LPGPU2 Font Renderer App

Modifications to work with the LPGPU2 Tool Suite

Introduction

As part of LPGPU2 Work Package 3, a font rendering app was developed to research the benefits of font rendering on the GPU (vs CPU as is normally done) More details on this can be found in D3.3 (which will be available on the website shortly after the end of the project).

This application was completed before the LPGPU2 tool was sufficiently mature to be used for profiling and as such its internal structure required some minor changes in order to work with the tool.

The initial GUI presented to the user displays a configuration screen consisting of a set of radio buttons for selecting the font rendering mode to test, a sliding scale to choose the volume of text to render, and a textbox for entering the text to render. This configuration screen is shown in Figure 1.

The sliding scale allows the user to tune the workload for frame rate. The value of the scale is the number of times the text is to be drawn per frame. The higher the value, the more text there is to render per frame and the lower the frame rate. The ‘Text to draw’ textbox on the interface is for specifying the text that will be rendered. In the present experiments, the default string was always used.

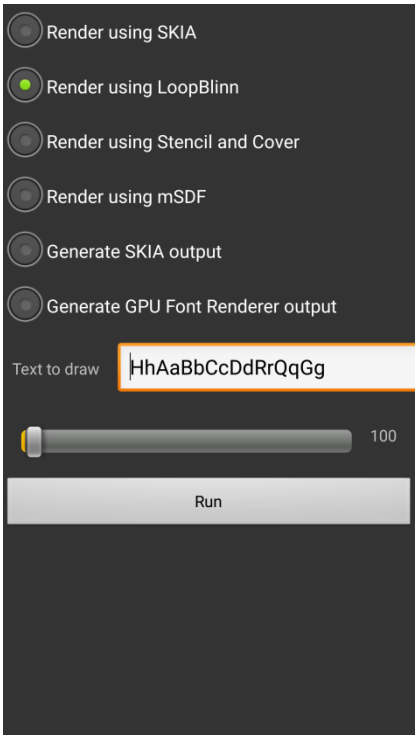
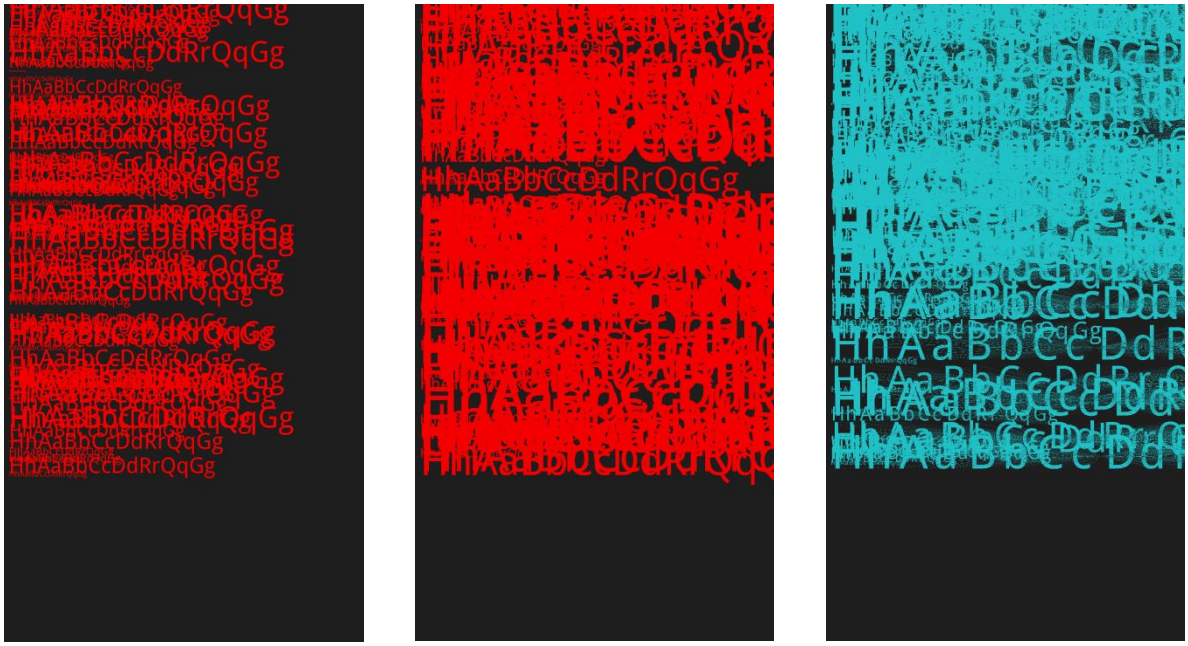


Figure 1 Font Renderer App Configuration Activity

When the user clicks Run, the app switches from the Java activity of the GUI to a native C++ activity to do the actual rendering. Figure 2 shows the output from three different rendering modes: SKIA – the default android mode, Loop-Blinn – an algorithm that shifts the compute burden of text rendering from the CPU to the GPU, and Stencil-and-Cover – an alternative mode provided for comparison, which uses the Loop-Blinn (GPU-heavy) method to render to the stencil buffer and then draw to the whole screen with two large triangles, allowing the text to 'show through' the stencil.

There is a qualitative difference in the rendering modes however for the purposes of this study we do not consider any visual disparities and only consider the implications to the system of rendering in the different modes.



a) SKIA b) Loop-Blinn c) Stencil and Cover

Figure 2 Font Rendering Modes

Build System

The existing Font Renderer app build system involved a sequence of command line actions for the native build followed by a standard Make Project type build in Android Studio. By contrast, the other LPGPU2 test apps – Vulkan, GL/EGL and OpenCL – all can be built entirely within the Android Studio environment with a single click. Deploying the existing apps to a target device is similarly trivial.

In order to have the same level of convenience and have the Font Renderer app behave in exactly the same way as all other LPGPU2 test apps two high level changes needed to be made: firstly, the existing Font Renderer app had to be moved from its present build system to a simpler, more intuitive process that can be operated entirely from Android Studio; and secondly, the app needed to use the LPGPU2 Shim version 2, which will allow the app to be detected by the LPGPU2 Remote Agent app and profiled with the LPGPU2 Profiling Tool.

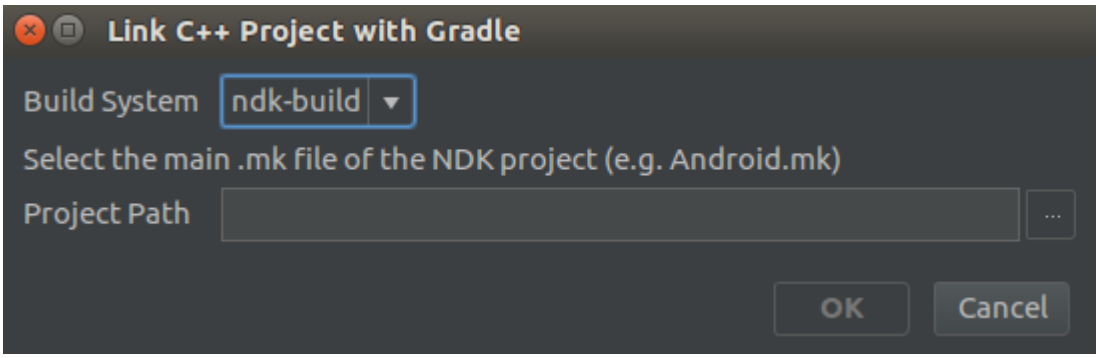
Improved Build System

The first decision to make was whether it would be more efficient to start with a fresh Android Studio project and migrate the components in bit by bit, or whether to take the existing application and incorporate the standalone C++ project into it.

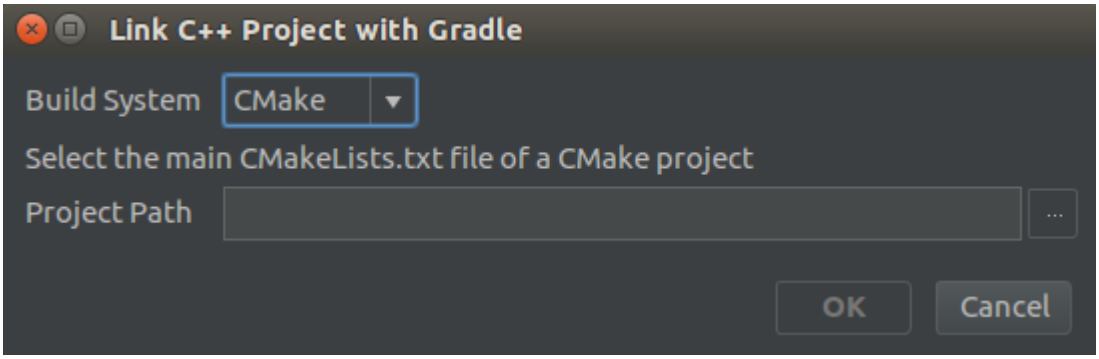
Starting with a greenfield project is always attractive, but the Font Renderer Java activity is much more involved than the existing LPGPU2 apps because of its comparatively busy layout and resources. On the other hand, the associated Android Studio Project had no concept of a C++ sub-project but used the pre-compiled shared objects directly, so it was not immediately obvious which path to take.

Adding native C++ to an existing Android Studio project is very easy (at least in version 2.3.3):-

To do this, right click on the module in the Project navigation tab, and select “Link C++ Project with Gradle”. Figure 3 shows the dialog that appears, asking the user to choose a build system and path to a valid *makefile*. The existing LPGPU2 test apps use *CMake*, so we specify *CMakeLists.txt*. That file does not yet exist but we can still specify it here.



Use the NDK build system



Use CMake build system

Figure 3 Android Studio provides two alternative build systems

The reason *CMakeLists.txt* does not exist is that the native part of the project uses *ndk-build* – the Native Development Kit (NDK) build system which looks for an *Android.mk* file - another type of make system used by the NDK. The first part of the migration was to recreate the build structure of *Android.mk* in *cmake*. This was not too difficult because apart from syntax, the two build systems are solving the same problem in a very similar way. Figure 4 compares a section of the *Android.mk f*ile and the equivalent section of the new *CMakeLists.txt*. Their mutual similarity is clear.

```
FREETYPE_SRC_PATH := ../../3rdparty/freetype2-android-master/

LOCAL_MODULE := freetype2-static

LOCAL_SRC_FILES := \
        $(FREETYPE_SRC_PATH)src/autofit/autofit.c \
        $(FREETYPE_SRC_PATH)src/base/basepic.c \
        $(FREETYPE_SRC_PATH)src/base/ftapi.c \
        $(FREETYPE_SRC_PATH)src/base/ftbase.c \
        $(FREETYPE_SRC_PATH)src/base/ftbbox.c \
```

```
set(FREETYPE_SRC_PATH "../../3rdparty/freetype2-android-master/")

set(FREETYPE_SRCS
 ${FREETYPE_SRC_PATH}/src/autofit/autofit.c
 ${FREETYPE_SRC_PATH}/src/base/basepic.c
 ${FREETYPE_SRC_PATH}/src/base/ftapi.c
 ${FREETYPE_SRC_PATH}/src/base/ftbase.c
 ${FREETYPE_SRC_PATH}/src/base/ftbbox.c
```

```
        $(FREETYPE_SRC_PATH)src/base/ftbitmap.c \
        $(FREETYPE_SRC_PATH)src/base/ftdbgmem.c \
        $(FREETYPE_SRC_PATH)src/base/ftdebug.c \
        $(FREETYPE_SRC_PATH)src/base/ftglyph.c \
        $(FREETYPE_SRC_PATH)src/base/ftinit.c \
        $(FREETYPE_SRC_PATH)src/base/ftpic.c \
        $(FREETYPE_SRC_PATH)src/base/ftstroke.c \
        $(FREETYPE_SRC_PATH)src/base/ftsynth.c \
        $(FREETYPE_SRC_PATH)src/base/ftsystem.c \
        $(FREETYPE_SRC_PATH)src/cff/cff.c \
        $(FREETYPE_SRC_PATH)src/pshinter/pshinter.c \
        $(FREETYPE_SRC_PATH)src/psnames/psnames.c \
        $(FREETYPE_SRC_PATH)src/raster/raster.c \
        $(FREETYPE_SRC_PATH)src/sfnt/sfnt.c \
        $(FREETYPE_SRC_PATH)src/smooth/smooth.c \
        $(FREETYPE_SRC_PATH)src/truetype/truetype.c

QTX_PATH := $(LOCAL_PATH)/../../Qt_modified/


### include paths ###
LOCAL_C_INCLUDES += $(LOCAL_PATH)/
LOCAL_C_INCLUDES += $(QTX_PATH)/global
LOCAL_C_INCLUDES += $(QTX_PATH)/geometry
LOCAL_C_INCLUDES += $(QTX_PATH)/containers
LOCAL_C_INCLUDES += $(QTX_PATH)/opengl
LOCAL_C_INCLUDES += $(QTX_PATH)/gui
LOCAL_C_INCLUDES += $(QTX_PATH)/utils
LOCAL_C_INCLUDES += $(LOCAL_PATH)/../../freetypeloader
LOCAL_C_INCLUDES += $(LOCAL_PATH)/../../loopblinn
LOCAL_C_INCLUDES += $(LOCAL_PATH)/../../loopblinntriangulator
LOCAL_C_INCLUDES += $(LOCAL_PATH)/../../textcontainers
LOCAL_C_INCLUDES += $(LOCAL_PATH)/../../fontbench
LOCAL_C_INCLUDES += $(LOCAL_PATH)/../../3rdparty/freetype2-android-
master/include



LOCAL_MODULE    := libspr2
```

a) Android.mk

```
 ${FREETYPE_SRC_PATH}/src/base/ftbitmap.c
 ${FREETYPE_SRC_PATH}/src/base/ftdbgmem.c
 ${FREETYPE_SRC_PATH}/src/base/ftdebug.c
 ${FREETYPE_SRC_PATH}/src/base/ftglyph.c
 ${FREETYPE_SRC_PATH}/src/base/ftinit.c
 ${FREETYPE_SRC_PATH}/src/base/ftpic.c
 ${FREETYPE_SRC_PATH}/src/base/ftstroke.c
 ${FREETYPE_SRC_PATH}/src/base/ftsynth.c
 ${FREETYPE_SRC_PATH}/src/base/ftsystem.c
 ${FREETYPE_SRC_PATH}/src/cff/cff.c
 ${FREETYPE_SRC_PATH}/src/pshinter/pshinter.c
 ${FREETYPE_SRC_PATH}/src/psnames/psnames.c
 ${FREETYPE_SRC_PATH}/src/raster/raster.c
 ${FREETYPE_SRC_PATH}/src/sfnt/sfnt.c
 ${FREETYPE_SRC_PATH}/src/smooth/smooth.c
 ${FREETYPE_SRC_PATH}/src/truetype/truetype.c
)

include_directories(
 ../../fontbench

 ../../Qt_modified/global
 ../../Qt_modified/geometry
 ../../Qt_modified/containers
 ../../Qt_modified/opengl
 ../../Qt_modified/gui
 ../../Qt_modified/utils

 ../../msdf
 ../../msdf/core

 ../../freetypeloader
 ../../loopblinn
 ../../loopblinntriangulator
 ../../textcontainers
 ../../stencilandcover

 ${FREETYPE_SRC_PATH}/include
)

add_library(spr2    SHARED ${SPR2_SRCS})
```

b) MakeLists.txt

Figure 4 NDK-Build and CMake build systems performing similar tasks

Now that the Project builds entirely from within Android Studio, deploys on a target device, and runs successfully, the next step is to insert the LPGPU2 Shim v2.

Updating existing apps to use the shim is a tried and tested process that we have performed many times. The next section outlines the steps needed to make an app profilable by the LPGPU2 Profiling Tool.

Inserting the LPGPU2 V2 Shim

Converting an app to work with the LPGPU2 Shim v2 involves a number of steps relating to the Java, native C++ code and libraries:

Java:

Import the following three java files into the build:

- LPGPU2DataPacket.java
- LPGPU2RMessengerInterface.java
- ShimActivity.java

Follow the instructions in the comments at the top of *ShimActivity.java* that basically say, change your custom Activity to derive from *ShimActivity* and edit *ShimActivity* to derive from whatever your custom Activity originally derived from.

Rename the Java package to anything involving the string “lpgpu” in it, such as:

- org.lpgpu.myapp
- com.domain.mylpgpuapp

Finally, for the Java part of the app, inserting the following code block into the static area of the Activity loads the shim and other libraries before they need to be used:

```
try {
      System.LoadLibrary(“spr2”); // Application Code
      System.LoadLibrary(“Gles2Shim”); // LPGPU2 Shim
      System.LoadLibrary(“gnustl_shared”); // C++ STL
} catch(UnsatisfiedLinkError e) {
      // deal with missing error
}
```

Any other parts of the Java code using these libraries (via JNI) may need to have this static library load-block too, because although these libraries are shared objects, it is not always clear which Java class will be instantiated first. To be safe, load them in the static area of all classes that will use them so that whichever class loads first, these libraries will be loaded once, and before the first class is instantiated.

C++:

Import the following C++ files into the native build:

- shim2ify.h
- lpgpu2_api.h

Edit *shim2ify.h* according the instructions in the comments at the top of that file that basically say, edit the JNI call macro to reflect the new package name:

```
#define LPGPU2_JNICALL(F) Java_org_lpgpu_fontrenderer_ShimActivity_##F
```

This macro is used throughout the file so all JNI calls in that file will now refer to the new package.

Libraries

Copy the *libGles2Shim.so* and *libgnustl_shared.so* shared object library files (*libGles2Shim.so* is the actual shim) into the *jniLibs* area of the project. In the build scripts, the link directives to GL and EGL must be replaced with a link to the shim. Android studio builds the code for all selected Application Binary Interfaces (ABIs). Different version of the libraries are built and put in different architecture directories. An example makes it clearer. Figure 5 shows a recursive directory listing of the libs folder of the new Font Renderer project.

```
jniLibs/:
arm64-v8a    armeabi    armeabi-v7a    x86

jniLibs/arm64-v8a:
libGles2Shim.so    libgnustl_shared.so

jniLibs/armeabi:
libGles2Shim.so    libgnustl_shared.so

jniLibs/armeabi-v7a:
libGles2Shim.so    libgnustl_shared.so

jniLibs/x86:
libGles2Shim.so    libgnustl_shared.so
```

Figure 5 Architecture dependent directories

Conclusion

Because this shimify process involves many steps, a script was developed to automate parts of the process, and although the script works only for specific apps, it will be provided with the LPGPU2 Profiling Tool infrastructure code as an example of how user can automate this process for their own apps.

The resulting project is now ready to be installed on a suitable target device, recognized by the LPGPU2 Remote Agent App. Figure 6 shows a screenshot of the LPGPU2 Profiling Tool capturing data from the new Font Renderer in flight.

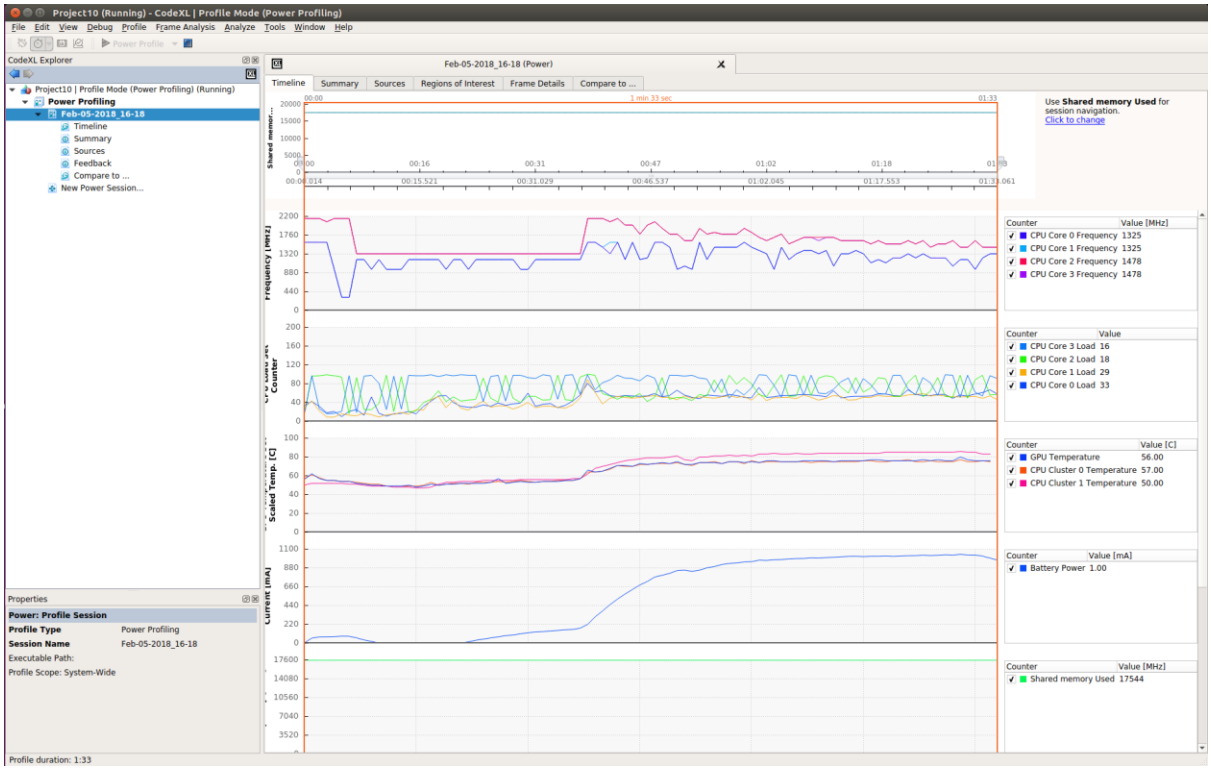


Figure 6 The LPGPU2 Profiling Tool capturing live data from the Font Renderer App

Next Steps

In an upcoming article we will discuss the results of analysing the CPU vs GPU font rendering implementations. Check back soon!