Vulkanised 2024

The 6th Vulkan Developer Conference
Sunnyvale, California | February 5-7, 2024

Writing an Efficient Vulkan Renderer for Quest Pro and Quest 3

Sergey Kosarevsky, Meta
Alexey Medvedev, Meta

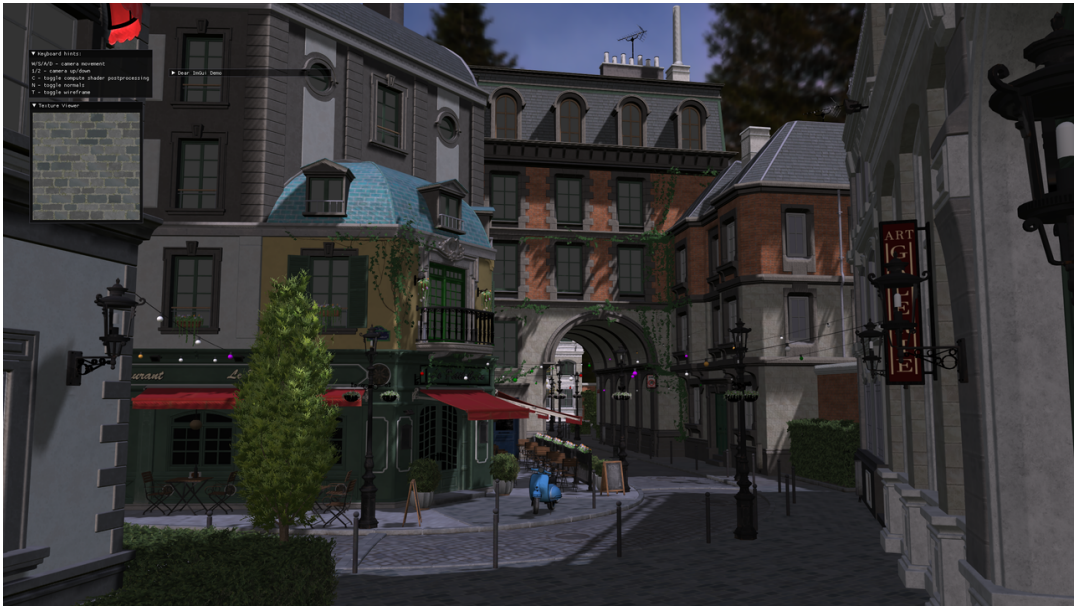
Agenda

- What is IGL?
- IGL Shell?
- Compiling GLSL shaders at run-time
- Command buffers
- Fences
- Staging device
- Render pipeline states
- Layout transitions & dependencies
- Future plans for IGL API

What is IGL?

- Cross-platform, graphics API agnostic rendering library
- Used in 20+ Meta's projects (internal and external)
- **Windows**, **macOS**, **iOS**, **Android**, **Linux**, **WebAssembly**, **OpenXR**
- **GLES 2**, **GLES 3+**, **WebGL 2**, **GL 4.1+**, **Metal 2+**
- **Vulkan 1.1** + optional bindless extensions VK_KHR_buffer_device_address, VK_EXT_descriptor_indexing
- **Open Source MIT project** https://github.com/facebook/igl

What is IGL?


▼ Keyboard hints:
W/S/A/D - camera movement
1/2 - camera up/down
C - toggle compute shader postprocessing
N - toggle normals
T - toggle wireframe
▼ Texture Viewer
▶ Dear ImGui Demo
ART

IGL Shell?

- Cross-platform, application agnostic framework
- Used as internal test farmework and external sample app
- RenderSession — "application"
- No changes required to run on any platform
- **Windows**, **macOS**, **iOS**, **Android**, **Linux**, **WebAssembly**, **OpenXR**

What is IGL?

Modelled after the Metal API and has a very strong OpenGL ES 2 influence:

```
class IRenderCommandEncoder
{
public:
  ...
  virtual void bindRenderPipelineState(IRenderPipelineState& pipelineState) = 0;
  virtual void bindBuffer(int index, IBuffer& buffer, size_t bufferOffset) = 0;
  virtual void bindSamplerState(int index, ISamplerState& samplerState) = 0;
  virtual void bindTexture(int index, ITexture& texture) = 0;
  virtual void drawIndexed(PrimitiveType primitiveType, uint32_t indexCount,
    IndexFormat indexFormat, IBuffer& indexBuffer, size_t indexBufferOffset) = 0;
  ...
};
```

Compiling GLSL shaders at run-time

- IGL allows GLSL shaders
- Vulkan accepts only binary SPIR-V
- Use ‘glslang‘ to compile GLSL at run-time

Compiling GLSL shaders at run-time

```
Result compileShader(VkDevice device,
  VkShaderStageFlagBits stage,
  const char* code,
  std::vector<uint32_t>& outSPIRV,
  const glslang_resource_t* glslLangResource)
{
  const glslang_input_t input = {
    .language = GLSLANG_SOURCE_GLSL,
    .stage = getGLSLangShaderStage(stage),
    .client = GLSLANG_CLIENT_VULKAN,
    .client_version = GLSLANG_TARGET_VULKAN_1_1,
    .target_language = GLSLANG_TARGET_SPV,
    .target_language_version = GLSLANG_TARGET_SPV_1_3,
    .code = code,
    .default_version = 100,
    .default_profile = GLSLANG_NO_PROFILE,
    .messages = GLSLANG_MSG_DEFAULT_BIT,
    .resource = glslLangResource,
  };

  glslang_shader_t* shader = glslang_shader_create(&input);
  IGL_SCOPE_EXIT { glslang_shader_delete(shader); };

  if (!glslang_shader_preprocess(shader, &input)) {
    IGL_LOG_ERROR("Preprocessing failed:\n%s\n%s\n",
      glslang_shader_get_info_log(shader),
      glslang_shader_get_info_debug_log(shader));
    logShaderSource(code);
    return Result(InvalidOperation);
  }
  if (!glslang_shader_parse(shader, &input)) {
    IGL_LOG_ERROR("Parsing failed:\n%s\n%s\n",
      glslang_shader_get_info_log(shader),
      glslang_shader_get_info_debug_log(shader));
    logShaderSource(
      glslang_shader_get_preprocessed_code(shader));
    return Result(InvalidOperation);
  }
  glslang_program_t* p = glslang_program_create();
  glslang_program_add_shader(p, shader);
  IGL_SCOPE_EXIT { glslang_program_delete(p); };
  if (!glslang_program_link(p, GLSLANG_MSG_SPV_RULES_BIT |
        GLSLANG_MSG_VULKAN_RULES_BIT)) {
    IGL_LOG_ERROR("Linking failed:\n%s\n%s\n",
      glslang_program_get_info_log(p),
      glslang_program_get_info_debug_log(p));
    return Result(InvalidOperation);
  }
  glslang_spv_options_t options = { .optimize_size = true };
  glslang_program_SPIRV_generate_with_options(
    p, input.stage, &options);
  if (glslang_program_SPIRV_get_messages(p)) {
    IGL_LOG_ERROR("%s\n",
      glslang_program_SPIRV_get_messages(p));
  }
  const uint32_t* ptr = glslang_program_SPIRV_get_ptr(p);
  outSPIRV = std::vector(ptr,
    ptr + glslang_program_SPIRV_get_size(p));
  return Result();
}
```

Command buffers

- Command pools and buffers are encapsulated in **VulkanImmediateCommands**
- Expose a high-level interface for command buffers

Command buffers

```
class VulkanImmediateCommands final {
 public:
  struct SubmitHandle {
    uint32_t bufferIndex_ = 0;
    uint32_t submitId_ = 0;
    bool empty() const { return submitId_ == 0; }
    uint64_t handle() const { return (uint64_t(submitId_) << 32) + bufferIndex_;}
  };
  struct CommandBufferWrapper {
    VkCommandBuffer cmdBuf_ = VK_NULL_HANDLE;
    VkFence fence_;
    VkSemaphore semaphore_;
    SubmitHandle handle_ = {};
  };
  const CommandBufferWrapper& acquire();
  SubmitHandle submit(const CommandBufferWrapper& wrapper);

  void waitSemaphore(VkSemaphore semaphore); // inject a swapchain semaphore etc

  bool isReady(SubmitHandle handle) const;
  void wait(SubmitHandle handle, uint64_t timeoutNanoseconds = UINT64_MAX);
  void waitAll();
  ...
 private:
  VkCommandPool commandPool_;
  std::vector<CommandBufferWrapper> buffers_;
  ...
};
```

Command buffers

```
VulkanImmediateCommands::SubmitHandle VulkanImmediateCommands::submit(const CommandBufferWrapper& wrapper) {
  VK_ASSERT(vkEndCommandBuffer(wrapper.cmdBuf_));

  VkSemaphore waitSemaphores[] = {VK_NULL_HANDLE, VK_NULL_HANDLE};
  uint32_t numWaitSemaphores = 0;
  if (waitSemaphore_) waitSemaphores[numWaitSemaphores++] = waitSemaphore_;
  if (lastSubmitSemaphore_) waitSemaphores[numWaitSemaphores++] = lastSubmitSemaphore_;

  const VkPipelineStageFlags waitStageMasks[] =
    { VK_PIPELINE_STAGE_ALL_COMMANDS_BIT, VK_PIPELINE_STAGE_ALL_COMMANDS_BIT };
  const VkSubmitInfo si = {
      .sType = VK_STRUCTURE_TYPE_SUBMIT_INFO,
      .waitSemaphoreCount = numWaitSemaphores,
      .pWaitSemaphores = numWaitSemaphores ? waitSemaphores : nullptr,
      .pWaitDstStageMask = waitStageMasks,
      .commandBufferCount = 1,
      .pCommandBuffers = &wrapper.cmdBuf_,
      .signalSemaphoreCount = 1,
      .pSignalSemaphores = &wrapper.semaphore_.vkSemaphore_,
  };
  VK_ASSERT(vkQueueSubmit(queue_, 1u, &si, wrapper.fence_.vkFence_));

  lastSubmitSemaphore_ = wrapper.semaphore_.vkSemaphore_;
  lastSubmitHandle_ = wrapper.handle_;
  waitSemaphore_ = VK_NULL_HANDLE;
  submitCounter_++;

  return lastSubmitHandle_;
}
```

Fences

- Vulkan synchronization is too low-level
- Expose a high-level interface for fences

```
1
2 /// GPU Fence Handle – exposed on the IGL API level
3 using SubmitHandle = uint64_t;
4
5 class ICommandQueue {
6 public:
7   virtual SubmitHandle submit(const ICommandBuffer& commandBuffer) = 0;
8 };
```

Fences

```
1
2  bool VulkanImmediateCommands::isReady(const SubmitHandle handle) const {
3    IGL_ASSERT(handle.bufferIndex_ < kMaxCommandBuffers);
4
5    if (handle.empty()) {
6      // a null handle
7      return true;
8    }
9
10   const CommandBufferWrapper& buf = buffers_[handle.bufferIndex_];
11
12   if (buf.cmdBuf_ == VK_NULL_HANDLE) {
13     // already recycled and not yet reused
14     return true;
15   }
16
17   if (buf.handle_.submitId_ != handle.submitId_) {
18     // already recycled and reused by another command buffer
19     return true;
20   }
21
22   return vkWaitForFences(device_, 1, &buf.fence_.vkFence_, VK_TRUE, 0) == VK_SUCCESS;
23 }
```

Staging device

- Upload texture/buffer data to GPU resources
- Manages lazily allocated staging buffers
- Can grow up to 256Mb size based on workload

Staging device

```
class VulkanStagingDevice final {
  struct MemoryRegion {
    VkDeviceSize offset = 0u;
    VkDeviceSize size = 0u;
    VkDeviceSize alignedSize = 0u;
    VulkanImmediateCommands::SubmitHandle handle;
  };

  MemoryRegion nextFreeBlock(VkDeviceSize size);
  void growStagingBuffer(VkDeviceSize minimumSize);
  ...

public:
  void bufferSubData(VulkanBuffer& buffer, size_t dstOffset, size_t size, const void* data);
  void getBufferSubData(VulkanBuffer& buffer, size_t srcOffset, size_t size, void* data);
  void imageData(VulkanImage& image,
                 TextureType type,
                 const TextureRangeDesc& range,
                 const TextureFormatProperties& properties,
                 uint32_t bytesPerRow,
                 const void* data);
  ...
};
```

Render pipeline states

- IGL has more dynamic state than what unextended Vulkan 1.1 can provide
- Don't enable extra extensions
- Use a hash map to map IGL render states to **VkPipeline** objects
- Treat Vulkan render passes as a part of dynamic state

Render pipeline states

```
1
2  class alignas(sizeof(uint64_t)) RenderPipelineDynamicState {
3    uint32_t topology_                : 4;
4    uint32_t depthCompareOp_          : 3;
5    uint32_t stencilFrontFailOp_      : 3;
6    uint32_t stencilFrontPassOp_      : 3;
7    uint32_t stencilFrontDepthFailOp_ : 3;
8    uint32_t stencilFrontCompareOp_   : 3;
9    uint32_t stencilBackFailOp_       : 3;
10   uint32_t stencilBackPassOp_       : 3;
11   uint32_t stencilBackDepthFailOp_  : 3;
12   uint32_t stencilBackCompareOp_    : 3;
13   uint32_t renderPassIndex_         : 8;
14   uint32_t depthBiasEnable_         : 1;
15   uint32_t depthWriteEnable_        : 1;
16
17   bool operator==(const RenderPipelineDynamicState& other) const {
18     return *(uint64_t*)this == *(uint64_t*)&other;
19   }
20   struct HashFunction {
21     uint64_t operator()(const RenderPipelineDynamicState& s) const { return *(const uint64_t*)&s; }
22   };
23 };
24
25 class RenderPipelineState final : public IRenderPipelineState {
26   ...
27   unordered_map<RenderPipelineDynamicState, VkPipeline, RenderPipelineDynamicState::HashFunction> pipelines_;
28 };
```

Layout transitions & dependencies

- Vulkan image layout transitions are too verbose — track internally
- Expose an API to declare dependencies between graphics and compute work

Layout transitions & dependencies

```
1
2  struct Dependencies {
3    static constexpr uint32_t IGL_MAX_TEXTURE_DEPENDENCIES = 4;
4    static constexpr uint32_t IGL_MAX_BUFFER_DEPENDENCIES = 4;
5    ITexture* textures[IGL_MAX_TEXTURE_DEPENDENCIES] = {};
6    IBuffer*  buffers[IGL_MAX_BUFFER_DEPENDENCIES] = {};
7  };
8
9  class ICommandBuffer {
10  public:
11    virtual std::unique_ptr<IRenderCommandEncoder> createRenderCommandEncoder(
12        const RenderPassDesc& renderPass,
13        IFramebuffer& framebuffer,
14        const Dependencies& dependencies,
15        Result* outResult) = 0;
16    virtual std::unique_ptr<IComputeCommandEncoder> createComputeCommandEncoder(const Dependencies& dependencies) = 0;
17    ...
18 };
```

Layout transitions & dependencies

```
1
2   for (auto& tex : dependencies.textures) {
3     transitionToShaderReadOnly(tex);
4   }
5   for (auto& buf : dependencies.buffers) {
6     if (!buf) continue;
7     VkPipelineStageFlags dstStageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
8     if ((buf->vkUsageFlags_ & VK_BUFFER_USAGE_INDEX_BUFFER_BIT) ||
9         (buf->vkUsageFlags_ & VK_BUFFER_USAGE_VERTEX_BUFFER_BIT)) {
10      dstStageFlags |= VK_PIPELINE_STAGE_VERTEX_INPUT_BIT;
11    }
12    if (buf->vkUsageFlags_ & VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT) {
13      dstStageFlags |= VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT;
14    }
15    bufferBarrier(buf, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, dstStageFlags);
16  }
```

Future plans for IGL API

- Better dependencies & barriers — a frame graph
- An API for Vulkan subpasses
- Replace smart pointers with integer handles

Questions?

Thank you!

Sergey Kosarevsky
corporateshark
sk@linderdaum.com

Alexey Medvedev
aleksrudybear
rudybear@gmail.com

https://github.com/facebook/igl
https://github.com/corporateshark/lightweightvk ← lots of bindless stuff in a fork