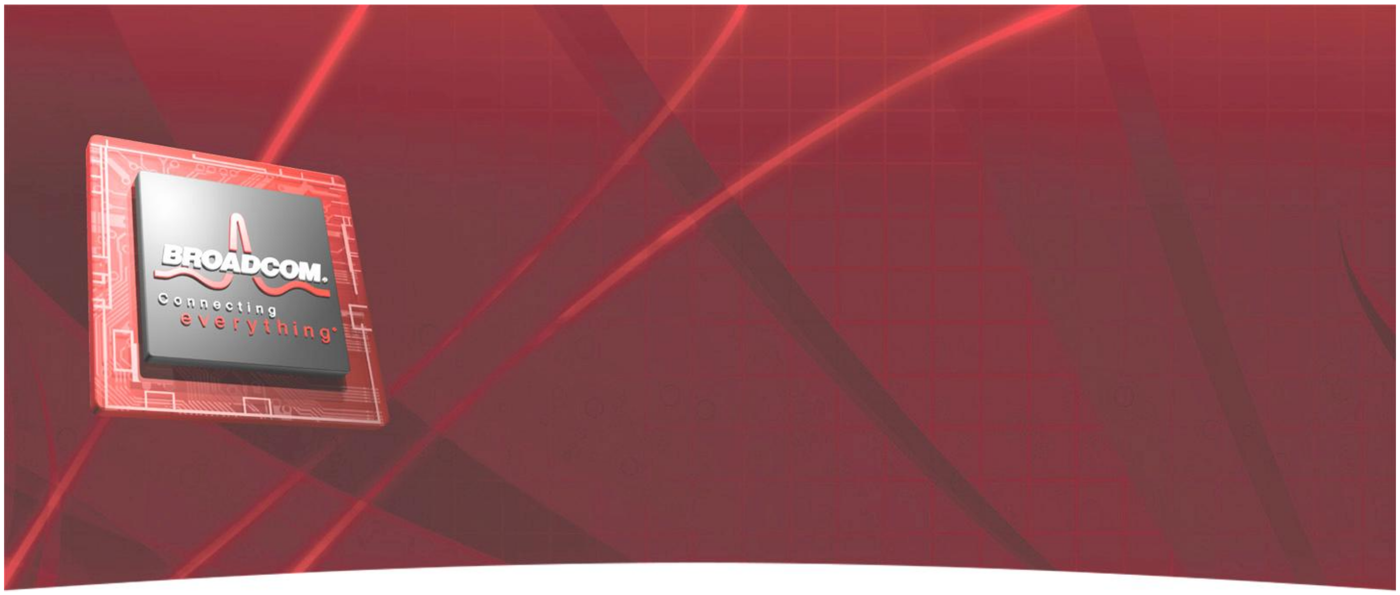


Nexus

Surface Compositor

Broadcom Corporation
5300 California Avenue
Irvine, California, USA 92617
Phone: 949-926-5000
Fax: 949-926-5203

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

Table of Contents

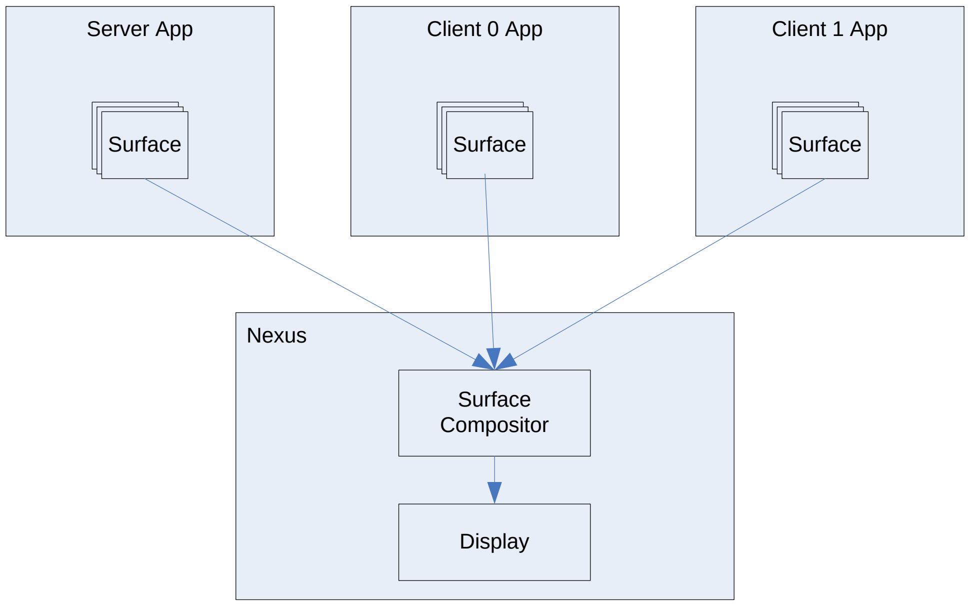
Introduction

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.
- Clean – the design must ensure no tearing or other rendering race conditions
- High-Performance – achieve optimal performance 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.
tunneled_client	A tunneled-mode client. Does small fills to the framebuffer.

Build the apps as follows:

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

Run the apps as follows:

```
# from STB console
nexus blit_server

# from STB telnet
nexus.client blit_client 0&
nexus.client animation_client 1&
nexus.client blit_client 2&
nexus.client blit_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 application may want to set the number of framebuffers to three or four to support multi-client tunneled mode. This is discussed in detail in the section on multi-client compositing.

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.

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 its own application IPC. See nexus/examples/multiprocess/refsw_server.c for an example of application IPC that does this.

Usage Modes

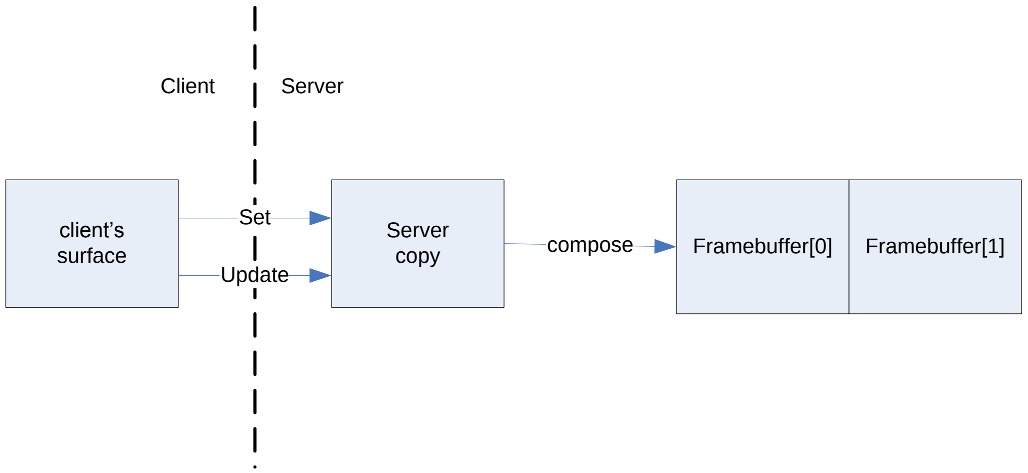
The client API (NEXUS_SurfaceClient) supports three usage modes:

- Incremental
- Multi-buffered
- Tunneled

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

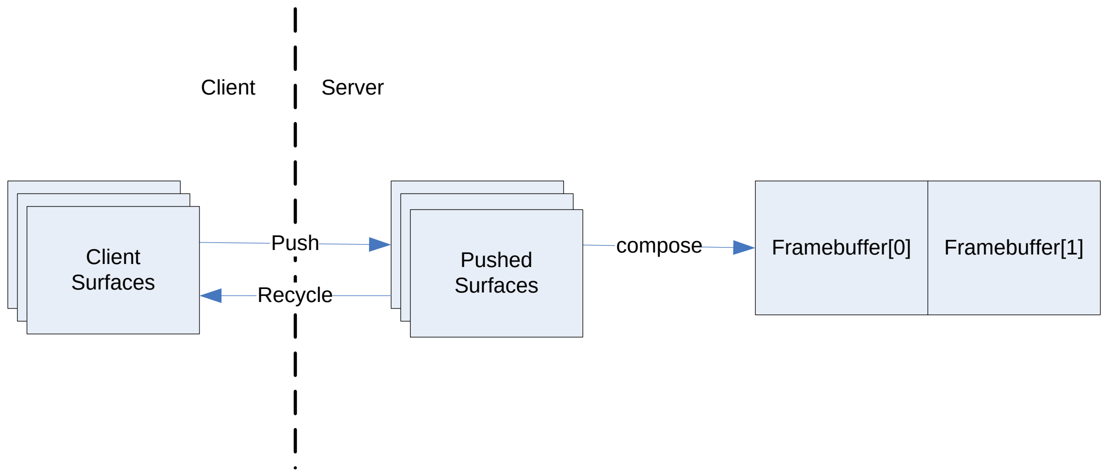
The key API's are NEXUS_SurfaceClient_SetSurface and UpdateSurface.



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

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.

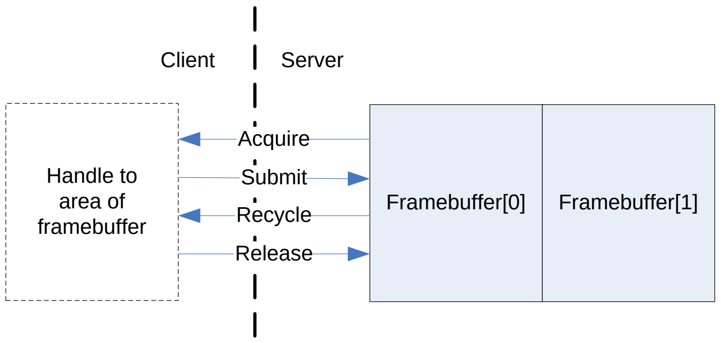
The server will consume one surface per vsync. It will repeat the last surface until a new surface arrives or the client is cleared (NEXUS_SurfaceClient_Clear).

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.

Tunneled Mode

The key API's are NEXUS_SurfaceClient_AcquireTunneledSurface, ReleaseTunneledSurface and SubmitTunneledSurface and RecycleSurface.



In tunneled mode the client does not allocate surfaces. Instead, it acquires surfaces which have been allocated by the server. This allows the client to get direct access to the framebuffer.[2] This results in optimal performance. However, the client is under certain restrictions.

The client must be trusted. If the client is only given access to a region of the framebuffer[3], there is nothing preventing that client from going outside of that region.

Only one tunneled client is allowed per server[4]. Also, the tunnel gives direct access to only one display. If you have more than one display, there must be a copy to the secondary displays.
In double buffered mode with a single client, the framebuffer is flipped on each submit. In single-buffered mode, the submit is a no-op; any update is immediately visible and tearing will result.

Multiple clients are allowed in tunneled mode, but it requires more complex processing and performance will degrade. This is covered in the next section.

[2] Theoretically, tunneled mode is not guaranteed to return the framebuffer. The client would not know. However, direct access to the framebuffer is the usual goal and the initial implementation.
[3] The NEXUS_SurfaceHandle returned to the client could be different from the NEXUS_SurfaceHandle for the framebuffer. It could point to the same memory, but have a different offset and size.
[4] Theoretically, SurfaceCompositor could allow multiple non-overlapping tunneled clients, but it would be complex to support all usage modes.

Publishing

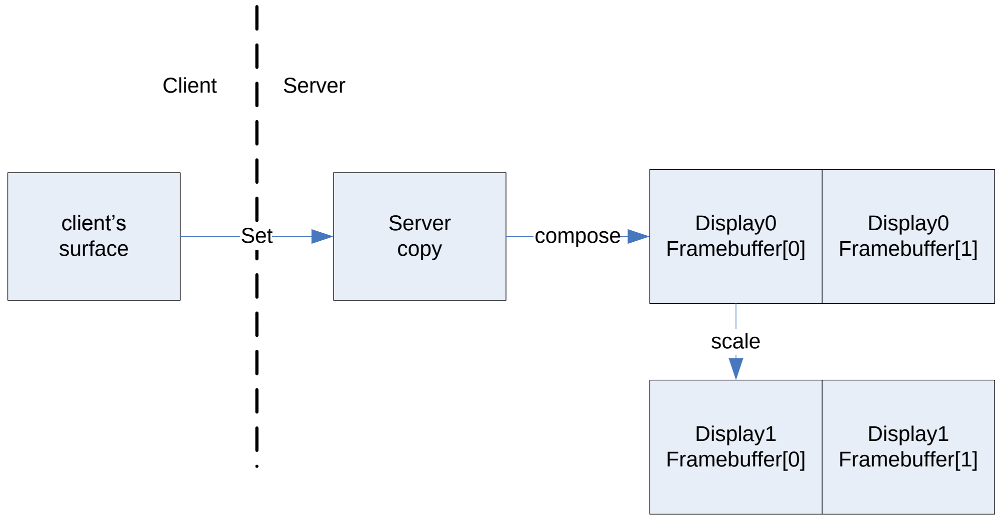
A client can publish the last submitted surface by calling NEXUS_SurfaceClient_PublishSurface. This works in all three modes (incremental, multi-buffered and tunneled).

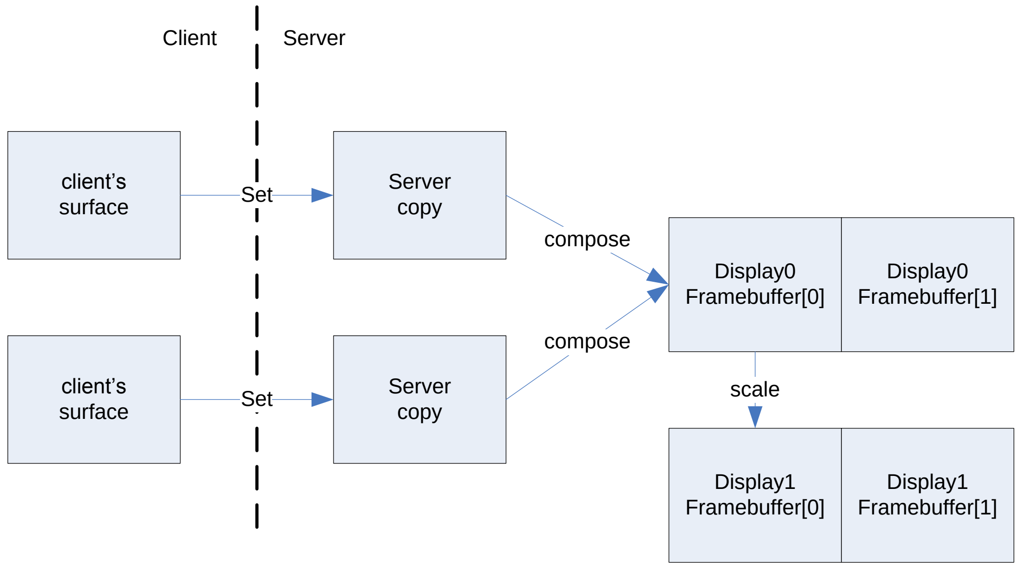
Publishing a surface tells SurfaceCompositor that it should re-render the surface on every vsync, without any further notification from the client.

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.



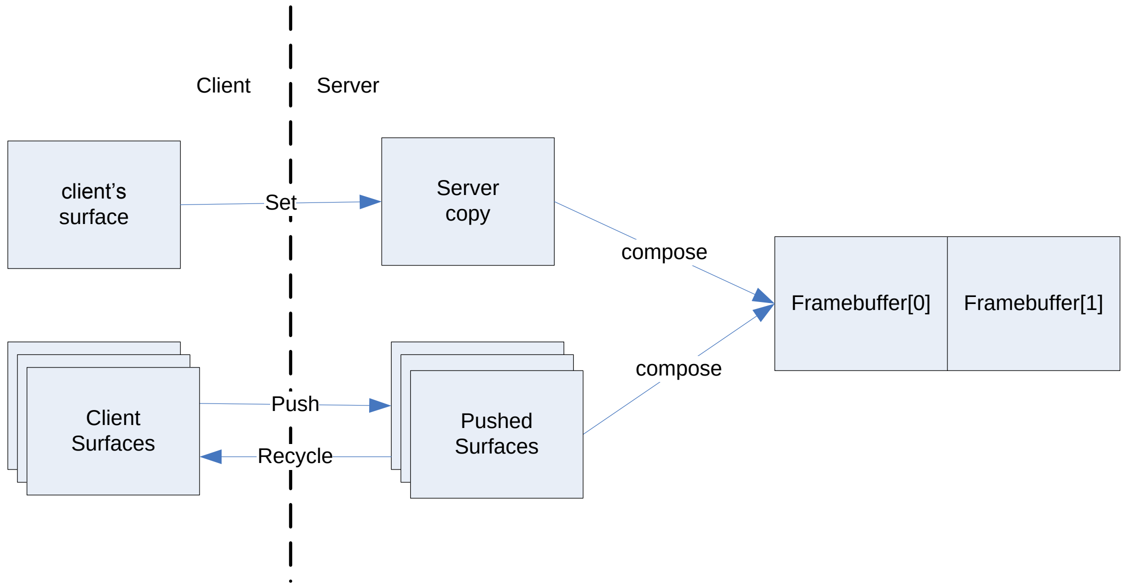


By doing a secondary rendering stage, we allow the rendering speed to be bound only by the main display. This results in a more optimal system.

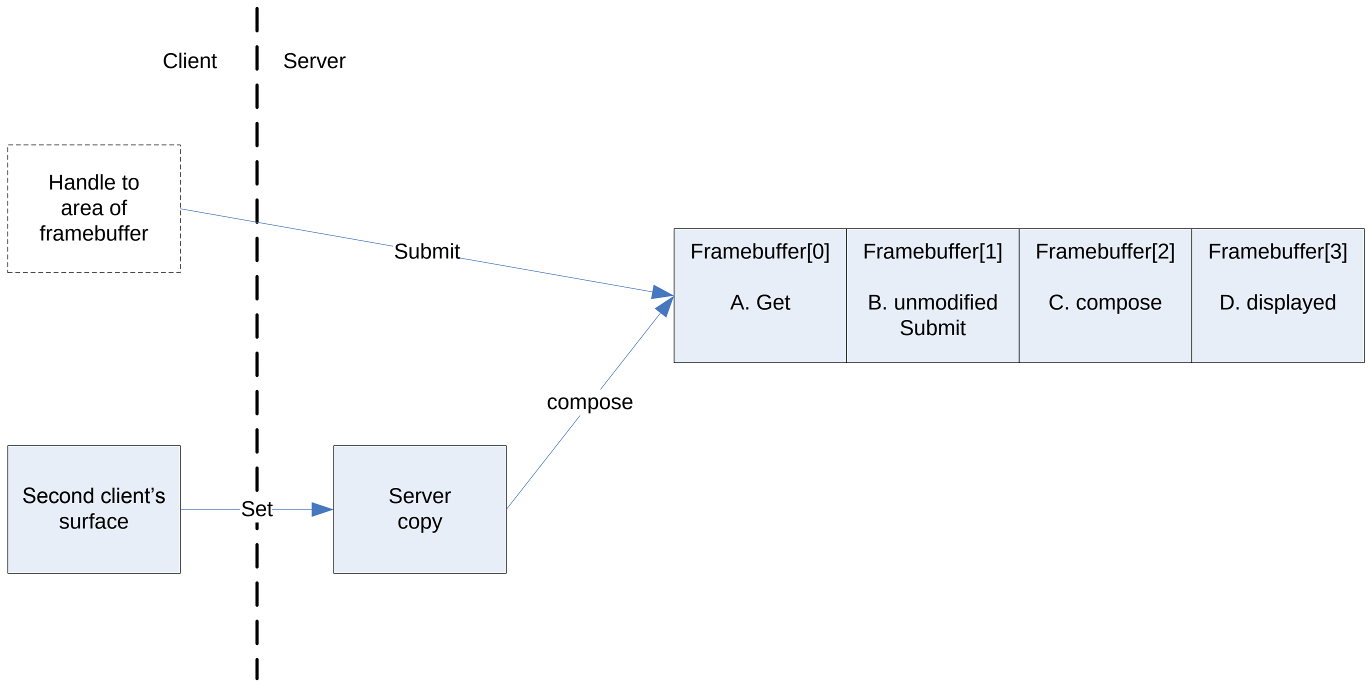
The downside is that a double-scale may happen on the secondary display. We believe the graphics quality will be acceptable.

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.



If one client is in tunneled mode, the server will require three or four framebuffers. The general case is that the non-tunneled client may overlap the tunneled client. This requires four framebuffers as follows:

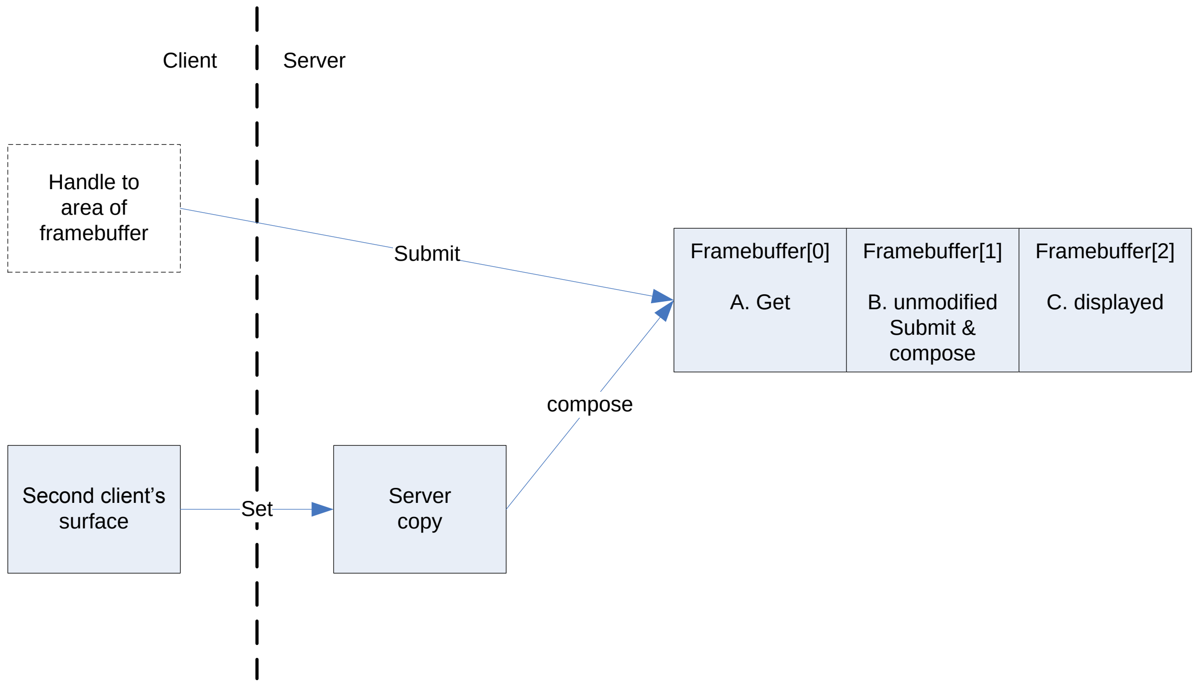

Client
Server
Handle to area of framebuffer
Submit
Framebuffer[0] A. Get
Framebuffer[1] B. unmodified Submit
Framebuffer[2] C. compose
Framebuffer[3] D. displayed
compose
Second client's surface
Set
Server copy

This requires no more memory and no more copies than if each client was in incremental mode. However, the performance of the tunneled client will go down when the second client is visible. But client does not need to be aware of the second client.

In this mode, two of the framebuffers (A and B) are acquired by the tunneled client and two framebuffers (C and D) are used for multi-buffered display.

When the second client disappears, SurfaceCompositor can seamlessly switch back to sending A and B to the display.

If the second client does not overlap with the tunneled client's region, a triple-buffered mode is possible.



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:

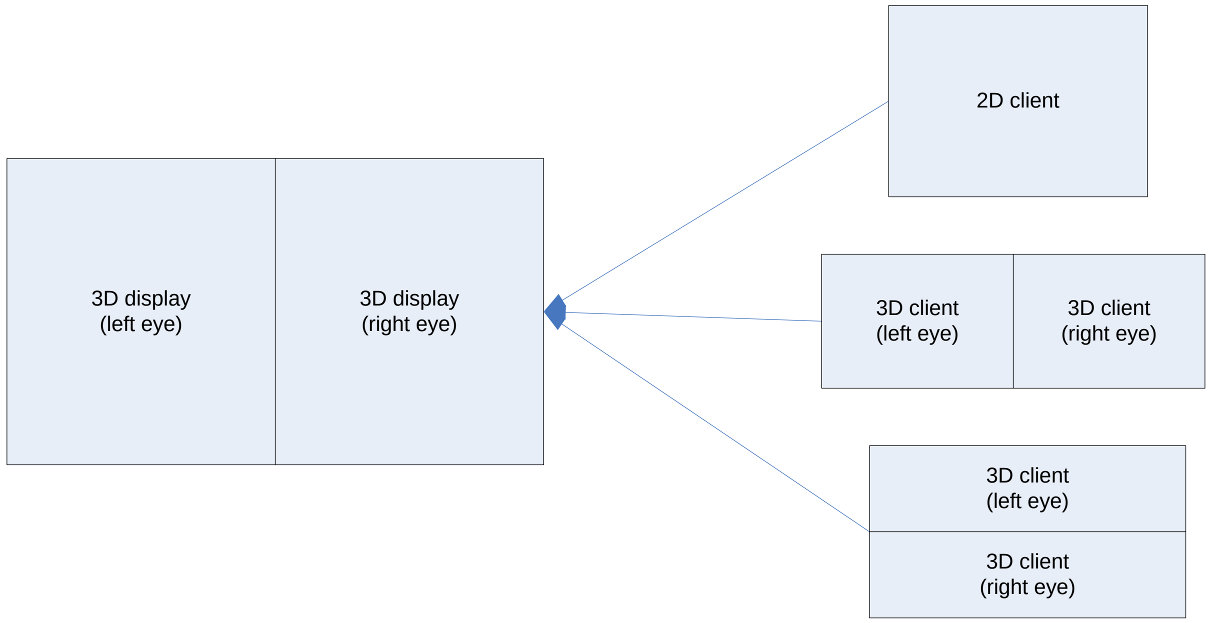
- For 40nm silicon, set NEXUS_DisplaySettings.display3DSettings.orientation.
- For 65nm silicon, set NEXUS_SurfaceCompositorSettings.display[].display3DSettings.overrideOrientation.
- Set the framebuffer's z-offset through NEXUS_GraphicsSettings.graphics3DSettings.rightViewOffset.

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

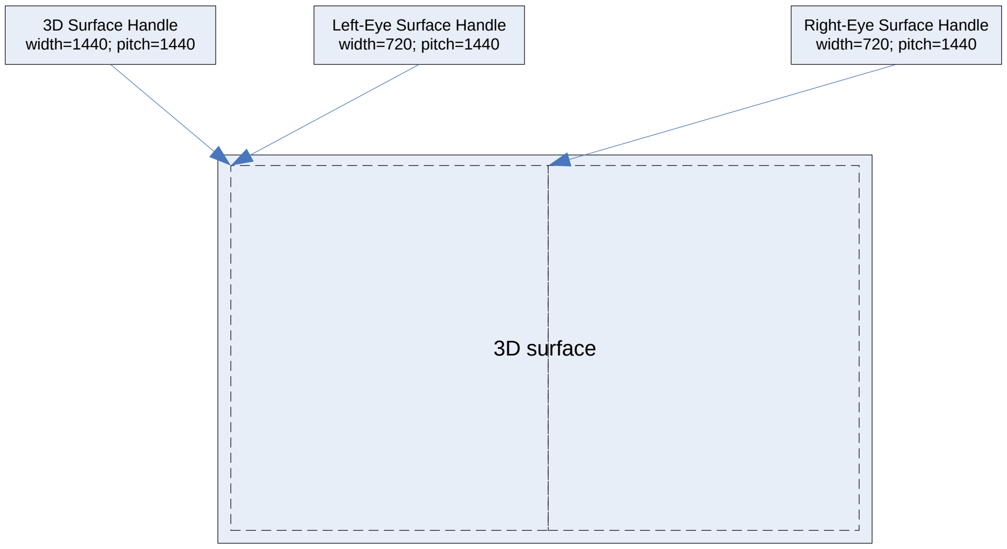
The server also determines the z-depth of each client by setting NEXUS_SurfaceCompositorClientSettings.composition.rightViewOffset.

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 attempts to minimize the impact on clients. Non-tunneled clients should see no change in behavior as the format changes. Tunneled clients will need to release tunneled surfaces.

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.
    - The server will also fire the NEXUS_SurfaceClientSettings.displayStatusChanged callback on all clients. If you are a non-tunneled client, you can ignore the callback. If you are a tunneled client, you must release all tunneled surfaces.
- When the server receives the inactive callback, it can change the display settings including format, framebuffer size, etc.
- The server should wait for clients to release all tunneled surfaces. It can poll GetStatus as long as it wants. If a client does not release its tunneled surfaces, the server can unregister the client.
- 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. If a tunneled client has not released its tunneled surfaces, the call will fail.
- When the surface compositor is enabled, all clients will receive a displayStateChanged callback. Tunneled clients can reacquire.

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.

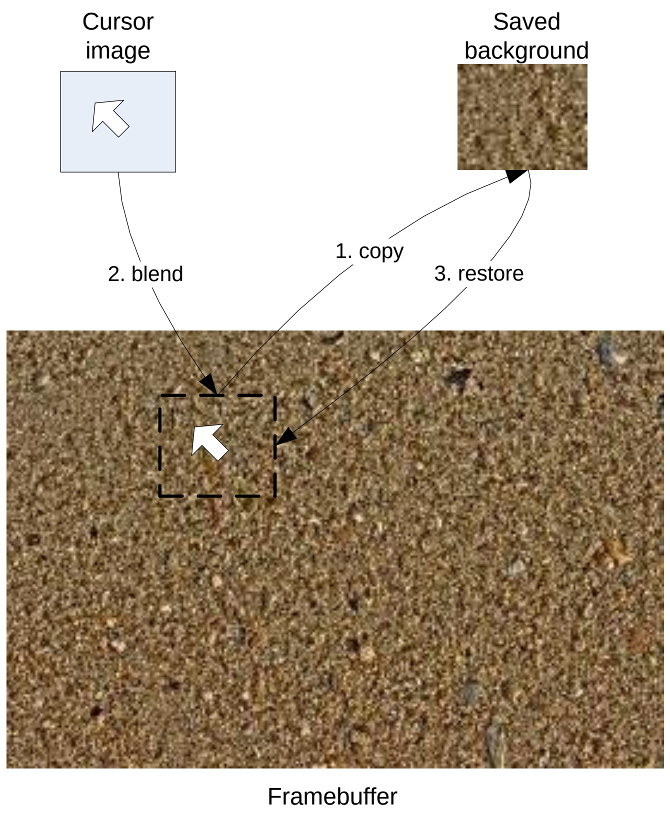
There are two cursor options when using tunneled mode: destructive and non-destructive.

The cursor must always be written into the framebuffer. If you have a tunneled client, that framebuffer may have been submitted by the client. In destructive mode, the cursor is written into a surface that will be returned to the tunneled client. In non-destructive mode, a copy is made so that the tunneled client never sees the cursor.

If there is no tunneled client, no extra copy is required.

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.



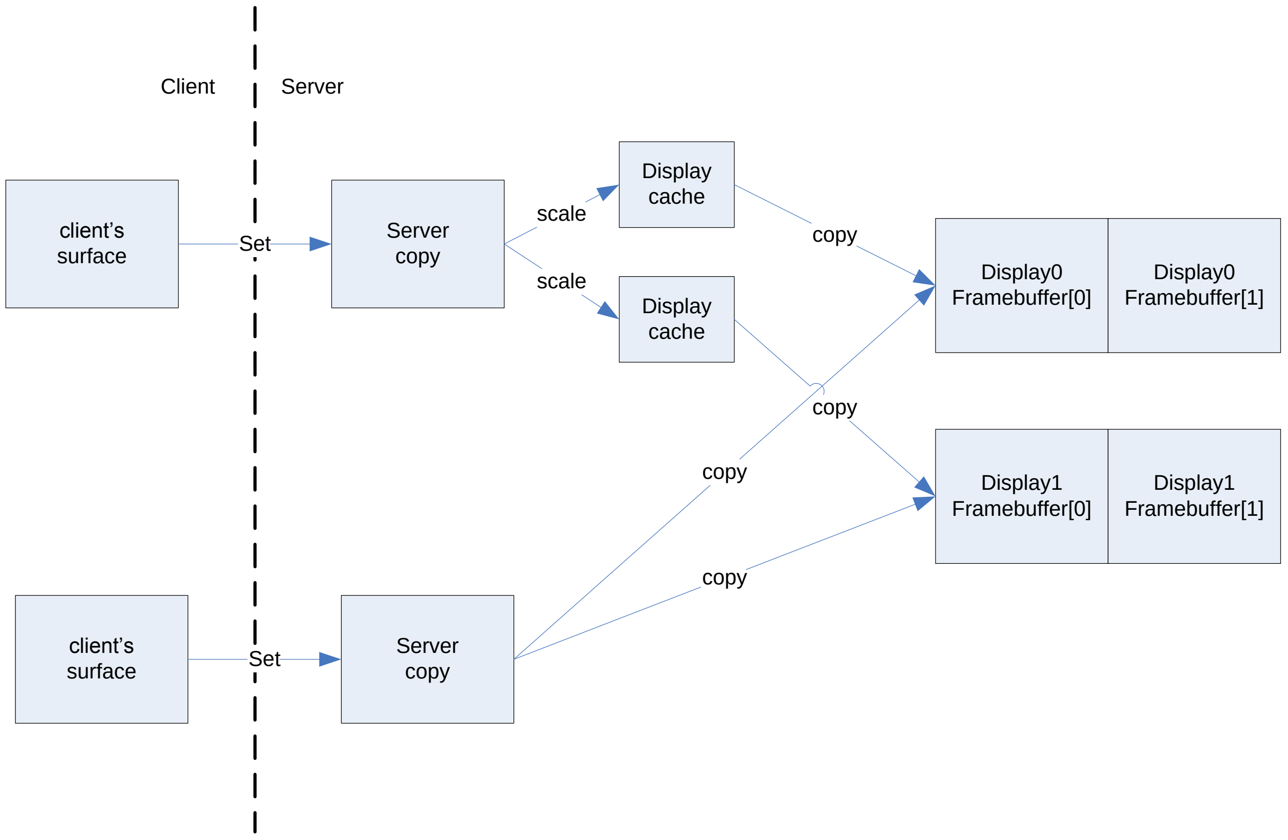
The position of the cursor is determined by the server. Any connection to a pointing device like a mouse must be done by the server application.

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

Display Cache

NOTE: *this feature will be removed when incremental scaled blits are implemented.*

An option available to clients is the “display cache”.



If the client knows it will submit updates infrequently in comparison to the number of framebuffer compositions, it will reduce M2MC bandwidth if scaling can be performed when the client updates, not when the framebuffer is composed. The M2MC bandwidth required for scaled blits is significantly higher than unscaled blits. The reduction of M2MC bandwidth comes at the price of higher memory usage.

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. If you system wants to allow dynamic creation of clients, it would require application IPC implemented above Nexus. See Nexus_MultiProcess.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.