

LunarG Surface Management Design

Chia-I Wu, olv@LunarG.com; Mike Stroyan, Mike@LunarG.com; Alan Ward, Alan@LunarG.com

1. Overview

This document describes the changes to Gallium3D to enable support for resource sharing between EGL and its rendering and non-rendering APIs (or, collectively media APIs). Initially these APIs will include EGL 1.4, OpenGL ES 2.0, OpenVG 1.1, and OpenMax IL 1.1.2. The motivation of the driver-level resource sharing is to allow EGL and its media APIs to support various EGLImage extensions.

2. Purpose and Goals:

In OpenGL ES, texture objects and renderbuffers have type GLuint. In OpenVG, an image object has type VGImage. In OpenMax IL, a buffer has type OMX_BUFFERHEADERTYPE. All these API resources are buffers themselves, yet a resource created by one API cannot be passed to another API. Gallium3D has solved the harder part of the issue: all resources are pipe_textures. What remains to be done is to define a way to look up a resource and return its pipe_texture/pipe_surface.

Rendering APIs are part of Gallium3D. The proposed solution is to add an interface that allows each of the rendering APIs to communicate with EGL both ways. It will allow EGL to look up a rendering API resource, and conversely, a rendering API to look up an EGL resource.

The only non-rendering API in existence is OpenMax. It is not part of Gallium3D. To extend pipe_textures and the look up mechanism for third party implementation of OpenMax or any future project that is not part of Gallium3D, a public interface needs to be defined.

3. Surface Manager

Behind the two interfaces for rendering and non-rendering APIs lies the pipe_screen. It is used to allocate pipe_textures for resources. A closer look at the use of pipe_screen in rendering APIs hints that, for resource management, we only need to

- Allocate/Destroy pipe_textures
- Map pipe_textures for CPU access

Since an EGLImage is a view into a pipe_texture, just like a pipe_surface, it motivates us to call this driver-level resource sharing mechanism Surface Manager, and thus the name of the public interface we are going to define. Surface Manager should be able to

- Allocate/Destroy surfaces (single-leveled pipe_textures with depth 1)
- Map surfaces for CPU access
- Create surfaces from EGLImages

4. Changes to Gallium3D:

To implement Surface Manager in Gallium3D, two changes will be made. st_api.h will be added and



implemented to support resource sharing between EGL and rendering APIs. A new state tracker will be added to support resource sharing between EGL and non-rendering APIs.

Other than a replacement for `st_public.h`, `st_api.h` provides also all required resource management functionality. As depicted below in Figure 4a, each rendering API implements the client side of `st_api.h`, while EGL implements the manager side of `st_api.h`. The interface gives rendering APIs access to the `pipe_screen` and resource lookup.

We've picked Bellagio as the reference OpenMax IL implementation. The Bellagio implementation of OpenMax IL is not Gallium3D-based. The public interface for non-rendering APIs is implemented as a state tracker, Surface Manager state tracker. It will install itself as a library. The existing Bellagio implementation with software driver can be left largely as is. Those components that work with EGLImage will be updated to use this state tracker, as depicted in the right-hand part of Figure 4a below.

Gallium3d Surface Management for API Integration

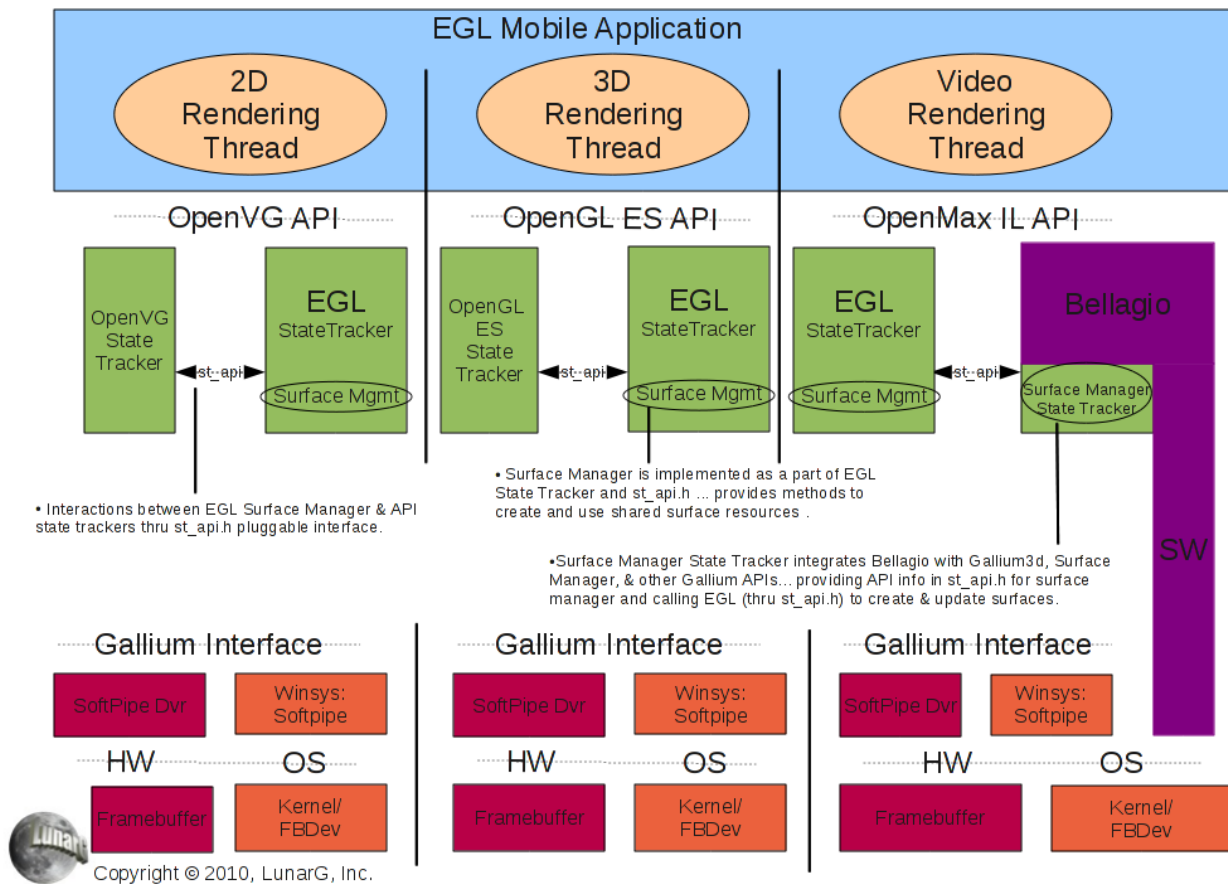


Figure 4a: Gallium3D Surface Management Block Diagram



Glossary

pipe_context

Gallium 3D abstraction of a rendering context. This extensible data structure combines data and callable methods. This parallels an EGL client API context.

pipe_screen

Gallium 3D abstraction of a screen. This extensible data structure combines data and callable methods. Basically everything hardware-specific that doesn't actually require a rendering context.

pipe_surface

Gallium 3D abstraction of a 2D surface. This extensible data structure combines data and callable methods. This is basically a view into a memory buffer. May be a renderbuffer, texture mipmap level, etc.

pipe_texture

Gallium 3D abstraction of a texture. This extensible data structure combines data and callable methods. It has format information, reference counting, pipe_surfaces, and information about the pipe_screen this texture belongs to.

state_tracker

Gallium 3D implementation of an API on top of the Gallium3D driver interface and pipe drivers. Each state tracker handles the specific entry points and state of a particular rendering API or management API. There is a state tracker for each of OpenGL ES, OpenVG, and EGL.

Appendix: st_api.h

The current version of st_api.h is available at http://cgit.freedesktop.org/~olv/st_api/. The details will be added as it is stabilized.

Appendix: Surface Manager State Tracker

Details in this section are our current specification. As we get into implementation, some of these details will probably change somewhat.

When an EGLImage or its targets are created, the buffer of the source may be reallocated. That is, the pixel data of the source may become undefined. This is for buffer compability (and/or performance) imposed by different hardware units.

This behavior is reasonable for targets that are created to render to the source. However, it is not suitable for targets that are created to "read" the source. An example of the second type is the classic texture-from-pixmap: A target is created to use a pixmap as a texture. This type of application is not possible if the pixel data of the source may become undefined. The issue is resolved through EGL_IMAGE_PRESERVED_KHR attribute. When set to true, the creation of an EGLImage target should fail if it cannot preserve the pixel data.

EGL_IMAGE_PRESERVED_KHR can always be supported by copying the pixel data from the old buffer to the newly allocated buffer when reallocation happens. It is however a heavy operation. This is especially true when used with video streaming. This state tracker is designed to make sure the



buffer allocated for a native pixmap, VGImage, texture image, or video frame can meet the hardware/performance requirement of its possible uses in the first place. When this is guaranteed, the creation of EGLImage or its targets will re-use the buffer of the EGLImage source. No reallocation nor copying will be required.

C Types

```
typedef void *SMSurface;  
typedef EGLint SMint;
```

C Tokens

```
typedef enum {  
    SM_SURFACE_FORMAT_NONE = 0,  
    SM_SURFACE_FORMAT_A8R8G8B8,  
    SM_SURFACE_FORMAT_YUV422,  
    SM_SURFACE_FORMAT_YUV422_PACKED,  
    SM_SURFACE_FORMAT_YUV422_SEMI,  
} SMSurfaceFormat;  
  
typedef enum {  
    SM_SURFACE_USAGE_SAMPLE      = 0x1,  
    SM_SURFACE_USAGE_RENDER      = 0x2,  
} SMSurfaceUsage;  
  
typedef enum {  
    SM_SURFACE_ATTRIB_WIDTH,  
    SM_SURFACE_ATTRIB_HEIGHT,  
    SM_SURFACE_ATTRIB_FORMAT,  
    SM_SURFACE_ATTRIB_USAGES,  
} SMSurfaceAttrib;  
  
typedef enum {  
    SM_SURFACE_MAP_HINT_READ      = 0x1,  
    SM_SURFACE_MAP_HINT_WRITE     = 0x2,  
} SMSurfaceMapHint;
```

C Functions

```
SMSurface smCreateSurface(EGLDisplay display,  
                          SMint width, SMint height,  
                          SMSurfaceFormat format, SMint usages);  
  
void smDestroySurface(SMSurface surface);  
  
SMint smQuerySurface(SMSurface surface,  
                    SMSurfaceAttrib attrib);  
  
void *smMapSurface(SMSurface surface,  
                  SMint hints, SMint *stride);  
  
void smUnmapSurface(SMSurface surface);
```

Specification

Surface Manager is thread-safe. Unlike EGL or media APIs, invalid uses of Surface Manager might crash the application. The application should be careful not to, for example, destroy a non-existent



surface, etc.

The command

```
SMSurface smCreateSurface(EGLDisplay display,  
                           SMint width, SMint height,  
                           SMSurfaceFormat format, SMint usages);
```

returns a surface. It returns NULL if the display is uninitialized. Once created, the surface remains valid even after the display is terminated.

The command

```
void smDestroySurface(SMSurface surface);
```

destroys a surface. If the surface is mapped, it is unmapped first.

The command

```
SMint smQuerySurface(SMSurface surface,  
                     SMSurfaceAttrib attrib);
```

can be used to query surface attributes. Querying an invalid attribute always returns 0.

The command

```
void *smMapSurface(SMSurface surface,  
                  SMint hints, SMint *stride);
```

maps a surface to the address space for CPU access. On errors, it returns NULL. Nested mapping is considered an error. A mapped surface is a CPU resource. If the surface is bound in the GPU pipeline, it should be unmapped before any rendering operation.

The command

```
void smUnmapSurface(SMSurface surface);
```

unmaps a surface. If the surface is not mapped, the command is no-op.

Issues:

1. Is `smReallocateSurface` missing?
RESOLVED: No. A new surface should be allocated to replace the old surface for reallocation.
2. What is the lifetime of a surface?
RESOLVED: As long as it is not destroyed. OpenMax IL components do not depend on an `EGLDisplay`. Even an `EGLContext` might live longer than its `EGLDisplay`. Once created, a surface should live until destroyed.
3. Can a surface be mapped recursively?
RESOLVED: No. It will make the semantics too complex.
4. Do we need an `EGLDisplay` to create a surface?



RESOLVED: Yes. It might be as well to take an EGLNativeDisplayType to create a surface. Using EGLDisplay has the benefit that Surface Manager need not define any native types. The downside is that EGL cannot be implemented by Surface Manager.

The reason we need an EGLDisplay or native display to create a surface is that, we want to use the display to select the GPU(s)/codec(s). On platforms with multiple displays, we might want to choose from which display the surface is created. But again, the downside is that the native display cannot be implemented by Surface Manager.

5. What flags should be available when a surface is created?

UNRESOLVED: Other than format and usages, it might also be desired to specify the access flags (GPU read/write? CPU read/write?) and the use pattern (is it mapped often?). It is also not clear how to describe usages. It might need to be more verbose (USAGE_GL_TEXTURE, and etc.)

6. Do we need more pixel data manipulation commands?

UNRESOLVED: Commands such as upload a CPU buffer to a surface, or copy between two surfaces are potentially interesting. We might want them someday.

