KDAB

Optimizing Rendering of Qt Quick 2 Applications

Qt World Summit 2019, Berlin

Giuseppe D'Angelo

giuseppe.dangelo@kdab.com

The Qt, OpenGL and C++ experts

About me

- Senior Software Engineer, KDAB
- Developer & Trainer
- Qt developer since ~2000
  - Qt contributor
- Ask me about Qt Core, Qt Gui, Qt Quick, …
  - And about Modern C++, 3D graphics

Agenda

Performance in Qt Quick: many aspects

- C++ performance
  - CPU, I/O, scalability, lock contention…
- JavaScript performance
  - QML compiler, incubation, bindings, animators…
- Rendering performance
  - Asset conditioning, offline processing…
  - Runtime performance when using OpenGL

You are here

Performance in rendering

- Draw **faster**
  - Draw the same things, in less time
  - Draw fewer things
  - Draw simpler things
  - Trade memory for more speed
- Use less resources (typically: **memory**)
- Other, somewhat related concerns:
  - Improve battery/power consumption
  - Reduce drawing latency

Detect and avoid overdraw

Overdrawing

- The easiest way of being faster is to *draw less*
- Ideally, we would like Qt Quick to draw all and only the elements the user can see
- However, Qt Quick draws *every* item with `visible` set to `true`, including:
  - Items that are out of bounds
  - Items that are completely obscured by opaque ones stacked on top
  - Items that are clipped by ancestors with `clip: true`

Overdrawing

- Qt Quick does not implement any optimization to prevent overdrawing
  - No Z-Fill pass, no frustum culling, no occlusion culling, etc.
- ☞ We must manually hide items that are not visible by the user
  - Using built-in elements (e.g. StackView) helps

Detecting Overdrawing

- GammaRay can visualize overdrawing, also highlighting items which are visible but out of view
  - Or: `export QSG_VISUALIZE=overdraw`
  - Or: use some OpenGL debugging utility; look for “overdraw”, “Z complexity”, “culled primitives”, etc.

KDAB

Consider caching items that are expensive to render.

Caching

- Caching means trading (video) memory for rendering speed
- Extremely useful in case we have some expensive element to render
- Usual culprit: shader effects
  - Blur, opacity masks, colorizations…

Why caching complex items?

- OpenGL does not support partial updates
- Every time anything changes in our scene, Qt Quick has to repaint *everything* from scratch
  - Including items that have not changed at all!

Detecting “expensive to render” items

- Use an OpenGL tracer/profiler and investigate
  - NSight, apitrace, …
- Shader effects (blur, shadows…) are good candidates for caching

Caching in Qt Quick

- Qt Quick has built-in support for caching
- Simply set: `layer.enabled: true`
- ☞ Consider caching bigger elements (root of trees containing many elements), and not each and every small one

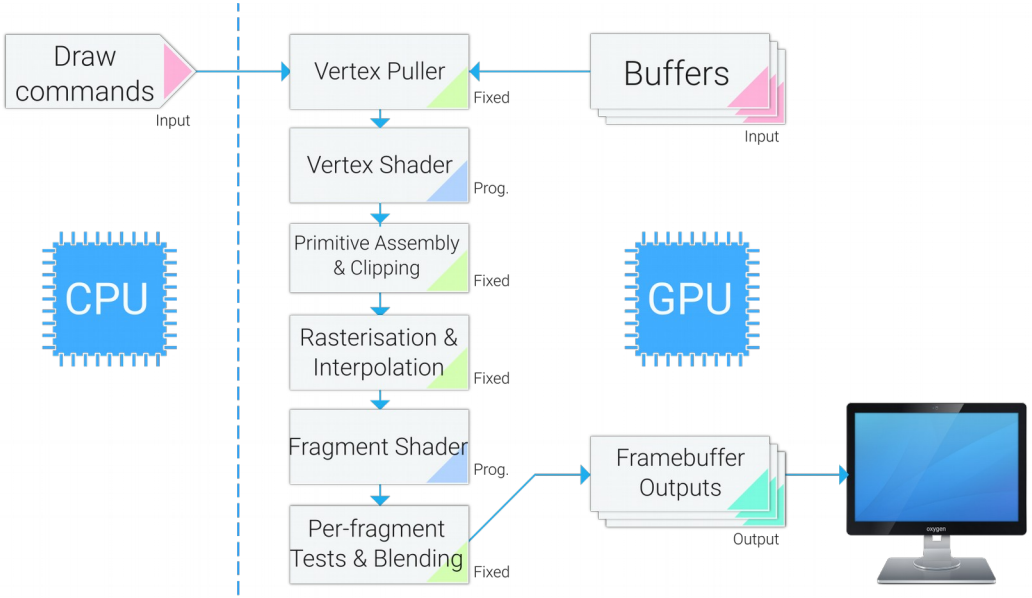
Understand OpenGL

OpenGL

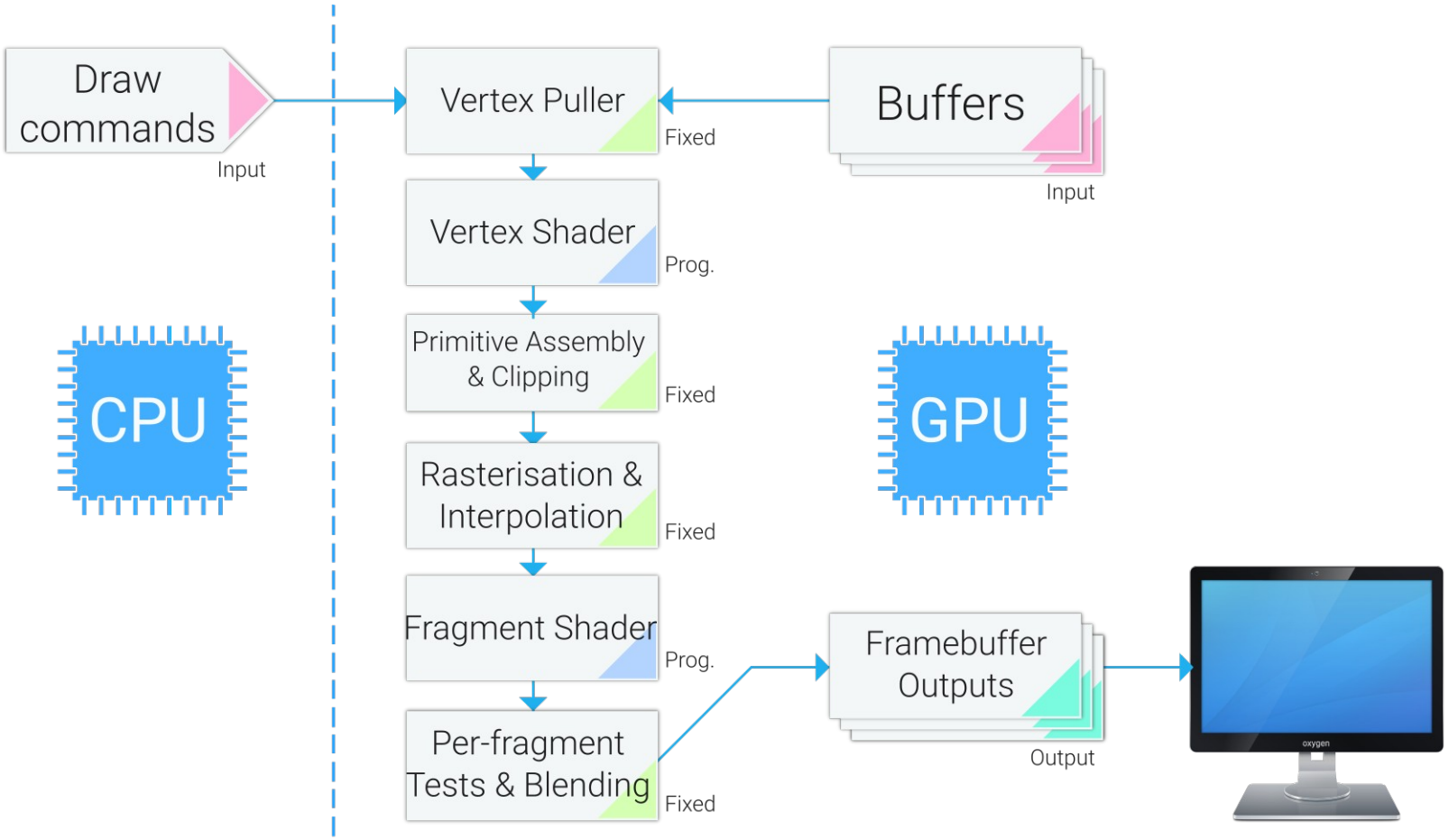
- OpenGL is (a specification for) a C API for 3D graphics
  - Cross-platform, royalty-free
- It allows us to exploit the computational power of GPUs
- Very long history and evolution
  - A couple of paradigm shifts happened along the way
  - Today we tend to use “Modern OpenGL”
- Qt has always had excellent support for OpenGL

OpenGL

- OpenGL defines a complex state machine around a dataflow processing pipeline
- Inputs of the pipeline get processed according to the current state
- The output is pixels on the screen



OpenGL pipeline



Drawing in OpenGL

- In order to draw something, we must set up lots of state upfront:
  - Inputs to the pipeline, usually in the form of Vertex Buffer Objects
  - What certain programmable processing steps should do (Vertex/Fragment Shaders)
  - Ancillary data, such as: uniforms, textures, etc.
  - Countless extra switches and knobs
- When everything is set up, we can issue draw commands
- State never changes during a draw command

Drawing in OpenGL

- Drawing multiple objects is usually a rinse-and-repeat process:
  1. Set all the required OpenGL state
  2. Issue one (or more) draw commands using that state
  3. Go back to 1. (until we've drawn everything)
- Moral lesson: we can't draw different objects together if they require different state

Understand OpenGL Performance

OpenGL Performance

- The best way to think about OpenGL performance is comparing OpenGL to a high-speed train.
- What are the performance charateristics of such a machine?

High-Speed Trains vs OpenGL

- Fast: capable of transporting hundreds of passengers at high speeds (> 300km/h).

- Fast: capable of rendering (hundreds of) thousands of geometric primitives per second.

High-Speed Trains vs OpenGL

- Constrained: only moves on certain paths.
- Tracks need to be laid down *before* we can have a train running on them.

- Constrained: only understands certain geometric primitives and draw commands.
- Lots of state needs to be set *before* drawing anything.

High-Speed Trains vs OpenGL

- Trains have a lot of inertia.
  - They take forever to speed up and slow down
- Stopping a high-speed train very often is a no go.

- OpenGL has lots of “inertia”
  - GPUs are complicated to set up
  - The command latency is a factor
- **Stopping the pipeline too often is a no go.**

Stopping too often

- We need to stop every time we need to change *any part* of the OpenGL state
  - Buffers to read from; that is: **which objects to render**
  - Which **texture(s) to use**
  - **Opacity** settings, **clipping** settings
  - **Shaders used** (material/appearance)
  - **Render target** (e.g. if drawing offscreen, maybe as part of a QQuickPaintedItem or a shader effect)
- As we have seen, having too many changes is detrimental to performance.

Stopping too often

- In the ideal scenario, we would set OpenGL up in a way that it can render a huge number of elements in the scene in one go (one draw command or so), without changing state.
  - A huge part of of “Modern OpenGL” is all about this: Instanced Drawing, Uniform Buffer Objects, Texture Arrays, Bindless Textures, Indirect Drawing…
- However, this is very very hard in practice
  - Generality of Qt Quick rendering (we're not building a specialized engine)
  - Support for legacy APIs in the Qt Quick renderer (hello, OpenGL ES 2)

Minimize state changes in Qt Quick

Qt Quick Rendering

- Qt Quick renders a given scene using OpenGL
- The elements that we add into a Qt Quick application (Image, Text, etc.) get converted into OpenGL commands
- The mapping between elements in a scene and OpenGL commands is not 1:1
  - It would issue lots of draw commands!

The Qt Quick Scenegraph

- The Qt Quick elements in a scene create a data structure called the **scenegraph**
- The scenegraph describes *how* to render a given scene:
  - What are the geometries to draw (usually triangles), the shaders, the textures
  - What is the necessary OpenGL state to set to draw them
- You can use tools such as GammaRay to visualize the scenegraph contents

GammaRay Quick Scenes Settings Help

- Actions
- Font Browser
- Graphics Scenes
- KJobs
- Locales
- Messages
- Meta Objects
- Meta Types
- Mime Types
- Models
- Objects
- Quick Scenes
- Resources
- Script Engines
- Selection Models
- Signals
- Standard Paths
- State Machines
- Styles
- Text Codecs
- Text Documents
- Timers
- Translations
- Web Inspector
- Widgets

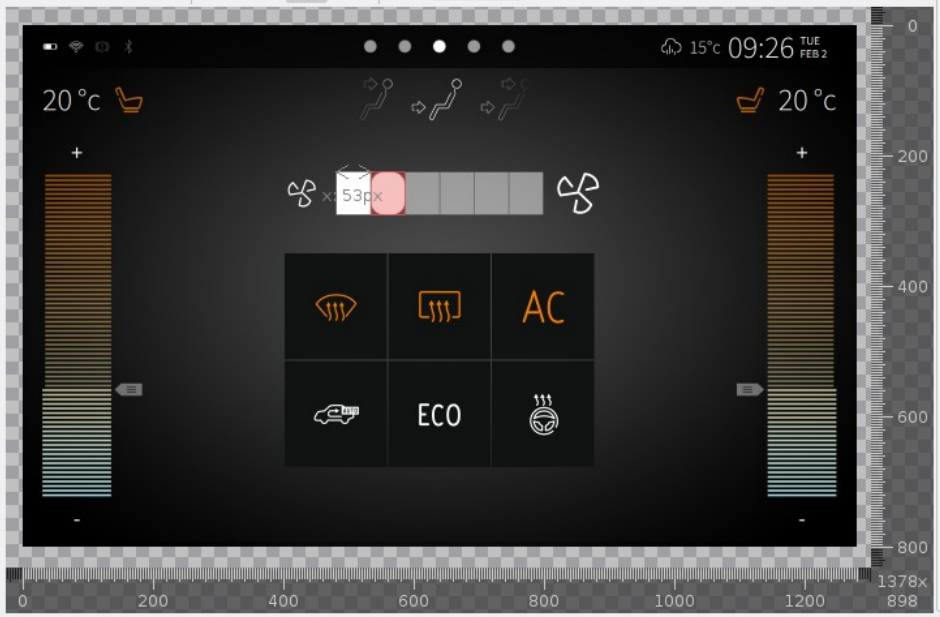
QQuickView[this=0x2ddaca0]

Search

Object	Type
0x28b01c0	Transform Node
0x28c3020	Transform Node
0x2f8f200	Transform Node
0x2fc07a0	Transform Node
0x2fc2420	Transform Node
0x2fc2590	Transform Node
0x2fc40a0	Transform Node
0x2fc41f0	Transform Node
0x2fe09d0	Transform Node
0x2fe5ea0	Geometry Node
0x2fe33e0	Transform Node
0x2fc4f30	Transform Node
0x2fc5d20	Transform Node
0x2fc5e90	Transform Node
0x2fc9620	Transform Node
0x2fd24a0	Geometry Node

Items | Scene Graph

50%

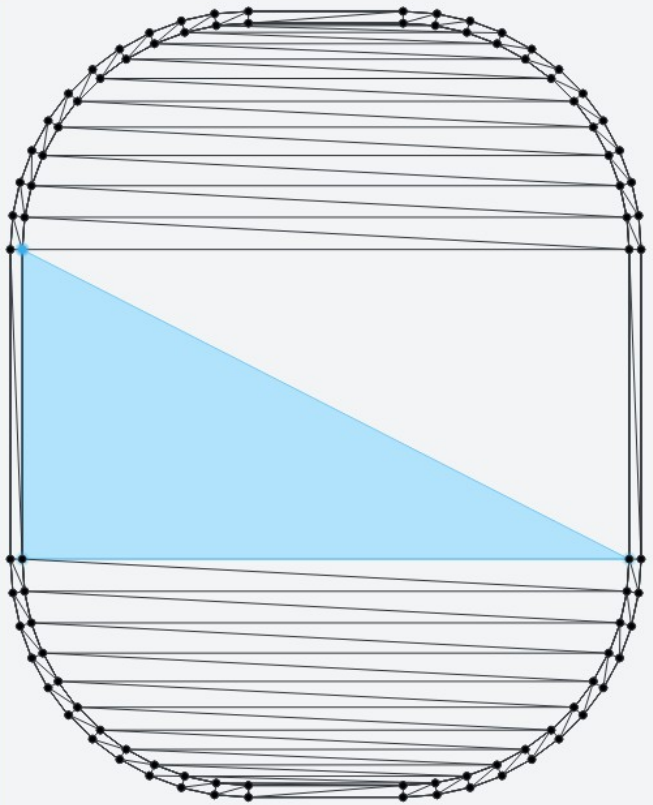


Properties | Material | Geometry

Raw Vertex Data

	vertex	vertexColor	vertexOffset
79	52.1899, 14.3656	0, 0, 0, 0	14.3656, 0
80	0.81007, 14.3656	0, 0, 0, 0	17.2963, nan
81	51.8066, 17.2963	255, 255, 255, 255	17.2963, nan
82	1.19335, 17.2963	255, 255, 255, 255	17.2963, nan
83	51.8066, 17.2963	170, 170, 170, 255	17.2963, nan
84	1.19335, 17.2963	170, 170, 170, 255	17.154, nan
85	52.7965, 17.154	170, 170, 170, 255	17.154, nan
86	0.203531, 17.154	170, 170, 170, 255	17.154, 0
87	52.7965, 17.154	0, 0, 0, 0	17.154, 0
88	0.203531, 17.154	0, 0, 0, 0	20.0003, nan
89	52, 20.0003	255, 255, 255, 255	20.0003, nan
90	0.999998, 20.0003	255, 255, 255, 255	20.0003, nan
91	52, 20.0003	170, 170, 170, 255	20.0003, nan
92	0.999998, 20.0003	170, 170, 170, 255	20.0003, nan
93	53, 20.0003	170, 170, 170, 255	20.0003, nan
94	-1.90735e-06, 20.0003	170, 170, 170, 255	20.0003, 0
95	53, 20.0003	0, 0, 0, 0	20.0003, 0
96	-1.90735e-06, 20.0003	0, 0, 0, 0	46, nan
97	52, 46	255, 255, 255, 255	46, nan
98	1, 46	255, 255, 255, 255	46, nan
99	52, 46	170, 170, 170, 255	46, nan
100	1, 46	170, 170, 170, 255	46, nan
101	53, 46	170, 170, 170, 255	46, nan
102	0, 46	170, 170, 170, 255	46, 0
103	53, 46	0, 0, 0, 0	46, 0
104	0, 46	0, 0, 0, 0	48.704, nan
105	51.8066, 48.704	255, 255, 255, 255	48.704, nan
106	1.1934, 48.704	255, 255, 255, 255	48.704, nan
107	51.8066, 48.704	170, 170, 170, 255	48.704, nan

Preview

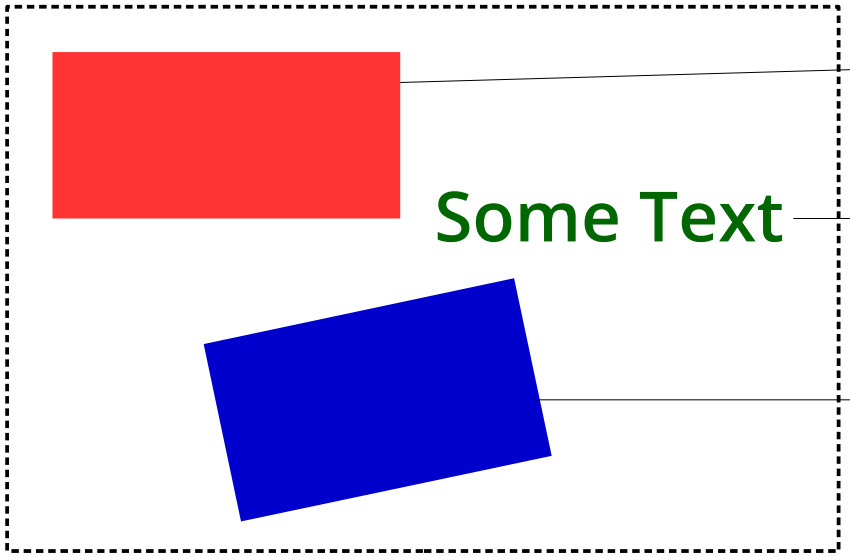


Drawing mode: GL_TRIANGLE_STRIP

The Qt Quick Renderer

- The Qt Quick Renderer traverses the scenegraph and renders its contents using OpenGL
- Using a scenegraph unlocks many optimization possibilities
- Qt Quick can **analyze** and **optimize** the scenegraph
  - As it contains *all* the data required to render

Rendering the Scenegraph



- Geometry
- Material (solid fill)

- Geometry
- Material (text)
- Textures

- Geometry
- Material (solid fill)
- Transformation

1. Coalesce the geometries for the rectangles.
2. Set the solid fill material.
3. **Draw both rectangles (in one draw call!).**
4. Set the text material.
5. **Draw the text.**

Qt Quick Scene

Scenegraph

Renderer

Batching

- In order to maximize OpenGL performance, Qt Quick will automatically try to draw together scene graph nodes that require the same OpenGL state
- This works by merging together the geometries of multiple elements, so to draw them all in one draw call
- We call this process **batching**

Visualizing Batching

- We can use GammaRay to visualize batching at runtime
  - Or set QSG_VISUALIZE=batches, QSG_RENDERER_DEBUG=render
  - QSG_VISUALIZE=clip for visualizing clipping
- Each different color means a different batch is being submitted to draw the corresponding elements
- ☞ Too many batches is bad!

When is batching applied?

- The Qt Quick Renderer uses a few simple heuristics to merge multiple elements in the same batch.
- Rule of thumb: any change of
  - Opacity
  - Clipping
  - Material (i.e. shader + textures + other uniforms)
  - Render target (ShaderEffect, layer, QQuickPaintedItem)

  results in a different batch.
- Visual overlapping and complex transformations also take a role.

Wait, is this a real problem?

- **YES!**
- I've seen so much code that tries to be “clever” and results in hundreds of unnecessary draw calls
  - The simplicity of Qt Quick 2 is a double-edge sword sometimes
- “Everything works fine on desktop but terribly slow on embedded/mobile”
  - “Qt Quick is terrible!” “Linux is terrible!” “Linux drivers are terrible!”

Questions?

KDAB

Thanks!

giuseppe.dangelo@kdab.com