Demo framework overview

[1. Summaries 4](#_Toc515881884)

[1.1. Executive summary 4](#_Toc515881885)

[1.2. Technical overview 4](#_Toc515881886)

[2. Introduction 5](#_Toc515881887)

[3. Design overview 6](#_Toc515881888)

[4. High level overview 7](#_Toc515881889)

[4.1. DemoMain 7](#_Toc515881890)

[4.2. DemoHost 7](#_Toc515881891)

[4.3. DemoApp 7](#_Toc515881892)

[5. Demo application details 8](#_Toc515881893)

[5.1. Demo method overview 8](#_Toc515881894)

[5.2. Fixed or variable timestep update 9](#_Toc515881895)

[5.3. Execution order of methods during a frame 9](#_Toc515881896)

[5.4. Exit 9](#_Toc515881897)

[5.5. Dealing with screen resolution changes 10](#_Toc515881898)

[5.6. Content loading 10](#_Toc515881899)

[5.7. Demo registration 11](#_Toc515881900)

[6. Demo playback 12](#_Toc515881901)

[6.1. Command line arguments 12](#_Toc515881902)

[6.2. Demo single stepping / pause. 12](#_Toc515881903)

[Appendix 1 - Helper Class Overview 13](#_Toc515881904)

[A1.1. FslBase 13](#_Toc515881905)

[A1.2. FslGraphics 15](#_Toc515881906)

[A1.3. FslUtil.OpenGLES2 18](#_Toc515881907)

[A1.4. FslUtil.OpenGLES3 19](#_Toc515881908)

[A1.5. FslUtil.OpenGLES3v1 19](#_Toc515881909)

[A1.6. FslUtil.OpenVG 19](#_Toc515881910)

[A1.7. FslGraphics3D 19](#_Toc515881911)

[A1.8. FslAssimp 19](#_Toc515881912)

[A1.9. FslGraphics3D.SceneFormat 20](#_Toc515881913)

[A1.10. FslSimpleUI 20](#_Toc515881914)

[Appendix 2 - FslBuild scripts 21](#_Toc515881915)

[A2.1. FslBuildGen.py 21](#_Toc515881916)

[A2.2. FslBuild.py 21](#_Toc515881917)

[A2.3. Important notes 21](#_Toc515881918)

[A2.4. Build system per platform: 21](#_Toc515881919)

[Appendix 3 - Android SDK+NDK on windows build guide 22](#_Toc515881920)

[A3.1. Prerequisites: 22](#_Toc515881921)

[A3.2. Environment setup: 22](#_Toc515881922)

[A3.3. To Compile and run an existing sample application. 23](#_Toc515881923)

[A3.4. To create a new GLES2 demo project named 'CoolNewDemo' 24](#_Toc515881924)

[A3.5. Using android studio. 25](#_Toc515881925)

[A3.6. Linux notes: 25](#_Toc515881926)

[A3.7. Notes 26](#_Toc515881927)

[Appendix 4 - Ubuntu build guide 27](#_Toc515881928)

[A4.1. Prerequisites: 27](#_Toc515881929)

[A4.2. Environment setup: 27](#_Toc515881930)

[A4.3. To Compile all samples 28](#_Toc515881931)

[A4.4. To Compile and run an existing sample application. 28](#_Toc515881932)

[A4.5. To create a new GLES2 demo project named 'CoolNewDemo' 29](#_Toc515881933)

[A4.6. NOTES: 30](#_Toc515881934)

[Appendix 5 - Windows build guide 31](#_Toc515881935)

[A5.1. Prerequisites: 31](#_Toc515881936)

[A5.2. Environment setup: 31](#_Toc515881937)

[A5.3. Compiling and running an existing sample application 31](#_Toc515881938)

[A5.4. To create a new GLES2 demo project named 'CoolNewDemo' 32](#_Toc515881939)

[A5.5. Notes 33](#_Toc515881940)

[Appendix 6 - Yocto build guide 34](#_Toc515881941)

[A6.1. Building using a prebuild Yocto SDK 34](#_Toc515881942)

[A6.2. Building using a full Yocto build 36](#_Toc515881943)

[A6.3. Using the demo framework 39](#_Toc515881944)

[A6.4. To Compile all samples 40](#_Toc515881945)

[A6.5. To Compile and run an existing sample application. 40](#_Toc515881946)

[A6.6. To create a new GLES2 demo project named 'CoolNewDemo' 41](#_Toc515881947)

[A6.7. NOTES: 42](#_Toc515881948)

[Appendix 7 - FslContentSync.py notes 43](#_Toc515881949)

[Appendix 8 - Known limitations 44](#_Toc515881950)

[A8.1. General 44](#_Toc515881951)

[A8.2. Android 44](#_Toc515881952)

[A8.3. Ubuntu 44](#_Toc515881953)

[A8.4. Windows 44](#_Toc515881954)

[Appendix 9 - Upgrading samples from earlier SDKs 45](#_Toc515881955)

[A9.1. From 2.0 to 2.1 45](#_Toc515881956)

[A9.2. From 2.1 to 2.2 45](#_Toc515881957)

[A9.3. From 2.2 to 2.3 45](#_Toc515881958)

[Appendix 10 - What’s new 46](#_Toc515881959)

# 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.[[1]](#footnote-1)
* 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[[2]](#footnote-2)
* 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.

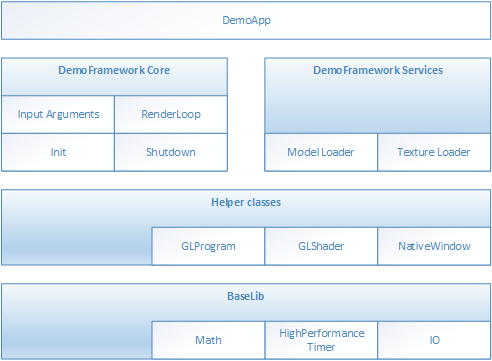
# Introduction

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)[[3]](#footnote-3) 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/)[[4]](#footnote-4) 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[[5]](#footnote-5).

# High level overview

The framework consists 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[[6]](#footnote-6).

// 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(const DemoTime& demoTime)

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)[[7]](#footnote-7).

### 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[[8]](#footnote-8).

### 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/)[[9]](#footnote-9) support both methods.

## 

## Execution order of methods during a frame

The methods will be called in this order

* Events (if any occurred)[[10]](#footnote-10)
* Resized[[11]](#footnote-11)
* 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[[12]](#footnote-12).

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[[13]](#footnote-13):

Bitmap bitmap;

contentManager->Read(bitmap, "Texture1.bmp", PixelFormat::R8G8B8\_UINT);

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 work 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 Fsl

{

namespace

{

// Custom EGL config (these will per default overwrite the custom settings.

// However, an exact EGL config can be used)

static const EGLint g\_eglConfigAttribs[] =

{

EGL\_SAMPLES, 0,

EGL\_RED\_SIZE, 8,

EGL\_GREEN\_SIZE, 8,

EGL\_BLUE\_SIZE, 8,

EGL\_ALPHA\_SIZE, 0, // buffers with the smallest alpha component size are preferred

EGL\_DEPTH\_SIZE, 24,

EGL\_SURFACE\_TYPE, EGL\_WINDOW\_BIT,

EGL\_NONE,

};

}

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

void ConfigureDemoAppEnvironment(HostDemoAppSetup& rSetup)

{

DemoAppHostConfigEGL config(g\_eglConfigAttribs);

DemoAppRegister::GLES2::Register<S01\_SimpleTriangle>(rSetup, "GLES2.S01\_SimpleTriangle", config);

}

}

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

### OpenGLES 3.X registration

To register a demo for OpenGLES 3.X you would use the GLES3 register method:

DemoAppRegister::GLES3::Register<S01\_SimpleTriangle>(rSetup, "GLES3.S01\_SimpleTriangle", config);

# 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. |
| --ScreenshotFrequency | Create a screenshot at the given frame frequency. |
| --ExitAfterFrame | Exit after the given number of frames has been rendered |
| --ContentMonitor | Monitor the Content directory for changes and restart the app on changes. WARNING: Might not work on all platforms and it might impact app performance (experimental) |

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 |
| Quaternion | Basic Quaternion operations. |

* + 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 |
| Color | RGBA color utility class. |
| PixelFormat | Various standardized pixel formats supported by the bitmap classes. |
| 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 |

### Font

|  |  |
| --- | --- |
| BasicFontKerning | Contains basic kerning information for a font. |
| BinaryFontBasicKerningLoader | Load basic kerning information from “fbk” files. |
| FontDesc | A very basic font description. |
| FontGlyphBasicKerning | Basic kerning for one glyph. |
| FontGlyphPosition | Position information for one glyph |
| FontGlyphRange | Font glyph range information. |
| IFontBasicKerning | Interface for extracting basic font kerning information. |
| TextureAtlasBitmapFont | Describes a bitmap font stored in a texture atlas. |
| TextureAtlasGlyphInfo | Texture atlas glyph information. |

### 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[[14]](#footnote-14).  See Content loading for more details. |

### Render

|  |  |
| --- | --- |
| AtlasFont | An atlas based bitmap font using a API independent texture. |
| AtlasTexture2D | An atlas based API independent texture. |
| BlendState | API independent blend states. |
| GenericBatch2D | A API independent 2D quad batcher. |
| Texture2D | A API independent texture representation. |

### TextureAtlas

|  |  |
| --- | --- |
| AtlasTextureInfo | Represents information about one texture that is stored in a texture atlas. |
| BasicTextureAtlas | A simple manager for looking up AtlasTextureInfo. |
| BinaryTextureAtlasLoader | A “BTA” basic texture atlas loader. |
| ITextureAtlas | Simple interface for accessing texture information. |
| NamedAtlasTexture | A named atlas texture. |
| TextureAtlasHelper | A simple way to extract AtlasTextureInfo from a texture atlas. |
| TextureAtlasMap | A more performance efficient way to extract AtlasTextureInfo from a texture atlas. |

* + 1. Vertices

API independent vertex helper classes.

|  |  |
| --- | --- |
| IndexConverter | Simple utility class to convert between index formats. It might not be efficient but it gets the job done. |
| VertexConverter | Simple utility class to convert between vertex formats. It might not be efficient but it gets the job done. |
| 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 |

### Window

|  |  |
| --- | --- |
| INativeWindow | An abstract from native windows. |

* 1. FslUtil.OpenGLES2

RAII based helper classes for common GLES2 operations.

|  |  |
| --- | --- |
| GLBatch2D | A specialization of GenericBatch2D GLES2. |
| GLBatch2DQuadRenderer | The GenericBatch2D backend for rendering quads. |
| GLCheck | Various helper macro’s for checking and transforming OpenGL ES errors to exception. |
| GLFrameBuffer | A RAII based frame buffer encapsulation. |
| GLIndexBuffer | A RAII based index buffer.   * uint8\_t & uint16\_t based index buffers. * Easy creation and update. |
| GlIndexBufferArray | A RAII based index buffer array.   * Improved efficiency when allocating many index buffers of the same format. |
| GLProgram | A RAII based GL program encapsulation.   * Vertex and fragment shader combination. |
| GLRenderBuffer | A RAII based GL render buffer encapsulation. |
| 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 |
| GLVertexBufferArray | A RAII based vertex buffer array.   * Improved efficiency when allocating many vertex buffers of the same vertex format. |
| NativeBatch2D | Extends GenericBatch2D with direct support for GLES2 native textures. |
| NativeTexture2D | Implements the INativeTexture2D for GLES2. This is used by the Batch2D system. |

* 1. FslUtil.OpenGLES3

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. FslUtil.OpenGLES3v1

RAII based helper classes for common GLES3.1 operation’s.

|  |  |
| --- | --- |
| GLProgramPipeline | A RAII based program pipeline encapsulation. |
| GLShaderProgram | A RAII based shader program encapsulation. |

* 1. FslUtil.OpenVG

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. FslGraphics3D

API independent descriptions of common 3D classes. This library is in development.

See the ModelLoaderBasics and ModelViewer samples for examples of how to use it.

|  |  |
| --- | --- |
| Mesh | A basic mesh |
| Scene | A basic scene |
| SceneNode | A basic node in the scene |

* 1. FslAssimp

The demo framework’s Assimp integration. Provides various helper classes that make it easier to work with assimp in the framework.

|  |  |
| --- | --- |
| MeshHelper | Helps to extract information from some assimp structures. |
| MeshImporter | Helps convert Assimp mesh structures to the FslGraphics3D ones. |
| SceneHelper | Extract basic information from a assimp scene. |
| SceneImporter | Helps convert Assimp scene structures to the FslGraphics3D ones. |

* 1. FslGraphics3D.SceneFormat

Code to load and save a very basic portable scene format.

|  |  |
| --- | --- |
| BasicSceneFormat | Load/save scene functionality. |

* 1. FslSimpleUI

A new experimental UI framework that makes it easy to get a basic UI up and running. The main code is API independent. It is not a show case of how to render a UI fast but only intended to allow you to quickly get a UI ready that is good enough for a demo.

You can look at:

* DFSimpleUI100
* DFSimpleUI101
* TessellationSample

To see how it’s used.

The next release of the framework should make it even easier to work with.

When working with the UI system its recommended to store all or at least the most used bitmaps in the same texture atlas. One commercially available texture packer is [Texture Packer](https://www.codeandweb.com/texturepacker) which can output a json file that we can convert to a binary format that can be loaded by the demo framework.

If you look at the DFSimpleUI100 sample, there is “OriginalContent/TextureAtlas” directory which contain a “MainAtlas.tps” file that can be loaded into texture packer. Pressing publish in texture packer produces a “MainAtlas.png” and “MainAtlas.json” file based on the files under “Main”. The “MainAtlas.png” can be copied directly to the samples “Content” directory but the json file needs to be converted to a binary file. For this we included the TPConvert python script that can be run like this:

TPConvert MainAtlas.json -f bta1

This will then produce a “MainAtlas.bta” file that can be copied to the “Content“ directory which contains all the needed atlas meta data.

Please beware that the default atlas is required to contain the default font as well. The documentation for creating the “MainAtlas.fbk” file has not been completed yet. The fbk file contains some basic font kerning information.

1. FslBuild scripts

For an easy to read text version of this document look in the demo framework “Doc/FslBuild\_toolchain\_readme.md” it contains more detailed information.

* 1. FslBuildGen.py

Is a cross-platform build-file generator. Which main purpose is to keep all build files consistent, in sync and up to date. See **FslBuildGen.docx** for details.

* 1. FslBuild.py

Extends the technology behind FslBuildGen with additional knowledge about how to execute the build system for a given platform.

So basically, FslBuild works like this

1. Invoke the build-file generator that updates all build files if necessary.
2. Filter the builds request based on the provided feature list.
3. Build all necessary build files in the correct order.
   * 1. Useful arguments

FslBuild comes with a few useful arguments

|  |  |
| --- | --- |
| --ListFeatures | List all features required by the build |
| --UseFeatures | Allows you to limit what’s build based on a provided feature list. For example [EGL,OpenGLES2]. This parameter defaults to all features. |
| -t 'sdk' | Build all demo framework projects |
| -v | Set verbosity level |
| -- | All arguments written after this is send directly to the native build system. |

* 1. Important notes
* Don’t modify the auto-generated files.

The FslBuild scripts are responsible for creating all the build files for a platform and verifying dependencies. Since all build files are auto generated you can never modify them directly as the next build will overwrite your changes.

Instead add your changes to the ‘Fsl.gen’ files as they control the build file generation!

* The ‘Fsl.gen’ file is the real build file.
* All include and source files in the respective folders are automatically added to the build files.
  1. Build system per platform:

|  |  |
| --- | --- |
| Android | gradle |
| Qnx | Make |
| Ubuntu | Make |
| Windows | Visual studio (IDE or nmake) |
| Yocto | make |

1. Android SDK+NDK on windows build guide

For an easy to read text version of this document look in the demo framework “Doc/Setup\_guide\_android.md” it contains more detailed information.

* 1. Prerequisites:
* Read Appendix 2 - so you know about the custom build system
* **IMPORTANT:** The way Gradle currently handles CMake builds on windows place some serious limits on the path length, so its recommended to either place the DemoFramework folder close to the root of the drive or to set the environment variable FSL\_GRAPHICS\_SDK\_ANDROID\_PROJECT\_DIR to a directory close to the root of the drive.
* [JDK (64 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](https://developer.android.com/studio/index.html)

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

**IMPORTANT:** Android studio must be at least 3.1

**IMPORTANT:** Get the android studio full package and enable the default packages.

Configure the SDK manager

* + - "SDK Platforms" add if necessary
      * Android 7.0 (Nougat)
    - "SDK Tools" add if necessary
      * CMake, LLDB, NDK, Android Support Repository

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

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

**IMPORTANT:** Make sure you have at least android-ndk-r16b

* Python 3.4.x or better. We highly recommend at least 3.5+
  + [For 64bit windows]((https:/www.python.org/ftp/python/3.6.2/python-3.6.2-amd64.exe)
  1. Environment setup:

Android projects are generated to the path specified in the environment variable FSL\_GRAPHICS\_SDK\_ANDROID\_PROJECT\_DIR. If it's not defined the 'prepare' script sets it to a default location.

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

1. Build a app for Android using gradle + cmake

FslBuild.py -p android

If you just want to regenerate the cmake build files then you can just run

FslBuildGen.py -p android

If you want to save a bit of compilation time you can build for the ANDROID ABI you need by adding

FslBuildGen.py --Variants [ANDROID\_ABI=armeabi-v7a]

or

FslBuild.py --Variants [ANDROID\_ABI=armeabi-v7a]

* 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 FslBuildNew.py script

FslBuildNew.py GLES2 CoolNewDemo

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

cd CoolNewDemo

1. Build a app for Android using gradle + cmake

FslBuild.py -p android

If you just want to regenerate the cmake build files then you can just run

FslBuildGen.py -p android

If you want to save a bit of compilation time you can build for the ANDROID ABI you need by adding

FslBuildGen.py --Variants [ANDROID\_ABI=armeabi-v7a]

or

FslBuild.py --Variants [ANDROID\_ABI=armeabi-v7a]

* 1. Using android studio.

1. Follow the instructions for "creating a new project" or "building an existing project".
2. As projects are generated to the path specified by the FSL\_GRAPHICS\_SDK\_ANDROID\_PROJECT\_DIR environment variable you can locate the project there and open it with android studio. Be sure to open Android studio in a correctly configured environment. Here it could be a good idea to create a script for launching android studio with the right environment.
   1. Linux notes:

* Install for private user and unzip android studio like this:

sudo unzip android-studio-ide\_FILENAME.zip -d ~/sdk

cd ~/sdk/android-studio/bin

./studio.sh

* In the ui make sure to install the sdk in a directory you have access to for example

~/sdk/android-sdk-linux

* 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

For an easy to read text version of this document look in the demo framework “Doc/Setup\_guide\_ubuntu16.04.md” it contains more detailed information.

* 1. Prerequisites:
* Read Appendix 2 - so you know about the custom build system
* Ubuntu16.04 64 bit
* Build tools and xrand

sudo apt-get install build-essential libxrandr-dev

* Python 3.4+

It should be part of the default Ubuntu16.04 install.

* An OpenGL ES 2+ emulator
  + Mesa OpenGL ES 2

sudo apt-get install libgles2-mesa-dev

* + [Arm Mali OpenGL ES 3.0 Emulator V3.0.2 (64 bit)](https://developer.arm.com/products/software-development-tools/graphics-development-tools/opengl-es-emulator/downloads)

wget https://armkeil.blob.core.windows.net/developer/Files/downloads/open-gl-es-emulator/3.0.2/Mali\_OpenGL\_ES\_Emulator-v3.0.2.g694a9-Linux-64bit.deb

sudo dpkg -i Mali\_OpenGL\_ES\_Emulator-v3.0.2.g694a9-Linux-64bit.deb

* DevIL
  + Developer's Image Library (DevIL)

sudo apt-get install libdevil-dev

* Assimp

Is now downloaded and build from source when needed. So its no longer necessary to run "sudo apt-get install libassimp-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

FslBuild.py -t sdk --BuildThreads 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 '--BuildThreads N' is number of cpu cores \* 2)

If you FslBuild without the --BuildThreads argument it will be set to 'auto' which uses your cpu core count.

FslBuild.py --BuildThreads 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 FslBuildNew.py.py script

FslBuildNew.py GLES2 CoolNewDemo

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

cd CoolNewDemo

1. Compile the project

FslBuild.py

Note:

Once a build has been done once you can just invoke the make file directly. However, this requires that you didn't change any dependencies or add files.

To do this run

make -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 or just make sure you always use the FslBuild.py script as it automatically adds files and regenerate build files as needed.

* 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

For an easy to read text version of this document look in the demo framework “Doc\Setup\_guide\_windows.md” it contains more detailed information.

* 1. Prerequisites:
* Read Appendix 2 - so you know about the custom build system
* Visual Studio 2017 (community edition or better)
* Python 3.5.x or newer
  + [For 64bit windows]((https:/www.python.org/ftp/python/3.6.2/python-3.6.2-amd64.exe)
* An OpenGL ES 2+ emulator
  + [Arm Mali OpenGL ES Emulator 3.0.2.g694a9 (64 bit)](https://developer.arm.com/products/software-development-tools/graphics-development-tools/opengl-es-emulator/downloads)
    - Please use the exact version (64bit) and use the installer to install it to the default location!
  + Vivante OpenGL ES Emulator

To get started its recommended to utilize the Arm Mali OpenGL ES 3.0.2 emulator (64 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 visual studio ```vcvarsall.bat x64``` to prepare your command line compiler environment for x64 compilation.

- For VS2017 its often located here:

"C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\vcvarsall.bat" x64

1. 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. Generate the build files

FslBuildGen.py

1. Launch visual studio using the Arm Mali Emulator:

.StartProject.bat arm

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

To utilize a different emulator the .StartProject.bat file can be launched with the following arguments

|  |  |
| --- | --- |
| arm | the arm mali emulator |
| powervr | the powervr emulator |
| qualcomm | the qualcomm andreno adreno emulator (expects its installed in "c:\AdrenoSDK”) |
| vivante | the vivante emulator |

If it is launched without an argument it defaults to the arm emulator.

* 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 FslBuildNew.py script

FslBuildNew.py 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

1. Launch visual studio using the Arm Mali Emulator:

.StartProject.bat arm

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

First you need to decide how you are going to be building for Yocto.

* Building using a prebuild Yocto SDK
* Building using a full Yocto build

For an easy to read text version of this document look in the demo framework “Doc/Setup\_guide\_yocto.md” it also more detailed information.

* 1. Building using a prebuild Yocto SDK

Building using a prebuild Yocto SDK and a prebuild sd-card image.

This tend to be the fastest way to get started.

* + 1. Prerequisites
* Read Appendix2 so you know about the custom build system.
* Ubuntu 16.04
* Python 3.5 (this is standard in Ubuntu 16.04)
* A prebuild sdk for your board typically called something like ‘toolchain.sh’
* A prebuild sd-card image for your board typically called ‘BoardName.rootfs.sdcard.bz2’
* Git: sudo apt-get install git

For this guide we will assume you are using a FB image.

* Download the DemoFramework source using git.

It's also a good idea to read the introduction to the FslBuild toolchain in “Doc/FslBuild\_toolchain\_readme.md”

* + 1. Preparing a Yocto SDK build

1. Start a terminal (ctrl+alt t)

2. Install the sdk:

./fsl-imx-internal-xwayland-glibc-x86\_64-fsl-image-gui-aarch64-toolchain-4.9.51-mx8-beta.sh

Chose where to install it, you can use the default location or a location of your choice.

For this example, we use "~/sdk/4.9.51-mx8-beta".

When the setup is complete it will list the configuration script you need to run to configure the sdk environment.

Something like this

$ . ~/sdk/4.9.51-mx8-beta/environment-setup-aarch64-poky-linux

Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.

3. Your SDK is now installed.

* + 1. Yocto SDK environment setup

1. Start a terminal (ctrl+alt t)

2. Prepare the yocto build environment by running the config command you got during the sdk install

. ~/sdk/4.9.51-mx8-beta/environment-setup-aarch64-poky-linux

3. You should now be ready to build using the demo framework. However, if you experience issues with the ‘prepare.sh’ script you can help it out by defining the platform name and the location of the root fs

export FSL\_PLATFORM\_NAME=Yocto

export ROOTFS=~/sdk/4.9.51-mx8-beta/sysroots/aarch64-poky-linux

Another possible error you can encounter is that the FslBuild.py scripts fail to include the 'typing' library.

This can happen because the SDK comes with a too old Python3 version or a incomplete Python3.5 version.

As a workaround for that you could delete the Python3 binaries from the SDK which will cause it to use the system Python3 version instead.

* + 1. Ready to build

You are now ready to start building Yocto apps using the demo framework.

Please continue the guide at “Using the demo framework”.

* 1. Building using a full Yocto build

Building using a full manually build Yocto build.

This process provides the most flexible solution but it also takes significantly longer to build the initial Yocto sdcard and toolchain.

* + 1. Prerequisites:
* Read Appendix 2 - so you know about the custom build system.
* The Ubuntu version required by the BSP release.
* Python 3.4 or newer

It should be part of the default Ubuntu install.

If you use 3.4 you need to install the 'typing' library manually so we highly recommended using 3.5 or newer.

To install the typing library in Python \*\*3.4\*\* run:

sudo apt-get install python3-pip

sudo pip3 install typing

* 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>

For this guide we will assume you are using a FB image.

* Download the DemoFramework source using git.
* It's also a good idea to read the introduction to the FslBuild toolchain at Doc/FslBuild\_toolchain\_readme.md
  + 1. Preparing a Yocto build

Before you build one of these yocto images you need to

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

MACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-x11 -e x11

3. Bake

bitbake fsl-image-gui

bitbake meta-toolchain

bitbake meta-ide-support

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

|  |
| --- |
| x11 yocto image  Example:  MACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-x11 -e x11  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-imx6qpsabresd.rootfs.tar.bz2` (you will need to locate your image name)  runqemu-extract-sdk ~/fsl-release-bsp/build-x11/tmp/deploy/images/imx6qpsabresd/fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-x11 |
| FB yocto image  Example:  ACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-fb -e fb  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-imx6qpsabresd.rootfs.tar.bz2` (you will need to locate your image name)  runqemu-extract-sdk ~/fsl-release-bsp/build-fb/tmp/deploy/images/imx6qpsabresd/fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-fb |

|  |
| --- |
| Wayland yocto image  Example:  MACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-wayland -e wayland  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-imx6qpsabresd.rootfs.tar.bz2` (you will need to locate your image name)  runqemu-extract-sdk ~/fsl-release-bsp/build-wayland/tmp/deploy/images/imx6qpsabresd/fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-wayland |

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

* + 1. Yocto environment setup:

Prepare the yocto build environment

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

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

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

export FSL\_PLATFORM\_NAME=Yocto

popd

* + 1. Ready to build

You are now ready to start building Yocto apps using the demo framework.

Please continue the guide at “Using the demo framework”.

* 1. Using the demo framework

1. Make sure that you performed the Yocto environment setup for your chosen Yocto environment.

- SDK build [Yocto SDK environment setup]

- Custom build [Yocto environment setup].cd to the demoframework folder

1. cd to 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

also verify that the script detect that you are doing a Yocto build by outputting

PlatformName: Yocto

If it doesn't you can override the platform auto detection by setting the environment variable

export FSL\_PLATFORM\_NAME=Yocto

Before running the prepare.sh script.

* 1. To Compile all samples

1. Make sure that you performed the demo framework environment setup
2. Compile everything

FslBuild.py --Variants [WindowSystem=FB] -t sdk

WindowSystem can be set to either: 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

FslBuild.py --Variants [WindowSystem=FB]

WindowSystem can be set to either: 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 FslBuildNew.py script

FslBuildNew.py GLES2 CoolNewDemo

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

cd CoolNewDemo

1. Compile the project

FslBuild.py --Variants [WindowSystem=FB]

WindowSystem can be set to either: FB, Wayland or X11

Note:

Once a build has been done once you can just invoke the make file directly. However, this requires that you didn't change any dependencies or add files. To do this run

make -f GNUmakefile\_Yocto -j 2 WindowSystem=FB

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 or just make sure you always use the FslBuild.py script as it automatically adds files and regenerate build files as needed.

* 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 FB backend will be called

S06\_Texturing\_FB\_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. 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.
  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. From 2.2 to 2.3

V2.2 can easily be upgraded to 2.3, just run FslBuildGen.py to update it.

1. What’s new

**Version 5.1**

* All ThirdParty code is now downloaded as needed instead of being included in the repo.
* Windows builds now default to Visual Studio 2017 instead of 2015.
* Basic support for changing the color-space via EGL.
* Examples of how to setup SRGB and HDR framebuffers.
* HDR to LDR display rendering examples with various basic tone-mapping algorithms.
* Vulkan enabled for the Yocto Wayland backend.
* Assimp upgraded to 4.1 on most platforms.
* GLES3.ColorspaceInfo
* GLES3.EquirectangularToCubemap
* GLES3.GammaCorrection demo.
* GLES3.HDR01\_BasicToneMapping
* GLES3.HDR02\_FBBasicToneMapping
* GLES3.HDR03\_SkyboxTonemapping
* GLES3.HDR04\_HDRFramebuffer
* GLES3.MultipleViewportsFractalShader demo.
* GLES3.Scissor101
* GLES3.Skybox
* GLES3.SRGBFramebuffer
* GLES3.TextureCompression demo.
* Vulkan.VulkanInfo demo.
* Android build now requires Android Studio 3.1 and the Android NDK16b or newer.

**Version 5.0.1**

* OpenVX.SoftISP demo.
* OpenCL.SoftISP demo.

**Version 5.0**

* Tools now require Python 3.4+ instead of python 2.7
* FslBuildNew script that can help you create a new project fast.
* Vulkan support is much closer to its final state.
* The application registration method has been changed so it’s more future proof and allow for greater customization.
* Prebuild binaries have been removed.
  + FslImageConvert.exe was removed as we now support saving screenshots directly in jpg.
  + Prebuild windows libraries removed as we now download and build them on demand instead.
* The directory structure was updated to make it simpler.
* Some tags in Fsl.gen xml files were deprecated.
* Gamepad support.
* New libraries
  + Stb, xinput, perfcounters.

**Version 4.0**

* First public release on github.
* Early access support for Vulkan, OpenCL, OpenCV and OpenVX.
  + Vulkan samples.
  + OpenVX samples.
  + OpenCL samples.
  + OpenCV samples.
* New libraries
  + GLI 0.8.10, GLM 0.9.7.6
* PixelFormats are now compatible with the vulkan pixel formats.
* FslBuild.py script introduced as a simple unified way to build on all platforms if so desired. It’s still possible to build using the native platform method.
* FslBuild scripts now support limited feature based filtering.
* Introduced a content pipeline to help build vulkan shaders.
* Windows builds
  + Visual Studio 2015 is now the default environment instead of 2013
  + We now use the OpenVG reference implementation to emulate OpenVG.

**Version 2.3**

* OpenGLES 3.1 support.
* A new ContentMonitor can reload your sample when it detects changes to the content folder (this does not work on Android). This allows for rapid prototyping on most platforms.
* New samples:
  + DFSimpleUI101, ModelLoaderBasics, ModelLoaderViewer, Tessellation101, TessellationSample.
* New libraries:
  + FslAssimp, FslGraphics3D, FslSceneFormat, FslSimpleUI, FslGraphicsGLES3v1
* New experimental UI framework intended to quickly create a UI for your sample app.
* Assimp support on most platforms. It is not supported on Android here we recommend using the FslSceneFormat instead. In general, it will be much more efficient to preprocess your model on a fast platform like a PC and save it in the FslSceneFormat instead of doing it on relatively slow target platform.
* Experimental support for generating Visual Studio 2015 projects (see the FslBuildgen documentation for details).
* Content loader for Binary texture and basic font kerning information.
* Windows PowerVR OpenGLES emulation support.

**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. “Yocto linux” or Yocto in the demo framework section means the BSP release. [↑](#footnote-ref-1)
2. We don’t use GPL or LGPL. [↑](#footnote-ref-2)
3. http://en.wikipedia.org/wiki/Resource\_Acquisition\_Is\_Initialization [↑](#footnote-ref-3)
4. http://assimp.sourceforge.net/ [↑](#footnote-ref-4)
5. We do however not utilize getopt to remain GPL free across platforms. [↑](#footnote-ref-5)
6. See DemoFramework\FslDemoApp\include\FslDemoApp\ADemoApp.hpp for a complete list. [↑](#footnote-ref-6)
7. This version of the framework always restart the app, so this will never be called. [↑](#footnote-ref-7)
8. This version uses a fixed update frequency of 60 ticks per second. This will be configurable in the future. [↑](#footnote-ref-8)
9. http://unity3d.com/ [↑](#footnote-ref-9)
10. For an example of event handling see the “DemoApps\GLES2\InputEvents” sample. [↑](#footnote-ref-10)
11. 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-11)
12. Future versions will allow demo apps to handle resize events if they so desire. [↑](#footnote-ref-12)
13. The current framework only png, bmp and jpeg images on all platforms but a few platforms has access to all formats supported by the DevIL library. [↑](#footnote-ref-13)
14. A future version will also add saving to the ContentManager. [↑](#footnote-ref-14)