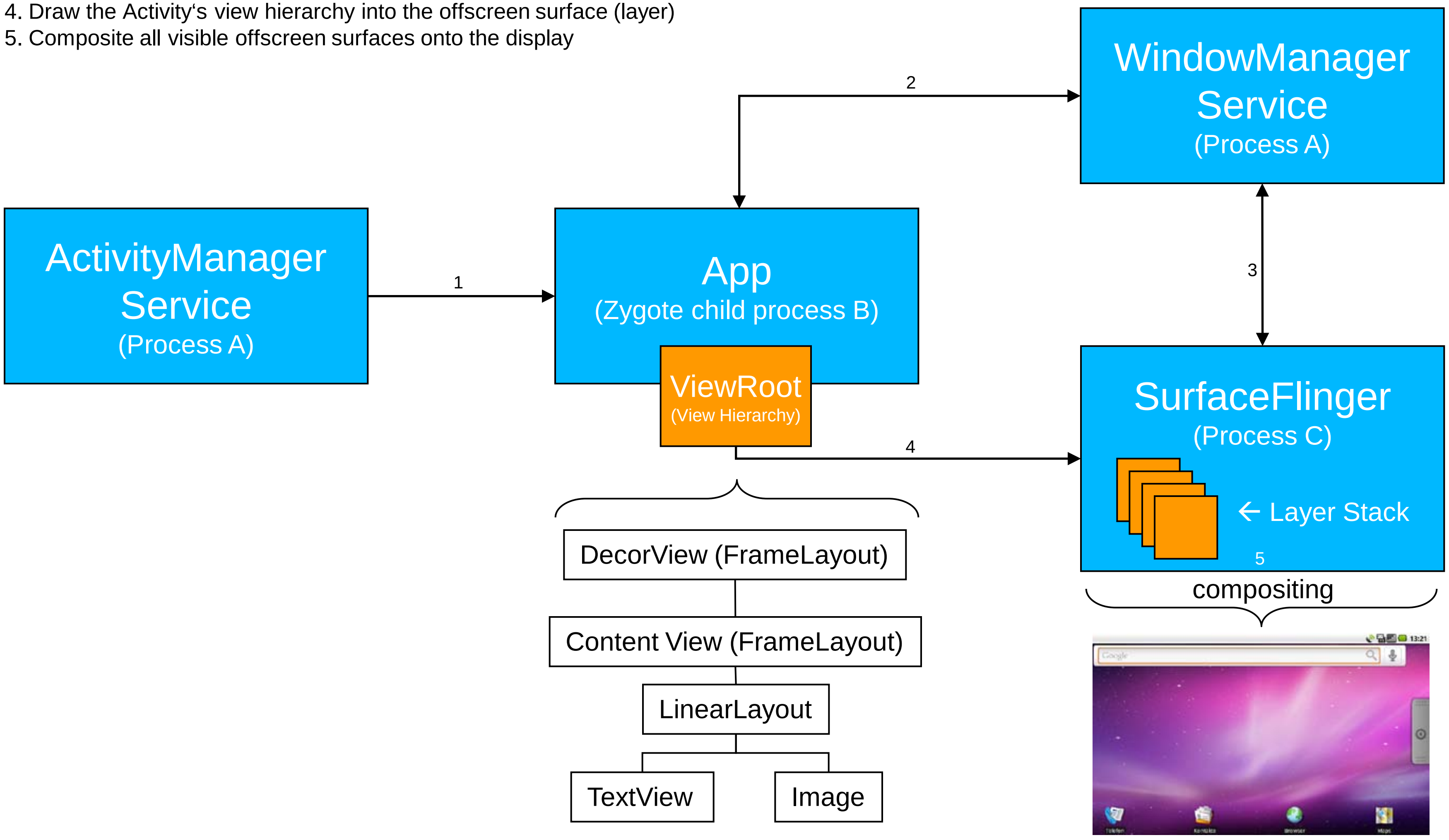
# Android Graphics Architecture I

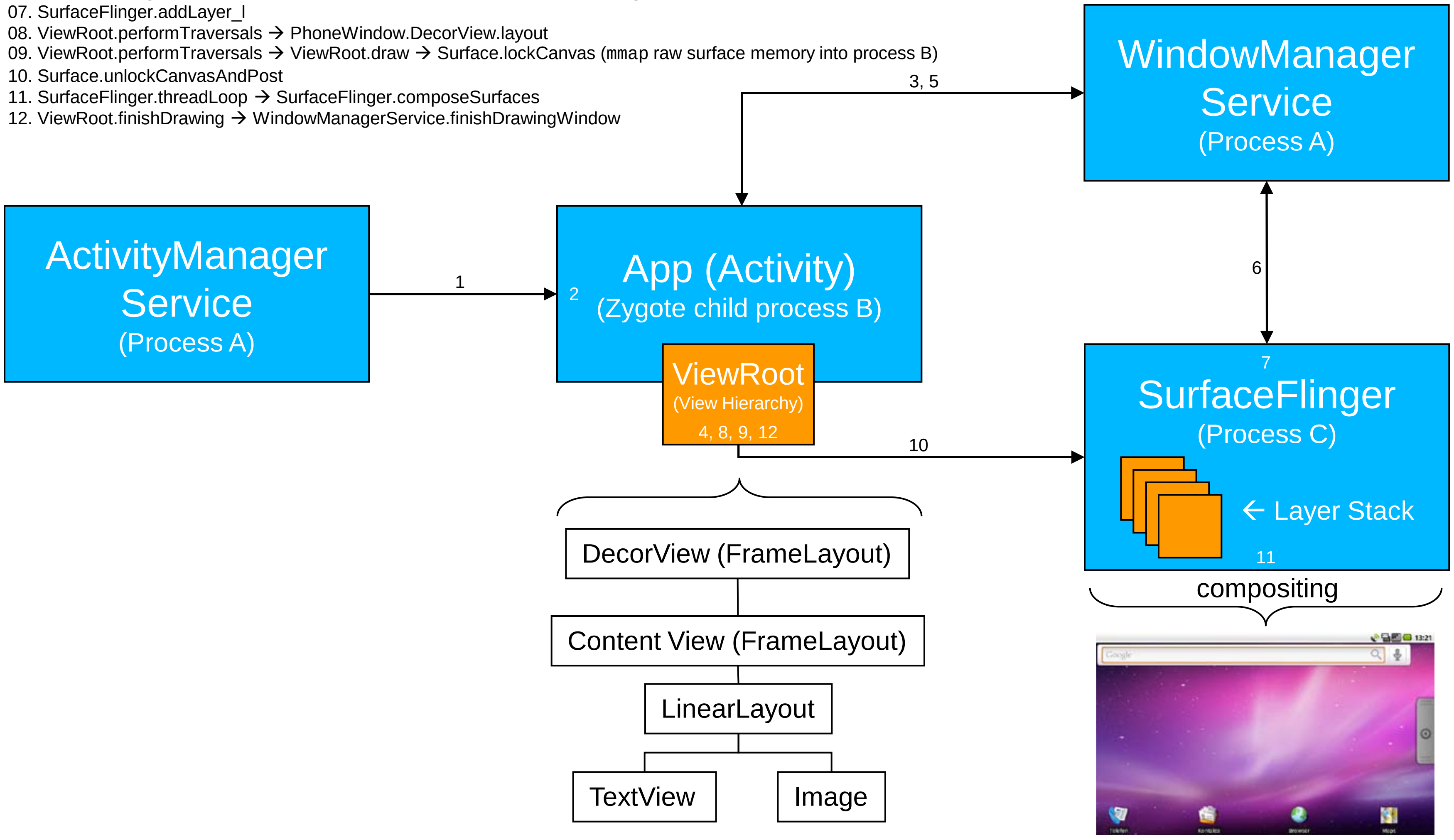
1. Launch a new Activity
2. Create a new window for the Activity and register it at the WindowManagerService
3. Create a new surface for the Activity's window and hand it over to the Activity
4. Draw the Activity's view hierarchy into the offscreen surface (layer)
5. Composite all visible offscreen surfaces onto the display



WindowManager Service (Process A)
2
ActivityManager Service (Process A)
1
App (Zygote child process B)
ViewRoot (View Hierarchy)
3
SurfaceFlinger (Process C)
4
Layer Stack
5
compositing
DecorView (FrameLayout)
Content View (FrameLayout)
LinearLayout
TextView
Image

# Android Graphics Architecture II

01. ActivityManagerService.startActivity
02. Activity.attach → PolicyManager.makeNewWindow
03. ViewRoot.setView → WindowManagerService.addWindow
04. ViewRoot.performTraversals → PhoneWindow.DecorView.measure
05. ViewRoot.relayoutWindow → WindowManagerService.relayoutWindow (fetches the surface created by the WMS also into process B)
06. WindowManagerService.WindowState.createSurfaceLocked → SurfaceFlinger.createSurface
07. SurfaceFlinger.addLayer_l
08. ViewRoot.performTraversals → PhoneWindow.DecorView.layout
09. ViewRoot.performTraversals → ViewRoot.draw → Surface.lockCanvas (`mmap` raw surface memory into process B)
10. Surface.unlockCanvasAndPost
11. SurfaceFlinger.threadLoop → SurfaceFlinger.composeSurfaces
12. ViewRoot.finishDrawing → WindowManagerService.finishDrawingWindow



# Activity.attach

## Call Stack:

Activity.attach (Activity.java)
    Activity.attachBaseContext (PhoneWindow.java)
    PolicyManager.makeNewWindow (PolicyManager.java) → creates the Activity‘s PhoneWindow
    → the UI thread is set to the calling thread which is the Activity‘s main thread
    Window.setWindowManager (Window.java)
        → creates the Windows‘s Window.LocalWindowManager instance

# Activity.setContentView

## Call Stack:

Activity.setContentView (Activity.java)

- PhoneWindow.setContentView (PhoneWindow.java)
  - PhoneWindow.installDecor (PhoneWindow.java)
  - → creates the window's PhoneWindow.DecorView instance
    - PhoneWindow.generateLayout (PhoneWindow.java)
    - → according to the window features the layout resource (window decoration) for the window is chosen
    - → e.g. com.android.internal.R.layout.screen_title chooses screen_title.xml (out/target/common/R/com/android/internal/R.java)
    - → the chosen layout resource is inflated into the PhoneWindow's DecorView
    - → each layout resource must have a „@android:id/content" FrameLayout container
    - → the PhoneWindow's background is set to the window's backdround drawable (normally a black background)
    - → the background could also be set to a transparent ColorDrawable
- → the Activity's layout resource provided as argument to setContentView is inflated into the „@android:id/content" FrameLayout container

# ActivityThread.handleResumeActivity

## Call Stack:

ActivityThread.handleResumeActivity (ActivityThread.java)
- Window.LocalWindowManager.addView (Window.java) → adds the PhoneWindow‘s DecorView to the Window Manager
  - WindowManagerImpl.addView (WindowManagerImpl.java) → creates the window‘s ViewRoot
    - ViewRoot.setView (ViewRoot.java) → sets the PhoneWindow‘s DecorView object to ViewRoot‘s root view (mView) → invalidates the ViewRoot‘s layout by calling ViewRoot.requestLayout
    - IWindowSession.add (IWindowSession.java) → adds the ViewRoot‘s IWindow.Stub (ViewRoot.W) to the WindowManagerService (WindowManagerService.java)

WindowManagerService.addWindow creates the window‘s WindowManagerService.WindowState instance for managing the Activity‘s remote window. E.g. the WindowManagerService delegates input events to the window, does visibility and focus handling for the window, etc.

# Android Graphics - Logs

Start a new Activity from Android‘s Launcher:
→ see „Android Graphics - startActivity Log1.txt“
→ see „Android Graphics - startActivity Log2.txt“

Start another Activity from the previous Activity.
The previous Activity gets invisible.
→ see „Android Graphics - startActivity and make previous Activity invisible.txt”

Start a new Activity from Android‘s Launcher that has a transparent background.
→ see „Android Graphics - startActivity which has a transparent background.txt”

# ViewRoot.performTraversals I

ViewRoot.performTraversals is responsible for drawing the Activity‘s view hierarchy into ViewRoot‘s offscreen surface. ViewRoot.performTraversals always runs inside of of the UI thread‘s context. ViewRoot.performTraversals is called every time a widget or layout manager calls its invalidate or requestLayout method.

## Call Stack:

Processes A, B, C and D. M is the main thread (also the UI thread), B is a Binder thread

```
B, M: ViewRoot.performTraversals (ViewRoot.java) → calls measure, layout and draw on its root view (mView)
B, M:     PhoneWindow.DecorView.measure (PhoneWindow.java)
          → if ViewRoot.performTraversals is called the first time after it was created, resized or has chaned its visibility …
          IWindowSession.relayout is called which then calls into WindowManagerService.relayoutWindow
A, B: WindowManagerService.relayoutWindow (WinodwManagerService.java)
A, B:     WindowManagerService.WindowState.createSurfaceLocked (WinodwManagerService.java)
A, B:         Surface.ctor (Surface.java)
A, B:             Surface.init (android_view_Surface.cpp)
A, B:                 SurfaceComposerClient.createSurface (SurfaceComposerClient.cpp)
A, B:                     BpSurfaceFlingerClient.createSurface (ISurfaceFlingerClient.cpp)
C, B: BnSurfaceFlingerClient.onTransact (ISurfaceFlingerClient.cpp)
C, B:     BClient.createSurface (SurfaceFlinger.cpp)
C, B:         SurfaceFlinger.createSurface (SurfaceFlinger.cpp)
C, B:             SurfaceFlinger.createNormalSurfaceLocked (SurfaceFlinger.cpp)
                  → create new Layer (Layer.cpp) and allocate GraphicBuffers for that Layer
C, B:                 SurfaceFlinger.addLayer_l (SurfaceFlinger.cpp) → add layer into layer stack
```

# ViewRoot.performTraversals II

## Call Stack continued:

Processes A, B, C and D. M is the main thread (also the UI thread), B is a Binder thread

```
C, B: BnSurfaceFlingerClient.onTransact (ISurfaceFlingerClient.cpp)
        → sends back a reference to the newly created surface (ISurface) to the WindowManagerService
A, B:                   BpSurfaceFlingerClient.createSurface (ISurfaceFlingerClient.cpp) → returns the ISurface object
A, B:               SurfaceComposerClient.createSurface (SurfaceComposerClient.cpp)
                    → creates a new SurfaceControll object that wraps the ISurface object
A, B:           Surface.init (android_view_Surface.cpp)
A, B:       Surface.ctor (Surface.java)
A, B:   WindowManagerService.WindowState.createSurfaceLocked (WinodwManagerService.java)
A, B: WindowManagerService.relayoutWindow (WinodwManagerService.java)
        → sends back the newly created surface object to ViewRoot as a parcel → the surface is used as ViewRoot‘s mSurface
B, M:   ViewRoot.relayoutWindow (ViewRoot.java)
B, M: ViewRoot.performTraversals (ViewRoot.java)
B, M:   PhoneWindow.DecorView.layout (PhoneWindow.java)
B, M:   ViewRoot.draw (ViewRoot.java)
B, M:       Surface.lockCanvas (Surface.java) → builds the drawing canvas
B, M:           Surface.lock (Surface.cpp) → lock the underlying GraphicBuffer
                → if the underlying graphic buffer is not already mapped into process B‘s address space this is done inside of
                  gralloc_map (gralloc.cpp and mapper.cpp)
B, M:       → a SkBitmap is initialized with the required format and the and the virtual address of the graphic buffer
B, M:           SkCanvas.setBitmapDevice (SkCanvas.cpp) → the Canvas drawing area is set to the SkBitmap instance
```

# ViewRoot.performTraversals III

## Call Stack continued:

Processes A, B, C and D. M is the main thread (also the UI thread), B is a Binder thread

```
B, M:     PhoneWindow.DecorView.draw (PhoneWindow.java) → craws the entire view hierarchy with the canvas into
          the surface‘s graphic buffer
B, M:     Surface.unlockCanvasAndPost (android_view_Surfrace.cpp)
B, M:       Surface.unlockAndPost (Surface.cpp)
B, M:         SurfaceComposerClient.signalServer (Surface.cpp)
B, M:           BpSurfaceComposer.signal (ISurfaceComposer.cpp)
C, B: BnSurfaceComposer.onTransact (ISurfaceComposer.cpp)
C, B:   SurfaceFlinger.signal (SurfaceFlinger.cpp) → triggers SurfaceFlinger.threadLoop
```

A widget draws itself with the help of the Canvas object provided as argument to the draw method, e.g. …

```
Canvas.drawLine (Canvas.java)
  Canvas.drawLine__FFFFPaint (Canvas.cpp)
    SkCanvas.drawLine (SkCanvas.cpp)
      SkCanvas.drawPoints (SkCanvas.cpp)
        SkDevice.drawPoints (SkDevice.cpp)
          SkDraw.drawPoints (SkDraw.cpp) → chooses an appropriate SkBlitter for the underlying SkBitmap
            SkBlitter.Choose (SkBlitter.cpp) → e.g. chooses an ARGB8888 blitter for drawing if the bitmap is also ARGB
          → draw the geometric shape into the SkBitmap with the chosen SkBlitter
```

## SurfaceFlinger I

The SurfaceFlinger is Android‘s window compositor. Each window is a OpenGL texture. The SurfaceFlinger just blends these OpenGL textures one upon the other. So SurfaceFlinger is completely implemented using OpenGL APIs.

## Call Stack:

Processes A, B, C and D. M is the main thread (also the UI thread), B is a Binder thread

```
C, B: SurfaceFlinger.threadLoop (SurfaceFlinger.cpp)
C, B:     SurfaceFlinger.waitForEvent (SurfaceFlinger.cpp) → block and wait for a redraw event
C, B:     SurfaceFlinger.handlePageFlip (SurfaceFlinger.cpp) → creates OpenGL textures from GraphicBuffers if needed
C, B:     SurfaceFlinger.handleRepaint (SurfaceFlinger.cpp)
C, B:         SurfaceFlinger.composeSurfaces (SurfaceFlinger.cpp) → draws the layer stack
C, B:             LayerBase.draw (LayerBase.cpp)
C, B:                 Layer.onDraw (Layer.cpp)
C, B:                     LayerBase.drawWithOpenGL (LayerBase.cpp) → blends the current layer as OpenGL texture onto the display
C, B:     SurfaceFlinger.postFramebuffer (SurfaceFlinger.cpp)
C, B: … SurfaceFlinger.threadLoop (SurfaceFLinger.cpp)
```

# SurfaceFlinger II

The SurfaceFlinger is 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.

## Call Stack:

WindowManagerService.WindowState.createSurfaceLocked (WindowManagerService.java)
- Surface ctor (Surface.java)
- Surface.openTransaction (Surface.java)
- Surface.setLayer (Surface.java)
- Surface.closeTransaction (Surface.java)
  - SurfaceComposerClient.closeTransaction (SurfaceComposerClient.cpp)
    - SurfaceFlinger.setClientState (SurfaceFlinger.cpp) → sets the new Z-order and reorders the layer list

→ The Z-order is calculated in the WindowManagerService.WindowState.ctor by calling PhoneWindowPolicy.windowTypeToLayerLw

→ The Surface.setLayer method is also called by the WMS.performLayoutAndPlaceSurfacesLockedInner method

→ The layer type of a window is specified in the LayoutParams of the call to WindowManager.addView which triggers ViewRoot.setView …

If a new layer should be added that is drawn above the status bar the following has to be done:

WMS.performLayoutAndPlaceSurfacesLockedInner
- PhoneWindowManager.beginLayoutLw → checks if the StatusBar is visible and computes some rects
- PhoneWindowManager.layoutWindowLw → just before calling WindowState.computeFrameLw the pf, df, cf and vf rectangles for that layer have to be adjusted to full size so the layer overlaps the status bar
- PhoneWindowManager.finishLayoutLw

# Basic Graphic Architecture Adjustments

## Draw Activities with transparent background

- → Give the Activity a transparent background (PhoneWindow.DecorView)
- → This adjustment can be done in PhoneWindow.generateLayout by setting the backdround drawable to a transparent ColorDrawable

ActivityManagerService removes all surfaces behind a fullscreen activity to speed up the surface compositing.
This is done in ActivityManagerService.ensureActivitiesVisibleLocked (`if (r.fullscreen) {…}`).
To deactivate this behaviour comment out the `if(r.fullscreen){…}` block.

## Activity window size

WindowManagerService uses PhoneWindowManager as window manager policy (sPolicy).
To adjust the Activity window sizes one can start with PhoneWindowManager.layoutWindowLw (PhoneWindowManager.java).

# Window Management I

**ViewRoot.performTraversals operates solely on some Activity‘s window and view hierarchy.**
**In contrast, WindowManagerService.performLayoutAndPlaceSurfacesLocked operates on all available windows.**

WMS.performLayoutAndPlaceSurfacesLockedInner is always called by WMS.performLayoutAndPlaceSurfacesLocked.

WindowManagerService.setAppVisibility is only called by ActivityManagerService! So ActivityManagerService controls the visibility of all Activitys.
→ WMS.setAppVisibility adds the app‘s AppWindowToken to mClosingApps if the app is no longer visible. During a WMS.performLayoutAndPlaceSurfacesLockedInner run mClosingApps is processed (WMS.setTokenVisibilityLocked is called on each AppWindowToken in mClosingApps).

If some app finishes (closes) it calls WMS.WindowState.finishExit. This method adds the WindowState‘s surface to mDestroySurface. These surfaces are deleted during a WMS.performLayoutAndPlaceSurfacesLockedInner run.

```
WindowManagerService. performLayoutAndPlaceSurfacesLockedInner (WindowManagerService.java)
    WindowManagerService.WindowState.destroySurfaceLocked (WindowManagerService.java)
        Surface_destroy (android_view_Surface.cpp)
            SurfaceControl.clear (Surface.cpp)
                SurfaceControl.destroy (Surface.cpp)
                    SurfaceComposerClient.destroySurface (SurfaceComposerClient.cpp)
                        BClient.destroySurface (SurfaceFlinger.cpp)
                            SurfaceFlinger.removeSurface (SurfaceFlinger.cpp)
                                SurfaceFlinger.purgatorizeLayer_l (SurfaceFlinger.cpp)
                                    SurfaceFlinger.removeLayer_l (SurfaceFlinger.cpp)
```
→ The second call to SurfaceFlinger.removeLayer_l comes from LayerBaseClient::Surface::~Surface (LayerBase.cpp)

WMS.WindowState.finishExit is either called by WMS.WindowState.stepAnimationLocked or by WMS.AppWindowToken.stepAnimationLocked. Both methods are only called during a WMS.performLayoutAndPlaceSurfacesLockedInner run.

# Window Management II

**WMS.performLayoutAndPlaceSurfacesLocked is called by one of the following methods (conditions):**

• WMS.removeWindowLocked
• WMS.removeWindowInnerLocked
• WMS.setInsetsWindow
• WMS.setWindowWallpaperPositionLocked
• WMS.relayoutWindow → triggered by ViewRoot.performTraversals
• WMS.finishDrawingWindow → triggered by ViewRoot.performTraversals
• WMS.removeWindowToken
• WMS.updateOrientationFromAppTokensUnchecked
• WMS.executeAppTransition → called only by the ActivityManagerService
• WMS.setAppStartingWindow
• WMS.setTokenVisibilityLocked
• WMS.unsetAppFreezingScreenLocked
• WMS.moveAppToken
• WMS.moveAppWindowsLocked
• WMS.setRotationUnchecked
• WMS.H.handleMessage(ANIMATE) → called only by the WMS and WMS.WindowState
• WMS.H.handleMessage(WINDOW_FREEZE_TIMEOUT)
• WMS.H.handleMessage(APP_TRANSITION_TIMEOUT)

For more details see the „Android Graphics – WMS.performLayoutAndPlaceSurfacesLocked.txt“ log file.
In this log WMS.performLayoutAndPlaceSurfacesLocked is called by:
• WMS.executeAppTransition
• WMS.relayoutWindow
• WMS.finishDrawingWindow
• WMS.H.handleMessage(ANIMATE)

# Window Management III

**Window Size and Positioning**

```
WindowManagerService.performLayoutAndPlaceSurfacesLocked
    WindowManagerService.performLayoutAndPlaceSurfacesLockedInner
        WindowManagerService.performLayoutLockedInner
            PhoneWindowManager.beginLayoutLw
            PhoneWindowManager.layoutWindowLw → computes the windows‘s size and position
                WindowManagerService.WindowState.computeFrameLw
            PhoneWindowManager.finishLayoutLw
```

→ ViewRoot gets the window‘s size via WindowManagerService.requestLayout

## Window Management IV

**Input Event Handling**

WindowManagerService.InputDispatcherThread.process → dispatches input events received by EventHub into the apps

## Window Management – Animations I

ActivityManagerService manages activities using HistoryRecords. HistoryRecords extend from IApplicationToken.
WindowManagerService.AppWindowTokens have a IApplicationToken object.
WindowManagerService.AppWindowTokens are created by ActivityManagerService.startActivityLocked while calling WindowManagerService.addAppToken. Here the ActivityManagerService uses some HistoryRecord to initialize the WindowManagerService.AppWindowToken‘s IApplicationToken field.

WindowManagerService.AppWindowTokens are used to control windows even if they are not already visible. E.g. these tokens control the orientation, visibility (WMS.setAppVisibility), animations, AppStartingWindow, etc.
In contrast WindowManagerService.WindowState objects contains the stuff needed by real „visible“ windows. E. g. the surface, size, positioning, etc. WindowManagerService.WindowState object‘s are created by WMS.addWindow.
WindowManagerService. WMS.WindowState also has a WindowManagerService.AppWindowToken field.

WMS.applyAnimationLocked chooses the disired animations
→ WMS.WindowState is responsible for window animations (Status Bar, Dialog, Options Panel, etc.)
→ WMS.AppWindowToken is responsible for activity and wallpaper animations
→ the animations are chosen from framework/base/core/res/res/values/styles.xml → framework/base/core/res/res/anim/*.xml

**WMS.performLayoutAndPlaceSurfacesLocked pass**

1. WindowManagerService.performLayoutLockedInner → compute the window sizes and positions

2. Surface.openTransaction

3. During a WMS.performLayoutAndPlaceSurfacesLocked pass the WMS.WindowState.stepAnimationLocked and WMS.AppWindowToken.stepAnimationLocked methods compute the transformation matrices for the desired animations (see Transformation.getTransformation → Transformation.applyTransformation).

4. Later during that WMS.performLayoutAndPlaceSurfacesLocked pass WMS.WindowState.computeShownFrameLocked computes the combined transformation matrix for that window. It also computes the size and position of the window.

5. Even later during that WMS.performLayoutAndPlaceSurfacesLocked pass Surface.setMatrix is called to transfer the computed transformation matrix to the SurfaceFlinger (LayerBase.setMatrix). Also Surface.setSize, Surface.setPosition and Surface.setAlpha are called to transfer this information into the Surface Flinger (LayerBase.cpp).

6. Surface.closeTransaction → triggers the SurfaceFlinger to perform a SurfaceFlinger.threadLoop pass.

7. At the end of WMS.performLayoutAndPlaceSurfacesLocked this method tiggers itself with a frequency of 60 Hz as long as some animation is running. This is done via WMS.requestAnimationLocked. WMS.requestAnimationLocked is called as long as WMS.WindowState.stepAnimationLocked or WMS.AppWindowToken.stepAnimationLocked return true.

## Window Management – Animations II

**SurfaceFlinger.threadLoop pass**
Meanwhile the SurfaceFlinger calls LayerBase.validateVisibility during a SurfaceFlinger.threadLoop pass. Here the new transformation matrix is applied and the new vertex coordinates for the texture mapping are calculated. These vertex coordinates are later used during this SurfaceFlinger.threadLoop pass to draw the window in LayerBase.drawWithOpenGL. The calculated vertexes are used in the call to glVertexPointer.

The texture coordinates used in LayerBase.drawWithOpenGL are multiplied by 65536 (1<<16) because the Skia framework multiplies transformation coordinates by 65536 transform from floating point arithmetic to fixed point arithmetic (see SkScalar.h and SkFixed.h).

The WindowManagerService is able to animate so called AppStartingWindows while the real app along with its window is starting up. The AppStartingWindow just displays an empty window (only window decoration and theme are applied) that is animated. To disable the AppStartingWindow set ActivityManagerService.SHOW_APP_STARTING_ICON to false.

→ See also „WindowManagerService animations discussion with Dianne Hackborn.txt“

## OpenGL Overlays

GLSurfaceView.java extends from SurfaceView.java

SurfaceView creates its own instance of class Surface and contains an IWindowSession object to communicate with the window manager.

SurfaceView.updateWindow adds the window to the WindowManagerService and also calls IWindowSession.relayout to allocate the surface‘s framebuffer memory just like ViewRoot does.

GLSurfaceView.EglHelper.createSurface then uses this surface framebuffer memory to initialize its EGL window surface using EGLWindowSurfaceFactory.createWindowSurface (→ EGLImpl.eglCreateWindowSurface).

Using SurfaceView.setZOrderOnTop one can also adjust the Z-order which the SurfaceFlinger uses to composite the multiple surfaces of the activity.