Contents

Enlighten™ Integration Guide SDK 3.08

Preface

About this book ...................................................................................................................... 7
Feedback .............................................................................................................................. 9

Chapter 1  Introduction

1.1 Prerequisites ............................................................................................................ 1-11
1.2 Integration overview ............................................................................................... 1-12

Chapter 2  Exporting your scene to high-level build system compatible file formats

2.1 Creating a Default.paramset file .............................................................................. 2-14
2.2 Creating a World.scene file ...................................................................................... 2-15
2.3 Creating a MyGeometry.geom file .......................................................................... 2-16
2.4 Creating .pim mesh files ......................................................................................... 2-17

Chapter 3  Testing the precompute and precomputing the scene

3.1 Loading precompute UVs ....................................................................................... 3-21
3.2 Rendering with Enlighten UVs .............................................................................. 3-22

Chapter 4  Setting up the update manager

4.1 Implementing the texture allocator class ................................................................. 4-24
4.2 Initializing the update manager and configuring the texture output format .......... 4-26
4.3 Adding systems, setting a default material for meshes, and adding lighting ......... 4-27
4.4 Queuing updates ...................................................................................................... 4-28
Enlighten™ Integration Guide
SDK 3.08
Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

Release Information

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0308-00</td>
<td>10 February 2017</td>
<td>Non-Confidential</td>
<td>First release</td>
</tr>
</tbody>
</table>

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement covering this document with ARM, then the 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.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU 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/about/trademark-usage-guidelines.php

Copyright © 2017, ARM Limited or its affiliates. All rights reserved.


110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.
Product Status
The information in this document is Final, that is for a developed product.

Web Address
http://www.arm.com
Preface

This preface introduces the *Enlighten™ Integration Guide SDK 3.08*. It contains the following:

About this book

This is a guide to integrating Enlighten in your engine. The guide uses a simple scene as an example, and provides instructions and example code for you to apply the integration into your own scene.

Product revision status

The rmpn identifier indicates the revision status of the product described in this book, for example, r1p2, where:

r Identifies the major revision of the product, for example, r1.
pn Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This resource is for users of UE4 who are beginning to use Enlighten.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This guide introduces the steps required to integrate Enlighten into your game engine. Before you begin to work through these steps, ensure that you have met the necessary prerequisites.

Chapter 2 Exporting your scene to high-level build system compatible file formats
How to export scene data to appropriate file types.

Chapter 3 Testing the precompute and precomputing the scene
How to test the precompute for your scene in GeoRadiosity and how to precompute the scene in your engine.

Chapter 4 Setting up the update manager
How to set up the update manager for Enlighten to solve global illumination at runtime.

Chapter 5 Rendering with Enlighten lightmaps
How to render the scene by reusing the vertex shader for visualizing UVs, updating the pixel shader, and passing the mesh color to the g_Colour constant.

Chapter 6 Supporting multiple systems
How to perform and test automatic system generation that splits scenes into systems.

Chapter 7 Lighting probe-lit objects
How to light object with Enlighten probes.

Chapter 8 Adding support for Probe Radiosity systems
How to add support for Probe Radiosity systems.

Chapter 9 Adding cubemap support for creating dynamic specular reflections
How to add Enlighten cubemaps for creating dynamic specular reflections.

Chapter 10 Summary
A summary of the steps outlined in this guide.

Appendix A Revisions
This appendix describes the technical changes between released issues of this guide.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the ARM Glossary for more information.
Typographic conventions

*italic*
Introduces special terminology, denotes cross-references, and citations.

**bold**
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

*monospace*
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

**monospace**
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

*monospace italic*
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

**monospace bold**
Denotes language keywords when used outside example code.

<&and;>
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

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

**SMALL CAPITALS**
Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, *IMPLEMENTATION DEFINED*, *IMPLEMENTATION SPECIFIC*, *UNKNOWN*, and *UNPREDICTABLE*.

Additional reading

**ARM publications**

**Other publications**
Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:
- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:
- The title Enlighten Integration Guide SDK 3.08.
- The number ARM 100835_0308_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.
Chapter 1
Introduction

This guide introduces the steps required to integrate Enlighten into your game engine. Before you begin to work through these steps, ensure that you have met the necessary prerequisites.

This is a beginner-level guide to integrating the Enlighten SDK into your game engine to enable real-time global illumination. Enlighten produces indirect lighting known as radiosity that is used to solve global illumination in your game engine. This guide does not explain how to generate the equivalent of offline baked lightmaps.

By the end of this guide you will be able to perform a basic integration of the main parts of Enlighten into your engine for a small scene. Enlighten scales to support large scenes and large worlds, but for simplicity this guide describes integration for a small scene.

Follow the tasks in this guide sequentially to configure and integrate Enlighten. The guide provides examples of code for you to use. To help you move your integration to larger scenes in your game engine, links to appropriate reference content are provided.

This guide references some advanced integration topics. These topics are not covered in depth but are provided to give you a starting point towards adapting and customizing your integration to your own requirements.

It contains the following sections:
• 1.1 Prerequisites on page 1-11.
• 1.2 Integration overview on page 1-12.
1.1 Prerequisites

This guide requires that you have prepared a simple scene, that you have installed the Enlighten SDK, and that you have access to the Enlighten high-level build system (HLBS), Forge, and GeoRadiosity. Before you integrate Enlighten, the following pre-requisites must be met:

Prepare a small scene

The integration method described in this guide is for integrating Enlighten for a small scene. A small scene precomputes quickly and is easier to debug if errors occur.

An example of a small scene is a street with buildings and some basic scenery.

Install tools

The Enlighten SDK is bundled with extra tools for evaluating and exploring Enlighten without integrating it into your engine. The high-level build system (HLBS), Forge, and GeoRadiosity are used to debug each step of your integration and provide you with feedback on the working state of your code. Install these tools to follow the instructions in this guide:

High-level build system

The HLBS uses the GeoPrecompute tool to precompute Enlighten runtime data, which consists of a mixture of XML and binary files that are exported from a scene. Enlighten also provides a low-level build system. Only consider using the low-level API if you have a specific requirement that the HLBS cannot meet. In all other cases, use the HLBS.

Note

Shipping the precompute libraries requires a special license agreement.

Forge

Forge is a lighting and material editor. Forge is designed for small scenes only, it does not scale to larger scenes. It can be used stand-alone, or you can import your scene directly from 3DS Max or Maya to visualize Enlighten. Also, scene data can be imported in the HLBS format to provide visualizations that can help debug both the setup of your scene and the integration into your engine.

Geomerics recommend running through the process of setting up a scene in Forge, because this helps provide an understanding of the workflow that is required in your tools. Because Forge requires no integration work, it is also a very useful tool for artists to explore Enlighten. Furthermore, it can provide your team with a better understanding of Enlighten and how it can be used in your game and in your engine.

GeoRadiosity

GeoRadiosity is a debugging tool. It contains several tools for visualizing data produced by Enlighten. In this guide GeoRadiosity is used to verify an exported scene.

Related information

Installing Enlighten

For a detailed description of the file format that the HLBS uses, see Defining a scene for the high-level build system.
1.2 Integration overview

Enlighten must be integrated into two parts of your engine. After configuring Enlighten in your scene, you must run the Enlighten precompute before you can generate dynamic runtime data.

Enlighten is integrated into the following parts of your engine:

- The editor that you use to lay out your scene.
- The rendering code of your engine.

The instructions in this guide explain how to integrate Enlighten into these parts.

This guide begins with applying Enlighten lightmaps to your scene. Probe lighting and reflection captures are introduced later in the guide.

In summary, you must provide data from a complete scene as input to Enlighten. This is done in your level editor.

Enlighten produces the following types of output:

- A UV coordinate stream that is used at render time to correctly sample the lightmaps that Enlighten produces at runtime for lightmap lit geometries.
- The precomputed Enlighten radiosity data, which is split into different files. These files must be packaged with your game data and provided to Enlighten at runtime.

Using the workflow that is outlined in this guide, after a successful precompute you can focus on generating dynamic runtime data.
Chapter 2
Exporting your scene to high-level build system compatible file formats

How to export scene data to appropriate file types.

To integrate Enlighten into a scene, you must export the scene data as files that Enlighten can read. A scene directory represents a scene.

The following files are required in the scene directory for Enlighten integration:

- The `Default.paramset` file. This file defines parameters of Enlighten systems, such as the size of the lightmap output pixels.
- `World/World.scene`. An XML file that references the instances of your exported geometries. Each instance has a transform to place the object into the world.
- A folder for each geometry, containing a file of the same name with the `.geom` extension. For example, `MyGeometry/MyGeometry.geom`. The `.geom` file is an XML file that references the mesh files and defines the geometry.
- A `.pim` mesh file for each mesh. `.pim` files are serialized binary files of the `IPrecomputeInputMesh` class.

It contains the following sections:

- 2.1 Creating a `Default.paramset` file on page 2-14.
- 2.2 Creating a `World.scene` file on page 2-15.
- 2.3 Creating a `MyGeometry.geom` file on page 2-16.
- 2.4 Creating `.pim` mesh files on page 2-17.
2.1 Creating a Default.paramset file

The Default.paramset file defines the parameters of Enlighten systems, including the size of the lightmap output pixels.

Use the following code to create a new Default.paramset file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<parameterSetList version="1">
  <parameterSet name="High" id="3"
    outputPixelSize="<your-value>
    clusterSize="<your-value"
    samplesPerCluster="32"
    irradBudget="64"
    irradQuality="8192"
    environmentResolution="8"/>
</parameterSetList>
```

You must change the values of the following parameters to suit the scale of your game:

**outputPixelSize**

The value of outputPixelSize determines the resolution of Enlighten light maps. A higher value decreases the resolution and improves performance, but reduces lighting quality. A lower value decreases the resolution and improves lighting quality, but reduces performance. Setting the value too low can cause the precompute to fail. For best results, load the geometry into Forge and use the Enlighten Scene Mark-Up Wizard to find a suitable setting for your scene.

**clusterSize**

After the outputPixelSize is defined, set the value of clusterSize to twice the value of the outputPixelSize.
2.2 Creating a World.scene file

The World.scene file is an XML file that references all instances of geometry in a scene.

The World.scene file references instances of geometry using instance tags. Multiple instances can point to the same geometry. Multiple instances can point to the same geometry.

Each instance has a name and an instance global unique identifier (GUID). GUIDs are 128-bit hexadecimal values. You can assign GUID values and then use them whenever you reference an instance.

The GUID is a global identifier and the same namespace is used for all types of objects, including instances, meshes, and materials.

The following code shows an example of a World.scene file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<scene name="World" version="1" axes="-x+y+z">

    <instance name="SimpleScene_Floor_0" instanceGuid="00000000000000000000000000000001" systemId="System_0" systemGuid="6f46be6c000000000000000000000000" paramSet="High"
                geometry="Floor" type="Radiosity" position="370.000000 0.000000 40.000000"
                rotation="0.000000 0.000000 0.000000 1.000000"/>

    <instance name="SimpleScene_Wall_0" instanceGuid="00000000000000000000000000000002" systemId="System_0" systemGuid="6f46be6c000000000000000000000000" paramSet="High"
                geometry="Wall" type="Radiosity" position="100.000000 0.000000 150.000000"
                rotation="0.000000 0.000000 0.000000 1.000000"/>

    <instance name="SimpleScene_MyGeometry_0" instanceGuid="00000000000000000000000000000003" systemId="System_0" systemGuid="6f46be6c000000000000000000000000" paramSet="High"
                geometry="MyGeometry" type="Radiosity" position="340.000000 70.000000 130.000000"
                rotation="0.000000 0.000000 0.000000 1.000000"/>

</scene>
```

In the example code, each instance uses Enlighten lightmaps. This is done by defining a paramset with the Radiosity type. For this guide, the paramset is named High.

The scene in the example code defines one system, named System_0.
2.3 Creating a MyGeometry.geom file

The .geom file is an XML file that references the mesh files and defines the geometry.

Use the following example code to create a MyGeometry.geom file in a directory called MyGeometry. Ensure that you update the guid attribute. The folder name and the name of the .geom file must match. Do this for all geometries referenced in the .scene file.

```xml
<?xml version="1.0" encoding="utf-8"?>
<geom name="MyGeometry" version="3" simpNumIterationsPerSimp="500" simpMaxNumSimps="250"
 simpUsePixelUnits="true">
  <mesh
    name="MyMesh"
    guid="7444ac65000000000000000000000000"
    filename="MyMesh.pim"
    direct="true"
    indirect="true"
    target="true"
    lodLevel="0"
    scaleCharts="true"
    simpMode="simplifyUsingUvs"/>
</geom>
```

In this code, the geometry uses the Enlighten AutoUV algorithm to automatically generate Enlighten UVs. To automatically generate Enlighten UVs, you must assign `simplifyUsingUvs` to the `simpMode` attribute. Geomerics recommend using AutoUVs for integrating Enlighten.

A .geom file references one or more mesh files, and these are serialized objects of the IPrecompInputMesh class. You only need to add vertices and triangles to an object of this class.

**Meshes and geometries**

The .geom file format uses instances of geometries, which consist of meshes. A common problem is correctly determining which primitives in your editor or engine you should map to the mesh or geometry.

For this guide, Geomerics recommend that you use the most straightforward mapping. For example, if your editor only exposes one kind of object, then export them as a geometry that contains only one mesh.
2.4 Creating .pim mesh files

.pim files are serialized binary files of the IPrecomputeInputMesh class.

Your editor uses a custom format to describe your geometry, commonly represented as a list of triangles referencing a list of vertices.

Every mesh must be exported as a .pim file.

The following example code exports a cube and a plane. The code uses the AddRectangle() helper function, which adds the correct vertices and triangles to a temporary array, which are sent to an IPrecomputeInputMesh instance.

```cpp
#include <string>
#include <Enlighten3/EnlightenUtils.h>
#include <GeoCore/GeoArray.h>
#include <GeoCore/GeoAutoPtr.h>

// Helper function to add a rectangle to a list of vertices and faces
bool AddRectangle(
    Geo::GeoArray<Enlighten::PrecompInputVertex>& vertices,
    Geo::GeoArray<Enlighten::PrecompInputFace>& faces,
    const Geo::GeoPoint3& origin, const Geo::GeoVector3& dir1, const Geo::GeoVector3& dir2,
    const Geo::GeoVector3& normal,
    const Geo::GeoPoint2& uvOrigin, const Geo::GeoPoint2& uvExtent
) {
    const Geo::s32 vertexOffset = vertices.GetSize();

    Enlighten::PrecompInputVertex v0;
    v0.m_Position = origin - dir1 - dir2;
    v0.m_Normal = normal;
    v0.m_AlbedoUV = Geo::GeoPoint2(0.f, 0.f);
    v0.m_ChartUV = uvOrigin;
    if (!vertices.Push(v0)) { return false; }

    Enlighten::PrecompInputVertex v1;
    v1.m_Position = origin + dir1 - dir2;
    v1.m_Normal = normal;
    v1.m_AlbedoUV = Geo::GeoPoint2(0.f, 0.f);
    v1.m_ChartUV = Geo::GeoPoint2(uvOrigin.X + uvExtent.X, uvOrigin.Y);
    if (!vertices.Push(v1)) { return false; }

    Enlighten::PrecompInputVertex v2;
    v2.m_Position = origin - dir1 + dir2;
    v2.m_Normal = normal;
    v2.m_AlbedoUV = Geo::GeoPoint2(0.f, 1.f);
    v2.m_ChartUV = Geo::GeoPoint2(uvOrigin.X, uvOrigin.Y + uvExtent.Y);
    if (!vertices.Push(v2)) { return false; }

    Enlighten::PrecompInputVertex v3;
    v3.m_Position = origin + dir1 + dir2;
    v3.m_Normal = normal;
    v3.m_AlbedoUV = Geo::GeoPoint2(1.f, 1.f);
    v3.m_ChartUV = Geo::GeoPoint2(uvOrigin.X + uvExtent.X, uvOrigin.Y + uvExtent.Y);
    if (!vertices.Push(v3)) { return false; }

    if (!faces.Push(Enlighten::PrecompInputFace(vertexOffset+0, vertexOffset+1, vertexOffset+2, 0))) { return false; }
    if (!faces.Push(Enlighten::PrecompInputFace(vertexOffset+1, vertexOffset+3, vertexOffset+2, 0))) { return false; }
}

bool WriteMesh(std::string filename) {
    // We build our mesh from multiple rectangles. In your code, you'd probably have access to the
    // data required here in some other format. So you will have to implement some translation
    // code that maps your data into the format the Enlighten needs.
    Geo::GeoRefReleasePtr<Enlighten::IPrecomputeInputMesh>
    mesh(Enlighten::IPrecomputeInputMesh::Create());

    Geo::GeoArray<Enlighten::PrecompInputVertex> vertices;
    Geo::GeoArray<Enlighten::PrecompInputFace> faces;
```
if (!AddRectangle(vertices, faces, Geo::GeoPoint3(0.f, 0.f, 0.f), Geo::GeoVector3(3.f, 0.f, 0.f), Geo::GeoVector3(0.f, 0.f, 1.f), Geo::GeoPoint2(0.f, 0.f), Geo::GeoPoint2(0.65f, 0.65f))) { return false; }

// And the box on top
if (!AddRectangle(vertices, faces, Geo::GeoPoint3(0.f, 0.f, 2.f), Geo::GeoVector3(1.f, 0.f, 0.f), Geo::GeoVector3(0.f, 1.f, 0.f), Geo::GeoVector3(0.f, 0.f, 1.f), Geo::GeoPoint2(0.7f, 0.7f), Geo::GeoPoint2(0.3f, 0.3f))) { return false; }
if (!AddRectangle(vertices, faces, Geo::GeoPoint3(1.f, 0.f, 1.f), Geo::GeoVector3(0.f, 1.f, 0.f), Geo::GeoVector3(0.f, 0.f, 1.f), Geo::GeoVector3(1.f, 0.f, 0.f), Geo::GeoPoint2(0.f, 0.7f), Geo::GeoPoint2(0.3f, 0.3f))) { return false; }
if (!AddRectangle(vertices, faces, Geo::GeoPoint3(-1.f, 0.f, 1.f), Geo::GeoVector3(0.f, -1.f, 0.f), Geo::GeoVector3(0.f, 0.f, 1.f), Geo::GeoVector3(-1.f, 0.f, 0.f), Geo::GeoPoint2(0.35f, 0.7f), Geo::GeoPoint2(0.3f, 0.3f))) { return false; }
if (!AddRectangle(vertices, faces, Geo::GeoPoint3(0.f, 1.f, 1.f), Geo::GeoVector3(-1.f, 0.f, 0.f), Geo::GeoVector3(0.f, 0.f, 1.f), Geo::GeoVector3(0.f, 1.f, 0.f), Geo::GeoPoint2(0.7f, 0.35f), Geo::GeoPoint2(0.3f, 0.3f))) { return false; }
if (!AddRectangle(vertices, faces, Geo::GeoPoint3(0.f, -1.f, 1.f), Geo::GeoVector3(1.f, 0.f, 0.f), Geo::GeoVector3(0.f, 0.f, 1.f), Geo::GeoVector3(0.f, -1.f, 0.f), Geo::GeoPoint2(0.7f, 0.f), Geo::GeoPoint2(0.3f, 0.3f))) { return false; }

if (!mesh->AddVertices(vertices.Begin(), vertices.End())) { return false; }
if (!mesh->AddFaces(faces.Begin(), faces.End())) { return false; }

// Save generated mesh
Geo::SaveInterface(mesh.GetPtr(), filename.c_str());
Chapter 3
Testing the precompute and precomputing the scene

How to test the precompute for your scene in GeoRadiosity and how to precompute the scene in your engine.

**Testing the precompute in GeoRadiosity**

To precompute the scene and preview it in GeoRadiosity:

1. In GeoRadiosity, select **Scene > Load Scene**. The scene will render in GeoRadiosity.

   --- **Note** ---

   The scene is shown in a debug view. This shows all of the faces with colors based on their orientation. If the orientation of the scene does not match what is in your engine, add the `axes` attribute to the scene tag in the `.scene` file, for example, `axes="+x-y+z"`. This describes the axis permutation for the scene. This has no impact on the precompute, but makes handling the scene in GeoRadiosity easier.

2. In GeoRadiosity, select **Precompute > Full precompute** > .

   A progress window opens, and closes when the precompute finishes.

3. Add some lights to the scene using the Lighting tab.

   When the precompute finishes, the scene is rendered with a gray material. This is because no materials have been exported.
If the precompute is successful, the scene is lit correctly with direct and indirect lighting. If this is not the case, the output pixel size is incorrect and must be changed.

4. In GeoRadiosity, select the **Chart texture** render mode. This mode renders every pixel as a square and assigns each UV chart a different color. If the pixels appear too large or look distorted then change the output pixel size.

To find a suitable output pixel size, import the scene in Forge and use the Forge Enlighten Markup Wizard. When you have chosen a suitable output pixel size, copy the value into the `Default.paramset` file and precompute the scene again in GeoRadiosity.

The precompute creates a larger number of files in the __Build_World__ directory.

If the precompute fails, see the precompute log for details.

**Precomputing the scene in your engine**

To instruct your engine editor to precompute, execute the GeoPrecompMonitor binary as shown in the following command:

```
GeoPrecompMonitor.exe /t:Precompute /p:ToolBinaries=<path to SDK bin directory> /p:InputScene=C:\myscene\myscene.scene /fl /flp:LogFile=C:\myscene\precompute.log
```

It contains the following sections:

- 3.1 Loading precompute UVs on page 3-21.
- 3.2 Rendering with Enlighten UVs on page 3-22.
3.1 Loading precompute UVs

Enlighten produces a new UV stream for all geometry lit by lightmaps. To render a mesh successfully in your engine, you must extract the Enlighten UV data and render it with your mesh.

For each system, there is a serialized object in the IPrecomputeLoaderSystem in the __Build_World__/precomputeloader/ directory. In the example code, this is called System_0.system.sse because the system is named System_0.

The precompute loader system stores the UV transform for each instance. This allows you to store the UV stream for each geometry and only store the transform for every instance of this geometry.

Use the following code to load an IPrecomputeLoaderSystem and retrieve the UV transform:

```cpp
void RetrieveEnlightenUVTransforms(const std::string& exportFolder, const std::string& sceneName, const std::string& systemId)
{
    Geo::GeoAutoReleasePtr<Enlighten::IPrecomputeLoaderSystem> systemData = Geo::LoadInterfaceCompressed<Enlighten::IPrecomputeLoaderSystem>((exportFolder + "/__Build_" + sceneName + "__/precomputeloader/" + systemId + "+.system.sse").c_str());

    Geo::s32 numInstances = systemData->GetUvTransformsCount();
    for (Geo::s32 i = 0; i < numInstances; ++i)
    {
        const Geo::s32 lod = 0; // for this example we will load the uv transforms at lod 0
        // The UV transforms are stored per instance.
        systemData->GetUvTransform(lod, i);
    }
}
```

To load the UV stream, load the IPrecomputeLoaderGeometry object. Every precompute loader geometry is also stored in __Build_World__/precomputeloader/. In the example code, this is called MyGeometry_High.geometry. The filename is always the geometry name and the paramset name that the instance uses joined together.

Use the following code to retrieve the UV coordinates, then add them to your rendered mesh:

```cpp
void RetrieveEnlightenGeometryUVs(const std::string& exportFolder, const std::string& sceneName, const std::string& geometryId, const std::string& paramset)
{
    Geo::GeoAutoReleasePtr<Enlighten::IPrecomputeLoaderGeometry> geometryData = Geo::LoadInterfaceCompressed<Enlighten::IPrecomputeLoaderGeometry>((exportFolder + "/__Build_" + sceneName + "__/precomputeloader/" + geometryId + "+.geometry").c_str());

    Geo::s32 numMeshes = geometryData->GetMeshCount();
    for (Geo::s32 i = 0; i < numMeshes; ++i)
    {
        const Geo::s32 lod = 0; // for this example we will load the uv's at lod 0
        // The UVs are stored in an array, one per mesh. This array is identical for all instances of one geometry
        geometryData->GetUvs(lod, i);
    }
}
```
3.2 Rendering with Enlighten UVs

Enlighten UVs are rendered by implementing a simple debug mode and transforming UVs with the per instance UV transform.

Implementing a simple debug mode verifies that Enlighten UVs are loaded correctly.

A simple debug mode is useful in later stages of integration because it allows you to visualize whether a scene has been correctly set up for Enlighten.

To implement a simple debug mode, you must write some code that creates a texture with a checkerboard pattern. The texture size is obtained from the IPrecomputeLoaderSystem. To determine the width and height of the lightmap size, use systemData->GetHeight() and systemData->GetWidth(). Disable texture filtering so that the checkerboard pattern is clearly visible.

Use the following vertex shader code to transform the UVs with the previously loaded per instance UV transform. Bind it to the g_LightmapUvLinearTransform constant.

```c
/// The output of the vertex shader (and input to the pixel shader) 
struct VS_Out 
{
    float4 ProjectedPosition : POSITION;
    float4 WorldPosition : TEXCOORD0;
    float3 Normal : TEXCOORD1;
    float2 LightmapUV : TEXCOORD2;
};

 rencont
VS_Out VS(float4 pos : POSITION, float4 normal : NORMAL, float2 lightmapUV : TEXCOORD0)
{
    VS_Out output;
    output.ProjectedPosition = mul(pos, g_ModelToClip);
    output.WorldPosition = mul(pos, g_ModelToWorld);
    output.Normal = float3(mul(normal, inverse(transpose(g_ModelToWorld))));
    //transform the lightmap UVs
    output.LightmapUV = float2(
        g_LightmapUvLinearTransform.x * lightmapUV.x + g_LightmapUvLinearTransform.z,
        g_LightmapUvLinearTransform.y * lightmapUV.y + g_LightmapUvLinearTransform.w);
    return output;
}
```

For the debug view, add a simple pixel shader that samples the texture that is bound to g_Checkerboard using the interpolated UVs.

```c
float4 PS(VS_Out input) : COLOR
{
    // sample the lightmap
    return tex2D(g_Checkerboard, input.LightmapUV);
}
```

To verify your output, compare your rendered output to the Chart texture render mode in GeoRadiosity. To navigate to the Chart texture render mode in GeoRadiosity, select Rendering > Charts > Chart texture. The visualizations should be close to identical, even though the pixel colors may be different. If they do not look close to identical, then you are not using the correct UV stream.
Chapter 4
Setting up the update manager

How to set up the update manager for Enlighten to solve global illumination at runtime.

The update manager is part of the high-level runtime. The update manager must be configured for Enlighten to solve global illumination during runtime in your engine.

It contains the following sections:

• 4.1 Implementing the texture allocator class on page 4-24.
• 4.2 Initializing the update manager and configuring the texture output format on page 4-26.
• 4.3 Adding systems, setting a default material for meshes, and adding lighting on page 4-27.
• 4.4 Queuing updates on page 4-28.
4.1 Implementing the texture allocator class

The update manager calls the texture allocator class to create the texture that Enlighten provides the lightmap data for.

You must implement a texture allocator class. Because every engine has its own way of managing textures, the update manager uses the texture allocator to obtain textures with the correct interface so that it can upload new data when ready.

Use the following code to implement a single virtual allocator:

```cpp
class MyTextureAllocator : public Enlighten::IGpuTextureAllocator
{
public:
    // EngineTexture is a resource-managing object for your own texture implementation. We assume that the textureFormat is always FP16.
    EngineTexture texture(width, height);
    return GEO_NEW_ARGS(MyTexture, (texture, width, height));
};
```

The code in this guide assumes a texture format of FP16, although Enlighten supports other formats.

With the allocator implemented, you must instruct the allocator to return an instance of the IGpuTexture interface.

Use the following code to do this:

```cpp
// Implement the IGpuTexture and IGpuTextureUpdater interface in one class. The IGpuTexture interface is very shallow.
class MyTexture : public IGpuTexture, public IGpuTextureUpdater
{
public:
    // EngineTexture is a resource-managing object for your own texture implementation. We assume that the textureFormat is always FP16.
    EngineTexture texture(width, height);
    return GEO_NEW_ARGS(MyTexture, (texture, width, height));
};
```

4.2 Setting up the update manager
m_Height(height)
{
  m_Memory = malloc(m_Height * GetRowPitch());
}

MyTexture::Update()
{
  // We need a way to upload or copy from the CPU memory to our texture.
  m_Texture.upload(m_Memory);
}
4.2 Initializing the update manager and configuring the texture output format

The update manager must be initialized as a multi-threaded version and with a definition of the texture output format.

Use the following code to instantiate a multi-threaded version of the update manager and configure the texture output format to be FP16:

```cpp
UpdateManagerProperties props;
props.m_IrradianceOutputFormat = OUTPUT_FORMAT_FP16;
props.m_SolveType = ENLIGHTEN_IRRADIANCE_ONLY;
props.m_TextureAllocator = GEO_NEW(MyTextureAllocator);
Enlighten::IUpdateManager* updateManager = MultithreadCpuUpdateManager::Create(props, NULL);
```
4.3 Adding systems, setting a default material for meshes, and adding lighting

The update manager must include systems, a default material, lights, and an emissive environment.

Add the systems that need to be solved. To do this, retrieve the contents from the previously loaded IPrecomputeLoaderSystem and pass them along, as shown in the following code:

```cpp
Enlighten::RadSystemCore* radCore = systemData->GetRadiosityCore();
Enlighten::InputWorkspace* inputWorkspace = systemData->GetInputWorkspace();
Enlighten::ISystemSolutionSpace* solutionSpace = updateManager->AllocateSystemSolutionSpace(radCore);
// Keep track so we can release the memory.
m_SolutionSpaces.insert(key, solutionSpace);
sys = updateManager->AllocateSystem(inputWorkspace, NULL);
sys->SetSystemSolutionSpace(solutionSpace);
updateManager->EnqueueAddSystem(sys);
```

Use the following code to set a default material for all meshes:

```cpp
const Enlighten::ClusterAlbedoWorkspaceMaterialData* CAWMaterialData = systemData->GetClusterAlbedoWorkspace();
Geo::u32 dynamicMaterialWorkspaceSize = Enlighten::CalcDynamicMaterialWorkspaceSize(CAWMaterialData);
void* dynamicMaterialWorkspaceMemory = GEO_ALIGNED_MALLOC(dynamicMaterialWorkspaceSize, 16);
dynamicMaterialWorkspace = Enlighten::CreateDynamicMaterialWorkspace(CAWMaterialData, dynamicMaterialWorkspaceMemory);
void* albedoWorkspaceMemory = GEO_ALIGNED_MALLOC(CalcMaterialGuidsLookupWorkspaceSize(CAWMaterialData), 16);
Enlighten::SetMaterialAlbedoColour(dynamicMaterialWorkspace, CAWMaterialData, GeoGuid::Invalid, GeoGuid::Invalid, GeoGuid::Invalid, VConstruct(0.7f, 0.7f, 0.7f, 1.f), albedoWorkspaceMemory);
Geo::u32 albedoBuffersSize = Enlighten::CalcAlbedoBufferSize(inputWorkspace);
void* albedoBufferMemory = GEO_ALIGNED_MALLOC(albedoBuffersSize, 16);
albedoBuffer = Enlighten::CreateAlbedoBuffer(inputWorkspace, albedoBufferMemory); // You need to hold on to this
Enlighten::InitialiseAlbedoBufferFromMaterialWorkspace(dynamicMaterialWorkspace, CAWMaterialData, albedoBuffer);
Enlighten::SystemAlbedoData albedoData;
albedoData.m_AlbedoBuffer = albedoBuffer;
albedoData.m_EmissiveBuffer = NULL;
albedoData.m_TransparencyBuffer = NULL;
Enlighten::EnqueueSetObjectParameter<Enlighten::BaseSystem, Enlighten::SystemAlbedoData>(
    updateManager, systemGuid, &Enlighten::BaseSystem::SetAlbedoData, albedoData);
```

Use the following code to add some light and an emissive environment:

```cpp
GeoGuid environmentGuid = GeoGuid::Create(0, 1);
Enlighten::EnqueueSetObjectParameter<Enlighten::BaseSystem, Geo::GeoGuid>(
    updateManager.GetPtr(), inputWorkspace->m_MetaData.m_SystemId, 
    &Enlighten::BaseSystem::SetEmissiveEnvironment, environmentGuid);
Geo::s32 resolution = 8; // This needs to match definition in .paramset file
GeoAutoPtr<v128, GeoDeleteArrayDestructor<v128>> values(GEO_NEW_ARRAY_ALIGNED(v128, 6 * resolution * resolution, 16));
for (s32 i = 0; i < 6 * resolution * resolution; i++)
{
    values[i] = VConstruct(0.7f, 0.7f, 0.7f, 1.f); // Hard coded just for test purposes.
}
updateManager->EnqueueUpdateEmissiveEnvironment(environmentGuid, resolution, values.GetPtr());
```
4.4 Queuing updates

The update manager must be set to queue updates.

Use the following code to set the update manager to queue updates:

```cpp
updateManager->EnqueueSetAllUpdateCounters(Enlighten::INFINITE_UPDATES);
```

In your render thread, you must call `updateManager->Update()`, once every frame.
Chapter 5
Rendering with Enlighten lightmaps

How to render the scene by reusing the vertex shader for visualizing UVs, updating the pixel shader, and passing the mesh color to the g_Colour constant.

When the update manager is correctly set up, you can render your scene using the lightmaps that have been generated.

To do this:
1. Reuse the vertex shader for visualizing UVs.
2. Update the pixel shader to sample from the lightmap.
3. The color of the mesh is passed in the g_Colour constant.

To implement these steps, use the following code:

```cpp
float4 PS(VS_Out input) : COLOR
{
    float4 result;
    // sample the lightmap
    result = tex2D(g_Lightmap, input.LightmapUV);
    // tint by the colour of the mesh
    result = result * g_Colour;
}
```

When you only read from the texture in your shader and output the color, without any other processing, you see a similar output in your engine to Forge when you only add a gray environment. Forge applies gamma correction to the final image and this might make the scene appear different. However, the distribution of light in your scene still resembles what you see in Forge.

Because Enlighten only uses gray materials and no lights have been added to the scene, the scene still does not look correct.
It contains the following sections:

- 5.1 Controlling albedo on page 5-31.
- 5.2 Adding light sources on page 5-33.
5.1 Controlling albedo

Enlighten can be configured to control the material property albedo to influence reflected light.

When light reflects off a surface, its color and intensity changes depending on the physical properties of the surface. The albedo is the proportion of incident light reflected by the surface.

In this guide, all materials have been set to gray. To make full use of your mesh textures, this must be changed.

For dynamic materials that can change efficiently at runtime, they must be set to use dynamic albedo.

Related information

For more information on setting up GUIDs for the precompute see Material IDs and GUIDs.

Albedo, emissive, and transparency are material components that can be controlled within Enlighten. Emissive and transparency are set the same way as albedo.

This section contains the following subsections:

- 5.1.1 Using texture albedo on page 5-31.
- 5.1.2 Using dynamic albedo on page 5-32.

5.1.1 Using texture albedo

The Enlighten AlbedoBuffer allows you to apply an albedo texture across a system.

Commonly, the color of materials is determined using complex blending of high-resolution textures. This can result in a varying albedo across the material. Enlighten allows you to initialize the AlbedoBuffer with texture data to get a more accurate representation of albedo across the system.

This is done by using the light map atlas, its UVs, and the material rendering capability of your engine. This enables you to use any material evaluation systems your engine uses.

To use texture albedo, you must prepare a single albedo texture for the whole system. To do this, render the light mapped geometry for a system, in light map UV space, into a render target and then read back the result into linear CPU memory. The final dimensions of this CPU texture can be at full or half of the Enlighten light map resolution. For best results, render at a higher resolution and then downsample. The pixel format is expected to be in power of two color space. This is close to the sRGB curve and is cheaper to process but is not an exact match so you might want to manually encode to gamma 2.

Note

The code that is required to generate the albedo texture is engine-specific. Because of this, an example is not provided here. To test the basic functionality, fill a texture with a single color and check whether this is reflected in the output.

Initialize the AlbedoBuffer using the generated texture with the following code:

```c
s32 width, height;
Enlighten::GetTextureSize(m_RadSystemCore, Enlighten::TextureResolutionScale::Half, width, height);
Geo::s32 pixelSizeInBytes = 4; // pixel format is expected to be gamma 2 encoded 32-bit pixel format

// Initialise the Albedo/EmissiveBuffers with albedo colors from the rendered textures
Enlighten::InputTextureSamplerParameters texParams;
texParams.m_TextureWidth = width;
texParams.m_TextureHeight = height;
texParams.m_TexturePitch = width*pixelSizeInBytes;
texParams.m_TextureData = m_AlbedoTexture;
if (m_AlbedoBuffer && m_AlbedoTexture)
{
    Enlighten::InitialiseAlbedoBufferFromTexture(m_DynamicMaterialWorkspace, m_CAWMaterialData, &texParams,
    Enlighten::TextureResolutionScale::Half, m_AlbedoBuffer);
}
```
Although the first parameter to the Enlighten::InitialiseAlbedoBufferFromTexture function is a DynamicMaterialWorkspace, any albedo colors set up in the workspace are ignored and the colors from the albedo texture are used instead. The reasons for this are:

• InitialiseAlbedoBufferFromTexture only uses the texture albedo for materials which are static. Dynamic materials are set from the colors in the DynamicMaterialWorkspace.
• By default, all materials in the DynamicMaterialWorkspace are set to be static unless explicitly initialized as dynamic.

The albedos of dynamic materials in the DynamicMaterialWorkspace take precedence over the colors in the albedo texture. This means that if you want to use the colors in the DynamicMaterialWorkspace for certain materials, you must mark each of those materials in the workspace as dynamic before the call to InitialiseAlbedoBufferFromTexture.

```
Enlighten::InitialiseMaterialAlbedoAsDynamic(
    system.m_DynamicMaterialWorkspace,
    system.m_CAWMaterialData,
    GeoGuid::Invalid,
    GeoGuid::Invalid,
    materialGuid,
    system.m_GuidLookupWorkspaceMemory);
```

### 5.1.2 Using dynamic albedo

Enlighten uses dynamic albedo to update the albedo of dynamic object at runtime. It is too computationally expensive to update the texture albedo at runtime. Instead, dynamic albedo updates the albedo of dynamic materials at runtime with a low performance overhead.

Use the following code to update each dynamic material in the DynamicMaterialWorkspace and update the AlbedoBuffer:

```
for (s32 m=0; m < m_NumDynamicMaterials; ++m)
{
    GeoGuid dynamicMaterialGuid = m_DynamicMaterialGuids[m];
    v128 newLinearMaterialColour = EvaluateLatestMaterialColour(dynamicMaterialGuid);
    Enlighten::SetMaterialAlbedoColour(
        system.m_DynamicMaterialWorkspace,
        system.m_CAWMaterialData,
        GeoGuid::Invalid,
        GeoGuid::Invalid,
        dynamicMaterialGuid,
        newLinearMaterialColour,
        system.m_GuidLookupWorkspaceMemory);
}
```

// Update the application albedo buffer with the latest dynamic material values.
Enlighten::UpdateAlbedoBuffer(
    system.m_DynamicMaterialWorkspace,
    system.m_CAWMaterialData,
    system.m_AlbedoBuffer);

// Set the albedo buffer on the UpdateManager system
Enlighten::SetAlbedoData(albedoData.m_AlbedoBuffer = system.m_AlbedoBuffer;
Enlighten::EnqueueSetObjectParameter<Enlighten::BaseSystem,
    Enlighten::SystemAlbedoData>(
    m_UpdateManager.GetPtr(), system.m_Id, &Enlighten::BaseSystem::SetAlbedoData, albedoData);
```

Calling UpdateAlbedoBuffer is only valid when the AlbedoBuffer has been initialized first. This allows you to preserve the high-quality static albedo that is already initialized from the albedo texture, but still have dynamic materials that are updated quickly at runtime.
5.2 Adding light sources

Light sources must be added to Enlighten so that they can contribute to the Enlighten radiosity.

To provide the light sources to Enlighten, you must register the scene lights in the update manager. These can also be removed.

The available light sources are directional, point, and spot lights.

The Enlighten SDK provides classes to support the registering of the light sources in the update manager. These are located in the Enlighten3/InputLighting directory.

This section contains the following subsections:
- 5.2.1 Adding directional lights on page 5-33.
- 5.2.2 Adding spot lights on page 5-34.
- 5.2.3 Adding point lights on page 5-34.
- 5.2.4 Removing a light source on page 5-34.

5.2.1 Adding directional lights

When adding a directional light to your scene, you must create an Enlighten light struct and add the data from the directional light.

For Enlighten, a directional light is defined using the struct Enlighten::DirectionalLight.

Set up a morning sun style of light using a directional light. To do this, add a directional light with the following settings:
- A direction vector gameLightDirection with an angle of 45 degrees pointing towards the scene. If you have a coordinate system that matches DX9 your vector is \{0.00 0.80 -0.59 0.00\}.
- An intensity vector gameLightIntensity to a light blue color.

For debugging reasons, leave out the visibility, meaning that the light is not blocked.

Use the following code to set up a light in the style of the sun:

```cpp
// Create an Enlighten light struct and populate with the data from your game light
Enlighten::DirectionalLight dirLight;
dirLight.m_Intensity = VConstruct(0.2, 0.2, 0.8);
dirLight.m_Direction = VConstruct(0.f, 0.8f, -0.59f, 0.f);
dirLight.m_VisibilityType = Enlighten::VisibilityFormat::BYTE_VISIBILITY;

// A guid that identifies this light
Geo::GeoGuid lightGuid = gameLightId;

// Enqueue a command to register the light in the update manager
Enlighten::UpdateLightCommandGeneric<Enlighten::DirectionalLight> updateLightCommand(lightGuid, dirLight, true, false, 0);
Enlighten::EnqueueCommand(updateManager.GetPtr(), updateLightCommand);
```

Now that you have light in your scene, you can add the precomputed directional visibility to your system. This data is passed when adding the system to the update manager.

The following code contains the necessary changes. When the precomputed directional visibility is added to the system, there is a change in the light bounce. Areas which looked unreasonably bright, now look more realistic and darker.

```cpp
// Load precomputed visibility
Geo::GeoFileString precompVisFilename = Geo::GeoFileString::Printf("%s\radiosity\%s.vis",
precomputeOutputPath.GetCString(), system.GetName());
precomputeDirectionalVisibility = Enlighten::ReadPrecomputedVisibilityDataFromFile(precompVisFilename.GetCString());

// Add system
Enlighten::RadSystemCore* radCore; // Load from disk (.rc)
Enlighten::InputWorkspace* inputWorkspace; // Load from disk (.iw)

system = updateManager->AllocateSystem(radCore, inputWorkspace, precomputeDirectionalVisibility);
updateManager->EnqueueAddSystem(system);
```
5.2.2 Adding spot lights

When adding a spot light to your scene, you must create a falloff table and an Enlighten light struct that contains the data of the light.

Use the following code to add spot lights to your scene:

```cpp
// Spot lights require a falloff table
Geo::GeoAutoPtr<Enlighten::InputLightFalloffTable> falloffTable = GEO_ALIGNED_NEW(Enlighten::InputLightFalloffTable, 16);

// Create an Enlighten light struct and populate with the data from your game light
Enlighten::Spotlight spotLight;
spotLight.m_Intensity = gameLightIntensity;
spotLight.m_Position = gameLightPosition;
spotLight.m_Direction = gameLightDirection;
spotLight.m_ConeAngle = gameLightConeAngle;
spotLight.m_InnerConeAngle = gameLightInnerConeAngle;
spotLight.m_Radius = gameLightRadius;
spotLight.m_CutOff = gameLightCutOff;
spotLight.m_FalloffTable = falloffTable;

// A guid that identifies this light
Geo::GeoGuid lightGuid = gameLightId;

// Enqueue a command to register the light in the update manager
Enlighten::UpdateLightCommandGeneric<Enlighten::Spotlight > updateLightCommand(lightGuid, spotLight, true, false, 0);
Enlighten::EnqueueCommand(updateManager.GetPtr(), updateLightCommand);
```

Spot lights require an Enlighten falloff table that describes the light falloff curve. The code uses the default Enlighten falloff table.

Visibility data is not added for spot lights. Enlighten uses your shadow map data if this is available.

5.2.3 Adding point lights

When adding a point light to your scene, you must create a falloff table and an Enlighten light struct that contains the data of the light.

Add point lights to the scene using the following code:

```cpp
// Point lights require a falloff table
Geo::GeoAutoPtr<Enlighten::InputLightFalloffTable> falloffTable = GEO_ALIGNED_NEW(Enlighten::InputLightFalloffTable, 16);

// Create an Enlighten light struct and populate with the data from your game light
Enlighten::PointLight pointLight;
pointLight.m_Intensity = gameLightIntensity;
pointLight.m_Position = gameLightPosition;
pointLight.m_Radius = gameLightRadius;
pointLight.m_CutOff = gameLightCutOff;
pointLight.m_FalloffTable = falloffTable;

// A guid that identifies this light
Geo::GeoGuid lightGuid = gameLightId;

// Enqueue a command to register the light in the update manager
Enlighten::UpdateLightCommandGeneric<Enlighten::PointLight > updateLightCommand(lightGuid, pointLight, true, false, 0);
Enlighten::EnqueueCommand(updateManager.GetPtr(), updateLightCommand);
```

5.2.4 Removing a light source

Light sources with light GUIDs can be efficiently updated or removed from Enlighten.

When a light source has been removed it does not contribute to Enlighten radiosity.

Use the following code to remove a light source:

```cpp
// A guid that identifies this light
Geo::GeoGuid lightGuid = gameLightId;
```
// Remove the light from update manager
updateManager->EnqueueRemoveLight(lightGuid);
Chapter 6
Supporting multiple systems

How to perform and test automatic system generation that splits scenes into systems.

Enlighten is made up of systems, each containing one lightmap. Enlighten provides automatic system generation to split scenes into appropriate systems.

An Enlighten system is the smallest unit that the Enlighten precompute and runtime handles.

Enlighten produces one lightmap for each system. So for a simple scene with one system, only one lightmap is produced. Because Enlighten lightmaps have a maximum size, larger scenes must be split into several systems. Splitting a scene into multiple systems also allows for better performance at runtime.

Enlighten provides automatic system generation that splits scenes into systems. To perform automatic system generation in your editor, you must perform a precompute using the high-level build system (HLBS) and a new .scene file must be read back. Automatic system generation can be tested in GeoRadiosity.

Related information

Internally, the voxelization method uses the functionality of the low-level API function CreateSystemsByVoxelisation.

It contains the following sections:

• 6.1 Testing automatic system generation using GeoRadiosity on page 6-37.
• 6.2 Performing automatic system generation on page 6-38.
• 6.3 Using multiple systems in the high-level runtime on page 6-39.
6.1 Testing automatic system generation using GeoRadiosity

Automatic system generation that splits scenes into systems can be tested using GeoRadiosity.

Follow these steps to test automatic system generation using GeoRadiosity. The different options in these steps are not required in an integration.

1. Ensure that all instances in your scene are placed in one system.
2. Ensure that each instance in the .scene file has a GUID. Using GUIDs lets you know which instances have been placed into which system.
3. Load your .scene file using the Load scene button in the Scene tab.
4. Under Scene Optimisation Method in the Scene tab, click on the Voxelisation radio button.
5. Set the voxel size slider anywhere between 0 and 512 pixel size units. A value of 0 pixel size units forces the precompute to read this value from the paramset.
6. Set the value of the Max Dependent sys distance slider anywhere between 0 and 1024 pixel size units. A value of 0 makes all systems independent of each other.
7. Select Optimise Scene.

A new .scene file is produced, with _Opt appended to the original scene name. The new file is opened in GeoRadiosity. The original .scene file is not modified. To see how Enlighten divides up the scene into systems, select the Systems visualization.
6.2 Performing automatic system generation

Automatic system generation can be performed from your editor.

When performing automatic system generation, ensure that all instances in your scene are placed in one system and that each instance in the .scene file has a GUID.

To add the automatic system generation functionality to your editor, use the following command to perform the automatic system generation. You must provide the original .scene file and the name of the optimized .scene file as parameters, and the voxel sizes:

```
C:\EnlightenSDK\Bin\Win64_2012\GeoPrecompMonitor.exe /t:OptimiseScene /p:SceneOptimisationMode=Voxelisation /p:InputScene=C:\myscene\myscene.scene /p:OutputScene=C:\myscene\myscene_optimised.scene /p:SystemGenerationVoxelSize=40 /p:SystemDependenciesMaxDistance=80
```

Setting the voxel size to a value of 40 is an approximate value that is suitable for most applications.
6.3 Using multiple systems in the high-level runtime

Multiple systems can be used in the high-level runtime (HLRT) by parsing the optimized scene file and then rendering the scene.

The optimized scene file must be parsed in your editor to get the list of newly created systems and the assignment of instances to these systems. The optimized scene file uses the same scene file format as the original World.scene file. Using any XML parser, implementing this functionality is straightforward. In this guide the GeoEn2Support library is used. The GeoEn2Support library makes use of TinyXML.

--- Note ---

Geomerics does not guarantee that the GeoEn2Support libraries have a stable interface. Geomerics recommends that you implement your own parsing code in your final integration.

---

Reading the scene file

The following code uses the GeoEn2Support library to load a .scene file. Replace the filename with the one used with the scene optimisation:

```cpp
GeoEngine::SceneDesc scene;
scene.LoadFromFile("myscene_optimised.scene");
```

This scene allows you to access all new systems. The following code will print out all new systems, including the instances assigned to them:

```cpp
for (Geo::s32 systemIdx = 0; systemIdx != scene.GetNumSystems(); ++systemIdx)
{
    const GeoEngine::SystemDesc* system = scene.GetSystemAtIdx(systemIdx);
    // The system name is required for loading the precompute data
    std::cout << system->GetSystemName() << "\n";
    for (Geo::GeoArray<GeoEngine::InstanceDesc>::const_iterator instance = system.Instances().Begin();
         instance != system.Instances().End(); ++instance)
    {
        // The new scene contains the same instances as the original scene with the same GUIDs
        std::cout << " " << instance->m_InstanceGuid;
    }
    std::cout << "\n";
}
```

Rendering the optimized scene

In your code, instead of printing out the assignment, store that data for rendering. All of the new systems must be loaded and added to the update manager. The update manager will produce one lightmap for each system and batching the instances by the automatically assigned system will improve performance.

Verify that this works by using a scene that consists of at least two systems after the automatic system generation. Verify both the lighting and the UV, using the debug code provided earlier, by comparing it with the output in Forge.
Chapter 7
Lighting probe-lit objects

How to light object with Enlighten probes.

Light probes provide radiosity for objects that are small, have complex geometry, or move at runtime. Place light probes in a scene and configure both the Enlighten high-level build system (HLBS) and the runtime for probes to contribute radiosity.

An object is suitable for probe lighting if it is small, made of complex geometry, or if the object is not static. An object is not static when it can move at runtime.

Small geometries only generate a small contribution to global illumination. This is because the object does not bounce much light back into the scene and does not occlude much indirect light. Geometries that are complex, such as trees, result in poor lightmap UV unwrapping. As such, probe lighting is more efficient for these geometries.

Examples of suitable candidates for probe lighting include small pieces of furniture, flower vases, objects that the player can move, and the player character.

Because spherical harmonic probes light probe-lit objects, it is not necessary to generate lightmap UVs for them.

Probe-lit objects do not contribute to global illumination computation. Therefore, they do not need to be passed to the Enlighten precompute.

Because probe-lit objects are not required to pass through the precompute, they can be added to a scene in your game editor and be lit correctly without an extra precompute.

To light probe-lit objects, probes must be placed in the scene and be passed through the precompute. Probes must be placed near probe-lit objects. Probes can be placed in the scene either as a 3D grid of probes or as an octree.
It contains the following sections:

- 7.1 *Adding a probe-lit object to the high-level build system* on page 7-42.
- 7.2 *Adding ProbeSet data to the high-level runtime* on page 7-43.
- 7.3 *Getting probe interpolants at runtime* on page 7-44.
- 7.4 *Adding probe-lit objects to the high-level runtime* on page 7-45.
- 7.5 *Shading probe-lit objects* on page 7-46.
7.1 Adding a probe-lit object to the high-level build system

Probe-lit objects are added to the high-level build system (HLBS) by declaring a probe-lit object in the HLBS scene description, adding a probe volume to the .scene file, and setting probe parameters.

Probe-lit objects do not need to be included in the Enlighten precompute. However, to visualize them in GeoRadiosity and Forge, probe-lit objects must be added to the high-level build system (HLBS) scene description.

To add a probe-lit object to the HLBS scene description:

1. Declare a probe-lit object in the HLBS scene description. To instruct the HLBS to ignore probe-lit objects, the value of their type is set to Fully Dynamic. Use the following example code to declare a probe-lit instance in the .scene file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<scene name="World" version="1">
<!-- other instances, including Radiosity ones, go here -->
<!-- declare Instance001 as probe-lit (fully dynamic) -->
<instance name="Instance001" geometry="Instance001" type="Fully Dynamic" transform="1.0 0.0 0.0 0.5 0.0 1.0 0.0 10.0 0.0 0.0 1.0 50.0 0.0 0.0 0.0 1.0"/>
</scene>
```

2. Add a probe volume ProbeSets to the .scene file. To do this, use the following XML code sample:

```xml
<volume name="DefaultProbeSet" guid="4237fc4ff21b94136b71b265d10cbaa5" probesParamSet="ProbesL1">
  <resolution x="3" y="13" z="3"/>
  <size x="200.0" y="200.0" z="200.0"/>
  <position x="0.0" y="0.0" z="0.0"/>
  <basis>
    <U x="1" y="0" z="0"/>
    <V x="0" y="1" z="0"/>
    <N x="0" y="0" z="1"/>
  </basis>
</volume>
```

The XML code declares a probe volume of dimensions (200, 200, 200), the corner of which is at position (0,0,0) in world space. The probe grid has a resolution of 3 on the X axis, 13 on the Y axis and 3 on the Z axis. The basis tag defines the orientation of the volume. This is X, Y, Z-aligned. The volume has the name DefaultProbeSet, a GUID, and a probesParamSet associated with it.

The probesParamSet that is named ProbesL1 must be declared in the Default.paramset file along with paramSets for all Enlighten systems and cube maps.

3. Specify that all probe volumes that use the ProbesL1 paramSet use L1 spherical harmonics and that probe culling is turned on. To do this, use the following code:

```xml
<probesParameterSet name="ProbesL1" id="1" useCulling="true" shNumCoefficients="4"/>
```

Probe culling removes probes that see mostly back faces. The result of probe culling does not affect nearby probe-lit objects.

After the precompute has run, ProbeSet data must be added to the high-level runtime (HLRT) to light objects in the scene.
7.2 Adding ProbeSet data to the high-level runtime

ProbeSet data is added to the high-level runtime (HLRT) by loading the precomputed data and adding the ProbeSet to the update manager.

A probe set is added to the HLRT using the update manager. Before a ProbeSet data can be added to the HLRT, load its precomputed data, the RadProbeSetCore. The RadProbeSetCore is generated during the precompute.

To add a probe set to the HLRT:

1. Load the IPrecomputeLoaderProbeSet in the HLRT using the following code:

```c++
// Load the IPrecomputeLoader
ProbeSet::Geo::GeoAutoReleasePtr<Enlighten::IPrecomputeLoaderProbeSet> probeSetData =
    Geo::LoadInterfaceCompressed<Enlighten::IPrecomputeLoaderProbeSet>(exportFolder +
    "__Build_" + sceneName + "__/precomputeloader/ProbesL1.probeset.sse").c_str());
Enlighten::RadProbeSetCore* probeSetRadCore = probeSetData->GetRadiosityCore();
Geo::GeoArray<Geo::v128> positions;
positions.SetCapacity(pInputProbeSet->GetNumProbes());

for(const v128* iter = pInputProbeSet->GetProbePositions(); iter != pInputProbeSet->
        GetProbePositions() + pInputProbeSet->GetNumProbes(); ++iter)
{
    positions.Push(*iter);
}
```

2. Add a ProbeSet to the UpdateManager. To do this, you must:
   a. Allocate an Enlighten::BaseProbeSet, passing the previously loaded RadProbeSetCore.
   b. Add the BaseProbeSet and an associated emissive environment to the UpdateManager.

Use the following code to add a ProbeSet to the update manager:

```c++
// Allocate a BaseProbeSet object (this will be owned by the UpdateManager)
Enlighten::BaseProbeSet* probeSet = updateManager->AllocateProbeSet(probeSetRadCore, false,
    NULL, Enlighten::SH_ORDER_L1);

// Add the BaseProbeSet to the UpdateManager
updateManager->EnqueueAddProbeSet(probeSet);

// Now set the emissive environment (without this probe lighting will not include
// contribution from the "sky")
Enlighten::EnqueueSetObjectParameter<Enlighten::BaseProbeSet, Geo::GeoGuid>(updateManager,
    probeSetRadCore->m_MetaData.m_ProbeSetId, &Enlighten::BaseProbeSet::SetEmissiveEnvironment,
    environmentGuid);
```
7.3 **Getting probe interpolants at runtime**

When a ProbeSet has been added to the UpdateManager, the high-level runtime (HLRT) ensures that the coefficients for each probe position are updated to represent the indirect lighting in the scene.

When a ProbeSet has been added to the UpdateManager, the HLRT ensures that the coefficients for each probe position are updated to represent the indirect lighting in the scene.

To correctly render a probe-lit object, an appropriate subset of these probes must be selected for rendering. It is possible to use the probe that is closest to the probe-lit object, but this results in the indirect lighting popping as the object moves. Instead, use an interpolation of the eight nearest probes. This corresponds to one cell of the regular grid of probes.

Probe interpolants depend on ProbeSets that are present in the scene and the position of the probe-lit object. Probe interpolants require recomputing if the object moves or if more ProbeSets are added.

Recalculation of the interpolants is a costly operation at runtime and Geomerics recommends avoiding this. Interpolants do not need to be recalculated if the lighting conditions change.

The HLRT calculates probe interpolants for all probe-lit objects that are added to the UpdateManager automatically. It also monitors the movement of probe-lit objects and only recalculates the interpolants when necessary.
7.4 Adding probe-lit objects to the high-level runtime

Probe-lit objects are added to the high-level runtime (HLRT) using the update manager. When the position of probe-lit objects change, the update manager must be notified.

Use the following code to add a probe-lit object to the update manager. This code uses a single-sample for the object, at the center of its bounding box. In the code, the object has been assigned a GUID and its bounding box has been calculated in local object space:

```cpp
Enlighten::DynamicObject dynamicObjectDesc;
// Set the GUID for this dynamic object
dynamicObjectDesc.m_Id = dynamicObjectGuid;
// Set its object space bounding box
dynamicObjectDesc.m_LocalBoundingBox = dynamicObjectBoundingBox;
// Use only one sample point per object
bool useVolumeTexture = false;
// Now allocate the dynamic object, which will be owned by the update manager
BaseDynamicObject* dynamicObject = updateManager->AllocateDynamicObject(dynamicObjectDesc, useVolumeTexture);
// Finally, add the newly created BaseDynamicObject to the update manager
updateManager->EnqueueAddDynamicObject(dynamicObject);
```

After the dynamic object has been added, you must notify the update manager when its position changes by setting its transform matrix.

Use the following code to notify the update manager of the position change:

```cpp
Enlighten::EnqueueSetObjectParameter<Enlighten::BaseDynamicObject, Geo::Matrix>( updateManager, dynamicObjectDesc.m_Id, &Enlighten::BaseDynamicObject::SetTransform, newTransform );
```
7.5 Shading probe-lit objects

Probe-lit objects are shaded by adding rendering and shading code.

To apply probe lighting to a dynamic object:

1. Retrieve the current spherical harmonic coefficients from the update manager and pass the values to
   the shader using the following code:

   ```cpp
   // Extract the SH coefficients for red, green and blue channels
   Geo::SHCoeff redCoeffs, greenCoeffs, blueCoeffs;
   updateManager->GetDynamicObjectShCoeff(dynamicObjectGuid, redCoeffs, greenCoeffs, blueCoeffs);

   v128 r, g, b;
   r = VConstruct(redCoeffs.GetL(0), redCoeffs.GetL(1), redCoeffs.GetL(2),
                  redCoeffs.GetL(3));
   g = VConstruct(greenCoeffs.GetL(0), greenCoeffs.GetL(1), greenCoeffs.GetL(2),
                  greenCoeffs.GetL(3));
   b = VConstruct(blueCoeffs.GetL(0), blueCoeffs.GetL(1), blueCoeffs.GetL(2),
                  blueCoeffs.GetL(3));

   // Set the shader uniforms with the coefficient data (render engine specific code)
   effect.SetVector(m_L1CoeffRHandle, "g_L1CoeffR", &r);
   effect.SetVector(m_L1CoeffGHandle, "g_L1CoeffG", &g);
   effect.SetVector(m_L1CoeffBHandle, "g_L1CoeffB", &b);
   ```

2. Compute the indirect probe lighting for a sample with a given surface normal using the following
   shader code:

   ```cpp
   // First L0 and L1 SH light coefficients
   float4 g_L1CoeffR;
   float4 g_L1CoeffG;
   float4 g_L1CoeffB;
   //------------------------------------------------------------------------
   ---------
   float NonlinearL1ComputeChannel(float3 normal, float4 coeffs)
   {
       float L0 = coeffs.x;
       float3 L1 = coeffs.yzw;
       float modL1 = length(L1);
       if (modL1 == 0.0f)
           return L0;

       float q = 0.5f + 0.5f * dot(normal, normalize(L1));
       float r = modL1 / L0;
       float p = 1.0f + 2.0f * r;
       float a = (1.0f - r) / (1.0f + r);
       return L0 * lerp((1.0f + p) * pow(q, p), 1.0f, a);
   }
   //------------------------------------------------------------------------
   ---------
   float3 CalcIrradianceSH(float3 normal)
   {
       float3 output;
       output.r = NonlinearL1ComputeChannel(normal, g_L1CoeffR);
       output.g = NonlinearL1ComputeChannel(normal, g_L1CoeffG);
       output.b = NonlinearL1ComputeChannel(normal, g_L1CoeffB);
       return max(float3(0.0f, 0.0f, 0.0f), output);
   }
   ```
Chapter 8
Adding support for Probe Radiosity systems

How to add support for Probe Radiosity systems

An object with complex geometry that results in a poor lightmap UV unwrapping and is large or important enough to contribute to the light transport by bouncing and occluding light must be added to a Probe Radiosity system. Some examples include, trees, complicated furniture, and statues.

Probe Radiosity geometry is lit using spherical harmonics and requires Enlighten probes to be placed in the scene. Probes can be added to a scene either as a probe set or as a probe octree.

Enlighten does not allocate any light-map pixels to Probe Radiosity instances. However, clusters are generated for geometry of this type. Clusters are required to calculate the light transport between other objects of this type and regular light-mapped geometry in the scene, to ensure that light is correctly bounced and occluded.

It contains the following sections:

• 8.1 Adding Probe Radiosity to high-level build system markup on page 8-48.
• 8.2 Adding Probe Radiosity systems to the high-level runtime on page 8-49.
• 8.3 Setting albedo buffers for contribute probe systems on page 8-50.
• 8.4 Rendering contribute Probe Radiosity meshes on page 8-51.
8.1 Adding Probe Radiosity to high-level build system markup

To add probe radiosity to high-level build system (HLBS) markup, Probe Radiosity geometry must be added to an Enlighten system and associated with a paramset.

Adding Probe Radiosity and light-mapped Radiosity instances to the same Enlighten system is not supported.

You can add multiple Probe Radiosity instances to the same Enlighten system.

Use the following example code to add Probe Radiosity geometry to the HLBS. The code defines one light-mapped Radiosity system containing one instance, and two Probe Radiosity systems containing one with one instance, and one with two instances.

```xml
<?xml version="1.0" encoding="utf-8"?>
<scene name="World" version="1">
  <!-- System001 is a Radiosity system with one instance in it -->
  <instance name="Instance001" systemId="System001" paramSet="Medium" geometry="Instance001" type="Radiosity" position="0.000000 0.000000 0.000000" rotation="0.0 0.0 0.0 1.0" />

  <!-- System002 is a Probe Radiosity system with one instance in it -->
  <instance name="Instance002" systemId="System002" paramSet="Medium" geometry="Instance002" type="Probe Radiosity" position="0.000000 0.000000 0.000000" rotation="0.0 0.0 0.0 1.0" />

  <!-- System003 is a Probe Radiosity system with two instances in it -->
  <instance name="Instance003" systemId="System003" paramSet="High" geometry="Instance003" type="Probe Radiosity" position="0.000000 0.000000 50.0" rotation="0.0 0.0 0.0 1.0" />
  <instance name="Instance004" systemId="System003" paramSet="High" geometry="Instance004" type="Probe Radiosity" position="0.000000 0.000000 100.0" rotation="0.0 0.0 0.0 1.0" />
</scene>
```

Probe Radiosity systems are associated with a paramset. The example code uses the paramset called High to define different quality settings for that system.

paramset settings impact runtime and precompute performance, and the final lighting quality. The most important parameter for Probe Radiosity systems is the cluster size. A cluster size that is too small results in a long precompute and a cluster size that is too large results in inadequate lighting quality.

Geomerics recommend that the cluster size be approximately twice the light-map pixel size of nearby light-mapped geometry. For human-scale objects such as houses and furniture Geomerics recommend a cluster size of approximately 2 meters. For large landscape features Geomerics recommend a value of approximately 20 meters or more.

The scene must contain one or more ProbeSet instances. Ensure that a ProbeSet is added to the HLBS .scene file.

Related information

Chapter 7 Lighting probe-lit objects on page 7-40.
8.2 Adding Probe Radiosity systems to the high-level runtime

Adding Probe Radiosity systems to the high-level runtime (HLRT) is done using the UpdateManager.

Adding Probe Radiosity systems to the high-level runtime (HLRT) is done in a similar way as adding Radiosity systems to the HLRT using the UpdateManager. However, when adding Probe Radiosity systems it is not necessary to allocate or set system solution spaces.

After creating an UpdateManager object, Probe Radiosity systems can be added and their material properties set. If the prerequisite ProbeSets have also been added to the UpdateManager, then Enlighten solves the Probe Radiosity systems.

Related information

Chapter 4 Setting up the update manager on page 4-23.
8.3 Setting albedo buffers for contribute probe systems

The correct albedo values must be set in the cluster albedo buffer for contribute probe systems.

Geomeric recommends setting albedo information for light-mapped Radiosity systems by rendering material albedo for each mesh to a texture in Enlighten lightmap UV space. This enables the materials to affect the color of the bounce lighting that Enlighten computes. This approach is not applicable for Probe Radiosity systems because these systems do not have light-map UVs or an associated light-map texture. As such, the integration must provide Enlighten with the correct albedo values in the cluster albedo buffer.

Albedo for contribute probe systems can be provided using the ColoursPerPoint API.

Related information

Controlling albedo on page 5-31

API module - Albedo handling
8.4 Rendering contribute Probe Radiosity meshes

Indirect lighting for Probe Radiosity meshes must be applied in the same way that indirect lighting applies to dynamic probe-lit objects.

Because Probe Radiosity geometry is probe-lit, indirect lighting is applied in the shader in the same way as it is applied for dynamic probe-lit objects.

Related information

Lighting probe-lit objects on page 7-40
Chapter 9
Adding cubemap support for creating dynamic specular reflections

How to add Enlighten cubemaps for creating dynamic specular reflections.

Enlighten provides cubemaps that generating radiosity through specular reflections. To configure these, you must first define the cubemap locations and parameters in the precompute, and then load the cubemap data at runtime.

It contains the following sections:
• 9.1 Setting up the precompute for cubemap support on page 9-53.
• 9.2 Loading cubemap data at runtime on page 9-54.
9.1 Setting up the precompute for cubemap support

To configure the precompute for cubemap support, cubemap locations and properties must be specified in the .scene file.

The pipeline for cubemaps is similar to probe sets. To use cubemaps in the Enlighten runtime, you must specify the cubemap locations to the precompute. Cubemaps are added as part of the .scene file in the high-level build system (HLBS). A <cubeMap> XML element represents each cubemap. At a minimum, each <cubeMap> element must include a unique name, face width, world space position, and orientation. The face width is the width in texels of a single face of the cubemap texture that will be generated. The following code shows an example <cubeMap> element. In the example, boxBasis is the orientation part.

```
<cubeMap name="Cube01">
  <faceWidth width="64"/>
  <position x="-4679.0" y="2000.0" z="450.0" />
  <boxBasis>
    <U x="1.000000" y="0.000000" z="0.000000" />
    <V x="0.000000" y="1.000000" z="0.000000" />
    <N x="0.000000" y="0.000000" z="1.000000" />
  </boxBasis>
</cubeMap>
```

Like Enlighten systems and probe volumes, cubemaps can have sets of parameters associated with them, defined in a .paramset file and referenced by name in the <cubeMap> XML element using the optional cubeMapParamSet attribute. The parameter defaults are set conservatively to ensure good quality cubemaps in a wide range of scenes.
9.2 Loading cubemap data at runtime

Cubemap data is loaded at runtime into an IPrecomputeLoaderCubemap object. A cubemap using this data must be loaded and added to the update manager.

The Enlighten precompute always creates the data required to generate both cluster-based and input lighting-based cubemaps. The HLBS allows you to specify the type of cubemaps to be assembled by the precompute loader API. At runtime, load the data for the type of cubemaps you require into a IPrecomputeLoaderCubeMap object:

```cpp
// Load the IPrecomputeLoaderCubeMap.
Geo::GeoAutoReleasePtr<Enlighten::IPrecomputeLoaderCubeMap> cubeMapData = Geo::LoadInterfaceCompressed<Enlighten::IPrecomputeLoaderCubeMap>((exportFolder + "__Build_" + sceneName + "__/precomputeloader/" + cubeMapName + ".cubemap.sse").c_str());
Enlighten::RadCubeMapCore* radCubeMapCore = cubeMapData->GetRadiosityCore();
```

You must allocate a cubemap using the loaded data, and then add it to the update manager. The following code shows an example of this allocation:

```cpp
BaseCubeMap* baseCubeMap = m_UpdateManager->AllocateCubeMap(radCubeMapCore);
m_UpdateManager->EnqueueAddCubeMap(baseCubeMap);
```

Ensure that you have implemented the CreateCubeMap() method in your IGpuTextureAllocator implementation and handle cubemaps in your IGpuTextureUpdater.

Various 32bpp HDR-encoded cubemap output texture formats are supported. However, Enlighten::OUTPUT_FORMAT_FP16 is the default output texture format and enables the solver to include an opacity mask in the alpha channel.

A cubemap pixel that does not fully cover geometry is given an alpha value of less than 1. This information can then be used to composite Enlighten local cubemaps with a high-resolution infinite cube map, such as a skybox, in your specular IBL shader.
Chapter 10
Summary

A summary of the steps outlined in this guide.

It contains the following section:
• 10.1 Summary of tasks undertaken on page 10-56.
10.1 Summary of tasks undertaken

By following the instructions in this guide, you can integrate Enlighten into a simple scene.

To summarize, the tasks in this guide include:

• Exporting your scene in a format that Enlighten can handle.
• Importing and precomputing your scene in GeoRadiosity.
• Adding a debug rendering to your engine to verify the Enlighten UVs.
• Successfully rendering a scene with Enlighten environment lighting.
• Passing materials and lights to Enlighten.
• Extending the basic integration to support larger scenes using multiple systems.
• Rendering probe-lit geometry.
• Rendering contribute probe-lit geometry.
• Using Enlighten cubemaps for specular materials

Use this guide as a basis for future Enlighten integrations and as a reference for links to advanced Enlighten topics.
Appendix A
Revisions

This appendix describes the technical changes between released issues of this guide.

It contains the following section:
• *A.1 Revisions* on page Appx-A-58.
A.1 Revisions

Each table lists the technical differences between successive issues of this tutorial.

Table A-1 Issue 0307-00

<table>
<thead>
<tr>
<th>Change</th>
<th>Location</th>
<th>Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td>First release</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table A-2 Issue 0308-00

<table>
<thead>
<tr>
<th>Change</th>
<th>Location</th>
<th>Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td>No changes</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>