DrawPath

Java onDraw->SkLite이
(op명령어 생성)

```
19034                 || !renderNode.isValid()
19035                 || (mRecreateDisplayList)) {
19036             // Don't need to recreate the display list, just need to tell our
19037             // children to restore/recreate theirs
19038             if (renderNode.isValid()
19039                     && !mRecreateDisplayList) {
19040                 mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
19041                 mPrivateFlags &= ~PFLAG_DIRTY_MASK;
19042                 dispatchGetDisplayList();
19043 
19044                 return renderNode; // no work needed
19045             }
19046 
19047             // If we got here, we're recreating it. Mark it as such to ensure that
19048             // we copy in child display lists into ours in drawChild()
19049             mRecreateDisplayList = true;
19050 
19051             int width = mRight - mLeft;
19052             int height = mBottom - mTop;
19053             int layerType = getLayerType();
19054 
19055             final DisplayListCanvas canvas = renderNode.start(width, height);
19056 
19057             try {
19058                 if (layerType == LAYER_TYPE_SOFTWARE) {
19059                     buildDrawingCache(true);
19060                     Bitmap cache = getDrawingCache(true);
19061                     if (cache != null) {
19062                         canvas.drawBitmap(cache, 0, 0, mLayerPaint);
19063                     }
19064                 } else {
19065                     computeScroll();
19066 
19067                     canvas.translate(-mScrollX, -mScrollY);
19068                     mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
19069                     mPrivateFlags &= ~PFLAG_DIRTY_MASK;
19070 
19071                     // Fast path for layouts with no backgrounds
19072                     if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
19073                         dispatchDraw(canvas);
19074                         drawAutofilledHighlight(canvas);
19075                         if (mOverlay != null && !mOverlay.isEmpty()) {
19076                             mOverlay.getOverlayView().draw(canvas);
19077                         }
19078                         if (debugDraw()) {
19079                             debugDrawFocus(canvas);
19080                         }
19081                     } else {
19082                         draw(canvas);
19083                     }
19084                 }
19085             } finally {
19086                 renderNode.end(canvas);
19087                 setDisplayListProperties(renderNode);
19088             }
19089         } else {
19090             mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
19091             mPrivateFlags &= ~PFLAG_DIRTY_MASK;
19092         }
19093         return renderNode;
19094     }
19095 
19096     private void resetDisplayList() {
```

xref: /frameworks/base/core/java/android/view/View.java

Home | History | Annotate | Line# | Navigate | Download Search only in View

```
20175     @CallSuper
20176     public void draw(Canvas canvas) {
20177         final int privateFlags = mPrivateFlags;
20178         final boolean dirtyOpaque = (privateFlags & PFLAG_DIRTY_MASK) == PFLAG_DIRTY_OPAQUE &&
20179                 (mAttachInfo == null || !mAttachInfo.mIgnoreDirtyState);
20180         mPrivateFlags = (privateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DRAWN;
20181 
20182         /*
20183          * Draw traversal performs several drawing steps which must be executed
20184          * in the appropriate order:
20185          *
20186          *      1. Draw the background
20187          *      2. If necessary, save the canvas' layers to prepare for fading
20188          *      3. Draw view's content
20189          *      4. Draw children
20190          *      5. If necessary, draw the fading edges and restore layers
20191          *      6. Draw decorations (scrollbars for instance)
20192          */
20193 
20194         // Step 1, draw the background, if needed
20195         int saveCount;
20196 
20197         if (!dirtyOpaque) {
20198             drawBackground(canvas);
20199         }
20200 
20201         // skip step 2 & 5 if possible (common case)
20202         final int viewFlags = mViewFlags;
20203         boolean horizontalEdges = (viewFlags & FADING_EDGE_HORIZONTAL) != 0;
20204         boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;
20205         if (!verticalEdges && !horizontalEdges) {
20206             // Step 3, draw the content
20207             if (!dirtyOpaque) onDraw(canvas);
20208 
20209             // Step 4, draw the children
20210             dispatchDraw(canvas);
20211 
20212             drawAutofilledHighlight(canvas);
```

xref: /frameworks/base/core/java/android/view/View.java

Home | History | Annotate | Line# | Navigate | Download Search only in View

```
20175     @CallSuper
20176     public void draw(Canvas canvas) {
20177         final int privateFlags = mPrivateFlags;
20178         final boolean dirtyOpaque = (privateFlags & PFLAG_DIRTY_MASK) == PFLAG_DIRTY_OPAQUE &&
20179                 (mAttachInfo == null || !mAttachInfo.mIgnoreDirtyState);
20180         mPrivateFlags = (privateFlags & ~PFLAG_DIRTY_MASK) | PFLAG_DRAWN;
20181
20182         /*
20183          * Draw traversal performs several drawing steps which must be executed
20184          * in the appropriate order:
20185          *
20186          *      1. Draw the background
20187          *      2. If necessary, save the canvas' layers to prepare for fading
20188          *      3. Draw view's content
20189          *      4. Draw children
20190          *      5. If necessary, draw the fading edges and restore layers
20191          *      6. Draw decorations (scrollbars for instance)
20192          */
20193
20194         // Step 1, draw the background, if needed
20195         int saveCount;
20196
20197         if (!dirtyOpaque) {
20198             drawBackground(canvas);
20199         }
20200
20201         // skip step 2 & 5 if possible (common case)
20202         final int viewFlags = mViewFlags;
20203         boolean horizontalEdges = (viewFlags & FADING_EDGE_HORIZONTAL) != 0;
20204         boolean verticalEdges = (viewFlags & FADING_EDGE_VERTICAL) != 0;
20205         if (!verticalEdges && !horizontalEdges) {
20206             // Step 3, draw the content
20207             if (!dirtyOpaque) onDraw(canvas);
20208
20209             // Step 4, draw the children
20210             dispatchDraw(canvas);
20211
20212             drawAutofilledHighlight(canvas);
```

중간 생략

xref: /frameworks/base/core/java/android/view/RecordingCanvas.java

Home | History | Annotate | Line# | Navigate | Download Search only

```
279     @Override
280     public final void drawPath(@NonNull Path path, @NonNull Paint paint) {
281         if (path.isSimplePath && path.rects != null) {
282             nDrawRegion(mNativeCanvasWrapper, path.rects.mNativeRegion, paint.getNativeInstance());
283         } else {
284             nDrawPath(mNativeCanvasWrapper, path.readOnlyNI(), paint.getNativeInstance());
285         }
286     }
287
```

xref: /frameworks/base/core/java/android/view/RecordingCanvas.java

Home | History | Annotate | Line# | Navigate | Download Search only

```
279     @Override
280     public final void drawPath(@NonNull Path path, @NonNull Paint paint) {
281         if (path.isSimplePath && path.rects != null) {
282             nDrawRegion(mNativeCanvasWrapper, path.rects.mNativeRegion, paint.getNativeInstance());
283         } else {
284             nDrawPath(mNativeCanvasWrapper, path.readOnlyNI(), paint.getNativeInstance());
285         }
286     }
287
```

RecordingCanvas.java -> android_graphics_Canvas.cpp (jni)

xref: /frameworks/base/core/jni/android_graphics_Canvas.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in

```
306
307 static void drawPath(JNIEnv* env, jobject, jlong canvasHandle, jlong pathHandle,
308                      jlong paintHandle) {
309     const SkPath* path = reinterpret_cast<SkPath*>(pathHandle);
310     const Paint* paint = reinterpret_cast<Paint*>(paintHandle);
311     get_canvas(canvasHandle)->drawPath(*path, *paint);
312 }
```

xref: /frameworks/base/core/jni/android_graphics_Canvas.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in

```
306
307 static void drawPath(JNIEnv* env, jobject, jlong canvasHandle, jlong pathHandle,
308                       jlong paintHandle) {
309     const SkPath* path = reinterpret_cast<SkPath*>(pathHandle);
310     const Paint* paint = reinterpret_cast<Paint*>(paintHandle);
311     get_canvas(canvasHandle)->drawPath(*path, *paint);
312 }
```

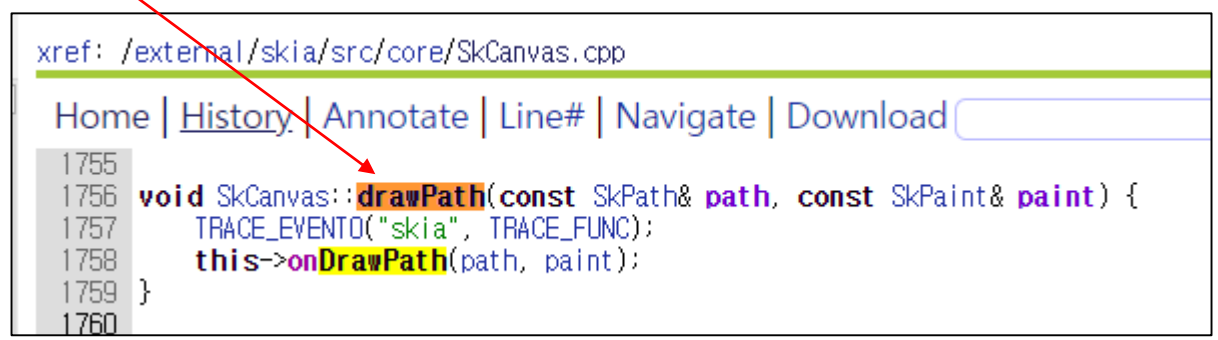
android_graphics_Canvas.cpp (jni) - >SkiaCanvas.cpp

xref: /frameworks/base/libs/hwui/SkiaCanvas.cpp

Home | History | Annotate | Line# | Navigate | Download Search

```
527
528 void SkiaCanvas::drawPath(const SkPath& path, const SkPaint& paint) {
529     if (CC_UNLIKELY(paint.nothingToDraw())) return;
530     if (CC_UNLIKELY(path.isEmpty() && (!path.isInverseFillType()))) {
531         return;
532     }
533     mCanvas->drawPath(path, paint);
534 }
```

SkiaCanvas.cpp - > SkCanvas.cpp (external)



xref: /external/skia/src/core/SkCanvas.cpp

Home | History | Annotate | Line# | Navigate | Download

```
1755
1756 void SkCanvas::drawPath(const SkPath& path, const SkPaint& paint) {
1757     TRACE_EVENT0("skia", TRACE_FUNC);
1758     this->onDrawPath(path, paint);
1759 }
1760
```

SkCanvas.cpp (external)

xref: /external/skia/src/core/SkLiteRecorder.cpp

Home | History | Annotate | Line# | Navigate | Download

```
66 void SkLiteRecorder::onDrawPath(const SkPath& path, const SkPaint& paint) {
67     fDL->drawPath(path, paint);
68 }
```

xref: /external/skia/src/core/SkCanvas.cpp

Home | History | Annotate | Line# | Navigate | Download

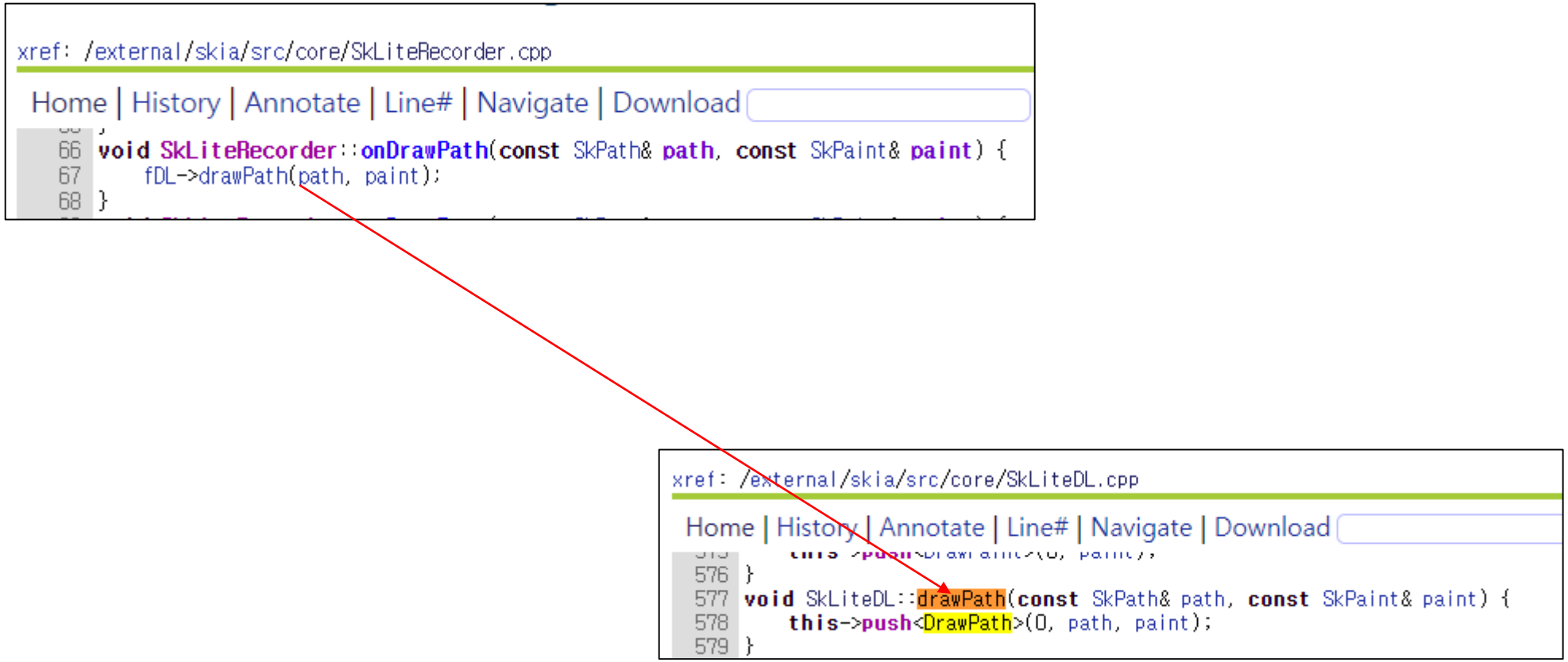
```
1755
1756 void SkCanvas::drawPath(const SkPath& path, const SkPaint& paint) {
1757     TRACE_EVENT0("skia", TRACE_FUNC);
1758     this->onDrawPath(path, paint);
1759 }
1760
```

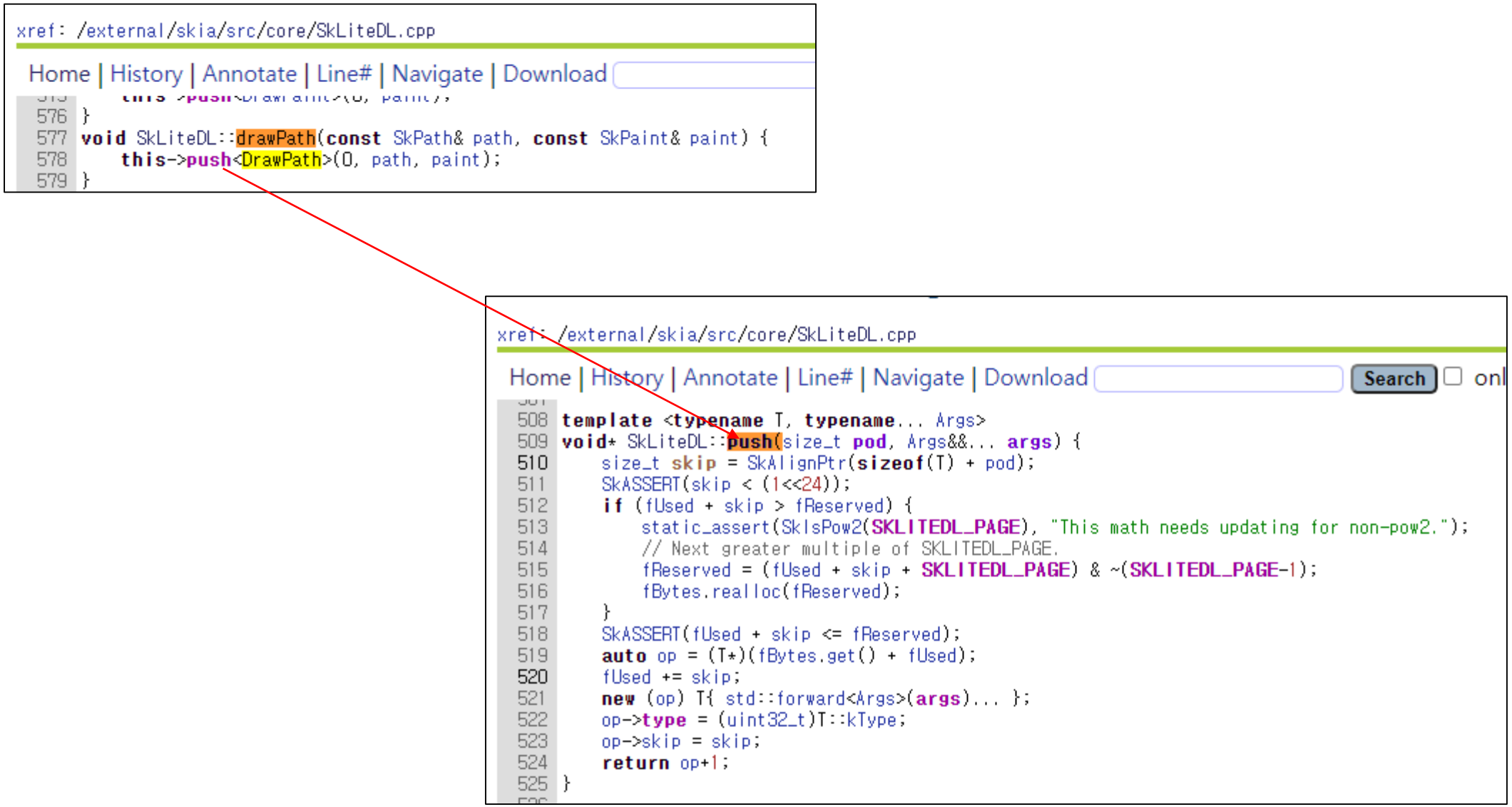
SkCanvas.cpp (external)

xref: /external/skia/src/core/SkLiteRecorder.cpp

Home | History | Annotate | Line# | Navigate | Download

```
66 void SkLiteRecorder::onDrawPath(const SkPath& path, const SkPaint& paint) {
67     fDL->drawPath(path, paint);
68 }
```


xref: /external/skia/src/core/SkLiteRecorder.cpp
Home | History | Annotate | Line# | Navigate | Download
66 void SkLiteRecorder::onDrawPath(const SkPath& path, const SkPaint& paint) {
67 fDL->drawPath(path, paint);
68 }
xref: /external/skia/src/core/SkLiteDL.cpp
Home | History | Annotate | Line# | Navigate | Download
576 }
577 void SkLiteDL::drawPath(const SkPath& path, const SkPaint& paint) {
578 this->push<DrawPath>(0, path, paint);
579 }


xref: /external/skia/src/core/SkLiteDL.cpp
Home | History | Annotate | Line# | Navigate | Download
576 }
577 void SkLiteDL::drawPath(const SkPath& path, const SkPaint& paint) {
578 this->push<DrawPath>(0, path, paint);
579 }
xref: /external/skia/src/core/SkLiteDL.cpp
Home | History | Annotate | Line# | Navigate | Download
Search
508 template <typename T, typename... Args>
509 void* SkLiteDL::push(size_t pod, Args&&... args) {
510 size_t skip = SkAlignPtr(sizeof(T) + pod);
511 SkASSERT(skip < (1<<24));
512 if (fUsed + skip > fReserved) {
513 static_assert(SkIsPow2(SKLITEDL_PAGE), "This math needs updating for non-pow2.");
514 // Next greater multiple of SKLITEDL_PAGE.
515 fReserved = (fUsed + skip + SKLITEDL_PAGE) & ~(SKLITEDL_PAGE-1);
516 fBytes.realloc(fReserved);
517 }
518 SkASSERT(fUsed + skip <= fReserved);
519 auto op = (T*)(fBytes.get() + fUsed);
520 fUsed += skip;
521 new (op) T{ std::forward<Args>(args)... };
522 op->type = (uint32_t)T::kType;
523 op->skip = skip;
524 return op+1;
525 }

DrawPath

Java-> native

(DisplayList 업데이트)

```
19034                 || !renderNode.isValid()
19035                 || (mRecreateDisplayList)) {
19036             // Don't need to recreate the display list, just need to tell our
19037             // children to restore/recreate theirs
19038             if (renderNode.isValid()
19039                     && !mRecreateDisplayList) {
19040                 mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
19041                 mPrivateFlags &= ~PFLAG_DIRTY_MASK;
19042                 dispatchGetDisplayList();
19043 
19044                 return renderNode; // no work needed
19045             }
19046 
19047             // If we got here, we're recreating it. Mark it as such to ensure that
19048             // we copy in child display lists into ours in drawChild()
19049             mRecreateDisplayList = true;
19050 
19051             int width = mRight - mLeft;
19052             int height = mBottom - mTop;
19053             int layerType = getLayerType();
19054 
19055             final DisplayListCanvas canvas = renderNode.start(width, height);
19056 
19057             try {
19058                 if (layerType == LAYER_TYPE_SOFTWARE) {
19059                     buildDrawingCache(true);
19060                     Bitmap cache = getDrawingCache(true);
19061                     if (cache != null) {
19062                         canvas.drawBitmap(cache, 0, 0, mLayerPaint);
19063                     }
19064                 } else {
19065                     computeScroll();
19066 
19067                     canvas.translate(-mScrollX, -mScrollY);
19068                     mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
19069                     mPrivateFlags &= ~PFLAG_DIRTY_MASK;
19070 
19071                     // Fast path for layouts with no backgrounds
19072                     if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
19073                         dispatchDraw(canvas);
19074                         drawAutofilledHighlight(canvas);
19075                         if (mOverlay != null && !mOverlay.isEmpty()) {
19076                             mOverlay.getOverlayView().draw(canvas);
19077                         }
19078                         if (debugDraw()) {
19079                             debugDrawFocus(canvas);
19080                         }
19081                     } else {
19082                         draw(canvas);
19083                     }
19084                 }
19085             } finally {
19086                 renderNode.end(canvas);
19087                 setDisplayListProperties(renderNode);
19088             }
19089         } else {
19090             mPrivateFlags |= PFLAG_DRAWN | PFLAG_DRAWING_CACHE_VALID;
19091             mPrivateFlags &= ~PFLAG_DIRTY_MASK;
19092         }
19093         return renderNode;
19094     }
19095 
19096     private void resetDisplayList() {
```

xref: /frameworks/base/core/java/android/view/RenderNode.java

Home | History | Annotate | Line# | Navigate | Download

```
228      */
229     public void end(DisplayListCanvas canvas) {
230         long displayList = canvas.finishRecording();
231         nSetDisplayList(mNativeRenderNode, displayList);
232         canvas.recycle();
233     }
234 
```

xref: /frameworks/base/core/java/android/view/RenderNode.java

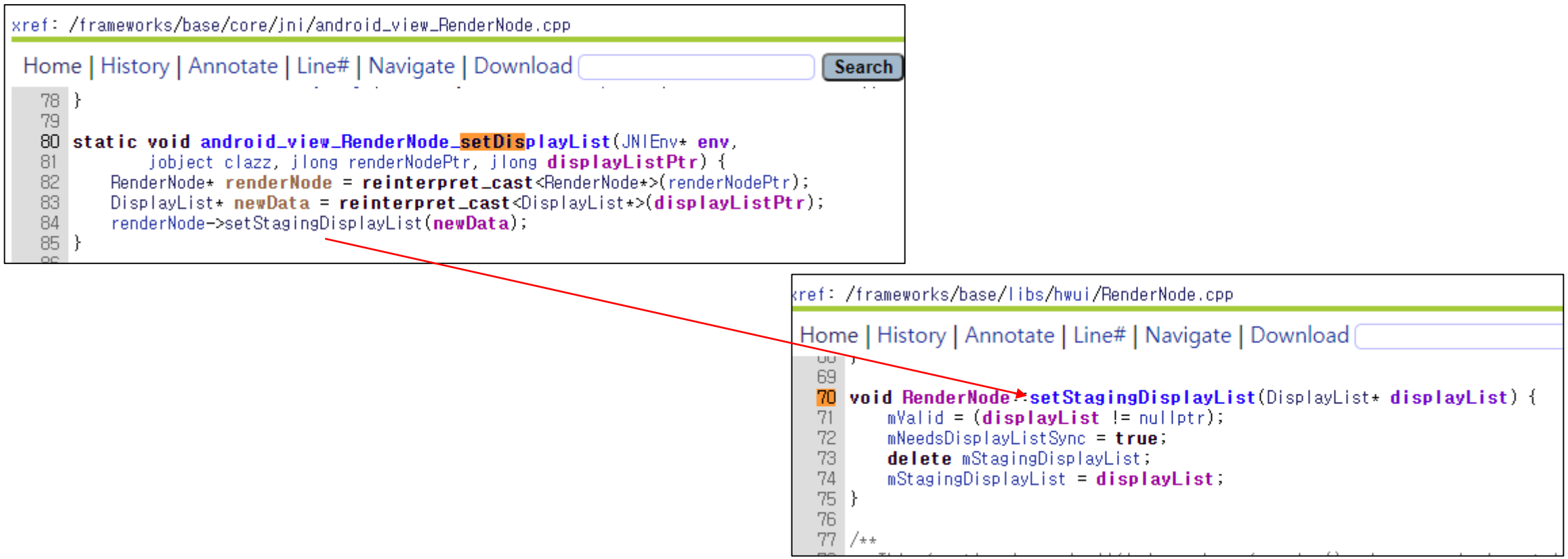
Home | History | Annotate | Line# | Navigate | Download

```
228      */
229     public void end(DisplayListCanvas canvas) {
230         long displayList = canvas.finishRecording();
231         nSetDisplayList(mNativeRenderNode, displayList);
232         canvas.recycle();
233     }
234
```

xref: /frameworks/base/core/jni/android_view_RenderNode.cpp

Home | History | Annotate | Line# | Navigate | Download Search

```
78 }
79
80 static void android_view_RenderNode_setDisplayList(JNIEnv* env,
81         jobject clazz, jlong renderNodePtr, jlong displayListPtr) {
82     RenderNode* renderNode = reinterpret_cast<RenderNode*>(renderNodePtr);
83     DisplayList* newData = reinterpret_cast<DisplayList*>(displayListPtr);
84     renderNode->setStagingDisplayList(newData);
85 }
```


xref: /frameworks/base/core/jni/android_view_RenderNode.cpp
Home | History | Annotate | Line# | Navigate | Download
Search
78 }
79
80 static void android_view_RenderNode_setDisplayList(JNIEnv* env,
81 jobject clazz, jlong renderNodePtr, jlong displayListPtr) {
82 RenderNode* renderNode = reinterpret_cast<RenderNode*>(renderNodePtr);
83 DisplayList* newData = reinterpret_cast<DisplayList*>(displayListPtr);
84 renderNode->setStagingDisplayList(newData);
85 }
xref: /frameworks/base/libs/hwui/RenderNode.cpp
Home | History | Annotate | Line# | Navigate | Download
69
70 void RenderNode::setStagingDisplayList(DisplayList* displayList) {
71 mValid = (displayList != nullptr);
72 mNeedsDisplayListSync = true;
73 delete mStagingDisplayList;
74 mStagingDisplayList = displayList;
75 }
76
77 /**

xref: /frameworks/base/libs/hwui/RenderNode.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in **RenderNode.cpp**

```
343
344 void RenderNode::syncDisplayList(TreeObserver& observer, TreeInfo* info) {
345     // Make sure we inc first so that we don't fluctuate between 0 and 1,
346     // which would thrash the layer cache
347     if (mStagingDisplayList) {
348         mStagingDisplayList->updateChildren([](RenderNode* child) { child->incParentRefCount(); });
349     }
350     deleteDisplayList(observer, info);
351     mDisplayList = mStagingDisplayList;
352     mStagingDisplayList = nullptr;
353     if (mDisplayList) {
354         mDisplayList->syncContents();
355     }
356 }
```

DrawPath

CanvasContext->SkLiteDL draw
(그리기 작업)

xref: /frameworks/base/libs/hwui/renderthread/CanvasContext.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in Canvas

```
439 void CanvasContext::draw() {
440     SkRect dirty;
441     mDamageAccumulator.finish(&dirty);
442
443     // TODO: Re-enable after figuring out cause of b/22592975
444     //     if (dirty.isEmpty() && Properties::skipEmptyFrames) {
445     //         mCurrentFrameInfo->addFlag(FrameInfoFlags::SkippedFrame);
446     //         return;
447     //     }
448
449     mCurrentFrameInfo->markIssueDrawCommandsStart();
450
451     Frame frame = mRenderPipeline->getFrame();
452
453     SkRect windowDirty = computeDirtyRect(frame, &dirty);
454
455     bool drew = mRenderPipeline->draw(frame, windowDirty, dirty, mLightGeometry, &mLayerUpdateQueue,
456                                       mContentDrawBounds, mOpaque, mWideColorGamut, mLightInfo,
457                                       mRenderNodes, &(profiler()));
458
459     int64_t frameCompleteNr = mFrameCompleteCallbacks.size() ? getFrameNumber() : -1;
460
461     waitOnFences();
462
463     bool requireSwap = false;
464     bool didSwap =
465             mRenderPipeline->swapBuffers(frame, drew, windowDirty, mCurrentFrameInfo, &requireSwap);
466
467     mIsDirty = false;
468
469     if (requireSwap) {
470         if (!didSwap) {  // some error happened
471             setSurface(nullptr);
472         }
473         SwapHistory& swap = mSwapHistory.next();
474         swap.damage = windowDirty;
475         swap.swapCompletedTime = systemTime(CLOCK_MONOTONIC);
476         swap.vsyncTime = mRenderThread.timeLord().latestVsync();
477         if (mNativeSurface.get()) {
478             int durationUs;
479             nsecs_t dequeueStart = mNativeSurface->getLastDequeueStartTime();
480             if (dequeueStart < mCurrentFrameInfo->get(FrameInfoIndex::SyncStart)) {
481                 // Ignoring dequeue duration as it happened prior to frame render start
482                 // and thus is not part of the frame.
483                 swap.dequeueDuration = 0;
484             } else {
485                 mNativeSurface->query(NATIVE_WINDOW_LAST_DEQUEUE_DURATION, &durationUs);
486                 swap.dequeueDuration = us2ns(durationUs);
487             }
488             mNativeSurface->query(NATIVE_WINDOW_LAST_QUEUE_DURATION, &durationUs);
489             swap.queueDuration = us2ns(durationUs);
490         } else {
491             swap.dequeueDuration = 0;
492             swap.queueDuration = 0;
493         }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaOpenGLPipeline.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in SkiaOper

```
59     return mEglManager.beginFrame(mEglSurface);
60 }
61
62 bool SkiaOpenGLPipeline::draw(const Frame& frame, const SkRect& screenDirty, const SkRect& dirty,
63                               const FrameBuilder::LightGeometry& lightGeometry,
64                               LayerUpdateQueue* layerUpdateQueue, const Rect& contentDrawBounds,
65                               bool opaque, bool wideColorGamut,
66                               const BakedOpRenderer::LightInfo& lightInfo,
67                               const std::vector<sp<RenderNode>>& renderNodes,
68                               FrameInfoVisualizer* profiler) {
69     mEglManager.damageFrame(frame, dirty);
70
71     // setup surface for fboID
72     GrGLFramebufferInfo fboInfo;
73     fboInfo.fFBOID = 0;
74     GrPixelConfig pixelConfig =
75             wideColorGamut ? kRGBA_half_GrPixelConfig : kRGBA_8888_GrPixelConfig;
76
77     GrBackendRenderTarget backendRT(frame.width(), frame.height(), 0, STENCIL_BUFFER_SIZE,
78                                     pixelConfig, fboInfo);
79
80     SkSurfaceProps props(0, kUnknown_SkPixelGeometry);
81
82     SkASSERT(mRenderThread.getGrContext() != nullptr);
83     sk_sp<SkSurface> surface(SkSurface::MakeFromBackendRenderTarget(
84             mRenderThread.getGrContext(), backendRT, kBottomLeft_GrSurfaceOrigin, nullptr, &props));
85
86     SkiaPipeline::updateLighting(lightGeometry, lightInfo);
87     renderFrame(*layerUpdateQueue, dirty, renderNodes, opaque, wideColorGamut, contentDrawBounds,
88                 surface);
89     layerUpdateQueue->clear();
90
91     // Draw visual debugging features
92     if (CC_UNLIKELY(Properties::showDirtyRegions ||
93                     ProfileType::None != Properties::getProfileType())) {
94         SkCanvas* profileCanvas = surface->getCanvas();
95         SkiaProfileRenderer profileRenderer(profileCanvas);
96         profiler->draw(profileRenderer);
97         profileCanvas->flush();
98     }
99
100    // Log memory statistics
101    if (CC_UNLIKELY(Properties::debugLevel != kDebugDisabled)) {
102        dumpResourceCacheUsage();
103    }
104
105    return true;
106 }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaOpenGLPipeline.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in SkiaOper

```
59      return mEglManager.beginFrame(mEglSurface);
60  }
61
62  bool SkiaOpenGLPipeline::draw(const Frame& frame, const SkRect& screenDirty, const SkRect& dirty,
63                                const FrameBuilder::LightGeometry& lightGeometry,
64                                LayerUpdateQueue* layerUpdateQueue, const Rect& contentDrawBounds,
65                                bool opaque, bool wideColorGamut,
66                                const BakedOpRenderer::LightInfo& lightInfo,
67                                const std::vector<sp<RenderNode>>& renderNodes,
68                                FrameInfoVisualizer* profiler) {
69      mEglManager.damageFrame(frame, dirty);
70
71      // setup surface for fbo0
72      GrGLFramebufferInfo fboInfo;
73      fboInfo.fFBOID = 0;
74      GrPixelConfig pixelConfig =
75              wideColorGamut ? kRGBA_half_GrPixelConfig : kRGBA_8888_GrPixelConfig;
76
77      GrBackendRenderTarget backendRT(frame.width(), frame.height(), 0, STENCIL_BUFFER_SIZE,
78                                      pixelConfig, fboInfo);
79
80      SkSurfaceProps props(0, kUnknown_SkPixelGeometry);
81
82      SkASSERT(mRenderThread.getGrContext() != nullptr);
83      sk_sp<SkSurface> surface(SkSurface::MakeFromBackendRenderTarget(
84              mRenderThread.getGrContext(), backendRT, kBottomLeft_GrSurfaceOrigin, nullptr, &props));
85
86      SkiaPipeline::updateLighting(lightGeometry, lightInfo);
87      renderFrame(*layerUpdateQueue, dirty, renderNodes, opaque, wideColorGamut, contentDrawBounds,
88                  surface);
89      layerUpdateQueue->clear();
90
91      // Draw visual debugging features
92      if (CC_UNLIKELY(Properties::showDirtyRegions ||
93                      ProfileType::None != Properties::getProfileType())) {
94          SkCanvas* profileCanvas = surface->getCanvas();
95          SkiaProfileRenderer profileRenderer(profileCanvas);
96          profiler->draw(profileRenderer);
97          profileCanvas->flush();
98      }
99
100     // Log memory statistics
101     if (CC_UNLIKELY(Properties::debugLevel != kDebugDisabled)) {
102         dumpResourceCacheUsage();
103     }
104
105     return true;
106 }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaPipeline.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in

```
322
323 void SkiaPipeline::renderFrame(const LayerUpdateQueue& layers, const SkRect& clip,
324                                 const std::vector<sp<RenderNode>>& nodes, bool opaque,
325                                 bool wideColorGamut, const Rect& contentDrawBounds,
326                                 sk_sp<SkSurface> surface) {
327     renderVectorDrawableCache();
328
329     // draw all layers up front
330     renderLayersImpl(layers, opaque, wideColorGamut);
331
332     // initialize the canvas for the current frame, that might be a recording canvas if SKP
333     // capture is enabled.
334     std::unique_ptr<SkPictureRecorder> recorder;
335     SkCanvas* canvas = tryCapture(surface.get());
336
337     renderFrameImpl(layers, clip, nodes, opaque, wideColorGamut, contentDrawBounds, canvas);
338
339     endCapture(surface.get());
340
341     if (CC_UNLIKELY(Properties::debugOverdraw)) {
342         renderOverdraw(layers, clip, nodes, contentDrawBounds, surface);
343     }
344
345     ATRACE_NAME("flush commands");
346     surface->getCanvas()->flush();
347 }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaPipeline.cpp

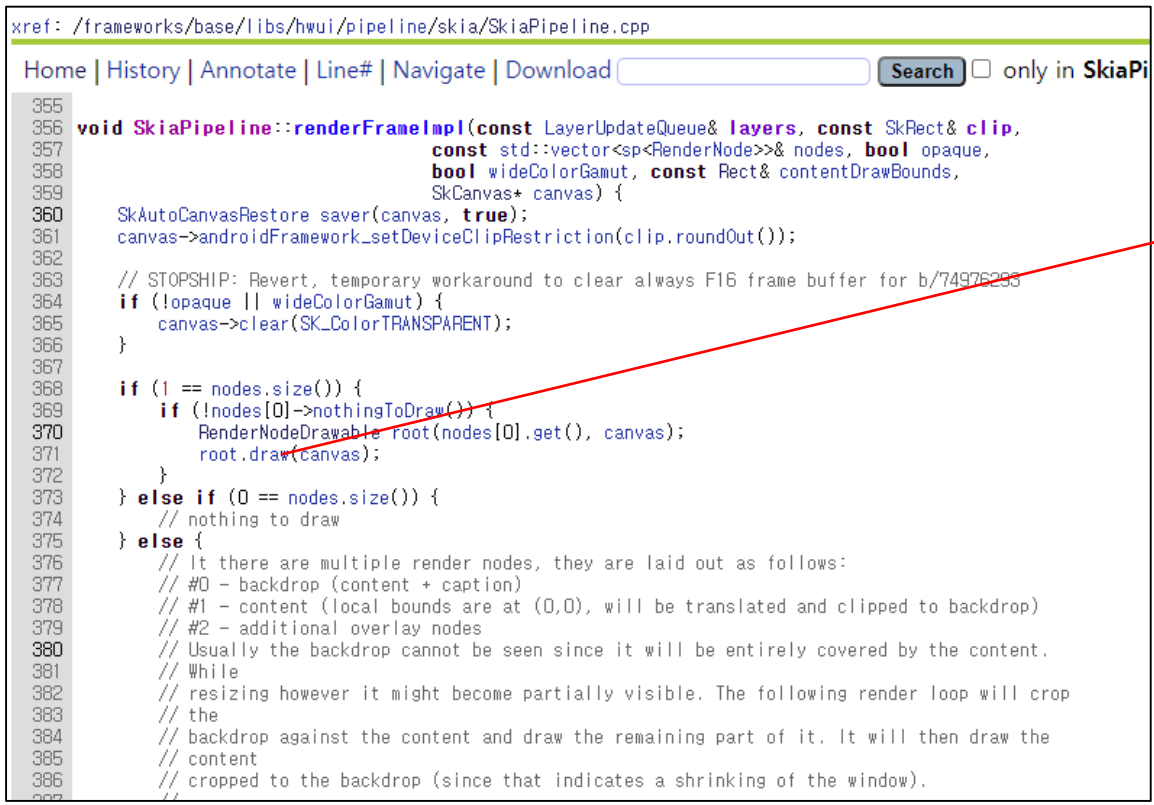
Home | History | Annotate | Line# | Navigate | Download Search only in

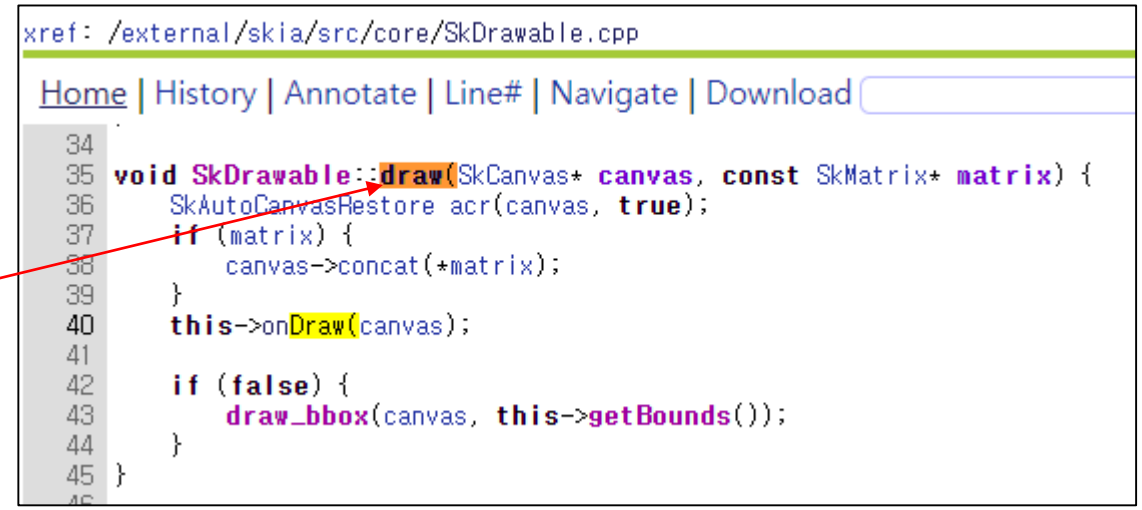
```
322
323 void SkiaPipeline::renderFrame(const LayerUpdateQueue& layers, const SkRect& clip,
324                                 const std::vector<sp<RenderNode>>& nodes, bool opaque,
325                                 bool wideColorGamut, const Rect& contentDrawBounds,
326                                 sk_sp<SkSurface> surface) {
327     renderVectorDrawableCache();
328
329     // draw all layers up front
330     renderLayersImpl(layers, opaque, wideColorGamut);
331
332     // initialize the canvas for the current frame, that might be a recording canvas if SKP
333     // capture is enabled.
334     std::unique_ptr<SkPictureRecorder> recorder;
335     SkCanvas* canvas = tryCapture(surface.get());
336
337     renderFrameImpl(layers, clip, nodes, opaque, wideColorGamut, contentDrawBounds, canvas);
338
339     endCapture(surface.get());
340
341     if (CC_UNLIKELY(Properties::debugOverdraw)) {
342         renderOverdraw(layers, clip, nodes, contentDrawBounds, surface);
343     }
344
345     ATRACE_NAME("flush commands");
346     surface->getCanvas()->flush();
347 }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaPipeline.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in SkiaPi

```
355
356 void SkiaPipeline::renderFrameImpl(const LayerUpdateQueue& layers, const SkRect& clip,
357                                    const std::vector<sp<RenderNode>>& nodes, bool opaque,
358                                    bool wideColorGamut, const Rect& contentDrawBounds,
359                                    SkCanvas* canvas) {
360     SkAutoCanvasRestore saver(canvas, true);
361     canvas->androidFramework_setDeviceClipRestriction(clip.roundOut());
362
363     // STOPSHIP: Revert, temporary workaround to clear always F16 frame buffer for b/74976293
364     if (!opaque || wideColorGamut) {
365         canvas->clear(SK_ColorTRANSPARENT);
366     }
367
368     if (1 == nodes.size()) {
369         if (!nodes[0]->nothingToDraw()) {
370             RenderNodeDrawable root(nodes[0].get(), canvas);
371             root.draw(canvas);
372         }
373     } else if (0 == nodes.size()) {
374         // nothing to draw
375     } else {
376         // It there are multiple render nodes, they are laid out as follows:
377         // #0 - backdrop (content + caption)
378         // #1 - content (local bounds are at (0,0), will be translated and clipped to backdrop)
379         // #2 - additional overlay nodes
380         // Usually the backdrop cannot be seen since it will be entirely covered by the content.
381         // While
382         // resizing however it might become partially visible. The following render loop will crop
383         // the
384         // backdrop against the content and draw the remaining part of it. It will then draw the
385         // content
386         // cropped to the backdrop (since that indicates a shrinking of the window).
```


xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaPipeline.cpp
Home | History | Annotate | Line# | Navigate | Download
Search
only in SkiaPi
355
356 void SkiaPipeline::renderFrameImpl(const LayerUpdateQueue& layers, const SkRect& clip,
357 const std::vector<sp<RenderNode>>& nodes, bool opaque,
358 bool wideColorGamut, const Rect& contentDrawBounds,
359 SkCanvas* canvas) {
360 SkAutoCanvasRestore saver(canvas, true);
361 canvas->androidFramework_setDeviceClipRestriction(clip.roundOut());
362
363 // STOPSHIP: Revert, temporary workaround to clear always F16 frame buffer for b/74976293
364 if (!opaque || wideColorGamut) {
365 canvas->clear(SK_ColorTRANSPARENT);
366 }
367
368 if (1 == nodes.size()) {
369 if (!nodes[0]->nothingToDraw()) {
370 RenderNodeDrawable root(nodes[0].get(), canvas);
371 root.draw(canvas);
372 }
373 } else if (0 == nodes.size()) {
374 // nothing to draw
375 } else {
376 // It there are multiple render nodes, they are laid out as follows:
377 // #0 - backdrop (content + caption)
378 // #1 - content (local bounds are at (0,0), will be translated and clipped to backdrop)
379 // #2 - additional overlay nodes
380 // Usually the backdrop cannot be seen since it will be entirely covered by the content.
381 // While
382 // resizing however it might become partially visible. The following render loop will crop
383 // the
384 // backdrop against the content and draw the remaining part of it. It will then draw the
385 // content
386 // cropped to the backdrop (since that indicates a shrinking of the window).


xref: /external/skia/src/core/SkDrawable.cpp
Home | History | Annotate | Line# | Navigate | Download
34
35 void SkDrawable::draw(SkCanvas* canvas, const SkMatrix* matrix) {
36 SkAutoCanvasRestore acr(canvas, true);
37 if (matrix) {
38 canvas->concat(*matrix);
39 }
40 this->onDraw(canvas);
41
42 if (false) {
43 draw_bbox(canvas, this->getBounds());
44 }
45 }

xref: /external/skia/src/core/SkDrawable.cpp

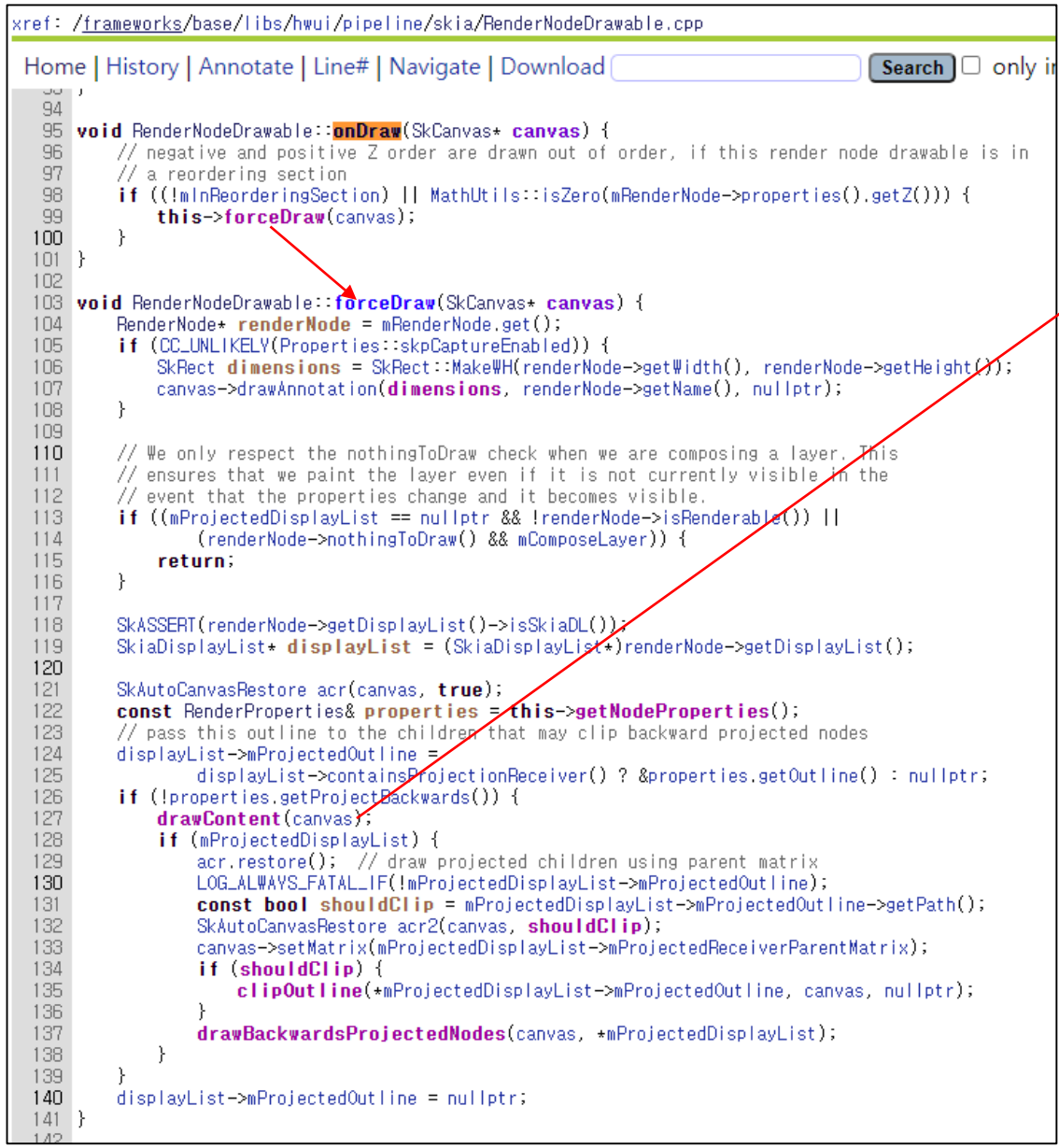
Home | History | Annotate | Line# | Navigate | Download

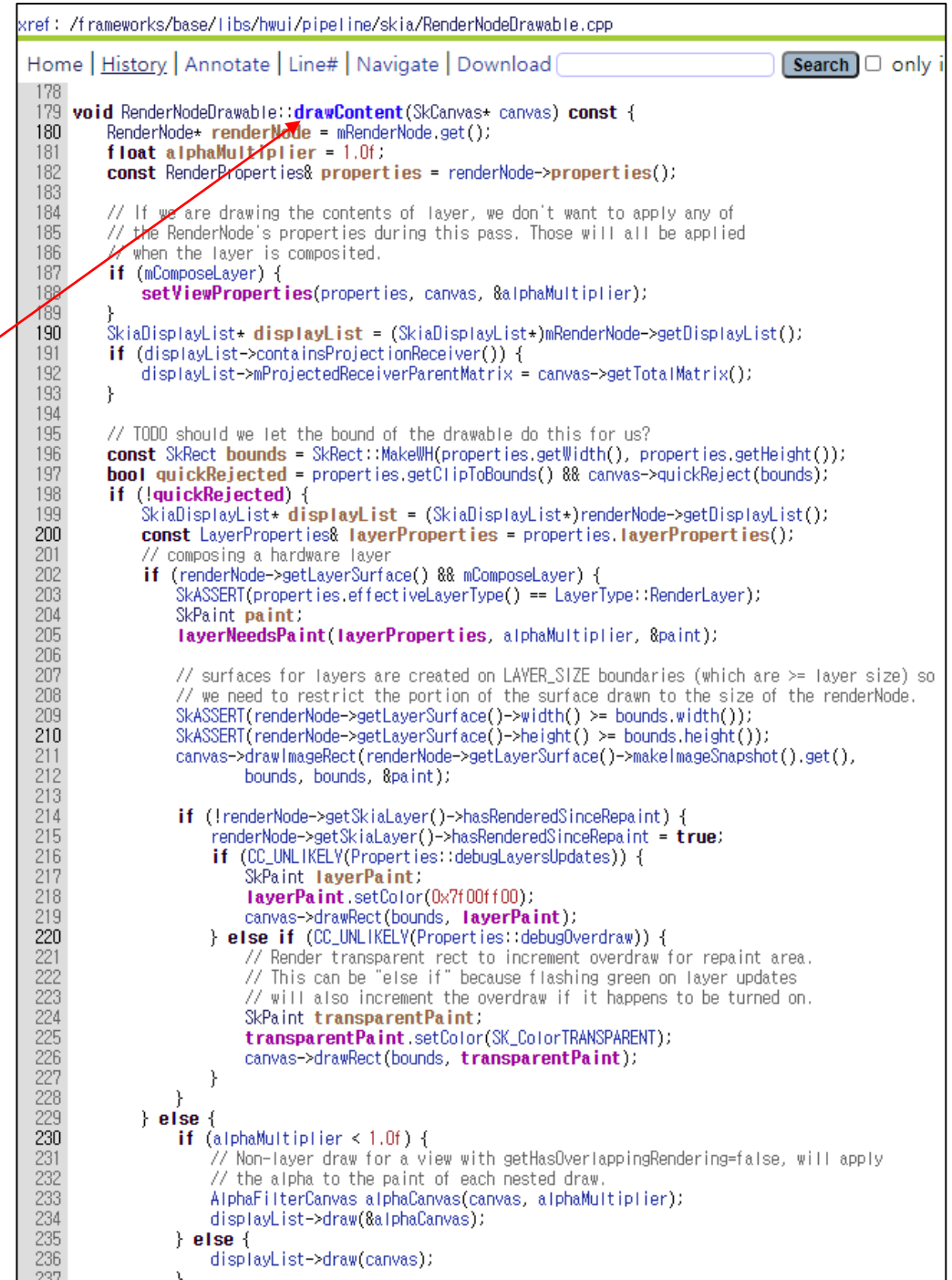
```
34
35 void SkDrawable::draw(SkCanvas* canvas, const SkMatrix* matrix) {
36     SkAutoCanvasRestore acr(canvas, true);
37     if (matrix) {
38         canvas->concat(*matrix);
39     }
40     this->onDraw(canvas);
41
42     if (false) {
43         draw_bbox(canvas, this->getBounds());
44     }
45 }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/RenderNodeDrawable.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in

```
94
95 void RenderNodeDrawable::onDraw(SkCanvas* canvas) {
96     // negative and positive Z order are drawn out of order, if this render node drawable is in
97     // a reordering section
98     if ((!mInReorderingSection) || MathUtils::isZero(mRenderNode->properties().getZ())) {
99         this->forceDraw(canvas);
100    }
101 }
102
103 void RenderNodeDrawable::forceDraw(SkCanvas* canvas) {
104     RenderNode* renderNode = mRenderNode.get();
105     if (CC_UNLIKELY(Properties::skpCaptureEnabled)) {
106         SkRect dimensions = SkRect::MakeWH(renderNode->getWidth(), renderNode->getHeight());
107         canvas->drawAnnotation(dimensions, renderNode->getName(), nullptr);
108     }
109
110     // We only respect the nothingToDraw check when we are composing a layer. This
111     // ensures that we paint the layer even if it is not currently visible in the
112     // event that the properties change and it becomes visible.
113     if ((mProjectedDisplayList == nullptr && !renderNode->isRenderable()) ||
114             (renderNode->nothingToDraw() && mComposeLayer)) {
115         return;
116     }
117
118     SkASSERT(renderNode->getDisplayList()->isSkiaDL());
119     SkiaDisplayList* displayList = (SkiaDisplayList*)renderNode->getDisplayList();
120
121     SkAutoCanvasRestore acr(canvas, true);
122     const RenderProperties& properties = this->getNodeProperties();
123     // pass this outline to the children that may clip backward projected nodes
124     displayList->mProjectedOutline =
125             displayList->containsProjectionReceiver() ? &properties.getOutline() : nullptr;
126     if (!properties.getProjectBackwards()) {
127         drawContent(canvas);
128         if (mProjectedDisplayList) {
129             acr.restore();  // draw projected children using parent matrix
130             LOG_ALWAYS_FATAL_IF(!mProjectedDisplayList->mProjectedOutline);
131             const bool shouldClip = mProjectedDisplayList->mProjectedOutline->getPath();
132             SkAutoCanvasRestore acr2(canvas, shouldClip);
133             canvas->setMatrix(mProjectedDisplayList->mProjectedReceiverParentMatrix);
134             if (shouldClip) {
135                 clipOutline(*mProjectedDisplayList->mProjectedOutline, canvas, nullptr);
136             }
137             drawBackwardsProjectedNodes(canvas, *mProjectedDisplayList);
138         }
139     }
140     displayList->mProjectedOutline = nullptr;
141 }
```


xref: /frameworks/base/libs/hwui/pipeline/skia/RenderNodeDrawable.cpp
Home | History | Annotate | Line# | Navigate | Download
Search
only i
94
95 void RenderNodeDrawable::onDraw(SkCanvas* canvas) {
96 // negative and positive Z order are drawn out of order, if this render node drawable is in
97 // a reordering section
98 if ((!mInReorderingSection) || MathUtils::isZero(mRenderNode->properties().getZ())) {
99 this->forceDraw(canvas);
100 }
101 }
102
103 void RenderNodeDrawable::forceDraw(SkCanvas* canvas) {
104 RenderNode* renderNode = mRenderNode.get();
105 if (CC_UNLIKELY(Properties::skpCaptureEnabled)) {
106 SkRect dimensions = SkRect::MakeWH(renderNode->getWidth(), renderNode->getHeight());
107 canvas->drawAnnotation(dimensions, renderNode->getName(), nullptr);
108 }
109
110 // We only respect the nothingToDraw check when we are composing a layer. This
111 // ensures that we paint the layer even if it is not currently visible in the
112 // event that the properties change and it becomes visible.
113 if ((mProjectedDisplayList == nullptr && !renderNode->isRenderable()) ||
114 (renderNode->nothingToDraw() && mComposeLayer)) {
115 return;
116 }
117
118 SkASSERT(renderNode->getDisplayList()->isSkiaDL());
119 SkiaDisplayList* displayList = (SkiaDisplayList*)renderNode->getDisplayList();
120
121 SkAutoCanvasRestore acr(canvas, true);
122 const RenderProperties& properties = this->getNodeProperties();
123 // pass this outline to the children that may clip backward projected nodes
124 displayList->mProjectedOutline =
125 displayList->containsProjectionReceiver() ? &properties.getOutline() : nullptr;
126 if (!properties.getProjectBackwards()) {
127 drawContent(canvas);
128 if (mProjectedDisplayList) {
129 acr.restore(); // draw projected children using parent matrix
130 LOG_ALWAYS_FATAL_IF(!mProjectedDisplayList->mProjectedOutline);
131 const bool shouldClip = mProjectedDisplayList->mProjectedOutline->getPath();
132 SkAutoCanvasRestore acr2(canvas, shouldClip);
133 canvas->setMatrix(mProjectedDisplayList->mProjectedReceiverParentMatrix);
134 if (shouldClip) {
135 clipOutline(*mProjectedDisplayList->mProjectedOutline, canvas, nullptr);
136 }
137 drawBackwardsProjectedNodes(canvas, *mProjectedDisplayList);
138 }
139 }
140 displayList->mProjectedOutline = nullptr;
141 }


xref: /frameworks/base/libs/hwui/pipeline/skia/RenderNodeDrawable.cpp
Home | History | Annotate | Line# | Navigate | Download
Search
only i
178
179 void RenderNodeDrawable::drawContent(SkCanvas* canvas) const {
180 RenderNode* renderNode = mRenderNode.get();
181 float alphaMultiplier = 1.0f;
182 const RenderProperties& properties = renderNode->properties();
183
184 // If we are drawing the contents of layer, we don't want to apply any of
185 // the RenderNode's properties during this pass. Those will all be applied
186 // when the layer is composited.
187 if (mComposeLayer) {
188 setViewProperties(properties, canvas, &alphaMultiplier);
189 }
190 SkiaDisplayList* displayList = (SkiaDisplayList*)mRenderNode->getDisplayList();
191 if (displayList->containsProjectionReceiver()) {
192 displayList->mProjectedReceiverParentMatrix = canvas->getTotalMatrix();
193 }
194
195 // TODO should we let the bound of the drawable do this for us?
196 const SkRect bounds = SkRect::MakeWH(properties.getWidth(), properties.getHeight());
197 bool quickRejected = properties.getClipToBounds() && canvas->quickReject(bounds);
198 if (!quickRejected) {
199 SkiaDisplayList* displayList = (SkiaDisplayList*)renderNode->getDisplayList();
200 const LayerProperties& layerProperties = properties.layerProperties();
201 // composing a hardware layer
202 if (renderNode->getLayerSurface() && mComposeLayer) {
203 SkASSERT(properties.effectiveLayerType() == LayerType::RenderLayer);
204 SkPaint paint;
205 layerNeedsPaint(layerProperties, alphaMultiplier, &paint);
206
207 // surfaces for layers are created on LAYER_SIZE boundaries (which are >= layer size) so
208 // we need to restrict the portion of the surface drawn to the size of the renderNode.
209 SkASSERT(renderNode->getLayerSurface()->width() >= bounds.width());
210 SkASSERT(renderNode->getLayerSurface()->height() >= bounds.height());
211 canvas->drawImageRect(renderNode->getLayerSurface()->makeImageSnapshot().get(),
212 bounds, bounds, &paint);
213
214 if (!renderNode->getSkiaLayer()->hasRenderedSinceRepaint) {
215 renderNode->getSkiaLayer()->hasRenderedSinceRepaint = true;
216 if (CC_UNLIKELY(Properties::debugLayersUpdates)) {
217 SkPaint layerPaint;
218 layerPaint.setColor(0x7f00ff00);
219 canvas->drawRect(bounds, layerPaint);
220 } else if (CC_UNLIKELY(Properties::debugOverdraw)) {
221 // Render transparent rect to increment overdraw for repaint area.
222 // This can be "else if" because flashing green on layer updates
223 // will also increment the overdraw if it happens to be turned on.
224 SkPaint transparentPaint;
225 transparentPaint.setColor(SK_ColorTRANSPARENT);
226 canvas->drawRect(bounds, transparentPaint);
227 }
228 }
229 } else {
230 if (alphaMultiplier < 1.0f) {
231 // Non-layer draw for a view with getHasOverlappingRendering=false, will apply
232 // the alpha to the paint of each nested draw.
233 AlphaFilterCanvas alphaCanvas(canvas, alphaMultiplier);
234 displayList->draw(&alphaCanvas);
235 } else {
236 displayList->draw(canvas);
237 }

```
178
179 void RenderNodeDrawable::drawContent(SkCanvas* canvas) const {
180     RenderNode* renderNode = mRenderNode.get();
181     float alphaMultiplier = 1.0f;
182     const RenderProperties& properties = renderNode->properties();
183
184     // If we are drawing the contents of layer, we don't want to apply any of
185     // the RenderNode's properties during this pass. Those will all be applied
186     // when the layer is composited.
187     if (mComposeLayer) {
188         setViewProperties(properties, canvas, &alphaMultiplier);
189     }
190     SkiaDisplayList* displayList = (SkiaDisplayList*)mRenderNode->getDisplayList();
191     if (displayList->containsProjectionReceiver()) {
192         displayList->mProjectedReceiverParentMatrix = canvas->getTotalMatrix();
193     }
194
195     // TODO should we let the bound of the drawable do this for us?
196     const SkRect bounds = SkRect::MakeWH(properties.getWidth(), properties.getHeight());
197     bool quickRejected = properties.getClipToBounds() && canvas->quickReject(bounds);
198     if (!quickRejected) {
199         SkiaDisplayList* displayList = (SkiaDisplayList*)renderNode->getDisplayList();
200         const LayerProperties& layerProperties = properties.layerProperties();
201         // composing a hardware layer
202         if (renderNode->getLayerSurface() && mComposeLayer) {
203             SkASSERT(properties.effectiveLayerType() == LayerType::RenderLayer);
204             SkPaint paint;
205             layerNeedsPaint(layerProperties, alphaMultiplier, &paint);
206
207             // surfaces for layers are created on LAYER_SIZE boundaries (which are >= layer size) so
208             // we need to restrict the portion of the surface drawn to the size of the renderNode.
209             SkASSERT(renderNode->getLayerSurface()->width() >= bounds.width());
210             SkASSERT(renderNode->getLayerSurface()->height() >= bounds.height());
211             canvas->drawImageRect(renderNode->getLayerSurface()->makeImageSnapshot().get(),
212                     bounds, bounds, &paint);
213
214             if (!renderNode->getSkiaLayer()->hasRenderedSinceRepaint) {
215                 renderNode->getSkiaLayer()->hasRenderedSinceRepaint = true;
216                 if (CC_UNLIKELY(Properties::debugLayersUpdates)) {
217                     SkPaint layerPaint;
218                     layerPaint.setColor(0x7f00ff00);
219                     canvas->drawRect(bounds, layerPaint);
220                 } else if (CC_UNLIKELY(Properties::debugOverdraw)) {
221                     // Render transparent rect to increment overdraw for repaint area.
222                     // This can be "else if" because flashing green on layer updates
223                     // will also increment the overdraw if it happens to be turned on.
224                     SkPaint transparentPaint;
225                     transparentPaint.setColor(SK_ColorTRANSPARENT);
226                     canvas->drawRect(bounds, transparentPaint);
227                 }
228             }
229         } else {
230             if (alphaMultiplier < 1.0f) {
231                 // Non-layer draw for a view with getHasOverlappingRendering=false, will apply
232                 // the alpha to the paint of each nested draw.
233                 AlphaFilterCanvas alphaCanvas(canvas, alphaMultiplier);
234                 displayList->draw(&alphaCanvas);
235             } else {
236                 displayList->draw(canvas);
237             }
238         }
239     }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaDisplayList.h

Home | History | Annotate | Line# | Navigate | Download | Search | only in Ski

```
118             TreeObserver& observer, TreeInfo& info, bool functorsNeedLayer,
119             std::function<void(RenderNode*, TreeObserver&, TreeInfo&, bool)> childFn) override;
120
121     /**
122      * Calls the provided function once for each child of this DisplayList
123      */
124     void updateChildren(std::function<void(RenderNode*)> updateFn) override;
125
126     /**
127      * Returns true if there is a child render node that is a projection receiver.
128      */
129     inline bool containsProjectionReceiver() const { return mProjectionReceiver; }
130
131     void attachRecorder(SkLiteRecorder* recorder, const SkIRect& bounds) {
132         recorder->reset(&mDisplayList, bounds);
133     }
134
135     void draw(SkCanvas* canvas) { mDisplayList.draw(canvas); }
```

xref: /frameworks/base/libs/hwui/pipeline/skia/SkiaDisplayList.h

Home | History | Annotate | Line# | Navigate | Download Search only in Ski

```
118         TreeObserver& observer, TreeInfo& info, bool functorsNeedLayer,
119         std::function<void(RenderNode*, TreeObserver&, TreeInfo&, bool)> childFn) override;
120
121     /**
122      * Calls the provided function once for each child of this DisplayList
123      */
124     void updateChildren(std::function<void(RenderNode*)> updateFn) override;
125
126     /**
127      * Returns true if there is a child render node that is a projection receiver.
128      */
129     inline bool containsProjectionReceiver() const { return mProjectionReceiver; }
130
131     void attachRecorder(SkLiteRecorder* recorder, const SkIRect& bounds) {
132         recorder->reset(&mDisplayList, bounds);
133     }
134
135     void draw(SkCanvas* canvas) { mDisplayList.draw(canvas); }
```

xref: /external/skia/src/core/SkLiteDL.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in S

```
682 void SkLiteDL::drawAtlas(const SkImage* atlas, const SkRSXform xforms[], const SkRect texs[],
683                          const SkColor colors[], int count, SkBlendMode xfermode,
684                          const SkRect* cull, const SkPaint* paint) {
685     size_t bytes = count*(sizeof(SkRSXform) + sizeof(SkRect));
686     if (colors) {
687         bytes += count*sizeof(SkColor);
688     }
689     void* pod = this->push<DrawAtlas>(bytes,
690                                       atlas, count, xfermode, cull, paint, colors != nullptr);
691     copy_v(pod, xforms, count,
692                 texs, count,
693                colors, colors ? count : 0);
694 }
695 void SkLiteDL::drawShadowRec(const SkPath& path, const SkDrawShadowRec& rec) {
696     this->push<DrawShadowRec>(0, path, rec);
697 }
698
699 typedef void(*draw_fn)(const void*,  SkCanvas*, const SkMatrix&);
700 typedef void(*void_fn)(const void*);
701
702 // All ops implement draw().
703 #define M(T) [](const void* op, SkCanvas* c, const SkMatrix& original) { \
704     ((const T*)op)->draw(c, original);                                    \
705 },
706 static const draw_fn draw_fns[] = { TYPES(M) };
707 #undef M
708
709 // Older libstdc++ has pre-standard std::has_trivial_destructor.
710 #if defined(__GLIBCXX__) && (__GLIBCXX__ < 20130000)
711     template <typename T> using can_skip_destructor = std::has_trivial_destructor<T>;
712 #else
713     template <typename T> using can_skip_destructor = std::is_trivially_destructible<T>;
714 #endif
715
716 // Most state ops (matrix, clip, save, restore) have a trivial destructor.
717 #define M(T) !can_skip_destructor<T>::value ? [](const void* op) { ((const T*)op)->~T(); } \
718                                             : (void_fn)nullptr,
719 static const void_fn dtor_fns[] = { TYPES(M) };
720 #undef M
721
722 void SkLiteDL::draw(SkCanvas* canvas) const {
723     SkAutoCanvasRestore acr(canvas, false);
724     this->map(draw_fns, canvas, canvas->getTotalMatrix());
725 }
```