Tizen 3.0's Window System Integration
Layer of OpenGLES/EGL & Vulkan Driver

(libtpl-egl, vulkan-wsi-tizen)

Mun Gwan-gyeong
Software R&D Center
Samsung Electronics
Agenda

- Tizen 3.0 Window System Architecture
- Tizen 3.0 Window System Integration Layer of OpenGL/EGL
  - libtpl-egl (Tizen Porting Layer for EGL)
- Tizen 3.0 Vulkan WSI for Tizen
  - vulkan-wsi-tizen
Tizen 3.0 Window System Architecture
Tizen 3.0 Window System Architecture

- Display Server (Forked from E20)
- Input manager
- TDM (Display manager)
- TBM (Buffer manager)
- GPU Vendor DDK
- Vulkan
- GL (EvasGL, CoreGL)
- Vulkan Loader (Khronos)
- 3D UI Toolkit
- libtpi-egl

Kernel layer – display sub system, input sub system, memory sub system

Tizen API

- Application
- Top-level Module
- Backend Module
- Uses

Copyright © 2016 Samsung Electronics, Co., Ltd. All rights reserved.
Components description

**TPL-EGL** is an abstraction layer for surface and buffer management on Tizen platform aimed to implement the EGL porting layer of OpenGLES driver over various display protocols.

**Vulkan-WSI-Tizen** wraps vendor's vulkan ICDs and provides the WSI(Window-System Interface) for the tizen.

**Tizen Buffer Manager (TBM)** provides the abstraction interface for the graphic buffer manager in Tizen.

**Tizen Display Manager (TDM)** provides the abstraction interface for the display server, such a wayland server, to allow the direct access to graphics hardware in a safe and efficient manner as a display HAL.
Components description (cont.)

- **EvasGL** is a kind of Evas Object image for opengl and it is a GLES Wrapper.

- **CoreGL** is an injection layer of OpenGL ES that provides the following capabilities:
  - Support for driver-independent optimization (FastPath)
  - EGL/OpenGL ES debugging
  - Performance logging
Tizen Porting Layer for EGL (libtpl-egl)
Tizen OpenGL ES and EGL Architecture

Diagram:

- **GLES Application**
  - GLESV1_CM (CoreGL Frontend)
  - GLESV2 (CoreGL Frontend)
  - EGL (CoreGL Frontend)

- **CoreGL Core**

- **GPU Vendor GL / EGL Driver**
  - libtpt-egl Frontend

- **GPU**
  - TBM backend
  - wayland-egl backend
  - GBM backend
**Tizen Porting Layer for EGL**

**Tizen Porting Layer (TPL) Architecture**
- TPL provides implementation of EGL platform functions on Tizen platform

![Diagram showing Tizen Porting Layer (TPL) Architecture]

**TPL?**
- Background
  - Various window system protocols in Tizen
    - Wayland, gbm, tbm, X11 (Tizen 3.0 Alpha)
  - Needs to separating common layer (frontend, duplicated code) and backend for maintaining
- Why TPL?
  - TPL-EGL APIs prevents burdens of EGL porting on various window system protocols.
  - Vendor GL Driver’s Window System Porting Layer functions treat only TPL-EGL APs.
  - If libtpl-egl has improved performance, then Vendor driver can get it without modification of code.
TPL Frontend Interface

Tizen Porting Layer Core Object

- **TPL Object**
  Base class for all TPL objects

- **TPL Display**
  Encapsulate native display object (wl_display, gbm_device, tbm_bufmgr )

- **TPL Surface**
  Encapsulate native drawable object (wl_surface, gbm_surface, tbm_surface_queue_h )
**Tizen Porting Layer for EGL**

**TPL**

- Provides TPL objects which correspond to EGL objects

---

![Diagram of Tizen Porting Layer for EGL]

- **EGLDisplay**
  - TPL Display
  - Native Display

- **EGLSurface**
  - TPL Surface (TBM_Surface_Queue)
  - Native Drawable

- **Driver's egl buffer**
  - Driver's internal memory handle

- **TBM_Surface**
  - DMABUF

---

GL Driver's Object  
TPL Object, TBM Surface  
Window System / DRM Object

---

Copyright © 2016 Samsung Electronics, Co., Ltd. All rights reserved. 11 / 33
GLES Drawing API Flow

EntryPoints Functions (egl entrypoints)
- eglGetDisplay
- eglInitialize
- eglChooseConfig
- eglCreateWindowSurface
- eglCreateContext
- GL calls (ex. glClear)
- eglSwapBuffer

GL Driver’s Window system porting Functions
- Create Driver’s display
- Create Driver’s window surface
- Get window target buffer
- Display window buffer

TPL(Tizen Porting Layer) (tpl.c)
- tpl_display_create
- tpl_surface_create
- tpl_surface_dequeue_buffer
- tpl_surface_enqueue_buffer

TPL display creation & initialization
Create TPL surface for the EGLSurface
Get the buffer of the current frame for render target
Called from GL Driver’s EGL Porting Layer when the rendering is done.
**Simple example** of the Tizen Porting Layer

```c
tpl_display_t *dpy = tpl_display_create(...);
tpl_surface_t *sfc = tpl_surface_create(dpy, ...);
tbm_surface_h buf;
while (1)
{
    buf = tpl_surface_dequeue_buffer(sfc); // get buffer
    /* Draw something */
    tpl_surface_enqueue_buffer(sfc, buf); // post buffer
}
```

[pseudo code] Using libtpl-egl api

In the GPU vendor driver, the "Draw something" part is what the GPU frame builder does.
TPL-EGL exposes the native platform buffer as `tbm_surface`. If `tbm` backend uses `drm-backend`, GL Driver can get `dma_buf` from `tbm_surface`'s buffer object.
### TPL Object

- Base class for all TPL objects
- Provide common functionalities of all TPL objects

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpl_object_reference</td>
<td>Increase reference count of the given TPL object</td>
</tr>
<tr>
<td>tpl_object_unreference</td>
<td>Decrease reference count and destroy it if it becomes 0</td>
</tr>
<tr>
<td>tpl_object_get_reference</td>
<td>Get reference count of the given TPL object</td>
</tr>
<tr>
<td>tpl_object_get_type</td>
<td>Get type of the object (display or surface)</td>
</tr>
<tr>
<td>tpl_object_set_user_data</td>
<td>Set user data and free callback for destruction</td>
</tr>
<tr>
<td>tpl_object_get_user_data</td>
<td>Get user data</td>
</tr>
</tbody>
</table>
TPL Frontend API (tpl_display)

**TPL Display**
- Encapsulate native display object (wl_display, gbm_device, tbm_bufmgr)
- Any other objects are created from TPL Display, they are inherited backend type from TPL Display.

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpl_display_create</td>
<td>Creates the TPL-EGL display object for the given native display</td>
</tr>
<tr>
<td>tpl_display_get</td>
<td>Retrieve the TPL-EGL display for the given native display handle</td>
</tr>
<tr>
<td>tpl_display_get_native_handle</td>
<td>Get native handle of the given display</td>
</tr>
<tr>
<td>tpl_display_query_config</td>
<td>Query pixel format information</td>
</tr>
<tr>
<td>tpl_display_get_native_window_info</td>
<td>Query information on the given native window.</td>
</tr>
<tr>
<td>tpl_display_get_nativePixmap_info</td>
<td>Query information on the given native pixmap.</td>
</tr>
<tr>
<td>tpl_display_get_buffer_from_nativePixmap</td>
<td>Get native buffer from the given native pixmap.</td>
</tr>
</tbody>
</table>
TPL Frontend API (tpl_surface)

TPL Surface

- Encapsulate native drawable object (wl_surface, gbm_surface, tbm_surface_queue_h)

Main Features

- Get the buffer for a surface
- Post the buffer to a surface of screen

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpl_surface_create</td>
<td>Create a TPL-EGL surface for the given native drawable</td>
</tr>
<tr>
<td>tpl_surface_get_display</td>
<td>Get TPL-EGL display of the given surface</td>
</tr>
<tr>
<td>tpl_surface_get_native_handle</td>
<td>Get native handle of the given surface</td>
</tr>
<tr>
<td>tpl_surface_get_type</td>
<td>Get type of the given surface (Window or Pixmap)</td>
</tr>
<tr>
<td>tpl_surface_get_size</td>
<td>Get size of the given surface</td>
</tr>
<tr>
<td>tpl_surface_dequeue_buffer</td>
<td>Get buffer (as TBM_SURFACE) of the current frame for the given surface</td>
</tr>
<tr>
<td>tpl_surface_validate</td>
<td>Check current buffer is valid</td>
</tr>
<tr>
<td>tpl_surface_set_post_interval</td>
<td>Set post interval</td>
</tr>
<tr>
<td>tpl_surface_get_post_interval</td>
<td>Get post interval</td>
</tr>
<tr>
<td>tpl_surface_enqueue_buffer</td>
<td>Post to screen</td>
</tr>
</tbody>
</table>
Wayland Server / Client on libtpl-egl

Wayland Display Server

Enlightenment

EVAS GL TBM / DRM

EGL

TBM / GBM

libtpl-egl

GPU Vendor GL Driver

libwayland-tbm

Wayland Client

Wayland-EGL Client

EGL

libwayland-egl

libtpl-egl
Buffer Flow (Wayland Server ↔ GLES/EGL Driver)

Buffer Flow Between the Wayland Server and GLES/EGL Driver
Vulkan WSI for Tizen (vulkan-wsi-tizen)
Tizen Vulkan Architecture

- Vulkan Application
- Khronos Vulkan Loader
- Vulkan WSI Tizen
- Vendor’s Vulkan ICD
- libtpl-egl
- GPU
Vulkan WSI for Tizen

**Objectives**
- Applications should be able to use khronos vulkan loader
- Do not modify khronos vulkan loader
- Separate WSI binary across multiple vendor ICDs
- Don’t do any platform specific things, use TPL instead

**Architecture**
- WSI wraps the ICD and act like a complete ICD

![Diagram of Vulkan WSI for Tizen architecture]

- **Vulkan Application**
- **Khronos Vulkan Loader**
- **Vulkan WSI Tizen**
- **Vendor ICD**
- **Tizen WSI/ICD Interface**
- **TPL/TBM**
- **Tizen Window System**
Vulkan Loader (Khronos Vulkan Loader)

- Loader exposes vulkan symbols to applications (libvulkan.so)
- Loader opens an ICD shared object file and dispatches ICD functions via vk_icdGetInstanceProcAddr()
  - This is recommended way according to the khronos loader document
- Application calls a loader function, then loader function finally calls the dispatched ICD function
  - Vulkan is layered architecture
Wrapping ICD (vulkan-wsi-tizen)

- Vulkan WSI Tizen acts like a complete ICD
  - Exposes vk_icdGetInstanceProcAddr() which dispatches all required Vulkan functions
- Some functions are implemented in vulkan-wsi-tizen, while others come from vendor ICD
- API Hooks
  - Vulkan WSI Tizen hooks desired Vulkan functions
  - Hooked vulkan-wsi-tizen functions are dispatched instead of ICD functions
  - vkGetInstanceProcAddr(), vkGetDeviceProcAddr() are hooked by default
    - If not, an (Vendor’s) ICD function might be dispatched even though it is hooked by WSI
Extension Merge (vulkan-wsi-tizen)

Extension Merge

- vulkan-wsi-tizen merges extensions from Vendor ICD and vulkan-wsi-tizen’s own extension
- vulkan-wsi-tizen hooks extension enumeration functions
- vkEnumerateInstanceExtensionProperties() in vulkan-wsi-tizen
  - Vendor ICD instance extension + VK_KHR_surface + VK_KHR_wayland_surface
- vkEnumerateDeviceExtensionProperties() in vulkan-wsi-tizen
  - Vendor ICD device extension + VK_KHR_swapchain
WSI Surface Functions (Khronos Vulkan Loader)

- WSI Surface Functions
  - Surface functions are implemented in the khronos loader
  - Surface object is passed to the vulkan-wsi-tizen when other WSI function is called
    - ex) vkCreateSwapchainKHR
  - Data structure for the loader surface object can be accessed via vk_icd.h (Khronos Vulkan Loader’s header file)

Ex) WSI Swapchain Functions

```
libvulkan.so
  (loader)

  foo()
  bar()
  baz()
  ...

vulkan-wsi-tizen.so

  foo()
  baz()
  ...

icd.so
  (vendor’s)

  foo()
  bar()
  ...
```
WSI Functions (vulkan-wsi-tizen)

- WSI functions except surface functions are implemented and hooked
- WSI function categories
  - Surface capability query functions
    - Formats, presentation support, ...
    - ex) vkGetPhysicalDeviceSurfaceCapabilitiesKHR(), vkGetPhysicalDeviceSurfaceFormatsKHR() ...
  - Swapchain functions
    - ex) vkCreateSwapchainKHR(), vkGetSwapchainImagesKHR(), vkAcquireNextImageKHR(), vkQueuePresentKHR() ...
  - Display functions
    - Required when presenting directly to a display device

Ex) WSI Swapchain Functions
Swapchain related API (vulkan-wsi-tizen)

Swapchain
- Manages image (buffer) queue
- `vkAcquireNextImageKHR()`
  - Acquire a next image from the presentation engine
- `vkQueuePresentKHR()`
  - Present the given image to the presentation engine
- Implemented using TPL surface

`vkAcquireNextImageKHR()`
- `tpl_surface_dequeue_buffer()`
- Find index of the dequeued buffer and return

`vkQueuePresentKHR()`
- `tpl_surface_enqueue_buffer()`
Vulkan WSI Tizen ↔ Vendor’s ICD interface

- Vulkan WSI Tizen ↔ Vendor’s ICD interface
  - Vendor’s ICD should provide functions required by the Vulkan WSI Tizen
  - `vk_tizen.h`
    - Defines functions ICD should provide
    - Vulkan WSI Tizen should be able to dispatch those functions via Vendor ICD’s `vk_icdGetInstanceProcAddr()`

- `vkCreateImageFromNativeBufferTIZEN()`
  - It creates a VkImage from a tizen native buffer (`tbm_surface_h`)
  - It is called by `vkCreateSwapchainKHR()` of `vulkan-wsi-tizen`
  - Arguments
    - [in] `VkDevice`
      - `VkDevice` is passed by `vkCreateSwapchainKHR()`
    - [in] `tbm_surface_h`
      - Native tizen buffer
    - [in] `const VkImageCreateInfo *`
      - Properties of the native tizen buffer (dimension, format, ...)
    - [in] `const VkAllocationCallbacks *`
      - Allocation callbacks used for host memory allocation
    - [out] `VkImage *`
      - Vendor ICD should create `vkImage` from `tbm_surface`.
      - `vkAcquireNextImageKHR()` uses this `VkImage`. 
Vulkan WSI Tizen ↔ Vendor’s ICD (cont.)

vkQueueSignalReleaseImageTIZEN()

- When the vendor’s vulkan driver ends up the handling of vkImage and it is ready to present (all waiting semaphores are triggered), Vendor ICD notifies to vulkan-wsi-tizen (NativeFenceFd is created by Vendor Driver.)
- It is called by vkQueuePresentKHR() of vulkan-wsi-tizen

Arguments

- [in] VkQueue
  - VKQueue is passed by vkQueuePresentKHR()
- [in] uint32_t
  - waitSemaphoreCount is passed by VkPresentInfoKHR of vkQueuePresentKHR()
- [in] const VkSemaphore *
  - WaitSemaphore list is passed by VkPresentInfoKHR of vkQueuePresentKHR()
- [in] VkImage
  - VkImage index is passed by VkPresentInfoKHR of vkQueuePresentKHR()
- [out] int *NativeFenceFd
  - Vendor ICD should create NativeFenceFd from WaitSemaphore list.
  - vulkan-wsi-tizen waits NativeFeceFd by tbm_sync_fence_wait().
Vulkan WSI Tizen ↔ Vendor’s ICD (cont.)

vkAcquireImageTIZEN()
- It notifies the acquired Image which is ready to use to the Vendor’s Vulkan Driver.
- It is called by vkAcquireNextImageKHR() of vulkan-wsi-tizen

Arguments
- [in] VkDevice
  - VkDevice is passed by vkAcquireNextImageKHR()
- [in] VkImage
  - VkImage index is passed by vkAcquireNextImageKHR()
- [in] int nativeFenceFD
  - Vulkan driver should wait this nativeFenceFD until Display Server triggers it. (Display Server uses tbm_sync_timeline_inc() for triggering)
  - nativeFenceFD is created by tbm_sync_fence_create()
- [in] VkSemaphore
  - Vendor ICD connects VkSemaphore to nativeFenceFD
  - When nativeFenceFD is triggered, Vendor ICD signals VkSemaphore
- [in] VkFence
  - Vendor ICD connects VkFence to nativeFenceFD
  - When nativeFenceFD is triggered, Vendor ICD signals VkFence
### Supported WSI Spec (Current State)

#### Surface & Swapchain Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCreateWaylandSurfaceKHR</td>
<td>Provided by khronos loader</td>
</tr>
<tr>
<td>vkDestroySurfaceKHR</td>
<td>Provided by khronos loader</td>
</tr>
<tr>
<td>vkGetPhysicalDeviceWaylandPresentationSupportKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkGetPhysicalDeviceSurfaceSupportKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkGetPhysicalDeviceSurfaceCapabilitiesKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkGetPhysicalDeviceSurfaceFormatsKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkGetPhysicalDeviceSurfacePresentModesKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkCreateSwapchainKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkCreateSharedSwapchainKHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>vkDestroySwapchainKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkGetSwapchainImagesKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkAcquireNextImageKHR</td>
<td>Done</td>
</tr>
<tr>
<td>vkQueuePresentKHR</td>
<td>Done</td>
</tr>
</tbody>
</table>
## Supported WSI Spec

### Present Modes

<table>
<thead>
<tr>
<th>Modes</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PRESENT_MODE_IMMEDIATE_KHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_MAILBOX_KHR</td>
<td>Done</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_FIFO_KHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>VK_PRESENT_MODE_FIFO_RELAXED_KHR</td>
<td>Not Implemented Yet</td>
</tr>
</tbody>
</table>

### Display Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCreateDisplaySurfaceKHR</td>
<td>Provided by khronos loader</td>
</tr>
<tr>
<td>vkGetPhysicalDeviceDisplayPropertiesKHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>vkGetPhysicalDeviceDisplayPlanePropertiesKHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>vkGetDisplayPlaneSupportedDisplaysKHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>vkGetDisplayModePropertiesKHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>vkCreateDisplayModeKHR</td>
<td>Not Implemented Yet</td>
</tr>
<tr>
<td>vkGetDisplayPlaneCapabilitiesKHR</td>
<td>Not Implemented Yet</td>
</tr>
</tbody>
</table>
References

Project Git repository (https://review.tizen.org/gerrit/#/admin/projects/)

<table>
<thead>
<tr>
<th>Project</th>
<th>Repository</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libtpl-egl</td>
<td>platform/core/uifw/libtpl-egl</td>
<td>Tizen Porting Layer for EGL</td>
</tr>
<tr>
<td>vulkan-wsi-tizen</td>
<td>platform/core/uifw/vulkan-wsi-tizen</td>
<td>vulkan wsi tizen icd, it wraps vendor icd and provides wsi for tizen</td>
</tr>
<tr>
<td>libtbm</td>
<td>platform/core/uifw/libtbm</td>
<td>The library for the Tizen Buffer Manager</td>
</tr>
<tr>
<td>coregl</td>
<td>platform/core/uifw/coregl</td>
<td>An injection layer of OpenGL ES / EGL</td>
</tr>
<tr>
<td>wayland-tbm</td>
<td>platform/core/uifw/wayland-tbm</td>
<td>Wayland tbm is a protocol for graphics memory management for Tizen</td>
</tr>
<tr>
<td>emulator-yagl</td>
<td>platform/adaptation/emulator/emulator-yagl</td>
<td>OpenGL ES / EGL driver for the emulator</td>
</tr>
<tr>
<td>tpl-novice</td>
<td>platform/core/uifw/ws-testcase</td>
<td>Novice test framework for TPL</td>
</tr>
</tbody>
</table>

libtpl-egl Reference Driver

- The Emulator YAGL (OpenGLES / EGL driver for the emulator) is implemented by libtpl-egl.
- The following commit explains how to port the driver with libtpl-egl from the traditional drm-based driver.
- Porting YAGL to the Tizen platform https://review.tizen.org/gerrit/#/c/67921/
Thank you