Drawing Path

Pie(9.0)

recordOp->openGL

자세한 코드

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

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

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

Home | History | Annotate | Line# | Navigate | Down

```
783
784 void SkCanvas::flush() {
785     this->onFlush();
786 }
787
788 void SkCanvas::onFlush() {
789     SkBaseDevice* device = this->getDevice();
790     if (device) {
791         device->flush();
792     }
793 }
794
```

자세한 코드

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

Home | History | Annotate | Line# | Navigate | Down

```
783
784 void SkCanvas::flush() {
785     this->onFlush();
786 }
787
788 void SkCanvas::onFlush() {
789     SkBaseDevice* device = this->getDevice();
790     if (device) {
791         device->flush();
792     }
793 }
794
```

xref: /external/skia/src/gpu/SkGpuDevice.cpp

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

```
1662 ////////////////////////////////////////////////////////////////////////////////
1663
1664 void SkGpuDevice::flush() {
1665     this->flushAndSignalSemaphores(0, nullptr);
1666 }
1667
1668 GrSemaphoresSubmitted SkGpuDevice::flushAndSignalSemaphores(int numSemaphores,
1669                                                             GrBackendSemaphore signalSemaphores[]) {
1670     ASSERT_SINGLE_OWNER
1671
1672     return fRenderTargetContext->prepareForExternalIO(numSemaphores, signalSemaphores);
1673 }
```

자세한 코드

xref: /external/skia/src/gpu/GrRenderTargetContext.cpp

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

```
1382 GrSemaphoresSubmitted GrRenderTargetContext::prepareForExternalIO(
1383         int numSemaphores, GrBackendSemaphore backendSemaphores[]) {
1384     ASSERT_SINGLE_OWNER
1385     if (this->drawingManager()->wasAbandoned()) { return GrSemaphoresSubmitted::kNo; }
1386     SkDEBUGCODE(this->validate();)
1387     GR_CREATE_TRACE_MARKER_CONTEXT("GrRenderTargetContext", "prepareForExternalIO", fContext);
1388
1389     return this->drawingManager()->prepareSurfaceForExternalIO(fRenderTargetProxy.get(),
1390                                                                numSemaphores,
1391                                                                backendSemaphores);
1392 }
```

xref: /external/skia/src/gpu/SkGpuDevice.cpp

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

```
1662 ////////////////////////////////////////////////////////////////////////////////
1663
1664 void SkGpuDevice::flush() {
1665     this->flushAndSignalSemaphores(0, nullptr);
1666 }
1667
1668 GrSemaphoresSubmitted SkGpuDevice::flushAndSignalSemaphores(int numSemaphores,
1669                                                             GrBackendSemaphore signalSemaphores[]) {
1670     ASSERT_SINGLE_OWNER
1671
1672     return fRenderTargetContext->prepareForExternalIO(numSemaphores, signalSemaphores);
1673 }
```

자세한 코드

xref: /external/skia/src/gpu/GrDrawingManager.cpp

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

```
326 GrSemaphoresSubmitted GrDrawingManager::prepareSurfaceForExternalIO(
327         GrSurfaceProxy* proxy, int numSemaphores, GrBackendSemaphore backendSemaphores[]) {
328     if (this->wasAbandoned()) {
329         return GrSemaphoresSubmitted::kNo;
330     }
331     SkASSERT(proxy);
332
333     GrSemaphoresSubmitted result = GrSemaphoresSubmitted::kNo;
334     if (proxy->priv().hasPendingIO() || numSemaphores) {
335         result = this->flush(proxy, numSemaphores, backendSemaphores);
336     }
337
338     if (!proxy->instantiate(fContext->contextPriv().resourceProvider())) {
339         return result;
340     }
341
342     GrGpu* gpu = fContext->contextPriv().getGpu();
343     GrSurface* surface = proxy->priv().peekSurface();
344
345     if (gpu && surface->asRenderTarget()) {
            gpu->resolveRenderTarget(surface->asRenderTarget());

        urn result;
```

xref: /external/skia/src/gpu/GrRenderTargetContext.cpp

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

```
1382 GrSemaphoresSubmitted GrRenderTargetContext::prepareForExternalIO(
1383         int numSemaphores, GrBackendSemaphore backendSemaphores[]) {
1384     ASSERT_SINGLE_OWNER
1385     if (this->drawingManager()->wasAbandoned()) { return GrSemaphoresSubmitted::kNo; }
1386     SkDEBUGCODE(this->validate();)
1387     GR_CREATE_TRACE_MARKER_CONTEXT("GrRenderTargetContext", "prepareForExternalIO", fContext);
1388
1389     return this->drawingManager()->prepareSurfaceForExternalIO(fRenderTargetProxy.get(),
1390                                                                numSemaphores,
1391                                                                backendSemaphores);
1392 }
```

자세한 코드

xref: /external/skia/src/gpu/GrDrawingManager.cpp

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

```
326 GrSemaphoresSubmitted GrDrawingManager::prepareSurfaceForExternalIO(
327         GrSurfaceProxy* proxy, int numSemaphores, GrBackendSemaphore backendSemaphores[]) {
328     if (this->wasAbandoned()) {
329         return GrSemaphoresSubmitted::kNo;
330     }
331     SkASSERT(proxy);
332
333     GrSemaphoresSubmitted result = GrSemaphoresSubmitted::kNo;
334     if (proxy->priv().hasPendingIO() || numSemaphores) {
335         result = this->flush(proxy, numSemaphores, backendSemaphores);
336     }
337
338     if (!proxy->instantiate(fContext->contextPriv().resourceProvider())) {
339         return result;
340     }
341
342     GrGpu* gpu = fContext->contextPriv().getGpu();
343     GrSurface* surface = proxy->priv().peekSurface();
344
345     if (gpu && surface->asRenderTarget()) {
346         gpu->resolveRenderTarget(surface->asRenderTarget());
347     }
348     return result;
349 }
```

xref: /external/skia/src/gpu/GrDrawingManager.h

me | History | Annotate | Line# | Navigate | Download

```
    GrSemaphoresSubmitted flush(GrSurfaceProxy* proxy,
                                int numSemaphores = 0,
                                GrBackendSemaphore backendSemaphores[] = nullptr) {
        return this->internalFlush(proxy, GrResourceCache::FlushType::kExternal,
                                   numSemaphores, backendSemaphores);
    }
```

자세한 코드

*이 다음부터 로그로 확인 못해봤습니다.

xref: /external/skia/src/gpu/GrDrawingManager.h

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

```
GrSemaphoresSubmitted flush(GrSurfaceProxy* proxy,
                            int numSemaphores = 0,
                            GrBackendSemaphore backendSemaphores[] = nullptr) {
    return this->internalFlush(proxy, GrResourceCache::FlushType::kExternal,
                               numSemaphores, backendSemaphores);
}
```

xref: /external/skia/src/gpu/GrDrawingManager.cpp

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

```
// MDB TODO: make use of the 'proxy' parameter.
GrSemaphoresSubmitted GrDrawingManager::internalFlush(GrSurfaceProxy*,
                                                      GrResourceCache::FlushType type,
                                                      int numSemaphores,
                                                      GrBackendSemaphore backendSemaphores[]) {
    GR_CREATE_TRACE_MARKER_CONTEXT("GrDrawingManager", "internalFlush", fContext);

    if (fFlushing || this->wasAbandoned()) {
        return GrSemaphoresSubmitted::kNo;
    }
    fFlushing = true;

    for (int i = 0; i < fOpLists.count(); ++i) {
        // Semi-usually the GrOpLists are already closed at this point, but sometimes Ganesh
        // needs to flush mid-draw. In that case, the SkGpuDevice's GrOpLists won't be closed
        // but need to be flushed anyway. Closing such GrOpLists here will mean new
        // GrOpLists will be created to replace them if the SkGpuDevice(s) write to them again.
        fOpLists[i]->makeClosed(*fContext->caps());
    }

#ifdef SK_DEBUG
    // This block checks for any unnecessary splits in the opLists. If two sequential opLists
    // share the same backing GrSurfaceProxy it means the opList was artificially split.
    if (fOpLists.count()) {
        GrRenderTargetOpList* prevOpList = fOpLists[0]->asRenderTargetOpList();
        for (int i = 1; i < fOpLists.count(); ++i) {
            GrRenderTargetOpList* curOpList = fOpLists[i]->asRenderTargetOpList();

            if (prevOpList && curOpList) {
                SkASSERT(prevOpList->fTarget.get() != curOpList->fTarget.get());
            }

            prevOpList = curOpList;
        }
    }
#endif

    if (fSortRenderTargets) {
        SkDEBUGCODE(bool result =) SkTTopoSort<GrOpList, GrOpList::TopoSortTraits>(&fOpLists);
        SkASSERT(result);
    }

    GrGpu* gpu = fContext->contextPriv().getGpu();

    GrOpFlushState flushState(gpu, fContext->contextPriv().resourceProvider(),
                              &fTokenTracker);
```

```
117 GrSemaphoresSubmitted GrDrawingManager::internalFlush(GrSurfaceProxy*,
118                                                       GrResourceCache::FlushType type,
119                                                       int numSemaphores,
120                                                       GrBackendSemaphore backendSemaphores[]) {
121     GR_CREATE_TRACE_MARKER_CONTEXT("GrDrawingManager", "internalFlush", fContext);
122
123     if (fFlushing || this->wasAbandoned()) {
124         return GrSemaphoresSubmitted::kNo;
125     }
126     fFlushing = true;
127
128     for (int i = 0; i < fOpLists.count(); ++i) {
129         // Semi-usually the GrOpLists are already closed at this point, but sometimes Ganesh
130         // needs to flush mid-draw. In that case, the SkGpuDevice's GrOpLists won't be closed
131         // but need to be flushed anyway. Closing such GrOpLists here will mean new
132         // GrOpLists will be created to replace them if the SkGpuDevice(s) write to them again.
133         fOpLists[i]->makeClosed(*fContext->caps());
134     }
135
136 #ifdef SK_DEBUG
137     // This block checks for any unnecessary splits in the opLists. If two sequential opLists
138     // share the same backing GrSurfaceProxy it means the opList was artificially split.
139     if (fOpLists.count()) {
140         GrRenderTargetOpList* prevOpList = fOpLists[0]->asRenderTargetOpList();
141         for (int i = 1; i < fOpLists.count(); ++i) {
142             GrRenderTargetOpList* curOpList = fOpLists[i]->asRenderTargetOpList();
143
144             if (prevOpList && curOpList) {
145                 SkASSERT(prevOpList->fTarget.get() != curOpList->fTarget.get());
146             }
147
148             prevOpList = curOpList;
149         }
150     }
```

```
        GrResourceAllocator alloc(fContext->contextPriv().resourceProvider());
        for (int i = 0; i < fOpLists.count(); ++i) {
            fOpLists[i]->gatherProxyIntervals(&alloc);
            alloc.markEndOfOpList(i);
        }

        GrResourceAllocator::AssignError error = GrResourceAllocator::AssignError::kNoError;
        while (alloc.assign(&startIndex, &stopIndex, &error)) {
            if (GrResourceAllocator::AssignError::kFailedProxyInstantiation == error) {
                for (int i = startIndex; i < stopIndex; ++i) {
                    fOpLists[i]->purgeOpsWithUninstantiatedProxies();
                }
            }

            if (this->executeOpLists(startIndex, stopIndex, &flushState)) {
                flushed = true;
            }
        }
    }

    fOpLists.reset();

    GrSemaphoresSubmitted result = gpu->finishFlush(numSemaphores, backendSemaphores);
```

xref: /external/skia/src/gpu/GrDrawingManager.cpp

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

```
245     return result;
246 }
247
248 bool GrDrawingManager::executeOpLists(int startIndex, int stopIndex, GrOpFlushState* flushState) {
249     SkASSERT(startIndex <= stopIndex && stopIndex <= fOpLists.count());
250
251     GrResourceProvider* resourceProvider = fContext->contextPriv().resourceProvider();
252     bool anyOpListsExecuted = false;
253
254     for (int i = startIndex; i < stopIndex; ++i) {
255         if (!fOpLists[i]) {
256             continue;
257         }
258
259         if (resourceProvider->explicitlyAllocateGPUResources()) {
260             if (!fOpLists[i]->isInstantiated()) {
261                 // If the backing surface wasn't allocated drop the draw of the entire opList.
262                 fOpLists[i] = nullptr;
263                 continue;
264             }
265         } else {
266             if (!fOpLists[i]->instantiate(resourceProvider)) {
267                 SkDebugf("OpList failed to instantiate.\n");
268                 fOpLists[i] = nullptr;
269                 continue;
270             }
271         }
272
273         // TODO: handle this instantiation via lazy surface proxies?
274         // Instantiate all deferred proxies (being built on worker threads) so we can upload them
275         fOpLists[i]->instantiateDeferredProxies(fContext->contextPriv().resourceProvider());
276         fOpLists[i]->prepare(flushState);
277     }
278
279     // Upload all data to the GPU
280     flushState->preExecuteDraws();
281
282     // Execute the onFlush op lists first, if any.
283     for (sk_sp<GrOpList>& onFlushOpList : fOnFlushCBOpLists) {
284         if (!onFlushOpList->execute(flushState)) {
285             SkDebugf("WARNING: onFlushOpList failed to execute.\n");
286         }
287         SkASSERT(onFlushOpList->unique());
288         onFlushOpList = nullptr;
```

xref: /external/skia/src/gpu/GrDrawingManager.cpp

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

```
    return result;
}

bool GrDrawingManager::executeOpLists(int startIndex, int stopIndex, GrOpFlushState* flushState) {
    SkASSERT(startIndex <= stopIndex && stopIndex <= fOpLists.count());

    GrResourceProvider* resourceProvider = fContext->contextPriv().resourceProvider();
    bool anyOpListsExecuted = false;

    for (int i = startIndex; i < stopIndex; ++i) {
        if (!fOpLists[i]) {
            continue;
        }

        if (resourceProvider->explicitlyAllocateGPUResources()) {
            if (!fOpLists[i]->isInstantiated()) {
                // If the backing surface wasn't allocated drop the draw of the entire opList.
                fOpLists[i] = nullptr;
                continue;
            }
        } else {
            if (!fOpLists[i]->instantiate(resourceProvider)) {
                SkDebugf("OpList failed to instantiate.\n");
                fOpLists[i] = nullptr;
                continue;
            }
        }

        // TODO: handle this instantiation via lazy surface proxies?
        // Instantiate all deferred proxies (being built on worker threads) so we can upload them
        fOpLists[i]->instantiateDeferredProxies(fContext->contextPriv().resourceProvider());
        fOpLists[i]->prepare(flushState);
    }

    // Upload all data to the GPU
    flushState->preExecuteDraws();

    // Execute the onFlush op lists first, if any.
    for (sk_sp<GrOpList>& onFlushOpList : fOnFlushCBOpLists) {
        if (!onFlushOpList->execute(flushState)) {
            SkDebugf("WARNING: onFlushOpList failed to execute.\n");
        }
        SkASSERT(onFlushOpList->unique());
        onFlushOpList = nullptr;
```

xref: /external/skia/include/private/GrOpList.h

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

```
    // These four methods are invoked at flush time
    bool instantiate(GrResourceProvider* resourceProvider);
    // Instantiates any "threaded" texture proxies that are being prepared elsewhere
    void instantiateDeferredProxies(GrResourceProvider* resourceProvider);
    void prepare(GrOpFlushState* flushState);
    bool execute(GrOpFlushState* flushState) { return this->onExecute(flushState); }
```

xref: /external/skia/include/private/GrOpList.h

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

```
34      // These four methods are invoked at flush time
35      bool instantiate(GrResourceProvider* resourceProvider);
36      // Instantiates any "threaded" texture proxies that are being prepared elsewhere
37      void instantiateDeferredProxies(GrResourceProvider* resourceProvider);
38      void prepare(GrOpFlushState* flushState);
39      bool execute(GrOpFlushState* flushState) { return this->onExecute(flushState); }
40
```

xref: /external/skqp/src/gpu/GrRenderTargetOpList.cpp

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

```
134 // Ops and instantiate them here.
135 bool GrRenderTargetOpList::onExecute(GrOpFlushState* flushState) {
136     if (0 == fRecordedOps.count() && GrLoadOp::kClear != fColorLoadOp) {
137         return false;
138     }
139
140     SkASSERT(fTarget.get()->priv().peekRenderTarget());
141 #ifdef SK_BUILD_FOR_ANDROID_FRAMEWORK
142     TRACE_EVENT0("skia", TRACE_FUNC);
143 #endif
144
145     // TODO: at the very least, we want the stencil store op to always be discard (at this
146     // level). In Vulkan, sub-command buffers would still need to load & store the stencil buffer.
147     std::unique_ptr<GrGpuRTCommandBuffer> commandBuffer = create_command_buffer(
148                                                 flushState->gpu(),
149                                                 fTarget.get()->priv().peekRenderTarget(),
150                                                 fTarget.get()->origin(),
151                                                 fColorLoadOp, fLoadClearColor,
152                                                 fStencilLoadOp);
153     flushState->setCommandBuffer(commandBuffer.get());
154     commandBuffer->begin();
155
156     // Draw all the generated geometry.
157     for (int i = 0; i < fRecordedOps.count(); ++i) {
158         if (!fRecordedOps[i].fOp) {
159             continue;
160         }
161 #ifdef SK_BUILD_FOR_ANDROID_FRAMEWORK
162         TRACE_EVENT0("skia", fRecordedOps[i].fOp->name());
163 #endif
164
165         GrOpFlushState::OpArgs opArgs {
166             fRecordedOps[i].fOp.get(),
167             fTarget.get()->asRenderTargetProxy(),
168             fRecordedOps[i].fAppliedClip,
169             fRecordedOps[i].fDstProxy
170         };
171
172         flushState->setOpArgs(&opArgs);
173         fRecordedOps[i].fOp->execute(flushState);
174         flushState->setOpArgs(nullptr);
175     }
176
177     finish_command_buffer(commandBuffer.get());
178     flushState->setCommandBuffer(nullptr);
179
180     return true;
```

xref: /external/skqp/src/gpu/GrRenderTargetOpList.cpp

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

```
// Ops and instantiate them here.
bool GrRenderTargetOpList::onExecute(GrOpFlushState* flushState) {
    if (0 == fRecordedOps.count() && GrLoadOp::kClear != fColorLoadOp) {
        return false;
    }

    SkASSERT(fTarget.get()->priv().peekRenderTarget());
#ifdef SK_BUILD_FOR_ANDROID_FRAMEWORK
    TRACE_EVENT0("skia", TRACE_FUNC);
#endif

    // TODO: at the very least, we want the stencil store op to always be discard (at this
    // level). In Vulkan, sub-command buffers would still need to load & store the stencil buffer.
    std::unique_ptr<GrGpuRTCommandBuffer> commandBuffer = create_command_buffer(
                                                    flushState->gpu(),
                                                    fTarget.get()->priv().peekRenderTarget(),
                                                    fTarget.get()->origin(),
                                                    fColorLoadOp, fLoadClearColor,
                                                    fStencilLoadOp);
    flushState->setCommandBuffer(commandBuffer.get());
    commandBuffer->begin();

    // Draw all the generated geometry.
    for (int i = 0; i < fRecordedOps.count(); ++i) {
        if (!fRecordedOps[i].fOp) {
            continue;
        }
#ifdef SK_BUILD_FOR_ANDROID_FRAMEWORK
        TRACE_EVENT0("skia", fRecordedOps[i].fOp->name());
#endif

        GrOpFlushState::OpArgs opArgs {
            fRecordedOps[i].fOp.get(),
            fTarget.get()->asRenderTargetProxy(),
            fRecordedOps[i].fAppliedClip,
            fRecordedOps[i].fDstProxy
        };

        flushState->setOpArgs(&opArgs);
        fRecordedOps[i].fOp->execute(flushState);
        flushState->setOpArgs(nullptr);
    }

    finish_command_buffer(commandBuffer.get());
    flushState->setCommandBuffer(nullptr);

    return true;
}
```

xref: /external/skia/src/gpu/ops/GrOp.h

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

```
    /**
     * Called prior to executing. The op should perform any resource creation or data transfers
     * necessary before execute() is called.
     */
    void prepare(GrOpFlushState* state) { this->onPrepare(state); }

    /** Issues the op's commands to GrGpu. */
    void execute(GrOpFlushState* state) { this->onExecute(state); }

    /** Used for spewing information about ops when debugging. */
    virtual SkString dumpInfo() const {
        SkString string;
        string.appendf("OpBounds: [L: %.2f, T: %.2f, R: %.2f, B: %.2f]\n",
                       fBounds.fLeft, fBounds.fTop, fBounds.fRight, fBounds.fBottom);
        return string;
    }

protected:
    /**
```

xref: /external/skia/src/gpu/ops/GrOp.h

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

```
144     /**
145      * Called prior to executing. The op should perform any resource creation or data transfers
146      * necessary before execute() is called.
147      */
148     void prepare(GrOpFlushState* state) { this->onPrepare(state); }
149
150     /** Issues the op's commands to GrGpu. */
151     void execute(GrOpFlushState* state) { this->onExecute(state); }
152
153     /** Used for spewing information about ops when debugging. */
154     virtual SkString dumpInfo() const {
155         SkString string;
156         string.appendf("OpBounds: [L: %.2f, T: %.2f, R: %.2f, B: %.2f]\n",
157                        fBounds.fLeft, fBounds.fTop, fBounds.fRight, fBounds.fBottom);
158         return string;
159     }
160
161 protected:
162     /**
```

xref: /external/skia/src/gpu/ops/GrMeshDrawOp.cpp

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

```
15 void GrMeshDrawOp::onPrepare(GrOpFlushState* state) { this->onPrepareDraws(state); }
16
17 void* GrMeshDrawOp::PatternHelper::init(Target* target, size_t vertexStride,
18                                         const GrBuffer* indexBuffer, int verticesPerRepetition,
19                                         int indicesPerRepetition, int repeatCount) {
20     SkASSERT(target);
21     if (!indexBuffer) {
22         return nullptr;
23     }
24     const GrBuffer* vertexBuffer;
25     int firstVertex;
26     int vertexCount = verticesPerRepetition * repeatCount;
27     void* vertices =
28             target->makeVertexSpace(vertexStride, vertexCount, &vertexBuffer, &firstVertex);
29     if (!vertices) {
30         SkDebugf("Vertices could not be allocated for instanced rendering.");
31         return nullptr;
32     }
33     SkASSERT(vertexBuffer);
34     size_t ibSize = indexBuffer->gpuMemorySize();
35     int maxRepetitions = static_cast<int>(ibSize / (sizeof(uint16_t) * indicesPerRepetition));
36
37     fMesh.setIndexedPatterned(indexBuffer, indicesPerRepetition, verticesPerRepetition,
38                               repeatCount, maxRepetitions);
39     fMesh.setVertexData(vertexBuffer, firstVertex);
40     return vertices;
41 }
42
43 void GrMeshDrawOp::PatternHelper::recordDraw(Target* target, const GrGeometryProcessor* gp,
44                                              const GrPipeline* pipeline) {
45     target->draw(gp, pipeline, fMesh);
46 }
47
48 void* GrMeshDrawOp::QuadHelper::init(Target* target, size_t vertexStride, int quadsToDraw) {
49     sk_sp<const GrBuffer> quadIndexBuffer = target->resourceProvider()->refQuadIndexBuffer();
50     if (!quadIndexBuffer) {
51         SkDebugf("Could not get quad index buffer.");
52         return nullptr;
53     }
54     return this->INHERITED::init(target, vertexStride, quadIndexBuffer.get(), kVerticesPerQuad,
55                                  kIndicesPerQuad, quadsToDraw);
56 }
57
58 void GrMeshDrawOp::onExecute(GrOpFlushState* state) {
59     state->executeDrawsAndUploadsForMeshDrawOp(this->uniqueID(), this->bounds());
60 }
61
```

xref: /external/skia/src/gpu/ops/GrMeshDrawOp.cpp

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

```
15  void GrMeshDrawOp::onPrepare(GrOpFlushState* state) { this->onPrepareDraws(state); }
16
17  void* GrMeshDrawOp::PatternHelper::init(Target* target, size_t vertexStride,
18                                          const GrBuffer* indexBuffer, int verticesPerRepetition,
19                                          int indicesPerRepetition, int repeatCount) {
20      SkASSERT(target);
21      if (!indexBuffer) {
22          return nullptr;
23      }
24      const GrBuffer* vertexBuffer;
25      int firstVertex;
26      int vertexCount = verticesPerRepetition * repeatCount;
27      void* vertices =
28              target->makeVertexSpace(vertexStride, vertexCount, &vertexBuffer, &firstVertex);
29      if (!vertices) {
30          SkDebugf("Vertices could not be allocated for instanced rendering.");
31          return nullptr;
32      }
33      SkASSERT(vertexBuffer);
34      size_t ibSize = indexBuffer->gpuMemorySize();
35      int maxRepetitions = static_cast<int>(ibSize / (sizeof(uint16_t) * indicesPerRepetition));
36
37      fMesh.setIndexedPatterned(indexBuffer, indicesPerRepetition, verticesPerRepetition,
38                                repeatCount, maxRepetitions);
39      fMesh.setVertexData(vertexBuffer, firstVertex);
40      return vertices;
41  }
42
43  void GrMeshDrawOp::PatternHelper::recordDraw(Target* target, const GrGeometryProcessor* gp,
44                                               const GrPipeline* pipeline) {
45      target->draw(gp, pipeline, fMesh);
46  }
47
48  void* GrMeshDrawOp::QuadHelper::init(Target* target, size_t vertexStride, int quadsToDraw) {
49      sk_sp<const GrBuffer> quadIndexBuffer = target->resourceProvider()->refQuadIndexBuffer();
50      if (!quadIndexBuffer) {
51          SkDebugf("Could not get quad index buffer.");
52          return nullptr;
53      }
54      return this->INHERITED::init(target, vertexStride, quadIndexBuffer.get(), kVerticesPerQuad,
55                                   kIndicesPerQuad, quadsToDraw);
56  }
57
58  void GrMeshDrawOp::onExecute(GrOpFlushState* state) {
59      state->executeDrawsAndUploadsForMeshDrawOp(this->uniqueID(), this->bounds());
60  }
61
```

xref: /external/skia/src/gpu/GrOpFlushState.cpp

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

```
32  GrGpuRTCommandBuffer* GrOpFlushState::rtCommandBuffer() {
33      return fCommandBuffer->asRTCommandBuffer();
34  }
35
36  void GrOpFlushState::executeDrawsAndUploadsForMeshDrawOp(uint32_t opID, const SkRect& opBounds) {
37      SkASSERT(this->rtCommandBuffer());
38      while (fCurrDraw != fDraws.end() && fCurrDraw->fOpID == opID) {
39          GrDeferredUploadToken drawToken = fTokenTracker->nextTokenToFlush();
40          while (fCurrUpload != fInlineUploads.end() &&
41                 fCurrUpload->fUpdateBeforeToken == drawToken) {
42              this->rtCommandBuffer()->inlineUpload(this, fCurrUpload->fUpload);
43              ++fCurrUpload;
44          }
45          SkASSERT(fCurrDraw->fPipeline->proxy() == this->drawOpArgs().fProxy);
46          this->rtCommandBuffer()->draw(*fCurrDraw->fPipeline, *fCurrDraw->fGeometryProcessor,
47                                        fMeshes.begin() + fCurrMesh, nullptr, fCurrDraw->fMeshCnt,
48                                        opBounds);
49          fCurrMesh += fCurrDraw->fMeshCnt;
50          fTokenTracker->flushToken();
51          ++fCurrDraw;
52      }
53  }
```

xref: /external/skia/src/gpu/GrOpFlushState.cpp

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

```
32 GrGpuRTCommandBuffer* GrOpFlushState::rtCommandBuffer() {
33     return fCommandBuffer->asRTCommandBuffer();
34 }
35
36 void GrOpFlushState::executeDrawsAndUploadsForMeshDrawOp(uint32_t opID, const SkRect& opBounds) {
37     SkASSERT(this->rtCommandBuffer());
38     while (fCurrDraw != fDraws.end() && fCurrDraw->fOpID == opID) {
39         GrDeferredUploadToken drawToken = fTokenTracker->nextTokenToFlush();
40         while (fCurrUpload != fInlineUploads.end() &&
41                fCurrUpload->fUploadBeforeToken == drawToken) {
42             this->rtCommandBuffer()->inlineUpload(this, fCurrUpload->fUpload);
43             ++fCurrUpload;
44         }
45         SkASSERT(fCurrDraw->fPipeline->proxy() == this->drawOpArgs().fProxy);
46         this->rtCommandBuffer()->draw(*fCurrDraw->fPipeline, *fCurrDraw->fGeometryProcessor,
47                                       fMeshes.begin() + fCurrMesh, nullptr, fCurrDraw->fMeshCnt,
48                                       opBounds);
49         fCurrMesh += fCurrDraw->fMeshCnt;
50         fTokenTracker->flushToken();
51         ++fCurrDraw;
52     }
53 }
```

xref: /external/skia/src/gpu/GrGpuCommandBuffer.cpp

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

```
18
19 void GrGpuRTCommandBuffer::clear(const GrFixedClip& clip, GrColor color) {
20 #ifdef SK_DEBUG
21     GrRenderTarget* rt = fRenderTarget;
22     SkASSERT(rt);
23     SkASSERT(!clip.scissorEnabled() ||
24              (SkIRect::MakeWH(rt->width(), rt->height()).contains(clip.scissorRect()) &&
25               SkIRect::MakeWH(rt->width(), rt->height()) != clip.scissorRect()));
26 #endif
27     this->onClear(clip, color);
28 }
29
30 void GrGpuRTCommandBuffer::clearStencilClip(const GrFixedClip& clip, bool insideStencilMask)
31     this->onClearStencilClip(clip, insideStencilMask);
32 }
33
34 bool GrGpuRTCommandBuffer::draw(const GrPipeline& pipeline,
35                                 const GrPrimitiveProcessor& primProc,
36                                 const GrMesh meshes[],
37                                 const GrPipeline::DynamicState dynamicStates[],
38                                 int meshCount,
39                                 const SkRect& bounds) {
40 #ifdef SK_DEBUG
41     SkASSERT(!primProc.hasInstanceAttribs() || this->gpu()->caps()->instanceAttribSupport());
42     SkASSERT(!primProc.willUsePrimitiveRestart() || this->gpu()->caps()->usePrimitiveRestart());
43     for (int i = 0; i < meshCount; ++i) {
44         SkASSERT(!GrPrimTypeRequiresGeometryShaderSupport(meshes[i].primitiveType()) ||
45                  this->gpu()->caps()->shaderCaps()->geometryShaderSupport());
46         SkASSERT(primProc.hasVertexAttribs() == meshes[i].hasVertexData());
47         SkASSERT(primProc.hasInstanceAttribs() == meshes[i].isInstanced());
48     }
49 #endif
50     auto resourceProvider = this->gpu()->getContext()->contextPriv().resourceProvider();
51
52     if (pipeline.isBad() || !primProc.instantiate(resourceProvider)) {
53         return false;
54     }
55
56     if (primProc.numAttribs() > this->gpu()->caps()->maxVertexAttributes()) {
57         this->gpu()->stats()->incNumFailedDraws();
58         return false;
59     }
60     this->onDraw(pipeline, primProc, meshes, dynamicStates, meshCount, bounds);
61     return true;
62 }
63
64
```

xref: /external/skia/src/gpu/GrGpuCommandBuffer.cpp

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

```
18
19 void GrGpuRTCommandBuffer::clear(const GrFixedClip& clip, GrColor color) {
20 #ifdef SK_DEBUG
21     GrRenderTarget* rt = fRenderTarget;
22     SkASSERT(rt);
23     SkASSERT(!clip.scissorEnabled() ||
24              (SkIRect::MakeWH(rt->width(), rt->height()).contains(clip.scissorRect()) &&
25               SkIRect::MakeWH(rt->width(), rt->height()) != clip.scissorRect()));
26 #endif
27     this->onClear(clip, color);
28 }
29
30 void GrGpuRTCommandBuffer::clearStencilClip(const GrFixedClip& clip, bool insideStencilMask)
31     this->onClearStencilClip(clip, insideStencilMask);
32 }
33
34 bool GrGpuRTCommandBuffer::draw(const GrPipeline& pipeline,
35                                 const GrPrimitiveProcessor& primProc,
36                                 const GrMesh meshes[],
37                                 const GrPipeline::DynamicState dynamicStates[],
38                                 int meshCount,
39                                 const SkRect& bounds) {
40 #ifdef SK_DEBUG
41     SkASSERT(!primProc.hasInstanceAttribs() || this->gpu()->caps()->instanceAttribSupport());
42     SkASSERT(!primProc.willUsePrimitiveRestart() || this->gpu()->caps()->usePrimitiveRestart());
43     for (int i = 0; i < meshCount; ++i) {
44         SkASSERT(!GrPrimTypeRequiresGeometryShaderSupport(meshes[i].primitiveType()) ||
45                  this->gpu()->caps()->shaderCaps()->geometryShaderSupport());
46         SkASSERT(primProc.hasVertexAttribs() == meshes[i].hasVertexData());
47         SkASSERT(primProc.hasInstanceAttribs() == meshes[i].isInstanced());
48     }
49 #endif
50     auto resourceProvider = this->gpu()->getContext()->contextPriv().resourceProvider();
51
52     if (pipeline.isBad() || !primProc.instantiate(resourceProvider)) {
53         return false;
54     }
55
56     if (primProc.numAttribs() > this->gpu()->caps()->maxVertexAttributes()) {
57         this->gpu()->stats()->incNumFailedDraws();
58         return false;
59     }
60     this->onDraw(pipeline, primProc, meshes, dynamicStates, meshCount, bounds);
61     return true;
62 }
63
64
```

xref: /external/skia/src/gpu/gl/GrGLGpuCommandBuffer.h

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

```
68
69     void insertEventMarker(const char* msg) override {
70         fGpu->insertEventMarker(msg);
71     }
72
73     void inlineUpload(GrOpFlushState* state, GrDeferredTextureUploadFn& upload) override
74         state->doUpload(upload);
75     }
76
77     void copy(GrSurface* src, GrSurfaceOrigin srcOrigin, const SkIRect& srcRect,
78               const SkIPoint& dstPoint) override {
79         fGpu->copySurface(fRenderTarget, fOrigin, src, srcOrigin, srcRect, dstPoint);
80     }
81
82     void submit() override {}
83
84 private:
85     GrGpu* gpu() override { return fGpu; }
86
87     void onDraw(const GrPipeline& pipeline,
88                 const GrPrimitiveProcessor& primProc,
89                 const GrMesh mesh[],
90                 const GrPipeline::DynamicState dynamicStates[],
91                 int meshCount,
92                 const SkRect& bounds) override {
93         SkASSERT(pipeline.renderTarget() == fRenderTarget);
94         fGpu->draw(pipeline, primProc, mesh, dynamicStates, meshCount);
95     }
96
```

xref: /external/skia/src/gpu/gl/GrGLGpuCommandBuffer.h

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

```
68
69      void insertEventMarker(const char* msg) override {
70          fGpu->insertEventMarker(msg);
71      }
72
73      void inlineUpload(GrOpFlushState* state, GrDeferredTextureUploadFn& upload) override
74          state->doUpload(upload);
75      }
76
77      void copy(GrSurface* src, GrSurfaceOrigin srcOrigin, const SkIRect& srcRect,
78                const SkIPoint& dstPoint) override {
79          fGpu->copySurface(fRenderTarget, fOrigin, src, srcOrigin, srcRect, dstPoint);
80      }
81
82      void submit() override {}
83
84  private:
85      GrGpu* gpu() override { return fGpu; }
86
87      void onDraw(const GrPipeline& pipeline,
88                  const GrPrimitiveProcessor& primProc,
89                  const GrMesh mesh[],
90                  const GrPipeline::DynamicState dynamicStates[],
91                  int meshCount,
92                  const SkRect& bounds) override {
93          SkASSERT(pipeline.renderTarget() == fRenderTarget);
94          fGpu->draw(pipeline, primProc, mesh, dynamicStates, meshCount);
95      }
96
```

xref: /external/skia/src/gpu/gl/GrGLGpu.cpp

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

```
2549
2550  void GrGLGpu::draw(const GrPipeline& pipeline,
2551                     const GrPrimitiveProcessor& primProc,
2552                     const GrMesh meshes[],
2553                     const GrPipeline::DynamicState dynamicStates[],
2554                     int meshCount) {
2555      this->handleDirtyContext();
2556
2557      bool hasPoints = false;
2558      for (int i = 0; i < meshCount; ++i) {
2559          if (meshes[i].primitiveType() == GrPrimitiveType::kPoints) {
2560              hasPoints = true;
2561              break;
2562          }
2563      }
2564      if (!this->flushGLState(pipeline, primProc, hasPoints)) {
2565          return;
2566      }
2567
2568      for (int i = 0; i < meshCount; ++i) {
2569          if (GrXferBarrierType barrierType = pipeline.xferBarrierType(*this->caps())) {
2570              this->xferBarrier(pipeline.renderTarget(), barrierType);
2571          }
2572
2573          if (dynamicStates) {
2574              if (pipeline.getScissorState().enabled()) {
2575                  GrGLRenderTarget* glRT = static_cast<GrGLRenderTarget*>(pipeline.renderTarget());
2576                  this->flushScissor(dynamicStates[i].fScissorRect,
2577                                     glRT->getViewport(), pipeline.proxy()->origin());
2578              }
2579          }
2580          if (this->glCaps().requiresCullFaceEnableDisableWhenDrawingLinesAfterNonLines() &&
2581              GrIsPrimTypeLines(meshes[i].primitiveType()) &&
2582              !GrIsPrimTypeLines(fLastPrimitiveType)) {
2583              GL_CALL(Enable(GR_GL_CULL_FACE));
2584              GL_CALL(Disable(GR_GL_CULL_FACE));
2585          }
2586          meshes[i].sendToGpu(primProc, this);
2587          fLastPrimitiveType = meshes[i].primitiveType();
2588      }
2589
```