

Hardware Accelerated 2D Rendering for Android

Jim Huang (黃敬群) <jserv@0xlab.org>

Developer, 0xlab

Feb 19, 2013 / Android Builders Summit

Rights to copy

© Copyright 2013 **0xlab**

<http://0xlab.org/>

contact@0xlab.org

Attribution – ShareAlike 3.0



You are free

Corrections, suggestions, contributions and translations
are welcome!

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Latest update: Feb 19, 2013

Under the following conditions

-  **Attribution.** You must give the original author credit.
-  **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: <http://creativecommons.org/licenses/by-sa/3.0/legalcode>



Agenda

(1) Concepts

(2) Performance Problems

(3) Hardware Accelerating

Case study: skia, webkit

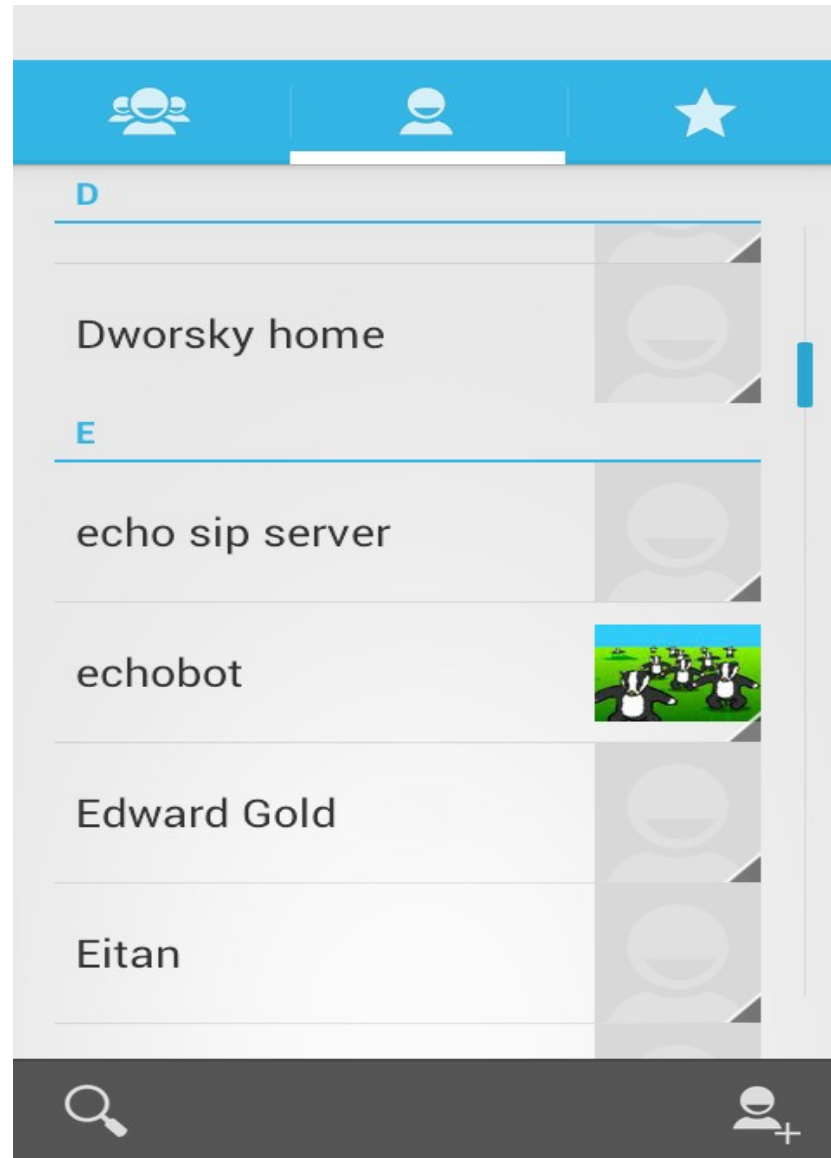


Concepts

Graphic Toolkit, Rendering, GPU operations

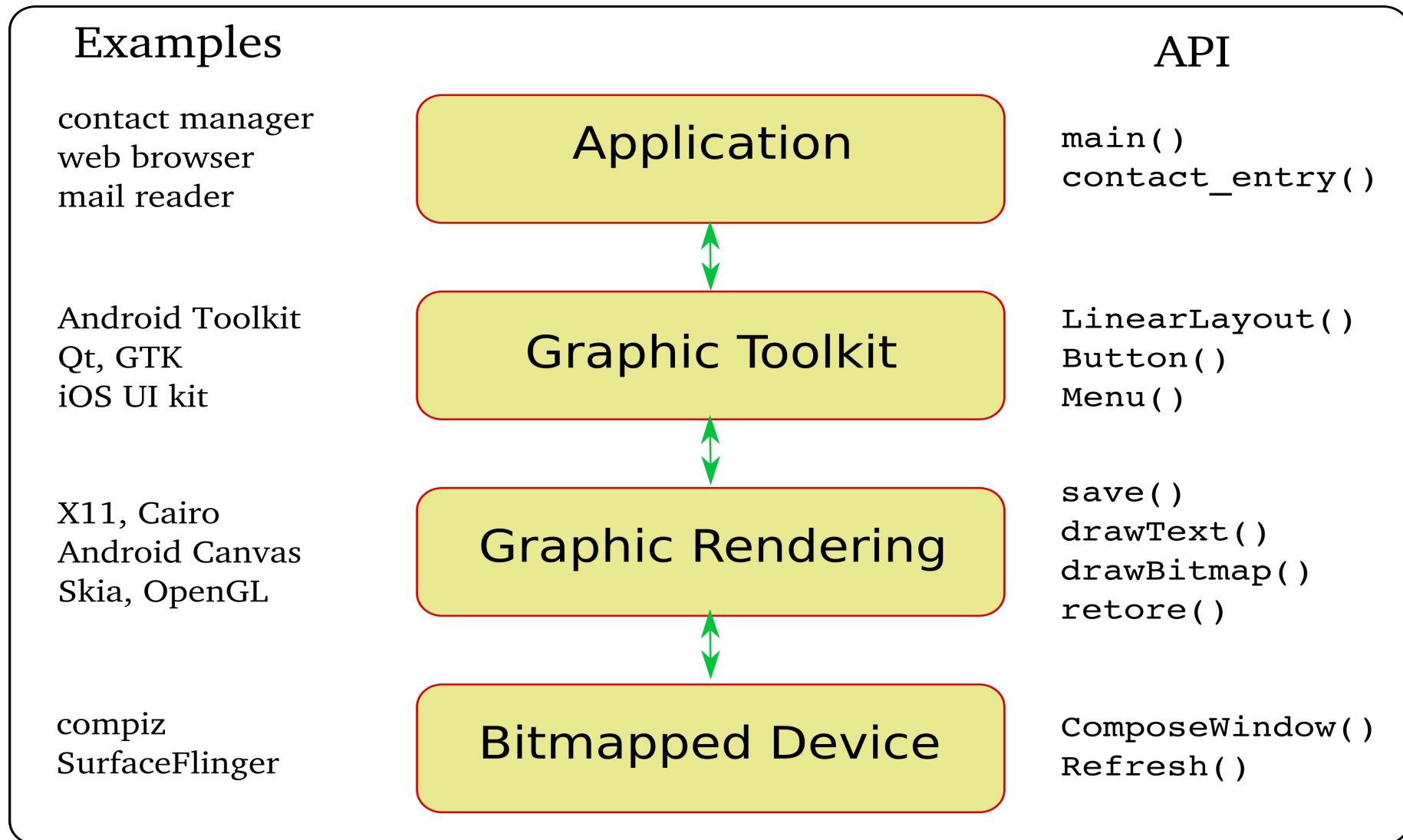


Revise what you saw on Android



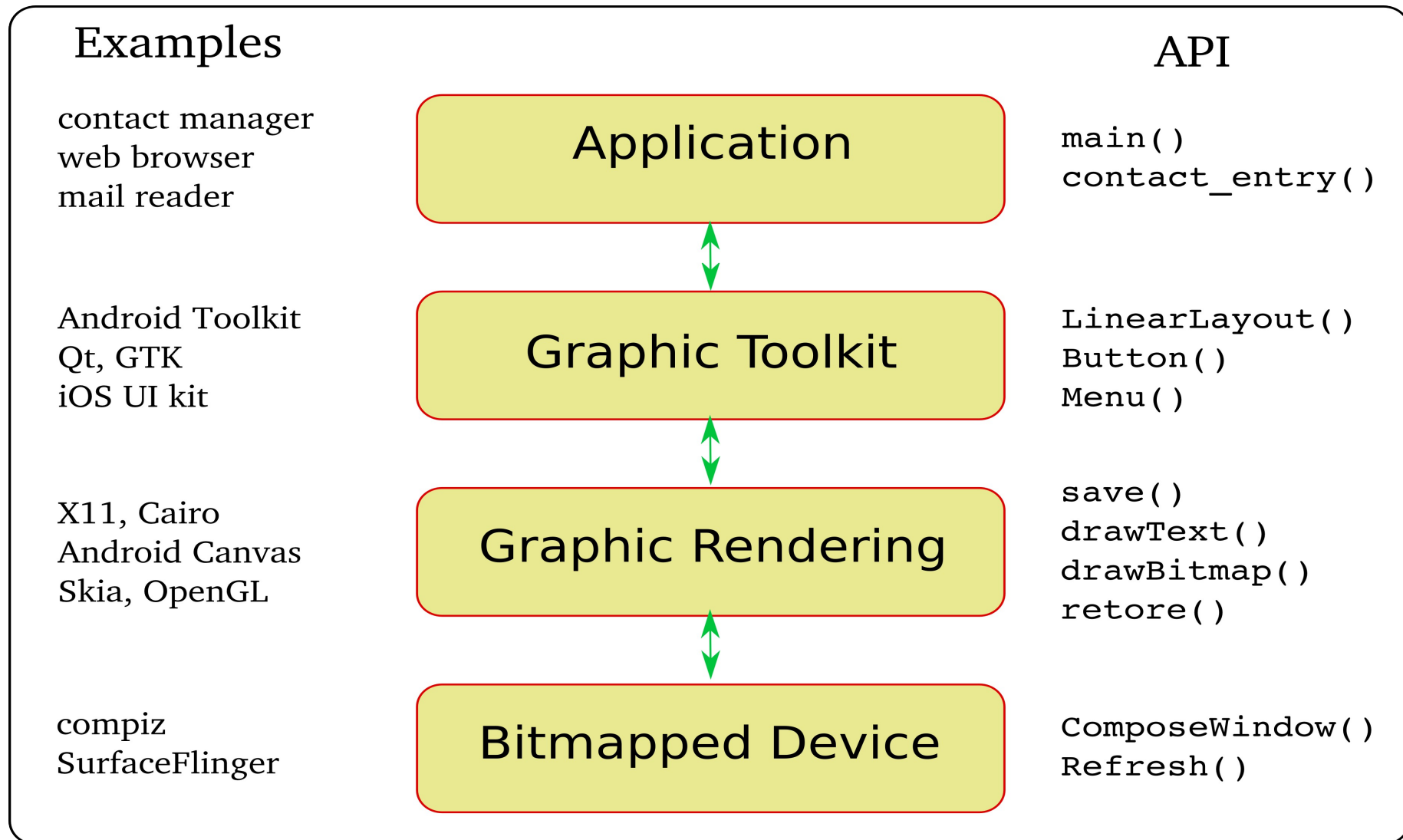
Exporting Graphics

- can be exported from any of the levels of the graphics stack
 - Application, Graphic Toolkit, Graphic Rendering, Bitmapped Device



Exporting Graphics - Application

- Normal way Linux/Android/Iphone runs apps.
 - The application itself is exported and run locally.



Exporting Graphics - Toolkit

- Technically very complex. Android has 15 different toolkit API variants.
- Every application can extend the toolkit with custom widgets (subclasses of `android.view.View`).

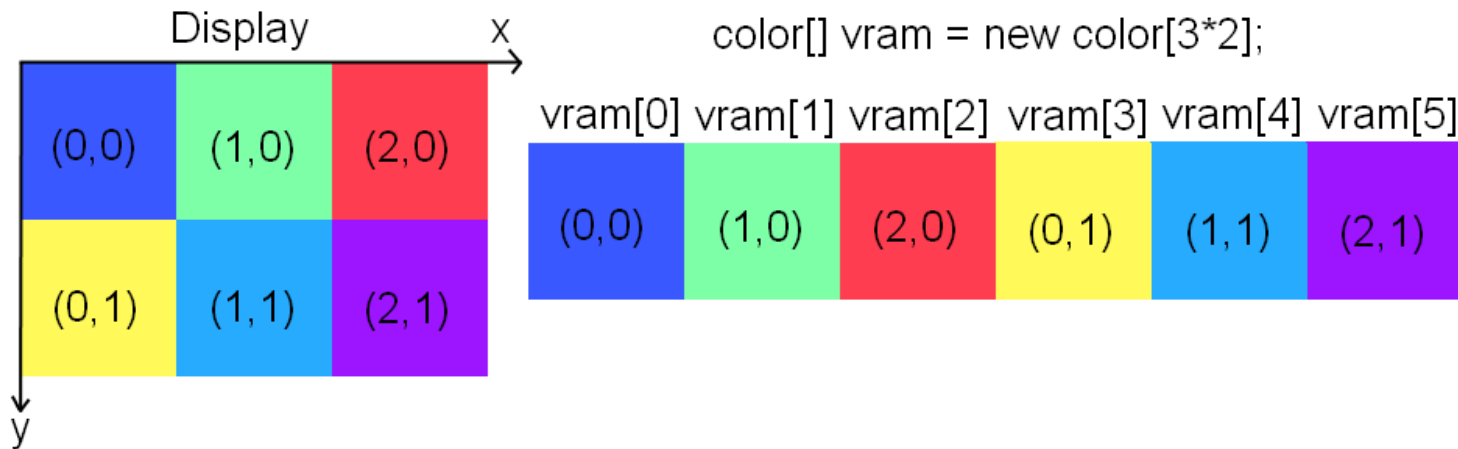
Exporting Graphics - Rendering

- Exports graphics at the rendering level.
- In Android there are a number of rendering interfaces that can be used:
 - skia graphics
 - OpenGL ES 1.1 or OpenGL ES 2.0
 - `Android.view.View`



2D Graphics

- The display presents us the contents of something called the framebuffer.
- The framebuffer is an area in (V)RAM
- For each pixel on screen there's a corresponding memory cell in the framebuffer
- Pixels are addressed with 2D coordinates.

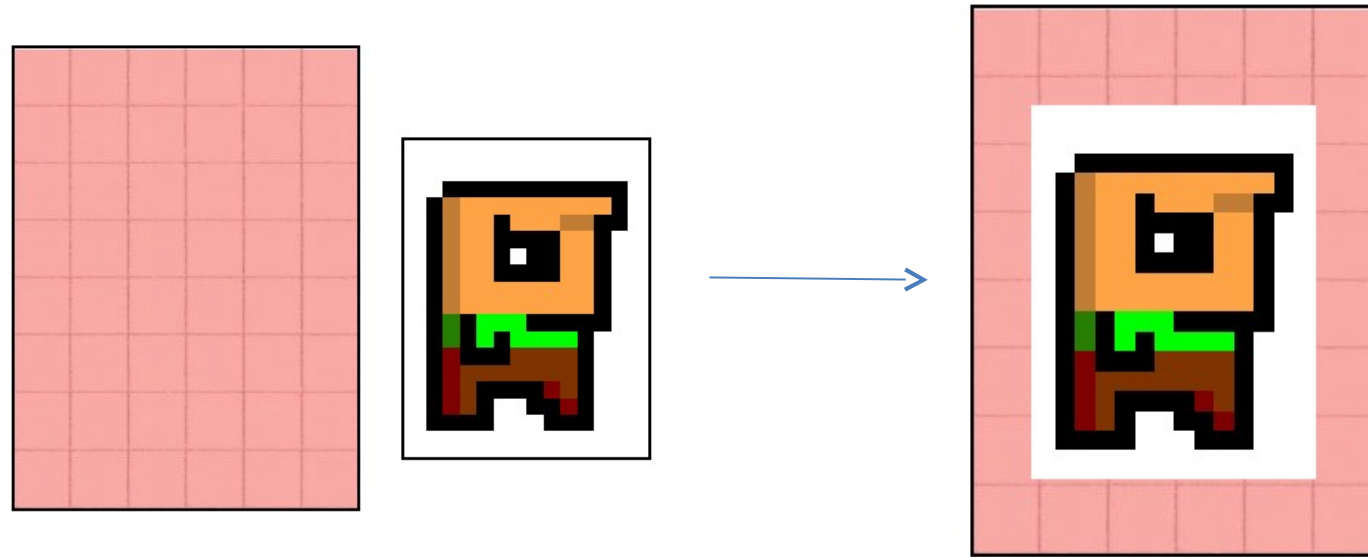


2D Graphics

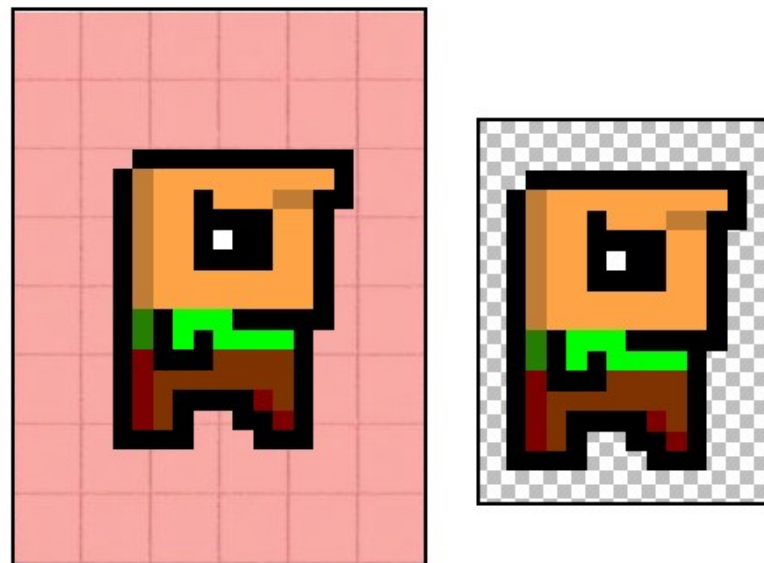
- To change what is displayed → change the colors of pixels in (V)RAM.
 - Pixel colors are encoded as RGB or RGBA
- To draw shapes → need to figure out which framebuffer pixels we have to set.
 - Images (bitmaps) are not special either
 - Pixels of the bitmap get stored in a memory area, just like we store framebuffer pixels.
- To draw a bitmap to the framebuffer → copy the pixels. (Blitting)
- We can perform the same operations on bitmaps as we perform on the framebuffer, e.g. draw shapes or other bitmaps.



Blitting: copy (parts of) one bitmap to another



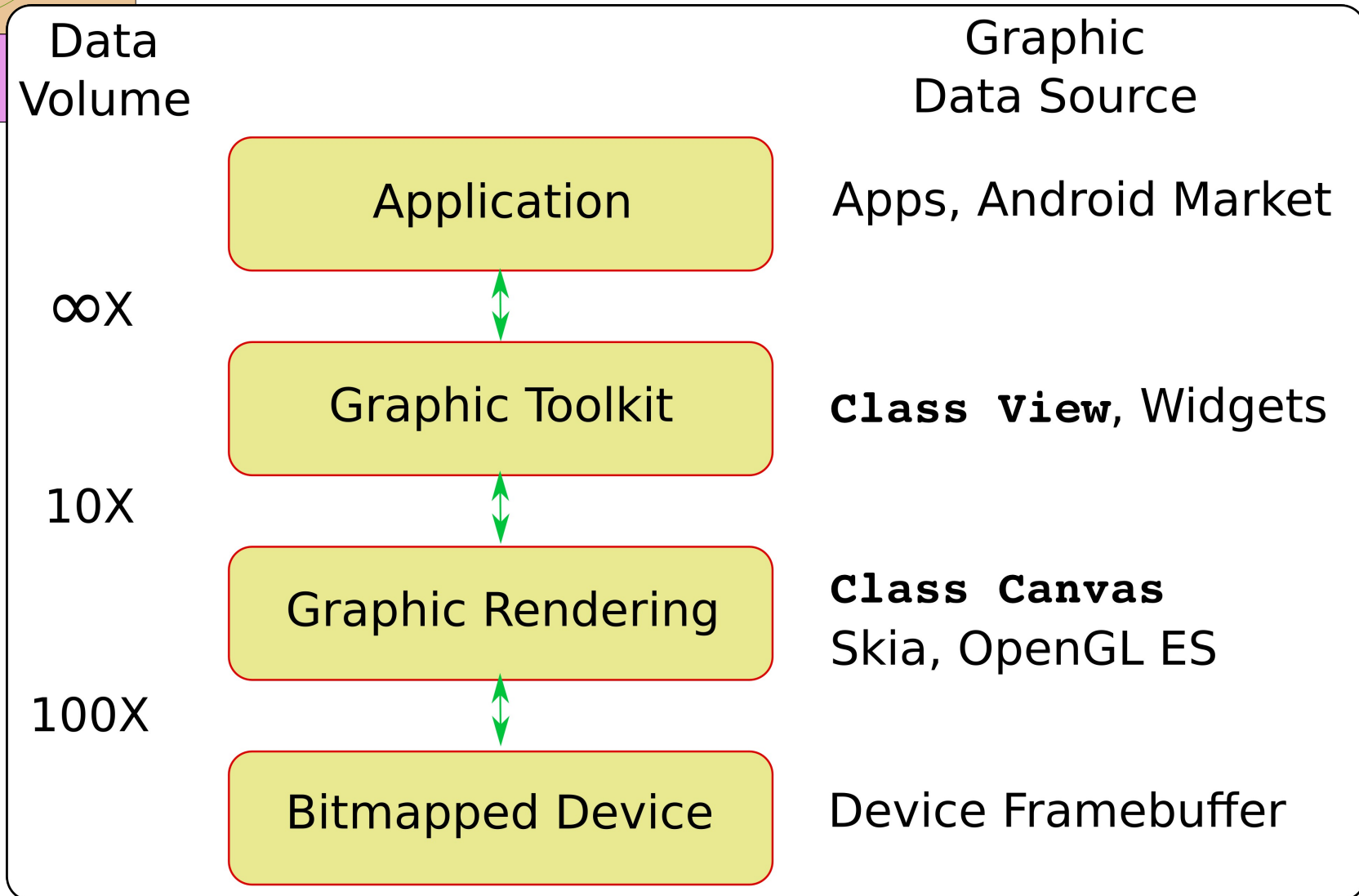
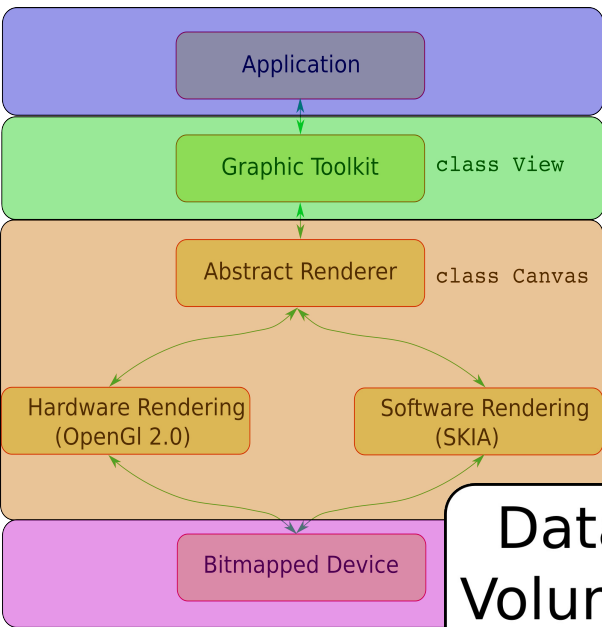
Alpha Compositing: blitting + alpha blending



- Alpha value of a pixel governs transparency
- Instead of overwriting a destination pixel we mix its color with the source pixel.

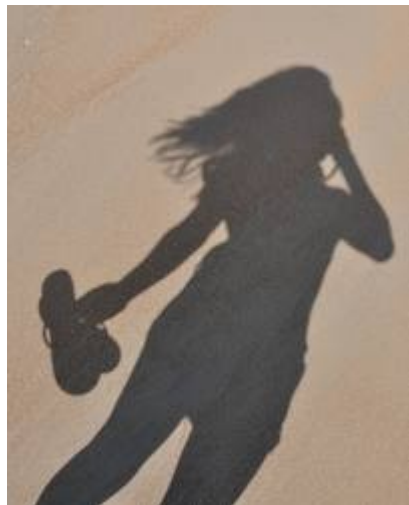


Android Graphics Stack



Rendering Level: skia

- The rendering level is the graphics layer that actually “colors” the pixels in the bitmap.
- skia is a compact open source graphics library written in C++.
- Currently used in Google Chrome, Chrome OS, and Android.



Skia is Greek for “shadow”



Rendering Level: skia

- skia is a complete 2D graphic library for drawing Text, Geometries, and Images. Features include:
 - 3x3 matrices w/ perspective
 - Antialiasing, transparency, filters
 - Shaders, xfermodes, maskfilters, patheffects



Rendering Level: skia

- Each skia call has two components:
 - the primitive being drawing (SkRect, SkPath, etc.)
 - color/style attributes (SkPaint)

- Usage example:

```
canvas.save();
```

```
canvas.rotate(45);
```

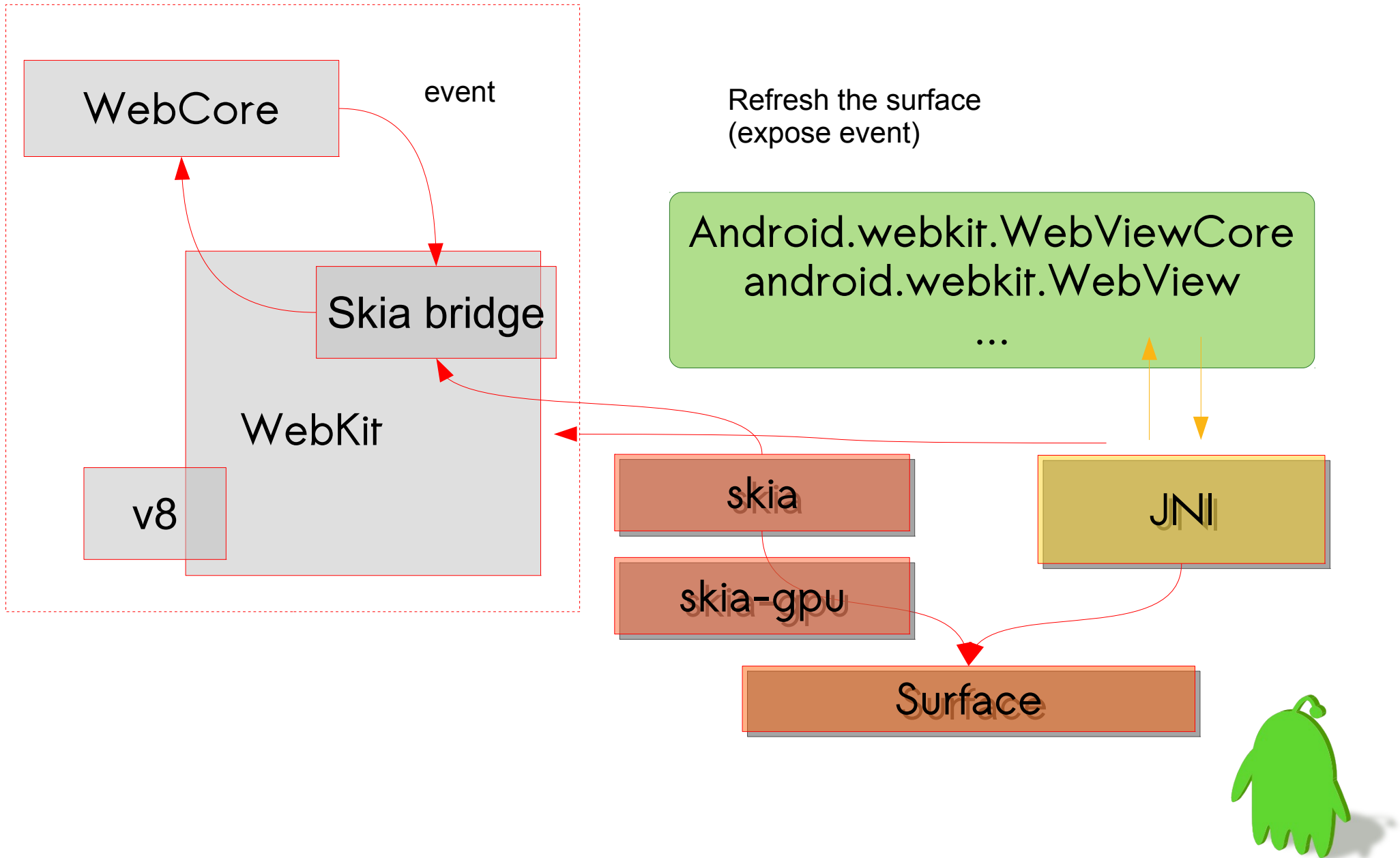
```
canvas.drawRect(rect, paint);
```

```
canvas.drawText("abc", 3, x, y, paint);
```

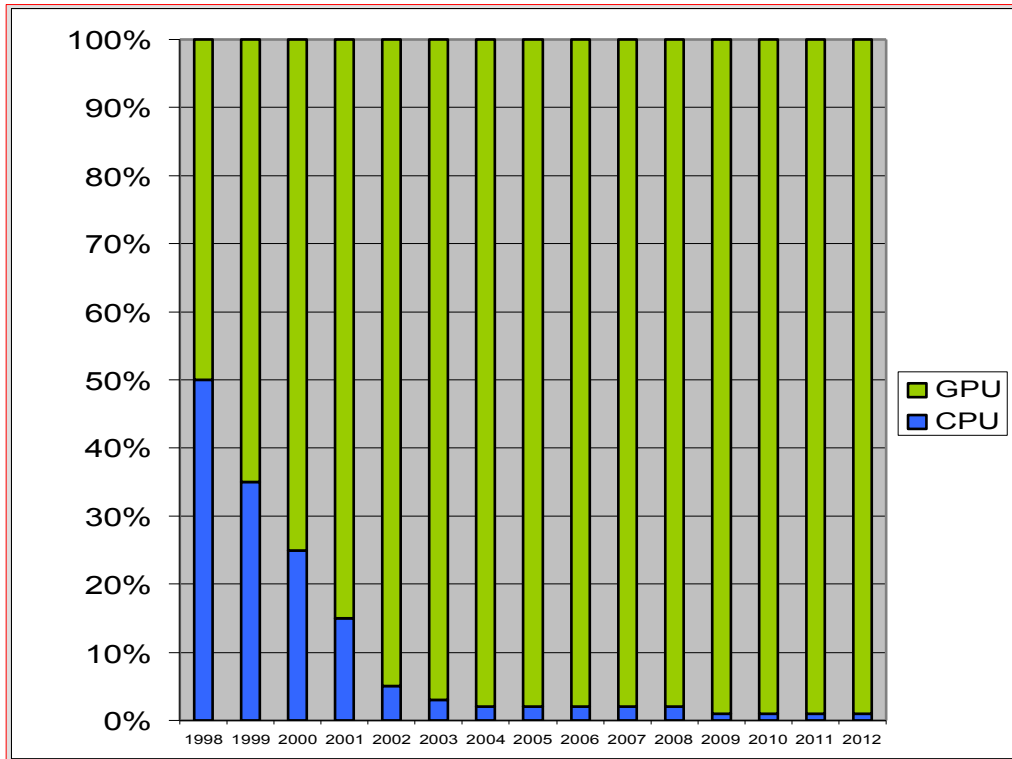
```
canvas.restore();
```



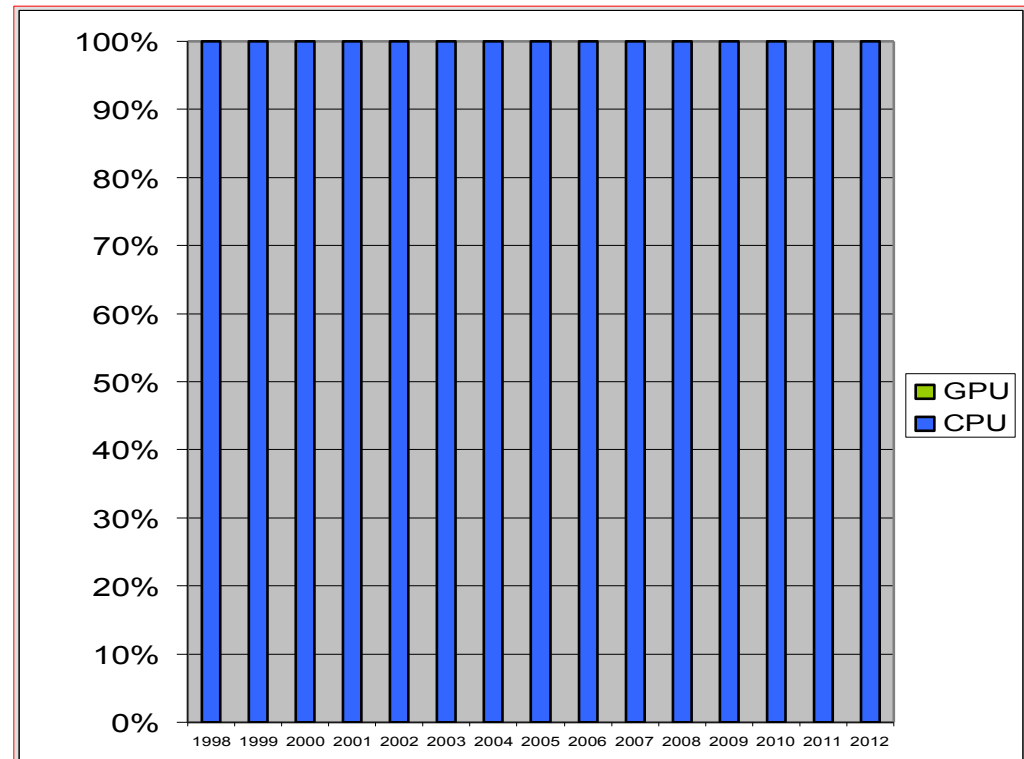
WebKit in Android



CPU vs. GPU Limited at Rendering Tasks over Time



Pipelined 3D Interactive Rendering



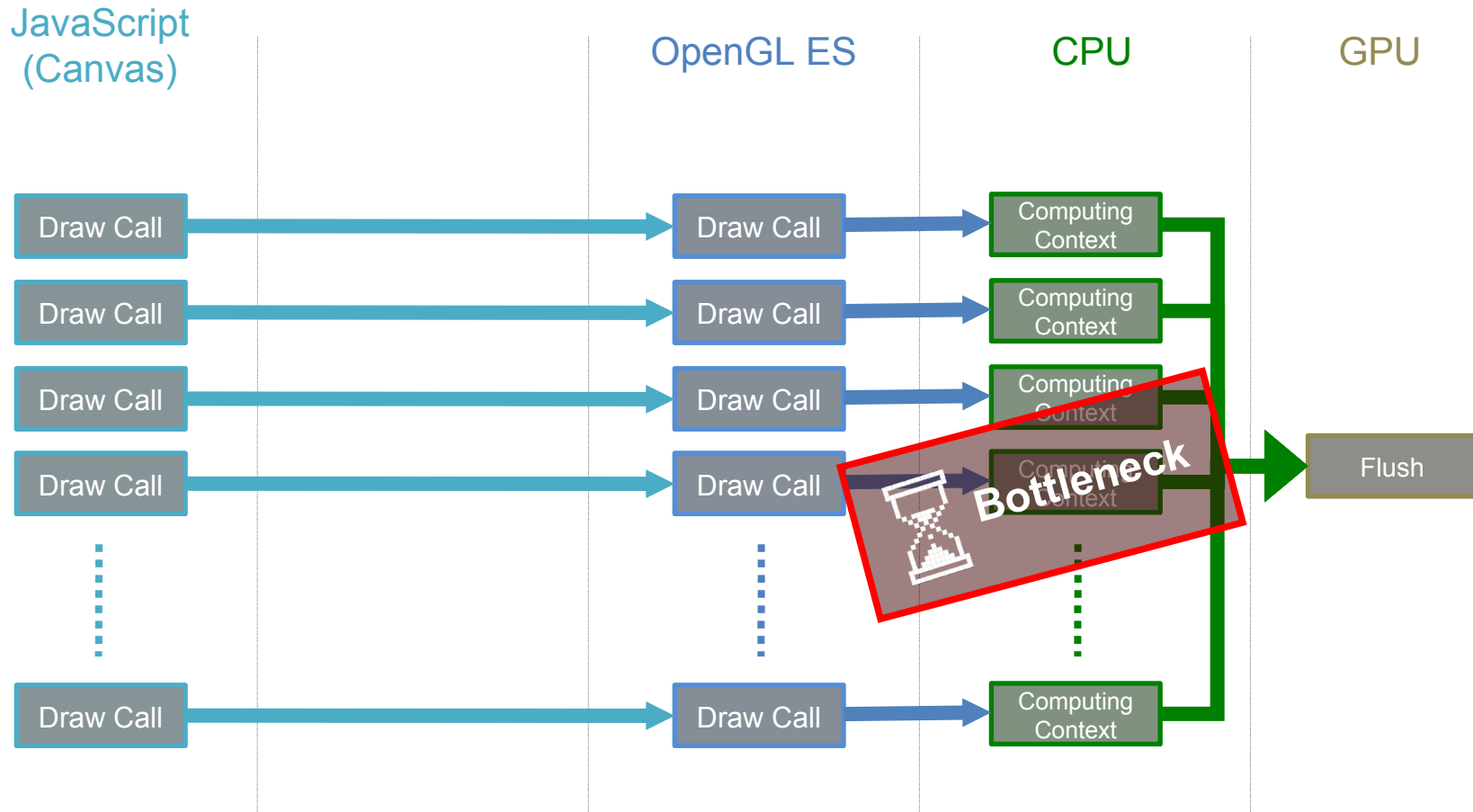
Path Rendering

*Goal of NV_path_rendering is to make path rendering a GPU-limited task
Render all interactive pixels, whether 3D or 2D or web content with the GPU*



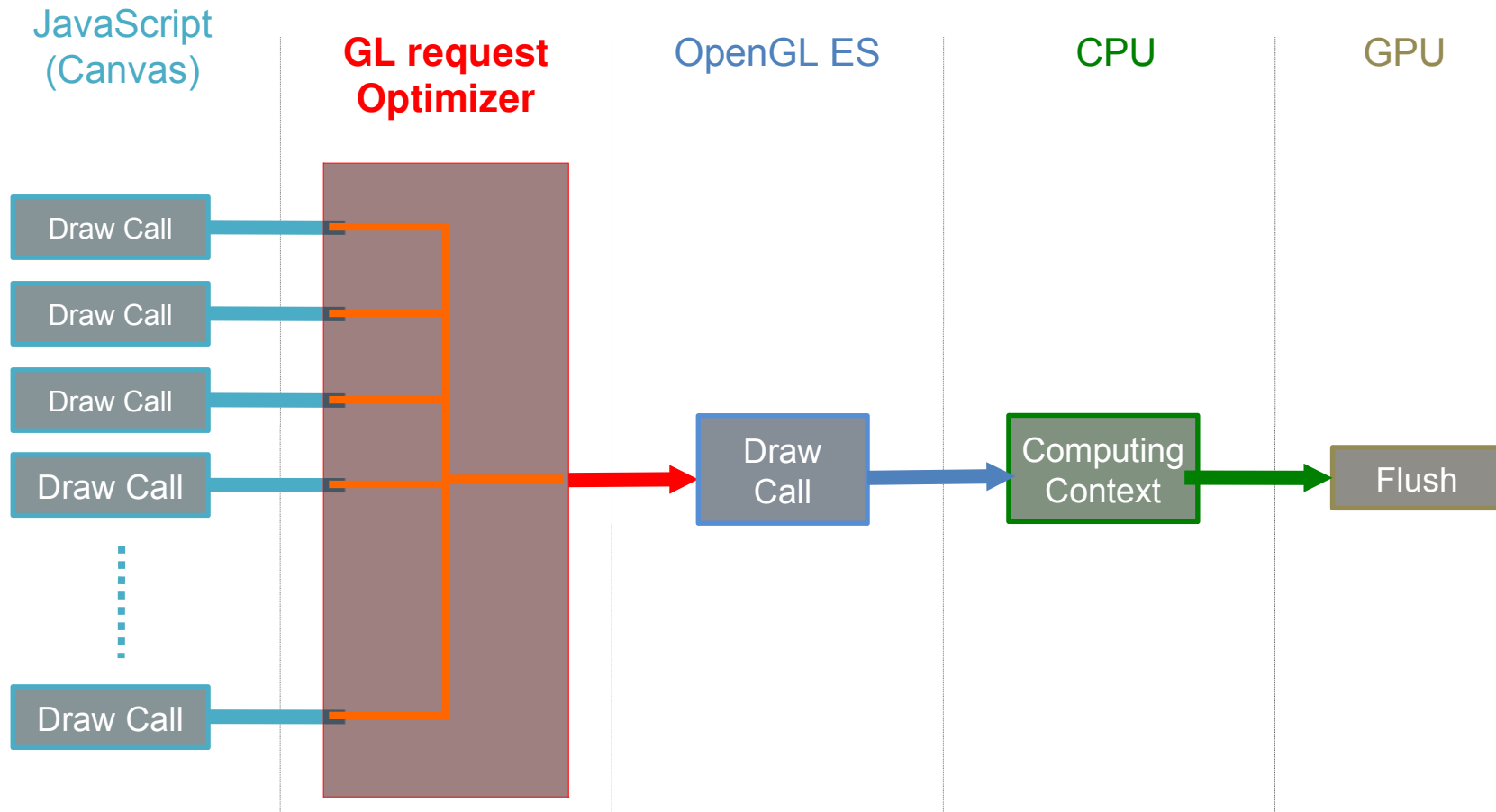
Rendering Paths

[skia + gpu; Chrome browser]



Rendering Paths

[ideal case]



- Problems
 - calls glDrawSomethings too many times
 - changes gl states too many times
 - switches FBO too many times
 - vector graphics APIs and shadows are really slow
- Increases dramatically CPU overhead

Draw Call At Once	Ideal GL	skia
Bitmap Sprite	Good	Good
Convex Path	Good	Poor
Concave Path	Good	Poor
Bitmap Sprite + Path	Good	Poor
Path + Different Shadow	Good	Poor
Text + Different Draw Call	Good	Poor



The performance problem is still
rendering...

Let's look into deeper.



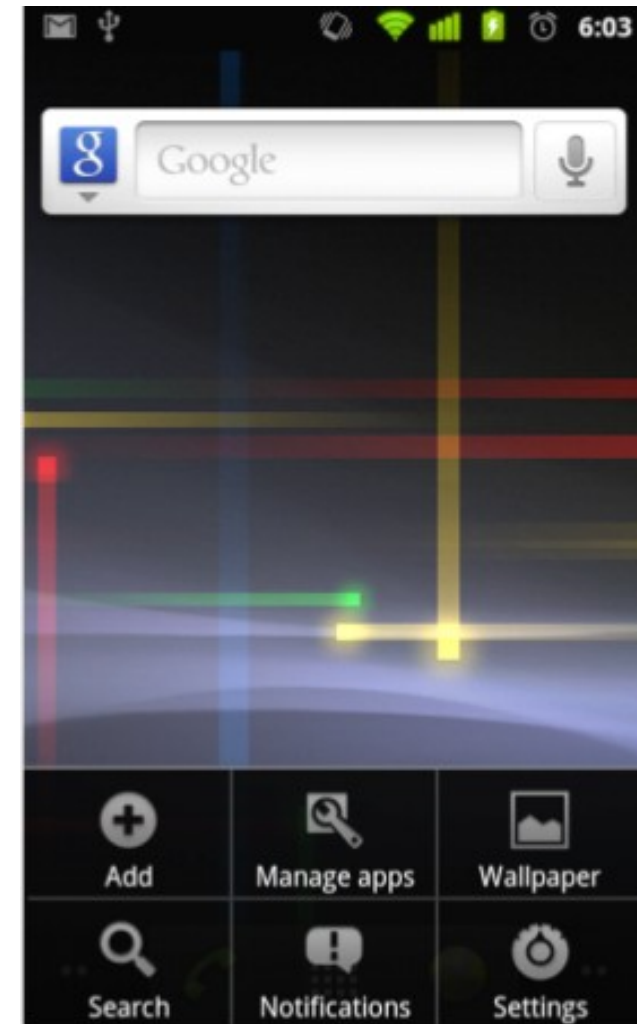
Hardware Accelerating

How Android utilizes GPU functionalities



Myths and Facts

- Myth: Android 1.x is slow because of no hardware accelerations
 - NOT TRUE! window compositing utilizes hardware accelerations.
 - But it is quite constrained
- There are 4 Window: Status Bar, Wallpaper, Launcher, and Menu in the right screenshot.
 - Hardware composites animations of Activity transition, the fading in/out of Menu.
- However, the content of Window (Canvas) is being accelerated by hardware since Android 3.x

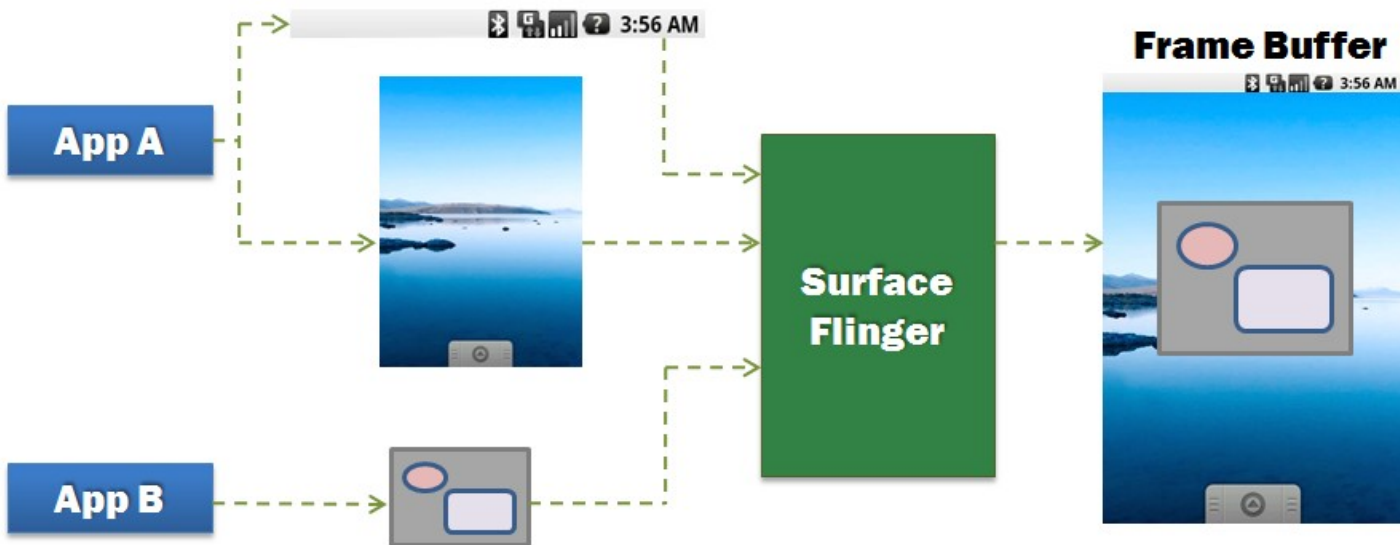
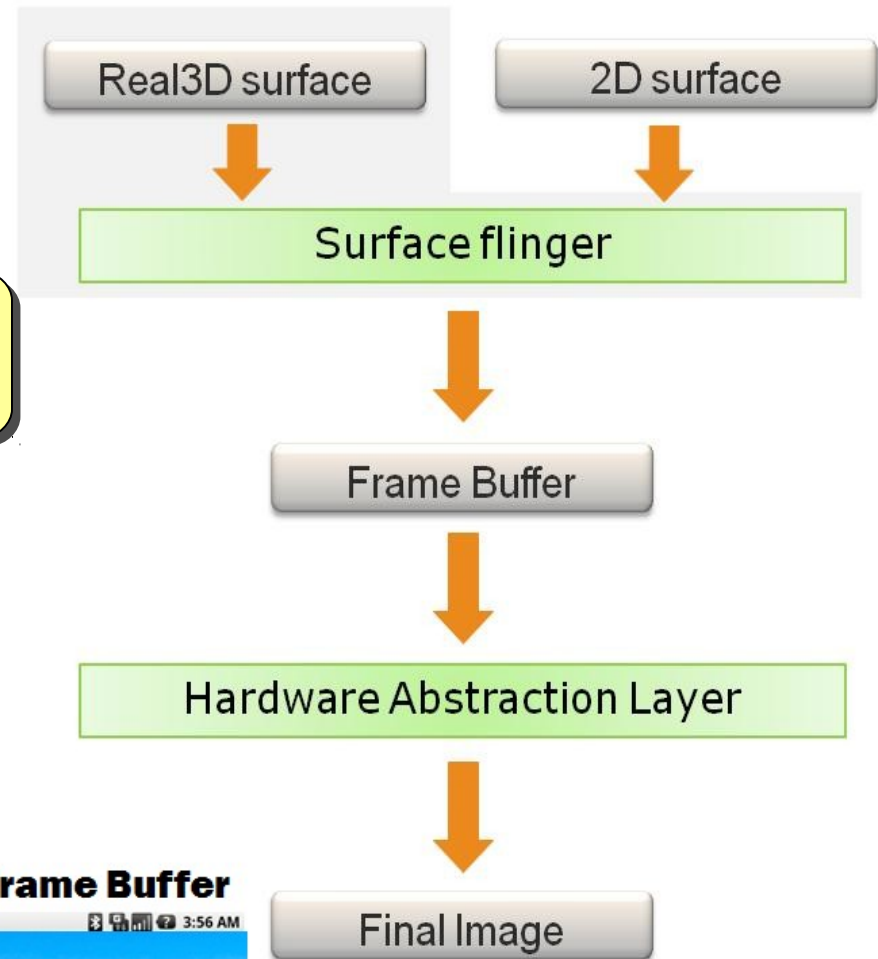
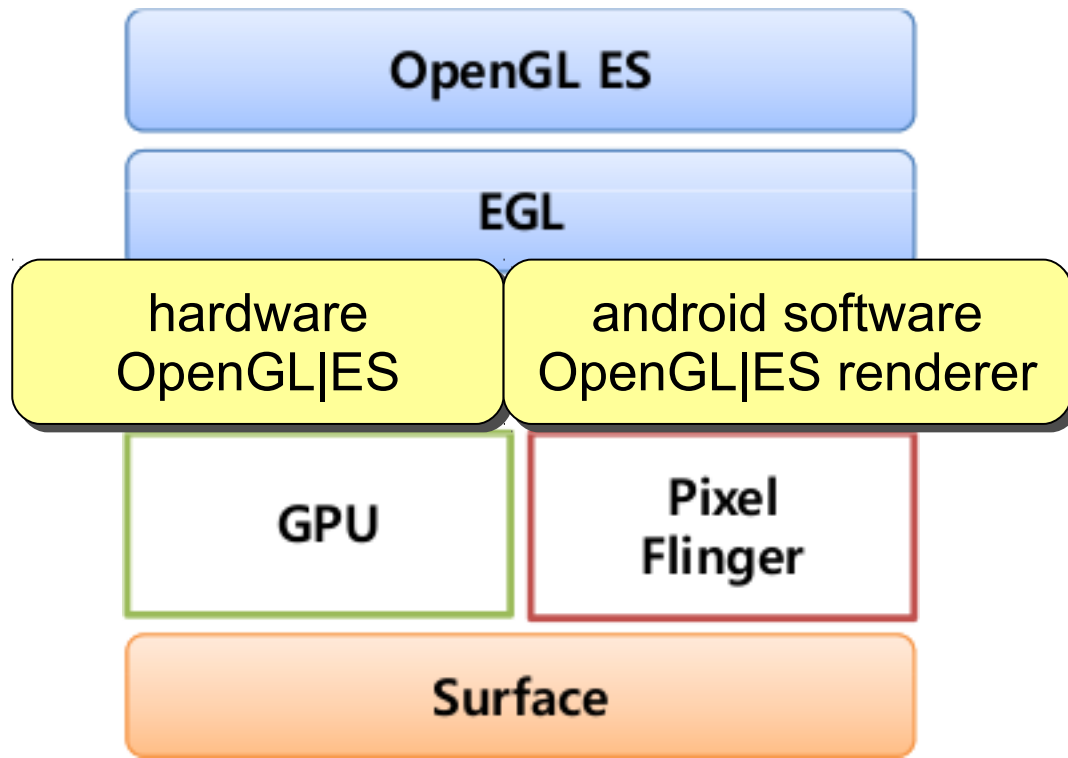


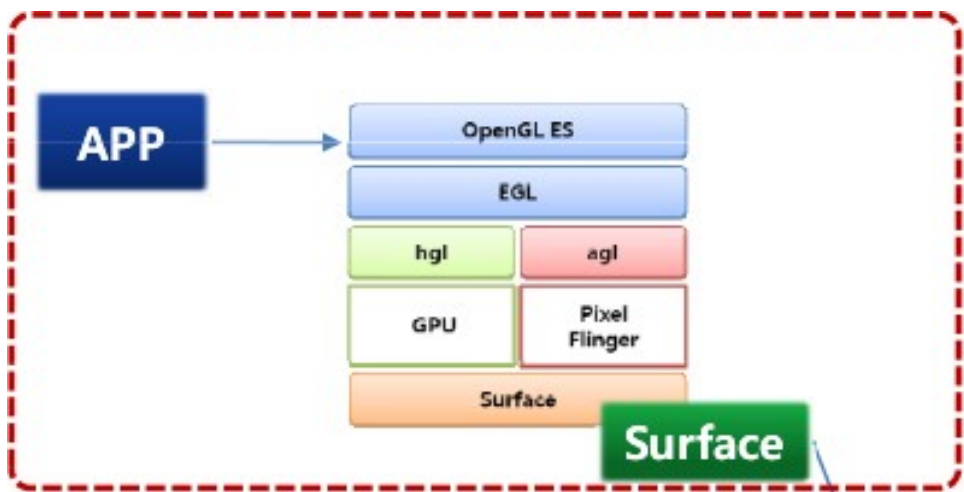
View & TextureView

- **View**
 - represents the basic building block for UI
 - occupies a rectangular area on the screen and is responsible for drawing and event handling.
- **SurfaceView**
 - provides a dedicated drawing surface embedded inside of a view hierarchy.
- **TextureView**
 - Since Android 4.0
 - Only activated when hardware acceleration is enabled !
 - has the same property of SurfaceView, but you can create GL surface and perform GL rendering above them.



from EGL to SurfaceFlinger



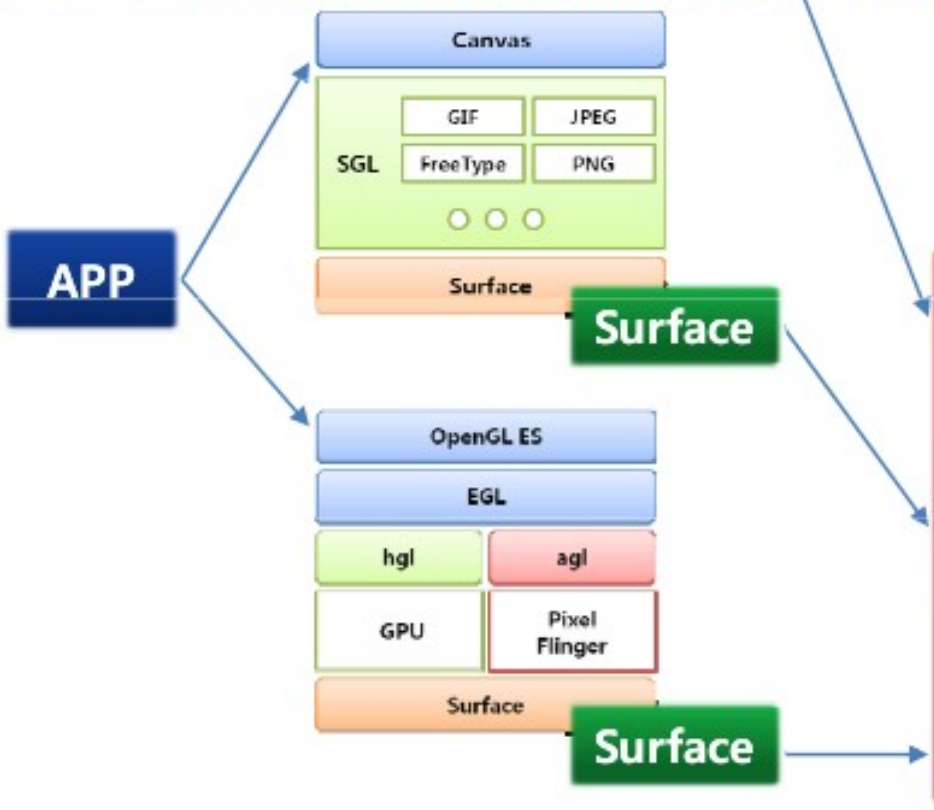


SurfaceFlinger::instantiate()

- AddSevice("Surface Flinger"..)

SurfaceFlinger::readyToRun()

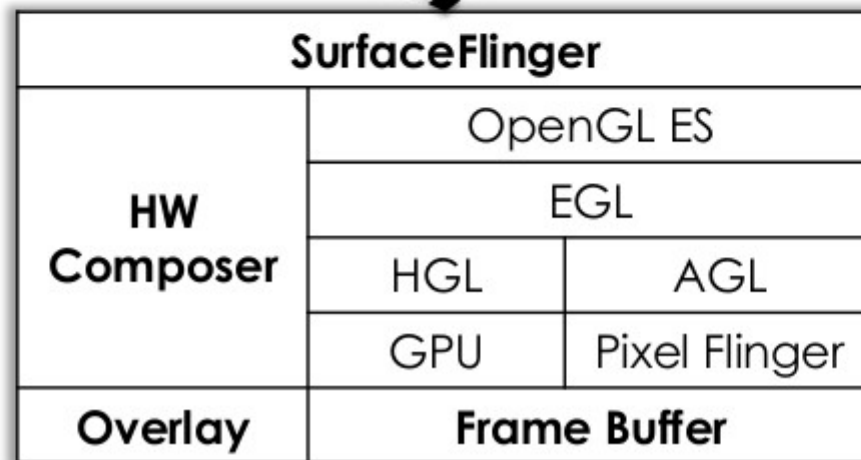
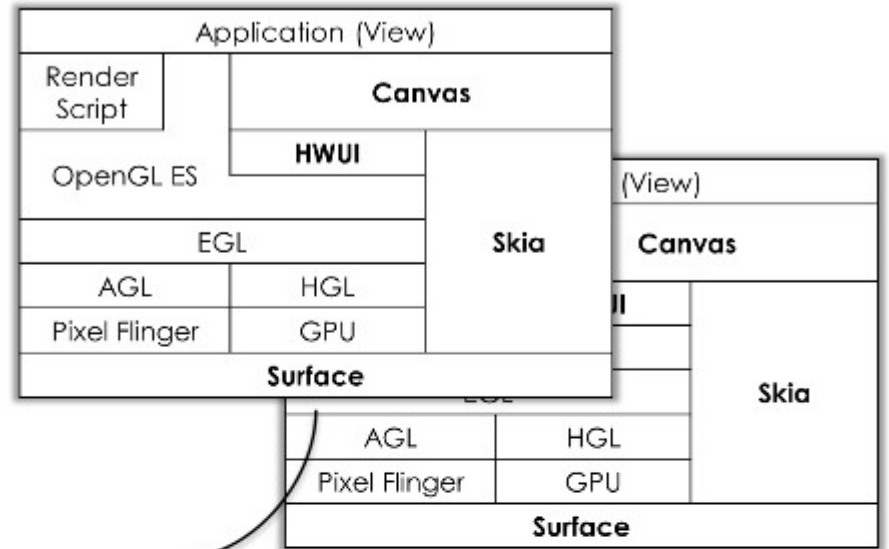
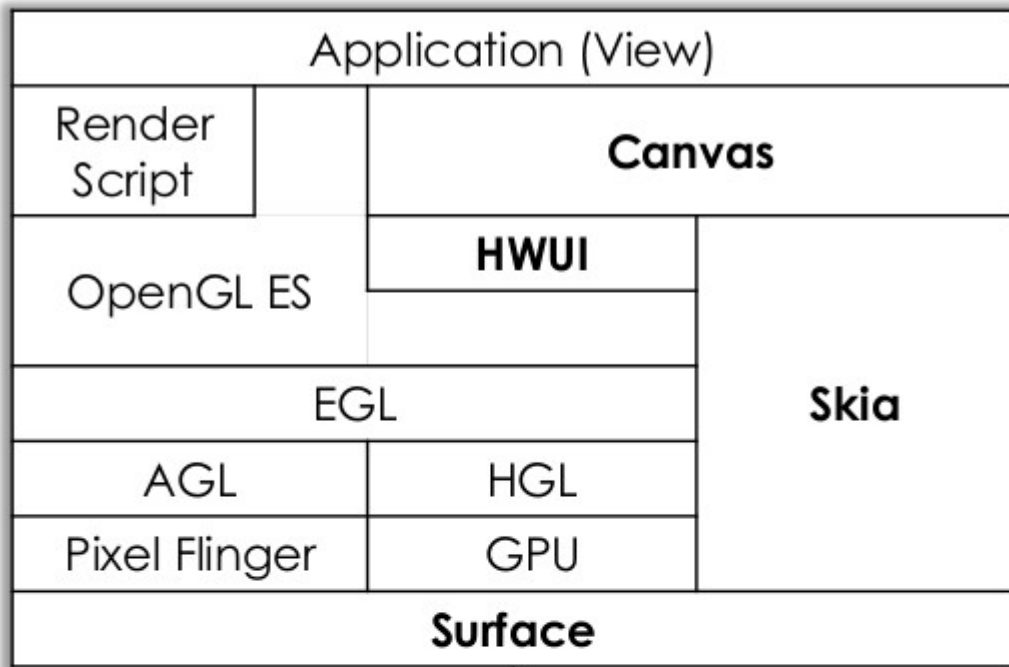
- Gather EGL extensions
- Create EGL Surface and Map Frame Buffer
- Create our OpenGL ES context
- Gather OpenGL ES extensions
- Init Display Hardware for GPU



SurfaceFlinger::threadLoop()

- Wait for Event
- Check for tranaction
- Post Surface (if needed)
- Post FrameBuffer ...





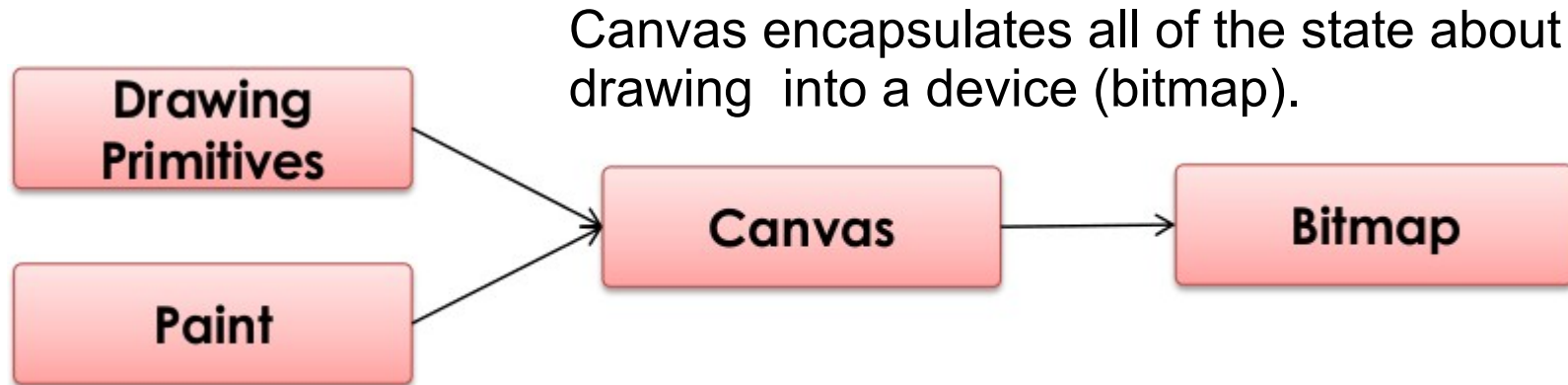
Case study: skia

Paint, Canvas, Backend



skia, again

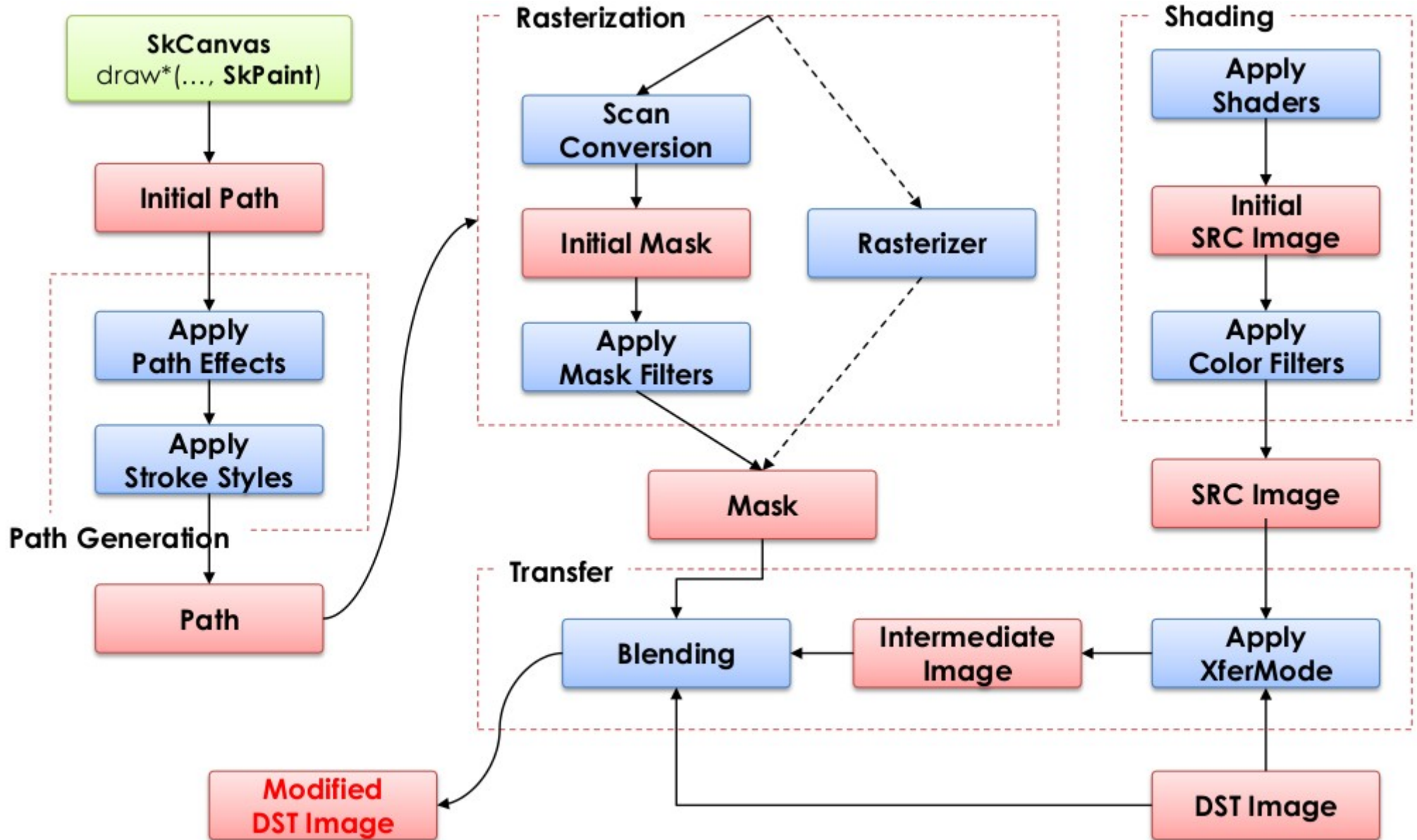
Drawing basic primitives include rectangles, rounded rectangles, ovals, circles, arcs, paths, lines, text, bitmaps and sprites. Paths allow for the creation of more advanced shapes.



While Canvas holds the state of the drawing device, the state (style) of the object being drawn is held by Paint, which is provided as a parameter to each of the draw() methods. Paint holds attributes such as color, typeface, textSize, strokeWidth, shader (e.g. gradients, patterns), etc.



skia rendering pipeline



Skia backends

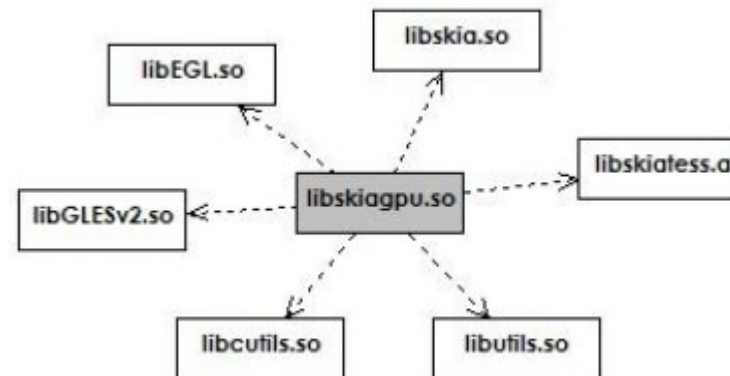
- Render in software

- create a native window and then
- wrap a pointer to its buffer as an SkBitmap
- Initialize an SkCanvas with the bitmap



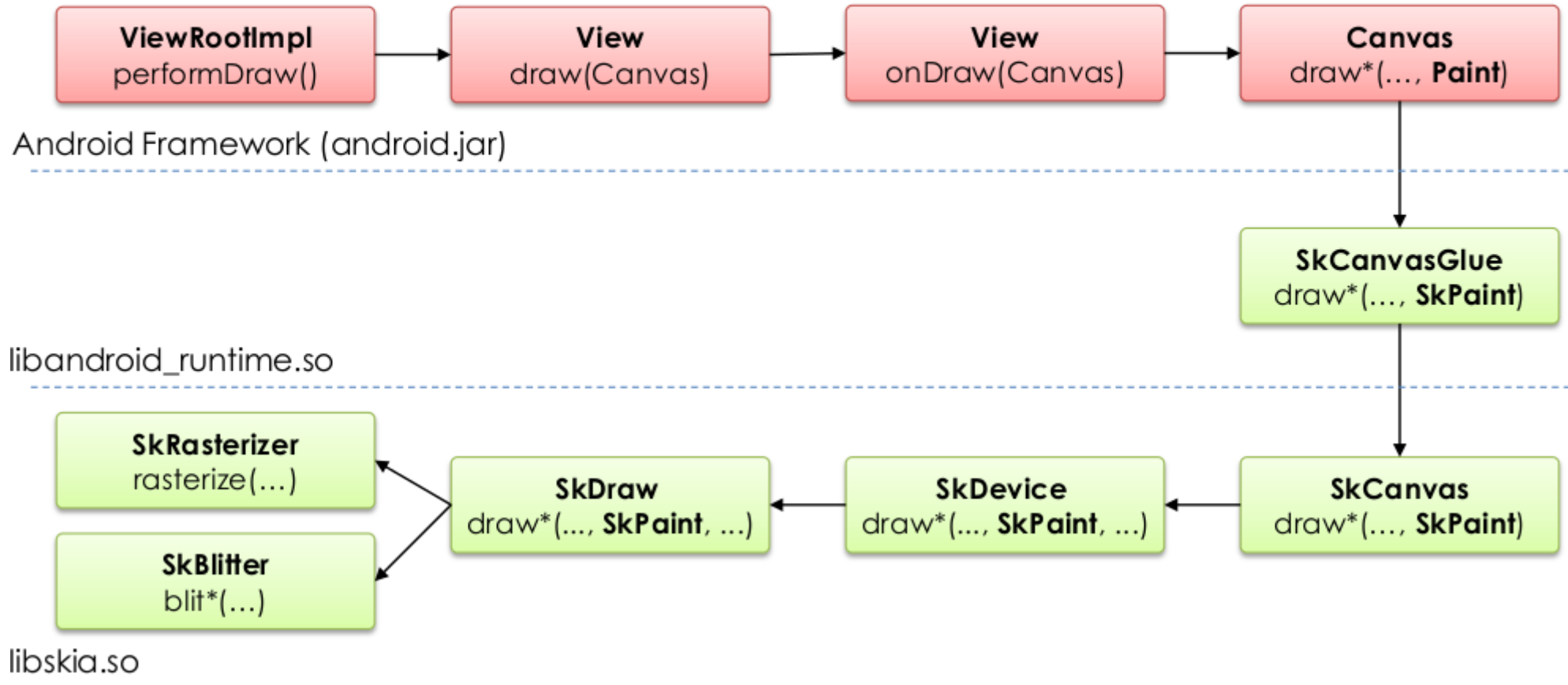
- Render in hardware acceleration

- create a GLES2 window or framebuffer and
- create the appropriate GrContext, SkGpuDevice, and SkGpuCanvas



How Views are Drawn [Android 2.x]

CPU Rasterization + GPU Composition



Hardware-accelerated 2D Rendering

- Since Android 3.x, more complex than before!
- Major idea: transform the implementation of 2D Graphics APIs into OpenGL|ES requests
- Texture, Shader, GLContext, pipeline, ...
- Major parts for hardware-accelerated 2D Rendering
 - Primitive Drawing: Shape, Text, Image
 - Layer/Surface Compositing



Control hardware accelerations

- Application level

 - `<application android:hardwareAccelerated="true">`

 - Default value

 - False in Android 3.x, True in Android 4.x

- Activity

- Window

 - `WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED`

- View

 - `setLayerType(View.LAYER_TYPE_SOFTWARE, null)`



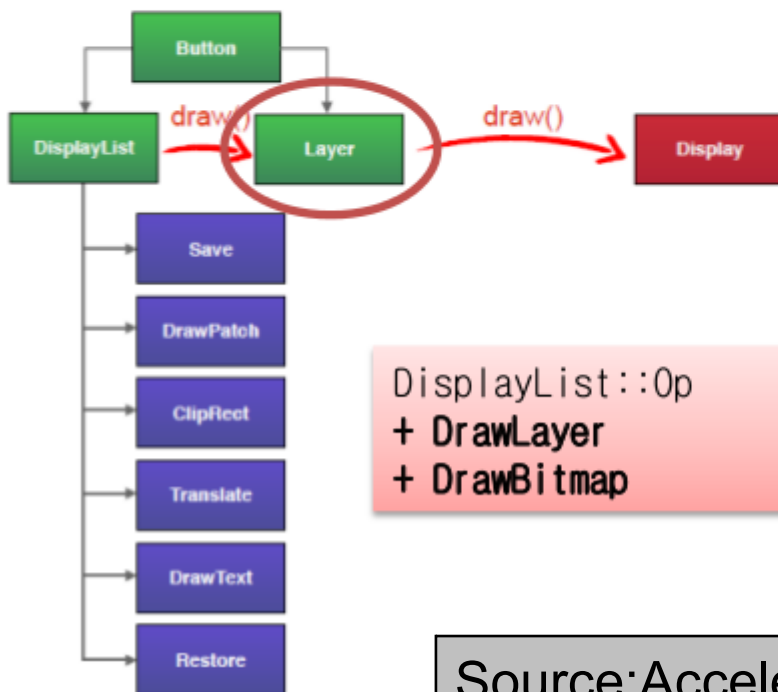
`View.setLayerType(int type, Paint p)`

Layers = Off-screen Buffers or Caches



View Layers since Android 3.x

type	View is H/W-Accelerated	View is NOT H/W Accelerated
LAYER_TYPE_NONE	<ul style="list-style-type: none"> Rendered in H/W Directly into surface 	<ul style="list-style-type: none"> Rendered in S/W Directly into surface
LAYER_TYPE_HARDWARE	<ul style="list-style-type: none"> Rendered in H/W Into hardware texture 	<ul style="list-style-type: none"> Rendered in S/W Into bitmap
LAYER_TYPE_SOFTWARE	<ul style="list-style-type: none"> Rendered in S/W Into bitmap 	<ul style="list-style-type: none"> Rendered in S/W Into bitmap



```

DisplayList::Op
+ DrawLayer
+ DrawBitmap
  
```

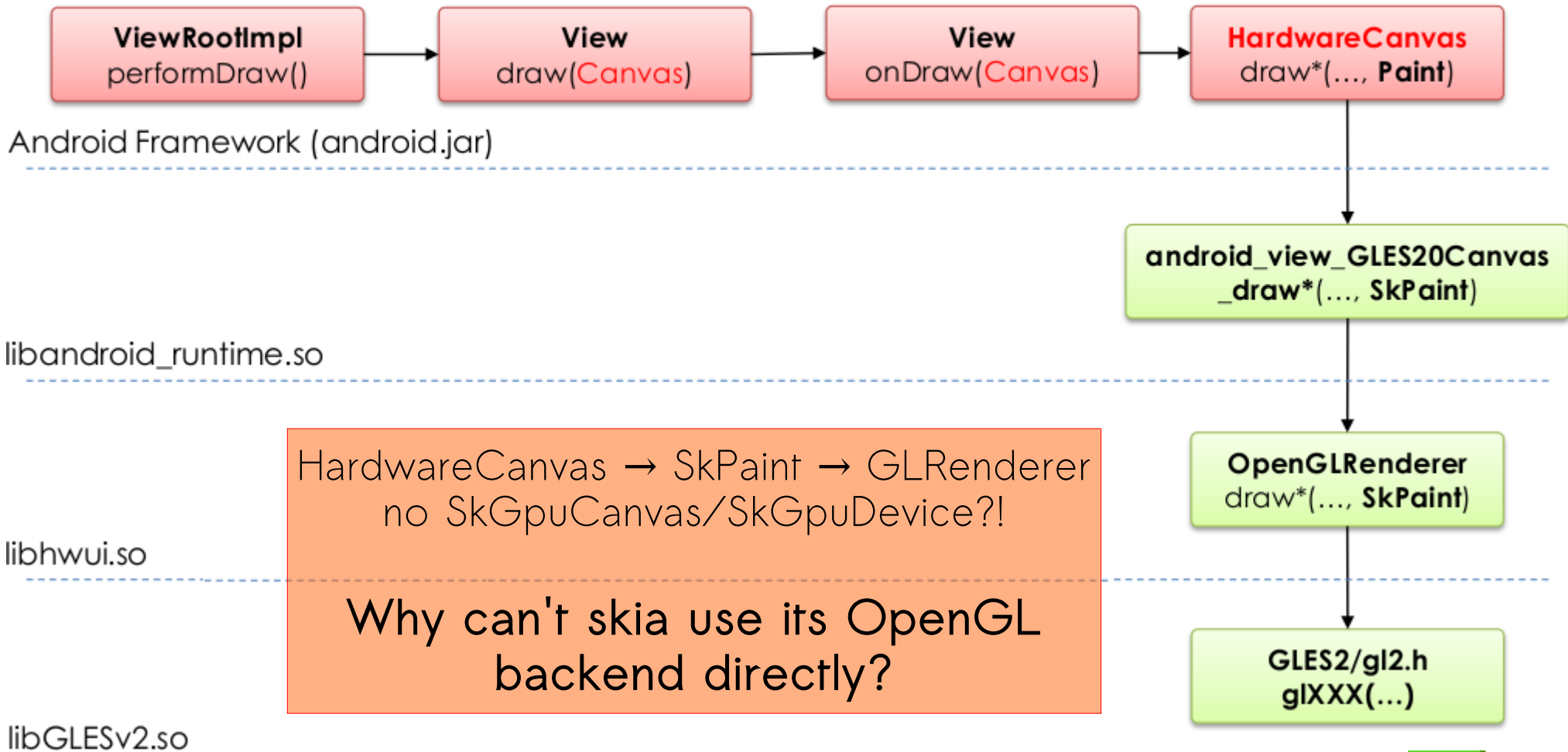
	Hardware layer	DisplayList	Software
Time in ms	0.009	2.1	10.3

*Measured when drawing a ListView with Android 3.0 on a Motorola XOOM

How Views are Drawn [Android 3.x]

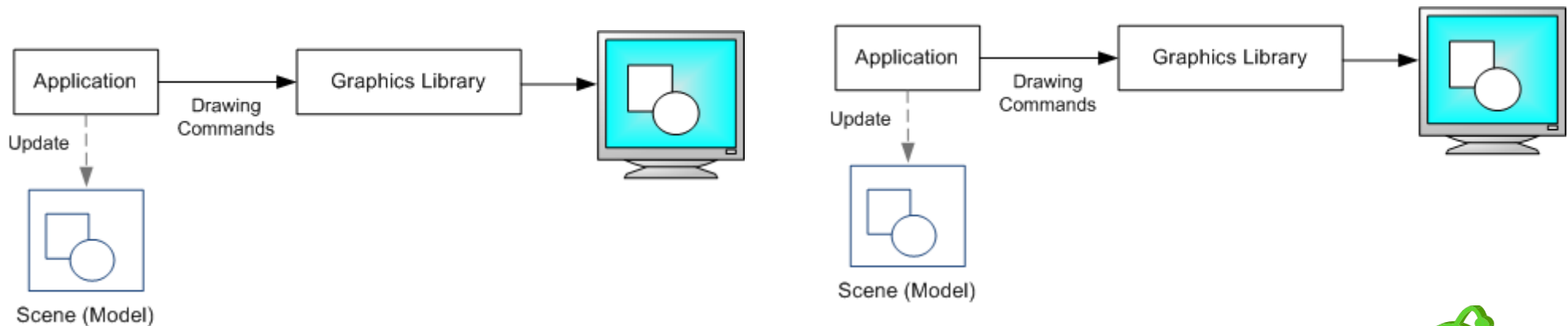
`android:hardwareAccelerated="true"`

GPU Rasterization + GPU Composition

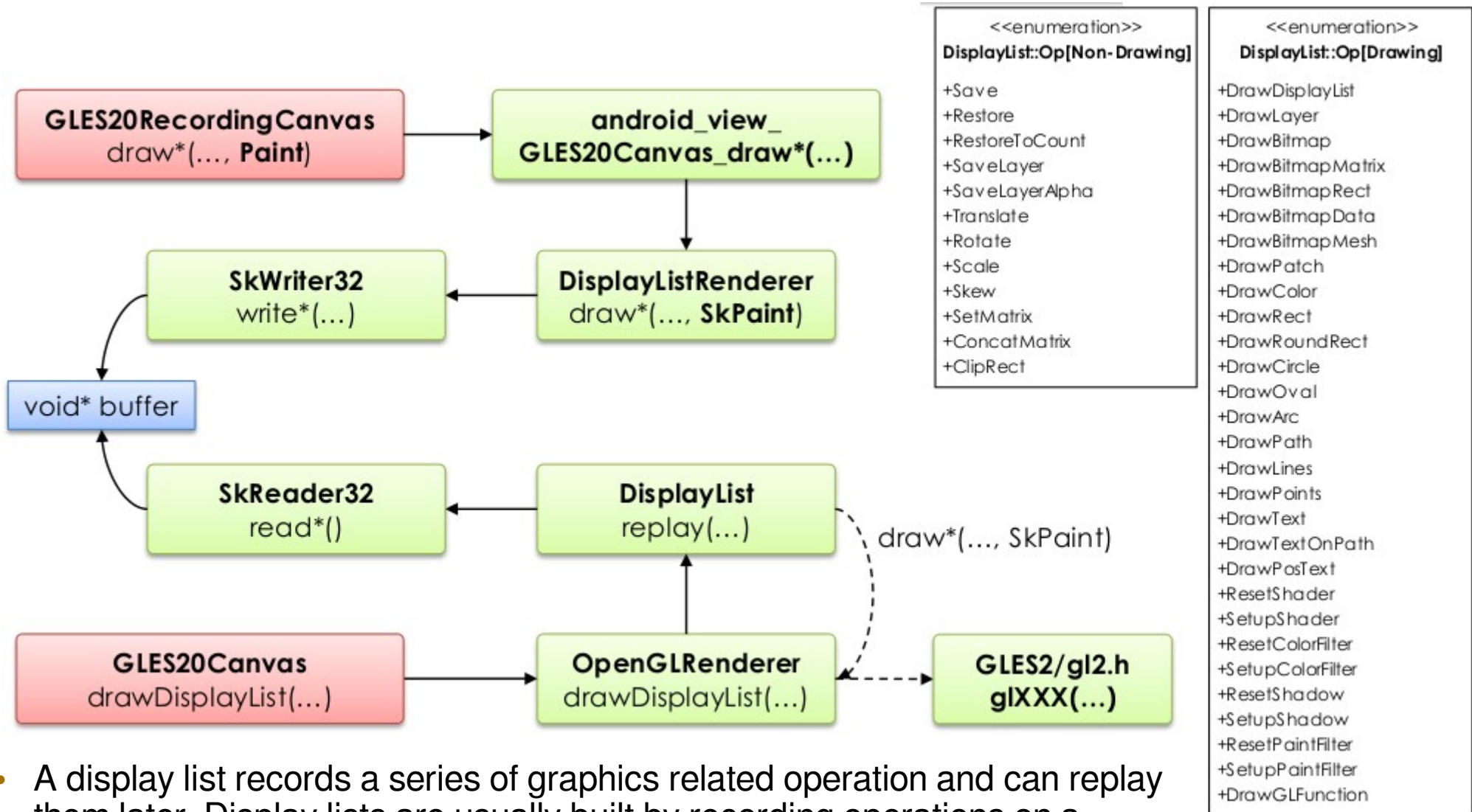


To answer the previous question, we have to learn Display List first

- A display list (or display file) is a series of graphics commands that define an output image. The image is created (rendered) by executing the commands.
- A display list can represent both two- and three-dimensional scenes.
- Systems that make use of a display list to store the scene are called retained mode systems as opposed to immediate mode systems.



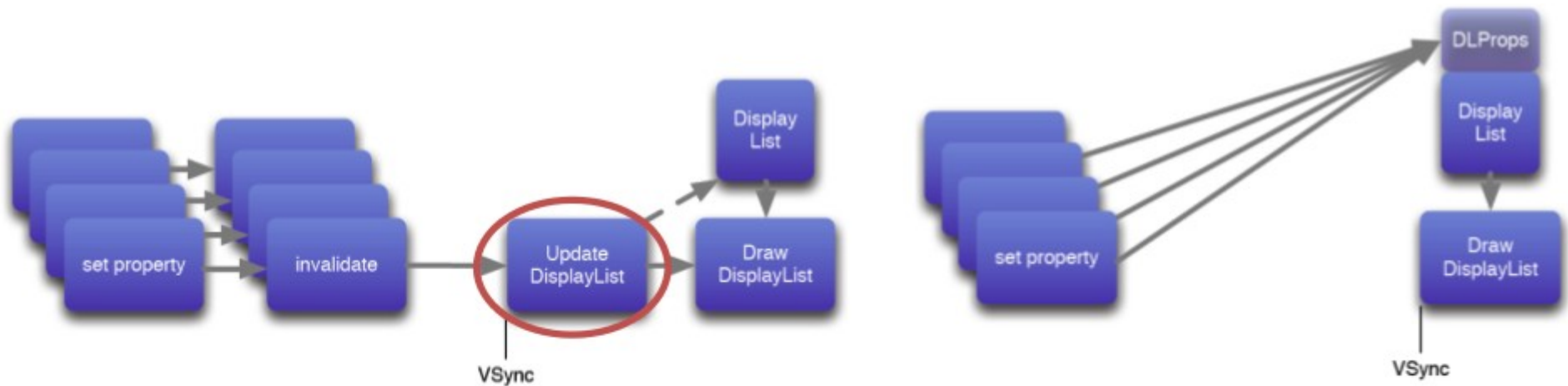
Display List [Android 3.x]



- A display list records a series of graphics related operation and can replay them later. Display lists are usually built by recording operations on a `android.graphics.Canvas`.
- Replaying the operations from a display list avoids executing views drawing code on every frame, and is thus much more efficient.

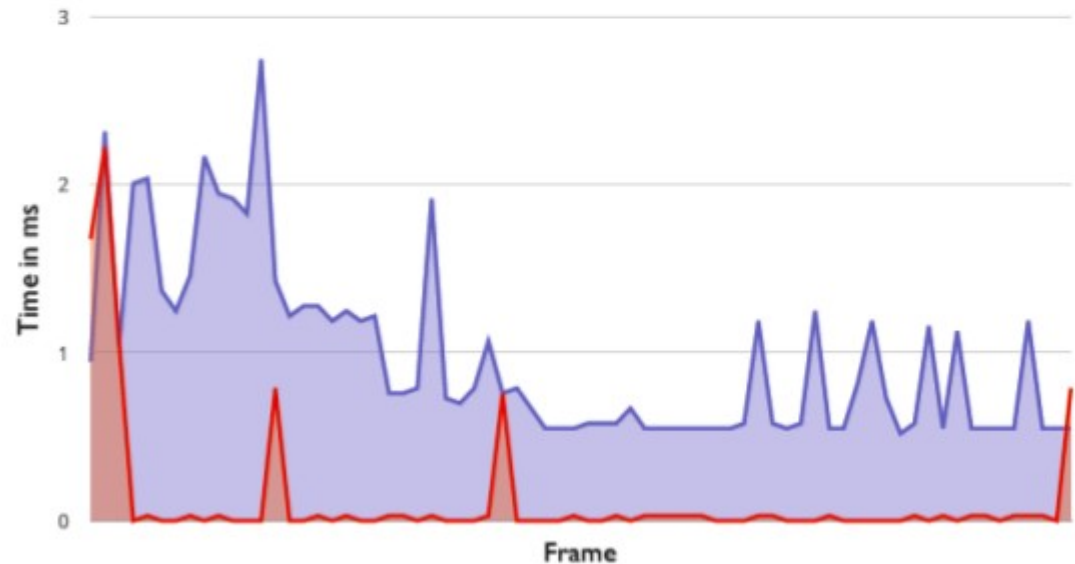


Display List [Android 4.1]



Without DisplayList Properties

With DisplayList Properties



```
+ bool mClipChildren;
+ float mAlpha;
+ int mMultipliedAlpha;
+ bool mHasOverlappingRendering;
+ float mTranslationX, mTranslationY;
+ float mRotation, mRotationX, mRotationY;
+ float mScaleX, mScaleY;
+ float mPivotX, mPivotY;
+ float mCameraDistance;
+ int mLeft, mTop, mRight, mBottom;
+ int mWidth, mHeight;
+ int mPrevWidth, mPrevHeight;
+ bool mPivotExplicitlySet;
+ bool mMatrixDirty;
+ bool mMatrixIsIdentity;
+ uint32_t mMatrixFlags;
+ SkMatrix* mTransformMatrix;
+ Sk3DView* mTransformCamera;
+ SkMatrix* mTransformMatrix3D;
+ SkMatrix* mStaticMatrix;
+ SkMatrix* mAnimationMatrix;
+ bool mCaching;
```

Source: For Butter or Worse, Google I/O 2012

Case study: webkit

RenderObjects, RenderTree, RenderLayers,
Accelerated Compositing, Rendering Flow, Tiled Texture



WebKit Rendering

- RenderObjects
- RenderTree
- RenderLayers



WebKit Rendering - RenderObject

- Each node in the DOM tree that produces visual output has a corresponding RenderObject.
- RenderObjects are stored in a parallel tree structure, called the Render Tree.
- RenderObject knows how to present (paint) the contents of the Node on a display surface.
- It does so by issuing the necessary draw calls to the GraphicsContext associated with the page renderer.
 - GraphicsContext is ultimately responsible for writing the pixels on the bitmap that gets displayed to the screen.



WebKit Rendering - RenderTree

RenderObjects are stored in a parallel tree structure, called Render Tree.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>transform example</title>
</head>
<style type="text/css">
div{
  width: 300px;
  margin: 150px auto;
  background-color: yellow;
  text-align: center;
  -webkit-transform: rotate(45deg);
  -moz-transform: rotate(45deg);
  -o-transform: rotate(45deg);
}
body{
  background-color: green;
}
</style>
<body>
<div>Example Text</div>
</body>
</html>
```

```
RenderView BF at (0,0) size 1600x781
  RenderBlock BF Rt {HTML} at (0,0) size 1600x320
    RenderBody BF {BODY} at (8,150) size 1584x20 [bgcolor=#008000]
      RenderBlock CI BF {DIV} at (642,0) size 300x20 [bgcolor=#FFFF00]
        RenderText {#text} at (108,0) size 84x19
          text run at (108,0) width 84: "Example Text"
```



WebKit Rendering - RenderLayers

- Each RenderObject is associated with a RenderLayer either directly or indirectly via an ancestor RenderObject.
- RenderObjects that share the same coordinate space (e.g. are affected by the same CSS transform) typically belong to the same RenderLayer.
- RenderLayers exist so that the elements of the page are composited in the correct order to properly display overlapping content, semitransparent elements, etc.



RenderLayers

- In general a RenderObject warrants the creation of a RenderLayer if
 - is the root object for the page
 - has explicit CSS position properties (relative, absolute or a transform)
 - is transparent
 - has overflow, an alpha mask or reflection
 - Corresponds to <canvas> element that has a 3D (WebGL) context Corresponds to a <video> element



```
<Layer> at (0,0) size 1600x781 - RenderView BF at (0,0) size 1600x781
  <Layer> at (0,0) size 1600x320 - RenderBlock BF Rt {HTML} at (0,0) size 1600x320
    <Layer> at (650,150) size 300x20 - RenderBlock CI BF {DIV} at (642,0) size 300x20 [bgcolor=#FFFF00]
```

WebKit Rendering

- RenderLayer hierarchy is traversed recursively starting from the root and the bulk of the work is done in `RenderLayer::paintLayer()`.
- `WebView` is the web page encapsulated in a UI component.
- Web page update → the redraw of `WebView`
 - Adjust layers structure according to the latest content and then render/record the updated
 - Render the updated content
- Various approaches of Rendering Architecture
 - Use texture or vector (backing store) as the internal representation
 - multithreaded, multiple processes.



Accelerated Compositing

- Idea: to optimize for cases where an element would be painted to the screen multiple times without its content changing.
 - For example, a menu sliding into the screen, or a static toolbar on top of a video.
- It does so by creating a scene graph, a tree of objects (graphics layers), which have properties attached to them - transformation matrix, opacity, position, effects etc., and also a notification when the layer's content needs to be re-rendered.




Elements = Layers



Fall leaves are beautiful...
on someone else's lawn.

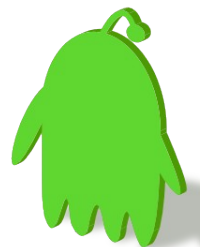
Dino's Lawn Care

 555-2143



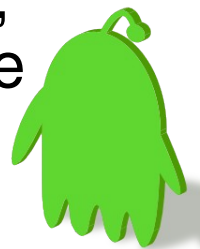


- When accelerated compositing is enabled, some (but not all) of the RenderLayer's get their own backing surface (compositing layer) into which they paint instead of drawing directly into the common bitmap for the page.
- Compositor is responsible for applying the necessary transformations (as specified by the layer's CSS transform properties) to each layer before compositing it.





- Since painting of the layers is decoupled from compositing, invalidating one of these layers only results in repainting the contents of that layer alone and recompositing.

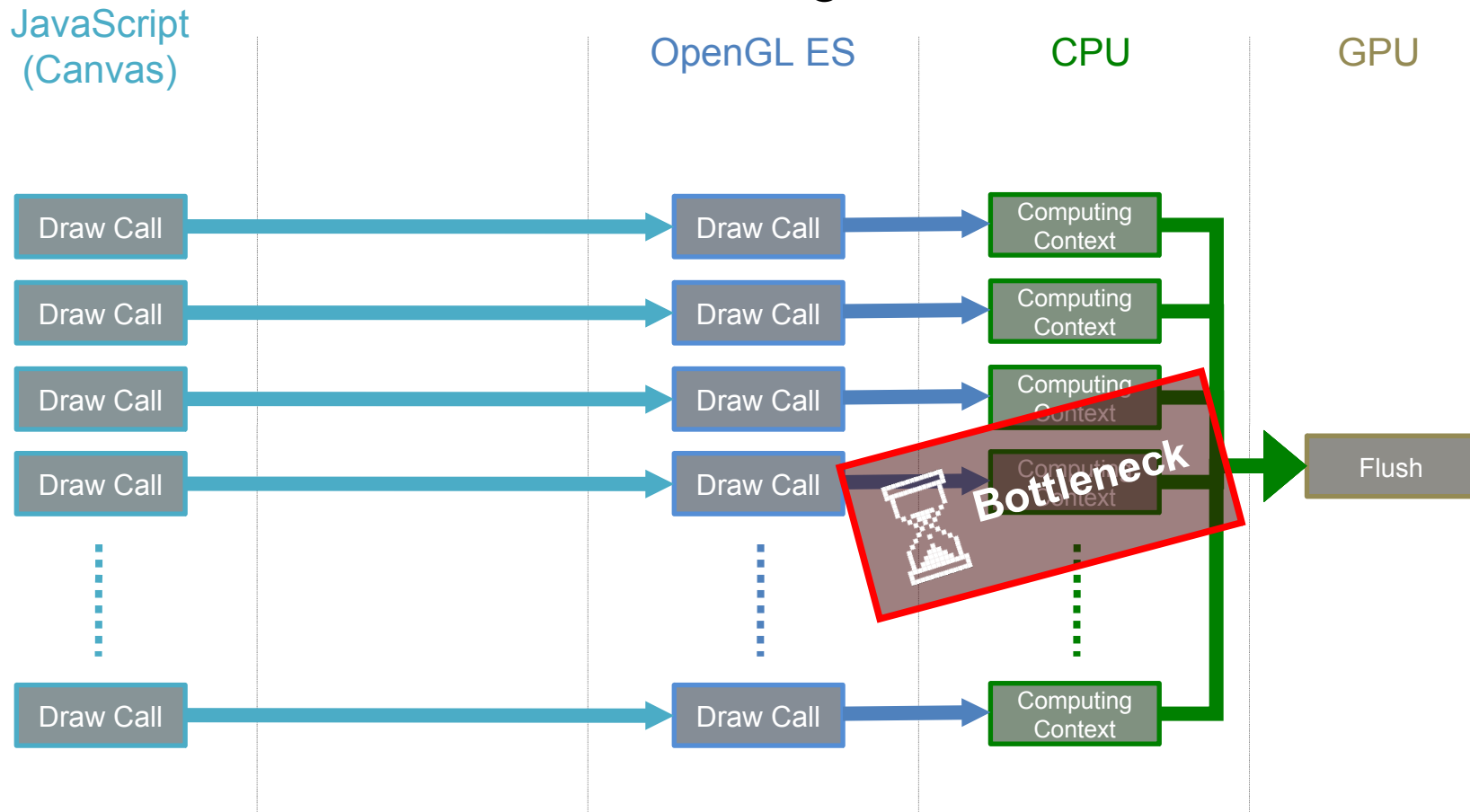


Rendering Flow

- Layers Sync
 - done by WebCore itself
- Layers Compositing
 - done by WebKit port (like Android)
- Android 4.x supports Accelerated Compositing and Hardware Accelerations
 - decided by the property of given Canvas



[skia + gpu; Chrome browser]



- When WebView is redrawn, UI thread performs compositing
 - First, TiledTexture in Root Layer of current Page ViewPort
 - Then, TiledTexture in other BackingLayers
- The generation of TiledTexture can utilize both CPU and GPU.
 - Android 4.0 still uses CPU.

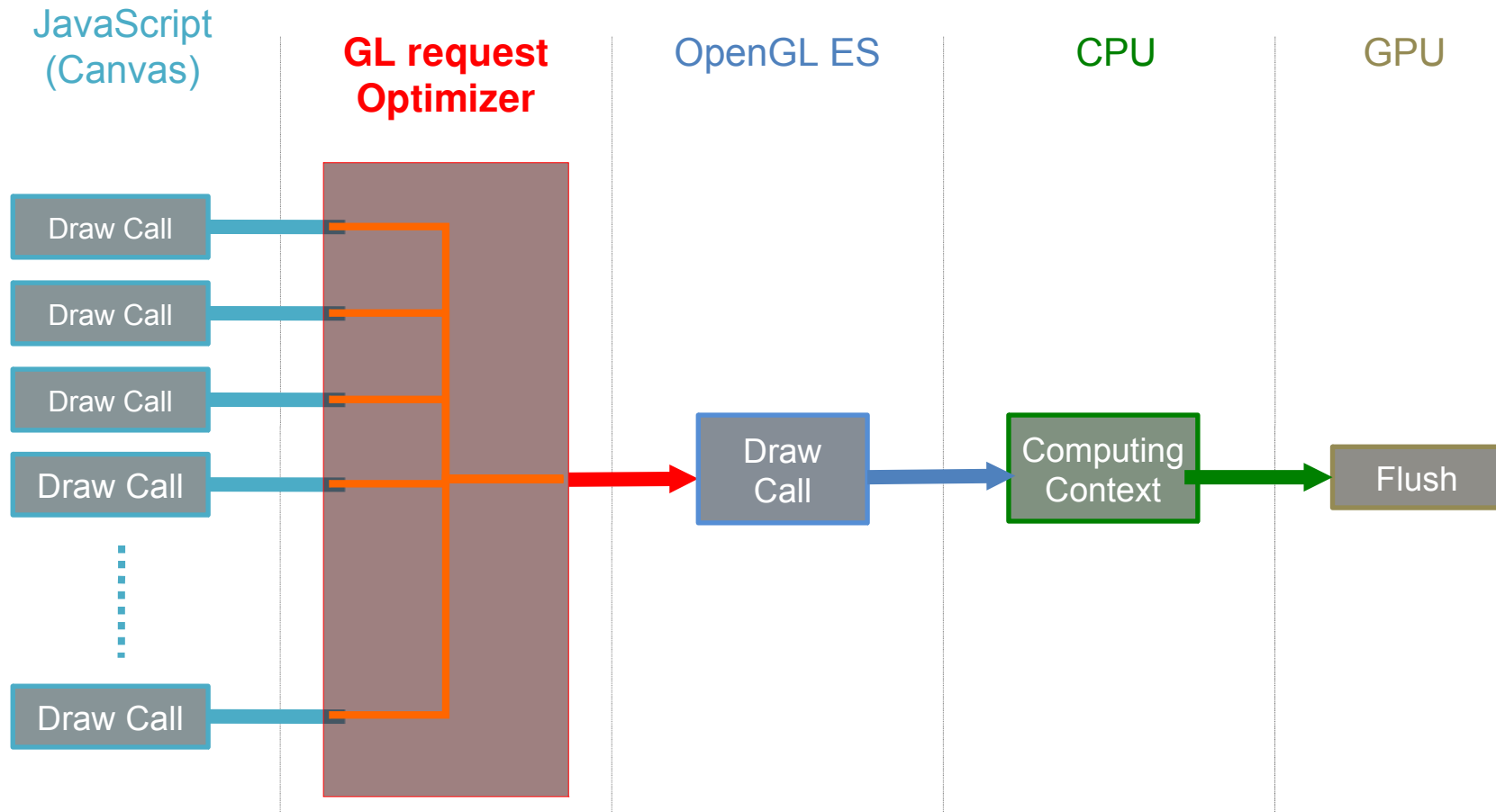


Flow of Generating Tiled Texture

- Using CPU
 - Take one global SkBitmap and reset (size equals to one Tile)
 - Draw SkPicture → global SkBitmap
 - Memory copy from SkBitmap to Graphics Buffer of Tile
- Using GPU
 - All real rendering occurs in TextureGenerator thread
 - Draw the pre-generated textures
 - Page → vector backing store, layer → texture



Techniques to make it better



- improve object lifetime management
- Use GPU specific Backing store implementation
- prefetch optimization for DOM Tree Traversal
- improve texture sharing mechanisms
- Eliminate the loading of U thread



SurfaceFlinger

- Android's window compositor
 - Each window is also a layer.
 - The layers are sorted by Z-order. The Z-order is just the layer type as specified in PhoneWindowManager.java.
- When adding a layer with a Z-order that is already used by some other layer in SurfaceFlinger's list of layers it is put on top of the layers with the same Z-order.
- Even many SurfaceFlinger rendering operations are inherently flat (2D), it uses OpenGL ES 1.1 for rendering
 - May be memory limited on devices with small displays
 - Copybit acceleration may be desirable for UI on some devices
 - deprecated since Android 2.3
 - It is known to improve UX with custom copybit module



(low-level) overhead

- **Composition overhead**
 - Android extensions such as “EGLImage from Android native buffer”, can employ copybit (2D) backend to further offload GPU
 - use non-linear textures for 3D applications to improve memory access locality
- **Native ↔ Java communication overhead**
 - Native code for key operations
 - Can be observed by TraceView tool
- **Cache management overhead**
 - range-based L1 and L2 cache functions (clean, invalidate, flush)
 - Normally uncached graphics memory is sufficient for gaming use cases
 - Cached buffers result in higher performance for CPU rendering in compositing systems



Performance Tips

- Display List is crucial to user experience, but it has to be scheduled properly, otherwise CPU loading gets high unexpectedly.
- Always verify and probe graphics system using Strict Mode
- When hardware acceleration is enable, prevent the following operations from being modified/created frequently:
 - Bitmap, Shape, Paint, Path



Reference

- Skia & FreeType: Android 2D Graphics Essentials, Kyungmin Lee, LG Electronics
- How about some Android graphics true facts? Dianne Hackborn (2011)
- Android 4.0 Graphics and Animations, Romain Guy & Chet Haase (2011)
- Learning about Android Graphics Subsystem, Bhanu Chetlapalli (2012)
- Service 與 Android 系統設計，宋寶華





<http://0xlab.org>