LibreOffice graphics subsystems:

System-Dependent PrimitiveRenderers (SDPR's)

Providing a working Example and report about progress/findings during development

Armin Le Grand

2023-02-04, 17:10, H.1308 Rolin

FOSDEM 2023

What is it about?

I talked about SDPRs at last LibreOffice Conference (link is here). SDPRs are an alternative to render **completely without** using VCL/OutputDevice or Backends. Instead, render Primitives to a system-specific graphic sub-system directly.

This next step in rendering for the office was planned since the design of Primitives, but is stuck due to missing resources.

When preparing that talk, I started to develop a reference implementation and promised to finish it – what I did in the meantime.

This talk is about that SDPR (using Direct2D) and some additional experiments I did to also offer possible solutions for ‘old’ Paint-code.

(1) What happens when Painting?

- EditViews use Primitives quite extensively, e.g. SdrObjects, full EditView display in Draw/Impress, all GraphicObjects, Overlay, TextSelection, …
- Created/Buffered Primitives get thrown at a Primitive-Processor (Renderer) to get them painted
- Currently this uses a Renderer that targets to use OutputDevice, called VCLProcessor, versions for targeting Pixel/Metafile exist
- The VCLPixelRenderer interprets/processes the Primitives, creates needed commands to call at OutputDevice to paint (tesselation), thus is hard coded/programmed against OutputDevice API

(2) What happens when Painting?

- OutputDevice does all possible stuff internally (partially crude, old, strange, hard-to-maintain stuff…) and calls a ‚Backend‘. Those Backends are the system-dependent implementations that e.g use Cairo/GDI+/X11/others to do the rendering. Originally this was GDI for Win and X11 for linux/unix
- The Primitives do eventually some decompositions - depending on the processor implementation, direct support of primitives to not trigger that is possible anytime :-)
- VCL‘s OutputDevice processes through about 4-6 layers internally, tessellating and preparing system-independent render data, old code paths/stuff. It already has some more modern parts (double, transformation, B2D* class usages), but ***still*** all old integer stuff and MapModes
- The Backends again can have quite some depth/overhead/buffering (have a look :-))

(1) Why did it develop to that State?

- Before Primitives/Processors there were direct ‚Paints' at SdrObjects (call it immediate mode) and everywhere unstructured distributed over the whole office (e.g. local ‚optimizied' repaints of single changes, etc…)
- These already used Backends, but there were only view of them. Often stuff was ‚added centrally' to OutputDevice (e.g. RTL anyone - of course not as transformation, but as hard repositioning)
- More hacks were/are in the upmost libraries/Apps (SW/SC) just because adding it there was known to be bad, but allows to do stuff without getting incompatible :-/ (build times in the past - meh)
- With Primitives things changed to a SceneGraph/Rendering approach for these objects. Just as in-between step and as proof of concept that VCL Processors were created initially, as a migration path.
- The long term/medium plan always was to have SDPRs. Unfortunately the transition to these and further work was interrupted by the fork to LibreOffice & the reactions to this, so we are stuck with these until today
- Originally I estimated to use these for a maximum of 4-5 years, replacing ASAP with SDPRs - in the order of importance of the target systems

(2) Why did it develop to that State?

- I did quite some presentations on Primitives and how they are intended to be used all those years, tried to find resources & filed tenders to continue that transition and be an ambassador to get support to drive this forward, but I barely succeeded
- I watched quite some work/resources being sunk into new Backends - what is one possibility to render using different renderer technologies on target systems, as e.g. OpenGL (did one already 20 years ago, same problems surfaced) and Skia (distributed throughout the whole office, many places with ‚adaptions‘,…)
- Think of It as a ‘Stack’ of four parts – Primives, Processor, OutputDevice, Backend. Just too complex and too deep
- All of these can - by principle - only be partially successful since they do *not* reduce the above described hierarchy/many layers of complexity and processing of the graphic data (sigh)
- A SDPR **replaces** the last three with a single, system-dependent Processor

(1) What did I do to impl that SDPR?

I had started to implement a demonstration/reference SDPR, using Direct2D on Windows. I had promised to continue that, and I did. We now have a Direct2D renderer for Win that can render Primitives *__without__* using OutputDevice/VCL or using any Backend at all, thus eliminating all these layers – completely. POC done.

It is feature-complete:

- All have-to support Primitives are implemented:
  - Four of them (Bitmap, PointArray, PolygonHairline, PolyPolygonColor)
- All have-to support grouping/encapsulating Primitives are implemented:
  - Three individual ones (Transparency, Invert, Mask)
  - Two standard ones (same in all processors: ModifiedColor, Transform)
- Quite some extended Primitives are added already:
  - Currently Eight, see below for more info

(2) What did I do to impl that SDPR?

The Direct2D SDPR renderer is implemented in a single source file in the drawinglayer project (2000+ lines?)

- It does not need any hacks/compromises/adaptions elsewhere in the office code to render on Direct2D
- It translates Primitive data directly to Direct2D calls/data as needed
- It uses the already available system-dependent buffering for path/bitmap data (which can be used for **any** system-dependent data anytime, but was not for ‚some' reasons..?)
- It does not need e.g. Bitmap/BitmapEx to be adapted in any form to that renderer
- It its quite fast, you can try it yourself: It's in master and can be activated by a ENV-Variable (TEST_SYSTEM_PRIMITIVE_RENDERER=true).

What extended Primitives are added?

- UnifiedTransparence: Optimizes the case that all child content is defined to have a single, unified transparency value
- PolygonStroke: Optimizes stroked lines, also fat lines (non-hairline) and the combination of both. On Direct2D, the tesselation could even be buffered. This is not done yet, but …
- FillGraphic: Optimizes PatternFilled areas, including that old ugly OffsetXY. Takes care of pixel fill, but also vector data like SVG/Metafile/PDF
- MarkerArray: e.g. Grid, BackgroundColor: fast color reset
- Line/FilledRectangle, SingleLine: Simplified Primitives for faster implementation of fallbacks (see below)

What else can be done?

What the renderer does not have (yet - can be extended in a compatible way anytime):

- Text - still uses decompose, so gets painted as AntiAliased PolyPolygons. Can be added anytime, e.g usage of DirectWrite, this time in an existing Direct2D **context**
- Gradients - many possibilities, e.g. just locally render non-AAed, but also may be done as custom ‚Effect' in Direct2D. A this place you **know** that a gradient needs to be painted and have the full definition and transformation
- Anything else that is still resource- or time-intense...

Thus, with some more work, this could easily be extended to product quality. It just would need the resources to do that.

It is far advanced over the state of a demonstration implementation already.

I used Direct2D as an example since I did not know too much about it (learning effect) and we do not have that yet - to test the concept and myself...

What also happened:

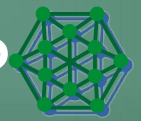
- Caolan started a SDPR implementation for Cairo based on that example implementation - thanks, man!
- It will need some more love, though, it‘s in a basic state but already renders something
- With a feature-complete SDPR for Cairo we would have a fast and working fallback for most/many cases, so this would be an important step to continue this to product quality state, too

So we now have POC of a working SDPR (even two) and the example implementation in drawinglayer (as orientation for others). No need to hack anything or to have Primitives in VCL library.

I also did some upstream stuff and added this to master during the journey - so this is out of the way, too, using that prototype experiment.

I also started a system-independent tooling library to support SDPR implementations. There are some cases between SDPRs where similar paint preparations have to be done.

How to get non-Primitive paint to SDPRs?



allotropia

So - this works for EditView visualizations, but

- not all Paints use Primitives fully (Writer/Calc/UI,...)
- what about printing?
  - Except Win all use PDF export anyways & we can keep VCLRenderer as fallback for some time, too…
  - OTOH Direct2D states to support printing, so another area to check before usage

Either change all those over time (hah!) or maybe there is a possibility to render VCL-calls to the OutputDevice API also on SDPRs? I checked that and did two further POC-experiments:

(A) Forward calls in the Backends to a SDPR

(B) Do a general DrawForwarder in OutputDevice

(A) Forward calls in the backends to SDPR

- Has not too many calls, quite lean API (14?)
- The Backends *are* libraries of their own, so they can just link against drawinglayer and use processors/primitives from there directly (thus no need to use primitives in VCL)
- Still has to step/chew through all that old stuff in all that layers of OutputDevice, so this concept will not get rid of that mayhem
- Has to create temporary Primitives to throw at the SDPR (could do on stack, but there is a clang-test that requires ref-counted stuff to be held/fed to a reference holder - is that clever..?)
- May need optimizations (Start|End/Draw for better optimizing in SDPRs, usage of draw cycles, processors as resources, …)
- Only a ***single*** Backend implementation is needed, that will work with any SDPR due to Primitives being unified in their definitions

It works just fine :-) I did a working prototype which redirects ::DrawRect(…), fill & line dependent on Pen/Brush, see gerrit (here). To test it out, just add as patch, compile & used mentioned ENV-Variable.

(B) Do a general DrawForwarder in OutputDevice

- I use a pure virtual class definition in VCL, offering calls to potentially all Paint-calls from OutputDevice you want/need to handle/forward (not all needed at once). All OutputDevice API that actually draws something may potentially use that
- If the forwarder says OK (using boolean retval), draw is done, else continue old stuff
- DrawForwarder would be a general way to isolate Metafile functionality, e.g. recording, PDF export, Printing, etc., and if set, use it - certainly too much cleanup work for now, maybe not worth it on OutputDevice, but systematically a good possible ‚split' of the do-it-all-in-one place old stuff. It would allow a 'structured' version of OutputDevice
- I Did a minimal DrawForwarder in drawinglayer, forwarding a factory/constructor for it to VCL
- VCL knows the pure virtual base class, so incarnates & calls (again) ::DrawRect(…) if activated
- The DrawForwarder implementation in drawinglayer can use all OutputDevice settings, the target OutputDevice is fully accessible to get data (Colors, LineStyle, FillStyle, Clipping/Region, Font, …)

It also works just fine :-), see working prototype at gerrit (here).

CAUTION: Both of these POCs ***need*** a SDPR, else you will get a loop-of-death immediately (!)

(1) How to go forward…?

Which way to continue? Which solution to use?

- Best:
  - Convert all LO to use Primitives everywhere
  - Implement SDPRs for all used/needed systems in order of importance
  - Keep VCLProcessor as fallback
- Medium:
  - Combine with solution (A) or (B)
    - I would strongly recommend (B), since (A) will keep the VCL implementations ‚alive' for eternity. When (B) is done, these layers of OutputDevice may be dismantled/removed completely (one day, just dreaming…)
    - (A) has the lighter API, but (B) has more potential from my POV
- Worst:
  - Keep stuff as it is, probably due to *still/again* not getting/activating resources for this, just because it's not visible/painful enough for the user – sigh...

(2) How to go forward…?

I am Still ready/capable/interested to do this, but...

- Already did the POC/Prototype & POCs of future migration possibilities – mostly in private
- Cannot continue doing that in private with the needed intensity to get that done

So - without getting **resources** - this will fail *__again__* and we will stay at VCL/OutputDevice forever - have fun then, and don’t complain!

We **have** choices - I presented the possibilities here…

(Probably no...) Time for Questions…?