Sceneri
Samsung

Vulkan for the masses
Building game engines for the next generation of creators
Vulkan is Ready

- Drivers are reaching maturity, improved validation
  Still some way to go for low end devices
- Exynos GPU brings solid Vulkan support
  Supports Ray Tracing, Variable Rate Shading, updatable drivers
- Making games on Android is still hard
  If you need help, get in touch! :)
- Working with Google on a new GPU Profiler for Android
GPU Profiler

- Based on Perfetto
- System Trace a la Android GPU Inspector
- Plug-in architecture to support wide variety of hardware
- Wraps GFXReconstruct to record and replay Vulkan calls
Samsung and Sceneri

- Deep Technical Collaboration
- Bridge the gap with Desktop and Console
- Expand Vulkan Reach
- Ray Tracing!
Team Origins

- Modding & demo scene
- Crytek, Massive, Avalanche and more
- Indie to AAA
- More than anything.. a passion to innovate
Sceneri

- One renderer, for 2D, 3D & UI
- Minimal footprint & load times
- Fully asynchronous
- Exclusive to “next-gen” graphics APIs
- Create on any device, from anywhere
- Scale from low to high-end graphics
- Assets streamed in from any source (disk, http etc)
- Instant publishing
Demo highlights

- Runs on Galaxy S24 w. new chip
- Ray-traced shadows & reflections
- Performant thanks to FSR & VRS
- Loads near-instantaneously
- Temporal Anti-Aliasing
- PBR, IBL & SSAO
- 100% dynamic, can dive in and edit in real-time on device
- Super quick iteration times
How did we build it?

Job system: Fully asynchronous from day one.
No main thread. One worker per logical core.

Rendering architecture: Vulkan first
General purpose low level API that helps guide engine architecture

One unified renderer to serve every platform with UI, 2D & 3D
Even more important with a smaller team, plus keeps performance in-house

More than anything, a constant mindset of leaving no stone unturned
“Why?” is the best question to ask. Working with restrictions leads to awesome discoveries
Job System

- One Job Runner per logical core
- Cheating on Vulkan using thread pool & vkWaitForFences (room for improvement!)
- std::condition_variable for battery life when threads are idle.
- Job sharing, not job stealing! Early design choice to fight contention
  - Share work to idle & less tasked threads before starting execution
  - Common API design based on Vulkan, Metal, WebGPU core
Job Runner Render Data

- Per runner resources
  - Descriptor pool
    - VkDescriptorPoolSize depending on type of runner (high priority, efficiency etc)
  - Command pool
    - Fast command buffer access
  - Per frame command pool
    - \( \rightarrow \) vkResetCommandPool at end of frame
    - No need for thread safety \( \rightarrow \) fast!
- Memory Pool
  - Cheap lock-free allocation in thread-only pages / pools
  - 256MB default size at startup (mostly, can be lower), additional space allocated in blocks of 64MB
- Resource deallocation
  - Lock-free is more important than ever for job systems - A stall is time wasted for everyone
Job System & Priorities

- Each job defines a priority from 0 - 255
  - Examples of highest priorities are: End / Start frame, record, submit & present
  - Lower end up being loading & deallocation

- Job priorities split into buckets
  - User Interactive - Per frame rendering, ECS update etc, immediate response
  - User Requests - Button presses, timers etc.
  - User Aware Background - Async loading & networking
  - Background - Unimportant events, low priority deallocations etc ← almost definitely running on efficiency cores

- Each bucket corresponds to job runner thread priority communicated to the OS
  - Note: Currently not pinning to logical core on PC, may revisit but suspecting it’ll mostly apply for consoles.

- Job are high performance, low performance or efficiency
  - Maps to heterogeneous processor scheduling, i.e. run on performance or efficiency core
  - Aiming to be even more dynamic here in cases clusters varying more than 2 or 3 types of cores.
Iteration time

- Asynchronous → near instant startup & load times
- Automatic deployment on devices
  - Web / WebAssembly / WebGPU invaluable to flow
- Quick testing across the board
- Multiple graphics APIs is key to maintaining a flexible & stable renderer

Soon enough.. RT on WebGPU!
Rendering architecture

- Automatic framegraph - Define passes & stages, barriers etc resolved without intervention
- Subpasses! Focus on framegraph helping here. Tiled rendering & mobile references.
  - Framegraph automates a lot of this. LOAD_OP_DONT_CARE, STORE_OP_DONT_CARE etc detected based on use.
- Prefer dynamic rendering if you’re starting with Vulkan today. Soon possible to do subpasses too!
  - VK_KHR_dynamic_rendering_local_read
- Framegraph defines barriers & queue ownership transfers that improve synchronization & overlap
- Multiple queue processing jobs, typically: 1x Graphics, 1x Transfer & 1x Compute
- Exclusivity to newer graphics APIs is key! Currently supporting Vulkan, Metal & WebGPU
Building the framegraph

1. **Describe all pass and attachment descriptions and compile in bulk**
   a. Defines render area, required attachments etc. Order suggests *preferred* execution order
   b. Three types of passes: Render pass (auto-subpass & merging), explicit render pass, & compute

2. **Temporarily store state info for each attachment including**
   a. Subresource range
   b. Current layout **per subresource** (mip, array level etc)
   c. First / current pass index (and index inside that pass)
   d. Current queue family index

3. **If attachment was referenced in a prior pass, insert dependency to imply execution order**
   a. Separating concept of record, submit and execution finished - this helps parallelize the hell out of command recording + simplifies job runner dependencies.

4. **Transition layouts for referenced entries**
   a. AKA insert pipeline barriers, semaphores where needed & queue family ownership transfers

5. **Create passes!**
   a. Asynchronously requests RT, texture, buffer creation / allocation
   b. Merges passes (where possible )& creates framebuffer + VkRenderPass

6. **Request transition to initial layouts** *(can be a no-op)*
   a. Can render to resources again in next frame
Framegraph example

// TODO: Move to data!

m_frameGraph.compile(
  Array {
    Framegraph::RenderPassStageDescription {
      .name = "Draw UI widgets",
      .usage = m_pidgeDrawStage.get(),
      .renderArea = outputArea,
      .colorAttachments = Array{
        Framegraph::ColorAttachmentDescription{
          .identifier = widgetRenderTargetInstanceIdentifier,
          .size = outputArea.GetSize(),
          .mipRange = MipRange(0, 1),
          .arrayRange = ArrayRange(0, 1),
          .clearColor = Math::Color(0.7, 0.7, 0.7, 0.7)
        },
      },
      .depthAttachment = Framegraph::DepthAttachmentDescription{
        .identifier = widgetDepthRenderTargetInstanceIdentifier,
        .size = outputArea.GetSize(),
        .mipRange = MipRange(0, 1),
        .arrayRange = ArrayRange(0, 1),
        .clearValue = DepthValue(0.7)
      }
    },
    Framegraph::RenderPassStageDescription {
      .name = "Copy UI to screen",
      .usage = m_pidgeCopyToScreenStage.get(),
      .renderArea = outputArea,
      .colorAttachments = Array{
        Framegraph::ColorAttachmentDescription{
          .identifier = renderOutputInstanceIdentifier,
          .size = outputArea.GetSize(),
          .mipRange = MipRange(0, 1),
          .arrayRange = ArrayRange(0, 1)
        }
      },
      .externalInputAttachments = Array{
        Framegraph::InputAttachmentDescription{
          .identifier = widgetRenderTargetInstanceIdentifier,
          .size = outputArea.GetSize(),
          .sourceImage = ImageSourceRanges{
            .color = ImageSourceRanges::Color,
            .mipRange = MipRange(0, 1),
            .arrayRange = ArrayRange(0, 1)
          }
        }
      }
    }
  }
);
Minimal API wrapper

- Optimizing for writing new functionality fast more than performance
  - Low-level belongs in a place hidden to most developers, not leaking through
- Find the most common ground between APIs
  - Being “next-gen” API exclusive is extremely important to drive decisions
  - Never ending journey. New features, deprecations etc → revisit API. This is a good thing!
- Resource loading details are not important, abstract away from the developer
  - Especially when it’s all asynchronous.
- Ensure parallelization at all steps
  - Avoid potentially blocking operations like the plague, defer and come back later.
- Move more boilerplate pipeline setup, shaders, framegraph etc to data
  - Enables extensive automatic optimization that everyone wants to do manually but won’t
  - The dream: Dynamic knowledge of full scene and automatic top to bottom optimization based on workload
Rendering architecture (per frame)
Raytracing & Indirect Draw

- Mesh, material (aka pipeline), material instance & texture cache define storage buffers & dynamic descriptor sets
  - Each buffer: VK_BUFFER_USAGE_STORAGE_BUFFER_BIT
  - Contiguous array of addresses
    - VkPhysicalDeviceVulkan12Features::runtimeDescriptorArray
    - VkPhysicalDeviceVulkan12Features::descriptorIndexing
    - VkPhysicalDeviceVulkan12Features::descriptorBindingPartiallyBound
    - VkPhysicalDeviceVulkan12Features::bufferDeviceAddress
    - vkGetBufferDeviceAddress
- On asset (i.e. a mesh) loaded: transfer buffer addresses → addresses buffer
  - Use staging buffers (make sure to create a pool and re-use!)
- Instance simply needs an index into the asset data buffer
Acceleration Structure Stage

- Responsible for building bottom and top level acceleration structure
  - **Bottom** = asset data = meshes, textures etc
    - VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR
    - VK_GEOMETRY_TYPE_TRIANGLES_KHR
      - Also possible to do bounding boxes, currently unused in our case
  - **Top** = instance data (references asset data)
    - VK_GEOMETRY_TYPE_INSTANCES_KHR
    - VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR
    - VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR
      - Even in our extreme case of 100% dynamic creation, fast trace is better than optimizing for build times.
      - Bottom / geometry should barely ever change after creation anyway.
  - VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR
    - Note: Performance penalties for partial update in our case, may be better now.
Super Resolution

- Based on AMD FSR1 and tweaked by Samsung
- Leveraged Sceneri’s rendering architecture to scale down all relevant render passes before FSR
- Most passes rendered @ lower resolution
  - Deferred Lighting (g-buffer + lighting + sky)
  - SSAO
  - Compositing & tone mapping
  - Benefits fragment heavy workloads the most
- Working on FSR2/3 on mobile
Optimized for mobile

- Original shader does the analysis 4 times
  - Bilinear interpolation on each input of a 12-tap Lanczos filter
- 'Mobile' / lite version only does it once!
  - Interpolates the inputs with fp16 instead of fp32
- Negligible impact on IQ for mobile
- Up to 40% reduction in GPU time on Xclipse 940
  - Down to ~1.4ms
Native

0.5x

60%

performance uplift (frame rate)
Scaling the renderer & future

- **Shader Graph & deep framegraph integration**
  - Pipeline libraries, SPIR-V linking and more!
- **DX12**
  - Not too important for us, Vulkan is great on Windows. Will wait until console is more relevant.
- **More indirect draw**
- **Descriptor buffers** (VK_EXT_descriptor_buffer)
  - Vulkan held us back but now getting better!
- **Bindless!**
- **No longer using MoltenVK.**
  - But.. we couldn’t be where we’re at without them. Amazing starting point, strongly recommend!
  - Starting with Metal is possible, but would likely lead to compromises due to simpler API
- **Leveraging Ray Tracing for Global Illumination**
  - Lots of interesting research to do to keep the real-time creation going & fluid
Vulkan & Mobile future & requests

- **Subpass compute + tiled memory!**
  - Tied into dynamic rendering?
  - Possible in Metal today
- **Validation!**
  - Synchronization improvements were amazing. More!!
- **Job system friendly extensions**
  - Polling, callbacks etc. How to avoid lazily polling all the time?
  - API to not assume need for blocking waits on threads?
  - Driver definitely has more tools, otherwise blocking wait would not work
    - Guessing there's a condition variable behind current vkWaitForFences operation
- **More extensions for letting us know when we hit a slow path**
  - We want to tell the driver / validation to slap us in the face when we hit a slow path
  - Very critical to tiled memory, subpasses etc
  - Also relevant to shader compilation, push constants backed by uniform & storage buffers etc.