Nexus

Surface Compositor

Broadcom
Corporate Headquarters: San Jose, CA

Broadcom Proprietary and Confidential

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.1	8/12/11	Initial draft
0.2	10/21/11	Added format change, 3DTV and cursor
0.3	2/5/13	Update example apps, remove display cache
0.4	1/17/14	Add section on virtual display coordinates and video window control
0.5	3/31/14	Remove tunneled mode
1.0	12/5/14	Reworked section into “Best Performance”

Table of Contents

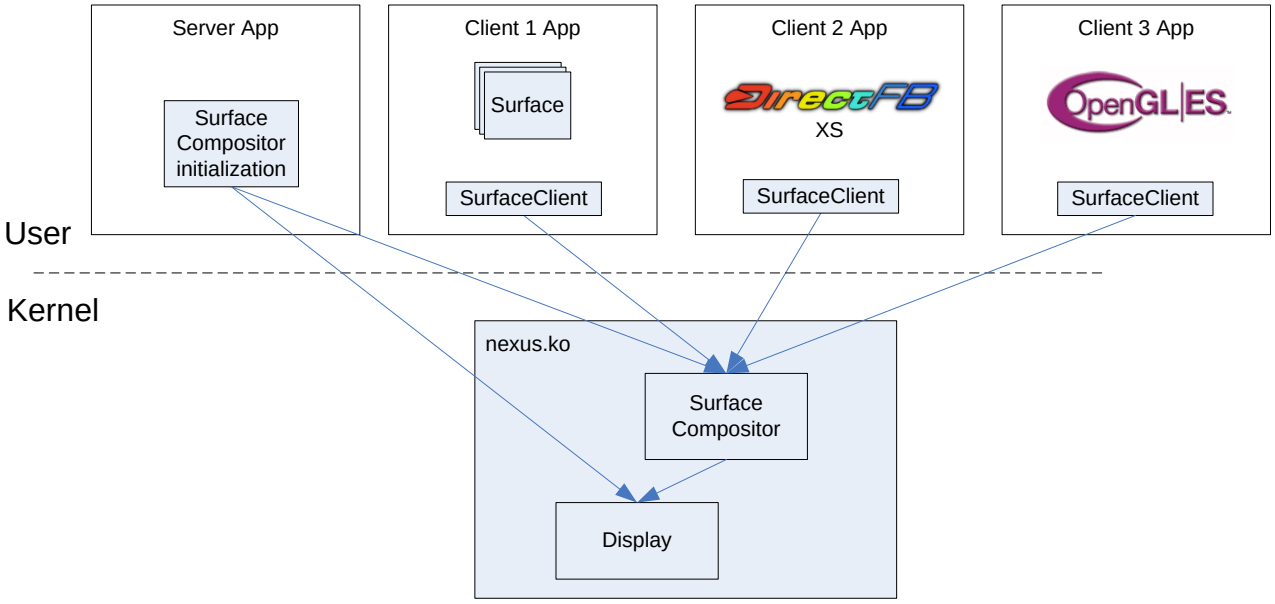
Introduction

SurfaceCompositor is a Nexus module which performs composition of multiple surfaces into a graphics framebuffer.

The design goals of SurfaceCompositor state that it must be:

- Multi-process - multiple clients can be in different processes, whether Nexus is running in Linux kernel mode or user mode.
- Secure – an untrusted client can submit a surface, but cannot compromise the system
- Minimal – the API is only intended to support framebuffer composition. It does not include support for fonts, image rendering, user input devices, display outputs, etc. The surface_compositor module is about 100K in size.
- Clean – the design must ensure no tearing or other rendering race conditions
- High-Performance – achieve optimal performance by running in kernel mode and by supporting multiple usage modes which minimizes copying and blocking.

A typical system looks like this:



Getting Started

The implementation for SurfaceCompositor is found in nexus/modules/surface_compositor. Because it was added in later Nexus releases, it may not be present in your release. Please ask your FAE for an update if you do not see the code.

You can run SurfaceCompositor using the example applications in nexus/examples/multiprocess. They are:

App	Description
blit_server	A sample SurfaceCompositor server. It runs with all of the clients listed below.
blit_client	An incremental-mode client. Does small blits into a 720x480 surface.
animation_client	A multibuffering-mode client. Feeds pipeline of animation frames.

Build the apps as follows:

```
# build nexus examples
cd nexus/examples/multiprocess
make server
make client
# binaries will copy to nexus/bin or $NEXUS_BIN_DIR
```

Run the apps as follows:

```
# from STB console
nexus blit_server

# from STB telnet
export LD_LIBRARY_PATH=.
blit_client –client 0&
animation_client –client 1&
blit_client –client 2&
blit_client –client 3&
```

Integration with Other Graphics Libraries

SurfaceCompositor is designed to be complementary with other graphics libraries like DirectFB, Qt, Microwindows and many others. Customers can use those libraries to render their surfaces, and then submit those rendered surfaces to SurfaceCompositor. But those libraries will stay within each client's process and will not composite the final framebuffer.

This may require stubbing out certain functions that do not apply. For instance, setting display outputs, changing the display format, etc. If the client wants to perform these functions, it could do its own IPC to the server application. SurfaceCompositor does not facilitate that.

The advantage of using SurfaceCompositor is that it is designed for multi-process and for security. It can be difficult to retrofit these capabilities into existing graphics libraries with fixed API's and legacy support requirements.

Multiprocess and Security

SurfaceCompositor uses Nexus' standard multiprocess mechanism. This means it uses Perl scripts to auto-generate ioctl code for Linux kernel-mode and UNIX-domain socket code for Linux user-mode. See nexus/docs/Nexus_MultiProcess.pdf for more information.

In kernel mode, SurfaceCompositor will run in the kernel. The overhead for ioctl's from each client is minimal and performance is optimal. In user mode, SurfaceCompositor will run in the server application's process. The overhead for socket communication from each client is substantial and performance may suffer. If the client is written to minimize the number of IPC calls (including the use of other techniques like Graphics2D packet blit) performance may be acceptable.

The SurfaceCompositor API is divided into two parts: a server API and a client API.

The server API (called SurfaceCompositor) is called by the one server application. The server app opens the display and video windows and passes those resources into SurfaceCompositor. The server also creates each client resource. No client can join the system unless the server has created it first. The server API is only callable by a trusted client.

The client API (called SurfaceClient) is called by each client application. Each client acquires a NEXUS_SurfaceClientHandle which has been created by the server. The client submits or acquires surfaces from the server. The client can be trusted or untrusted. If the client is untrusted, it has no ability to access the server. If the client crashes, all resources will be released and any surface which has visible on the display will be removed.

Display Framebuffer Multi-buffering

The server app tells SurfaceCompositor how many framebuffers to create per display by setting NEXUS_SurfaceCompositorSettings.display[].numFramebuffers.

The default value is two (called double buffering). At least two buffers are required to prevent tearing on the display. Tearing is when a partial graphics update is momentarily visible on the screen.

The application can set the number of framebuffers to one (called single buffering.) In this mode, tearing is inevitable. Even if the client does rendering in an off-screen buffer, the blit to the single framebuffer will be visible as a momentary stair-stepped pattern which is the M2MC's striped pattern. Single buffering can be useful for performance tests when only rendering-time is to be measured.

The server has a background color for each display. If no surface is visible, it will paint the background color. The server also has a clipping algorithm to prevent drawing hidden background or hidden surfaces.

Composition Aggregation

Composition of the framebuffer occurs based on an aggregation of requests from the clients. If any client makes a change, a re-composition of all clients is scheduled. However, any other client change that has been received will also be included in that composition. This means that (for example) 5 client changes will not take 5 display vsyncs. In the general case, they can all be accomplished in one vsync.

There is a start-up delay in composition which may cause near-simultaneous composition requests to be delayed to 2 vsyncs. If (for example) 5 clients make a change, the first change will trigger the re-composition. The other 4 may make it into that first re-composition, or because of minor timing differences, they may have to wait for the next vsync. Once that start up delay occurs, all clients are able to submit changes with equal priority.

The Server's Client API

The server has an API to control each client.

The server must create each client that can join the compositor.

The server controls the size, position, z-order and alpha blending of each top-level surface for each client. This is necessary to ensure a secure system. SurfaceCompositor does not allow a client to take over the display. It can only work within the box that the server allows.

If the client application wants to specify its size and location, it can communicate with the server via application IPC. See nexus/nxclient and its NxClient_SetSurfaceClientComposition function for an example of this.

Usage Modes

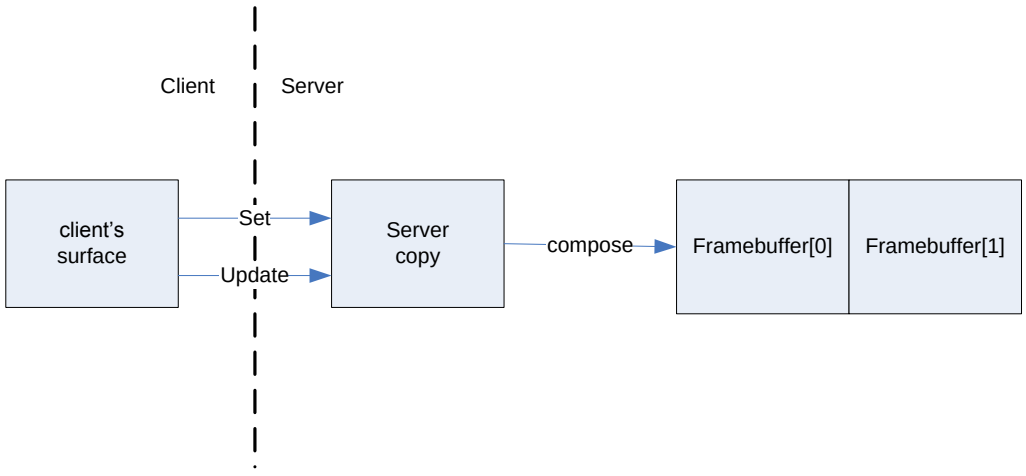
The client API (NEXUS_SurfaceClient) supports two usage modes:

- Incremental
- Multi-buffered

Each mode is designed to be optimal for a certain usage mode. A client can switch between these three modes as needed. Only one mode is active at a time.

Incremental Mode (SetSurface)

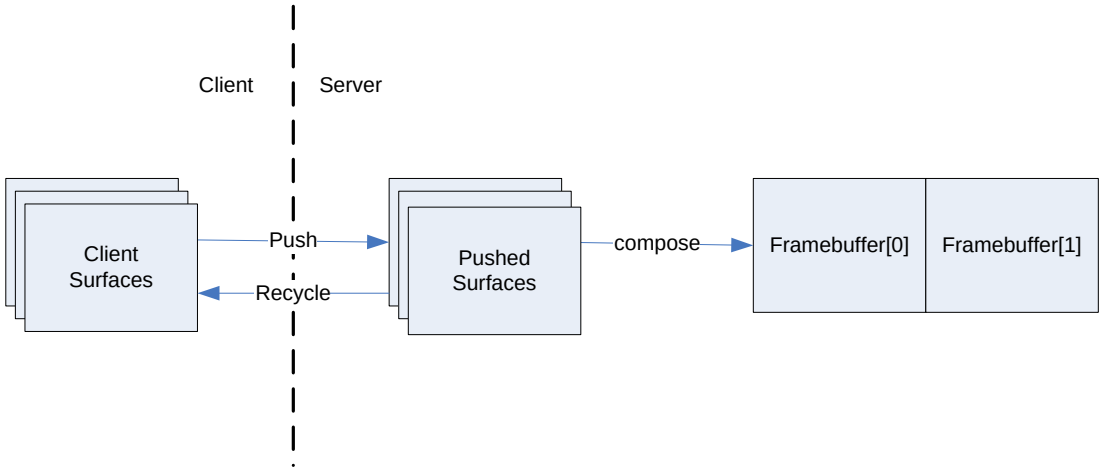
The key API's are NEXUS_SurfaceClient_SetSurface and UpdateSurface.



After calling SetSurface, the server will make a full copy of the client's surface. When the client calls UpdateSurface, the server will copy the updated portion. The server will compose the framebuffers whenever the client sets or updates the surface. It is called incremental mode because the client does not need to re-render the entire surface on every Update.

Multi-buffered Mode (PushSurface)

The key API's are NEXUS_SurfaceClient_PushSurface and RecycleSurface.



This mode is useful if the client is produces fully rendered surfaces[1]. This could be used with an OpenGL client in pipelined mode. Instead of OpenGL submitting the completed frames directly to the display, it would submit them to SurfaceCompositor. It should not know the difference.

The client's server-side state machine keeps a FIFO of pushed surfaces. If that FIFO has a depth greater than one, each display vsync will consume a new surface and the client will drive new composition. But as soon as the FIFO reaches one, the client will simply repeat the last remaining surface whenever a composition is needed. The client will not drive new composition itself. It will get repeated only if another client is driving new composition. The FIFO depth will fall to zero if the client is cleared (NEXUS_SurfaceClient_Clear).

The Push API has an optional update rectangle. The client can report the portion of the surface which differs from the previously submitted surface. The server can use this information to optimize framebuffer composition.

[1] This does not mean that the client must completely re-render each frame. It could keep track of what it has rendered in each surface in the queue and only update those regions that differ. SurfaceCompositor would have no involvement in that update algorithm.

Best Performance

Monitoring frame rate

You can measure the SurfaceCompositor composition frame rate using the /proc interface. In kernel mode, run "cat /proc/brcm/surface_compositor". In user mode, run "echo surface_compositor >/proc/bcmdriver/debug". You will see something like:

```
00:16.551 surface_comp: SurfaceCompositor 0x135038
00:16.551 surface_comp: display 0: 0x10c678, 1920x1080, 232 compositions
00:16.551 surface_comp: display 1: 0x107bc0, 720x480, 230 compositions
00:16.551 surface_comp: client 0x856128f8: 0,0,1920,1080; visible y; zorder 0
00:16.551 surface_comp:   pushed 3, recycled 0, total 228

00:21.556 surface_comp: SurfaceCompositor 0x135038
00:21.556 surface_comp: display 0: 0x10c678, 1920x1080, 532 compositions
00:21.556 surface_comp: display 1: 0x107bc0, 720x480, 530 compositions
00:21.556 surface_comp: client 0x856128f8: 0,0,1920,1080; visible y; zorder 0
00:21.556 surface_comp:   pushed 3, recycled 0, total 528
```

You can see on the HD display that 300 compositions were done in 5 seconds. That's 60 frames/second.

Recycled Callback

After setting, updating or pushing a surface, the application must learn from SurfaceCompositor that the surface is available again.

If you are using SetSurface, the recycled callback fires when the surface has been copied. Because there's no FIFO, no function call is required.

If you are using PushSurface, the recycled callback fires when a surface is no longer being displayed. The application must call NEXUS_SurfaceClient_RecycleSurface to get the surface. Proper management of the FIFO is critical to getting optimal performance, so this is covered in detail below.

Call RecycleSurface and wait for the recycled callback only when you must

After calling PushSurface, it's tempting to think you should wait for the recycled callback and call RecycleSurface. You should not. For best performance (and certainly for 60fps maximum performance) you must build and maintain a queue of at least 2 surfaces at all times in

SurfaceCompositor (see the "pushed #" count in /proc output). If the queue drops to 1, then SurfaceCompositor will be repeating frames on the next vsync and your composition frame rate will drop. It may look like SurfaceCompositor is suffering from slow M2MC or MEMC performance, when it's really poor queue management.

Instead, follow these two rules:

- Only call NEXUS_SurfaceClient_RecycleSurface when you need a surface and don't have one. If you wait until you actually need it, it increases the chances that it's already available.
- Only wait for the recycled callback when NEXUS_SurfaceClient_RecycleSurface returns with no surfaces and you still need one. Then you have to wait. But if you have at least 3 surfaces to work with, SurfaceCompositor should always have a queue of 2.

The following are various tips and things to avoid.

Don't use Push/Recycle with only two surfaces

Unless your render time is super short, you won't be able to hit 60 fps. You need 3 surfaces to allow for double buffering at the display and one surface in-flight between surface compositor and the app.

Don't wait for the recycled event after NEXUS_SurfaceClient_PushSurface

The first problem is that if you've only pushed one surface, the recycled event will never come. If you've pushed two, then the recycled event will come, but only when the SurfaceCompositor queue drops to one, which will lead to a repeat and lower frame rate.

Don't always wait for the recycled event before calling RecycleSurface

The problem is that if there's a surface already available, you're waiting for no reason. It's possible the wait will always return right away, but why bother. Only wait when you didn't get what you need.

Don't count callbacks

After receiving a recycled callback, you must call NEXUS_SurfaceClient_RecycleSurface to learn which surface(s) have actually been returned. If you are using more than two surfaces (which is recommended), there is no way to know which are recycled by the callback alone.

Be aware that callbacks are like interrupts – two can be collapsed into one when the system is busy. The following illustrates this:

> *Your app has an animation which is a circular queue of 5 surfaces. You push surface 0, 1, 2, 3, and 4, and then wait for the recycled callback before cycling back and pushing 0 again. How many recycled callbacks will you receive? Depending on the performance of the system and any delay between your pushes, you may receive anywhere between 1 and 4 callbacks. You must call NEXUS_SurfaceClient_RecycleSurface and see which surfaces are recycled.*

Decrease your framebuffer size

If you are M2MC or MEMC bandwidth limited, if you decrease the framebuffer size, it will increase performance. The trade-off is graphics quality due to GFD upscaling. Consider 1280x720 even if your output is 1920x1080. Or consider 960x1080, has GFD horizontal upscale for 720p and 1080p, but no GFD vertical upscale. Don't assume that your framebuffer must be 1920x1080.

Display SD path if bandwith limited

SurfaceCompositor is designed to blit the SD path without delaying the HD rendering path. It recycles client surfaces after the HD framebuffer is composed, and then renders the SD framebuffer from the HD framebuffer. However, if you are M2MC or MEMC bandwidth limited, even the SD path could slow down the system. If you disable SD, that bandwidth is released.

The only valid use of the displayed callback

If the recycled callback should be used for managing surface life cycle, why have the displayed callback at all?

The only use case is if you are using set/update and want to pace your updates based on the framerate of the display. If you only use recycled, you will be able to set/update as fast as the blitter can go. This may be much faster than the display and you may miss displaying some frames. If you wait for the displayed callback, you will match the display's frame rate.

Note that even in this case, you may still want to use the recycled callback in order to start client side composition as soon as possible. Recycled will be called first, when the server's copy blit is done. Displayed will be called later, when the server's copy finally makes it to the display.

Virtual Display Coordinates

When positioning a SurfaceClient on the screen, or positioning a child window within a parent, the caller has two options for coordinate systems: native coordinates or virtual coordinates.

The easiest option is virtual coordinates. When setting NEXUS_SurfaceComposition.position, the caller also specifies NEXUS_SurfaceComposition.virtualDisplay. The units for 'position' are 'virtualDisplay', regardless of the actual display coordinates. For example, if virtualDisplay is 1920x1080, then a position rectangle of (960,0,960,540) will always be in the upper right quadrant, regardless of current display resolution.

For native coordinates, the units of 'position' can vary:

For a top-level window, it is the clipped framebuffer coordinates. A typical system may use a 1280x720 framebuffer for both 720p and 1080p/i resolutions, but it will need to clip this to 720x480 for 480p/i. A client using native coordinates would need to know the current resolution and clipping in order to position itself correctly.

For a child window, for example a video window, actual coordinates are whatever the dimensions of the last submitted surface.

Because actual coordinates may vary in ways the client cannot predict, we recommend always specifying virtualDisplay coordinates.

Video Window Control

SurfaceCompositor has an option of resizing video windows along with graphics. If the server connects the video window to a client's NEXUS_SurfaceCompositorClientSettings, then the client can call NEXUS_SurfaceClient_AcquireVideoWindow and obtain a handle for a child window. The child window can be positioned relative to the parent graphics window. This allows an application to render a GUI with a video window placed within that GUI, and if someone outside of the client repositions or resizes the top-level GUI, the video window will be moved and scaled with it.

If the server does not connect the video window to SurfaceCompositor, then graphics and video are controlled separately.

Publishing

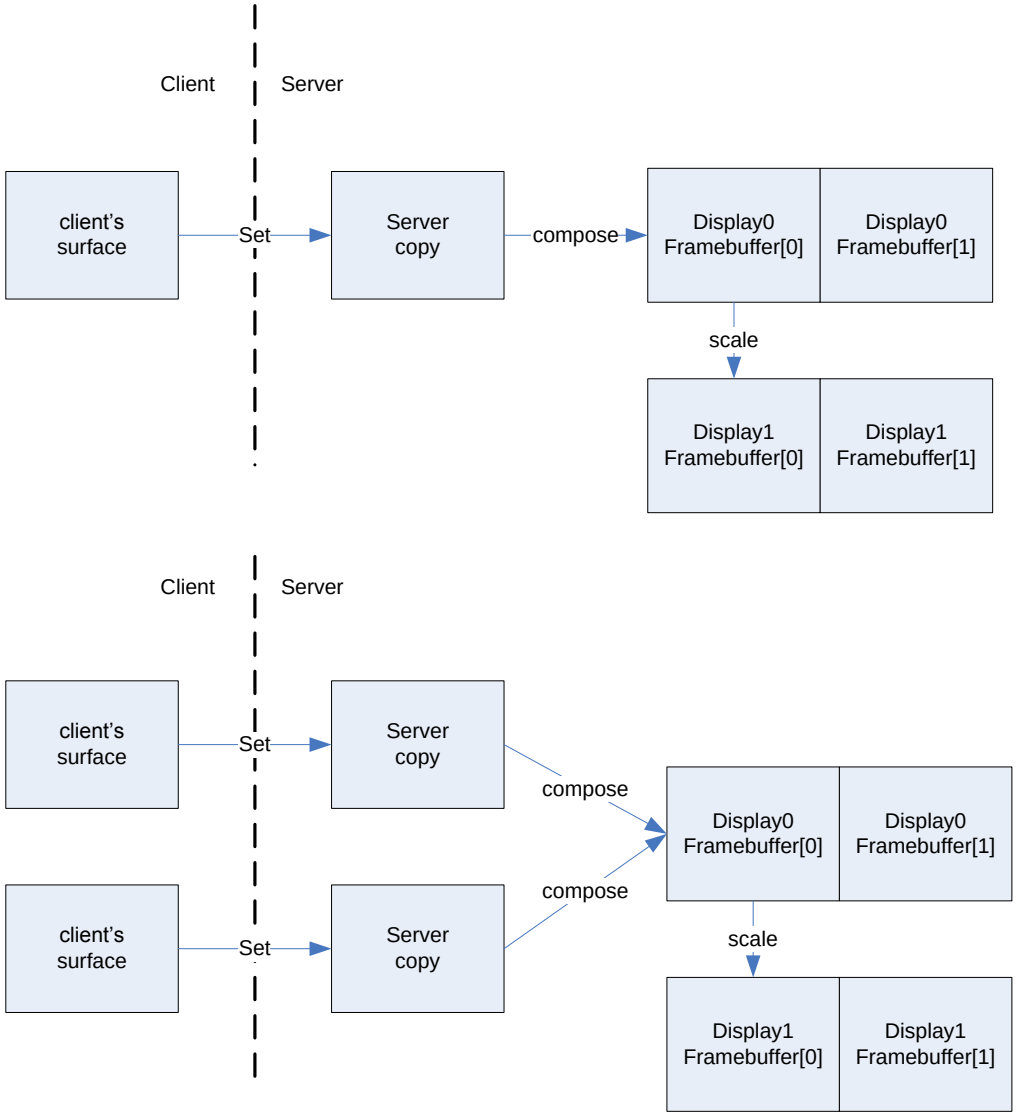
A client can publish the last submitted surface by calling NEXUS_SurfaceClient_PublishSurface. This works in incremental and multi-buffered modes.

Publishing a surface tells SurfaceCompositor that it should re-render the surface on every vsync, without any further notification from the client.

This mode is not recommended because it cannot be used without tearing. However, it has been added for backward compatibility with certain graphics libraries that expect the functionality.

Multiple Displays

SurfaceCompositor supports rendering to multiple displays, often referred to as “HD/SD simul” mode. In the composition stage, framebuffers for the main display are rendered first. Then, the secondary display is rendered from the primary display.

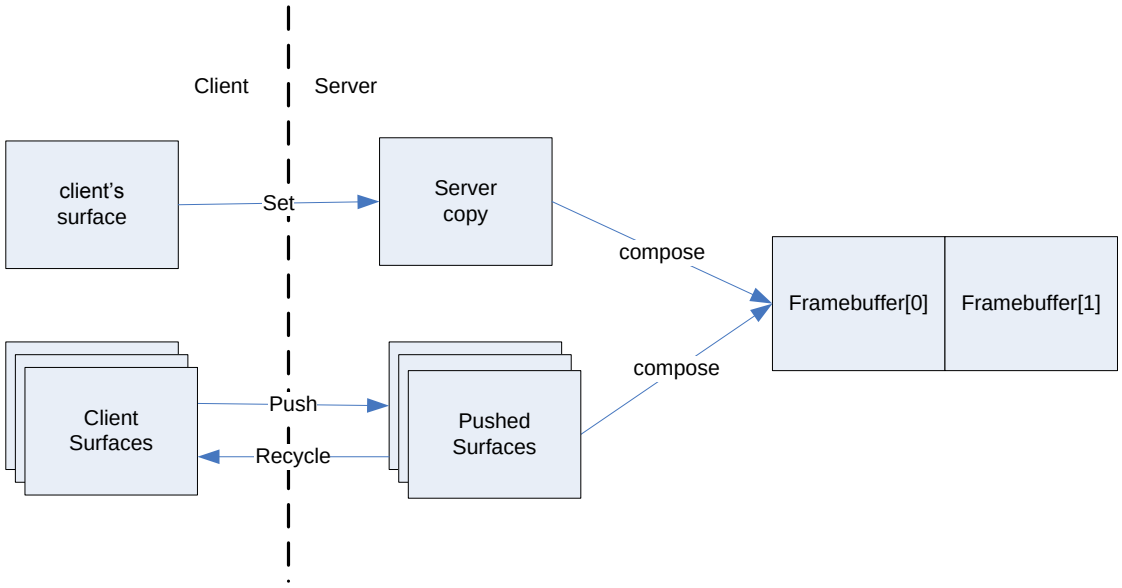


By doing a secondary rendering stage, we allow the rendering speed to be bound only by the main display. This results in a more optimal system.

The downside is that a double-scale may happen on the secondary display. We believe the graphics quality will be acceptable.

Multi-client composition

The server can composite multiple clients, each working in a different mode, into the same framebuffer. If the clients are in incremental or multi-buffering mode, composition is simply a matter of copying all of the clients in correct zorder.



Stereoscopic TV (3DTV)

SurfaceCompositor supports stereoscopic TV's (aka 3DTV). For 65nm silicon, 3DTV requires software support and an extra blit. For 40nm silicon, 3DTV is supported by hardware and the extra blit may be avoided. The API is similar for both types of silicon.

There are both server-side and client-side API's. For 40nm silicon, you set the 3D video orientation in the NEXUS_DisplaySettings and so Nexus SurfaceCompositor simply reads that back and determines its mode. For 65nm silicon, there are no 3DTV display settings, so the app must tell surface compositor to render as half-res 3DTV. The API's are as follows:

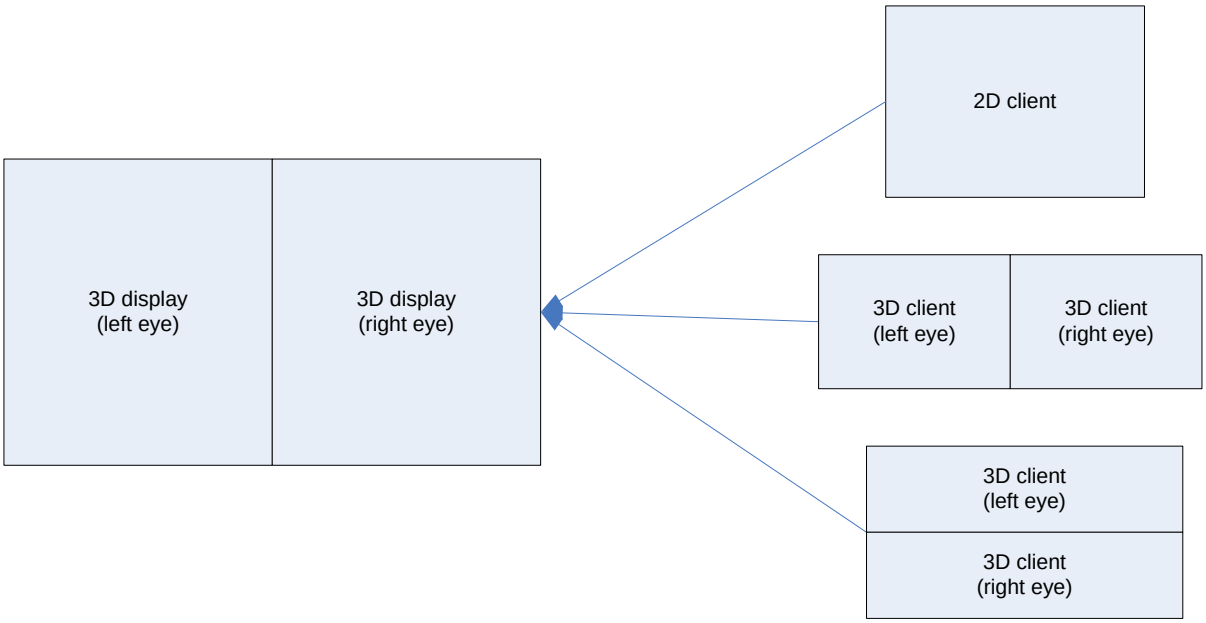
- For 40nm silicon, set NEXUS_DisplaySettings.display3DSettings.orientation.
- For 65nm silicon, set NEXUS_SurfaceCompositorSettings.display[].display3DSettings.overrideOrientation.
- Set the framebuffer's z-offset through NEXUS_GraphicsSettings.graphics3DSettings.rightViewOffset.

NOTE: NEXUS_GraphicsSettings is tunneled through surface compositor using NEXUS_SurfaceCompositorSettings.display[].graphicsSettings.

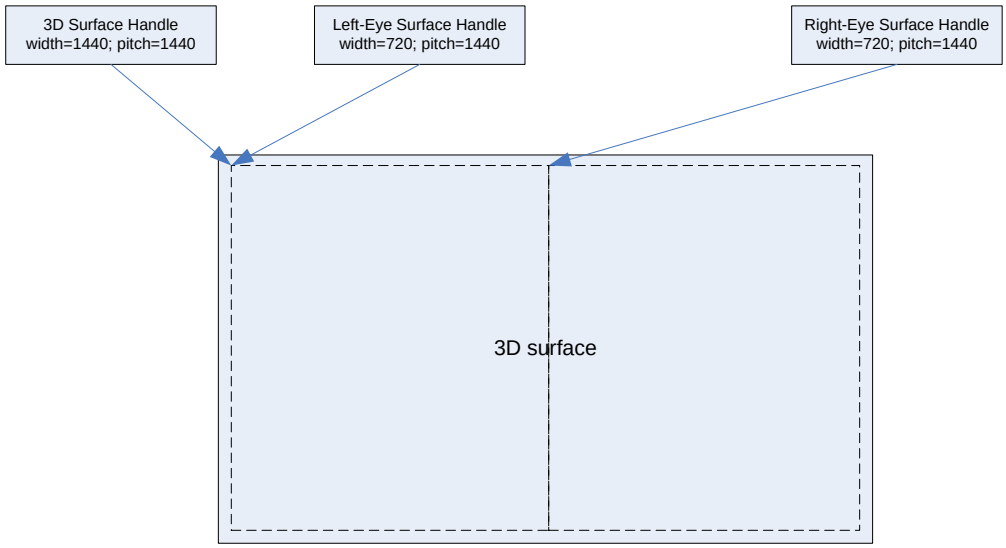
The server also determines the z-depth of each client by setting NEXUS_SurfaceCompositorClientSettings.composition.rightViewOffset.

The client has the ability to render itself in a 2D or 3D mode. In a 3D mode, the client renders separate right and left eye images. In 2D mode, the client renders a single image. There is a mapping between every combination of 2D/3D clients and every combination of 2D/3D displays. See the header file comments for NEXUS_SurfaceClientSettings.orientation.

The following shows some possible combinations:



The client always submits a single surface, even if it is rendering a separate left and right eye. If you want one 720x480 surface for left eye and one 720x480 surface for right eye, then you must create a 1440x480 surface and render left eye in the left side, right eye in the right side. If your application needs separate surface handles for left and right eye, you can create those handles, but map to the single stereoscopic surface. Your memory layout will look like this:



Display Format Change

Changing the display format requires some complex internal and external synchronization. Our design minimizes the impact on clients.

The steps are as follows:

- Server app disables the server by setting NEXUS_SurfaceCompositorSettings.enabled = false, then waits for inactive callback.
- Once enabled is set to false, the server will stop submitting new work to the blitter and will stop submitting framebuffers to the display and will disable graphics. Once all checkpoint callbacks and framebuffer callbacks have been received, the surface compositor is inactive and fires the inactive callback.
    - During this time, all client calls will succeed, even though no action will be taken internally. The surface compositor will also simulate a “displayed” callback. This allows clients to continue operating normally.
- When the server receives the inactive callback, it can change the display settings including format, framebuffer size, etc.
- Once the display format is changed, the server can update NEXUS_SurfaceCompositorDisplaySettings with new framebuffer sizes and set enabled = true.
    - If NEXUS_SurfaceCompositorDisplaySettings are changed when the surface compositor is not in a inactive state, the call will fail.
    - At this point, framebuffers may be destroyed and recreated.
- When the surface compositor is enabled, all clients will receive a displayStateChanged callback.

This supports display format change on the server. If the server app wants to expose display format change to clients, it must use application IPC.

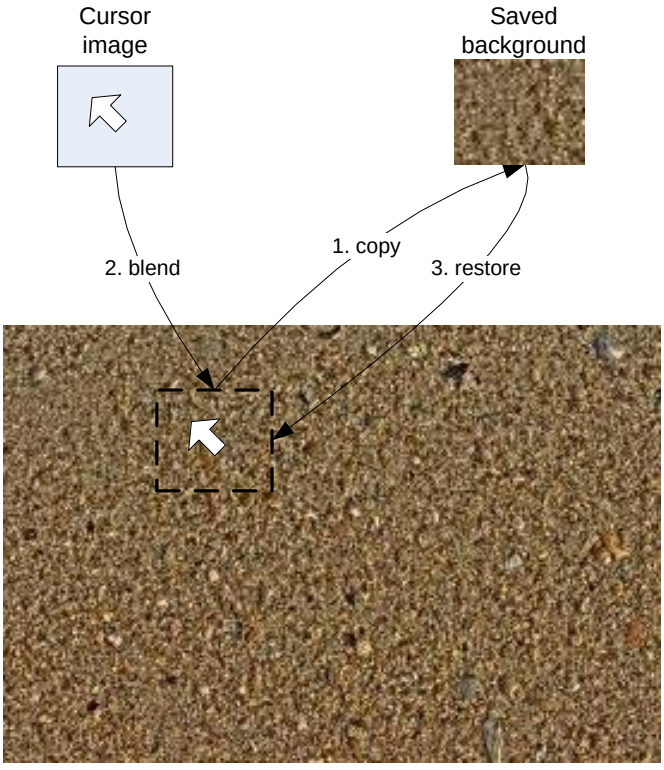
Cursor Support

Surface Compositor supports a cursor (a moving pointer object) on top of all composited surfaces.

For most silicon, this is a software-rendered cursor. For silicon where a hardware cursor is available, it is preferred.

The cursor rendering algorithm works as follows:

- After rendering the framebuffer, the contents of the location of the cursor is saved.
- The cursor is rendered into that location.
- If the cursor is moved before a new framebuffer is rendered, the old location of the cursor is restored, the new location is saved and the cursor is drawn.



The position of the cursor is determined by the server. Any connection to a pointing device like a mouse must be done by the server application.

The default image of the cursor is set by the server. The client can set a local override for its cursor.

Other Usage Modes

Multiple graphics feeders

If your silicon has more than one graphics feeder (GFD) per display, we recommend that each GFD have its own instance of SurfaceCompositor. Any blending interaction between the GFD's would be done by the server application.

Single-process applications

SurfaceCompositor is designed to make multi-process GUI's possible, but it can be used in a single process context as well. You can create and use multiple clients in a single process, from one or more threads.

Dynamically-created clients

The server must explicitly creating clients before they can be acquired and used by client applications. NxClient was designed to provide dynamic creation of clients by using application IPC implemented above Nexus. See nexus/nxclient/docs/NxClient.pdf.

Dynamically-sized clients

The server must explicitly set the position and size of client surfaces on the display. If your system wants to allow the client to dynamically position itself on the screen, there are two options: implement application IPC above Nexus, or create a full-screen client and have the client use an alpha hole for the region it is not using. NxClient was designed to provide the first option. See nexus/nxclient/docs/NxClient.pdf.