Demo framework overview

[1. Summaries 3](#_Toc423685380)

[1.1. Executive summary 3](#_Toc423685381)

[1.2. Technical overview 3](#_Toc423685382)

[2. Introduction 4](#_Toc423685383)

[3. Design overview 5](#_Toc423685384)

[4. High level overview 6](#_Toc423685385)

[4.1. DemoMain 6](#_Toc423685386)

[4.2. DemoHost 6](#_Toc423685387)

[4.3. DemoApp 6](#_Toc423685388)

[5. Demo application details 7](#_Toc423685389)

[5.1. Demo method overview 7](#_Toc423685390)

[5.2. Fixed or variable timestep update 8](#_Toc423685391)

[5.3. Execution order of methods during a frame 8](#_Toc423685392)

[5.4. Exit 8](#_Toc423685393)

[5.5. Dealing with screen resolution changes 9](#_Toc423685394)

[5.6. Content loading 9](#_Toc423685395)

[5.7. Demo registration 10](#_Toc423685396)

[6. Demo playback 11](#_Toc423685397)

[6.1. Command line arguments 11](#_Toc423685398)

[6.2. Demo single stepping / pause. 11](#_Toc423685399)

[Appendix 1 - Helper Class Overview 12](#_Toc423685400)

[A1.1. FslBase 12](#_Toc423685401)

[A1.2. FslGraphics 14](#_Toc423685402)

[A1.3. FslGraphicsGLES2 16](#_Toc423685403)

[A1.4. FslGraphicsGLES3 16](#_Toc423685404)

[A1.5. FslGraphicsVG 16](#_Toc423685405)

[Appendix 2 - Android SDK+NDK on windows build guide 17](#_Toc423685406)

[A2.1. Prerequisites: 17](#_Toc423685407)

[A2.2. Environment setup: 17](#_Toc423685408)

[A2.3. To Compile and run an existing sample application. 17](#_Toc423685409)

[A2.4. To create a new GLES2 demo project named 'CoolNewDemo' 18](#_Toc423685410)

[A2.5. Notes 19](#_Toc423685411)

[Appendix 3 - Ubuntu build guide 20](#_Toc423685412)

[A3.1. Prerequisites: 20](#_Toc423685413)

[A3.2. Environment setup: 20](#_Toc423685414)

[A3.3. To Compile all samples 20](#_Toc423685415)

[A3.4. To Compile and run an existing sample application. 20](#_Toc423685416)

[A3.5. To create a new GLES2 demo project named 'CoolNewDemo' 22](#_Toc423685417)

[A3.6. NOTES: 23](#_Toc423685418)

[Appendix 4 - Windows build guide 24](#_Toc423685419)

[A4.1. Prerequisites: 24](#_Toc423685420)

[A4.2. Environment setup: 24](#_Toc423685421)

[A4.3. Compiling and running an existing sample application 24](#_Toc423685422)

[A4.4. To create a new GLES2 demo project named 'CoolNewDemo' 25](#_Toc423685423)

[A4.5. Notes 26](#_Toc423685424)

[Appendix 5 - Yocto build guide 27](#_Toc423685425)

[A5.1. Prerequisites: 27](#_Toc423685426)

[A5.2. Yocto environment setup: 30](#_Toc423685427)

[A5.3. Demo framework environment setup: 30](#_Toc423685428)

[A5.4. To Compile all samples 30](#_Toc423685429)

[A5.5. To Compile and run an existing sample application. 30](#_Toc423685430)

[A5.6. To create a new GLES2 demo project named 'CoolNewDemo' 31](#_Toc423685431)

[A5.7. NOTES: 32](#_Toc423685432)

[Appendix 6 - FslContentSync.py notes 33](#_Toc423685433)

[Appendix 7 - Roadmap – Upcoming features 34](#_Toc423685434)

[A7.1. Technical overview 34](#_Toc423685435)

[Appendix 8 - Known limitations 35](#_Toc423685436)

[A8.1. General 35](#_Toc423685437)

[A8.2. Android 35](#_Toc423685438)

[A8.3. Ubuntu 35](#_Toc423685439)

[A8.4. Windows 35](#_Toc423685440)

[Appendix 9 - Upgrading samples from earlier SDKs 36](#_Toc423685441)

[A9.1. From 2.0 to 2.1 36](#_Toc423685442)

[A9.2. From 2.1 to 2.2 36](#_Toc423685443)

[Appendix 10 - What’s new 37](#_Toc423685444)

# Summaries

This document describes the Freescale Demo Framework, targeted at platform agnostic development of graphical demos. It covers the goals, architecture and instructions of how to use it across platforms, examples and best practices.

## Executive summary

* Write a demo application once.
* Run it on Android, Yocto linux, Ubuntu and MS Windows.
* Easily portable to additional platforms.
* Supports: OpenGL ES2, OpenGL ES3, OpenVG and experimental G2D support.

## Technical overview

* Written in a limited subset of C++11 and uses [RAII](http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization) to manage resources.
* Uses a limited subset of STL to make it easier to port.
* No copyleft restrictions from GPL / L-GPL licenses
* Allows for direct access to the expected API’s (EGL,ES2, ES3, VG)
* Provides optional helper classes for commonly used tasks
  + Matrix, Vector3, GLShader, GLTexture, etc
* Services
  + Keyboard & mouse
  + Persistent data manager
  + Assets management (models, textures)
* Defines a standard way for handling
  + Init, shutdown & window resize.
  + Program input arguments.
  + Input events like keyboard, mouse and touch.
  + Fixed time-step and variable time-step demo implementations.
  + Logging functionality.

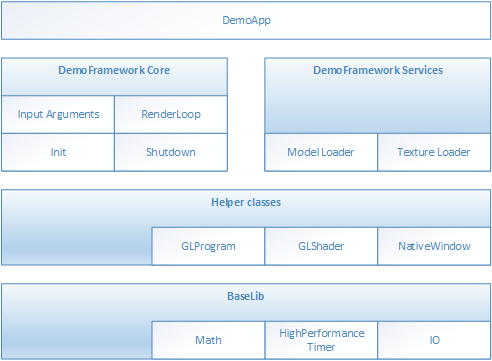
# Introduction

The Demo Framework is a multi-platform framework that enables demos to run on various platforms without any changes. The framework abstracts away all the boilerplate & OS specific code of allocating surfaces, creating the context, model loading, texture loading, shader compilation, render loop, animation ticks, benchmarking graph overlays etc. This allows the demo/benchmark developer to focus on writing rendering code. It also enables them to develop demos on PC or Android where the tool chain and debug facilities allows for faster turnaround time and then take the working code and deploy without code changes to the supported platforms. The platforms we currently support are Windows (for development via emulated backends), Android NDK and Linux with various windowing systems. The framework allows us to provide ‘real’ comparative benchmarks between the different OS and windowing systems we support, since we can run the exact same demo/benchmark code on them all.

The long term plans for the framework include extending it with support for other relevant API’s.

# Design overview

The framework is written in C++ and uses [RAII](http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization)[[1]](#footnote-1) to manage resources. The resource management code focuses on ‘ease of use’ over raw performance, since it’s mainly run on construction and destruction of the demo.

To allow the demo framework to be easily portable to new platforms its functionality is split into two parts: ‘core’ and ‘services’. The core framework depends on a limited subset of STL to make it easier to port. Framework services come with their own set of library requirements. The model importer [Assimp](http://assimp.sourceforge.net/)[[2]](#footnote-2) requires boost to be available on the platform.

Beside the demo framework core and demo framework services there is a set of helper classes for commonly used functionality, which makes it easier to write demo’s for the API’s we support. The helper classes do not depend on the demo framework and can be used in any program for the given API. For example for OpenGL ES, there is a GLShader and GLProgram class which hides away the complexities of compiling the shader object and linking the program object and since they are [RAII](http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization) objects, they also clean up after themselves once you are done with them.

Since our primarily supported BSPs are linux based, we decided to utilize an input argument framework that is compatible with the standard Unix parameter format, like the one exposed by getopt[[3]](#footnote-3).

# High level overview

The framework consist of three high level domains.

## DemoMain

Classes-HighLevelAll the code that binds everything together and it is platform independent.

1. It gets the current demo setup
   1. Which demo host to utilize for the demo.
   2. Which demo app that needs to be run.
2. It parses the input arguments
3. It launches the demo host.
4. It logs any errors that might occur.

## DemoHost

The demo-host is responsible for init & shutdown of the host environment and running the main loop.

The main loop utilizes the DemoAppManager to control the life of the DemoApp.

In other words, the DemoHost is the graphics API specific code needed to initialize and shutdown a given API and some code to run a render loop. All the API and platform independent code of the render loop resides inside the DemoAppManager class.

The exact capabilities of a DemoHost are also platform dependent. For example, some EGL implementations support running OpenVG and OpenGL ES, allowing a demo app to utilize both API’s at once. This is not something that is supported by most windows emulation layers.

## DemoApp

A demo application written for one or more specific APIs which are supported by a specific DemoHost. The demo is usually platform independent – the exception to the rule is if it depends on specific features that only exist on certain platforms.

# Demo application details

The following description of the demo application details uses a GLES2 demo named ‘S01\_SimpleTriangle’ as example. It lists the default methods that a demo should implement, the way it can provide customized parameters to the windowing system and how asset management is made platform agnostic.

## Demo method overview

This is a list of the methods that every Demo App is most likely to override[[4]](#footnote-4).

// Init

S01\_SimpleTriangle(const DemoAppConfig& config)

// Shutdown

~S01\_SimpleTriangle()

// OPTIONAL: Custom resize logic (if the app requested it). The default logic is to

// restart the app.

void Resized(const Point2& size)

// OPTIONAL: Fixed time step update method that will be called the set number of times

// per second. The fixed time step update is often used for physics.

void FixedUpdate(const DemoTime& demoTime)

// OPTIONAL: Variable time step update method.

void Update(const DemoTime& demoTime)

// Put the rendering calls here

void Draw()

When the constructor is invoked, the Demo Host API will already be setup and ready for use, the demo framework will use EGL to configure things as requested by your EGL config and API version.

It is recommended that you do all your setup in the constructor.

This also means that you should never try to shutdown EGL in the destructor since the framework will do it at the appropriate time. The destructor should only worry about resources that your demo app actually allocated by itself.

### Resized

The resized method will be called if the screen resolution changes (if your app never changes resolution this will never be called)[[5]](#footnote-5).

### FixedUpdate

Is a fixed time-step update method that will be called the set number of times per second. The fixed time step update is often used for physics[[6]](#footnote-6).

### Update

Will be called once before every draw call and you will normally update your animation using delta time.

For example if you need to move your object 10 units horizontally per second you would do something like

m\_positionX += 10 \* demoTime.DeltaTime;

### Draw

Should be used to render graphics.

## Fixed or variable timestep update

Depending on what your demo is doing, you might use one or the other - or both. It’s actually a very complex topic once you start to dig into it, but in general anything that need precision and predictable/repeatable calculations, like for example physics, often benefits from using fixed time steps. It really depends on your algorithm and it’s recommended to do a couple of google searches on fixed vs variable, since there are lots of arguments for both. It’s also worth noting that game engines like [Unity3D](http://unity3d.com/)[[7]](#footnote-7) support both methods.

## 

## Execution order of methods during a frame

The methods will be called in this order

* Events (if any occurred)[[8]](#footnote-8)
* Resized[[9]](#footnote-9)
* FixedUpdate (0-N calls. The first frame will always have a FixedUpdate call)
* Update
* Draw

After the draw call, a swap will occur.

## 

## Exit

The demo app can request an exit to occur, or it can be terminated via an external request.

In both cases one of the following things occur.

1. If the app has been constructed and has received a FixedUpdate, then it will finish its FixedUpdate, Update, Draw, swap sequence before its shutdown.
2. If the app requests a shutdown during construction, the app will be destroyed before calling any other method on the object (and no swap will occur).

The app can request an exit to occur by calling:

GetDemoAppControl()->RequestExit(1);

## Dealing with screen resolution changes

Per default the app is destroyed and recreated when a resolution change occurs[[10]](#footnote-10).

It is left up to the DemoApp to save and restore demo specific state.

## Content loading

The framework supports loading files from the Content folder on all platforms.

Given a content folder like this:

Content/Texture1.bmp

Content/Stuff/Readme.txt

You can load the files via the *IContentManager* service that can be accessed by calling

std::shared\_ptr<IContentManager> contentManager = GetContentManager();

You can then load files like this:

Binary file:

std::vector<uint8\_t> content;

contentManager->ReadAllBytes(content, "MyData.bin");

Text file:

const std::string content = contentManager->ReadAllText("Stuff/Readme.txt");

Bitmap file[[11]](#footnote-11):

Bitmap bitmap;

contentManager->Read(bitmap, "Texture1.bmp", PixelFormat::RGB888);

If you prefer to control the loading yourself you can retrieve the path to the files like this:

IO::Path contentPath = contentManager->GetContentPath();

IO::Path myData = IO::Path::Combine(contentPath, "MyData.bin");

IO::Path readmePath = IO::Path::Combine(contentPath, "Stuff/Readme.txt");

IO::Path texture1Path = IO::Path::Combine(contentPath, "Texture1.bmp");

You can then open the files with any method you prefer.

Both methods works for all supported platforms.

For detailed information about how the content is handled on each platform, see the build guide appendixes.

The details of the available helper classes for a Demo Application are described in Appendix 1.

## Demo registration

This is done in the S01\_SimpleTriangle\_Register.cpp file.

namespace

{

// Custom EGL config (Will overwrite the settings for the listed values.

// however an exact EGL config can be used)

static const *EGLint* g\_eglConfigAttribs[] =

{

*EGL\_RED\_SIZE*, 5,

*EGL\_GREEN\_SIZE*, 6,

*EGL\_BLUE\_SIZE*, 5,

*EGL\_ALPHA\_SIZE*, 0,

*EGL\_SAMPLES*, 0,

*EGL\_NONE*

};

}

// configure the demo environment to run this demo app in a OpenGLES2 host environment

FSL\_REGISTER\_OPENGLES2\_DEMO(S01\_SimpleTriangle,DemoAppHostConfigEGL(g\_eglConfigAttribs));

Since the demo framework is controlling the main method, you need to register your application with the Demo Host specific macro (in this case the OpenGL ES2 host), for the framework to register your demo class.

# Demo playback

## Command line arguments

All demos support various command line arguments.

|  |  |
| --- | --- |
| **Key** | **Function** |
| -h | Show the command line argument help. |
| --Stats | Show a performance graph. |
| --LogStats | Log various stats to the console. |

Use –h on a demo for a complete list

## Demo single stepping / pause.

Under windows all samples support time stepping which can be useful for debugging. It might also be available on under platforms that support the given keys.

|  |  |
| --- | --- |
| **Key** | **Function** |
| Pause | Pause the sample. |
| PageDown | Move forward one timestep. |
| Delete | Toggle between normal and Slow 2x playback |
| End | Toggle between normal and Slow 4x playback |
| Insert | Toggle between normal and fast 2x playback. |
| Home | Toggle between normal and fast 4x playback. |

1. Helper Class Overview
   1. FslBase

Provides basic functionality missing from C++ standard libraries.

* + 1. Bits

|  |  |
| --- | --- |
| BitsUtil | Utility methods for working with bits |
| ByteArrayUtil | Utility methods for reading and writing values from byte arrays in a specific endian format. This functionality is useful when working on platform independent load and save methods. |

* + 1. IO

Platform independent IO.

|  |  |
| --- | --- |
| Directory | Helper methods for working on directories.   * GetCurrentWorkingDirectory. |
| File | Helper methods for working with files   * Checking if file exists. * File length. * Read all content from a file. |
| Path | A UTF8 path class and helper methods for working on it.   * Combing paths. * Extracting directory or filename. * Getting the full path from a relative path. |

* + 1. Log

Platform independent logging.

Instead of using printf or std::cout to log information it’s better to utilize the provided logging macro’s since work across all supported platforms.

|  |  |
| --- | --- |
| Log | Various logging macros   * FSLLOG * FSLLOG\_IF * FSLLOG\_WARNING * FSLLOG\_WARNING\_IF * FSLLOG\_ERROR * FSLLOG\_ERROR\_IF |

* + 1. Math

Mainly focused on math functionality useful for working with graphics. It focuses on ease of use instead of raw performance.

|  |  |
| --- | --- |
| MathHelper | Various commonly used helper methods and constants like   * PI * Clamping * Lerp * Conversions between radians and angles * PowerOfTwo |
| Matrix | Matrix helper methods like   * Perspective * Rotate * Translate * Scale * Multiply |
| Point2 | A 2D integer point. |
| Rectangle | A integer based rectangle with helper methods like   * Union * Intersection |
| Vector2 | A 2d float point with helper methods like   * Dot * Length * Lerp * Min, max * Normalize * Reflect |
| Vector3 | A 3d float point with helper methods like   * Cross * Dot * Length * Lerp * Min, max * Normalize * Reflect * Transform by matrix |
| Vector4 | A 4d float point with helper methods like   * Dot * Length * Lerp * Min, max * Normalize * Reflect * Transform by matrix |

* + 1. String

Various string functionality

|  |  |
| --- | --- |
| StringParseUtil | Various utility method for converting a string to a number. |
| UTF8String | A UTF8 string representation. |

* + 1. System

|  |  |
| --- | --- |
| HighResolutionTimer | A platform independent high resolution timer. |

* 1. FslGraphics

|  |  |
| --- | --- |
| Bitmap | A RAII class to manage bitmap data. |
| BitmapUtil | Contains various helper methods that works on the bitmap class.   * Horizontal flip * Pixel format conversion |
| RawBitmap | Read only bitmap information. |
| RawBitmapEx | Writeable access to bitmap information |
| RawBitmapUtil | Low level helper methods that work on RawBitmap’s   * Horizontal flip * Padding clear * Swizzle |

### IO

|  |  |
| --- | --- |
| BMPUtil | A simple helper class for loading and saving BMP images.  It’s not recommended to utilize it directly. Instead utilize the framework for loading images[[12]](#footnote-12).  See Content loading for more details. |

* + 1. Vertices

API independent vertex helper classes.

|  |  |
| --- | --- |
| VertexDeclaration | Defines how a vertex is constructed in an API independent way. |
| VertexElementEx | Defines a vertex element |
| VertexPositionColor | A vertex comprised of   * position * color. |
| VertexPositionColorNormalTexture | A vertex comprised of   * position * color * normal * texture coordinates |
| VertexPositionColorTexture | A vertex comprised of   * position * color * texture coordinates |
| VertexPositionNormalTexture | A vertex comprised of   * position * normal * texture coordinates |
| VertexPositionTexture | A vertex comprised of   * position * texture coordinates |

* 1. FslGraphicsGLES2

RAII based helper classes for common GLES2 operations.

|  |  |
| --- | --- |
| GLCheck | Various helper macro’s for checking and transforming OpenGL ES errors to exception. |
| GLIndexBuffer | A RAII based index buffer.   * uint8\_t & uint16\_t based index buffers. * Easy creation and update. |
| GLProgram | A RAII based GL program encapsulation.   * Vertex and fragment shader combination. |
| GLShader | A RAII based GL shader encapsulation.   * Compilation and logging. |
| GLTexture | A RAII based GL texture encapsulation.   * Can be created from either FslGraphics RawBitmap’s or Bitmaps. * Easy content update. * Supports both normal and cubemap textures. |
| GLUtil | Contains various utility methods for OpenGL ES2   * Capture screenshots |
| GLVertexBuffer | A RAII based vertex buffer.   * Easy creation and updating from Custom or FslGraphics.Vertices. * Helper methods for quickly enabling/disabling Attribs |

* 1. FslGraphicsGLES3

RAII based helper classes for common GLES3 operations.

GLES3 has the exact same helper classes as GLES2 and the following additions:

|  |  |
| --- | --- |
| GLVertexArray | A RAII based vertex array.   * Easy creation |

* 1. FslGraphicsVG

RAII based helper classes for common OpenVG operations.

|  |  |
| --- | --- |
| VGPathBuffer | A RAII based path buffer   * Easy creation |
| VGUtil | Contains various utility methods for OpenVG   * Capture screenshots |
| VGCheck | Various helper macro’s for checking and transforming OpenVG errors to exception. |

1. Android SDK+NDK on windows build guide
   1. Prerequisites:

* [JDK (32 bit)](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)

**IMPORTANT:** Make sure to configure JAVA\_HOME to point to the JDK directory

* [Android SDK (32 bit)](https://developer.android.com/sdk/index.html?hl=i#download)

Once it’s installed it’s a good idea to run "SDK Manager.exe" and make sure everything is up to date.

**IMPORTANT:** Make sure to configure ANDROID\_HOME to point to the android sdk directory

**IMPORTANT:** Make sure that you have the Android 4.4.2 (API 19) SDK Platform installed.

* [Android NDK (32 bit)](https://developer.android.com/tools/sdk/ndk/index.html)

**IMPORTANT:** Make sure to configure ANDROID\_NDK to point to the android ndk directory

* [Ant](http://ant.apache.org/bindownload.cgi)

**IMPORTANT:** Make sure to configure ANT\_HOME to point to the ant directory

Extra info: <http://www.androidengineer.com/2010/06/using-ant-to-automate-building-android.html>

* Python 2.7.x
  + [For 32bit windows](https://www.python.org/ftp/python/2.7.8/python-2.7.8.msi)
  + [For 64bit windows](https://www.python.org/ftp/python/2.7.8/python-2.7.8.amd64.msi)
  1. Environment setup:

1. Start a windows console (cmd.exe) in the DemoFramework folder.
2. Run the 'prepare.bat' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.bat file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).
   1. To Compile and run an existing sample application.

In this example we will utilize the GLES2 S06\_Texturing app.

1. Make sure that you performed the environment setup.
2. Change directory to the sample directory:

cd DemoApps\GLES2\S06\_Texturing\Android

1. Build and install the app apk[[13]](#footnote-13)

ant debug install

* 1. To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the environment setup.
2. Change directory to the GLES2 sample directory:

cd DemoApps/GLES2

1. Create the project template using the FslNewDemoProject.py script

FslNewDemoProject.py all -t GLES2 CoolNewDemo

1. Change directory to the newly created project folder 'CoolNewDemo'

cd CoolNewDemo

1. Generate build files for Android, Ubuntu and Yocto (this step will be simplified soon)

FslBuildGen.py

When you add the generated build.sh to git on windows then please remember to set the executable bit using:

git update-index --chmod=+x build.sh

1. Change directory to the android folder 'CoolNewDemo'

cd Android

1. Build and install the app apk[[14]](#footnote-14)

ant debug install

If you add source files to a project or change the Fsl.gen file then run the FslBuildGen.py script in the project root folder to regenerate the various build files.

* 1. Notes
     1. Content

As long as you utilize one of the methods above to load the resources, you don’t really need to know the following. However if you experience problems it might be useful for you to know.

Under android builds we package all content using the Android 'assets' system. Since the system requires that the asset files are located under it's 'assets' folder (located at Android/assets in our samples) we utilize a one way folder synchronization utility called 'FslContentSync.py' to ensure that all files and directories under Content exist inside the asset folder as well. The synchronization script is automatically invoked during the android build process. To complicate things further the Android assets cannot normally be accessed via filenames using standard C/C++ methods. Because of this the assets are 'unpacked' on target to either the external or internal file system which allows us to open the files any way we like. Unfortunately this means that there will be a slight unpacking delay the first time a sample is executed.

* + 1. Command line app building via Ant

<http://developer.android.com/tools/building/building-cmdline.html>

1. Ubuntu build guide
   1. Prerequisites:

* Ubuntu14.04 64 bit
* Build tools and xrand

sudo apt-get install build-essential libxrandr-dev

* Python 2.7

It should be part of the default Ubuntu14.04 install.

* A OpenGL ES 2+ emulator
  + Mesa OpenGL ES 2

sudo apt-get install libgles2-mesa-dev

* + [Arm Mali OpenGL ES 3.0 Emulator V1.4.1 (64 bit)](http://malideveloper.arm.com/develop-for-mali/tools/opengl-es-3-0-emulator/)

wget http://malideveloper.arm.com/downloads/tools/emulator/1.4.1/Mali\_OpenGL\_ES\_Emulator-1.4.1-Linux-64bit.deb

sudo dpkg -i Mali\_OpenGL\_ES\_Emulator-1.4.1-Linux-64bit.deb

* OpenVG
  + Mesa OpenVG

sudo apt-get install libopenvg1-mesa-dev

* DevIL
  + Developer's Image Library (DevIL)

sudo apt-get install libdevil-dev

* 1. Environment setup:

1. Start a terminal (ctrl+alt t) in the DemoFramework folder
2. Run the 'prepare.sh' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.sh file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).

source prepare.sh

* 1. To Compile all samples

1. Make sure that you performed the environment setup
2. Compile everything (a good rule of thumb for '-j N' is number of cpu cores \* 2)

./build.sh -j 2

* 1. To Compile and run an existing sample application.

In this example we will utilize the GLES2 S06\_Texturing app.

1. Make sure that you performed the environment setup
2. Change directory to the sample directory:

cd DemoApps/GLES2/S06\_Texturing

1. Compile the project (a good rule of thumb for '-j N' is number of cpu cores \* 2)

./build.sh -j 2

* 1. To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the environment setup
2. Change directory to the GLES2 sample directory:

cd DemoApps/GLES2

1. Create the project template using the FslNewDemoProject.py script

FslNewDemoProject.py all -t GLES2 CoolNewDemo

1. Change directory to the newly created project folder 'CoolNewDemo'

cd CoolNewDemo

1. Generate build files for Android, Ubuntu and Yocto (this step will be simplified soon)

FslBuildGen.py

chmod u+x build.sh

1. Compile the project (a good rule of thumb for '-j N' is number of cpu cores \* 2)

./build.sh -j 2

If you add source files to a project or change the Fsl.gen file then run the FslBuildGen.py script in the project root folder to regenerate the various build files.

* 1. NOTES:
     1. Content

As long as you utilize one of the methods above to load the resources, you don’t really need to know the following. However if you experience problems it might be useful for you to know.

The ubuntu build expects the content folder to be located at "<executable directory>/content". Since the binary is put in the sample root directory where the content folder is located, there should be no problem loading the resources.

### 

* + 1. Manual environment setup:

1. Configure your FSL\_GRAPHICS\_SDK to point to the downloaded sdk without the ending backslash:

export FSL\_GRAPHICS\_SDK=~/fsl/YourDemoFrameworkFolder

1. For easy access to the python scripts (not required for building)

PATH=$PATH:$FSL\_GRAPHICS\_SDK/.Config

* + 1. Override platform auto-detection

To override the platform auto detection code set the following variable

export FSL\_PLATFORM\_NAME=Ubuntu

* + 1. Executable location

The final executable will be placed in the root of the demo application folder. If it is moved the content folder (if it exist) needs to be copied to the same location.

1. Windows build guide
   1. Prerequisites:

* Visual Studio 2013 (community edition or better)
* Python 2.7.x
  + [For 32bit windows](https://www.python.org/ftp/python/2.7.8/python-2.7.8.msi)
  + [For 64bit windows](https://www.python.org/ftp/python/2.7.8/python-2.7.8.amd64.msi)
* A OpenGL ES 2+ emulator
  + [Arm Mali OpenGL ES 3.0 Emulator V2.2 (32 bit)](http://malideveloper.arm.com/develop-for-mali/tools/opengl-es-3-0-emulator/)
  + Vivante OpenGL ES Emulator

To get started its recommended to utilize the Arm Mali OpenGL ES 3.0 emulator (32 bit) which this guide will assume you are using.

* 1. Environment setup:

1. Start a windows console (cmd.exe) in the DemoFramework folder
2. Run the 'prepare.bat' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.bat file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).

* 1. Compiling and running an existing sample application

In this example we will utilize the GLES2 S06\_Texturing app.

1. Make sure that you performed the environment setup
2. Change directory to the sample directory:

cd DemoApps\GLES2\S06\_Texturing

1. Launch visual studio using the Arm Mali Emulator:

.StartProject\_Arm.bat

1. Compile and run the project (The default is to press F5)

To utilize the vivante emulator use .StartProject\_Vivante.bat instead of .StartProject\_Arm.bat

* 1. To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the environment setup
2. Change directory to the GLES2 sample directory:

cd DemoApps/GLES2

1. Create the project template using the FslNewDemoProject.py script

FslNewDemoProject.py all -t GLES2 CoolNewDemo

1. Change directory to the newly created project folder 'CoolNewDemo'

cd CoolNewDemo

1. Generate build files for Android, Ubuntu and Yocto (this step will be simplified soon)

FslBuildGen.py

When you add the generated build.sh to git on windows then please remember to set the executable bit using:

git update-index --chmod=+x build.sh

1. Launch visual studio using the Arm Mali Emulator:

.StartProject\_Arm.bat

1. Compile and run the project (The default is to press F5) or start creating your new demo.

If you add source files to a project or change the Fsl.gen file then run the FslBuildGen.py script in the project root folder to regenerate the various build files.

* 1. Notes
     1. Content

As long as you utilize one of the methods above to load the resources, you don’t really need to know the following. However if you experience problems it might be useful for you to know.

The windows build expects the content folder to be located at "<current working directory>/content". When you launch the sample via the visual studio project the current working directory will be equal to the sample root directory where the content folder is located, so there should be no problem loading the resources.

* + 1. Switching between emulators

The visual studio projects have been configured so that emulator builds can co-exist without interfering with each other. Furthermore the only the emulator dependent parts will be rebuild when changing emulator.

So all in all it ought to be very fast to switch between emulators.

* + 1. Executable location

The executable location is based upon the build type release/debug and which emulator you are using and

So the executable for a demo called S06\_Texturing build as debug and using the arm emulator will be located under

bin\S06\_Texturing\Debug\_ARM\

The content folder is located at

Content

If you want to move them then make sure that both the S06\_Texturing.exe and Content folder is moved to the same location like this:

S06\_Texturing.exe

Content

1. Yocto build guide
   1. Prerequisites:

* Python 2.7

It should be part of the default Ubuntu14.04 install.

* A working yocto build

For example follow one of these:

* + <http://git.freescale.com/git/cgit.cgi/imx/fsl-arm-yocto-bsp.git/>
  + <https://community.freescale.com/docs/DOC-94866>

Before you build one of these yocto images you need to

1. Unpack the meta-gtec.tar.gz file found in the demoframework root directory to

<fsl\_yocto\_bsp>/sources/meta-gtec

Example:

mkdir ~/fsl-release-bsp/sources/meta-gtec

tar -xvzf meta-gtec.tar.gz -C ~/fsl-release-bsp/sources/meta-gtec

2. Run the yocto build setup (X11 example).

MACHINE=imx6qsabreauto source fsl-setup-release.sh -b build-x11 -e x11

3. Edit the <build directory>/conf/bblayers.conf file and add the line:

BBLAYERS += " ${BSPDIR}/sources/meta-gtec "

4. Edit the <build directory>/conf/local.conf file and add the line:

CORE\_IMAGE\_EXTRA\_INSTALL += "devil "

5. Beware that running the "MACHINE" command overwrites the changes done in step 2+3!!!

You can now build one of the images below (or a custom one)

|  |
| --- |
| x11 yocto image  Example:  <Perform step1>  MACHINE=imx6qsabreauto source fsl-setup-release.sh -b build-x11 -e x11  <Perform step3+4>  bitbake fsl-image-gui  bitbake meta-toolchain  bbitbake meta-ide-support  Extracted rootfs  We assume your yocto build dir is located at ~/fsl-release-bsp/build-x11 and that the rootfs will be unpacked to ~/unpacked-rootfs/build-x11 and the image is called fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 (you will need to locate your image name)  runqemu-extract-sdk ~/fsl-release-bsp/build-x11/tmp/deploy/images/imx6qsabresd/fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-x11 |
| FB yocto image  Example:  <Perform step1>  MACHINE=imx6qsabreauto source fsl-setup-release.sh -b build-fb -e fb  <Perform step3+4>  bitbake fsl-image-gui  bitbake meta-toolchain  bitbake meta-ide-support  Extracted rootfs  We assume your yocto build dir is located at ~/fsl-release-bsp/build-fb and that the rootfs will be unpacked to ~/unpacked-rootfs/build-fb and the image is called fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 (you will need to locate your image name)  runqemu-extract-sdk ~/fsl-release-bsp/build-fb/tmp/deploy/images/imx6qsabresd/fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-fb |

|  |
| --- |
| Wayland yocto image  Example:  <Perform step1>  MACHINE=imx6qsabreauto source fsl-setup-release.sh -b build-wayland -e wayland  <Perform step3+4>  bitbake fsl-image-gui  bitbake meta-toolchain  bitbake meta-ide-support  Extracted rootfs  We assume your yocto build dir is located at ~/fsl-release-bsp/build-wayland and that the rootfs will be unpacked to ~/unpacked-rootfs/build-wayland and the image is called fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 (you will need to locate your image name)  runqemu-extract-sdk ~/fsl-release-bsp/build-wayland/tmp/deploy/images/imx6qsabresd/fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-wayland |
| DirectFB yocto image  Example:  <Perform step1>  MACHINE=imx6qsabresd source fsl-setup-release.sh -b build-dfb -e dfb  <Perform step3+4>  bitbake fsl-image-gui  bitbake meta-toolchain  bitbake meta-ide-support  Extracted rootfs  We assume your yocto build dir is located at ~/fsl-release-bsp/build-dfb and that the rootfs will be unpacked to ~/unpacked-rootfs/build-dfb and the image is called fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 (you will need to locate your image name)  runqemu-extract-sdk ~/fsl-release-bsp/build-dfb/tmp/deploy/images/imx6qsabresd/fsl-image-gui-imx6qsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-dfb |

For this guide we will assume you are using an X11 image.

* 1. Yocto environment setup:

Prepare the yocto build environment

pushd ~/fsl-release-bsp/build-x11/tmp

source environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi

export ROOTFS=~/unpacked-rootfs/build-x11

popd

* 1. Demo framework environment setup:

1. Make sure that you performed the Yocto setup
2. cd to the demoframework folder
3. Run the 'prepare.sh' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.sh file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).

source prepare.sh

* 1. To Compile all samples

1. Make sure that you performed the demo framework environment setup
2. Compile everything (a good rule of thumb for '-j N' is number of cpu cores \* 2)

./build.sh -f GNUmakefile\_Yocto -j 2 EGLBackend=X11

EGLBackend can be set to either: DirectFB, FB, Wayland or X11

* 1. To Compile and run an existing sample application.

In this example we will utilize the GLES2 S06\_Texturing app.

1. Make sure that you performed the demo framework environment setup.
2. Change directory to the sample directory:

cd DemoApps/GLES2/S06\_Texturing

1. Compile the project (a good rule of thumb for '-j N' is number of cpu cores \* 2)

./build.sh -f GNUmakefile\_Yocto -j 2 EGLBackend=X11

EGLBackend can be set to either: DirectFB, FB, Wayland or X11

* 1. To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the demo framework environment setup.
2. Change directory to the GLES2 sample directory:

cd DemoApps/GLES2

1. Create the project template using the FslNewDemoProject.py script

FslNewDemoProject.py all -t GLES2 CoolNewDemo

1. Change directory to the newly created project folder 'CoolNewDemo'

cd CoolNewDemo

1. Generate build files for Android, Ubuntu and Yocto (this step will be simplified soon)

FslBuildGen.py

chmod u+x build.sh

1. Compile the project (a good rule of thumb for '-j N' is number of cpu cores \* 2)

./build.sh -f GNUmakefile\_Yocto -j 2 EGLBackend=X11

EGLBackend can be set to either: DirectFB, FB, Wayland or X11

If you add source files to a project or change the Fsl.gen file then run the FslBuildGen.py script in the project root folder to regenerate the various build files.

* 1. NOTES:

* + 1. Content

As long as you utilize one of the methods above to load the resources, you don’t really need to know the following. However if you experience problems it might be useful for you to know.

The Yocto build expects the content folder to be located at "<executable directory>/content".

* + 1. Manual environment setup:

Configure your FSL\_GRAPHICS\_SDK to point to the downloaded sdk without the ending backslash:

export FSL\_GRAPHICS\_SDK=~/fsl/YourDemoFrameworkFolder

1. For easy access to the python scripts

PATH=$PATH:$FSL\_GRAPHICS\_SDK/.Config

* + 1. Override platform auto-detection

To override the platform auto detection code set the following variable

export FSL\_PLATFORM\_NAME=Yocto

### 

* + 1. Building for multiple backends

The makefiles have been configured so that the builds for all backends can co-exist without interfering with each other. Furthermore the only the backend dependent parts will be rebuild when changing backend.

So all in all it ought to be very fast to switch between backends.

The demo app executables will be post fixed with the backend its build for to ensure no conflicts occurs.

* + 1. Executable location

The final executable will be placed in the root of the demo application folder. If it is moved the content folder (if it exist) needs to be copied to the same location.

The executables follows this naming scheme:

<DemoAppName>\_<BackendName>[<TargetPostFix>]

So a debug build of S06\_Texturing for the DirectFB backend will be called

S06\_Texturing\_DirectFB\_d

A release build of S06\_Texturing for the X11 backend will be called

S06\_Texturing\_X11

1. FslContentSync.py notes

* Does not copy files that start with a '.' in its file or directory name.
* Does not allow files to contain ".." in its name.
* Do **not** utilize file names that only differ by casing like this:
  + Shader.txt
  + shader.txt
* Due to the android asset packer it’s not recommended to use Unicode file names as they are unsupported by the android tool at the moment.

1. Roadmap – Upcoming features
   1. Technical overview

* Graphics API support
  + Vulkan
* Services
  + Model loader via Assimp

1. Known limitations
   1. General

* Android, Ubuntu and Windows OpenVG support is considered experimental for this release.
* G2D support is experimental and it’s not recommended to use it yet.
  1. Android
* Android does not handle Unicode file names inside the 'content' folder. So do not utilize Unicode for filenames stored in Content. The culprit is the android assets folder which we utilize for content files.
* .png and .jpgs content are not supported yet.
  1. Ubuntu
* OpenGLES3 is currently unsupported on Ubuntu, as we rely on the Mesa 3D graphics library for OpenGLES emulation.
* OpenVG is emulated via the Mesa 3D graphics library and it might contain unsupported features.
  1. Windows
* OpenVG is emulated via the Mesa 3D graphics library and it might contain unsupported features.

1. Upgrading samples from earlier SDKs

To convert a sample to the newest sdk start at the SDK version you are using and upgrade the app one step at a time. So a 2.0 app needs to be updated to 2.1 before it can be updated to 2.2.

* 1. From 2.0 to 2.1

Since version 2.1 contains minor incompatibilities with 2.0, any existing application will have to be upgraded. The easiest way to upgrade a sample is to rename the old directory, then run

* FslNewDemoProject.py all -t <type> <name>
* cd <name>
* FslBuildGen.py

Then do a two way merge of the old source directory and the new one. If any dependencies were manually added to Fsl.gen in the sample, they will have to be re-added to the new one.

Then run

* FslBuildGen.py

The project should now be converted.

* 1. From 2.1 to 2.2

V2.1 can easily be upgraded to 2.2, just run FslBuildGen.py to update it.

1. What’s new

**Version 2.2**

* Demo content can now be stored in bmp, png and jpeg format on all platforms.
  + Some platforms support extra formats via the DevIL image library.
* Onscreen performance graph support that can be augmented with custom data.
* Pause and single stepping during demo playback.
* Added infrastructure that allows samples to share a library. See DemoApps/Shared for example libraries.
* Lots of new samples.
  + The Blur, FractalShader, FurShellRendering and DirectMultiSamplingVideoYUV are functional but experimental.
* Experimental G2D support.
* Experimental NativeBatch2D support under 3D api’s. See the DFNativeBatch2D samples for an example of how it works.
* Experimental –mmdc parameter for Yocto builds. If it shows the wrong info then run mmdc2 before running the sample as it will reset things correctly.

**Version 2.1**

* OpenVG support.
* OpenVG examples
* Examples: T3DstressTest for GLES2 + GLES3
* Most samples were upgraded to use the Content system to load their shaders and graphics.
* All samples now support the following arguments
  + –LogStats = Log basic rendering stats
  + –ScreenshotFrequency <frequency> = Create a screenshot at the given frame frequency (Not supported for OpenVG).

1. http://en.wikipedia.org/wiki/Resource\_Acquisition\_Is\_Initialization [↑](#footnote-ref-1)
2. http://assimp.sourceforge.net/ [↑](#footnote-ref-2)
3. We do however not utilize getopt to remain GPL free across platforms. [↑](#footnote-ref-3)
4. See DemoFramework\FslDemoApp\include\FslDemoApp\ADemoApp.hpp for a complete list. [↑](#footnote-ref-4)
5. This version of the framework always restart the app, so this will never be called. [↑](#footnote-ref-5)
6. This version uses a fixed update frequency of 60 ticks per second. This will be configurable in the future. [↑](#footnote-ref-6)
7. http://unity3d.com/ [↑](#footnote-ref-7)
8. For an example of event handling see the “DemoApps\GLES2\InputEvents” sample. [↑](#footnote-ref-8)
9. In this version of the framework this is never called as the app will be recreated on screen size changes (future versions will allow demo apps to handle resize events if they so desire) [↑](#footnote-ref-9)
10. Future versions will allow demo apps to handle resize events if they so desire. [↑](#footnote-ref-10)
11. The current framework only png, bmp and jpeg images on all platforms. [↑](#footnote-ref-11)
12. A future version will also add saving to the ContentManager. [↑](#footnote-ref-12)
13. See the ant notes for more details [↑](#footnote-ref-13)
14. See the ant notes for more details [↑](#footnote-ref-14)