Leia Creative Toolkit

Android SDK Guide

Leia Loft™

v 0.0.5

July 3, 2018

Table of Contents

Leia Loft Native SDK Contents **4**

Native Libraries and Headers **4**

- Samples 4
  - TeapotsWithLeia 4
  - JNISample 4
- Source 4
- Prebuilt Applications 4

Getting Started **5**

- Prerequisites 5
- Project Setup Steps 5
- Design Considerations 8

Working with the Code **9**

- Objects 9
- LeiaCameraData 9
- LeiaCameraView 9

Rendering For Leia **10**

- LeiaInitializeCameraData 10
- LeiaCalculateViews 11
- Updating Your Matrices 12
- Shaders 13
  - View Interlacing 13
  - Depth Of Field (DOF) 13
  - View Sharpening 14
- Accessing the Shaders 15
- Leia Render Functions 15
- Extensions For The LeiaNativeSDK Rendering Functions 17
- Querying the Leia System Values 20
  - Including the Leia module in Android Studio 20
  - Accessing Leia System Functionality Through Java 20
    - Interacting with the 3D Backlight 21

Querying Leia System Parameters 22
Getting System Parameters Into Native Code 23

Updating The GL Pipeline 24
Render Target Size 24
Managing Depth 26
Updating the Shader Uniforms 27
Texture Parameters 27

Overview of Lightfield Rendering 28

Content Design 30
Scene Design 30
Controlling Depth 30

Tips and Tricks 31
Shader Compilation 31
Setup Cameras Once 31
UI, Menus and Text 31
Minimize GL draws with culling objects by area 31

Common Issues 33

Appendix A: Example build.grade 34

Appendix B: Example CMakeLists.txt 36

Leia Loft Native SDK Contents

Native Libraries and Headers

The LeiaNativeSDK directory contains .so libraries and the headers needed in order to build for an NDK application. Your applications will need to link to these libraries based on their platform, as well as add the header(s) to the projects in order to use the classes provided.

Samples

TeapotsWithLeia

Inside the samples directory the basic sample code used to highlight how to use the libraries from the LeiaNativeSDK directory. This is a sample downloaded from the Google NDK samples called “Teapots”, and Leia extended this application since it is something every developer can get familiar with and see how we have modified it.

JNISample

Another sample is to highlight how to add Java and JNI code to your NativeActivity project if you currently do not include any. This is necessary in order to interact with the Leia system service, which provides hardware specific information needed by the library.

Source

The source folder has useful code which can be used to simplify different operations in a project. For example, the LeiaJNIDisplayParameters folder contains two files for simplifying implementing the JNI interface which carries information about the Leia hardware through Java and into the Native code.

Prebuilt Applications

The sample projects provided are fully functional, but to see them in action without building the projects separately, the application files are provided here.

Getting Started

Prerequisites

1. Android Studio
    a. NDK installed
2. Leia Native SDK

Project Setup Steps

This section explains the steps required to prepare your Android Studio project to use the LeiaNativeSDK.

To start, open your existing project or start a new NDK project in Android Studio.

Next connect to the Java module that gives access to the Leia Display and the hardware values associated with that.

In Android Studio, every project contains modules, and both the project and modules contain build.gradle files. The modules are where the local code exists, so any module that will be interacting with the Leia Display needs to be adjusted. You can add the Java module to your own application module:

Note: These changes can be found in Appendix A or in the TeapotsWithLeia sample at:
<LeiaNativeSDK dir>/samples/TeapotsWithLeia/classic-teapot/build.gradle

1. In your module's folder, open the build.gradle file
2. Add the following to the build.gradle file:

```
repositories {
    maven {
        url 'https://leiainc.jfrog.io/leiainc/gradle-release'
    }
}
```

3. Add a dependency to the Leia library:

```
dependencies {
    implementation 'com.leia:leiasdk:4+'
}
```

NOTE: On older projects, "implementation" will need to be changed to "compile"

- TEST: You can now include Leia specific imports if everything went well. If adding imports from the Leia module works, then you have successfully pulled it into your application. Rebuilding your project should work without error. Errors like "error: package com.leia.android.lights does not exist" show the above did not work.
  Try adding the following code to your application module and if the rebuild works you can move on to the next step:

```
import com.leia.android.lights.LeiaSDK;
```

Now that the build.gradle knows how to find the Android module, you want to update your application module to build for the correct target. In this case, the build targets supported by Leia currently are "arm64-v8a" and "armeabi-v7a". The way we can add this to the module is by doing the following:

Note: These changes can be found in Appendix A or in the TeapotsWithLeia sample at:

<LeiaNativeSDK dir>/samples/TeapotsWithLeia/classic-teapot/build.gradle

1. Open the build.gradle file which you opened for the previous step to include the Leia Android module.
2. Add the following to the build.gradle file:

```
android {
        ...
        defaultConfig {
                ...
                ndk {
                        abiFilters 'arm64-v8a', 'armeabi-v7a'
                }
                ...
        }
        ...
}
```

Once the Java module added, it is necessary to link to the native portion. Android Studio supports CMakeLists.txt for building native applications, this is the approach used to add the native LeiaNativeSDK library. First add the header files and libraries into the project by creating a common directory. Then update the CMakeLists.txt file in the source code directory. The steps below show how to do this:

1. In the application root directory, create a folder for common libraries

   Note: These changes can be found in:

   <LeiaNativeSDK dir>/samples/TeapotsWithLeia/

   A. referenced in future steps as <application root>/<distribution folder>

```
< application root >/distribution/
```

2. Inside the <distribution folder>, create a folder named "leia_sdk"

   Note: These changes can be found in:

   <LeiaNativeSDK dir>/samples/TeapotsWithLeia/distribution/

   A. referenced in future steps as <application root>/<distribution folder>/leia_sdk

```
< application root >/distribution/leia_sdk
```

3. Inside the leia folder just created, add the folders found in the LeiaNativeSDK folder.
   Note: These changes can be found in:
   <LeiaNativeSDK dir>/samples/TeapotsWithLeia/distribution/leia_sdk

```
<application root>/<distribution folder>/leia_sdk/include
<application root>/<distribution folder>/leia_sdk/lib
```

With the directory created, you have added the Leia native libraries to your project. Now you need to tell the project how to find them and include them in the application. This can be done accomplished by the following steps:

Note: These changes can be found in Appendix A or in the TeapotsWithLeia sample at:
<LeiaNativeSDK dir>/samples/TeapotsWithLeia/classic-teapot/build.gradle

1. Update the build.gradle for the application's module which will be using the Leia library by adding jniLibs.srcDirs with the distribution directory from the previous step. If the build.gradle already contains jniLibs.srcDirs, and the distribution directory from the previous step to the list instead.

```
android {
    sourceSets {
        main {
            jniLibs.srcDirs = ['./../distribution/leia_sdk/lib']
        }
    }
}
```

Your application now understands how to interact with the Leia Java module, and it knows how to find the Leia library, and include it in the package. The next step is to make the library visible to the native code, so you can make LeiaNativeSDK library calls. To do this, follow the instructions below:

1. Include the library in the modules CMakeLists.txt
   Note: These changes can be found in Appendix B or in the TeapotsWithLeia sample at:
   <LeiaNativeSDK dir>/samples/TeapotsWithLeia/classic-teapot/src/main/cpp/CMakeLists.txt

   A. Add a Cmake variable for the distribution directory (and change <distribution dir> to your directory) by calling the "set" function, near the top of the file.

```
set(distribution_DIR ${CMAKE_SOURCE_DIR}/../../../../<distribution dir>)
```

   B. Add a library variable to the project by adding a call to "add_library"

```
add_library(lib_leia_sdk STATIC IMPORTED)
```

   C. Set the library variable properties by calling "set_target_properties" directly below the "add library" call in (B)

```
set_target_properties(lib_leia_sdk PROPERTIES IMPORTED_LOCATION
            ${distribution_DIR}/leia_sdk/lib/${ANDROID_ABI}/libleiasdk.so)
```

   D. Add the include directory for the headers (and change ACTIVITY NAME to the name of activity running your module) by adding a call to "target_include_directories" function.

```
target_include_directories( <ACTIVITY NAME> PRIVATE
                              ${distribution_DIR}/leia_sdk/include)
```

E. Link to the library (and change ACTIVITY NAME to the name of activity running your module) by adding “lib_leia_sdk” to the “target_link_libraries” function

```
target_link_libraries(<ACTIVITY NAME>
                      lib_leia_sdk)
```

TEST: Add one of the header files included in <application root>/<distribution folder>/leia_sdk/include and from the Build menu select Rebuild Project to verify the application can now see the include directory

Design Considerations

1. Set the focal plane to where the viewer should be looking
    a. Text and UI look best when in focus
    b. Game objects between the near plane and the convergence plane will appear to pop out of the screen
    c. Game objects between the convergence plane and far plane will appear inside the device
    d. Set baseline as needed to determine the amount depth objects have

2. Use a DOF post effect to blur areas that are not near the convergence plane
    a. This allows you to make the 3D effect more pronounced without breaking the user's suspension of disbelief.

3. Switch the 3D backlight off when only showing 2D content.
    a. Improves the visual quality and saves power.

Working with the Code

Objects

Leia currently provides two main objects in the LeiaNativeSDK. These are the following:

- LeiaCameraData
- LeiaCameraView

Together, they allow applications to make full use of the Leia display with minimal changes to their current applications.

There are also 2 Java objects to note:

- LeiaDisplayManager
- SimpleDisplayQuery

While these are not strictly part of the LeiaNativeSDK, these are the classes that allow you to get information about the Leia hardware from the Android system, and this information is needed by the LeiaNativeSDK. Usage of these objects are discussed at the end of this section.

LeiaCameraData

The LeiaCameraData struct is a storage variable that the rest of the Native SDK can use to better understand the application. It stores values relevant to the perspective matrices which will be generated for your application later. By keeping this data together, it allows the LeiaCameraView struct to stay small, enabling it to scale better based on the necessary number of cameras and removing duplicate data.

LeiaCameraView

An important part of the LeiaNativeSDK is to provide perspective matrice, simplifying the work behind creating 3D effects. Each LeiaCameraView struct contains a perspective matrix. Your application will use a 2D array of these and the LeiaNativeSDK library will be able to properly fill the matrices inside for you to use in your render loop. This matrix will replace the standard camera projection matrix your application would normally use.

Rendering For Leia

We want to work with developers to help them bring their games to life with our lightfield technology. In order to do this, our first priority is to make sure we fit into existing systems rather than force changes on them.

LeiaInitializeCameraData

At the beginning initialization part of the scene, your application will need to fill a LeiaCameraData object, which the rest of the library uses. Any time the camera parameters get updated, this object needs to be updated appropriately. In order to initialize this data the first time, a LeiaCameraData object should be passed into leiaInitializeCameraData, using the below function:

```
void leiaInitializeCameraData(LeiaCameraData * data,
                              int num_horizontal_views,
                              int num_vertical_views,
                              float system_disparity_in_pixels,
                              float baseline_scaling,
                              float convergence_distance,
                              float vertical_field_of_view_degrees,
                              float near, float far,
                              int view_resolution_x_pixels,
                              int view_resolution_y_pixels);
```

data: get filled with a number of useful values used to compute the perspective matrices.
num_horizontal_views and *num_vertical_views:* give the library an idea of how many cameras will be used during rendering.
system_disparity_in_pixels: a value retrieved from the Leia System Service.
baseline_scaling: this is a scalar for how far beyond the system disparity you would like. For example, by setting *baseline_scaling* to 2.0, you will double the amount of disparity compared to if you set *baseline_scaling* to 1.0. The larger the value, the more objects will appear to pop out or into the screen, but larger values may cause you to lose the effect altogether.
convergence_distance: In world space, this is the location where an object looks exactly the same from all views. This value should fall between *near* and *far*. Objects between *near* and *convergence_distance* (but not AT *convergence_distance*) will appear to pop out of the screen, while objects between *convergence_distance* and *far* will appear to fall into the screen.
vertical_field_of_view_degrees: The number of degrees vertically of the field of view, which is a standard value used for creating a perspective matrix.
near and *far*: the near and far planes of the main camera frustum.
view_resolution_x_pixels and *view_resolution_y_pixels*: the size of each view the Leia Camera will render into. These values can be retrieved from Android using SimpleDisplayQuery, and are a value

smaller than the resolution of the screen based on orientation.

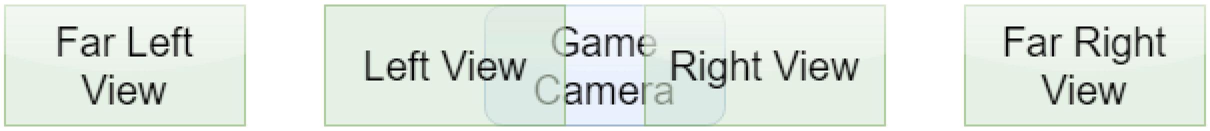
LeiaCalculateViews

This is the main function in the SDK, simplifying the math involved in creating the different views. To do this we give native access to leiaCalculateViews. By providing a LeiaCameraView object for each view to this function (along with some other data) the LeiaNativeSDK will populate each view with a perspective matrix to be used later in rendering. The function looks like the following:

```
int leiaCalculateViews(LeiaCameraData * data,
                       LeiaCameraView * out_views,
                       int len_views_in_x,
                       int len_views_in_y);
```

data: A LeiaCameraData object previously filled using leiaIntializeCameraData
out_views: A two dimensional array of LeiaCameraView objects. If the views were declared as *LeiaCameraView views[1][4]*, 1 is the number of views vertically, and 4 is the number of views horizontally.
len_views_in_x and *len_views_in_y*: The number of views in the x and y direction respectively. These values must match the numbers provided in leiaInitializeCameraData.

In the approved layout Leia supports, the normal set of views is 1x4, with 2 views to the left and 2 to the right of the dividing line.



All of the cameras will share the plane at the convergence distance, each camera will have a slightly less-wide projection plane than normal. It also means all objects which are exactly on the convergence plane will appear the same to each camera. Only objects not on the plane of convergence will appear in different locations, which is where the effect for depth comes from.

Now a perspective matrix exists; we have to use these new matrices. It is very common for games to have a view matrix for the camera, which handles positioning, while the projection for the camera is a different matrix. In this case, the matrices provided by the LeiaCameraView will be the projection matrices. The LeiaCameraView member *matrix* contains the projection matrix as a float array with 16 values. In order to handle the multiplication correctly, it is important to know how the values are laid out in memory, so below gives a mapping (though the description below is not how we compute the final matrix):
{(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (transX, transY, transZ, 1)}.

Multiplying the view matrix with the perspective matrix inside the LeiaCameraView object will provide the correct final matrix, which can be accessed like below:

```
Mat4 perspective = Mat4(view.matrix);
mat_vp = perspective * mat_v;
```

Updating Your Matrices

The LeiaCameraData and LeiaCameraView objects are both structs. This leaves the data inside them readily accessible for you to use in your application. However, these should be considered read-only. The data members of both structs are highly dependent on each other and the math involved in creating them. This means the objects cannot be directly updated by simply changing one of the internal values of the struct. For example, your application should not do:

```
myLeiaCameraData.mVerticalFieldOfView = 70.0f;
```

While this is entirely valid code which will compile, this does not ensure that the entire struct is completely updated, and that the view objects get updated as well. In order to update the LeiaCameraData struct properly, your application should use one of the “setter” functions provided:

```
void leiaSetNumberOfViews(LeiaCameraData* data,
                           int num_horizontal_views,
                           int num_vertical_views);
void leiaSetSystemDisparityInPixels(LeiaCameraData* data,
                                     float disparity_in_pixels);
void leiaSetApplicationBaselineScale(LeiaCameraData* data,
                                      float baseline_scaling);
void leiaSetFieldOfView(LeiaCameraData* data,
                         float fov_in_degrees);
void leiaSetFrustumPlanes(LeiaCameraData* data,
                           float near,
                           float focal_distance,
                           float far);
void leiaSetViewSizeInPixels(LeiaCameraData* data,
                              int resolution_x,
                              int resolution_y);
```

It is good to note that these functions are just simpler versions of the original function used to fill the

LeiaCameraData object (leiaInitializeCameraData). If the plan is to update many of the data members of the LeiaCameraData struct, it is better to simply call leiaInitializeData rather than call each setter of the struct.

Once the LeiaCameraData struct has been updated appropriately, you must again use the “leiaCalculateViews” function to update each view object. If this is not done, then the changes made to the LeiaCameraData struct will not affect the LeiaCameraView objects. For a complete example of using these objects, please take a look at the Samples directory.

Shaders

One of the most important aspects of the LeiaNativeSDK comes from the shaders made available to you. The three combinations provided are effects for handling view interlacing (the different views into a format the display understands), a DOF effect to be used as a final pass across each view, and a sharpening effect used to improve the visual clarity of the final rendering.

View Interlacing

The Leia display provides visuals in new and unique ways. In order for games utilize this technology, games need to draw in a specific way. At Leia, we call this “view Interlacing”. View interlacing is the process of stitching multiple views together into the Leia format. Using shaders provided by the SDK, view interlacing becomes trivial to manage.
In the vertex shader, not much is happening. The shader expects to render to a quad, which has attributes for a vec2 for position and a vec2 for texture coordinates.

In the fragment shader a few more variables are needed. View interlacing itself takes a few different inputs. The first uniform required is to have 4 textures. In this case, the shader expects them in array form. The only other uniform value required is the offset which needs to be queried from the Leia System Service. The shader also needs the texture coordinates passed from the vertex shader so we know where to sample from in our textures. Simple, but effective, which allows high quality content to be displayed when using the Leia platform.

Depth Of Field (DOF)

The DOF effect is significantly more involved than the view interlacing shader. DOF is used to provide an artistic blurring across different depths of the frustum. DOF has another use in 3D though. When pushing the limits of 3D, often users will have trouble visualizing it because the differences between the views becomes too much. DOF can be used to smooth the views, making it much easier to increase the disparity of the views.

The vertex shader is very simple, managing the points of a quad for the view. This requires the position and texture coordinates as vec2 attributes.

The fragment shader is the most complex shader currently provided by the LeiaNativeSDK. This shader needs multiple textures, one for the colored rendering of the final scene with all objects, and the depth buffer filled during that rendering. The aspect ratio is also passed in so the blurring effect will always occur in a circular area. The view width is needed as well to help provide a useful blur radius. Aperture is how the application can scale the blur radius, and “f in pixels” is a value retrieved by using the LeiaCameraData object built when preparing to compute the perspective matrices. Baseline describes the distance between the cameras for each view, and near and far are the world-space values used for the near and far planes.

View Sharpening

It’s possible to see objects with high-levels of 3D, this can be improved further than if we stopped at view interlacing. The next rendering pass handles an effect called view sharpening. View sharpening is a sharpening algorithm specifically designed to make sure the end-user's eyes are getting the most crisp, beautiful image possible.

The view sharpening vertex shader is exactly the same as we discussed for view interlacing. Since the shader is rendering a quad with a texture, the only attributes needed for the shader are position and texture, both of which are vec2s. From here we pass the texture coordinates to the fragment shader, and the interesting work can begin.

The most important part of the of fragment shader is the interlaced_view texture, which is a TEXTURE_2D of the previous pass – view interlacing. The output of the view interlacing pass is passed in for this texture. Another uniform we need is the pixel width. This tells us the size of a single pixel to allow us to run our convolution horizontally. “a” and “b” are hardware values that should be queried from Android, which again can be retrieved using SimpleDisplayQuery.

The texture coordinates of the vertex shader are used as our center-point of the convolution. With this information the shader is fully capable of taking yet another step to improve visual quality and give more impressive 3D.

Accessing the Shaders

In order to use the shaders listed above, the LeiaNativeSDK provides a public function which allows you to query for each vertex or fragment shader as needed. This function returns the shader source as a string, which you can compile using the provided shader compilation function, or with your own application shader functionality. The function which gives you access to the shaders is:

```
const char * leiaGetShader(enum LeiaShader shader,
                           unsigned int * out_size);
```

shader: an enumerated value, listed in the LeiaNativeSDK.h header file. All available shaders have enumerated value, and only values in this list will be accepted by the function.
out_size: the length of the string returned to you as an out parameter.

In order to compile a vertex and fragment pair, the following function is also provided:

```
GLuint leiaCreateProgram(const char * vert_source,
                         const char * frag_source);
```

By providing the shader source from leiaGetShader (or other source files you provide yourself) to this function, the shaders will be compiled into a program which can be used later for rendering.

Leia Render Functions

This section describes how to use the shaders described above with the LeiaNativeSDK.

If you wish for the LeiaNativeSDK to manage the DOF effect, you can invoke the leiaDOF function provided, which looks like the following:

```
void leiaDOF(GLuint rendered_texture,
             GLuint depth_texture,
             LeiaCameraData * data,
             GLuint dof_program,
             GLuint fbo_target,
             float aperture);
```

This helper function minimizes the setup required by your application to use DOF. After fully rendering a scene to a given view, calling this function will update the view appropriately.
rendered_texture: the texture the scene was rendered onto while rendering all the objects.
depth_texture: the depth buffer from rendering the same view.
data: the same structure created when building the camera matrices needed to render each view.
dof_program: a GL program that has previously been compiled and linked. This program is expected to have all the same uniform values as the DOF shaders provided through leiaGetShader.
fbo_target: an fbo with at least one color attachment that can be used in future passes of rendering the

frame.
aperture: used by you to scale the amount of blur used.

After all the different views have been rendered, with or without the DOF effect applied, view interlacing is required. To minimize the work your application is required to do, another helper function is provided called:

```
void leiaInterlace(GLuint * views_as_texture_2d,
                   LeiaCameraData * data,
                   GLuint interlace_program,
                   GLuint fbo_target,
                   int screen_width_pixels,
                   int screen_height_pixels,
                   int alignment_offset);
```

If this function is called with the correct views provided, the final rendering from this function will allow the Leia display to show 3D.
views_as_texture_2d: an array of texture object ids which are the previous view's color buffers.
data: the LeiaCameraData object that was filled when creating the perspective matrices.
interlace_program: a previously compiled and linked program which manages putting the views together into one colored image.
fbo_target: allows you to select which FBO will be rendered into, as long as it has at least one color attachment.
screen_width_pixels and *screen_height_pixels:* the size of the screen in pixels. *alignment_offset:* a value that needs to be retrieved from Android through the SimpleDisplayQuery object.

At the end of the Leia pipeline comes the final pass:

```
void leiaViewSharpening(GLuint interlaced_texture,
                        LeiaCameraData * data,
                        GLuint view_sharpening_program,
                        GLuint fbo_target,
                        int screen_width_pixels,
                        float* act_coefficients,
                        int num_act_coefficients);
```

This is the last function that is provided by the LeiaNativeSDK, and it will allow the final interlaced view to be sharpened even further, improved the 3D quality.
interlaced_texture: the interlaced color buffer from the previous pass.
data: again the LeiaCameraData object filled previously for the perspective matrices.
view_sharpening_program: needs to be a previously compiled and linked program, which can be done using the leiaGetShader function provided along with the leiaCreateProgram function.
fbo_target: allows you to decide which FBO the function will render into.

screen_width_pixels: the size of the screen width currently.
act_coefficients: array of values retrieved from Android through the SimpleDisplayQuery object.
num_act_coefficients: the number of values inside the act_coefficients array.

A general example of rendering a single frame could look similar to the following:

```
for (unsigned int y = 0; y < CAMERAS_HIGH; ++y) {
    for (unsigned int x = 0; x < CAMERAS_WIDE; ++x) {
        unsigned int index = y * CAMERAS_WIDE + x;
        glBindFramebuffer(GL_FRAMEBUFFER, fbos[index]);
        glClearColor(1.0, 0.0, 1.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        RenderView(x, y);
        leiaDOF(render_textures[index],
                depth_textures[index],
                &data, dof_shader.program_,
                fbo_dof[index], 1.0f);
    }
}
leiaViewInterlace(texture_dof, &data, view_interlacing_shader.program_,
                  fullscreen_fbo, screen_width_pixels_, screen_height_pixels_,
                  LeiaJNIDisplayParameters::mAlignmentOffset);
leiaViewSharpening(fullscreen_texture, &data, view_sharpening_shader.program_, 0,
                   screen_width_pixels_,
                   LeiaJNIDisplayParameters::mViewSharpeningParams, 2);
```

Extensions For The LeiaNativeSDK Rendering Functions

Rendering pipelines are varied, and each has their own way of managing objects and effects for display. It is expected that not all applications can use these functions, and others may choose they do not want to. In an attempt to provide as much flexibility as possible while creating your application, the LeiaNativeSDK API for helping render allows for more granular work as well.

Leia provides shaders for your application and if your rendering framework allows it, it is easiest to use the functions previously listed. These manage filling in the shader variables with the needed data, and manage rendering the fullscreen quads as well.

Notice the two larger pieces these functions manage:

1. Setting up the shader
2. Rendering the quad

These two layers are exposed in the LeiaNativeSDK header for you to use.

The following functions allow your application to setup the shaders retrieved from the SDK to simplify setting up the uniform values:

```
void leiaPrepareViewInterlace(GLuint* views_as_texture_2d,
                              LeiaCameraData* data,
                              GLuint view_interlace_program,
                              GLuint fbo_target,
                              int screen_width_pixels,
                              int screen_height_pixels,
                              int alignment_offset,
                              float debug);
void leiaPrepareViewSharpening(GLuint interlaced_texture,
                               LeiaCameraData* data,
                               GLuint view_sharpening_program,
                               GLuint fbo_target,
                               int screen_width_pixels,
                               float* act_coefficients,
                               int num_act_coefficients,
                               float debug);
void leiaPrepareDOF(GLuint rendered_texture,
                    GLuint depth_texture,
                    LeiaCameraData* data,
                    GLuint dof_program,
                    GLuint fbo_target,
                    float aperture,
                    float debug);
```

These functions are the same as their previous respective function, with two large exceptions:

1. No drawing occurs inside these functions. They setup uniform values for the shader you are passing in, allowing your application to manage the quad rendering.
2. A debug value is added, which will allow for mild debugging of the shaders and the rendering when the value is greater than 0.0.

The debugging flag is good to prove a few different aspects of the Leia rendering. Outside of shaders, enabling debugging causes a glClear to occur. This allows us to see if any rendering is occurring, indicating if the draw method is working appropriately. The colors used for clearing are:

leiaPrepareViewInterlacing:	glClearColor(0, 0, 1, 1)
leiaPrepareViewSharpening:	glClearColor(0, 1, 0, 1)
leiaPrepareDOF:	glClearColor(1, 1, 0, 1)

To see how the debug value will affect particular shader passes, please refer to the specific shader being debugged.

An additional helper function has been provided to manage drawing as well. The function looks like:

```
void leiaDrawQuad(GLint program_id,
                  int draw_immediate_mode,
                  unsigned int vbo_id);
```

This function can draw a fullscreen quad for your application. This function requires the same position and texture input variables for the vertex shader. If you choose to modify the shaders this function will continue to work unless the input texture and position variable names change.
program_id: The program being used for rendering which has already been compiled and linked.
draw_immediate_mode: This value flags if the draw will use client side rendering or vertex buffer object rendering.
vbo_id: a vertex buffer object id previously returned from the LeiaNativeSDK library.

This helper function can simplify some work on your side for implementation purposes. However, there are important aspects to note to use this function efficiently. The LeiaNativeSDK library is unable to ensure the GL context life or the life of the VBO id. Due to this, it is not possible to manage VBO ids longer than the lifetime of the single function call. This means that every time this function is called with *draw_immediate_mode* as 0, the function will always recreate a new VBO id and destroy it at the end of the function. By setting *draw_immediate_mode* to 1, or by leaving *draw_immediate_mode* to 0 and giving a previously created VBO id (by the LeiaNativeSDK) to *vbo_id*, the creation and deletion of the VBO id is no longer occurring.

The variable *draw_immediate_mode* is not entirely accurate. When this is true, the LeiaNativeSDK use client-side vertices to render instead of VBOs. In general, this is typically considered slower than using VBOs. However, in this particular case (when *draw_immediate_mode* and *vbo_id* are 0), leiaDrawQuad can render more efficiently. It is important to note that some hardware drivers have issues mixing client and vbo rendering, so even though it may be more efficient this path may not rendering at all and may leave a black screen displayed instead.

We have already stated that *vbo_id* needs to be a previously created by the LeiaNativeSDK. To do this, use the following function:

```
int leiaBuildQuadVertexBuffer(GLint program_id);
```

This function builds a VBO id.
And when your application is finished rendering, it can destroy the id with:

```
void leiaDestroyQuadVertexBuffer(unsigned int vbo_id);
```

An example for using this set of functions (leiaPrepare..., leiaDrawQuad, leiaBuildQuadVertexBuffer, and leiaDestroyQuadVertexBuffer) can be found in the samples provided, inside “TeapotsWithLeia”.

Querying the Leia System Values

The Leia Native SDK helps enable your application for rendering for the Leia hardware, but in order to use the hardware we have to interact with the Android system. In your application, it will be necessary to interact with the Java layer to easily communicate the needs of your application to the hardware. In an effort to minimize the work involved in this, Leia provides a module which can be easily included in your Android Studio project.

Including the Leia module in Android Studio

The first step to interacting with the Leia system is to include the Android module in your build.gradle file for the module which will be using the Leia Display. You can do this simply by adding two snippets:

```
repositories {
    maven {
        url 'https://leiainc.jfrog.io/leiainc/gradle-release'
    }
}
```

The above snippet shows how to access the module directly, and Android Studio will pull from this repository. You still need to inform Android Studio what it needs to find at this repository, which is where the second snippet comes in:

```
dependencies {
    implementation 'com.leia:leiasdk:4+'
}
```

It should be noted that “implementation” is the newer keyword. Older projects may require this to become “compile”.

Once this is added to the build.gradle, sync the project so some Java code can be added in.

Accessing Leia System Functionality Through Java

The Leia Native SDK is built to enable applications running from C/C++ code to take full advantage of the Leia device. However, a Java interface is the standard way to interact with the software system on Android. The additional code come in two ways: You need to be able to control the Leia Display by choosing when to enable or disable the 3D backlight, and you will need to provide system values to the native code in order to maximize quality.

Interacting with the 3D Backlight

Changing the 3D backlight is designed to be easy, giving you control when and how you need it. The easiest way to interact with the 3D backlight is by creating a LeiaDisplayManager object, and using this to enable or disable it. Below is how you can instantiate the object:

```
import com.leia.android.lights.LeiaDisplayManager;
import static com.leia.android.lights.LeiaDisplayManager.BacklightMode.MODE_2D;
import static com.leia.android.lights.LeiaDisplayManager.BacklightMode.MODE_3D;

public class MyNativeActivity extends NativeActivity {
    private LeiaDisplayManager mDisplayManager;
}
```

Here you can see how to import the manager into the code, and then we create a class variable for it. From here, you can follow it up by creating the actual instance:

```
mDisplayManager = LeiaSDK.getDisplayManager(this);
```

Nothing special is required here, except that the LeiaDisplayManager requires the context to be passed in.

For a simple way to interact with the Leia Display, you can enable or disable easily with functions similar to the below:

```
public void Enable3DBacklight() {
    if (mDisplayManager != null) {
        mDisplayManager.setBacklightMode(MODE_3D);
    }
}
public void Disable3DBacklight() {
    if (mDisplayManager != null) {
        mDisplayManager.setBacklightMode(MODE_2D);
    }
}
```

With these added to your NativeActivity class (or the class you derived from NativeActivity), it is possible to turn the 3D backlight on and off. If the application is using the Leia Native SDK and the backlight is on, you will likely see the objects in the scene appearing to pop out or fall into the screen.
In the sample provided, the Enable3DBacklight and Disable3DBacklight functions are used in the activity lifecycle to ensure the backlight is only on when the application is open. It will be important to

test all scenarios for your specific application to ensure the backlight is enabled at the appropriate times.
The Manager object allows you to access any information needed about the Leia Display. You could learn everything you need in order to fully utilize the hardware. Even though it gives you complete information, Leia provides an easier way to get the values needed for the LeiaNativeSDK.

Querying Leia System Parameters

As you did above, you could use the Display Manager object to query all the system values needed. However, this could be simplified by using a SimpleDisplayQuery object. First you must create the Query object:

```
import com.leia.android.lights.SimpleDisplayQuery;

public class MyNativeActivity extends NativeActivity {
    static SimpleDisplayQuery mLeiaQuery;
}
```

This code gives us the ability to reference the Query object through static functions, which is how you can more simply talk to the Native code in your application, as we will see later.
We create the SimpleDisplayQuery object in the OnCreate function as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mLeiaQuery = new SimpleDisplayQuery(this);
}
```

Now you can directly, with a single line, retrieve the data needed for the LeiaNativeSDK functions. In the provided samples, The native C++ code will use JNI to query the android system for this information. The Java layer of this code is seen here:

```
public float[] GetViewSharpening() {
    return mLeiaQuery.GetViewSharpening(mIsDeviceCurrentlyInPortraitMode);
}

public float GetAlignmentOffset() {
    return mLeiaQuery.GetAlignmentOffset(mIsDeviceCurrentlyInPortraitMode);
}
public int GetSystemDisparity() {
    return mLeiaQuery.GetSystemDisparity();
}

public int[] GetScreenResolution() {
    return mLeiaQuery.GetScreenResolution(mIsDeviceCurrentlyInPortraitMode);
}

public int[] GetNumAvailableViews() {
    return mLeiaQuery.GetNumAvailableViews(mIsDeviceCurrentlyInPortraitMode);
}

public int[] GetViewResolution() {
    return mLeiaQuery.GetViewResolution(mIsDeviceCurrentlyInPortraitMode);
}
```

The SimpleDisplayQuery object requires information about the current orientation of the device, which is provided by the Boolean variable:

```
private boolean mIsDeviceCurrentlyInPortraitMode;
```

And the samples currently make sure it is correct by updating it inside of onResume in order to make sure after each rotation it is properly updated:

```
mIsDeviceCurrentlyInPortraitMode = IsPortraitCurrentOrientation();
```

The samples use the following method to check the device orientation:

```
public boolean IsPortraitCurrentOrientation() {
    boolean is_portrait_current_orientation = false;
    DisplayMetrics dm = new DisplayMetrics();
    getWindowManager().getDefaultDisplay().getMetrics(dm);
    int width = dm.widthPixels;
    int height = dm.heightPixels;
    if (height > width) {
        is_portrait_current_orientation = true;
    }
    else {
        is_portrait_current_orientation = false;
    }
    return is_portrait_current_orientation;
}
```

IsPortraitCurrentOrientation asks for the current orientation of the device from the operating system. Then, based on how Android responds, report that as “is portrait” or not. This is the final piece needed on the Java side before we request this from the native code.

Getting System Parameters Into Native Code

In order to access the system parameters, there are two possibilities: Use JNI code to interact with the system directly, or use the android_app pointer given inside the NDK android_main function. Given how

the Native Activity works, it is easiest to use the android_app pointer, which gives you access to your java object. In this documentation, the class you have access to is MyNativeActivity (which we reference above on page 17). The samples will have their own names, such as “TeapotNativeActivity” or “MoreTeapotsNativeActivity”.
From the C/C++ code, you have direct access to your main activity class, you can call functions from this class in the C/C++ code without too much trouble. Here, you will see a quick description for how you can do so.
In the provided samples, the system parameters are updated every time the display is initialized which should include every orientation change. The function the samples use is:

```
int Engine::InitDisplay(android_app * app) {
    GetSystemParameters();
}
```

The GetSystemParameters function was added to the Engine class, and uses a class implemented by Leia, called LeiaJNIDisplayParameters:

```
bool Engine::GetSystemParameters(void) {
    return LeiaJNIDisplayParameters::ReadSystemParameters(app_->activity);
}
```

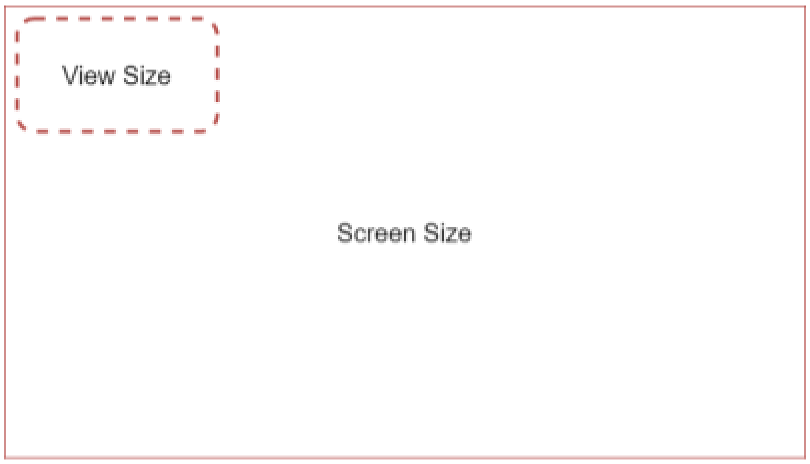
Once the ReadSystemParameters function is called, the LeiaJNIDisplayParameters object holds on to the data and you can pull the values out whenever it is convenient for you and your application. This class is provided in order to give a complete understanding of how you might create your own implementation, or if this handles all situations your application requires it is available for your use. Please refer to the source of LeiaJNIDisplayParameters for how it works.

Updating The GL Pipeline

When integrating the Leia Native SDK, you may want to update the OpenGL pipeline. The modifications listed below are the recommended approach to handle this update to work efficiently.

Render Target Size

One of the most important parts of the rendering setup is the size of the render targets each part the pipeline will be filling. It is common to have the main camera rendering into a view the size of the screen, which fills the number of pixels in a display.

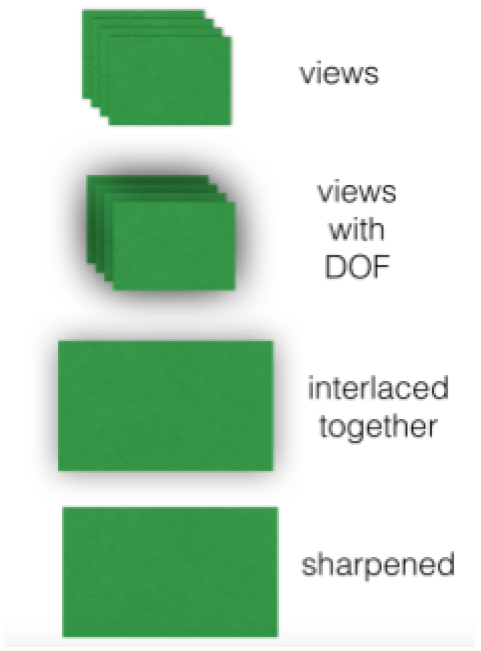


This shows the difference in size of a view versus the screen. By having views that are 1/16th the size of the original screen, much less rendering is occurring.

One of the most important changes to rendering with the Leia Native SDK is rendering four views instead of just one from the main camera. The Leia-enabled display shows the four views; each of which vary depending on how the device is positioned in front of you. By splitting the display into multiple views, the resolution of each view is reduced, and in this case the reduction is one fourth the width and one fourth the height. Effectively the four views can render one sixteenth the number of pixels. When the number of pixels are added together from each view, this renders one fourth the total number of pixels the original full screen application would render. It is important to note that the CPU effort is increased due to the number of draw calls increasing by a factor of four (the same number of draws from the original full screen game are processed per view).

This same optimization works for the Depth of Field (DOF) effect. The DOF effect works on each of the four views, allowing the render to occur at one fourth the width and one fourth the height of the display resolution.

The end of the Leia Native SDK pipeline operates on the larger view size, which is equivalent to the final resolution of the application. With the view interlacing and view sharpening passes rendering at the full screen resolution, the rendering load is potentially less than that of the original application because of overdraw content and the CPU effort is increased slightly.



The renderloop described by surfaces rendered. Note that the views (including depth of field) require rendering 4 times, but they are at 1/16th the size of the screen size. The interlacing and sharpening steps are screen size and rendered once.

One final optimization which is possible is to render the view interlacing and view sharpening passes at one fourth the height. The current implementation of the view interlacing shader is to copy a set of four pixels (one for each view) to be four high (one for each potential view). This means that the current shader renders a block of four pixels high as the exact same color for every view. If the shader were to be modified to only render one fourth the height, then the view sharpening pass were ran, a final pass could be used to scale the height. It is also possible that instead of scaling the height the default render target which appears on the display could be declared at one fourth the height and the display hardware could upsample the final output directly, saving even more rendering and CPU time.

Managing Depth

One of the most common states of GL to update is the depth test. For normal rendering (the rendering of objects in a scene) depth is usually turned on to limit overdraw, keeping the object at each pixel which is closest to the camera as what is finally rendered. However, this test becomes less important when rendering a quad which covers the full view, such as what occurs for the Depth of Field effect, view interlacing effect, and view sharpening effect. These functions explicitly disable this test due to potential FBO management issues, meaning two things for your pipeline. The first being if you want the depth to be turned on after calling one of the rendering functions from the Leia Native SDK, it will be necessary for you to enable it from your pipeline. The second is, because the depth test will be disabled during the rendering of the depth of field, view sharpening, and view interlacing passes, having the depth attached becomes useless. Since it will not be used, taking the time to prepare the depth target for the framebuffer uses resources from the CPU unnecessarily.

Updating the Shader Uniforms

It is important for you to note that the Leia rendering functions (leiaDOF, leiaViewInterlacing, leiaViewSharpening) will attempt to interact with specific uniform and vertex data. The uniform values are expected to continue to be available by the string name. The shaders are provided in case experienced users wish to change the functionality to work better for their own needs, but in order for the render functions provided by Leia to work correctly the uniform and input variables which are already in use should remain.

Texture Parameters

When using the leiaDOF function, a kernel is being used on the input image data which effects the output. The kernel size can be fairly large, and around the edges this causes some points of the convolution to be outside the texture bounds. When this occurs, the way the texture is created and the parameters provided to it (or the sampler) become important. For example, when the texture/sampler wrap mode (GL_TEXTURE_WRAP_S/T) is not set to GL_CLAMP_TO_EDGE or GL_CLAMP_TO_BORDER then odd rendering artifacts appear. It is recommended that wrapping modes which enable reading from another edge of a texture be disabled to reduce the artifacts of rendering near the edges.

Overview of Lightfield Rendering

Traditional 3D Rendering

In traditional 3D computer graphics, scenes are drawn with respect to a camera. This camera is the idealized pinhole camera, with a viewable area which is known as a frustum. The frustum is calculated based on a field of view of the camera, as well as the closest and furthest distances the camera is able to see from its current position. When all this data is put together, a perspective matrix can be computed, which is later used for rendering geometry. The frustum for this type of camera is symmetric, so when divided in half vertically or horizontally, the frustum is the same size. Objects are then drawn onto the near plane based on this projection matrix. The entire scene is drawn, with potentially many objects. When finished, the final frame will show everything in focus, unless effort is taken to create visual effects to make the scene appear similar to what the world would look like from a human eye (such as blurring, lighting changes, etc).

Lightfield Rendering

When rendering for the Leia display, which is using lightfield technology, everything becomes more interesting. In the current version, Leia suggests having a total of 4 cameras. While drawing objects with these cameras is similar to the traditional method, due to the requirements of Leia rendering we need to change some aspects.

The human brain is able to visual depth because the two eyes see a very similar world, but their location is different. Each eye sees the world, and the brain takes the image, combines it with the image from the other eye, and during this combination we are able to visualize objects being closer or more distant than others.

In order for Leia rendering to accomplish the same job, we need more than the normal, single camera. The more cameras used, the more of a smooth 3D effect we can see. The current Leia display technology recommends using a set of 4 by 1 camera setup. This means we need a total of 4 cameras next to each other. These cameras are centered around one point, and the distance each is away from their neighbor is called the camera **baseline**.

This is different from traditional computer graphics because the cameras need to be parallel. However, the camera frustums of the 4 cameras must also intersect at a single plane. We must apply a shear to the perspective matrices to achieve this. If the scene only contained a single object, say a square, that existed exactly on the plane of intersection, then all of the cameras would see the exact same view. This plane of intersection is called the **convergence plane.** When an application is running, the elements the developer wants the user paying attention to should be on or very near this plane, and if necessary new perspective matrices should be created when the applications main focus moves around in the frustum.

When objects are in front of the focal plane, or behind it, the object will look slightly different to each view. The difference, in pixels, between one point of an object from views which are next to each other is called **disparity**. The further the cameras are away from each other, the more disparity will exist between objects when projected, and the more different the neighboring views will be.

When these 4 views are appropriately combined, a process called **view interlacing**, a 3D effect will appear on the Leia display. From a correctness perspective, everything is working.

In order to create a bigger effect of 3D, we need to view objects further from the focal plane, or move the cameras further apart. In either of these cases, the effect will be that the views are more different. When the disparity becomes “too large”, humans have a harder time mentally putting the images together. This makes the scene confusing or worse. At this point we have to either:

- Keep the views fairly similar in order to keep the brain believing in the depth shown
- Blur each view more the further from the focal plane the pixel is based on depth.

Leia recommends applying a DOF effect to each view before the view interlacing occurs. Without DOF, a general statement of 6 pixels of disparity are acceptable. However, when DOF is properly added, disparity can become significantly larger, giving the developer and designer the opportunity to push the 3D effect as far as possible. As the developer, this allows the views to be significantly more different, creates a more impressive 3D, as well as providing the end-user with a specific and obvious location to focus on for a minimal amount of computational effort in traditional 3D computer graphics, scenes are drawn with respect to a camera.

Content Design

Scene Design

On the Leia display, managing the depth of your scene is an important design consideration. The most important content in your scene should intersect with the convergence Plane.

The convergence plane is the area of the scene with the sharpest area of focus. The convergence plane sits at the position of the physical surface of the display. Content that is in front of the convergence plane will extend out from screen, and content that is behind will recede into the screen.

Design your content to have layers of depth to accentuate the parallax motion between foreground, midground and background objects. Placing content at different depths that move with parallax greatly enhance the sense of 3D in your projects.

Controlling Depth

Adjusting the baseline scaling between each virtual camera controls the depth of the 3D effect of your rendering. As these cameras move farther apart, the apparent 3D effect increases but area of critical focus becomes shallower.

If all cameras are positioned with a baseline of zero (when baseline scaling is set to 0.0), all cameras are seeing an identical image and the 3D effect disappears. If the cameras are spaced farther apart, the 3D effect and parallax effects are increased. However individual views of objects farther away from the convergence plane become more apparent.

Tuning the baseline scaling of the LeiaCameraData object will allow you to balance the depth of the rendering and the image quality of the scene.

Tips and Tricks

Shader Compilation

To improve startup time of an application is to compile all shaders and later link them. This way a graphics driver can batch-compile them, saving significant time. When using the leiaCreateProgram function, this type of optimization is not possible unless these are the last two shaders compiled, and all other application shaders are linked into programs after.

Handling all shaders together is a much better approach. In this case, compiling and linking would be done in two different steps, and the Leia shaders would be included with those from the original program.

Setup Cameras Once

In most games, the way the camera renders, as defined by its projection matrix, tends not to differ between frames. While there are plenty of scenarios where the camera may zoom in, on average the projection matrix does not change. Later, the camera will move and the projection matrix is multiplied by the camera's position matrix and you now have a scene.
When using the Leia technology, you need to render the scene from each view. These extra views increase the number of matrix calculations that are necessary. Though setting up the camera matrices does not require lots of computation, it is generally more preferred to be as optimal as possible in order to minimize heat and power loss. Because of this, Leia recommends computing the matrices once or as needed instead of every frame.

UI, Menus and Text

In gaming, it is typical to find entire screens which are of 2D. Even if the buttons are lit and rounded, they still tend to be just textured as if they existed in 3D, instead of actually in 3D. For example, the start menu to most games tends to be entirely flat. In situations where the objects in the game are all rendered very near the same plane (such as menus), enabling the Leia 3D will not yield any results.
While Leia does everything possible to have the best performance that is feasible, some performance impact is expected over 2D rendering. Due to this, when applications are using only 2D rendering or only rendering objects that are in a similar plane, do not enable 3D. Doing so will cause extra work to render in ¼ the resolution and will have no 3D effect, making the loss of extra power wasted.

Minimize GL draws with culling objects by area

In the average 3D application, often there are many objects in the world. One of the most costly processes in rendering with OpenGL ES is calling or similar functions.

```
glDrawArrays(...)
```

The outcome of this is a large use of CPU computation and power. In the general case (without using the multiview extension), the Leia display requires the application to render each object for each view. This drastically increases the number of draws, and therefore also increases the CPU effort of the device.
Many games implement a data structure to minimize the number of objects that are required to be drawn (KD-Tree is one example). With the number of draws doubling because of increase views, this type of data structure would pay for itself very quickly in performance.

Common Issues

Step	Failed to resolve: com.leia:leiasdk:4.1.0	
1	Verify the url value is correct	url 'https://leiainc.jfrog.io/leiainc/gradle-release'

Step	Failed to resolve: ...	
1	Verify the leia library package name is correct	implementation 'com.leia:leiasdk:4.1.0'

Appendix A: Example build.grade

```
apply plugin: 'com.android.application'

repositories {
    maven {
        url 'https://leiainc.jfrog.io/leiainc/gradle-release-local'
    }
}

android {
    compileSdkVersion = 23
    defaultConfig {
        applicationId = 'com.sample.teapot'
        minSdkVersion 23
        targetSdkVersion 23
        ndk {
            abiFilters 'arm64-v8a'
        }
        externalNativeBuild {
            cmake {
                cppFlags "-std=c++11 -stdlib=libc++ -O2"
                arguments '-DANDROID_PLATFORM=android-21',
                        '-DANDROID_TOOLCHAIN=clang', '-DANDROID_STL=c++_static'
            }
        }
    }
    buildTypes {
        release {
            minifyEnabled = false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                    'proguard-rules.pro'
        }
    }
    externalNativeBuild {
        cmake {
            path 'src/main/cpp/CMakeLists.txt'
        }
    }
    sourceSets {
        main {
            jniLibs.srcDirs = ['./../distribution/leia_sdk/lib']
        }
    }
    compileSdkVersion 23
    productFlavors {
    }
}

dependencies {
    compile 'com.leia:leiasdk:4.2.1'
    compile fileTree(include: ['*.jar'], dir: '.')
    compile 'com.android.support:appcompat-v7:23.4.0'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
}
```

Appendix B: Example CMakeLists.txt

```
#
# Copyright (C) The Android Open Source Project
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

cmake_minimum_required(VERSION 3.4.1)

# configure import libs

# build native_app_glue as a static lib
add_library(native_app_glue STATIC
    ${ANDROID_NDK}/sources/android/native_app_glue/android_native_app_glue.c)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=gnu++11 -Wall -fno-exceptions -fno-rtti")

# build the ndk-helper library
set(ndk_helper_dir ../../../../common/ndk_helper)
add_subdirectory(${ndk_helper_dir} ndk_helper)

# Export ANativeActivity_onCreate(),⍰
# Refer to: https://github.com/android-ndk/ndk/issues/381.
set(CMAKE_SHARED_LINKER_FLAGS
    "${CMAKE_SHARED_LINKER_FLAGS} -u ANativeActivity_onCreate")

# now build app's shared lib
add_library(MoreTeapotsNativeActivity SHARED
    LeiaJNIDisplayParameters.cpp
    MoreTeapotsNativeActivity.cpp
    MoreTeapotsRenderer.cpp)

target_include_directories(MoreTeapotsNativeActivity PRIVATE
    ${ANDROID_NDK}/sources/android/cpufeatures
    ${ANDROID_NDK}/sources/android/native_app_glue
    ${ndk_helper_dir})

# add lib dependencies
target_link_libraries(MoreTeapotsNativeActivity
    android
    native_app_glue
    atomic
    EGL
    GLESv2
    log
    ndk-helper)

set(distribution_DIR ${CMAKE_SOURCE_DIR}/../../../../distribution)
add_library(lib_leia_sdk STATIC IMPORTED)
set_target_properties(lib_leia_sdk PROPERTIES IMPORTED_LOCATION
    ${distribution_DIR}/leia_sdk/lib/${ANDROID_ABI}/libleiasdk.so)
target_include_directories(MoreTeapotsNativeActivity PRIVATE
    ${distribution_DIR}/leia_sdk/include)
target_link_libraries(MoreTeapotsNativeActivity
    lib_leia_sdk)
```