GLOVE™

GL over Vulkan User Manual

Version v0.4

January 20, 2020

Disclaimer

Questions or comments may be directed to:

Think Silicon S.A
Suite B8
Patras Science Park Rion Achaias
26504, Greece
web: http://www.think-silicon.com
email: info@think-silicon.com
Tel:+30 2610 910650

Contents

List of Figures

List of Tables

1 Introduction

GLOVE™ (GL Over Vulkan) is a cross-platform software library that acts as an intermediate layer between an OpenGL® ES application and Vulkan® . GLOVE™ is focused towards embedded systems and is comprised of OpenGL® ES and EGL™ implementations, which translate at runtime all OpenGL ES / EGL calls and ESSL shaders to Vulkan® commands and SPIR-V™ shader respectively and as a final step, relay them to the underlying Vulkan driver.

GLOVE™ has been designed towards facilitating developers to easily build and integrate new features, allowing at the same time its further extension, portability and interoperability. Currently, GLOVE™ supports OpenGL® ES 2.0 and EGL™ 1.4 on Linux and Android platforms, but the modular design can be easily extended to encompass implementations of other client APIs as well.

GLOVE™ is considered as a work-in-progress and is open-sourced under the LGPL v3 license. It is provided as free software with unlimited use for educational and research purposes available in Think Silicon's GitHub repository: https://github.com/Think-Silicon/GLOVE. Future extensions of GLOVE™ are planned to include support of OpenGL® ES 3.x and OpenGL® applications.

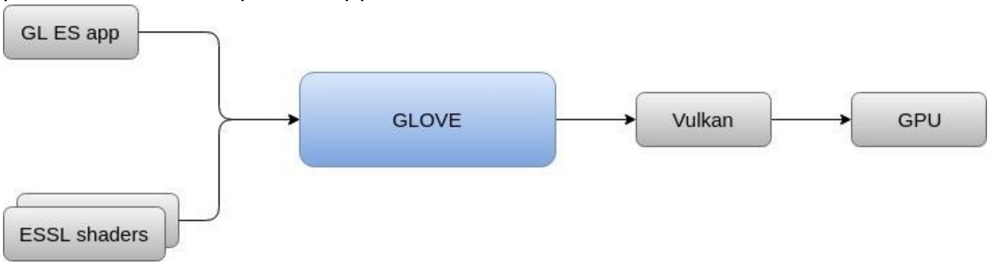

Figure 1: GLOVE™ functionality

2 System Architecture

GLOVE™ is a software library that acts as a bridge between an OpenGL® ES application and a Vulkan® GPU driver. To accomplish this, GLOVE™ offers implementations of OpenGL® ES and EGL™ (Figure 2) and is comprised of two shared libraries: *libGLESv2.so* and *libEGL.so*. Additionally, the translation from ESSL shaders to SPIR-V™ (needed by Vulkan) is handled by the external glslang library. The latter is statically linked to *libGLESv2.so*.

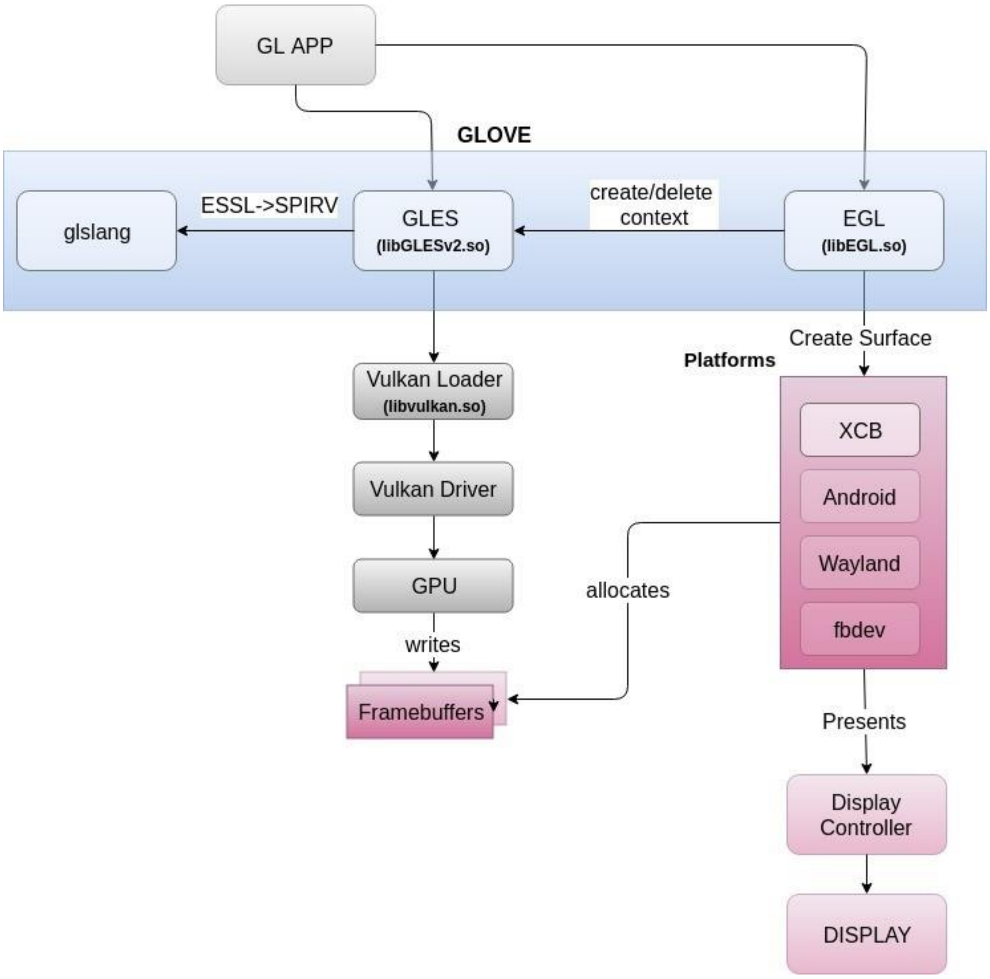

Figure 2: GLOVE™ system architecture

Currently, GLOVE™ supports OpenGL ES 2.0 and EGL 1.4 versions and has been tested with mesa Vulkan Intel driver version 1.0.54.

As a prerequisite for correct function, GLOVE™ must be linked to a Vulkan driver implementation which supports *VK_KHR_maintenance1* extension. This is mandatory for OpenGL to Vulkan coordinates conversion (left handed to right handed coordinate system). The minimum Vulkan loader version must be 1.0.24.

GLOVE™ EGL implementation can be connected to one or more window platforms such as XCB, Wayland, Android or fbdev, which handle framebuffer allocation / deallocation and presentation onto the system's display. Currently EGL supports XCB back-end, but it can be easily extended to support more back-ends (more details in Section 3.2)

2.1 GLOVE™ EGL

EGL folder structure is shown in Figure 3. GLOVE™ EGL implementation is comprised of 2 parts:

- **Rendering Thread**: This part implements rendering thread calls such as *eglBindAPI*, *eglQueryAPI*, *eglCreateContext*, etc. It connects EGL to client APIs and maintains rendering contexts. Currently, GLOVE™ supports connection only to OpenGL ES 2.0, but hosts hooks to enable the connection to other APIs (see Section 3.1)
- **Display Driver**: This part is responsible for creating and maintaining rendering surfaces as well as connecting to a window platform like XCB or Wayland. Platform part is implemented with abstract classes (*platformWindowInterface*, *platformResources*) that can be extended to support any desired platform (more details in section 3.2). Currently, GLOVE™ EGL implements connections to Vulkan WSI using XCB and native rendering (useful on embedded platforms) through the *VK_KHR_xcb_surface* and *VK_KHR_display* extensions, respectively.

2.2 GLOVE™ GLES

GLES folder structure is shown in Figure 4. GLOVE™ GLES implementation is split into 3 main layers:

- **API and Context Layer**: This layer implements all OpenGL ES calls within the scope of a rendering context. According to the user input, it triggers either the *GL State* or *GL Resources* modules.
- **GL State & GL Resources Layer** : *GL State* module is responsible for maintaining the GL state of a rendering context (e.g., activeTexture, activeProgram, CullFace, FrontFace, PolygonOffset). The *GL Resources* module tracks the resources of a rendering context such as textures, shaders, framebuffers and vertex buffers. *Shader* and *ShaderProgram* modules use *glslang* module to compile and link shaders, in order to transform ESSL sources to SPIR-V.
- **Vulkan API Layer**: This layer provides the interface to the Vulkan driver. It is responsible for creating and maintaining all Vulkan objects that are needed to construct and use a rendering pipeline through a Vulkan GPU driver.

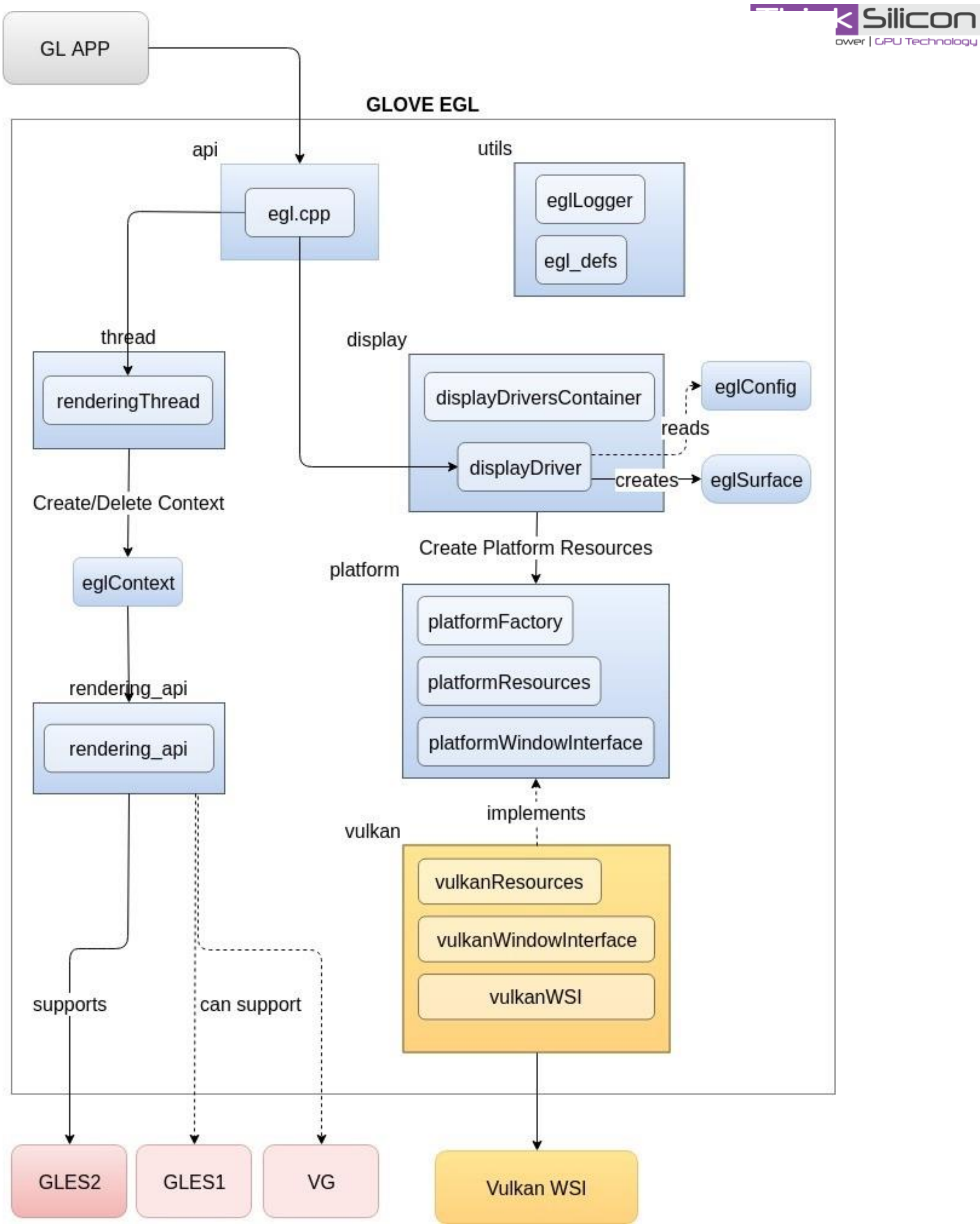

Figure 3: GLOVE™ EGL components

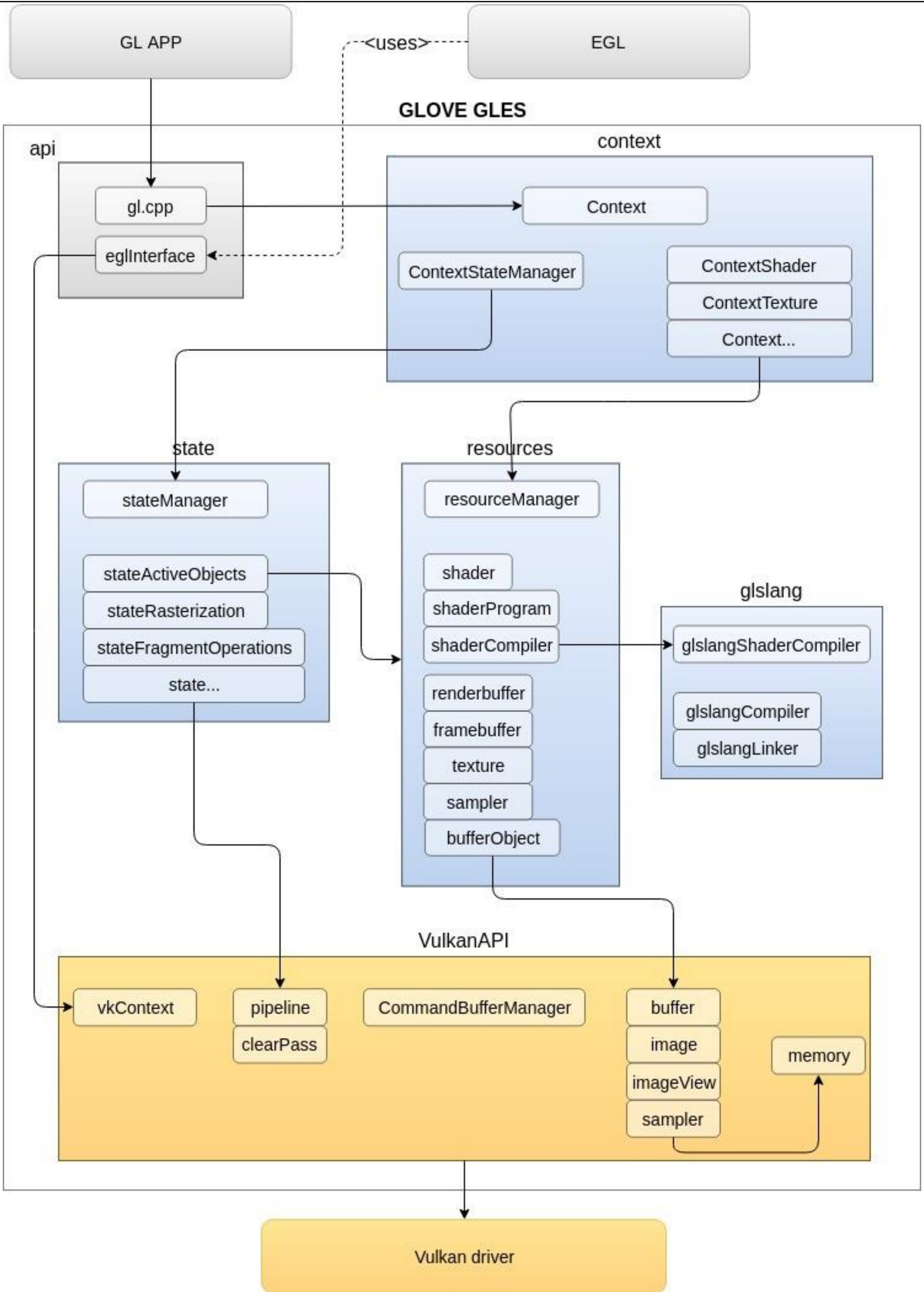

Figure 4: GLOVE™ GLES components

3 GLOVE™ Extensions

3.1 EGL Front-end (Client API)

GLOVE™ EGL can be connected to OpenGL® ES or OpenVG™ client APIs at run-time using the *dlopen* system call. The interface to client APIs is provided via a set of callback functions used in *EGLContext* scope. The client API callbacks are defined in the header file: *EGL/include/rendering_api_interface.h*.

```
typedef struct rendering_api_interface {
        api_state_t state;
        init_API_cb_t init_API_cb;
        terminate_API_cb_t terminate_API_cb;
        create_context_cb_t create_context_cb;
        set_write_surface_cb_t set_write_surface_cb; set_read_surface_cb_t set_read_surface_cb;
       delete_context_cb_t delete_context_cb;
        release_system_fbo_cb_t release_system_fbo_cb; set_next_image_index_cb_t
       set_next_image_index_cb;
        finish_cb_t finish_cb;
} rendering_api_interface_t;
```

To connect a client API to GLOVE™ EGL, the user has to implement the client API callback functions and hold them inside a structure with the following defined names:

1. **GLES1Interface** for OpenGL® ES 1.1
2. **GLES2Interface** for OpenGL® ES 2.0
3. **VGInterface** for OpenVG™

The GLOVE™ EGL will get the client API callbacks by resolving the above symbols names at runtime via *dlsym*. The following code, that implements the **GLES2Interface**, is givena as an example.

```
# include " rendering_api_interface .h"
# include " context / context .h"
# include " glFunctions .h"

static vkInterface_t vkInterface ;
api_state_t gles2_state = nullptr ;

api_state_t init_API ();
void terminate_API ();
api_context_t create_context ();
void set_read_write_surface ( api_context_t api_context ,
EGLSurfaceInterface * eglReadSurfaceInterface , EGLSurfaceInterface *
eglWriteSurfaceInterface );
void delete_context ( api_context_t api_context );
void release_system_fbo ( api_context_t api_context );
void set_next_image_index ( api_context_t api_context , uint32_t index )
;
GLPROC get_proc_addr ( const char * procname );
void flush ( api_context_t api_context );
```

```
void finish ( api_context_t api_context );
void bind_to_texture ( api_context_t api_context , uint32_t bind );

static void FillInVkInterface ( vulkanAPI :: vkContext_t * vkContext );

rendering_api_interface_t GLES2Interface = {
gles2_state ,
init_API ,
terminate_API ,
create_context ,
set_read_write_surface ,
delete_context ,
release_system_fbo ,
set_next_image_index ,
get_proc_addr ,
flush ,
finish ,
bind_to_texture
};

static void FillInVkInterface ( vulkanAPI :: vkContext_t * vkContext )
{
vkInterface . vkInstance = vkContext -> vkInstance ;
vkInterface . vkGpus = & vkContext -> vkGpus [0];
vkInterface . vkQueue = vkContext -> vkQueue ;
vkInterface . vkGraphicsQueueNodeIndex = vkContext -> vkGraphicsQueueNodeIndex ;
vkInterface . vkDeviceMemoryProperties = vkContext -> vkDeviceMemoryProperties ;
vkInterface . vkDevice = vkContext -> vkDevice ;
vkInterface . vkSyncItems = vkContext -> vkSyncItems ;
}

api_state_t init_API ()
{

FUN_ENTRY ( GL_LOG_DEBUG );

vulkanAPI :: InitContext ();

FillInVkInterface ( vulkanAPI :: GetContext ());

return reinterpret_cast < api_state_t >(& vkInterface );
}

void terminate_API ()
{
FUN_ENTRY ( GL_LOG_DEBUG );

vulkanAPI :: TerminateContext ();
GLLogger :: Shutdown ();
}

api_context_t create_context ()
{
FUN_ENTRY ( GL_LOG_DEBUG );

Context *ctx = new Context ();
        return ctx;
```

```
}

void set_read_write_surface(api_context_t api_context, EGLSurfaceInterface * eglReadSurfaceInterface,
EGLSurfaceInterface *eglWriteSurfaceInterface) {
        FUN_ENTRY(GL_LOG_DEBUG);

        Context *ctx = reinterpret_cast<Context *>(api_context);
        ctx->SetReadWriteSurfaces(eglReadSurfaceInterface, eglWriteSurfaceInterface);

        SetCurrentContext(ctx);
}

void delete_context(api_context_t api_context)
{
        FUN_ENTRY(GL_LOG_DEBUG);

        Context *ctx = reinterpret_cast<Context *>(api_context); delete ctx;
}

void release_system_fbo(api_context_t api_context)
{
        FUN_ENTRY(GL_LOG_DEBUG);

        Context *ctx = reinterpret_cast<Context *>(api_context); ctx->ReleaseSystemFBO();
}

void set_next_image_index(api_context_t api_context, uint32_t index)
{
        FUN_ENTRY(GL_LOG_DEBUG);

        Context *ctx = reinterpret_cast<Context *>(api_context); ctx->SetNextImageIndex(index);
}

GLPROC get_proc_addr(const char* procname)
{
        FUN_ENTRY(GL_LOG_DEBUG);

        return GetGLProcAddr(procname);
}

void flush(api_context_t api_context)
{
        FUN_ENTRY(GL_LOG_DEBUG);

        Context *ctx = reinterpret_cast<Context *>(api_context); ctx->Flush();
}

void finish(api_context_t api_context)
{
        FUN_ENTRY(GL_LOG_DEBUG);
```

```
        Context *ctx = reinterpret_cast<Context *>(api_context); ctx->Finish();
}

void bind_to_texture(api_context_t api_context, uint32_t bind)
{
        FUN_ENTRY(GL_LOG_DEBUG);

        Context *ctx = reinterpret_cast<Context *>(api_context); ctx->BindToTexture(bind);
}
```

3.2 EGL Back-end (Platform Support)

GLOVE™ EGL can be connected to any window platform via the platform hooks in *EGL/source/platform* folder of GitHub repository. To insert a new platform, the following classes need to be extended:

1. **platformWindowInterface**: Window Surface creation/ destroy

```
class PlatformWindowInterface
{ public:
        PlatformWindowInterface() { } virtual
        ~PlatformWindowInterface() { }

        virtual EGLBoolean     Initialize() = 0;
        virtual EGLBoolean     Terminate() = 0;;
        virtual EGLBoolean     CreateSurface(EGLDisplay dpy,
              EGLNativeWindowType win, EGLSurface_t *surface) = 0;
              virtual void         AllocateSurfaceImages(EGLSurface_t *surface) =
              0;
             virtual void                                DestroySurfaceImages(EGLSurface_t *eglSurface)
              = 0;
              virtual void DestroySurface(EGLSurface_t *eglSurface) = 0;
              virtual EGLBoolean AcquireNextImage(EGLSurface_t *surface,
              uint32_t *imageIndex) = 0;
            virtual EGLBoolean                             PresentImage(EGLSurface_t *eglSurface) = 0;
};
```

2. **platformResources**: Swap Chain resources holder

```
class PlatformResources
{ public:
        PlatformResources()   { }

        virtual ~PlatformResources() { }

        virtual uint32_t           GetSwapchainImageCount() = 0;
        virtual void   *GetSwapchainImages()       = 0;
};
```

After the platform classes are created, they need to be added in the **PlatfomFactory** class, for GLOVE™ EGL to allocate the appropriate objects for these new platforms. Further modifications are not needed since the general-based platform classes are connected to the rest of the EGL code.

3.3 EGL and GLES Loggers

GLOVE™ EGL and GLOVE™ GLES support logging when "*trace-build*" building flag is enabled. The existing logging modules are based on printf calls and output logs to a standard output. However, it is possible to replace the default logging modules of EGL Logger with a custom logging framework, when the *EGLLoggerImpl* class found in *EGL/source/utils* folder of GitHub repository, is modified. Similar to EGL Logger, a custom logging framework can be added to the GLES Logger when the *GLLoggerImpl* class, found in *GLES/source/utils* folder of GitHub repository, is modified. To use a custom logger, the *SimpleLoggerImpl()* should be replaced with the custom implementation in the following lines of the *eglLogger.cpp* code for EGL and of *glLogger.cpp* for GLES.

```
void
EGLLogger::SetLoggerImpl()
{
if(!mLoggerImpl) { mLoggerImpl = new SimpleLoggerImpl();
        }
}

void
GLLogger::SetLoggerImpl()
{ if(!mLoggerImpl) {
mLoggerImpl = new SimpleLoggerImpl();
        }
}
```

4 GLOVE™ Supported Configurations

Demos for Linux and Android platforms have been used to test the GLOVE™ functionality (Section 6). The configurations that have been successfully tested and that GLOVE™ supports are shown in Table 1

GL version	Graphics Card	Vulkan Driver	Vulkan API	OS	Windows Platform	Status
ES 2.0	Intel Ivybridge Desktop	Mesa 17.3.3	1.0.54	Ubuntu 16.04	XCB	success
ES 2.0	Intel HD Graphics 530 (Skylake GT2)	Mesa 18.0.5	1.0.57	Ubuntu 16.04	XCB	success
ES 2.0	Intel HD Graphics 630 (Kabylake GT2)	Mesa 18.0.5	1.0.61	Ubuntu 16.04	XCB	success
ES 2.0	Intel Ivybridge Desktop	Mesa 17.3.3	1.0.54	Ubuntu 16.04	WAYLAND	success
ES 2.0	Intel HD Graphics 530 (Skylake GT2)	Mesa 18.0.5	1.0.57	Ubuntu 16.04	WAYLAND	success
ES 2.0	Radeon RX 550 Series	Mesa 18.0.5	1.0.61	Ubuntu 16.04	XCB	success
ES 2.0	Radeon RX 550 Series	AMDGPU-Pro v18.40	1.1.77	Ubuntu 16.04	XCB	success
ES 2.0	GeForce 940M	NVIDIA 396.51	1.1.70	Ubuntu 16.04	XCB	success
ES 2.0	GeForce GTX 670	NVIDIA 396.54	1.1.70	Ubuntu 18.04	XCB	success
ES 2.0	Mali-G71	ARM 482.381.3347	1.0.26	Android 7.0	Android	success
ES 2.0	Mali-G71	ARM 485.111.1108	1.0.65	Android 8.0	Android	success
ES 2.0	GeForce GTX 1050	NVIDIA 416.83	1.1.84	Windows 10	Windows	success
ES 2.0	Intel Iris Graphics 6100	MoltenVK v1.0.38	1.1.126	macOS Catalina	MacOS	success

Table 1: GLOVE™ demos for Linux and Windows platforms

5 Installation Instructions

To install GLOVE™, the following steps should be performed:

1. **Download the Repository**

To create your local git repository:

```
git clone https://github.com/Think-Silicon/GLOVE.git
```

2. **Required Packages**

2.1 Required Packages for Linux

To install all required packages:

```
sudo apt-get install git cmake extra-cmake-modules libvulkan-dev vulkan-utils build-essential libx11-xcb-dev
```

Optionally "mesa-vulkan-drivers" package is needed if no other Vulkan driver is available. The compiler minimum version that this project is built with, is GCC 4.9.3, although earlier versions may work.

2.2 Required Packages for MS Windows

To compile GLOVE on Windows, you need

- MS Visual Studio 2019 (Download here), with CMake enabled
- Python3 (Download here)

2.3 Vulkan SDK

To facilitate running and debugging GLOVE on MS Windows, it is recommended to download Vulkan SDK .

2.4 Required Packages for MacOS

Python3 and cmake are required for MacOS. You can install them via homebrew with the following commands

```
brew install cmake
brew install python3
```

2.5 MoltenVK

GLOVE has been tested in macOS, using MoltenVK (Vulkan to Metal middleware), which creates the necessary Vulkan headers and Vulkan loader (libMoltenVK.dylib). Instructions on how to build MoltenVk can be found here.

3. **External Repositories Dependencies**

Khronos glslang repository is mandatory for compiling, validating and generating SPIR-V from ESSL shaders.

Google googletest repository is used for unit testing.

To get and build the above projects:

```
python3.x update_external_sources.py
```

ATTENTION: Python 3 is supported only, so you need to install python 3.x version

Linux Users can also use the equivalent bash shell script, as follows

```
./update_external_sources.sh
```

5.1 Building GLOVE™ for Linux

The building process has been tested on Ubuntu 16.04 and 18.04.

```
$ ./configure.sh [-options]
```

The configuration options are listed in Table 2:

Option	Default	Description
-a | --arm-compile	*OFF*	*Enable cross building for ARM platform*
-d | --debug	*OFF*	*Enable building Debug mode*
-e | --werror	*OFF*	*Turn all compilation warnings into errors*
-f | --use-surface	*XCB*	*Sets the windowing system (Options: XCB, WAYLAND, ANDROID, NATIVE, WINDOWS, MACOS)*
-i | --install-prefix (dir)	*System Installation Prefix (/usr/local)*	*Set custom installation prefix path*
-s | --sysroot (dir)	-	*Set sysroot for cross compilation*
-t | --trace-build	*OFF*	*Enable logs*
-u | --vulkan-include-path (dir)	*System Include Path*	*Set custom Vulkan include path*
-v | --vulkan-loader (lib)	*System Vulkan Loader*	*Set custom Vulkan loader library*

Table 2: Configuration building options for Linux

The Project is built using the following command:

```
$ make
```

To install all the necessary files to system directories the following command is used (superuser privilege maybe required):

```
$ make install
```

Similarly, to uninstall the libraries from the system directories the following command is used (superuser privilege maybe required):

```
$ make uninstall
```

5.2 Building GLOVE™ for Windows

The building process has been tested on Windows 10, using MS Visual Studio 2019.

At first, you should create an MS Visual Studio Project by cloning GLOVE from github. Afterwards, you should resolve the external dependencies, as described here

Option	Default	Description
-d | --debug	*OFF*	*Enable building Debug mode*
-t | --trace-build	*OFF*	*Enable logs*

example of CMakeSettings json file

```
{
    "configurations": [
        {
            "name": "x64-Release",
            "generator": "Ninja",
            "configurationType": "Release",
            "inheritEnvironments": [ "msvc_x64_x64" ],
            "buildRoot": "${projectDir}\\out\\build\\${name}",
            "installRoot": "${projectDir}\\out\\install\\${name}",
            "cmakeCommandArgs": "",
            "buildCommandArgs": "-v",
            "ctestCommandArgs": "",
            "variables": [
                {
                    "name": "TRACE_BUILD",
                    "value": "false",
                    "type": "BOOL"
                }
            ]
        }
    ]
}
```

ATTENTION: Since glslang is built with Release option, it is easier to build GLOVE with Release flag as well. If you need to build GLOVE in Debug mode, you must build glslang in Debug mode as well, otherwise MSVC compiler complains about this. In order to do the latter, you have to modify the 'Build' function of update_external_sources.py script.

To build the Project, use MS Visual Studio GUI (Build->Build All)

5.3 Building GLOVE™ for macOS

The building process has been tested on macOS Catalina (10.15).

GLOVE building can be configured according to the options listed in the following table:

```
./configure.sh [-options]
```

Option	Default	Description
-d | --debug	*OFF*	*Enable building Debug mode*
-e | --werror	*OFF*	*Turn all compilation warnings into errors*
-f | --use-surface	*XCB*	*Sets the windowing system* **(MACOS option must be set for macOS)**
-i | --install-prefix (dir)	*System Installation Prefix (/usr/local)*	*Set custom installation prefix path*
-t | --trace-build	*OFF*	*Enable logs*
-u | --vulkan-include-path (dir)	*System Include Path*	*Set custom Vulkan include path*
-v | --vulkan-loader (lib)	*System Vulkan Loader*	*Set custom Vulkan loader library*

In macOS, the configure.sh script calls CMake with "-G Xcode" argument, thus preparing all necessary files for opening GLOVE in Xcode. Build files are stored in "build" folder.

Open GLOVE.xcodeproj (<GLOVE_root>/build/GLOVE.xcodeproj) with Xcode and build the Project from the tool (Product | Build).

5.4 Building GLOVE™ for Android

The building process has been tested on Android 7 and 8.
To build GLOVE™ for Android, Java 8 is required.

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
```

Download Android Studio on Ubuntu at https://developer.android.com/studio/
Downgrade Android-SDK to version 25:

```
cd <android-sdk-dir>/
mv tools tools_back
wget http://dl.google.com/android/repository/tools_r25-linux.zip
unzip tools_r25-linux.zip
```

Required packages for Android building:

```
sudo apt-get install android-platform-build-headers xcb-proto android-platform-frameworks-native-headers android-platform-system-core-headers android-libcutils-dev ant
```

GLOVE building can be configured according to the options listed in the following table:

```
./android_build.sh  [-options]
```

Option	Default	Description
-d | --debug	*OFF*	*Enable building Debug mode*
-t | --trace-build	*OFF*	*Enable logs*

Table 3: Configuration Options

The above process builds GLOVE for Android and generates an apk to be later installed on the Android device. See the installation process here.

6 GLOVE™ Demos

6.1 GLOVE™ Demos for Linux

GLOVE™ is accompanied by a demo SDK that contains fully commented, highly optimized C applications with ESSL shader source code. These demos demonstrate some simple rendering techniques with different geometry complexities, as they were designed with the restrictions of low-power embedded platforms in mind.

Table 4 show the names of the demos name and a short description.

Name	Dimension	Shading Functionality
triangle2d_one_color	*2D*	*Draw a single 2D triangle with* **white color**
triangle2d_split_colors	*2D*	*A 2D triangle is rendered on the screen. The color of each non-empty pixel is dynamically chosen at runtime between* **2 fixed colors** *(red & green) based on its screen-space position.*
circle2d_sdf	*2D*	*A dynamically generated circle is drawn (with time-varying radius) using* **signed distance fields**. *Note that no geometry (for the 2D circle) is streamed to hardware.*
texture2d_color	*2D*	*A* ***fullscreen quad rendering*** *(using two triangles) is used to draw a 2D texture.*
cube3d_vertexcolors	*3D*	*Draw a 3D rotated (in Y axis) cube in the center of the screen with* ***per-vertex colors***. *Note that this example also supports* **transparency** *via blending operations and* **orthographic** *camera projection (default: perspective).*
cube3d_textures	*3D*	*Similar graphics rendering framework as '* **cube3d_vertexcolors** *' but with different shading method:* **Mixing two textures.**
render_to_texture_filter_gamma	*2D/3D*	*2D image processing effects on a texture generated by off-screen rendering a 3D rotated cube (using demo example* **'cube3d_texture** *'). To achieve this, we initially render scene to a Frame Buffer Object (FBO) and then perform post-processing* **Gamma correction** *filtering on the generated texture.*
render_to_texture_filter_invert	*2D/3D*	*Similar graphics rendering framework as '* **render_to_texture_filter_gamma** *' but with different filtering effect:* **Color Invert**
render_to_texture_filter_grayscale	*2D/3D*	*Similar graphics rendering framework as '* **render_to_texture_filter_gamma** *' but with different filtering effect:* **Grayscale**

Name	Dimension	Shading Functionality
render_to_texture_filter_sobel	*2D/3D*	*Similar graphics rendering framework as '* **render_to_texture_filter_gamma** *' but with different filtering effect:* **Sobel**
render_to_texture_filter_boxblur	*2D/3D*	*Similar graphics rendering framework as '* **render_to_texture_filter_gamma** *' but with different filtering effect:* **Box Blur**

Table 4: GLOVE™ demos for Linux platform

6.1.1 Supported Window Platforms

X11/XCB

By default, GLOVE uses X11/XCB as a Window system. If no particular "--use-surface" option is given in configure.sh, GLOVE builds the XCB backend for EGL.

WAYLAND

For Linux OS, there is a second Window backend alternative, the **Wayland Window System**. If configure.sh is executed with "--use-surface WAYLAND" option, GLOVE builds the WAYLAND backend for EGL.

IMPORTANT: To run GLOVE with WAYLAND, Vulkan Loader must be built with "BUILD_WSI_WAYLAND_SUPPORT" option ON (see details in Vulkan Loader Khronos page).

Once GLOVE is built with WAYLAND backend, GLOVE Demos for Linux can be tested with Weston Server (see HowTo run Weston here).

6.1.2 Execution

Open a new terminal and navigate to the ' **build/Demos/demos**' directory ($ cd build/Demos/demos/), then run all examples by typing this command:

```
$ ./run_all_samples.sh
```

Note that file logging of OpenGL debug and Vulkan profile is supported (use **--help** for details).

6.1.3 Configuration

A number of object-like and conditional macros have been used to offer debug and profiling features as well as to simplify the setting process of the demo configuration (see Table 5).

Name	Values	Functionality
DEBUG	(a)DEBUG_OPENGLES (b)DEBUG_ASSET_MANAGEMENT	*Enable to log errors for (a) OpenGL ES API and (b) asset management respectively.*
PROFILE	(a)FPS_DISPLAY, FPS_TIME_PERIOD = **X** (b)INFO_DISPLAY	*Enable to report (a) processed time in fps and ms for the time period of* **X** *seconds and (b) shading and rendering settings respectively.*
CONFIG	KILL_APP = **Y**	*Enable to auto terminate the application after the time period of* **Y** *seconds (suggestion: Y>X).*
BINARY_PROG	DBINARY_PROG	*Enable to load from a precompiled shader program. Disable to load from vertex and fragment shaders.*
WINDOW_SIZE	WIDTH = W, HEIGHT = H	*Set the dimensions of the application window to be* **[W,H].** *Note that this macro definition is mandatory.*

Table 5: Makefile macros used for the GLOVE™ demos

6.1.4 Key Bindings

A list of key bindings are provided for information and testing purposes (see Table 6). Note that the keys 't' and 'p' are available only for the 'cube3d_vertexcolors' and 'cube3d_textures' demos that handles 3d shapes.

Key	Functionality	Mode
‘Esc’	Exit the application	-
‘t’	Changes the material type of all objects	Opaque/Transparent
‘p’	Changes the projection transformation of the camera	Perspective/Orthographic

Table 6: Key bindings for GLOVE™ demos

6.1.5 Offline Shader Compilation Tool

An offline compiler interface of input (vertex and fragment) shader sources is also provided ensuring that the resulting shader program binary can be efficiently loaded into the graphics application. This is a very useful path for applications that wish to remain portable by shipping pure ESSL source shaders, yet would like to avoid the cost of compiling their shaders at runtime. The tool, which can be found in the ' **build/Demos/tools**' folder ($ cd build/Demos/tools/), works by compiling and linking the given shaders and returns back the final program binary through a command-line interface. If the vertex and fragment ESSL source shader names are '**sh.vert**' and '**sh.frag**' respectively, then the generated precompiled binary shader program, named '**sh.bin**', can be generated by executed this command:

```
$ ./offline_shader_compiler -v sh.vert -f sh.frag -o sh.bin
```

Note that, the BINARY_PROG macro preprocessor in the ' CMakeLists.txt' file has to be provided in the CMAKE_C_FLAGS to inform graphics applications to use precompiled shaders (Table 5).

6.2 GLOVE demos for Windows

GLOVE demos described in previous section are supported on Windows as well.

6.2.1 Execution

By using either command prompt or Windows explorer, navigate to **GLOVE\out\build<build_name>\bin** and execute **run_all_samples** batch file to run all samples.

```
run_all_samples.bat
```

6.3 GLOVE demos for MacOS

GLOVE demos described in the first section are supported on MacOS as well.

6.3.1 Execution

Navigate to <GLOVE_root>/build/Demos/demos and execute each one of the applications under <build_type> folder (e.g. for Debug build execute triangle2d_one_color.app under Debug folder).

To execute all samples in a row, just execute the following script.

```
./run_all_samples_mac.sh <build_type>
```

build_type: Debug/Release

6.4 GLOVE™ Demos for Android

Currently, GLOVE uses the es2gears demo (official link here) as a demo application for Android.

6.4.1 Installation

The installation procedure requires a device that supports Vulkan. Ensure that the device is connected to the building machine, open a new terminal and type the following command:

```
$ adb install android/bin/NativeActivity-debug.apk
```

The application will be automatically installed to the Android device.

6.4.2 Execution

To execute the application, tap on the application icon named *GLOVE_Demo*.

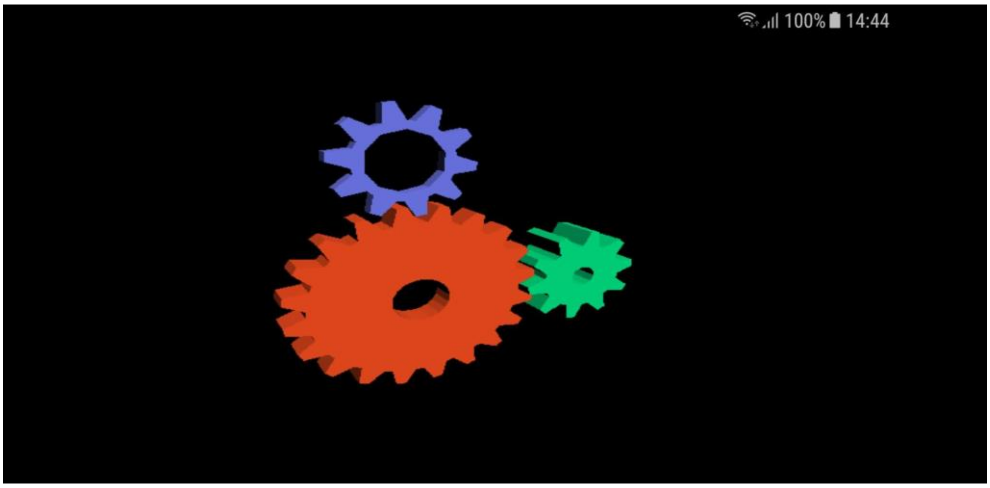

Figure 5: GLOVE™ Demo for Android platform

7 GLOVE™ Benchmarking

GLOVE™ aims to take advantage of Vulkan in terms of performance. The preliminary results are very promising and further major performance upgrades are in progress.

glmark2 is an OpenGL 2.0 and ES 2.0 benchmark used for GLOVE™ testing. To clone the *glmark2* repository, the following command is used:

```
$ git clone https://github.com/glmark2/glmark2.git
```

The build and install procedures can be found in Section 5. Note that - -*with-flavors=x11-glesv2* must be used in build configuration.

To run *glmark2* benchmark, the following command is used:

```
$ <path to glmark2-es2 executable>/glmark2-es2 --reuse-context -f <path to GLOVE root
      >/Benchmarking/glmark/glmark2_benchmarks_options
```

- - -*reuse-context* option is needed since GLOVE™ does not yet fully support multiple contexts
- *glmark2_benchmarks_options* file contains a list of the so far supported benchmarks by GLOVE™