Freedesktop Planet - Latest News

- Natalie Vock: Inside Mesa 26.0’s RADV RT improvements (2026/01/30 00:00)
  Mesa 26.0 is big for RADV’s ray tracing. In fact, it’s so big it single-handedly revived this blog. There are a lot of improvements to talk about, and some of them were in the making for a little over two years at this point. In this blog post I’ll focus on the things I myself worked on specifically, most of which revolve around how ray tracing pipelines are compiled and dispatched. Of course, there’s more than just what I did myself: Konstantin Seurer worked on a lot of very cool improvements to how we build BVHs, the data structure that RT hardware uses for the triangle soup making up the geometry in game scenes so the HW can trace rays against them efficiently. RT pipeline compilation The rest of this blog post will assume some basic idea of how GPU ray tracing and ray tracing pipelines work. I’ve written about this in more detail one-and-a-half years ago, in my blog post about RT pipeline being enabled by default. Let’s take a bit of a closer look at what I said about RT pipelines in RADV back then. In a footnote, I said: Any-hit and Intersection shaders are still combined into a single traversal shader. This still shows some of the disadvantages of the combined shader method, but generally compile times aren’t that ludicrous anymore. I spent a significant amount of time in that blogpost detailing about how there tend to be a really large number of shaders, and combining them into a single megashader is very slow because shader sizes get genuinely ridiculous at that point. So clearly, it was only a matter of time until the any-hit/intersection shader combination would blow up spectacularly on a spectacular number of shaders, as well. So there’s this thing called Unreal Engine For illustrating the issues with inlined any-hit/intersection shaders, I’ll use Unreal Engine as an example because I noticed it being particularly egregious here. This definitely was an issue with other RT games/workloads as well, and function calls will provide improvements there too. There’s a lot of people going around making fun of Unreal Engine these days, to the point of entire social media presences being built around mocking the ways in which UE is inefficient, slow, badly-designed bloatware and whatnot. Unfortunately, the most popular critics often know the least what they’re actually talking about. I feel compelled to point out here that while there certainly are reasonable complaints to be raised about UE and games made with it, I explicitly don’t want this section (or anything else in this post, really) to be misconstrued as “UE doing a bad thing”. As you’ll see, Unreal is really just using the RT pipeline API as designed. With the disclaimer aside, what does Unreal actually do here that made RADV fall over so hard? Let’s talk a bit about how big game engines handle shading and materials. As you’ll probably know already, to calculate how lighting interacts with objects in a scene, an application will usually run small programs called “shaders” on the GPU that, among other things1, calculate the colors different pixels have according to the material at that pixel. Different materials interact with light differently, and in a large world with tons of different materials, you might end up having a ton of different shaders. In a traditional raster setup you draw each object separately, so you can compile a lot of graphics pipelines for all of your materials, and then bind the correct one whenever you draw something with that material. However, this approach falls apart in ray tracing. Rays can shoot through the scene randomly and they can hit pretty much any object that’s loaded in at the moment. You can only ever use one ray tracing pipeline at once, so every single material that exists in your scene and may be hit by a ray needs to be present in the RT pipeline. The more materials a game has, the more ludicrous the number of shaders gets. Usually, this is most relevant for closest-hit shaders, because these are the shaders that get called for the object hit by the ray (where shading needs to be calculated). However, depending on your material setup, you may have something like translucent materials - where parts of the

material are “see-through”, and rays should go through these parts to reveal the scene behind it instead of stopping. This is where any-hit shaders come into play - any-hit shaders can instruct the driver to ignore a ray hitting a geometry, and instead keep searching for the next hit. If you have a ton of (potentially) translucent materials, that would translate into a lot of any-hit shaders being compiled for these materials. The design of RT pipelines is quite obviously written in a way that accounts for this. In the previous blogpost I already mentioned pipeline libraries - the idea is that a material could just be contained in a “library”, and if RT pipelines want to use it, they just need to link to the library instead of compiling the shading code all over again. This also allows for easy addition/removal of materials: Even though you have to re-create the RT pipeline, all you need to do is link to the already compiled libraries for the different materials. UE, particularly UE4, is a heavy user of libraries, which makes a lot of sense: It maps very well to what it’s trying to achieve. Everything’s good, as long as the driver doesn’t do silly things. Silly things like, for example, combining any-hit shaders into one big traversal shader. Doing something like that pretty much entirely side-steps the point of libraries. The traversal shader can only be compiled when all any-hit shaders are known, which is only at the very final linking step, which is supposed to be very fast… And if UE4, assuming the linking step is very fast, does that re-linking over and over, very often, what you end up with is horrible pipeline compilation stutter every few seconds. And in this case, it’s not really UE’s fault, even! Sorry for that, Unreal. Why can’t we just compile any-hit/intersection separately? Clearly, inlining all the any-hit and intersection shaders won’t work. So why not just compile them separately? To answer that, I’ll try to start with explaining some assumptions that lie at the base of RADV’s shader compilation. When ACO (and NIR, too) were written, shaders were usually incredibly simple. They had some control flow, ifs, loops and whatnot, but all the code that would ever execute was contained in one compact program executing top-to-bottom. This perfectly matched what graphics/compute shaders looked like in the APIs, and what the API does is what you want to optimize for. Unfortunately, this means RADV’s shader compilation stack got hit extra hard by the paradigm shift introduced by RT pipelines. Dynamic linking of different programs, and calls across the dynamic link boundaries, is something common in CPU programming languages (C/C++, etc.), but Mesa never really had to deal with something like that before2. One specific core assumption that prevents us from compiling any-hit/intersection shaders separately just like that is that every piece of code assumes it has exclusive and complete access to things like registers and other hardware resources. Comparing to CPU again, most of the program code is contained in some functions, and those functions will be called from somewhere else3. Those functions will have used CPU registers and stack memory and so on before, and code inside that function can’t write to just any CPU register, or any location on stack. Which registers are writable by a function and which ones must have their values preserved (so that the function callers can store values of their own there without them being overwritten) are governed by little specifications called “calling conventions”. In Mesa, the shader compiler generally used to have no concept of calling conventions, or a concept of “calling” something, for that matter. There was no concept of a register having some value from a function caller and needing to be preserved - if a register exists, the shader might end up writing its own value to it. In cases of graphics/compute shaders, this wasn’t a problem - the registers only ever had random uninitialized values in them. This has always been a problem for separately compiling shaders in RT pipelines, but we had a different solution: At every point a shader called another shader, we’d split the shader in half: One half containing everything before the call, and the other half containing everything after. Of course, sometimes the second half needed variables coming from the first half of the shader. All these variables would be stored to memory in the first half. Then, the first half ends, and execution jumps to the called shader. Once the end of the called shader is reached, execution returns to the second half. This was good enough for things like calling into traceRay to trace a ray and execute all the associated closest hit/miss shaders. Usually, applications wouldn’t have that many variables needing to be backed up to memory, and tracing a ray is supposed to be expensive. But that concept completely breaks down when you apply it to any-hit shaders. At the point an any-hit shader is called, you're right in the middle of ray traversal. Ray traversal has lots of internal state variables that you really want to keep in registers at all times. If you call an any-hit shader with this approach, you'd have to back up all of these state variables to memory and reload them back afterwards. Any-hit shaders are supposed to be relatively cheap and called potentially lots of times during traversal. All these memory stores and reloads you'd need to insert would completely ruin performance. So, separately compiling any-shaders was an absolute no-go. At least, unless someone were to go off the deep end and change the entire compiler stack to fix the assumptions at their heart. "So, where have you been the last two years?" I went and changed more or less the entire compiler stack to fix these assumptions and introduce proper function calls. The biggest part of this work by far were the absolute basics. How do we best teach the compiler that certain registers need to be preserved and are best left alone? How should the compiler figure out that something like a call instruction might randomly overwrite other registers? How do we represent a calling convention/ABI specification in the driver? All of these problems can be tackled with different approaches and at different stages of compilation, and nailing down a clean solution is pretty important in a rework as fundamental as this one. I started out with applying function calls to the shaders that were already separately compiled - this means that the function call work itself didn't improve performance by too much, but in retrospect I think it was a very good idea to make sure the baseline functionality is rock-solid before moving on to separately-compiling any-hit shaders. Indeed, once I finally got around to adding the code that splits out any-hit/intersection shaders and use function calls for them, things worked nearly out of the box! I opened the associated merge request a bit over two weeks ago and got everything merged within a week. (Of course, I would never have gotten it in that fast without all the reviewers teaming up to get everything in ASAP! Big thank you to Daniel, Rhys and Konstantin) In comparison, I started work on function calls in January of 2024 and got the initial code in a good enough shape to open a merge request in June that year, and the code only got merged on the same day I opened the above merge request, two years after starting the initial drafting (although to be fair, that merge request also had periods of being stalled due to personal reasons). Shader compilation with function calls Function calls makes shader compilation work in arguably a much more straightforward way. For the most part, the shader just gets compiled like any other - there's no fancy splitting or anything going on. If a shader calls another shader, like when executing traceRay, or when calling an any-hit shaders, a call instruction is generated. When the called shader finishes, execution resumes after the call instruction. All the magic happens in ACO, the compiler backend. I've documented the more technical design of how calls and ABIs are represented in a docs article. At first, call instructions in the NIR IR are translated to a p_call "pseudo" instruction. It's not actually a hardware instruction, but serves as a placeholder for the eventual jump to the callee. This instruction also carries information about which specific registers parameters will be stored in, and which registers may be overwritten by the call instruction. ACO's compiler passes have special handling for calls wherever necessary: For example, passes analyzing how many registers are required in all the different parts of the code take special care to take into account that in call instructions, fewer registers may be available to store values in (because all other values are overwritten). ACO also has a spilling pass for moving register values to memory whenever the amount of used registers exceeds the available amount. Another fundamental change is that function calls also introduce a call stack. In CPUs, this is no big deal - you have one stack pointer register, and it points to the stack region that your program uses. However, on GPUs, there isn't just one stack - remember that GPUs are highly parallel, and every thread running on the GPU needs its own stack! Luckily, this sounds worse at first than it actually is. In fact, the hardware already has facilities to help manage stacks. AMD GPUs ever since Vega4 have the concept of "scratch memory" - a memory pool in VRAM where the hardware ensures that each thread has its own private "scratch region". There are special scratch_* memory instructions that load and store from this scratch area. Even though they're also VRAM loads/stores, they don’t take any address, just an offset, and for each thread return the value stored in that thread’s own scratch memory region. In my blog post about RT pipeline being enabled by default I claimed AMD GPUs don’t implement a call stack. This is actually misleading - the scratch memory functionality is all you need to implement a stack yourself. The “stack pointer” here is just the offset you pass to the scratch_* memory instruction. Pushing to the stack increases the stack offset, and popping from it decreases the offset5. Eventually, when it comes to converting a call to hardware instructions, all that is needed is to execute the s_swappc instruction. This instruction automatically writes the address of the next instruction to a register before jumping to the called shader. When the called shader wants to return, it merely needs to jump to the address stored in that register, and execution resumes from right after the call instruction. Finally, any-hit separate compilation was a straightforward task as well - it was merely an issue of defining an ABI that made sure that a ton of registers stay preserved and the caller can stash its values there. In practice, all of the traversal state will be stashed in these preserved registers. No expensive spilling to memory needed, just a quick jump to the any-hit shader and back. Performance considerations If you look at the merge request, the performance benefits seem pretty obvious. Ghostwire Tokyo’s RT passes speed up by more than 2x, and of course pipeline compilation times improved massively. The compilation time difference is quite easy to explain. Generally, compilers will perform a ton of analysis passes on shader code to find everything they can to optimize it to death. However, these analysis passes often require going over the same code more than once, e.g. after gathering more context elsewhere in the shader. This also means that a shader that doubles in size will take more than twice as long to compile. When inlining hundreds or thousands of shaders into one, that also means that shader’s compile time grows by a lot more than just a hundred or a thousand times. Thus, if we reverse things and are suddenly able to stop inlining all the shaders into one, that scaling effect means all the shaders will take less total time to compile than the one big megashader. In practice, all modern games also offload shader compilation to multiple threads. If you can compile the any-hit shaders separately, the game can compile them all in parallel - this just isn’t possible with the single megashader which will always be compiled on a single thread. In the runtime performance department, moving to just having a single call instruction instead of hundreds of shaders in one place means the loop has a much smaller code size. In a loop iteration where you don’t call any any-hit shaders, you would still need to jump over all of the code for those shaders, almost certainly causing instruction cache misses, stalls and so on. Also, forcing any-hit/intersection shaders to be separate also means that any-hit/intersection shaders that consume tons of registers despite nearly never getting called won’t have any negative effects on ray traversal as a whole. ACO has heuristics on where to optimally insert memory stores in case something somewhere needs more registers than available. However, these heuristics may decide to insert memory stores inside the generic traversal loop, even if the problematic register usage only comes from a few rarely-called inlined shaders. These stores in the generic loop would now mean that the whole shader is slowed down in every case. However, separate compilation doesn’t exclusively have advantages, either. In an inlined shader, the compiler is able to use the context surrounding the (now-inlined) shader to optimize the code itself. A separately-compiled shader needs to be able to get called from any imaginable context (as long as it conforms to ABI), and this inhibits optimization. Another consideration is that the jump itself has a small cost (not as big as you’d think, but it does have a cost). RADV currently keeps inlining any-hit shaders as long as you don’t have too many of them, and as long as doing so wouldn’t inhibit the ability to compile the shaders in parallel. About that big UE5 Lumen perf improvement I also openend a merge request that provided massive performance improvements to Lumen’s RT right before the branchpoint. However, these improvements are completely unrelated to function calls. In fact, they’re a tiny bit embarrassing, because all that changed was that RADV doesn’t make the hardware do ridiculously inefficient things anymore. Let’s talk about dispatching RT shaders. The Vulkan API provides a vkCmdTraceRaysKHR command that takes in the number of rays to dispatch for X, Y and Z dimensions.

Usually, compute dispatches are described in terms of how many thread groups to dispatch, but RT is special because one ray corresponds to one thread. So here, we really get the dispatch sizes in threads, not groups. By itself, that's not an issue. In fact, AMD hardware has always been able to specify dispatch dimensions in threads instead of groups. In that case, the hardware takes the job of assembling just enough groups that hold the specified number of threads. The issue here comes from how we describe that group to the hardware. The workgroup size itself is also per-dimension, and the simplest case of 32x1x1 threads (i.e. a 1D workgroup) is actually not always the best. Let's consider a very common ray tracing use case: You might want to trace a ray for each pixel in a 1920x1080 image. That's pretty easy, you just call vkCmdTraceRaysKHR to dispatch 1920 rays in the X dimension and 1080 in the Y dimension. When you dispatch a 32x1x1 workgroup, the coordinates for each thread in a workgroup look like this: thread id | 0 | 1 | 2 | ... | 16 | 17 |...| 63 | coord |(0,0)|(1,0)|(2,0)| ... |(16,0)|(17,0)|...|(63,0)| Or, if you consider how the thread IDs are laid out in the image: ------------------- 0 | 1 | 2 | 3 | .. ------------------- That's a straight line in image space. That's not the best, because it means that the pixels will most likely cover different objects which may have very different trace characteristics. This means divergence during RT will be higher, which can make the overall process slower. Let's look instead what happens when you make the workgroup 2D, with a 8x4 size: thread id | 0 | 1 | 2 | ... | 16 | 17 |...| 63 | coord |(0,0)|(1,0)|(2,0)| ... |(0,2) |(1,2) |...|(7,3) | In image space: ------------------- 0 | 1 | 2 | 3 | .. ------------------ 8 | 9 | 10| 11| .. ------------------ 16| 17| 18| 19| .. ------------------- That's much better. Threads are now arranged in a little square, and these squares are much more likely to all cover the same objects, have similar RT characteristics, etc. This is why RADV used 8x4 workgroups as well. Now let's get to when this breaks down. What if the RT dispatch doesn't actually have 2 dimensions? What if there are 1920 rays in the X dimension, but the Y dimension is just 1? It turns out that the hardware can only run 8 threads in a single wavefront in this case. This is because the rest of the workgroup is out-of-bounds of the dispatch - it has a non-zero Y coordinate, but the size in the Y dimension is only 1, so it would exceed the dispatch bounds. The hardware also can't pull in threads from other workgroups, because one wavefront can only ever execute one workgroup. The end result is that the wave runs with only 8 out of 32 threads active - at 1/4 theoretical performance. For no real reason. I actually had noticed this issue years ago (with UE4, ironically). Back then I worked around it by rearranging the game's dispatch sizes into a 2D one behind its back, and recalculating a 1-dimensional dispatch ID inside the RT shader so the game doesn't notice. That worked just fine… as long as we're actually aware about the dispatch sizes. UE5 doesn't actually use vkCmdTraceRaysKHR. It uses vkCmdTraceRaysIndirectKHR, a variant of the command where the dispatch size is read from GPU memory, not specified on the CPU. This command is really cool and allows for some and nifty GPU-driven rendering setups where you only dispatch as many rays as you're definitely going to trace (as determined by previous GPU commands). This command also rips a giant hole in the approach of rearranging dispatch sizes, because we don't even know the dispatch size before the dispatch is actually executed. That means the super simple workaround I built was never hit, and we had the same embarrassingly inefficient RT performance as a few years ago all over again. Obviously, if UE5 is too smart for your workaround, then the solution is to make an even smarter workaround. The ideal solution would work with a 1D thread ID (so that we don't run into any more issues when there is a 1D dispatch, but if a 2D dispatch is detected, we turn that "line" of 1D IDs into a "square". The whole idea about turning a linear coordinate into a square reminded me a lot of how Z-order curves work. In fact, the GPU arranges things like image data on a Z-order curve by interleaving the address bits from X and Y already, because nearby pixels are often accessed together and it's better if they're close to each other. However, instead of interleaving a X and Y coordinate pair to make a linear memory address, we want the opposite: We have a linear dispatch ID, and we want to recover a 2D coordinate inside a square from it. That's not too hard, you just do the opposite operation: Deinterleave the bits, where every odd/even bit of the dispatch ID forms the X/Y coordinate. As it turned out, you can actually do this entirely from inside the shader with just a few bit twiddling tricks, so this approach work for both indirect and direct (non-indirect) trace commands. With that approach, dispatch IDs and coordinates look something like this: thread id | 0 | 1 | 2 | ... | 16 | 17 |...| 63 | coord |(0,0)|(1,0)|(0,1)| ... |(4,0) |(5,0) |...|(7,3) | In image space: ------------------- 0 | 1 | 4 | 5 | .. ------------------ 2 | 3 | 6 | 7 | .. ------------------ 8 | 9 | 12| 13| .. ------------------- 10| 11| 14| 15| .. ------------------- Not only are the thread IDs now arranged in squares, the squares themselves get recursively subdivided into more squares! I think theoretically this should be a further improvement w.r.t divergence, but I don't think it has resulted in measurable speedup in practice anywhere. The most important thing, though, is that now UE5 RT doesn't run 4x slower than it should. Oops. Bonus content: Function call bug bonanza The second most fun thing about function calls is that you can just jump to literally any program anywhere, provided the program doesn't completely thrash your preserved registers and stack space. The most fun thing about function calls is what happens when the program does just that. I'm going to use this section to scream in the void about two very real function call bugs that were reported after I already merged the MR. This is not an exhaustive list, you can trust I've had much much more fun just like what I'll be presenting here while I was testing and developing function calls. Avowed gets stuck in an infinite loop On the scale of function call bugs, this one was rather tame, even. Having infinite loops isn't the most optimal for hang debugging, but it does mean that you can use a tool like umr to sample which wavefronts are active, and get some register dumps. The program counter will at least point to some instruction in the loop that it's stuck in, and you can get yourself the disassembly of the whole shader to try and figure out what's going in the loop and why the exit conditions aren't met. The loop in Avowed was rather simple: It traced a ray in a loop, and when the loop counter was equal to an exit value, control flow would break out of the loop. The register dumps also immediately highlighted the loop exit counter being random garbage. So far so good. During the traceRay call, the loop exit counter was backed up to the shader's stack. Okay, so it's pretty obvious that the stack got smashed somehow and that corrupted the loop exit counter. What was not obvious, however, was what smashed the stack. Debugging this is generally a bit of an issue - GPUs are far, far away from tools like AddressSanitizer, especially at a compiler level. There are no tools that would help me catch a faulty access at runtime. All I could really do was look at all the shaders in that ray tracing pipeline (luckily that one didn't have too many) and see if they somehow store to wrong stack locations. All shaders in that pipeline were completely fine, though. I checked every single scratch instruction in every shader if the offsets were correct (luckily, the offsets are constants encoded in the disassembly, so this part was trivial). I also verified that the stack pointer was incremented by the correct values - everything was completely fine. No shader was smashing its callers' stack. I found the bug more or less by complete chance. The shader code was indeed completely correct, there were no miscompilations happening. Instead, the "scratch memory" area the HW allocated was smaller than what each thread actually used, because I forgot to multiply by the number of threads in a wavefront in one place. The stack wasn't smashed by the called function, it was smashed by a completely different thread. Whether your stack would get smashed was essentially complete luck, depending on where the HW placed your scratch memory area, other wavefront's scratch, and how those wavefronts' execution was timed relative to yours. I don't think I would ever have been able to deduce this from any debugger output, so I should probably count myself lucky I stumbled upon the fix regardless. Silent Hill 2's reflections sample the sky color Did I talk about Unreal Engine yet? Let's talk about Unreal Engine some more. Silent Hill 2 uses Lumen for its reflection/GI system, and somehow Lumen from UE 5.3 specifically was the only thing that seemed to reproduce this particular bug. In every way the Avowed bug was tolerable to debug, this one was pure suffering. There were no GPU hangs, all shaders ran completely fine. That means using umr and getting a rough idea of where the issue is was off the table from the start. Unfortunately, the RT pipeline was also way too large to analyze - there were a few hundred hit shaders, but there also were seven completely different ray generation shaders. Having little other recourse, I started trying to at least narrow down the ray generation shader that triggered the fault. I used Mesa’s debugging environment variables to dump the SPIR-V of all the shaders the driver encountered, and then used spirv-cross on all of them to turn them into editable GLSL. For each ray generation shader, I’d comment out the imageStore instructions that stored the RT result to some image, recompiled the modified GLSL to SPIR-V, and instructed Mesa to sneakily swap out the original ray-gen SPIR-V with my modified one. Then I re-ran the game to see if anything changed. This indeed led me to find the correct ray generation shader, but the lead turned into a dead end - there was little insight other than that the ray was indeed executing the miss shader. Everything seemed correct so far, and if I hadn’t known these rays didn’t miss about 3 commits ago, I honestly wouldn’t even have suspected anything was wrong at all. The next thing I tried was commenting out random things in ray traversal code. Skipping over all any-hit/intersection shaders yielded no change, and neither did replacing the ray flags/culling masks with known good constants to rule out wrong values being passed as parameters. What did “fix” the result, however, was… commenting out the calls to closest-hit shaders. Now, if closest-hit shaders get called and that makes miss shaders execute somehow, you’d perhaps think we’d be calling the wrong function. Maybe we confuse the shader binding table where we load the addresses of shaders to call from? To verify that assumption, I also disabled calling any and all miss shaders. I zeroed out the addresses in the shader handles to make extra sure there was no possible way that a miss shader could ever get called. To keep things working, I replaced the code that calls miss shaders with the relevant code fragment from UE’s miss shader (essentially inlining the shader myself). Nothing changed from that. That means a closest-hit shader being executed somehow resulted in a ray traversal itself returning a miss, not the wrong function being called. Perhaps the closest-hit shaders corrupt some caller values again? Since the RT pipeline was too big to analyze, I tried to narrow down the suspicious shaders by only disabling specific closest-hit shaders. I also discovered that just making all closest-hit shaders no-ops “fixed” things as well, even if they do get called. Sure enough, at some point I had a specific closest-hit shader where the issue went away once I deleted all code from it/made it a no-op. I even figured out a specific register that, if explicitly preserved, would make the issue go away. The only problem was that this register corresponded to one part of the return value of the closest-hit shader - that is, a register that the shader was supposed to overwrite. From here on out it gets completely nonsensical. I will save you the multiple days of confusion, hair-pulling, desperation and agony over the complete and utter undebuggableness of Lumen’s RT setup and skip to the solution: It turned out the “faulty” closest-hit shader I found was nothing but a red herring. Lumen’s RT consists of 6+ RT dispatches, most of which I haven’t exactly figured out the purpose of, but what I seemed to observe was that the faulty RT dispatch used the results of the previous RT dispatch to make decisions on whether to trace any rays or not. Making the closest-hit shaders a no-op did nothing but disable the subsequent traceRays that actually exhibited the issue. Since these RT dispatches used the same RT pipelines, that meant virtually any avenue I had of debugging this driver-side was completely meaningless. Any hacks inside the shader compiler might actually work around the issue, or just affect a conceptually unrelated dispatch that happens to disable the actually problematic rays. Determining which was the case was nearly impossible, especially in a general case. I never really figured out how to debug this issue. Once again, what saved me was a random epiphany out of the blue. In fact, now that I know what the bug was, I’m convinced I would’ve never found this through a debugger either. The issue turned out to be in an optimization for what’s commonly called tail-calls. If you have a function that calls another function at the very end just before returning, a common optimization is to simply turn that call into a jump, and let the other function return directly to the caller. Imagine ray traversal working a bit like this C code: /* hitT is the t value of the ray at the hit point */ payload closestHit(float hitT); /* tMax is the maximum range of the ray, if there is * no hit with a t <= tMax, the ray misses instead */ payload traversal(float tMax) { do something; if (hit) return closestHit(hitT); // gets replaced with a jmp, closestHit returns directly to traversal's caller } More specifically, the bug was with how preserved parameters and tail-calls interact.

Function callers are generally supposed to assume that preserved parameters do not change their value over the function call. That means it's safe to reuse that register after the call and assuming it still has the value the caller put in. However, in the example above, let's assume closestHit has the same calling convention as traversal. That means closestHit's parameter needs to go into the same register as traversal's parameter, and thus the register gets overwritten. If traversal's caller was assuming that the parameter is preserved, that would mean the value of tMax has just been overwritten with the value of hitT without the caller knowing. If traversal now gets called again from the same place, the value of tMax is not the intended value, but the hitT value from the previous iteration, which is definitely smaller than tMax. Put shortly: If all these conditions are met, a smaller-than-intended tMax could cause rays to miss when they were intended to hit. Once again, I got incredibly lucky and stumbled upon the bug by complete chance. The GPU gods seem to be in good spirits for my endeavours. I pray it stays this way. Footnotes "Shader" in this context really means any program that runs on the GPU. The RT pipeline is also made of shaders, shaders determine where the points and triangles making up each object end up on screen, there are compute shaders for generic computing, and so on... ↩ There actually is another use-case where this becomes relevant on GPU - and that is GPGPU code like CUDA/HIP/OpenCL. CUDA/HIP allow you to write C++ for the GPU in a much more "CPU-like" programming environment (OpenCL uses C), and you run into all the same problems there. This also means all the major GPU vendors had already written their solutions for these problems when raytracing came around. There are OpenCL kernels that end up really really bad if you don't have proper function calls in the compiler (which Rusticl suffers from right now), and the function calls work in RADV/ACO may end up proving useful for those as well. ↩ Even your main function works like that, actually. Unless you have some form of freestanding environment, all your program code works like that. ↩ In RADV, the stack pointer is actually constant across a function, and pushing/popping to/from the stack is implemented by adding another offset to the constant stack pointer in load/store instructions. This allows to make the stack pointer an SGPR instead of a VGPR and simplifies stack accesses that aren't push/pop. ↩ We support raytracing before Vega too. We support function calls on all GPUs, as well, through a little magic in dreaming up a buffer descriptor with specific memory swizzling to achieve the same addressing that scratch_* instructions use on Vega and later. ↩

- Lennart Poettering: Introducing Amutable (2026/01/26 23:00)
  Today, we announce Amutable, our new company. We – @blixtra@hachyderm.io, @brauner@mastodon.social, @davidstrauss@mastodon.social, @rodrigo_rata@mastodon.social, @michaelvogt@mastodon.social, @pothos@fosstodon.org, @zbyszek@fosstodon.org, @daandemeyer@mastodon.social @cyphar@mastodon.social, @jrocha@floss.social and yours truly – are building the next generation of Linux systems, with integrity, determinism, and verification – every step of the way. For more information see → https://amutable.com/blog/introducing-amutable
- Mike Blumenkrantz: Unpopular Opinion (2026/01/23 00:00)
  A Big Day For Graphics Today is a big day for graphics. We got shiny new extensions and a new RM2026 profile, huzzah. VK_EXT_descriptor_heap is huge. I mean in terms of surface area, the sheer girth of the spec, and the number of years it's been under development. Seriously, check out that contributor list. Is it the longest ever? I'm not about to do comparisons, but it might be. So this is a big deal, and everyone is out in the streets (I assume to celebrate such a monumental leap forward), and I'm not. All hats off. Person to person, let's talk. Power Overwhelming It's true that descriptor heap is incredibly powerful. It perfectly exemplifies everything that Vulkan is: low-level, verbose, flexible. vkd3d-proton will make good use of it (eventually), as this more closely relates to the DX12 mechanics it translates. Game engines will finally have something that allows them to footgun as hard as they deserve. This functionality even maps more closely to certain types of hardware, as described by a great

gfxstrand blog post. There is, to my knowledge, just about nothing you can’t do with VK_EXT_descriptor_heap. It’s really, really good, and I’m proud of what the Vulkan WG has accomplished here. But I don’t like it. What Is This Incredibly Hot Take? It’s a risky position; I don’t want anyone’s takeaway to be “Mike shoots down new descriptor extension as worst idea in history”. We’re all smart people, and we can comprehend nuance, like the difference between rb and ab in EGL patch review (protip: if anyone ever gives you an rb, they’re fucking lying because nobody can fully comprehend that code). In short, I don’t expect zink to ever move to descriptor heap. If it does, it’ll be years from now as a result of taking on some other even more amazing extension which depends on heaps. Why is this, I’m sure you ask. Well, there’s a few reasons: Code Complexity Like all things Vulkan, “getting it right” with descriptors meant creating an API so verbose that I could write novels with fewer characters than some of the struct names. Everything is brand new, with no sharing/reuse of any existing code. As anyone who has ever stepped into an unfamiliar bit of code and thought “this is garbage, I should rewrite it all” knows too well, existing code is always the worst code–but it’s also the code that works and is tied into all the other existing code. Pretty soon, attempting to parachute in a new descriptor API becomes rewriting literally everything because it’s all incompatible. Great for those with time and resources to spare, not so great for everyone else. Gone are image views, which is cool and good, except that everything else in Vulkan still uses them, meaning now all image descriptors need an extra pile of code to initialize the new structs which are used only for heaps. Hope none of that was shared between rendering and descriptor use, because now there will be rendering use and descriptor use and they are completely separate. Do I hate image views? Undoubtedly, and I like this direction, but hit me up in a few more years when I can delete them everywhere. Shader interfaces are going to be the source of most pain. Sure, it’s very possible to keep existing shader infrastructure and use the mapping API with its glorious nested structs. But now you have an extra 1000 lines of mapping API structs to juggle on top. Alternatively, you can get AI to rewrite all your shaders to use the new spirv extension and have direct heap access. Performance Descriptor heap maps closer to hardware, which should enable users to get more performant execution by eliminating indirection with direct heap access. This is great. Full stop. …Unless you’re like zink, where the only way to avoid shredding 47 CPUs every time you change descriptors is to use a “sliding” offset for descriptors and update it each draw (i.e., VK_DESCRIPTOR_MAPPING_SOURCE_HEAP_WITH_PUSH_INDEX_EXT). Then you can’t use direct heap access. Which means you’re still indirecting your descriptor access (which has always been the purported perf pain point of 1.0 descriptors and EXT_descriptor_buffer). You do not pass Go, you do not collect $200. All you do is write a ton of new code. Opinionated Development There’s a tremendous piece of exposition outlining the reasons why EXT_descriptor_heap exists in the proposal. None of these items are incorrect. I’ve even contributed to this document. If I were writing an engine from scratch, I would certainly expect to use heaps for portability reasons (i.e., in theory, it should eventually be available on all hardware). But as flexible and powerful as descriptor heap is, there are some annoying cases where it passes the buck to the user. Specifically, I’m talking about management of the sampler heap. 1.0 descriptors and descriptor buffer just handwave away the exact hardware details, but with VK_EXT_descriptor_heap, you are now the captain of your own destiny and also the manager of exactly how the hardware is allocating its samplers. So if you’re on NVIDIA, where you have exactly 4096 available samplers as a hardware limit, you now have to juggle that limit yourself instead of letting the driver handle it for you. This also applies to border colors, which has its own note in the proposal. At an objective, high-view level, it’s awesome to have such fine-grained control over the hardware. Then again, it’s one more thing the driver is no longer managing. I Don’t Have A Better Solution That’s certainly the takeaway here. I’m not saying go back to 1.0 descriptors. Nobody should do that. I’m not saying stick with descriptor buffers either. Descriptor heap has been under development since before I could legally drive, and I’m certainly not smarter than everyone (or anyone, most likely) who worked on it. Maybe this is the best we’ll get. Maybe the future of descriptors really is micromanaging every byte of device memory and material stored within because we haven't read every blog post in existence and don't trust driver developers to make our shit run good. Maybe OpenGL, with its drivers that "just worked" under the hood (with the caveat that you, the developer, can't be an idiot), wasn't what we all wanted. Maybe I was wrong, and we do need like five trillion more blog posts about Vulkan descriptor models. Because releasing a new descriptor extension is definitely how you get more of those blog posts. I'm tired, boss.

- Simon Ser: Status update, January 2026 (2026/01/21 22:00)
  Hi! Last week I've released Goguma v0.9! This new version brings a lot of niceties, see the release notes for more details. New since last month are audio previews implemented by delthas, images for users, channels & networks, and usage hints when typing a command. Jean THOMAS has been hard at work to update the iOS port and publish Goguma on AltStore PAL. It's been a while since I've started a NPotM, but this time I have something new to show you: nagjo is a small IRC bot for Forgejo. It posts messages on activity in Forgejo (issue opened, pull request merged, commits pushed, and so on), and it expands references to issues and pull requests in messages (writing "can you look at #42?" will reply with the issue's title and link). It's very similar to glhf, its GitLab counterpart, but the configuration file enables much more flexible channel routing. I hope that bot can be useful to others too! Up until now, many of my projects have moved to Codeberg from SourceHut, but the issue tracker was still stuck on todo.sr.ht due to a lack of a migration tool. I've hacked together srht2forgejo, a tiny script to create Forgejo issues and comments from a todo.sr.ht archive. It's not perfect since the author is the migration user's instead of the original one, but it's good enough. I've now completely migrated all of my projects to Codeberg! I've added a server implementation and tests to go-smee, a small Go library for a Web push forwarding service. It comes in handy when implementing Web push receivers because it's very simple to set up, I've used it when working on nagjo. I've extended the haproxy PROXY protocol to add a new client certificate TLV to relay the raw client certificate from a TLS terminating reverse proxy to a backend server. My goal is enabling client certificate authentication when the soju IRC bouncer sits behind tlstunnel. I've also sent patches for the kimchi HTTP server and go-proxyproto. Because sending a haproxy patch involved git-send-email, I've noticed I've started hitting a long-standing hydroxide signature bug when sending a message. I wasn't previously impacted by this, but some users were. It took a bit of time to hunt down the root cause (some breaking changes in ProtonMail's crypto library), but now it's fixed. Félix Poisot has added two new color management options to Sway: the color_profile command now has separate gamma22 and srgb transfer functions (some monitors use one, some use the other), and a --device-primaries flag to read color primaries from the EDID (as an alternative to supplying a full ICC profile). With the help of Alexander Orzechowski, we've fixed multiple wlroots issues regarding toplevel capture (aka. window capture) when the toplevel is completely hidden. It should all work fine now, except one last bug which results in a frozen capture if you're unlucky (aka. you loose the race). I've shipped a number of drmdb improvements. Plane color pipelines are now supported and printed on the snapshot tree and properties table. A warning icon is displayed next to properties which have only been observed on tainted or unstable kernels (as is usually the case for proprietary or vendor kernel modules with custom properties). The device list now shows vendor names for platform devices (extracted from the kernel table). Devices using the new "faux" bus (e.g. vkms) are now properly handled, and all of the possible cursor sizes advertised via the SIZE_HINTS property are now printed. I've also done some SQLite experiments, however they turned out unsuccessful (see that thread and the merge request for more details). delthas has added a new allow_proxy_ip directive to the kimchi HTTP server to mark IP addresses as trusted proxies, and has made it so Forwarded/X-Forwarded-For header fields are not overwritten when the previous hop is a trusted proxy. That way, kimchi can be used in more scenario: behind another HTTP reverse proxy, and behind a TCP proxy which doesn't have a loopback IP address (e.g. tlstunnel in Docker). See you next month!

- Christian Schaller: Can AI help ‘fix’ the patent system? (2026/01/21 18:35)
  So one thing I think anyone involved with software development for the last decades can see is the problem of “forest of bogus patents”. I have recently been trying to use AI to look at patents in various ways. So one idea I had was “could AI help improve the quality of patents and free us from obvious ones?” Lets start with the justification for patents existing at all. The most common argument for the patent system I hear is this one : “Patents require public disclosure of inventions in exchange for protection. Without patents, inventors would keep innovations as trade secrets, slowing overall technological progress.”. This reasoning is something that makes sense to me, but it is also screamingly obvious to me that for it to hold true you need to ensure the patents granted are genuinely inventions that otherwise would stay hidden as trade secrets. If you allow patents on things that are obvious to someone skilled in the art, you are not enhancing technological progress, you are hampering it because the next person along will be blocking from doing it. So based on this justification the question then becomes does for example the US Patents Office do a good job filtering out obvious patents? And I believe the answer is “No they don’t”. Having been working in the space of software for many decades now it is very clear to me that the patent office do very little to avoid patents getting approved for obvious things. And there are many reasons for why that happens I think. First of all if you are an expert in a field you would likely not be working as a case worker in the patent office, no disrespect for the people working there, but for instance if you are a really good software engineer there are much better paying options to be found elsewhere, same goes for medical researchers for example. So we have people who have limited understanding of the field of the patent trying to evaluate if the patent is obvious or not. Secondly the patent office is not really incentivized to deny patents as they are a self funded entity and their ‘success’ and revenue is based on how many patent applications are submitted. So the lower the bar of approval, the more patents applications get submitted and thus the patent office makes more money and is more “successful”. This is a problem, but maybe if we fix the first issue I mention it is maybe possible to reduce the impact of this second issue. So what can AI do to help here? It can help patent reviewers at the patent office with evaluating patents for obviousness due to having been trained on these fields. If we required every patent applied for to be run through a AI engine for review then the patent office reviewer would have to justify why he or she believes something is non-obvious after the AI has looked at it and concluded it is not. My assumption would be that they would very rarely do so. To test out this I decided to ask Gemini to review a H264 video patent that is still valid and which has been used in litigation before, to see if it could do a decent job of reviewing the patent application and judge if this is a true invention or not. Here is what Gemini came up with. US Patent 7,916,782 (Samsung Electronics) is one of the most famous “battleground” patents in the history of smartphone litigation. It has been asserted against Apple, Huawei, and others. It covers a specific part of the H.264 standard known as CAVLC (Context-Adaptive Variable Length Coding). The Patent: US 7,916,782 : Method and apparatus for video coding and decoding The Mechanism: Efficiently coding the count of zero-valued pixels in a video block (“TotalZeros”). The Problem In video compression, you divide the image into small 4×4 pixel blocks. After processing (transform and quantization), most of the 16 pixels in the block end up being zero. You only have a few non-zero numbers left. Goal: You need to tell the decoder how many zeros are in the block so it can reconstruct the empty space. Constraint: You already told the decoder how many non-zero coefficients exist (let’s say you have 5 non-zeros). The “Invention” The patent claims a method where the encoder selects a specific lookup table (VLC table) to encode the “TotalZeros” value, and—crucially—this selection is based on the number of non-zero coefficients (TotalCoeff) that were just processed. If TotalCoeff is 1, use Table A (allows for many zeros).If TotalCoeff is 10, use Table B (allows for fewer zeros). The “Skilled Engineer” Test Imagine you are a video coding engineer in 2002. You are tasked with compressing the “TotalZeros” value. Here is the logical deduction chain you would inevitably follow: The Constraint Check (Basic Math) Engineer’s Thought: “I am working with a 4×4 block.

That's 16 pixels total." Engineer's Thought: "I just transmitted the number of non-zero pixels. Let's say there are 5 non-zeros. Deduction: "Therefore, the maximum number of zeros possible is 16−5=11." The Optimization (Common Sense) Engineer's Thought: "I need a Huffman table (lookup table) to compress the TotalZeros value." Engineer's Thought: "Should I use a generic table that goes from 0 to 16?" Deduction: "No, that's wasteful. I know for a fact the value cannot be 12, 13, 14, 15, or 16. If I use a table that includes those impossible numbers, I'm wasting bits on shorter codes for values that will never happen." The Solution (The "Invention") Engineer's Thought: "I should use a smaller, custom table that only contains values 0 through 11." Engineer's Thought: "Actually, I should just make a different table for every possible number of non-zeros. If I have 15 non-zeros, the only possible zero count is 1. I don't even need a table for that; it's implicit!" The Verdict: Valid or Broken? Engineer's View (Obvious): This is not an invention; it is constraint propagation. If you have a container of capacity 16, and you fill it with 5 rocks, it is "obvious" that the remaining space for air is 11. Creating a table sized to "11" isn't a creative spark; it is the mathematical consequence of the block size. To an engineer, failing to do this would be incompetence, not a design choice. Patent Office View (Novel): The USPTO granted this because the patent didn't just claim "optimizing tables." It claimed the specific structure of the solution: Check TotalCoeff.If TotalCoeff is 1, select Table_1. If TotalCoeff is 2, select Table_2. …and so on. Because no prior paper had explicitly written down "Use these specific tables in this specific order for H.264," it was deemed novel. Conclusion US 7,916,782 is a prime example of the "broken" system because it effectively patents arithmetic efficiency. By granting a monopoly on the logic "Use a small table when you have small space," the patent office allowed Samsung to assert ownership over the only logical way to implement the H.264 standard efficiently. An engineer trying to solve this problem without seeing the patent would essentially be forced to "re-invent" it, because the math leaves no other optimal path. Conclusion: I have a hard time believing a USPTO reviewer would have green lighted this patent after getting this feedback from the AI engine and thus hopefully over time having something like this in place could help us reduce the patent pool to things that genuinly deserve patent protection.

- Sebastian Wick: Best Practices for Ownership in GLib (2026/01/21 15:31)
  For all the rightful criticisms that C gets, GLib does manage to alleviate at least some of it. If we can't use a better language, we should at least make use of all the tools we have in C with GLib. This post looks at the topic of ownership, and also how it applies to libdex fibers. Ownership In normal C usage, it is often not obvious at all if an object that gets returned from a function (either as a real return value or as an out-parameter) is owned by the caller or the callee: MyThing *thing = my_thing_new (); If thing is owned by the caller, then the caller also has to release the object thing. If it is owned by the callee, then the lifetime of the object thing has to be checked against its usage. At this point, the documentation is usually being consulted with the hope that the developer of my_thing_new documented it somehow. With gobject-introspection, this documentation is standardized and you can usually read one of these: The caller of the function takes ownership of the data, and is responsible for freeing it. The returned data is owned by the instance. If thing is owned by the caller, the caller now has to release the object or transfer ownership to another place. In normal C usage, both of those are hard issues. For releasing the object, one of two techniques are usually employed: single exit MyThing *thing = my_thing_new (); gboolean c; c = my_thing_a (thing); if (c) c = my_thing_b (thing); if (c) my_thing_c (thing); my_thing_release (thing); /* release thing */ goto cleanup MyThing *thing = my_thing_new (); if (!my_thing_a (thing)) goto out; if (!my_thing_b (thing)) goto out; my_thing_c (thing); out: my_thing_release (thing); /* release thing */ Ownership Transfer GLib provides automatic cleanup helpers (g_auto, g_autoptr, g_autofd, g_autolist). A macro associates the function to release the object with the type of the object (e.g. G_DEFINE_AUTOPTR_CLEANUP_FUNC). If they are being used, the single exit and goto cleanup approaches become unnecessary:

g_autoptr(MyThing) thing = my_thing_new (); if (!my_thing_a (thing)) return; if (!my_thing_b (thing)) return; my_thing_c (thing); The nice side effect of using automatic cleanup is that for a reader of the code, the g_auto helpers become a definite mark that the variable they are applied on own the object! If we have a function which takes ownership over an object passed in (i.e. the called function will eventually release the resource itself) then in normal C usage this is indistinguishable from a function call which does not take ownership: MyThing *thing = my_thing_new (); my_thing_finish_thing (thing); If my_thing_finish_thing takes ownership, then the code is correct, otherwise it leaks the object thing. On the other hand, if automatic cleanup is used, there is only one correct way to handle either case. A function call which does not take ownership is just a normal function call and the variable thing is not modified, so it keeps ownership: g_autoptr(MyThing) thing = my_thing_new (); my_thing_finish_thing (thing); A function call which takes ownership on the other hand has to unset the variable thing to remove ownership from the variable and ensure the cleanup function is not called. This is done by “stealing” the object from the variable: g_autoptr(MyThing) thing = my_thing_new (); my_thing_finish_thing (g_steal_pointer (&thing)); By using g_steal_pointer and friends, the ownership transfer becomes obvious in the code, just like ownership of an object by a variable becomes obvious with g_autoptr. Ownership Annotations Now you could argue that the g_autoptr and g_steal_pointer combination without any conditional early exit is functionally exactly the same as the example with the normal C usage, and you would be right. We also need more code and it adds a tiny bit of runtime overhead. I would still argue that it helps readers of the code immensely which makes it an acceptable trade-off in almost all situations. As long as you haven’t profiled and determined the overhead to be problematic, you should always use g_auto and g_steal! The way I like to look at g_auto and g_steal is that it is not only a mechanism to release objects and unset variables, but also annotations about the ownership and ownership transfers. Scoping One pattern that is still somewhat pronounced in older code using GLib, is the declaration of all variables at the top of a function: static void foobar (void) { MyThing *thing = NULL; size_t i; for (i = 0; i < len; i++) { g_clear_pointer (&thing); thing = my_thing_new (i); my_thing_bar (thing); } } We can still avoid mixing declarations and code, but we don’t have to do it at the granularity of a function, but of natural scopes: static void foobar (void) { for (size_t i = 0; i < len; i++) { g_autoptr(MyThing) thing = NULL; thing = my_thing_new (i); my_thing_bar (thing); } } Similarly, we can introduce our own scopes which can be used to limit how long variables, and thus objects are alive: static void foobar (void) { g_autoptr(MyOtherThing) other = NULL; { /* we only need `thing` to get `other` */ g_autoptr(MyThing) thing = NULL; thing = my_thing_new (); other = my_thing_bar (thing); } my_other_thing_bar (other); } Fibers When somewhat complex asynchronous patterns are required in a piece of GLib software, it becomes extremely advantageous to use libdex and the system of fibers it provides. They allow writing what looks like synchronous code, which suspends on await points: g_autoptr(MyThing) thing = NULL; thing = dex_await_object (my_thing_new_future (), NULL); If this piece of code doesn’t make much sense to you, I suggest reading the libdex Additional Documentation. Unfortunately the await points can also be a bit of a pitfall: the call to dex_await is semantically like calling g_main_loop_run on the thread default main context. If you use an object which is not owned across an await point, the lifetime of that object becomes critical. Often the lifetime is bound to another object which you might not control in that particular function. In that case, the pointer can point to an already released object when dex_await returns: static DexFuture * foobar (gpointer user_data) { /* foo is owned by the context, so we do not use an autoptr */ MyFoo *foo = context_get_foo (); g_autoptr(MyOtherThing) other = NULL; g_autoptr(MyThing) thing = NULL; thing = my_thing_new (); /* side effect of running g_main_loop_run */ other = dex_await_object (my_thing_bar (thing, foo), NULL); if (!other) return dex_future_new_false (); /* foo here is not owned, and depending on the lifetime * (context might recreate foo in some circumstances), * foo might point to an already released object */ dex_await (my_other_thing_foo_bar (other, foo), NULL); return dex_future_new_true (); } If we assume that context_get_foo returns a different object when the main loop runs, the code above will not work. The fix is simple: own the objects that are being used across await points, or re-acquire an object. The correct choice depends on what semantic is required. We can also combine this with improved scoping to only keep the objects alive for as long as required. Unnecessarily keeping objects alive across await points can keep resource usage high and might have unintended consequences. static DexFuture * foobar (gpointer user_data) { /* we now own foo */ g_autoptr(MyFoo) foo = g_object_ref (context_get_foo ()); g_autoptr(MyOtherThing) other = NULL; { g_autoptr(MyThing) thing = NULL; thing = my_thing_new (); /* side effect of running g_main_loop_run */ other = dex_await_object (my_thing_bar (thing, foo), NULL); if (!other) return dex_future_new_false (); } /* we own foo, so this always points to a valid object */ dex_await (my_other_thing_bar (other, foo), NULL); return dex_future_new_true (); } static DexFuture * foobar (gpointer user_data) { /* we now own foo */ g_autoptr(MyOtherThing) other = NULL; { /* We do not own foo, but we only use it before an * await point. * The scope ensures it is not being used afterwards. */ MyFoo *foo = context_get_foo (); g_autoptr(MyThing) thing = NULL; thing = my_thing_new (); /* side effect of running g_main_loop_run */ other = dex_await_object (my_thing_bar (thing, foo), NULL); if (!other) return dex_future_new_false (); } { MyFoo *foo = context_get_foo (); dex_await (my_other_thing_bar (other, foo), NULL); } return dex_future_new_true (); } One of the scenarios where re-acquiring an object is necessary, are worker fibers which operate continuously, until the object gets disposed. Now, if this fiber owns the object (i.e. holds a reference to the object), it will never get disposed because the fiber would only finish when the reference it holds gets released, which doesn’t happen because it holds a reference. The naive code also suspiciously doesn’t have any exit condition. static DexFuture * foobar (gpointer user_data) { g_autoptr(MyThing) self = g_object_ref (MY_THING (user_data)); for (;;) { g_autoptr(GBytes) bytes = NULL; bytes = dex_await_boxed (my_other_thing_bar (other, foo), NULL); my_thing_write_bytes (self, bytes); } } So instead of owning the object, we need a way to re-acquire it. A weak-ref is perfect for this. static DexFuture * foobar (gpointer user_data) { /* g_weak_ref_init in the caller somewhere */ GWeakRef *self_wr = user_data; for (;;) { g_autoptr(GBytes) bytes = NULL; bytes = dex_await_boxed (my_other_thing_bar (other, foo), NULL); { g_autoptr(MyThing) self = g_weak_ref_get (&self_wr); if (!self) return dex_future_new_true (); my_thing_write_bytes (self, bytes); } } } Conclusion Always use g_auto/g_steal helpers to mark ownership and ownership transfers (exceptions do apply) Use scopes to limit the lifetime of objects In fibers, always own objects you need across await points, or re-acquire them

- Mike Blumenkrantz: 2026 Status (2026/01/14 00:00)
  Not A Real Post Still digging myself out of a backlog (and remembering how to computer), so probably no real post this week. I do have some exciting news for the blog though. Now that various public announcements have been made, I can finally reveal the reason why I’ve been less active in Mesa of late is because I’ve been hard at work on Steam Frame. There’s a lot of very cool tech involved, and I’m planning to do some rundowns on the software-related projects I’ve been tackling. Temper your expectations: I won’t be discussing anything hardware-related, and there will likely be no mentions of any specific game performance/issues.
- Sebastian Wick: Improving the Flatpak Graphics Drivers Situation (2026/01/05 23:30)
  Graphics drivers in Flatpak have been a bit of a pain point. The drivers have to be built against the runtime to work in the runtime. This usually isn’t much of an issue but it breaks down in two cases: If the driver depends on a specific kernel version If the runtime is end-of-life (EOL) The first issue is what the proprietary Nvidia drivers exhibit. A specific user space driver requires a specific kernel driver. For drivers in Mesa, this isn’t an issue. In the medium term, we might get lucky here and the Mesa-provided Nova driver might become competitive with the proprietary driver. Not all hardware will be supported though, and some people might need CUDA or other proprietary features, so this problem likely won’t go away completely. Currently we have runtime extensions for every Nvidia driver version which gets matched up with the kernel version, but this isn’t

great. The second issue is even worse, because we don’t even have a somewhat working solution to it. A runtime which is EOL doesn’t receive updates, and neither does the runtime extension providing GL and Vulkan drivers. New GPU hardware just won’t be supported and the software rendering fallback will kick in. How we deal with this is rather primitive: keep updating apps, don’t depend on EOL runtimes. This is in general a good strategy. A EOL runtime also doesn’t receive security updates, so users should not use them. Users will be users though and if they have a goal which involves running an app which uses an EOL runtime, that’s what they will do. From a software archival perspective, it is also desirable to keep things working, even if they should be strongly discouraged. In all those cases, the user most likely still has a working graphics driver, just not in the flatpak runtime, but on the host system. So one naturally asks oneself: why not just use that driver? That’s a load-bearing “just”. Let’s explore our options. Exploration Attempt #1: Bind mount the drivers into the runtime. Cool, we got the driver’s shared libraries and ICDs from the host in the runtime. If we run a program, it might work. It might also not work. The shared libraries have dependencies and because we are in a completely different runtime than the host, they most likely will be mismatched. Yikes. Attempt #2: Bind mount the dependencies. We got all the dependencies of the driver in the runtime. They are satisfied and the driver will work. But your app most likely won’t. It has dependencies that we just changed under its nose. Yikes. Attempt #3: Linker magic. Until here everything is pretty obvious, but it turns out that linkers are actually quite capable and support what’s called linker namespaces. In a single process one can load two completely different sets of shared libraries which will not interfere with each other. We can bind mount the host shared libraries into the runtime, and dlmopen the driver into its own namespace. This is exactly what libcapsule does. It does have some issues though, one being that the libc can’t be loaded into multiple linker namespaces because it manages global resources. We can use the runtime’s libc, but the host driver might require a newer libc. We can use the host libc, but now we contaminate the apps linker namespace with a dependency from the host. Attempt #4: Virtualization. All of the previous attempts try to load the host shared objects into the app. Besides the issues mentioned above, this has a few more fundamental issues: The Flatpak runtimes support i386 apps; those would require a i386 driver on the host, but modern systems only ship amd64 code. We might want to support emulation of other architectures later It leaks an awful lot of the host system into the sandbox It breaks the strict separation of the host system and the runtime If we avoid getting code from the host into the runtime, all of those issues just go away, and GPU virtualization via Virtio-GPU with Venus allows us to do exactly that. The VM uses the Venus driver to record and serialize the Vulkan commands, sends them to the hypervisor via the virtio-gpu kernel driver. The host uses virglrenderer to deserializes and executes the commands. This makes sense for VMs, but we don’t have a VM, and we might not have the virtio-gpu kernel module, and we might not be able to load it without privileges. Not great. It turns out however that the developers of virglrenderer also don’t want to have to run a VM to run and test their project and thus added vtest, which uses a unix socket to transport the commands from the mesa Venus driver to virglrenderer. It also turns out that I’m not the first one who noticed this, and there is some glue code which allows Podman to make use of virgl. You can most likely test this approach right now on your system by running two commands: rendernodes=(/dev/dri/render*) virgl_test_server --venus --use-gles --socket-path /tmp/flatpak-virgl.sock --rendernode "${rendernodes[0]}" & flatpak run --nodevice=dri --filesystem=/tmp/flatpak-virgl.sock --env=VN_DEBUG=vtest --env=VTEST_SOCKET_NAME=/tmp/flatpak-virgl.sock org.gnome.clocks If we integrate this well, the existing driver selection will ensure that this virtualization path is only used if there isn’t a suitable driver in the runtime. Implementation Obviously the commands above are a hack. Flatpak should automatically do all of this, based on the availability of the dri permission. We actually already start a host program and stop it when the app exits: xdg-dbus-proxy. It’s a bit involved because we have to wait for the program (in our case virgl_test_server) to provide the service before starting the app. We also have to shut it down when the app exits, but flatpak is not a supervisor.

You won't see it in the output of ps because it just execs bubblewrap (bwrap) and ceases to exist before the app even started. So instead we have to use the kernel's automatic cleanup of kernel resources to signal to virgl_test_server that it is time to shut down. The way this is usually done is via a so called sync fd. If you have a pipe and poll the file descriptor of one end, it becomes readable as soon as the other end writes to it, or the file description is closed. Bubblewrap supports this kind of sync fd: you can hand in a one end of a pipe and it ensures the kernel will close the fd once the app exits. One small problem: only one of those sync fds is supported in bwrap at the moment, but we can add support for multiple in Bubblewrap and Flatpak. For waiting for the service to start, we can reuse the same pipe, but write to the other end in the service, and wait for the fd to become readable in Flatpak, before exec'ing bwrap with the same fd. Also not too much code. Finally, virglrenderer needs to learn how to use a sync fd. Also pretty trivial. There is an older MR which adds something similar for the Podman hook, but it misses the code which allows Flatpak to wait for the service to come up, and it never got merged. Overall, this is pretty straight forward. Conclusion The virtualization approach should be a robust fallback for all the cases where we don't get a working GPU driver in the Flatpak runtime, but there are a bunch of issues and unknowns as well. It is not entirely clear how forwards and backwards compatible vtest is, if it even is supposed to be used in production, and if it provides a strong security boundary. None of that is a fundamental issue though and we could work out those issues. It's also not optimal to start virgl_test_server for every Flatpak app instance. Given that we're trying to move away from blanket dri access to a more granular and dynamic access to GPU hardware via a new daemon, it might make sense to use this new daemon to start the virgl_test_server on demand and only for allowed devices.

- Timur Kristóf: A love song for Linux gamers with old GPUs (EOY 2025) (2026/01/01 00:00)
AMD GPUs are famous for working very well on Linux. However, what about the very first GCN GPUs? Are they working as well as the new ones? In this post, I'm going to summarize how well these old GPUs are supported and what I've been doing to improve them. This story is about the first two generations of GCN: Southern Islands (aka. SI, GCN1, GFX6) and Sea Islands (aka. CIK, GCN2, GFX7). Working on old GPUs While AMD GPUs generally have a good reputation on Linux, these old GCN graphics cards have been a sore spot for as long as I've been working on the driver stack. It occurred to me that resolving some of the long-standing issues on these old GPUs might be a great way to get me started on working on the amdgpu kernel driver and would help improve the default user experience of Linux users on these GPUs. I figured that it would give me a good base understanding, and later I could also start contributing code and bug fixes to newer GPUs. Where I started The RADV team has supported RADV on SI and CIK GPUs for a long time. RADV support was already there even before I joined the team in mid-2019. Daniel added ACO support for GFX7 in November 2019, and Samuel added ACO support for GFX6 in January 2020. More recently, Martin added a Tahiti (GFX6) and Hawaii (GFX7) GPU to the Mesa CI which are running post-merge "nightly" jobs. So we can catch regressions and test our work on these GPUs quite quickly. The kernel driver situation was less fortunate. On the kernel side, amdgpu (the newer kernel driver) has supported CIK since June 2015 and SI since August 2016. DC (the new display driver) has supported CIK since September 2017 (the beginning), and SI support was added in July 2020 by Mauro. However, the old radeon driver was the default driver. Unfortunately, radeon doesn't support Vulkan, so in the default user experience, users couldn't play most games or benefit from any of the Linux gaming related work we've been doing for the last 10 years. In order to get working Vulkan support on SI and CIK, we needed to use the following kernel params: radeon.si_support=0 radeon.cik_support=0 amdgpu.si_support=1 amdgpu.cik_support=1 Then, you could boot with amdgpu and enjoy a semblance of a good user experience until the GPU crashed / hung, or until you tried to use some functionality which was missing from amdgpu, or until you plugged in a display which the display driver couldn't handle. It was… not the best user experience. Where to go from there? The first question that came to

mind is, why wasn’t amdgpu the default kernel driver for these GPUs? Since the “experimental” support had been there for 10 years, we had thought the kernel devs would eventually just enable amdgpu by default, but that never happened. At XDC 2024 I met Alex Deucher, the lead developer of amdgpu and asked him what was missing. Alex explained to me that the main reason the default wasn’t switched is to avoid regressions for users who rely on some functionality not supported by amdgpu: Display features: analog connectors in DC (or DP/HDMI audio support in non-DC) VCE1 for video coding on SI It doesn’t seem like much, does it? How hard can it be? Display features On a 2025 summer afternoon… I messaged Alex Deucher to get some advice on where to start. Alex was very considerate and helped me to get a good understanding of how the code is organized, how the parts fit together and where I should start reading. Harry Wentland also helped a lot with making a plan how to fit analog connectors in DC. Then, I plugged my monitors into my Raphael iGPU to be used as a primary GPU, then plugged in an old Oland card as a secondary GPU, and started hacking. Focus For the display, I decided that the best way forward is to add what is missing from DC for these GPUs and use DC by default. That way, we can eventually get rid of the legacy display code (which was always meant as a temporary solution until DC landed). Additionally, I decided to focus on dedicated GPUs because these are the most useful for gaming and are easy to test using a desktop computer. There is still work left to do for CIK APUs. Analog connector support in DC Analog connectors have been actually quite easy to deal with, once I understood the structure of the DC (display core) codebase. I could use the legacy display code as a reference. The DAC (digital to analog converter) is actually programmed by the VBIOS, and the driver just needs to call the VBIOS to tell it what to do. Easier said than done, but not too hard. It also turned out that some chips that already defaulted to DC (eg. Tonga, Hawaii) also have analog connectors, which apparently just didn’t work on Linux by default. I managed to submit the first version of this in July. Then I was sidetracked with a lot of other issues, so I submitted the second version of the series in September, which then got merged. Going shopping It is incredibly difficult to debug issues when you don’t have the hardware to reproduce them yourself. Some developers have a good talent for writing patches to fix issues without actually seeing the issue, but I feel I still have a long way to go to be that good. It was pretty clear from the beginning that the only way to make sure my work actually works on all SI/CIK GPUs is to test all of them myself. So, I went ahead an acquired at least one of each SI and CIK chip. I got most of them from used hardware ad sites, and Leonardo Frassetto sent me a few as well. Fixing DC support on SI (DCE6) After I got the analog connector working using the old GPUs as a secondary GPU, I thought it’s time to test how well it works as a primary GPU. You know, the way most actual users would use them. So I disabled the iGPU and booted my computer with each dGPU with amdgpu.dc=1 to see what happens. This is where things started going crazy… Tahiti (R9 280X) booted into “no signal”, absolutely dead Oland (R7 250) had heavy flickering and hung very quickly Oland (Radeon 520) booted into “unsupported signal” with DC and massive flickering with non-DC Pitcairn (R9 270X) had some artifacts Cape Verde (HD 7770) I didn’t even plug it in at this point… Hainan fortunately doesn’t have DCE (display controller engine) so that wasn’t a problem The way to debug these problems is the following: Boot with amdgpu.dc=0, dump all DCE registers using umr: umr -r oland.dce600..* > oland_nodc_good.txt Boot with amdgpu.dc=1, dump all DCE registers using umr: umr -r oland.dce600..* > oland_dc_bad.txt Compare the two DCE register dumps using a diff viewer, eg. Meld. Try to find what are the register differences that are responsible for the bad behaviour. Use umr (either the GUI or CLI) to try to change the registers in real time, poke at it until the issue is gone. Wait until headache is gone. Read the code that sets the problematic registers and develop an actual fix. I decided to fix the bugs before adding new features. I sent a few patch series to address a bunch of display issues mainly with SI (DCE6): Fixed broken PLL programming and some mistakes Fixed DC “overclocking” the display clock and a few other issues — many years ago someone fixed an issue on Polaris by unconditionally raising the display clock by 15%, but unfortunately they also applied this to older GPUs; additionally the display clock was already set higher than the max Fixed DVI-D/HDMI adapters — these would just give a black screen when I plugged in a 4K monitor While at it, I also fixed them in the legacy code Fixed a freeze caused by relying on a non-existent interrupt — it seems that DCE6 is not capable of VRR, so we just shouldn't try to enable it or rely on interrupts that don't exist on this HW Fixed another black screen by rejecting too high pixel clocks — technically, DP supports the bandwidth required by 4K 120Hz using 6-bit color with YUV420 on SI/CIK/VI, so DC would happily advertise this mode, but the GPUs didn't actually support a high enough display clock for 4K 120Hz Fixed an issue with the DCE6 scaler not being properly disabled — apparently the VBIOS sets up the scaler automatically on some GPUs, which needs to be disabled by the kernel, otherwise you get weird artifacts DisplayPort/HDMI audio support on SI (DCE6) I noticed that HDMI audio worked alright on all GPUs with DC (as expected), however DP audio didn't (which was unexpected). However it worked when both DP and HDMI were plugged in… After consulting with Alex and doing some trial and error, it turned out that this was just due to setting some clock frequency in the wrong way: Fix DP audio DTO1 clock source on DCE6. In order to figure out the correct frequencies, I wrote a script that set the frequency using umr then played a sound. I just laid back and let the script run until I heard the sound. Then it was just a matter of figuring out why that frequency was correct. A small fun fact: it turns out that DP audio on Tahiti didn't work on any Linux driver before. Now it works with DC. Poweeeeer The DCE (Display Controller Engine), just like other parts of the GPU, has its own power requirements and needs certain clocks, voltages, etc. It is the responsibility of the power management code to make sure DCE gets the power it needs. Unfortunately, DC didn't talk to the legacy power management code. Even more unfortunately, the power management code was buggy so that's what I started with. SI power management fixes — contains some fixes to get Tahiti to boot with DC, also it turns out that the SMC (system management controller) needs a longer timeout to complete some tasks More SI power management and PLL fixes — most importantly this disables ASPM on SI, which caused "random hangs" Finally, this series hooks up SI to the legacy DPM code — this is required because the power management code needs to know how many and what kind of displays are connected After I was mostly done with SI, I also fixed an issue with CIK, where the shutdown temperature was incorrectly reported. VCE1 video encoding on SI Video encoding is usually an afterthought, not something that most users think about unless they are interested in streaming or video transcoding. It was definitely an afterthought for the hardware designers of SI, which has the first generation VCE (video coding engine) that only supports H264 and only up to 2048 x 1152. However, the old radeon kernel driver supports this engine and for anyone relying on this functionality, it would be a regression when switching to amdgpu. So we need to support it. There was already some work by Alexandre Demers to support VCE1, but that work was stalled due to issues caused by the firmware validation mechanism. In order to switch SI to amdgpu by default, I needed to deal with VCE1. So I started a conversation with Christian König (amdgpu expert) to identify what the problem actually was, and with Alexandre to see how far along his work was. It turns out that due to some HW/FW limitations, the firmware (VCPU BO) needs to be mapped at a low 32-bit address. It also needs to be in VRAM for optimal performance (otherwise it would incur too many roundtrips to system RAM). However, there was no way for amdgpu to place something in VRAM and map it in the low 32-bit address space. (Note, it can actually do this for the address space of userspace apps, just not inside the kernel itself.) Christian helped me a lot with understanding how the memory controller and the page table work. After I got over the headache, I came up with this idea: Let amdgpu place the VCPU BO in VRAM Map the GART (graphics address remapping table) in the low 32-bit address space (instead of using best fit) Insert a few page table entries in the GART which would practically map the VCPU BO into the low 32-bit address space With that out of the way, the rest of the work on VCE1 was pretty straightforward. I could use Alexandre's research, as well as the VCE2 code from amdgpu and the VCE1 code from radeon as a reference. Finally, a few reviews and three revisions later, the VCE1 series was accepted. Final thoughts Who is this for? In the current economic situation of our world, I expect that people are going to use GPUs for much longer, and replace them less often. And when an old GPU is replaced, it doesn't die, it goes to somebody who upgrades an even older GPU. Eventually it will reach somebody that can't afford a better one. There are some efforts to use Linux to keep old computers alive, for example this one. My goal with this work is to make Linux gaming a good experience also for those who use old GPUs. Other than that, I also did it for myself. Not because I want to run old GPUs myself, but because it has been a great learning experience to get into the amdgpu kernel driver. Why amdgpu? Why DC? The open source community including AMD themselves as well as other entities like Valve, Igalia, Red Hat etc. have invested a lot of time and effort into amdgpu and DC, which now support many generations of AMD GPUs: GCN1-5, RDNA1-4, as well as CDNA. In fact amdgpu supports more generations of GPUs now than what came before, and it looks like it will support many generations of future GPUs. By making amdgpu work well with SI and CIK, we ensure that these GPUs remain competently supported for the foreseeable future. By switching SI and CIK to use DC by default, we enable display features like atomic modesetting, VRR, HDR, etc. and this also allows the amdgpu maintainers to eventually move on from the legacy display code without losing functionality. What is left to do? Now that amdgpu is at feature parity with radeon on old GPUs, we switched the default to amdgpu on SI and CIK dedicated GPUs. It's time to start thinking about what else is left to do. Add support for DRM format modifiers for all SI/CIK/VI/Polaris GPUs. This would be a huge step forward for the Vulkan ecosystem, it would enable using purely Vulkan based compositors, Zink, and other features. Add support for TRAVIS and NUTMEG display bridges, so that we can also switch to amdgpu by default for CIK APUs. I couldn't find the hardware for this work (mainly looking for Kaveri APUs), if you have it and want to help, please reach out. Your dmesg log will mention if the APU uses TRAVIS or NUTMEG. Refactor SI and KV power management so that we can retire the legacy power management code, which would further ease the maintenance burden of these GPUs. Eventually retire the non-DC legacy display code from amdgpu to ease the maintenance burden. Deal with a few lingering bugs, such as power limit on Radeon 430, black screen with the analog connector on Radeon HD 7790, as well as reenable compute queues on SI, mitigate VM faults on SI/CIK, etc. Verify sparse mapping (PRT) support. I already wrote a kernel fix and a Mesa MR for enabling it. Implement transfer queue support for old GPUs in RADV. What have I learned from all this? It isn't that scary Kernel development is not as scary as it looks. It is a different technical challenge than what I was used to, but not in a bad way. Just needed to figure out a good workflow for how to configure a code editor, as well as what is a good way to test my work without rebuilding everything all the time. Maintainers are friendly AMD engineers have been very friendly and helpful to me all the way. Although there are a lot of memes and articles on the internet about attitude and rude/toxic messages by some kernel developers, I didn't see that in amdgpu at least. My approach was that even before I wrote a single line of code, I started talking to the maintainers (who would eventually review my patches) to find out what would be the good solution to them and how to get my work accepted. Communicating with the maintainers saved a lot of time and made the work faster, more pleasant and more collaborative. Development latency Sadly, there is a huge latency between a Linux kernel developer working on something and the work reaching end users. Even if the patches are accepted quickly, it can take 3~6 months until users can actually use it. In Mesa, we can merge any features to the next Mesa release up to the branch point, and afterwards we backport bug fixes to that release. Simple and efficient. In the Linux kernel, there is a deadline for merging new features to the next release, (it's not clearly communicated when that is). Bug fixes may be backported to previous releases any time, but upstream maintainers aren't involved in that. In hindsight, if I had focused on finishing the analog support and VCE1 first (instead of fixing all the bugs I found), my work would have ended up in Linux 6.18 (including the bug fixes, as there is no deadline for those). Due to how I prioritized bug fixing, the features I've developed are only included in Linux 6.19, so this will be the version where SI and CIK will default to amdgpu by default. XDC 2025 I presented a lightning talk on this topic at XDC 2025, where I talked about the state of SI and CIK support as of September 2025. You can find the slide deck here and the video here. Acknowledgements I'd like to say a big thank you to all of these people. All of the work I mentioned in this post would not have been possible without them. Alex Deucher, Christian König (amdgpu devs) Harry Wentland, Rodrigo Siquiera (DC devs) Marek Olšák, Pierre-Eric Pelloux-Prayer (radeonsi devs) Bas Nieuwenhuizen, Samuel Pitoiset (radv devs) Martin Roukala, né Peres (CI expert) Tom St Dennis (umr dev) Mauro Rossi (DCE6 in DC) Alexandre Demers (VCE1 research) Leonardo Frassetto (HW donation) Roman Elshin and others (testing) Pierre-Loup Griffais (Valve) Which graphics cards are affected exactly? When in doubt, consult Wikipedia. GFX6 aka. GCN1 - Southern Islands (SI) dedicated GPUs: amdgpu is now the default kernel driver as of Linux 6.19. The DC display driver is now the default and is now usable for these GPUs. DC now supports analog connectors, power management is less buggy, and video encoding is now supported by amdgpu. Tahiti Radeon HD 7870 XT, 7950, 7970, 7990, 8950, 8970, 8990 Radeon R9 280, 280X FirePro W8000, W9000, D500, D700, S9000, S9050, S10000 Radeon Sky 700, 900 Pitcairn Radeon HD 7850, 7870, 7970M, 8870, 8970M Radeon R9 265, 270, 270X, 370, 370X, M290X, M390 FirePro W5000, W7000, D300, R5000, S7000 Cape Verde Radeon HD 7730, 7750, 7770, 8730, 8760 Radeon R7 250E, 250X, 350, 450 FirePro W600, W4100, M4000, M6000 Oland Radeon HD 8570, 8670 Radeon R5 240, 250, 330, 340, 350, 430, 520, 610 FirePro W2100 various mobile GPUs Hainan various mobile GPUs GFX7 aka. GCN2 - Sea Islands (CIK) dedicated GPUs: amdgpu is now the default kernel driver as of Linux 6.19. The DC display driver is now the default for Bonaire (was already the case for Hawaii). DC now supports analog connectors. Minor bug fixes. Hawaii Radeon R9 290, 290X, 295X2, 390, 390X FirePro W8100, W9100, S9100, S9150, S9170 Bonaire Radeon HD 7790/8870 Radeon R7 260/360/450 Radeon RX 455, FirePro W5100, etc. various mobile GPUs GFX8 aka. GCN3 - Volcanic Islands (VI) dedicated GPUs: DC now supports analog connectors. (Note that amdgpu and DC were already supported on these GPUs since release.) Tonga Radeon R9 285, 380, 380X (other chips of this family are not affected by the work in this post)

- Lennart Poettering: Mastodon Stories for systemd v259 (2025/12/30 23:00)
  On Dec 17 we released systemd v259 into the wild. In the weeks leading up to that release (and since then) I have posted a series of serieses of posts to Mastodon about key new features in this release, under the #systemd259 hash tag. In case you aren't using Mastodon, but would like to read up, here's a list of all 25 posts: Post #1: systemd-resolved Hooks Post #2: dlopen() everything Post #3: systemd-analyze dlopen-metadata Post #4: run0 --empower Post #5: systemd-vmspawn --bind-user= Post #6: Musl libc support Post #7: systemd-repart without device name Post #8: Parallel kmod loading in systemd-modules-load.service Post #9: NvPCR Support Post #10: systemd-analyze nvcpcrs Post #11: systemd-repart Varlink IPC API Post #12: systemd-vmspawn block device serial Post #13: systemd-repart --defer-partitions-empty= + --defer-partitions-factory-reset= Post #14: userdb support for UUID queries Post #15: Wallclock time in service completion logging Post #16: systemd-firstboot --prompt-keymap-auto Post #17: $LISTEN_PIDFDID Post #18: Incremental partition rescanning Post #19: ExecReloadPost= Post #20: Transaction order cycle tracking Post #21: systemd-firstboot facelift Post #22: Per-User systemd-machined + systemd-importd Post #23: systemd-udevd's OPTIONS="dump-json" Post #24: systemd-resolved's DumpDNSConfiguration() IPC Call Post #25: DHCP Server EmitDomain= + Domain= I intend to do a similar series of serieses of posts for the next systemd release (v260), hence if you haven't left tech Twitter for Mastodon yet, now is the opportunity. My series for v260 will begin in a few weeks most likely, under the #systemd260 hash tag. In case you are interested, here is the corresponding blog story for systemd v258, here for v257, and here for v256.
- Timur Kristóf: Understanding your Linux open source drivers (2025/12/21 23:52)
  After introducing how graphics drivers work in general, I'd like to give a brief overview about what is what in the Linux graphics stack, what are the important parts and what the key projects are where the development happens, as well as what you need to do to get the best user

experience out of it. The open source Linux graphics driver stack Please refer to my previous post for a more detailed general explanation on graphics drivers in general. This post focuses on how things work in the open source graphics stack on Linux. Which GPUs are supported? We have open source drivers for the GPUs from all major manufacturers with varying degrees of success. Some companies (eg. AMD, Intel and others) choose to participate in the open source community and develop open source drivers themselves, with contributions also coming from other parties (eg. Valve, Red Hat, Google, etc). If you have an AMD or Intel GPU, their open source drivers typically work better than their alternatives (if any), and have been for a long time. Others (eg. NVidia and Apple) don't recognize the benefits of this style of development and leave it to the community to create drivers based on reverse-engineering or sometimes minimal help from the manufacturer. If you have an NVidia GPU, as of late 2025 the open source drivers may not be quite ready just yet. In this post I am not going to discuss the proprietary drivers. What parts do you need? The components you need in order to get your GPU working on open source drivers on a Linux distro are the following: Linux kernel (obviously) — contains the open source kernel drivers (KMD). linux-firmware — contains the necessary firmware. Note that even when the actual drivers are open source, most (all?) GPUs require closed source firmware to function. Mesa — contains the most userspace drivers (relevant to gaming) as well as a shader compiler stack. LLVM — needed by some Mesa drivers for shader compilation. Some vendors have other projects (eg. AMD ROCm) for supporting other features that aren't part of Mesa. These parts are out of scope for this blog post. To make your GPU work, you need new enough versions of the Linux kernel, linux-firmware and Mesa (and LLVM) that include support for your GPU. To make your GPU work well, I highly recommend to use the latest stable versions of all of the above. If you use old versions, you are missing out. By using old versions you are choosing not to benefit from the latest developments (features and bug fixes) that open source developers have worked on, and you will have a sub-par experience (especially on relatively new hardware). Wait, aren't the drivers in the kernel? If you read Reddit posts, you will stumble upon some people who believe that "the drivers are in the kernel" on Linux. This is a half-truth. Only the KMDs are part of the kernel, everything else (linux-firmware, Mesa, LLVM) is distributed in separate packges. How exactly those packages are organized, depends on your distribution. What is the Mesa project? Mesa is a collection of userspace drivers which implement various different APIs. It is the cornerstone of the open source graphics stack. I'm going to attempt to give a brief overview of what are the most relevant parts of Mesa. Gallium An important part of Mesa is the Gallium driver infrastructure, which contains a lot of common code for implementing different APIs, such as: Graphics: OpenGL, OpenGL ES, EGL Compute: OpenCL Video decoding and encoding: VAAPI (previously also VDPAU) Vulkan Mesa also contains a collection of Vulkan drivers. Originally, Vulkan was deemed "lower level than Gallium", so Vulkan drivers are not part of the Gallium driver infrastructure. However, Vulkan has a lot of overlapping functionality with the aforementioned APIs, so Vulkan drivers still share a lot of code with their Gallium counterparts when appropriate. NIR Another important part of Mesa is the NIR shader compiler stack, which is at the heart of every Mesa driver that is still being maintained. This enables sharing a lot of compiler code across different drivers. I highly recommend Faith Ekstrand's post In defense of NIR to learn more about it. Compatibility layers and API translation Technically they are not drivers, but in practice, if you want to run Windows games, you will need a compatibility layer like Wine or Proton, including graphics translation layers. The recommended ones are: DXVK: translates DirectX 8-11 to Vulkan. VKD3D-Proton: translates DirectX 12 to Vulkan. Those are default in Proton and offer the best performance. However, for "political" reasons, these are sadly not the defaults in Wine, so either you'll have to use Proton or make sure to install the above in Wine manually. Just for the sake of completeness, I'll also mention the Wine defaults: WineD3D: translates DirectX 1-11 to OpenGL. It is what we all used before Vulkan and DXVK existed, sadly its performance and compatibility has always been lacking. VKD3D: translates DirectX 12 to Vulkan. Not practically usable. Can only actually run a select few games, and those with lackluster performance. (Not to be confused with VKD3D-Proton, which is actually fully-featured.) Side note about window systems Despite the X server being abandoned for a long time, there is still a debate between Linux users whether to use a Wayland compositor or the X server. I'm not going to discuss the advantages and disadvantages of these, because I don't participate in their development and I feel it has already been well-explained by the community. I'm just going to say that it helps to choose a competent compositor that implements direct scanout. This means that the frames as rendered by your game can be sent directly to the display without the need for the compositor to do any additional processing on it. In this blog post I focus on just the driver stack, because that is largely shared between both solutions. Making your games run (well) Sadly, many Linux distributions choose to ship old versions of the kernel and/or other parts of the driver stack, giving their users a sub-par experience. Debian, Ubuntu LTS and their derivatives like Mint, Pop OS, etc. are all guilty of this. They justify this by claiming that older versions are more reliable, but this is actually not true. In reality, us driver developers as well as the developers of the API translation layers work hard to implement new features that are needed to get new games working, as well as fixing bugs that are exposed by new games (or updates of old games). Regressions are real, but they are usually quickly fixed thanks to the extensive testing that we do: every time we merge new code, our automated testing system runs the full Vulkan conformance test suite to make sure that all functionality is still intact, thanks to Martin's genious.

- Simon Ser: Status update, December 2025 (2025/12/21 22:00)
  Hi all! This month the new KMS plane color pipeline API has finally been merged! It took multiple years and continued work and review by engineers from multiple organizations, but at last we managed to push it over the finish line. This new API exposes to user-space new hardware blocks: these applying color transformations before blending multiple KMS planes as a final composited image to be sent on the wire. This API unlocks power-efficient and low-latency color management features such as HDR. Still, much remains to be done. Color pipelines are now exposed on AMD and VKMS, Intel and other vendors are still working on their driver implementation. Melissa Wen has written a drm_info patch to show pipeline information, some more work is needed to plumb it through drmdb. Some patches have been floated to leverage color pipelines for post-blending transforms too (currently KMS only supports a fixed rudimentary post-blending pipeline with two LUTs and one 3×3 matrix). On the wlroots side, Félix Poisot has redesigned the way post-blending color transforms are applied by the renderer. The API used to be a mix of descriptive (describing which primaries and transfer functions the output buffer uses) and prescriptive (passing a list of operations to apply). Now it's fully prescriptive, which will help for offloading these transformations to the DRM backend. GnSight has contributed support for the wlr-foreign-toplevel-management-v1 protocol to the Cage kiosk compositor. This enables better control over windows running inside the compositor: external tools can close or bring windows to the front. mhorky has added client support for one-way method calls to go-varlink, as well as a nice Registry enhancement to add support for the org.varlink.service interface for free, for discovery and introspection of Varlink services. Now that the module is feature-complete I've released version 0.1.0. delthas has introduced support for authenticating with the soju IRC bouncer via TLS client certificates. He has contributed a simple audio recorder to the Goguma mobile IRC client, plus new buttons above the reaction list to be able to easily +1 another user's reaction. Hubert Hirtz has sent a collection of bug fixes and has added a button to reveal the password field contents on the connection screen. I've resurrected work on some old projects I'd almost forgotten about. I've pushed a few patches for libicc, adding support for encoding multi-process transforms, luminance and metadata. I've added a basic test suite to libjsonschema, and improved handling of objects and arrays without enough information to automatically generate types from. But the old project I've spent most of my time on is go-mls, a Go implementation of the Messaging Layer Security (MLS) protocol. MLS is an end-to-end encryption protocol for chat messages. My goal is twofold: learn how MLS works under the hood (implementing something is one of the best ways for me to understand that something),

and lay the groundwork for a future end-to-end encryption IRC extension. This month I've fixed up the remaining failures in the test suite and I've implemented just enough to be able to create a group, add members to it, and exchange an encrypted message. I'll work on remaining group operations (e.g. removing a member) next. Last, I've migrated FreeDesktop's Mailman 3 installation to PostgreSQL from SQLite. Mailman 3's SQLite integration had pretty severe performance issues, these are gone with PostgreSQL. The migration wasn't straightforward: there is no tooling to migrate Mailman 3 core's data between database engines, so I had to manually fill the new database with the old data. I've migrated two more mailing lists to Mailman 3: fhs and nouveau. I plan to continue the migration in the coming months, and hopefully we'll be able to decommission Mailman 2 in a not-so-distant future. See you next year!

- Timur Kristóf: How do graphics drivers work? (2025/12/16 00:09)
  I'd like to give an overview on how graphics drivers work in general, and then write a little bit about the Linux graphics stack for AMD GPUs. The intention of this post is to clear up a bunch of misunderstandings that people have on the internet about open source graphics drivers. What is a graphics driver? A graphics driver is a piece of software code that is written for the purpose of allowing programs on your computer to access the features of your GPU. Every GPU is different and may have different capabilities or different ways of achieving things, so they need different drivers, or at least different code paths in a driver that may handle multiple GPUs from the same vendor and/or the same hardware generation. The main motivation for graphics drivers is to allow applications to utilize your hardware efficiently. This enables games to render pretty pixels, scientific apps to calculate stuff, as well as video apps to encode / decode efficiently. Organization of graphics drivers Compared to drivers for other hardware, graphics is very complicated because the functionality is very broad and the differences between each piece of hardware can be also vast. Here is a simplified explanation on how a graphics driver stack usually works. Note that most of the time, these components (or some variation) are bundled together to make them easier to use. GPU firmware (FW) — low-level code for power management, context switching, command processing, display engine, video encoding/decoding, etc. Kernel driver, aka. kernel-mode driver (KMD) — makes it possible for multiple userspace applications to submit commands to the GPU, and is responsible for memory management and display functionality. Userspace driver, aka. user-mode driver (UMD) — responsible for implementing an API, such as Vulkan, OpenGL, etc. For each piece of hardware, there may be multiple different UMDs implementing different APIs. Shader compiler — a userspace library that compiles shader programs for your GPU from the HW-independent code that applications have. Can be possibly shared between UMDs, sometimes developed as a separate project. I'll give a brief overview of each component below. GPU firmware Most GPUs have additional processors (other than the shader cores) which run a firmware that is responsible for operating the low-level details of the hardware, usually stuff that is too low-level even for the kernel. The firmware on those processors are responsible for: power management, context switching, command processing, display, video encoding/decoding etc. Among other things it parses the commands we submitted to it, launches shaders, distributes work between the shader cores etc. Some GPU manufacturers are moving more and more functionality to firmware, which means that the GPU can operate more autonomously and less intervention is needed by the CPU. This tendency is generally positive for reducing CPU time spent on programming the GPU (as well as "CPU bubbles"), but at the same time it also means that the way the GPU actually works becomes less transparent. Kernel driver You might ask, why not implement all driver functionality in the kernel? Wouldn't it be simpler to "just" have everything in the kernel? The answer is no, mainly because there is a LOT going on which nobody wants in the kernel. You don't want to have your kernel crash when a game misbehaves. Sadly it can still happen, but it would happen a lot more if the kernel and userspace components weren't separated. You definitiely don't want to run a fully-fledged compiler inside your kernel which takes arbitrary input from the user. You want to avoid having to upgrade your

kernel to deploy most fixes and improvements to the graphics stack. (This is not always avoidable but can be minimized.) So, usually, the KMD is only left with some low-level tasks that every user needs: Command submission userspace API: an interface that allows userspace processes to submit commands to the GPU, query information about the GPU, etc. Memory management: deciding which process gets to use how much VRAM, defining GTT, handling low-memory situations, etc. Display functionality: making display connectors work, by programming the registers of the display controller. There is also a separate uAPI for just this purpose. Power management: making sure the GPU doesn't draw too much power when not needed, and also making sure applications can get the best clock speeds etc. when that is needed, in cooperation with the power management firmware. GPU recovery: when the GPU hangs or crashes for some reason, it's the kernel's responsibility to ensure that the GPU can be recovered and that the crash doesn't affect other processes. Userspace driver Applications interact with userspace drivers instead of the kernel (or the hardware directly). Userspace drivers are compiled as shared libraries and are responsible for implementing one or more specific APIs for graphics, compute or video for a specific family of GPUs. (For example, Vulkan, OpenGL or OpenCL, etc.) Each graphics API has entry points which load the available driver(s) for the GPU(s) in the user's system. The Vulkan loader is an example of this; other APIs have similar components for this purpose. The main functionality of a userspace driver is to take the commands from the API (for example, draw calls or compute dispatches) and turn them into low level commands in a binary format that the GPU can understand. In Vulkan, this is analogous to recording a command buffer. Additionally, they utilize a shader compiler to turn a higher level shader language (eg. GLSL) or bytecode (eg. SPIR-V) into hardware instructions which the GPU's shader cores can execute. Furthermore, userspace drivers also take part in memory management, they basically act as an interface between the memory model of the graphics API and kernel's memory manager. The userspace driver calls the aforementioned kernel uAPI to submit the recorded commands to the kernel which then schedules it and hands it to the firmware to be executed. Shader compiler If you've seen a loading screen in your favourite game which told you it was "compiling shaders…" you probably wondered what that's about and why it's necessary. Unlike CPUs which have converged to a few common instruction set architectures (ISA), GPUs are a mess and don't share the same ISA, not even between different GPU models from the same manufacturer. Although most modern GPUs have converged to SIMD based architectures, the ISA is still very different between manufacturers and it still changes from generation to generation (sometimes different chips of the same generation have slightly different ISA). GPU makers keep adding new instructions when they identify new ways to implement some features more effectively. To deal with all that mess, graphics drivers have to do online compilation of shaders (as opposed to offline compilation which usually happens for apps running on your CPU). This means that shaders have to be recompiled when the userspace graphics driver is updated either because new functionality is available or because bug fixes were added to the driver and/or compiler. But I only downloaded one driver! On some systems (especially proprietary operating systems like Windows), GPU manufacturers intend to make users' lives easier by offering all of the above in a single installer package, which is just called "the driver". Typically such a package includes: Firmware files for all hardware that the package supports A kernel driver Several userspace drivers for various APIs A shader compiler (sometimes more) that is used by those userspace drivers A "user-friendly" application (ie. a control panel) to present all the functionality to the user Various other utilities and libraries (that you may or may not need). But I didn't download any drivers! On some systems (typically on open source systems like Linux distributions), usually you can already find a set of packages to handle most common hardware, so you can use most functionality out of the box without needing to install anything manually. Neat, isn't it? However, on open source systems, the graphics stack is more transparent, which means that there are many parts that are scattered across different projects, and in some cases there is more than one driver available for the same HW. To end users, it can be very confusing. However, this doesn't mean that open source drivers are designed worse. It is just that due to their community oriented nature, they are organized differently. One of the main sources of confusion is that various Linux distributions mix and match different versions of the kernel with different versions of different UMDs which means that users of different distros can get a wildly different user experience based on the choices made for them by the developers of the distro. Another source of confusion is that we driver developers are really, really bad at naming things, so sometimes different projects end up having the same name, or some projects have nonsensical or outdated names. The Linux graphics stack In the next post, I'll continue this story and discuss how the above applies to the open source Linux graphics stack.

- Hari Rana: Please Fund My Continued Accessibility Work on GNOME! (2025/12/16 00:00)
  Hey, I have been under distress lately due to personal circumstances that are outside my control. I cannot find a permanent job that allows me to function, I am not eligible for government benefits, my grant proposals to work on free and open-source projects got rejected, paid internships are quite difficult to find, especially when many of them prioritize new contributors. Essentially, I have no stable, monthly income that allows me to sustain myself. Nowadays, I mostly volunteer to improve accessibility throughout GNOME apps, either by enhancing the user experience for people with disabilities, or enabling them to use them. I helped make most of GNOME Calendar accessible with a keyboard and screen reader, with additional ongoing effort involving merge requests !564 and !598 to make the month view accessible, all of which is an effort no company has ever contributed to, or would ever contribute to financially. These merge requests require literal thousands of hours for research, development, and testing, enough to sustain me for several years if I were employed. I would really appreciate any kinds of donations, especially ones that happen periodically to increase my monthly income. These donations will allow me to sustain myself while allowing me to work on accessibility throughout GNOME, essentially 'crowdfunding' development without doing it on the behalf of the GNOME Foundation or another organization. Donate on Liberapay Support on Ko-fi Sponsor on GitHub Send via PayPal
- Sebastian Wick: Flatpak Pre-Installation Approaches (2025/12/13 17:17)
  Together with my then-colleague Kalev Lember, I recently added support for pre-installing Flatpak applications. It sounds fancy, but it is conceptually very simple: Flatpak reads configuration files from several directories to determine which applications should be pre-installed. It then installs any missing applications and removes any that are no longer supposed to be pre-installed (with some small caveats). For example, the following configuration tells Flatpak that the devel branch of the app org.test.Foo from remotes which serve the collection org.test.Collection, and the app org.test.Bar from any remote should be installed: [Flatpak Preinstall org.test.Foo] CollectionID=org.test.Collection Branch=devel [Flatpak Preinstall org.test.Bar] By dropping in another confiuration file with a higher priority, pre-installation of the app org.test.Foo can be disabled: [Flatpak Preinstall org.test.Foo] Install=false The installation procedure is the same as it is for the flatpak-install command. It supports installing from remotes and from side-load repositories, which is to say from a repository on a filesystem. This simplicity also means that system integrators are responsible for assembling all the parts into a functioning system, and that there are a number of choices that need to be made for installation and upgrades. The simplest way to approach this is to just ship a bunch of config files in /usr/share/flatpak/preinstall.d and config files for the remotes from which the apps are available. In the installation procedure, flatpak-preinstall is called and it will download the Flatpaks from the remotes over the network into /var/lib/flatpak. This works just fine, until someone needs one of those apps but doesn't have a suitable network connection. The next way one could approach this is exactly the same way, but with a sideload repository on the installation medium which contains the apps that will get pre-installed. The flatpak-preinstall command needs to be pointed at this repository at install time, and the process which creates the installation medium needs to be adjusted to create this repository. The installation process now works without a

network connection. System updates are usually downloaded over the network, just as new pre-installed applications will be. It is also possible to simply skip flatpak-preinstall, and use flatpak-install to create a Flatpak installation containing the pre-installed apps which get shipped on the installation medium. This installation can then be copied over from the installation medium to /var/lib/flatpak in the installation process. It unfortunately also makes the installation process less flexible because it becomes impossible to dynamically build the configuration. On modern, image-based operating systems, it might be tempting to just ship this Flatpak installation on the image because the flexibility is usually neither required nor wanted. This currently does not work for the simple reason that the default system installation is in /var/lib/flatpak, which is not in /usr which is the mount point of the image. If the default system installation was in the image, then it would be read-only because the image is read-only. This means we could not update or install anything new to the system installation. If we make it possible to have two different system installations — one in the image, and one in /var — then we could update and install new things, but the installation on the image would become useless over time because all the runtimes and apps will be in /var anyway as they get updated. All of those issues mean that even for image-based operating systems, pre-installation via a sideload repository is not a bad idea for now. It is however also not perfect. The kind of "pure" installation medium which is simply an image now contains a sideload repository. It also means that a factory reset functionality is not possible because the image does not contain the pre-installed apps. In the future, we will need to revisit these approaches to find a solution that works seamlessly with image-based operating systems and supports factory reset functionality. Until then, we can use the systems mentioned above to start rolling out pre-installed Flatpaks.

- Dave Airlie (blogspot): fedora 43: bad mesa update oopsie (2025/11/24 01:42)
  F43 picked up the two patches I created to fix a bunch of deadlocks on laptops reported in my previous blog posting. Turns out Vulkan layers have a subtle thing I missed, and I removed a line from the device select layer that would only matter if you have another layer, which happens under steam.The fedora update process caught this, but it still got published which was a mistake, need to probably give changes like this more karma thresholds.I've released a new update https://bodhi.fedoraproject.org/updates/FEDORA-2025-2f4ba7cd17 that hopefully fixes this. I'll keep an eye on the karma.
- Juan A. Suarez: Major Upgrades to the Raspberry Pi GPU Driver Stack (XDC 2025 Recap) (2025/11/23 23:00)
  XDC 2025 happened at the end of September, beginning of October this year, in Kuppelsaal, the historic TU Wien building in Vienna. XDC, The X.Org Developer's Conference, is truly the premier gathering for open-source graphics development. The atmosphere was, as always, highly collaborative and packed with experts across the entire stack. I was thrilled to present, together with my workmate Ella Stanforth, on the progress we have made in enhancing the Raspberry Pi GPU driver stack. Representing the broader Igalia Graphics Team that work on this GPU, Ella and I detailed the strides we have made in the OpenGL driver, though part of the improvements affect also the Vulkan driver. The presentation was divided in two parts. In the first one, we talked about the new features that we were implementing, or are under implementation, mainly to make the driver more closely aligned with OpenGL 3.2. Key features explained were 16-bit Normalized Format support, Robust Context support, and Seamless cubemap implementation. Beyond these core OpenGL updates, we also highlighted other features, such as NIR printf support, framebuffer fetch or dual source blend, which is important for some game emulators. The second part was focused on specific work done to improve the performance. Here, we started with different traces from the popular GFXBench application, and explained the main improvements done throughout the year, with a look at how much each of these changes improved the performance for each of the benchmarks (or in average). At the end, for some benchmarks we nearly doubled the performance compared to last year. I won't explain

here each of the changes done, But I encourage the reader to watch the talk, which is already available. For those that prefer to check the slides instead of the full video, you can view them here: Outside of the technical track, the venue's location provided some excellent down time opportunities to have lunch at different nearby places. I need to highlight here one that I really enjoyed: An's Kitchen Karlsplatz. This cozy Vietnamese street food spot quickly became one of my favourite places, and I went there a couple of times. On the last day, I also had the opportunity to visit some of the most recomendable sightseeings spots in Vienna. Of course, one needs more than a half-day to do a proper visit, but at least it helps to spark an interest to write it down to pay a full visit to the city. Meanwhile, I would like to thank all the conference organizers, as well as all the attendees, and I look forward to see them again.

- Lennart Poettering: Mastodon Stories for systemd v258 (2025/11/17 23:00)
  Already on Sep 17 we released systemd v258 into the wild. In the weeks leading up to that release I have posted a series of serieses of posts to Mastodon about key new features in this release, under the #systemd258 hash tag. It was my intention to post a link list here on this blog right after completing that series, but I simply forgot! Hence, in case you aren't using Mastodon, but would like to read up, here's a list of all 37 posts: Post #1: systemctl start -v Post #2: Home areas Post #3: systemd-resolved delegate zones Post #4: Foreign UID range Post #5: /etc/hostname ??? wildcards Post #6: Quota on /tmp/ Post #7: ConcurretnySoftMax= + ConcurrencyHardMax= Post #8: Product UUID in ConditionHost= Post #9: Context OSC terminal sequences Post #10: uki-url Boot Loader Spec Type #1 fields Post #11: rd.break= boot breakpoints Post #12: Factory Reset Rework Post #13: systemd-resolved DNS Configuration Change IPC Subscription API Post #14: io.systemd.boot-entries.extra= SMBIOS Type #11 Key Post #15: Bring Your Own Firmware Post #16: userdb record aliases Post #17: systemd-validatefs and its xattrs Post #18: Offline Signing of Artifacts Post #19: PAMName= in services hooked up to ask-password protocol Post #20: x-systemd.graceful-option= mount option Post #21: systemd-userdb-load-credentials.service Post #22: systemd-vmspawn --grow-image=a Post #23: systemd-notify --fork Post #24: $TERM auto-discovery Post #25: Rebooting/Powering off systemd-nspawn containers via hotkey Post #26: ExecStart= | modifier Post #27: systemctl reload reloads confexts Post #28: Server side userdb filtering Post #29: Quota on StateDirectory= and friends Post #30: systemd-analyze unit-shell Post #31: /etc/issue.d/ drop-in for AF_VSOCK CID Post #32: fsverity in systemd-repart Post #33: AcceptFileDescriptor= + PassPIDFD= Post #34: Tab completion in interactive systemd-firstboot Post #35: rd.systemd.pull= kernel command line option/Boot into tarball Post #36: ConditionKernelModuleLoaded= Post #37: systemd-analyze chid Post #38: homectl list-signing-keys/get-signing-key/add-signing-key/remove-signing-key Post #39: DDI Image Filters Post #40: Android USB Debugging udev rules Post #41: systemd-vmspawn's --smbios11= switch Post #42: $MAINPIDFDID + $MANAGERPIDFDID Post #43: $DEBUG_INVOCATION=1 Respected by all systemd services Post #44: LoaderDeviceURL EFI Variable and systemd.pull='s origin kernel command line switch Post #45: cgroupv1 removal Post #46: ProtectHostname=private Post #47: homectl adopt + homectl register Post #48: systemd-machined Varlink APIs Post #49: DeferTrigger and "lenient" job mode Post #50: Automatic Removal of foreign UID owned delegate subgroups in the per-user service manager Post #51: Per-user ask-password protocol Post #52: PrivateUsers=full Post #53: LoadCredentialEncrypted= in the per-user service manager Post #54: dissect_image builtin in systemd-udevd Post #55: BPF Delegation via Tokens I intend to do a similar series of serieses of posts for the next systemd release (v259), hence if you haven't left tech Twitter for Mastodon yet, now is the opportunity. We intend to shorten the release cycle a bit for the future, and in fact managed to tag v259-rc1 already yesterday, just 2 months after v258. Hence, my series for v259 will begin soon, under the #systemd259 hash tag. In case you are interested, here is the corresponding blog story for systemd v257, and here for v256.
- Rodrigo Siqueira: XDC 2025 (2025/11/17 00:00)

It has been a long time since I published any update in this space. Since this was a year of colossal changes for me, maybe it is also time for me to make something different with this blog and publish something just for a change — why not start talking about XDC 2025? This year, I attended XDC 2025 in Vienna as an Igalia developer. I was thrilled to see some faces from people I worked with in the past and people I'm working with now. I had a chance to hang out with some folks I worked with at AMD (Harry, Alex, Leo, Christian, Shashank, and Pierre), many Igalians (Žan, Job, Ricardo, Paulo, Tvrtko, and many others), and finally some developers from Valve. In particular, I met Tímur in person for the first time, even though we have been talking for months about GPU recovery. Speaking of GPU recovery, we held a workshop on this topic together. The workshop was packed with developers from different companies, which was nice because it added different angles on this topic. We began our discussion by focusing on the topic of job resubmission. Christian began sharing a brief history of how the AMDGPU driver started handling resubmission and the associated issues. After learning from erstwhile experience, amdgpu ended up adopting the following approach: When a job cause a hang, call driver specific handler. Stop the scheduler. Copy all jobs from the ring buffer, minus the job that caused the issue, to a temporary ring. Reset the ring buffer. Copy back the other jobs to the ring buffer. Resume the scheduler. Below, you can see one crucial series associated with amdgpu recovery implementation: https://lore.kernel.org/amd-gfx/20250701184451.11868-1-alexander.deucher@amd.com/T/#m5df543b5cbbdc31e2834c955d1e3e30b939cb77 The next topic was a discussion around the replacement of drm_sched_resubmit_jobs() since this function became deprecated. Just a few drivers still use this function, and they need a replacement for that. Some ideas were floating around to extract part of the specific implementation from some drivers into a generic function. The next day, Philipp Stanner continued to discuss this topic in his workshop, DRM GPU Scheduler. Another crucial topic discussed was improving GPU reset debuggability to narrow down which operations cause the hang (keep in mind that GPU recovery is a medicine, not the cure to the problem). Intel developers shared their strategy for dealing with this by obtaining hints from userspace, which helped them provide a better set of information to append to the devcoredump. AMD could adopt this alongside dumping the IB data into the devcoredump (I am already investigating this). Finally, we discussed strategies to avoid hang issues regressions. In summary, we have two lines of defense: IGT: At the IGT level, we can have more tests that insert malicious instructions into the ring buffer, forcing the driver into an invalid state and triggering the recovery process. HangTest suite: HangTest suite is a tool that simulates some potential hangs using Vulkan. Some tests are already available in this suite, but we should explore more creative combinations for trying to trigger hangs. Lighting talk This year, as always, XDC was super cool, packed with many engaging presentations which I highly recommend everyone check out. If you are interested, check the schedule and the presentation recordings available on the X.Org Foundation Youtube page. Anyway, I hope this blog post marks the inauguration of a new era for this site, where I will start posting more content ranging from updates to tutorials. See you soon.

- Simon Ser: Status update, November 2025 (2025/11/15 22:00)
  Hi! This month a lot of new features have added to the Goguma mobile IRC client. Hubert Hirtz has implemented drafts so that unsent text gets saved and network disconnections don't disrupt users typing a message. He also enabled replying to one's own messages, changed the appearance of short messages containing only emoji, upgraded our emoji library to Unicode version 16, fixed some linkifier bugs and added unit tests. Markus Cisler has added a new option in the message menu to show a user's profile. I've added an on-disk cache for images (with our own implementation, because the widely used cached_network_image package is heavyweight). I've been working on displaying network icons and blocking users, but that work is not finished yet. I've also contributed some maintenance fixes for our webcrypto.dart dependency (toolkit upgrades and CI fixes). The soju IRC bouncer has also got some love this month. delthas has contributed support for labeled-response for soju

clients, allowing more reliable matching of server replies with client commands. I’ve introduced a new icon directive to configure an image representing the bouncer. soju v0.10.0 has been released, followed by soju v0.10.1 including bug fixes from Karel Balej and Taavi Väänänen. In Wayland news, wlroots v0.19.2 and v0.18.3 have been released thanks to Simon Zeni. I’ve added support for the color-representation protocol for the Vulkan renderer, allowing clients to configure the color encoding and range for YCbCr content. Félix Poisot has been hard at work with more color management patches: screen default color primaries are now extracted from the EDID and exposed to compositors, the cursor is now correctly converted to the output’s primaries and transfer function, and some work-in-progress patches switch the renderer API from a descriptive model to a prescriptive model. go-webdav v0.7.0 has been released with a patch from prasad83 to play well with Thunderbird. I’ve updated clients to make multi-status errors non-fatal, returning partial data alongside the error. I’ve released drm_info v2.9.0 with improvements mentioned in the previous status update plus support for the TILE connector property. See you next month!

- Dave Airlie (blogspot): a tale of vulkan/nouveau/nvk/zink/mutter + deadlocks (2025/11/10 03:16)
 I had a bug appear in my email recently which led me down a rabbit hole, and I'm going to share it for future people wondering why we can't have nice things.Bug:1. Get an intel/nvidia (newer than Turing) laptop.2. Log in to GNOME on Fedora 42/43 3. Hotplug a HDMI port that is connected to the NVIDIA GPU.4. Desktop stops working.My initial reproduction got me a hung mutter process with a nice backtrace which pointed at the Vulkan Mesa device selection layer, trying to talk to the wayland compositor to ask it what the default device is. The problem was the process was the wayland compositor, and how was this ever supposed to work. The Vulkan device selection was called because zink called EnumeratePhysicalDevices, and zink was being loaded because we recently switched to it as the OpenGL driver for newer NVIDIA GPUs.I looked into zink and the device select layer code, and low and behold someone has hacked around this badly already, and probably wrongly and I've no idea what the code does, because I think there is at least one logic bug in it. Nice things can't be had because hacks were done instead of just solving the problem. The hacks in place ensured under certain circumstances involving zink/xwayland that the device select code to probe the window system was disabled, due to deadlocks seen. I'd no idea if more hacks were going to help, so I decided to step back and try and work out better.The first question I had is why WAYLAND_DISPLAY is set inside the compositor process, it is, and if it wasn't I would never hit this. It's pretty likely on the initial compositor start this env var isn't set, so the problem only becomes apparent when the compositor gets a hotplugged GPU output, and goes to load the OpenGL driver, zink, which enumerates and hits device select with env var set and deadlocks.I wasn't going to figure out a way around WAYLAND_DISPLAY being set at this point, so I leave the above question as an exercise for mutter devs.How do I fix it?Attempt 1:At the point where zink is loading in mesa for this case, we have the file descriptor of the GPU device that we want to load a driver for. We don't actually need to enumerate all the physical devices, we could just find the ones for that fd. There is no API for this in Vulkan. I wrote an initial proof of concept instance extensions call VK_MESA_enumerate_devices_fd. I wrote initial loader code to play with it, and wrote zink code to use it. Because this is a new instance API, device-select will also ignore it. However this ran into a big problem in the Vulkan loader. The loader is designed around some internals that PhysicalDevices will enumerate in similiar ways, and it has to trampoline PhysicalDevice handles to underlying driver pointers so that if an app enumerates once, and enumerates again later, the PhysicalDevice handles remain consistent for the first user. There is a lot of code, and I've no idea how hotplug GPUs might fail in such situations. I couldn't find a decent path forward without knowing a lot more about the Vulkan loader. I believe this is the proper solution, as we know the fd, we should be able to get things without doing a full enumeration then picking the answer using the fd info. I've asked Vulkan WG to take a look at this, but I still need to fix the bug.Attempt 2:Maybe I can just turn off device selection, like the current hacks do, but in a better manner. Enter VK_EXT_layer_settings. This extensions

allows layers to expose a layer setting in the instance creation. I can have the device select layer expose a setting which says don't touch this instance. Then in the zink code where we have a file descriptor being passed in and create an instance, we set the layer setting to avoid device selection. This seems to work but it has some caveats, I need to consider, but I think should be fine.zink uses a single VkInstance for it's device screen. This is shared between all pipe_screens. Now I think this is fine inside a compositor, since we shouldn't ever be loading zink via the non-fd path, and I hope for most use cases it will work fine, better than the current hacks and better than some other ideas we threw around. The code for this is in [1].What else might be affected:If you have a vulkan compositor, it might be worth setting the layer setting if the mesa device select layer is loaded, esp if you set the DISPLAY/WAYLAND_DISPLAY and do any sort of hotplug later. You might be safe if you EnumeratePhysicalDevices early enough, the reason it's a big problem in mutter is it doesn't use Vulkan, it uses OpenGL and we only enumerate Vulkan physical devices at runtime through zink, never at startup.AMD and NVIDIA I think have proprietary device selection layers, these might also deadlock in similiar ways, I think we've seen some wierd deadlocks in NVIDIA driver enumerations as well that might be a similiar problem. [1] https://gitlab.freedesktop.org/mesa/mesa/-/merge_requests/38252

- Sebastian Wick: Flatpak Happenings (2025/11/04 20:28)
  Yesterday I released Flatpak 1.17.0. It is the first version of the unstable 1.17 series and the first release in 6 months. There are a few things which didn’t make it for this release, which is why I’m planning to do another unstable release rather soon, and then a stable release still this year. Back at LAS this year I talked about the Future of Flatpak and I started with the grim situation the project found itself in: Flatpak was stagnant, the maintainers left the project and PRs didn’t get reviewed. Some good news: things are a bit better now. I have taken over maintenance, Alex Larsson and Owen Taylor managed to set aside enough time to make this happen and Boudhayan Bhattcharya (bbhtt) and Adrian Vovk also got more involved. The backlog has been reduced considerably and new PRs get reviewed in a reasonable time frame. I also listed a number of improvements that we had planned, and we made progress on most of them: It is now possible to define which Flatpak apps shall be pre-installed on a system, and Flatpak will automatically install and uninstall things accordingly. Our friends at Aurora and Bluefin already use this to ship core apps from Flathub on their bootc based systems (shout-out to Jorge Castro). The OCI support in Flatpak has been enhanced to support pre-installing from OCI images and remotes, which will be used in RHEL 10 We merged the backwards-compatible permission system. This allows apps to use new, more restricting permissions, while not breaking compatibility when the app runs on older systems. Specifically access to input devices such as gamepads, and access to the USB portal can now be granted in this way. It will also help us to transition to PipeWire. We have up-to-date docs for libflatpak again Besides the changes directly in Flatpak, there are a lot of other things happening around the wider ecosystem: bbhtt released a new version of flatpak-builder Enhanced License Compliance Tools for Flathub Adrian and I have made plans for a service which allows querying running app instances (systemd-appd). This provides a new way of authenticating Flatpak instances and is a prerequisite for nested sandboxing, PipeWire support, and getting rid of the D-Bus proxy. My previous blog post went into a few more details. Our friends at KDE have started looking into the XDG Intents spec, which will hopefully allow us to implement deep-linking, thumbnailing in Flatpak apps, and other interesting features Adrian made progress on the session save/restore Portal Some rather big refactoring work in the Portals frontend, and GDBus and libdex integration work which will reduce the complexity of asynchronous D-Bus What I have also talked about at my LAS talk is the idea of a Flatpak-Next project. People got excited about this, but I feel like I have to make something very clear: If we redid Flatpak now, it would not be significantly better than the current Flatpak! You could still not do nested sandboxing, you would still need a D-Bus proxy, you would still have a complex permission system, and so on. Those problems require work outside of Flatpak, but have to integrate with

Flatpak and Flatpak-Next in the future. Some of the things we will be doing include: Work on the systemd-appd concept Make varlink a feasible alternative to D-Bus D-Bus filtering in the D-Bus daemons Network sandboxing via pasta PipeWire policy for sandboxes New Portals So if you're excited about Flatpak-Next, help us to improve the Flatpak ecosystem and make Flatpak-Next more feasible!

- Melissa Wen: Kworkflow at Kernel Recipes 2025 (2025/11/03 21:30)
This was the first year I attended Kernel Recipes and I have nothing but say how much I enjoyed it and how grateful I'm for the opportunity to talk more about kworkflow to very experienced kernel developers. What I mostly like about Kernel Recipes is its intimate format, with only one track and many moments to get closer to experts and people that you commonly talk online during your whole year. In the beginning of this year, I gave the talk Don't let your motivation go, save time with kworkflow at FOSDEM, introducing kworkflow to a more diversified audience, with different levels of involvement in the Linux kernel development. At this year's Kernel Recipes I presented the second talk of the first day: Kworkflow - mix & match kernel recipes end-to-end. The Kernel Recipes audience is a bit different from FOSDEM, with mostly long-term kernel developers, so I decided to just go directly to the point. I showed kworkflow being part of the daily life of a typical kernel developer from the local setup to install a custom kernel in different target machines to the point of sending and applying patches to/from the mailing list. In short, I showed how to mix and match kernel workflow recipes end-to-end. As I was a bit fast when showing some features during my presentation, in this blog post I explain each slide from my speaker notes. You can see a summary of this presentation in the Kernel Recipe Live Blog Day 1: morning. Introduction Hi, I'm Melissa Wen from Igalia. As we already started sharing kernel recipes and even more is coming in the next three days, in this presentation I'll talk about kworkflow: a cookbook to mix & match kernel recipes end-to-end. This is my first time attending Kernel Recipes, so lemme introduce myself briefly. As I said, I work for Igalia, I work mostly on kernel GPU drivers in the DRM subsystem. In the past, I co-maintained VKMS and the v3d driver. Nowadays I focus on the AMD display driver, mostly for the Steam Deck. Besides code, I contribute to the Linux kernel by mentoring several newcomers in Outreachy, Google Summer of Code and Igalia Coding Experience. Also, by documenting and tooling the kernel. And what's this cookbook called kworkflow? Kworkflow (kw) Kworkflow is a tool created by Rodrigo Siqueira, my colleague at Igalia. It's a single platform that combines software and tools to: optimize your kernel development workflow; reduce time spent in repetitive tasks; standardize best practices; ensure that deployment data flows smoothly and reliably between different kernel workflows; It's mostly done by volunteers, kernel developers using their spare time. Its features cover real use cases according to kernel developer needs. Basically it's mixing and matching the daily life of a typical kernel developer with kernel workflow recipes with some secret sauces. First recipe: A good GPU driver for my AMD laptop So, it's time to start the first recipe: A good GPU driver for my AMD laptop. Before starting any recipe we need to check the necessary ingredients and tools. So, let's check what you have at home. With kworkflow, you can use: kw device: to get information about the target machine, such as: CPU model, kernel version, distribution, GPU model, kw remote: to set the address of this machine for remote access kw config: you can configure kw with kw config. With this command you can basically select the tools, flags and preferences that kw will use to build and deploy a custom kernel in a target machine. You can also define recipients of your patches when sending it using kw send-patch. I'll explain more about each feature later in this presentation. kw kernel-config manager (or just kw k): to fetch the kernel .config file from a given machine, store multiple .config files, list and retrieve them according to your needs. Now, with all ingredients and tools selected and well portioned, follow the right steps to prepare your custom kernel! First step: Mix ingredients with kw build or just kw b kw b and its options wrap many routines of compiling a custom kernel. You can run kw b -i to check the name and kernel version and the number of modules that will be compiled and kw b --menu to change kernel configurations. You can also pre-configure compiling preferences in kw config regarding kernel

building. For example, target architecture, the name of the generated kernel image, if you need to cross-compile this kernel for a different system and which tool to use for it, setting different warning levels, compiling with CFlags, etc. Then you can just run kw b to compile the custom kernel for a target machine. Second step: Bake it with kw deploy or just kw d After compiling the custom kernel, we want to install it in the target machine. Check the name of the custom kernel built: 6.17.0-rc6 and with kw s SSH access the target machine and see it's running the kernel from the Debian distribution 6.16.7+deb14-amd64. As with building settings, you can also pre-configure some deployment settings, such as compression type, path to device tree binaries, target machine (remote, local, vm), if you want to reboot the target machine just after deploying your custom kernel, and if you want to boot in the custom kernel when restarting the system after deployment. If you didn't pre-configured some options, you can still customize as a command option, for example: kw d --reboot will reboot the system after deployment, even if I didn't set this in my preference. With just running kw d --reboot I have installed the kernel in a given target machine and rebooted it. So when accessing the system again I can see it was booted in my custom kernel. Third step: Time to taste with kw debug kw debug wraps many tools for validating a kernel in a target machine. We can log basic dmesg info but also tracking events and ftrace. With kw debug --dmesg --history we can grab the full dmesg log from a remote machine, if you use the --follow option, you will monitor dmesg outputs. You can also run a command with kw debug --dmesg --cmd="<my command>" and just collect the dmesg output related to this specific execution period. In the example, I'll just unload the amdgpu driver. I use kw drm --gui-off to drop the graphical interface and release the amdgpu for unloading it. So I run kw debug --dmesg --cmd="modprobe -r amdgpu" to unload the amdgpu driver, but it fails and I couldn't unload it. Cooking Problems Oh no! That custom kernel isn't tasting good. Don't worry, as in many recipes preparations, we can search on the internet to find suggestions on how to make it tasteful, alternative ingredients and other flavours according to your taste. With kw patch-hub you can search on the lore kernel mailing list for possible patches that can fix your kernel issue. You can navigate in the mailing lists, check series, bookmark it if you find it relevant and apply it in your local kernel tree, creating a different branch for tasting… oops, for testing. In this example, I'm opening the amd-gfx mailing list where I can find contributions related to the AMD GPU driver, bookmark and/or just apply the series to my work tree and with kw bd I can compile & install the custom kernel with this possible bug fix in one shot. As I changed my kw config to reboot after deployment, I just need to wait for the system to boot to try again unloading the amdgpu driver with kw debug --dmesg --cm=modprobe -r amdgpu. From the dmesg output retrieved by kw for this command, the driver was unloaded, the problem is fixed by this series and the kernel tastes good now. If I'm satisfied with the solution, I can even use kw patch-hub to access the bookmarked series and marking the checkbox that will reply the patch thread with a Reviewed-by tag for me. Second Recipe: Raspberry Pi 4 with Upstream Kernel As in all recipes, we need ingredients and tools, but with kworkflow you can get everything set as when changing scenarios in a TV show. We can use kw env to change to a different environment with all kw and kernel configuration set and also with the latest compiled kernel cached. I was preparing the first recipe for a x86 AMD laptop and with kw env --use RPI_64 I use the same worktree but moved to a different kernel workflow, now for Raspberry Pi 4 64 bits. The previous compiled kernel 6.17.0-rc6-mainline+ is there with 1266 modules, not the 6.17.0-rc6 kernel with 285 modules that I just built&deployed. kw build settings are also different, now I'm targeting a arm64 architecture with a cross-compiled kernel using aarch64-linu-gnu- cross-compilation tool and my kernel image calls kernel8 now. If you didn't plan for this recipe in advance, don't worry. You can create a new environment with kw env --create RPI_64_V2 and run kw init --template to start preparing your kernel recipe with the mirepoix ready. I mean, with the basic ingredients already cut… I mean, with the kw configuration set from a template. And you can use kw remote to set the IP address of your target machine and kw kernel-config-manager to fetch/retrieve the .config file from your target machine. So just run kw bd to compile and install a upstream kernel for

Raspberry Pi 4. Third Recipe: The Mainline Kernel Ringing on my Steam Deck (Live Demo) Let’s show you how easy is to build, install and test a custom kernel for Steam Deck with Kworkflow. It’s a live demo, but I also recorded it because I know the risks I’m exposed to and something can go very wrong just because of reasons :) Report: how was the live demo For this live demo, I took my OLED Steam Deck to the stage. I explained that, if I boot mainline kernel on this device, there is no audio. So I turned it on and booted the mainline kernel I’ve installed beforehand. It was clear that there was no typical Steam Deck startup audio when the system was loaded. As I started the demo in the kw environment for Raspberry Pi 4, I first moved to another environment previously used for Steam Deck. In this STEAMDECK environment, the mainline kernel was already compiled and cached, and all settings for accessing the target machine, compiling and installing a custom kernel were retrieved automatically. My live demo followed these steps: With kw env --use STEAMDECK, switch to a kworkflow environment for Steam Deck kernel development. With kw b -i, shows that kw will compile and install a kernel with 285 modules named 6.17.0-rc6-mainline-for-deck. Run kw config to show that, in this environment, kw configuration changes to x86 architecture and without cross-compilation. Run kw device to display information about the Steam Deck device, i.e. the target machine. It also proves that the remote access - user and IP - for this Steam Deck was already configured when using the STEAMDECK environment, as expected. Using git am, as usual, apply a hot fix on top of the mainline kernel. This hot fix makes the audio play again on Steam Deck. With kw b, build the kernel with the audio change. It will be fast because we are only compiling the affected files since everything was previously done and cached. Compiled kernel, kw configuration and kernel configuration is retrieved by just moving to the “STEAMDECK” environment. Run kw d --force --reboot to deploy the new custom kernel to the target machine. The --force option enables us to install the mainline kernel even if mkinitcpio complains about missing support for downstream packages when generating initramfs. The --reboot option makes the device reboot the Steam Deck automatically, just after the deployment completion. After finishing deployment, the Steam Deck will reboot on the new custom kernel version and made a clear resonant or vibrating sound. [Hopefully] Finally, I showed to the audience that, if I wanted to send this patch upstream, I just needed to run kw send-patch and kw would automatically add subsystem maintainers, reviewers and mailing lists for the affected files as recipients, and send the patch to the upstream community assessment. As I didn’t want to create unnecessary noise, I just did a dry-run with kw send-patch -s --simulate to explain how it looks. What else can kworkflow already mix & match? In this presentation, I showed that kworkflow supported different kernel development workflows, i.e., multiple distributions, different bootloaders and architectures, different target machines, different debugging tools and automatize your kernel development routines best practices, from development environment setup and verifying a custom kernel in bare-metal to sending contributions upstream following the contributions-by-e-mail structure. I exemplified it with three different target machines: my ordinary x86 AMD laptop with Debian, Raspberry Pi 4 with arm64 Raspbian (cross-compilation) and the Steam Deck with SteamOS (x86 Arch-based OS). Besides those distributions, Kworkflow also supports Ubuntu, Fedora and PopOS. Now it’s your turn: Do you have any secret recipes to share? Please share with us via kworkflow. Useful links Talk Recording of Kworkflow at Kernel Recipes 2025 on Igalia’s Channel Talk Abstract, Recording and Slide Deck of Kworkflow at Kernel Recipes 2025 on Kernel Recipes Website Talk Slide Deck for Download with some Videos instead of GIFs

- Mike Blumenkrantz: Hibernate On (2025/10/31 00:00)
  Take A Break We’ve reached Q4 of another year, and after the mad scramble that has been crunch-time over the past few weeks, it’s time for SGC to once again retire into a deep, restful sleep. 2025 saw a lot of ground covered: NVK-Zink synergy Continued Rusticl improvements Viewperf perf and general CPU overhead reduction Tiler GPU perf Mesh shaders apitrace perf More GL extensions released than any other year this decade It’s been a real roller coaster ride of a year as always, but I can say authoritatively that fans of the blog, you need to take care of

yourselves. You need to use this break time wisely. Rest. Recover. Train your bodies. Travel and broaden your horizons. Invest in night classes to expand your minds. You are not prepared for the insanity that will be this blog in 2026.

- Mike Blumenkrantz: Apitrace Goes Vroom (2025/10/27 00:00)
  First Time Today marks the first post of a type that I've wanted to have for a long while: a guest post. There are lots of graphics developers who work on cool stuff and don't want to waste time setting up blogs, but with enough cajoling they will write a single blog post. If you're out there thinking you just did some awesome work and you want the world to know the grimy, gory details, let me know. The first victimrecipient of this honor is an individual famous for small and extremely sane endeavors such as descriptor buffers in Lavapipe, ray tracing in Lavapipe, and sparse support in Lavapipe. Also wrangling ray tracing for RADV. Below is the debut blog post by none other than Konstantin Seurer. What is apitrace? Apitrace is a powerful tool for capturing and replaying traces of GL and DX applications. The problem is that it is not really suitable for performance testing. This blog post is about implementing a faster method for replaying traces. About six weeks ago, Mike asked me if I wanted to work on this. [6:58:58 pm] <zmike> on the topic of traces [6:59:08 pm] <zmike> I have a longer-term project that could use your expertise [6:59:19 pm] <zmike> it's low work but high complexity [7:00:12 pm] <zmike> specifically I would like apitrace to be able to competently output C code from traces and to have this functionality merged upstream low work Sure. The state of glretrace This first obvious step was measuring how glretrace currently performs. Mike kindly provided a couple of traces from his personal collection, and I immediately timed a trace of the only relevant OpenGL game: $ time ./glretrace -b minecraft-perf.trace /Users/Cortex/Downloads/graalvm-jdk-23.0.1+11.1/bin/java Rendered 1261 frames in 10.4269 secs, average of 120.937 fps real 0m10.554s user 0m12.938s sys 0m2.712s This looks fine, but I have no idea how fast the application is supposed to run. Running the same trace with perf reveals that there is room for improvement. 2/3 of frametime is spent parsing the trace. Implementation An apitrace trace stores API call information in an object-oriented style. This makes basic codegen really easy because the objects map directly to the generated C/C++ code. However, not all API calls are made equal, and the countless special cases that I needed to handle are what made this project take so long. glretrace has custom implementations for WSI API calls, and it would be a shame not to use them. The easiest way of doing that is generating a shared library instead of an executable and having glretrace load it. The shared library can then provide a bunch of callbacks for the call sequences we can do codegen for and Call objects for everything else. Besides WSI, there are also arguments and return values that need special treatment. OpenGL allows the application to create all kinds of objects that are represented using IDs. Those IDs are assigned by the driver, and they can be different during replay. glretrace remaps them using std::maps which have non-trivial overhead. I initially did that as well for the codegen to get things up and running, but it is actually possible to emit global variables and have most of the remapping run logic during codegen. Data streaming With the main replay overhead being taken care of, a major amount of replay time is now spent loading texture and buffer data. In large traces, there can also be >10GiB of data, so loading everything upfront is not an option. I decided to create one thread for reading the data file and nproc decompression threads. The read thread will wait if enough data has been loaded to limit memory usage. Decompression threads are needed because decompression is slower than reading the compressed data. codegen in action The results speak for themselves: $ ./glretrace --generate-c minecraft-perf minecraft-perf.trace /Users/Cortex/Downloads/graalvm-jdk-23.0.1+11.1/bin/java Rendered 0 frames in 79.4072 secs, average of 0 fps $ cd minecraft-perf $ echo "Invoke the superior build tool" $ meson build --buildtype release $ ninja -Cbuild $ time ../glretrace build/minecraft-perf.so info: Opening 'minecraft-perf.so'... (0.00668795 secs) warning: Waited 0.0461142 secs for data (sequence = 19) Rendered 1261 frames in 5.19587 secs, average of 242.693 fps real 0m5.415s user 0m5.429s sys 0m4.983s Nice. Looking at perf most CPU time is now spent in driver code or streaming

binary data for stuff like textures on a separate thread. If you are interested in trying this out yourself, feel free to build the upstream PR and report on bugs unintended features. It would also be nice to have DX support in the future, but that will be something for the dxvk developers unless I need something to procrastinate from doing RT work. - Konstantin

- Simon Ser: Status update, October 2025 (2025/10/15 22:00)
  Hi! I skipped last month's status update because I hadn't collected a lot of interesting updates and I've dedicated my time to writing an announcement for the first vali release. Earlier this month, I've taken the train to Vienna to attend XDC 2025. The conference was great, I really enjoyed discussing face-to-face with open-source graphics folks I usually only interact with online, and meeting new awesome people! Since I'm part of the X.Org Foundation board, it was also nice to see the fruit of our efforts. Many thanks to all organizers! We've discussed many interesting topics: a new API for 2D acceleration hardware, adapting the Wayland linux-dmabuf protocol to better support multiple GPUs, some ways to address current Wayback limitations, ideas to improve libliftoff, Vulkan use in Wayland compositors, and a lot more. On the wlroots side, I've worked on a patch to fallback to the renderer to apply gamma LUTs when the KMS driver doesn't support them (this also paves the way for applying color transforms in KMS). Félix Poisot has updated wlroots to support the gamma 2.2 transfer function and use it by default. llyyr has added support for the BT.1886 transfer function and fixed direct scanout for client using the gamma 2.2 transfer function. I've sent a patch to add support for DisplayID v2 CTA-861 data blocks, required for handling some HDR screens. I've reviewed and merged a bunch of gamescope patches to avoid protocol errors with the color management protocol, fix nested mode under a Vulkan compositor, fix a crash on VT switch and modernize dependencies. I've worked a bit on drm_info too. I've added a JSON schema to describe the shape of the JSON objects, made it so EDIDs are included in the JSON output as base64-encoded strings, and added the EDID make/model/serial + bus info to the pretty-printed output. delthas has added soju support for user metadata, introduced a new work-in-progress metadata key to block users, and made it so soju cancels Web Push notifications if a client marks a message as read (to avoid opening notifications for a very short time when actively chatting with another user). Markus Cisler has revamped Goguma's message bubbles: they look much better now! See you next month!
- Sebastian Wick: SO_PEERPIDFD Gets More Useful (2025/10/10 15:04)
  A while ago I wrote about the limited usefulness of SO_PEERPIDFD. for authenticating sandboxed applications. The core problem was simple: while pidfds gave us a race-free way to identify a process, we still had no standardized way to figure out what that process actually was - which sandbox it ran in, what application it represented, or what permissions it should have. The situation has improved considerably since then. cgroup xattrs Cgroups now support user extended attributes. This feature allows arbitrary metadata to be attached to cgroup inodes using standard xattr calls. We can change flatpak (or snap, or any other container engine) to create a cgroup for application instances it launches, and attach metadata to it using xattrs. This metadata can include the sandboxing engine, application ID, instance ID, and any other information the compositor or D-Bus service might need. Every process belongs to a cgroup, and you can query which cgroup a process belongs to through its pidfd - completely race-free. Standardized Authentication Remember the complexity from the original post? Services had to implement different lookup mechanisms for different sandbox technologies: For flatpak: look in /proc/$PID/root/.flatpak-info For snap: shell out to snap routine portal-info For firejail: no solution … All of this goes away. Now there's a single path: Accept a connection on a socket Use SO_PEERPIDFD to get a pidfd for the client Query the client's cgroup using the pidfd Read the cgroup's user xattrs to get the sandbox metadata This works the same way regardless of which sandbox engine launched the application. A Kernel Feature, Not a systemd One It's worth emphasizing: cgroups are a Linux kernel feature. They have no dependency on systemd or any other userspace component. Any process can manage cgroups and attach xattrs to

them. The process only needs appropriate permissions and is restricted to a subtree determined by the cgroup namespace it is in. This makes the approach universally applicable across different init systems and distributions. To support non-Linux systems, we might even be able to abstract away the cgroup details, by providing a varlink service to register and query running applications. On Linux, this service would use cgroups and xattrs internally. Replacing Socket-Per-App The old approach - creating dedicated wayland, D-Bus, etc. sockets for each app instance and attaching metadata to the service which gets mapped to connections on that socket - can now be retired. The pidfd + cgroup xattr approach is simpler: one standardized lookup path instead of mounting special sockets. It works everywhere: any service can authenticate any client without special socket setup. And it's more flexible: metadata can be updated after process creation if needed. For compositor and D-Bus service developers, this means you can finally implement proper sandboxed client authentication without needing to understand the internals of every container engine. For sandbox developers, it means you have a standardized way to communicate application identity without implementing custom socket mounting schemes.
- Mike Blumenkrantz: Mesh Shaders In The Current Year (2025/10/09 00:00)
  It Happened. Just a quick post to confirm that the OpenGL/ES Working Group has signed off on the release of GL_EXT_mesh_shader. Credits This is a monumental release, the largest extension shipped for GL this decade, and the culmination of many, many months of work by AMD. In particular we all need to thank Qiang Yu (AMD), who spearheaded this initiative and did the vast majority of the work both in writing the specification and doing the core mesa implementation. Shihao Wang (AMD) took on the difficult task of writing actual CTS cases (not mandatory for EXT extensions in GL, so this is a huge benefit to the ecosystem). Big thanks to both of you, and everyone else behind the scenes at AMD, for making this happen. Also we have to thank the nvidium project and its author, Cortex, for single-handedly pushing the industry forward through the power of Minecraft modding. Stay sane out there. Support Minecraft mod support is already underway, so expect that to happen "soon". The bones of this extension have already been merged into mesa over the past couple months. I opened a MR to enable zink support this morning since I have already merged the implementation. Currently, I'm planning to wait until either just before the branch point next week or until RadeonSI merges its support to merge the zink MR. This is out of respect: Qiang Yu did a huge lift for everyone here, and ideally AMD's driver should be the first to be able to advertise that extension to reflect that. But the branchpoint is coming up in a week, and SGC will be going into hibernation at the end of the month until 2026, so this offer does have an expiration date. In any case, we're done here.
- Simon Ser: Announcing vali, a C library for Varlink (2025/10/03 22:00)
  In the past months I've been working on vali, a C library for Varlink. Today I'm publishing the first vali release! I'd like to explain how to use it for readers who aren't especially familiar with Varlink, and describe some interesting API design decisions. What is Varlink anyways? Varlink is a very simple Remote Procedure Call (RPC) protocol. Clients can call methods exposed by services (ie, servers). To call a method, a client sends a JSON object with its name and parameters over a Unix socket. To reply to a call, a service sends a JSON object with response parameters. That's it. Here's an example request with a bar parameter containing an integer: { "method": "org.example.service.Foo", "parameters": { "bar": 42 } } And here's an example response with a baz parameter containing a list of strings: { "parameters": { "baz": ["hello", "world"] } } Varlink also supports calls with no reply or with multiple replies, but let's leave this out of the picture for simplicity's sake. Varlink services can describe the methods they implement with an interface definition file. method Foo(bar: int) -> (baz: []string) Coming from the Wayland world, I love generating code from specification files. This removes all of the manual encoding/decoding boilerplate and is more type-safe. Unfortunately the official libvarlink library doesn't support code generation (and is not actively maintained anymore), so I've decided to write my own. vali is the

result! vali without code generation To better understand the benefits of code generation and vali design decisions, let’s take a minute to have a look at what usage without code generation looks like. A client first needs to connect via vali_client_connect_unix(), then call vali_client_call() with a JSON object containing input parameters. It’ll get back a JSON object containing output parameters, which needs to be parsed. struct vali_client *client = vali_client_connect_unix("/run/org.example.service"); if (client == NULL) { fprintf(stderr, "Failed to connect to service\n"); exit(1); } struct json_object *in = json_object_new_object(); json_object_object_add(in, "bar", json_object_new_int(42)); struct json_object *out = NULL; if (!vali_client_call(client, "org.example.service.Foo", in, &out, NULL)) { fprintf(stderr, "Foo request failed\n"); exit(1); } struct json_object *baz = json_object_object_get(out, "baz"); for (size_t i = 0; i < json_object_array_length(baz); i++) { struct json_object *item = json_object_array_get_idx(baz, i); printf("%s\n", json_object_get_string(item)); } This is a fair amount of boilerplate. In case of a type mismatch, the client will silently print nothing, which isn’t ideal. The last parameter of vali_client_call() is an optional struct vali_error *: if set to a non-NULL pointer and the service replies with an error, the struct is populated, otherwise it’s zero’ed out: struct vali_error err; if (!vali_client_call(client, "org.example.service.Foo", in, &out, &err)) { if (err.name != NULL) { fprintf(stderr, "Foo request failed: %s\n", err.name); } else { fprintf(stderr, "Foo request failed: internal error\n"); } vali_error_finish(&err); exit(1); } How does the service side look like? A service first calls vali_service_create() to initialize a fresh service, defines a callback to be invoked when a Varlink call is performed by a client via vali_service_set_call_handler(), and sets up a Unix socket via vali_service_listen_unix(). Let’s demonstrate how a service accesses a shared state by printing the number of calls done so far when the callback is invoked. The callback needs to end the call with vali_service_call_close_with_reply(). void handle_call(struct vali_service_call *call, void *user_data) { int *call_count_ptr = user_data; (*call_count_ptr)++; printf("Received %d-th client call\n", *call_count_ptr); struct json_object *baz = json_object_new_array(); json_object_array_add(baz, json_object_new_string("hello")); json_object_array_add(baz, json_object_new_string("world")); struct json_object *params = json_object_new_object(); json_object_object_add(params, "baz", baz); vali_service_call_close_with_reply(call, params); } int main(int argc, void *argv[]) { int call_count = 0; struct vali_service_call_handler handler = { .func = handle_call, .user_data = &call_count, }; struct vali_service *service = vali_service_create(); vali_service_set_call_handler(service, &handler); vali_service_listen_unix(service, "/run/org.example.ftl"); while (vali_service_dispatch(service)); return 0; } In a prior iteration of the API, the callback would return the reply JSON object. This got changed to vali_service_call_close_with_reply() so that services can handle a call asynchronously. If a service needs some time to reply (e.g. because it needs to send data over a network, or perform a long computation), it can give back control to its event loop so that other clients are not blocked, and later call vali_service_call_close_with_reply() from another callback. Why bundle the callback and the user data pointer together in a struct, rather than pass them as two separate parameters to vali_service_set_call_handler()? The answer is two-fold: Conceptually, the user data pointer is tied to the callback. Other programming languages with support for lambdas just capture variables. Standard C doesn’t have lambdas, and the user data pointer is just a way to pass state to the callback. Bundling the callback and the user data pointer together as a single fat pointer unlocks more ergonomic and safer APIs: a function can return a single struct vali_service_call_handler without making the caller manipulate two separate variables to pass it down to vali_service_set_call_handler() (and risk mixing them up in case there are multiple). This design makes wrapping a handler much easier (to create middlewares and routers, more on that below). This all might sound familiar to folks who’ve written an HTTP server: indeed, struct vali_service_call_handler is inspired from Go’s net/http.Handler. Client side with code generation Given the method definition from the article introduction, vali generates the following client function: struct example_Foo_in { int bar; }; struct example_Foo_out { char **baz; size_t baz_len; }; bool example_Foo(struct vali_client *client, const struct example_Foo_in *in, struct example_Foo *out, struct vali_error *err); It can be used this way to send the JSON request we’ve seen earlier: struct vali_client *client = vali_client_connect_unix("/run/org.example.service"); if (client == NULL) { fprintf(stderr, "Failed to connect to service\n"); exit(1); } const struct example_Foo_in in = { .bar = 42, }; struct example_Foo_out out; if (!example_Foo(client, &in, &out, NULL)) { fprintf(stderr, "Foo request failed\n"); exit(1); } for (size_t i = 0; i < out.baz_len; i++) { printf("%s\n", out.baz[i]); } example_Foo_out_finish(&out); Why does vali generates these twin structs, one for input parameters and the other for output parameters, instead of passing all parameters as function arguments? This does make calls slightly more verbose, but this has a few upsides: There is a clear split between input and output parameters, instead of having a variable number of function arguments for each. No need for the API user to remember where input parameters end and when output parameters begin, especially when there are a lot of these. On the wire and in the interface definition file, input and output parameters are objects. vali always generates structs for all objects. This is more consistent. If a new backwards-compatible version of the interface is published, the newly generated code is also backwards-compatible: old callers will still compile and work fine against the newly generated code. For instance, if a new optional field is added to the input parameters, it will naturally left as NULL by the caller when re-generating the code (because omitted fields are zero-initialized in C). Service side with code generation The service side is more complicated because it needs to handle multiple connections concurrently and needs to be asynchronous. Being asynchronous is important to not block other clients when processing a call. The generator for the service code spits out one struct per method and a function to send a reply (and destroy the call): struct example_Foo_service_call { struct vali_service_call *base; }; void example_Foo_close_with_reply(struct example_Foo_service_call call, const struct example_Foo_out *params); The per-call struct wrapping the struct vali_service_call * makes functions sending replies strongly tied to a particular call, and provides type safety: a Foo reply cannot be sent to a Bar call. Additionally, the generator also provides a handler struct with one callback per method, and a function to obtain a generic handler from an interface handler: struct example_handler { void (*Foo)(struct example_Foo_service_call call, const struct example_in *in); }; struct vali_service_call_handler example_get_call_handler(const struct example_handler *handler); To use all of these toys, a service implementation can define a handler for the Foo method, then feed the result of example_get_call_handler() to vali_service_set_call_handler(): static void handle_foo(struct example_Foo_service_call call, const struct example_Foo_in *in) { printf("Foo called with bar=%d\n", in->bar); example_Foo_close_with_reply(call, &(const struct example_Foo_out){ .baz = (char *[]){ "hello", "world" }, .baz_len = 2, }); } static const struct example_handler example_handler = { .Foo = handle_foo, }; int main(int argc, void *argv[]) { struct vali_service *service = vali_service_create(); vali_service_set_call_handler(service, example_get_call_handler(&example_handler)); vali_service_listen_unix(service, "/run/org.example.ftl"); while (vali_service_dispatch(service)); } Service registry Some more elaborated services might want to implement more than a single interface. Additionally, services might want to add support for the org.varlink.service interface, which provides introspection: a client can query metadata about the service (e.g. service name, version) and the definition of each interface. vali makes this easy thanks to struct vali_registry. A service can initialize a new registry via vali_registry_create(), then register each interface by passing its definition and handler to vali_registry_add(). The generated code exposes the interface definition as an example_interface constant. Finally, the registry can be wired up to the struct vali_service by feeding the result of vali_registry_get_call_handler() to vali_service_set_call_handler(). const struct vali_registry_options registry_options = { .vendor = "emersion", .product = "example", .version = "1.0.0", .url = "https://example.org", }; struct vali_registry *registry = vali_registry_create(&registry_options); vali_registry_add(registry, &example_interface, example_get_call_handler(&example_handler)); vali_registry_add(registry, &another_example_interface, another_example_get_call_handler(&another_example_handler)); struct vali_service *service = vali_service_create();

vali_service_set_call_handler(service, vali_registry_get_call_handler(registry)); vali_service_listen_unix(service, "/run/org.example.ftl"); while (vali_service_dispatch(service)); This is where the struct vali_service_call_handler fat pointer really shines: the wire-level struct vali_service and the higher-level registry can stay entirely separate. struct vali_service invokes the registry's handler, then the registry is responsible for routing the call to the correct interface-specific handler. The registry's internal state remains hidden away in the handler's opaque user data pointer. A complete client and service example is available in vali's example/ directory. What's next? I plan to leverage vali in the next version of the kanshi Wayland output management daemon. We've discussed about async on the service side above, but we haven't discussed async on the client side. That can be useful too, especially when a client needs to juggle with multiple sockets, and is still a TODO. Something I'm still unhappy about is the lack of const fields generated structs. Let's have a look at the struct for output parameters given above: struct example_Foo_out { char **baz; size_t baz_len; }; If a service has a bunch of const char * variables it wants to send as part of the reply, it needs to cast them to char * or strdup() them. None of these options are great. static const char hiya[] = "hiya"; static void handle_foo(struct example_Foo_service_call call, const struct example_Foo_in *in) { example_Foo_close_with_reply(call, &(const struct example_Foo_out){ // Type error: implicit cast from "const char *" to "char *" .baz = (char *[]){ hiya }, .baz_len = 1, }); } On the other hand, making all struct fields const would be cumbersome when dynamically constructing nested structs in replies, and would be a bit of a lie when passing a reply to example_Foo_out_finish() (that function frees all fields). Generating two structs (one const, one not) is not an option since types are shared between client and service, and some types can be referenced from both a call input and another call's output. Ideally, C would provide a way to propagate const-ness to fields, but that's not a thing. Oh well, that's life. If you need an IPC mechanism for your tool, please consider giving vali a shot! Feel free to reach out to report any bugs, questions or suggestions.

- Iago Toral: XDC 2025 (2025/10/03 06:58)
  It has been a while since my last post, I know. Today I just want to thank Igalia for continuing to give me and many other Igalians the opportunity to attend XDC. I had a great time in Vienna where I was able to catch up with other Mesa developers (including Igalians!) I rarely have the opportunity to see face to face. It is amazing to see how Mesa continues to gain traction and interest year after year, seeing more actors and vendors getting involved in one way or another… the push for open source drivers in the industry is real and it is fantastic to see it happening. I'd also like to thank the organization, I know all the work that goes into making these things happen, so big thanks to everyone who was involved, and to the speakers, the XDC program is getting better every year. Looking forward to next year already
- Hans de Goede: Fedora 43 will ship with FOSS Meteor, Lunar and Arrow Lake MIPI camera support (2025/09/30 18:55)
  Good news the just released 6.17 kernel has support for the IPU7 CSI2 receiver and the missing USBIO drivers have recently landed in linux-next. I have backported the USBIO drivers + a few other camera fixes to the Fedora 6.17 kernel.I've also prepared an updated libcamera-0.5.2 Fedora package with support for IPU7 (Lunar Lake) CSI2 receivers as well as backporting a set of upstream SwStats and AGC fixes, fixing various crashes as well as the bad flicker MIPI camera users have been hitting with libcamera 0.5.2.Together these 2 updates should make Fedora 43's FOSS MIPI camera support work on most Meteor Lake, Lunar Lake and Arrow Lake laptops!If you want to give this a try, install / upgrade to Fedora 43 beta and install all updates. If you've installed rpmfusion's binary IPU6 stack please run: sudo dnf remove akmod-intel-ipu6 'kmod-intel-ipu6*'to remove it as it may interfere with the FOSS stack and finally reboot. Please first try with qcam:sudo dnf install libcamera-qcamqcamwhich only tests libcamera and after that give apps which use the camera through pipewire a try like gnome's "Camera" app (snapshot) or video-conferencing in Firefox.Note snapshot on Lunar Lake triggers a bug in the LNL Vulkan code, to avoid this start snapshot from a terminal

with:GSK_RENDERER=gl snapshotIf you have a MIPI camera which still does not work please file a bug following these instructions and drop me an email with the bugzilla link at hansg@kernel.org. comments

- Sebastian Wick: XDG Intents Updates (2025/09/24 16:57)
Andy Holmes wrote an excellent overview of XDG Intents in his “Best Intentions” blog post, covering the foundational concepts and early proposals. Unfortunately, due to GNOME Foundation issues, this work never fully materialized. As I have been running into more and more cases where this would provide a useful primitive for other features, I tried to continue the work. The specifications have evolved as I worked on implementing them in glib, desktop-file-utils and ptyxis. Here’s what’s changed: Intent-Apps Specification Andy showed this hypothetical syntax for scoped preferences: [Default Applications] org.freedesktop.Thumbnailer=org.gimp.GIMP org.freedesktop.Thumbnailer[image/svg+xml]=org.gnome.Loupe;org.gimp.GIMP We now use separate groups instead: [Default Applications] org.freedesktop.Thumbnailer=org.gimp.GIMP [org.freedesktop.Thumbnailer] image/svg+xml=org.gnome.Loupe;org.gimp.GIMP This approach creates a dedicated group for each intent, with keys representing the scopes. This way, we do not have to abuse the square brackets which were meant for translatable keys and allow only very limited values. The updated specification also adds support for intent.cache files to improve performance, containing up-to-date lists of applications supporting particular intents and scopes. This is very similar to the already existing cache for MIME types. The update-desktop-database tool is responsible for keeping the cache up-to-date. This is implemented in glib!4797, desktop-file-utils!27, and the updated specification is in xdg-specs!106. Terminal Intent Specification While Andy mentioned the terminal intent as a use case, Zander Brown tried to upstream the intent in xdg-specs!46 multiple years ago. However, because it depended on the intent-apps specification, it unfortunately never went anywhere. With the fleshed-out version of the intent-apps specification, and an implementation in glib, I was able to implement the terminal-intent specification in glib as well. With some help from Christian, we also added support for the intent in the ptyxis terminal. This revealed some shortcomings in the proposed D-Bus interface. In particular, when a desktop file gets activated with multiple URIs, and the Exec line in the desktop entry only indicates support for a limited number of URIs, multiple commands need to be launched. To support opening those commands in a single window but in multiple tabs in the terminal emulator, for example, those multiple commands must be part of a single D-Bus method call. The resulting D-Bus interface looks like this: <interface name="org.freedesktop.Terminal1"> <method name="LaunchCommand"> <arg type='aa{sv}' name='commands' direction='in' /> <arg type='ay' name='desktop_entry' direction='in' /> <arg type='a{sv}' name='options' direction='in' /> <arg type='a{sv}' name='platform_data' direction='in' /> </method> </interface> This is implemented in glib!4797, ptyxis!119 and the updated specification is in xdg-specs!107. Deeplink Intent Andy’s post discussed a generic “org.freedesktop.UriHandler” with this example: [org.freedesktop.UriHandler] Supports=wise.com; Patterns=https://*.wise.com/link?urn=urn%3Awise%3Atransfers; The updated specification introduces a specific org.freedesktop.handler.Deeplink1 intent where the scheme is implicitly http or https and the host comes from the scope (i.e., the Supports part). The pattern matching is done on the path alone: [org.freedesktop.handler.Deeplink1] Supports=example.org;extensions.gnome.org example.org=/login;/test/a?a extensions.gnome.org=/extension/*/*/install;/extension/*/*/uninstall This allows us to focus on deeplinking alone and allows the user to set the order of handlers for specific hosts. In this example, the app would handle the URIs http://example.org/login, http://example.org/test/aba, http://extensions.gnome.org/extension/123456/BestExtension/install and so on. There is a draft implementation in glib!4833 and the specification is in xdg-specs!109. Deeplinking Issues and Other Handlers I am still unsure about the Deeplink1 intent. Does it make sense to allow schemes other than http and https? If yes, how should the priority of applications be determined when opening a URI? How

complex does the pattern matching need to be? Similarly, should we add an org.freedesktop.handler.Scheme1 intent? We currently abuse MIME handlers for this, so it seems like a good idea, but then we need to take backwards compatibility into account. Maybe we can modify update-desktop-database to add entries from org.freedesktop.handler.Scheme1 to mimeapps.list for that? If we go down that route, is there a reason not to also do the same for MIME handlers and add an org.freedesktop.handler.Mime1 intent for that purpose with the same backwards compatibility mechanism? Deeplinking to App Locations While working on this, I noticed that we are not great at allowing linking to locations in our apps. For example, most email clients do not have a way to link to a specific email. Most calendars do not allow referencing a specific event. Some apps do support this. For example, Zotero allows linking to items in the app with URIs of the form zotero://select/items/0_USN95MJC. Maybe we can improve on this? If all our apps used a consistent scheme and queries (for example xdg-app-org.example.appid:/some/path/in/the/app?name=Example), we could render those links differently and finally have a nice way to link to an email in our calendar. This definitely needs more thought, but I do like the idea. Security Considerations Allowing apps to describe more thoroughly which URIs they can handle is great, but we also live in a world where security has to be taken into account. If an app wants to handle the URI https://bank.example.org, we better be sure that this app actually is the correct banking app. This unfortunately is not a trivial issue, so I will leave it for the next time.

- Sebastian Wick: Integrating libdex with GDBus (2025/09/18 18:58)
Writing asynchronous code in C has always been a challenge. Traditional callback-based approaches, including GLib’s async/finish pattern, often lead to the so-called callback hell that’s difficult to read and maintain. The libdex library offers a solution to this problem, and I recently worked on expanding the integration with GLib’s GDBus subsystem. The Problem with the Sync and Async Patterns Writing C code involving tasks which can take non-trivial amount of time has traditionally required choosing between two approaches: Synchronous calls - Simple to write but block the current thread Asynchronous callbacks - Non-blocking but result in callback hell and complex error handling Often the synchronous variant is chosen to keep the code simple, but in a lot of cases, blocking for potentially multiple seconds is not acceptable. Threads can be used to prevent the other threads from blocking, but it creates parallelism and with it the need for locking. It also can potentially create a huge amount of threads which mostly sit idle. The asynchronous variant has none of those problems, but consider a typical async D-Bus operation in traditional GLib code: static void on_ping_ready (GObject *source_object, GAsyncResult *res, gpointer data) { g_autofree char *pong = NULL; if (!dex_dbus_ping_pong_call_ping_finish (DEX_BUS_PING_PONG (source_object), &pong, res, NULL)) return; // handle error g_print ("client: %s\n", pong); } static void on_ping_pong_proxy_ready (GObject *source_object, GAsyncResult *res, gpointer data) { DexDbusPingPong *pp dex_dbus_ping_pong_proxy_new_finish (res, NULL); if (!pp) return; // Handle error dex_dbus_ping_pong_call_ping (pp, "ping", NULL, on_ping_ready, NULL); } This pattern becomes unwieldy quickly, especially with multiple operations, error handling, shared data and cleanup across multiple callbacks. What is libdex? Dex provides Future-based programming for GLib. It provides features for application and library authors who want to structure concurrent code in an easy to manage way. Dex also provides Fibers which allow writing synchronous looking code in C while maintaining the benefits of asynchronous execution. At its core, libdex introduces two key concepts: Futures: Represent values that will be available at some point in the future Fibers: Lightweight cooperative threads that allow writing synchronous-looking code that yields control when waiting for asynchronous operations Futures alone already simplify dealing with asynchronous code by specefying a call chain (dex_future_then(), dex_future_catch(), and dex_future_finally()), or even more elaborate flows (dex_future_all(), dex_future_all_race(), dex_future_any(), and dex_future_first()) at one place, without the typical callback hell. It still requires splitting things into a bunch of functions

and potentially moving data through them. static DexFuture * lookup_user_data_cb (DexFuture *future, gpointer user_data) { g_autoptr(MyUser) user = NULL; g_autoptr(GError) error = NULL; // the future in this cb is already resolved, so this just gets the value // no fibers involved user = dex_await_object (future, &error); if (!user) return dex_future_new_for_error (g_steal_pointer (&error)); return dex_future_first (dex_timeout_new_seconds (60), dex_future_any (query_db_server (user), query_cache_server (user), NULL), NULL); } static void print_user_data (void) { g_autoptr(DexFuture) future = NULL; future = dex_future_then (find_user (), lookup_user_data_cb, NULL, NULL); future = dex_future_then (future, print_user_data_cb, NULL, NULL); future = dex_future_finally (future, quit_cb, NULL, NULL); g_main_loop_run (main_loop); } The real magic of libdex however lies in fibers and the dex_await() function, which allows you to write code that looks synchronous but executes asynchronously. When you await a future, the current fiber yields control, allowing other work to proceed while waiting for the result. g_autoptr(MyUser) user = NULL; g_autoptr(MyUserData) data = NULL; g_autoptr(GError) error = NULL; user = dex_await_object (find_user (), &error); if (!user) return dex_future_new_for_error (g_steal_pointer (&error)); data = dex_await_boxed (dex_future_first (dex_timeout_new_seconds (60), dex_future_any (query_db_server (user), query_cache_server (user), NULL), NULL), &error); if (!data) return dex_future_new_for_error (g_steal_pointer (&error)); g_print ("%s", data->name); Christian Hergert wrote pretty decent documentation, so check it out! Bridging libdex and GDBus With the new integration, you can write D-Bus client code that looks like this: g_autoptr(DexDbusPingPong) *pp = NULL; g_autoptr(DexDbusPingPongPingResult) result = NULL; pp = dex_await_object (dex_dbus_ping_pong_proxy_new_future (connection, G_DBUS_PROXY_FLAGS_NONE, "org.example.PingPong", "/org/example/pingpong"), &error); if (!pp) return dex_future_new_for_error (g_steal_pointer (&error)); res = dex_await_boxed (dex_dbus_ping_pong_call_ping_future (pp, "ping"), &error); if (!res) return dex_future_new_for_error (g_steal_pointer (&error)); g_print ("client: %s\n", res->pong); This code is executing asynchronously, but reads like synchronous code. Error handling is straightforward, and there are no callbacks involved. On the service side, if enabled, method handlers will run in a fiber and can use dex_await() directly, enabling complex asynchronous operations within service implementations: static gboolean handle_ping (DexDbusPingPong *object, GDBusMethodInvocation *invocation, const char *ping) { g_print ("service: %s\n", ping); dex_await (dex_timeout_new_seconds (1), NULL); dex_dbus_ping_pong_complete_ping (object, invocation, "pong"); return G_DBUS_METHOD_INVOCATION_HANDLED; } static void dex_dbus_ping_pong_iface_init (DexDbusPingPongIface *iface) { iface->handle_ping = handle_ping; } pp = g_object_new (DEX_TYPE_PING_PONG, NULL); dex_dbus_interface_skeleton_set_flags (DEX_DBUS_INTERFACE_SKELETON (pp), DEX_DBUS_INTERFACE_SKELETON_FLAGS_HANDLE_METHOD_INVOCATIONS_IN_FIBER); This method handler includes a 1-second delay, but instead of blocking the entire service, it yields control to other fibers during the timeout. The merge request contains a complete example of a client and service communicating with each other. Implementation Details The integration required extending GDBus's code generation system. Rather than modifying it directly, the current solution introduces a very simple extension system to GDBus' code generation. The generated code includes: Future-returning functions: For every _proxy_new() and _call_$method() function, corresponding _future() variants are generated Result types: Method calls return boxed types containing all output parameters Custom skeleton base class: Generated skeleton classes inherit from DexDBusInterfaceSkeleton instead of GDBusInterfaceSkeleton, which implements dispatching method handlers in fibers Besides the GDBus code generation extension system, there are a few more changes required in GLib to make this work. This is not merged at the time of writing, but I'm confident that we can move this forward. Future Directions I hope that this work convinces more people to use libdex! We have a whole bunch of existing code bases which will have to stick with C in the foreseeable future, and libdex provides tools to make incremental improvements. Personally, I want to start using in in the xdg-desktop-portal project.

- Mike Blumenkrantz: Now We CAD (2025/09/16 00:00)
  Perf Must Increase. After my last post, I'm sure everyone was speculating about the forthcoming zink takeover of the CAD industry. Or maybe just wondering why I'm bothering with this at all. Well, the answer is simple: CAD performance is all performance. If I can improve FPS in viewperf, I'm decreasing CPU utilization in all apps, which is generally useful. As in the previous post, the catia section of viewperf was improved to a whopping 34fps against the reference driver (radeonsi) by eliminating a few hundred thousand atomic operations per frame. An interesting observation here is that while eliminating atomic operations in radeonsi does improve FPS there by ~5% (105fps), there is no bottlenecking, so this does not "unlock" further optimizations in the same way that it does for zink. I speculate this is because zink has radv underneath, which affects memory access across ccx in ways that do not affect radeonsi. In short: a rising tide lifts all ships in the harbor, but since zink was effectively a sunken ship, it is rising much more than the others. Even More Improvements Since that previous post, I and others have been working quietly in the background on other improvements, all of which have landed in mesa main already: A nice 35% improvement, largely from three MRs: zink draw optimizations zink vbo binding optimizations radv dynamic state optimizations That's right. In my quest to maximize perf, I have roped in veteran radv developer and part-time vacation enthusiast, Samuel Pitoiset. Because radv is slow. vkoverhead exists to target noticeably slow cases, and by harnessing the forbidden power of rewriting the whole driver, it was possible for a lone Frenchman to significantly reduce bottlenecking during draw emission. This Isn't Even My Final Form Obviously. I'm not about to say that I'll only stop when I reach performance parity, but the FPS can still go up. At this point, however, it's becoming less useful (in zink) to look at flamegraphs. There's only so much optimization that can be done once the code has been simplified to a certain extent, and eventually those optimizations will lead to obfuscated code which is harder to maintain. Thus, it's time to step back and look architecturally. What is the app doing? How does that reach the driver? Can it be improved? GALLIUM_TRACE is a great tool for this, as it logs the API stream as it reaches the backend driver, and there are parser tools to convert the output XML to something readable. Let's take a look at a small cross-section of the trace: pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10043], [is_user_buffer = 0, buffer_offset = 7440, buffer.resource = resource_10043]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10044], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10045], [is_user_buffer = 0, buffer_offset = 7632, buffer.resource = resource_10045]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10046], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10047], [is_user_buffer = 0, buffer_offset = 7680, buffer.resource = resource_10047]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10048], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10049], [is_user_buffer = 0, buffer_offset = 7656, buffer.resource = resource_10049]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1,

min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10050], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10051], [is_user_buffer = 0, buffer_offset = 7752, buffer.resource = resource_10051]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10052], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10053], [is_user_buffer = 0, buffer_offset = 7800, buffer.resource = resource_10053]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10054], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10055], [is_user_buffer = 0, buffer_offset = 7968, buffer.resource = resource_10055]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10056], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10057], [is_user_buffer = 0, buffer_offset = 7968, buffer.resource = resource_10057]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10058], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10059], [is_user_buffer = 0, buffer_offset = 8136, buffer.resource = resource_10059]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10060], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10061], [is_user_buffer = 0, buffer_offset = 8280, buffer.resource = resource_10061]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10062], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10063], [is_user_buffer = 0, buffer_offset = 8040, buffer.resource = resource_10063]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10064], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10065], [is_user_buffer = 0, buffer_offset = 7608, buffer.resource = resource_10065]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart =

0, restart_index = 0, index.resource = resource_10066], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) As expected, a huge chunk of the runtime is just set_vertex_buffers -> draw_vbo. Architecturally, this leads to a lot of unavoidably wasted cycles in drivers: set_vertex_buffers "binds" vertex buffers to the context and flags state updates draw_vbo checks all of the driver's update-able states, updates the flagged ones, and then emits draws But in the scenario where the driver can know ahead of time exactly what states will be updated, couldn't that yield an improvement? For example, bundling these two calls into a single draw call would eliminate: "binding" of vertex buffers vbo state update flagging draw-time validation calling multiple driver entrypoints In theory, it seems like this should be pretty good. And now that vertex buffer lifetimes have been reworked to use explicit ownership rather than garbage collection, it's actually possible to do this. The optimal site for the optimization would be in threaded-context, where similar types of draw merging are already occurring. The result looks something like this in a comparable trace: pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1141, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 163536, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 191032, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771328, buffer.resource = resource_29602]], draws = [[start = 1141, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1146, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 218528, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 246144, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771360, buffer.resource = resource_29602]], draws = [[start = 1146, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1151, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 273760, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 301496, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771392, buffer.resource = resource_29602]], draws = [[start = 1151, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1156, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 329232, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 357088, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771424, buffer.resource = resource_29602]], draws = [[start = 1156, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1161, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 384944, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 412920, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771456, buffer.resource = resource_29602]], draws = [[start = 1161, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1166, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 440896, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 468992, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771488, buffer.resource = resource_29602]], draws = [[start = 1166, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1171, max_index = 5, primitive_restart = 0, restart_index

= 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 497088, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 525304, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771520, buffer.resource = resource_29602]], draws = [[start = 1171, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1176, max_index = 11, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 553520, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 582000, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771552, buffer.resource = resource_29602]], draws = [[start = 1176, count = 11]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1187, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 610480, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 639080, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771584, buffer.resource = resource_29602]], draws = [[start = 1187, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1192, max_index = 6, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 667680, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 696424, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771616, buffer.resource = resource_29602]], draws = [[start = 1192, count = 6]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1198, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 725168, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 754032, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771648, buffer.resource = resource_29602]], draws = [[start = 1198, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1203, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 782896, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 811880, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771680, buffer.resource = resource_29602]], draws = [[start = 1203, count = 5]], num_draws = 1) It's more compact, which is nice, but how does the perf look? About another 40% improvement, now over 60fps: nearly double the endpoint of the last post. Huge. And this is driving ecosystem improvements which will affect other apps and games which don't even use zink. Stay winning, Open Source graphics.

- Dave Airlie (blogspot): radv takes over from AMDVLK (2025/09/15 19:08)
  AMD have announced the end of the AMDVLK open driver in favour of focusing on radv for Linux use cases.When Bas and I started radv in 2016, AMD were promising their own Linux vulkan driver, which arrived in Dec 2017. At this point radv was already shipping in most Linux distros. AMD strategy of having AMDVLK was developed via over the wall open source releases from internal closed development was always going to be a second place option at that point.When Valve came on board and brought dedicated developer power to radv, and the aco compiler matured, there really was no point putting effort into using AMDVLK which was hard to package and impossible to contribute to meaningfully for external developers.radv is probably my proudest contribution to the Linux ecosystem, finally disproving years of idiots saying an open source driver could never compete with a vendor provided driver, now it is the vendor provided driver.I think we will miss the open source PAL repo as a reference source and I hope AMD engineers can bridge that gap, but it's often hard to find workarounds you don't know exist to ask about them.

I'm also hoping AMD will add more staffing beyond the current levels especially around hardware enablement and workarounds.Now onwards to NVK victory :-)[1] https://github.com/GPUOpen-Drivers/AMDVLK/discussions/416

- Christian Schaller: More adventures in the land of AI and Open Source (2025/09/09 14:39)
I been doing a lot of work with AI recently, both as part of a couple of projects I am part of at work, but I have also taken a personal interest in understanding the current state and what is possible. My favourite AI tool currently is Claude.ai. Anyway I have a Prusa Core One 3D printer now that I also love playing with and one thing I been wanting to do is to print some multicolor prints with it. So the Prusa Core One is a single extruder printer, which means it only has 1 filament loaded at any given time. Other printers on the market, like the PrusaXL has 5 extruders, so it can have 5 filaments or colors loaded at the same time. Prusa Single Extruder Multimaterial setting The thing is that the Prusa Slicer (the slicer is the software that takes a 3d model and prepares the instructions for the printer based on that 3d model) got this feature called Single Extruder Multi Material. And while it is a process that wastes a lot of filament and takes a lot of manual intervention during the print, it does basically work. What I quickly discovered was that using this feature is non-trivial. First of all I had to manually add some G Code to the model to actually get it to ask me to switch filament for each color in my print, but the bigger issue is that the printer will ask you to change the color or filament, but you have no way of knowing which one to switch to, so for my model I had 15 filament changes and no simple way of knowing which order to switch in. So people where solving this among other things through looking through the print layer by layer and writing down the color changes, but I thought that this must be possible to automate with an application. So I opened Claude and started working on this thing I ended up calling Prusa Color Mate.. So the idea for the application was simple enough, have it analyze the project file, extract information about the order of color changes and display them for the user in a way that allows them to manually check of each color as its inserted. So I started off with doing a simple python script that would just print to the console. So it quickly turned out that the hard part of this project was to parse the input files and it was made worse by my ignorance. So what I learned the hard way is that if you store a project in Prusa Slicer it will use this format called 3mf. So my thought was, lets just analyze the 3mf file and extract the information I need. It took my quite a bit of back and forth with Claude, feeding claude source code from Prusa’s implementation and pdf files with specifications, but eventually the application did spit out a list of 15 toolchanges and the colors associated with them. So I happily tried to use it to print my model. I quickly discovered that the color ordering was all wrong. And after even more back and forth with Claude and reading online I realized that the 3mf file is a format for storing 3d models, but that is not what is being fed your 3d printer, instead for the printer the file provided is a bgcode file. And while the 3mf file did contain the information that you had to change filament 15 times, the information on in which order is simply not stored in the 3mf file as that is something chosen as part of composing your print. That print composition file is using a file format called bgcode. So I now had to extract the information from the bgcode file which took me basically a full day to figure out with the help of Claude. I could probably have gotten over the finish line sooner by making some better choices underway, but the extreme optimism of the AI probably lead me to believe it was going to be easier than it was to for instance just do everything in Python. At first I tried using this libbgcode library written in C++, but I had a lot of issues getting Claude to incorporate it properly into my project, with Meson and CMAKE interaction issues (in retrospect I should have just made a quick RPM of libbgcode and used that). After a lot of struggles with this Claude thought that parsing the bgcode file in python natively would be easier than trying to use the C++ library, so I went down that route. I started by feeding Claude a description of the format that I found online and asked it to write me a parser for it. It didn’t work very well and I ended up having a lot of back and forth, testing and debugging, finding more documentation, including a blog post about this meatpack format used inside the file, but it still didn’t really work very well. In the end what probably helped the most was

asking it to use the relevant files from libbgcode and Prusa Slicer as documentation, because even if that too took a lot of back and forth, eventually I had a working application that was able to extract the tool change data and associated colors from the file. I ended up using one external dependency which was the heatshrink2 library that I PIP installed, but while that worked correctly, it took a look time for me and Claude to figure out exactly what parameters to feed it to work with the Prusa generated file. Screenshot of Prusa Color Mate So know I had the working application going and was able to verify it with my first print. I even polished it up a little, by also adding detection of the manual filament change code, so that people who try to use the application will be made aware they need to add that through Prusa Slicer. Maybe I could bake that into the tool, but atm I got only bgcode decoders, not encoders, in my project. Warning showed for missing G Code Dialog that gives detailed instructions for how to add G Code So to conclude, it probably took me 2.5 days to write this application using Claude, it is a fairly niche tool, so I don't expect a lot of users, but I made it to solve a problem for myself. If I had to write this pre-AI myself it would have taken me weeks, like figuring out the different formats and how library APIs worked etc. would have taken me a long time. So I am not an especially proficient coder, so a better coder than me could probably put this together quicker than I would, but I think this is part of what I think will change with AI, that even with limited time and technical skills you can put together simple applications like this to solve your own problems. If you are a Prusa Core One user and would like to play with multicolor prints you can find Prusa Color Mate on Gitlab. I have not tested it on any other system or printer than my own, so I don't even know if it will work with other non-Core One Prusa printers. There are rpms for Fedora you can download in the packaging directory of the gitlab repo, which also includes a RPM for the heatshrink2 library. As for future plans for this application I don't really have any. It solves my issue the way it is today, but if there turns out to be an interested user community out there maybe I will try to clean it up and create a proper flatpak for it.

- Mike Blumenkrantz: Big Lifts (2025/09/09 00:00)
  New Record For months now I've been writing increasingly unhinged patchsets. Sometimes it might even seem like there is no real point to what I'm doing. Or that I'm just churning code to have something to do. But I'm here today to tell you that finally, the long journey is over. We have reached the promised land of perf. Huge. Many months ago, I began examining viewperf, AKA the final frontier of driver performance. What makes this the final frontier? some of you might be asking. Imagine an application which does 10,000 individual draws per frame, each with their own vertex buffer bindings. That's a lot of draws. Now imagine an application which does ten times that many draws per frame. This is viewperf, which represents common use cases of CAD-adjacent technologies. Where other applications might hammer on the GPU, viewperf tests the CPU utilization. It's what separates the real developers from average, sane people. So all those months ago, I ran viewperf on zink, and I ended up here: 18fps. This is on threadripper 5975WX with RADV; not the most modern or powerful CPU, but it's still pretty quick. Then I loaded up radeonsi and got 100fps. Brutal. Plumbing The Abyss Examining this was where I entered into into realms of insanity not known to mere mortals. perf started to fail and give confusing results, other profilers just drew a circle around the driver and pointed to the whole thing as the problem area, and some tools just gave up entirely. No changes affected the performance in any way. This is when the savvy hacker begins profiling by elimination: delete as much code as possible and try to force changes. Thus, I deleted a lot of code to see what would pop out, and eventually I discovered the horrifying truth: I was being bottlenecked by the sheer number of atomic operations occurring. Like I said before, viewperf does upwards of 100,000 draw calls per frame. This means 100,000 draw calls, 100,000 vertex buffer binds (times two because there are two vertex buffers), 100,000 index buffer binds, and a few shader changes sprinkled in. The way that mesa/gallium work means that every single vertex buffer and index buffer which get sent to the driver incur multiple atomic operations (each) along the way for refcounting: because gallium uses

refcounting rather than an ownership model since it is much easier to manage. That means we’re talking about upwards of 300,000 atomic operations per frame. Unfortunately, hackily deleting all the refcounting made the FPS go brrrrr, and it was a long road to legitimately get there. A very, very long road. Six months, in fact. But all the unhinged MRs above landed, reducing the surface area of the refcounting to just buffers, which put me in a position to do this pro gamer move where I also am removing all the refcounting from the buffers. This works, roughly speaking, by enforcing ownership on the buffers and then releasing them when they are no longer used. Sounds simple, but plumbing it through all the gallium drivers without breaking everything was less so. Let’s see where moving to that model gets the numbers: One more frame. Tremendous. But wait, there’s more. The other part of that MR further deletes all the refcounting in zink for buffers, fully removing the atomics. And… Blammo, that doubles the perf and manages to eliminate the bottleneck, which sets the stage for further improvements. The gap is still large, but it’s about to close real fast. Shout out to Marek for heroically undertaking the review of this leviathan.

- Mike Blumenkrantz: Mesh Shader Progress (2025/09/05 00:00)
  VKCTS Tests: 27,890 | GLCTS Tests: 227 | Percentage of Vulkan Drivers With Mesh Bugs: 100%
- Hans de Goede: Leaving Red Hat (2025/09/03 18:46)
  After 17 years I feel that it is time to change things up a bit and for a new challenge. I'm leaving Red Hat and my last day at Red Hat will be October 31st.I would like to thank Red Hat for the opportunity to work on many interesting open-source projects during my time at Red Hat and for all the things I've learned while at Red Hat.I want to use this opportunity to thank everyone I've worked with, both my great Red Hat colleagues, as well as everyone from the community for all the good times during the last 17 years.I've a pretty good idea of what will come next, but this is not set in stone yet. I definitely will continue to work on open-source and on Linux hw-enablement. comments
- Mike Blumenkrantz: Tiler Improvements (2025/08/29 00:00)
  Super Late Code Meant to blog about this last quarter, but somehow another two months went by and here we are. A while back, I did some work to improve zink performance on tiling GPUs. Namely this entailed adding renderpass tracking into threaded-context, and also implementing command stream reordering, and inlining swapchain resolves, and framebuffer discards, and actually maybe it’s more than just “some” work. All of this amounted to improved performance by reducing memory bandwidth. How much improved performance? All of it. And then, around two months ago, a colleague told me he was no longer going to use zink on his tiling GPU. Devastated Some of you noticed that the blog has gone quiet in recent times. I’m going to take this opportunity to foist all the blame onto that colleague: to preserve his identity, let’s just call him Gabe. Gabe came to me a few months ago and told me zink was too slow. Vulkan was better. Faster. More “reliable”. I said there’s no way that could be true; I’ve put way more bugs into Vulkan than I have into zink. Unblinking, he stared at me across the digital divide. I task-switched to important whitespace cleanups. Time passed, and I pulled myself together. I compiled some app traces. Analyzed them. Did some deep thinking. There was one place where zink indeed could be less performant than this “Vulkan” thing. The final frontier of driver performance. Some call it graphics heaven. I call it hell. Web Browsers Chrome is the web browser, and, statistically, everyone uses it. It ships on desktops and phones, embeds in apps, and even allows you to read this blog. Haters will say No I uSe FiReFoX, but they may as well be Netscape users in the year 2000. In the past, Chrome defaulted to using GL, which made testing easy. Now, however, --disable-features=Vulkan is needed to return to the comfort of an API so reliable it no longer receives versioned updates. Looking at an apitrace of Chrome, I saw a disturbing rendering pattern that went something like this: draw some element on a page using multisampled FBO1 resolve FBO1 to texture1 composite texture1 onto larger FBO2/texture2 composite texture2 onto even larger, multisampled FBO3 resolve FBO3 to swapchain present In this case, zink would correctly

inline the FBO3/swapchain resolve at the end, but the intermediate multisampled rendering on FBO1 would pay the full performance penalty of storing the multisampled image data and then loading it again for the separate resolve operation. I'd like to say it was simple to inline this intermediate resolve. That I just slapped a single MR into mesa and it magically worked. Unfortunately, nothing is ever that simple. There were minor fixups all over the place. And this brought me to the real insanity. Chrome has bugs too. Literal Hell Let's take a concrete example: launch Chrome with --disable-features=Vulkan and check out this tiny SVG: chromebug.html This is most likely what you see: The reason you see this is because you are on a big, strong desktop GPU which doesn't give a shit about load/store ops or uninitialized GPU memory. You're driving a giant industrial bulldozer on your morning commute: traffic no longer exists and stop signals are fully optional. On a wimpy tiling GPU, however, things are different. Using a recent version of zink, even on a desktop GPU, you can run the same Chrome browser using ZINK_DEBUG=rp,rploads to enable the same codepaths used by tilers and also clear all uninitialized memory to red. Now load the same SVG, and you'll see this: It took nearly a week of pair debugging and a new zink debug mode to prune down test cases and figure out what was happening. All around the composited SVG texture, memory is uninitialized. But this only shows up on tiling GPUs. And only if the driver is doing near-lethal amounts of very legal renderpass optimizations. This fast is too fast.

- Alyssa Rosenzweig: Dissecting the Apple M1 GPU, the end (2025/08/26 05:00)
  In 2020, Apple released the M1 with a custom GPU. We got to work reverse-engineering the hardware and porting Linux. Today, you can run Linux on a range of M1 and M2 Macs, with almost all hardware working: wireless, audio, and full graphics acceleration. Our story begins in December 2020, when Hector Martin kicked off Asahi Linux. I was working for Collabora working on Panfrost, the open source Mesa3D driver for Arm Mali GPUs. Hector put out a public call for guidance from upstream open source maintainers, and I bit. I just intended to give some quick pointers. Instead, I bought myself a Christmas present and got to work. In between my university coursework and Collabora work, I poked at the shader instruction set. One thing led to another. Within a few weeks, I drew a triangle. In 3D graphics, once you can draw a triangle, you can do anything. Pretty soon, I started work on a shader compiler. After my final exams that semester, I took a few days off from Collabora to bring up an OpenGL driver capable of spinning gears with my new compiler. Over the next year, I kept reverse-engineering and improving the driver until it could run 3D games on macOS. Meanwhile, Asahi Lina wrote a kernel driver for the Apple GPU. My userspace OpenGL driver ran on macOS, leaving her kernel driver as the missing piece for an open source graphics stack. In December 2022, we shipped graphics acceleration in Asahi Linux. In January 2023, I started my final semester in my Computer Science program at the University of Toronto. For years I juggled my courses with my part-time job and my hobby driver. I faced the same question as my peers: what will I do after graduation? Maybe Panfrost? I started reverse-engineering of the Mali Midgard GPU back in 2017, when I was still in high school. That led to an internship at Collabora in 2019 once I graduated, turning into my job throughout four years of university. During that time, Panfrost grew from a kid's pet project based on blackbox reverse-engineering, to a professional driver engineered by a team with Arm's backing and hardware documentation. I did what I set out to do, and the project succeeded beyond my dreams. It was time to move on. What did I want to do next? Finish what I started with the M1. Ship a great driver. Bring full, conformant OpenGL drivers to the M1. Apple's drivers are not conformant, but we should strive for the industry standard. Bring full, conformant Vulkan to Apple platforms, disproving the myth that Vulkan isn't suitable for Apple hardware. Bring Proton gaming to Asahi Linux. Thanks to Valve's work for the Steam Deck, Windows games can run better on Linux than even on Windows. Why not reap those benefits on the M1? Panfrost was my challenge until we "won". My next challenge? Gaming on Linux on M1. Once I finished my coursework, I started full-time on gaming on Linux. Within a month, we shipped OpenGL 3.1 on Asahi Linux. A few weeks later, we passed official conformance for OpenGL

ES 3.1. That put us at feature parity with Panfrost. I wanted to go further. OpenGL (ES) 3.2 requires geometry shaders, a legacy feature not supported by either Arm or Apple hardware. The proprietary OpenGL drivers emulate geometry shaders with compute, but there was no open source prior art to borrow. Even though multiple Mesa drivers need geometry/tessellation emulation, nobody did the work to get there. My early progress on OpenGL was fast thanks to the mature common code in Mesa. It was time to pay it forward. Over the rest of the year, I implemented geometry/tessellation shader emulation. And also the rest of the owl. In January 2024, I passed conformance for the full OpenGL 4.6 specification, finishing up OpenGL. Vulkan wasn't too bad, either. I polished the OpenGL driver for a few months, but once I started typing a Vulkan driver, I passed 1.3 conformance in a few weeks. What remained was wiring up the geometry/tessellation emulation to my shiny new Vulkan driver, since those are required for Direct3D. Et voilà, Proton games. Along the way, Karol Herbst passed OpenCL 3.0 conformance on the M1, running my compiler atop his "rusticl" frontend. Meanwhile, when the Vulkan 1.4 specification was published, we were ready and shipped a conformant implementation on the same day. After that, I implemented sparse texture support, unlocking Direct3D 12 via Proton. …Now what? Ship a great driver? Check. Conformant OpenGL 4.6, OpenGL ES 3.2, and OpenCL 3.0? Check. Conformant Vulkan 1.4? Check. Proton gaming? Check. That's a wrap. We've succeeded beyond my dreams. The challenges I chased, I have tackled. The drivers are fully upstream in Mesa. Performance isn't too bad. With the Vulkan on Apple myth busted, conformant Vulkan is now coming to macOS via LunarG's KosmicKrisp project building on my work. Satisfied, I am now stepping away from the Apple ecosystem. My friends in the Asahi Linux orbit will carry the torch from here. As for me? Onto the next challenge!

- Sebastian Wick: Testing with Portals (2025/08/21 21:00)
  At the Linux App Summit (LAS) in Albania three months ago, I gave a talk about testing in the xdg-desktop-portal project. There is a recording of the presentation, and the slides are available as well. To give a quick summary of the work I did: Revamped the CI Reworked and improved the pytest based integration test harness Added integration tests for new portals Ported over all the existing GLib/C based integration tests Support ASAN for detecting memory leaks in the tests Made tests pretend to be either a host, Flatpak or Snap app The hope I had is that this will result in: Fewer regressions Tests for new features and bug fixes More confidence in refactoring More activity in the project While it's hard to get definite data on those points, at least some of it seems to have become reality. I have seen an increase in activity (there are other factors to this for sure), and a lot of PRs already come with tests without me even having to ask for it. Canonical is involved again, taking care of the Snap side of things. So far it seems like we didn't introduce any new regressions, but this usually shows after a new release. The experience of refactoring portals also became a lot better because there is a baseline level of confidence when the tests pass, as well as the possibility to easily bisect issues. Overall I'm already quite happy with the results. Two weeks ago, Georges merged the last piece of what I talked about in the LAS presentation, so we're finally testing the code paths that are specific to host, Flatpak and Snap applications! I also continued a bit with improving the tests, and now they can be run with Valgrind, which is super slow and that's why we're not doing it in the CI, but it tends to find memory leaks which ASAN does not. With the existing tests, it found 9 small memory leaks. If you want to improve the Flatpak story, come and contribute to xdg-desktop-portal. It's now easier than ever!
- Peter Hutterer: Why is my device a touchpad and a mouse and a keyboard? (2025/08/20 01:12)
  If you have spent any time around HID devices under Linux (for example if you are an avid mouse, touchpad or keyboard user) then you may have noticed that your single physical device actually shows up as multiple device nodes (for free! and nothing happens for free these days!). If you haven't noticed this, run libinput record and you may be part of the lucky roughly 50% who get free extra event nodes. The pattern is always

the same. Assuming you have a device named FooBar ExceptionalDog 2000 AI[1] what you will see are multiple devices /dev/input/event0: FooBar ExceptionalDog 2000 AI Mouse /dev/input/event1: FooBar ExceptionalDog 2000 AI Keybard /dev/input/event2: FooBar ExceptionalDog 2000 AI Consumer Control The Mouse/Keyboard/Consumer Control/... suffixes are a quirk of the kernel's HID implementation which splits out a device based on the Application Collection. [2] A HID report descriptor may use collections to group things together. A "Physical Collection" indicates "these things are (on) the same physical thingy". A "Logical Collection" indicates "these things belong together". And you can of course nest these things near-indefinitely so e.g. a logical collection inside a physical collection is a common thing. An "Application Collection" is a high-level abstractions to group something together so it can be detected by software. The "something" is defined by the HID usage for this collection. For example, you'll never guess what this device might be based on the hid-recorder output: # 0x05, 0x01, // Usage Page (Generic Desktop) 0 # 0x09, 0x06, // Usage (Keyboard) 2 # 0xa1, 0x01, // Collection (Application) 4 ... # 0xc0, // End Collection 74 Yep, it's a keyboard. Pop the champagne[3] and hooray, you deserve it. The kernel, ever eager to help, takes top-level application collections (i.e. those not inside another collection) and applies a usage-specific suffix to the device. For the above Generic Desktop/Keyboard usage you get "Keyboard", the other ones currently supported are "Keypad" and "Mouse" as well as the slightly more niche "System Control", "Consumer Control" and "Wireless Radio Control" and "System Multi Axis". In the Digitizer usage page we have "Stylus", "Pen", "Touchscreen" and "Touchpad". Any other Application Collection is currently unsuffixed (though see [2] again, e.g. the hid-uclogic driver uses "Touch Strip" and other suffixes). This suffix is necessary because the kernel also splits out the data sent within each collection as separate evdev event node. Since HID is (mostly) hidden from userspace this makes it much easier for userspace to identify different devices because you can look at a event node and say "well, it has buttons and x/y, so must be a mouse" (this is exactly what udev does when applying the various ID_INPUT properties, with varying levels of success). The side effect of this however is that your device may show up as multiple devices and most of those extra devices will never send events. Sometimes that is due to the device supporting multiple modes (e.g. a touchpad may by default emulate a mouse for backwards compatibility but once the kernel toggles it to touchpad mode the mouse feature is mute). Sometimes it's just laziness when vendors re-use the same firmware and leave unused bits in place. It's largely a cosmetic problem only, e.g. libinput treats every event node as individual device and if there is a device that never sends events it won't affect the other event nodes. It can cause user confusion though: "why does my laptop say there's a mouse?" and in some cases it can cause functional degradation - the two I can immediately recall are udev detecting the mouse node of a touchpad as pointing stick (because i2c mice aren't a thing), hence the pointing stick configuration may show up in unexpected places. And fake mouse devices prevent features like "disable touchpad if a mouse is plugged in" from working correctly. At the moment we don't have a good solution for detecting these fake devices - short of shipping giant databases with product-specific entries we cannot easily detect which device is fake. After all, a Keyboard node on a gaming mouse may only send events if the user configured the firmware to send keyboard events, and the same is true for a Mouse node on a gaming keyboard. So for now, the only solution to those is a per-user udev rule to ignore a device. If we ever figure out a better fix, expect to find a gloating blog post in this very space. [1] input device naming is typically bonkers, so I'm just sticking with precedence here [2] if there's a custom kernel driver this may not apply and there are quirks to change this so this isn't true for all devices [3] or sparkling wine, let's not be regionist here

- Simon Ser: Status update, August 2025 (2025/08/16 22:00)
  Hi! This month I've spent quite some time working on vali, a C library and code generator for the Varlink IPC protocol. It was formerly named "varlinkgen", but the new name is shorter and more accurate (the library can be used without the code generator). I've fixed a bunch of bugs,

updated the service implementation to use non-blocking IO, added some tests and support for continued calls (which are Varlink's way to emit events from a service). I've also written a patch to port the kanshi output configuration tool to vali. Speaking of kanshi, I've released version 1.8. A new kanshictl status command shows the current mode, and wildcard patterns are now supported to match outputs. I want to finish up relative output positioning for the next release, but some tricky usability issues need to be sorted out first. Support for toplevel capture in xdg-desktop-portal-wlr has been merged. This was the last missing piece to be able to share an individual window from Web browsers. libdisplay-info v0.3 has been released with support for many new CTA data blocks and groundwork for DisplayID v2. José Expósito has sent libdrm and drm_info patches to add user-space support for the special "faux" bus used in cases where a device isn't backed by hardware (up until now, the platform bus was abused). The Goguma mobile IRC client now displays a small bubble when someone else mentions you, making these easier to spot at a glance: Jean THOMAS has added a new option to choose between the in-app Web view and an external browser when opening links. Hubert Hirtz has tweaked the login forms to play better with the autofill feature some password managers provide. I've released go-imap v2 beta 6, with support for SPECIAL-USE and CHILDREN thanks to Dejan Štrbac and legacy RECENT thanks to fox.cpp. I'd like to eventually ship v2, but there are still some rough edges that I'd like to smooth out. I now realize it's been more than 2 years since the first v2 alpha release, maybe I should listen a bit more to my bio teacher who used to repeat "perfect is the enemy of good". Anyways, that's all for now, see you next month!
- Sebastian Wick: Display Next Hackfest 2025 (2025/08/15 15:41)
A few weeks ago, a bunch of display driver and compositor developers met once again for the third iteration of the Display Next Hackfest. The tradition was started by Red Hat, followed by Igalia (thanks Melissa), and now AMD (thanks Harry). We met in the AMD offices in Markham, Ontario, Canada; and online, to discuss issues, present things we worked on, figure out future steps on a bunch of topics related to displays, GPUs, and compositors. The Display Next Hackfest in the AMD Markham offices It was really nice meeting everyone again, and also seeing some new faces! Notably, Charles Poynton who "decided that HD should have 1080 image rows, and square pixels", and Keith Lee who works for AMD and designed their color pipeline, joined us this year. This turned out to be invaluable. It was also great to see AMD not only organizing the event, but also showing genuine interest and support for what we are trying to achieve. This year's edition is likely going to be the last dedicated Display Next Hackfest, but we're already plotting to somehow fuse it with XDC next year in some way. If you're looking for a more detailed technical rundown of what we were doing there, you can read Xaver's, or Louis' blog posts, or our notes. With all that being said, here is an incomplete list of things I found exciting: The biggest update of the Atomic KMS API (used to control displays) is about to get merged. The Color Pipeline API is something I came up with three years ago, and thanks to the tireless efforts of AMD, Intel and Igalia and others, this is about to become reality. Read Melissa's blog post for more details. As part of the work enabling displaying HDR content in Wayland compositors, we've been unhappy with the current HDR modes in displays, as they are essentially created for video playback and have lots of unpredictable behavior. To address this, myself and Xaver have since last year been lobbying for displays to allow the use of Source Based Tone Mapping (SBTM), and this year, it seems that what we have asked for have made it to the right people. Let's see! In a similar vein, on mobile devices we want to dynamically increase or decrease the HDR headroom, depending on what content applications want to show. This requires backlight changes to be somewhat atomic and having a mapping to luminance. The planned KMS backlight API will allow us to expose this, if the platform supports it. I worked a lot on backlight support in mutter this year so we can immediately start using this when it becomes available. Charles, Christopher, and I had a discussion about compositing HDR and SDR content, and specifically about how to adjust content that was mastered for a dark viewing environment that is being shown in a bright viewing environment, so that the perception is maintained. I believe that we now

have a complete picture of how compositing should work, and I’m working on documenting this in the color-and-hdr repo. For Variable Refresh Rates (VRR) we want a new KMS API to set the minimum and maximum refresh cycle, where setting min=max gives us a fixed refresh rate without a mode set. To make use of VRR in more than the single-fullscreen-window case, we also agreed that a Wayland protocol letting clients communicate their preferred refresh rate would be a good idea. Like always, lots of work ahead of us, but it’s great to actually see the progress this year with the entire ecosystem having HDR support now. Sampling local craft beers See you all at XDC this year (or at least the one next year)!

- Peter Hutterer: xkeyboard-config 2.45 has a new install location (2025/08/11 01:44)
  This is a heads ups that if you install xkeyboard-config 2.45 (the package that provides the XKB data files), some manual interaction may be needed. Version 2.45 has changed the install location after over 20 years to be a) more correct and b) more flexible. When you select a keyboard layout like "fr" or "de" (or any other ones really), what typically happens in the background is that an XKB parser (xkbcomp if you're on X, libxkbcommon if you're on Wayland) goes off and parses the data files provided by xkeyboard-config to populate the layouts. For historical reasons these data files have resided in /usr/share/X11/xkb and that directory is hardcoded in more places than it should be (i.e. more than zero). As of xkeyboard-config 2.45 however, the data files are now installed in the much more sensible directory /usr/share/xkeyboard-config-2 with a matching xkeyboard-config-2.pc for anyone who relies on the data files. The old location is symlinked to the new location so everything keeps working, people are happy, no hatemail needs to be written, etc. Good times. The reason for this change is two-fold: moving it to a package-specific directory opens up the (admittedly mostly theoretical) use-case of some other package providing XKB data files. But even more so, it finally allows us to start versioning the data files and introduce new formats that may be backwards-incompatible for current parsers. This is not yet the case however, the current format in the new location is guaranteed to be the same as the format we've always had, it's really just a location change in preparation for future changes. Now, from an upstream perspective this is not just hunky, it's also dory. Distributions however struggle a bit more with this change because of packaging format restrictions. RPM for example is quite unhappy with a directory being replaced by a symlink which means that Fedora and OpenSuSE have to resort to the .rpmmoved hack. If you have ever used the custom layout and/or added other files to the XKB data files you will need to manually move those files from /usr/share/X11/xkb.rpmmoved/ to the new equivalent location. If you have never used that layout and/or modified local you can just delete /usr/share/X11/xkb.rpmmoved. Of course, if you're on Wayland you shouldn't need to modify system directories anyway since you can do it in your $HOME. Corresponding issues on what to do on Arch and Gentoo, I'm not immediately aware of other distributions's issues but if you search for them in your bugtracker you'll find them.
- Sebastian Wick: GNOME 49 Backlight Changes (2025/08/08 15:26)
  One of the things I’m working on at Red Hat is HDR support. HDR is inherently linked to luminance (brightness, but ignoring human perception) which makes it an important parameter for us that we would like to be in control of. One reason is rather stupid. Most external HDR displays refuse to let the user control the luminance in their on-screen-display (OSD) if the display is in HDR mode. Why? Good question. Read my previous blog post. The other reason is that the amount of HDR headroom we have available is the result of the maximum luminance we can achieve versus the luminance that we use as the luminance a sheet of white paper has (reference white level). For power consumption reasons, we want to be able to dynamically change the available headroom, depending on how much headroom the content can make use of. If there is no HDR content on the screen, there is no need to crank up the backlight to give us more headroom, because the headroom will be unused. To work around the first issue, mutter can change the signal it sends to the display, so that white is not a signal value of 1.0 but somewhere else

between 0 and 1. This essentially emulates a backlight in software. The drawback is that we’re not using a bunch of bits in the signal anymore and issues like banding might become more noticeable, but since we’re using this only with 10 or 12 bits HDR signals, this isn’t an issue in practice. This has been implemented in GNOME 48 already, but it was an API that mutter exposed and Settings showed as “HDR Brightness”. "HDR Brightness" in GNOME Settings The second issue requires us to be able to map the backlight value to luminance, and change the backlight atomically with an update to the screen. We could work towards adding those things to the existing sysfs backlight API but it turns out that there are a number of problems with it. Mapping the sysfs entry to a connected display is really hard (GNOME pretends that there is only one single internal display that can ever be controlled), and writing a value to the backlight requires root privileges or calling a logind dbus API. One internal panel can expose multiple backlights, the value of 0 can mean the display turns off or is just really dim. So a decision was made to create a new API that will be part of KMS, the API that we use to control the displays. The sysfs backlight has been controlled by gnome-settings-daemon and GNOME Shell called a dbus API when the screen brightness slider was moved. To recap: There is a sysfs backlight API for basically one internal panel requires logind or a setuid helper executable gnome-settings-daemon controlled this single screen sysfs backlight Mutter has a “software backlight” feature KMS will get a new backlight API that needs to be controlled by mutter Overall, this is quite messy, so I decided to clean this up. Over the last year, I moved the sysfs backlight handling from gnome-settings-daemon into mutter, added logic to decide which backlight to use (sysfs or the “software backlight”), and made it generic so that any screen can have a backlight. This means mutter is the single source of truth for the backlight itself. The backlight itself gets its value from a number of sources. The user can configure the screen brightness in the quick settings menu and via keyboard shortcuts. Power saving features can kick in and dim the screen. Lastly, an Ambient Light Sensor (ALS) can take control over the screen brightness. To make things more interesting, a single “logical monitor” can have multiple hardware monitors which each can have a backlight. All of that logic is now neatly sitting in GNOME Shell which takes signals from gnome-settings-daemon about the ALS and dimming. I also changed the Quick Settings UI to make it possible to control the brightness on multiple screens, and removed the old “HDR Brightness” from Settings. There should have been a video here but your browser does not seem to support it. All of this means that we can now handle screen brightness on multiple monitors and when the new KMS backlight API makes it upstream, we can just plug it in, and start to dynamically create HDR headroom.

- Peter Hutterer: libinput and Lua plugins (Part 2) (2025/08/07 10:00)
  Part 2 is, perhaps suprisingly, a follow-up to libinput and lua-plugins (Part 1). The moon has circled us a few times since that last post and some update is in order. First of all: all the internal work required for plugins was released as libinput 1.29 but that version does not have any user-configurable plugins yet. But cry you not my little jedi and/or sith lord in training, because support for plugins has now been merged and, barring any significant issues, will be in libinput 1.30, due somewhen around October or November. This year. 2025 that is. Which means now is the best time to jump in and figure out if your favourite bug can be solved with a plugin. And if so, let us know and if not, then definitely let us know so we can figure out if the API needs changes. The API Documentation for Lua plugins is now online too and will auto-update as changes to it get merged. There have been a few minor changes to the API since the last post so please refer to the documentation for details. Notably, the version negotiation was re-done so both libinput and plugins can support select versions of the plugin API. This will allow us to iterate the API over time while designating some APIs as effectively LTS versions, minimising plugin breakages. Or so we hope. What warrants a new post is that we merged a new feature for plugins, or rather, ahaha, a non-feature. Plugins now have an API accessible that allows them to disable certain internal features that are not publicly exposed, e.g. palm detection. The reason why libinput doesn't have a lot of configuration options have been

explained previously (though we actually have quite a few options) but let me recap for this particular use-case: libinput doesn't have a config option for e.g. palm detection because we have several different palm detection heuristics and they depend on device capabilities. Very few people want no palm detection at all[1] so disabling it means you get a broken touchpad and we now get to add configuration options for every palm detection mechanism. And keep those supported forever because, well, workflows. But plugins are different, they are designed to take over some functionality. So the Lua API has a EvdevDevice:disable_feature("touchpad-palm-detection") function that takes a string with the feature's name (easier to make backwards/forwards compatible this way). This example will disable all palm detection within libinput and the plugin can implement said palm detection itself. At the time of writing, the following self-explanatory features can be disabled: "button-debouncing", "touchpad-hysteresis", "touchpad-jump-detection", "touchpad-palm-detection", "wheel-debouncing". This list is mostly based on "probably good enough" so as above - if there's something else then we can expose that too. So hooray for fewer features and happy implementing! [1] Something easily figured out by disabling palm detection or using a laptop where palm detection doesn't work thanks to device issues

- Peter Hutterer: libinput and Lua plugins (2025/08/07 00:44)
First of all, what's outlined here should be available in libinput 1.29 1.30 but I'm not 100% certain on all the details yet so any feedback (in the libinput issue tracker) would be appreciated. Right now this is all still sitting in the libinput!1192 merge request. I'd specifically like to see some feedback from people familiar with Lua APIs. With this out of the way: Come libinput 1.29 1.30, libinput will support plugins written in Lua. These plugins sit logically between the kernel and libinput and allow modifying the evdev device and its events before libinput gets to see them. The motivation for this are a few unfixable issues - issues we knew how to fix but we cannot actually implement and/or ship the fixes without breaking other devices. One example for this is the inverted Logitech MX Master 3S horizontal wheel. libinput ships quirks for the USB/Bluetooth connection but not for the Bolt receiver. Unlike the Unifying Receiver the Bolt receiver doesn't give the kernel sufficient information to know which device is currently connected. Which means our quirks could only apply to the Bolt receiver (and thus any mouse connected to it) - that's a rather bad idea though, we'd break every other mouse using the same receiver. Another example is an issue with worn out mouse buttons - on that device the behavior was predictable enough but any heuristics would catch a lot of legitimate buttons. That's fine when you know your mouse is slightly broken and at least it works again. But it's not something we can ship as a general solution. There are plenty more examples like that - custom pointer deceleration, different disable-while-typing, etc. libinput has quirks but they are internal API and subject to change without notice at any time. They're very definitely not for configuring a device and the local quirk file libinput parses is merely to bridge over the time until libinput ships the (hopefully upstreamed) quirk. So the obvious solution is: let the users fix it themselves. And this is where the plugins come in. They are not full access into libinput, they are closer to a udev-hid-bpf in userspace. Logically they sit between the kernel event devices and libinput: input events are read from the kernel device, passed to the plugins, then passed to libinput. A plugin can look at and modify devices (add/remove buttons for example) and look at and modify the event stream as it comes from the kernel device. For this libinput changed internally to now process something called an "evdev frame" which is a struct that contains all struct input_events up to the terminating SYN_REPORT. This is the logical grouping of events anyway but so far we didn't explicitly carry those around as such. Now we do and we can pass them through to the plugin(s) to be modified. The aforementioned Logitech MX master plugin would look like this: it registers itself with a version number, then sets a callback for the "new-evdev-device" notification and (where the device matches) we connect that device's "evdev-frame" notification to our actual code: libinput:register(1) -- register plugin version 1 libinput:connect("new-evdev-device", function (_, device) if device:vid() == 0x046D and device:pid() == 0xC548 then device:connect("evdev-frame", function (_, frame) for _, event in ipairs(frame.events)

do if event.type == evdev.EV_REL and (event.code == evdev.REL_HWHEEL or event.code == evdev.REL_HWHEEL_HI_RES) then event.value = -event.value end end return frame end) end end) This file can be dropped into /etc/libinput/plugins/10-mx-master.lua and will be loaded on context creation. I'm hoping the approach using named signals (similar to e.g. GObject) makes it easy to add different calls in future versions. Plugins also have access to a timer so you can filter events and re-send them at a later point in time. This is useful for implementing something like disable-while-typing based on certain conditions. So why Lua? Because it's very easy to sandbox. I very explicitly did not want the plugins to be a side-channel to get into the internals of libinput - specifically no IO access to anything. This ruled out using C (or anything that's a .so file, really) because those would run a) in the address space of the compositor and b) be unrestricted in what they can do. Lua solves this easily. And, as a nice side-effect, it's also very easy to write plugins in.[1] Whether plugins are loaded or not will depend on the compositor: an explicit call to set up the paths to load from and to actually load the plugins is required. No run-time plugin changes at this point either, they're loaded on libinput context creation and that's it. Otherwise, all the usual implementation details apply: files are sorted and if there are files with identical names the one from the highest-precedence directory will be used. Plugins that are buggy will be unloaded immediately. If all this sounds interesting, please have a try and report back any APIs that are broken, or missing, or generally ideas of the good or bad persuation. Ideally before we ship it and the API is stable forever :) [1] Benjamin Tissoires actually had a go at WASM plugins (via rust). But ... a lot of effort for rather small gains over Lua

- Peter Hutterer: unplug - a tool to test input devices via uinput (2025/08/04 06:09)
Yet another day, yet another need for testing a device I don't have. That's fine and that's why many years ago I wrote libinput record and libinput replay (more powerful successors to evemu and evtest). Alas, this time I had a dependency on multiple devices to be present in the system, in a specific order, sending specific events. And juggling this many terminal windows with libinput replay open was annoying. So I decided it's worth the time fixing this once and for all (haha, lolz) and wrote unplug. The target market for this is niche, but if you're in the same situation, it'll be quite useful. Pictures cause a thousand words to finally shut up and be quiet so here's the screenshot after running pip install unplug[1]: This shows the currently pre-packaged set of recordings that you get for free when you install unplug. For your use-case you can run libinput record, save the output in a directory and then start unplug path/to/directory. The navigation is as expected, hitting enter on the devices plugs them in, hitting enter on the selected sequence sends that event sequence through the previously plugged device. Annotation of the recordings (which must end in .yml to be found) can be done by adding a YAML unplug: entry with a name and optionally a multiline description. If you have recordings that should be included in the default set, please file a merge request. Happy emulating! [1] And allowing access to /dev/uinput. Details, schmetails...
- Christian Schaller: Artificial Intelligence and the Linux Community (2025/07/29 16:24)
I have wanted to write this blog post for quite some time, but been unsure about the exact angle of it. I think I found that angle now where I will root the post in a very tangible concrete example. So the reason I wanted to write this was because I do feel there is a palpable skepticism and negativity towards AI in the Linux community, and I understand that there are societal implications that worry us all, like how deep fakes have the potential to upend a lot of things from news disbursement to court proceedings. Or how malign forces can use AI to drive narratives in social media etc., is if social media wasn't toxic enough as it is. But for open source developers like us in the Linux community there is also I think deep concerns about tooling that deeply incurs into something that close to the heart of our community, writing code and being skilled at writing code. I hear and share all those concerns, but at the same time having spent time the last weeks using Claude.ai I do feel it is not something we can

afford not to engage with. So I know people have probably used a lot of different AI tools in the last year, some being more cute than useful others being somewhat useful and others being interesting improvements to your Google search for instance. I think I shared a lot of those impressions, but using Claude this last week has opened my eyes to what AI engineers are going to be capable of going forward. So my initial test was writing a python application for internal use at Red Hat, basically connecting to a variety of sources and pulling data and putting together reports, typical management fare. How simple it was impressed me though, I think most of us having to deal with pulling data from a new source know how painful it can be, with issues ranging from missing, outdated or hard to parse API documentation. I think a lot of us also then spend a lot of time experimenting to figure out the right API calls to make in order to pull the data we need. Well Claude was able to give me python scripts that pulled that data right away, I still had to spend some time with it to fine tune the data being pulled and ensuring we pulled the right data, but I did it in a fraction of the time I would have spent figuring that stuff out on my own. The one data source Claude struggled with Fedora’s Bohdi, well once I pointed it to the URL with the latest documentation for that it figured out that it would be better to use the bohdi client library to pull data and once it had that figured out it was clear sailing. So coming of pretty impressed by that experience I wanted to understand if Claude would be able to put together something programmatically more complex, like a GTK+ application using Vulkan. [Note: should have checked the code better, but thanks to the people who pointed this out. I told the AI to use Vulkan, which it did, but not in the way I expected, I expected it to render the globe using Vulkan, but it instead decided to ensure GTK used its Vulkan backend, an important lesson in both prompt engineering and checking the code afterwards).]So I thought what would be a good example of such an application and I also figured it would be fun if I found something really old and asked Claude to help me bring it into the current age. So I suddenly remembered xtraceroute, which is an old application orginally written in GTK1 and OpenGL showing your traceroute on a 3d Globe.Screenshot of the original Xtraceroute application I went looking for it and found that while it had been updated to GTK2 since last I looked at it, it had not been touched in 20 years. So I thought, this is a great testcase. So I grabbed the code and fed it into Claude, asking Claude to give me a modern GTK4 version of this application using Vulkan. Ok so how did it go? Well it ended up being an iterative effort, with a lot of back and forth between myself and Claude. One nice feature Claude has is that you can upload screenshots of your application and Claude will use it to help you debug. Thanks to that I got a long list of screenshots showing how this application evolved over the course of the day I spent on it. This screenshot shows Claudes first attempt of transforming the 20 year old xtraceroute application into a modern one using GTK4, Vulkan and also adding a Meson build system. My prompt to create this was feeding in the old code and asking Claude to come up with a GTK4 and Vulkan equivalent. As you can see the GTK4 UI is very simple, but ok as it is. The rendered globe leaves something to be desired though. I assume the old code had some 2d fall backcode, so Claude latched onto that and focused on trying to use the Cairo API to recreate this application, despite me telling it I wanted a Vulkan application. What what we ended up with was a 2d circle that I could spin around like a wheel of fortuen. The code did have some Vulkan stuff, but defaulted to the Cairo code. Second attempt at updating this application Anyway, I feed the screenshot of my first version back into Claude and said that the image was not a globe, it was missing the texture and the interaction model was more like a wheel of fortune. As you can see the second attempt did not fare any better, in fact we went from circle to square. This was also the point where I realized that I hadn’t uploaded the textures into Claude, so I had to tell it to load the earth.png from the local file repository. Third attempt from Claude.Ok, so I feed my second screenshot back into Claude and pointed out that it was no globe, in fact it wasn’t even a circle and the texture was still missing. With me pointing out it needed to load the earth.png file from disk it came back with the texture loading. Well, I really wanted it to be a globe, so I said thank you for loading the texture, now do it on a globe. This is the output of the 4th attempt. As you can see, it did bring back a circle, but the texture was gone again. At this point I also decided I didn’t want Claude to waste anymore time on the Cairo code, this was meant to be a proper 3d application. So I told Claude to drop all the Cairo code and instead focus on making a Vulkan application. So now we finally had something that started looking like something, although it was still a circle, not a globe and it got that weird division of 4 thing on the globe. Anyway, I could see it using Vulkan now and it was loading the texture. So I was feeling like we where making some decent forward movement. So I wrote a longer prompt describing the globe I wanted and how I wanted to interact with it and this time Claude did come back with Vulkan code that rendered this as a globe, thus I didn’t end up screenshoting it unfortunately. So with the working globe now in place, I wanted to bring in the day/night cycle from the original application. So I asked Claude to load the night texture and use it as an overlay to get that day/night effect. I also asked it to calculate the position of the sun to earth at the current time, so that it could overlay the texture in the right location. As you can see Claude did a decent job of it, although the colors was broken. So I kept fighting with the color for a bit, Claude could see it was rendering it brown, but could not initally figure out why. I could tell the code was doing things mostly right so I also asked it to look at some other things, like I realized that when I tried to spin the globe it just twisted the texture. We got that fixed and also I got Claude to create some tests scripts that helped us figure out that the color issue was a RGB vs BRG issue, so as soon as we understood that then Claude was able to fix the code to render colors correctly. I also had a few iterations trying to get the scaling and mouse interaction behaving correctly. So at this point I had probably worked on this for 4-5 hours, the globe was rendering nicely and I could interact with it using the mouse. Next step was adding the traceroute lines back. By default Claude had just put in code to render some small dots on the hop points, not draw the lines. Also the old method for getting the geocoordinates, but I asked Claude to help me find some current services which it did and once I picked one it on first try gave me code that was able to request the geolocation of the ip addresses it got back. To polish it up I also asked Claude to make sure we drew the lines following the globes curvature instead of just drawing straight lines. Final version of the updated Xtraceroute application. It mostly works now, but I did realize why I always thought this was a fun idea, but less interesting in practice, you often don’t get very good traceroutes back, probably due to websites being cached or hosted globally. But I felt that I had proven that with a days work Claude was able to help me bring this old GTK application into the modern world. Conclusions So I am not going to argue that Xtraceroute is an important application that deserved to be saved, in fact while I feel the current version works and proves my point I also lost motivation to try to polish it up due to the limitations of tracerouting, but the code is available for anyone who finds it worthwhile. But this wasn’t really about Xtraceroute, what I wanted to show here is how someone lacking C and Vulkan development skills can actually use a tool like Claude to put together a working application even one using more advanced stuff like Vulkan, which I know many more than me would feel daunting. I also found Claude really good at producing documentation and architecture documents for your application. It was also able to give me a working Meson build system and create all the desktop integration files for me, like the .desktop file, the metainfo file and so on. For the icons I ended up using Gemini as Claude do not do image generation at this point, although it was able to take a png file and create a SVG version of it (although not a perfect likeness to the original png). Another thing I want to say is that the way I think about this, it is not that it makes coding skills less valuable, AIs can do amazing things, but you need to keep a close eye on them to ensure the code they create actually do what you want and that it does it in a sensible manner. For instance in my reporting application I wanted to embed a pdf file and Claude initial thought was to bring in webkit to do the rendering. That would have worked, but would have added a very big and complex dependency to my application, so I had to tell it that it could just use libpoppler to do it, something Claude agreed was a much better solution. The bigger the codebase the harder it also becomes for the AI to deal with it, but I think it hose circumstances what you can do is use the AI to give you sample code for the functionality you want in the programming language you want and then you can just work on incorporating that into your big application. The other part here if course in terms of open source is how should contributors and projects deal with this? I know there are projects where AI generated CVEs or patches are drowning them and that helps nobody. But I think if we see AI as a developers tool and that the developer using the tool is responsible for the code generated, then I think that mindset can help us navigate this. So if you used an AI tool to create a patch for your favourite project, it is your responsibility to verify that patch before sending it in, and with that I don't mean just verifying the functionality it provides, but that the code is clean and readable and following the coding standards of said upstream project. Maintainers on the other hand can use AI to help them review and evaluate patches quicker and thus this can be helpful on both sides of the equation. I also found Claude and other AI tools like Gemini pretty good at generating test cases for the code they make, so this is another area where open source patch contributions can improve, by improving test coverage for the code. I do also believe there are many areas where projects can greatly benefit from AI, for instance in the GNOME project a constant challenge for extension developers have been keeping their extensions up-to-date, well I do believe a tool like Claude or Gemini should be able to update GNOME Shell extensions quite easily. So maybe having a service which tries to provide a patch each time there is a GNOME Shell update might be a great help there. At the same time having a AI take a look at updated extensions and giving an first review of the update might help reduce the load on people doing code reviews on extensions and help flag problematic extensions. I know for a lot of cases and situations uploading your code to a webservice like Claude, Gemini or Copilot is not something you want or can do. I know privacy is a big concern for many people in the community. My team at Red Hat has been working on a code assistant tool using the IBM Granite model, called Granite.code. What makes Granite different is that it relies on having the model run locally on your own system, so you don't send your code or data of somewhere else. This of course have great advantages in terms of improving privacy and security, but it has challenges too. The top end AI models out there at the moment, of which Claude is probably the best at the time of writing this blog post, are running on hardware with vast resources in terms of computing power and memory available. Most of us do not have those kind of capabilities available at home, so the model size and performance will be significantly lower. So at the moment if you are looking for a great open source tool to use with VS Code to do things like code completion I recommend giving Granite.code a look. If you on the other hand want to do something like I have described here you need to use something like Claude, Gemini or ChatGPT. I do recommend Claude, not just because I believe them to be the best at it at the moment, but they also are a company trying to hold themselves to high ethical standards. Over time we hope to work with IBM and others in the community to improve local models, and I am also sure local hardware will keep improving, so over time the experience you can get with a local model on your laptop at least has less of a gap than what it does today compared to the big cloud hosted models. There is also the middle of the road option that will become increasingly viable, where you have a powerful server in your home or at your workplace that can at least host a midsize model, and then you connect to that on your LAN. I know IBM is looking at that model for the next iteration of Granite models where you can choose from a wide variety of sizes, some small enough to be run on a laptop, others of a size where a strong workstation or small server can run them or of course the biggest models for people able to invest in top of the line hardware to run their AI. Also the AI space is moving blazingly fast, if you are reading this 6 Months from now I am sure the capabilities of online and local models will have changed drastically already. So to all my friends in the Linux community I ask you to take a look at AI and what it can do and then lets work together on improving it, not just in terms of capabilities, but trying to figure out things like societal challenges around it and sustainability concerns I also know a lot of us got. Whats next for this code As I mentioned I while I felt I got it to a point where I proved to myself it worked, I am not planning on working anymore on it. But I did make a cute little application for internal use that shows a spinning globe with all global Red Hat offices showing up as little red lights and where it pulls Red Hat news at the bottom. Not super useful either, but I was able to use Claude to refactor the globe rendering code from xtraceroute into this in just a few hours. Red Hat Offices Globe and news.

- Tomeu Vizoso: Rockchip NPU update 6: We are in mainline! (2025/07/28 07:02)
  The kernel portion of the Linux driver for the Rockchip NPUs has been merged into the maintainer tree, and will be sent in the next pull request to Linus. The userspace portion of the driver has just been merged as well, in the main Mesa repository.This means that in the next few weeks the two components of the Rocket driver will be in official releases of the Linux and Mesa projects, and Linux distributions will start to pick them up and package. Once that happens, we will have seamless accelerated inference on one more category of hardware.It has been a bit over a year since I started working on the driver, though the actual feature implementation took just over two months of that. The rest of the time was spent waiting for reviews and reacting to excellent feedback from many contributors to the Linux kernel. The driver is now much better because of that frank feedback.What I see in the near future for this driver is support for other Rockchip SoCs and some performance work, to match that of the proprietary driver. But of course, with it being open source, contributors can just start hacking on it and sending patches over for review and merging.I'm now working on further improvements to the Etnaviv driver for the Vivante NPUs, and have started work with Arm engineers on a new driver for their Ethos line of NPUs. So stay tuned for more news on accelerated inference on the edge in mainline Linux!
- Bastien Nocera: Digitising CDs (aka using your phone as an image scanner) (2025/07/27 19:39)
  I recently found, under the rain, next to a book swap box, a pile of 90's “software magazines” which I spent my evening cleaning, drying, and sorting in the days afterwards.Magazine cover CDs with nary a magazine Those magazines are a peculiar thing in France, using the mechanism of “Commission paritaire des publications et des agences de presse” or “Commission paritaire” for short. This structure exists to assess whether a magazine can benefit from state subsidies for the written press (whether on paper at the time, and also the internet nowadays), which include a reduced VAT charge (2.1% instead of 20%), reduced postal rates, and tax exemptions.In the 90s, this was used by Diamond Editions[1] (a publisher related to tech shop Pearl, which French and German computer enthusiasts probably know) to publish magazines with just enough original text to qualify for those subsidies, bundled with the really interesting part, a piece of software on CD.If you were to visit a French newsagent nowadays, you would be able to find other examples of this: magazines bundled with music CDs, DVDs or Blu-rays, or even toys or collectibles. Some publishers (including the infamous and now shuttered Éditions Atlas) will even get you a cheap kickstart to a new collection, with the first few issues (and collectibles) available at very interesting prices of a couple of euros, before making that “magazine” subscription-only, with each issue being increasingly more expensive (article from a consumer protection association).Other publishers have followed suite.I guess you can only imagine how much your scale model would end up costing with that business model (50 eurocent for the first part, 4.99€ for the second), although I would expect them to have given up the idea of being categorised as “written press”.To go back to Diamond Editions, this meant the eventual birth of 3 magazines: Presqu'Offert, BestSellerGames and StratéJ. I remember me or my dad buying a few of those, an older but legit and complete version of ClarisWorks, CorelDraw or a talkie version of a LucasArt point'n'click was certainly a more interesting proposition than a cut-down warez version full of viruses when budget was tight.3 of the magazines I managed to rescue from the rainYou might also be interested in the UK “covertape wars”.Don't stress the technique This brings us back to today and while the magazines are still waiting for scanning, I tried to get a wee bit organised and digitising the CDs.Some of them will have printing that covers the whole of the CD, a fair few use the foil/aluminium backing of the CD as a blank surface, which will give you pretty bad results when scanning them with a flatbed scanner: the light source keeps moving with the sensor, and what you'll be scanning is the sensor's reflection on the CD.My workaround for this is to use a

digital camera (my phone's 24MP camera), with a white foam board behind it, so the blank parts appear more light grey. Of course, this means that you need to take the picture from an angle, and that the CD will appear as an oval instead of perfectly circular.I tried for a while to use GIMP perspective tools, and “Multimedia” Mike Melanson's MobyCAIRO rotation and cropping tool. In the end, I settled on Darktable, which allowed me to do 4-point perspective deskewing, I just had to have those reference points.So I came up with a simple "deskew" template, which you can print yourself, although you could probably achieve similar results with grid paper.My janky setup The resulting picture After opening your photo with Darktable, and selecting the “darkroom” tab, go to the “rotate and perspective tool”, select the “manually defined rectangle” structure, and adjust the rectangle to match the centers of the 4 deskewing targets. Then click on “horizontal/vertical fit”. This will give you a squished CD, don't worry, and select the “specific” lens model and voilà.Tools at the readyTargets acquired Straightened but squishedYou can now export the processed image (I usually use PNG to avoid data loss at each step), open things up in GIMP and use the ellipse selection tool to remove the background (don't forget the center hole), the rotate tool to make the writing straight, and the crop tool to crop it to size.And we're done! The result of this example is available on Archive.org, with the rest of my uploads being made available on Archive.org and Abandonware-Magazines for those 90s magazines and their accompanying CDs.[1]: Full disclosure, I wrote a couple of articles for Linux Pratique and Linux Magazine France in the early 2000s, that were edited by that same company.

- Hari Rana: GNOME Calendar: A New Era of Accessibility Achieved in 90 Days (2025/07/25 00:00)
Note Please consider supporting my effort in making GNOME apps accessible for everybody. Thanks! Liberapay Ko-fi GitHub Sponsors Introduction There is no calendaring app that I love more than GNOME Calendar. The design is slick, it works extremely well, it is touchpad friendly, and best of all, the community around it is just full of wonderful developers, designers, and contributors worth collaborating with, especially with the recent community growth and engagement over the past few years. Georges Stavracas and Jeff Fortin Tam are some of the best maintainers I have ever worked with. I cannot express how thankful I am of Jeff’s underappreciated superhuman capabilities to voluntarily coordinate huge initiatives and issue trackers. One of Jeff’s many initiatives is gnome-calendar#1036: the accessibility initiative, which is a big and detailed list of issues related to accessibility. In my opinion, GNOME Calendar’s biggest problem was the lack of accessibility support, which made the app completely unusable for people exclusively using a keyboard, or people relying on assistive technologies. This article will explain in details about the fundamental issues that held back accessibility in GNOME Calendar since the very beginning of its existence (12 years at a minimum), the progress we have made with accessibility as well as our thought process in achieving it, and the now and future of accessibility in GNOME Calendar. Calendaring Complications Note On a desktop or tablet form factor, GNOME Calendar has a month view and a week view, both of which are a grid comprising of cells representing a time frame. In the month view, each row is a week, and each cell is a day. In the week view, the time frame within cells varies on the zooming level. There are mainly two reasons that made GNOME Calendar inaccessible: firstly, GTK’s accessibility tree does not cover the logically and structurally complicated workflow and design that is a typical calendaring app; and secondly, the significant negative implications of accessibility due to reducing as much overhead as possible. Accessibility Trees Are Insufficient for Calendaring Apps GTK’s accessibility tree, or rather any accessibility tree, is rendered insufficient for calendaring apps, mainly because events are extremely versatile. Tailoring the entire interface and experience around that versatility pushes us to explore alternate and custom structures. Events are highly flexible, because they are time-based. An event can last a couple of minutes, but it can as well last for hours, days, weeks, or even months. It can start in the middle of a day and end on the upcoming day; it can start by the end of a week and end at the beginning of the upcoming week. Essentially, events are limitless, just like time itself. Since events can last more than a day, cell widgets cannot

hold a meaningful link with event widgets, because otherwise event widgets would not be capable of spanning across cells. As such, event widgets are overlaid on top of cell widgets and positioned based on the coordinates, width, and height of each widget. As a consequence, the visual representation of GNOME Calendar is fundamentally incompatible with accessibility trees. GNOME Calendar’s month and week views are visually 2.5 dimensional: A grid layout by itself is structurally two-dimensional, but overlaying event widgets that is capable of spanning across cells adds an additional layer. Conversely, accessibility trees are fundamentally two-dimensional, so GNOME Calendar’s visual representation cannot be sufficiently adapted into a two-dimensional logical tree. In summary, accessibility trees are insufficient for calendaring apps, because the versatility and high requirements of events prevents us from linking cell widgets with event widgets, so event widgets are instead overlaid on top, consequently making the visual representation 2.5 dimensional; however, the additional layer makes it fundamentally impossible to adapt to a two-dimensional accessibility tree. Negative Implications of Accessibility due to Maximizing Performance Unlike the majority of apps, GNOME Calendar’s layout and widgetry consist of custom widgets and complex calculations according to several factors, such as: the size of the window; the height and width of each cell widget to figure out if one or more event widgets can perceptibly fit inside a cell; the position of each event widget to figure out where to position the event widget, and where to reposition all the event widgets around it if necessary; what went wrong in my life to work on a calendaring app written in C. Due to these complex calculations, along with the fact that it is also possible to have tens, hundreds, or even thousands of events, nearly every calendar app relies on maximizing performance as much as possible, while being at the mercy of the framework or toolkit. Furthermore, GNOME Calendar supports smooth scrolling and kinetic scrolling, so each event and cell widget’s position needs to be recalculated for every pixel when the user scrolls or swipes with a mouse or touchpad. One way to minimize that problem is by creating custom widgets that are minimal and only fulfill the purpose we absolutely need. However, this comes at the cost of needing to reimplement most functionality, including most, if not all accessibility features and semantics, such as keyboard focus, which severely impacted accessibility in GNOME Calendar. While GTK’s widgets are great for general purpose use-cases and do not have any performance impact with limited instances of them, performance starts to deteriorate on weaker systems when there are hundreds, if not thousands of instances in the view, because they contain a lot of functionality that event widgets may not need. In the case of the GtkButton widget, it has a custom multiplexer, it applies different styles for different child types, it implements the GtkActionable interface for custom actions, and more technical characteristics. Other functionality-based widgets will have more capabilities that might impact performance with hundreds of instances. To summarize, GNOME Calendar reduces overhead by creating minimal custom widgets that fulfill a specific purpose. This unfortunately severely impacted accessibility throughout the app and made it unusable with a keyboard, as some core functionalities, accessibility features and semantics were never (re)implemented. Improving the Existing Experience Despite being inaccessible as an app altogether, not every aspect was inaccessible in GNOME Calendar. Most areas throughout the app worked with a keyboard and/or assistive technologies, but they needed some changes to improve the experience. For this reason, this section is reserved specifically for mentioning the aspects that underwent a lot of improvements. Improving Focus Rings The first major step was to improve the focus ring situation throughout GNOME Calendar. Since the majority of widgets are custom widgets, many of them require to manually apply focus rings. !563 addresses that by declaring custom CSS properties, to use as a base for focus rings. !399 tweaks the style of the reminders popover in the event editor dialog, with the addition of a focus ring. We changed the behavior of the event notes box under the “Notes” section in the event editor dialog. Every time the user focuses on the event notes box, the focus ring appears and outlines the entire box until the user leaves focus. This was accomplished by subclassing AdwPreferencesRow to inherit its style, then applying the .focused class whenever the user focuses on the notes. Improving the Calendar Grid

Note The calendar grid is a 7×6 grid of buttons representing each day. The horizontal axis represents the day of the week, and the vertical axis represents the week number. The calendar grid on the sidebar suffered from several issues when it came to keyboard navigation, namely: pressing ↹ would focus the next cell in the grid up until the last cell; when out of bounds, there would be no auditory feedback; on the last row, pressing ↓ would focus a blank element; and pressing → in left-to-right languages, or ← in right-to-left languages, on the last column would move focus to a completely different widget. While the calendar grid can be interacted with a keyboard, the keyboard experience was far from desired. !608 addresses these issues by overriding the Gtk.Widget.focus () virtual method. Pressing ↹ or Shift+↹ skips the entire grid, and the grid is wrapped to allow focusing between the first and last columns with ← and →, while notifying the user when out of bounds. Improving the Calendar List Box Note The calendar list box holds a list of available calendars, all of which can be displayed or hidden from the week view and month view. Each row is a GtkListBoxRow that holds a GtkCheckButton. The calendar list box had several problems in regards to keyboard navigation and the information each row provided to assistive technologies. The user was required to press ↹ a second time to get to the next row in the list. To elaborate: pressing ↹ once focused the row; pressing it another time moved focus to the check button within the row (bad); and finally pressing the third time focused the next row. Row widgets had no actual purpose besides toggling the check button upon activation. Similarly, the only use for a check button widget inside each row was to display the “check mark” icon if the calendar was displayed. This meant that the check button widget held all the desired semantics, such as the “checkbox” role and the “checked” state; but worst of all, it was getting focus. Essentially, the check button widget was handling responsibilities that should have been handled by the row. Both inconveniences were addressed by !588. The check button widget was replaced with a check mark icon using GtkImage, a widget that does not grab focus. The accessible role of the row widget was changed to “checkbox”, and the code was adapted to handle the “checked” state. Implementing Accessibility Functionality Accessibility is often absolute: there is no ‘in-between’ state; either the user can access functionality, or they cannot, which can potentially make the app completely unusable. This section goes in depth with the widgets that were not only entirely inaccessible but also rendered GNOME Calendar completely unusable with a keyboard and assistive technology. Making the Event Widget Accessible Note GcalEventWidget, the name of the event widget within GNOME Calendar, is a colored rectangular toggle button containing the summary of an event. Activating it displays a popover that displays additional detail for that event. GcalEventWidget subclasses GtkWidget. The biggest problem in GNOME Calendar, which also made it completely impossible to use the app with a keyboard, was the lack of a way to focus and activate event widgets with a keyboard. Essentially, one would be able to create events, but there would be no way to access them in GNOME Calendar. Quite literally, this entire saga began all thanks to a dream I had, which was to make GcalEventWidget subclass GtkButton instead of GtkWidget directly. The thought process was: GtkButton already implements focus and activation with a keyboard, so inheriting it should therefore inherit focus and activation behavior. In merge request !559, the initial implementation indeed subclassed GtkButton. However, that implementation did not go through, due to the reason outlined in § Negative Implications of Accessibility due to Maximizing Performance. Despite that, the initial implementation instead significantly helped us figure out exactly what were missing with GcalEventWidget: specifically, setting Gtk.Widget:receives-default and Gtk.Widget:focusable properties to “True”. Gtk.Widget:receives-default makes it so the widget can be activated how ever desired, and Gtk.Widget:focusable allows it to become focusable with a keyboard. So, instead of subclassing GtkButton, we instead reimplemented GtkButton’s functionality in order to maintain performance. While preliminary support for keyboard navigation was added into GcalEventWidget, accessible semantics for assistive technologies like screen readers were severely lacking. This was addressed by !587, which sets the role to “toggle-button”, to convey that GcalEventWidget is a toggle button. The merge request also indicates that the widget has a popup for the event popover, and has the means to update the “pressed” state of the widget. In summary, we first made GcalEventWidget accessible with a keyboard by reimplementing some of GtkButton’s functionality. Then, we later added the means to appropriately convey information to assistive technologies. This was the worst offender, and was the primary reason why GNOME Calendar was unusable with a keyboard, but we finally managed to solve it! Making the Month and Year Spin Buttons Accessible Note GcalMultiChoice is the name of the custom spin button widget used for displaying and cycling through months and/or years. It comprises of a “decrement” button to the start, a flat toggle button in the middle that contains a label that displays the value, and an “increment” button to the end. Only the button in the middle can gain keyboard focus throughout GcalMultiChoice. In some circumstances, GcalMultiChoice can display a popover for increased granularity. GcalMultiChoice was not interactable with a keyboard, because: it did not react to ↑ and ↓ keys; and the “decrement” and “increment” buttons were not focusable. For a spin button widget, the “decrement” and “increment” buttons should generally remain unfocusable, because ↑ and ↓ keys already accomplish that behavior. Furthermore, GtkSpinButton’s “increment” (+) and “decrement” (-) buttons are not focusable either, and the Date Picker Spin Button Example by the ARIA Authoring Practices Guide (APG) avoids that functionality as well. However, since GcalMultiChoice did not react to ↑ and ↓ keys, having the “decrement” and “increment” buttons be focusable would have been a somewhat acceptable workaround. Unfortunately, since those buttons were not focusable, and ↑ and ↓ keys were not supported, it was impossible to increment or decrement values in GcalMultiChoice with a keyboard without resorting to workarounds. Additionally, GcalMultiChoice lacked the semantics to communicate with assistive technologies. So, for example, a screen reader would never say anything meaningful. All of the above problems remained problems until merge request !603. For starters, it implements GtkAccessible and GtkAccessibleRange, and then implements keyboard navigation. Implementing GtkAccessible and GtkAccessibleRange The merge request implements the GtkAccessible interface to retrieve information from the flat toggle button. Fundamentally, since the toggle button was the only widget capable of gaining keyboard focus throughout GcalMultiChoice, this caused two distinct problems. The first issue was that assistive technologies only retrieved semantic information from the flat toggle button, such as the type of widget (accessible role), its label, and its description. However, the toggle button was semantically just a toggle button; since it contained semantics and provided information to assistive technologies, the information it provided was actually misleading, because it only provided information as a toggle button, not a spin button! So, the solution to this is to strip the semantics from the flat toggle button. Setting its accessible role to “none” makes assistive technologies ignore its information. Then, setting the accessible role of the top-level (GcalMultiChoice) to “spin-button” gives semantic meaning to assistive technologies, which allows the widget to appropriately convey these information, when focused. This led to the second issue: Assistive technologies only retrieved information from the flat toggle button, not from the top-level. Generally, assistive technologies retrieve information from the focused widget. Since the toggle button was the only widget capable of gaining focus, it was also the only widget providing information to them; however, since its semantics were stripped, it had no information to share, and thus assistive technologies would retrieve absolutely nothing. The solution to this is to override the Gtk.Accessible.get_platform_state () virtual method, which allows us to bridge communication between the states of child widgets and the top-level widget. In this case, both GcalMultiChoice and the flat toggle button share the state—if the flat toggle button is focused, then GcalMultiChoice is considered focused; and since GcalMultiChoice is focused, assistive technologies can then retrieve its information and state. The last issue that needed to be addressed was that GcalMultiChoice was still not providing any of the values to assistive technologies. The solution to this is straightforward: implementing the GtkAccessibleRange interface, which makes it necessary to set values for the following accessible properties: “value-max”, “value-min”, “value-now”, and “value-text”. After all this effort, GcalMultiChoice now provides correct semantics to assistive technologies. It appropriately reports its role, the current textual value, and whether it contains a popover. To summarize: The flat toggle button was the only widget conveying information to assistive technologies, as it was the only widget capable of gaining focus and providing semantic information. To solve this, its semantics were stripped away. The top-level, being GcalMultiChoice, was assigned the "spin-button" role to provide semantics; however, it was still incapable of providing information to assistive technologies, because it was never getting focused. To solve this, the state of the toggle button, including the focused state, carried over to the top-level to allow assistive technologies to retrieve information from the top-level. GcalMultiChoice still did not provide its values to assistive technologies. This is solved by implementing the GtkAccessibleRange interface. Providing Top-Level Semantics to a Child Widget As Opposed to the Top-Level Widget Is Discouraged As you read through the previous section, you may have asked yourself: "Why go through all of those obstacles and complications when you could have just re-assigned the flat toggle button as "spin-button" and not worry about the top-level's role and focus state?" Semantics should be provided by the top-level, because they are represented by the top-level. What makes GcalMultiChoice a spin button is not just the flat toggle button, but it is the combination of all the child widgets/objects, event handlers (touch, key presses, and other inputs), accessibility attributes (role, states, relationships), widget properties, signals, and other characteristics. As such, we want to maintain that consistency for practically everything, including the state. The only exception to this is widgets whose sole purpose is to contain one or more elements, such as GtkBox. This is especially important for when we want it to communicate with other widgets and APIs, such as the Gtk.Widget::state-flags-changed signal, the Gtk.Widget.is_focus () method, and other APIs where it is necessary to have the top-level represent data accurately and behave predictably. In the case of GcalMultiChoice, we set accessible labels at the top-level. If we were to re-assign the flat toggle button's role as "spin-button", and set the accessible label to the top-level, assistive technologies would only retrieve information from the toggle button while ignoring the labels defined at the top-level. For the record, GtkSpinButton also overrides Gtk.Accessible.get_platform_state (): static gboolean gtk_spin_button_accessible_get_platform_state (GtkAccessible *self, GtkAccessiblePlatformState state) { return gtk_editable_delegate_get_accessible_platform_state (GTK_EDITABLE (self), state); } static void gtk_spin_button_accessible_init (GtkAccessibleInterface *iface) { … iface->get_platform_state = gtk_spin_button_accessible_get_platform_state; } To be fair, assigning the "spin-button" role to the flat toggle button is unlikely to cause major issues, especially for an app. Re-assigning the flat toggle button was my first instinct. The initial implementation did just that as well. I was completely unaware of the Gtk.Accessible.get_platform_state () virtual method before finalizing the merge request, so I initially thought that was the correct way to do. Even if the toggle button had the "spin-button" role instead of the top-level, it would not have stopped us from implementing workarounds, such as a getter method that retrieves the flat toggle button that we can then use to manipulate it. In summary, we want to provide semantics at the top-level, because they are structurally part of it. This comes with the benefit of making the widget easier to work with, because APIs can directly communicate with it, instead of resorting to workarounds. The Now and Future of Accessibility in GNOME Calendar All these accessibility improvements will be available on GNOME 49, but you can download and install the pre-release on the "Nightly GNOME Apps" DLC Flatpak remote on nightly.gnome.org. In the foreseeable future, I want to continue working on !564, to make the month view itself accessible with a keyboard, as seen in the following: A screen recording demoing keyboard navigation within the month view. Focus rings appear and disappear as the user moves focus between cells. Going out of bounds in the vertical axis scrolls the view to the direction, and going out of bounds in the horizontal axis moves focus to the logical sibling. However, it is already adding 640 lines of code, and I can only see it increasing overtime. We also want to make cells in the week view accessible, but this will also be a monstrous merge request, just like the above merge request. Most importantly, we want (and need) to collaborate and connect with people who rely on assistive technologies to use their computer, especially when everybody working on GNOME Calendar does not rely on assistive technologies themselves. Conclusion I am overwhelmingly satisfied of the progress we have made with accessibility on GNOME Calendar in six months. Just a year ago, if I was asked about what needs to be done to incorporate accessibility features in GNOME Calendar, I would have shamefully said “dude, I don’t know where to even begin”; but as of today, we somehow managed to turn GNOME Calendar into an actual, usable calendaring app for people who rely on assistive technologies and/or a keyboard. Since this is still Disability Pride Month, and GNOME 49 is not out yet, I encourage you to get the alpha release of GNOME Calendar on the “Nightly GNOME Apps” Flatpak remote at nightly.gnome.org. The alpha release is in a state where the gays with disabilities can organize and do crimes using GNOME Calendar /j

- Dave Airlie (blogspot): ramalama/mesa : benchmarks on my hardware and open source vs proprietary (2025/07/24 22:19)
One of my pet peeves around running local LLMs and inferencing is the sheer mountain of shit^W^W^W complexity of compute stacks needed to run any of this stuff in an mostly optimal way on a piece of hardware.CUDA, ROCm, and Intel oneAPI all to my mind scream over-engineering on a massive scale at least for a single task like inferencing. The combination of closed source, over the wall open source, and open source that is insurmountable for anyone to support or fix outside the vendor, screams that there has to be a simpler way. Combine that with the pytorch ecosystem and insanity of deploying python and I get a bit unstuck.What can be done about it?llama.cpp to me seems like the best answer to the problem at present, (a rust version would be a personal preference, but can't have everything). I like how ramalama wraps llama.cpp to provide a sane container interface, but I'd like to eventually get to the point where container complexity for a GPU compute stack isn't really needed except for exceptional cases.On the compute stack side, Vulkan exposes most features of GPU hardware in a possibly suboptimal way, but with extensions all can be forgiven. Jeff Bolz from NVIDIA's talk at Vulkanised 2025 started to give me hope that maybe the dream was possible.The main issue I have is Jeff is writing driver code for the NVIDIA proprietary vulkan driver which reduces complexity but doesn't solve my open source problem.Enter NVK, the open source driver for NVIDIA GPUs. Karol Herbst and myself are taking a look at closing the feature gap with the proprietary one. For mesa 25.2 the initial support for VK_KHR_cooperative_matrix was landed, along with some optimisations, but there is a bunch of work to get VK_NV_cooperative_matrix2 and a truckload of compiler optimisations to catch up with NVIDIA.But since mesa 25.2 was coming soon I wanted to try and get some baseline figures out.I benchmarked on two systems (because my AMD 7900XT wouldn't fit in the case). Both Ryzen CPUs. The first I used system I put in an RTX5080 then a RTX6000 Ada and then the Intel A770. The second I used for the RX7900XT. The Intel SYCL stack failed to launch unfortunately inside ramalama and I hacked llama.cpp to use the A770 MMA accelerators. ramalama bench hf://unsloth/Qwen3-8B-GGUF:UD-Q4_K_XL I picked this model at random, and I've no idea if it was a good idea. Some analysis:The token generation workload is a lot less matmul heavy than prompt processing, it also does a lot more synchronising. Jeff has stated CUDA wins here mostly due to CUDA graphs and most of the work needed is operation fusion on the llama.cpp side. Prompt processing is a lot more matmul heavy, extensions like NV_coopmat2 will help with that (NVIDIA vulkan already uses it in the above), but there may be further work to help close the CUDA gap. On AMD radv (open source) Vulkan is already better at TG than ROCm, but behind in prompt processing. Again coopmat2 like extensions should help close the gap there.NVK is starting from a fair way behind, we just pushed support for the most basic coopmat extension and we know there is a long way to go, but I think most of it is achievable as we move forward and I hope to update with new scores on a semi regular basis. We also know we can definitely close the gap on the NVIDIA proprietary Vulkan driver if we apply enough elbow grease and register allocation :-)I think it might also be worth putting some effort into radv coopmat2 support, I think if radv could

overtake ROCm for both of these it would remove a large piece of complexity from the basic users stack.As for Intel I've no real idea, I hope to get their SYCL implementation up and running, and maybe I should try and get my hands on a B580 card as a better baseline. When I had SYCL running once before I kinda remember it being 2-4x the vulkan driver, but there's been development on both sides. (The graphs were generated by Gemini.)
- Simon Ser: Status update, July 2025 (2025/07/18 22:00)
Hi! Sway’s patch to add HDR support has finally be merged! It can be enabled via output <name> hdr on, and requires the Vulkan renderer (which can be selected via WLR_RENDERER=vulkan). Still, lots remains to be done to improve tone mapping and compositing. Help is welcome if you have ideas! I’ve also added support for toplevel tags to Sway. Toplevel tags provide a stable key to select a particular window of a multi-window application: for instance, the favorites window of a Web browser might carry the tag “favorites”. Once support is added to various clients, this should be a more robust way to target windows than title regular expressions. David Turner has contributed support for color-representation-v1 to wlroots. Thanks to this protocol, clients can describe how channels of a buffer should be interpreted, in particular pre-multiplied vs. straight alpha, YUV matrix coefficients and full vs. limited range. The renderer and backends bits haven’t been merged yet, but work-in-progress patches have been posted. Wayland 1.24 has been released, with a new global to free up wl_registry objects, a new “repeated” state for keyboard keys, and other new utility functions. grim 1.5 adds support for ext-image-copy-capture-v1, the new screen capture protocol. grim can now capture individual toplevel windows. In IRC news, accessibility for the gamja Web client has been improved with ARIA attributes. A pending patch for Goguma adds support for user and channel avatars (via the metadata-2 extension). I’ve sent a draft for a new user query extension to synchronize opened direct message conversations across clients. Last, qugu2427 has contributed go-smtp support for DELIVERBY (to ask the server to deliver a message before a timestamp) and MT-PRIORITY (to indicate a priority level for a message). See you next month!
- Sebastian Wick: Blender HDR and the reference white issue (2025/07/13 14:26)
The latest alpha of the upcoming Blender 5.0 release comes with High Dynamic Range (HDR) support for Linux on Wayland which will, if everything works out, make it into the final Blender 5.0 release on October 1, 2025. The post on the developer forum comes with instructions on how to enable the experimental support and how to test it. If you are using Fedora Workstation 42, which ships GNOME version 48, everything is already included to run Blender with HDR. All that is required is an HDR compatible display and graphics driver, and turning on HDR in the Display Settings. It’s been a lot of personal blood, sweat and tears, paid for by Red Hat across the Linux graphics stack for the last few years to enable applications like Blender to add HDR support. From kernel work, like helping to get the HDR mode working on Intel laptops, and improving the Colorspace and HDR_OUTPUT_METADATA KMS properties, to creating a new library for EDID and DisplayID parsing, and helping with wiring things up in Vulkan. I designed the active color management paradigm for Wayland compositors, figured out how to properly support HDR, created two wayland protocols to let clients and compositors communicate the necessary information for active color management, and created documentation around all things color in FOSS graphics. This would have also been impossible without Pekka Paalanen from Collabora and all the other people I can’t possibly list exhaustively. For GNOME I implemented the new API design in mutter (the GNOME Shell compositor), and helped my colleagues to support HDR in GTK. Now that everything is shipping, applications are starting to make use of the new functionality. To see why Blender targeted Linux on Wayland, we will dig a bit into some of the details of HDR! HDR, Vulkan and the reference white level Blender’s HDR implementation relies on Vulkan’s VkColorSpaceKHR, which allows applications to specify the color space of their swap chain, enabling proper HDR rendering pipeline integration. The key color space in question is VK_COLOR_SPACE_HDR10_ST2084_EXT, which corresponds to the HDR10

standard using the ST.2084 (PQ) transfer function. However, there’s a critical challenge with this Vulkan color space definition: it has an undefined reference white level. Reference white indicates the luminance or a signal level at which a diffuse white object (such as a sheet of paper, or the white parts of a UI) appears in an image. If images with different reference white levels end up at different signal levels in a composited image, the result is that “white” in one of the images is still being perceived as white, while the “white” from the other image is now being perceived as gray. If you ever scrolled through Instagram on an iPhone or played an HDR game on Windows, you will probably have noticed this effect. The solution to this issue is called anchoring. The reference white level of all images needs to be normalized in order for “white” ending up on the same signal level in the composited image. Another issue with the reference white level specific to PQ is the prevalent myth, that the absolute luminance of a PQ signal must be replicated on the actual display a user is viewing the content at. PQ is a bit of a weird transfer characteristic because any given signal level corresponds to an absolute luminance with the unit cd/m² (also known as nit). However, the absolute luminance is only meaningful for the reference viewing environment! If an image is being viewed in the reference viewing environment of ITU-R BT.2100, (essentially a dark room) and the image signal of 203 nits is being shown at 203 nits on the display, it makes the image appear as the artist intended. The same is not true when the same image is being viewed on a phone with the summer sun blasting on the screen from behind. PQ is no different from other transfer characteristics in that the reference white level needs to be anchored, and that the anchoring point does not have to correspond to the luminance values that the image encodes. Coming back to the Vulkan color space VK_COLOR_SPACE_HDR10_ST2084_EXT definition: “HDR10 (BT2020) color space, encoded according to SMPTE ST2084 Perceptual Quantizer (PQ) specification”. Neither ITU-R BT.2020 (primary chromaticity) nor ST.2084 (transfer characteristics), nor the closely related ITU-R BT.2100 define the reference white level. In practice, the reference level of 203 cd/m² from ITU-R BT.2408 (“Suggested guidance for operational practices in high dynamic range television production”) is used. Notable, this is however not specified in the Vulkan definition of VK_COLOR_SPACE_HDR10_ST2084_EXT. The consequences of this? On almost all platforms, VK_COLOR_SPACE_HDR10_ST2084_EXT implicitly means that the image the application submits to the presentation engine (what we call the compositor in the Linux world) is assumed to have a reference white level of 203 cd/m², and the presentation engine adjusts the signal in such a way that the reference white level of the composited image ends up at a signal value that is appropriate for the actual viewing environment of the user. On GNOME, the way to control this currently is the “HDR Brightness” slider in the Display Settings, but will become the regular screen brightness slider in the Quick Settings menu. On Windows, the misunderstanding that a PQ signal value must be replicated one to one on the actual display has been immortalized in the APIs. It was only until support for HDR was added to laptops that this decision was revisited, but changing the previous APIs was already impossible at this point. Their solution was exposing the reference white level in the Win32 API and tasking applications to continuously query the level and adjust the image to match the new level. Few applications actually do this, with most games providing a built-in slider instead. The reference white level of VK_COLOR_SPACE_HDR10_ST2084_EXT on Windows is essentially a continuously changing value that needs to be queried from Windows APIs outside of Vulkan. This has two implications: It is impossible to write a cross-platform HDR application in Vulkan (if Windows is one of the targets) On Wayland, a “standard” HDR signal can just be passed on to the compositor, while on Windows, more work is required While the cross-platform issue is solvable, and something we’re working on, the way Windows works also means that the cross-platform API might become harder to use because we cannot change the underlying Windows mechanisms. No Windows Support The result is that Blender currently does not support HDR on Windows. <figcaption> Jeroen-Bakker explaining the lack of Windows support </figcaption> The design of the Wayland color-management protocol, and the resulting active color-management paradigm of Wayland compositors was a good choice, making it easy for developers to do the right thing, while also giving them more control if they so chose. Looking forward We have managed to transition the compositor model from a dumb blitter to a component which takes an active part in color management, we have image viewers and video players with HDR support and now we have tools for producing HDR content! While it is extremely exciting for me that we have managed to do this properly, we also have a lot of work ahead of us, some of which I will hopefully tell you about in a future blog post!

- Hans de Goede: Recovering a FP2 which gives "flash write failure" errors (2025/07/04 16:19)
  This blog post describes my successful os re-install on a fairphone 2 which was giving "flash write failure" errors when flashing it with fastboot, with the flash_FP2_factory.sh script. I'm writing down my recovery steps for this in case they are useful for anyone else.I believe that this is caused by the bootloader code which implements fastboot not having the ability to retry recoverable eMMC errors. It is still possible to write the eMMC from Linux which can retry these errors.So we can recover by directly fastboot-ing a recovery.img and then flashing things over adb.(See step by step instructions...) comments
- Dave Airlie (blogspot): nvk: blackwell support (2025/07/01 10:20)
  Blog posts are like buses sometimes...I've spent time over the last month enabling Blackwell support on NVK, the Mesa vulkan driver for NVIDIA GPUs. Faith from Collabora, the NVK maintainer has cleaned up and merged all the major pieces of this work and landed them into mesa this week. Mesa 25.2 should ship with a functioning NVK on blackwell. The code currently in mesa main passes all tests in the Vulkan CTS.Quick summary of the major fun points:Ben @ NVIDIA had done the initial kernel bringup in to r570 firmware in the nouveau driver. I worked with Ben on solidifying that work and ironing out a bunch of memory leaks and regressions that snuck in.Once the kernel was stable, there were a number of differences between Ada and Blackwell that needed to be resolved. Thanks to Faith, Mel and Mohamed for their help, and NVIDIA for providing headers and other info.I did most of the work on a GB203 laptop and a desktop 5080.1. Instruction encoding: a bunch of instructions changed how they were encoded. Mel helped sort out most of those early on.2. Compute/QMD: the QMD which is used to launch compute shaders, has a new encoding. NVIDIA released the official QMD headers which made this easier in the end.3. Texture headers: texture headers were encoded different from Hopper on, so we had to use new NVIDIA headers to encode those properly4. Depth/Stencil: NVIDIA added support for separate d/s planes and this also has some knock on effects on surface layouts. 5. Surface layout changes. NVIDIA attaches a memory kind to memory allocations, due to changes in Blackwell, they now use a generic kind for all allocations. You now longer know the internal bpp dependent layout of the surfaces. This means changes to the dma-copy engine to provide that info. This means we have some modifier changes to cook with NVIDIA over the next few weeks at least for 8/16 bpp surfaces. Mohamed helped get this work and host image copy support done.6. One thing we haven't merged is bound texture support. Currently blackwell is using bindless textures which might be a little slower. Due to changes in the texture instruction encoding, you have to load texture handles to intermediate uniform registers before using them as bound handles. This causes a lot of fun with flow control and when you can spill uniform registers. I've written a few efforts at using bound textures, so we understand how to use them, just have some compiler issues to maybe get it across the line.7. Proper instruction scheduling isn't landed yet. I have a spreadsheet with all the figures, and I started typing, so will try and get that into an MR before I take some holidays.
- Dave Airlie (blogspot): radv: VK_KHR_video_encode_av1 support (2025/07/01 09:27)
  I should have mentioned this here a week ago. The Vulkan AV1 encode extension has been out for a while, and I'd done the initial work on enabling it with radv on AMD GPUs. I then left it in a branch, which Benjamin from AMD picked up and fixed a bunch of bugs, and then we both got distracted. I realised when doing VP9 that it hasn't landed, so did a bit of cleanup. Then David from AMD picked it up and carried it over the

last mile and it got merged last week.So radv on supported hw now supports all vulkan decode/encode formats currently available.