지 않으므로 보다 현실적인 추정치를 구하려면 각 값에 1.5를 곱하는 것이 가장 좋습니다.

### 4.4.4 Mali™ GPU 파이프라인에 대한 최적화

Mali 오프라인 쉐이더 컴파일러는 각 파이프라인에서 사용되는 사이클 수를 제공합니다. 쉐이더는 파이프라인에서 가장 느리고 사이클 수가 가장 많습니다. 쉐이더가 가장 느린 파이프라인을 타겟으로 한 최적화를 사용하여 쉐이더를 최적화하십시오.

Mali GPU는 세 유형의 처리 파이프라인을 포함하고 있습니다.

- 산술 파이프라인.
- 로드/저장 파이프라인.
- 텍스처 파이프라인.

파이프라인은 모두 병렬로 실행됩니다. 일반적으로 쉐이더는 세 유형의 파이프라인을 모두 사용합니다.

#### 산술 파이프라인

모든 산술 연산은 산술 파이프라인에서 사이클을 소비합니다.

다음은 산술 파이프라인 사용을 줄일 수 있는 방법들입니다.

- 다음과 같이 복잡한 산술 연산을 사용하지 않습니다.
  — 역 매트릭스 함수.
  — 모듈로 연산자.
  — 나눗셈.
  — 행렬식.
  — 사인.
  — 코사인.
- 정수 피연산자의 경우 시프트와 같은 연산을 사용하여 나눗셈, 모듈로 및 곱셈을 계산합니다.
- 직교 매트릭스에 대해 역 대신 전치를 사용합니다.
- 전치 연산을 피하기 위해, 매트릭스-벡터 또는 매트릭스-매트릭스 곱셈에서 매트릭스 중 하나가 전치되는 경우 피연산자의 순서를 변경합니다. 예를 들면 다음과 같습니다.

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

부하를 다른 파이프라인으로 이동하여 산술 파이프에 대한 부하를 줄일 수도 있습니다.

- 매트릭스를 연산하는 대신 uniform 변수로 전달합니다. 그러면 로드/저장 파이프라인이 사용됩니다.
- 텍스처를 사용하여 사인 또는 코사인과 같은 함수를 나타내는 사전 연산된 값 세트를 저장합니다. 그러면 부하가 텍스처 파이프라인으로 전환됩니다.

#### 로드/저장 파이프라인

로드/저장 파이프라인은 유니폼 버퍼 객체 또는 쉐이더 저장 버퍼 객체와 같은 쉐이더에서 uniform 변수를 읽고, varying 변수를 쓰고, 버퍼에 액세스하는 데 사용됩니다.

응용 프로그램이 로드/저장 파이프라인에 바운드된 경우 다음 기법을 사용해 보십시오.

- 쉐이더에서 데이터를 읽을 때 버퍼 객체 대신 텍스처를 사용합니다.
- 산술 연산을 사용하여 데이터를 연산합니다.
- uniform 및 varying 변수를 압축하거나 줄입니다.

#### 텍스처 파이프라인

텍스처 액세스는 텍스처 파이프라인에서 사이클을 소비하며 메모리 대역폭을 사용합니다. 대규모 텍스처를 사용하면 성능에 악영향을 미칠 수 있습니다. 캐시 미스 가능성이 높아지고 데이터를 대기하느라 여러 스레드가 중단될 수 있기 때문입니다.

텍스처 파이프라인의 성능을 개선하려면 다음을 시도해 보십시오.

밉맵을 사용
: 밉맵은 텍스처 좌표의 변동을 기준으로 사용할 텍스처의 최상 해상도를 선택하기 때문에 캐시 히트 비율이 증가합니다.

텍스처 압축을 사용
: 이 기법도 메모리 대역폭을 줄이고 캐시 히트 비율을 높이는 데 유용합니다. 압축된 각 블록에는 복수의 텍셀이 포함되므로 블록에 액세스할 때 캐시 히트 비율이 높아집니다.

삼선형 또는 이방성 필터링을 사용하지 않음
: 삼선형 및 이방성 필터링은 텍셀을 가져와야 하는 연산 수를 증가시킵니다. 절대적으로 필요하지 않으면 이들 기법을 사용하지 마십시오.

### 4.4.5 파이프라인 사이클을 감소하기 위한 추가 기법

각 파이프라인에서 사용되는 사이클을 줄이기 위해 사용할 있는 추가 기법은 여러 가지가 있습니다.

#### 레지스터 스필링(spilling)을 방지합니다.

Mali 오프라인 쉐이더 컴파일러는 쉐이더에서 레지스터 스필링이 발생하는지 알려줍니다. 레지스터 스필링은 일반적으로 변수가 너무 많아 레지스터 세트에 완전히 들어갈 수 없는 경우 스레드에서 발생합니다.

레지스터 스필링은 일반적으로 다음 항목이 많은 경우 스레드에서 발생합니다.

- 입력 uniform.
- varying.
- 임시 변수.

또한 변수 정밀도가 높은 경우에도 레지스터 스필링이 발생할 수 있습니다.

레지스터 스필링이 발생할 경우 Mali GPU가 메모리에서 일부 uniform 변수를 읽어 들이므로 로드/저장 유닛에 대한 부하가 증가하고 성능이 저하됩니다. 이 문제를 해결하려면 쉐이더에 제공하는 uniform 변수의 개수 및 정밀도를 낮춰 보십시오.

얼음 동굴 데모에서는 일부 쉐이더에서 레지스터 스필링이 발생했습니다. 예:

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

그림 4-24 레지스터 스필링이 발생한 쉐이더.

허용된 uniform 변수 개수를 줄이는 것으로 이 문제가 해결되었으며, 그 결과 성능이 향상되었습니다. 예:

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

그림 4-25 레지스터 스필링이 없는 쉐이더.

### varying 및 uniform 변수의 정밀도를 감소

사용자 지정 쉐이더를 작성할 때 32비트 float 또는 16비트 half-float를 사용하여 uniform 및 varying 변수의 부동 소수점 정밀도를 지정할 수 있습니다. 정밀도는 변수가 나타낼 수 있는 값의 최소/최대 범위 및 단위를 결정합니다.

half-float를 사용할 경우 여러 장점이 있습니다.

- 대역폭 사용이 감소합니다.
- 산술 파이프라인에서 사용되는 사이클이 감소합니다. 쉐이더 컴파일러가 더 많은 병렬 연산이 가능하도록 코드를 최적화할 수 있기 때문입니다.
- 필요한 uniform 레지스터 수가 감소하며, 따라서 레지스터 스필링의 위험이 감소합니다.

다음 코드는 얼음 동굴 데모에 포함되어 있는 간단한 조각 쉐이더 변형의 예입니다. 쉐이더는 Mali 오프라인 쉐이더 컴파일러를 사용하여 두 번 컴파일됩니다.

첫 번째 코드 예제는 float를 사용하여 컴파일되었습니다.

```
$ 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.
```

그림 4-26 Float를 사용하여 컴파일된 쉐이더

두 번째 코드 예제는 half-float를 사용하여 컴파일되었습니다.

```
$ 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.
```

그림 4-27 half-float를 사용하여 컴파일된 쉐이더

half-float 버전에서는 로드/저장 명령어 수가 감소합니다. 사용되는 작업 및 uniform 레지스터의 수가 감소하고 레지스터 스필링이 발생하지 않습니다.

half-float를 사용하여 생성된 코드는 float를 사용하여 생성된 코드보다 크기도 작습니다. 이는 Mali GPU에서 캐시 히트 비율을 개선하여 성능이 향상됩니다.

### 정적 객체에 대해 월드 공간 노말 맵을 사용

탄젠트 공간 노말 맵을 사용하여 지오메트리 디테일은 증가시키지 않고 모델 디테일을 높일 수 있습니다. 애니메이션화된 객체는 메시의 각 삼각형에 대한 지역성(locality) 덕분에 객체를 수정하지 않아도 탄젠트 공간 노말 맵을 사용할 수 있습니다.

아쉽게도 이를 위해 쉐이더에서 더 많은 산술 연산을 수행해야 올바른 결과를 얻을 수 있습니다. 정적 객체의 경우 일반적으로 이러한 계산이 불필요합니다.

대신, 로컬 공간 노말 맵 또는 월드 공간 노말 맵을 사용할 수 있습니다. 로컬 공간 노말 맵을 사용하면 쉐이더에서 수행하는 계산이 감소하지만 모델에서의 변환이 샘플링된 노말에 적용되어야 합니다. 월드 공간 노말 맵은 어떤 변환도 필요하지 않지만 이들은 정적이며 객체가 이동할 수 없습니다. 얼음 동굴 데모에서는 동굴 및 기타 고품질 객체가 정적이며 월드 공간 노말 맵을 사용하여 쉐이더에 필요한 ALU 연산을 크게 줄였습니다. 대부분의 일반적인 3D 모델링 도구는 월드 공간 노말 맵을 생성할 수 있으며, 사용자가 오프라인 프로세스를 통해 코드로 생성할 수도 있습니다.

# 제 5 장
# 실시간 3D 아트 모범 사례: 지오메트리

이 장은 3D 애셋의 주요 지오메트리 최적화를 중심으로 설명합니다. 지오메트리 최적화는 효율적인 게임을 만들고 모바일 플랫폼에서 게임 성능 목표를 달성하는 데 도움이 됩니다.

——— 주의 ———

이 콘텐츠의 최신 버전은 Arm 개발자 웹 사이트 *https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/real-time-3d-art-best-practices-geometry*에서 확인할 수 있습니다.

———————

이 장은 다음 단원으로 구성되어 있습니다.

## 5.1 지오메트리란 무엇인가

지오메트리(또는 폴리곤메시)는 3D 오브젝트 모양을 구성하는 정점, 에지, 면의 모음입니다. 이때 3D 오브젝트는 차량, 무기, 환경, 캐릭터 등 비디오 게임에서 사용되는 모든 애셋이 될 수 있습니다.

3D 오브젝트의 지오메트리를 구성하는 3가지 요소는 다음과 같습니다.

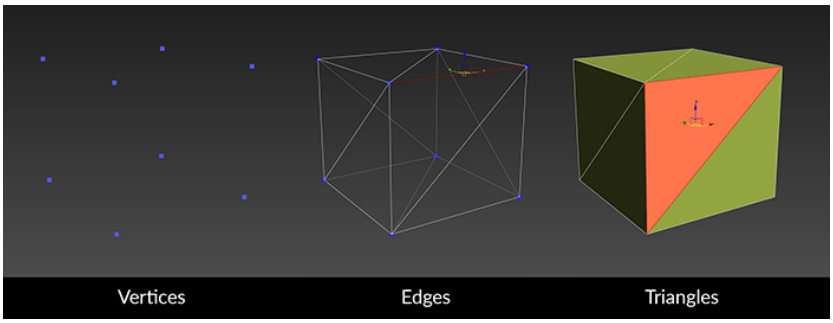


그림 5-1 3D 오브젝트의 지오메트리를 구성하는 3가지 요소

정점

정점은 3D 오브젝트의 표면을 구성하는 점입니다.

에지

에지는 두 정점이 일직선으로 연결될 때 발생합니다.

삼각형

삼각형은 3개의 에지와 연결되는 3개의 정점으로 구성되며 폴리곤이나 면이라고도 합니다. Max, Maya, Blender 같은 3D 소프트웨어에서는 주로 쿼드라는 4면 폴리곤을 사용합니다. 쿼드는 수정이 편리하며 이런 3D 프로그램 내에서 작업이 가능합니다. 화면에서 렌더링될 때 이 폴리곤들은 삼각형으로 표시됩니다.

## 5.2 삼각형과 폴리곤 사용

게임 성능을 최적화하려면, 언제나 화면에 표시되는 삼각형 수를 세야한다.

3D 오브젝트나 모델에서 원하는 품질을 내면서 일정한 성능을 낼 수 있도록 적절히 균형 잡힌 최소한의 삼각형을 사용하는 게 중요합니다.

우리는 다음 팁을 시도하기를 권장합니다.

- 적은 삼각형을 사용하면 성능이 향상됩니다.
  — 모바일 플랫폼에서 콘텐츠를 생산할 때는 삼각형 수를 유지하는 게 중요합니다.
  — 삼각형이 적으면 GPU가 정점을 적게 처리합니다.
  — 정점 처리는 컴퓨팅 비용이 높은 프로세스입니다. 따라서 처리되는 정점이 적으므로 전체 성능이 좋습니다.
- 삼각형을 적게 사용하면 게임을 더 많은 장치에 릴리스할 수 있습니다. 가장 강력한 GPU를 사용한 최신 장치가 아니어도 됩니다.

다음 이미지는 2개의 3D 오브젝트를 비교한 것입니다. 하나는 584개의 삼각형을 사용했고 다른 하나는 704개의 삼각형을 사용했습니다. 두 오브젝트는 쉐이드 모드에서 동일하게 보입니다. 에지를 제거해도 실루엣에 변화가 없습니다.

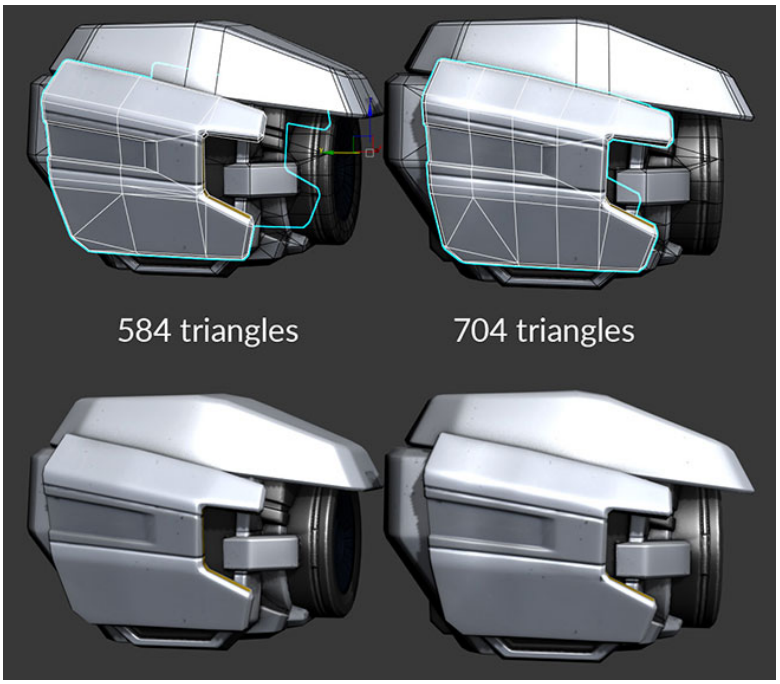


그림 5-2 원하는 결과를 얻기 위해 가장 적은 수의 삼각형 사용

모바일 장치에서 3D 오브젝트나 메시에 사용할 수 있는 최대 정점 수는 65,535입니다. 따라서 이 숫자보다 낮은 수를 사용하십시오. Mali-400을 사용하는 Android 장치 등 오래된 GPU는 그 이상을 지원하지 않습니다. 이런 장치는 더 많은 정점이 있는 3D 오브젝트를 렌더링하지 않습니다.

게임을 타겟 장치 또는 가능한 많은 타켓 장치에서 보거나 테스트하는 것이 중요합니다. 컴퓨터 화면에서 게임을 테스팅하는 것은 최적화에 필요한 정보를 제공하지 않습니다.

모바일 장치 화면은 평균적인 컴퓨터 모니터보다 작다는 것을 기억하십시오. 따라서 많은 삼각형을 사용해 생성한 디테일은 모바일 장치에서 보이지 않을 수도 있습니다.

카메라와 가까운 전경에 위치한 3D 오브젝트는 보다 많은 삼각형을 사용하고, 멀리 떨어져 배경쪽에 있는 3D 오브젝트는 보다 적은 삼각형을 사용하십시오. 이 테크닉은 정적 카메라 *시점*(Point-of-View, POV)을 사용하는 게임일 경우 더 큰 도움이 됩니다.

다음 이미지는 3D 모델이 전경에서 사용되고, 저품질 3D 모델이 2D 배경으로 사용된 예를 보여줍니다.

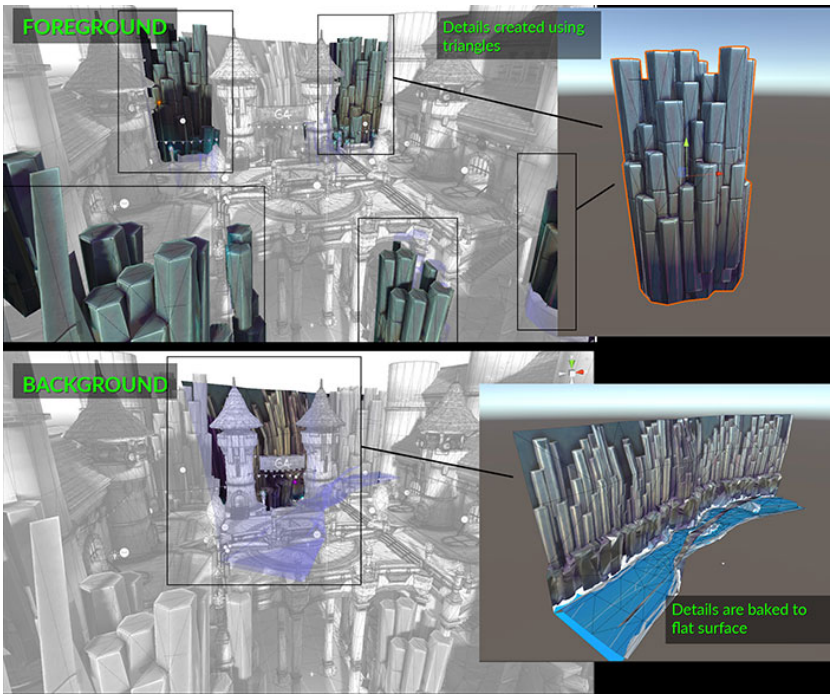


그림 5-3 전경과 배경 오브젝트의 삼각형 사용 예

3D 오브젝트에서 사용할 수 있는 최대 삼각형 수는 고정되어 있지 않지만 화면에 다수의 3D 오브젝트가 동시에 출현하면 오브젝트당 더 적은 삼각형을 사용할 수 있습니다. 그러나 화면에 3D 오브젝트를 적게 표시한다면 더 많은 삼각형을 사용할 수 있습니다.

대상 장치도 중요합니다. 최신 Samsung Galaxy S 시리즈 등 최신 휴대 전화는 오래된 모바일 장치보다 더 복잡한 지오메트리를 처리할 수 있습니다.

다음 예는 두 데모의 캐릭터를 보여줍니다. Circuit VR 데모에는 로봇 캐릭터가 단 하나 등장하므로 폴리곤 수가 많은 모델을 사용할 수 있습니다. Armies 데모는 한 프레임에 수백 명의 군인이 출현하므로 적은 수의 폴리곤이 필요합니다.

그림 5-4 삼각형 수 차이 예

Armies 데모는 Unity에서 빌드한 64비트 모바일 장치 기술 데모로 카메라는 정적이며 움직이는 캐릭터가 많습니다. 각 프레임에 렌더링된 삼각형은 전부 합해 약 210,000개로, 이 삼각형 수는 데모를 30FPS *(Frame Per Second)* 수준으로 매끄럽게 실행시킬 수 있습니다.

———— 참고 ————

사용되는 삼각형 수는 제작하는 게임 유형이나 대상 장치 사양에 따라 달라집니다.

————————————

다음 예는 우리의 기술 데모에 사용된 전체 삼각형 수를 보여줍니다.


그림 5-5 우리의 예제 장면에서 사용된 전체 삼각형 수

장면에서 가장 큰 오브젝트인 포탄 탑은 개당 약 3000개의 삼각형으로 구성되어 있으며 화면에서 큰 비중을 차지합니다.

캐릭터는 약 360개의 삼각형만 사용했습니다. 캐릭터 수가 많고 멀리 있기 때문입니다. 카메라 시점에서는 화면에 캐릭터가 잘 표시됩니다.

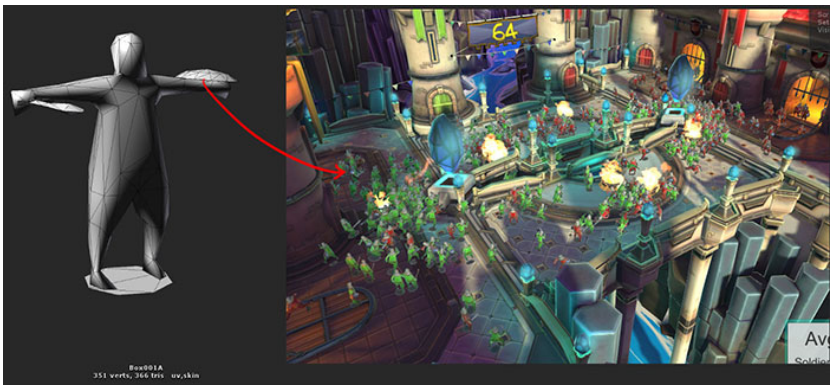


그림 5-6 군인의 적은 삼각형 수

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *5.2.1 관련 영역에 삼각형 분배* 페이지의 5-79.
- *5.2.2 마이크로 삼각형이 나쁜 이유* 페이지의 5-80.
- *5.2.3 길거나 얇은 삼각형이 나쁜 이유* 페이지의 5-82.

### 5.2.1 관련 영역에 삼각형 분배

모바일 플랫폼에서 폴리곤과 정점은 컴퓨팅 비용이 높습니다. 게임의 시각 품질에 확실히 공헌하는 영역에 폴리곤과 정점을 배치하면 처리 비용을 낭비하지 않습니다.

게임 레벨에 3D 오브젝트가 있는 대부분의 장치는 화면 크기가 작으므로 3D 오브젝트의 작은 삼각형 디테일은 게임 내에서 보이지 않습니다. 대신 오브젝트의 디테일보다 실루엣을 결정하는 큰 모양이나 부분에 집중합니다.

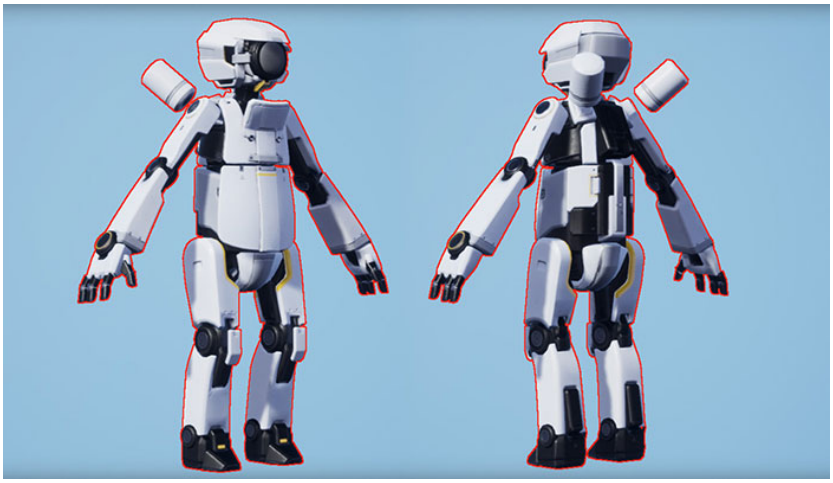

그림 5-7 큰 모양과 실루엣에 집중

화면에 잘 보이지 않는 영역에는 삼각형을 적게 사용합니다. 차량의 바닥이나 옷장의 뒤가 이런 예시입니다.

작은 디테일을 모델링할 때 고밀도 삼각형 메시를 사용하지 않습니다. 대신 섬세한 디테일에는 텍스처와 노말 맵을 사용합니다.

———— 참고 ————

노말 맵은 각 픽셀의 표면 방향을 저장하는 텍스처 맵입니다.

————————————

다음 그림은 동일한 메시에서 노말 맵이 있는 경우와 없는 경우를 보여줍니다.

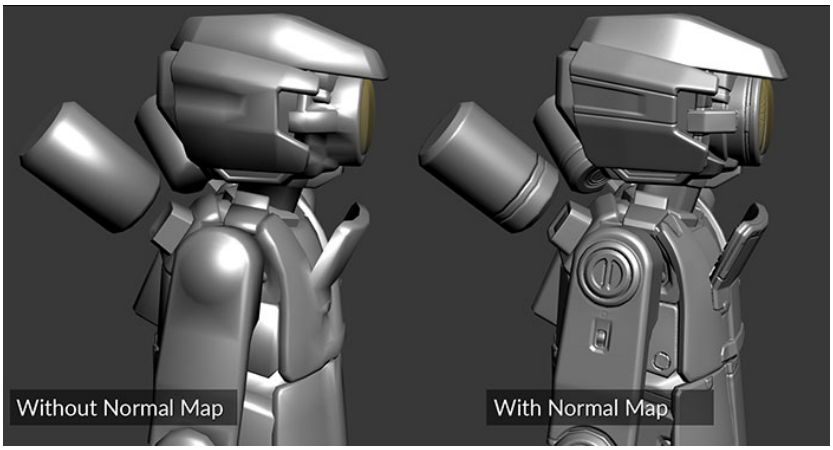


그림 5-8 작은 디테일에는 노말 맵과 텍스처를 사용합니다

카메라 시점에서 절대 보이지 않는 오브젝트의 뒷면, 바닥, 일부에 대한 삭제를 고려하십시오.

그러나 이렇게 하면 해당 장면의 재사용 가능성을 제한할 수 있으므로 신중하게 적용해야 합니다. 예를 들어 최종 사용자에게 절대 보이지 않는다고 생각하고 테이블 메시의 바닥을 삭제합니다. 이렇게 하면 해당 모델을 뒤집어 배치하거나 다른 용도로 재사용할 수 없습니다.

### 5.2.2 마이크로 삼각형이 나쁜 이유

마이크로 삼각형은 오브젝트나 장면의 최종 표현에 크게 영향을 주지 않는 작은 삼각형입니다.

폴리곤 수가 많은 3D 오브젝트가 카메라에서 멀어지면 마이크로 삼각형 문제가 발생합니다. 마이크로 삼각형은 장치에서 1-10픽셀 사이의 삼각형을 말합니다.

마이크로 삼각형이 좋지 않은 이유는 GPU가 모든 삼각형을 처리해야 하기 때문입니다. 하지만 보이지 않을 정도로 작으면 최종 이미지에 영향을 끼치지도 않습니다. 정점은 처리하는 컴퓨팅 비용이 높으며, 화면에 한 번에 표시되는 작은 삼각형이 많으면 처리해야 할 정점도 늘어난다는 것을 기억하십시오.

다음 두 방식은 마이크로 삼각형의 원인이 됩니다.
- 디테일이 너무 작거나 많은 삼각형으로 구성된 경우.
- 카메라와 거리가 먼 삼각형이 많은 오브젝트.

다음 그림은 카메라에서 가깝고 멀 때 3D 오브젝트의 삼각형 수를 보여줍니다. 왼쪽은 적은 삼각형을 사용했고 오른쪽은 동일한 시각적 디테일을 표시하는 대신 노말 맵을 사용했습니다.

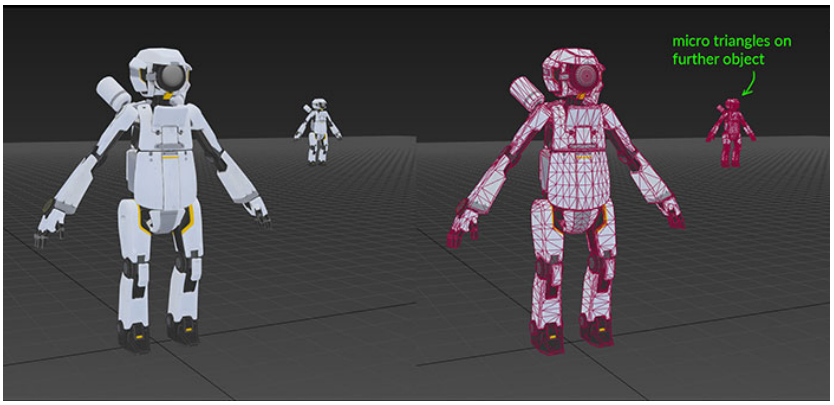


그림 5-9 카메라와 먼 오브젝트의 마이크로 삼각형

다음 이미지에서 강조된 부분의 삼각형 대부분은 너무 작아서 모바일 장치에서 보이지 않습니다. 따라서 최종 표현에 별로 영향을 주지 않습니다.

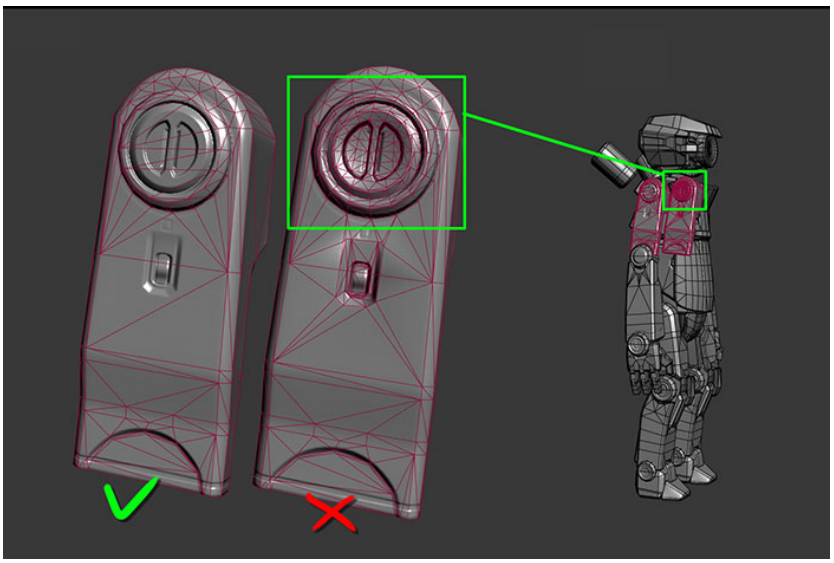

그림 5-10 가까운 마이크로 삼각형

### 이 문제를 최소화하는 방법

이 문제를 완화하는 방법은 몇 가지가 있습니다.
- 카메라와 거리가 변경되는 오브젝트의 경우 *LOD*(Level of Detail)를 사용합니다. 올바른 LOD를 사용하면 거리가 먼 오브젝트를 단순화하여 삼각형을 적게 사용합니다.
- 배경 오브젝트에 적은 수의 삼각형을 사용합니다.

- 모드에서 섬세한 디테일을 표현할 때 폴리곤을 사용하지 않습니다. 대신 텍스처와 노말 맵을 조합해 표현합니다.
- 화면에 보이지 않을 정도로 너무 작거나 최종 이미지에 큰 영향을 주지 않는 정점이나 삼각형은 병합합니다.
- 영역에서 삼각형을 10픽셀 이상으로 유지하십시오.

#### 마이크로 삼각형의 사용을 줄이는 것이 중요한 이유

마이크로 삼각형의 사용을 줄이는 것이 중요한 이유는 여러 가지가 있습니다. 이는 다음과 같습니다.

- GPU는 모든 삼각형과 정점을 처리해야 합니다. 화면에 표시되는 최종 장면에 아무 값을 더하지 않는 경우에도 마찬가지입니다.
- GPU가 처리할 수 있도록 더 많은 데이터를 보내야 하기 때문에 메모리 대역폭이 부정적인 영향을 받습니다.
- 필요한 처리량은 모바일 장치의 배터리 성능에 직접 영향을 줍니다. 따라서 GPU가 적은 데이터를 처리할수록 배터리 수명이 길어집니다.

### 5.2.3 길거나 얇은 삼각형이 나쁜 이유

길거나 얇은 삼각형은 최종 이미지에 렌더링할때 10픽셀 보다 작으면서 정점들이 화면을 따라 펼쳐져 있습니다. 길거나 얇은 삼각형은 일반 삼각형보다 GPU에서 처리하는 비용이 비싸기 때문에 좋지 않습니다.

다음 스크린샷은 길거나 얇은 삼각형이 사용된 예를 보여줍니다. 이 스크린샷은 멀리서 보았을 때 기둥의 경사를 강조합니다. 그러나 가까이에서 보는 경우 경사는 문제가 되지 않습니다.

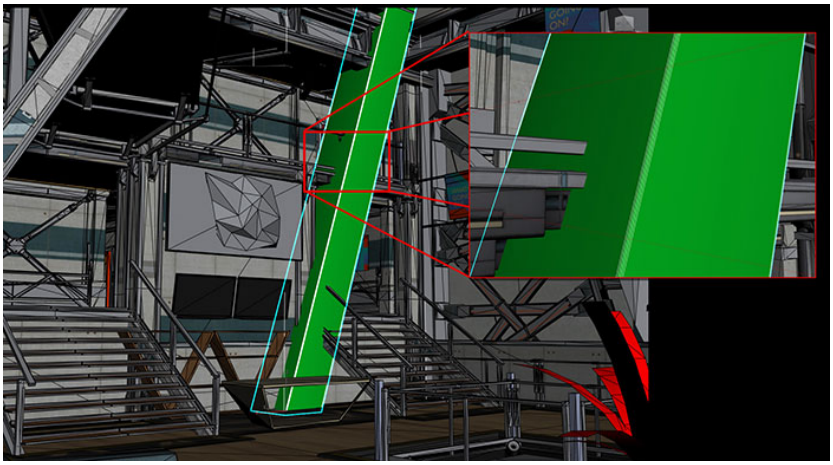

그림 5-11 길고 얇은 삼각형 예

#### 이 문제를 최소화하는 방법

이 문제를 완화하는 방법은 몇 가지가 있습니다.

- 가능한 경우 모든 오브젝트에서 당신에게 보이는 길거나 얇은 삼각형을 제거합니다. 불가능한 경우에는 길거나 얇은 삼각형을 완전히 제거하는 것이 좋습니다.
- 길거나 얇은 삼각형이 있는 오브젝트는 플리커링이 발생할 수 있으므로 빛나는 머티리얼을 사용하지 마십시오.
- *LOD*를 사용하고 화면에서 멀어지는 경우 길거나 얇은 삼각형을 제거하십시오.
- 기술적으로 삼각형을 정삼각형이 되도록 하는 것이 좋습니다. 에지보다 내부에 더 많은 삼각형을 사용하십시오.
- 이 문제에 대한 더욱 자세한 기술적 설명은 *http://www.humus.name/index.php?page=News&ID=228*을 참조하십시오.

## 5.3 Level of Detail(LOD)

*LOD*는 뷰어에서 오브젝트가 멀어질수록 메시 복잡성을 줄이는 테크닉입니다.

LOD는 처리해야 할 정점 수를 줄입니다. 또한 LOD는 마이크로 삼각형 문제를 피하고 보통은 장면에서 멀리 배치된 오브젝트의 시각 품질을 더 좋게 합니다. 따라서 우리는 카메라와의 거리가 크게 변화하는 모든 3D 오브젝트의 LOD를 최적화할 것을 권장합니다.

다음 그림은 LOD 관리를 활용해 3D 모델의 복잡성을 줄이면서 모델이 카메라에서 멀어져도 적절한 디테일 레벨을 유지하는 방법을 보여줍니다.

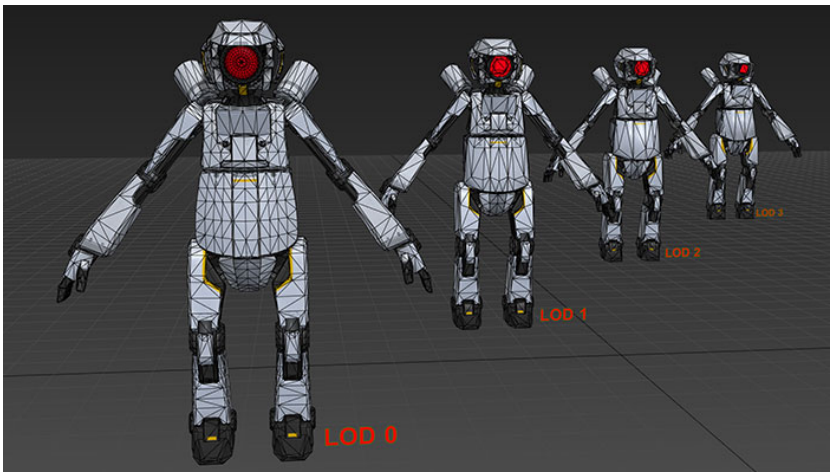


그림 5-12 4 레벨 LOD 예

LOD 사용을 최적화할 때 고려해야 할 몇 가지 조언은 다음과 같습니다.

- LOD의 삼각형 수를 줄일 때 삼각형이 오브젝트의 실루엣에 어떤 영향을 주는지 고려하십시오.
- LOD는 셰이더 복잡성에도 적용할 수 있습니다. 멀어지는 3D 오브젝트에 대해 셰이더와 머티리얼을 최적화할 수 있습니다. 예를 들면 사용되는 텍스처 수를 줄이는 것입니다.
- 평평한 영역의 폴리곤을 더 많이 제거합니다.
- 텍스처의 LOD로 밉맵을 사용합니다.

### LOD를 사용하지 않아야 하는 경우

모든 상황에 LOD 사용이 권장되는 것은 아닙니다. 예를 들어 카메라 시점과 오브젝트가 고정된 게임에서는 LOD를 사용하지 마십시오. 이미 오브젝트의 폴리곤 수가 적은 경우에도 권장하지 않습니다.

LOD는 메모리 오버헤드를 일으킬 수 있으며 파일 사이즈도 큽니다. 모든 LOD 메시 데이터는 메모리에 저장되어야 하므로 실시간으로 활용할 수 있습니다.

다음 그림은 장면이 정적이라 LOD를 사용하지 않은 장면의 예를 보여줍니다. 대신 플레이어에게 절대 보이지 않는 폴리곤을 제거하는 등 다른 최적화 트릭을 사용할 수 있습니다.


그림 5-13 정적인 장면에서 LOD를 사용하지 않은 예

### 왜 LOD를 사용할까요?

오브젝트가 카메라에서 멀어지면 오브젝트의 디테일이 덜 보입니다. 거리가 20미터 떨어질 경우 삼각형이 200개인 오브젝트와 2000개인 오브젝트의 차이가 거의 보이지 않습니다. 따라서 장면에 값을 추가하지 않는 추가 삼각형을 많이 사용할 필요가 없습니다.

다른 핵심 이점은 다음과 같습니다.

- 처리해야 하는 삼각형이 적어서 성능이 향상됩니다.
- LOD는 마이크로 삼각형이 일으킬 수 있는 문제를 완화할 수 있습니다.

다음 그림은 폴리곤 수가 달라도 먼 오브젝트가 얼마나 동일하게 보이는지 보여줍니다.

그림 5-14 원거리 모델 LOD 예

### 각 LOD에 적절한 수의 삼각형을 사용

각 LOD에서 적절한 삼각형 수를 결정하는 데 중요한 포인트는 다음과 같습니다.

- 보편적으로 각 LOD 레벨마다 삼각형 수를 50%씩 줄이는 것이 좋습니다.
- 낮은 LOD에서 많은 수의 삼각형을 사용하지 마십시오. 낮은 LOD는 오브젝트가 멀어질 때만 보입니다.
- 카메라에서 보이는 정확한 거리에서 LOD가 어떻게 보이는지 확인하십시오. 낮은 LOD는 가까운 거리에서 시각 품질이 떨어지지만 그렇게 보일 일이 없다면 괜찮습니다.

다음 그림은 레벨마다 50%씩 줄어드는 폴리곤을 사용했을 때 오브젝트가 어떻게 보이는지 보여줍니다.

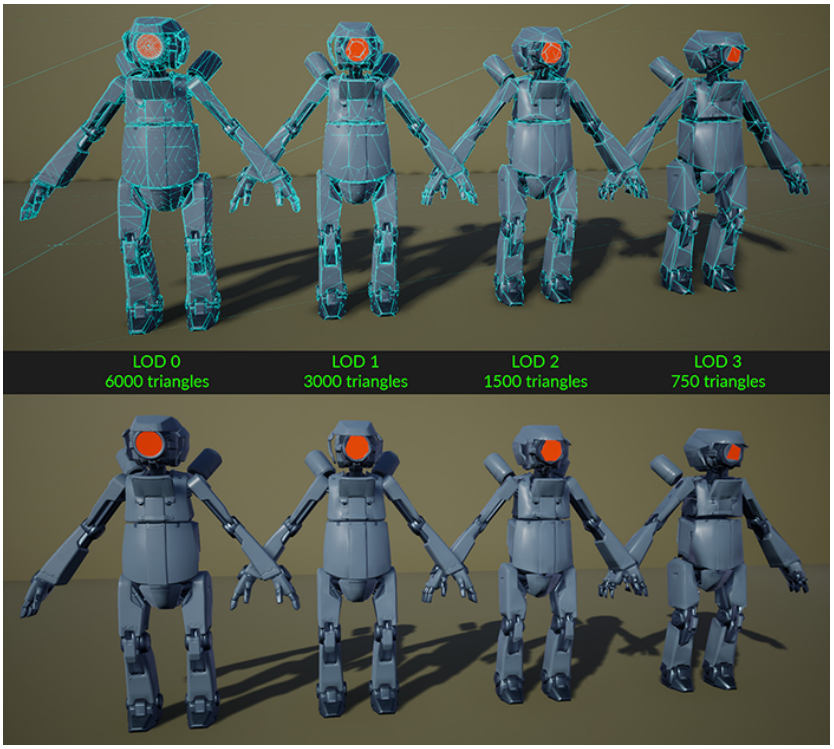


그림 5-15 폴리곤 수 LOD 예

다음 그림은 폴리곤 수가 적은 오브젝트가 화면에서 멀어질 때 어떻게 보이는지 예시를 보여줍니다. 또한 화면에서 너무 가까운 경우 낮은 LOD의 시각 품질이 얼마나 낮은지 강조합니다.

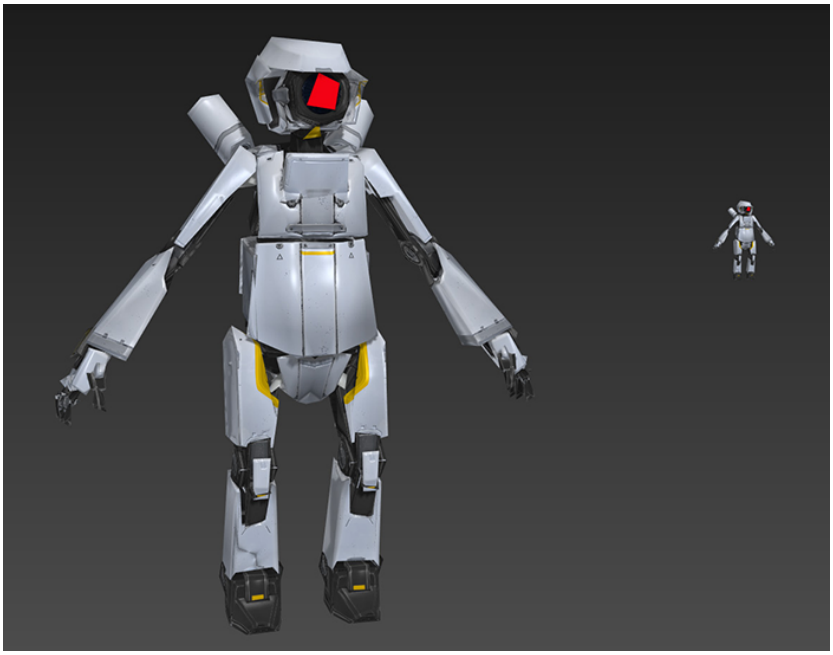

그림 5-16 LOD 거리 예

낮은 LOD 오브젝트에서 폴리곤 수를 충분히 줄이지 않으면 게임 성능에 부정적인 영향을 줄 수 있기 때문에 LOD 삼각형 수가 중요합니다. CPU가 필요 이상으로 정점을 처리하기 때문입니다.

낮은 LOD 오브젝트의 폴리곤 수를 너무 적극적으로 줄이는 경우 아이템 디테일이 실시간으로 나타났다 사라지는 것처럼 보입니다. 이런 등장 효과는 사용자가 눈치챌 수 있으므로 잠재적으로 게임의 몰입도를 해칩니다.

### 오브젝트에 사용할 적절한 LOD 레벨 수는 얼마입니까?

오브젝트가 소유할 수 있는 LOD 수는 정해져 있지 않습니다. LOD 수는 오브젝트의 크기와 중요도에 따라 결정됩니다. 예를 들어 액션 게임의 캐릭터나 레이싱 게임의 차량은 나무나 깡통 등 소형 배경 오브젝트가 있는 것보다 LOD 레벨을 많이 사용하는 것이 유리합니다.

LOD 레벨을 너무 적게 사용할 때의 효과는 다음과 같습니다.

- LOD 레벨 간 줄어드는 폴리곤 수가 충분하지 않으면 성능 저하가 발생합니다.
- 폴리곤 감소가 너무 크면 LOD 전환 시 발생하는 팝 효과가 더 잘 보입니다.

LOD 레벨을 너무 많이 사용할 때의 효과는 다음과 같습니다.

- 표시될 LOD를 결정하는 데 더 많은 처리가 필요하므로 CPU 워크로드가 증가합니다.
- 파일 크기가 큰 경우 추가 메시를 저장해야 하므로 메모리 사용량이 증가합니다.
- 그러나 이때 가장 큰 비용은 이 LOD 모델을 생성하고 검증하는 데 필요합니다. 특히 아티스트가 수동으로 이를 생성하는 경우에 그렇습니다.

### LOD 메시 생성 방법

3D 소프트웨어에서 LOD 메시를 수동으로 생성할 때 3D 오브젝트의 에지 루프를 제거하거나 정점의 수를 줄입니다. 아티스트가 더 많이 제어할 수 있지만 작업 시간은 더 많이 걸립니다.

LOD 메시를 자동으로 생성할 때 내장 한정자나 별도의 LOD 생성 소프트웨어를 사용합니다. 3DS Max의 내장 한정자라면 ProOptimizer 함수를 사용하고, Maya라면 LOD 메시 생성 함수를 사용합니다.

### Unity에서 LOD 구현

Unity에서 LOD를 구현하는 방법은 다음 링크 *Unity에서 LOD 구현* 페이지의 11-259을 클릭합니다.

## 5.4 추가 지오메트리 모범 사례

게임 성능을 최적화하는 데 도움이 되는 테크닉과 트릭이 몇 가지 더 있습니다.

### 스무딩 그룹

에지의 강도를 정의하거나 모델의 외양을 변경할 때 스무딩 그룹 또는 사용자 지정 정점 노말을 사용합니다. 스무딩 그룹은 아트 디렉션이 의도적으로 적은 수의 폴리곤을 사용할 때 더 나은 셰이딩을 생성하는 데 도움이 됩니다.

스무딩 그룹은 모델의 UV 아일랜드와 베이킹 시 노말 맵의 품질에도 영향을 주기 때문에 특히 주의해야 합니다. 관련 내용은 *텍스처링 모범 사례 장* 페이지의 6-90에 자세히 설명되어 있습니다.

3D 모델에 스무딩을 구현하려면 3D 소프트웨어에서 내보낸 뒤 엔진으로 가져와야 합니다.

다음 그림은 오브젝트에 스무딩이 적용되었을 때의 예를 보여줍니다.

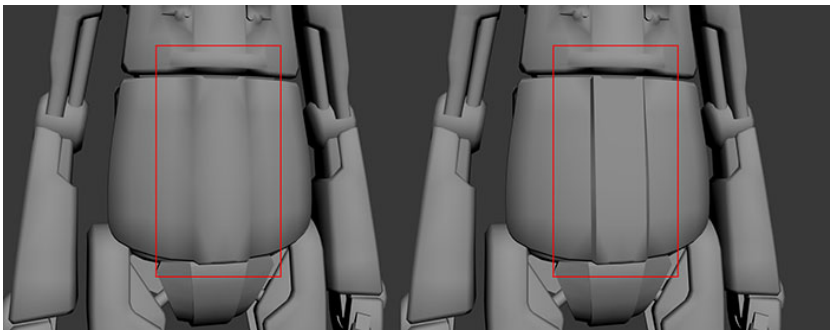

그림 5-17 스무딩 그룹 응용 예

### 메시 토폴로지

메시 토폴로지를 사용할 때 다음 팁을 기억하십시오.

- 3D 애셋을 생성할 때 비교적 깔끔하고 정돈된 토폴로지를 사용합니다.
- 깔끔한 토폴로지는 변형되거나 움직이는 캐릭터나 다른 오브젝트에 필수적입니다.
- 3D 모델에서 완벽한 토폴로지를 갖으려고 애쓰지 마십시오. 모든 오브젝트에 완벽한 에지 플로우가 필요하지 않으므로 모델을 깔끔하게 유지하도록 노력합니다.
  — 플레이어나 사용자는 3D 모델의 와이어프레임을 보지 않습니다.
  — 3D 모델의 외양에는 메시에 적용된 텍스처와 머티리얼이 더 큰 영향을 줍니다.

다음 그림은 간단한 지오메트리와 토폴로지를 사용한 바위 절벽 메시의 와이어프레임 예시를 보여줍니다. 적용된 머티리얼이 있을 경우 바위 절벽의 시각 품질이 더 좋습니다. 따라서 토폴로지 관련 문제는 사라집니다.

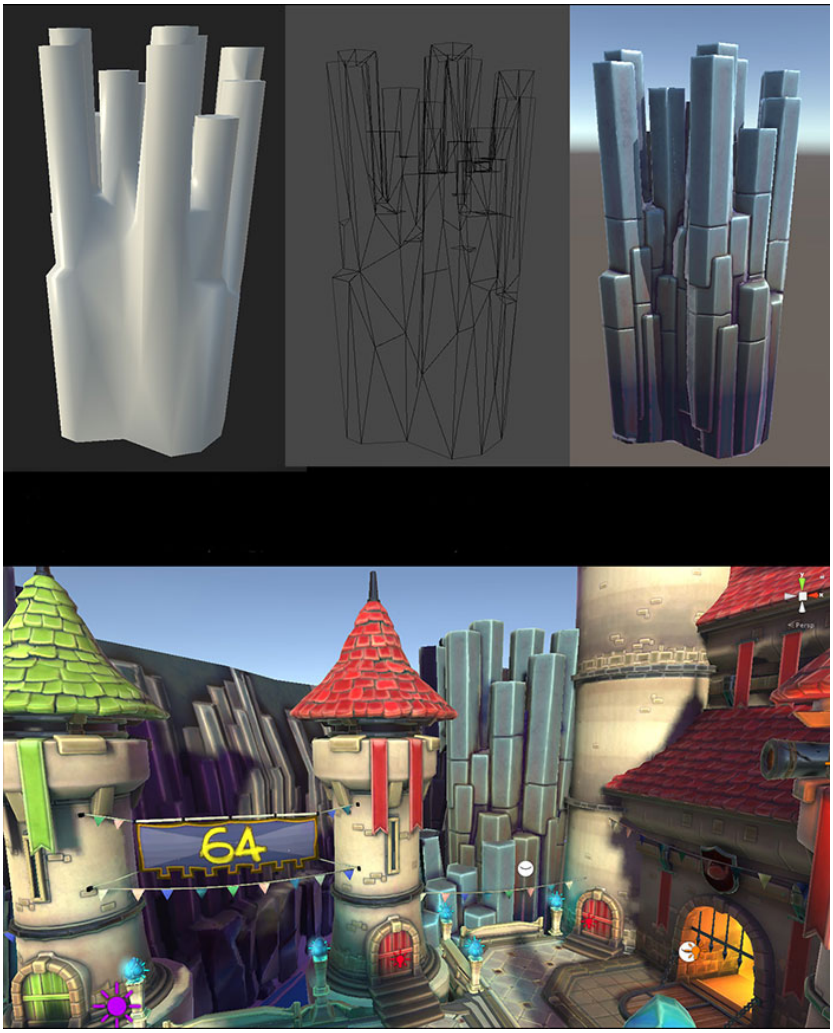


그림 5-18 메시 토폴로지 예

### 모양 과장

모양 과장은 특정 부분이나 모양을 보통보다 더 크게 하여 가독성을 높이는 테크닉입니다. 그러나 이 접근 방식이 게임에 적절한지는 제작하는 게임의 유형이나 스타일에 따라 달라집니다.

예를 들면 다음과 같습니다.

- 모바일 장치 화면은 작기 때문에 특정 모양이 너무 작은 경우 포착하기 어려울 때가 있습니다. 모양 과장은 이 문제를 해결하는 데 도움이 됩니다. 예를 들어 캐릭터의 손을 더 과장하면 작은 화면에서도 잘 보입니다.

다음 그림은 손의 크기를 확대하면 어떻게 보이는지 나타냅니다. 손, 칼, 일반 신체 비율은 다양한 방식으로 강조될 수 있습니다. 이는 사용된 폴리곤의 적은 수를 처리하며 가시성을 개선하는 데 사용됩니다.

그림 5-19 모양 과장 예

# 제 6 장
# 실시간 3D 아트 모범 사례: 텍스처링

이 장은 게임의 시각 품질을 높이고 부드럽게 실행되도록 돕는 다양한 텍스처 최적화를 다룹니다.

——— 주의 ———

이 콘텐츠의 최신 버전은 Arm 개발자 웹 사이트 *https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/real-time-3d-art-best-practices-texturing*에서 확인할 수 있습니다.

———

이 장은 다음 단원으로 구성되어 있습니다.

## 6.1 텍스처 아틀라스, 필터링, 밉맵 텍스처 아틀라스

텍스처 아틀라스는 하나로 패킹된 여러 작은 이미지의 데이터를 포함하는 하나의 이미지입니다. 메시당 텍스처를 별도로 사용하지 않고 여러 메시가 하나의 큰 텍스처를 공유할 수 있습니다.

텍스처 아틀라스는 애셋을 만들기 전에 생성할 수 있으며 텍스처 아틀라스에 따라 애셋은 UV 언래핑됩니다. 즉 텍스처를 생성할 때 조기 계획이 필요하다는 의미입니다.

텍스처 아틀라스는 페인팅 소프트웨어에서 텍스처가 병합되어 애셋이 완성된 뒤에도 만들 수 있습니다. 그러나 이 역시 텍스처에 따라 UV 아일랜드를 재조정해야 합니다.

———— 참고 ————

UV 아일랜드는 텍스처 맵에 있는 폴리곤 연결 그룹입니다.

————————————

다음 그림은 하나의 텍스처 세트를 사용하는 3D 오브젝트를 강조해 보여줍니다.

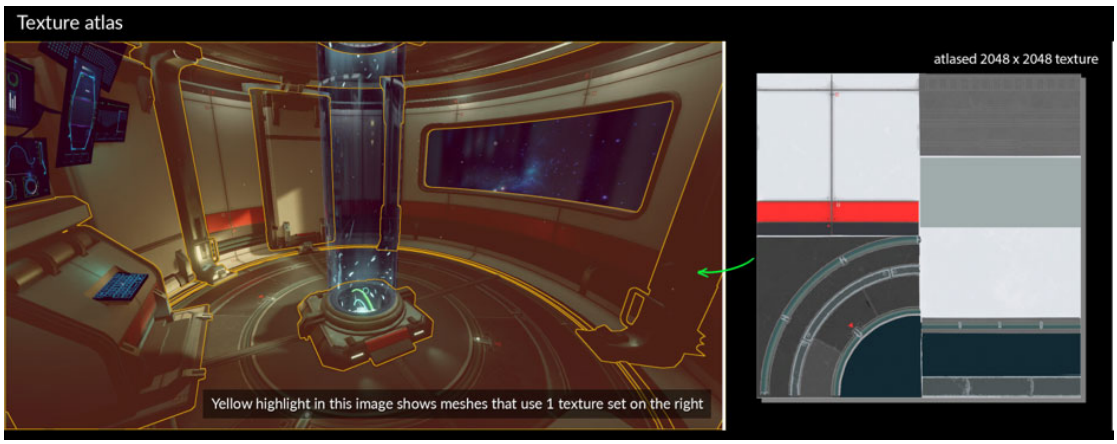


그림 6-1 텍스처 아틀라스 예

### 텍스처 아틀라스를 사용하는 이유

텍스처 아틀라스는 해당 텍스처 아틀라스와 동일한 머티리얼을 공유하는 여러 정적 오브젝트를 배칭할 수 있습니다. 배칭하면 드로우 콜 수를 줄이고, 드로우 콜이 감소하면 게임이 CPU의 처리량이 많을때 성능이 개선됩니다.

Unity 게임 엔진은 오브젝트가 정적인 경우 배칭하는 기능이 있습니다. 이 작업은 오브젝트를 수동으로 병합하지 않아도 가능합니다. 이에 대한 자세한 내용은 *Unity 웹 사이트를 참조하십시오*.

또한 게임 내에서 텍스처가 패킹되었기 때문에 텍스처 아틀라스는 텍스처가 적게 필요합니다. 압축하면 텍스처의 메모리 소비를 낮출 수 있습니다.

## 6.2 텍스처 필터링

텍스처 필터링은 장면에서 텍스처 품질을 개선하는 데 사용되는 방법입니다. 텍스처 필터링을 사용하지 않으면 일반적으로 에일리어싱 같은 아티팩트는 더 나빠 보입니다. Unity 텍스처 필터링에서 사용할 수 있는 옵션에는 여러 가지가 있습니다.

텍스처 필터링은 텍스처의 블록화를 줄여 품질을 높여줍니다. 이렇게 하면 일반적으로 게임 화면이 좋아집니다.

그러나 텍스처 필터링을 사용해 품질이 좋아진다는 것은 더 많은 처리가 필요하다는 뜻이므로 성능 저하가 발생할 수 있습니다. 텍스처 필터링은 GPU 에너지 소비량의 절반을 차지할 수 있습니다. 따라서 더 간단한 텍스처 필터를 선택할수록 응용 프로그램에서 필요한 에너지를 줄일 수 있습니다.

Nearest 또는 Point 필터링
: 근접 보기 시 Nearest 필터링은 텍스처를 블록 모양으로 보이게 합니다. 이것이 가장 간단하고 비용이 낮은 텍스처 필터링입니다.

Bilinear 필터링
: Bilinear 필터링을 사용하면 가까이에 있는 텍스처의 블러 현상이 심해집니다. 가장 가까운 네 개의 텍셀을 샘플링한 뒤 평균을 내어 메인 픽셀을 색칠합니다. Nearest 필터링과 달리 Bilinear 필터링은 픽셀이 부드럽게 그라데이션되므로 픽셀 블록화가 덜합니다.

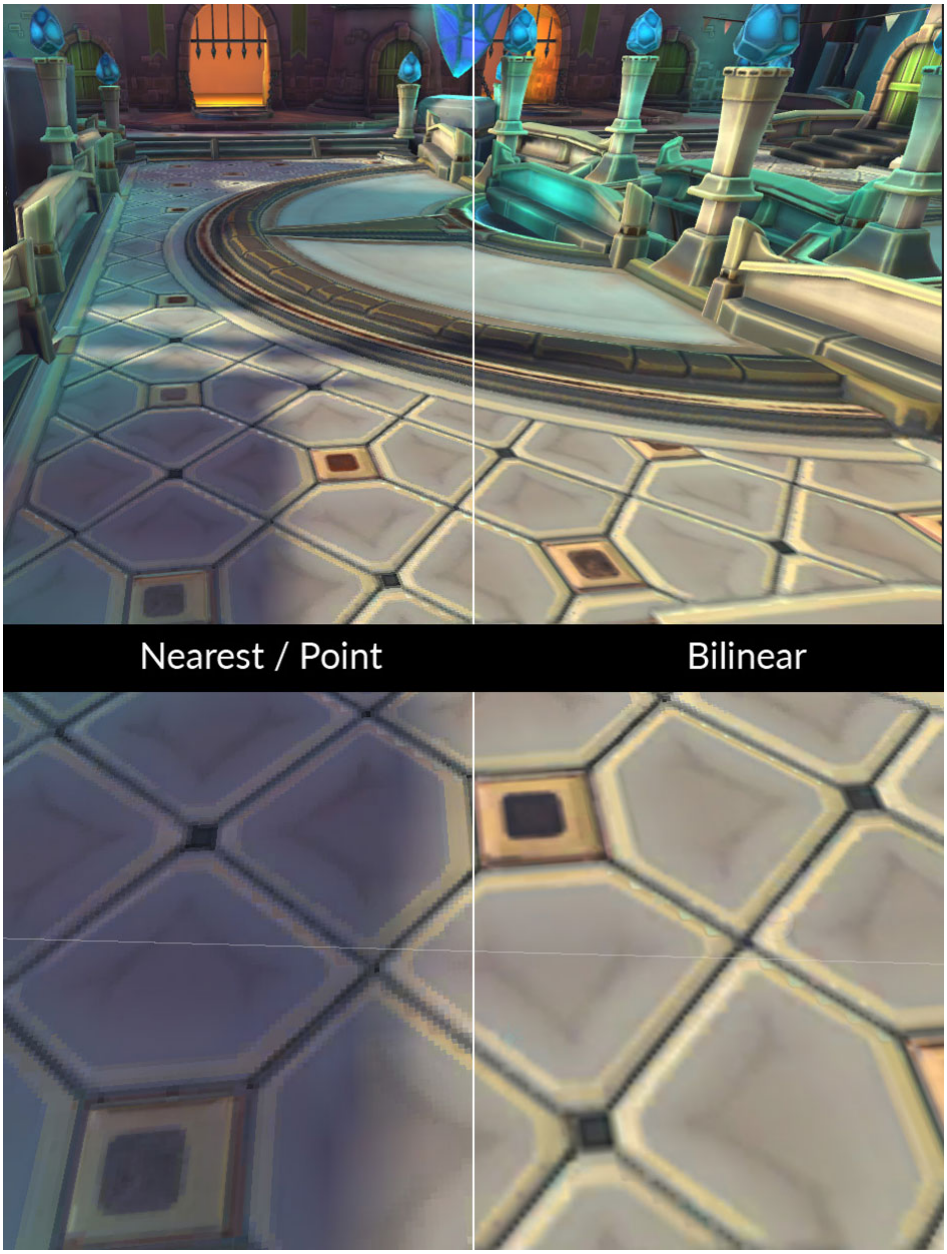


그림 6-2 근접 및 Bilinear 텍스처 필터링 예

Trilinear 필터링

Trilinear 필터링은 Bilinear 필터링과 유사하지만 밉맵 레벨 간 블렌딩이 추가됩니다. Trilinear 필터링은 밉맵을 부드럽게 전환하여 밉맵 간 전환이 눈에 띄지 않도록 합니다.

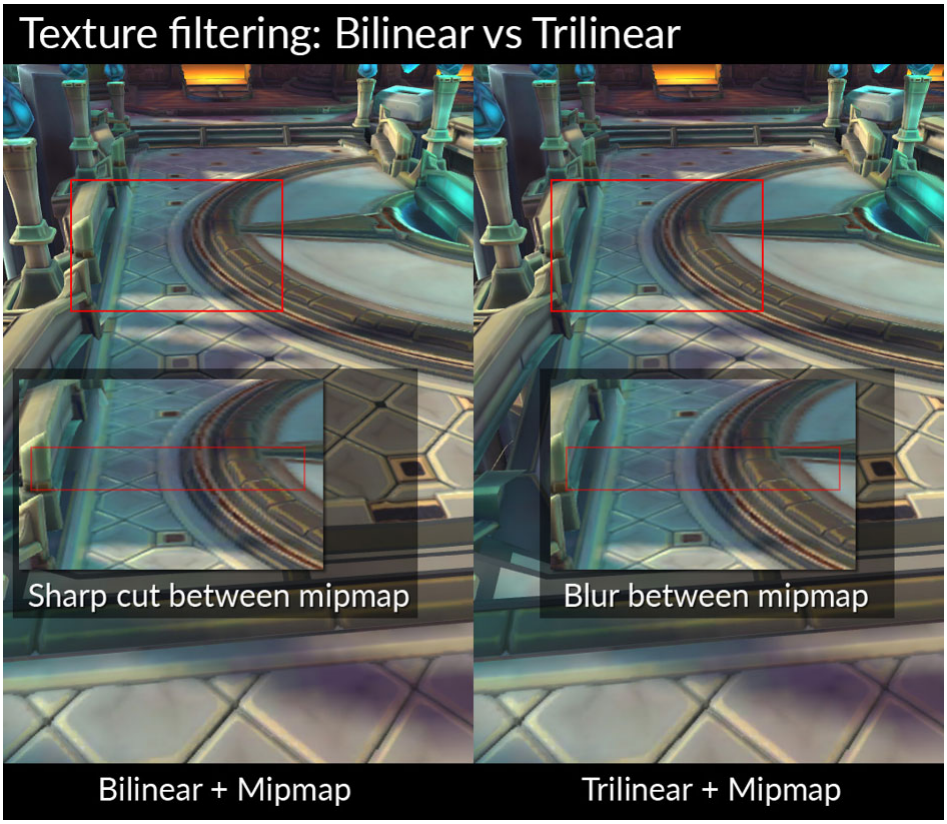


그림 6-3 Bilinear과 Trilinear 텍스처 필터링 비교 예

———— 참고 ————

Bilinear과 Trilinear 필터링은 더 많은 픽셀을 샘플링해야 하므로 더 많은 연산이 필요합니다.

————————————

Anisotropic 필터링
: Anisotropic 필터링은 한 각도에서 보았을 때의 텍스처 품질을 높입니다. 예를 들어 그라운드 레벨 텍스처에 사용하면 좋습니다.


그림 6-4 Anisotropic 텍스처 필터링 예

**텍스처 필터링 모범 사례**

우리는 다음 텍스처 필터링 팁을 해 보기를 권장합니다.

- Bilinear 필터링을 사용해 성능과 시각 품질의 균형을 잡습니다.
- Trilinear 필터링은 Bilinear 필터링보다 메모리 대역폭을 더 많이 사용하므로 선택적으로 사용합니다.
- Trilinear 필터링과 1x Anisotropic 필터링 대신 Bilinear 필터링과 2x Anisotropic 필터링 사용하는 것이 성능 및 품질 면에서 더 낫습니다.
- Anisotropic 레벨을 낮게 유지합니다. 중요 게임 애셋의 경우 두 단계 대신 한 단계 높은 레벨을 사용합니다.

## 6.3 밉매핑

밉맵은 원본 텍스처와 이를 여러 개의 더 작은 해상도들로 구성된 텍스처의 집합입니다. 따라서 밉맵은 텍스처의 *LOD*라고 할 수 있습니다.

프래그먼트가 텍스처 공간을 얼마나 차지하는지에 따라 적절한 샘플링 레벨이 선택됩니다. 오브젝트가 카메라에서 멀수록 저해상도 텍스처가 적용됩니다. 오브젝트가 카메라에서 가까울수록 고해상도 텍스처가 적용됩니다.

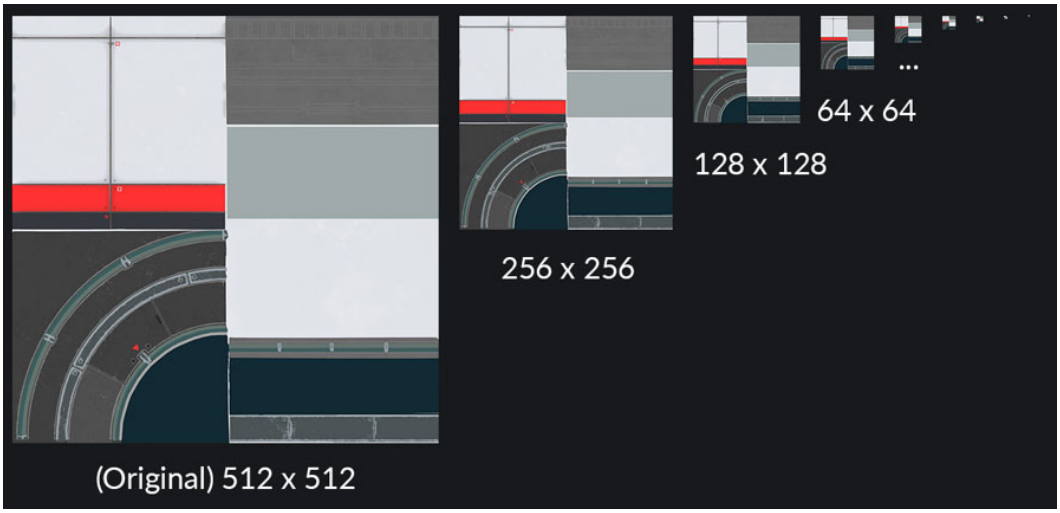


그림 6-5 밉맵 체인 예

### 모범 사례

밉맵은 성능과 품질을 개선할 수 있으므로 밉맵을 사용했는지 확인합니다. 밉매핑을 사용하면 GPU가 카메라에서 먼 오브젝트의 텍스처를 전체 해상도로 렌더링할 필요가 없으므로 GPU 성능이 개선됩니다.

또한 밉매핑은 텍스처 에일리어싱을 감소시키고 최종 이미지 품질을 높입니다. 텍스처 에일리어싱은 카메라에서 먼 영역에 플리커링 효과를 발생시킵니다.

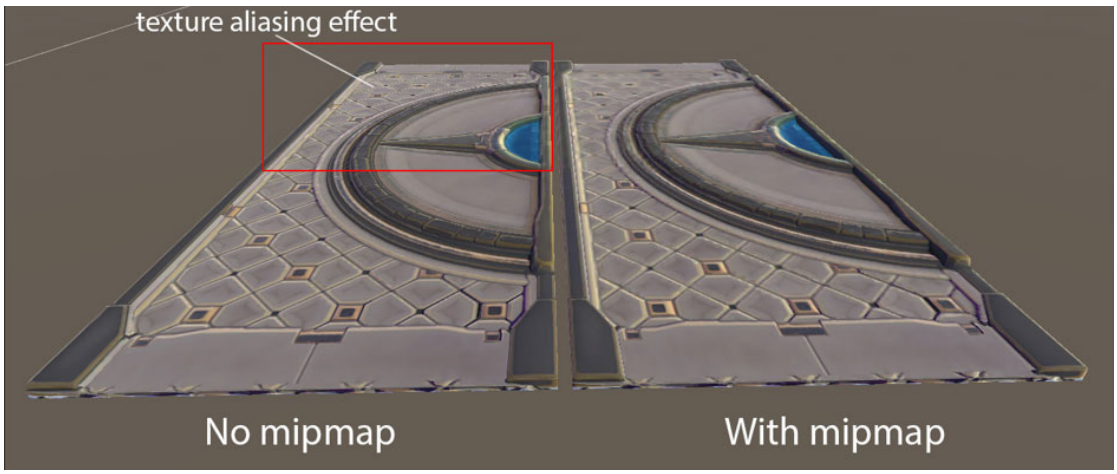


그림 6-6 밉맵 비교 예

Unity는 가져오기에 필요한 만큼 자동으로 밉맵을 생성하고 2의 배수가 아닌 텍스처를 다시 스케일합니다. 이에 대한 자세한 내용은 *Unity 웹 사이트를 참조하십시오*.

## 6.4 텍스처 크기, 컬러 스페이스, 압축

필요한 품질에 맞는 크기 이상의 텍스처를 생성하지 마십시오. 또한 여러 메시에서 공유하는 여러 텍스처를 포함한 대형 텍스처 아틀라스를 사용하는 것이 좋습니다.

### 텍스처 크기

텍스처 크기는 다양할 수 있습니다. 디테일이 적게 필요한 텍스처의 크기를 줄이면 대역폭 레벨을 감소시키는 데 도움이 됩니다. 예를 들어 디퓨즈 텍스처는 1024x1024로 설정하고 러프니스와 메탈릭은 512x512로 매핑할 수 있습니다.

선별하여 텍스처 크기를 줄이고 이어서 시각 품질이 감소되었는지 확인하십시오.

### 텍스처 컬러 스페이스

대부분의 Adobe Photoshop이나 Substance Painter 같은 텍스처링 소프트웨어는 sRGB 컬러 스페이스를 사용해 작업하고 내보냅니다.

우리는 다음을 시도하기를 권장합니다.

- sRGB 컬러 스페이스에서 디퓨즈 텍스처를 사용합니다.
- 색상으로 처리되지 않는 텍스처는 sRGB 컬러 스페이스에 포함되지 않아야 합니다. 메탈릭, 러프니스, 노말 맵 등이 일부 예가 될 수 있습니다.
  — 맵이 색상이 아닌 데이터를 사용하기 때문입니다.
  — 이런 맵에서 sRGB를 사용하면 표시, 시각 품질, 머티리얼이 잘못 표시될 수 있습니다.

———— 참고 ————

Inspector 창에서 텍스처의 sRGB(Color Texture) 설정을 할 때 러프니스, 반사, 노말 맵 등을 선택하면 안 됩니다.

————————————

다음 스크린샷은 이러한 텍스처에 sRGB가 잘못 적용되는 경우를 보여줍니다.


그림 6-7 sRGB 적용 텍스처 비교 예

### 텍스처 압축

텍스처 압축은 텍스처 데이터 크기를 줄이면서 시각 품질 손실을 최소화하는 이미지 압축입니다. 개발 시에는 TGA나 PNG 같은 일반 포맷으로 텍스처를 내보냈습니다. 이런 포맷은 사용이 편리하며 주요 이미지 소프트웨어도 이를 지원합니다.

이런 포맷은 특별 이미지 포맷보다 액세스와 샘플링 속도가 느리므로 최종 렌더링에 사용되면 안 됩니다. Android에서는 *Adaptive Scalable Texture Compression*(ASTC)와 *Ericsson Texture Compression*(ETC) 1, ETC2 등의 옵션이 있습니다.

### 텍스처 압축 모범 사례

Unity는 ARM이 개발한 ASTC 기술을 사용하는 것을 권장합니다. ASTC를 사용해야 하는 몇 가지 이유는 다음과 같습니다.

- ASTC는 ETC와 동일한 메모리 크기일 때 품질이 더욱 좋습니다.
- 또한 ASTC는 ETC보다 적은 메모리 크기로도 동일한 품질을 구현합니다.
- ASTC는 ETC보다 인코딩이 오래 걸려 게임 패키징 프로세스에 더 오랜 시간이 걸립니다. 이게 문제가 되는 경우 게임의 최종 패키징에 ASTC를 사용하는 것이 좋습니다.
- ASTC는 블록 크기를 설정할 수 있어 품질 면에서 더 많은 제어가 가능합니다. 블록 크기로 가장 적절한 단일 기본 크기가 없기 때문에 블록 크기는 5x5나 6x6으로 시작하면 좋습니다.

가끔 장치에 게임을 빠르게 배포해야 하는 경우 ETC로 개발하는 것이 더 빠를 수 있습니다. 배포 시간이 증가하는 경우 빠른 압축 설정의 ASTC를 사용할 수 있습니다. 인코딩할 때 품질과 크기를 희생하는 대신 속도를 선택할 수 있는 옵션이 있습니다. 최종 빌드의 시각 품질과 파일 크기 면에서 ASTC는 가장 균형 잡힌 최고의 선택입니다.

게임 엔진은 게임을 패키징할 때 텍스처 압축을 처리합니다. 하지만 그 방식은 선택할 수 있습니다. 포맷을 선택하여 이 단계를 건너뛰지 않도록 합니다.

다음 스크린샷은 Unity에서 Android 패키지를 빌드할 때 ASTC를 선택하는 방법을 보여줍니다.

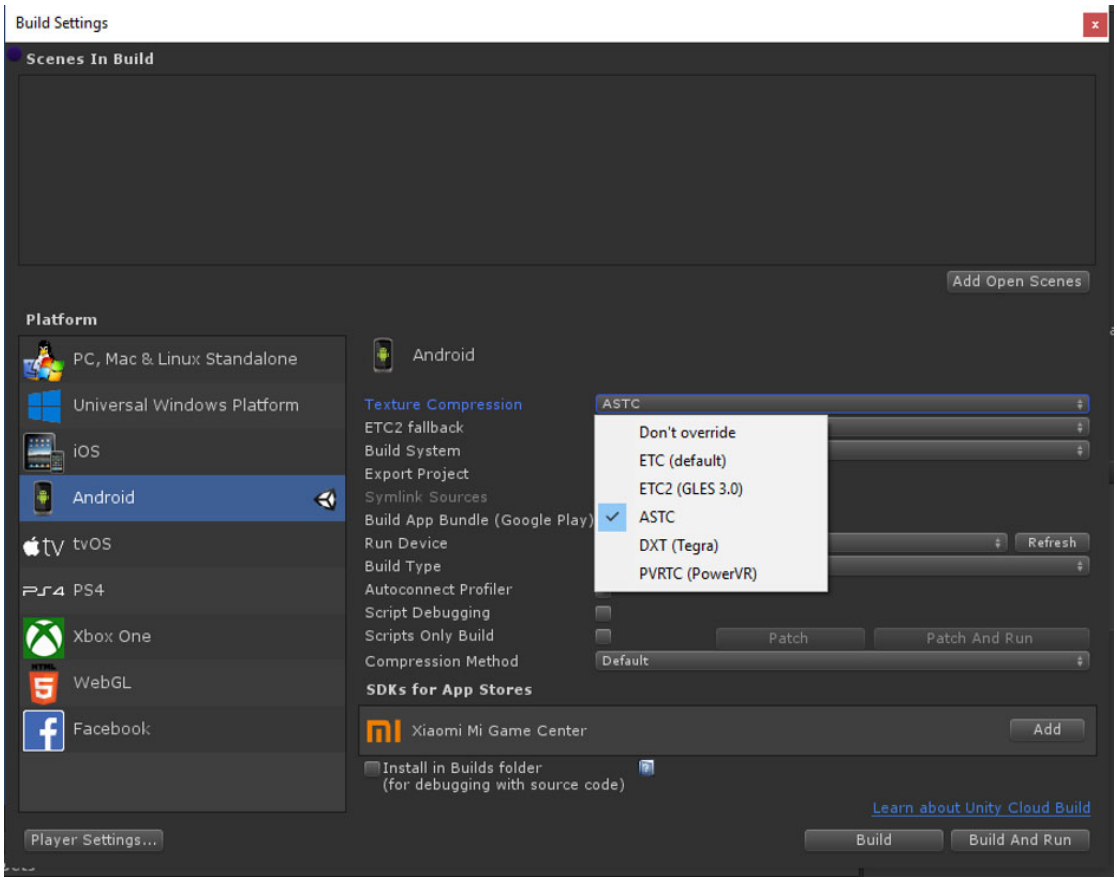


그림 6-8 Unity에서 Android용 ASTC 텍스처 압축 사용

다음 그림은 ETC와 ASTC 압축의 품질과 파일 크기의 차이를 보여줍니다.

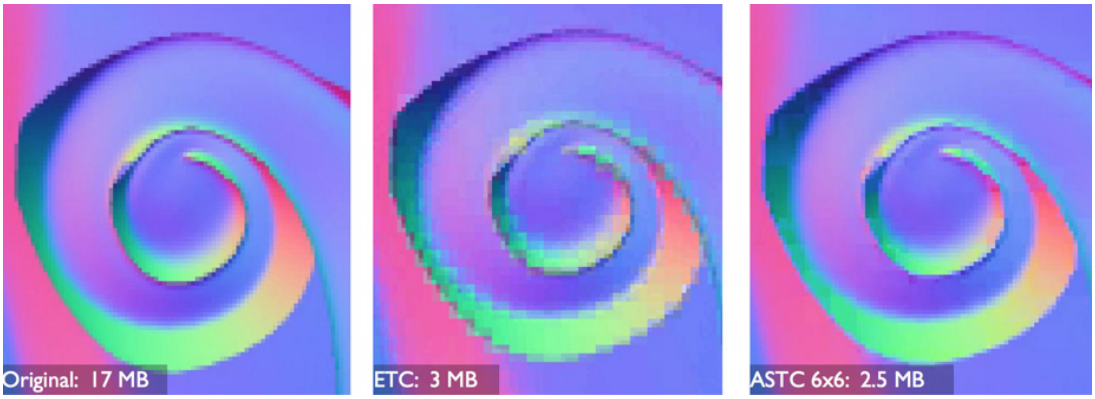


그림 6-9 ETC와 ASTC 텍스처 압축 비교

## 6.5 UV 언랩, 시각 효과, 텍스처 채널 패킹

UV 맵은 2D 텍스처를 3D 모델의 표면에 투영합니다. UV 언래핑은 UV 맵을 생성하는 프로세스입니다.

### UV 언래핑

UV 아일랜드를 그대로 두는 것이 가장 좋습니다.

———— 참고 ————

UV 아일랜드는 텍스처 맵에 있는 폴리곤 연결 그룹입니다.

————————————

그 이유는 다음과 같습니다.

- UV 아일랜드에 쉽게 패킹하여 낭비 공간을 줄입니다.
- 직선 UV는 텍스처에 발생하는 계단 효과를 줄입니다.
- 모바일 플랫폼에서 텍스처 크기가 게임 콘솔이나 PC보다 작으므로 텍스처 공간이 제한됩니다. 좋은 UV 패킹은 텍스처에서 최대한의 해상도를 얻었는지 보장해주는 것입니다.
- 전체적으로 우수한 텍스처 품질을 위해 UV의 직선을 유지하는 것보다 약간 일그러진 UV를 갖는 것이 나을 수 있습니다.

너무 눈에 띄지 않을 곳에 UV를 배치합니다. 시각 품질 목적이므로 모델의 텍스처 표현은 좋지 않을 수 있습니다. 따라서 UV 아일랜드에 공간이 있고 에지가 날카로우면 UV 아일랜드를 분할하십시오. 이렇게 하면 베이킹 처리로 더 나은 노말 맵을 생성할 수 있습니다.

다음 그림은 텍스처 공간을 최대화하기 위해 UV 언래핑을 사용한 방법을 보여줍니다.

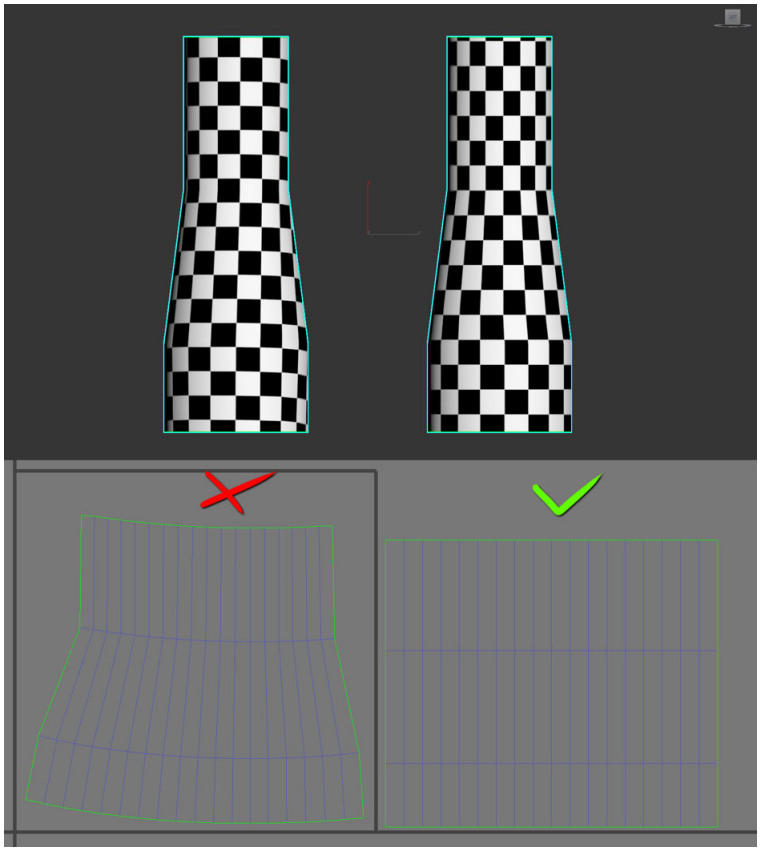

그림 6-10 텍스처 공간을 최대화하기 위한 직선 UV 언래핑 예

### 시각 효과

보이는 디테일만 만들도록 하십시오. 휴대 전화 화면은 작으므로 아주 작은 디테일은 보이지 않습니다. 텍스처를 만들 땐 이를 고려하십시오. 예를 들어 방 구석에 있어 거의 보이지 않는 의자에 디테일이 섬세한 4K 텍스처를 사용할 필요는 없습니다.

다음 스크린샷은 필요한 디테일만 사용한 작은 군인 텍스처의 예입니다.


그림 6-11 적절한 디테일만 사용한 작은 텍스처 예

어떤 경우에는 형태 인식률을 높이기 위해 에지나 셰이딩을 강조하거나 확대할 필요가 있습니다. 모바일 플랫폼은 보통 작은 텍스처를 사용한다는 것을 기억하십시오. 작은 텍스처에서는 필요한 모든 디테일을 전부 포함하기 어려울 수 있습니다.

적은 텍스처를 사용하고 추가 디테일은 하나의 텍스처에 베이킹하십시오. 이것이 중요한 이유는 다음과 같습니다.

- 휴대 전화 화면은 작고 일부 디테일은 디퓨즈 텍스처로 베이킹되어 디테일을 표시하는 것이 낫기 때문입니다.
- 앰비언트 오클루전이나 작은 정반사광 같은 요소도 베이킹 후 디퓨즈 텍스처에 추가할 수 있습니다.

이 방식은 정반사나 앰비언트 오클루전을 표현할 때 셰이더나 엔진 기능에 너무 의존할 필요가 없다는 것을 의미합니다.

다음 스크린샷은 텍스처에 디테일을 베이킹하는 예입니다.


그림 6-12 텍스처에 디테일을 베이킹하는 예

가능하면 셰이더에서 컬러 틴팅이 가능한 그레이스케일 텍스처를 사용합니다. 이렇게 하면 틴팅을 수행하는 커스텀 셰이더 생성 비용으로 텍스처 메모리를 절약할 수 있습니다.

모든 오브젝트에 대해서 이 방법이 맞는 것은 아니기 때문에 이 테크닉을 사용할 때는 신중하시기 바랍니다. 동일하거나 유사하거나 색상이 있는 오브젝트에 적용하면 좋습니다.

RGB 마스크를 사용해 해당 마스크의 색상 범위를 바탕으로 텍스처를 적용하는 방법도 있습니다.

다음 그림은 틴팅된 기둥에 그레이스케일 텍스처가 적용된 예입니다.


그림 6-13 그레이스케일 텍스처 예

### 텍스처 채널 패킹

텍스처 채널을 사용해 여러 텍스처를 하나로 패킹합니다. 우리는 아래의 방법을 추천합니다.

- 패킹 테크닉을 사용하면 3개의 맵을 하나의 텍스처로 만들기 때문에 텍스처 메모리를 절약할 수 있습니다. 즉 텍스처 샘플러가 적게 사용됩니다.
- 이 텍스처 패킹 테크닉은 일반적으로 러프니스나 스무드니스, 메탈릭을 하나의 텍스처로 패킹하는 데 사용됩니다. 하지만 어떤 텍스처 마스크에도 사용할 수 있습니다.

녹색 채널을 사용해 더 중요한 마스크를 저장할 수 있습니다. 녹색 채널은 비트가 더 많은 편입니다. 우리 눈이 녹색에 민감하고 파란색에 덜 민감하기 때문입니다. 우리는 아래의 방법을 추천합니다.

- 러프니스나 스무드니스의 경우는 메탈릭보다 맵에 더 많은 디테일이 포함되므로 녹색 채널에 배치합니다.
- 이런 텍스처의 경우 sRGB 컬러 스페이스 대신 선형이나 RGB로 설정합니다.

다음 이미지는 텍스처 패킹 예를 보여줍니다.

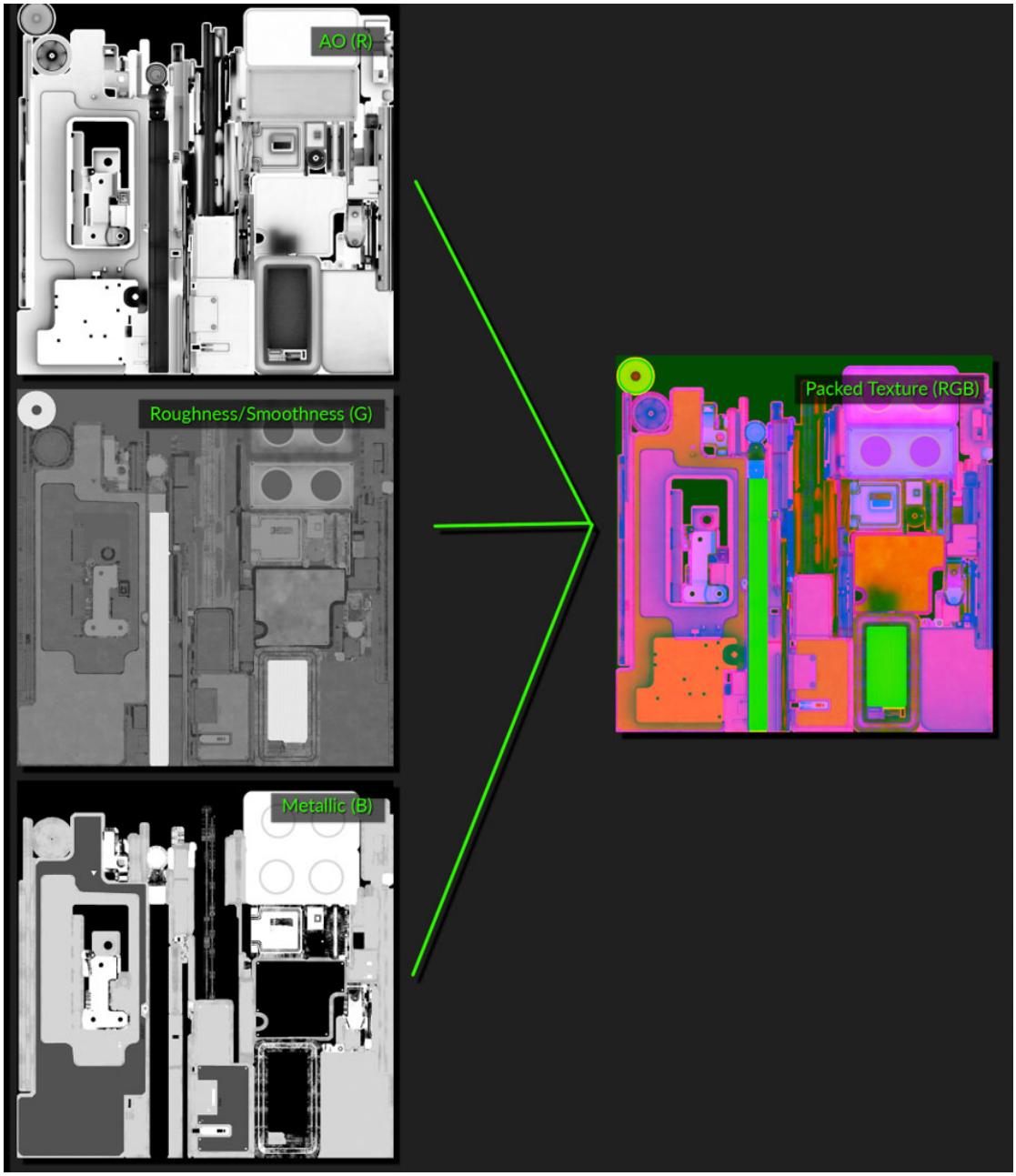


그림 6-14 텍스처 채널 패킹 예

색상 민감도 그래프 읽기에 대한 자세한 내용은 *이곳을 참조하십시오.*.

## 6.6 알파 채널과 노말 맵 모범 사례

다음은 게임에서 알파 채널과 노말 맵을 사용하려는 경우 추천하는 모범 사례들입니다.

### 알파 채널 사용

텍스처에 알파 채널을 추가할 때는 주의가 필요합니다. 투명도를 추가하면 텍스처 파일을 32비트 포맷으로 변환하므로 전체 사용 대역폭을 증가시켜 파일 크기가 커지는 경향이 있습니다.

알파 채널을 저장하는 또 다른 방법은 러프니스 또는 메탈릭 텍스처에 추가 채널을 사용하는 것입니다. Unity에서 이런 텍스처는 3개 중 러프니스(G)와 메탈릭(B)이라는 2개의 채널만 사용하므로 (R) 채널을 자유롭게 사용할 수 있습니다.

자유 채널을 사용해 알파 마스크를 저장하면 디퓨즈 텍스처를 16비트로 분산시킬 수 있어 파일 크기를 반으로 줄일 수 있습니다. 앰비언트 오클루전 맵은 보통 디퓨즈 맵에서 베이킹됩니다.

다음 그림은 레드 채널에서 투명도 맵을 저장하는 예시를 보여줍니다.

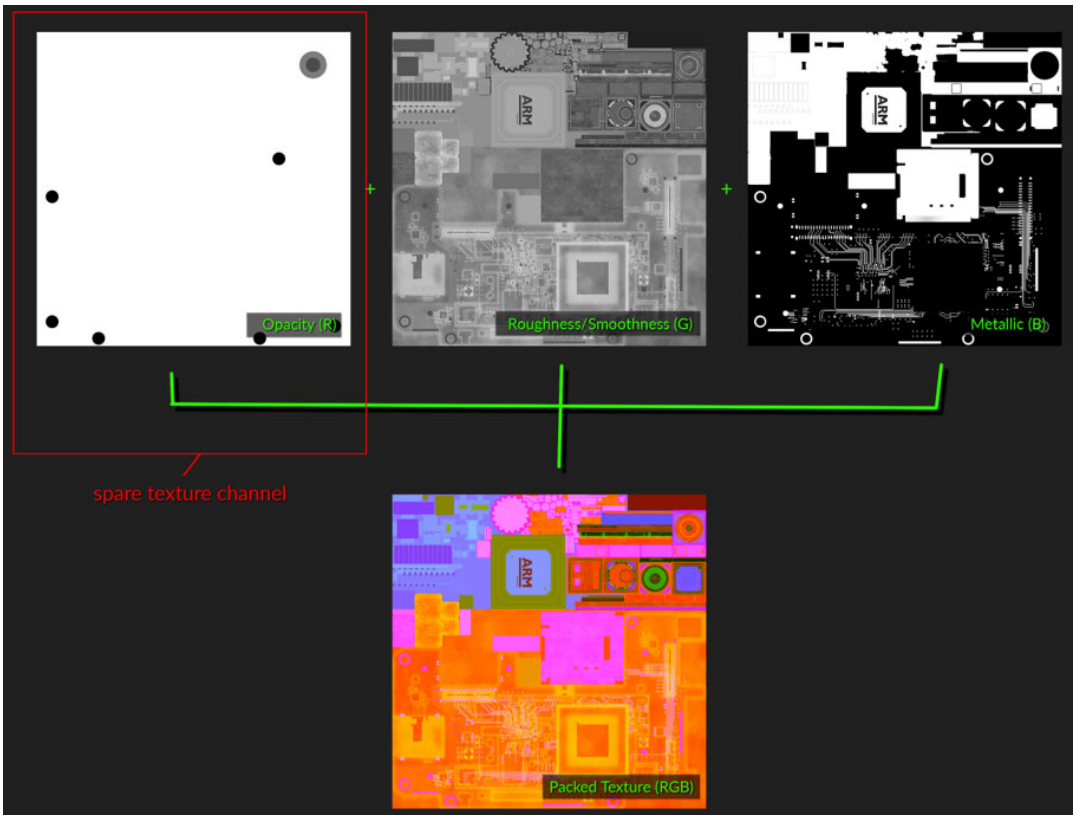


그림 6-15 레드 채널에 투명도 맵 저장

### 노말 맵 모범 사례

노말 말은 3D 오브젝트에 디테일을 추가하는 좋은 방법입니다. 주름과 볼트같이 작은 디테일을 추가할 때와 많은 삼각형을 필요로 하는 디테일을 추가하는데 좋습니다. 노말 맵 사용은 게임의 유형 및 아트 디렉션에 따라 달라집니다.

우리의 내부 프로젝트 대부분은 눈에 띄는 성능 저하 없이 노말 맵을 사용합니다. 대부분의 데모는 고사양 장치를 대상으로 하므로 저사양 장치에서는 다른 결과가 나타날 수 있습니다.

노말 맵을 사용하면 적은 비용이라도 비용이 발생합니다. 아래의 사항들을 염두하십시오.

- 노말 맵은 추가 텍스처이므로 더 많은 텍스처를 가져오기 위해 더 많은 대역폭을 사용하게 됩니다.
- 저사양 장치를 대상으로 할 때는 노말 맵을 적게 사용합니다.

그러나 노말 맵을 사용해 지오메트리의 삼각형 수가 적어지면 성능이 개선될 수 있습니다.

다음 그림은 작은 디테일에 노말 맵과 텍스처를 사용하는 예시를 보여줍니다.

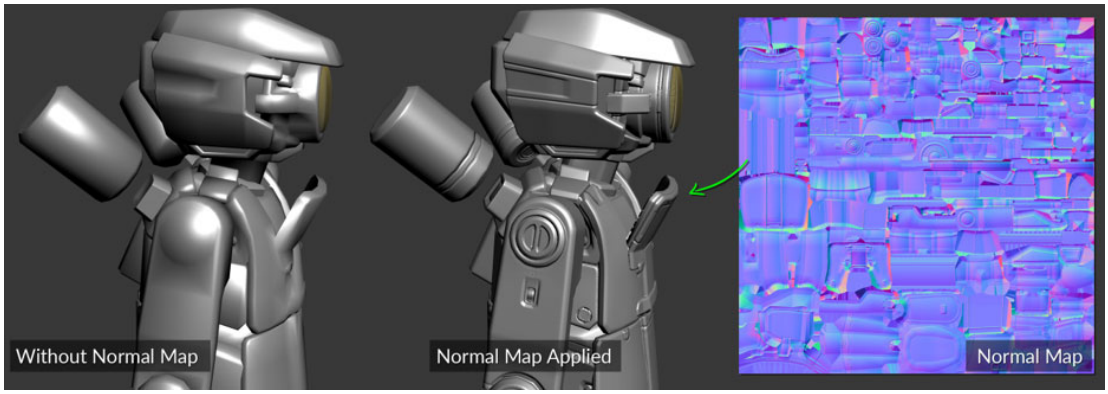


그림 6-16 노말 맵과 텍스처 예

### 노말 맵 베이킹 모범 사례

베이킹하는 표면 유형과 상관없이 고품질 노말 맵을 얻기 위해 케이지를 사용하면 좋습니다. 일반적인 노말 맵 소프트웨어는 자동으로 케이지를 생성합니다. 필요한 경우, 저 폴리곤 모델에서 만든 노말 맵을 만들고 복사한 뒤 크기를 약간 늘릴 수 있습니다.

프로그램은 케이지를 사용해 베이킹을 할 때 노말을 계산할 때 사용되는 방향을 변경합니다. 이렇게 하면 스플릿 노말이나 하드 에지에 더 좋은 결과를 얻을 수 있습니다.

케이지는 폴리곤 수가 적은 모델을 키운 버전이므로 모델이 밀려난 것처럼 보입니다. 베이킹을 잘 적용하려면 물리적으로 폴리곤 수가 많은 모델을 덮어야 합니다.

## 6.7 노말 맵 베이킹 모범 사례

베이킹하는 표면 유형과 상관없이 고품질 노말 맵을 얻기 위해 케이지를 사용하면 좋습니다. 대부분의 노말 맵 소프트웨어는 자동으로 케이지를 만들 수 있지만, 저 폴리곤 모델을 복사해 스케일을 약간 변형시켜 노말 맵을 만들 수 있습니다.

케이지의 사용목적은 프로그램이 베이킹을 할 때 노말을 계산할 때 사용되는 방향을 바꾸기 위함입니다. 이렇게 하면 다음 그림과 같이 스플릿 노말과 하드 에지에서 더 나은 결과를 보여줍니다.

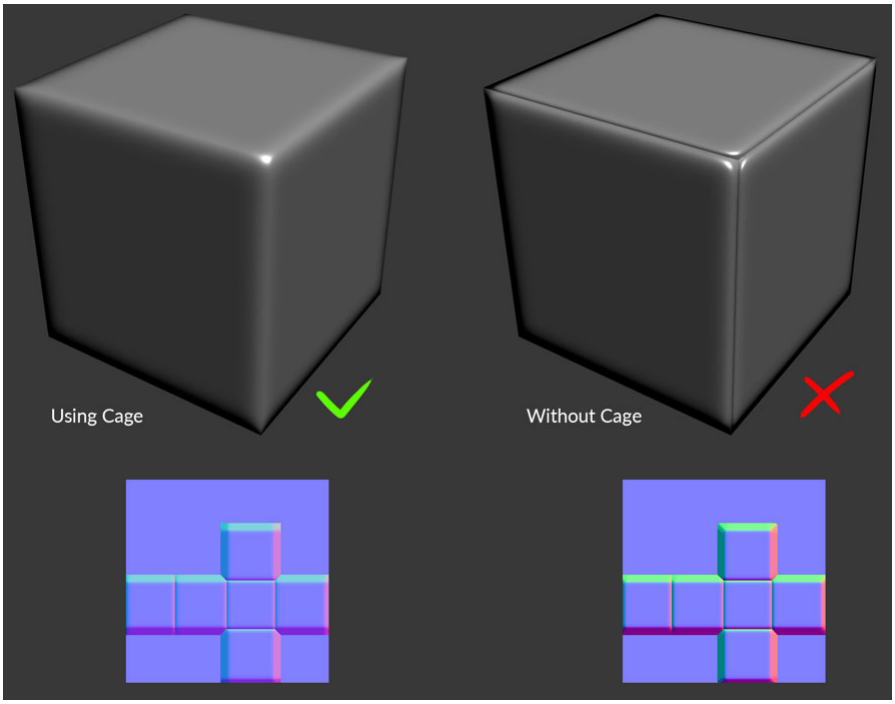


그림 6-17 케이지 사용 비교

케이지는 폴리곤 수가 적은 모델을 키우거나 밀어낸 버전입니다. 베이킹을 잘 적용하려면 물리적으로 폴리곤 수가 많은 모델을 포함해야 합니다.

메시 케이지는 노말 맵 베이킹 중 사용되는 레이 캐스트 거리를 제한하는 데 사용됩니다. 케이지는 다음 그림과 같이 노말 맵에서 스플릿 노말이 보여주는 문제를 해결할 수도 있습니다.

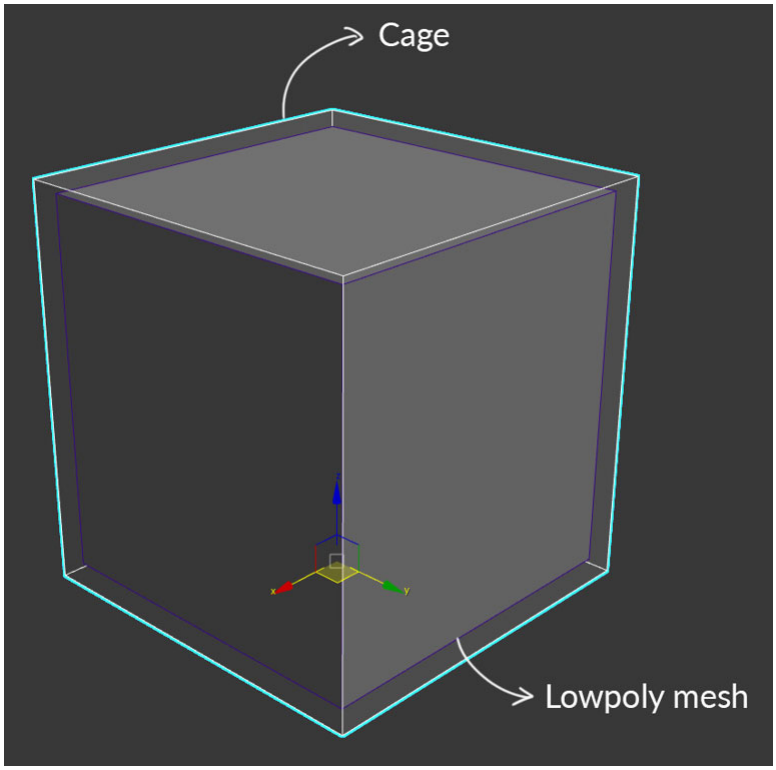


그림 6-18 메시 케이지의 예

베이킹 소프트웨어가 지원한다면 메시 이름과 동일하게 베이킹됩니다. 이렇게 하면 잘못된 노말 맵 프로젝션 생성 문제를 예방합니다. 거리가 너무 가까운 오브젝트는 노말 맵의 다른 면에 예측과 다르게 프로젝트 될 수 있습니다. 이 방법을 사용하면 맞는 이름으로 정확한 면에 베이킹이 이뤄집니다.

이름과 메시를 일치하는 것에 대한 자세한 내용은 *Marmoset Toolbag 튜토리얼*과 함께 *substance3d 웹사이트에서 확인하세요*.

베이킹에서 메시 이름을 일치시키지 않으면 메시를 해체합니다. 메시를 해체하면 노말 맵이 의도하지 않은 면에 프로젝트되지 않도록 파트를 떨어뜨려 놓는다는 것입니다. 이렇게 하면 올바르지 않은 노말 맵 프로젝션도 막아줍니다.

다음 이미지는 해체된 메시 예를 보여줍니다.

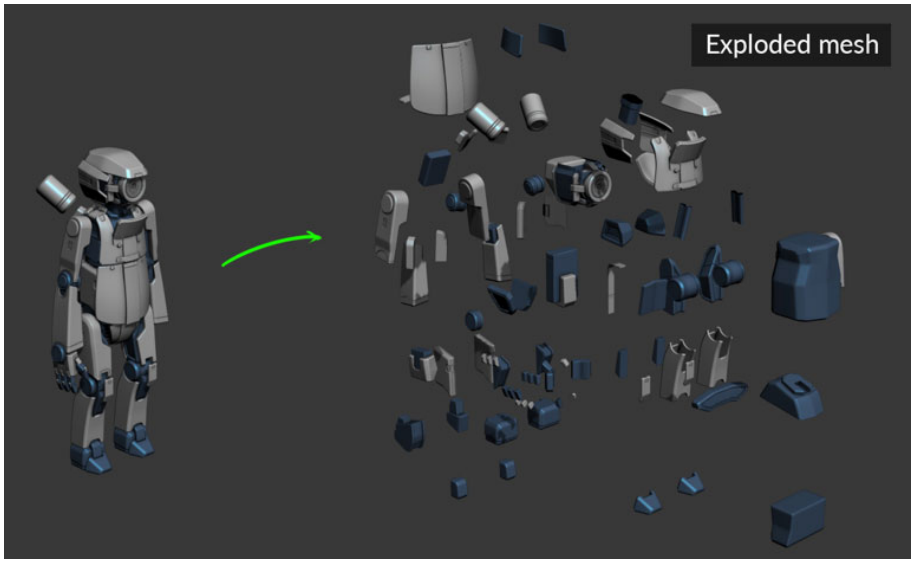


그림 6-19 노말 베이킹을 위해 해체된 메시 예

앰비언트 오클루전을 위해 분리 베이킹 (separate bake)을 할 땐 이 방법이 필요할 수 있습니다. 그래서 하드 에지의 연속적인 UV는 시각적 경계선을 유발하므로 하드 에지의 UV를 나눕니다. 일반적인 규칙은 각도를 90도 이하로 하거나 다른 스무딩 그룹으로 놓는 것입니다. 삼각형에서 삼각형의 다른 스무딩 그룹과 UV 경계선을 일치시킨다.

다음 그림은 하드 에지에서 분할 UV가 어떻게 보이는지 나타냅니다.

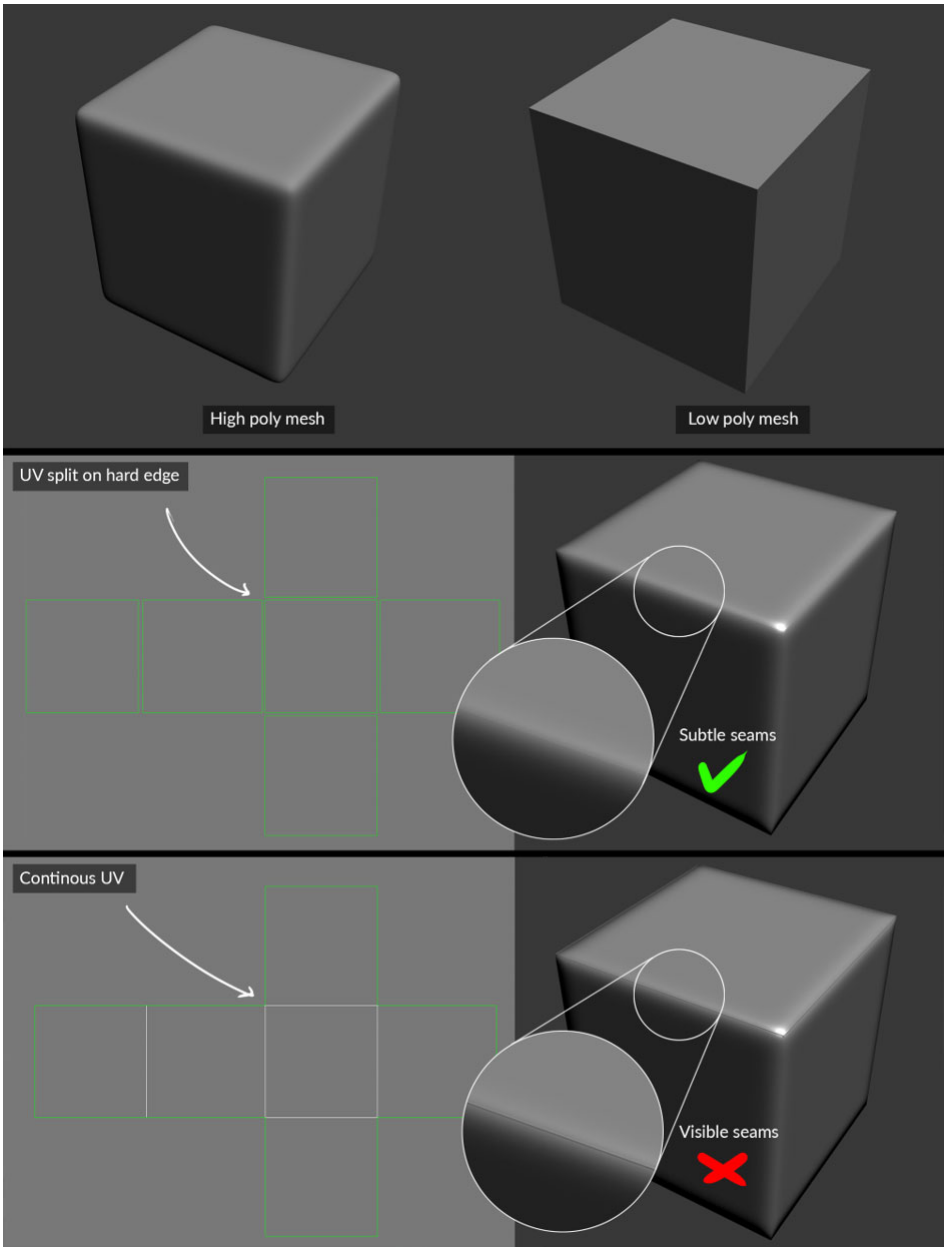


그림 6-20 하드 에지에서 분할 UV 예

## 6.8 텍스처 설정 편집

Unity는 텍스처 설정 편집을 편리하게 만들어 왔습니다. 편집하려는 텍스처를 선택하고 텍스처 설정을 표시하는 검사기 창을 확인합니다.

텍스처 설정을 편집할 때 유용한 팁은 다음과 같습니다.

- 텍스처 유형은 텍스처 유형을 제어하므로 엔진에서 텍스처가 사용되는 방법을 나타냅니다.
- 텍스처 모양은 일부 텍스처 유형에 2D 대신 큐브 맵을 사용하는 것입니다.
- 선택한 텍스처 유형에 따라 2가지 섹션이 표시됩니다.
  — 텍스처 설정은 텍스처가 원하는 작업이 가능하도록 특별하게 제어합니다.
  — 고급 설정은 sRGB, 알파 소스, 알파 투명, 읽기/쓰기 사용, 밉맵 생성 등을 다룹니다.
- 랩 모드는 텍스처의 UV 래핑 방식을 제어합니다. 사용 가능한 옵션은 다음과 같습니다.
  — 반복은 텍스처의 타일링 반복을 의미하며 패턴을 반복하는 데 사용됩니다.
  — 클램프는 에지의 최종 픽셀에 텍스처를 고정합니다.
  — 미러는 반복과 동일하지만 미러는 텍스처의 모든 반복을 반영합니다.
  — 미러 한 번은 텍스처를 한 번 반영하고 텍스처를 픽셀 모서리에 고정합니다.
- 필터 모드는 텍스처에서 사용되는 텍스처 필터를 제어합니다.
- 텍스처 압축 상자에서는 최대 크기, 크기 조절 알고리즘, 형식, 압축, 크런치 압축 사용을 제어할 수 있습니다.

# 제 7 장
# 실시간 3D 아트 모범 사례: 머티리얼과 셰이더

이 장에서는 게임의 시각 품질을 높이고 효율적으로 실행되도록 하는 다양한 머티리얼 및 셰이더 최적화를 다룹니다.

——— 주의 ———

이 콘텐츠의 최신 버전은 Arm 개발자 웹 사이트 *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*에서 확인할 수 있습니다.

---

이 장은 다음 단원으로 구성되어 있습니다.

## 7.1 셰이더와 머티리얼 소개

머티리얼과 셰이더는 3D 오브젝트가 보이는 방식을 결정하므로 각각의 정의와 최적화 방법을 아는 것이 중요합니다.

셰이더는 GPU에 화면에 오브젝트를 그리는 방법과 오브젝트에 필요한 모든 계산을 알려주는 작은 프로그램입니다. 셰이더는 머티리얼과 결합되어 있을 때만 사용할 수 있습니다.

셰이더를 작성할 때는 *고차원 셰이딩 언어*(HLS)과 *OpenGL 셰이딩 언어*(GLSL)이라는 두 개의 스크립팅 언어가 주로 사용됩니다.

머티리얼은 오브젝트나 메시에 적용되어 오브젝트의 시각적 표현을 결정합니다. 머티리얼은 셰이더에서 사용 가능한 특정 매개변수 값을 설정하는 데 사용됩니다. 색상, 텍스처, 숫자 값 등이 그 예입니다.

다음 두 스크린샷은 셰이더를 생성하는 두 가지 방식과 다른 머티리얼에서 사용된 경우를 보여줍니다.

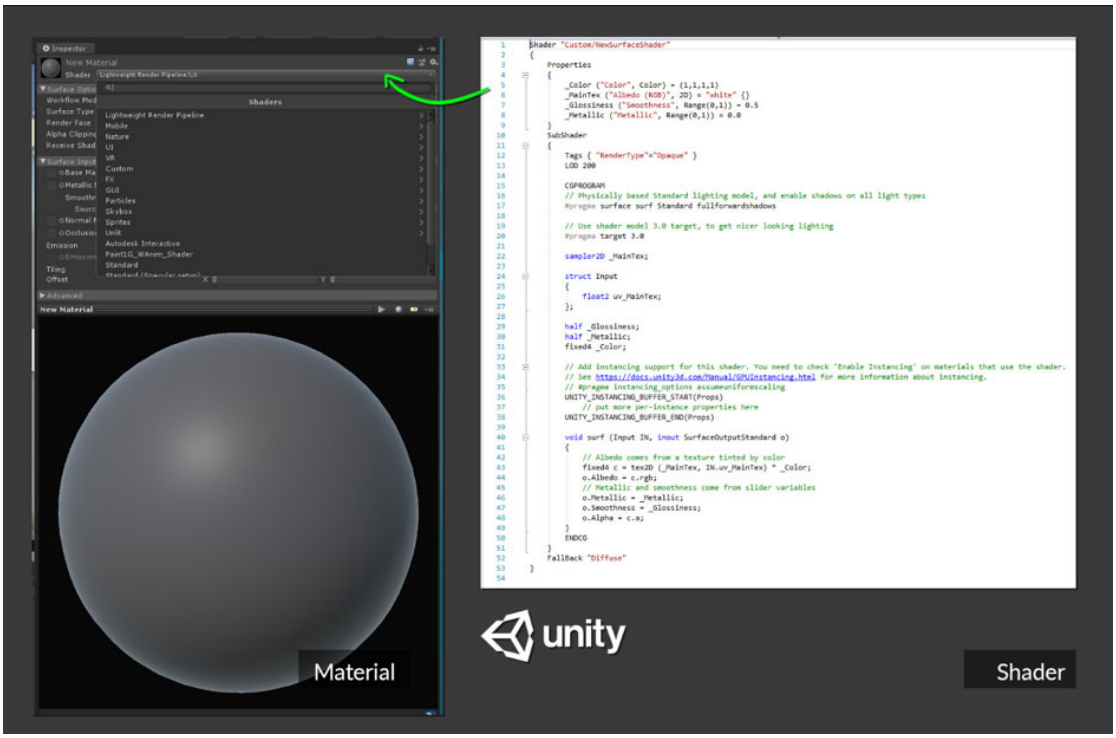


그림 7-1 셰이더와 머티리얼

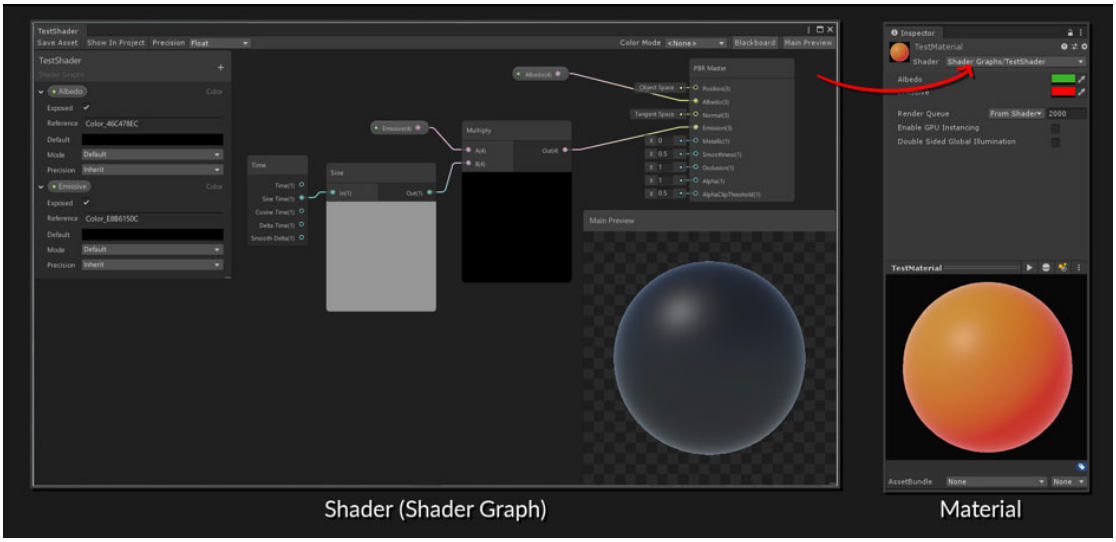


그림 7-2 셰이더 그래프와 머티리얼 메뉴

## 7.2 모바일 플랫폼용 최적화된 셰이더 사용

Unity는 모바일 플랫폼에 최적화된 셰이더 모음을 제공합니다. 이런 셰이더는 Unity의 모바일 카테고리에 있습니다.

이 최적화된 셰이더는 단순화된 기능을 포함하지만 모바일 기반 플랫폼의 성능을 향상시켜줍니다. 그러나 모바일용이 아닌 표준 셰이더에 비해 사용할 수 있는 기능이 적습니다. 예를 들면 컬러 틴팅 기능이 없습니다.

다음 스크린샷은 모바일 전용 셰이더의 위치를 보여줍니다.

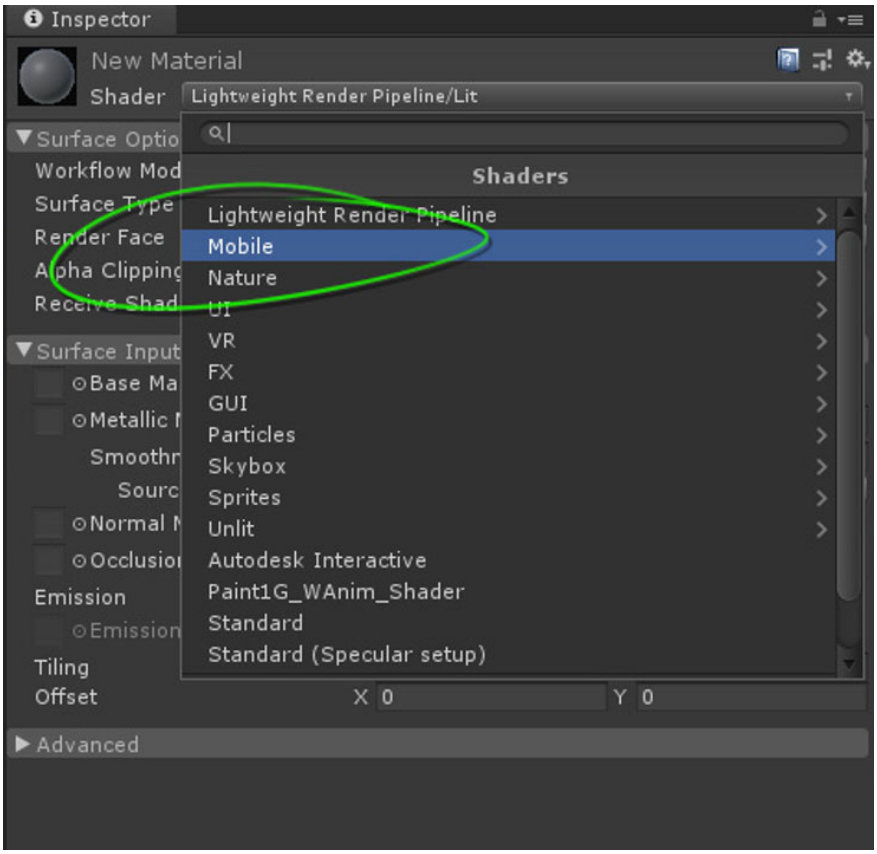


그림 7-3 Unity 모바일 셰이더 위치

필요한 기능만 사용하십시오. Unity는 머티리얼에서 선택되어 구성된 셰이더에서 최적화된 런타임 버전을 생성합니다. 셰이더 복잡성을 줄이면 모바일 플랫폼의 성능에 도움이 됩니다.

## 7.3 텍스처 최적화

모바일 플랫폼에서는 텍스처를 가능한 적게 사용하십시오. 많은 텍스처를 사용하면 텍스처 가져오기를 더 많이 하기 때문입니다. 텍스처 가져오기를 많이 하려면 더 많은 대역폭을 사용하므로 장치의 배터리 수명에 영향을 줍니다.

더 많은 텍스처를 메모리에 저장하기 때문에 응용 프로그램 크기도 증가합니다. 러프니스 텍스처와 메탈릭 텍스처에 개별 텍스처를 사용하는 대신 단일 텍스처의 채널로 묶을 수 있습니다. 이 테크닉은 텍스처 묶음이라고 하며 사용되는 텍스처의 수를 줄여줍니다.

다음 그림은 3개의 개별 텍스처를 단일 채널로 묶어 대역폭을 절약하는 예시를 보여줍니다.

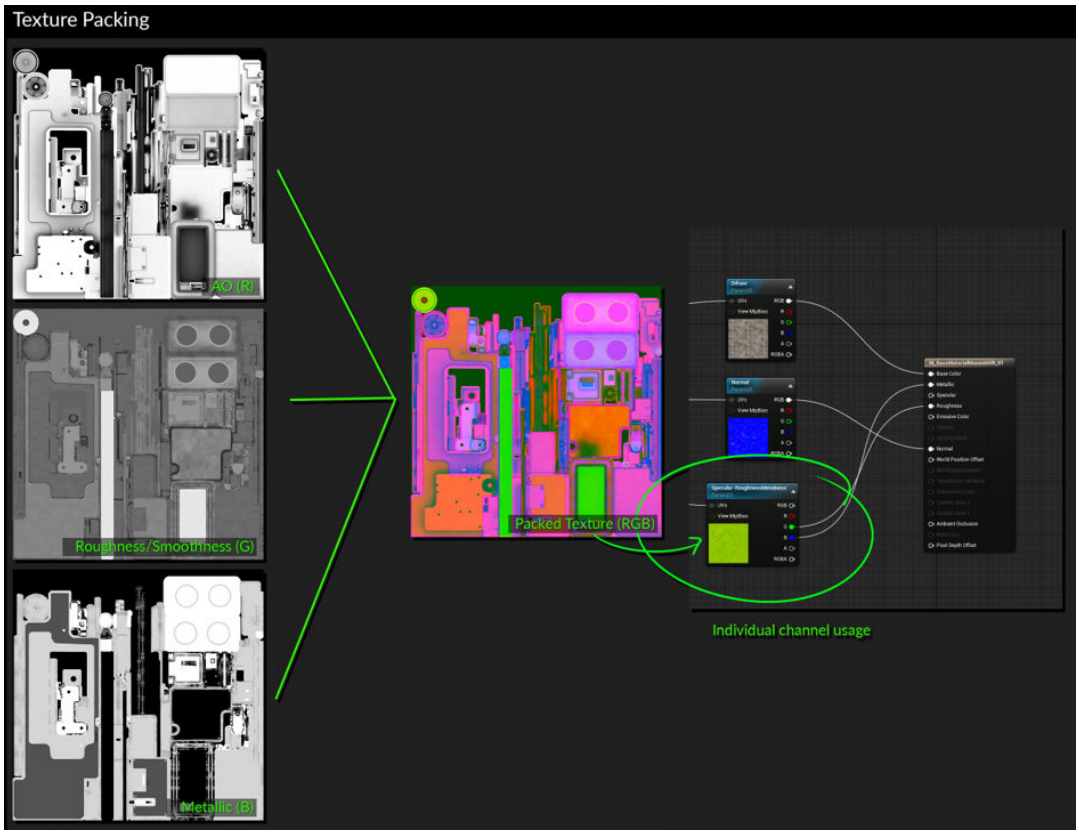


그림 7-4 Unity의 텍스처 묶음

메탈릭, 러프니스, 스무드니스 같은 일부 매개변수에 하나의 텍스처를 사용하는 대신 숫자를 사용할 수 있습니다. 가능한 경우 이 테크닉을 사용하고 시각 품질에 영향을 주는지 확인하십시오. 숫자 값을 사용하면 사용되는 텍스처의 수를 더 줄일 수 있습니다.

———— 참고 ————

Unity에서는 셰이더에 값을 추가해야 합니다.

————————

조명 없는 셰이더를 사용하면 라이트가 머티리얼에 영향을 주지 않으므로 사용되는 텍스처의 수를 줄일 수 있습니다. 즉 러프니스나 메탈릭 텍스처가 필요하지 않습니다.

## 7.4 언릿 셰이더와 릿 셰이더 비교

셰이더를 생성할 때 머티리얼이 빛에 어떻게 반응할지 결정할 수 있습니다. 모바일 게임에서 사용되는 대부분의 셰이더는 릿이나 언릿으로 나뉩니다.

언릿

언릿 셰이더는 가장 빠르고 저렴한 셰이딩 모델입니다. 저사양 장치를 대상으로 하는 경우 이 유형을 사용합니다. 이때 고려해야 할 점은 다음과 같습니다.

- 라이팅은 언릿 셰이딩 모델에는 영향을 주지 않고 색상 표시 결과에만 영향을 줍니다.
- 따라서 반사 같은 많은 계산이 필요하지 않습니다. 저렴하고 빠른 렌더링으로 결과를 얻을 수 있습니다.
- 언릿 셰이딩은 만화 같은 스타일의 아트 디렉션과 잘 어울립니다. 따라서 모바일 플랫폼으로 게임을 만드는 경우 이러한 아트 스타일을 고려하면 좋습니다.

릿

릿 셰이더는 언릿 셰이더보다 더 많은 처리 능력을 사용합니다. 그러나 다음 사항에 유의해야 합니다.

- 빛이 릿 셰이더에 영향을 주며 표면 반사가 가능합니다.
- 오늘날 모바일 게임에서 가장 많이 사용되는 셰이딩 모델일 것입니다.

다음 이미지는 릿 오브젝트와 언릿 오브젝트를 비교하는 것입니다.

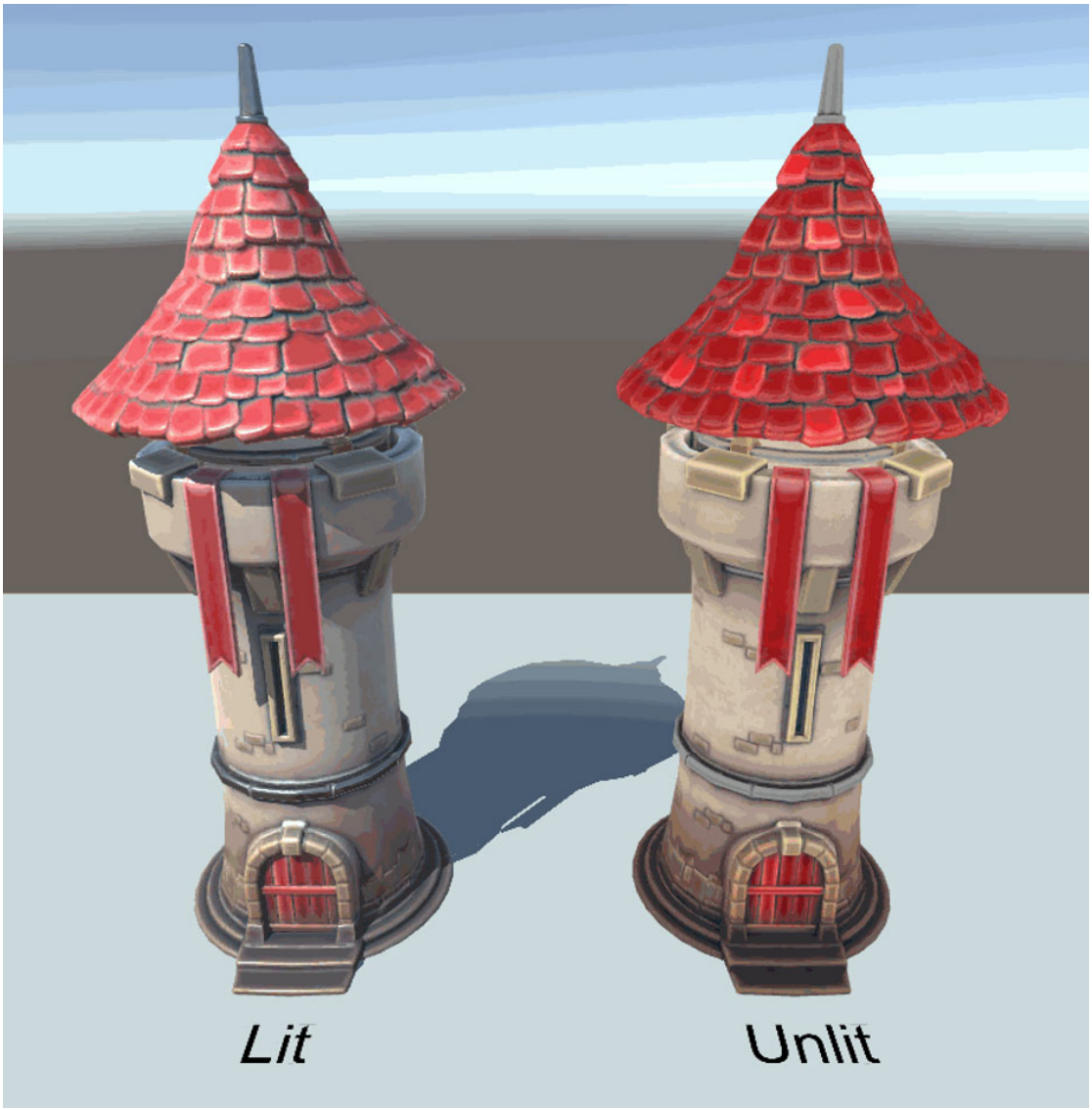


그림 7-5 릿 셰이더와 언릿 셰이더 비교

동일한 타워 메시와 텍스처 설정이 적용되었지만 다른 셰이더를 사용합니다. 빛은 언릿 셰이더에 영향을 주지 않으므로 컴퓨팅이 덜 필요합니다. 따라서 특히 저사양 장치의 게이밍 성능을 개선합니다.

## 7.5 투명도를 사용할 때는 주의하십시오.

가능하면 불투명한 머티리얼을 사용하십시오. 모바일 플랫폼에서 필요하지 않은 경우 투명도를 사용하지 않는 게 좋습니다.

투명도가 있는 오브젝트를 렌더링하면 반투명한 오브젝트를 렌더링하는 것보다 더 많은 GPU 리소스를 사용합니다. 투명한 오브젝트를 많이 사용하면 모바일 플랫폼의 성능에 영향을 줍니다. 특히 투명 오브젝트가 다른 오브젝트에 여러 번 렌더링되는 경우가 그렇습니다.

동일한 픽셀이 화면에 여러 번 그려지는 경우를 오버드로우라고 합니다. 투명도 레이어가 많을수록 렌더링 비용이 비싸지기 때문에 오버드로우가 문제가 됩니다. 모바일 플랫폼의 경우 오버드로우는 성능에 큰 영향을 미칩니다.

다음 스크린샷은 푸른 조명이 반투명 머티리얼로도 투명 효과를 낼 수 있는 방법을 보여줍니다.


그림 7-6 반투명 머티리얼 사용

따라서 레벨을 빌드할 때 오버드로우는 최소한으로 유지하려고 노력하십시오. Unity에는 장면 내 오버드로우의 양을 볼 수 있는 모드가 있습니다. 이 모드는 다음 스크린샷에 강조되어 있습니다.

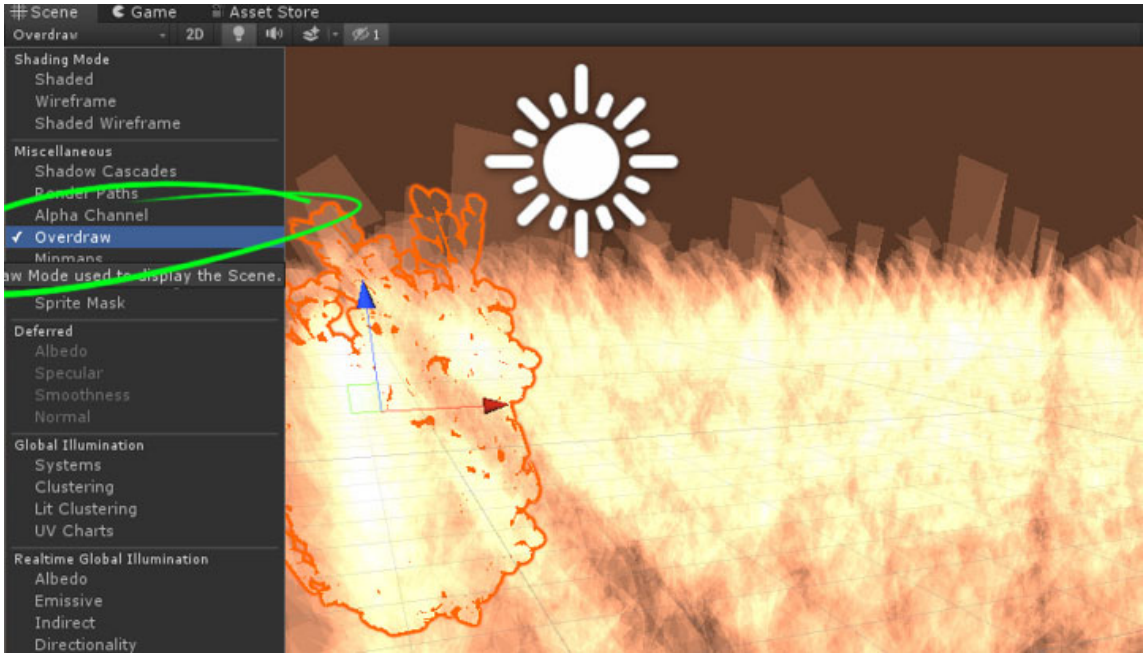


그림 7-7 Unity의 오버드로우 시각화

## 7.6 프로파일과 투명 비교 구현

셰이더에서 투명도를 구현하는 것과 가장 자주 사용되는 방식은 알파 테스트와 알파 블렌드로 다릅니다. 그러나 사용 사례에 따라 최선의 성능을 내는 결과는 다릅니다. ARM은 둘 간 성능 차이를 항상 프로파일링하는 것을 권장합니다.

모바일 플랫폼에서 ARM 툴인 *스트림라인*을 사용해 장치의 성능 데이터를 수집할 수 있습니다.

알파 테스트 또는 컷아웃
: 알파 테스트 구현은 오브젝트 머티리얼을 100% 반투명 또는 100% 투명으로 표현합니다. 마스크의 컷아웃 한계를 설정할 수 있습니다. Unity에서는 이런 형태의 투명도를 컷아웃이라고 합니다.

알파 블렌드
: 시각적으로 알파 블렌드는 투명 범위의 머티리얼을 허용해 오브젝트가 반투명하거나 완전히 투명하지 않고 일부만 투명하게 할 수 있습니다. Unity는 이런 블렌드 모드를 투명이라고 합니다.

알파 블렌드는 부분 투명을 허용하고 알파 테스트는 딱 떨어지는 컷아웃을 만들어냅니다. 알파 블렌드와 알파 테스트는 다음과 같이 비교할 수 있습니다.

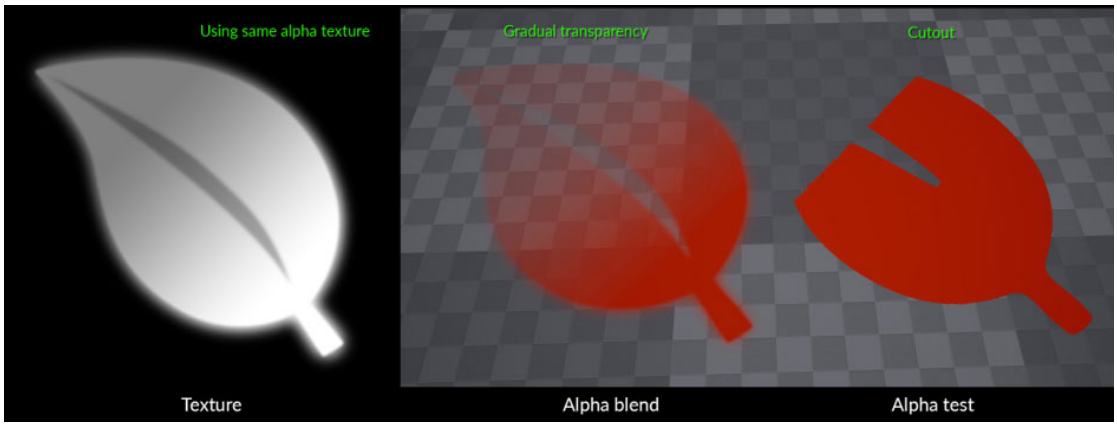


그림 7-8 알파 블렌드와 알파 테스트의 비교

### 알파 테스트

만약 게임에서 알파 테스트를 사용하고 싶다면 다음 팁을 기억하십시오.

- 필요하지 않다면 알파 테스트나 컷아웃을 사용하지 않는 것이 좋습니다.
- 알파 테스트는 머티리얼이 100% 반투명 또는 100% 투명으로 표현됩니다.
- 알파 테스트는 GPU에서 일부 최적화 기능을 사용하지 못하게 하므로 대상 모바일 플랫폼에서 사용 테스트를 할 것을 권장합니다. 또한 프로파일링으로 알파 블렌드와 성능 차이가 있는지 비교하십시오.

### 알파 블렌드

만약 게임에서 알파 블렌드를 사용하고 싶다면 다음 팁을 기억하십시오.

- 모바일 플랫폼의 경우 Unity *는 알파 테스트보다 알파 블렌드를 사용하는 것을 권장합니다*. 실제 적용 시 이 방법은 콘텐츠 의존도가 높으므로 알파 테스트와 알파 블렌드를 프로파일링하고 성능을 비교해서 수치화하는 것이 필요합니다.
- 일반적으로 모바일 플랫폼에서는 필요하지 않은 경우 알파 블렌드를 사용하지 마십시오.
- 알파 블렌드가 필요한 경우 블렌딩 영역을 적게 만드십시오.

### 나뭇잎에 알파 테스트 투명도 사용

나뭇잎의 정적 보기에서 에지가 부드러운 알파 블렌딩이 더 잘 표현됩니다. 다음 그림에서 날카롭게 베어내는 알파 테스트와 비교할 수 있습니다.

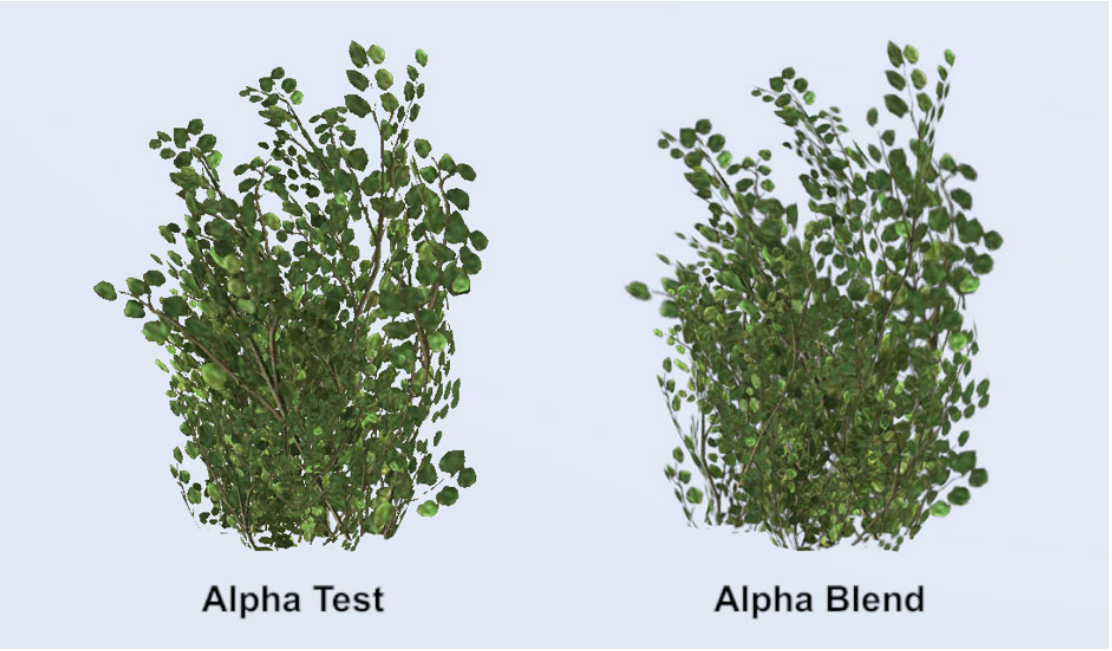


그림 7-9 알파 테스트와 알파 블렌드의 나뭇잎 예

그러나 움직이는 경우 알파 블렌드는 표현이 어색하고 정확한 순서로 렌더링되지 않을 수 있습니다. 알파 테스트는 투명도를 처리하지만 알파 블렌드는 남김을 처리합니다. 그러나 알파 블렌드에 비해 에지가 더 날카롭거나 에일리어싱입니다.

일반적으로 움직이는 경우 에일리어싱 에지가 선명하지 않아 알파 테스트도 허용 가능한 수준의 시각 품질을 보여줍니다. 그러나 알파 블렌드를 사용하면 나뭇잎이나 가지가 앞뒤로 이동해 옵티컬 일루전이 망가집니다.

## 7.7 추가 머티리얼 및 셰이더 모범 사례

ARM은 게임을 개발할 때 이 추가 모범 사례 팁을 항상 생각하도록 권장합니다.

오버드로우를 피할 수 없는 상황이라면 셰이더를 최대한 단순하게 합니다. 이 원칙을 기억하십시오.

- 언릿 등 최대한 간단한 셰이더를 사용하고 불필요한 기능은 사용하지 않습니다.
- Unity는 특별히 파티클에 맞게 개발된 셰이더가 있으므로 처음 구현할 때 좋을 것입니다.
- 오버드로우를 최소화하십시오. 파티클의 숫자와 크기를 줄이면 오버드로우 최소화에 도움이 됩니다.

### 프로파일 셰이더 복잡성

텍스처 샘플러, 투명도, 다른 기능을 더 많이 추가하면 셰이더가 복잡해지고 렌더링에 영향을 줄 수 있습니다. 그러므로 셰이더를 자주 프로파일하는 것이 좋습니다.

ARM은 *Mali 오프라인 셰이더 컴파일러* 및 *스트림라인* 같은 프로파일 툴을 제공합니다. 그러나 이런 툴로 GPU의 작동 방식을 잘 이해하기 위해서는 상위 수준의 그래픽 지식이 필요합니다. 셰이더를 편집한다면 이 분야를 더욱 깊이 공부하는 것이 좋습니다.

### 정점 셰이더에서 최대한 많은 작업을 합니다.

프로젝트가 특정 비주얼을 표현하기 위해 정점 셰이더, 픽셀 또는 프래그먼트, 셰이더를 혼용해 사용하는 것은 일반적입니다. 정점 셰이더는 모든 정점에 대해 작업하며 픽셀 셰이더는 모든 픽셀에 대해 실행됩니다.

일반적으로 화면의 정점보다 더 많은 픽셀이 렌더링됩니다. 즉 픽셀 셰이더는 정점 셰이더보다 더 자주 실행됩니다. ARM은 가능하면 픽셀 셰이더보다 정점 셰이더를 사용해 컴퓨팅을 줄이도록 권장합니다.

정점 셰이더로 작업을 이동시킨다는 것은 보통 처리된 데이터를 varying을 통해 픽셀 셰이더로 이동시킨다는 것을 말합니다. 따라서 보통은 좋은 생각이라 할지라도 병목 현상이 일어나지 않도록 타일 생성기에 주의해야 합니다. 일반적으로 최적화 작업을 마치면 추가로 프로파일링을 해야 합니다.

### 가능하면 복잡한 수학 연산은 사용하지 마십시오.

셰이더 내의 수학 연산을 사용해 룩과 동작을 맞춤 설정하십시오. 이때 사용되는 수학 연산으로는 곱셈, 덧셈, 거듭제곱, 올림, 로그, 제곱근 등이 있습니다.

이 수학 연산은 성능 비용 면에서 동일하지 않습니다. 따라서 사용 시 주의해야 합니다. sin, pow, cos, divide, noise 연산은 좀 더 복잡합니다.

덧셈이나 곱셈 같은 기본 연산은 빠르게 처리할 수 있으므로 느린 수학 연산의 수는 최소한으로 유지합니다. GLES 2.0을 사용하는 등 오래된 기기에서는 복잡한 수학 연산 사용을 적게 유지해야 합니다.

### 항상 성능 프로파일링을 합니다.

응용 프로그램에서 실제 병목 현상이 발생하는 부분을 이해하기 위해 성능 프로파일링을 하십시오. 사용한 최적화 효과 전후를 비교할 때도 프로파일링을 추천합니다.

# 제 8 장
# 실시간 3D 아트 모범 사례: 조명

이 장은 게임이 부드럽게 실행되며 시각적으로 개선되도록 돕는 여러 조명 최적화를 다룹니다.

——— 주의 ———

이 콘텐츠의 최신 버전은 Arm 개발자 웹 사이트 *https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/real-time-3d-art-best-practices-lighting*에서 확인할 수 있습니다.

———————

이 장은 다음 단원으로 구성되어 있습니다.

## 8.1 조명 소개

조명은 게임에서 가장 중요한 측면 중 하나입니다. 조명을 통해 분위기를 결정하고, 게임 플레이를 유도하며, 위협 및 목표를 식별할 수 있기 때문입니다.

조명은 게임의 비주얼을 완성하기도 하지만 파괴하기도 합니다. 예를 들어 훌륭한 조명 기술을 활용한 단순한 모델은 인게임 비주얼을 개선하지만, 서툰 조명 기술을 사용한 섬세한 모델은 인게임 비주얼 품질을 낮춥니다.

Unity 게임 엔진은 조명 작업을 단순하고 이해하기 쉽게 만들어 줍니다. 모바일 게임 성능은 조명 결정에 영향을 받으므로 조명은 효과적으로 사용해야 합니다.

이 단원에서는 다음 내용을 설명합니다.

- 정적 및 동적 라이트 간의 차이를 설명합니다.
- 라이트를 최적화합니다.
- 페이크 라이트 사용 방법을 안내합니다.
- 실시간 라이트 및 라이트 유형을 설명합니다.

## 8.2 Render Pipeline

Unity에 내장된 Render Pipeline에서는 포워드 렌더링과 디퍼드 렌더링이라는 두 가지 렌더링 경로를 제공합니다.

포워드 렌더링에서 실시간 라이트는 비용이 매우 높습니다. 그러나 이 비용을 상쇄하기 위해 픽셀당 렌더링될 라이트 수를 선택할 수 있습니다.

지연 쉐이딩은 GPU 지원이 필요하지만 가능한 하드웨어에서는 더 많은 실시간 라이트를 렌더링할 수 있으며 수준 높은 조명 재현도를 보입니다.

라이트 표현력이 좋은 PC나 콘솔 게임에서는 지연 쉐이딩이 매력적이지만 모바일 GPU에서는 좋은 성능을 내지 못합니다. 이는 양산형 장치의 대역폭이 낮기 때문입니다. 모바일 타이틀을 제작할 때는 최대한 많은 장치에서 부드럽게 실행될 수 있도록 작업하는 것이 중요합니다. 사용자를 지원하기 위해 Unity에서는 *Universal Render Pipeline*(URP)를 제공합니다.

이는 성능에 맞춰 사전 제작된 Scriptable Render Pipeline(SRP)입니다. URP는 앱 성능을 최대한 향상시킬 수 있는 다양한 기능을 제공하며 자세한 정보는 *여기*에서 확인하실 수 있습니다. 이는 조명 프로젝트에서 제안하는 최적화 방식에서 URP를 사용하기 때문입니다.

## 8.3 라이트 모드

라이트에는 여러 가지 모드가 있습니다. 이런 모드는 라이트 이동성 및 장면 내 사용 방식과 관계가 있습니다. 모드는 성능 면에서 차이가 있으므로 라이트를 구현할 때는 라이트 모드를 고려하는 것이 중요합니다.

3가지 라이트 모드(베이크드, 혼합, 실시간)를 사용했을 때의 장점과 단점을 살펴보겠습니다.

### 베이크드

베이크드 라이트 모드는 정적 조명을 제공합니다. 즉 객체는 런타임 중 조명을 변경하지 않습니다. 라이트 베이킹은 텍스터 맵의 조명 데이터를 게임 실행 전에 저장하는 과정입니다.

베이크드 라이트 모드의 핵심 기능은 다음과 같습니다.

- 런타임 중 라이트를 수정할 수 없습니다. 라이트와 그림자가 라이트맵에 베이킹됩니다. 이 과정은 Unity에서 조명이 생성될 때 완료되며 런타임 성능에 영향을 주지 않습니다.
- 그림자는 정적입니다. 즉 게임 플레이 중 동적 또는 이동하는 객체에 사용될 때 어색할 수 있습니다.
- 베이크드 라이트 모드는 이 가이드에서 다루는 방법 중 컴퓨팅 비용이 가장 저렴한 방법입니다.

### 혼합

혼합 라이트 모드는 움직이는 객체에 정적 라이트를 제공합니다. 혼합 라이트 모드는 두 개의 다른 방식을 혼합한 것으로도 볼 수 있습니다.

혼합 라이트 모드의 핵심 기능은 다음과 같습니다.

- 동적 직접 조명 및 그림자.
- 라이트는 정적 객체의 라이트맵 계산에 포함될 수 있습니다.
- 라이트는 동적 객체에 영향을 줍니다. 이때 해당 객체의 그림자 생성도 포함됩니다.
- 강도는 런타임 중 변경될 수 있지만 직접 조명만 업데이트됩니다.
- 혼합 라이트 모드는 컴퓨팅 비용이 높은 방법입니다.

### 실시간

실시간 라이트 모드는 동적 또는 이동 가능한 라이트를 제공하며 조명 방식 중 가장 비용이 높고 복잡한 방법입니다.

실시간 라이트 모드의 핵심 기능은 다음과 같습니다.

- 동적 라이트 및 그림자는 라이트맵에 베이킹된 경우와 달리 런타임 중 변경 가능한 속성을 지닙니다.
- 실시간 라이트 모드는 이 가이드에서 다루는 방법 중 컴퓨팅 비용이 가장 높은 방법입니다.

Unity 조명 파이프라인에 대한 자세한 설명은 Unity 설명서에서 확인하실 수 있습니다. *관련 정보*링크를 확인하세요.

## 8.4 최대한 정적 라이트 사용하기

모바일 장치에서 작업할 때는 정적 라이트를 사용하는 것이 핵심입니다. 이는 장치 실행 비용이 낮고 게임 최종 사용자에게 더 나은 경험을 제공합니다.

동적이나 실시간 조명은 프레임마다 계산 후 업데이트되므로 이동하는 객체에 효과적이며, 상호작용성을 높이고 장면에 감성을 부여합니다.

반면 정적 라이트 정보는 베이킹이 가능합니다. Unity는 Unity Editor로 베이크드 라이트 계산을 수행하며 그 결과를 조명 데이터로 저장합니다. 이 프로세스를 베이킹이라고 합니다.

계산이 완료되면 장면에서 필요한 것은 런타임 값뿐입니다. 정적 라이트는 동적 라이트에 비해 언제나 훨씬 낮은 비용을 사용합니다. 리소스가 제한적인 모바일 게임에서 구현할 때는 정적 라이트를 가장 우선해서 선택해야 한다는 것입니다.

## 8.5 최대한 베이킹하기

모바일 플랫폼에서 Unity를 활용할 때 가장 기본적인 접근 방식은 조명을 최대한 베이킹하는 것입니다.

라이트맵 베이킹은 조명, 그림자 등 라이트 효과를 계산하고, 이 정보를 라이트맵이라는 별도의 텍스처에 저장하는 과정입니다. 라이트맵은 객체의 외관 품질을 높이는 데 사용될 수 있습니다. 이런 베이킹 과정을 통해 당신은 오프라인 컴퓨팅을 한번만 수행하면 됩니다. 런타임 중 추가 성능 비용이 필요하지 않습니다.

프리베이크드 조명은 장면에서 동적이거나 이동하는 요소를 다루지 않습니다. 그러나 프리베이크드 조명은 모든 정적 요소에 대한 전역 조명을 포함합니다. 이는 각 정적 요소는 객체에서 반사된 간접 조명과 정적 조명의 직접 조명을 받는다는 의미입니다. 다음 이미지는 완전히 베이킹된 장면의 예를 보여줍니다.


그림 8-1 Armies의 베이크드 조명 설정을 보여주는 기술 데모

Unity에서는 라이트 베이킹이 간편합니다. 라이트를 베이킹하기 전 2개의 주요 단계를 설정해야 합니다.

1. Windows > Rendering > Lighting Settings을 클릭하고 Mixed 또는 Baked 중 베이크하려는 조명을 설정합니다.
   a. 모바일 타이틀의 경우 가능하면 혼합 라이트보다 베이크드 라이트를 사용하십시오. 이는 라이트 옵션 중 베이크드 라이트의 비용이 가장 저렴하기 때문입니다.
2. 베이크드 라이트를 받는 객체를 정적으로 표시:
   a. 정적으로 표시할 객체의 최적화 옵션은 다양하지만 이 설정에서는 일반적으로 모두를 선택합니다. 객체를 정적으로 표시하면 Unity는 라이트 베이킹에 이를 포함시킵니다.

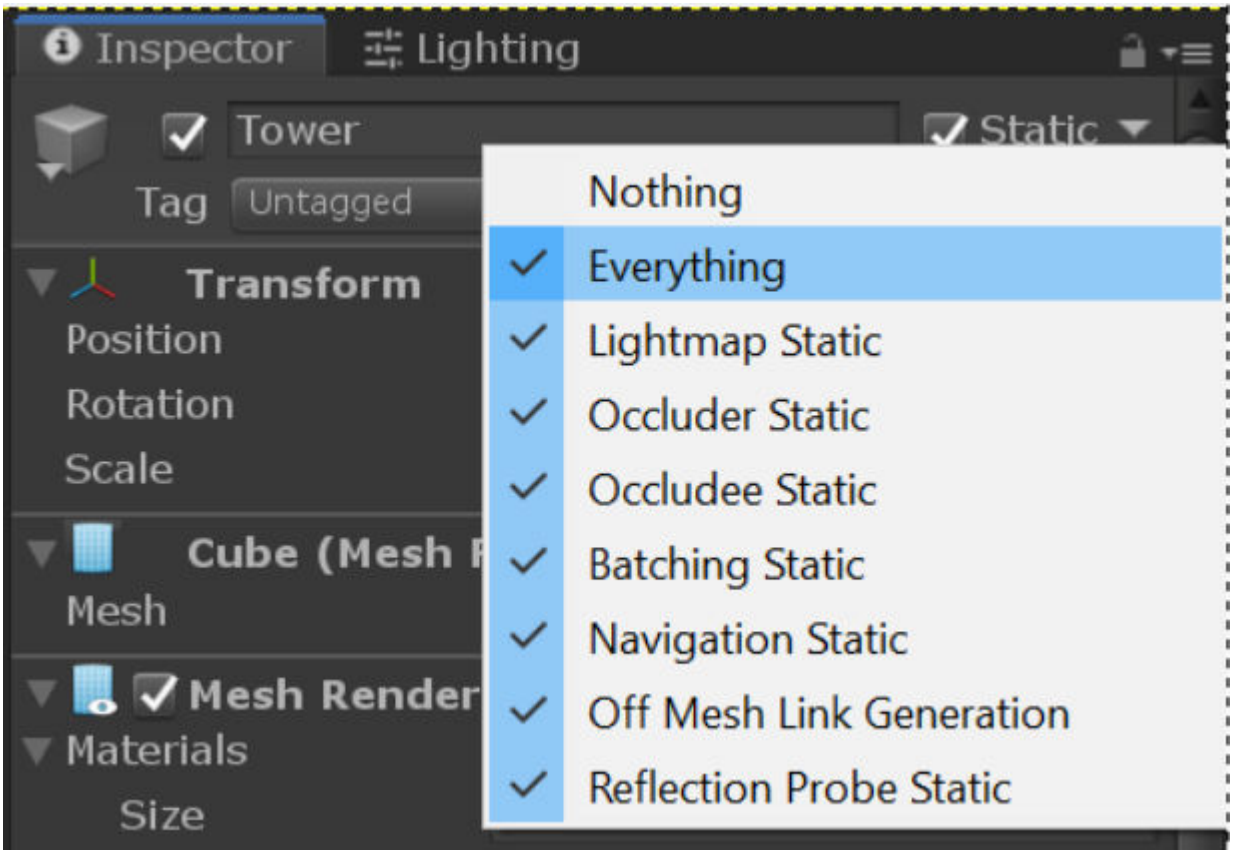


그림 8-2 베이크드 라이트를 정적으로 표시

참고

정적 배칭(Batching Static)이 활성화되면 표시된 객체를 이동시키거나 애니메이션화할 수 없습니다. 이는 또 다른 최적화 방식으로 가능한 경우 그대로 유지해야 합니다.

라이트를 베이킹할 때는 베이크를 시작할 때 활성화된 장면을 기반으로 데이터가 저장된다는 것을 기억하십시오. 방금 베이킹한 장면과 동일한 이름의 폴더가 생성됩니다. 이 폴더는 조명 데이터의 모든 구성 요소가 저장되는 장소입니다.

프로젝트가 동시에 여러 장면을 로딩하는 경우 각 장면마다 베이크드 라이트가 필요합니다. 장면을 조정하려면 라이트를 다시 베이킹해야 합니다.

다음 이미지는 폴더가 생성되는 위치를 나타냅니다.

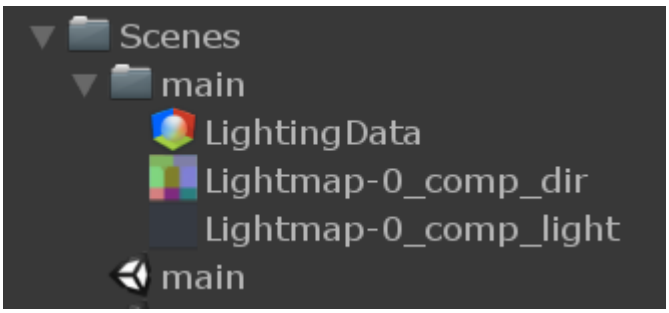


그림 8-3 베이크드 라이트 폴더

## 8.6 라이트맵 최적화

베이킹할 라이트를 구성했다면 베이킹된 맵을 최적화해야 합니다.

라이트맵은 베이킹된 설정에 따라 크기가 달라집니다. 모바일 플랫폼에서는 메모리 사용량을 최소화해야 하므로 라이트맵 크기를 모니터링할 필요가 있습니다. 다음 예시 이미지에는 1024x1024 픽셀 크기의 라이트맵 7개가 있습니다.

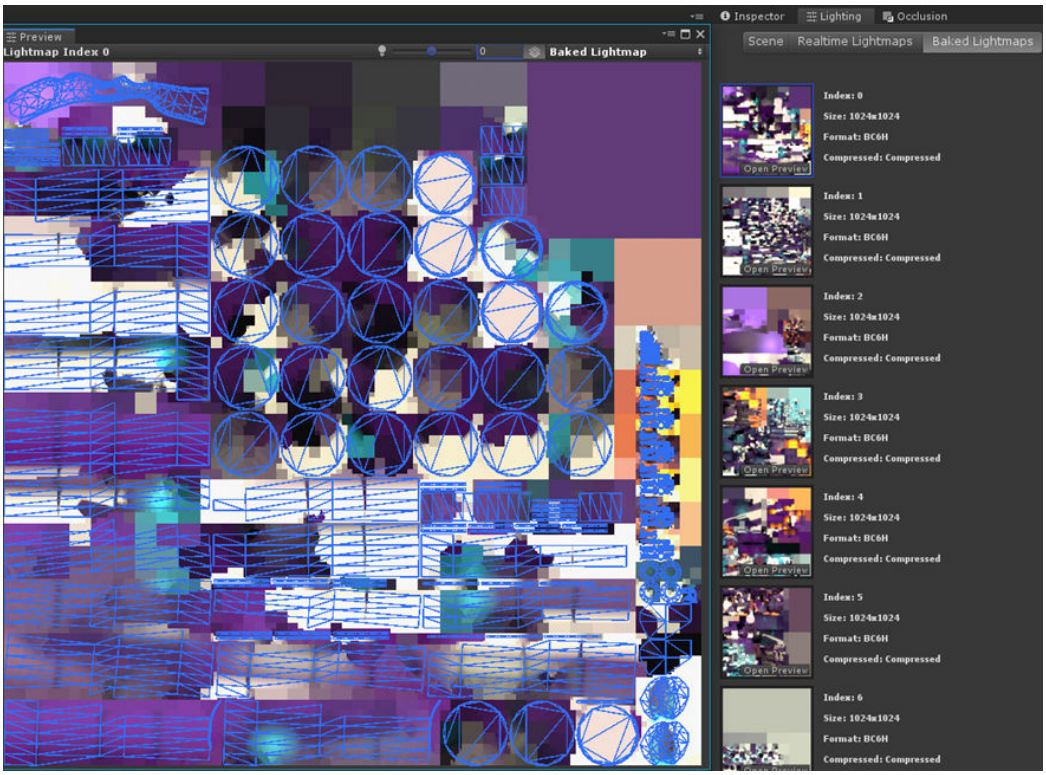


그림 8-4 베이킹된 라이트맵 인덱스 예

맵 미리보기에서는 맵에 있는 메시를 확인할 수 있고 선택된 메시가 강조됩니다.

라이트매핑(Lightmapping)의 일부 설정과 실제 맵 크기는 사용 공간을 결정합니다.

대부분의 중요 설정은 다음 단원에서 상세하게 설명합니다.

### 라이트 매퍼

Unity의 라이트 매퍼는 장면의 라이트를 베이킹할 때 세 가지 방법을 사용합니다.

- Enlighten
- 프로그레시브 CPU
- 프로그레시브 GPU

Unity의 신규 작업에는 프로그레시브 옵션 중 하나를 사용해야 합니다. 프로그레시브 라이트 매퍼는 라이트맵을 프로그레시브로 생성하므로 엄청난 시간을 절약할 수 있다는 장점이 있습니다. 보기 우선 옵션이 선택되면 장면 보기에 있는 영역을 우선합니다. 보기 우선은 장면 조명 설정 이터레이션 시간을 단축합니다.

CPU와 GPU 프로그레시브 라이트 매퍼의 가장 큰 차이는 라이트맵이 생성되는 장소가 CPU인지 GPU인지 여부입니다. 두 옵션의 결과는 동일하지만 강력한 GPU가 있다면 해당 옵션의 속도가 훨씬 빠릅니다. GPU 옵션의 요구 사항이나 설정 단계에 대한 자세한 내용은 *여기*에서 확인하십시오.

### 텍셀

텍셀 또는 텍스처 요소는 텍스처 맵의 개별 픽셀입니다. 예를 들어 텍셀은 라이트맵에서 라이트가 각 객체에 닿는 지점에 대한 조명 정보를 저장합니다. 사용되는 텍셀 수를 계산하여 라이트 베이킹에 필요한 작업량을 측정할 수 있습니다.

텍셀의 정의와 조절 방법을 파악하는 것은 중요합니다. 이는 조명 품질, 베이킹에 소요되는 컴퓨팅 시간, 디스크 저장 비용, 라이트맵의 VRAM 비용에 영향을 줄 수 있기 때문입니다.

필요한 라이트맵 데이터의 양에 가장 큰 영향을 주기 때문에 베이킹되는 각 유닛의 텍셀 수를 조정해야 합니다. 이 작업은 Lightmapping Settings에서 수행할 수 있습니다. 이 설정은 예시 이미지에서 확인할 수 있듯 베이킹에 사용되는 각 객체에 표시되는 텍셀 수 등 라이트맵을 관리할 수 있습니다.

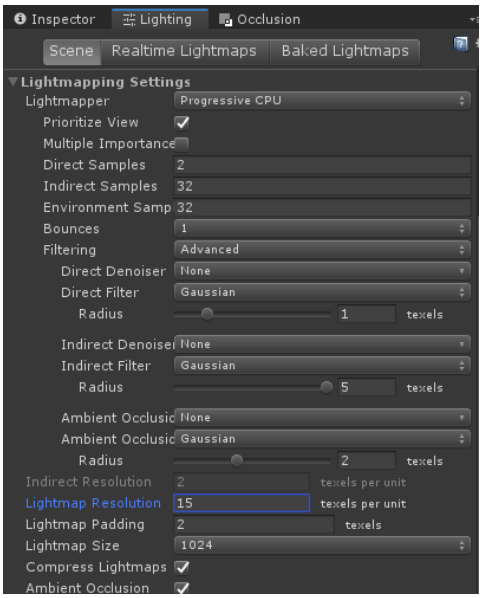


그림 8-5 라이트매핑 설정

라이트매핑 설정에는 Lightmap Resolution라는 옵션이 포함됩니다. 이 옵션은 라이트맵의 각 유닛에 사용되는 텍셀 수를 설정합니다.

장면에 텍셀이 놓이는 방식을 확인하는 방법은 다음과 같습니다.

- Scene view의 draw mode 드롭다운을 클릭합니다.
- Lightmap Indices를 찾아 클릭합니다.

이제 베이킹되는 오브젝트는 체커보드 오버레이로 덮입니다. 이는 라이트를 베이킹할 때 텍셀이 분포되는 방식입니다.

다음 스크린샷에서 다양한 Lightmap Resolution 설정에 따른 큐브의 예시를 볼 수 있습니다. 왼쪽 이미지에는 1개가 설정되어 있고, 가운데는 2개가 설정되어 있으며, 오른쪽 이미지는 5개가 설정되어 있습니다.

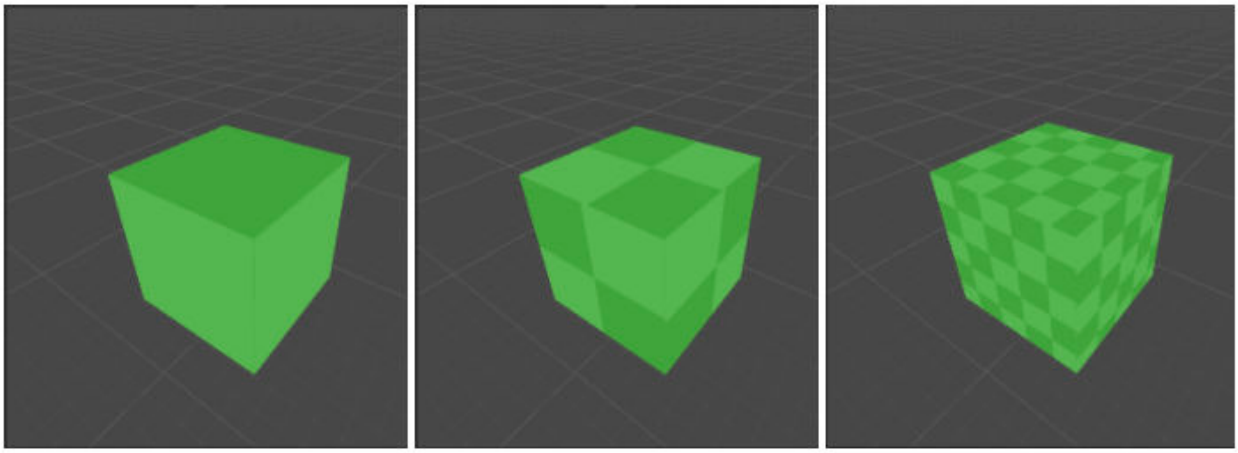

그림 8-6 라이트맵 해상도 설정

해상도가 높으면 필요한 작업량이 빠르게 증가하는 것을 볼 수 있습니다. 5~10 사이의 낮은 라이트맵 해상도에서 시작해 장면에서 요구하는 정도에 따라 해상도를 높이거나 낮추는 것이 좋습니다. 라이트맵 해상도를 높이면 각 이터레이션 크기가 엄청나게 증가합니다.

예를 들어 데모 예시의 라이트맵 해상도를 15에서 12로 낮추면 다음 이미지와 같이 필요한 라이트맵 수가 7개에서 4개로 감소합니다.

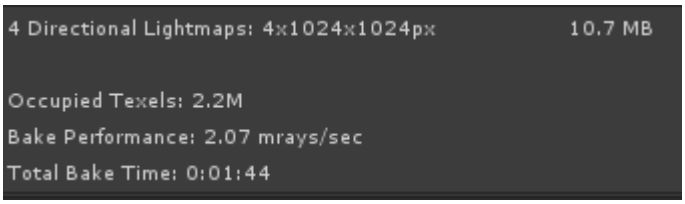


그림 8-7 라이트맵 해상도 예

### 텍셀 사용

라이트매핑 설정에서는 장면에서 유닛당 사용할 텍셀 수를 설정할 수 있는데, 텍셀을 많이 사용하지 않으려는 객체도 있을 수 있습니다.

Unity에서는 각 객체에서 사용할 수 있는 텍셀 수를 제어할 수 있습니다. 객체의 검사기 > 메시 렌더러에는 라이트맵에서 스케일이라는 매개변수가 포함됩니다. 라이트맵에서 스케일을 조정해 라이트맵의 객체가 사용할 텍셀 양을 변경합니다.

다음 스크린샷에서 왼쪽의 경우 평균적인 객체가 베이킹 단위당 라이트맵 정보가 있는 텍셀을 5개 가지고 있습니다. 이는 라이트맵 해상도가 5로 설정되었기 때문입니다. 오른쪽의 상자는 라이트맵에서 스케일이 0.5로 설정되어 있습니다.

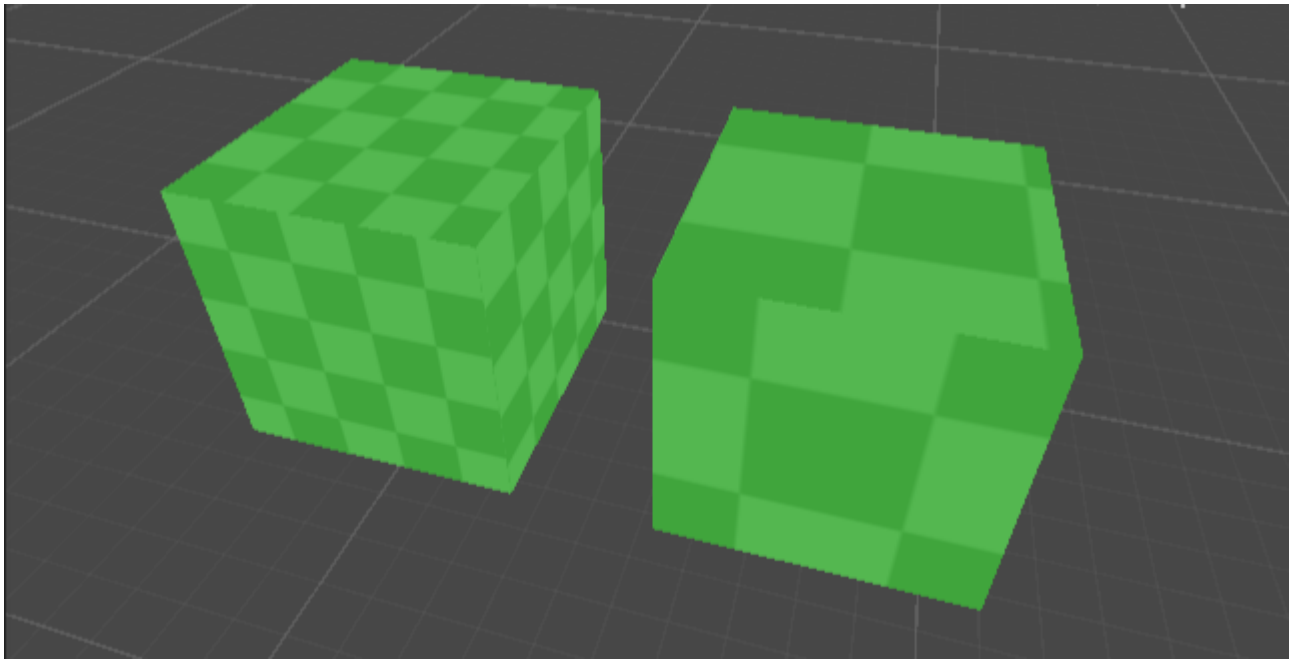

그림 8-8 라이트맵 예

라이트맵에서 오른쪽 상자는 왼쪽 상자보다 적은 공간을 사용합니다. 다음 스크린샷에서 라이트맵 설정을 확인할 수 있습니다.

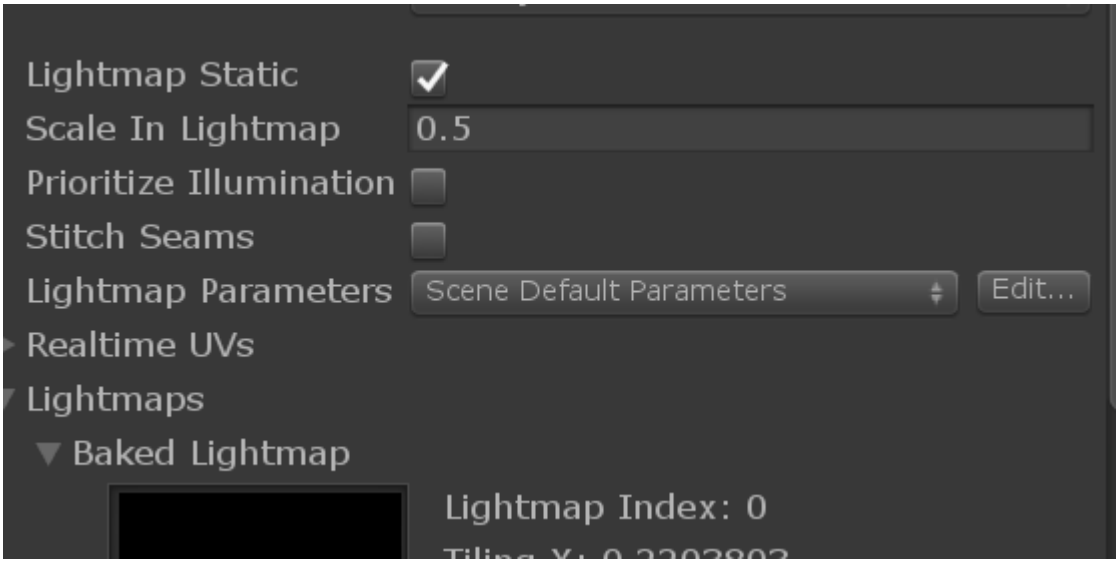


그림 8-9 라이트맵 설정 예

다음 요소에는 텍셀 사용을 피하는 것이 좋습니다.

- 플레이어가 보지 못하는 표면 및 객체. 이는 화면에 표시되지 않는 디테일로 인해 라이트맵이 커져 불필요한 메모리 소모를 방지하기 위함입니다.
- 조명 변화가 적은 표면(예: 그림자 속 객체나 단일 조명을 받는 객체).
- 작거나 얇은 객체. 조명량이 작거나 얇은 객체는 장면 최종 렌더링에서 추가될 정보가 많지 않습니다.

## 8.7 최대한 속이기

실제 그림자는 컴퓨팅 비용이 높습니다. 동적 오브젝트의 그림자를 표현하기 위해 동적 라이트에 의존하는 것보다 페이크 그림자를 구현하는 것을 추천합니다.

실시간 그림자는 주로 섀도우 맵이라는 기술을 사용해 생성됩니다. 섀도우 맵에 렌더링되는 장면 지오메트리 비용은 장면에 있는 정점 수에 따라 다릅니다. 섀도우 캐스팅 지오메트리 수와 실시간 섀도우 캐스팅 라이트의 수를 제한하는 것이 중요합니다.

페이크 그림자를 구현하는 몇 가지 방법은 다음과 같습니다.

- 캐릭터 아래에 놓이는 3D 메시, 평면, 쿼드에 블러 처리된 텍스처를 적용해 활용합니다.
- Unity에 내장된 Render Pipeline 기능을 활용해 *프로젝터*를 이용한 동적 블롭 그림자를 적용할 수 있습니다. 이 방법은 캐릭터 아래에 쿼드를 사용하는 것보다 계산 비용이 비싸며 모바일 개발에 추천하는 Universal Render Pipeline에서는 이용할 수 없습니다.
- 맞춤 쉐이더를 작성하면 더욱 정교한 블롭 그림자를 생성할 수 있습니다.

Armies의 예제 데모에 대한 다음 스크린샷에서 섀도우 메시를 사용한 섀도우 구현을 보여줍니다.

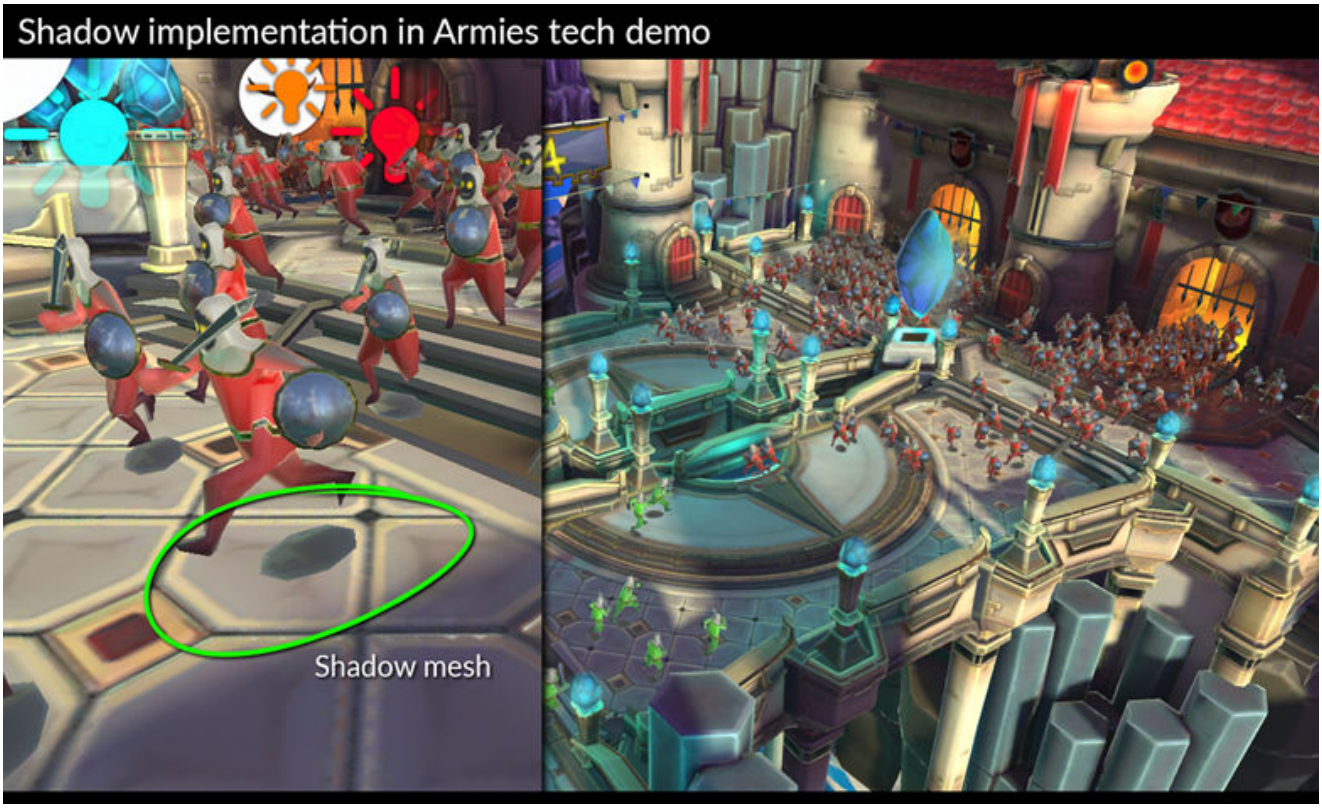


그림 8-10 그림자 구현 예

조명 정보를 텍스처에 직접 덮어씌웁니다. 라이트 쉐이딩의 일부를 텍스처에 덮어씌우면 추가 라이트에서 요구하는 추가 컴퓨팅 비용을 줄일 수 있습니다. 또한 텍스처에 직접 덮어씌우면 장면에 라이트를 베이킹할 때 텍스처 메모리를 적게 요구하기 때문에 메모리를 절약할 수 있습니다.

쉐이더나 머티리얼을 활용해 조명을 시뮬레이션합니다. 커스텀 머티리얼을 사용해 조명 효과를 시뮬레이션할 수 있습니다. 예를 들어 게임 레벨에서 캐릭터의 가시성이나 비주얼을 개선하기 위해 테두리 라이트를 사용하려는 경우가 있습니다. 이런 효과를 내기 위해 라이트를 사용하는 대신 쉐이더 효과를 활용해 라이트 일루션을 만들 수 있습니다.

쉐이더를 사용하면 게임에 유용한 효과를 다양하게 추가할 수 있습니다. 자세한 내용은 *쉐이더 및 머티리얼 모범 사례 가이드*를 참고하십시오.

## 8.8 라이트 프로브

라이트 프로브는 크게 두 가지 이유로 활용됩니다. 첫 번째는 장면에서 움직이는 오브젝트에 (간접 반사광을 포함한) 고품질 조명을 제공하는 것입니다.

라이트 프로브가 사용되는 두 번째 경우는 정적 장면이 Unity의 *Level of Detail*(LoD) 시스템을 사용하여 조명 정보를 제공해야 할 때입니다.

베이크드 조명을 활용한 동적 객체를 활용할 때 일반적으로 라이트 프로브는 라이트맵의 영향을 받지 않습니다. 그래서 어색하고 장면과 조화되지 않는 것처럼 보일 수 있습니다.

이 문제를 해결하는 것이 라이트 프로브입니다. 라이트 프로브는 오프라인으로 계산할 수 있는 조명 데이터를 저장한다는 점에서 라이트맵의 장점과 여러 가지가 동일합니다. 마찬가지로 라이트 프로브는 컴퓨팅 비용 대부분을 런타임이 아닌 편집 시로 이동합니다.

라이트맵은 장면의 표면에 대해 지정된 텍셀에서 수신된 조명을 인코딩하지만, 라이트 프로브는 장면의 빈 공간을 통과하는 라이트를 저장합니다. 이 데이터는 동적 객체 조명에도 활용될 수 있어 장면에서 라이트맵이 활용된 객체와 시각적으로 통일성을 갖추도록 돕습니다.

라이트 프로브는 정적 장면의 라이트와 그림자만을 저장하고 표시합니다. 이는 라이트 프로브 역시 프리베이크드되기 때문입니다. 이 방식은 동적 객체, 실시간 라이트, 셀프 섀도우(self-shadow)에 대한 그림자나 라이트 솔루션은 아닙니다. 그러나 라이트 프로브는 장면 대부분에서 사용되는 조명을 제공할 수 있습니다.

라이트 프로브는 다음과 같은 2가지 경우에 주로 사용됩니다.

- 장면의 움직이는 객체에 조명 사용: 라이트 프로브는 베이크드 라이트의 장점을 활용하므로 객체는 장면에서 동일한 조명을 받게 됩니다. 라이트 프로브로 동적 객체에 조명을 사용하면 실시간 라이트보다 비용이 저렴합니다.
- LoD 시스템을 활용하는 정적 표시된 객체에 조명 정보를 제공.

라이트 프로브에 대해 더 자세히 알고 싶은 경우 *관련 정보*를 확인하십시오.

다음 스크린샷은 라이트 프로브가 사용된 예를 보여줍니다.

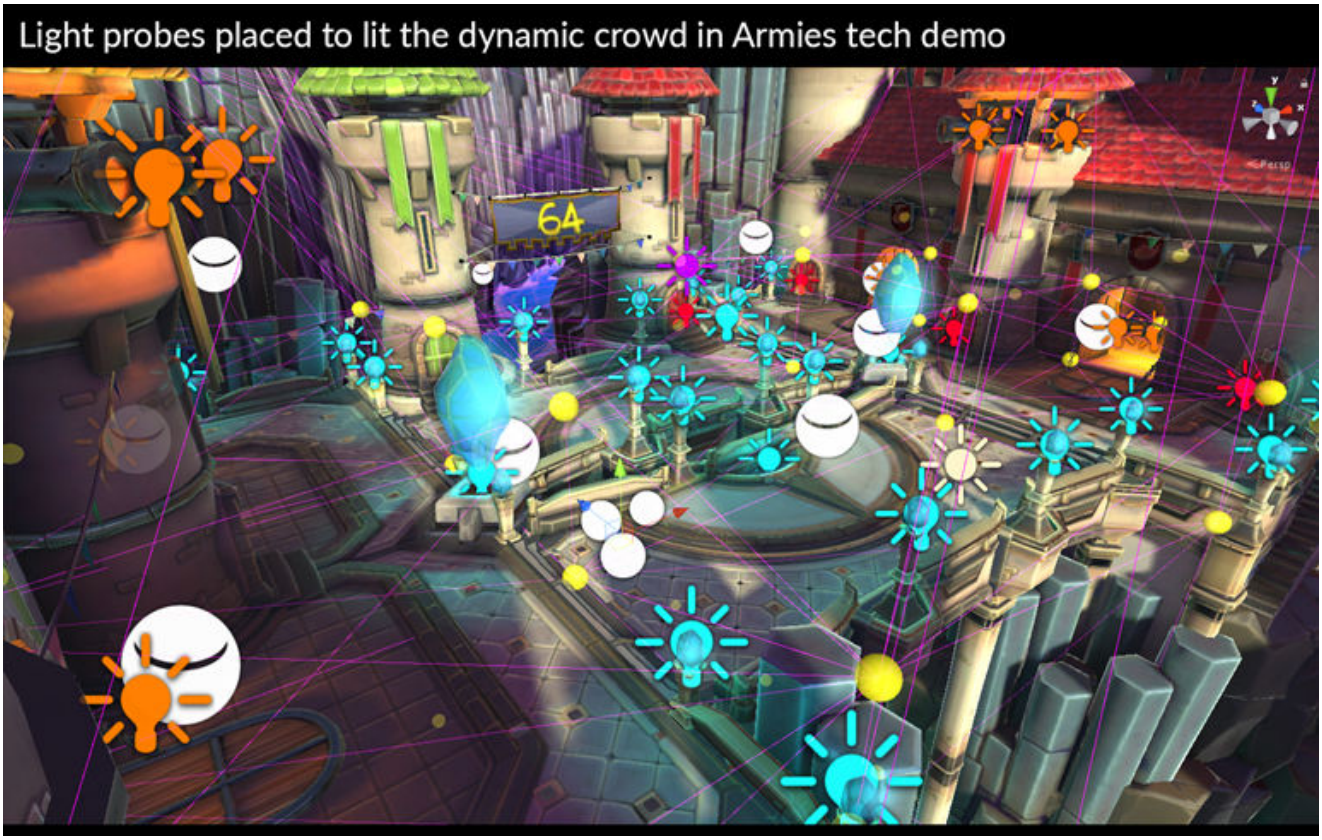


그림 8-11 라이트 프로브 예

## 8.9 메시 렌더러 설정

장면에서 사용하는 조명 종류와 관계없이 올바른 메시 렌더러 설정을 하는 것은 중요합니다.

가장 좋은 원칙은 사용하지 않는 조명을 전부 끄는 것입니다. Cast Shadows 같은 설정은 객체에 조명이 꺼진 경우에도 장면 렌더링 비용을 추가합니다. 다음 스크린샷은 장면에 Mesh Renderer 설정이 사용된 예를 보여줍니다.

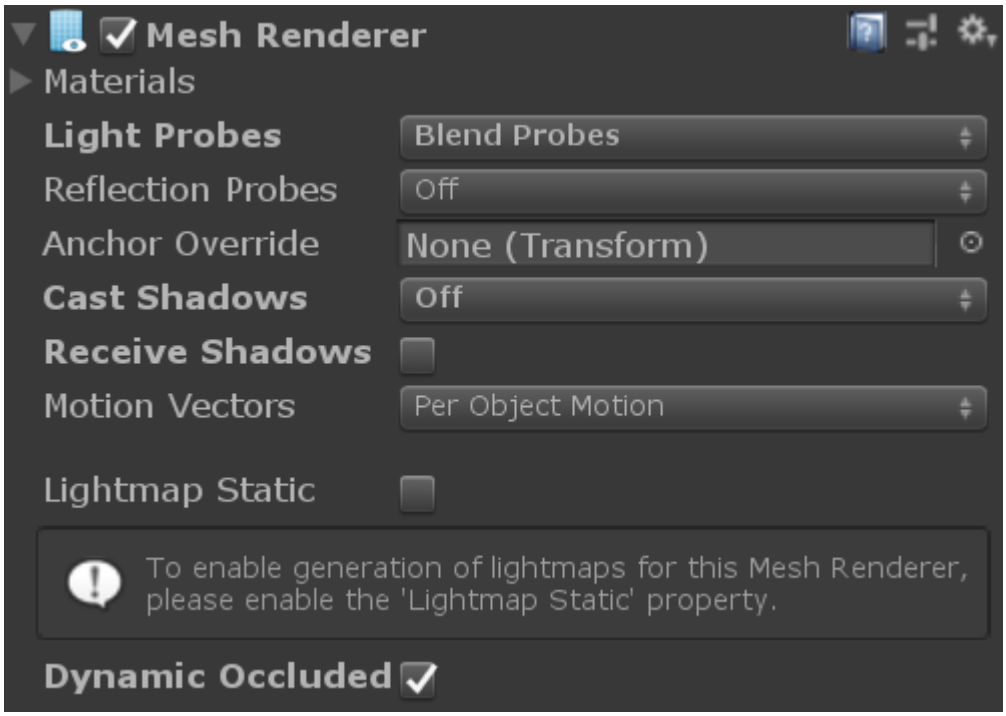


그림 8-12 메시 렌더러 설정 예

이 옵션은 이동 시 조명 정보를 가장 가까운 프로브에서 받아 부드럽게 혼합하여 반사 없이 조명만 받도록 설정되어 있습니다. 섀도우 캐스트는 블롭 방식을 사용하기 때문에 꺼져 있습니다. Receive Shadows도 꺼져 있습니다. 이 장면은 베이킹된 장면이며 실시간 섀도우 캐스트가 없기 때문입니다.

## 8.10 실시간 라이트 및 라이트 유형

베이크드 조명, 라이트 프로브, 머티리얼 효과를 활용한 라이트를 전부 다뤄보는 것이 좋습니다. 가끔 실시간 라이트가 필요할 때가 있습니다. 그런 경우 어떤 유형의 실시간 라이트를 사용할지 생각해야 합니다.

실시간 라이트 각 유형마다 계산해야 할 비용이 다릅니다.

방향성

단방향이며 밝기 저하가 없는 경우 실시간 라이트 중 가장 저렴한 것은 방향성 라이트입니다. 일반적으로 방향성 라이트는 전체 장면을 밝힐 수 있기 때문에 한 개면 됩니다. 즉 포워드 렌더링에서 Unity는 항상 한 개의 방향성 라이트만 렌더링한다는 것을 의미합니다. 장면에 방향성 라이트가 없는 경우에도 마찬가지입니다.

포인트

포인트 라이트는 공간의 한 지점에 위치하여 모든 방향으로 동일하게 조명을 비춥니다.

스폿

스폿 라이트는 구형 포인트 라이트보다 더 많은 객체를 배제하므로 스폿 라이트는 두 번째로 저렴한 실시간 조명입니다. 원주 넓이를 좁게 유지하며 선택된 객체에만 조명을 비추게 하면 스폿 라이트의 효과를 극대화할 수 있습니다.

모든 방향에 라이트를 비추는 것은 유용하지만 그 비용은 상당히 비쌉니다. 방향성 라이트는 어디서나 계산 비용이 비교적 저렴합니다. 스폿 라이트는 좁은 영역으로 비용이 비싼 부분이 제한되며, 포인트 라이트는 더 넓은 영역에 걸쳐 비용이 비쌉니다. 또한 그림자 계산은 조명에서 가장 계산 비용이 높기 때문에, 모든 방향에 조명을 비추면 비용이 증가합니다.

동적 라이트는 렌더링 비용이 비싸므로 모바일 게임에서는 피하는 것이 좋습니다. 가끔 동적 라이트는 사용되는 장치와 그래픽 API에 따라 사용이 제한될 수 있습니다. 예를 들어 OpenGL ES 2.0을 사용하는 Unity Universal Render Pipeline 포워드 렌더러에서는 객체당 4개의 라이트로 제한됩니다.

# 제 9 장
# 고급 그래픽 기법

이 장에서는 몇 가지 고급 그래픽 기법을 소개합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 9.1 사용자 지정 쉐이더

이 단원에서는 사용자 지정 쉐이더에 대해 설명합니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.1.1 사용자 지정 쉐이더 개요

Unity 5 이상에는 자료와 라이트 간 상호 작용을 시뮬레이션하는 Physically Based Shading(*PBS*) 모델이 포함되어 있습니다. 이 모델은 고도의 사실성을 제공하며 다양한 조명 상태에서 일관적인 모양을 구현할 수 있습니다.

표준 쉐이더에서 PBS를 사용할 수 있습니다. 자체 자료를 생성하는 경우 자동으로 표준 쉐이더로 할당됩니다.

표준 쉐이더는 용이하게 액세스할 수 있습니다. 자체 자료를 생성하는 경우 표준 쉐이더가 해당 자료에 할당됩니다. 초심자에게 매우 유용한 다른 내장 쉐이더가 다수 제공됩니다. 검사기에서 shader 드롭다운 메뉴를 클릭하면 사용 가능한 모든 내장 쉐이더가 계열로 구분되어 표시됩니다.

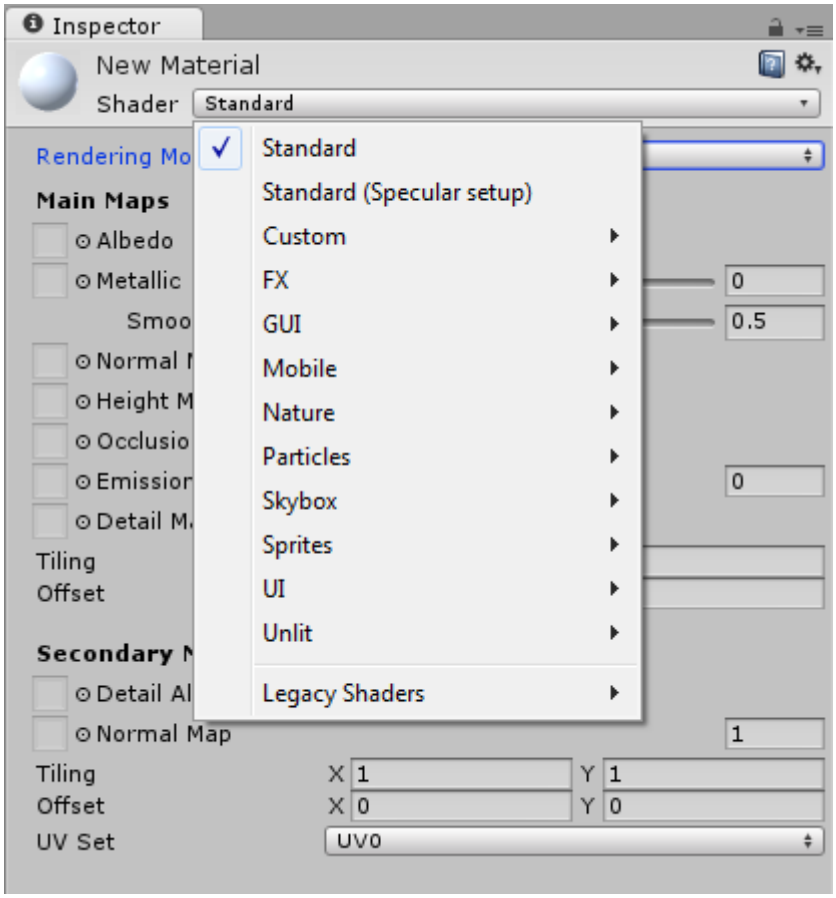


그림 9-1 Unity 내장 쉐이더

내장 쉐이더의 소스 코드는 120여 쉐이더가 포함되어 있는 Unity 다운로드 아카이브 *http://unity3d.com/*에서 확인할 수 있습니다. 이러한 쉐이더의 코드를 읽고 이해하려고 노력하는 과정에서 많은 것을 배울 수 있습니다.

이밖에 기존 쉐이더로는 구현할 수 없는 효과가 많이 있습니다. 로컬 큐브맵을 기반으로 반사를 구현하는 쉐이더가 한 예입니다. 자세한 내용은 *9.2 로컬 큐브맵을 사용하여 반사 구현* 페이지의 9-153을 참조하십시오.

Unity에서 쉐이더를 작성하는 방법은 기본적으로 두 가지입니다.

표면 쉐이더
: 이 쉐이더는 쉐이더가 라이트와 그림자에 의해 영향을 받을 때 흔히 사용됩니다. Unity가 사용자 대신 조명 모델과 관련된 작업을 수행하여 보다 콤팩트하게 쉐이더를 작성할 수 있게 해줍니다.

정점 및 조각 쉐이더
: 이들 쉐이더는 가장 유연한 쉐이더이지만 사용자가 모든 것을 구현해야 합니다. Unity ShaderLab이 정점 및 조각 쉐이더보다 많은 기능을 수행하지만, 이들 쉐이더는 모든 쉐이딩이 이루어지는 그래픽 파이프라인 내 주요 프로그래밍 부분에 포함되므로 사용자 지정 정점 및 조각 쉐이더를 작성하는 방법을 아는 것이 중요합니다.

### 9.1.2 쉐이더 구조

다음 코드는 정점 또는 조각 쉐이더에 필요한 요소를 대부분 포함하는 매우 간단한 정점 및 조각 쉐이더를 보여줍니다.

이 쉐이더 예제는 Cg로 작성된 것입니다. Unity는 쉐이더 코드 조각용으로 HLSL 언어도 지원합니다.

```
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"

            // User-specified uniforms
            uniform float4 _AmbientColor;
            uniform sampler2D _MainTex;

            struct vertexInput
            {
                float4 vertex : POSITION;
                float4 texCoord : TEXCOORD0;
            };
            struct vertexOutput
            {
                float4 pos : SV_POSITION;
                float4 tex : TEXCOORD0;
            };

            // Vertex shader.
            vertexOutput vert(vertexInput input)
            {
                vertexOutput output;

                output.tex = input.texCoord;
                output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                return output;
            }

            // Fragment shader.
            float4 frag(vertexOutput input) : COLOR
            {
                float4 texColor = tex2D(_MainTex, float2(input.tex));
                return _AmbientColor + texColor;
            }

            ENDCG
        }
    }
    Fallback "Diffuse"
}
```

첫 번째 키워드는 Shader이고 쉐이더의 *path/name* 이 이어집니다. 경로는 자료를 설정할 때 드롭다운 메뉴에서 쉐이더가 표시되는 카테고리를 정의합니다. 이 예제에서 쉐이더는 드롭다운 메뉴에서 Custom 쉐이더 카테고리 아래에 표시됩니다.

Properties{} 블록은 검사기에서 표시되는 쉐이더 매개 변수와 사용자가 상호 작용할 수 있는 매개 변수를 나열합니다.

Unity에서 각 쉐이더는 일련의 서브 쉐이더로 구성됩니다. Unity는 메시를 렌더링할 때 사용할 쉐이더를 검색하여 그래픽 카드에서 실행될 수 있는 첫 번째 서브 쉐이더를 선택합니다. 이러한 방식으로 다양한 쉐이더 모델을 지원하는 다양한 그래픽 카드에서 쉐이더가 올바로 실행됩니다. GPU 하드웨어 및 API는 계속 발전하고 있으므로 이 기능이 중요합니다. 예를 들어 Mali Midgard GPU를 타겟으로 메인 쉐이더를 작성하여 OpenGL ES 3.0의 최신 기능을 사용하고, 별도의 서브 쉐이더에서 OpenGL ES 2.0 이하를 지원하는 그래픽 카드용으로 대체 쉐이더를 작성할 수 있습니다.

Pass 블록은 객체의 지오메트리를 한 번에 렌더링하게 해줍니다. 쉐이더는 하나 이상의 패스를 포함할 수 있습니다. 구형 하드웨어에서, 또는 특수 효과를 구현하기 위해 다중 패스를 사용할 수 있습니다.

Unity가 지오메트리를 올바로 렌더링할 수 있는 쉐이더의 본문에서 서브 쉐이더를 찾지 못할 경우 Fallback 문 뒤에 정의된 다른 쉐이더로 롤백합니다. 예제에서는 Diffuse 내장 쉐이더가 그것입니다.

Cg 프로그램 코드 조각은 CGPROGRAM과 ENDCG 사이에 작성됩니다.

### 9.1.3 컴파일 지시문

컴파일 지시문을 #pragma 문으로 전달합니다. 컴파일 지시문은 컴파일할 쉐이더 함수를 가리킵니다.

각 컴파일 지시문은 적어도 정점과 조각 쉐이더를 컴파일하는 지시문을 포함해야 합니다. #pragma vertex name, #pragma fragment name.

기본적으로 Unity는 쉐이더를 쉐이더 모델 2.0으로 컴파일합니다. 지시문 #pragma target을 사용하면 쉐이더를 다른 기능 레벨로 컴파일할 수 있습니다. 쉐이더 크기가 커질 경우 다음 유형의 오류가 발생합니다.

```
Shader error in 'Custom/MyShader': Arithmetic instruction limit of 64 exceeded; 83
arithmetic instructions needed to compile program;
```

이 경우 #pragma target 3.0 문을 추가하여 쉐이더 모델 2.0에서 쉐이더 모델 3.0으로 변경해야 합니다. 쉐이더 모델 3.0은 명령어 제한이 훨씬 큽니다.

정점 쉐이더에서 조각 쉐이더로 여러 varying 변수를 전달할 경우 다음과 같은 오류가 발생할 수 있습니다.

```
Shader error in 'Custom/MyShader': Too many interpolators used (maybe you want
#pragma glsl?) at line 75.
```

이 경우 컴파일 지시문 #pragma glsl을 추가하십시오. 이 지시문은 Cg 또는 HLSL 코드를 GLSL로 변환합니다.

#pragma only_renderers 지시문.

Unity는 gles, gles3, opengl, d3d11, d3d11_9x, xbox360, ps3, flash 등 여러 렌더링 플랫폼을 지원합니다. 기본적으로 쉐이더는 사용자가 #pragma only_renderers를 사용하고 그 다음에 공백으로 구분한 원하는 렌더 API를 추가하여 이 숫자를 명시적으로 제한하지 않는 한 이들 플랫폼 전부로 컴파일됩니다.

모바일 디바이스를 타겟으로 하는 경우 쉐이더 컴파일을 gles 및 gles3으로 제한하십시오. Unity 편집기에서 사용하는 opengl 및 d3d11 렌더러도 추가해야 합니다.

```
#pragma only_renderers gles gles3 [opengl, d3d11]
```

### 9.1.4 Include

Unity 사전 정의 변수 및 도우미 함수를 사용하여 쉐이더에서 include 파일을 추가할 수 있습니다.

C:\Program Files \Unity\Editor\Data\CGIncludes에서 사용 가능한 포함 파일을 확인할 수 있습니다. 예를 들어 포함 파일 UnityCG.cginc에서 수많은 표준 셰이더에 사용되는 여러 가지 유용한 도우미 함수와 매크로를 찾을 수 있습니다. 이들을 사용하려면 셰이더에 포함 파일을 선언하십시오.

다수의 Unity 기본 변수를 쉐이더에서 사용할 수 있습니다. 이러한 변수는 include UnityShaderVariables.cginc에 포함되어 있습니다. Unity가 자동으로 이 파일을 쉐이더에 포함시키므로 사용자가 직접 포함시킬 필요는 없습니다. 다수의 유용한 변환 매트릭스 및 크기를 쉐이더에서 직접 사용할 수 있습니다. 중복 작업을 피하기 위해 이러한 매트릭스 및 크기를 모두 숙지하는 것이 중요합니다. 예를 들어 특정 매트릭스를 어떻게 쉐이더, 카메라 위치 또는 투

영 매개 변수 또는 라이트 매개 변수로 전달할지 고려하기 전에 이러한 매트릭스를 제공하는 include가 이미 존재하는지 확인하십시오.

성능을 개선하기 위해 때로는 정점 쉐이더에서 모든 정점에 대한 연산을 실행하는 대신 CPU에서 연산을 실행하고 결과를 GPU에 전달하는 것이 바람직한 경우도 가끔 있습니다. 예를 들어 매트릭스 uniform 변수의 곱셈이 그렇습니다. Unity가 다수의 복합 매트릭스를 기본 uniform 변수로 제공하는 이유가 이 때문입니다. 다음 표에 일부 중요한 Unity 쉐이더 기본 값이 나와 있습니다.

표 9-1 중요 Unity 쉐이더 기본 값

| 기본 Uniform 변수 | 설명 |
|---|---|
| UNITY_MATRIX_V | 현재 뷰 매트릭스 |
| UNITY_MATRIX_P | 현재 투영 매트릭스 |
| Object2World | 현재 모델 매트릭스 |
| _World2Object | 현재 월드 매트릭스의 역 |
| UNITY_MATRIX_VP | 현재 뷰 * 투영 매트릭스 |
| UNITY_MATRIX_MV | 현재 모델 * 뷰 매트릭스 |
| UNITY_MATRIX_MVP | 현재 모델 * 뷰 * 투영 매트릭스 |
| UNITY_MATRIX_IT_MV | 현재 모델 * 뷰 매트릭스의 역 전치 |
| _WorldSpaceCameraPos | 월드 공간 내 카메라 위치 |
| _ProjectionParams | 벡터의 성분으로서 근거리 평면, 원거리 평면 및 1/원거리 평면 |
| _Time | 벡터 내 현재 시간(t/20, t, t*2, t*3) |

### 9.1.5 OpenGL ES 3.0 그래픽 파이프라인

그래픽 파이프라인에서 프로그래밍 가능한 정점 및 조각 쉐이더가 어디에 위치하는지 아는 것이 중요합니다.

다음 그림은 OpenGL ES 3.0 그래픽 파이프라인 흐름을 개략적으로 보여줍니다. OpenGL ES 3.0은 임베디드 그래픽의 진화 과정에서 중요한 단계이며 OpenGL 3.3 사양에서 파생되었습니다.

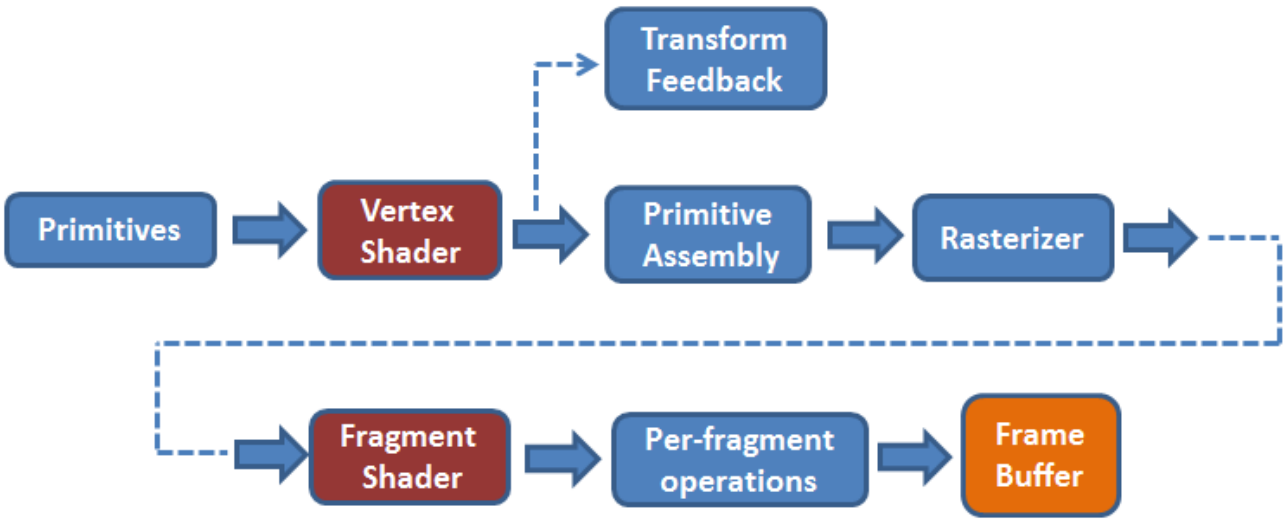


그림 9-2 OpenGL ES 3.0 프로그래밍 가능 파이프라인

프리미티브
: 프리미티브 단계에서 파이프라인은 정점, 점, 직선 및 폴리곤에 의해 설명되는 기하학적 프리미티브에서 작동합니다.

정점 쉐이더
: 정점 쉐이더는 정점에 대해 연산하기 위한 프로그래밍 가능 범용 메서드를 구현합니다. 정점 쉐이더는 정점을 변환하고 조명합니다.

프리미티브 어셈블리
: 프리미티브 어셈블리에서 정점은 기하학적 프리미티브로 어셈블됩니다. 결과 프리미티브는 클리핑 볼륨에 클립되어 래스터라이저로 보내집니다.

래스터화
: 정점 쉐이더의 출력 값은 생성된 모든 조각에 대해 계산됩니다. 이 프로세스를 보간이라고 합니다. 래스터화하는 동안 프리미티비는 일련의 2D 조각으로 변환된 다음 조각 쉐이더로 보내집니다.

변환 피드백
: 변환 피드백은 정점 쉐이더가 출력하고 나중에 정점 쉐이더로 다시 보내지는 출력 버퍼에 대한 선택적 쓰기를 쓸 수 있도록 활성화합니다. 이 기능은 Unity에 의해 표시되는 것이 아니라 예를 들어 캐릭터의 스키닝을 최적화하기 위해 내부적으로 사용됩니다.

조각 쉐이더
: 조각 쉐이더는 다음 단계로 보내기 전에 조각에 대해 연산하기 위한 프로그래밍 가능 범용 메서드를 구현합니다.

조각별 연산
: 조각별 연산에서는 각 조각에 여러 함수 및 테스트가 적용됩니다. 픽셀 소유권 테스트, 가위 테스트, 스텐실 및 깊이 테스트, 블렌딩, 디더링. 이 조각별 단계의 결과로 각 조각이 폐기되거나 조각 색상, 깊이 또는 스텐실 값이 화면 좌표로 프레임 버퍼에 기록됩니다.

### 9.1.6 Vertex shaders

정점 쉐이더 예제는 지오메트리의 각 정점에 대해 한 번만 실행됩니다. 정점 쉐이더의 목적은 객체의 로컬 좌표로 주어지는 각 정점의 3D 위치를 화면 공간의 투영된 2D 위치로 변환하고 Z 버퍼 깊이 값을 계산하는 것입니다.

정점 쉐이더 예제 샘플 코드는 *9.1.2 쉐이더 구조* 페이지의 9-141를 참조하십시오.

변환된 위치는 정점 쉐이더의 출력에서 예상됩니다. 정점 쉐이더가 값을 반환하지 않을 경우 콘솔에 다음 오류가 표시됩니다.

```
Shader error in 'Custom/ctTextured': '' : function does not return a value: vert at
line 36
```

예제에서 정점 쉐이더는 로컬 공간 정점 좌표와 텍스처 좌표를 입력으로 수신합니다. 정점 좌표는 Unity 기본 값인 Model View Projection 매트릭스 `UNITY_MATRIX_MVP`를 사용하여 로컬 공간에서 화면 공간으로 변환됩니다.

```
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
```

텍스처 좌표는 조각 쉐이더에 varying 변수로 전달되지만, 이는 이들 좌표가 변환되지 않는다는 의미는 아닙니다.

노말은 다른 방식으로 객체 공간에서 월드 공간으로 변환됩니다. 노말이 차등 스케일링 연산 이후에도 삼각형에 대해 노말이 되도록 보장하려면 노말에 변환 매트릭스의 역의 전치를 곱해야 합니다. 전치 연산을 적용하려면 곱셈에서 인수의 순서를 뒤집습니다. 로컬-월드 매트릭스의 역은 기본 `World2Object` Unity 매트릭스입니다. 이 매트릭스는 4x4 매트릭스이므로 3성분 노멀 입력 벡터에서 4성분 벡터를 생성합니다.

```
float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);
```

4성분 벡터를 생성할 때 4번째 성분으로 0을 추가합니다. 이 과정이 4차원 공간에서 벡터 변환을 올바로 처리하기 위해 필요하며, 좌표에 대해서는 4번째 성분이 반드시 하나의 유닛이어야 합니다.

노말이 이미 월드 좌표로 제공되는 경우 노말 변환 프로세스를 건너뛸 수 있습니다. 그러면 정점 쉐이더에서 작업이 줄어듭니다. 객체 메시가 Unity 내장 쉐이더에 의해 처리될 가능성이 있는 경우 이 힌트를 사용하지 마십시오. 이런 경우에는 노말이 객체 좌표로 예상되기 때문입니다.

대부분의 그래픽 효과는 조각 쉐이더에서 구현되지만 일부 효과는 정점 쉐이더에서도 구현할 수 있습니다. 정점 변위 매핑(변위 매핑이라고도 함)은 예를 들어 높이 맵을 사용하여 지형 생성 시 표면 디테일을 추가하는 경우와 같이 텍스처를 사용하여 폴리곤 메시를 변형(deform)시킬 수 있는 잘 알려진 기법입니다. 정점 쉐이더에서 변위 맵이라고도 하는 이 텍스처에 액세스하려면 pragma 지시문 `#pragma target 3.0`을 추가해야 합니다. 이것이 쉐이더 모델 3.0에서는 유일하게 사용 가능하기 때문입니다. 쉐이더 모델 3.0에 따라 적어도 4개의 텍스처 유닛이 정점 쉐이더 내부에서 액세스 가능해야 합니다. 편집기가 OpenGL 렌더러를 사용하도록 강제 적용할 경우에는 `#pragma glsl` 지시문도 추가해야 합니다. 이 지시문을 선언하지 않을 경우 다음과 같이 이에 대한 오류 메시지가 표시됩니다.

```
Shader error in 'Custom/ctTextured': function "tex2D" not supported in this profile
(maybe you want #pragma glsl?) at line 57
```

또한 정점 쉐이더에서는 “절차적 애니메이션(procedural animation)” 기법을 사용하여 정점을 애니메이션화할 수 있습니다. 쉐이더에서 시간 변수를 사용하면 시간의 함수로서 정점 좌표를 수정할 수 있습니다. 메시 스키닝은 정점 쉐이더에서 구현되는 또 하나의 기능입니다. Unity는 이 기능을 사용하여 캐릭터 골격과 연결된 메시의 정점을 애니메이션화합니다.

### 9.1.7 정점 쉐이더 입력

정점 쉐이더의 입력 및 출력은 구조를 통해 정의됩니다. 예제의 입력 구조에서는 정점 속성 위치 및 텍스처 좌표만 선언합니다.

다음 시맨틱을 사용하여 예를 들어 두 번째 텍스처 좌표 세트, 객체 좌표의 노말, 색상, 탄젠트 등 더 많은 속성을 입력으로 정의할 수 있습니다.

```
struct vertexInput
{
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    fixed4 color : COLOR;
};
```

시맨틱은 매개 변수의 사용에 관한 정보를 제공하는, 쉐이더 입력 또는 출력과 연결된 문자열입니다. 쉐이더 단계 사이에서 전달되는 모든 변수에 대해 시맨틱을 지정해야 합니다.

`float3 tangent2 : TANGENTIAL`과 같이 잘못된 시맨틱을 사용할 경우 다음과 같은 유형의 오류가 발생합니다.

```
Shader error in 'Custom/ctTextured': unknown semantics "TANGENTIAL" specified for
"tangent2" at line 32
```

성능을 위해 반드시 필요한 입력 구조에서만 매개 변수를 지정해야 합니다. Unity는 다음과 같이 가장 일반적인 입력 매개 변수 조합에 대해 미리 정의된 입력 구조를 제공합니다. `appdata_base`, `appdata_tan` 및 `appdata_full`. 이들 구조는 `UnityCG.cginc` include 파일에 설명되어 있습니다. 위의 정점 입력 구조 예제는 `appdata_full`에 해당합니다. 이 경우에서는 구조는 선언할 필요가 없고 include 파일만 선언합니다.

### 9.1.8 정점 쉐이더 출력 및 varying 변수

정점 쉐이더 출력은 반드시 정점 변환 좌표를 포함하는 출력 구조로 정의됩니다. 다음 예제에서는 출력 구조가 매우 단순하지만 다른 크기를 추가할 수 있습니다.

다음 코드는 Unity에서 지원하는 시맨틱(semantics)을 나열합니다.

```
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;
};
```

변환된 정점 좌표는 시맨틱 SV_POSITION을 사용하여 정의됩니다. 텍스처 2개, 여러 벡터, 그리고 시맨틱 TEXCOORDn을 호출하는 다양한 공간의 좌표도 조각 쉐이더로 전달됩니다.

TEXCOORD0은 일반적으로 UV용으로 예약되고 TEXCOORD1은 라이트맵 UV용으로 예약됩니다. 하지만 기술적으로는 모든 데이터를 TEXCOORD0~TEXCOORD7에서 조각 쉐이더로 보낼 수 있습니다. 각 보간기, 즉 각 시맨틱은 최대 4개의 float를 처리할 수 있다는 점을 유의해야 합니다. 매트릭스와 같이 더 큰 변수는 여러 보간기에 전달합니다. 이는 매트릭스를 varying 변수로 전달되도록 정의할 경우(float4x4 myMatrix : TEXCOORD2) Unity는 보간기 TEXCOORD2~TEXCOORD5를 사용합니다.

정점 쉐이더에서 조각 쉐이더로 전달하는 모든 데이터는 기본적으로 선형으로 보간됩니다. 정점 V1, V2 및 V3에 의해 정의되는 삼각형 안의 모든 픽셀에 대해 그래픽 파이프라인에서 정점 쉐이더와 조각 쉐이더 사이에 위치하는 래스터라이저가 무게 중심 좌표 $\lambda 1$, $\lambda 2$, $\lambda 3$를 사용하여 정점 좌표의 선형 보간으로 픽셀 좌표를 계산합니다.

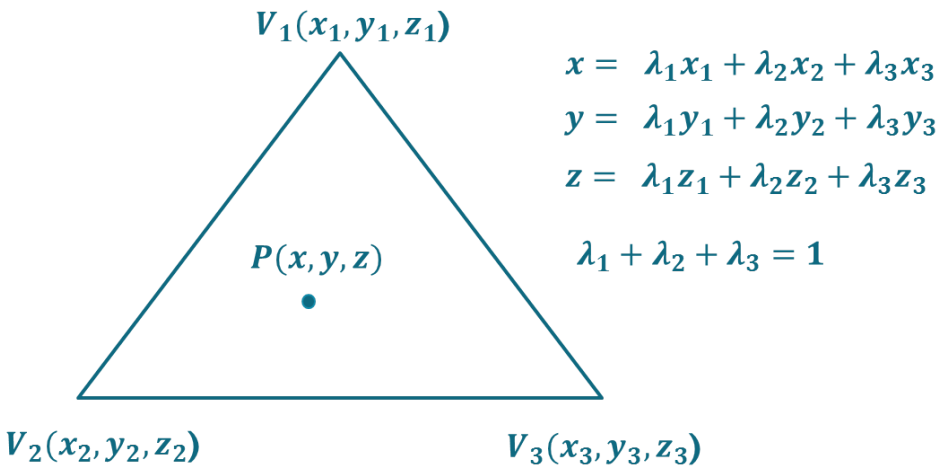


그림 9-3 무게 중심 좌표를 사용한 선형 보간

다음 다이어그램은 정점 색상이 빨간색, 녹색, 파란색인 삼각형에서 색상 보간의 결과를 보여줍니다.

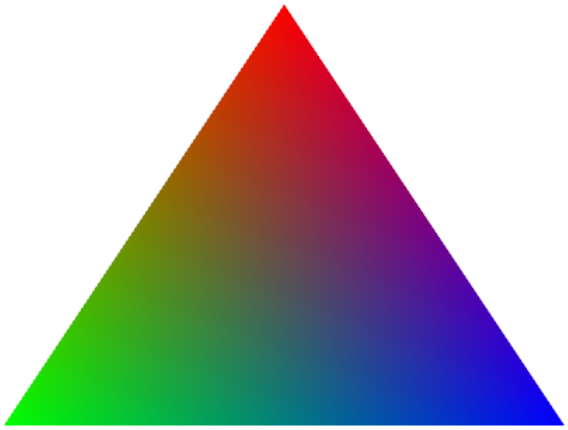

그림 9-4 색상 보간

조각 쉐이더로 전달되는 모든 varying 변수에 동일한 보간이 적용됩니다. 하드웨어 선형 보간기가 있기 때문에 이는 매우 강력한 도구입니다. 예를 들어 평면에서 중심 C와의 거리의 함수로서 색을 적용하려는 경우, 중심 C의 좌표를 정점 쉐이더로 전달하여 정점과 C 사이의 거리를 제곱한 다음 이 크기를 조각 쉐이더에 전달합니다. 모든 삼각형의 모든 픽셀에서 거리 값이 자동으로 보간됩니다.

값은 선형으로 보간되므로 정점별 계산을 수행하고 조각 쉐이더에서 재사용할 수 있습니다. 즉, 조각 쉐이더에서 선형으로 보간될 수 있는 값은 정점 쉐이더에서 대신 계산될 수 있습니다. 정점 쉐이더는 조각 쉐이더보다 훨씬 작은 데이터 세트에서 실행될 수 있으므로 이를 통해 상당한 성능 향상이 가능합니다.

특히, 많은 게임에서 성능 및 메모리 대역폭 소비가 매우 중요한 성공 요소가 되는 모바일에서는 특히 varying 변수를 신중하게 사용해야 합니다. varying 변수가 많을수록 정점 액세스 및 조각 쉐이더 varying 변수 읽기 대역폭이 증가합니다. varying 변수를 사용할 때 적절한 균형을 추구하십시오.

### 9.1.9 조각 쉐이더

조각 쉐이더는 프리미티브 래스터화 이후의 그래픽 파이프라인 단계입니다.

프리미티브에 의해 커버되는 픽셀의 각 샘플에 대해 조각이 생성됩니다. 조각 쉐이더 코드는 생성된 각 조각에 대해 실행됩니다. 조각은 정점보다 훨씬 많으므로 조각 쉐이더에서 실행되는 연산 횟수에 주의를 기울여야 합니다.

조각 쉐이더에서는 무엇보다 정점 쉐이더의 모든 보간된 정점별 출력 값을 포함하는 창 공간에서 조각 좌표에 액세스할 수 있습니다.

*9.1.2 쉐이더 구조* 페이지의 9-141의 쉐이더 예제에서는 조각 쉐이더가 정점 쉐이더로부터 보간된 텍스처 좌표를 수신하고 텍스처 검색을 수행하여 이들 좌표에서 색상을 구합니다. 조각 쉐이더는 이 색상을 주변 색상과 결합하여 최종 출력 색상을 생성합니다. 조각 쉐이더 `float4 frag(vertexOutput input) : COLOR`의 선언에서 조각 색상이 생성될 것임을 분명히 알 수 있습니다. 조각 쉐이더는 연산을 실행하여 필요한 효과를 구현하는 위치입니다. 이 쉐이더는 궁극적으로 조각에 올바른 색상을 할당하는 프로세스로 이루어집니다.

### 9.1.10 쉐이더에 데이터 제공

Pass 블록에서 uniform 변수로 선언된 데이터는 정점 및 조각 쉐어더 모두에 사용할 수 있습니다.

uniform 변수는 쉐이더 내에서 수정할 수 없기 때문에 일종의 전역 상수 변수로 볼 수 있습니다.

이 uniform 변수를 다음 방법으로 쉐이더에 제공할 수 있습니다.

- Properties 블록을 사용.
- 프로그래밍을 통해 스크립트에서.

Properties 블록을 사용하여 검사기에서 대화식으로 uniform 변수를 정의할 수 있습니다. Properties 블록에서 선언된 모든 변수는 자료 검사기에 변수 이름과 함께 나열됩니다.

다음 코드는 자료 ctSphereMat과 연결된 쉐이더 예제의 Properties 블록입니다

```
Properties
{
    _AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)
    _MainTex ("Base (RGB)", 2D) = "white" {}
}
```

Properties 블록에서 각각 Ambient Color 및 Base (RGB)의 이름으로 선언된 변수 _AmbientColor 및 _MainTex가 해당 이름과 함께 자료 검사기에 표시됩니다.

다음 그림은 자료 검사기의 속성 부분입니다.

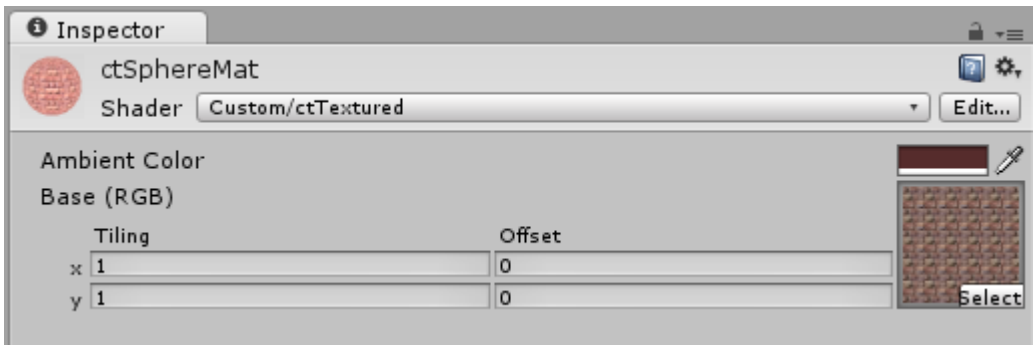


그림 9-5 자료 검사기의 속성

Properties 블록을 통해 쉐이더로 데이터를 전달하는 것은 쉐이더 개발 단계에서 특히 유용합니다. 대화식으로 데이터를 변경하고 런타임에서 효과를 확인할 수 있기 때문입니다.

Properties 블록에 지정할 수 있는 변수 형식은 다음과 같습니다.

- Float.
- Color.
- Texture 2D.
- Cubemap.
- Rectangle.
- Vector.

Properties 블록은 예를 들어 이전 계산으로부터 데이터를 필요로 하거나 특정 시점에 데이터를 전달해야 하는 경우에는 데이터를 전달하는 유용한 방법이 아닙니다.

쉐이더로 데이터를 전달하는 다른 방법은 프로그래밍을 통해 스크립트에서 전달하는 것입니다.

자료 클래스는 특정 자료와 연결된 데이터를 쉐이더로 전달하는 데 사용할 수 있는 여러 메서드를 보여줍니다. 다음 표에는 가장 일반적인 메서드가 나와 있습니다.

표 9-2 특정 자료와 연결된 데이터를 쉐이더로 전달하는 가장 일반적인 메서드

| 메서드 |
|---|
| SetColor (propertyName: string, color: Color); |
| SetFloat (propertyName: string, value: float); |
| SetInt (propertyName: string, value: int); |
| SetMatrix (propertyName: string, matrix: Matrix4x4); |

표 9-2 특정 자료와 연결된 데이터를 쉐이더로 전달하는 가장 일반적인 메서드 (계속)

| 메서드 |
| --- |
| SetVector (propertyName: string, vector: Vector4); |
| SetTexture (propertyName: string, texture: Texture); |

다음 코드에서는 메인 카메라가 장면을 렌더링하기 직전에 두 번째 카메라 shwCam이 메인 카메라 렌더 패스와 결합될 텍스처로 그림자를 렌더링합니다.

그림자 텍스처 투영 프로세스의 경우 정점이 편리한 방식으로 변환되어야 합니다. 그림자 카메라 투영 매트릭스(shwCam.projectionMatrix), 월드-로컬 변환 매트릭스 (shwCam.transform.worldToLocalMatrix) 및 렌더링된 그림자 텍스처(m_ShadowsTexture)가 쉐이더로 전달됩니다.

이들 값은 쉐이더에서 이름이 _ShwCamProjMat, _ShwCamViewMat 및 m_ShadowsTexture인 uniform 변수로 사용 가능합니다.

다음 코드는 어떻게 매트릭스 및 텍스처가 shwMats 목록에 포함된 자료를 통해 쉐이더로 전달되는지 보여줍니다.

```
// Called before object is rendered.

public void OnWillRenderObject()
{
    // Perform different checks.
    ...
    CreateShadowsTexture();
    // Set up shadows camera shwCam.
    ...
    // Pass matrices to the shader
    for(int i = 0; i < shwMats.Count; i++)
    {
        shwMats[i].SetMatrix("_ShwCamProjMat", shwCam.projectionMatrix);
        shwMats[i].SetMatrix("_ShwCamViewMat", shwCam.transform.worldToLocalMatrix);
    }
    // Render shadows texture
    shwCam.Render();
    for(int i = 0; i < shwMats.Count; i++)
    {
        shwMats[i].SetTexture( "_ShadowsTex", m_ShadowsTexture );
    }
    s_RenderingShadows = false;
}
```

### 9.1.11 Unity에서 쉐이더 디버깅

Unity에서는 종래의 코드와 동일한 방식으로 쉐이더를 디버깅할 수 없습니다. 하지만 조각 쉐이더의 출력을 사용하여 디버깅하려는 값을 시각화할 수 있습니다. 그런 다음 생성된 이미지를 해석해야 합니다.

다음 그림은 *9.2 로컬 큐브맵을 사용하여 반사 구현* 페이지의 9-153의 바닥 반사 표면에 적용된 쉐이더 ctReflLocalCubemap.shader의 출력입니다.


그림 9-6 반사가 있는 체스룸

다음 조각 쉐이더에서는 출력 색상이 정규화된 로컬 수정 반사 벡터로 대체되었습니다.

```
return float4(normalize(localCorrReflDirWS), 1.0);
```

반사된 이미지 대신, 색상으로 정규화된 반사 벡터의 구성요소를 시각화합니다.

바닥의 붉은 영역은 반사 벡터가 강한 X 구성요소를 가짐을, 즉 대부분 X축으로 향해 있다는 것을 나타냅니다. 붉은 부분은 반사가 해당 방향, 즉 창이 나있는 벽에서 온다는 것을 보여줍니다.

푸르스름한 영역은 Z축으로 향하는 반사 벡터, 즉 오른쪽 벽에서 반사된 빛이 우세함을 나타냅니다.

검은색 영역에서는 벡터가 주로 –Z를 향하지만, 음의 구성요소가 0으로 클램프되므로 색상은 양의 구성요소만 가질 수 있습니다.

다음 그림은 조각의 출력 색상을 정규화된 로컬 반사 벡터로 대체한 결과입니다.

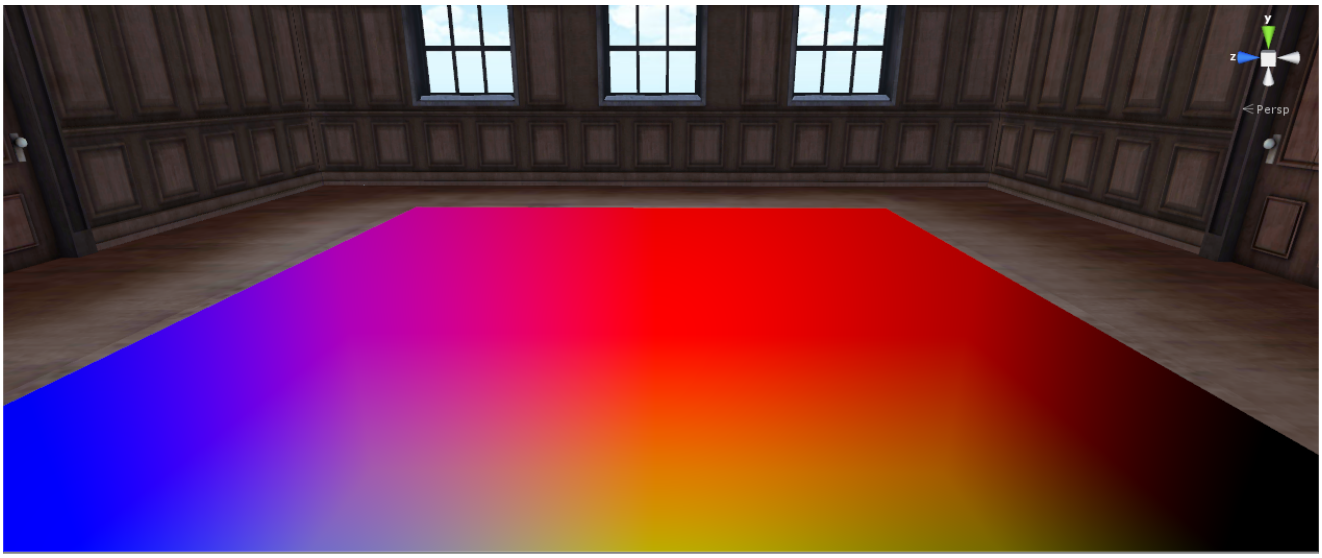


그림 9-7 다중 색상을 사용한 쉐이더 디버깅

디버깅을 수행하는 동안 색상의 의미를 해석하기가 처음에는 어려울 수 있으므로 한 가지 색상 구성요소에만 집중해 보십시오. 예를 들어 정규화된 로컬 수정 반사 벡터의 Y 구성요소만 반환할 수 있습니다.

```
float3 normLocalCorrReflDirWS = normalize(localCorrReflDirWS);
return float4(0, normLocalCorrReflDirWS.y, 0, 1);
```

이 경우, 출력은 카메라 위쪽 지붕에서 주로 나오는 반사뿐입니다. 즉, 실내의 Y축을 향하는 부분입니다 벽으로부터의 반사는 X, Z 및 –Z 방향에서 나오므로 이들은 검은색으로 렌더링됩니다.

다음 그림은 단일 색상을 사용한 쉐이더 디버깅을 보여줍니다.


그림 9-8 단일 색상을 사용한 쉐이더 디버깅

색상을 사용하여 디버깅하는 크기가 0~1 사이인지 확인하십시오. 다른 값은 자동으로 클램프됩니다. 음의 값은 영으로 할당되고 1보다 큰 값은 1로 할당됩니다.

## 9.2 로컬 큐브맵을 사용하여 반사 구현

로컬 큐브맵에 기반한 반사는 모바일 디바이스에서 반사를 렌더링하는 데 유용한 기법입니다.

Unity 버전 5 이상에서는 로컬 큐브맵에 기반한 반사를 반사 프로브로 구현합니다. 이러한 반사를 사용자 지정 쉐이더에서 런타임 시 렌더링되는 반사와 같이 다른 유형의 .반사와 결합할 수 있습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.2.1 반사 구현의 역사

그래픽 개발자는 낮은 연산 비용으로 반사를 구현하기 위한 방법을 찾기 위해 늘 노력해 왔습니다.

최초의 솔루션 중 하나는 구면 매핑입니다. 이 기법은 연산 비용이 높은 광선 추적 또는 조명 계산을 사용하지 않고 객체에 대한 반사 또는 조명을 시뮬레이션합니다.

다음 그림은 구체 위의 환경 맵입니다.

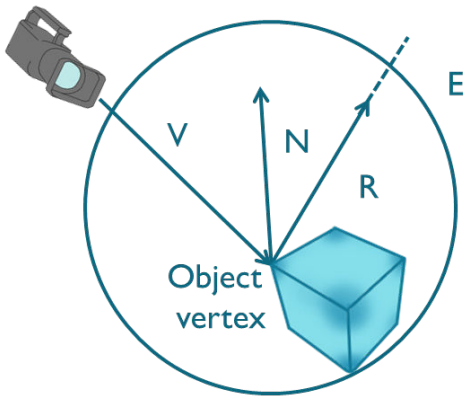


그림 9-9 구체 위 환경 맵

다음 그림은 구면을 2차원으로 매핑하기 위한 방정식을 보여줍니다.

The spherical surface is mapped into 2D:

$$u = \frac{R_x}{m} + \frac{1}{2}$$

$$v = \frac{R_y}{m} + \frac{1}{2}$$

$$m = 2 \cdot \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

그림 9-10 구면 2D 매핑 방정식

이 기법에는 여러 단점이 있지만, 주된 문제는 그림을 구체에 매핑할 때 발생하는 왜곡입니다. 1999년, 하드웨어 가속을 기반으로 큐브맵을 사용할 수 있게 되었습니다. 큐브맵은 이미지 왜곡, 시점 종속성,구면 매핑과 관련된 비효율적 연산이라는 문제를 해결했습니다.

다음 그림은 펼친 큐브입니다

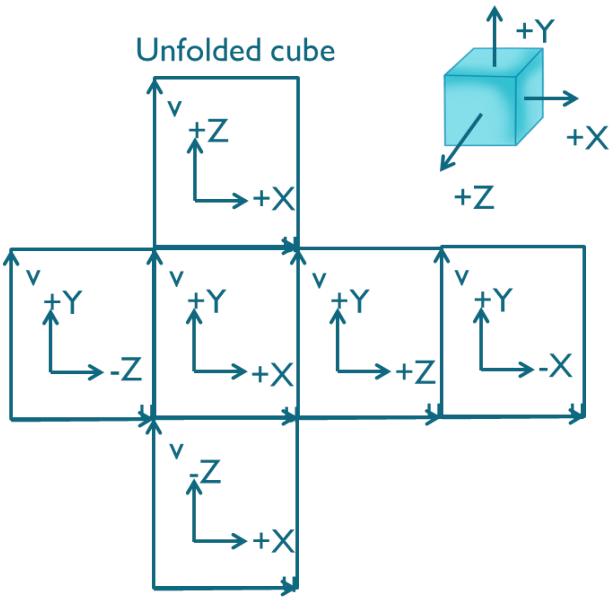


그림 9-11 펼친 큐브

큐브매핑은 맵 형태로 큐브의 6면을 사용합니다. 환경은 큐브의 각 면으로 투영되고 6개의 정사각형 텍스처로 저장되거나 단일 텍스처의 6개 영역으로 펼쳐집니다. 큐브맵은 각 큐브 면을 표현하는 6개 카메라 방향(90도 뷰 프루스툼)을 사용하여 특정 위치에서 장면을 렌더링하여 생성됩니다. 소스 이미지는 직접 샘플링됩니다. 중간 환경 맵으로 다시 샘플링하여 왜곡이 발생하지 않습니다.

다음 그림은 무한 반사입니다.

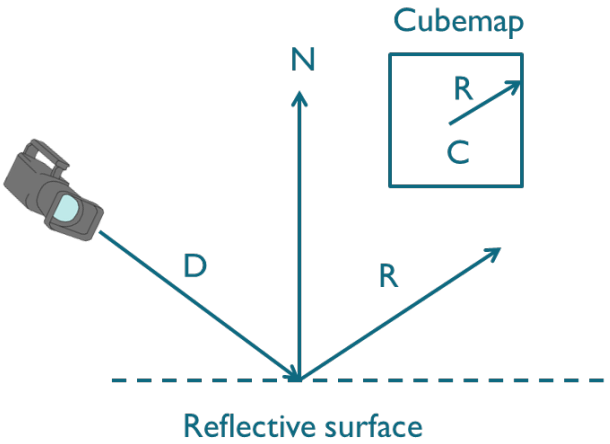


그림 9-12 무한 반사

큐브맵을 기반으로 반사를 구현하려면 반사된 벡터 R을 평가하고 이 벡터로 lookup 함수 texCUBE()를 사용하여 큐브맵 _Cubemap으로부터 텍셀을 가져옵니다.

```
float4 color = texCUBE(_Cubemap, R);
```

노말 N 및 뷰 벡터 D가 조각 및 정점 쉐이더로 전달됩니다. 조각 쉐이더가 큐브맵에서 텍스처 색상을 가져옵니다.

```
float3 R = reflect(D, N);
float4 color = texCUBE(_Cubemap, R);
```

이 방법은 큐브맵 위치가 상관 없는 원거리 환경에서만 올바르게 반사를 재현할 수 있습니다. 이 간단하고 효과적인 기법은 실외 조명에서 주로 사용됩니다(예: 하늘의 반사를 추가).

다음 그림은 잘못된 반사를 보여줍니다.

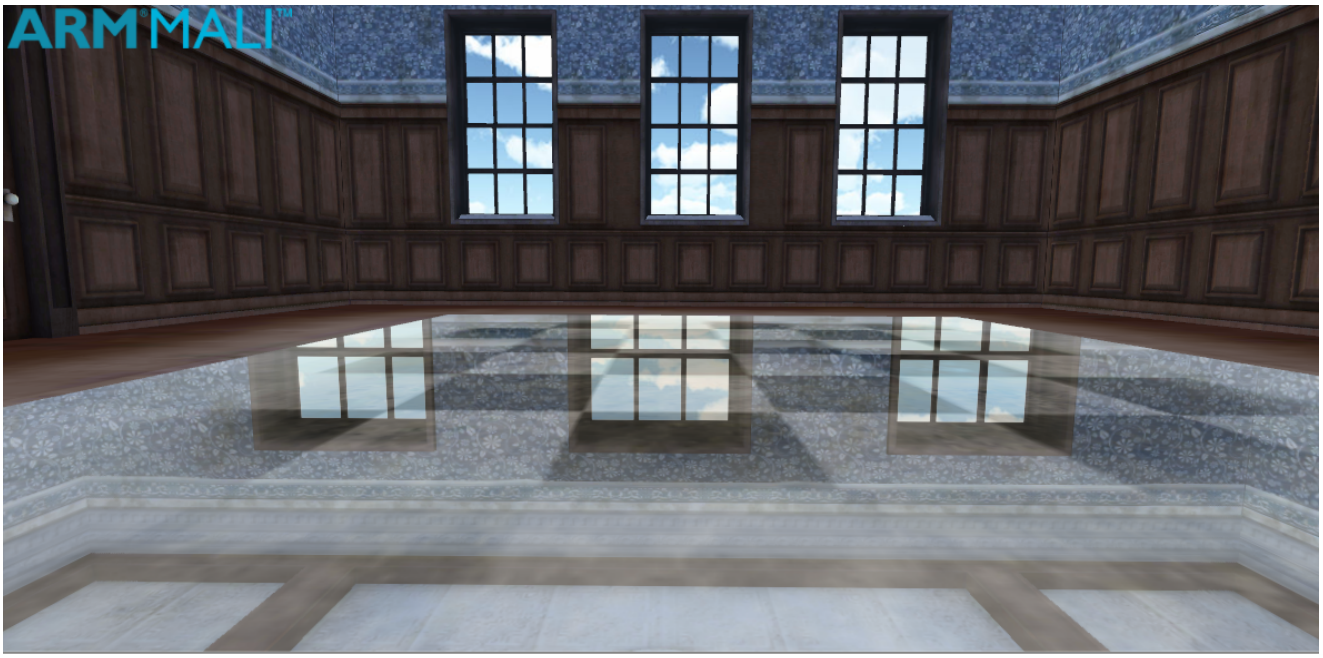


그림 9-13 잘못된 반사

로컬 환경에서 이 기법을 사용할 경우 잘못된 반사가 생성됩니다. 반사가 부정확한 이유는 식 `float4 color = texCUBE(_Cubemap, R);`에 로컬 지오메트리에 대한 바인딩이 없기 때문입니다. 예를 들어 동일한 각도로 반사성 바닥을 바라보며 그 위를 걸어갈 경우 반사는 계속 똑같이 보입니다. 반사된 벡터는 항상 동일하고 식은 항상 동일한 결과를 제공합니다. 이는 뷰 벡터의 방향이 바뀌지 않기 때문입니다. 실제 세계에서는 반사가 보는 각도와 보는 위치에 따라 달라집니다.

### 9.2.2 로컬 큐브맵을 사용하여 올바른 반사를 생성

이 문제를 해결하는 한 솔루션은 반사를 계산하는 프로시저에서 로컬 지오메트리에 바인딩하는 것입니다.

이 솔루션은 Randima Fernando(시리즈 편집자)의 *GPU Gems: 실시간 그래픽 프로그래밍 기법, 팁 및 힌트*에 설명되어 있습니다.

다음 그림은 경계 구체를 사용한 로컬 수정을 보여줍니다.

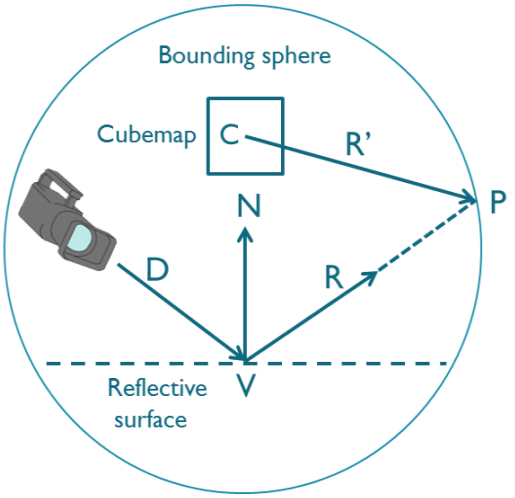


그림 9-14 경계 구체를 사용한 로컬 수정

경계 구체는 반사될 장면을 구분하는 프록시 볼륨으로 사용됩니다. 반사된 백터 R을 사용하여 큐브맵으로부터 텍스처를 가져오는 대신 새로운 벡터 R'이 사용됩니다. 이 새 벡터를 만들려면 로컬 지점 V로부터 반사된 벡터 R 방향으로 광선이 경계 구체와 교차하는 지점 P를 찾습니다. 큐브맵이 생성된 큐브맵 중심 C로부터 교차점 P까지 새 벡터 R'를 만듭니다. 이 벡터를 사용하여 큐브맵에서 텍스처를 가져옵니다.

```
float3 R = reflect(D, N);
Find intersection point P
Find vector R' = CP
float4 col = texCUBE(_Cubemap, R');
```

이 방법은 거의 구형에 가까운 객체의 표면에서 양호한 결과를 가져오지만 평면 반사 표면에서는 반사가 왜곡됩니다. 이 방법의 또 다른 결점은 경계 구체와의 교차점을 계산하는 알고리즘이 2차 방정식을 풀어야 하는데 이것이 비교적 복잡하다는 것입니다.

지난 2010년, 한 개발자가 *http://www.gamedev.net*의 한 포럼에서 보다 향상된 솔루션을 제안했습니다. 이 방법은 이전의 경계 구체를 상자로 대체하고 이전 방법의 왜곡 및 복잡성 문제를 해결했습니다. 자세한 내용은 *http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/?&p=4637262*을 참조하십시오.

다음 그림은 경계 상자를 사용한 로컬 수정을 보여줍니다.

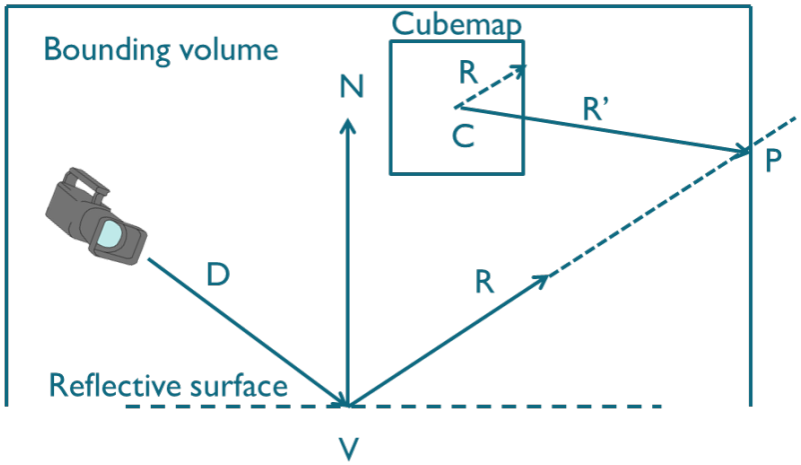


그림 9-15 경계 상자를 사용한 로컬 수정

보다 최근인 2012년, Sebastien Lagarde에 의한 작업에서는 이 새로운 방법을 사용하여 여러 큐브맵을 사용하는 보다 복잡한 주변 반사 조명을 시뮬레이션했고 각 큐브맵의 기여를 평가하고 효율적으로 GPU에서 블렌드하는 알고리즘을 사용했습니다. *http://seblagarde.wordpress.com*을 참조하십시오.

표 9-3 무한 큐브맵과 로컬 큐브맵의 차이

| 무한 큐브맵 | 로컬 큐브맵 |
|---|---|
| • 원거리 환경으로부터의 조명을 표현하기 위해 주로 실외에서 사용됩니다.<br>• 큐브맵 위치는 상관이 없습니다. | • 유한한 로컬 환경으로부터의 조명을 표현합니다.<br>• 큐브맵 위치가 상관이 있습니다.<br>• 이러한 큐브맵으로부터의 조명은 큐브맵이 생성된 위치에서만 올바릅니다.<br>• 큐브맵 고유의 로컬 환경에 대한 무한 속성을 조정하기 위해 반드시 로컬 수정을 적용해야 합니다. |

다음 그림은 로컬 큐브맵을 사용하여 생성된 올바른 반사가 적용된 장면을 보여줍니다.

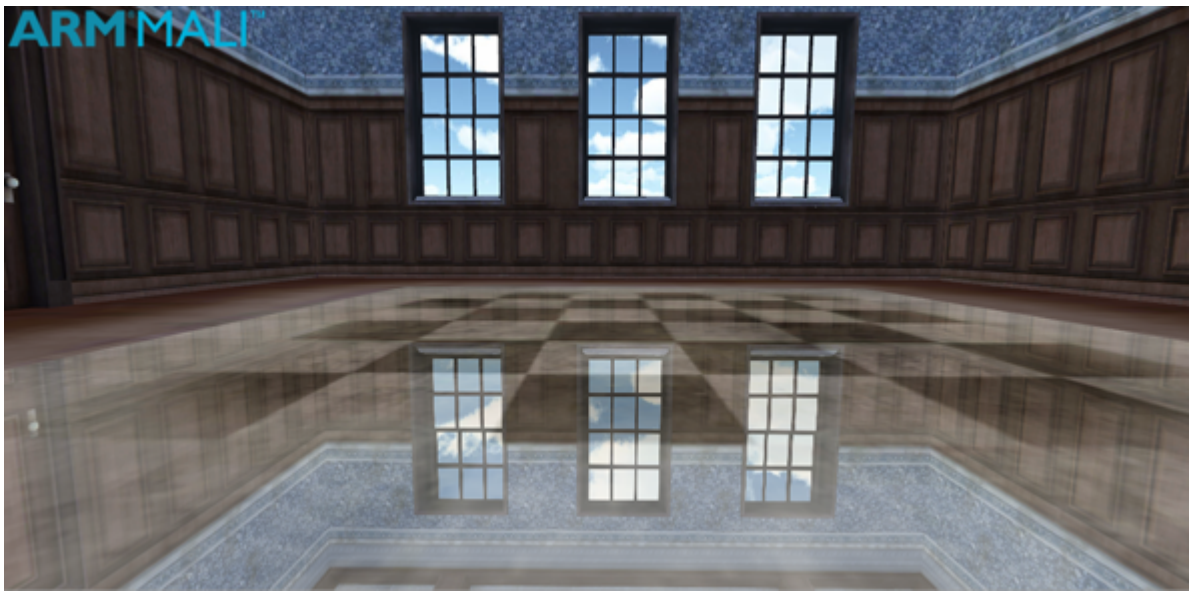


그림 9-16 올바른 반사

### 9.2.3 쉐이더 구현

이 단원에서는 로컬 큐브맵을 사용하여 반사를 구현하는 쉐이더를 설명합니다.

정점 쉐이더는 조각 쉐이더로 전달되는 3개의 크기를 보간된 값으로 계산합니다.

- 정점 위치.
- 뷰 방향.
- 노말.

이들 값은 월드 좌표입니다.

다음 코드는 Unity에서 로컬 큐브맵을 사용하여 반사를 구현한 쉐이더입니다.

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;
    // Transform vertex coordinates from local to world.
    float4 vertexWorld = mul(_Object2World, input.vertex);
    // Transform normal to world coordinates.
    float4 normalWorld = mul(float4(input.normal,0.0), _World2Object);
    // Final vertex output position.  output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    // ----------- Local correction ------------
    output.vertexInWorld = vertexWorld.xyz;
    output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
```

```
    output.normalInWorld = normalWorld.xyz;
    return output;
}
```

볼륨 박스 내 교차점과 반사된 벡터는 조각 쉐이더에서 연산됩니다. 로컬 수정된 반사 벡터를 새로 만들고 이 벡터를 사용하여 로컬 큐브맵에서 반사 텍스처를 가져옵니다. 그런 다음 텍스처와 반사를 결합하여 출력 색상을 생성합니다.

```
float4 frag(vertexOutput input) : COLOR
{
    float4 reflColor = float4(1, 1, 0, 0);
    // Find reflected vector in WS.
    float3 viewDirWS = normalize(input.viewDirInWorld);
    float3 normalWS = normalize(input.normalInWorld);
    float3 reflDirWS = reflect(viewDirWS, normalWS);
    // Working in World Coordinate System.
    float3 localPosWS = input.vertexInWorld;
    float3 intersectMaxPointPlanes = (_BBoxMax - localPosWS) / reflDirWS;
    float3 intersectMinPointPlanes = (_BBoxMin - localPosWS) / reflDirWS;
    // Looking only for intersections in the forward direction of the ray.
    float3 largestParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);
    // Smallest value of the ray parameters gives us the intersection.
    float distToIntersect = min(min(largestParams.x, largestParams.y), largestParams.z);
    // Find the position of the intersection point.
    float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;
    // Get local corrected reflection vector.
    Float3 localCorrReflDirWS = intersectPositionWS - _EnviCubeMapPos;
    // Lookup the environment reflection texture with the right vector.
    reflColor = texCUBE(_Cube, localCorrReflDirWS);
    // Lookup the texture color.
    float4 texColor = tex2D(_MainTex, float2(input.tex));
    return _AmbientColor + texColor * _ReflAmount * reflColor;
}
```

위의 조각 쉐이더 코드에서 크기 _BBoxMax 및 _BBoxMin은 경계 볼륨의 최대 및 최소 지점입니다. 변수 _EnviCubeMapPos는 큐브맵이 생성된 위치입니다. 이들 값을 다음 스크립트에서 쉐이더로 전달합니다.

```
[ExecuteInEditMode]
public class InfoToReflMaterial : MonoBehaviour
{
    // The proxy volume used for local reflection calculations.
    public GameObject boundingBox;

    void Start()
    {
        Vector3 bboxLenght = boundingBox.transform.localScale;
        Vector3 centerBBox = boundingBox.transform.position;

        // Min and max BBox points in world coordinates
        Vector3 BMin = centerBBox - bboxLenght/2;
        Vector3 BMax = centerBBox + bboxLenght/2;

        // Pass the values to the material.
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMin", BMin);
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMax", BMax);
        gameObject.renderer.sharedMaterial.SetVector("_EnviCubeMapPos", centerBBox);
    }
}
```

_AmbientColor, _ReflAmount, 메인 텍스처 및 큐브맵 텍스처 값을 uniform 변수로 속성 블록에서 쉐이더로 전달합니다.

```
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
  {
    Pass
```

```
    {
      CGPROGRAM
      #pragma glsl
      #pragma vertex vert
      #pragma fragment frag
      #include "UnityCG.cginc"

      // User-specified uniforms
      uniform sampler2D _MainTex;
      uniform samplerCUBE _Cube;
      uniform float4 _AmbientColor;
      uniform float _ReflAmount;
      uniform float _ToggleLocalCorrection;
      // ----Passed from script InfoRoReflmaterial.cs --------
      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;
      };

    정점 쉐이더   { }
    조각 쉐이더   { }

    ENDCG
    }
  }
}
```

경계 볼륨에서 교차점을 계산하는 알고리즘은 로컬 위치 또는 조각에서 반사된 광선의 매개 변수 표현의 사용을 기반으로 합니다. 광선-상자 교차 알고리즘에 대한 설명은 *9.2.5 광선-상자 교차 알고리즘* 페이지의 9-163을 참조하십시오.

### 9.2.4 큐브맵 필터링

로컬 큐브맵을 사용하여 반사를 구현할 경우 장점 중 하나는 큐브맵이 정적이라는 것입니다. 즉, 큐브맵은 런타임이 아니라 개발 도중 생성됩니다. 그러므로 효과를 구현하기 위해 큐브맵 이미지에 필터링을 적용할 수 있는 기회가 주어집니다.

CubeMapGen은 큐브맵에 필터링을 적용하는 AMD 도구입니다. AMD 개발자 웹 사이트 (*http://developer.amd.com*)에서 CubeMapGen을 다운로드할 수 있습니다.

큐브맵 이미지를 Unity에서 CubeMapGen으로 내보내려면 각 큐브맵 이미지를 개별적으로 저장해야 합니다. 이미지를 저장하는 도구의 소스 코드는 *9.2.6 큐브맵을 생성할 편집기 스크립트용 소스 코드* 페이지의 9-166를 참조하십시오. 이 도구는 큐브맵을 생성하고 선택적으로 각 큐브맵 이미지를 개별적으로 저장할 수 있습니다.

이 도구의 스크립트를 Asset 디렉터리의 `Editor` 폴더에 배치해야 합니다.

큐브맵 편집기 도구를 사용하는 방법

1. 큐브맵을 생성합니다.
2. GameObject 메뉴에서 Bake CubeMap 도구를 시작합니다.
3. 큐브맵 및 카메라 렌더 위치를 제공합니다.
4. 선택적으로, 큐브맵에 필터링을 적용하려면 개별 이미지를 저장합니다.

다음 그림은 Bake CubeMap 도구 인터페이스입니다.

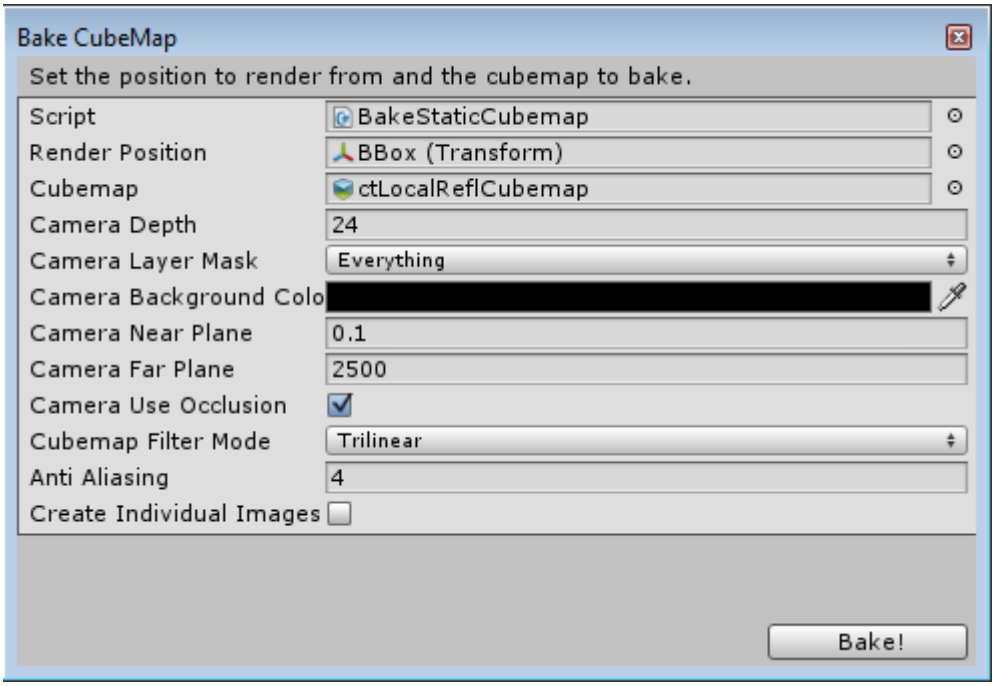


그림 9-17 Bake CubeMap 도구 인터페이스

CubeMapGen을 사용하여 큐브맵의 각 이미지를 개별적으로 로드할 수 있습니다.

Select Cube Face 드롭다운 메뉴에서 로드할 면을 선택하고 Load Cubemap Face 버튼을 누릅니다. 모든 면이 로드되면 큐브맵을 회전시키며 올바른지 확인하는 것이 가능합니다.

CubeMapGen은 Filter Type 드롭다운 메뉴에서 다양한 필터링 옵션을 사용할 수 있습니다. 필요한 필터 설정을 선택하고 Filter Cubemap을 눌러 필터를 적용합니다. 필터링은 큐브맵 크기에 따라 몇 분이 걸릴 수 있습니다. 실행 취소 옵션이 없으므로 필터링을 적용하기 전에 큐브맵을 단일 이미지로 저장하십시오. 필터링 결과가 예상한 것과 다르다면 큐브맵을 다시 로드하여 매개 변수를 조정할 수 있습니다.

큐브맵을 CubeMapGen으로 가져올 때는 다음 절차를 사용합니다.

1. 큐브맵을 베이크할 때 개별 이미지를 저장하는 확인란을 선택합니다.
2. CubeMapGen 도구를 시작하고 다음 표에 나열된 관계에 따라 큐브맵 이미지를 로드합니다.
3. 큐브맵을 단일 dds 또는 큐브 크로스 이미지로 저장합니다. 실행 취소가 불가능하므로 필터를 사용하여 실험하는 경우 이를 통해 큐브맵을 다시 로드할 수 있습니다.
4. 결과가 만족스러울 때까지 필요한 필터를 큐브맵에 적용합니다.
5. 큐브맵을 개별 이미지로 저장합니다.

다음 표는 CubeMapGen과 Unity 사이의 큐브맵 페이스 인덱스 등가 관계를 보여줍니다.

표 9-4 CubeMapGen과 Unity 사이의 큐브맵 페이스 인덱스 등가 관계

| AMD CubeMapGen | Unity |
|---|---|
| X+ | -X |
| X- | +X |
| Y+ | +Y |
| Y- | -Y |
| Z+ | +Z |
| Z- | -Z |

다음 그림은 6개 큐브맵 이미지가 로딩된 CubeMapGen을 보여줍니다.


그림 9-18 CubeMapGen

다음 그림은 CubeMapGen이 가우시안 필터링을 적용하여 *서리* 효과를 구현한 결과입니다.

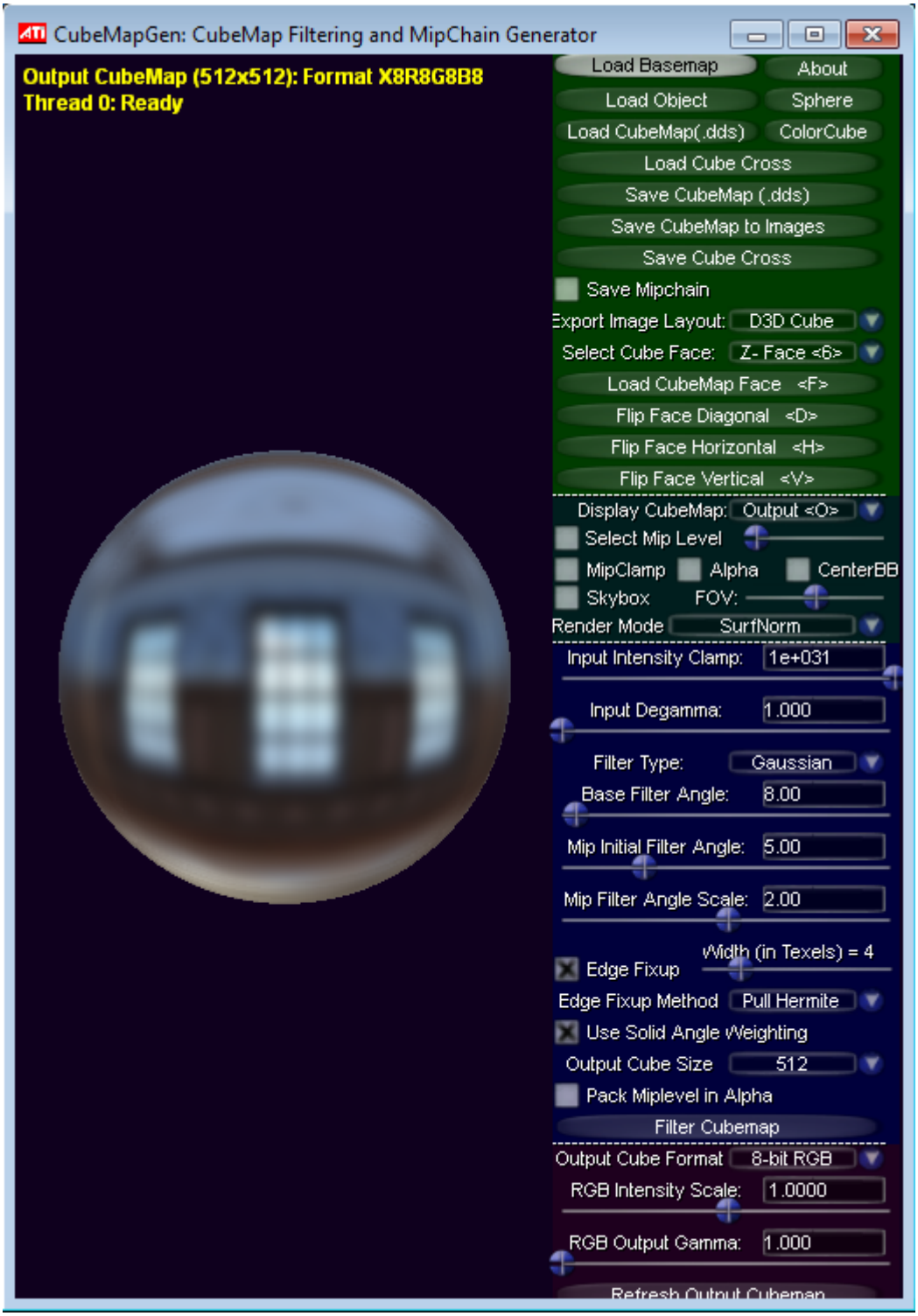


그림 9-19 서리 효과를 보여주는 CubeMapGen

다음 표는 서리 효과를 구현하기 위해 Gaussian 필터와 함께 사용되는 필터 매개 변수입니다.

표 9-5 반사에서 서리 효과를 구현하기 위해 CubeMapGen에서 사용되는 매개 변수.

| 필터 설정 | 값 |
|---|---|
| 형식 | 가우시안 |
| 기본 필터 각도 | 8 |
| 밉 초기 필터 각도 | 5 |
| 밉 필터 각도 스케일 | 2.0 |
| 에지 픽스업 | 선택 |
| 에지 픽스업 너비 | 4 |

다음 그림은 서리 효과를 사용하여 큐브맵과 함께 생성된 반사를 보여줍니다.


그림 9-20 서리 효과가 있는 반사

다음 그림에 CubeMapGen 도구를 사용하여 Unity에 필터링을 적용하는 워크플로가 요약되어 있습니다.

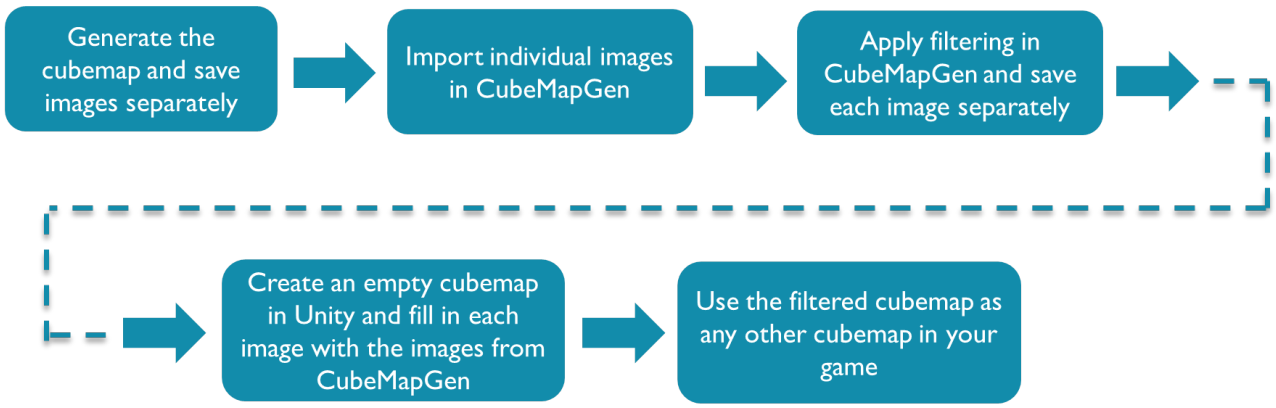


그림 9-21 큐브맵 필터링 워크플로

### 9.2.5 광선-상자 교차 알고리즘

이 단원에서는 광선-상자 교차 알고리즘에 대해 설명합니다.

다음 그림은 직선 방정식 그래프입니다.

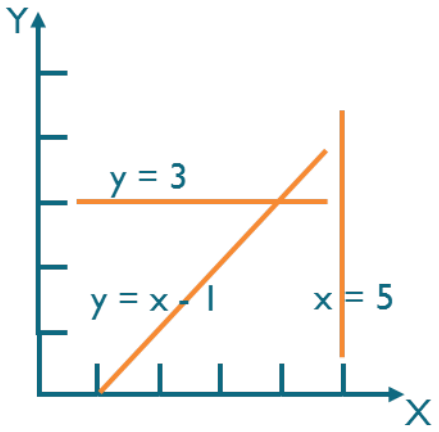


그림 9-22 선 방정식 포함 그래프

직선 방정식

```
y = mx + b
```

이 방정식의 벡터형은 다음과 같습니다.

```
r = O + t*D
```

인수 설명:

O: 원점

D: 방향 벡터

t: 매개 변수

다름 그림은 축 정렬된 경계 상자입니다.

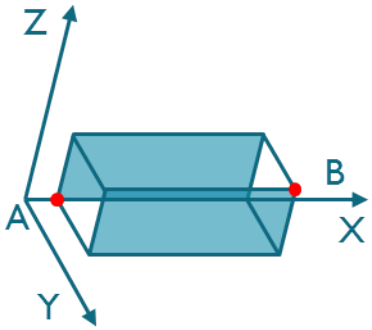


그림 9-23 축 정렬된 경계 상자

축에 맞춰 정렬된 경계 상자 AABB는 최소 점과 최대 점 A와 B로 정의할 수 있습니다.

AABB는 좌표 축과 평행한 직선 세트를 정의합니다. 직선의 각 구성요소는 다음 방정식으로 정의할 수 있습니다.

$$x = A_x;\ y = A_y;\ z = A_z x = B_x;\ y = B_y;\ z = B_z$$

광선이 어디에서 이들 직선 중 하나와 교차하는지 확인하려면 두 방정식을 풉니다. 예를 들면 다음과 같습니다.

$$O_x + t_x * D_x = A_x$$

해를 다음과 같이 쓸 수 있습니다.

$$t_{Ax} = (A_x - O_x) / D_x$$

동일한 방법으로 두 교차점의 모든 구성요소에 대해 해를 구합니다.

$$t_{Ax} = (Ax - Ox) / Dx$$
$$t_{Ay} = (Ay - Oy) / Dy$$
$$t_{Az} = (Az - Oz) / Dz$$
$$t_{Bx} = (Bx - Ox) / Dx$$
$$t_{By} = (By - Oy) / Dy$$
$$t_{Bz} = (Bz - Oz) / Dz$$

벡터형에서는 다음과 같습니다.

$$t_A = (A - O) / D$$
$$t_B = (B - O) / D$$

이는 육면체의 표면에 의해 정의된 평면과 직선이 교차하는 위치를 찾지만 교차점이 육면체 상에 존재하는지는 보장하지 않습니다.

다음 그림은 광선-상자 교차의 2D 표현입니다.

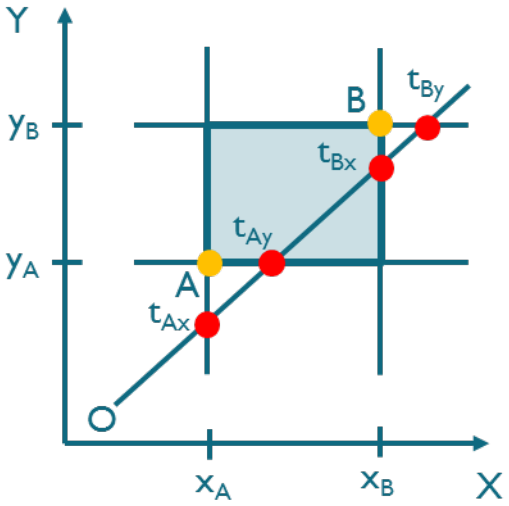


그림 9-24 광선-상자 교차 2D 표현

실제로 어느 해가 상자와의 교차인지 확인하려면 최소 평면에서의 교차에는 더 높은 t 값이 필요합니다.

$$t_{min} = (t_{Ax} > t_{Ay}) \ ? \ t_{Ax} : t_{Ay}$$

최대 평면에서의 교차에는 더 작은 t 값이 필요합니다.

$$t_{min} = (t_{Ax} > t_{Ay}) \ ? \ t_{Ax} : t_{Ay}$$

또한 교차가 없을 경우도 고려해야 합니다.

다음 그림은 교차가 없는 광선-상자입니다.

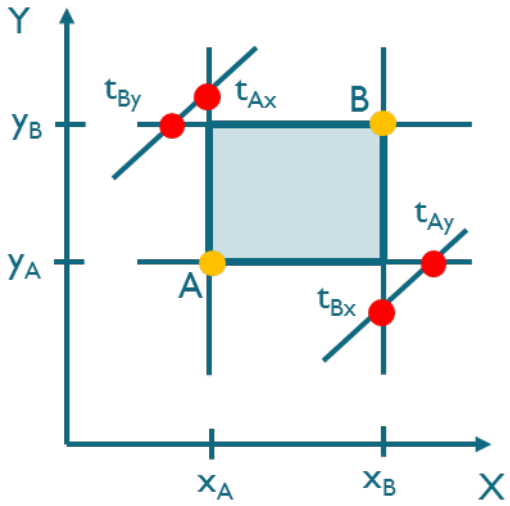


그림 9-25 교차 없는 광선-상자

BBox에 의해 둘러싸인 반사 표면이 없는 것이 확실한 경우, 즉 반사된 광선의 원점이 BBox 내부에 있는 경우 상자에는 항상 2개의 교차점이 있으며 다양한 경우의 처리가 간소해집니다.

다음 그림은 BBox에서의 광선-상자 교차입니다.

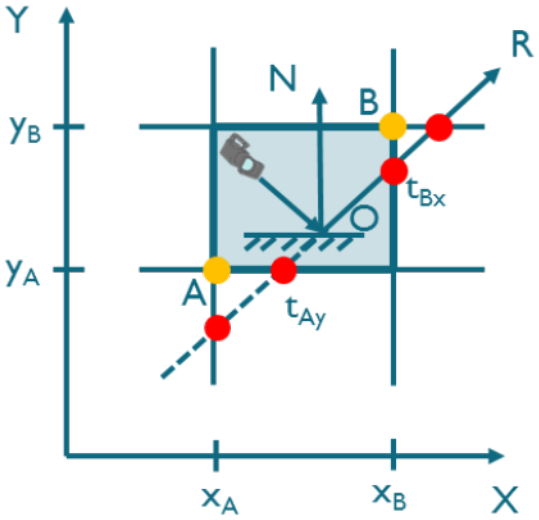


그림 9-26 BBox에서의 광선-상자 교차

### 9.2.6 큐브맵을 생성할 편집기 스크립트용 소스 코드

이 단원에서는 큐브맵을 생성할 편집기 스크립트용 소스 코드를 제공합니다.

```
/*
* 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
```

```
* 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 = "렌더링을 시작할 위치와 베이킹할 큐브맵을 설정합니다.";
      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.
```

```
        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("큐브맵 베이킹", typeof(BakeStaticCubemap),"베이킹합니
다.");
    }
}
```

## 9.3 반사 결합

이 단원에서는 반사 결합을 설명합니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.3.1 반사 결합 개요

로컬 큐브맵 기법을 사용한 반사는 정적 로컬 큐브맵을 기반으로 고품질 반사를 효율적으로 렌더링할 수 있게 해줍니다. 하지만 객체가 동적일 경우 정적 로컬 큐브맵은 더 이상 유효하지 않아 이 기법을 사용할 수 없습니다.

정적 반사를 동적으로 생성된 반사와 결합하여 이 문제를 해결할 수 있습니다.

다음 그림은 정적 및 동적 지오메트리 반사의 결합을 보여줍니다.

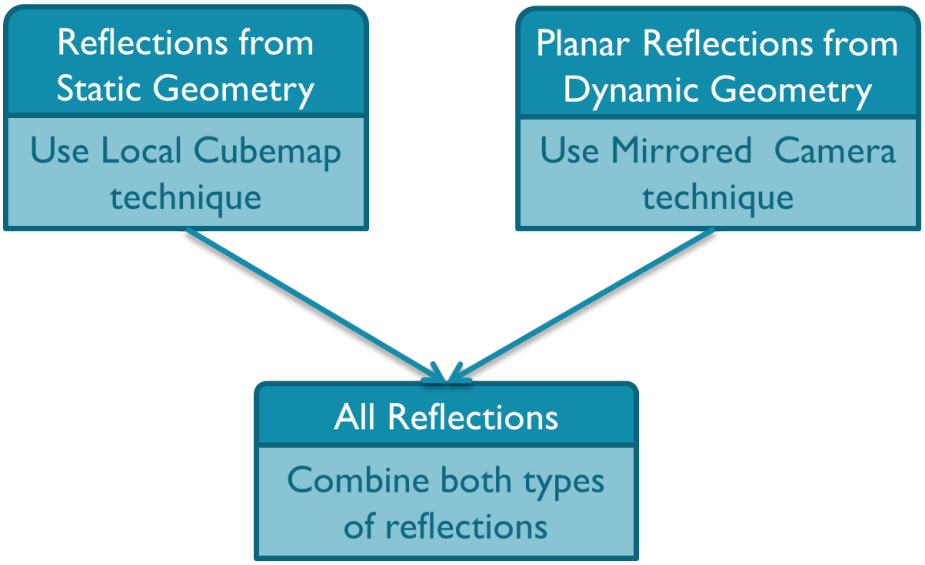


그림 9-27 정적 및 동적 지오메트리 반사의 결합

반사 표면이 평면일 경우 미러 카메라를 사용하여 동적 반사를 생성할 수 있습니다.

미러 카메라를 생성하려면 런타임 시 반사를 렌더링하는 메인 카메라의 위치 및 방향을 계산합니다.

반사 평명을 기준으로 메인 카메라의 위치 및 방향을 미러링합니다.

다음 그림은 미러 카메라 기법을 보여줍니다.

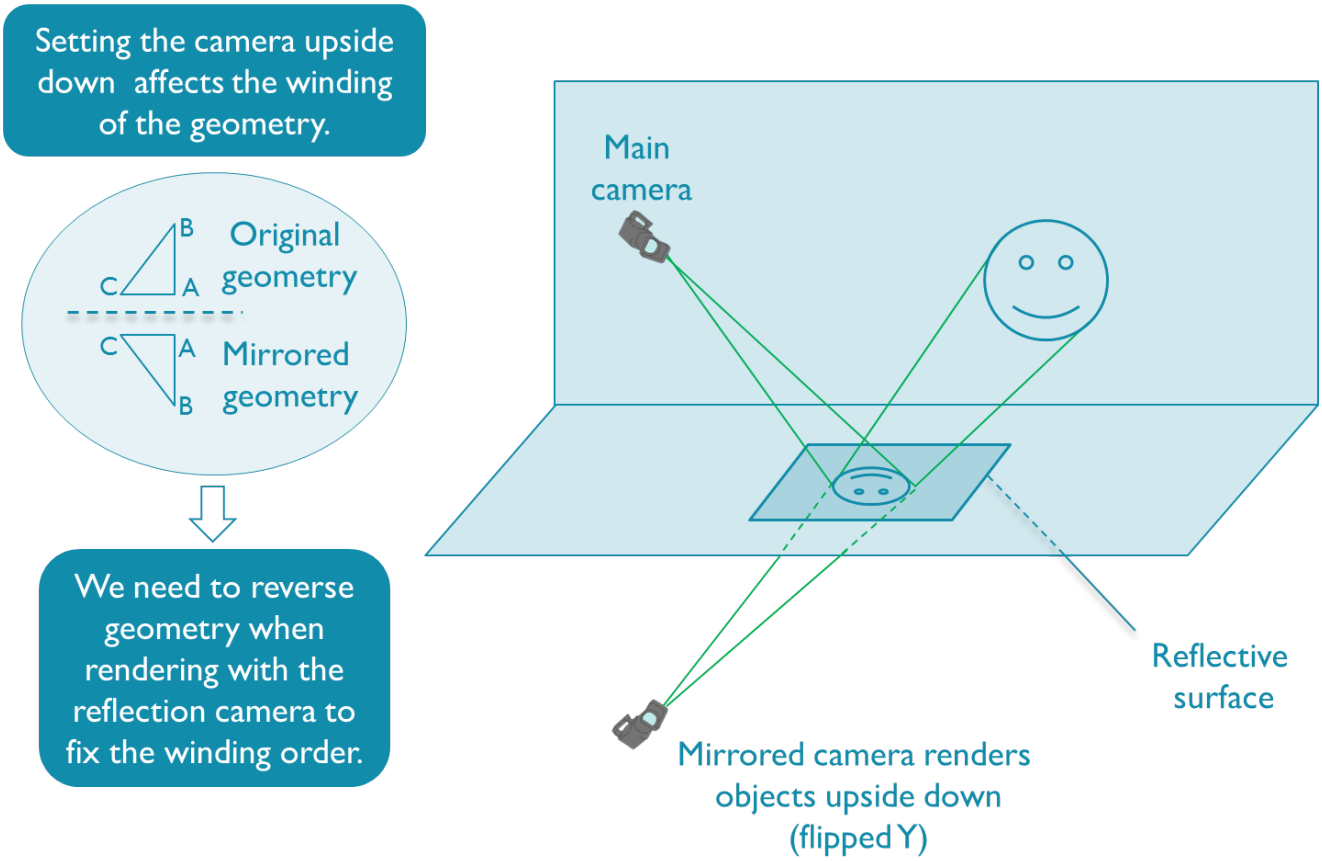


그림 9-28 평면 반사를 렌더링하기 위한 미러 카메라 기법

미러링 프로세스에서 새 반사 카메라는 축이 반대 방향에 위치하게 됩니다. 실제 거울과 마찬가지 방식으로 좌측 및 우측의 반사가 반전됩니다. 이는 반사 카메라가 반대 방향 와인딩으로 지오메트리를 렌더링함을 의미합니다.

지오메트리를 올바로 렌더링하려면 반사를 렌더링하기 전에 지오메트리의 와인딩을 반전시켜야 합니다. 반사 렌더링이 완료되면 원래 와인딩으로 복원하십시오.

다음 그림은 미러 카메라를 설정하고 반사를 렌더링하는 데 필요한 단계를 순서대로 보여줍니다.

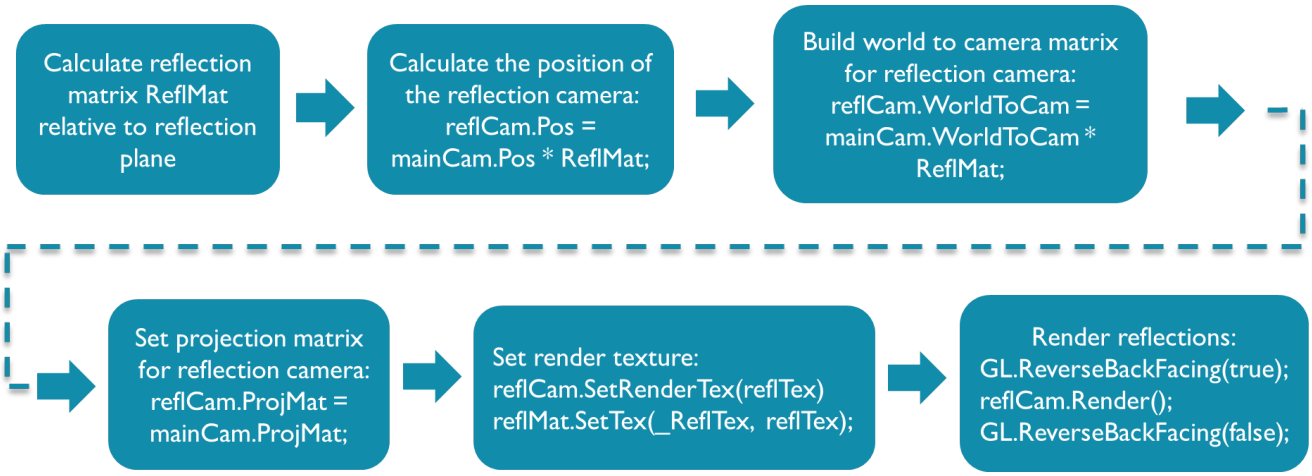


그림 9-29 미러 카메라 설정 및 반사 렌더링 주요 단계

미러 반사 변환 매트릭스를 만듭니다. 이 매트릭스를 사용하여 반사 카메라의 위치 및 월드-카메라 변환 매트릭스를 계산합니다.

다음 그림은 미러 반사 변환 매트릭스입니다.

$$R = \begin{bmatrix} 1-2n_x^2 & -2n_xn_y & -2n_xn_z & -2n_zn_w \\ -2n_xn_y & 1-2n_y^2 & -2n_yn_z & -2n_yn_w \\ -2n_xn_z & -2n_yn_z & 1-2n_z^2 & -2n_zn_w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$n_x = planeNormal.x$$
$$n_y = planeNormal.y$$
$$n_z = planeNormal.z$$
$$n_w = -dot(planeNormal, planePos)$$

그림 9-30 미러 반사 변환 매트릭스

반사 매트릭스 변환을 메인 카메라의 위치 및 월드-카메라 매트릭스에 적용합니다. 그러면 반사 카메라의 위치 및 월드-카메라 매트릭스가 구해집니다.

반사 카메라의 투영 매트릭스는 메인 카메라의 투영 매트릭스와 동일해야 합니다.

반사 카메라가 반사를 텍스처로 렌더링합니다.

양호한 결과를 얻으려면 렌더링하기 전에 이 텍스처를 올바로 설정해야 합니다.
- 밉맵을 사용하십시오.
- 필터링 모드를 삼선형으로 설정하십시오.
- 멀티샘플링을 사용하십시오.

텍스처 크기가 반사 표면의 면적과 비례해야 합니다. 텍스처가 클수록 반사가 작게 픽셀화됩니다.

미러 카메라 스크립트 예제는 *http://wiki.unity3d.com/index.php/File:MirrorReflection.png*를 참조하십시오.

### 9.3.2 반사 결합 쉐이더 구현

쉐이더에서 정적 환경 반사를 동적 평면 반사와 결합할 수 있습니다.

쉐이더에서 반사를 결합하려면 *9.2.3 쉐이더 구현* 페이지의 9-157에서 제공한 쉐이더 코드를 수정해야 합니다.

쉐이더는 런타임 시 반사 카메라를 사용하여 렌더링되는 평면 반사를 통합해야 합니다. 이를 구현하기 위해 반사 카메라의 텍스처 `_ReflectionTex`가 uniform 변수로 조각 쉐이더로 전달되고 `lerp()` 함수를 사용하여 평면 반사 결과와 결합됩니다.

로컬 수정과 관련된 데이터 이외에 정점 쉐이더는 내장 함수 `ComputeScreenPos()`를 사용하여 정점의 화면 좌표도 계산합니다. 이 좌표가 조각 쉐이더로 전달됩니다.

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;

    // Transform vertex coordinates from local to world.
    float4 vertexWorld = mul(_Object2World, input.vertex);
```

```
    // Transform normal to world coordinates.
    float4 normalWorld = mul(float4(input.normal,0.0), _World2Object);

    // Final vertex output position.
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);

    // ----------- Local correction ------------
    output.vertexInWorld = vertexWorld.xyz;
    output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
    output.normalInWorld = normalWorld.xyz;

    // ----------- Planar reflections ------------
    output.vertexInScreenCoords = ComputeScreenPos(output.pos);
    return output;
}
```

평면 반사는 조각 쉐이더가 조각의 화면 좌표에 액세스할 수 있도록 텍스처로 렌더링됩니다. 이를 구현하기 위해 정점 화면 좌표를 조각 쉐이더에 varying 변수로 전달합니다.

조각 쉐이더에서

- 반사 벡터에 로컬 수정을 적용합니다.
- 로컬 큐브맵에서 환경 반사의 색상 `staticReflColor`를 검색합니다.

다음 코드는 로컬 큐브맵 기법을 사용한 정적 환경 반사를 미러 카메라 기법을 사용하여 런타임 시 렌더링되는 동적 평면 반사와 결합하는 방법입니다.

```
float4 frag(vertexOutput input) : COLOR
{
    float4 staticReflColor = float4(1, 1, 1, 1);

    // Find reflected vector in WS.
    float3 viewDirWS = normalize(input.viewDirInWorld);
    float3 normalWS = normalize(input.normalInWorld);
    float3 reflDirWS = reflect(viewDirWS, normalWS);

    // Working in World Coordinate System.
    float3 localPosWS = input.vertexInWorld;
    float3 intersectMaxPointPlanes = (_BBoxMax - localPosWS) / reflDirWS;
    float3 intersectMinPointPlanes = (_BBoxMin - localPosWS) / reflDirWS;

    // Look only for intersections in the forward direction of the ray.
    float3 largestParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);

    // Smallest value of the ray parameters gives us the intersection.
    float distToIntersect = min(min(largestParams.x, largestParams.y), largestParams.z);

    // Find the position of the intersection point.
    float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;

    // Get local corrected reflection vector.
    float3 localCorrReflDirWS = intersectPositionWS - _EnviCubeMapPos;

    // Lookup the environment reflection texture with the right vector.
    float4 staticReflColor = texCUBE(_Cube, localCorrReflDirWS);

    // Lookup the planar runtime texture
    float4 dynReflColor = tex2Dproj(_ReflectionTex,
    UNITY_PROJ_COORD(input.vertexInScreenCoords);

    //Revert the blending with the background color of the reflection camera
    dynReflColor.rgb /= (dynReflColor.a < 0.00392)?1:dynReflColor.a;

    // Combine static environment reflections with dynamic planar reflections
    float4 combinedRefl = lerp(staticReflColor.rgb, dynReflColor.rgb, dynReflColor.a);

    // Lookup the texture color.
    float4 texColor = tex2D(_MainTex, float2(input.tex));
    return _AmbientColor + texColor * _ReflAmount * combinedRefl;
}
```

평면 런타임 반사 텍스처 `_ReflectionTex`에서 텍스처 색상 `dynReflColor`를 추출합니다.

쉐이더에서 `_ReflectionTex`를 uniform 변수로 선언합니다.

속성 블록에서 `_ReflectionTex`를 선언합니다. 그러면 런타임 시 텍스처가 어떻게 보이는지 확인이 가능하고, 이는 게임을 개발하는 동안 디버깅에 도움이 됩니다.

텍스처 조회를 위해 텍스처 좌표를 투영합니다. 즉, 텍스처 좌표를 좌표 벡터의 마지막 성분으로 나눕니다. 이를 위해 Unity 내장 함수 UNITY_PROJ_COORD()를 사용할 수 있습니다.

lerp() 함수를 사용하여 정적 환경 반사와 동적 평면 반사를 결합합니다. 다음을 결합합니다.

- 반사 색상.
- 반사 표면의 텍스처 색상.
- 주변 색상 구성요소.

### 9.3.3 원거리 환경 반사 결합

정적 및 동적 객체로부터 반사를 렌더링할 때 원거리 환경으로부터의 반사도 고려해야 할 수 있습니다. 예를 들어 로컬 환경의 창을 통한 하늘로부터의 반사입니다.

이 경우 세 유형의 반사를 결합해야 합니다.

- 로컬 큐브맵 기법을 사용한 정적 환경의 반사.
- 미러 카메라 기법을 사용한 동적 객체의 평면 반사.
- 표준 큐브맵 기법을 사용한 스카이박스의 반사. 큐브맵에서 텍스처를 가져오기 전에 반사 벡터를 수정할 필요가 없습니다.

스카이박스로부터의 반사를 통합하려면 반사 벡터 reflDirWS를 사용하여 스카이박스 큐브맵에서 텍셀을 가져옵니다. 스카이박스 큐브맵 텍스처를 쉐이더로 uniform 변수로 전달합니다.

———— 참고 ————

로컬 수정은 적용하지 마십시오.

————————————

창에서 스카이박스만 보이도록 정적 큐브맵의 반사를 베이크할 때 알파 채널에서 장면의 투명도를 렌더링합니다.

불투명 지오메트리에는 값 1을 할당하고, 지오메트리가 없거나 완전히 투명한 경우 값 0을 할당합니다. 예를 들어 알파 채널이 0인 창에 해당하는 픽셀을 렌더링합니다.

스카이박스 큐브맵 _Skybox를 쉐이더에 uniform 변수로 전달합니다.

*9.3.2 반사 결합 쉐이더 구현* 페이지의 9-171의 조각 쉐이더 코드에서 다음 주석을 찾습니다.

```
// Lookup the planar runtime texture
```

이 주석 앞에 다음 행을 삽입합니다.

```
float4 skyboxReflColor = texCUBE(_Skybox, reflDirWS);
staticReflColor = lerp(skyboxReflColor.rgb, staticReflColor.rgb, staticReflColor.a);
```

이 코드는 정적 반사를 스카이박스로부터의 반사와 결합합니다.

다음 그림은 다른 유형의 반사의 결합을 보여줍니다.


그림 9-31 다른 유형의 반사를 결합

## 9.4 로컬 큐브맵을 기반으로 한 동적 소프트 그림자

이 기법은 로컬 큐브맵을 사용하여 정적 객체의 환경 투명도를 표시하는 텍스처를 유지합니다. 이는 고품질 소프트 그림자를 생성하는 매우 효율적인 기법입니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.4.1 로컬 큐브맵을 기반으로 한 동적 소프트 그림자 개요

장면에는 움직이는 객체와 방과 같은 정적 객체가 있습니다. 이 기법을 사용하면 프레임마다 정적 지오메트리를 그림자 맵으로 렌더링할 필요가 없습니다. 그러므로 텍스처를 사용하여 그림자를 표현할 수 있습니다.

큐브맵은 얼음 동굴 데모의 동굴과 같이 불규칙한 형태를 비롯해 수많은 종류의 정적 환경에 대한 훌륭한 근시치가 될 수 있습니다. 또한 알파 채널은 방으로 들어오는 빛의 양을 나타냅니다.

움직이는 객체는 일반적으로 방을 제외한 모든 것입니다. 다음과 같은 객체입니다.

- 태양.
- 카메라.
- 동적 객체.

방 전체를 큐브 텍스처로 표현하면 조각 쉐이더 내에서 임의의 환경 텍셀에 액세스할 수 있습니다. 그러므로 예를 들어 태양을 임의의 위치에 배치할 수 있고 큐브맵에서 가져온 값을 기준으로 조각에 닿는 빛의 양을 계산할 수 있습니다.

알파 채널, 즉 투명도는 로컬 환경으로 들어오는 빛의 양을 나타냅니다. 장면에서 그림자를 추가하려는 정적 및 동적 객체를 렌더링하는 조각 쉐이더에 큐브맵 텍스처를 연결합니다.

### 9.4.2 그림자 큐브맵 생성

환경 외부의 광원으로부터의 그림자를 적용할 로컬 환경부터 시작합니다. 예를 들어 방, 동굴, 케이지입니다.

이 기법은 로컬 큐브맵을 기반으로 한 반사와 유사합니다. 자세한 내용은 *9.2 로컬 큐브맵을 사용하여 반사 구현* 페이지의 9-153을 참조하십시오.

그림자 큐브맵을 반사 큐브맵과 동일한 방식으로 생성하되 반드시 알파 채널도 추가해야 합니다. 알파 채널, 즉 투명도는 로컬 환경으로 들어오는 빛의 양을 나타냅니다.

큐브맵의 6면을 렌더링할 기준 위치를 결정합니다. 대부분의 경우 이 위치는 로컬 환경의 경계 상자의 중심입니다. 큐브맵을 생성하려면 이 위치가 필요합니다. 또한 로컬 수정 벡터를 계산하여 큐브맵에서 올바른 텍셀을 가져오려면 이 위치를 쉐이더로 전달해야 합니다.

큐브맵의 중심을 배치할 위치를 결정했으면 큐브맵 텍스처의 모든 면을 렌더링하고 로컬 환경의 투명도(알파)를 기록할 수 있습니다. 영역이 투명할수록 환경으로 들어오는 빛이 많습니다. 지오메트리가 없는 경우 영역은 완전히 투명합니다. 필요한 경우, RGB 채널을 사용하여 스테인드 글라스, 반사, 굴절과 같은 유색 그림자의 환경 색상을 저장할 수 있습니다.

### 9.4.3 그림자 렌더링

월드 공간에서 하나의 정점 또는 조각에서 라이트(들)까지의 벡터 $P_iL$을 만들고 이 벡터를 사용하여 큐브맵 그림자를 가져옵니다.

각 텍셀을 가져오기 전 $P_iL$ 벡터에 로컬 수정을 적용해야 합니다. ARM는 보다 정확하게 그림자를 구현하기 위해 프래그먼트 셰이더에서 로컬 수정을 연산하도록 권장합니다.

로컬 수정을 연산하려면 환경 경계 상자와 조각-조명 벡터의 교차점을 계산해야 합니다. 이 교차점을 사용하여 큐브맵 원점 위치 C에서 교차점 P까지 벡터를 또 만듭니다. 이 벡터가 텍셀을 가져올 때 사용하는 최종 벡터 CP입니다.

로컬 수정을 연산하려면 다음 입력 매개 변수가 필요합니다.

- `_EnviCubeMapPos` 큐브맵 원점 위치.
- `_BBoxMax` 환경 경계 상자의 최대점.
- `_BBoxMin` 환경 경계 상자의 최소점.
- `Pi` 월드 공간 내 조각 위치.
- `PiL` 월드 공간 내 정규화된 조각-조명 벡터.

출력 값 CP를 계산합니다. 이 값이 그림자 큐브맵에서 텍셀을 가져올 때 사용하는 수정된 조각-조명 벡터입니다.

다음 그림은 조각-조명 벡터의 로컬 수정을 보여줍니다.

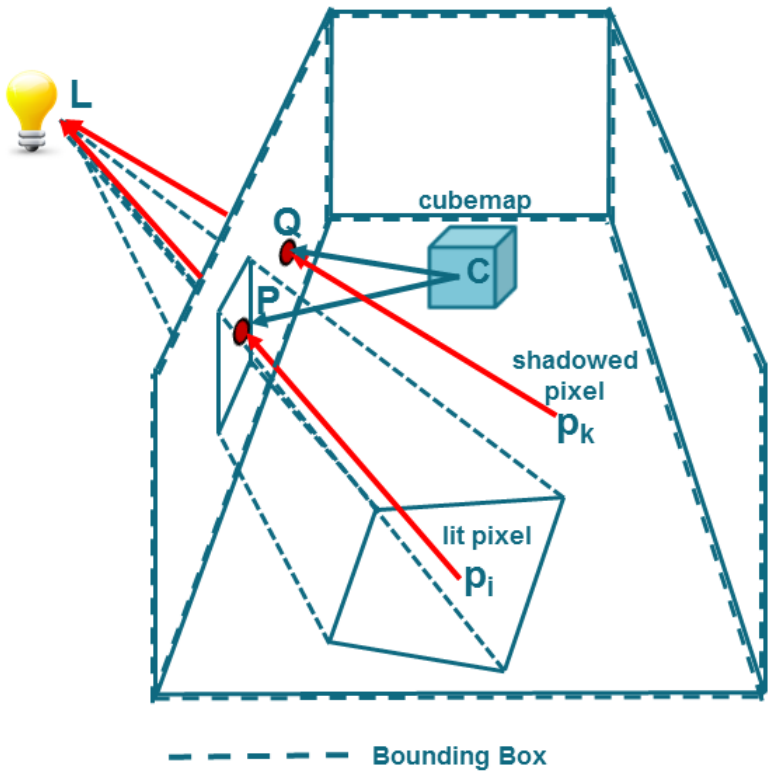


그림 9-32 조각-조명 벡터의 로컬 수정

다음 예제 코드는 올바른 CP 벡터를 계산하는 방법을 보여줍니다.

```
// Working in World Coordinate System.
vec3 intersectMaxPointPlanes = (_BBoxMax - Pi) / PiL;
vec3 intersectMinPointPlanes = (_BBoxMin - Pi) / PiL;

// Looking only for intersections in the forward direction of the ray.
vec3 largestRayParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);

// Smallest value of the ray parameters gives us the intersection.
float dist = min(min(largestRayParams.x, largestRayParams.y), largestRayParams.z);

// Find the position of the intersection point.
vec3 intersectPositionWS = Pi + PiL * dist;

// Get the local corrected vector.
CP = intersectPositionWS - _EnviCubeMapPos;
```

CP 벡터를 사용하여 큐브맵에서 텍셀을 가져옵니다. 텍셀의 알파 채널은 조각에 조명 또는 그림자를 얼마나 적용할지에 대한 정보를 제공합니다.

```
float shadow = texCUBE(cubemap, CP).a;
```

다음 그림은 하드한 그림자가 적용된 체스룸을 보여줍니다.


그림 9-33 하드 그림자가 적용된 체스룸

이 기법은 장면에서 작용하는 그림자를 생성하지만 추가로 2개 단계를 통해 그림자 품질을 개선할 수 있습니다.

- 그림자 내 후면
- 부드러움

### 그림자 내 후면

큐브맵 그림자 기법은 깊이 정보를 사용하여 그림자를 적용하지 않습니다. 이는 그림자 안에 있어야 할 일부 면에 잘못해서 빛이 비친다는 의미입니다.

이 문제는 표면이 조명과 반대 방향을 향할 때만 발생합니다. 이 문제를 해결하려면 노말 벡터와 조각-조명 벡터 사이의 각도 $P_iL$을 확인합니다. 각도가 -90도 ~ 90도 단위를 벗어날 경우 표면이 그림자 안에 들어갑니다.

다음 코드 조각은 이 확인을 위한 것입니다.

```
if (dot(PiL,N) < 0)
shadow = 0.0;
```

위 코드는 각 삼각형을 조명에서 음영으로 하드 스위칭합니다. 부드럽게 전환하려면 다음 공식을 사용합니다.

```
shadow *= max(dot(PiL, N), 0.0);
```

인수 설명:

- shadow는 그림자 큐브맵에서 가져온 알파 값입니다.
- $P_iL$은 월드 공간 내 조각-조명 벡터입니다.
- N은 월드 공간 내 표면의 노말 벡터입니다.

다음 그림은 그림자 내 후면이 적용된 체스룸을 보여줍니다.


그림 9-34 그림자 내 후면이 적용된 체스룸

### 부드러움

이 그림자 기법은 장면에서 보다 사실적인 부드러운 그림자를 구현합니다.

1. 밉맵을 생성하고 큐브맵 텍스처에 대해 삼선형 필터링을 설정합니다.
2. 조각-교차점 벡터의 길이를 측정합니다.
3. 길이에 계수를 곱합니다.

이 계수는 환경에서 밉맵 레벨 수까지 최대 거리의 노멀라이저입니다. 이 값을 경계 볼륨 및 밉맵 레벨을 기준으로 자동으로 계산할 수 있습니다. 계수를 장면에 맞게 사용자 정의해야 합니다. 그러면 환경에 적합하게 설정을 수정하여 이미지 품질을 개선할 수 있습니다. 예를 들어 얼음 동굴 프로젝트에 사용된 계수는 `0.08`입니다.

로컬 수정에서 얻은 계산 결과를 재사용할 수 있습니다. 조각-조명 벡터와 경계 상자의 교차점에서 조각 위치까지 선분 길이로 로컬 수정을 위한 코드 조각의 `dist`를 재사용합니다.

```
float texLod = dist;
```

`texLod`에 거리 계수를 곱합니다.

```
texLod *= distanceCoefficient;
```

부드러움을 구현하려면 Cg 함수 `texCUBElod()` 또는 GLSL function `textureLod()`를 사용하여 텍스처의 올바른 밉맵 레벨을 가져옵니다.

`XYZ`가 방향 벡터를 나타내고 `W` 성분이 LOD를 나타내는 `vec4`를 만듭니다.

```
CP.w = texLod;
shadow = texCUBElod(cubemap, CP).a;
```

이 기법은 장면에서 고품질의 부드러운 그림자를 구현합니다.

다음 그림은 부드러운 그림자가 적용된 체스룸을 보여줍니다.


그림 9-35 부드러운 그림자

### 9.4.4 큐브맵과 그림자 맵 결합

동적 콘텐츠에서 그림자를 완전하게 구현하려면 큐브맵 그림자 기법과 전통적인 그림자 맵 기법과 결합해야 합니다. 그러면 작업은 늘어나지만 동적 객체만 그림자 맵에 렌더링하면 되므로 충분한 가치가 있습니다.

다음 그림은 부드러운 그림자만 적용된 체스룸을 보여줍니다.


그림 9-36 부드러운 그림자

다음 그림은 동적 그림자와 결합된 부드러운 그림자가 적용된 체스룸을 보여줍니다.


그림 9-37 동적 그림자와 결합된 부드러운 그림자

### 9.4.5 큐브맵 그림자 기법의 결과

전통적인 기법에서는 그림자를 만드는 각 광원의 퍼스펙티브에 대해 전체 장면을 렌더링하므로 그림자 렌더링은 연산 비용이 상당히 높을 수 있습니다. 여기서 설명하는 큐브맵 그림자 기법은 대부분이 프리베이크되므로 향상된 성능을 제공합니다.

또한 이 기법은 출력 해상도와 독립적입니다. 1080p, 720p 및 기타 해상도에서 시각적 품질이 동일합니다.

부드러움 필터링은 하드웨어에서 계산되므로 부드러움을 구현하는 연산 비용이 거의 없습니다. 그림자가 부드러울수록 기법이 더 효율적입니다. 밉맵 레벨이 낮을수록 전통적 그림자 맵 기법보다 데이터가 감소하기 때문입니다. 전통적 기법에서는 그림자를 시각적으로 보다 매력적으로 보일 정도로 부드럽게 만들려면 대규모 커널이 필요합니다. 그러면 큰 메모리 대역폭이 요구되므로 성능이 저하됩니다.

큐브맵 그림자 기법을 통해 실현되는 품질은 기대보다 높을 수 있습니다. 사실적인 부드러움과 에지 시머링 없는 안정적인 그림자를 구현합니다. 에지 시머링은 전통적 그림자 맵 기법을 사용할 때 래스터화 및 앨리어싱 효과로 인해 관찰될 수 있습니다. 하지만 어떤 안티앨리어싱 알고리즘도 이 문제를 완벽하게 해결하지는 못합니다.

큐브맵 그림자 기법은 시머링 문제가 없습니다. 렌더 타겟보다 훨씬 낮은 해상도를 사용하더라도 엣지가 안정적입니다. 출력보다 4배 낮은 해상도를 사용할 수 있으며 아티팩트 또는 원치 않는 시머링이 없습니다. 또한 4배 낮은 해상도를 사용하면 메모리 대역폭이 절약되어 성능이 개선됩니다.

이 기법은 OpenGL ES 2.0 이상을 사용하는 것과 같은 쉐이더를 지원하는 모든 상용 디바이스에서 사용할 수 있습니다. 로컬 큐브맵 기법을 기반으로 언제 어디서 반사를 사용할지 이미 알고 있는 경우 그림자 기법을 구현에 용이하게 적용할 수 있습니다.

———— 참고 ————

이 기법은 장면의 모든 곳에 사용할 수는 없습니다. 예를 들어 동적 객체는 큐브맵으로부터 그림자를 받지만 큐브맵 텍스처리 프리베이크될 수는 없습니다. 동적 객체의 경우 그림자 맵을 큐브맵 그림자 기법과 결합하여 그림자를 생성하십시오.

———————————

다음 그림은 그림자가 있는 얼음 동굴입니다.


그림 9-38 그림자가 있는 얼음 동굴

다음 그림은 부드러운 그림자가 적용된 얼음 동굴입니다.


그림 9-39 부드러운 그림자가 적용된 얼음 동굴

다음 그림은 부드러운 그림자가 적용된 얼음 동굴입니다.

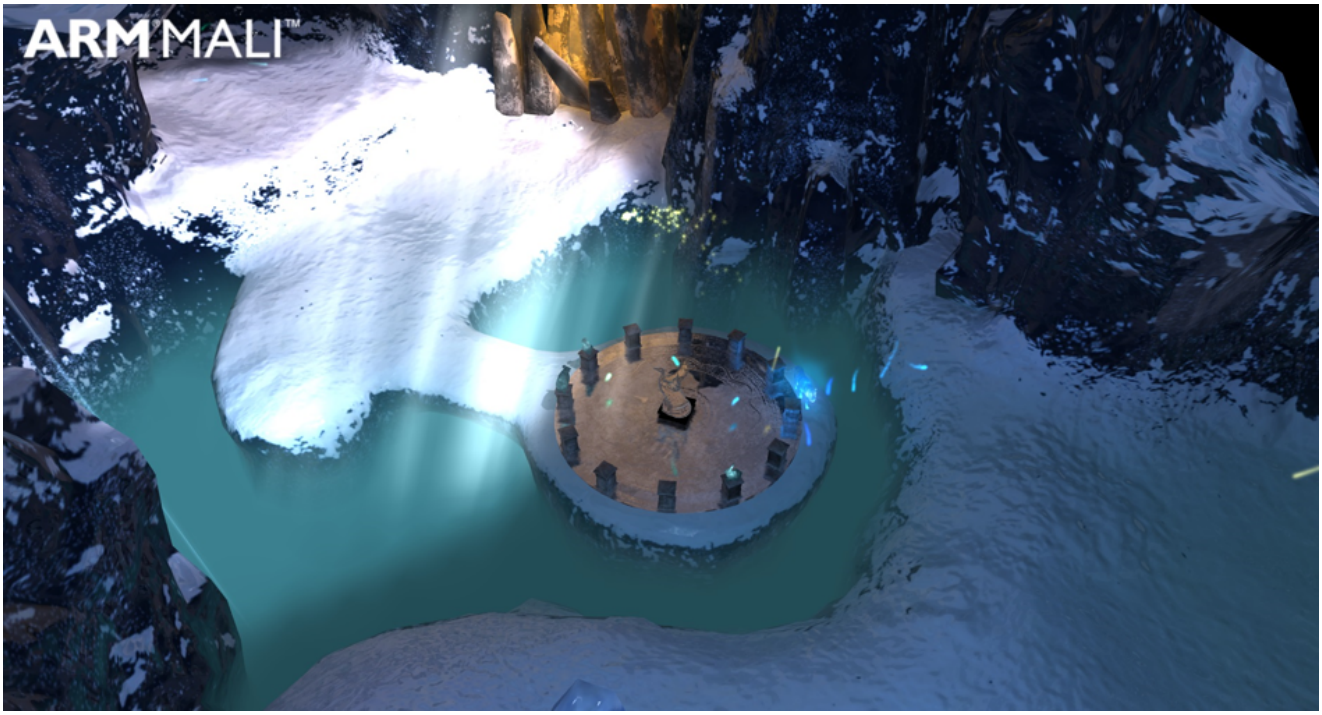


그림 9-40 부드러운 그림자가 적용된 얼음 동굴

## 9.5 로컬 큐브맵을 기반으로 한 굴절

로컬 큐브맵을 사용하여 고품질 굴절을 구현할 수 있습니다. 런타임 시 굴절을 반사와 결합할 수 있습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.5.1 굴절 개요

게임 개발자는 게임에서 시각적으로 인상적인 효과를 구현할 효율적인 방법을 늘 찾고 있습니다. 모바일 플랫폼을 대상으로 하는 경우에는 이것이 특히 중요한데, 성능을 극대화하기 위해 세심하게 리소스 균형을 맞춰야 하기 때문입니다.

굴절은 빛이 통과하는 매질의 변화로 인한 광파 방향의 변화입니다. 반투명 지오메트리에서 추가로 사실성을 원한다면 고려해야 할 중요한 효과는 굴절입니다.

굴절률에 따라 빛이 어떤 재료로 입사될 때 얼마나 많이 꺾이거나 굴절되는지 결정됩니다. 굴절은 빛이 굴절률 $n_1$의 한 매질에서 굴절률 $n_2$의 다른 매질로 통과할 때 빛이 꺾이는 것으로 정의됩니다.

굴절률과 입사각 및 굴절각의 사인 사이의 관계는 스넬의 법칙을 사용하여 계산합니다.

다음 그림은 스넬의 법칙과 굴절을 보여줍니다.

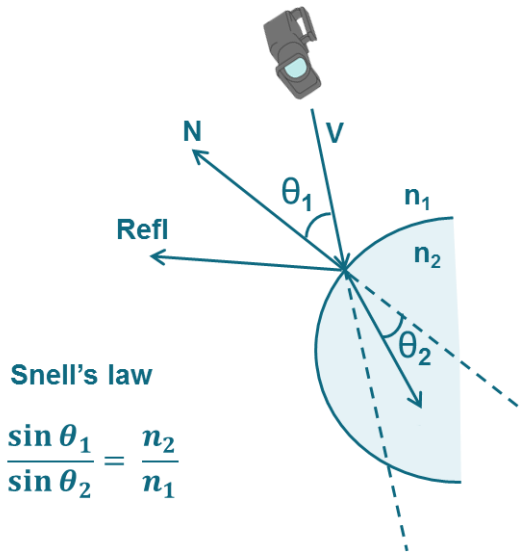


그림 9-41 빛이 한 매질을 통해 다른 매질로 들어갈 때 빛의 굴절

### 9.5.2 굴절 구현

개발자들은 반사를 렌더링하기 시작한 이후 굴절을 렌더링하기 위해 노력해 왔습니다. 이러한 프로세스는 모든 반투명 표면에서 발생하기 때문입니다. 반사를 렌더링하는 기법은 다수 있지만 굴절의 경우에는 별로 없습니다.

런타임 시 굴절을 구현하는 기존의 방법은 굴절 유형에 따라 다릅니다. 대부분의 기법은 런타임 시 굴절성 객체 뒤 장면을 텍스처로 렌더링한 후 두 번째 패스에서 텍스처 왜곡을 굴절된 모양에 적용합니다. 텍스처 왜곡에 따라 이 방법을 사용하여 물, 아지랑이, 유리 등과 같은 굴절 효과를 렌더링할 수 있습니다.

이 중 일부 기법은 양호한 결과를 얻을 수 있지만 텍스처 왜곡이 물리적 기반이 아니므로 결과가 항상 정확하지는 않습니다. 예를 들어 굴절 카메라의 시점에서 텍스처를 렌더링할 경우 카메라에서 직접 보이지는 않지만 물리적 기반 굴절에서는 보이는 영역이 있을 수 있습니다.

render-to-texture 방법을 사용할 경우 주된 제약은 품질입니다. 카메라가 이동할 경우 픽셀 시머링 또는 픽셀 불안정성이 자주 관찰됩니다.

### 9.5.3 로컬 큐브맵을 기반으로 한 굴절 개요

로컬 큐브맵은 반사를 렌더링하기 위한 뛰어난 기법이며 개발자들은 이 기법이 개발된 이후로 정적 큐브맵을 사용하여 반사와 굴절을 모두 구현해 왔습니다.

하지만 로컬 환경에서 정적 큐브맵을 사용하여 반사 또는 굴절을 구현할 경우, 로컬 수정을 적용하지 않으면 결과가 부정확해집니다.

여기서 설명하는 기법에서는 올바른 결과를 얻기 위해 수정을 적용합니다. 이 기법은 고도로 최적화되어 있습니다. 런타임 리소스가 제한되어 세심하게 균형을 맞춰야 하는 모바일 디바이스에서 특히 유용합니다.

### 9.5.4 큐브맵 준비

굴절 구현에서 사용할 큐브앱을 준비해야 합니다.

큐브맵을 준비하려면 다음을 수행합니다.

1. 카메라를 굴절성 지오메트리의 중앙에 배치합니다.
2. 굴절성 객체를 숨기고 주위의 정적 환경을 6방향에서 큐브맵으로 렌더링합니다. 이 큐브맵을 굴절 및 반사를 구현하는 데 모두 사용할 수 있습니다.
3. 굴절성 객체를 둘러싼 환경을 정적 큐브맵으로 베이크합니다.
4. 굴절 벡터의 방향을 결정하고 로컬 환경의 경계 상자와 교차하는 지점을 찾습니다.
5. *9.4 로컬 큐브맵을 기반으로 한 동적 소프트 그림자* 페이지의 9-175에서와 동일한 방식으로 로컬 수정을 적용합니다.
6. 큐브맵이 생성된 위치에서 교차점까지 새 벡터를 빌드합니다. 이 최종 벡터를 사용하여 큐브맵에서 텍셀을 가져와 굴절성 객체의 뒤쪽을 렌더링합니다.

굴절된 벡터 $R_{rf}$를 사용하여 큐브맵으로부터 텍셀을 가져오는 대신, 굴절된 벡터가 경계 상자와 교차하는 지점 P를 찾고 큐브맵의 중앙 C에서 교차점 P까지 새로운 벡터 $R'_{rf}$을 빌드합니다. 이 새 벡터를 사용하여 큐브맵에서 텍스처 색상을 가져옵니다.

```
float eta=n2/n1;
```

float3$R_{rf}$ = refract(D,N,eta);

Find intersection point P

Find vector $R'_{rf}$ = CP;

Float4 col = texCube(Cubemap, $R'_{rf}$);

다음 그림은 큐브맵 및 굴절 벡터가 있는 장면입니다.

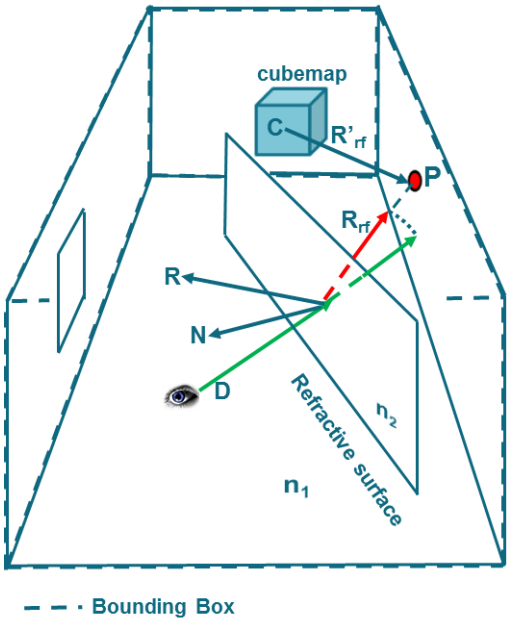


그림 9-42 굴절 벡터에 대한 로컬 수정

이 기법에 의해 생성되는 굴절은 스넬 법칙을 사용하여 굴절 벡터의 방향이 계산되므로 정확하게 물리에 기반한 것입니다.

또한 엄격하게 스넬 법칙에 따라 굴절 벡터 R을 찾기 위해 쉐이더에서 사용할 수 있는 내장 함수도 있습니다.

```
R = refract( I, N, eta);
```

인수 설명:

- I는 정규화된 뷰 또는 입사 벡터입니다.
- N은 정규화된 노말 벡터입니다.
- eta는 굴절 지수 비율 $n_1/n_2$입니다.

다음 그림은 로컬 큐브맵을 기반으로 굴절을 구현하는 쉐이더의 흐름입니다.

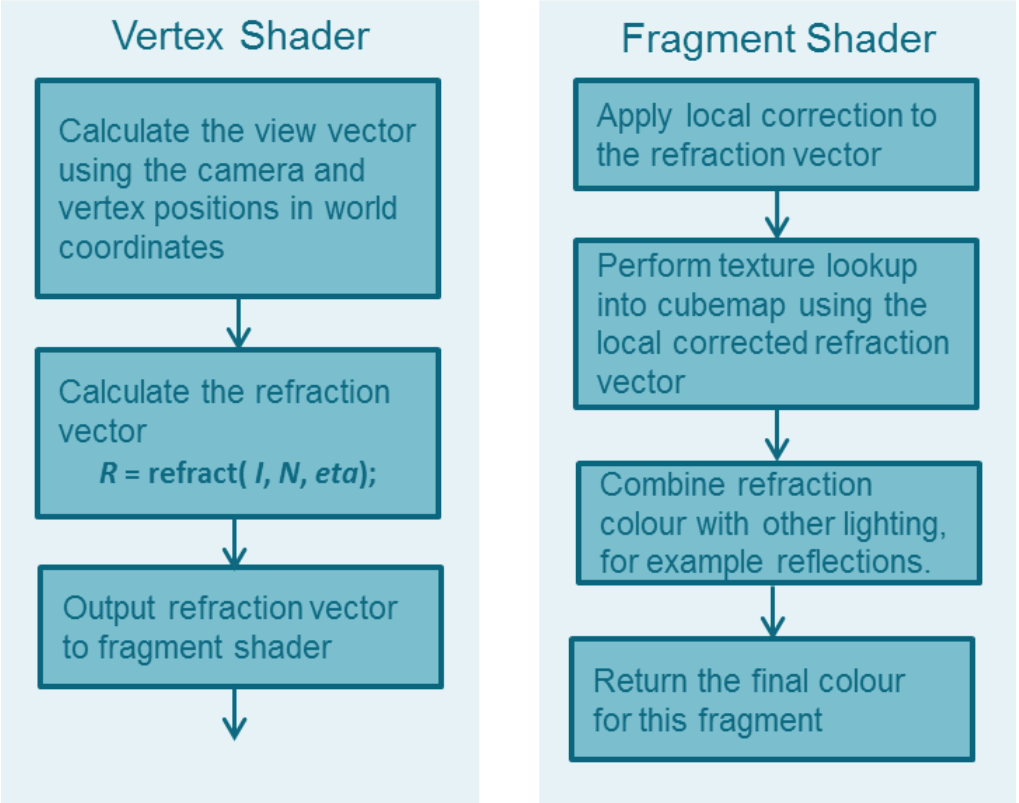


그림 9-43 로컬 큐브맵을 기반으로 한 굴절의 쉐이더 구현

### 9.5.5 쉐이더 구현

로컬 수정된 굴절 방향에 따라 텍셀을 가져올 때 굴절 색상을 다른 조명과 결합할 수 있습니다. 예를 들어 굴절과 동시에 발생하는 반사입니다.

굴절 색상을 다른 조명과 결합하려면 추가 뷰 벡터를 조각 쉐이더에 전달하고 여기에 로컬 수정을 적용해야 합니다. 이 결과를 사용하여 동일한 큐브맵에서 굴절 색상을 가져옵니다.

다음 코드 조각은 반사와 굴절을 결합하여 최종 출력 색상을 생성하는 방법을 보여줍니다.

```
// ----------------- Environment reflections ----------------
float3 newReflDirWS = LocalCorrect(input.reflDirWS, _BBoxMin, _BBoxMax, input.posWorld,
_EnviCubeMapPos);
float4 staticReflColor = texCUBE(_EnviCubeMap, newReflDirWS);
// ----------------- Environment refractions ----------------
float3 newRefractDirWS = LocalCorrect(RefractDirWS, _BBoxMin, _BBoxMax, input.posWorld,
_EnviCubeMapPos);
float4 staticRefractColor = texCUBE(_EnviCubeMap, newRefractDirWS);
// ----------------- Combined reflections and refractions ----------------
float4 combinedReflRefract = lerp(staticReflColor, staticRefractColor, _ReflAmount);

float4 finalColor = _AmbientColor + combinedReflRefract;
```

계수 `_ReflAmount`는 조각 쉐이더에 uniform 변수로 전달됩니다. 이 계수를 사용하여 반사와 굴절 간 균형을 조정합니다. 사용자가 직접 `_ReflAmount`를 조정하여 필요한 시각적 효과를 달성할 수 있습니다.

`LocalCorrect` 함수의 구현은 반사 블로그(*http://community.arm.com/groups/arm-mali-graphics/blog/2014/08/07/reflections-based-on-local-cubemaps*)를 참조하십시오.

굴절성 지오메트리가 중공 객체인 경우 굴절 및 반사가 전방 및 후방 표면 모두에서 발생합니다.

다음 그림은 로컬 큐브맵을 기반으로 한 유리 비숍에서의 굴절입니다.

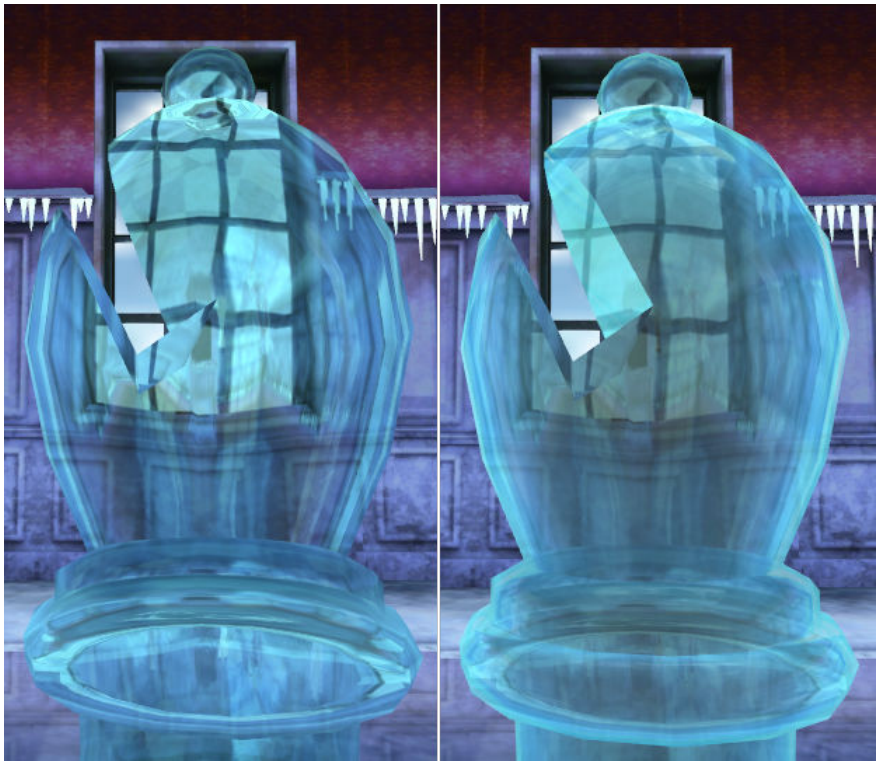

그림 9-44 로컬 큐브맵을 기반으로 한 유리 비숍에서의 굴절

좌측 이미지는 로컬 굴절 및 반사를 포함하여 후면만 렌더링하는 첫 번째 패스를 보여줍니다.

우측 이미지는 로컬 굴절 및 반사와 첫 번째 패스와의 알파 블렌딩을 포함하여 전면만 렌더링하는 두 번째 패스를 보여줍니다.

- 첫 번째 패스에서는 반투명 객체를 불투명 지오메트리를 렌더링하는 방식과 동일하게 렌더링합니다. 전면 컬링이 켜진 객체는 마지막에 렌더링합니다. 즉, 후면만 렌더링합니다. 다른 객체는 오클루드하면 안 될 경우 깊이 버퍼에는 쓰지 마십시오.

  후면 색상은 반사, 굴절에서 계산된 색상과 객체 자체의 디퓨즈 색상을 혼합하여 구합니다.
- 두 번째 패스에서는 후면 컬링을 포함하여 전면을 렌더링합니다. 이 단계는 렌더링 큐에서 마지막으로 실행합니다. 깊이 쓰기가 꺼져 있는지 확인하십시오. 굴절 및 반사 텍스처를 디퓨즈 색상과 혼합하여 전면 색상을 구합니다. 두 번째 패스의 굴절은 최종 렌더링에서 사실성을 더해줍니다. 후면에서의 굴절이 효과를 강조하는 데 충분할 경우 이 단계를 건너뛸 수 있습니다.
- 최종 패스에서 결과 색상을 첫 번째 패스와 알파 블렌딩합니다.

다음 그림은 얼음 동굴 데모의 반투명 불사조에 대해 로컬 큐브맵을 기반으로 굴절을 구현한 결과입니다.

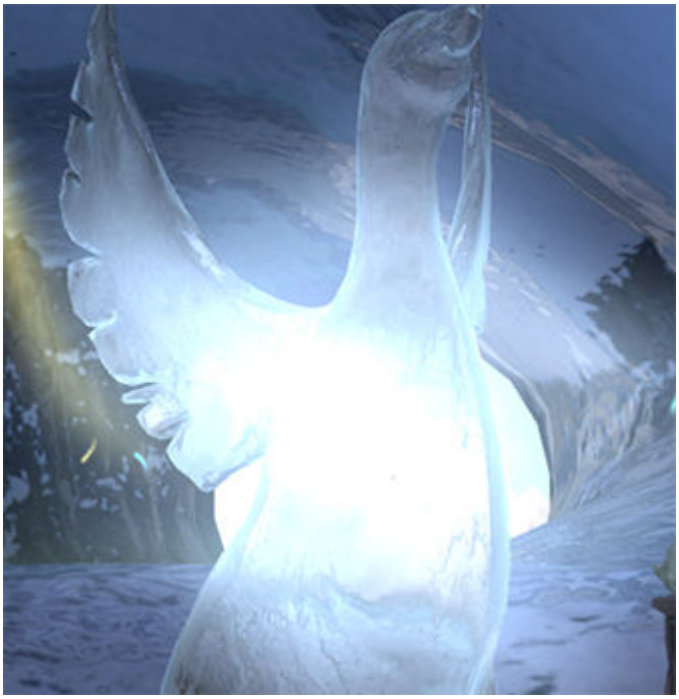

그림 9-45 반투명 불사조 굴절

다음 그림은 반투명 불사조 날개입니다.

그림 9-46 반투명 불사조 날개

## 9.6 얼음 동굴 데모의 반사 효과

얼음 동굴 데모의 반사 효과는 Blinn 기법을 사용하여 구현됩니다. 이 기법은 매우 효율적이며, 뛰어난 결과를 보여줍니다.

다음 코드는 Blinn 기법을 사용하여 반사 효과를 구현하는 방법을 보여줍니다.

```
// Returns intensity of a specular effect without taking into account shadows
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 기법의 단점은 일부 상황에서 잘못된 결과를 생성할 수 있다는 점입니다. 예를 들어 그림자 영역에서 반사 효과가 나타날 수 있습니다.

다음 그림은 반사 효과가 없는 그림자 영역의 예입니다.

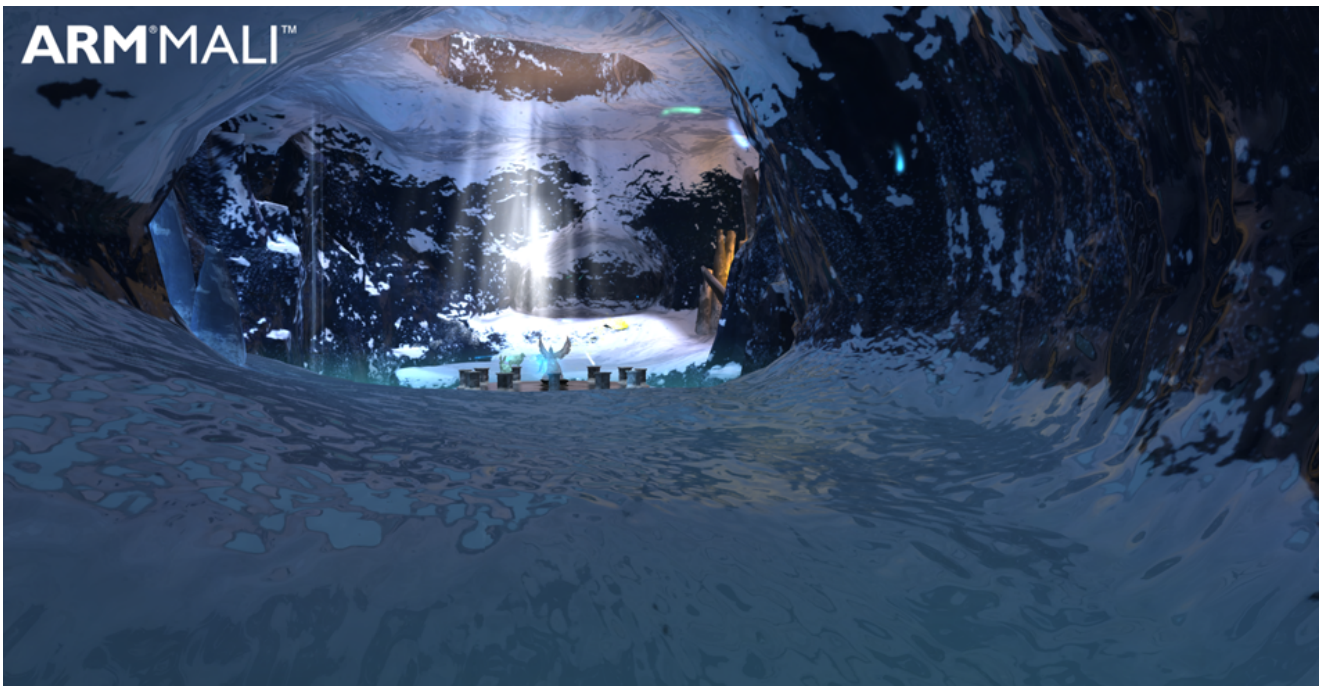


그림 9-47 그림자 영역

다음 그림은 그림자 영역에 나타난 반사 효과의 예입니다. 이는 잘못된 것입니다.

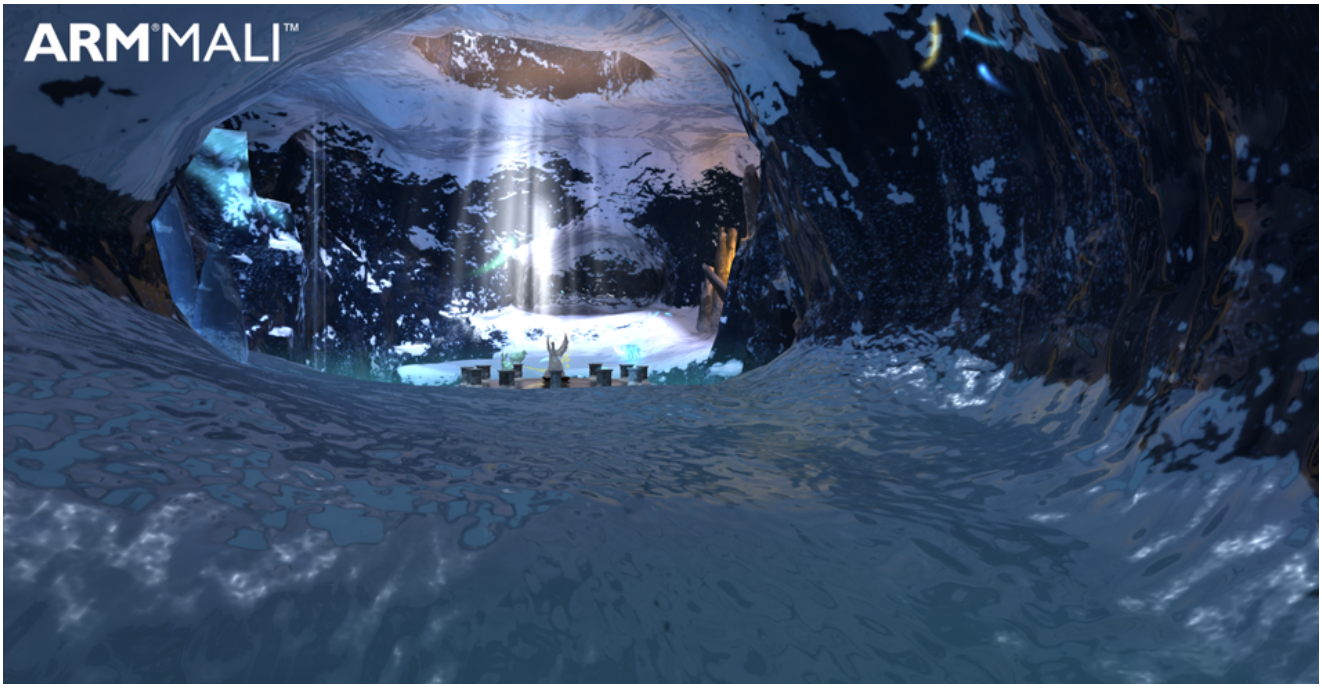


그림 9-48 잘못된 반사 효과가 있는 그림자 영역

다음 그림은 조명 영역에서 보이는 반사 효과의 예입니다. 이는 올바른 것입니다.

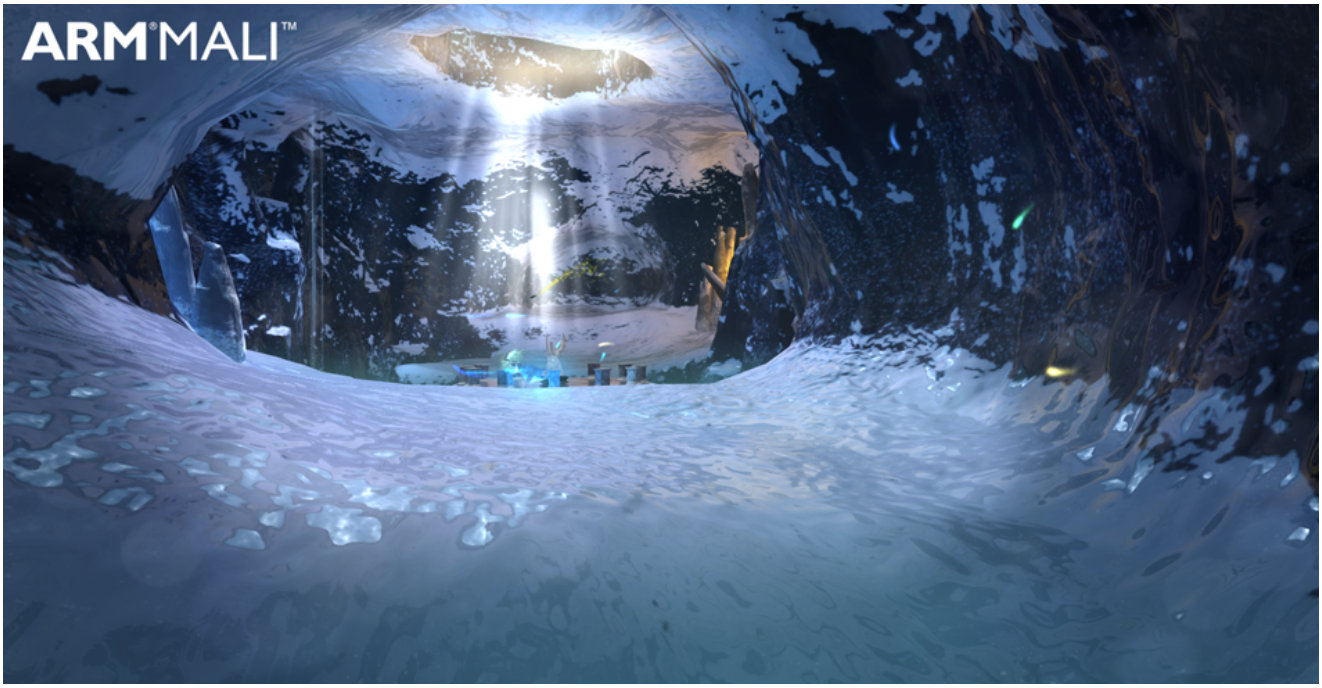


그림 9-49 올바른 반사 효과가 있는 조명 영역

그림자는 반사 표면에 닿는 빛에 따라 반사 효과를 더 강하거나 더 약하게 만들어야 합니다. 반사 효과의 강도를 수정하는 데 필요한 모든 정보는 얼음 동굴 데모에 이미 있으므로 수정은 비교적 간단합니다. 반사 및 그림자 효과에 사용되는 환경 큐브맵 텍스처는 두 가지 유형의 정보를 포함합니다. RGB 채널은 반사에 사용되는 환경 색상을 포함합니다. 알파 채널은 불투명도를 포함하며 이 정보는 그림자에 사용됩니다. 알파 채널을 사용하여 반사 강도를 결정할 수 있습니다. 알파 채널은 동굴에서 환경으로 빛이 들어오는 구멍을 나타내기 때문입니다. 따라서 알파 채널은 반사 효과가 조명이 비추는 표면에만 적용되도록 하는 데 사용됩니다.

이렇게 하려면 조각 쉐이더에서 수정된 반사 벡터를 계산하여 반사 효과를 생성합니다. 수정된 반사 벡터 생성에 대한 자세한 내용은 *9.2 로컬 큐브맵을 사용하여 반사 구현* 페이지의 9-153을 참조하십시오. 이 벡터를 사용하여 큐브맵 텍스처에서 RGBA 텍셀을 가져옵니다.

```
// Locally corrected static reflections
 const half4 reflColor = SampleCubemapWithLocalCorrection(
        ReflDirectionWS,
        _ReflBBoxMinWorld,
        _ReflBBoxMaxWorld,
        input.vertexInWorld,
        _ReflCubePosWorld,
        _ReflCube);
```

`SampleCubemapWithLocalCorrection()` 함수에 대한 자세한 내용은 *9.2 로컬 큐브맵을 사용하여 반사 구현* 페이지의 9-153을 참조하십시오.

`reflColor`는 RGBA 형식으로, RGB 구성요소가 반사의 색상 데이터를 포함하고 알파 채널이 반사 효과의 강도를 포함합니다. 얼음 동굴 데모에서는 알파 채널에 반사 색상(Blinn 기법을 사용하여 계산됨)이 곱해집니다.

```
half3 specular = _SpecularColor.rgb *
        SpecularBlinn(
            input.vertexToLight01InWorld,
            viewDirInWorld,
            normalInWorld,
            _SpecularPower) *
        reflColor.a;
```

`specular` 값은 최종 반사 색상을 나타냅니다. 조명 모델에 이 값을 추가할 수 있습니다.

## 9.7 Early-z 사용

Mali GPU에는 Early-Z 알고리즘을 처리하는 기능이 포함되어 있습니다. Early-Z는 오버드로된 조각을 제거하여 성능을 개선합니다.

일반적으로 Mali GPU는 대부분의 콘텐츠에 Early-Z 알고리즘을 실행하지만 정확성을 유지하기 위해 이 알고리즘을 실행하지 않는 경우도 있습니다. 이 알고리즘은 Unity 엔진과 컴파일러에서 생성된 코드 모두에 의존하므로 Unity에서는 이를 제어하기 힘들 수 있습니다. 하지만 몇 가지 징후를 찾아볼 수 있습니다.

쉐이더를 모바일용으로 컴파일하고 코드를 확인합니다. 쉐이더가 다음 범주 중 하나에 속하지 않는지 확인합니다.

쉐이더에 부작용이 있음
: 이 의미는 쉐이더 스레드가 실행 도중 전역 상태를 수정하므로 쉐이더를 두 번째 실행하면 다른 결과가 나올 수 있다는 것입니다. 일반적으로 이는 쉐이더가 쉐이더 저장 버퍼 객체 또는 이미지와 같은 공유된 읽기/쓰기 메모리 버퍼에 기록한다는 것을 의미합니다. 예를 들어 성능을 측정하기 위해 카운터를 증분하는 쉐이더를 작성할 경우 이 쉐이더에는 부작용이 있습니다.
다음은 부작용으로 분류되지 않습니다.
- 읽기 전용 메모리에 액세스.
- 쓰기 전용 버퍼에 기록.
- 순수 논리 메모리에 액세스.

쉐이더가 discard()를 호출
: 조각 쉐이더가 실행 도중 `discard()`를 호출할 수 있는 경우 Mali GPU가 Early-Z를 활성화할 수 없습니다. 조각 쉐이더가 현재 조각을 폐기할 수 있지만 Early-Z 테스트에 의해 이미 수정된 깊이 값은 복구될 수 없기 때문입니다.

Alpha-to-coverage가 활성화됨
: Alpha-to-coverage가 활성화된 경우 조각 쉐이더가 나중에 액세스된 데이터를 계산하여 알파를 정의합니다.
예를 들어 나뭇잎을 렌더링하는 경우 일반적으로 이들은 평면으로 표현되고 텍스처는 나뭇잎의 어떤 부분이 투명 또는 불투명한지 정의합니다. Early-z가 활성화된 경우 정면의 일부가 평면의 투명 부분에 의해 오클루드될 수 있어 잘못된 결과가 얻어집니다.

Depth source not fixed 함수
: 깊이 테스트에 사용되는 깊이 값은 정점 쉐이더에서 제공하지 않습니다. 조각 쉐이더가 `gl_FragDepth`에 기록할 경우 Mali GPU는 Early-Z 테스트를 실행할 수 없습니다.

## 9.8 더티 렌즈 효과

더티 렌즈 효과를 사용하여 드라마의 느낌을 유발할 수 있습니다. 흔히 렌즈 플레어 효과와 함께 사용됩니다.

더티 렌즈 효과는 모바일 디바이스에 적합한 매우 가볍고 간단한 방식으로 구현할 수 있습니다.

얼음 동굴 데모에서 더티 렌즈 효과는 장면 위에 가변 강도, 전체 스크린 쿼드를 렌더링하는 최소 쉐이더에서 구현됩니다. 쿼드 강도는 스크립트에 의해 전달됩니다.

쉐이더는 모든 투명 지오메트리가 렌더링된 이후 마지막으로 가산식 알파 블렌딩을 사용하여 쿼드를 렌더링합니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *9.8.1 쉐이더 구현* 페이지의 9-193.
- *9.8.2 스크립트 구현* 페이지의 9-195.
- *9.8.3 더티 렌즈 쉐이더 코드* 페이지의 9-195.

### 9.8.1 쉐이더 구현

이 단원에서는 더티 렌즈 쉐이더에 대해 설명합니다.

더티 렌즈 쉐이더의 전체 소스 코드는 *9.8.3 더티 렌즈 쉐이더 코드* 페이지의 9-195를 참조하십시오.

다음의 서브 쉐이더 태그는 전체 화면 쿼드를 모든 불투명 지오메트리 이후, 그리고 다른 9개 투명 객체 이후에 렌더링하도록 명령합니다.

```
Tags {"Queue" = "Transparent+10"}
```

쉐이더는 다음 명령을 사용하여 깊이 버퍼 쓰기를 비활성화합니다. 그러면 쿼드가 그 뒤쪽의 지오메트리를 오클루드하지 못합니다.

```
ZWrite Off
```

블렌딩 단계에서 조각 쉐이더의 출력이 이미 프레임 버퍼 안에 있는 픽셀 색상과 혼합됩니다.

쉐이더는 가산적 블렌딩 유형 `Blend One One`을 지정합니다. 이 블렌딩 유형에서는 소스 및 대상 인수가 모두 `float4 (1.0, 1.0, 1.0, 1.0)`입니다.

이 유형의 블렌딩은 투명하고 빛을 밝히는 불과 같은 효과를 나타내는 파티클 시스템에서 종종 사용됩니다.

쉐이더는 더티 렌즈 효과가 항상 렌더링되도록 컬링 및 ZTest를 비활성화합니다.

쿼드의 정점은 뷰포트 좌표에서 정의됩니다. 따라서 정점 쉐이더에서는 정점 변환이 발생하지 않습니다.

조각 쉐이더는 텍스처에서 텍셀만 가져와 효과의 강도와 비례하는 인수를 적용합니다.

다음 그림은 더티 렌즈 효과를 위해 전체 화면 쿼드 텍스처로 사용되는 이미지입니다.

그림 9-50 얼음 동굴 데모에서 더티 렌즈 효과용으로 사용되는 텍스처

다음 그림은 카메라가 동굴 입구에서 비추는 일광을 향할 때 더티 렌즈 효과가 얼음 동굴 데모에서 어떻게 보이는지 보여줍니다.


그림 9-51 얼음 동굴 데모에서 구현된 더티 렌즈 효과

### 9.8.2 스크립트 구현

간단한 스크립트로 전체 화면 쿼드를 생성하고 조각 쉐이더로 전달되는 강도 인수를 계산합니다.

다음 함수는 시작 함수에서 쿼드의 메시를 생성합니다.

```
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();

    //Increase bounds to avoid frustum clipping.
    bigBounds.SetMinMax(new Vector3(-100, -100, -100), new Vector3(100, 100, 100));
    mesh.bounds = bigBounds;
}
```

메시가 생성되면 절대로 절두체 클립되지 않도록 그 경계가 증가합니다. 장면의 크기에 따라 경계의 크기를 설정합니다.

스크립트는 강도 인수를 계산하여 조각 쉐이더로 전달합니다. 이는 카메라–태양 벡터 및 카메라 포워드 벡터의 상대 방향을 기반으로 합니다. 효과의 최대 강도는 카메라가 태양을 직시할 때 이루어집니다.

다음 코드는 강도 인수 계산입니다.

```
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 더티 렌즈 쉐이더 코드

다음은 DirtyLensEffect.shader의 코드입니다.

```
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);
```

## 9.9 라이트 샤프트

라이트 샤프트는 후광, 대기 산란 또는 음영의 효과를 시뮬레이션합니다. 이들 효과는 장면에 깊이와 사실성을 더하기 위해 사용됩니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.9.1 라이트 샤프트 개요

얼음 동굴 데모에서 라이트 샤프트는 동굴 위쪽 구멍에서 비추는 태양 광선의 산란을 시뮬레이션합니다.

다음 그림은 얼음 동굴 데모의 라이트 샤프트입니다.

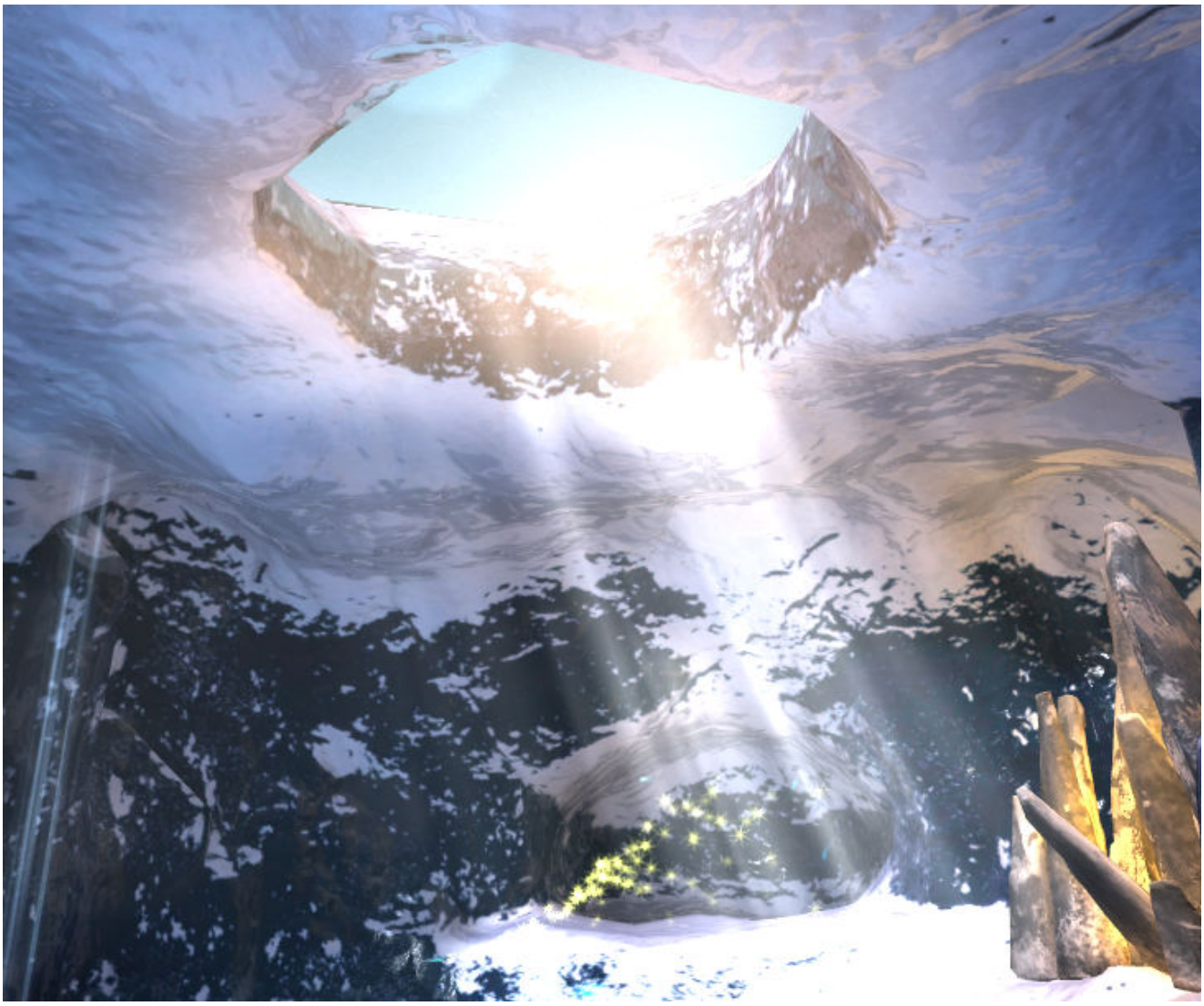

그림 9-52 얼음 동굴 데모의 라이트 샤프트

라이트 샤프트는 원추대를 기반으로 합니다. 원추는 윗부분이 항상 고정된 상태로 광선 방향을 추종합니다.

다음 그림은 원추의 지오메트리입니다.

- a는 라이트 샤프트의 기본 지오메트리가 상단 및 하단에 단면이 2개 있는 원통이라는 것을 보여줍니다.
- b는 하단면이 사용자가 정의하는 각도 $\theta$ 에 따라 확장되어 원추대 지오메트리를 구현하는 것을 보여줍니다.
- c는 상단면은 고정되고 하단면이 태양 광선 방향에 따라 수평으로 변위하는 것을 보여줍니다.

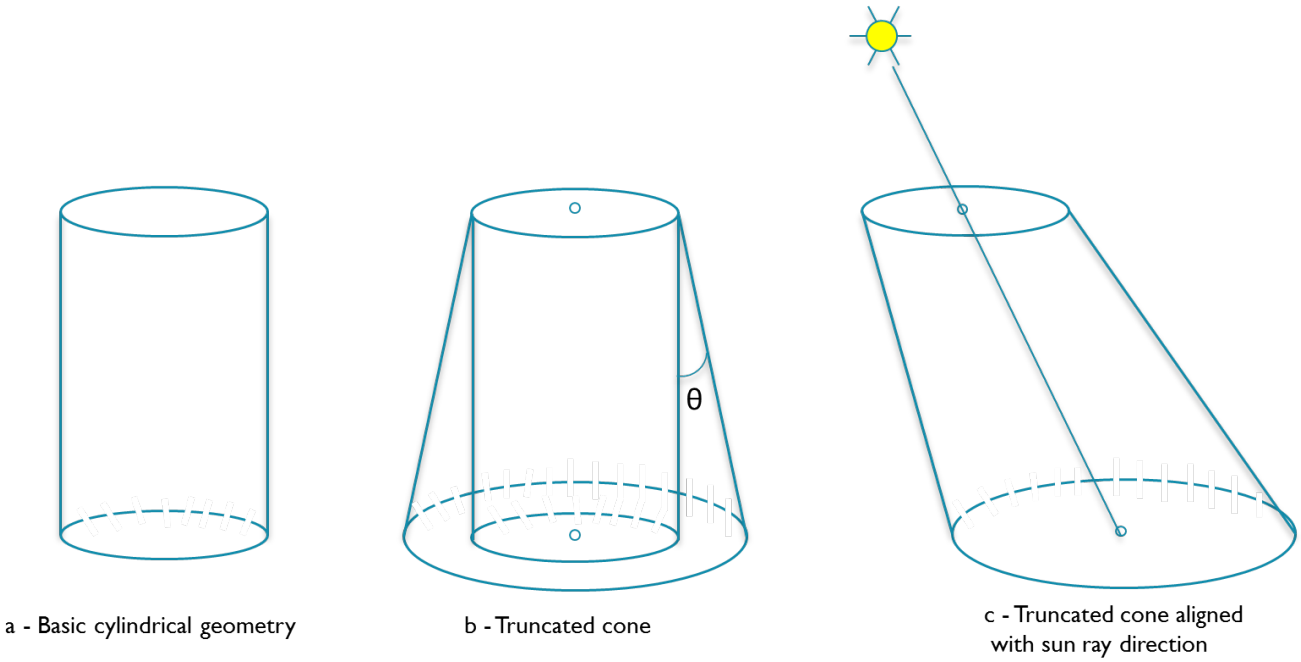


그림 9-53 라이트 샤프트 지오메트리

스크립트는 태양의 위치를 사용하여 다음 값을 계산합니다.

- 입력으로 제공되는 각도 $\theta$ 에 따른 원추 하단면 확장 크기.
- 하단면 변위 방향 및 크기.

정점 쉐이더가 이 데이터를 기반으로 변환을 적용합니다. 변형은 라이트 샤프트의 로컬 좌표로 원래 원통형 지오메트리의 정점에 적용됩니다.

라이트 샤프트를 렌더링할 때 지오메트리를 노출시키는 하드 에지는 렌더링하지 마십시오. 이렇게 하려면 상단 및 하단을 완만하게 페이드 아웃하는 텍스처 마스크를 사용할 수 있습니다.

다음 그림은 라이트 샤프트 텍스처입니다.

그림 9-54 라이트 샤프트 텍스처. 왼쪽: 마스크 텍스처. 오른쪽: 빔 텍스처

### 9.9.2 콘 에지 페이드 아웃

횡단면과 평행한 평면에서 카메라-정점 상대 방향에 따라 라이트 샤프트 강도를 페이드 아웃합니다.

정점 쉐이더에서 카메라 위치를 횡단면에 투영합니다.

횡단면의 중심에서 투영으로 새 벡터를 만들고 정규화합니다.

정점 노말을 사용하여 이 벡터의 내적을 계산한 다음 그 결과를 지수로 올립니다.

결과를 varying 변수로 조각 쉐이더로 전달합니다. 이 값은 라이트 샤프트 강도를 조절하는 데 사용됩니다.

정점 쉐이더는 이러한 계산을 *로컬 좌표계*(LCS)에서 수행합니다.

다음 코드는 정점 쉐이더입니다.

```
// Project camera position onto cross section
float3 axisY = float3(0, 1, 0);
float dotWithYAxis = dot(camPosInLCS, axisY);
float3 projOnCrossSection = camPosInLCS - (axisY * dotWithYAxis);
projOnCrossSection = normalize(projOnCrossSection);

// Dot product to fade the geometry at the edge of the cross section
float dotProd = abs(dot(projOnCrossSection, input.normal));
output.overallIntensity = pow(dotProd, _FadingEdgePower) * _CurrLightShaftIntensity;
```

계수 _FadingEdgePower를 사용하여 라이트 샤프트 에지의 페이딩을 미세하게 조정할 수 있습니다.

이 스트립트는 계수 _CurrLightShaftIntensity를 전달합니다. 그러면 카메라가 라이트 샤프트에 접근할 때 라이트 샤프트가 페이드 아웃됩니다.

다음 코드는 스크립트에서 텍스처를 천천히 아래로 스크롤하여 라이트 샤프트에 추가되는 마지막 터치를 보여줍니다.

```
void Update()
{
    float localOffset = (Time.time * speed ) + offset;
    localOffset = localOffset % 1.0f;
    GetComponent<Renderer>().material.SetTextureOffset("_MainTex", new Vector2(0,
```

```
localOffset));
}
```

조각 쉐이더는 빔 및 마스크 텍스처를 가져온 다음 강도 인수와 결합합니다.

```
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;
}
```

라이트 샤프트 지오메트리는 모든 불투명 지오메트리 다음에 투명 큐에서 렌더링됩니다.

쉐이더는 가산적 블렌딩을 사용하여 조각 색상을 프레임 버퍼의 대응 픽셀과 결합합니다.

또한 쉐이더는 다른 객체가 오클루드되지 않도록 깊이 버퍼 쓰기 및 컬링을 비활성화합니다.

패스 설정은 다음과 같습니다.

```
Blend One One
Cull Off
ZWrite Off
```

## 9.10 안개 효과

안개 효과는 장면에 분위기를 더합니다. 안개를 생성하기 위해 고급 구현이 필요하지는 않습니다. 단순한 안개 효과만으로도 효과적일 수 있습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.10.1 안개 효과 개요

실제 세계에서는 멀리 바라볼수록 색이 희미해집니다. 안개가 끼어야만 이 효과가 보이는 것이 아니라 맑은 날에도, 특히 산을 바라볼 때 잘 보입니다.

이 효과는 실제 세계에서 매우 흔하므로 게임에 이 효과를 추가하면 사실성이 향상됩니다.

이 단원에서는 안개 효과의 두 버전을 설명합니다.

- 절차적 선형 안개.
- 파티클 기반 안개.

두 효과를 동시에 적용할 수 있습니다. 얼음 동굴 데모에서는 두 기법을 모두 사용합니다.

### 9.10.2 절차적 선형 안개

객체가 멀어질수록 객체 색상이 정의된 안개 색상으로 페이드되도록 합니다. 조각 쉐이더에서 이 효과를 구현하려면 조각 색상과 안개 색상 사이에 간단한 선형 보간을 사용할 수 있습니다.

다음 예제 코드는 카메라와의 거리를 기반으로 정점 쉐이더에서 안개 색상을 계산하는 방법을 보여줍니다. 이 색상은 varying 변수로 조각 쉐이더로 전달됩니다.

```
output.fogColor = _FogColor * clamp(vertexDistance * _FogDistanceScale, 0.0, 1.0);
```

인수 설명:

- `vertexDistance`는 정점과 카메라 간 거리입니다.
- `_FogDistanceScale`는 uniform 변수로 쉐이더로 전달되는 인수입니다.
- `_FogColor`는 사용자가 정의하는 기본 안개 색상이며 uniform 변수로 전달됩니다.

조각 쉐이더에서 보간된 `input.fogColor`가 조각 색상 `output.Color`와 결합됩니다.

```
outputColor = lerp(outputColor, input.fogColor.rgb, input.fogColor.a);
```

다음 그림은 장면에서 구현되는 결과입니다.

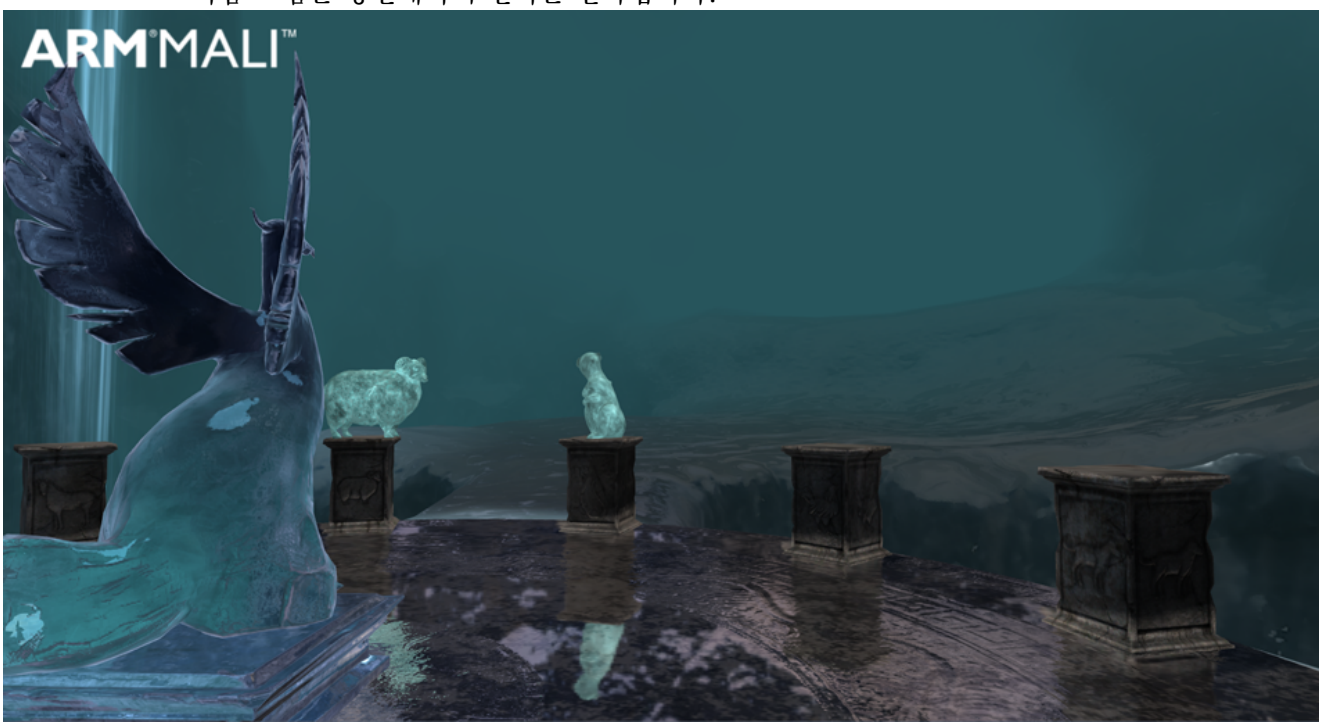


그림 9-55 거리 기반 선형 안개

참고

쉐이더에서 안개를 계산한다는 것은 안개 효과를 생성하기 위해 추가로 어떤 후처리도 수행할 필요가 없다는 의미입니다.

가능한 한 많은 효과를 한 쉐이더로 병합하도록 노력하십시오. 하지만 캐시를 초과할 경우 성능이 저하될 수 있으므로 성능을 확인해야 합니다. 쉐이더를 2개 이상의 패스로 분할해야 할 수도 있습니다.

안개 색상 계산은 정점 또는 조각 쉐이더에서 가능합니다. 조각 쉐이더에서의 계산이 더 정확하지만 모든 조각에 대해 계산되므로 더 높은 컴퓨팅 성능을 요구합니다.

정점 쉐이더에서의 계산은 정밀도가 더 낮지만 정점에 대해 한 번만 계산되므로 성능도 더 높습니다.

### 9.10.3 높이 포함 선형 안개

안개가 장면에 걸쳐 균일하게 적용됩니다. 밀도를 높이에 따라 변경하여 보다 사실적으로 만들 수 있습니다.

안개를 아래로 갈수록 진하게, 위로 갈수록 옅게 만드십시오. 수동으로 조정할 수 있도록 높이를 노출시킬 수 있습니다.

다음 그림은 거리 및 높이를 기반으로 한 선형 안개입니다.


그림 9-56 거리 및 높이 기반 선형 안개

### 9.10.4 비균일 안개

안개가 균일할 필요는 없습니다. 약간의 노이즈를 섞어 안개를 시각적으로 보다 흥미롭게 만들 수 있습니다.

노이즈 텍스처를 적용하여 비균일 안개를 생성할 수 있습니다.

보다 복잡한 효과를 위해 여러 노이즈 텍스처를 적용하고 서로 다른 속도로 미끄러지게 할 수도 있습니다. 예를 들어 먼 거리에 위치한 노이즈 텍스처가 카메라에서 가까운 텍스처보다 느리게 미끄러집니다.

한 쉐이더에서 단일 패스에 복수의 텍스처를 적용하고 거리에 따라 노이즈 텍스처를 구부릴 수 있습니다.

### 9.10.5 프리베이크된 안개

카메라가 가까이 다가가지 않을 것을 아는 경우 안개를 텍스처로 프리베이크할 수 있습니다. 그러면 필요한 컴퓨팅 파워가 감소합니다.

### 9.10.6 파티클을 사용한 입체적 안개

파티클로 입체적 안개를 시뮬레이션할 수 있습니다. 그러면 고품질 결과를 얻을 수 있습니다.

파티클 수를 최소한으로 유지하십시오. 파티클은 오버드로를 증가시킵니다. 조각당 더 많은 쉐이더가 실행되기 때문입니다. 더 많은 파티클을 생성하는 대신 더 큰 파티클을 사용해 보십시오.

얼음 동굴 데모에서 최대 15개의 파티클이 동시에 사용됩니다.

#### 빌보드 대신 지오메트리 렌더링

파티클 시스템의 렌더링 모드를 빌보드 대신 메시로 설정합니다. 이는 입체적 효과를 얻으려면 개별 파티클이 무작위 회전해야 하기 때문입니다.

다음 그림은 텍스처가 없는 파티클 지오메트리입니다.

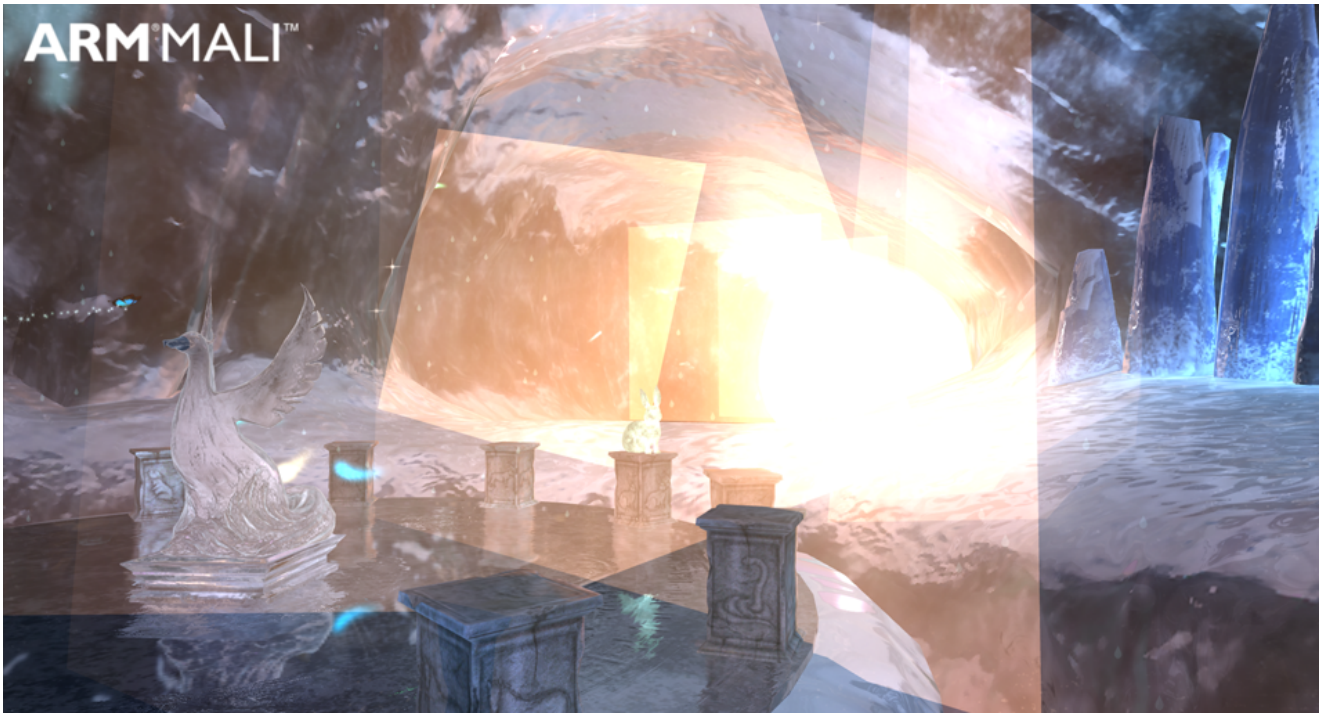


그림 9-57 텍스처 없는 파티클

다음 그림은 텍스처가 있는 파티클 지오메트리입니다.

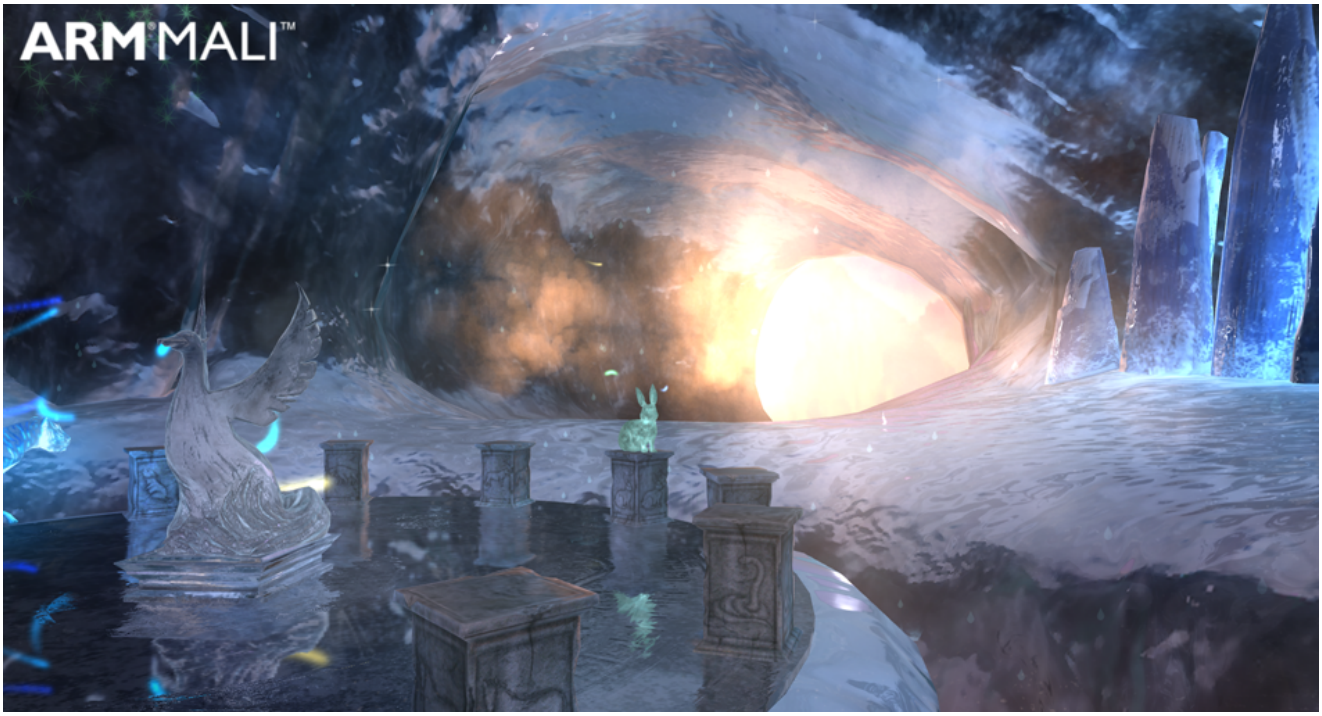


그림 9-58 텍스처 있는 파티클

다음 그림은 각 파티클에 적용된 텍스처입니다.

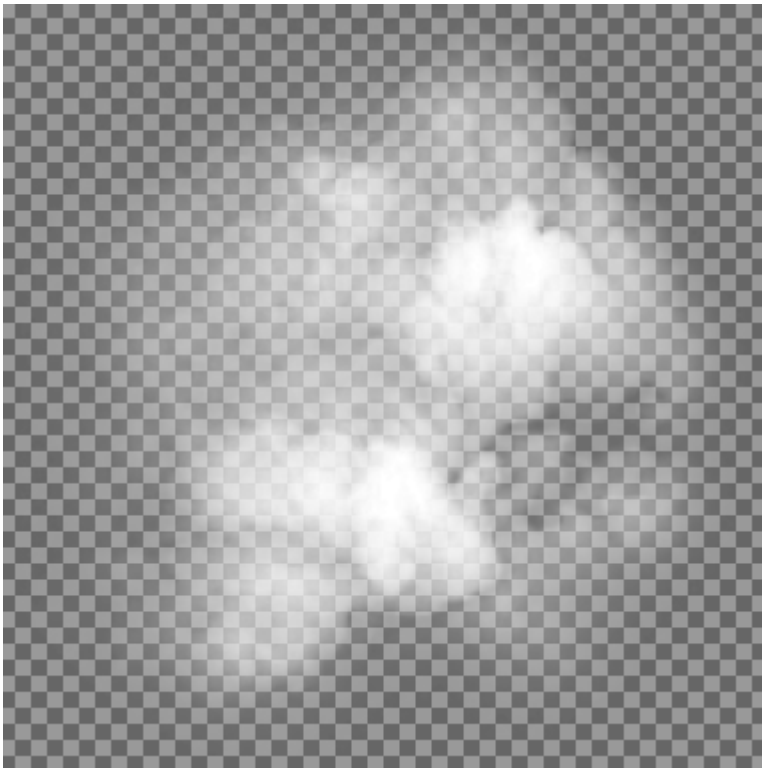

그림 9-59 개별 파티클의 텍스처

### 앵글 페이드 효과

개별 파티클을 카메라 위치를 기준으로 한 방향에 따라 페이드 인 및 아웃합니다. 파티클을 페이드 인 및 아웃하지 않을 경우 파티클의 날카로운 에지가 보입니다.

다음 예제 코드는 페이딩을 수행하는 정점 쉐이더입니다.

```
 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;  // instead of power of 2
```

varying 변수 `output.visibility`는 파티클 폴리곤에서 보간됩니다. 조각 쉐이더에서 이 값을 읽고 투명도를 적용합니다.

다음 코드는 그 방법을 보여줍니다.

```
half4 diffuseTex = _Color * tex2D(_MainTex, half2(input.texCoord));
diffuseTex *= input.visibility;
return diffuseTex;
```

### 파티클 렌더링

파티클을 렌더링하려면 다음 절차를 사용합니다.

1. 파티클을 프레임 내부의 마지막 프리미티브로 렌더링합니다.

   ```
   Tags { "Queue" = "Transparent+10" }
   ```

   +10은 얼음 동굴 데모가 파티클 이전에 9개의 다른 투명한 객체를 렌더링하기 때문에 이 예제에 포함된 것입니다.
2. 적절한 블렌딩 모드를 설정합니다.

   쉐이더 패스의 시작 부분에 다음 행을 추가합니다.

   ```
   Blend SrcAlpha One
   ```
3. z 버퍼에 대한 쓰기를 비활성화합니다.

다음 행을 추가합니다.

```
ZWrite Off
```

### 파티클 시스템 설정

다음 그림은 얼음 동굴 데모에서 사용되는 파티클 시스템 설정입니다.

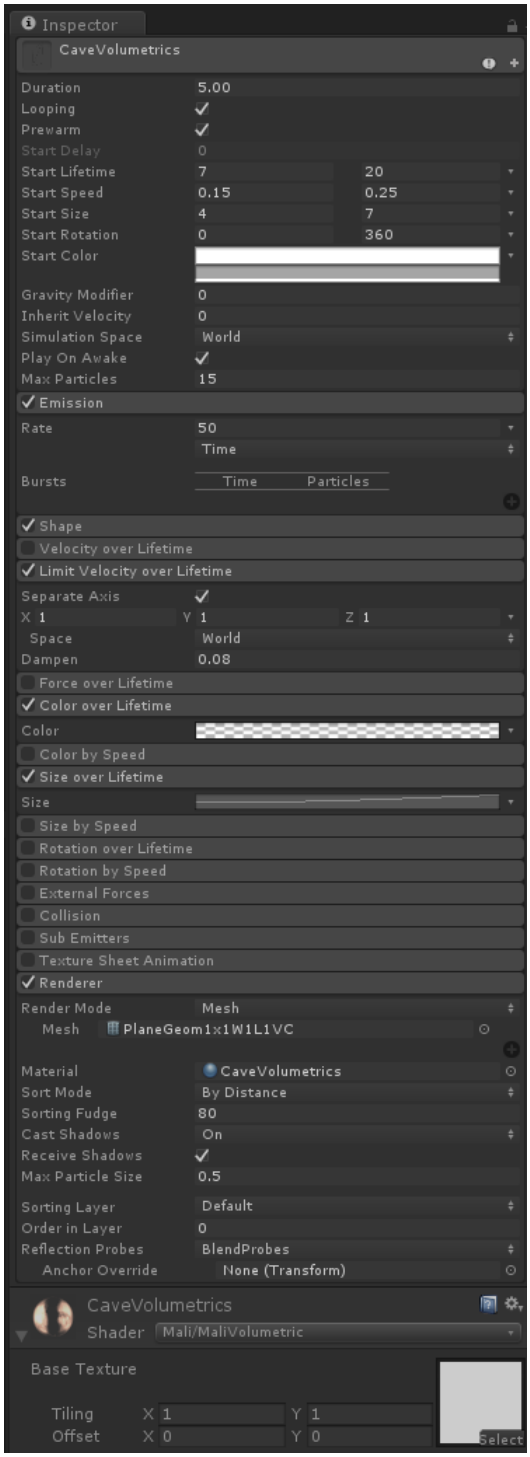


그림 9-60 얼음 동굴 데모의 파티클 시스템 매개 변수

다음 그림은 파티클이 스폰되는 영역을 보여줍니다. 이 영역은 이미지 안의 박스로 표시됩니다. 박스는 Unity Particle System의 내장 옵션 Shape로 정의됩니다.


그림 9-61 파티클이 스폰되는 파티클 박스 정의

## 9.11 블룸

블룸은 밝은 환경에서 사진을 촬영할 때 실제 카메라에서 발생하는 효과를 재현합니다. 블룸 효과는 밝은 영역의 경계에서 뻗어 나오는 빛의 테두리를 시뮬레이션합니다.

다음 그림은 얼음 동굴 데모에서 동굴 입구에서의 블룸을 보여줍니다.

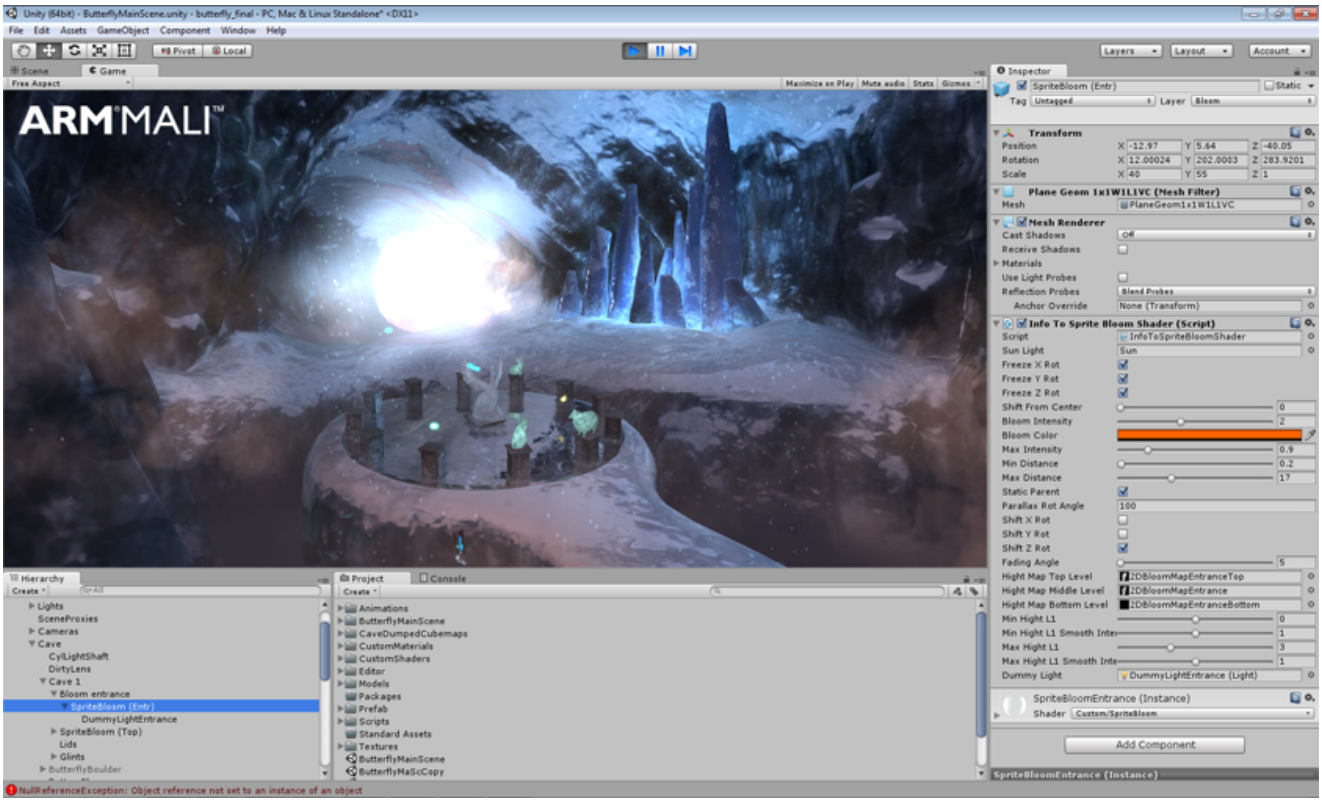


그림 9-62 얼음 동굴 데모에서 구현된 동굴 입구에서의 블룸

실제 렌즈에서 블룸 효과는 완벽하게 초점을 맞출 수 없기 때문에 발생합니다. 빛이 카메라의 조리개를 통과할 때 빛 일부가 회절되어 이미지 주위로 밝은 고리를 만듭니다. 일반적으로 이 효과는 대부분의 조건에서 잘 보이지 않지만 강한 조명 아래에서는 보입니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *9.11.1 블룸 구현* 페이지의 9-207.
- *9.11.2 블룸 조절 인수 스크립트 만들기* 페이지의 9-208.
- *9.11.3 정점 쉐이더* 페이지의 9-208.
- *9.11.4 조각 쉐이더* 페이지의 9-208.
- *9.11.5 블룸 효과 수정* 페이지의 9-209.
- *9.11.6 오클루전 맵 간 보간* 페이지의 9-212.

### 9.11.1 블룸 구현

블룸 효과는 일반적으로 후처리 효과로 구현됩니다. 이러한 방식으로 효과를 생성할 경우 대용량의 컴퓨팅 파워를 사용하게 되어 모바일 게임에서는 적합하지 않을 수 있습니다. 게임이 얼음 동굴 데모와 같이 복잡한 효과를 많이 사용할 경우 특히 그렇습니다.

한 가지 대안에서는 단순한 평면을 사용합니다. 카메라와 광원 사이에 평면을 배치하는 것입니다. 이 평면은 노말이 카메라가 잡아야 할 장면의 일부를 가리키도록 방향을 설정해야 합니다.

이러한 방식으로 블룸 효과를 생성하려면 평면을 블룸 효과가 발생하기를 원하는 위치에 배치합니다. 평면 노말 및 광원과 함께 뷰 벡터의 정렬에 기반한 인수를 사용하여 효과의 강도를 조절합니다. 이 방법은 얼음 동굴 데모에 구현되어 있습니다.

다음 그림은 평면을 사용하여 구현된 블룸 효과입니다.

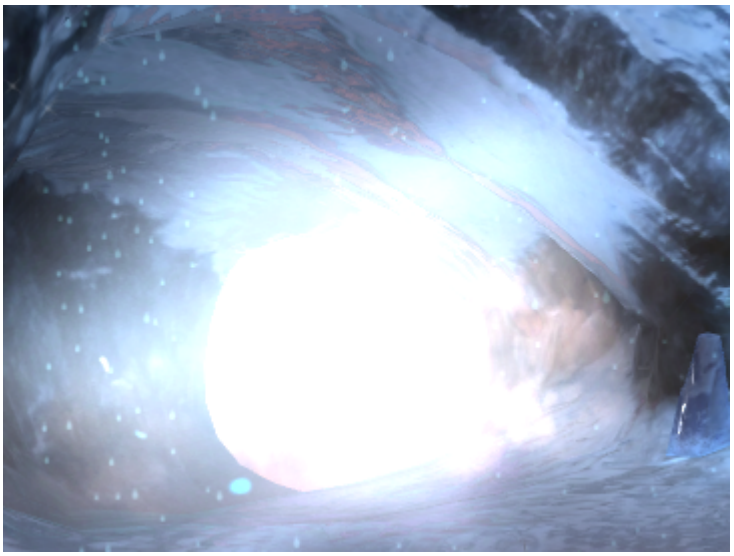

그림 9-63 동굴 입구의 평면이 블룸을 구현

### 9.11.2 블룸 조절 인수 스크립트 만들기

정렬 인수는 스크립트를 사용하여 계산할 수 있습니다. 이 인수는 카메라가 효과를 촬영하는 각도에 따라 블룸 효과를 변화시킵니다.

예를 들어 다음 스크립트는 스크립트가 연결된 평면의 정렬 인수를 계산합니다.

```
// Light-plane
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);
```

정렬 인수 `sunLightPlaneCameraAlignment`가 쉐이더로 전달되어 렌더링된 색상의 강도를 조절합니다.

### 9.11.3 정점 쉐이더

정점 쉐이더는 `sunLightPlainCameraAlignment` 인수를 수신하여 렌더링된 블룸 색상의 강도를 조절하는 데 사용합니다.

정점 쉐이더는 MVP 매트릭스를 적용하고, 텍스처 좌표를 조각 쉐이더로 전달하고, 정점 좌표를 출력합니다.

다음 코드는 이 프로세스가 어떻게 구현되는지 보여줍니다.

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;
    output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    return output;
}
```

### 9.11.4 조각 쉐이더

조각 쉐이더는 uniform 변수로 전달되는 `CurrBloomColor` 틴트 색상을 사용하여 텍스처 색상 및 증분을 가져옵니다. 조각 인수는 색상을 사용되기 전에 조절합니다.

다음 코드는 이 프로세스가 어떻게 수행되는지 보여줍니다.

```
half4 frag(vertexOutput input) : COLOR
{
    half4 textureColor = tex2D(_MainTex, input.tex.xy);
    textureColor += textureColor * _CurrBloomColor;
    return textureColor * _AligmentFactor;
}
```

블룸 평면은 모든 불투명 지오메트리 다음에 투명 큐에서 렌더링됩니다. 쉐이더에서는 다음 큐 태그를 사용하여 렌더링 순서를 설정합니다.

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

블룸 평면이 가산적 1:1 블렌딩을 사용하여 적용되어 조각 색상을 프레임 버퍼에 이미 저장된 대응 픽셀과 결합합니다. 또한 쉐이더는 기존 개체가 오클루드되지 않도록 `ZWrite Off` 명령어를 사용하여 깊이 버퍼에 대한 쓰기를 비활성화합니다.

다음 그림은 얼음 동굴 데모가 블룸 효과를 위해 사용하는 텍스처와 그 결과를 보여줍니다.

그림 9-64 불투명 주추 뒤의 오클루딩 영역으로 들어가는 카메라를 보여주는 시퀀스

### 9.11.5 블룸 효과 수정

평면을 사용하여 블룸 효과를 생성하는 것은 간단하고 모바일 디바이스에 적합합니다. 하지만 불투명한 객체가 블룸을 생성하는 광원과 카메라 사이에 위치할 경우 기대한 결과가 얻어지지 않습니다. 오클루전 맵을 사용하여 이를 수정할 수 있습니다.

불투명한 객체 뒤에서 블룸 효과가 잘못 나타나는 것은 효과가 투명 큐에서 렌더링되기 때문입니다. 그러므로 블렌딩이 블룸 평면 앞의 객체 위에서 이루어집니다.

다음 그림은 수정 없이 생성된 잘못된 블룸 효과의 예입니다.

그림 9-65 불투명 주추 위에서 표시되는 잘못된 블룸 효과

이러한 오류를 방지하기 위해 정렬 인수를 계산하는 스크립트를 추가 오클루전 맵을 처리하도록 변경할 수 있습니다. 이 오클루전 맵은 카메라가 일반적으로 블룸 효과를 잘못 적용하는 영역을 설명합니다. 이러한 충돌 영역은 오클루전 맵 값이 0으로 지정됩니다. 이 인수는 정렬 인수와 결합되어 이러한 장소에서 인수 강도를 0으로 설정합니다. 이 변경은 블룸이 더 이상 블룸을 오클루드해야 하는 객체에서 렌더링되지 않도록 블룸을 0으로 설정합니다. 다음 코드는 이러한 정렬을 구현합니다.

```
IntensityFactor = alignmentFactor * occlusionMapFactor
```

얼음 동굴 데모 카메라는 3차원을 자유롭게 이동할 수 있으므로 각각 다른 높이를 커버하는 3개의 그레이스케일 맵이 사용됩니다. 검은색은 0을 의미합니다. 따라서 오클루전 맵 인수에 정렬 인수를 곱하면 최종 강도가 0이 되어 블룸을 오클루드합니다. 흰색은 1을 의미합니다. 따라서 강도가 정렬 인수와 동일하고 블룸이 오클루드되지 않습니다.

다음 그림은 지면 레벨의 `occlusionMapFactor`를 보여줍니다.

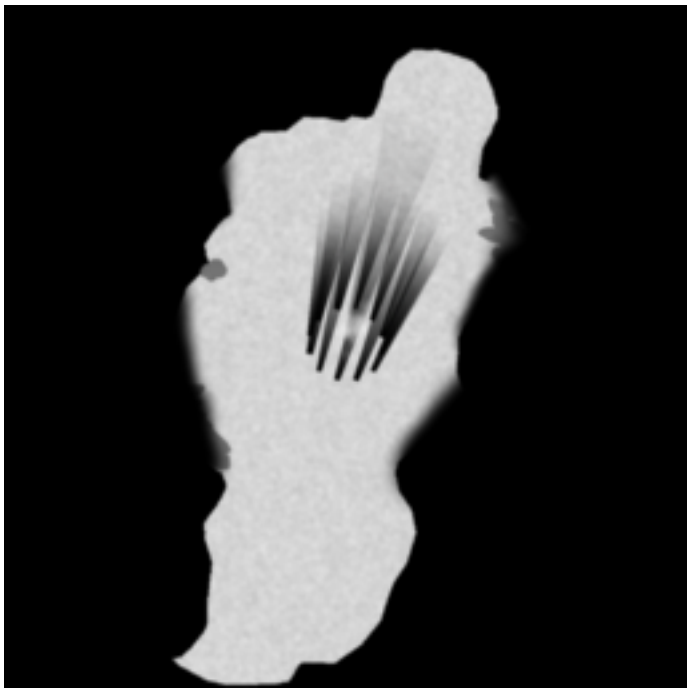

그림 9-66 지면 레벨 블룸 오클루전 맵

———— 참고 ————

맵의 중앙에서 검은색으로 빛나는 부분은 동굴의 입구에서 블룸 효과를 오클루드하는 불투명 주추 뒤쪽 영역입니다.

————————

지상 레벨 $H_{max}$ 높이에서는 오클루딩 객체가 없습니다. 이는 지상 레벨 블룸 오클루전 맵이 흰색이라는 의미입니다. 다음 그림은 지상 레벨의 `occlusionMapFactor`를 보여줍니다.

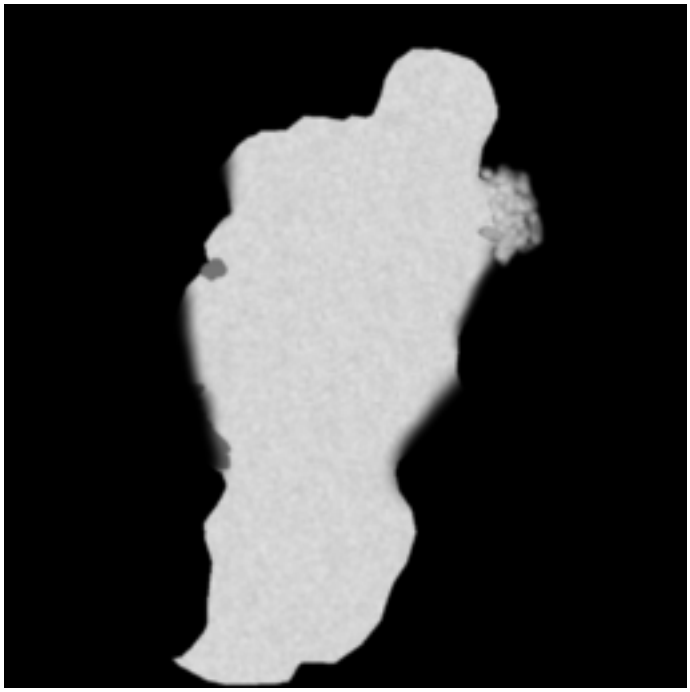

그림 9-67 지상 레벨 블룸 오클루전 맵

$H_{min}$ 높이 아래의 지하 레벨에서는 블룸 효과가 완전히 오클루드됩니다. 이는 지하 레벨 블룸 오클루전 맵이 완전히 검은색이라는 의미입니다. 맵이 보이도록 흰색 윤곽이 추가됩니다. 다음 그림은 지하 레벨의 `occlusionMapFactor`를 보여줍니다.

그림 9-68 지하 레벨 블룸 오클루전 맵

### 9.11.6 오클루전 맵 간 보간

얼음 동굴 데모에서 스크립트는 프레임마다 2D 케이브 맵 위 카메라 위치의 XZ 프로젝션을 계산하여 결과를 0과 1 사이로 정규화합니다.

카메라 높이가 $H_{min}$보다 낮거나 $H_{max}$보다 높을 경우 정규화된 좌표를 사용하여 단일 맵에서 색상 값을 가져옵니다. 카메라 높이가 $H_{min}$과 $H_{max}$ 사이일 경우 색상은 양쪽 맵에서 가져와 보간됩니다.

이 보간을 통해 다양한 높이의 효과 사이에서 부드러운 전환이 이루어집니다.

얼음 동굴 데모는 `GetPixelBilinear()` 함수를 사용하여 맵(들)로부터 색상을 가져옵니다. 이 함수는 다음 코드를 사용하여 필터 색상 값을 반환합니다.

```
float groundOcclusionFactor = groundOcclusionMap.GetPixelBilinear(camPosXZNormalized.x,
camPosXZNormalized.y)).r;
```

블룸 오클루전 맵을 사용하면 불투명한 오클루딩 객체 위에서는 블렌딩이 이루어지지 않습니다. 다음 그림은 그 효과를 보여줍니다. 카메라가 들어가고 나갈 때 오클루드하는 검은색이 불투명 주추 뒤에서 잘못된 블룸을 방지합니다.

다음 그림은 불투명 주추 뒤에서 오클루딩 영역으로 들어가는 카메라를 보여주는 시퀀스입니다.

그림 9-69 불투명 주추 뒤에서 오클루딩 영역으로 들어감

## 9.12 빙벽 효과

얼음 동굴 데모는 동굴의 빙벽에서 섬세한 반사 효과를 사용합니다. 이 효과는 섬세하지만 장면에 추가적인 사실성과 분위기를 더합니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.12.1 빙벽 효과 개요

얼음은 빛이 표면의 작은 디테일에 따라 다양한 방식으로 산란하기 때문에 재현하기 어려운 자료일 수 있습니다. 반사는 완전히 명확하거나, 완전히 왜곡되거나, 또는 그 중간 어디쯤일 수 있습니다. 얼음 동굴 데모는 이 효과를 표시하며 사실성을 높이기 위해 패럴랙스 효과를 포함합니다.

다음 그림은 얼음 동굴 데모에서 이 효과가 구현된 모습입니다.


그림 9-70 얼음 동굴 데모

다음 그림은 이 효과의 클로즈업입니다.

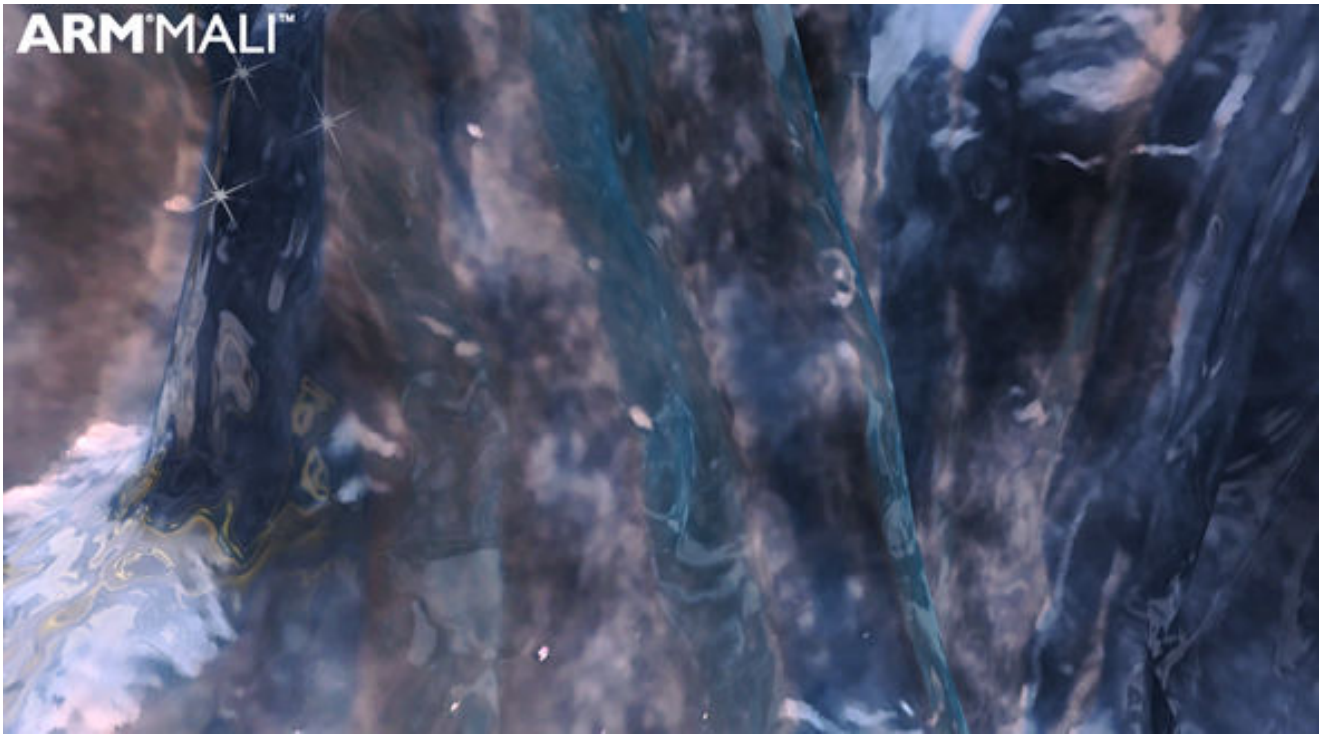


그림 9-71 반사성 빙벽의 클로즈업

### 9.12.2 반사에 영향을 주기 위해 노말 맵 수정 및 결합

얼음 동굴 데모의 반사 효과는 탄젠트 공간 노말 맵과 계산된 그레이스케일 페이크 노말 맵을 사용합니다. 이들 두 맵을 일부 맵과 결합하면 데모에서의 효과가 생성됩니다.

그레이스케일 페이크 노말 맵은 탄젠트 공간 노말 맵의 그레이스케일입니다. 얼음 동굴 데모에서 그레이스케일 맵의 값은 대부분 0.3-0.8 범위입니다. 다름 그림은 얼음 동굴 데모에서 사용되는 탄젠트 공간 노말 맵과 그레이스케일 페이크 노말 맵입니다.

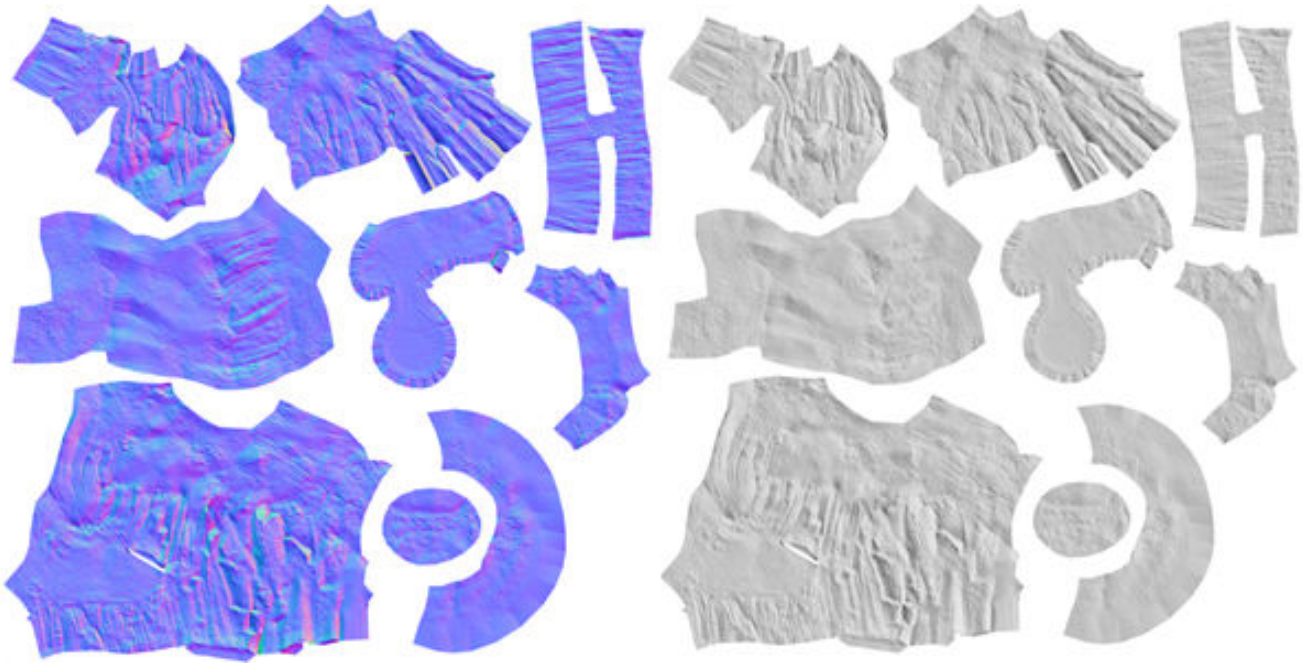

그림 9-72 탄젠트 공간 노말 맵 및 그레이스케일 페이크 노말 맵

### 탄젠트 공간 노말 맵을 사용하는 이유

탄젠트 공간 노말 맵은 결과 그레이스케일의 값 범위가 작기 때문에 얼음 동굴 데모에서 사용됩니다. 이는 탄젠트 공간 노말 맵이 이 렌더링 프로세스의 후반 단계에서 잘 작용한다는 것을 의미합니다.

한 가지 대안은 객체 공간 노말 맵을 사용하는 것입니다. 이들 맵은 탄젠트 공간 노말 맵과 동일한 디테일을 보여주지만 빛이 닿는 부분도 보여줍니다. 그러므로 결과 객체 공간 노말 맵 그레이스케일의 값 범위가 너무 커 렌더링 프로세스의 후반 단계에서 제대로 작동하지 않습니다. 다음 그림은 이 효과를 보여줍니다.

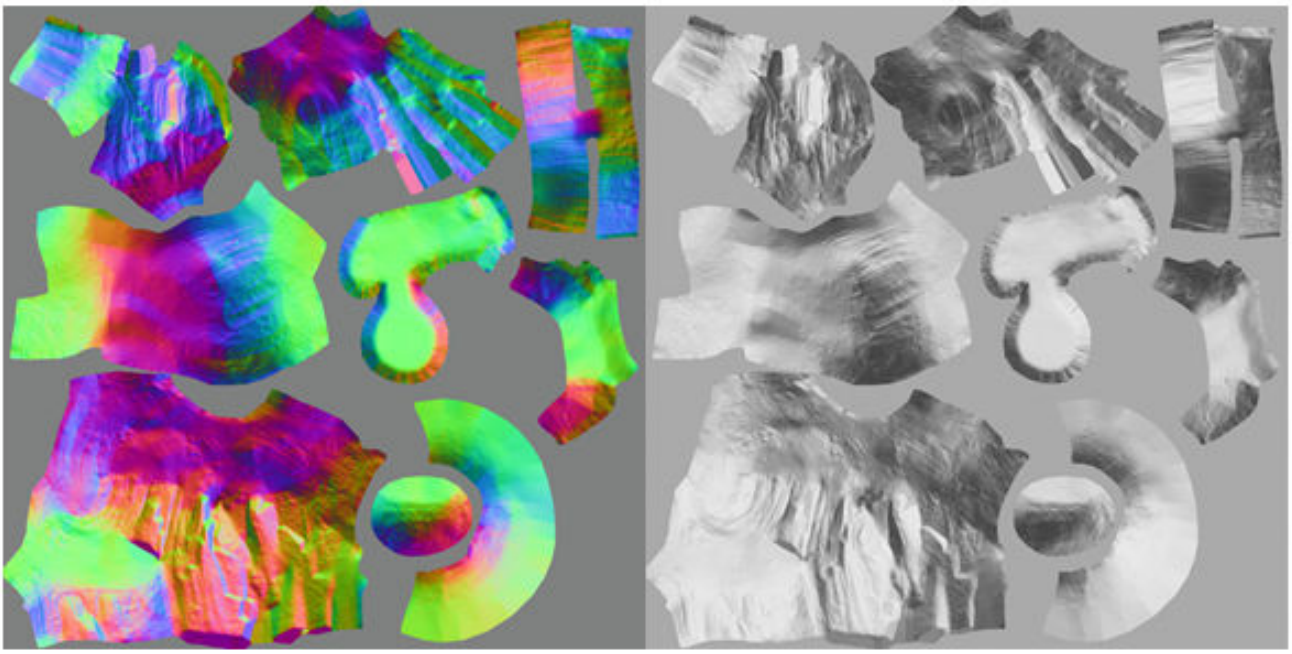

그림 9-73 객체 공간 노말 맵 및 그레이스케일 효과

### 노말 맵의 일부에 투명성 적용

그레이스케일 페이크 노말 맵은 눈이 없는 영역에만 적용됩니다. 이들 맵이 눈 덮인 영역에 적용되지 않도록 그레이스케일 페이크 노말 맵이 동일한 표면에 대해 디퓨즈 텍스처 맵을 사용하도록 수정됩니다. 이 수정을 통해 텍스처 맵에서 눈이 나타나는 위치에서 그레이스케일 페이크 노말 맵의 알파 구성요소가 증가합니다.

다음 그림은 얼음 동굴 데모 정적 표면을 위한 디퓨즈 텍스처 맵과 이러한 맵이 그레이스케일 페이크 노말 맵에 적용되었을 때의 결과를 보여줍니다.

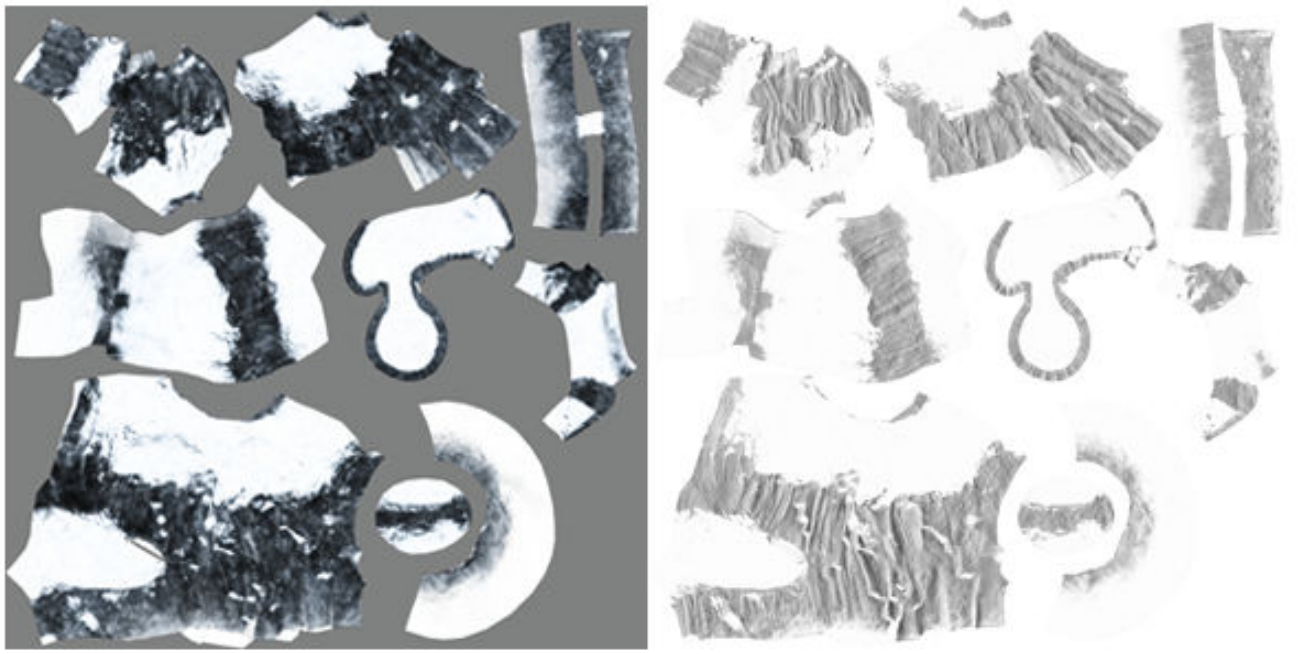

그림 9-74 투명 영역을 포함하는 디퓨즈 텍스처 및 최종 페이크 노말 맵

### 9.12.3 다양한 노말 맵에서 반사 생성

투명도 조정 그레이스케일 페이크 노말 맵과 트루 노말 맵을 조합하면 얼음 동굴 데모에서 보여지는 반사 효과가 생성됩니다.

투명도 조정 그레이스케일 페이크 노말 맵(bumpFake)과 트루 노말 맵(bumpNorm)은 비례적으로 결합됩니다. 결합에는 다음 함수가 사용됩니다.

```
half4 bumpNormalFake = lerp(bumpNorm, bumpFake, amountFakeNormalMap);
```

이 코드는 동굴의 더 어두운 부분에서는 그레이스케일 페이크 노말이 반사에 주로 기여함을 의미합니다. 동굴의 눈 덮인 부분에서는 효과가 객체 공간 노말에서 나옵니다.

그레이스케일 페이크 노말 맵을 사용하여 효과를 적용하려면 그레이스케일을 노말 벡터로 변환해야 합니다. 이 프로세스를 시작하기 위해 노말 벡터의 세 구성요소가 그레이스케일 값과 동일하게 설정됩니다. 얼음 동굴 데모에서 이는 구성요소 벡터가 (0.3, 0.3, 0.3) ~ (0.8, 0.8, 0.8) 사이임을 의미합니다. 모든 구성요소가 동일하도록 설정되므로 모든 노말 벡터가 동일한 방향을 가리킵니다.

쉐이더는 노말 구성요소에 변환을 적용합니다. 0~1 범위의 값을 −1~1 범위의 값으로 변경하는 변환이 일반적으로 사용됩니다. 이 변환을 수행하는 방정식은 다음과 같습니다.
 = 2 * value − 1. 이 방정식은 노말 벡터를 이전에 향하던 방향 또는 반대 방향을 가리키도록 변경합니다. 예를 들어 원래 노말의 구성요소가 (0.3, 0.3, 0.3)일 경우 결과 노말은 (−0.4, −0.4, −0.4)입니다. 원래 노말의 구성요소가 (0.8, 0.8, 0.8)일 경우 결과 노말은 (0.6, 0.6, 0.6)입니다.

−1~1 범위로 변환 후 벡터는 reflect() 함수에 입력됩니다. 이 함수는 정규화된 노말 벡터를 사용하도록 설계되었지만, 이 경우에는 비정규화된 노말이 함수로 전달됩니다. 다음 코드는 쉐이더 내장 함수 reflect()가 어떻게 작용하는지 보여줍니다.

```
R = reflect(I, N) = I - 2 * dot(I, N) * N
```

이 함수를 길이가 1 미만인 비정규화된 입력 노말과 함께 사용하면 반사 벡터가 반사 법칙에 따라 노말에서 예상되는 것보다 커집니다.

노말 벡터의 구성요소 값이 0.5 미만일 경우 반사 벡터는 반대 방향으로 전환됩니다. 그러면 큐브맵의 다른 부분이 판독됩니다. 이처럼 큐브맵에서 부분 간 전환이 발생하면 큐브맵의 암석

부분의 반사 옆에 있는 큐브맵의 흰색 부분을 반영하는 불균일한 스폿의 효과가 생성됩니다. 그레이스케일 페이크 노말 맵에서 양수 노말과 음수 노말 간 전환을 초래한 영역은 반사된 벡터에서 가장 왜곡된 각도를 생성하는 영역이기도 하므로 이는 흥미로운 스월 효과를 만듭니다. 다음 그림은 이 효과를 보여줍니다.

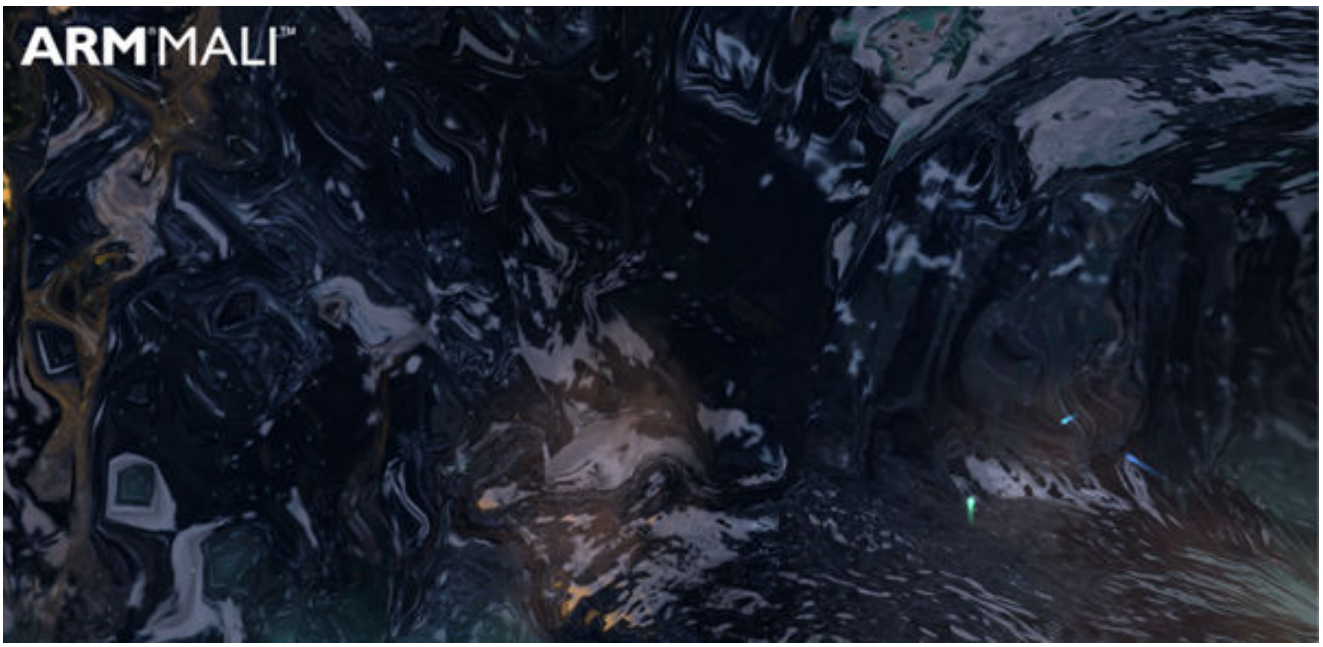


그림 9-75 얼음 반사 내 스월 효과

쉐이더가 비정규화된 클램프 스테이지 없이 페이크 노말 값을 사용하도록 프로그래밍된 경우 결과는 대각선 밴드입니다. 이 차이는 상당하며 이러한 스테이지가 중요함을 나타냅니다. 다음 그림은 클램프 스테이지 없이 구현되는 효과입니다.

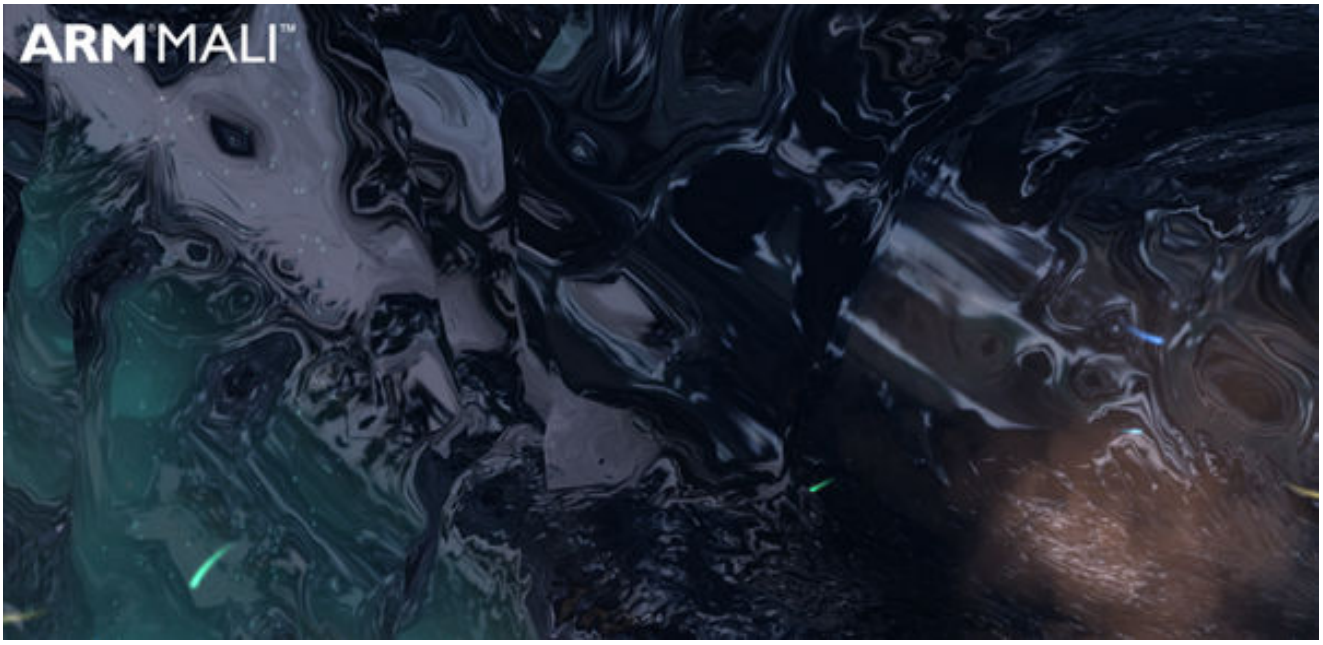


그림 9-76 클램프 스테이지 없는 얼음 반사

### 9.12.4 반사에 로컬 수정 적용

반사 벡터에 로컬 수정을 적용하면 효과의 사실성이 개선됩니다. 수정이 없을 경우, 실제 반사와 마찬가지로 카메라의 위치에 따라 바뀌지 않습니다. 카메라가 횡방향으로 이동하는 경우 특히 그렇습니다.

*관련 정보*

*9.2.2 로컬 큐브맵을 사용하여 올바른 반사를 생성* 페이지의 9-155

## 9.13 절차적 스카이박스

얼음 동굴 데모는 일중 시간 시스템을 사용하여 로컬 큐브맵을 사용하여 구현할 수 있는 동적 그림자 효과를 표시합니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.13.1 절차적 스카이박스 개요

동적 일중 시간 효과를 구현하기 위해 다음 요소가 결합됩니다.

- 절차적으로 생성된 태양.
- 주야 주기를 표현하는 일련의 페이드하는 스카이박스 배경 큐브맵.
- 스카이박스 구름 큐브맵.

절차적 태양 및 분리된 구름 텍스처도 낮은 연산 비용으로 표면하 산란 효과를 생성합니다.

다음 그림은 스카이박스의 모습입니다.

그림 9-77 얼음 동굴 데모에서 표면하 산란을 적용한 태양 렌더링

얼음 동굴 데모는 뷰 방향을 사용하여 큐브맵으로부터 샘플링합니다. 그러므로 데모가 스카이박스의 반구를 렌더링하지 않을 수 있습니다. 대신, 동굴의 구멍 가까이에서 평면을 사용합니다.

이렇게 하면 다른 지오메트리에 의해 대부분 오클루드된 반구를 렌더링하는 것과 비교해 성능이 향상됩니다.

다음 그림은 평면을 사용하여 렌더링된 스카이박스입니다.


그림 9-78 평면을 사용하여 렌더링된 스카이박스

### 9.13.2 일중 시간 관리

이 효과는 C# 스크립트를 사용하여 일중 시간에 대한 계산과 주야 주기의 애니메이션을 관리합니다. 그런 다음 쉐이더가 태양과 스카이맵을 결합합니다.

스크립트에서 다음 값을 지정해야 합니다.

- 스카이박스 배경 페이딩 단계 수.
- 주야 주기의 최대 지속 시간.

각 프레임에 대해 스크립트가 함께 블렌딩할 스카이박스 큐브맵을 선택합니다. 선택된 스카이박스가 쉐이더의 텍스처로 설정되고 쉐이더는 텍스처를 렌더링하면서 블렌딩합니다.

텍스처를 설정하기 위해 얼음 동굴 데모는 Unity 쉐이더 전역 기능을 사용합니다. 이 기능을 통해 사용자가 응용 프로그램의 모든 쉐이더가 사용할 수 있는 텍스처를 한 장소에 설정할 수 있습니다. 다음 코드가 이를 보여줍니다.

```
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);
```

———— 참고 ————

사용된 샘플러의 이름이 쉐이더가 로컬에서 정의한 이름과 충돌하면 안 됩니다.

————————

다음은 스크립트 코드에서 설정됩니다

- 보간되는 두 큐브맵. 값 `idx1` 및 `idx2`는 경과 시간을 기준으로 연산됩니다.
- 쉐이더에서 두 큐브맵을 블렌딩하는 데 사용되는 `blendAlpha` 인수.
- 태양 구체를 렌더링하는 데 사용되는 정규화된 태양 위치.
- 다수의 태양 매개 변수.

- 태양의 색상.
- 구름 큐브맵. ShaderCubemap1 및 ShaderCubemap2는 고유한 샘플러 이름(이 경우 _SkyboxCubemap1 및 _SkyboxCubemap2)을 포함하는 두 개의 문자열입니다.

쉐이더에서 이들 텍스처에 액세스하려면 다음 코드를 사용하여 해당 텍스처를 선언해야 합니다.

```
samplerCUBE _SkyboxCubemap1;
samplerCUBE _SkyboxCubemap2;
```

스크립트는 사용자가 지정한 목록에 따라 태양 색상 및 주변 색상을 선택합니다. 이들 색상은 각 단계에 대해 보간됩니다.

태양을 올바른 색상으로 렌더링하기 위해 태양 색상이 쉐이더로 전달됩니다.

주변 색상은 Unity 변수 RenderSettings.ambientSkyColor를 동적으로 설정하는 데 사용됩니다.

```
RenderSettings.ambientSkyColor = _ambientColor;
```

이 변수를 설정하면 모든 머티리얼이 올바른 주변색을 수신합니다. 얼음 동굴 데모에서 이 효과는 일중 단계에 따라 장면의 전체 색상을 점진적으로 변화시킵니다.

### 9.13.3 태양 렌더링

태양을 렌더링하려면 조각 쉐이더에서 스카이박스의 각 픽셀에 대해 해당 픽셀이 태양의 원주 내부에 위치하는지 확인해야 합니다.

이를 위해 쉐이더가 렌더링되는 픽셀의 월드 좌표로 정규화된 태양 위치 벡터와 정규화된 뷰 방향 벡터의 내적을 계산합니다.

다음 그림은 태양 렌더링을 보여줍니다. 정규화된 뷰 벡터와 태양 위치의 내적은 조각이 P2로서 하늘로 렌더링되는지, 또는 P1으로서 태양으로 렌더링되는지 여부를 결정하는 데 사용됩니다.

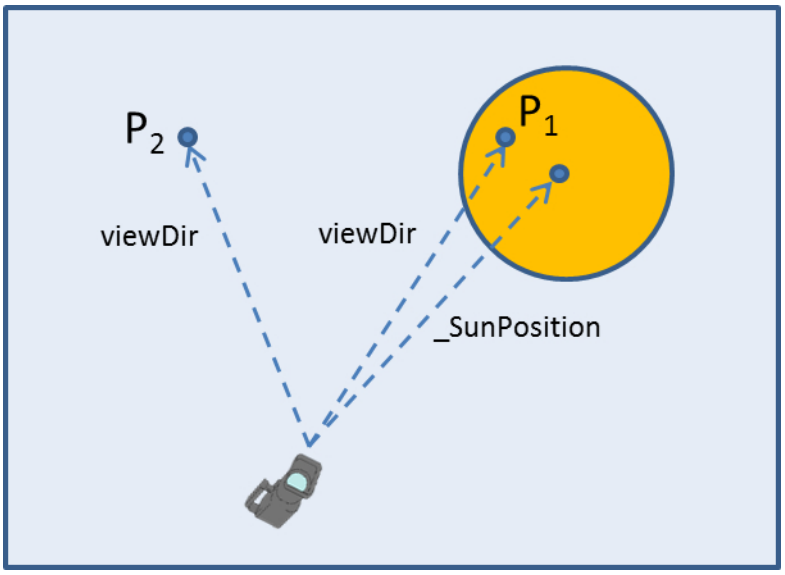


그림 9-79 태양 렌더링

정규화된 태양 위치 벡터는 C# 스크립트에 의해 쉐이더로 전달됩니다.

- 결과가 특정 임계값보다 큰 경우 픽셀이 태양 색상으로 채색됩니다.
- 결과가 특정 임계값보다 작은 경우 픽셀이 하늘 색상으로 채색됩니다.

또한 내적은 태양의 가장자리로 가면서 페이드 효과도 생성합니다.

```
half _sunContribution = dot(viewDir,_SunPosition);
```

다음 그림에서는 태양이 또렷하게 보입니다.

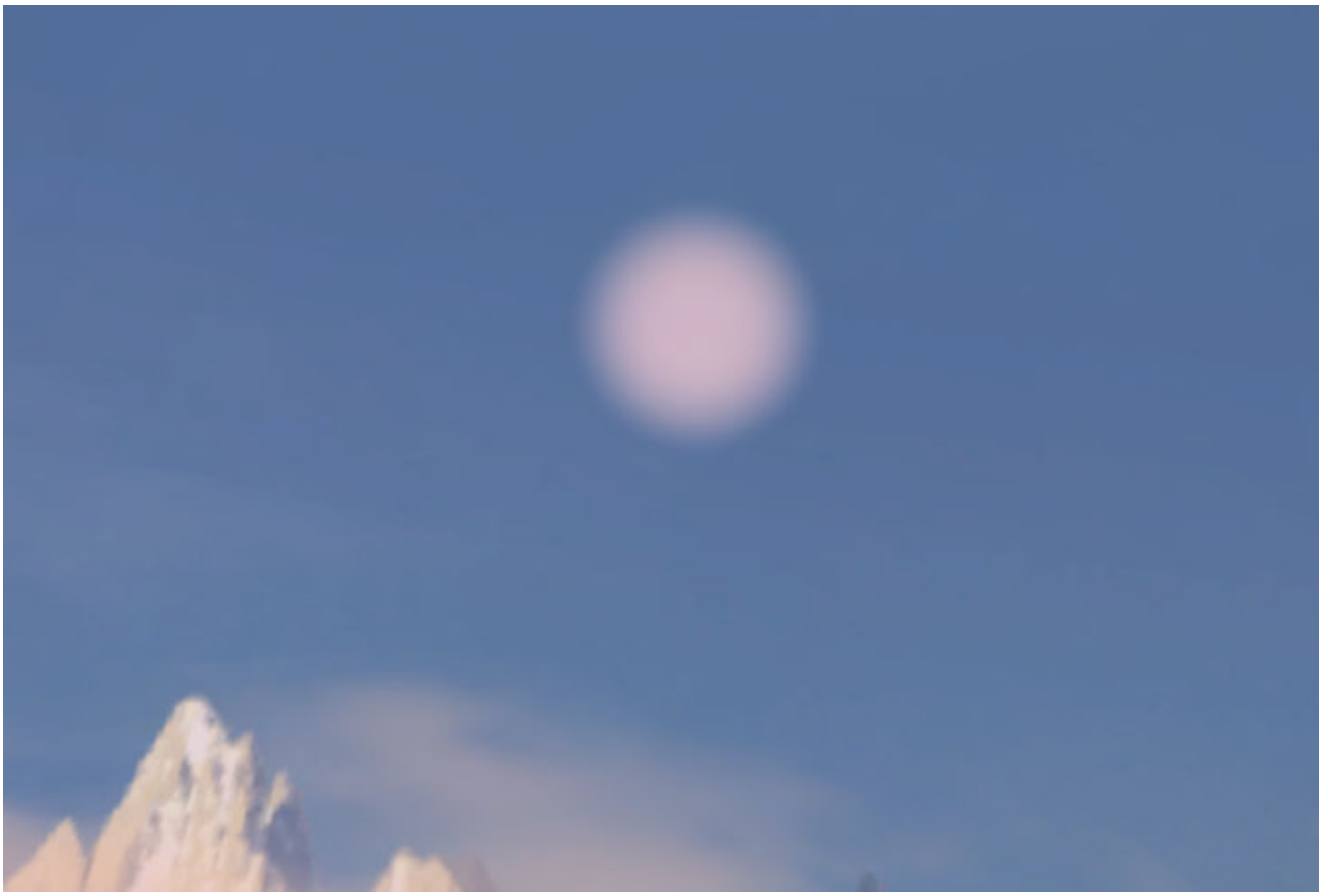

그림 9-80 맑은 하늘 조건에서 절차적 태양 렌더링

### 9.13.4 태양을 산 뒤로 페이드

태양이 하늘에 낮게 떠 있는 경우 문제가 있습니다. 실제 세계에서는 산 뒤쪽으로 사라질 것입니다.

이 효과를 구현하기 위해 큐브맵의 알파 채널이 사용되어 텍셀이 하늘을 표현할 경우 값 `0`을 저장하고, 텍셀이 산을 표현할 경우 값 `1`을 저장합니다.

태양을 렌더링하는 동안 텍스처가 샘플링되고 알파가 사용되어 태양을 산 뒤로 페이드합니다. 이 샘플링은 텍스처가 이미 산을 렌더링하기 위해 샘플링되어 있기 때문에 실제로 연산 비용이 거의 없습니다.

또한 산의 눈 덮인 가장자리 또는 부분 가까이에서 점진적으로 알파를 페이드할 수도 있습니다. 그러면 연산이 거의 필요 없이 눈에서 반사되는 태양의 효과가 생성됩니다.

유사한 기법이 낮은 연산 비용으로 구름의 표면하 산란 효과를 생성하기 위해 사용됩니다.

원래의 단계 큐브맵이 두 개의 그룹으로 분할됩니다.

- 한 큐브맵 세트는 하늘과 산을 포함합니다. 알파 값은 하늘이 0, 산이 1로 설정됩니다.
- 다른 큐브맵 세트는 구름을 포함합니다. 알파 값은 구름이 없는 부위가 0으로 설정되고 구름이 짙어질수록 점차 1로 증가합니다.

다음 그림은 산 텍스처입니다. 산은 알파 값이 1, 하늘은 알파 값이 0입니다.

그림 9-81 산 텍스처

구름 스카이박스는 구름이 없는 부분의 알파 값이 0이고, 구름이 낀 부분은 점차 1로 페이드됩니다. 구름이 인위적으로 하늘 끝에 걸쳐진 것처럼 보이지 않도록 알파 채널이 완만하게 페이드됩니다. 다음 그림은 알파가 적용된 구름 텍스처입니다.

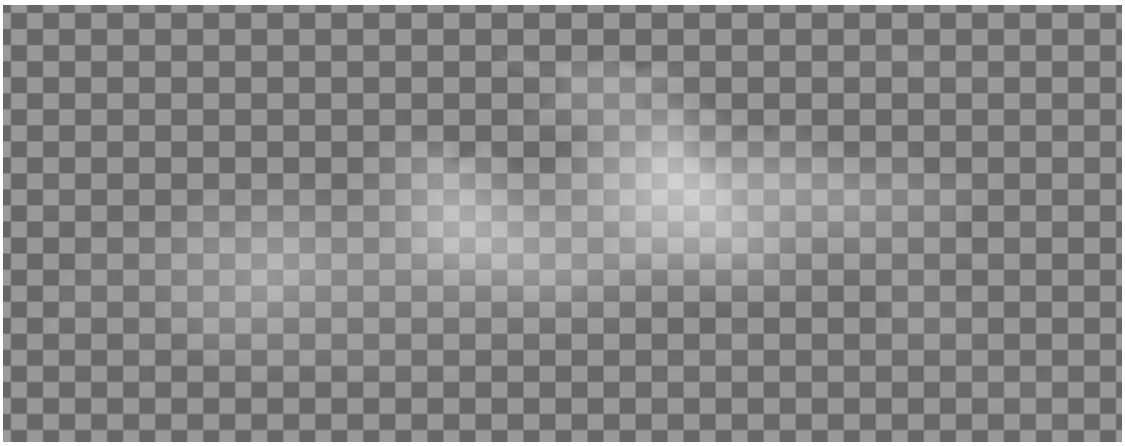

그림 9-82 구름 텍스처

쉐이더는 다음 작업을 수행합니다.

1. 현재 일중 단계를 표현하는 두 개의 스카이박스를 샘플링합니다.
2. C# 스크립트에 의해 계산된 블렌딩 인수에 따라 두 색상을 블렌딩합니다.
3. 구름 스카이박스를 샘플링합니다.
4. 구름 알파를 사용하여 2번의 색상을 구름의 색상과 블렌딩합니다.
5. 구름 알파와 스카이박스 알파를 더합니다.
6. 함수를 호출하여 현재 픽셀의 태양 색상 기여를 계산합니다.
7. 6번의 결과를 앞서 블렌딩한 스카이박스 및 구름 색상에 더합니다.

———— 참고 ————

하늘, 산, 구름을 단일 스카이박스에 넣어 이 시퀀스를 최적화할 수 있습니다. 얼음 동굴 데모에서는 이들이 분리되어 있으므로 아티스트가 용이하게 스카이박스와 구름을 개별적으로 수정할 수 있습니다.

————————

### 9.13.5 표면하 산란

알파 정보를 사용하여 태양의 반경을 늘리는 표면하 산란 효과를 추가할 수 있습니다.

실제 세계에서는 태양이 맑은 하늘에 떠 있을 경우 눈으로 들어오는 빛을 산란시키는 구름이 없기 때문에 그 크기가 비교적 작게 보입니다. 보이는 것은 태양에서 직접 비치는, 대기에 의해 아주 약간만 퍼진 광선입니다.

태양이 구름에 의해 오클루드되지만 완전히 가려지지 않은 경우에는 일부 빛이 구름 주위로 산란됩니다. 이 빛이 태양에서 약간 떨어진 방향에서 눈에 닿을 수 있습니다. 그러면 태양이 실제보다 크게 보일 수 있습니다.

다음 코드는 동굴 데모에서 이 효과를 구현하기 위해 사용된 것입니다.

```
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 );
}
```

함수 매개 변수는 viewDirection과 구름 알파와 스카이박스 알파를 더해 계산된 알파입니다.

태양 위치와 뷰 방향의 내적이 태양 중심으로부터 현재 픽셀까지의 거리를 나타내는 스케일링 인수를 계산하는 데 사용됩니다.

_sunDistanceFade 계산은 smoothstep() 함수를 사용하여 태양의 중심에서 가장자리 부근의 하늘까지 점진적 페이드를 제공합니다. 또한 구름 부근에서 태양의 반경을 늘려 표면하 산란 효과를 시뮬레이션할 수도 있습니다.

이 함수는 알파를 기반으로 한 변수 도메인을 가지며, 맑은 하늘의 경우 알파가 0이고 범위는 _SunParameters.z와 1.0 사이입니다. 이 경우 _SunParameters.z는 C# 스크립트에서 0.995로 초기화되며, 이 값은 5도 직경의 태양에 해당합니다(cos(5 degrees) = 0.995).

처리되는 픽셀이 구름을 포함하는 경우 태양 반경이 13도로 증가하여 구름에 접근할 때 늘어난 산란 효과를 생성할 수 있습니다.

_sunOcclusionFade 인수는 산과 구름으로 인한 오클루전을 기반으로 태양의 기여를 축소하는 데 사용됩니다.

다음 그림은 구름에 의해 오클루드되지 않은 태양입니다.

그림 9-83 구름에 의해 오클루드되지 않은 태양

다음 그림은 구름에 의해 오클루드된 태양입니다.

그림 9-84 구름에 의해 오클루드된 태양

## 9.14 반딧불이

반딧불이는 빛을 내며 날아다니는 곤충으로 얼음 동굴 데모에 역동성을 더하기 위해 사용되었습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.14.1 반딧불이 개요

반딧불이는 다음 구성요소로 이루어집니다.

- 런타임 시 인스턴스화되는 프리팹 객체.
- 반딧불이가 날아다닐 수 있는 영역을 제한하는 데 사용되는 상자 콜라이더.

이들 두 구성요소가 반딧불이의 움직임을 관리하고 반딧불이가 그리는 궤적을 정의하는 C# 스크립트에 의해 결합됩니다.

다음 그림은 반딧불이를 보여줍니다.

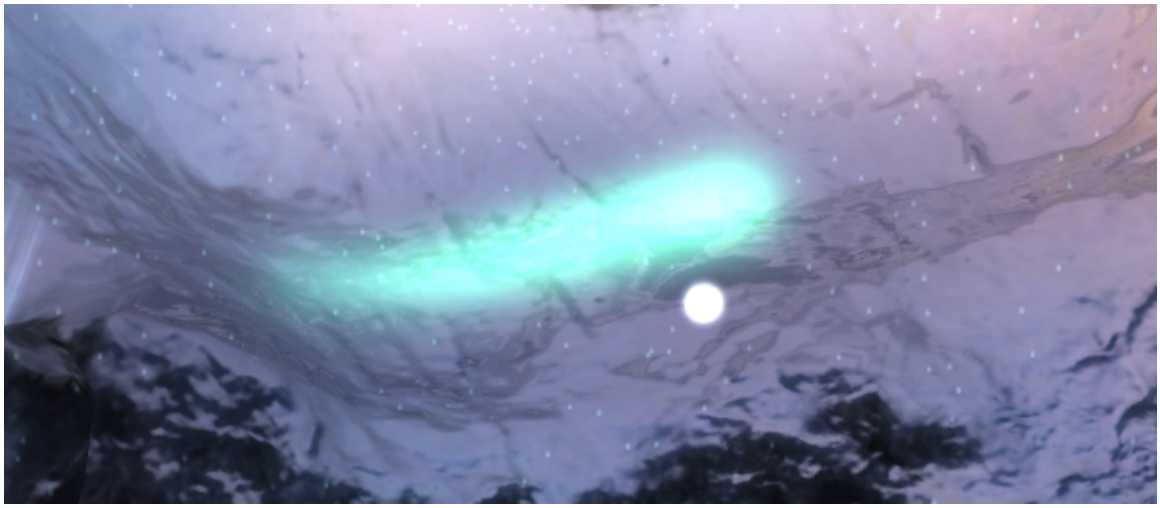

그림 9-85 반딧불이

반딧불이 프리팹은 Unity 표준 파티클 시스템을 사용하여 반딧불이의 궤적을 생성합니다.

다음 그림은 반딧불이에 의해 사용되는 Unity 파티클 시스템 설정입니다.

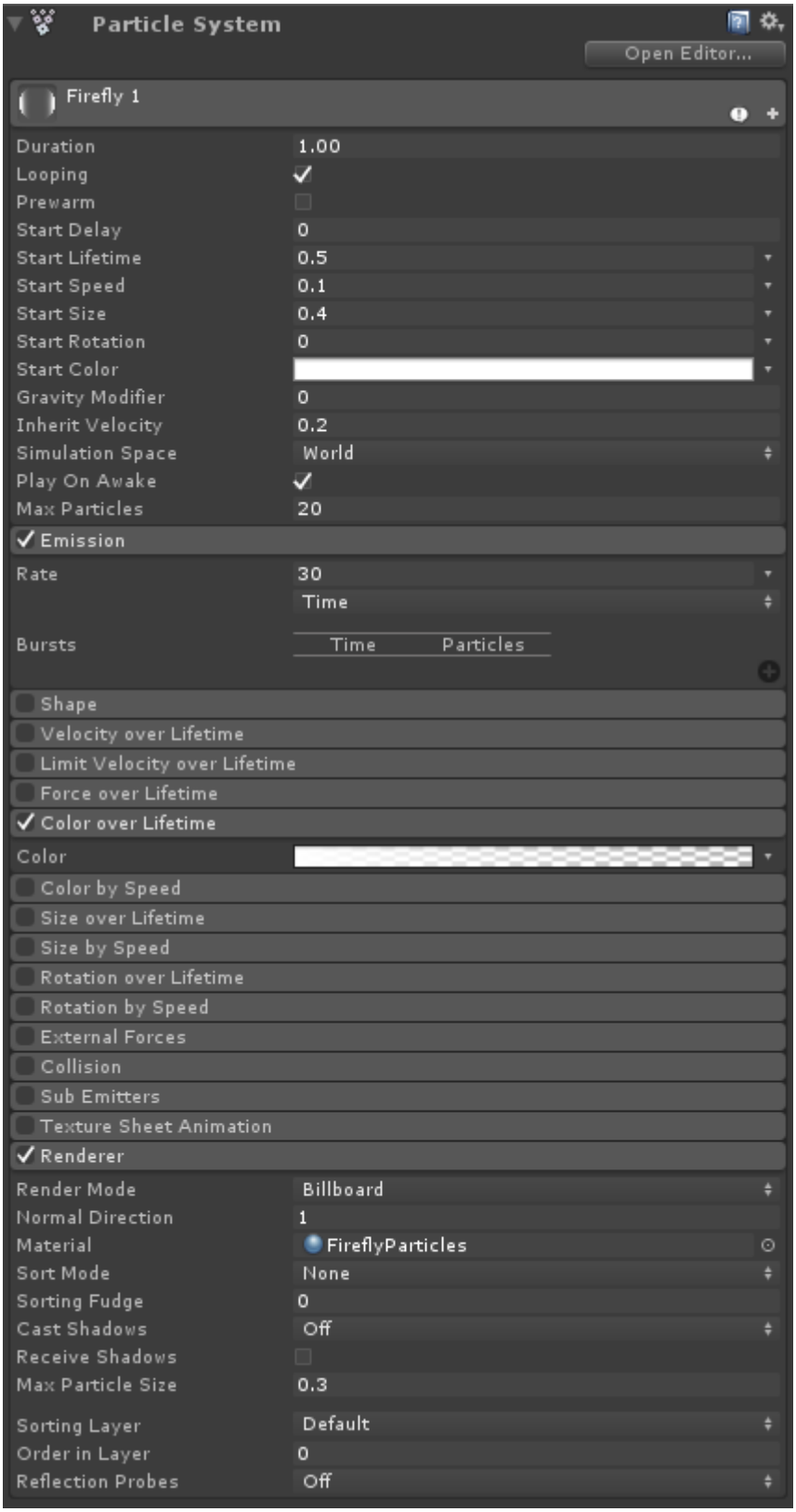


그림 9-86 Unity 파티클 시스템 설정

구현하려는 효과에 따라 Unity 트레일 렌더러를 사용하여 보다 연속적인 모양을 만들 수 있습니다.

트레일 렌더러는 각 궤적에 대해 많은 수의 삼각형을 생성합니다. 트레일 렌더러 설정 Min Vertex Distance를 수정하여 삼각형의 개수를 변경할 수 있지만 이 값이 크면 소스가 너무 빠르게 움직일 경우 궤적이 갑자기 이동할 수 있습니다.

Min Vertex Distance 옵션은 궤적을 형성하는 정점 사이의 최소 거리를 정의합니다. 이 값이 크면 직선 궤적에서는 문제가 없지만 곡선 궤적에서는 매끄럽게 보이지 않습니다.

생성된 궤적은 항상 카메라를 마주봅니다. 따라서 갑작스럽게 움직이는 소스가 궤적과 중첩될 수 있습니다. 그러면 궤적을 형성하는 삼각형의 블렌딩에 의해 아티팩트가 발생할 수 있습니다.

다음 그림은 중첩 궤적으로 인한 아티팩트를 보여줍니다.

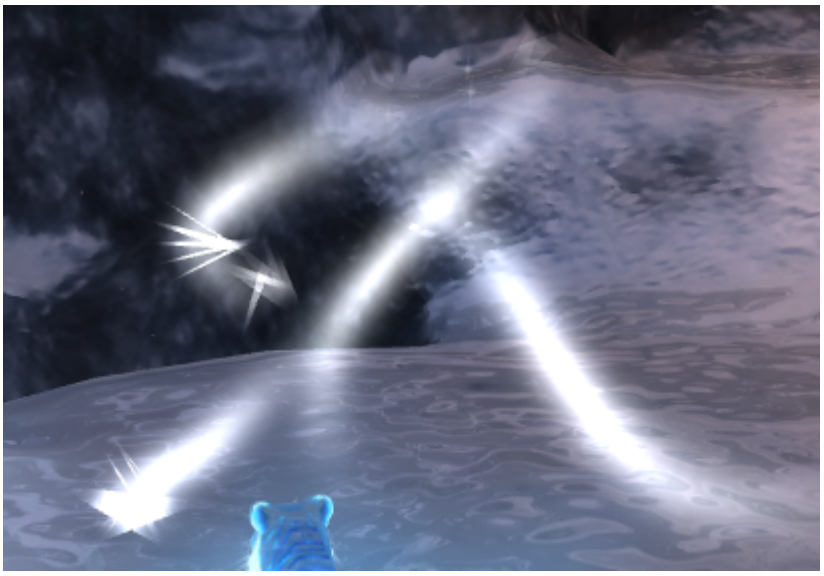

그림 9-87 중첩 궤적 아티팩트

프리팹에 추가되는 마지막 구성요소는 이동하는 동안 장면에 빛을 비추는 포인트 라이트입니다.

### 9.14.2 반딧불이 생성기 프리팹

반딧불이 생성기 프리팹은 반딧불이 생성을 관리하고 프레임마다 반딧불이를 업데이트합니다. 이 프리팹은 업데이트를 위한 C# 스크립트와 각 반딧불이가 그 안에서 움직일 수 있는 볼륨을 구획하는 상자 콜라이더로 구성됩니다.

스크립트는 생성할 반딧불이 숫자를 매개 변수로 사용하고, 프리팹이 반딧불이 객체를 인스턴스화하게 합니다. 반딧불이의 움직임이 경계 상자 내에서 무작위적이므로 반딧불이가 움직이는 방향의 변화가 특정 각도로 제한됩니다. 따라서 반딧불이가 급격하게 방향을 변화할 수 없습니다.

무작위 움직임을 생성하기 위해 구간적 3차 에르미트 보간을 통해 제어점이 생성됩니다. 에르미트 보간은 다른 경로가 서로 연결되더라도 올바로 동작하는 유연 연속 함수를 제공합니다. 끝점에서의 1차 도함수도 연속이므로 급격한 속도 변화가 없습니다.

이 보간은 시작 및 끝에 대해 각각 제어점이 필요하며, 각 제어점에 대해 2개의 탄젠트가 필요합니다. 제어점을 무작위로 생성되므로 스크립트가 3개의 제어점과 2개의 탄젠트를 저장할 수 있습니다. 첫 번째 및 두 번째 제어점의 위치를 사용하여 첫 번째 제어점의 탄젠트가 정의되고, 두 번째 및 세 번째 제어점을 사용하여 두 번째 제어점의 탄젠트가 정의됩니다.

스크립트는 로딩 시 각 반딧불이에 대해 다음을 생성합니다.

- 초기 위치.
- Unity 함수 `Random.onUnitSphere()`를 사용하여, 초기 방향.

다음 코드는 제어점이 초기화되는 방식을 보여줍니다.

```
_fireflySpline[i*_controlPoints] = initialPosition;
Vector3 randDirection = Random.onUnitSphere;
_fireflySpline[i*_controlPoints+1] = initialPosition + randDirection;
_fireflySpline[i*_controlPoints+2] = initialPosition + randDirection * 2.0f;
```

초기 제어점은 직선 상에 놓입니다. 탄젠트는 이들 제어점을 사용하여 생성됩니다.

```
//The tangent for the first point is in the same direction as the initial direction vector
_fireflyTangents[i*_controlPoints] = randDirection;

//This code computes the tangent from the control point positions. It is shown here for
// reference because it can be set to randDirection at initialization.
_fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
_fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1] -
_fireflySpline[i*_controlPoints])/2;
```

반딧불이 초기화를 완료하려면 반딧불이의 색상과 현재 주기 지속 시간을 설정해야 합니다.

각 프레임마다 스크립트가 다음 코드로 에르미트 보간을 계산하여 반딧불이의 위치를 업데이트합니다.

```
// t is the parameter that defines where in the curve the firefly is placed. It represents
// the ratio of the time the firefly has traveled along the path to the total time.
float t = _fireflyLifetime[i].y / _fireflyLifetime[i].x;

//Hermite interpolation parameters
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);
//Firefly updated position
_fireflyObjects[i].transform.position = h00 * A + h10 * _fireflyTangents[i*_controlPoints]
+ h01 * B + h11 * _fireflyTangents[i*_controlPoints+1];
```

반딧불이가 무작위로 생성된 경로의 전체 구간을 완료하면 스크립트가 현재 구간의 끝에서 시작하는 새로운 무작위 구간을 생성합니다.

```
//t > 1.0 indicates the end of the current path
if( t >= 1.0 )
{

    //Update the new position
    //Shift the second point to the first as well as the tangent
    _fireflySpline[i*_controlPoints] = _fireflySpline[i*_controlPoints+1];
    _fireflyTangents[i*_controlPoints] = _fireflyTangents[i*_controlPoints+1];

    //Shift the third point to the second, this point doesn't have a tangent
    _fireflySpline[i*_controlPoints+1] = _fireflySpline[i*_controlPoints+2];

    //Get new random control point within a certain angle from the current fly direction
    _fireflySpline[i*_controlPoints+2] = GetNewRandomControlPoint();

    //Compute the tangent for the central point
    _fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
    _fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1] -
    _fireflySpline[i*_controlPoints])/2;

    //Set how long should take to navigate this part of path
    _fireflyLifetime[i].x = _fireflyMinLifetime;

    //Timer used to check how much we traveled along the path
    _fireflyLifetime[i].y = 0.0f;
}
```

## 9.15 탄젠트 공간-월드 공간 노말 변환 도구

탄젠트 공간-월드 공간 노말 변환 도구는 C# 스크립트와 쉐이더로 구성됩니다. 이 도구는 Unity 편집기 내부에서 오프라인으로 실행되며 게임의 런타임 성능에는 영향을 미치지 않습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 9.15.1 탄젠트 공간-월드 공간 변환 도구

얼음 동굴 데모 프로파일링에서 산술 파이프라인의 병목이 확인되었습니다. 얼음 동굴 데모는 부하를 줄이기 위해 정적 지오메트리에 대해 탄젠트 공간 노말 맵이 아니라 월드 공간 노말 맵을 사용합니다.

탄젠트 공간 노말 맵은 애니메이션화된 객체와 동적 객체에 유용하지만 샘플링된 노말을 올바로 지향시키기 위해 추가 연산이 필요합니다.

얼음 동굴 데모의 지오메트리는 대부분 정적이므로 노말 맵이 월드 공간 노말 맵으로 변환되었습니다. 그러면 텍스처에서 샘플링된 노말이 월드 공간에서 올바로 지향됩니다. 얼음 동굴 데모 조명이 사용자 지정 쉐이더에서 연산되는 반면 Unity 표준 쉐이더는 탄젠트 공간 노말 맵을 사용하므로 이 변경이 가능합니다.

변환 도구는 다음 구성요소로 이루어집니다.

- 편집기에 새 옵션을 추가하는 C# 스크립트.
- 변환을 수행하는 쉐이더.

이 도구는 Unity 편집기 내부에서 오프라인으로 실행되며 게임의 런타임 성능에는 영향을 미치지 않습니다.

### 9.15.2 C# 스크립트

C# 스크립트를 Unity `Assets/Editor` 디렉터리에 배치해야 합니다. 그러면 스크립트가 Unity 편집기의 GameObject 메뉴에 새 옵션을 추가합니다. 이 디렉터리가 없을 경우 새로 만드십시오.

다음 코드는 Unity 편집기에 새 옵션을 추가하는 방법을 보여줍니다.

```
[MenuItem("GameObject/World Space Normals Creator")]

static void CreateWorldSpaceNormals ()
{
    ScriptableWizard.DisplayWizard("Create World Space Normal",
    typeof(WorldSpaceNormalsCreator),"Create");
}
```

다음 그림은 스크립트가 추가한 GameObject 메뉴 옵션을 보여줍니다.

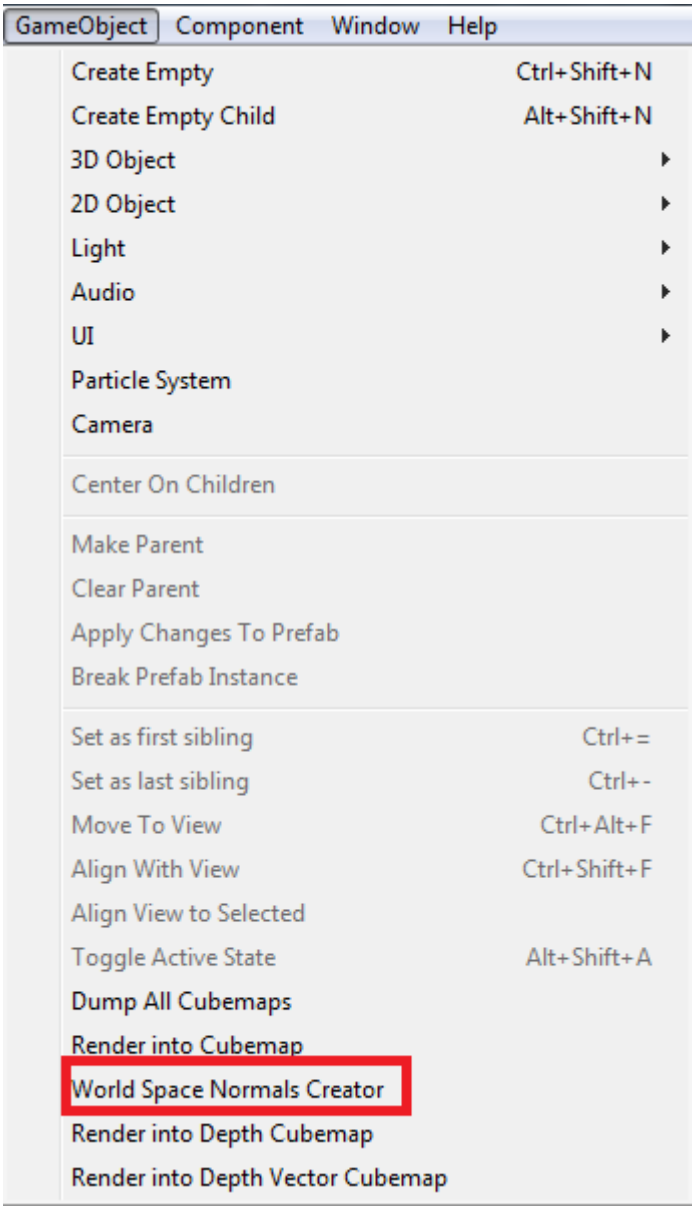


그림 9-88 스크립트에 의해 추가된 GameObject 메뉴 옵션

C# 스크립트에서 정의된 클래스는 Unity `ScriptableWizard` 클래스에서 파생된 것이며 그 멤버 중 일부에 액세스할 수 있습니다 이 클래스에서 파생하여 편집기 마법사를 만듭니다. 편집기 마법사는 일반적으로 메뉴 항목을 사용하여 엽니다.

`OnWizardUpdate` 코드에서 `helpString` 변수는 마법사가 생성된다는 창에 표시되는 도움말 메시지를 포함합니다.

`isValid` 멤버는 올바른 매개 변수가 모두 선택되고 Create 버튼이 사용 가능한 때를 정의하는 데 사용됩니다. 이 경우 `_currentObj` 멤버가 유효한 객체를 가리키는지 확인됩니다.

마법사 창의 필드는 클래스의 퍼블릭 멤버입니다. 이 경우 `_currentObj`만 퍼블릭이므로 마법사 창에 하나의 필드만 있습니다.

다음 그림은 사용자 지정 마법사 창입니다.

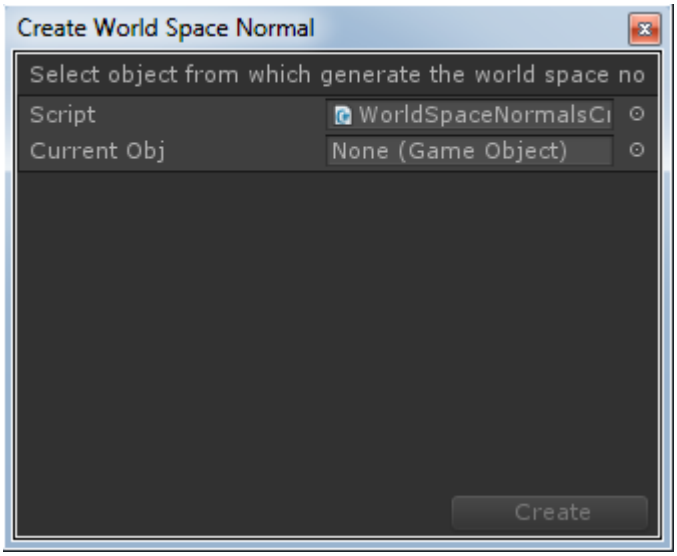


그림 9-89 사용자 지정 마법사 창

객체를 선택하고 Create 버튼을 클릭하면 OnWizardCreate() 함수가 호출됩니다.

OnWizardCreate() 함수가 변환 작업을 대부분 수행합니다.

노말로 변환하기 위해 이 도구가 새로운 월드 공간 노말을 RenderTexture로 렌더링하는 임시 카메라를 생성합니다. 이를 위해 카메라가 직교 모드로 설정되고 객체의 레이어가 미사용 수준으로 변경됩니다. 이는 카메라가 이미 장면의 일부이더라도 자체적으로 객체를 렌더링할 수 있음을 의미합니다.

다음 코드는 카메라가 설정되는 방식을 보여줍니다.

```
// Set antialiasing
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
```

이 스크립트는 변환을 실행하는 대체 쉐이더를 설정합니다.

```
_renderCamera.SetReplacementShader (wns,null);
_renderCamera.useOcclusionCulling = false;
```

카메라는 객체를 향해 배치됩니다. 그러면 프루스텀 컬링 도중 객체가 렌더링에서 제거되는 것이 방지됩니다.

```
_renderCamera.transform.rotation = Quaternion.LookRotation (_currentObj.transform.position -
_renderCamera.transform.position);
```

객체에 할당된 각 자료에 대해 스크립트가 _BumpMap 텍스처를 찾습니다. 이 텍스처는 쉐이더 전역 함수를 사용하여 대체 쉐이더의 소스 텍스처로 설정됩니다.

음의 방향을 가리키는 노말이 표현되어야 하므로 투명 색상이 (0.5,0.5,0.5)로 설정됩니다.

```
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;
    }
    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);
```

카메라가 장면을 렌더링한 후 픽셀이 PNG 이미지로 저장됩니다.

```
    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);
    }
```

카메라 컬링 마스크는 16진수 형식으로 표현되는 이진 마스크를 사용하여 렌더링할 레이어를 지정합니다.

이 경우 레이어 30이 사용되었습니다.

```
_currentObj.layer = 30;
```

16진수는 30번째 비트가 1로 설정되었으므로 0x40000000입니다.

### 9.15.3 WorldSpaceNormalCreator 쉐이더

변환을 구현하는 쉐이더 코드는 상당히 간단합니다. 이 쉐이더는 실제 정점 위치가 아니라 정점의 텍스처 좌표를 그 위치로 사용합니다. 그러면 텍스처링과 마찬가지로 객체가 2D 평면에 투영됩니다.

OpenGL 파이프라인이 올바로 작동하도록 UV 좌표가 표준적인 `[0,1]` 범위에서 `[-1,1]` 범위로 이동되고 Y 좌표가 반전됩니다. Z 좌표는 사용되지 않습니다. 따라서 `0` 또는 근거리 및 원거리 클립 평면 내의 임의의 값으로 설정할 수 있습니다.

```
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;
```

노말, 탄젠트 및 바이탄젠트는 정점 쉐이더에서 계산되어 변환을 실행할 조각 쉐이더로 전달됩니다.

```
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);
```

조각 쉐이더:

1. 노말을 탄젠트 공간에서 월드 공간으로 변환합니다.
2. 노말을 `[0,1]` 범위로 스케일링합니다.
3. 노말을 새 텍스처로 출력합니다.

다음 코드가 이를 보여줍니다.

```
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);
```

다음 그림은 처리되기 전의 탄젠트 공간 노말 맵입니다.

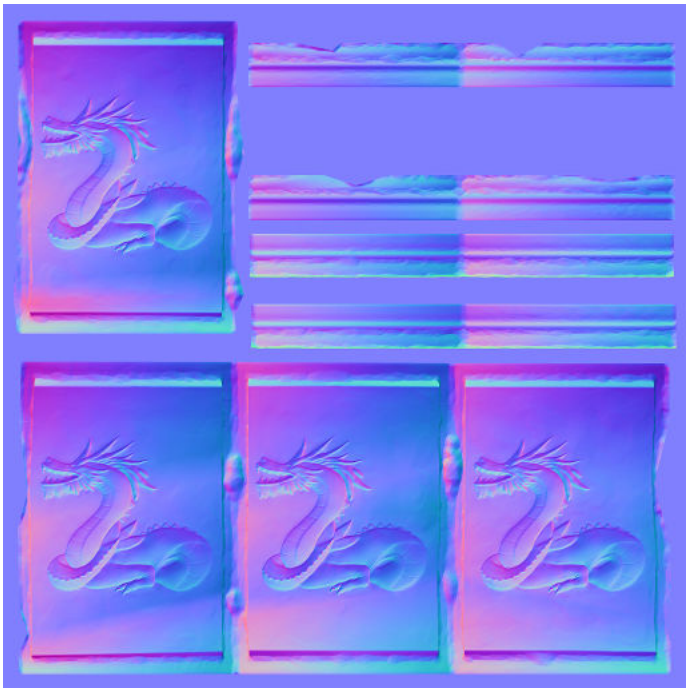

그림 9-90 원래 탄젠트 공간 노말 맵

다음 그림은 도구에 의해 생성된 월드 공간 노말 맵입니다.

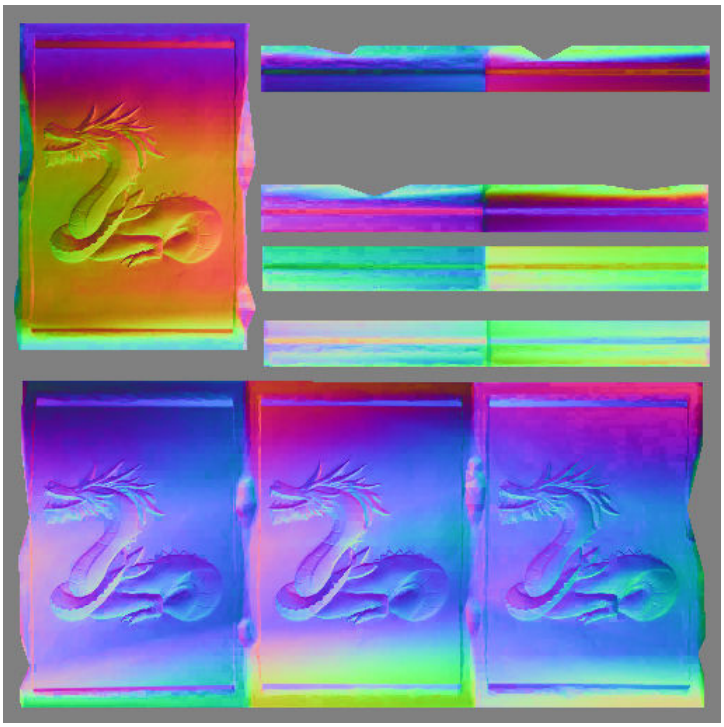

그림 9-91 생성된 월드 공간 노말 맵

### 9.15.4 WorldSpaceNormalsCreators C# 스크립트

다음은 WorldSpaceNormalsCreators C# 스크립트의 코드입니다.

```
using UnityEngine;
using UnityEditor;
using System.Collections;

public class WorldSpaceNormalsCreator : ScriptableWizard
```

```
{
    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 ()
    {
        // Set antialiasing
        QualitySettings.antiAliasing = 4;
        Shader wns = Shader.Find ("Custom/WorldSpaceNormalCreator");
        GameObject go = new GameObject( "WorldSpaceNormalsCam", typeof(Camera) );

        //Set the new camera to perform orthographic projection
        _renderCamera = go.GetComponent<Camera> ();
        _renderCamera.orthographic = true;
        _renderCamera.nearClipPlane = 0.0f;
        _renderCamera.farClipPlane = 10f;
        _renderCamera.orthographicSize = 1.0f;

        //Save the current object layer and set it to a unused one
        int prevObjLayer = _currentObj.layer;
        _currentObj.layer = 30; //0x40000000

        //Set the replacement shader for the camera
        _renderCamera.SetReplacementShader (wns,null);
        _renderCamera.useOcclusionCulling = false;

        //Rotate the camera to look at the object to avoid frustum culling
        _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;
            }

            //Render the world space normal maps to a texture
            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;

            //Save it to PNG
            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 ()
```

```
    {
        ScriptableWizard.DisplayWizard("Create World Space Normal",
        typeof(WorldSpaceNormalsCreator),"Create");
    }

}
```

### 9.15.5 WorldSpaceNormalCreator 쉐이더 코드

다음은 WorldSpaceNormalCreator 셰이더의 코드입니다.

```
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
        }
    }
}
```

# 제 10 장
# 가상현실

이 장에서는 응용 프로그램 또는 게임을 가상현실 하드웨어에서 실행되도록 조정하는 프로세스와 가상현실에서 반사를 구현할 때 몇몇 차이점에 대해 설명합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 10.1 가상현실 하드웨어의 Unity 지원

여러 유형의 *가상현실*(VR) 하드웨어가 Unity를 지원합니다. Unity는 기본으로 일부 디바이스를 지원합니다. 플러그인을 통해 일부 다른 디바이스가 지원될 수 있습니다.

네이티브 Unity VR 지원을 제공하는 디바이스가 플러그인을 사용하여 지원되는 디바이스보다 뛰어난 성능을 발휘합니다. Unity가 네이티브 Unity VR 지원을 제공하는 디바이스에 대해 내부 최적화를 구현하기 때문입니다.

다음 디바이스는 네이티브 Unity VR 지원을 제공합니다.

- Oculus Rift.
- 삼성 기어 VR.
- PlayStation VR.
- Microsoft Hololens.

다음 디바이스는 플러그인을 통해 Unity VR 지원이 가능합니다.

- Google Cardboard.
- Moverio.
- HTC Vive 및 기타 SteamVR 플랫폼을 지원하는 디바이스.

다수의 다른 디바이스가 Unity용 *Open Source Virtual Reality*(OSVR) 플러그인을 사용하여 지원될 수 있습니다.

## 10.2 Unity VR 포팅 프로세스

응용 프로그램 또는 게임을 네이티브 Unity VR로 포팅하는 프로세스는 여러 단계로 구성됩니다.

응용 프로그램을 Unity VR로 포팅하는 데 필요한 단계는 다음과 같습니다.

1. Unity 버전 5.1 이상을 설치합니다. Unity 버전 5.1 이상은 기본으로 VR을 지원합니다.
2. 필요한 경우, 해당 웹 사이트에서 디바이스별 서명 파일을 구해 디바이스의 해당 폴더에 저장합니다.

   얼음 동굴 데모를 실행하는 삼성 기어 VR의 경우 Oculus 개발자 웹 사이트 *https://developer.oculus.com/osig*입니다. 삼성 디바이스용 서명 파일은 `Plugins/Android/assets` 폴더에 저장해야 합니다.
3. Unity에서 Open File > Build Settings > Player Settings를 선택합니다. Player Settings 창이 나타납니다. Other Settings 섹션에서 Virtual Reality Supported 옵션을 켭니다.

   다음 그림은 이 창의 스크린샷입니다.

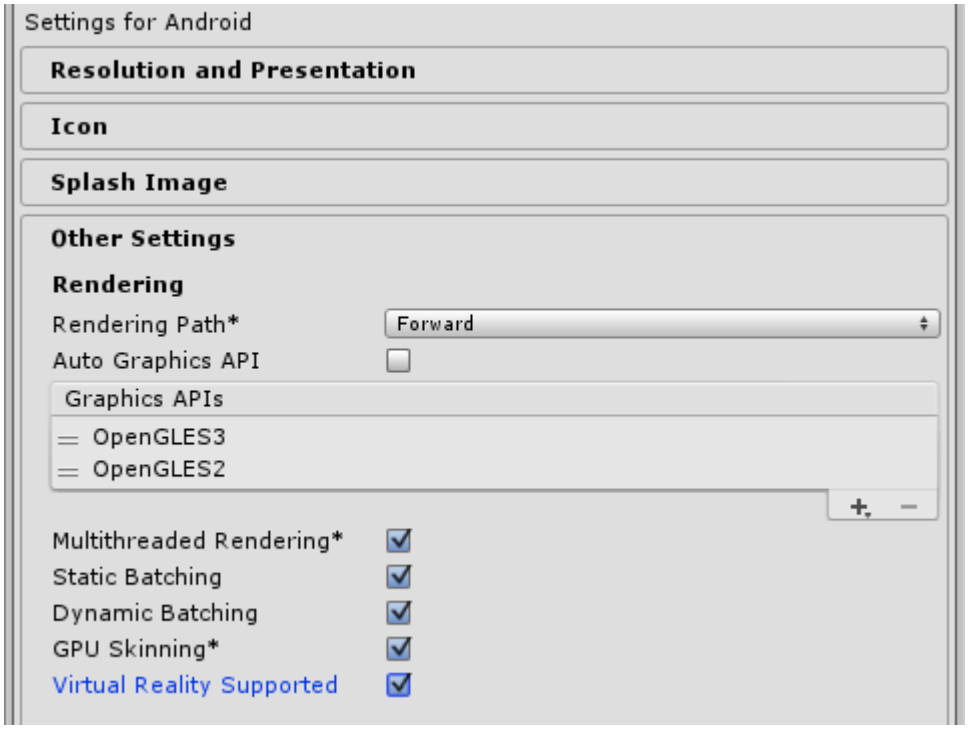


그림 10-1 Player Settings 창

4. 카메라의 상위를 설정합니다. 모든 카메라 컨트롤은 카메라 위치 및 방향을 이 카메라 상위로 설정해야 합니다.
5. 필요할 경우 카메라 컨트롤을 VR 헤드셋 터치 패드와 연결합니다.
6. Android 디바이스의 경우 Developer options 메뉴를 사용으로 설정하고 USB debugging을 켭니다.

   다음 그림은 Developer options 메뉴입니다.

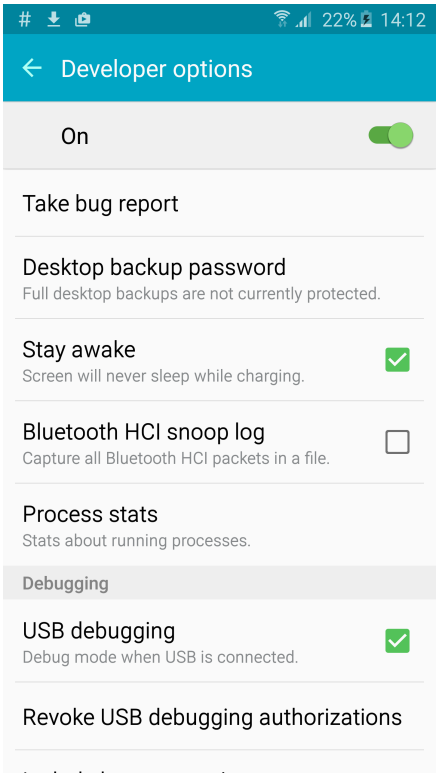


그림 10-2 Developer options 메뉴

7. 응용 프로그램을 제작하여 디바이스에 설치합니다.
8. 응용 프로그램을 시작합니다.

삼성 디바이스에서 응용 프로그램을 시작하면 디바이스를 헤드셋에 삽입하라는 메시지가 표시됩니다. 디바이스가 VR 준비가 되지 않은 경우 네트워크에 연결하여 삼성 VR 소프트웨어를 다운로드하라는 메시지가 표시됩니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *10.2.1 삼성 기어 VR 개발자 모드 활성화* 페이지의 10-242.

### 10.2.1 삼성 기어 VR 개발자 모드 활성화

개발자 모드에서는 VR 헤드셋에 디바이스를 삽입하지 않고 실행 중인 VR 응용 프로그램을 시각화할 수 있습니다.

서명된 VR 응용 프로그램을 이미 설치한 경우 삼성 기어 VR 개발자 모드를 활성화할 수 있습니다. 서명된 VR 응용 프로그램을 이미 설치하지 않은 경우에는 이 모드를 활성화할 수 있도록 서명된 VR 응용 프로그램을 설치할 수 있습니다.

개발자 모드를 활성화하는 방법:

1. 삼성 디바이스에서 Settings > Application Manager > Gear VR Service를 선택합니다.
2. Manage storage를 선택합니다.
3. VR 서비스 버전을 6번 탭합니다.
4. 스캔 프로세스가 끝날 때까지 기다립니다. 개발자 모드 토글이 나타납니다.

———— 참고 ————

개발자 모드를 사용하면 헤드셋을 사용하지 않을 경우 헤드셋을 끄는 모든 설정이 재정의되므로 스마트폰의 배터리 수명이 줄어듭니다.

————————

다음 그림은 삼성 기어 VR 개발자 모드 뷰의 얼음 동굴 예시 스크린샷입니다.

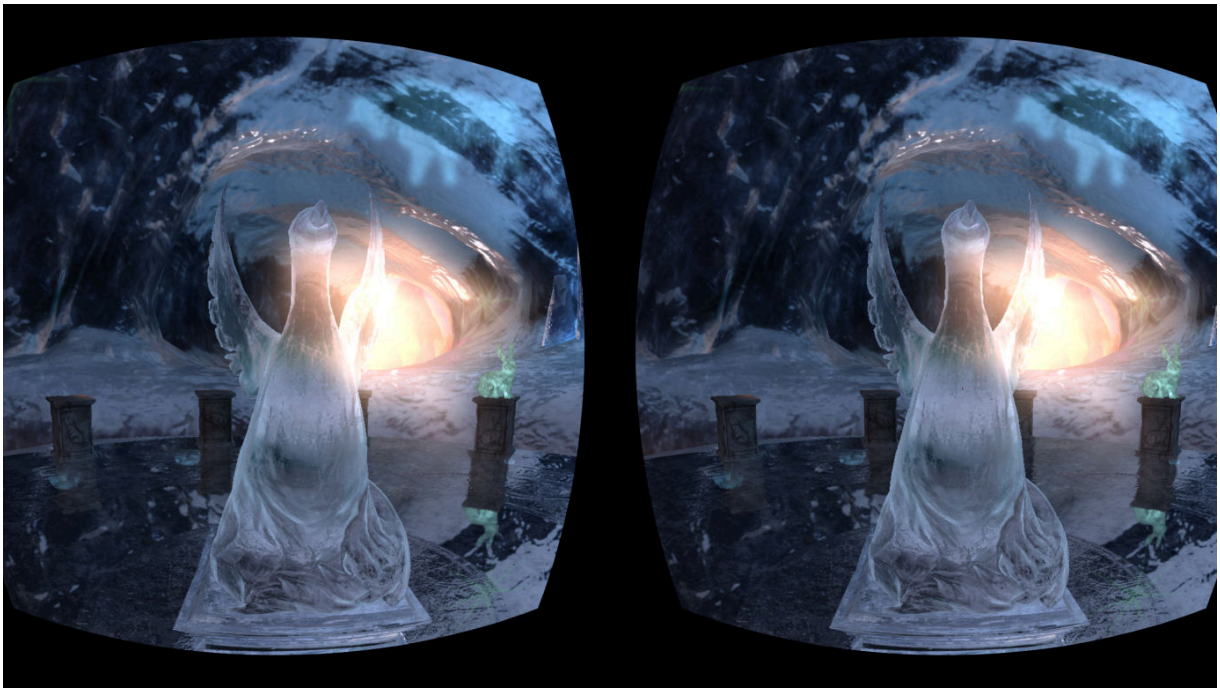

그림 10-3 삼성 기어 VR 개발자 모드에서 실행되는 VR 응용 프로그램의 예시 스크린샷

## 10.3 VR로 포팅 시 고려할 사항

VR은 다른 응용 프로그램 유형과는 매우 다른 사용자 경험을 만듭니다. 이는 비 VR 응용 프로그램 또는 게임에서는 유효하지만 VR에서는 그렇지 않은 요소들이 있음을 의미합니다.

몇몇 사용자에서 응용 프로그램을 테스트하여 각 사용자의 쾌적도를 파악하여 이들이 경험을 쾌적하게 생각하도록 코드를 수정하십시오.

비 VR 게임에서는 쾌적한 카메라 애니메이션이 동일한 게임의 VR 버전에서 불쾌할 수 있습니다. 예를 들어 VR이 없는 얼음 동굴 데모에는 카메라 애니메이션 모드가 있습니다. 일부 사용자는 이 모드를 불쾌하게 생각하고, 특히 카메라가 뒤로 이동할 경우 멀미를 호소합니다. 이 모드를 제거하면 이러한 불쾌한 경험이 방지됩니다.

비 VR 응용 프로그램은 스마트폰의 터치 스크린을 사용하거나 스마트폰을 기울여 제어할 수 있습니다. 이러한 컨트롤 메커니즘이 VR 응용 프로그램에서는 가능하지 않을 수 있습니다. 예를 들어 오리지널 얼음 동굴 데모는 2개의 가상 조이스틱을 사용하여 카메라를 제어하는데, VR 디바이스에서는 터치 스크린에 액세스할 수 없으므로 이렇게 할 수 없습니다. 얼음 동굴 VR 데모는 삼성 기어 VR에서 실행하도록 설계되었는데, 이 디바이스는 헤드셋의 측면에 터치 패드가 있습니다. 터치 스크린 대신 터치 패드를 사용하면 이 문제가 해결됩니다.

다음 그림은 삼성 기어 VR 헤드셋의 터치 패드입니다.

그림 10-4 삼성 기어 VR 헤드셋의 터치 패드

비 VR 응용 프로그램에서는 제대로 보이는 시각적 효과 중 일부가 VR 응용 프로그램에서는 제대로 보이지 않을 수 있습니다. 예를 들어 비 VR 얼음 동굴 데모는 카메라-태양 정렬에 따라 강도가 변하는 더티 렌즈 효과를 사용합니다. 이 효과를 VR에서 테스트했을 때는 제대로 보이지 않았기 때문에 제거되었습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 10.3.1 카메라의 외부 디바이스 제어

VR은 일반적으로 스마트폰 및 모바일 디바이스와 연결되지 않는 컨트롤 방법을 활용할 수 있습니다. 응용 프로그램의 타겟 사용자에 따라 이러한 방법 중 일부는 고려할 가치가 있습니다.

블루투스를 통해 VR 디바이스에 연결할 수 있는 컨트롤러가 있습니다. 이를 구현하기 위해 얼음 동굴 데모는 Unity가 Android 블루투스 이벤트를 해석할 수 있도록 기능을 확장하는 사용자 지정 플러그인을 사용합니다. 이러한 이벤트는 카메라 동작을 트리거합니다.

다음 그림은 얼음 동굴 데모에서 사용되는 블루투스 컨트롤러입니다.


그림 10-5 얼음 동굴 데모를 제어하는 블루투스 컨트롤러

## 10.4 VR에서의 반사

반사는 모든 VR 응용 프로그램에서 중요합니다. 실제 세계에서는 수많은 반사가 발생하므로 게임 또는 응용 프로그램에서 반사가 없다면 사람들이 알아차립니다.

VR에서의 반사는 전통적 게임에서와 동일한 기법을 사용하지만, 사용자가 보는 입체 시각적 출력에서 작동하도록 수정해야 합니다.

반사를 제대로 구현할 경우 응용 프로그램 또는 게임이 보다 사실적이고 몰입적으로 느껴질 수 있습니다. 하지만 VR에서 반사를 구현할 때 추가로 고려할 사항이 몇 가지 있습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *10.4.1 로컬 큐브맵을 사용한 반사* 페이지의 10-246.
- *10.4.2 다양한 유형의 반사를 결합* 페이지의 10-246.
- *10.4.3 입체 반사* 페이지의 10-246.

### 10.4.1 로컬 큐브맵을 사용한 반사

로컬 큐브맵을 사용하여 반사를 생성할 수 있습니다. 이 방법은 각 프레임마다 반사 텍스처를 생성하는 대신, 미리 렌더링된 큐브맵에서 반사 텍스처를 가져옵니다. 이 방법은 큐브맵이 생성된 위치와 장면 경계 상자를 기반으로 반사 벡터에 로컬 수정을 적용한 다음, 이 정보를 사용하여 올바른 텍스처를 가져옵니다.

로컬 큐브맵을 사용하여 생성된 반사는 각 프레임에 대해 런타임 시 반사가 생성될 경우 발생할 수 있는 픽셀 불안정성 또는 픽셀 시머링의 영향을 받지 않습니다.

로컬 큐브맵을 사용한 반사에 대한 자세한 내용은 *9.2 로컬 큐브맵을 사용하여 반사 구현* 페이지의 9-153을 참조하십시오.

### 10.4.2 다양한 유형의 반사를 결합

최상의 성능 및 효과를 구현하려면 다양한 반사 생성 기법이 필요합니다. 어느 기법이 필요한지는 반사 표면의 형태에 따라, 또한 반사 표면과 반사되는 객체가 정적 또는 동적인지에 따라 달라집니다. 다양한 반사 생성 기법의 결과를 결합하여 사용자가 보는 결과를 생성해야 합니다.

다양한 반사 유형을 결합하는 자세한 내용은 *9.3 반사 결합* 페이지의 9-169을 참조하십시오.

### 10.4.3 입체 반사

비 VR 게임에서는 카메라 관점이 하나뿐입니다. VR 게임에서는 카메라 관점이 눈마다 하나씩 있습니다. 이는 반사가 좌안 및 우안에 대해 개별적으로 계산되어야 한다는 의미합니다.

두 눈이 동일한 반사에서 표시될 경우 사용자가 반사에 입체감이 없음을 금세 알아차립니다. 이는 사용자의 예상과 어긋나며 몰입감을 방해해 VR 경험의 품질에 악영향을 미칩니다.

이 문제를 해결하려면 2개의 반사를 계산하여 게임에서 사용자가 시선을 돌릴 때 각 눈의 위치에 대해 정확한 조정을 적용하여 표시해야 합니다.

얼음 동굴 데모는 이러한 반사를 구현하기 위해 동적 객체로부터의 평면 반사의 경우 2개의 반사 텍스처를 사용하며, 정적 객체 반사의 경우 2개의 로컬 수정 반사 벡터를 사용하여 단일 로컬 큐브맵으로부터 텍스처를 가져옵니다.

반사는 동적 또는 정적 객체일 수 있습니다. 각 반사 유형이 VR에서 작동하기 위해서는 서로 다른 변경이 필요합니다.

#### Unity VR에서 입체 평면 반사 구현

VR 게임에서 입체 평면 반사를 구혀하려면 비 VR 게임 코드에서 몇몇 조정이 필요합니다.

시작하기 전에 Unity에서 가상현실 지원을 활성화했는지 확인하십시오. 이렇게 하려면 Build Settings > Player Settings > Other Settings를 선택하고 Virtual Reality Supported 확인란을 선택합니다.

### 동적 입체 평면 반사

동적 반사에서는 두 눈에 올바른 결과를 생성하기 위해 약간의 변경이 필요합니다.

2대의 카메라, 그리고 각 카메라가 렌더링할 대상 텍스처가 있어야 합니다. 각 렌더링이 프로그램에 의해 실행되도록 양쪽 카메라를 비활성화합니다. 그런 다음 뒤이은 스크립트를 양쪽 카메라에 연결합니다.

```
void OnPreRender(){
        SetUpReflectionCamera();
        // Invert winding
        GL.invertCulling = true;
}
void OnPostRender(){
        // Restore winding
        GL.invertCulling = false;
}
```

이 스크립트는 메인 카메라의 위치 및 방향을 사용하여 반사 카메라의 위치 및 방향을 설정합니다. 이를 위해 스크립트는 좌측 및 우측 반사 카메라가 렌더링하기 직전에 `SetUpReflectionCamera()` 함수를 호출합니다. 다음 코드는 이 함수가 어떻게 구현되는지 보여줍니다.

```
public  GameObject reflCam;
public  float clipPlaneOffset ;
…
private void SetUpReflectionCamera(){
        // Find out the reflection plane: position and normal in world space
        Vector3 pos = gameObject.transform.position;

        // Reflection plane normal in the direction of Y axis
        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);

        // Update reflection camera considering main camera position and orientation
        // Set view matrix
        Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;
        reflCam.GetComponent<Camera>().worldToCameraMatrix = m;

        // Set projection matrix
        reflCam.GetComponent<Camera>().projectionMatrix = Camera.main.projectionMatrix;
}
```

이 함수는 반사 카메라의 뷰 및 투영 매트릭스를 계산합니다. 또한 메인 카메라의 뷰 매트릭스에 적용할 반사 변환 `worldToCameraMatrix`를 결정합니다.

각 눈에 대해 카메라의 위치를 설정하려면 `Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;` 행 다음에 다음 코드를 추가합니다.

좌안의 경우

```
m[12] += stereoSeparation;
```

우안의 경우

```
m[12] -= stereoSeparation;
```

시프트 값 `stereoSeparation`은 0.011입니다. `stereoSeparation` 값은 양안 분리 값의 절반입니다.

다른 스크립트를 메인 카메라에 연결하여 좌측 및 우측 반사 카메라의 렌더링을 제어합니다. 다음 코드는 이 스크립트의 얼음 동굴 구현을 보여줍니다.

```
public class RenderStereoReflections : MonoBehaviour
{
    public GameObject reflectiveObj;
    public GameObject leftReflCamera;
    public GameObject rightReflCamera;
    int eyeIndex = 0;
```

```
    void OnPreRender(){
        if (eyeIndex == 0){
            // Render Left camera
            leftReflCamera.GetComponent<Camera>().Render();
            reflectiveObj.GetComponent<Renderer>().material.SetTexture(
                "_DynReflTex", leftReflCamera.GetComponent<Camera>().targetTexture);
        }
        else{
            // Render right camera
            rightReflCamera.GetComponent<Camera>().Render();
            reflectiveObj.GetComponent<Renderer>().material.SetTexture(
                "_DynReflTex", rightReflCamera.GetComponent<Camera>().targetTexture);
        }
        eyeIndex = 1 - eyeIndex;
    }
}
```

이 스크립트는 메인 카메라의 OnPreRender() 콜백 함수에서 좌측 및 우측 반사 카메라의 렌더링을 처리합니다. 이 스크립트는 좌안에 대해 한 번 호출된 다음 우안에 대해 한 번 호출됩니다. eyeIndex 변수는 각 반사 카메라에 대해 올바른 렌더링 순서를 할당하고 메인 카메라의 각 눈에 올바른 반사를 적용합니다. 콜백 함수는 처음 호출될 때 좌안에 대해 호출된 것으로 가정합니다. 이는 Unity가 OnPreRender() 메서드를 호출하는 순서입니다.

### 각 눈에 서로 다른 텍스처가 사용되는지 확인

스크립트가 각 눈에 서로 다른 렌더 텍스처를 올바로 생성하는지 확인하는 것이 중요합니다.

각 눈에 올바른 텍스처가 표시되는지 확인하는 방법:

#### 절차

1. 스크립트를 쉐이더에 eyeIndex 값을 uniform 변수로 전달하도록 변경합니다.
2. 각 eyeIndex 값에 하나씩, 2개의 반사 텍스처 색상을 사용합니다.

스크립트가 올바로 작동할 경우, 출력은 다음 그림과 같이 두 가지 안정적 반사가 보이는 스크린샷과 비슷합니다.

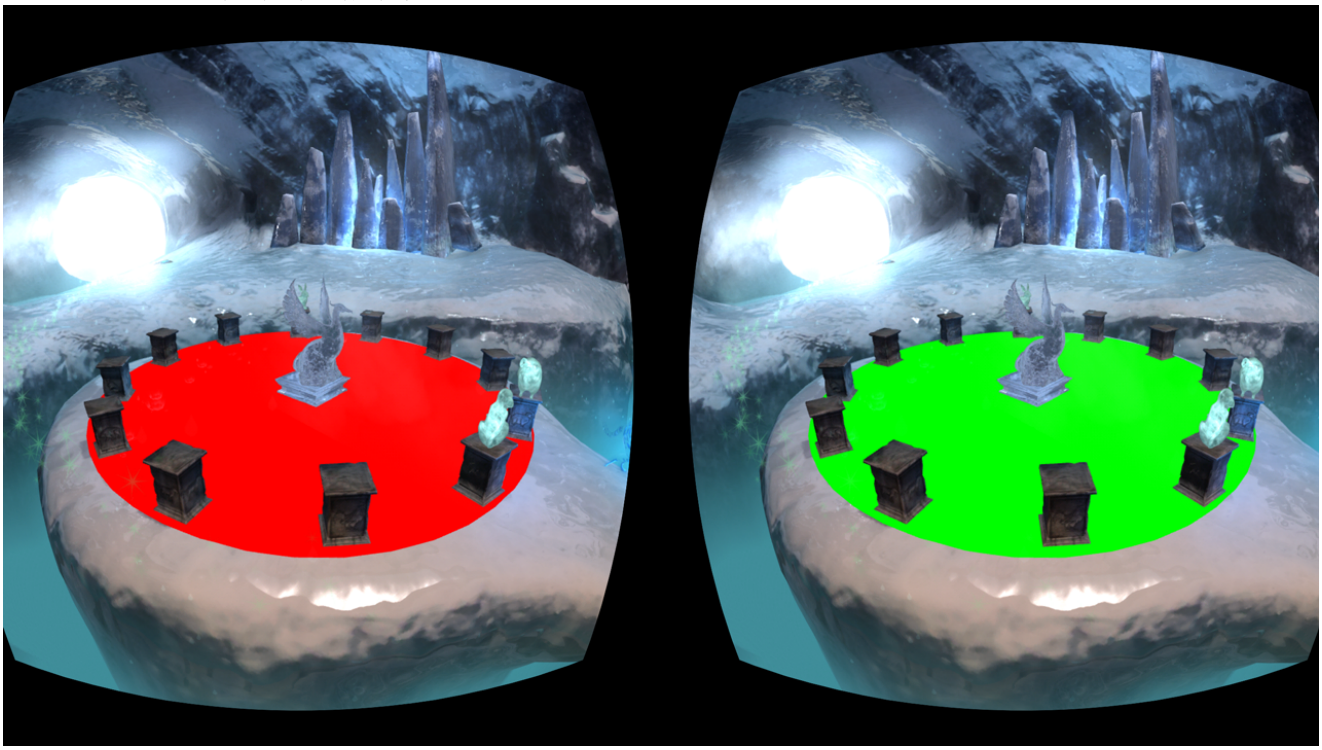

그림 10-6 올바른 반사 텍스처 출력 확인 예

### 정적 입체 평면 반사

큐브맵을 사용하여 정적 객체로부터 입체 반사를 효율적으로 생성할 수 있습니다. 유일한 차이는 큐브맵에서 텍셀을 가져오기 위해 각 눈마다 하나씩 두 개의 반사 벡터를 사용해야 한다는 점입니다.

Unity는 세계 좌표로 카메라 위치에 액세스하기 위한 기본 값을 쉐이더에서 제공합니다.

`_WorldSpaceCameraPos`.

하지만 VR에서는 좌측 및 우측 카메라의 위치가 필요합니다. `_WorldSpaceCameraPos`는 좌측 및 우측 카메라의 위치를 제공하지 못합니다. 따라서 스크립트를 사용하여 좌측 및 우측 카메라의 위치를 계산하고 그 결과를 단일 uniform 변수로 쉐이더에 전달해야 합니다.

쉐이더에서 카메라 위치 정보를 전달할 수 있는 새 uniform 변수를 선언합니다.

```
uniform float3 _StereoCamPosWorld;
```

좌측 및 우측 카메라 위치를 계산하기 위한 최상의 위치는 메인 카메라에 연결된 스크립트입니다. 메인 카메라 뷰 매트릭스에 쉽게 액세스할 수 있기 때문입니다. 다음 코드는 `eyeIndex = 0` 사례에서 이렇게 하는 방법입니다.

이 코드는 메인 카메라의 뷰 매트릭스를 수정하여 로컬 좌표로 좌안 위치를 설정하도록 수정합니다. 좌안 위치는 역 매트릭스가 구해지도록 세계 좌표로 필요합니다. 좌안 카메라 위치는 uniform `_StereoCamPosWorld`를 통해 쉐이더로 전달됩니다.

```
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));
```

코드는 우안도 동일하지만 입체 분리가 `mWorldToCamera[12]`에 추가되는 것이 아니라 차감되는 점이 다릅니다.

정점 쉐이더에는 다음과 같이 뷰 벡터를 계산하는 행이 있습니다.

```
output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
```

이 행을 세계 좌표의 새 좌안 및 우안 카메라 위치를 사용하는 행으로 대체합니다.

```
output.viewDirInWorld = vertexWorld.xyz - _ StereoCamPosWorld;
```

입체 반사가 구현되면 편집기 모드에서 응용 프로그램을 실행할 때 볼 수 있습니다. 좌안에서 우안으로 반복적으로 전환될 때 반사 텍스처가 깜박이기 때문입니다. 각 눈에 다른 텍스처가 사용되므로 VR 디바이스에서는 이 깜박임이 보이지 않습니다.

### 입체 반사 최적화

추가 최적화 없이는 입체 반사 구현이 항상 실행됩니다. 즉, 반사가 보이지 않을 때는 반사에 프로세싱 시간이 낭비되는 것입니다.

반사 자체에 대한 작업을 수행하기 전에 반사 표면이 보이는지 여부를 확인하는 코드를 삽입하십시오. 이렇게 하려면 반사성 객체에 다음 코드 예제와 유사한 코드를 연결합니다.

```
public class IsReflectiveObjectVisible : MonoBehaviour
{
        public bool reflObjIsVisible;

        void Start(){
                reflObjIsVisible = false;
        }

        void OnBecameVisible(){
                reflObjIsVisible = true;
        }

        void OnBecameInvisible(){
                reflObjIsVisible = false;
```

```
        }
}
```

이 클래스를 정의한 후, 반사성 객체가 보일 때만 입체 반사 계산이 실행되도록 메인 카메라에 연결된 스크립트에 다음 `if` 문을 사용합니다.

```
void OnPreRender(){
        if (reflectiveObjetc.GetComponent<IsReflectiveObjectVisible>().reflObjIsVisible){
        …
        }
}
```

코드의 나머지 부분은 이 `if` 문 안에 들어갑니다. 이 `if` 문은 클래스 `IsReflectiveObjectVisible`을 사용하여 반사성 객체가 보이는지 여부를 확인합니다. 반사성 객체가 보이지 않으면 반사가 계산되지 않습니다.

## 10.5 결과

이 작업은 몰입감을 높여주는 입체 반사를 구현하는 VR 버전의 게임을 생성하며, 이는 전체 VR 사용자 경험을 개선합니다. 입체 반사로 전환할 경우 사용자 경험이 현저히 개선됩니다. 입체 반사가 구현되지 않으면 사람들이 반사에서 입체감이 빠진 것을 알아차리기 때문입니다.

다음 그림은 개발자 모드에서 실행되는 얼음 동굴 데모에서 구현된 입체 반사를 보여주는 예시 스크린입니다.

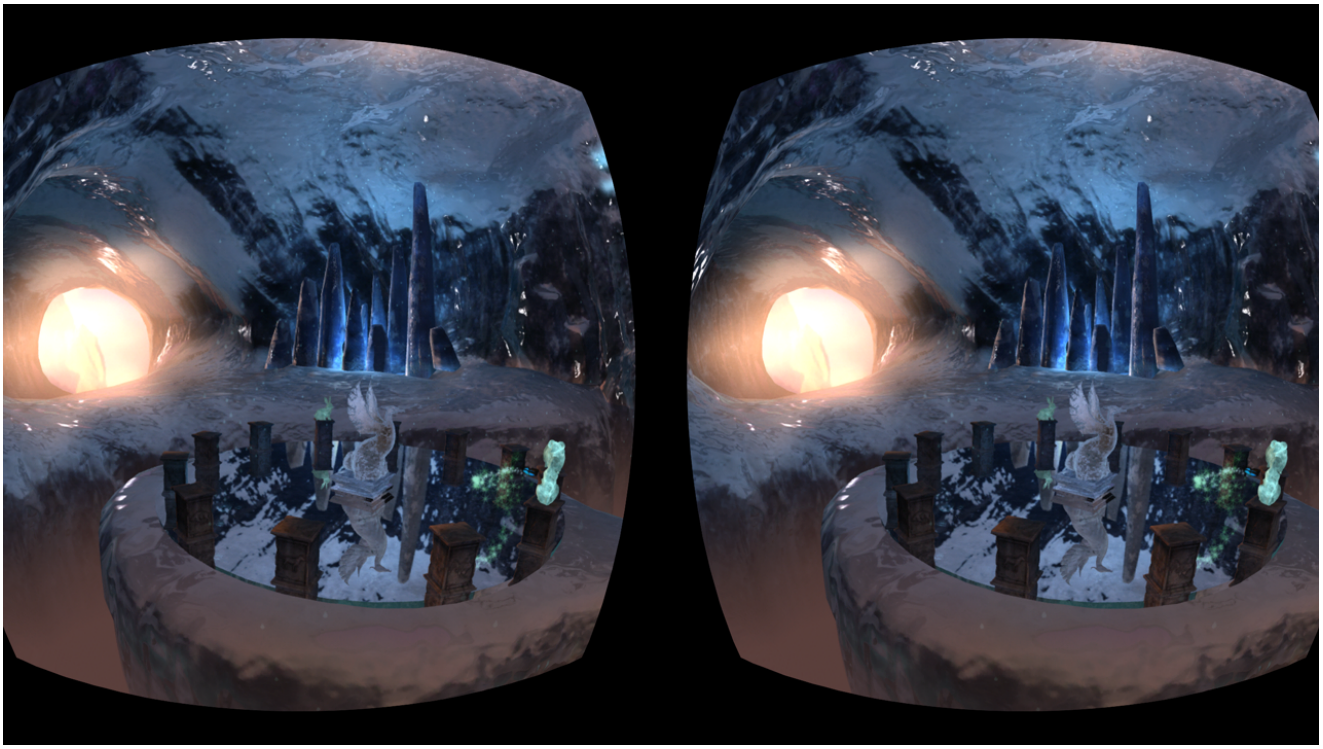

그림 10-7 얼음 동굴 데모의 스크린샷

# 제 11 장
# 고급 VR 그래픽 기법

이 장은 가상 현실 응용 프로그램의 그래픽 성능을 향상시키는 데 사용할 수 있는 다양한 기술을 설명합니다.

———— 주의 ————

이 콘텐츠의 최신 버전은 Arm 개발자 웹 사이트 *https://developer.arm.com/solutions/graphics-and-gaming/gaming-engine/unity/arm-guide-for-unity-developers/advanced-vr-graphics-techniques*에서 확인할 수 있습니다.

———————————

이 장은 다음 단원으로 구성되어 있습니다.

## 11.1 에일리어싱

에일리어싱은 실용성을 위해 오브젝트를 정확하게 재현할 만큼 충분한 샘플을 사용하지 않고 정보를 파악할 때 발생하며 무시할 수 없는 신호 처리 양상입니다.

오디오와 비디오의 원본 소리나 색상 데이터를 별개의 디지털 신호로 변환할 때 에일리어싱이 발생합니다. 에일리어싱은 고주파수 오디오나 비디오 디테일이 자세한 경우에 더욱 분명히 드러납니다.

에일리어싱은 원본 데이터를 드물게 샘플링하는 경우에 발생합니다. 콘텐츠에서 가장 높은 주파수보다 속도가 절반 미만일 때 발생합니다. 그래픽의 경우, 디스플레이를 형성하는 픽셀 그리드에 맞지 않게 움직이는 카메라에서 오브젝트를 표시하려고 래스터라이제이션 프로세스를 수행할 때 자주 보이는 현상입니다.

그래픽 래스터라이제이션 프로세스는 디스플레이를 형성하는 픽셀 그리드에 맞지 않게 이동하는 카메라에서 오브젝트를 표시하는 직접적인 결과로도 에일리어싱이 발생합니다.

VR 환경에서는 지오메트릭 에일리어싱과 정반사 에일리어싱 2가지가 특히 두드러집니다.

지오메트릭 에일리어싱은 장면에 상대적으로 낮은 주파수의 픽셀 샘플에 표시되는 컬러 스페이스에 고주파수 입력 신호를 포함하는 경우, 예를 들어 두 개의 대조적인 색이 빠르게 전환할 때 흔히 보입니다.

특히 카메라가 움직일 때 직선이 기어오르는 것 같은 에일리어싱을 볼 수 있습니다 이런 지오메트릭 에일리어싱은 다음 이미지에 표시된 형태의 붉은 에지에서 볼 수 있습니다.

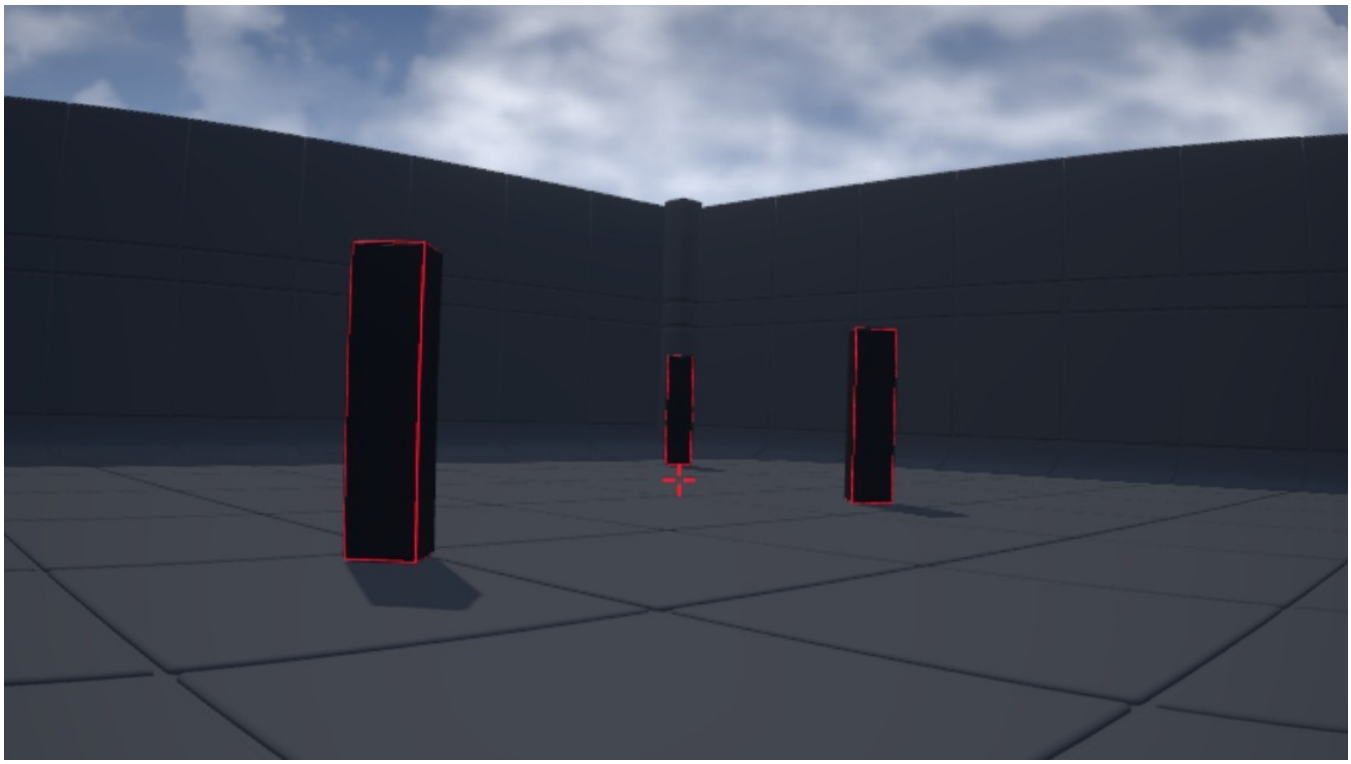

그림 11-1 지오메트릭 에일리어싱 예

지오메트릭 에일리어싱은 폴리곤 수가 많은 메시가 저해상도 디스플레이에 렌더링되어 폴리곤 수로 인해 자세한 디테일이 적은 픽셀 범위에 표시되는 경우에도 발생할 수 있습니다. 폴리곤 수가 많으면 에지가 많으므로 이를 렌더링할 픽셀 수가 적으면 에일리어싱이 더 많이 발생합니다.

또한 삼각형 밀도가 높으면 GPU가 최종 화질에 큰 차이를 주지 않는 너무 작은 지오메트리에 많은 사이클을 소모하여 GPU에서 래스터화가 충분하지 않습니다. 또한 GPU가 지오메트리 래스터화에 너무 많은 사이클을 사용하면 프래그먼트 셰이더를 실행할 사이클이 적어집니다.

GPU가 프래그먼트 셰이더 실행보다 더 많은 사이클을 지오메트리를 래스터화 하는 프로세스에 소모해야 하기 때문입니다.

정반사 에일리어싱은 카메라나 오브젝트가 이동하여 갑자기 '튀어나오'거나 사라지는 뾰족한 하이라이트가 있는 오브젝트에서 발생합니다. 정반사 효과가 한 프레임에는 등장했다가 다음 프레임에서는 사라지기 때문에 프레임에서 픽셀이 반짝이는 것처럼 보입니다. 이러한 시머링 효과는 사용자의 시선을 끌어 번거롭게 하기 때문에 몰입도에 영향을 주며 사용자 경험의 질도 감소합니다. 아래 이미지에서 벽의 솟은 부분은 정반사 에일리어싱을 보여줍니다.

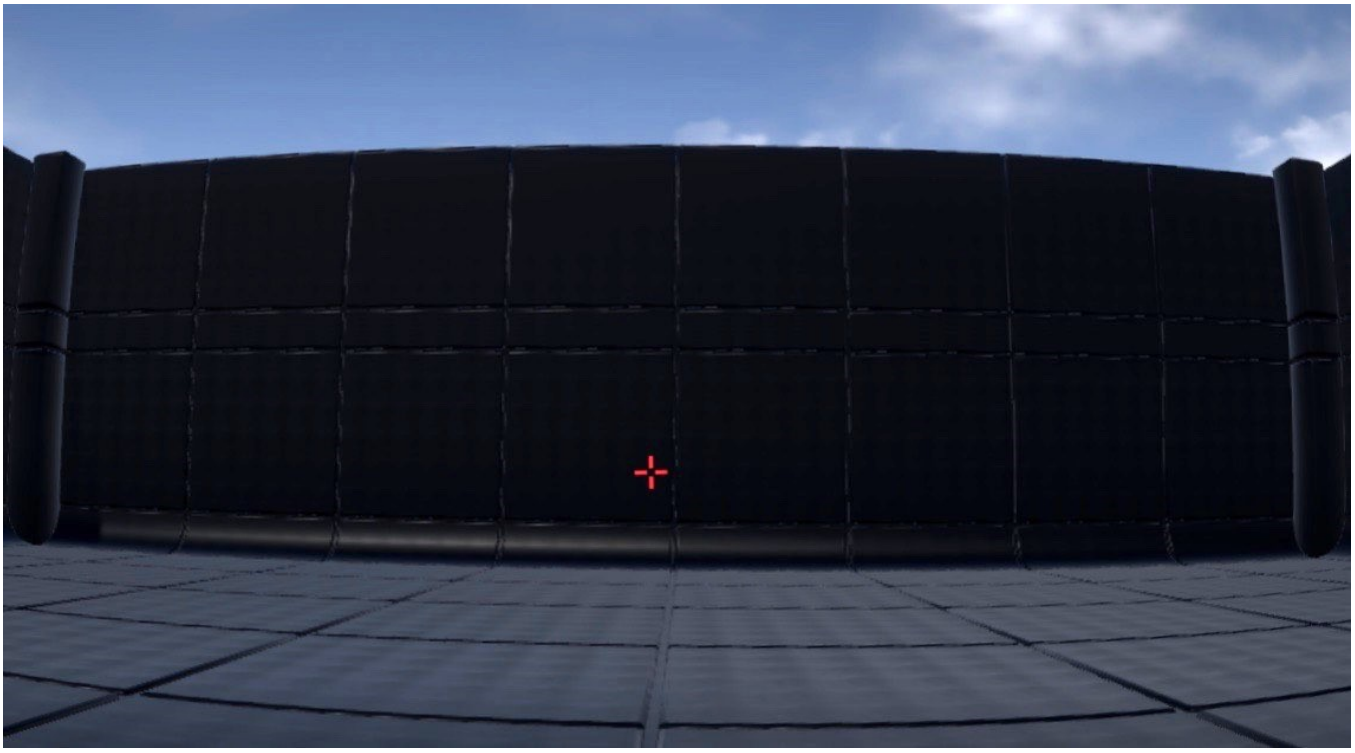

그림 11-2 정반사 에일리어싱 예

슈퍼샘플링 같은 일반적인 안티앨리어싱 테크닉은 컴퓨팅 복잡성 때문에 모바일에는 적합하지 않으며, 효과적이며 성능도 우수한 *Multi-Sampling Anti-Aliasing*(MSAA)은 지오메트릭 에일리어싱에만 적용되며 정반사 에일리어싱 같은 셰이더 내에서 발생하는 언더샘플링 아티팩트에는 효과적이지 않습니다.

컴퓨터 그래픽, 특히 VR에서 발생하는 에일리어싱은 종류가 다양해 완화하는 데 여러 테크닉이 필요합니다.

## 11.2 멀티 샘플 안티앨리어싱

*멀티 샘플 안티앨리어싱*(MSAA)는 슈퍼샘플링보다 더욱 효율적인 안티앨리어싱 테크닉입니다. 슈퍼샘플링은 표시 해상도를 다운스케일링하기 전에 고해상도에서 모든 픽셀에 조각 셰이딩을 수행하여 높은 내부 해상도에서 이미지를 렌더링합니다.

MSAA는 보통 정점 셰이딩을 수행하지만 각 픽셀은 하위 샘플 비트 단위 적용 마스크를 사용하여 테스트된 하위 샘플로 나눠집니다. 하위 샘플이 이 적용 테스트를 통과하면 조각 셰이딩을 수행하여 적용 테스트를 통과한 하위 샘플마다 조각 셰이더의 결과가 저장됩니다.

MSAA는 픽셀마다 조각 셰이더를 한 번만 실행하므로 슈퍼샘플링보다 훨씬 효율적이지만 이 테크닉은 두 삼각형이 교차하는 지점의 지오메트릭 별칭만 완화합니다.

VR 응용 프로그램의 경우 MSAA 4회 사용 시 삼각형 모서리의 '뾰족함'을 줄이면서 비용 대비 품질을 개선하므로 가능한 경우 언제든 사용해야 합니다.

Mali GPU는 MSAA 4회 사용 시 전체 조각 처리량을 감당할 수 있도록 설계되었으므로 이를 사용하면 삼각형 모서리를 따라 생성되는 추가 조각으로 인한 성능 저하가 거의 없습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *11.2.1 Unity에서 Multisample Anti-Aliasing 구현* 페이지의 11-255.

### 11.2.1 Unity에서 Multisample Anti-Aliasing 구현

Unity에서 *Multisample Anti-Aliasing*(MSAA)은 *Universal Render Pipeline*(URP)를 사용할 경우 URP 설정에서 설정합니다. 그렇지 않으면 Project Quality Settings 패널에서 설정합니다.

URP 패널을 사용해 MSAA를 사용하는 방법은 다음과 같습니다.

1. Assets 창으로 이동합니다.
2. Inspector 창에서 품질 드롭다운 메뉴로 이동합니다.
3. 드롭다운 메뉴에서 필요한 MSAA 레벨을 선택합니다.

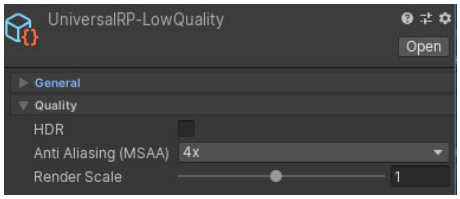


그림 11-3 URP MSAA 설정 패널

Project Quality Settings 패널을 사용해 MSAA를 사용하는 방법은 다음과 같습니다.

1. Edit에서 Project Settings으로 이동합니다.
2. Quality을 선택하고 Project Quality Settings 패널을 엽니다.
3. 이제 Anti Aliasing 드롭다운 메뉴를 선택하고 적절한 설정을 선택합니다. 가능한 경우 4x Multi Sampling을권장합니다.

———— 참고 ————

모든 품질 수준에 대해 안티앨리어싱 값을 설정했는지 확인하십시오.

————————

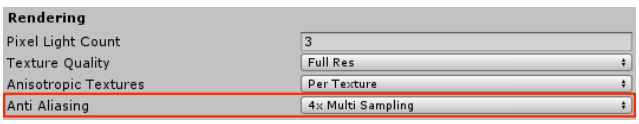


그림 11-4 MSAA 설정

## 11.3 밉매핑

밉매핑은 고해상도 텍스처를 다운스케일하고 필터링하여 하위 밉 레벨을 이전 레벨 영역의 1/4로 축소하는 테크닉입니다. 이것은 텍스처와 생성된 모든 밉이 원래 텍스처 크기의 1.5배 이하라는 의미입니다.

밉맵은 아티스트가 직접 생성하거나 컴퓨터가 생성할 수 있으며 GPU에 업로드됩니다. GPU는 샘플링을 수행하기에 가장 적합한 밉을 선택합니다. 작은 밉 레벨에서 샘플링을 하면 텍스처 에일리어싱을 최소화하며 표면 텍스처의 정의를 유지하고 거리가 먼 표면에도 모아레 패턴이 형성되는 것을 막아줍니다.

그림 11-5 9개의 밉 레벨(1x1 해상도까지)을 갖고 있는 512x512 텍스처

단일 텍스처 아틀라스 내에 밉맵 레벨을 과도하게 많이 저장하면 포비티드 렌더링이 사용될 때 시각화 문제가 발생할 수 있으니 주의하십시오.

포비티드 렌더링은 사용자가 현재 보고 있는 곳에 고품질 그래픽을 표시하는 아이트래킹을 사용합니다. 시야 밖의 이미지 품질은 크게 감소할 수 있습니다.

하나의 타일을 기본 해상도로 렌더링할 때 문제가 발생하지만 인접한 타일은 1/4 해상도로 렌더링됩니다. 이 경우 이웃한 타일은 2레벨 낮은 밉맵에서 텍스처를 샘플링한 것입니다.

샘플링 중에도 결과 문제가 발생합니다. 예를 들어 텍셀이 텍스처 필터링 중 이웃 텍셀과 함께 블러처리되면 텍스처 아틀라스의 이웃 레벨에서 잘못된 색상 누출이 발생할 수 있습니다.

그 결과 두 개의 다른 포비티드 지역에 있는 두 개의 이웃한 개별 타일이 색상 차이를 보일 수 있습니다. 텍스처의 에지를 확장해 아틀라스의 엔트리 사이를 분리하는 갭을 만들면 이 문제를 해결할 수 있습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 11.3.1 Unity에서 밉맵 구현

밉맵 자동 생성은 Inspector 창에서 설정할 수 있습니다.

밉맵 생성을 사용하려면 다음과 같습니다.

1. 프로젝트 창의 Asset 섹션에서 텍스처를 선택하고 Texture Inspector 창을 엽니다.
2. Generate Mip Maps 옵션을 켭니다.

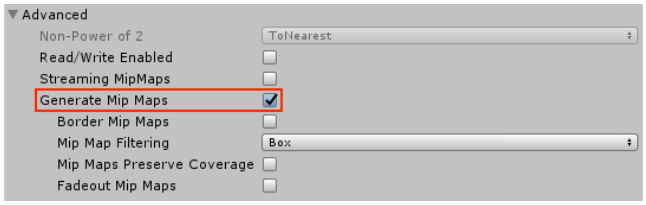


그림 11-6 밉맵 생성 설정

## 11.4 Level of Detail(LOD)

*Level of Detail*(LOD)은 오브젝트와 뷰어 간 거리가 증가할수록 오브젝트의 디테일과 복잡성을 줄이는 테크닉입니다.

LOD를 사용하면 뷰어가 오브젝트와 가까울 때 오브젝트의 지오메트리 디테일 수준이 높습니다. 따라서 오브젝트가 자세하고 정확하게 표시됩니다. 오브젝트가 화면에서 멀어질수록 자동으로 낮은 디테일 모델로 전환하여 플레이어에게 디테일이 적은 모델을 표시합니다.

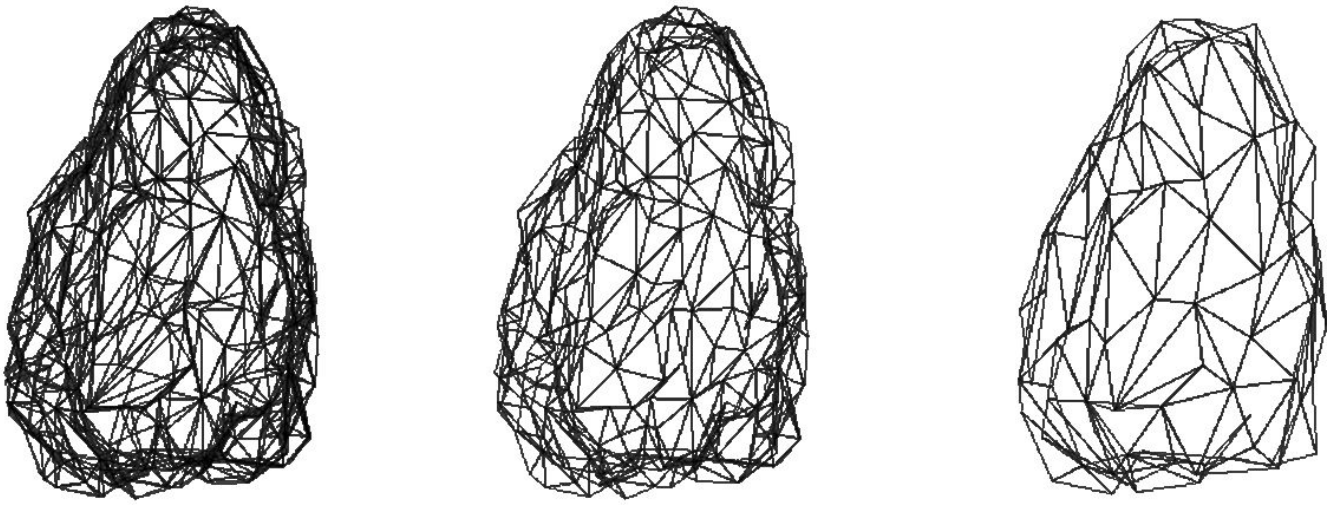

그림 11-7 단일 메시의 3개 LOD

모델을 디테일 레벨이 낮은 모델로 전환하면 다음과 같은 두 가지 장점이 있습니다.

- 지오메트릭 에일리어싱 발생 확률이 감소하여 아티팩트 레벨이 감소합니다.
- 셰이딩이 필요한 정점 수가 감소하여 성능이 향상됩니다.

지오메트리가 작은 경우 불필요한 오버샘플링을 방지하기 위해 각 LOD 레벨 간 정점 수가 크게 차이 나는 LOD를 생성해야 합니다.

LOD에 대한 자세한 내용은 *지오메트리 모범 사례 장* 페이지의 5-83의 LOD 섹션을 참조하십시오.

### LOD를 보여주는 시각화 예

다음 그림과 비디오는 오브젝트가 카메라에서 가까워지거나 멀어질 때 LOD가 어떻게 작동하는지 예시를 보여줍니다.

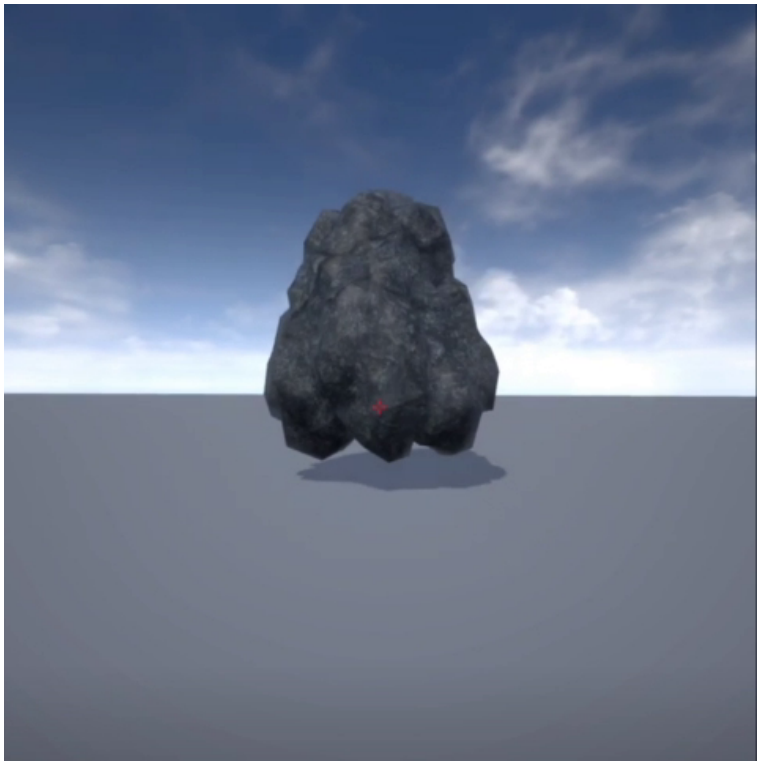

그림 11-8 최종 LOD 렌더링 예

다음 웹 주소 링크의 비디오는 카메라가 오브젝트와 가까워질 때 메시가 3개의 개별 LOD를 통과하는 경우를 보여줍니다. 이 비디오는 카메라가 오브젝트와 가까워질 때 보이고 멀어지면 보이지 않는 섬세한 디테일을 보여줍니다. 그러나 가까이에서 보이는 섬세한 디테일은 오브젝트의 사실성을 높여줍니다.

*https://developer.arm.com/graphics/videos/level-of-detail-example*

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *11.4.1 Unity에서 LOD 구현* 페이지의 11-259.

### 11.4.1 Unity에서 LOD 구현

Unity의 구성요소 메뉴 옵션에서 *LOD*를 구현합니다.

모델에서 LOD를 사용하려면 다음과 같습니다.

1. 필요한 모델을 선택합니다.
2. Object Inspector Panel에서 Component로 이동합니다.
3. Rendering을 선택합니다.
4. LOD Group을 선택합니다.

그림 11-9 오브젝트에 LOD Group 사용

LOD Group을 선택하면 Object Inspector Panel에서 LOD Group 창을 표시합니다. 다른 LOD 레벨을 선택하면 Renderers panel이 노출되는데 이곳에서 필요한 복잡성 수준에 맞는 메시를 선택할 수 있습니다. 사용 가능한 LOD 레벨은 응용 프로그램에서 LOD 간 이동하기 원하는 시점에 따라 크기를 조절할 수 있습니다.

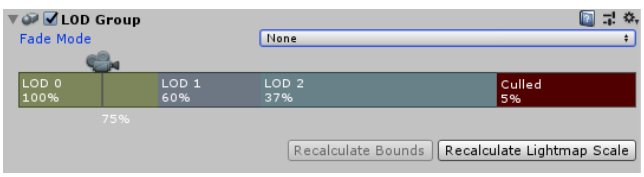


그림 11-10 LOD 설정

필요하면 Fade Mode를 설정하거나 Cross Fade 또는 SpeedTree를 설정할 수 있습니다. 오브젝트의 응용에 따라 Fade Mode는 셰이더에서 액세스할 수 있는 블렌드 요소를 노출합니다. 블렌드 요소를 사용해 여러 LOD를 부드럽게 블렌딩할 수 있습니다.

Unity는 `Shader.globalMaximumLOD`나 `Shader.maximumLOD`의 현재 값에 따라 적용되는 여러 셰이더나 셰이더 패스를 사용하는 셰이더 LOD를 지원합니다.

셰이더 LOD를 사용하면 더 효율적인 셰이더를 적용하여 셰이더의 하드웨어 사용 비용이 너무 높지 않도록 할 수 있습니다.

## 11.5 컬러 스페이스

컬러 스페이스는 각 컬러가 할당되는 숫자 값을 지시합니다. 컬러 스페이스는 공간에서 각 색상이 할당되는 영역과 해당 색상 내에서 변화 정도를 정의합니다.

원래 렌더링은 감마 컬러 스페이스에서 수행되었습니다. 모니터는 감마 컬러 스페이스 이미지를 표시하므로 그렇지 않은 경우는 모니터에 표시되기 전 최종 이미지에 감마 보정이 필요했습니다.

*물리 기반 렌더링*(PBR)의 출현으로 리니어 컬러 스페이스의 렌더링으로 전환되었습니다. 이렇게 하면 다양한 광원 값을 셰이더 내에 쉽고 정확하게 누산할 수 있습니다. 감마 컬러 스페이스의 경우 감마 컬러 스페이스 렌더링을 위한 커브 문제로 이 합계가 물리적으로 정확하지 않았습니다.

리니어 컬러 스페이스 렌더링은 반사 별칭을 줄이는 데 훨씬 좋습니다. 이 렌더링 방식이 더 유리한 이유는 감마 컬러 스페이스 렌더링의 경우 장면에서 밝기를 증가시키면 객체를 빠르게 흰색으로 만들어 반사 별칭 효과가 발생하기 때문입니다. 리니어 컬러 스페이스에서는 객체가 점진적으로 밝아지기 때문에 객체가 급격하게 하얀색으로 변하지 않으므로 반사 별칭 발생 위험이 감소합니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *11.5.1 Unity에서 컬러 스페이스를 선택하는 방법* 페이지의 11-261.

### 11.5.1 Unity에서 컬러 스페이스를 선택하는 방법

Unity는 Project Player Settings에서 컬러 스페이스를 선택합니다.

Linear 컬러 스페이스를 사용하는 방법은 다음과 같습니다.

1. Edit에서 Project Settings으로 이동합니다.
2. Player를 선택하고 Project Player Settings 패널을 엽니다.
3. Player Settings, Other Settings 내에 있는 Color Space 드롭다운 옵션을 엽니다.
4. Rendering 섹션의 드롭다운 메뉴에서 Linear 옵션을 선택합니다.

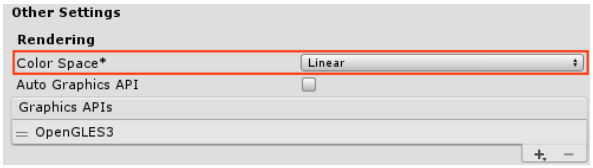


그림 11-11 컬러 스페이스 설정

참고

응용 프로그램을 Linear 컬러 스페이스로 설정하는 경우 Graphics API가 OpenGLES 3.0으로 설정되어야 합니다. 올바른 API를 선택하려면 Auto Graphics API 옵션 선택을 해제하고 명명된 그래픽 API 목록에서 OpenGLES2를 제거합니다. Identification 섹션에서 Minimum API Level 설정은 최소 API 레벨 18인 Android 4.3 'Jelly Bean' 이상으로 설정되어야 합니다.

## 11.6 텍스처 필터링

텍스처 필터링은 텍스처에서 샘플링할 때 발생하는 에일리어싱을 줄이기 위해 사용되는 테크닉입니다.

에일리어싱은 화면에서 렌더링하는 픽셀이 렌더링하는 오브젝트에 매핑되는 텍스처 내의 픽셀 그리드에 정확하게 맞지 않는 경우 발생합니다. 그 대신 텍스처가 매핑될 오브젝트가 뷰어와 임의의 거리 및 방향에 있으므로 약간 보정됩니다.

텍스처 픽셀 또는 텍셀을 스크린 픽셀로 매핑할 때 발생하는 두 가지 경우가 있는데, 두 경우 모두 텍셀이 스크린 픽셀보다 큰 경우 스크린 픽셀에 맞추기 위해 텍셀이 축소되어야 합니다. 한편 텍셀이 스크린 픽셀보다 작은 경우 스크린 픽셀에 맞추기 위해 여러 텍셀이 결합되어야 합니다.

텍스처 필터링은 밉맵에 크게 의존합니다. 확대(Magnification) 시 가져오는 텍셀 수는 4개를 초과하지 않기 때문입니다. 그러나 축소(Minification) 시 텍스처가 적용된 오브젝트가 멀어지면 전체 텍스처가 하나의 픽셀에 맞게 됩니다. 이 경우 정확한 결과를 위해 모든 텍셀을 가져와 병합하는 것은 GPU 구현 비용이 매우 비쌉니다.

그래서 4개의 샘플을 선택하는데 이때 데이터에 예상하지 못한 언더 샘플링 현상이 발생할 수 있습니다. 밉맵은 이것을 다른 크기의 텍스처를 사전 필터링함으로써 오브젝트가 멀어지면 작은 크기의 텍스처가 적용되게 하였습니다.

Bilinear, Trilinear, Anisotropic 필터링 같은 자주 구현되는 필터링 방식은 샘플링되는 텍셀 수, 결합 방식, 필터링 처리 과정에서 밉맵을 활용하는지 여부에 따라 차이가 있습니다.

각 필터링 방식의 성능 비용은 다릅니다. 따라서 사용할 필터링 유형을 선택할 때는 필터링 방식을 사용하는 성능 비용이 제공되는 시각적 이득과 비교해 적절한지에 따라 개별로 처리해야 합니다.

예를 들어 Trilinear 필터링은 Bilinear 필터링보다 2배의 비용이 들지만 거리가 먼 오브젝트에 적용될 때 텍스처가 항상 시각적 이점이 명백하지는 않습니다. 그 대신 성능과 품질 면에서 더 나은 2x Anisotropic 필터링 사용을 권장합니다.

Anisotropic 필터링을 사용할 때는 최대 샘플 수를 정해야 합니다. 그러나 모바일에서는 성능에 영향을 주기 때문에 샘플 수 8개 이상 많은 샘플을 사용할 때는 특히 주의해야 합니다. 이 테크닉은 먼 거리의 지면 같은 비스듬한 표면 텍스처에 적절합니다.

다음 이미지는 여러 유형의 텍스처 필터링의 차이를 보여줍니다. 특별히 주의해야 할 것은 2x 샘플 Anisotropic 필터링이 사용되었을 경우 Bilinear과 Trilinear 필터링 간 차이가 거의 드러나지 않는다는 것입니다. 그러나 Trilinear 필터링은 더 많은 리소스 비용을 사용합니다.

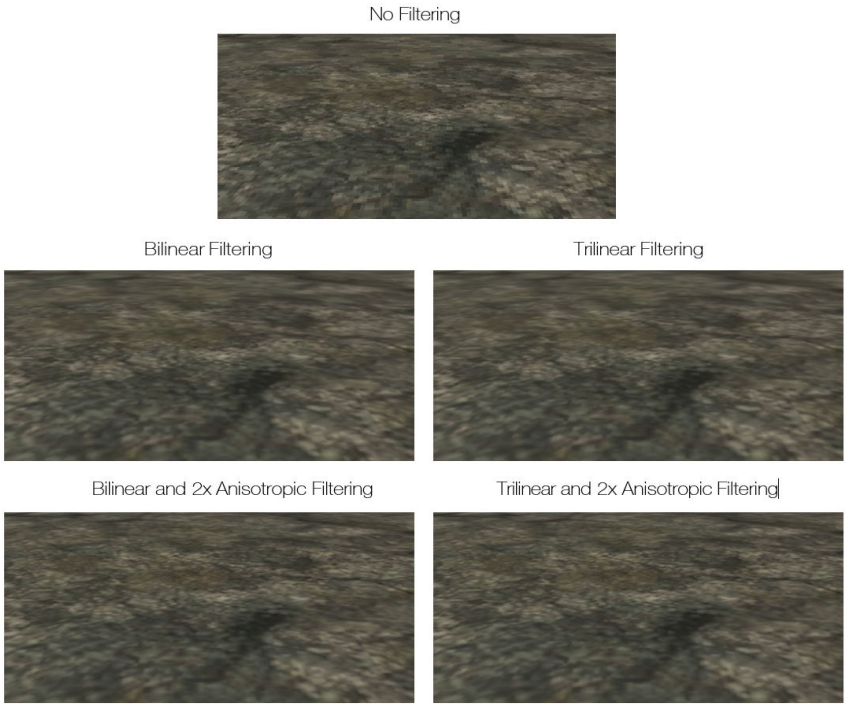


그림 11-12 다양한 텍스처 필터링 방법 비교

텍스처 필터링에 대한 자세한 내용은 *텍스처링 모범 사례 장* 페이지의 6-92을 참조하십시오.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *11.6.1 Unity에서 텍스처 필터링 구현* 페이지의 11-263.

### 11.6.1 Unity에서 텍스처 필터링 구현

Unity에서 텍스처 필터링은 Project Quality Settings으로 설정합니다.

유닛의 텍스처에 텍스처 필터링 설정을 하려면 다음과 같습니다.

1. Edit으로 이동합니다.
2. Project Settings을 선택합니다.
3. Quality을 선택하고 Project Quality Settings 패널을 엽니다.
4. 창에서 Anisotropic Textures 드롭다운을 선택합니다.
5. 마지막으로 Per Texture 옵션을 선택합니다.

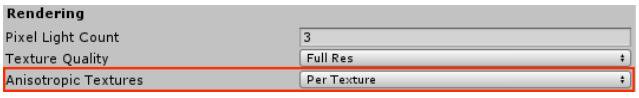


그림 11-13 Per texture로 텍스처 필터링 설정

각 텍스처에서 Project 창의 Assets 섹션에서 선택하여 Texture Inspector 창이 열리도록 합니다. 창이 열리면 텍스처에 Filter Mode와 Aniso Level 설정을 설정합니다.

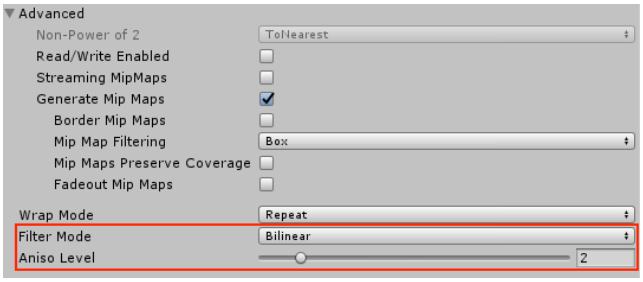


그림 11-14 텍스처에 텍스처 필터링 설정을 선택

## 11.7 알파 합성

알파 합성은 배경 이미지에 이미지를 결합하여 투명도 있는 합성 이미지를 만드는 테크닉입니다.

알파 테스팅은 널리 구현되는 형태의 알파 합성입니다. 그러나 알파 채널이 비트 단위이기 때문에 오브젝트 에지에 심각한 에일리어싱 효과를 발생시킬 수 있고 에지 간 블렌딩이 없습니다.

이 경우 셰이더가 각 픽셀에 한 번만 실행되므로 멀티 샘플링은 아무런 영향을 주지 않습니다. 각 서브 샘플은 에지의 에일리어싱을 그대로 둔 채 동일한 알파 값을 반환합니다.

알파 블렌딩은 이를 대신하는 해결책입니다. 그러나 폴리곤 정렬을 하지 않으면 블렌딩이 실패하여 오브젝트가 제대로 렌더링되지 않습니다. 그러나 정렬을 사용하면 처리 비용이 높으므로 성능이 저하됩니다.

*Alpha to Coverage*(ATOC)은 에일리어싱을 줄이는 또 다른 알파 합성 방법입니다. ATOC는 프래그먼트 셰이더의 알파 구성요소 출력을 커버리지 마스크로 변환하여 이를 멀티 샘플링 마스크와 결합합니다. ATOC는 AND 연산자를 사용해 이 연산을 통과하는 픽셀만 렌더링합니다.

**알파 테스팅과 알파 커버리지가 어떻게 보이는지 시각화된 예**

다음 링크의 이미지와 비디오는 왼쪽의 기본 알파 테스트 구현과 오른쪽의 알파에서 커버리지 구현을 비교해서 보여줍니다.

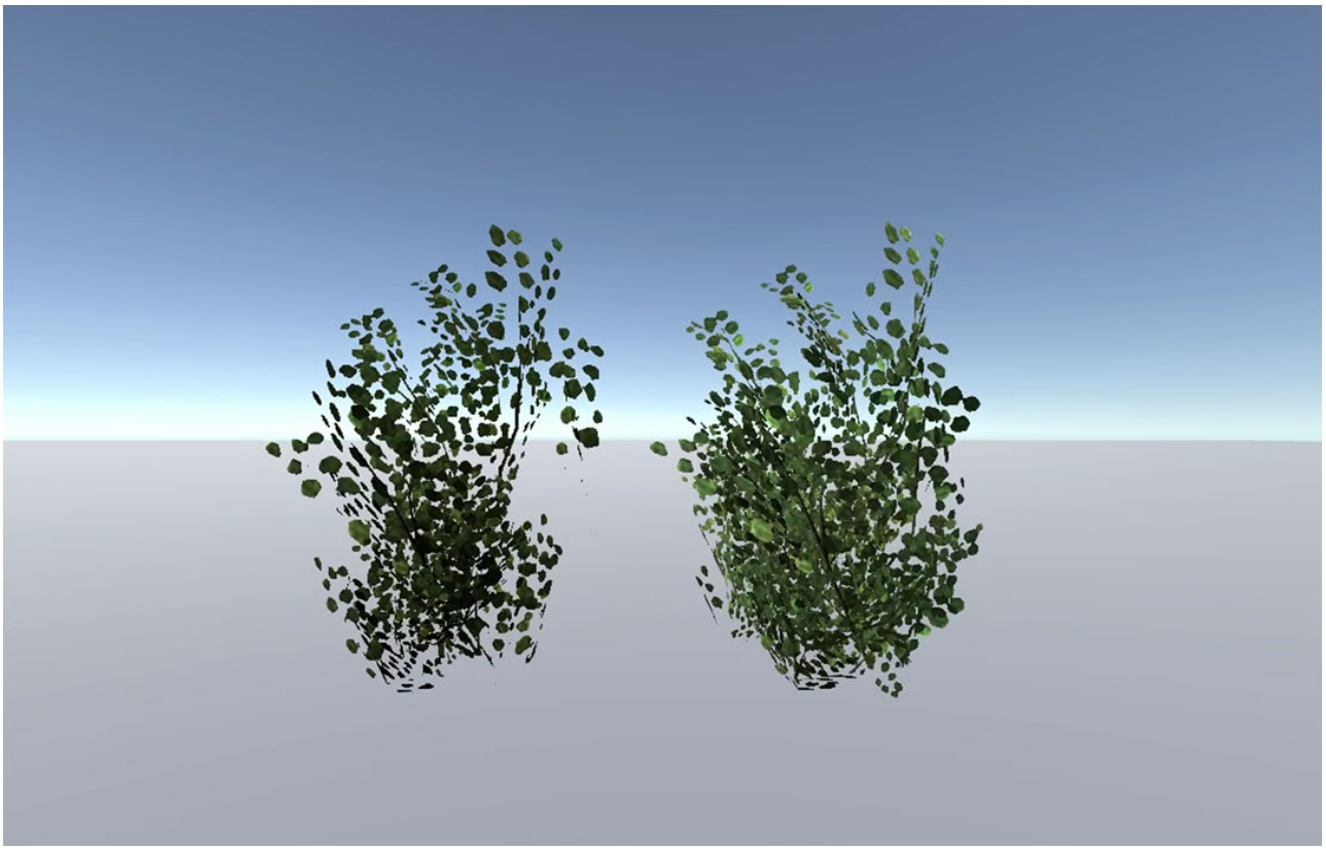

그림 11-15 알파 합성 예

*https://developer.arm.com/graphics/videos/alpha-compositing-example*

ATOC 구현의 경우 에일리어싱이 훨씬 적게 발생하고 깜박임도 줄어듭니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.



### 11.7.1 Unity에서 알파 합성 구현

알파에서 적용은 Unity 셰이더에서 구현됩니다.

#### Mip maps preserve coverage 옵션 - Unity 2017 이후 버전

Mip maps preserve coverage 옵션은 이전 버전과 달리 밉맵 레벨 계산을 요구하지 않습니다. Texture Inspector에서 실행할 수 있습니다.

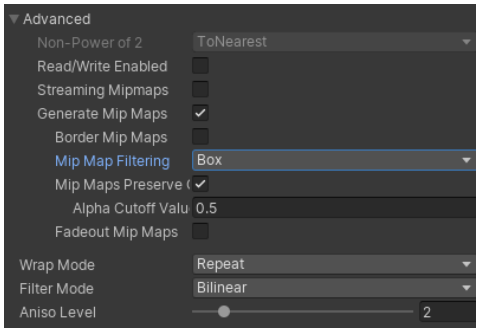


그림 11-16 Unity 2017 이후 버전용 Mip Maps Preserve Coverage 옵션

Unity 2017 이후 버전부터 ATOC를 사용하려면 먼저 프로젝트 창에서 새 셰이더를 생성하고 다음 셰이더 코드를 입력합니다.

———— 참고 ————

이 셰이더를 머티리얼에 적용하고 알파 합성이 필요한 오브젝트에 머티리얼을 설정합니다.

————————————

```
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;
                }
```

### Unity 2017 이전 버전

Unity 2017 이전 버전에서 ATOC를 사용하려면 셰이더 코드에 밉맵 레벨 계산을 추가해야 합니다. Unity 2017 및 이후 버전부터는 먼저 프로젝트 창에서 새 셰이더를 생성해야 합니다. 이후 다음 셰이더 코드를 입력합니다.

———— 참고 ————

이 셰이더를 머티리얼에 적용하고 알파 합성이 필요한 오브젝트에 머티리얼을 설정합니다.

————————————

```
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
          }
     }
}
```

## 11.8 레벨 디자인

에일리어싱을 최소화하는 데 도움이 되는 모든 기술적 옵션이 있지만 공들인 레벨 디자인은 아티팩트 에일리어싱 감소에 여전히 큰 도움이 됩니다. 레벨 디자인 선택이 좋지 않으면 모든 기술적 솔루션의 의미가 없으므로 현명한 레벨 디자인 선택은 에일리어싱 감소에 여전히 중요한 역할을 합니다.

장면에 지오메트리를 생성하는 경우 메시 에지가 날카롭거나 갑작스러운 에지가 발생하지 않도록 모델링에 주의를 기울여야 합니다. 계단마다 평평한 모서리가 있는 계단의 경우 뷰어가 고개를 돌렸을 때 에일리어싱이 발생할 수 있습니다.

그러나 멀리서 계단을 보았을 때 각 계단 자체에서도 에일리어싱이 발생할 수 있습니다. 먼 거리에서 각 계단은 얇은 오브젝트이므로 뷰어가 이동할 때 깜박일 수 있습니다.

다음 그림은 비탈길 대신 계단을 제작했을 때 여러 수준의 에일리어싱 예시를 보여줍니다.

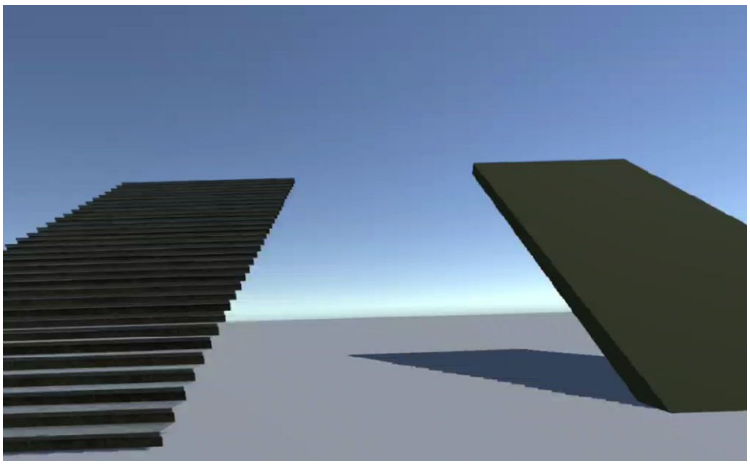

그림 11-17 계단과 비탈길의 에일리어싱 비교

### 계단과 비탈길의 에일리어싱 비교 비디오

다음 비디오는 비탈길 대신 계단을 제작했을 때 여러 수준의 에일리어싱 예시를 보여줍니다.

*https://developer.arm.com/graphics/videos/ramp-vs-stairs-aliasing-example*

가능한 경우 단단한 모서리나 경사진 면 대신 부드럽고 둥그런 형태를 사용합니다. 전선이나 케이블 같은 얇은 오브젝트는 멀리서 볼 때 상당한 에일리어싱이 발생하는 경향이 있습니다. 얇은 오브젝트는 선으로 렌더링되기 때문에 에일리어싱이 발생합니다.

메탈릭 머티리얼을 사용할 때는 특히 주의해야 합니다. 조명이 있을 때 메탈릭 오브젝트가 반사 효과를 발생하는 것은 가능하면 최소화해야 합니다. 반사 효과는 뷰어가 이동할 때 깜박이며 에일리어싱을 일으키기 때문입니다. 따라서 매트(matte)한 머티리얼 사용을 권장합니다.

다음 그림은 메탈릭 머티리얼과 매트 머티리얼에서 발생하는 에일리어싱을 비교해서 보여줍니다. 상단 이미지는 메탈릭 머티리얼을, 하단 이미지는 매트 머티리얼을 보여줍니다.

그림 11-18 메탈릭 머티리얼과 매트 머티리얼 비교

밝은 조명은 릿 오브젝트에 정반사 효과를 발생시키므로 장면 조명 구현은 주의해야 합니다. 이 경우 눈에 띄는 정반사 에일리어싱이 발생할 수 있습니다. 눈은 픽셀의 '깜박임'에 끌리므로 프레임 사이에 정반사 효과가 나타났다가 사라집니다.

비용이 많이 드는 실시간 조명 계산을 피하고 밴딩 같은 조명 아티팩트의 존재를 줄이기 위해 장면이 빌드될 때 조명을 사전 베이킹할 수 있습니다. 조명을 사전 베이킹하려면 장면을 디자인하여 적절한 사전 베이킹으로 이동해야 하는 오브젝트를 줄이고 라이브로 낮밤 주기를 발생시키지 않아야 합니다.

조명 프로브를 활용하면 필요한 사전 베이킹 양을 최소화하며 6방향이 자유로운 VR에서 특히 유리합니다.

VR에서 반사는 일반적인 사용 사례보다 더 주의해야 합니다. 스크린 스페이스 반사 같은 기술은 VR에 사용하기에 컴퓨팅 비용이 매우 큽니다. 따라서 반사에 다른 테크닉을 사용해야 합니다. 고품질 반사가 필요한 오브젝트에 적용할 수 있는 테크닉으로는 런타임에서 반사를 사용하기 위해 사전 렌더링이 가능한 리플렉션 프로브, 큐브 맵 등이 있습니다.

에일리어싱을 숨기기 위해 안개나 연기 같은 파티클 효과를 사용할 수 있습니다. 짧은 렌더링 거리를 숨길 때 오랫동안 사용되어 온 이런 테크닉은 동일 거리의 오브젝트에서 발생하는 에일리어싱을 감추는 데도 사용할 수 있습니다.

다음 그림은 파티클 효과를 추가한 뒤 에일리어싱이 얼마나 감소했는지 보여주는 적절한 그림입니다.

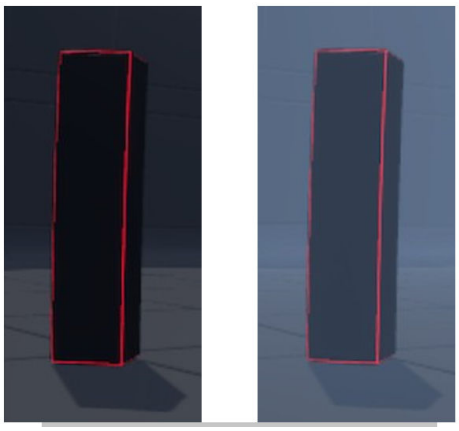

그림 11-19 파티클 효과 적용 전후 비교

## 11.9 밴딩

밴딩은 픽셀당 주어진 숫자 내에서 필요한 색상을 정확하게 표시할 수 없을 때 발생합니다. VR에서 밴딩은 밴드 간에 갑작스러운 변화로 인해 구별되는 색상 밴드로 표시됩니다.

밴드는 사용자가 장면에 몰입하는 VR에서 주로 발견됩니다. 사용자의 눈은 장면의 라이트 레벨에 적응합니다. 눈이 적응하면 밴드가 더욱 뚜렷해집니다. 밴드가 많으면 눈이 계속해서 바뀌는 라이트 레벨에 맞춰 적응하기 때문에 피로감을 느낍니다.

#### 완화 기술 - 디더링

디더링은 밴딩 머티리얼이나 뷰어에게 노이즈를 보여주는 프로세스입니다. 이 프로세스를 통해 구별되는 색상 밴드가 쪼개지고 파괴되어 눈에 덜 띄게 됩니다.

다양한 형태의 노이즈를 적용하거나 적용된 노이즈를 수집할 때 다른 접근법을 사용하는 등 다양한 형태의 디더링을 소개할 수 있습니다. 예를 들면 실시간 노이즈 생성이나 텍스처에서 노이즈를 샘플링해 랜덤 노이즈를 만들어낼 수 있습니다.

#### 완화 기술 - 톤 매핑

톤 매핑은 *고 다이내믹 레인지*(HDR) 색상을 HDR을 표시할 수 없는 화면에 표시가 가능하도록 *저 다이내믹 레인지*(LDR)에 맞도록 변화시키는 색상 변화 하위 세트입니다. 톤 매핑은 *조회 테이블*(LUT)을 사용하여 각 색상이 새 톤에 맞는 해당 컬러로 매핑되도록 합니다.

라이팅 레벨이 낮거나 텍스처의 그라데이션이 날카로워 밴딩이 발생하는 경우 장면에 톤 매핑을 적용해 이를 줄일 수 있습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.
- *11.9.1 Unity에서 디더링 구현* 페이지의 11-269.
- *11.9.2 Unity에서 톤 매핑 구현* 페이지의 11-270.

### 11.9.1 Unity에서 디더링 구현

Unity의 디더링은 주로 추가와 설정이 필요한 포스트 프로세싱 패키지를 통해 처리됩니다. 또는 Universal Render Pipeline에서 카메라에 빌드되는 효과도 있습니다.

디더링을 사용하려면 다음을 따릅니다.

1. Window에서 Package Manager로 이동해 포스트 프로세싱을 검색하고 패키지를 설치합니다. 사용 버전이 2.x인지 확인합니다.
2. Project Window에서 오른쪽 클릭하여 프로젝트 창에 Post Processing Profile을 만듭니다.
3. Create에서 Post-processing Profile을 선택합니다.
4. 카메라 섹션의 Inspector에서 디더링이 필요한 장면 내의 각 카메라에 Post Process Volume과 Post Process Layer 구성요소를 추가합니다.
5. 마지막으로 Post Processing Volume에서 생성된 Post Processing Profile을 프로파일로 설정합니다.

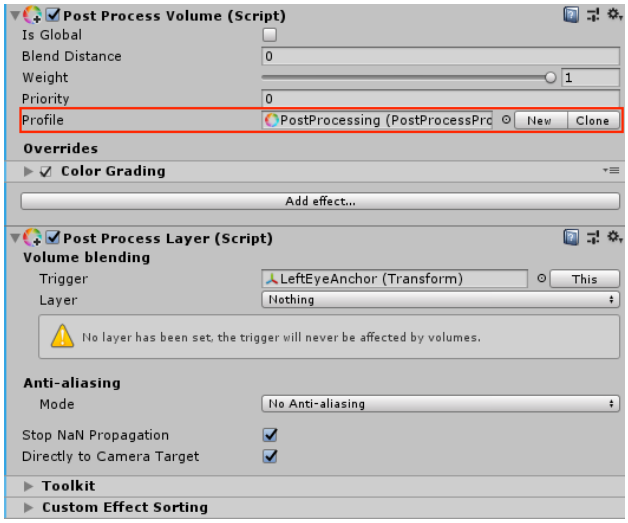


그림 11-20 Post Processing Profile 설정

다음 스크린샷처럼 Unity의 Universal Render Pipeline에서 카메라의 디더링을 사용할 수 있습니다.

1. 렌더링 섹션 내 카메라의 검사기 창에서 Post-Processing 상자에 체크합니다.
2. 그 아래 Dithering 상자를 체크합니다.

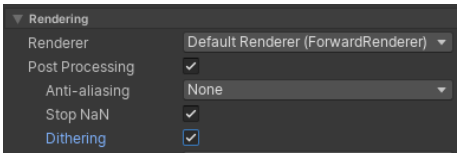


그림 11-21 카메라 디더링을 사용하는 방법

### 11.9.2 Unity에서 톤 매핑 구현

Unity에서 톤 매핑을 구현하려면 Post Processing 패키지를 설치하고 설정해야 합니다.

톤 매핑은 다음과 같은 방법으로 사용합니다.

1. Window에서 Package Manager로 이동해 Post Processing을 검색하고 패키지를 설치합니다. 사용 버전이 2.x인지 확인합니다.
2. 다음으로 Project Window에서 오른쪽 클릭하여 Project Window에 Post Processing Profile을 만듭니다.
3. Create에서 Post-processing Profile을 선택합니다.
4. 이제 이 Post-processing Profile을 선택합니다.
5. Inspector에서 Add effect...를 선택합니다.
6. Unity에서 Unity Color Grading을 선택하고 Mode 옵션을 하이 데피니션 레인지로 선택합니다.
7. 모드에서 톤 매핑 상자에 체크하고 콤보 상자에서 ACES를 선택합니다.
8. 카메라 섹션의 검사기에서 톤 매핑이 필요한 장면 내의 각 카메라에 포스트 프로세싱 볼륨과 포스트 프로세스 레이어 구성요소를 추가합니다.
9. 마지막으로 포스트 프로세싱 볼륨에서 사전 생성된 포스트 프로세싱 프로필을 프로필로 선택합니다.

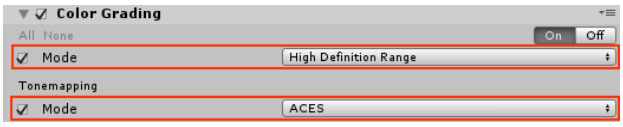


그림 11-22 톤 매핑 설정

## 11.10 범프 매핑

범프 매핑은 메시의 정점 수를 줄이고 객체 표면의 돌출부 같은 세부 디테일을 시뮬레이션하는 데 사용하는 테크닉입니다. 라이팅 계산에 사용되기 전 필요 시뮬레이션은 객체의 노말을 처리하여 수행합니다.

대부분의 경우 노말 매핑 테크닉을 사용하는 게 적절하지만 플레이어가 노말 매핑된 텍스처의 시야각을 쉽게 바꿀 수 있는 VR에서는 효과적이지 않습니다. 따라서 노말 맵 테크닉에서는 이런 퍼스펙티브 변화가 계산되지 않아 뎁스 일루전이 파괴됩니다.

또한 노말은 하나의 시점에서만 생성되므로 노말 매핑은 VR 헤드셋에서 사용되는 입체 렌즈 사용에 적합하지 않습니다. 따라서 각 눈이 동일한 노말을 수신하므로 인간의 눈에서는 부정확하게 보입니다.

#### 범프 매핑 방법 비교 비디오

다음 비디오는 노말 매핑과 패럴랙스 오클루전 매핑을 비교해 보여줍니다.

*https://developer.arm.com/graphics/videos/bump-mapping-example*

#### 완화 기술 – 노말 매핑

노말 매핑은 범프 매핑 시 가장 주로 사용되는 구현이며 모델링 과정에서 메시의 폴리곤 수가 적은 것과 많은 것을 동시에 생성합니다. 이후 노말 맵은 폴리곤 수가 많은 버전에서 내보내기로 생성되며 세부 디테일의 노말은 노말 맵 텍스처에 저장됩니다.

렌더링될 때 샘플 값에서 노말 맵과 노말의 조각 셰이더 샘플이 생성됩니다. 생성된 노말은 라이팅 계산에 사용되기 전 폴리곤 수가 적은 버전의 표면 노말과 결합됩니다. 이렇게 하면 라이팅은 세부 표면 디테일의 개별 정점을 렌더링할 필요 없이 해당 디테일을 표시합니다.

노말 매핑이 일반적으로 VR에서는 효과적이지 않지만 플랫 머티리얼에서는 여전히 노말 맵이 효과적입니다. 특히 장면에 라이팅을 사용할 때 주의해서 노말 맵을 배치하는 경우가 그러합니다.

#### 완화 기술 – 패럴랙스 오클루전 매핑

패럴랙스 오클루전 매핑은 노말 매핑과 유사한 테크닉입니다. 그러나 텍스처 좌표를 대체할 때 표면 노말 대비 뷰어의 각도를 처리합니다.

따라서 좁은 시야각에서 텍스처 좌표는 높은 각도로 대체됩니다. 이렇게 해서 뎁스 일루전을 유지합니다.

패럴랙스 오클루전 매핑은 컴퓨팅 비용이 높은 프로세스입니다. 따라서 이 테크닉은 뷰어가 가까이 다가갈 수 있는 소형 머티리얼에만 사용합니다. 먼 곳의 텍스처는 시야각이 크게 변하지 않으므로 패럴랙스 오클루전 매핑 사용이 효과적이지 않습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *11.10.1 Unity에서 노말 매핑 구현* 페이지의 11-271.
- *11.10.2 Unity에서 패럴랙스 오클루전 매핑 구현* 페이지의 11-272.

### 11.10.1 Unity에서 노말 매핑 구현

노말 매핑은 Unity 내의 여러 셰이더에서 사용할 수 있습니다.

머티리얼에 노말 맵을 추가하려면 다음을 참조합니다.

1. 프로젝트 창에서 머티리얼을 선택합니다.
2. 머티리얼 검사기 패널을 열고 표준, 범용 렌더링 파이프라인, 모바일의 경우 범프 디퓨즈 등 노말 맵 지원을 포함하는 셰이더를 선택합니다.
3. 마지막으로 필요한 노말 맵 텍스처를 설정합니다.

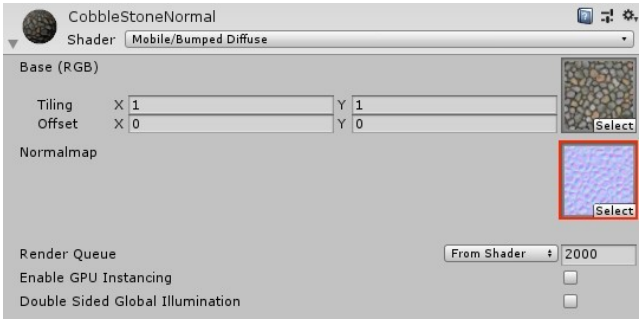


그림 11-23 노말 맵 텍스처 추가

### 11.10.2 Unity에서 패럴랙스 오클루전 매핑 구현

Unity의 내장 셰이더로 패럴랙스 오클루전 매핑을 구현합니다.

머티리얼에 패럴랙스 오클루전을 매핑을 추가하려면 다음을 참조합니다.

1. 프로젝트 창에서 머티리얼을 선택합니다.
2. 머티리얼 검사기 패널을 열고 패럴랙스 디퓨즈 매핑을 지원하는 셰이더를 선택합니다. 예를 들면 표준 셰이더가 있습니다.
3. 마지막으로 필요한 반사율, 노말 맵, 높이 맵 텍스처를 설정합니다.

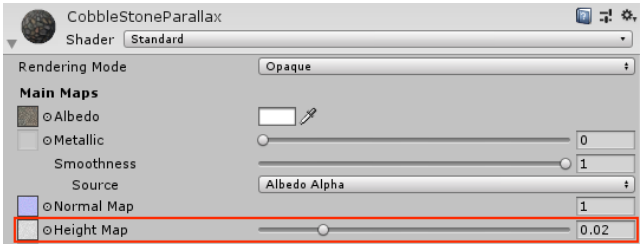


그림 11-24 머티리얼에 패럴랙스 오클루전 매핑 추가

## 11.11 그림자

모바일 장치에서 그림자 매핑을 하는 기존 방식은 컴퓨팅 소모량이 큽니다. 그림자 매핑은 여분의 프레임 버퍼와 렌더링 패스가 필요하므로 성능이 크게 저하됩니다.

따라서 모바일 장치의 그림자 버퍼는 저해상도이며 필터링을 사용하지 않는 경우가 많습니다. 대신 그림자는 대량의 에일리어싱이 발생하므로 그림자 점이나 단단한 그림자 같은 아티팩트가 시뮬레이션 사실성을 손상시킵니다.

다음 스크린샷의 왼쪽에는 일반적인 그림자를 표시하고 오른쪽에는 블롭 그림자를 표시해 비교했습니다.

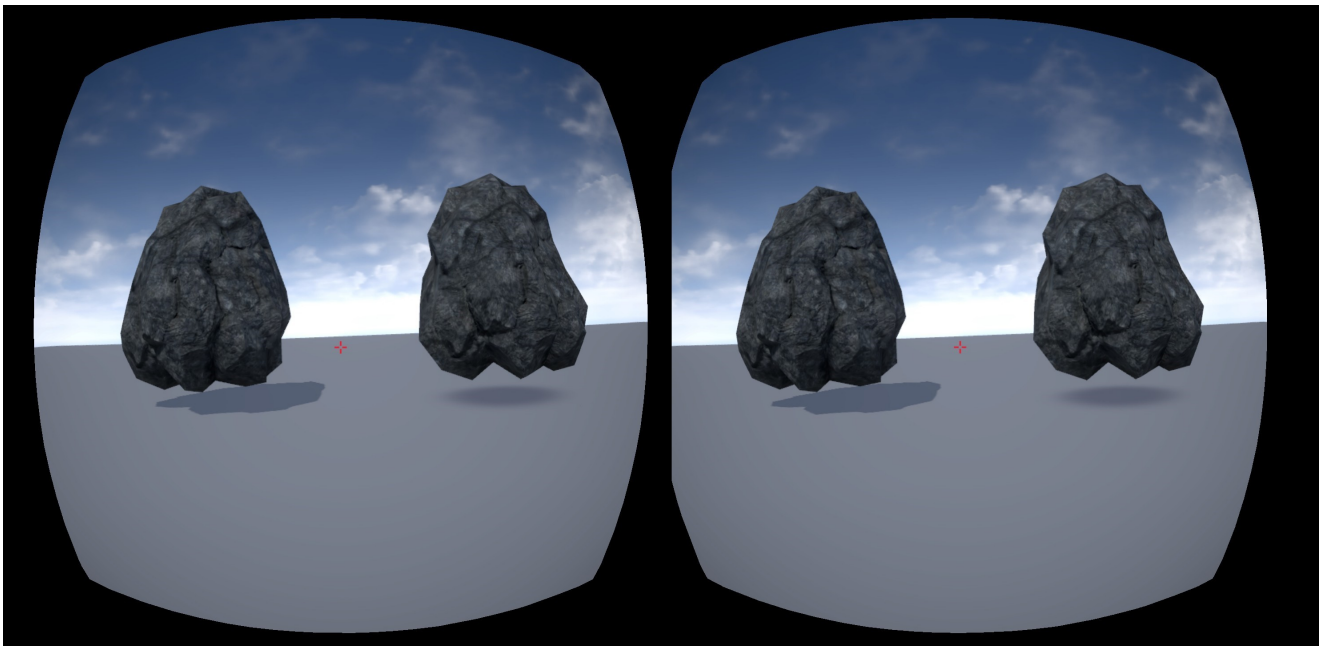

그림 11-25 그림자 매핑과 블롭 그림자 비교

고려할 만한 테크닉으로는 그림자 베이킹이 있습니다. 그림자 매핑과 달리 그림자 베이킹은 여분의 렌더링 패스가 필요하지 않습니다. 대신 그림자가 비치는 텍스처에 고품질 그림자를 베이크합니다.

### 완화 기술 - 블롭 그림자

VR에서는 가능하면 그림자를 렌더링하지 않는 것을 권장하는 편입니다. 하지만 플레이어 밑 등 그림자가 필요한 경우는 오브젝트 밑에 블롭 그림자를 렌더링할 수 있습니다. 블롭 그림자는 일반적인 그림자 매핑 기술보다 에일리어싱이 적게 생기고 성능도 더 낫습니다.

이 장은 다음의 세부 단원으로 구성되어 있습니다.

- *11.11.1 Unity에서 블롭 그림자 구현* 페이지의 11-273.

### 11.11.1 Unity에서 블롭 그림자 구현

블롭 그림자는 장면 구성요소로서 Unity의 애셋 스토어에서 무료로 제공합니다.

블롭 그림자를 구현하려면 다음과 같습니다.

1. Unity 애셋 스토어에서 Unity 표준 애셋을 가져옵니다.
2. 애셋 > 표준 애셋 > 프로젝트 이펙트 > 프리팹에서 블롭 그림자 프로젝트 프리팹으로 이동합니다.
3. 블롭 그림자 프로젝트 프리팹을 계층에서 블롭 그림자가 필요한 오브젝트의 하위 프리팹으로 드래그합니다.

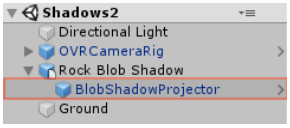


그림 11-26 하위 오브젝트 설정

완료되면 블롭 그림자가 나타납니다. 블롭 그림자를 설정을 완료하려면 다음과 같습니다.

1. 프리팹 옵션을 편집합니다. 그림자가 캐스팅되는 오브젝트와 관련된 블롭 그림자의 위치를 포함했는지 확인합니다.
2. 그림자가 캐스팅되는 오브젝트를 선택하고 그림자 캐스트 옵션을 끄기로 선택합니다.

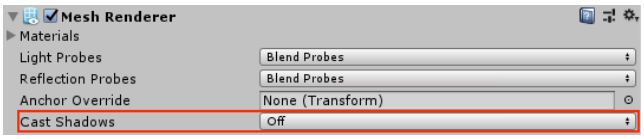


그림 11-27 오브젝트의 그림자 캐스트 사용 중지

# 제 12 장
# Vulkan

이 장에서는 Vulkan과 Vulkan을 활성화하는 방법을 설명합니다.

Vulkan은 Khronos 그룹의 교차 플랫폼 그래픽 및 컴퓨팅 API로, OpenGL과 OpenGL ES보다 많은 이점을 제공합니다. 다음과 같은 이점이 있습니다.

- 모바일, 데스크톱, 콘솔, 서버 및 임베디드 시스템을 위해 하나의 통합된 API 프레임워크를 제공합니다.
- 다양한 기능으로 하드웨어를 지원합니다.
- 최소한의 드라이버 오버헤드로 ARM Mali GPU 하드웨어에서 높은 성능을 지원합니다.
- 응용 프로그램이 GPU와 컴퓨팅 리소스를 통해 더 많은 하위 수준 액세스를 얻습니다.
- 응용 프로그램 프로세서 병목 현상이 줄어듭니다.
- 다중 스레딩 및 다중 처리를 지원합니다.
- 여러 응용 프로그램 프로세서를 효율적으로 사용합니다.
- 셰이더를 위해 SPIR-V 중간 언어를 사용해 런타임 커널 컴파일 시간을 단축합니다.
- 셰이더 소스 코드를 탑재할 필요가 없습니다.
- 낮은 응용 프로그램 프로세서 오버헤드, 간소화된 드라이버, 더 많은 온칩 메모리 사용으로 인해 에너지 소비량이 감소합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 12.1 Vulkan 정보

Vulkan은 Khronos 그룹의 교차 플랫폼 그래픽 및 컴퓨팅 API입니다.

Vulkan은 기존의 OpenGL 및 OpenGL ES 표준보다 다음과 같은 많은 장점이 있습니다.

통합 및 이식 가능
: Vulkan은 모바일, 데스크톱, 콘솔 및 임베디드 시스템을 위해 하나의 통합된 API 프레임워크를 제공합니다.
폭넓고 다양한 구현을 통해 플랫폼 간에 이식성이 있고 광범위한 응용 프로그램에 유용합니다.

더욱 단순한 드라이버
: Vulkan은 더욱 단순한 드라이버를 사용하여 드라이버 오버헤드를 최소화합니다. 낮은 대기시간과 높은 효율은 Vulkan을 사용한 응용 프로그램이 OpenGL ES 3.1을 사용했을 때보다 더 좋은 성능을 달성할 수 있다는 의미입니다.
이처럼 더 단순한 드라이버는 응용 프로그램 프로세서 병목 현상을 줄여줍니다.
응용 프로그램이 리소스 관리를 수행하고 GPU에 대해 하위 수준에서 직접 제어할 수 있습니다.

다중 스레딩 및 다중 처리
: Vulkan은 여러 응용 프로그램 프로세서에 걸쳐 다중 스레딩을 지원합니다. 이 기능 덕분에 여러 응용 프로그램 프로세서를 효율적으로 사용하여 처리 부하와 소비전력을 낮출 수 있습니다.
응용 프로그램이 스레드 관리와 동기화를 제어합니다.

커맨드 버퍼
: 멀티 스레딩을 사용하여 커맨드 버퍼를 위한 명령 생성을 병렬로 수행할 수 있습니다. 또한 별개의 전송 스레드를 사용하여 커맨드 버퍼를 커맨드 큐에 넣을 수 있습니다.
개발자는 그래픽, 컴퓨팅 및 DMA에 대한 커맨드 버퍼를 추가할 수 있습니다.
다양한 그래픽 큐, DMA 큐 및 컴퓨팅 큐가 작업 디스패치를 위한 유연성을 제공합니다.
다중 스레드 방식의 명령 생성을 사용하여 여러 응용 프로그램 프로세서 코어에서 코드를 실행하여 성능을 높일 수 있습니다. 클록 속도가 높은 프로세서 하나만 사용하기보다는 더 낮은 클록 속도로 작동하는 여러 개의 응용 프로그램 프로세서를 사용해도 소비전력이 감소됩니다.

**SPIR-V**
: Vulkan은 공통 언어 프런트 엔드를 사용할 수 있게 해주는 SPIR-V 중간 언어를 사용합니다.
SPIR-V는 병렬 컴퓨팅과 그래픽을 위한 중간 언어인 다중 API입니다. SPIR-V는 흐름 제어, 그래픽 및 병렬 컴퓨팅 생성 기능을 포함합니다.
SPIR-V는 Vulkan 셰이더 및 OpenCL 커널 소스 언어를 위한 네이티브 표현을 제공합니다.
여러 플랫폼이 동일한 SPIR-V 프런트 엔드 컴파일러를 사용하여 사전 컴파일된 셰이더를 생성할 수 있습니다.
SPIR-V를 사용한다는 것은 Vulkan 드라이버에 프런트 엔드 컴파일러가 없으므로 드라이버가 더욱 단순하고 셰이더 컴파일이 더 빠르게 수행된다는 의미입니다.
중간 언어를 사용한다는 것은 응용 프로그램에 셰이더 소스 코드를 포함할 필요가 없다는 의미입니다. 향후 다른 셰이딩 언어를 사용할 수 있는 유연성도 제공됩니다.

로드 가능한 레이어
: Vulkan을 사용하면 개발 중에 테스트와 디버깅을 위해 소프트웨어 레이어를 로드할 수 있습니다.
제품화를 위해 추가 소프트웨어 레이어를 제거할 수 있으므로 제품 출고 시 테스트 오버헤드가 발생하지 않습니다.

멀티패스 렌더링

멀티패스 렌더링은 렌더 패스에 앞서 모든 것을 선언하는 기법입니다. 각 하위 패스에 대해 제각기 다른 출력을 지정하고 이들을 함께 연결할 수 있습니다.
멀티패스를 사용하면 한 하위 패스의 픽셀이 같은 픽셀 위치에 있는 이전 하위 패스의 결과에 액세스할 때 드라이버가 최적화를 수행할 수 있습니다. 이런 식으로 고속 온칩 메모리에 데이터를 넣음으로써 대역폭과 전력을 절감할 수 있습니다. 이 프로세스는 Mali GPU와 같은 타일 기반 GPU에서 더욱 효율적입니다.
멀티패스 렌더링에 대한 예시 사용 사례는 다음과 같습니다.

- 지연 렌더링 (deferred rendering)
- 소프트 파티클
- 톤 매핑

다음 다이어그램은 Vulkan의 구조를 보여줍니다.

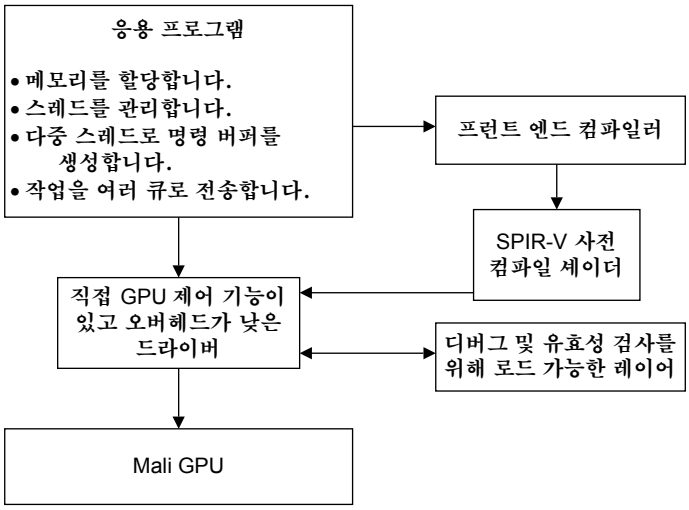


그림 12-1 Vulkan 구조

## 12.2 Unity의 Vulkan 정보

Unity에서 Vulkan을 사용할 때의 장점.

Vulkan의 많은 장점은 Vulkan의 드라이버가 OpenGL 및 OpenGL ES에 비해 간단하다는 사실에서 비롯됩니다.

하지만 관리 작업 중 많은 부분을 응용 프로그램 내에서 처리해야 한다는 단점이 있는데, 이러한 하위 수준 액세스 때문에 응용 프로그램을 작성하기 더 복잡해집니다.

Unity는 이러한 작업을 대부분 자동으로 처리하므로 응용 프로그램은 Vulkan 덕분에 발생한 성능 향상 이익을 자동적으로 얻게됩니다.

Unity에서는 Unity가 응용 프로그램을 빌드할 때 사용할 수 있는 그래픽 API 목록에 Vulkan을 추가하기만 하면 됩니다.

Vulkan은 OpenGL ES보다 더 효율적이고 소비전력을 줄이고 배터리 수명은 늘려주므로 사용 가능하다면 대체로 더 나은 선택이라 할 수 있습니다. 다음과 같은 경우 Vulkan 사용을 고려하십시오.

- 응용 프로그램의 최대 성능을 원하는 경우.
- 응용 프로그램 프로세서에 발이 묶여 충분한 성능을 내지 못하는 경우.
- OpenGL 또는 OpenGL ES 드라이버가 문제를 일으키고 있는 것으로 의심될 경우.

## 12.3 Unity에서 Vulkan 활성화

여기서는 Unity 5.6에서 Vulkan을 활성화하는 절차를 설명합니다.

Vulkan을 활성화하는 방법은 다음과 같습니다.

1. File > Build Settings …를 선택합니다.
2. 새 창의 대화 상자에서 Player Settings … 버튼을 누릅니다.
3. Player Settings 창에서 Other Settings 섹션을 찾습니다.
4. Auto Graphics API 확인란이 선택되지 않은 상태임을 확인합니다. 이 상태에서 API를 수동으로 선택할 수 있습니다.
5. Vulkan을 API로 추가하려면 +를 눌러 새 API를 목록에 추가하고 Vulkan을 추가합니다. Vulkan이 마지막 옵션으로 추가됩니다.
6. Vulkan을 기본 API로 만들려면 Vulkan을 선택하고 목록 맨 위로 이동합니다.

이제는 Vulkan이 플레이어의 기본 API입니다. 다음 이미지에 이 내용이 표시되어 있습니다.

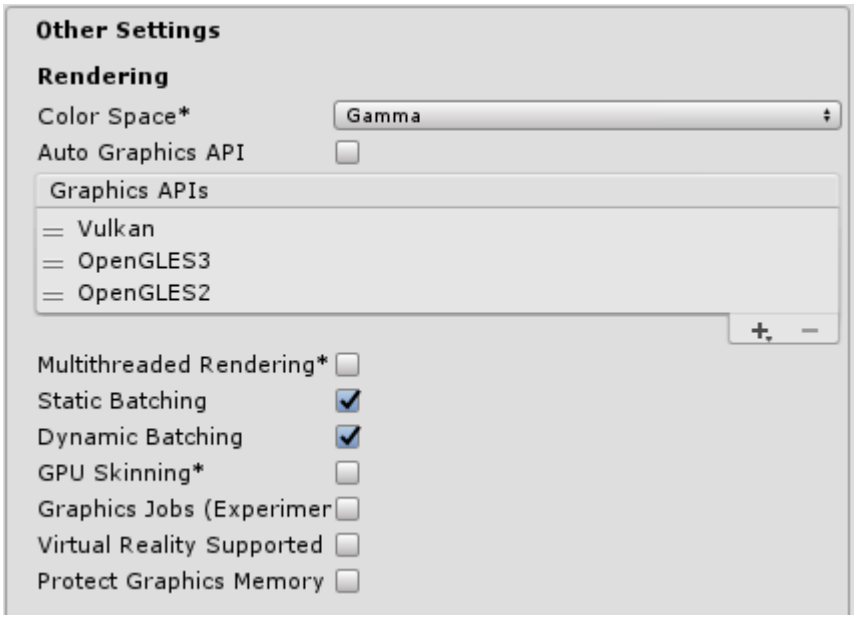


그림 12-2 Unity API 선택

다른 둘을 제거하여 Vulkan만 유일한 API로 남길 수 있습니다.

이렇게 하려면 API를 선택하고 –를 눌러 해당 API를 제거합니다.

## 12.4 Vulkan 사례 연구

게임 개발사 Infinite Dreams가 Vulkan을 사용해 성능을 개선한 방법.

모바일 게임 개발사 Infinite Dreams는 인기 모바일 게임 Sky Force Reloaded를 개발하는 회사입니다. Sky Force Reloaded는 응용 프로그램 프로세서와 GPU 모두 매우 높은 수준의 성능을 요구하는 고품질의 그래픽 환경에서 격렬한 액션이 펼쳐지는 게임입니다. Sky Force Reloaded는 원래 Unity와 OpenGL ES를 함께 사용하여 제작되었습니다.

그림 12-3 Sky Force Reloaded

### Sky Force Reloaded 최적화

이 게임의 초기 개발 작업에서는 필요한 고품질의 그래픽 환경을 만들었습니다. 다음 단계는 최적화였습니다.

이 게임은 화면에 많은 것이 동시에 표현되어 그래픽적으로 복잡하므로 개발팀에서는 최적화 작업에서 가장 큰 부분을 차지하는 것이 필 레이트일 것이라 생각했습니다. 필 레이트와 관련된 문제는 보통 프레임 버퍼의 해상도를 줄이면 해결할 수 있습니다.

하지만 낮은 해상도에서 게임을 렌더링했는데도 여전히 성능 문제가 발생하는 것이었습니다. 고사양 기기에서도 60*FPS(frames per second)*를 항상 유지할 수 있는 건 아니었습니다. 개발팀은 게임에서 다수의 드로우 콜이 이루어지고 있다는 사실을 발견했습니다. 때로는 프레임당 최대 1,000회의 드로우 콜이 이루어지기도 했습니다.

드로우 콜마다 계산 오버헤드가 발생하므로 드로우 콜이 많이 이루어진다는 건 결국 컴퓨터 리소스를 많이 소모하는 것입니다. 이로 인해 OpenGL ES 드라이버가 GPU를 위한 데이터를 준비하느라 오랜 시간 동안 응용 프로그램 프로세서를 계속 사용하게 됩니다. 결국 고사양 모바일 기기에서도 속도가 느려질 수 있는 것입니다.

개발팀은 최적화를 위해 드로우 콜 횟수를 최소화하거나 게임 엔진이 드로우 콜을 일괄 처리하도록 수정할 수 있을 것입니다. 하지만 항상 게임의 품질을 떨어뜨리지 않고 이를 실현할 수 있는 것은 아닙니다.

### Vulkan 테스트

개발팀은 Vulkan에 대한 소식을 들었고 Vulkan으로 게임 성능을 향상할 수 있을지 살펴보기로 했습니다.

Unity에서는 Vulkan이 렌더링 API로 사용되며 Unity가 어려운 작업을 죄다 도맡아 합니다. 개발팀은 Unity에서 Vulkan을 그냥 활성화하는 것만으로 Vulkan을 사용할 수 있었습니다.

개발팀은 OpenGL ES와 Vulkan의 차이점을 테스트하기 위해 게임을 분석했습니다. 그 결과 게임에서 OpenGL이 60FPS의 속도를 내지 못하는 가장 느린 부분 중 하나를 찾았습니다. OpenGL ES와 Vulkan 사이의 성능 차이를 측정하기 위해 개발팀은 게임에서 이 부분을 기준으로 가상 벤치마크를 만들었습니다.

이 벤치마크에서는 API가 렌더링할 반복 가능한 장면을 사용하여 두 API에 모두 일관된 테스트를 제공합니다.

다음 그래프에서는 OpenGL ES는 어려움을 겪지만 Vulkan은 대부분의 시간 동안 60FPS를 유지할 수 있다는 점을 확인할 수 있습니다. Vulkan의 전체적인 성능은 OpenGL ES에 비해 15% 높습니다. 개발팀은 Vulkan이 그래픽 API일 뿐이라는 점을 생각해보면 이는 매우 좋은 결과라고 생각했습니다.

다음 이미지는 OpenGL ES와 Vulkan을 비교한 벤치마크 결과를 보여줍니다.

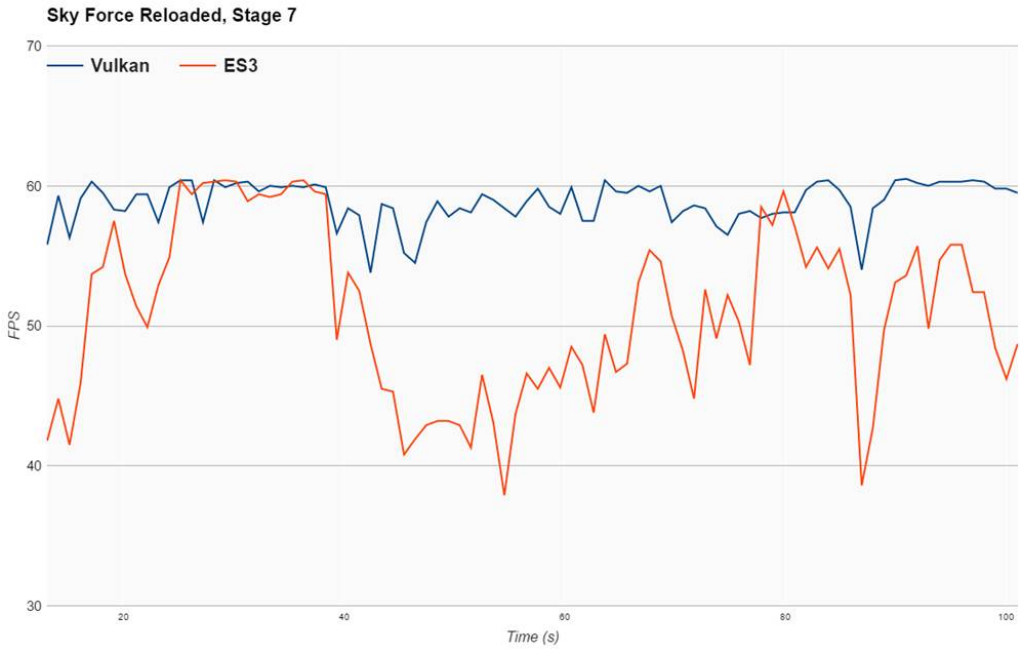


그림 12-4 OpenGL ES와 Vulkan을 비교한 Sky Force Reloaded 벤치마크

### Vulkan에서 얻는 추가 성능 테스트

Vulkan을 사용했더니 대부분의 시간 동안 60FPS로 작동할 수 있다는 점을 확인한 개발팀은 Vulkan을 이용해 얼마나 더 많은 성능을 얻을 수 있을지 궁금했습니다.

그래서 Vulkan조차도 60FPS를 달성하지 못하는 지점까지 벤치마크 수준에 다른 오브젝트를 더 추가하기 시작했는데, 이로써 OpenGL ES와 Vulkan 사이의 격차는 더 커졌습니다. Vulkan이 OpenGL ES보다 평균 32% 더 빠른 것으로 나타났기 때문입니다.

다음 그래프는 OpenGL ES와 Vulkan을 비교한 벤치마크 결과를 보여주는데, 이번에는 장면에 다른 오브젝트를 더 추가한 경우입니다.

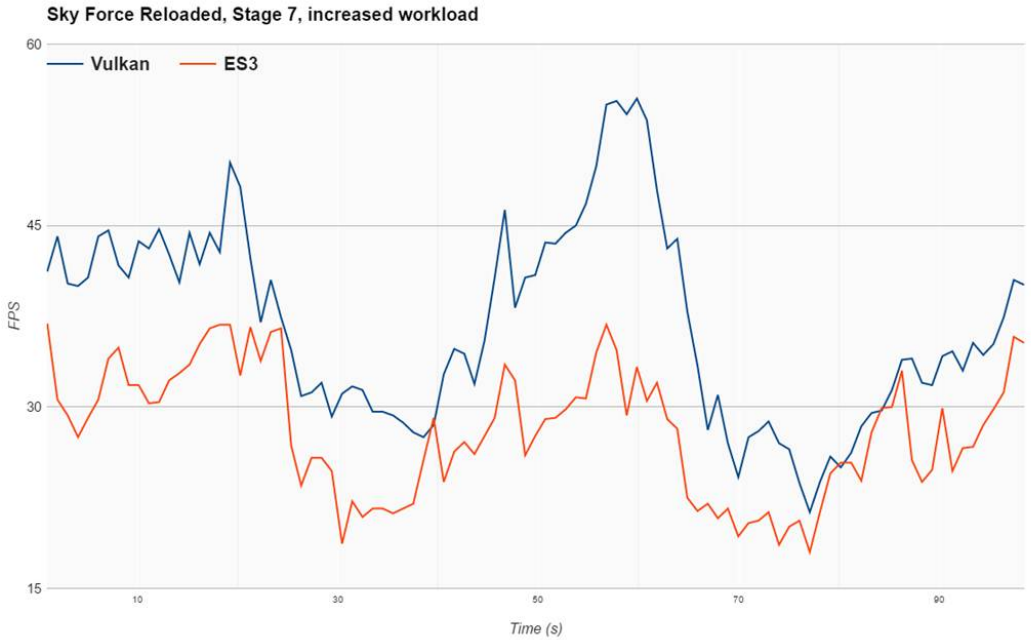


그림 12-5 객체를 추가한 Sky Force Reloaded 벤치마크

분명히 여분의 성능을 더 끌어낼 수 있었으므로 개발팀은 이 여분의 성능을 게임에서 어떻게 활용할 수 있을지 자문했습니다. 개발팀은 그래픽, 파티클, 오브젝트, 애니메이션을 더 많이 추가했습니다. 개발팀은 게임을 훨씬 더 풍성하게 보이도록 만들면서도 60FPS를 유지할 수 있다는 사실을 발견했습니다. Vulkan 덕분에 이 모든 것이 사실상 공짜로 얻은 성과인 셈이었습니다.

### Vulkan 비디오 비교

*https://www.youtube.com/watch?v=VCSkp-QZ37M*에서 비디오 비교 내용을 확인할 수 있습니다.

이 비디오에서는 Vulkan 버전과 OpenGL ES 버전의 성능을 나란히 놓고 비교한 화면을 보여줍니다. 비디오 왼쪽은 게임에 OpenGL ES를 사용한 장면입니다. 비디오 오른쪽은 게임에 Vulkan을 사용한 장면입니다. 두 버전 모두 초당 60개의 프레임으로 실행 중입니다.

이 프레임 속도에서, Vulkan은 OpenGL ES 버전에 비해 별은 6배, 총알은 2배 더 많이 렌더링할 수 있습니다. OpenGL ES를 사용해 같은 수의 오브젝트를 렌더링하면 복잡하게 돌아가는 장면 중에 프레임 속도가 현저히 떨어지게 됩니다. Vulkan을 사용하면 OpenGL ES에 비해 같은 프레임 속도에서 화면에 더 많은 지오메트리를 추가할 수 있습니다.

다음 이미지는 비교 비디오에서 발췌한 프레임을 보여줍니다.

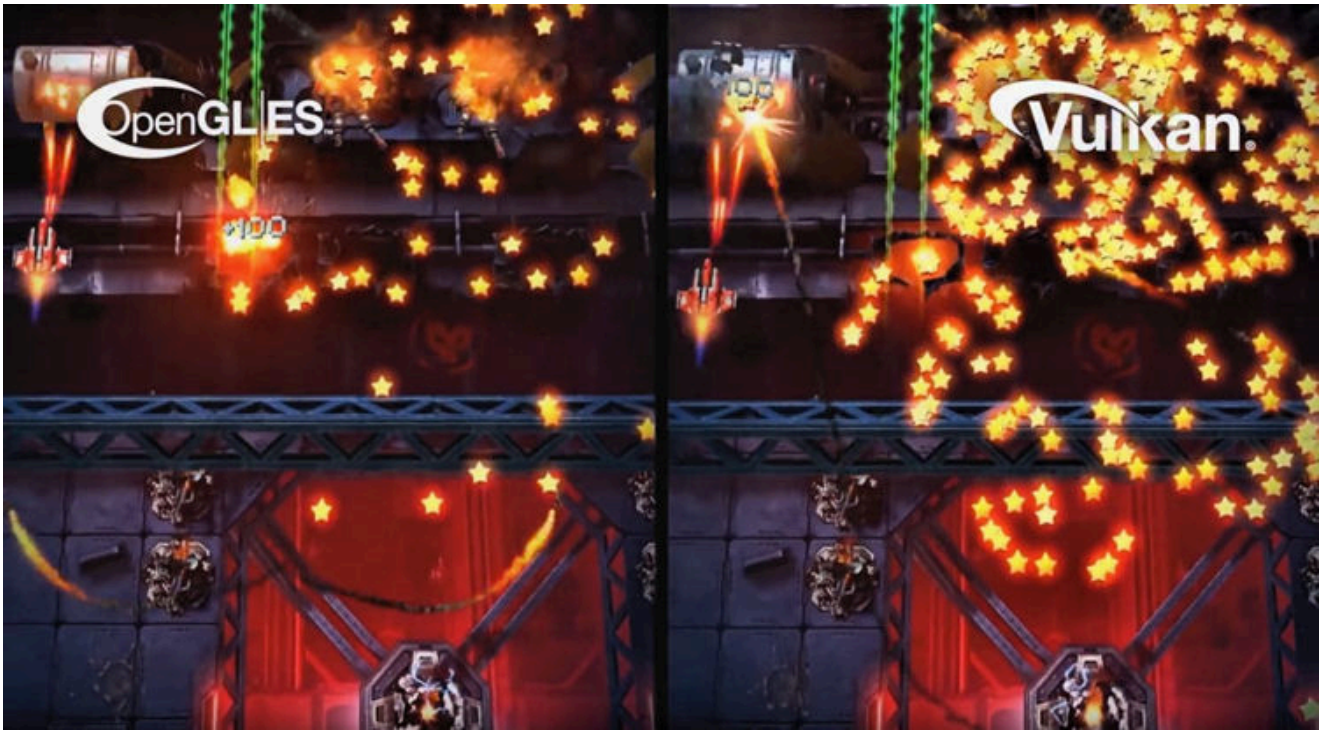


그림 12-6 OpenGL ES와 Vulkan을 비교한 Sky Force Reloaded 프레임

### Vulkan 소비전력

Sky Force Reloaded는 응용 프로그램 프로세서와 GPU의 사용량이 무척 많은 게임입니다. 일부 플레이어는 이 게임을 하다 보면 배터리가 너무 빨리 닳는다는 불만도 제기했습니다. 개발팀은 배터리 소모량을 줄이려고 콘솔 게임과 같은 수준의 게임 품질을 조금이라도 떨어뜨리고 싶지는 않았고, 그래서 혹시 Vulkan이 도움이 될 수 있을지 살펴봤습니다. Vulkan이 소비전력을 줄여 배터리 수명을 늘려줄지 알아보는 테스트를 실시했습니다. Unity에서 Vulkan을 활성화했더니 소비전력이 10~12% 정도 줄어 같은 배터리로 10~12%에 해당하는 시간 동안 더 오래 게임을 즐길 수 있었습니다.

*https://www.youtube.com/watch?v=WI7nXq8oozw*에서 비디오 비교 내용을 확인할 수 있습니다.

# 제 13 장
# ARM Mobile Studio

이 장은 Arm Mobile Studio 도구를 다룹니다.

이 장은 다음 단원으로 구성되어 있습니다.

## 13.1 Arm Mobile Studio 정보

Arm Mobile Studio 제품군의 성능 분석 도구는 루팅되지 않은 Android 장치에서 게임 개발 워크플로 중 다양한 수준의 디테일 성능을 평가하는 데 도움이 됩니다.

성능 분석 도구 제품군에는 다음 제품들이 포함됩니다.

- Streamline
- Performance Advisor
- Graphics Analyzer
- Mali Offline Compiler.

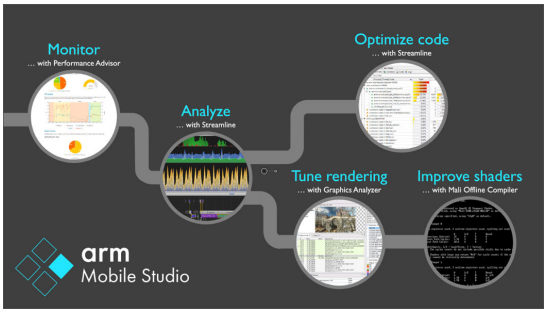


그림 13-1 ARM Mobile Studio

### Streamline

*Streamline*은 게임이 실행 중일 때 Android 장치에서 성능 데이터를 캡처합니다. 캡처가 완료되면 데이터를 그래프 방식으로 확인하여 게임에서 리소스를 정확히 어떻게 사용하는지 볼 수 있습니다.

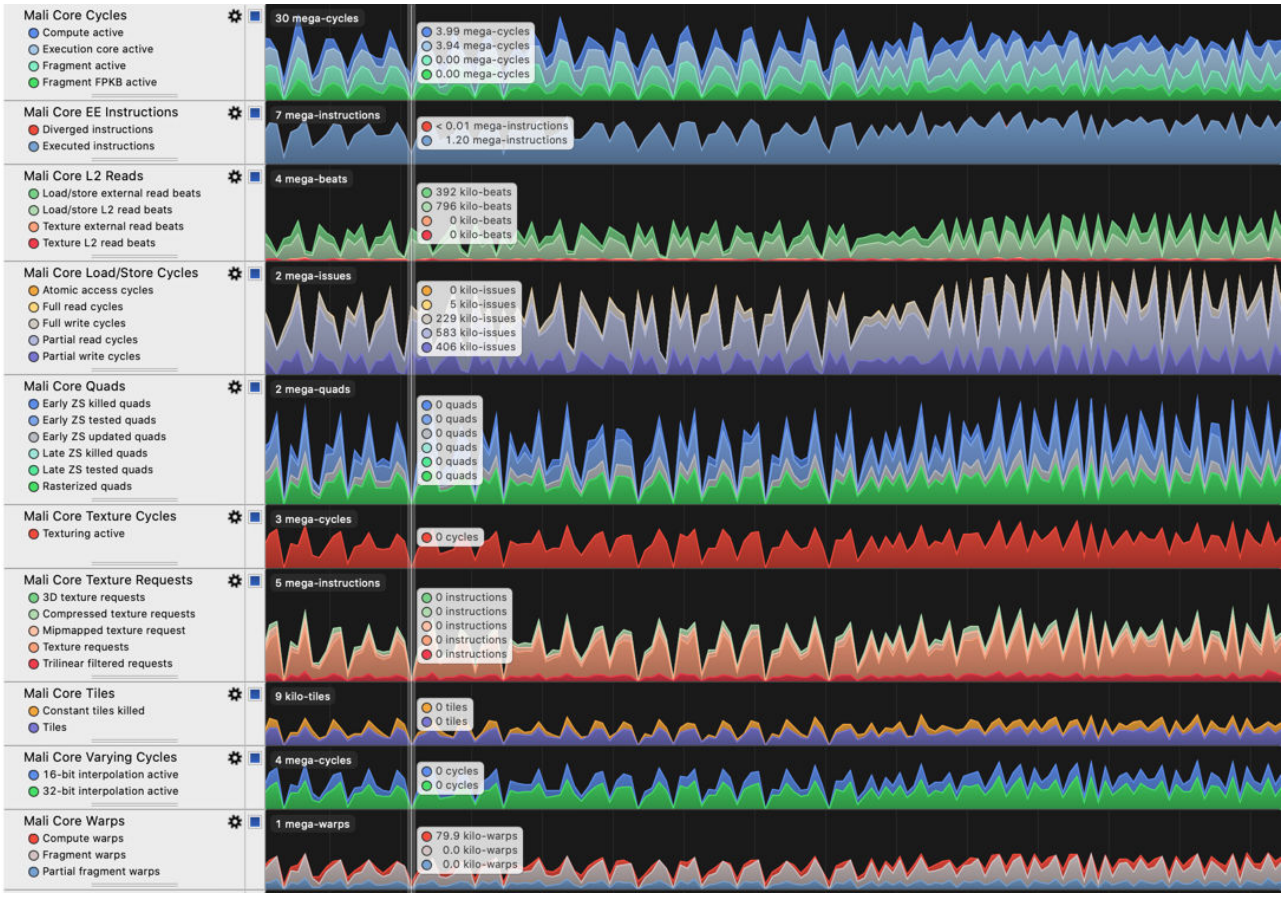


그림 13-2 Streamline에서 성능 카운터 그래프 보기

Streamline을 사용해 성능 문제를 확인할 때 정확히 어떤 부분에 주목해야 할지 모르는 경우에는 많은 시간이 소요될 수 있습니다. *Performance Advisor*는 가벼운 리포팅 툴로 Streamline의 캡처를 보기 쉬운 보고서 형태로 변환하여 응용 프로그램 성능을 간략하게 보여주고 최적화 조언을 제공합니다.

### Performance Advisor

Performance Advisor 보고서의 처음 부분에서는 캡처 기간 중 전체 성능 요약을 보여줍니다. 평균 FPS, 작업량 분석, 장치의 평균 CPU 및 GPU 사용량을 확인할 수 있습니다.


그림 13-3 Performance Advisor 보고서

이러한 간편한 메트릭은 일일 실행 모니터링을 통해 개발 중 응용 프로그램 성능 변화 정도를 파악하는 데 유용합니다.

더욱 자세한 FPS 분석 그래프는 시간에 따른 응용 프로그램 성능을 보여줍니다. 응용 프로그램의 성능에 문제가 없는 경우 그래프의 배경색이 초록색입니다.

성능에 문제가 있는 구역에서는 배경색을 통해 문제를 나타냅니다. 여기에서 그래프 대부분은 파란색이므로 장치의 GPU가 조각 작업량 처리에 문제를 겪고 있음을 알 수 있습니다.

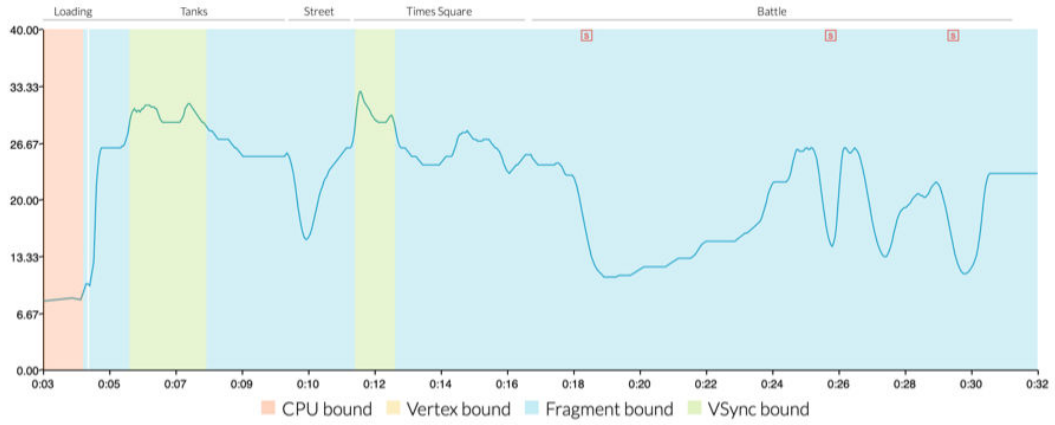


그림 13-4 FPS 분석 그래프

추가적인 그래프로 작업량, 콘텐츠 속성, GPU의 기능 유닛 활용 방식에 대한 정보를 제공합니다. 또한 각 그래프는 FPS를 표시하므로 문제 발생과 관련이 있는 영역을 확인할 수 있습니다.

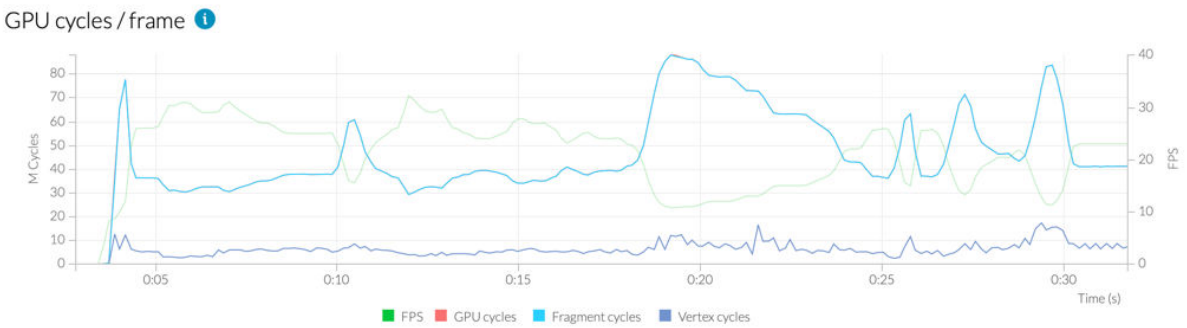


그림 13-5 GPU 주기 그래프

Performance Advisor는 잠재적 문제를 발견하는 경우 이를 표시하며, 문제를 해결하고 성능을 개선할 수 있는 Arm 개발자 웹 사이트의 *최적화 팁* 링크를 안내합니다.

Performance Advisor에서 성능 문제를 발견한 경우 Arm Mobile Studio 제품군의 다른 도구를 활용해 문제를 더욱 자세히 파악할 수 있습니다.

### Graphics Analyzer

*Graphics Analyzer*는 Android 장치에서 실행되는 응용 프로그램의 모든 OpenGL ES 및 Vulkan API 호출을 평가할 수 있습니다. 게임의 장면을 프레임 및 드로우 콜별로 탐색하여 렌더링 문제를 찾고 성능 최적화 기회를 발견할 수 있습니다.

드로우 콜을 단계별로 실행해 프레임 구성을 확인할 수 있습니다. 그래픽 API 호출과 객체 지오메트리, 프레임 버퍼 출력을 동시에 확인해 각 드로우 콜이 장면에 어떤 영향을 주는지 평가합니다.

여러 쉐이더가 장면에서 사용되는 방식, 쉐이더별 드로우되는 프래그먼트 수, 쉐이더 드로잉에 사용되는 GPU 사이클 수를 확인합니다.

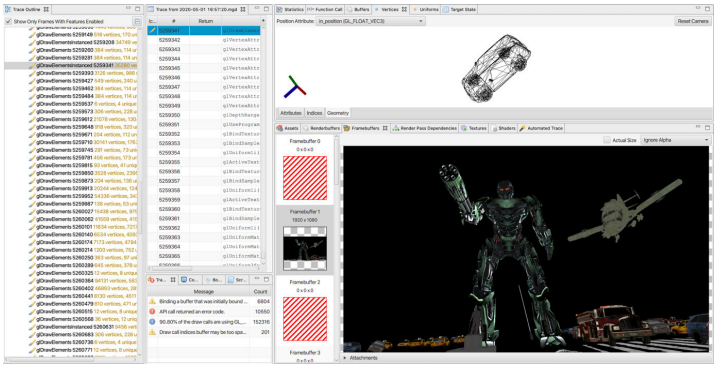

그림 13-6 Graphics Analyzer에서 드로우 콜, 지오메트리, 프레임 버퍼 출력 확인하기

### Mali Offline Compiler

*Mali Offline Complier*는 쉐이더 프로그램을 컴파일하고 Mali GPU에서 쉐이더 프로그램의 실행을 분석하기 위한 커맨드 라인(command-line) 도구입니다. 디자인 내 주요 기능 유닛에 대한 대략의 사이클(cycle) 비용 분석을 확인하여 쉐이더 프로그램의 최적화 방법을 확인합니다.

Mali Offline Compiler 성능 보고서 예:

```
Configuration
=============

Hardware: Mali-G76 r0p0
Driver: Bifrost r19p0-00rel0
Shader type: OpenGL ES Fragment

Main shader
===========

Work registers: 32
Uniform registers: 34
Stack spilling: False

A   LS    V    T  Bound
Total Instruction Cycles:  4.5  0.0  0.2  2.5      A
Shortest Path Cycles:      1.0  0.0  0.2  2.5      T
Longest Path Cycles:       4.5  0.0  0.2  2.5      A

A = Arithmetic, LS = Load/Store, V = Varying, T = Texture

Shader properties
=================

Uniform computation: False
```

*Arm 개발자 웹 사이트*에 방문해 Arm Mobile Studio에 대한 자세한 정보를 확인하고 *Starter Edition 다운로드*를 무료로 받을 수 있습니다. 지속 통합 워크플로의 일부로서 자동 분석에 사용될 수 있는 Arm Mobile Studio Professional Edition에 관심이 있는 경우 *문의하십시오*.

# 부록 A
# 버전

이 부록에서는 이 설명서의 버전 간 변경 사항을 설명합니다.

이 장은 다음 단원으로 구성되어 있습니다.

## A.1 버전

이 부록에서는 이 설명서 버전에 추가된 사항을 설명합니다.

표 A-1 버전 3.0 추가 사항

| 추가 | 위치 | 버전 |
|---|---|---|
| 사용자 지정 쉐이더에서 Enlighten 사용 | - | 버전 3.0 |
| 반사 결합 | *9.3 반사 결합* 페이지의 9-169 | 버전 3.0 |
| Early-z 사용 | *9.7 Early-z 사용* 페이지의 9-192 | 버전 3.0 |
| 더티 렌즈 효과 | *9.8 더티 렌즈 효과* 페이지의 9-193 | 버전 3.0 |
| 라이트 샤프트 | *9.9 라이트 샤프트* 페이지의 9-196 | 버전 3.0 |
| 안개 효과 | *9.10 안개 효과* 페이지의 9-200 | 버전 3.0 |
| 블룸 | *9.11 블룸* 페이지의 9-207 | 버전 3.0 |
| 빙벽 효과 | *9.12 빙벽 효과* 페이지의 9-214 | 버전 3.0 |
| 절차적 스카이박스 | *9.13 절차적 스카이박스* 페이지의 9-220 | 버전 3.0 |
| 반딧불이 | *9.14 반딧불이* 페이지의 9-228 | 버전 3.0 |
| 탄젠트 공간-월드 공간 변환 도구. | *9.15 탄젠트 공간-월드 공간 노말 변환 도구* 페이지의 9-232 | 버전 3.0 |

표 A-2 버전 3.0_01 변경 사항

| 변경 | 위치 | 버전 |
|---|---|---|
| Early-z 사용에서 한 목록 항목을 제거 | *9.7 Early-z 사용* 페이지의 9-192 | 버전 3.0 |

표 A-3 버전 3.1 변경 사항

| 변경 | 위치 | 버전 |
|---|---|---|
| 가상 현실에 관한 장을 추가 | *제 장 10 가상현실* 페이지의 10-239 | 버전 3.1 |

표 A-4 버전 3.2의 변경 사항

| 변경 | 위치 | 영향을 받는 대상 |
|---|---|---|
| Enlighten의 정보 및 구조 장 업데이트 | - | 버전 3.2 최초 릴리스 |

표 A-5 버전 3.3의 변경 사항

| 변경 | 위치 | 영향을 받는 대상 |
|---|---|---|
| Vulkan에 관한 장 추가 | *제 장 12 Vulkan* 페이지의 12-275 | 버전 3.3 최초 릴리스 |
| Mali Graphics Debugger에 관한 장 추가 | *제 장 13 ARM Mobile Studio* 페이지의 13-284 | 버전 3.3 최초 릴리스 |

표 A-6 버전 3.3_01의 변경 사항

| 변경 | 위치 | 영향을 받는 대상 |
|---|---|---|
| 섹션 8.3의 스크린샷 업데이트 | *12.3 Unity에서 Vulkan 활성화* 페이지의 12-279 | 버전 3.3 두 번째 릴리스 |

표 A-7 버전 4.0 변경 사항

| 변경 | 위치 | 버전 |
|---|---|---|
| ARM Mobile Studio 장 추가. | *제 장 13 ARM Mobile Studio* 페이지의 13-284 | 버전 4.0 최초 릴리스 |
| Mali Graphics Debugger의 모든 인스턴스를 Graphics Analyzer로 재명명. | 이 문서 전체 | 버전 4.0 최초 릴리스 |
| Graphics Analyzer 콘텐츠를 ARM Mobile Studio 장으로 이동. | *제 장 13 ARM Mobile Studio* 페이지의 13-284 | 버전 4.0 최초 릴리스 |
| ARM 브랜딩 이미지 업데이트. | 이 문서 전체 | 버전 4.0 최초 릴리스 |
| 본문에서 강조된 모든 참조 삭제. | 이 문서 전체 | 버전 4.0 최초 릴리스 |

표 A-8 버전 4.1 변경 사항

| 변경 | 위치 | 버전 |
|---|---|---|
| 지오메트리 모범 사례 장 추가. | *제 장 5 실시간 3D 아트 모범 사례: 지오메트리* 페이지의 5-75 | 버전 4.1 최초 릴리스 |
| 텍스처링 모범 사례 추가. | *제 장 6 실시간 3D 아트 모범 사례: 텍스처링* 페이지의 6-90 | 버전 4.1 최초 릴리스 |
| 머티리얼 및 쉐이더 모범 사례 추가. | *제 장 7 실시간 3D 아트 모범 사례: 머티리얼과 셰이더* 페이지의 7-111 | 버전 4.1 최초 릴리스 |
| 고급 VR 그래픽 기법 장 추가. | *제 장 11 고급 VR 그래픽 기법* 페이지의 11-252 | 버전 4.1 최초 릴리스 |

표 A-9 버전 4.2 변경 사항

| 변경 | 위치 | 버전 |
|---|---|---|
| 라이팅 모범 사례 장 추가. | *제 장 8 실시간 3D 아트 모범 사례: 조명* 페이지의 8-123 | 버전 4.2 최초 릴리스 |
| 성능 분석 장 업데이트. | *제 장 3 성능 분석* 페이지의 3-22 | 버전 4.2 최초 릴리스 |
| Arm Mobile Studio 장 업데이트. | *제 장 13 ARM Mobile Studio* 페이지의 13-284 | 버전 4.2 최초 릴리스 |