

Indigo Renderer SDK Documentation

Copyright Glare Technologies Limited 2019-



Render: Poustinia 2 by bubs

Contents

1	<u>Indigo Renderer SDK Documentation</u>
2	<u>Contents</u>
3	<u>Introduction</u>
4	<u>Included Materials</u>
4	<u>Binaries and support files</u>
4	<u>Documentation</u>
5	<u>C++ Header Files</u>
5	<u>Example Program</u>
6	<u>Indigo SDK Introduction</u>
6	<u>Initialisation</u>
6	<u>Using the message queue</u>
7	<u>Constructing the scene graph</u>
12	<u>Contact</u>
13	<u>Changes</u>

Introduction

The Indigo Renderer SDK allows developers to harness the full power of Indigo, in order to create highly realistic renders from a host application.

The Indigo Renderer SDK provides a C++ API, which allows the Indigo functionality to be used in any C++ program.

The Indigo SDK is currently available for Windows and Mac OS X.

Included Materials

The Indigo SDK consists of the following components:

Binaries and support files

The *binaries* directory contains the 64-bit indigo DLL or SO file, other DLLs needed to run the indigo DLL, and other data and configuration files needed by Indigo.

In particular, you will need to distribute the contents of one of the subdirectories of *binaries* with your program.

The *vs2010_64* directory contains the 64-bit binaries for Visual Studio 2010.

The *vs2012_64* directory contains the 64-bit binaries for Visual Studio 2012.

The *vs2015_64* directory contains the 64-bit binaries for Visual Studio 2015.

The *vs2017_64* directory contains the 64-bit binaries for Visual Studio 2017.

Documentation

Indigo SDK.pdf

This file.

changelog.txt

This is the changelog for Indigo itself. This file is also included in the normal Indigo distribution, but is relevant for the SDK as well.

Indigo Technical Reference.pdf

This file contains in-depth explanations of some Indigo scene file XML elements, plus shader language description, command line parameters and other information. The technical reference is now moving online, to <https://www.indigorenderer.com/indigo-technical-reference>, which has newer information but is not yet complete.

API Reference (doxygen_html)

This directory contains in-depth API documentation of the C++ classes that make up the Indigo SDK.

Please view the index.html file in the doxygen_html directory to get started.

C++ Header Files

The `include` directory contains all the C++ header files needed in order to use the Indigo API.

We try to make our header files nice and readable, so please do look at our header files to understand the Indigo API!

Example Program

The `indigo_dll_example_vs201x` directories contains example program source code (`Driver.cpp`), and a Visual Studio 201x project, showing how to initialise the Indigo API, create a scene graph, start a render, and save the resulting image to disk.

Building and launching the example program

Open the solution file in the `indigo_dll_example_vs201x` directory, and press Control+F5 to build and run the example.

The example will now launch and should render successfully. The output image is saved as `render.bmp`.

Indigo View example code

This is located in `documentation\Indigo_View_example_code`, and consists of four files:

`IndigoView.cpp`, `IndigoView.h`, `IndigoConversion.cpp`, `IndigoConversion.h`.

This code is an example of fully-functional code that embeds a realtime-updating Indigo Renderer widget in a 3d visualisation Qt app, that updates the rendered view in realtime as materials are changes, the camera is moved around, etc..

`IndigoConversion.cpp/.h` shows one way to convert different material and object formats into the format Indigo requires.

Indigo SDK Introduction

Initialisation

The Indigo SDK is distributed as a library, which is started at run-time by creating an Indigo context. This context is then initialised with the Indigo data path (to access licence files etc.), after which a renderer can be created from the context.

Objects created by the Indigo SDK are typically accessed through references that do reference counting, so the creator doesn't need to worry about freeing the memory when it's no longer used. Many functions return an `indResult` to indicate success or failure via error codes.

Please see `Driver.cpp` in `indigo_dll_example_vs2012` for an example of how to initialise an Indigo context, and then to create a renderer.

The renderer object encapsulates the rendering of a scene, and returns the final HDR and LDR images via the `RenderBuffer` and `UInt8Buffer`, and `FloatBuffer` buffers specified when calling its `initialiseWithScene()` method. Note that this **must** be called before rendering can begin.

The call is non-blocking and so returns almost immediately, however the initialisation can take a while to process large scenes. The message queue can be checked for a `ProgressMessageInterface` message, as described in the next section.

Using the message queue

Renderers communicate with the host application in a thread-safe way via a message queue, which should be periodically checked using the `getMessages` method provided by the renderer.

A specific kind of reference is used for messages in the queue, called a handle; these are needed to ensure the SDK works properly across multiple compilers and platforms, but are not functionally different from references.

The following code illustrates basic message handling:

```
Vector<Handle<MessageInterface> > messages;
renderer->getMessages(messages);

for(size_t m = 0; m < messages.size(); ++m)
{
    Handle<MessageInterface> message = messages[m];
    MessageInterface::Type msg_type = message->getType();

    // Switch on the method type
    switch(msg_type)
    {
        case MessageInterface::ERROR_MESSAGE:
            // Report error...
            exit(1);
            break;
        case MessageInterface::LOG_MESSAGE:
            // Report message...
            break;
        case MessageInterface::PROGRESS_MESSAGE:
            // Report build progress...
            break;
        // Handle other message types...
    }
}
renderer->releaseMessageQueueLock();
```

The messages are placed into the `messages` Vector by the renderer, after which each message is processed by an appropriate case in the `switch` statement by inspecting its type.

For a comprehensive list of message types, please consult the API Reference for `MessageInterface`.

Constructing the scene graph

Introduction

The scene graph (SG) consists of scene nodes arranged in a hierarchy. SI units (metres and seconds) are used for data where applicable.

The root node of a SG always has a special type, `SceneNodeRoot`, which provides additional methods to access the scene media and materials, physical sun and sky parameters, render settings etc.

```
SceneNodeRootRef root = new SceneNodeRoot();
```

After constructing a root node, several required child nodes are created: the renderer settings, camera settings, and tone mapping settings. In each case, after construction and initialisation, it must be made a child of the root node.

The following code illustrates the construction of the renderer settings node:

```
Indigo::SceneNodeRenderSettingsRef settings =
Indigo::SceneNodeRenderSettings::getDefaults();

settings->metropolis.setValue(false);
settings->bidirectional.setValue(false);
settings->image_save_period.setValue(10000000.0);
settings->width.setValue(render_width);
settings->height.setValue(render_height);
settings->info_overlay.setValue(true);

root->addChildNode(settings);
```

After the scene graph has been constructed, a call to the root node's `finalise()` method **must** be made with the path to the scene file (if required, otherwise a dummy string can be passed in).

Camera nodes

A camera node must also be created for the scene to be valid, and currently only a single camera node is supported. Further information on the elements can be found in the API Reference.

```
SceneNodeCameraRef cam = new SceneNodeCamera();
cam->lens_radius = 0.0001;
cam->autofocus = false;
cam->exposure_duration = 1.0 / 30.0;
cam->focus_distance = 1.0;
cam->lens_sensor_dist = 0.03;
cam->lens_shift_right_distance = 0;
cam->lens_shift_up_distance = 0;
cam->sensor_width = 0.035;

cam->forwards = Vec3d(0, 1, 0);
cam->up = Vec3d(0, 0, 1);
cam->setPos(Vec3d(0, -2, 0.3));
```

```

root->addChildNode(cam);

SceneNodeTonemappingRef tone_mapping(new
SceneNodeTonemapping());
tone_mapping->setType(SceneNodeTonemapping::Reinhard);
tone_mapping->pre_scale = 1;
tone_mapping->post_scale = 1;
tone_mapping->burn = 6;

root->addChildNode(tone_mapping);

```

Mesh nodes

Indigo meshes are comprised of triangles and quads. The use of quads is recommended when the source data is naturally quad-based, since the subdivision surface algorithms perform better using a combination of both triangles and quads, compared to triangulated data.

A simple mesh node with a single quad can be created as follows:

```

SceneNodeMeshRef mesh_node(new SceneNodeMesh());
mesh_node->mesh = new Mesh();
mesh_node->mesh->num_uv_mappings = 0;

// Make a single quad
Indigo::Quad q;

// Set the material index to the first material in the scene
q.mat_index = 0;

// Set the vertex and UV indices
for(int i = 0; i < 4; ++i)
{
    q.vertex_indices[i] = i;
    q.uv_indices[i] = 0;
}

// Add it to mesh's quads
mesh_node->mesh->quads.push_back(q);

// Define the vertices
mesh_node->mesh->vert_positions.push_back(Vec3f(0, 0, 2));
mesh_node->mesh->vert_positions.push_back(Vec3f(0, 1, 2));
mesh_node->mesh->vert_positions.push_back(Vec3f(1, 1, 2));
mesh_node->mesh->vert_positions.push_back(Vec3f(1, 0, 2));

mesh_node->mesh->endOfModel(); // Build the mesh

root->addChildNode(mesh_node);

```

Model nodes

A model node associates a mesh node with one or more material node and a (key framed) transformation, allowing an instance of the mesh with a given transformation path and appearance to be created:

```
SceneNodeModelRef model(new SceneNodeModel());

model->setName("Instanced object");
model->setGeometry(mesh_node);
model->keyframes.push_back(KeyFrame());
model->rotation = new MatrixRotation();
model->setMaterials(Vector<SceneNodeMaterialRef>(1, mat));

root->addChildNode(model);
```

Material nodes

Material nodes define an Indigo material as a combination of various basic components. The `albedo` parameter defines the diffuse reflectance of the material; in our example a `TextureWavelengthDependentParam` spectrum is used, which also requires the definition of texture co-ordinates. The basic `UVTexCoordGenerator` generates UVs using the UVs specified in the mesh.

For more information on the material and spectrum types, please consult the [API Reference for the Material, TextureWavelengthDependentParam and WavelengthIndependentParam classes](#).

```
SceneNodeMaterialRef mat(new SceneNodeMaterial());

DiffuseMaterial* diffuse = new DiffuseMaterial();
diffuse->random_triangle_colours = false;
diffuse->layer = 0;

// Create a texture.
Indigo::Texture texture;
texture.path = "ColorChecker_sRGB_from_Ref.jpg";
texture.tex_coord_generation = new UVTexCoordGenerator();

diffuse->albedo = new TextureWavelengthDependentParam(
    texture
);
```

Light sources

A valid Indigo scene must have at least one light source, defined to be a model with a material with non-zero emission. The emission is controlled through the `base_emission` and `emission` components, which respectively define the source light spectrum and a multiplied spectrum.

```
SceneNodeMaterialRef mat(new SceneNodeMaterial());

DiffuseMaterial* diffuse = new DiffuseMaterial();
diffuse->random_triangle_colours = false;
diffuse->layer = 0;
diffuse->base_emission = new ConstantWavelengthDependentParam(
    new UniformSpectrum(1.0e10));
diffuse->emission = new ConstantWavelengthDependentParam(
    new UniformSpectrum(1.0));
mat->material = diffuse;

root->addChildNode(mat);
```

Reading the rendered image back

Once Indigo has done some rendering, you can retrieve the rendered image back to your code, in order to display it in the host application.

Accessing images is generally done once they have been tone-mapped.

Indigo will tonemap the image in the render buffer, and place it in the `UInt8Buffer` object that you created and passed to the `ToneMapper` constructor.

Indigo will tonemap an image (and copy it to the `UInt8Buffer` as part of the tonemapping operation) in various circumstances:

- 1) You can explicitly request a tonemap with `ToneMapper::startToneMapping()`. This is a non-blocking call, so tonemapping will take place in another thread. You can query whether the tonemap is done with `ToneMapper::isToneMappingDone()`.
- 2) You can call `ToneMapper::toneMapBlocking()`. This call blocks until the tone mapping operation is complete. You may want to call this when the user presses a 'refresh image' button, for example.
- 3) Indigo will trigger tonemaps itself at various stages of the rendering process, for example after render passes are done. You can query `ToneMapper::isImageFresh()` to see if this has occurred. `isImageFresh` will return true if a tonemap has been completed, and `isImageFresh()` has not already been called since the tonemap was completed. **This is the recommended way of receiving images from the Indigo SDK.** Using this technique takes advantage of the Indigo core tonemapping more frequently during the early stages of the render, and then less frequently as the render converges. See `IndigoView.cpp` for an example of this technique.

If you want to get floating point tonemapped data instead of 8-bit data, you can construct a `FloatBuffer` object and pass it to the `ToneMapper` constructor.

You can also use `Indigo::ImageSaver` (in `IndigoImageSaver.h`) to save tone-mapped (and non-tonemapped) images directly to disk.

Enabling OpenCL (GPU) rendering

The first step is to query available OpenCL devices:

```
// Query GPU devices
Indigo::HardwareInfoRef hardware_info = new Indigo::HardwareInfo(this->context);

const Indigo::Vector<Indigo::OpenCLDevice> devices =
hardware_info->queryOpenCLDevices()->getOpenCLDevices();
```

Once the devices have been queried, you can choose which ones to use. Be aware that some devices may be CPU devices, which are generally not very efficient to use. For example, to select all GPU devices:

```
// Work out which devices to use. We will use all GPU devices.
Indigo::Vector<Indigo::OpenCLDevice> devices_to_use;
for(size_t i=0; i<devices.size(); ++i)
{
    if(!devices[i].is_cpu_device) // If this is a GPU device:
    {
        conPrint("Using device " +
toStdString(devices[i].device_name));
        devices_to_use.push_back(devices[i]);
    }
}
```

Finally, we can tell Indigo to use the devices we selected, by setting `enabled_opengl_devices` in the Render Settings scene node:

```
if(!devices_to_use.empty()) // If we found at least once GPU device:
{
    settings_node->gpu.setValue(true); // Enable GPU rendering
    settings_node->enabled_opengl_devices = devices_to_use; // Tell
indigo which devices to use.
}
```

See `IndigoView.cpp` for this example code.

(documentation/Indigo_View_example_code/IndigoView.cpp)

Contact

If you need technical support with the Indigo Renderer SDK, please contact

support@indigorenderer.com

SDK Changelog

The following changes are in addition to the changes to the Indigo render-core itself, which can be found in changelog.txt.

4.4.1

- * Added Indigo View example code to SDK distribution.
- * Made it so that passes are done (e.g. PassDoneMessages are emitted) in CPU rendering mode even after realtime mode is finished. Allows SDK users to just use `ToneMapper::isImageFresh()`.
- * Added the 'Reading the rendered image back' documentation section to Indigo SDK.pdf.
- * Simplified some code in indigo example Driver.cpp.
- * **BREAKING CHANGE:** API users now set the OpenCL device to use by setting it in `SceneNodeRenderSettings` instead of modifying `settings.xml`. However the device selection is not serialised with `SceneNodeRenderSettings`.
- * Added 'explicit' to `Indigo::Vector` count constructor to prevent accidental type conversion.
- * Cleaned up `IndigoMaterial.h` a bit and added some convenience constructors to various node classes.
- * Added initial implementation of in-mem tex map in the API.
- * `Indigo::Texture` now has a default `UVTexCoordGenerator` `tex_coord_generator`.
- * Throwing exception from scene loader if texture `tex_coord_generation` is null, instead of crashing.
- * Added `SceneNodeCamera::setPosAndForwards()`.
- * Added some new `Matrix2` constructors for easier use.
- * Made it so that `IndigoMesh::endOfModel()` is called automatically in `SceneNodeRoot::finalise()` if it hasn't been already. This is to make using the API a bit easier.
- * `SceneNodeRoot::addChildNode` now adds node dependencies (e.g. other nodes that this node depends on, such as children of a blend material) to the scene graph as well, if they are not already added. Made it so that `addChildNode` has no effect if the node is already a child.
- * added `setSoloLayer` to `Indigo::ToneMapper`
- * Added `Indigo::Renderer::getRegionSamplesPerPixel()`
- * Removed some old, unused header files.

4.0.50

- * All `updateSequenceNumber()` calls can be removed (automatically done in `setDirtyFlags()` now)
- * `ToneMappingDoneMessage` message has been removed. Instead use `Indigo::ToneMapper::isImageFresh()`.
(`Indigo::ToneMapper::isToneMappingDone()` also works but `isImageFresh()` is preferred.)
- * You can now set render settings more directly instead of looking up by string, e.g.

```
settings->bidirectional.setValue(false);
```


instead of

```
settings->setBoolValue("bidirectional", false);
```
- * Texture are now passed directly into `TextureWavelengthDependentParam`, instead of referenced by index.
- * `renderer->initialiseWithScene()` now takes a tone mapper reference. This is so tonemapping can be done automatically by the

renderer after realtime changes.