HLSL shaders compilation pipeline in BG3

LARIAN STUDIOS

Content

About me:

- Name is Mikhail Korolev.
- Work as senior engine programmer for Larian.
- Work on Vulkan back-end and general rending refactoring since early 2018.

In this presentation:

- our setting which is important for solutions for given problems;
- bindings problems and solutions;
- shader reflection requirements and implementation;
- few words on offline validation;

Setting

We use usual HLSL as a shader's source language

```
struct VsIn
{
    float2 position : Position0;
    float2 texCoord : TexCoord0;
};

struct VsOut
{
    float4 position : SV_Position0;
    float2 texCoord : TexCoord0;
};

VsOut main(in const VsIn vsIn)
{
    VsOut vsOut;
    vsOut.position = float4(vsIn.position, 1.0f, 1.0f);
    vsOut.texCoord = ConvertTexCoordsVPtoRT(vsIn.texCoord);
    return vsOut;
}
```

Setting

We use single `.shd` file for all needed stages in the shader.

```
[Vertex shader]
#include "Shaders/HLSL/Preamble.shdh"
#include "Shaders/HLSL/PPDefaultVS.shdh"

[Fragment shader]
#include "Shaders/HLSL/Preamble.shdh"
#include "Shaders/HLSL/CombineUI.shdh"
```

Same with the binary format: we use single `.bshd` file for all needed stages and meta data:

- vertex format
- used descriptor sets mask
- list of `VkDescriptorSetLayoutBinding` to create shader's descriptor set layout
- list of combinations of descriptor set index and binding for validation purposes

Setting

Tools used for Vulkan shader generation:

- dxc to compile HLSL into SPIR-V.
- SPIRV-Cross to extract reflection from SPIR-V module.
- SPIRV-Tools to strip reflection when we are done with it and additional optimize pass.

Bindings

Problem: in DX bindings are different for different shader stages so having different constant buffers in register `b0` for VS and PS stages is perfectly fine.

```
[Vertex shader]
cbuffer VsCb : register (b0)
{
      float4x4 WorldMatrix;
}

[Fragment shader]
cbuffer PsCb : register (b0)
{
      float4 color;
}
```

Code above is valid for both HLSL and for DX but doesn’t work for Vulkan.

Bindings

Solution? Wrap everything in `[[vk::binding(X, DS_Shader)]]`!

```
[Vertex shader]
VK_DESCRIPTOR(DS_Shader, 0) cbuffer VsCb : register (b0)
{
      float4x4 WorldMatrix;
}

[Fragment shader]
VK_DESCRIPTOR(DS_Shader, 1) cbuffer PsCb : register (b0)
{
      float4 color;
}
```

Bindings

New problems:

- This is boring and annoying.
- Same collisions as before (but now handwritten!).
- Holes in descriptor maps.

```
VK_DESCRIPTOR(DS_Shader, 0) Texture2D foo;
VK_DESCRIPTOR(DS_Shader, 2) Texture2D bar;
// where is VK_DESCRIPTOR(DS_Shader, 1)...
```

Bindings

Better solution - remap bindings during compilation automatically.

1. Compile all shader stages and remap all unspecified bindings to descriptor set 100

```
-auto-binding-space 100
```

Bindings

2. Using SPIRV-Cross collect all bindings in ds 100 across all stages, save their types/names/IDs

```
const auto processResources = [&](const auto& resources)
{
   for (const spirv_cross::Resource& i : resources) {
        const uint32 descriptorSetNumber
            = stage.Compiler->get_decoration(i.id, spv::DecorationDescriptorSet);
        if (descriptorSetNumber == 100) {
            InternalLocalBinding binding;
            binding.Name = stage.Compiler->get_name(i.id);
            binding.Type = SPIRVTypeToString(stage.Compiler->get_type(i.type_id));
            binding.Binding = stage.Compiler->get_decoration(i.id, spv::DecorationBinding);
            binding.ID = i.id;
            stage.UnspecifiedBindings.Append(std::move(binding));
        }
    }
};

const auto shaderResources = stage.Compiler->get_shader_resources();
processResources(shaderResources.uniform_buffers);
// ...
```

Bindings

3. Specify new bind points in descriptor set DS_Shader for unspecified bindings

```
for (int i = 0; i < rf::SHADER_STAGES_NUMBER; ++i) {
    auto& iStage = m_InternalPerStageData[i];
    for (auto& iBinding : iStage.UnspecifiedBindings) {
        // check if it was specified anywhere before
        bool found = false;
        for (int j = 0; j < rf::SHADER_STAGES_NUMBER; ++j) {
            auto& jStage = m_InternalPerStageData[j];
            for (const auto& jBinding : jStage.SpecifiedBindings) {
                if (iBinding == jBinding) {
                    iBinding.Binding = jBinding.Binding;
                    found = true;
                    break;
                }
            }
            if (found) break;
        }
        // if it wasn't -- assign next available binding
        if (!found)
            iBinding.Binding = ++maxBindIndex;
    }
}
```

Bindings

4. Walk through SPIR-V and remap it in binary

```
const auto id = spirvBinaryPtr[j + 1];
const auto decorationType = spirvBinaryPtr[j + 2];
auto& decorationValue = spirvBinaryPtr[j + 3];
if (decorationType == spv::DecorationDescriptorSet && decorationValue == 100) {
    decorationValue = ls::checked_numcast<uint32>(rf::sb::DescriptorSet::Shader);
} else if (decorationType == spv::DecorationBinding) {
    for (const auto& unspecifiedBinding : internalStageData.UnspecifiedBindings)
    {
        if (unspecifiedBinding.ID == id)
        {
            decorationValue = unspecifiedBinding.Binding;
            break;
        }
    }
}
```

Bindings

- Now our HLSL code is usual HLSL code again.
- Bindings are assigned automatically without any human involvement.
- Ho holes, no bindings collisions across shader stages.

Reflection

Setting:

- we have engine shaders which are more or less stable and where bindings are bound by code
- we have data-driven materials where bindings are bound by simply iterating through material's resources

In DOS2 everything was bound by string name of the resource:

```
commandBuffer.SetTexture("MyTexture", myTextureHandle);
```

Reflection

Better option – have a data structure that represents resource's bind point. Should be `constexpr` variable for engine shaders and embedded directly into material file.

```
struct Binding
{
    constexpr Binding() noexcept { /* ... */ }

    constexpr Binding(/* ... */,
                      const rf::sb::DescriptorSet descriptorSetIndex,
                      const int8 descriptorSetBinding) noexcept
        : /* ... */
        , DescriptorSetIndex(descriptorSetIndex)
        , DescriptorSetBinding(descriptorSetBinding)
    {
    }

    /* ... */
    rf::sb::DescriptorSet DescriptorSetIndex;
    int8 DescriptorSetBinding;
};

static constexpr rf::sb::Binding Base = {/* ... */, rf::sb::DescriptorSet(1), 1};
```

Reflection

Another wanted features are:

Compile time constant buffers layout checking (we use DX layout)

```
static_assert(offsetof(rf::sb::Tex::TexParams, scaleBias) == 0);
static_assert(offsetof(rf::sb::Tex::TexParams, color) == 16);
static_assert(offsetof(rf::sb::Tex::TexParams, offset) == 32);
static_assert(offsetof(rf::sb::Tex::TexParams, PADDING) == 40);
static_assert(sizeof(rf::sb::Tex::TexParams) == 48);
static_assert((sizeof(rf::sb::Tex::TexParams) % 16) == 0);
```

Constant buffers size for data-driven materials

```
EngineMaterialParamFlags EngineParamFlags;
uint32 EngineCBSize;
uint32 MaterialCBSize;
MaterialBindPoint EngineCBBinding;
MaterialBindPoint MaterialCBBinding;
```

Reflection

Implementation – SPIRV-Cross (again)!

```
const auto processResources = [&](const auto& resources, const ResKind type) {
    for (const auto& i : resources)
    {
        const uint32 dsn = internalStageData.Compiler->get_decoration(i.id, spv::DecorationDescriptorSet);
        stageData.UsedSets.Set(ls::checked_numcast<uint8>(dsn), true);
        switch (type)
        {
        case ResKind::UBO: FillUniformBuffer(stageIdx, i); break;
        case ResKind::SBO: FillStorageBuffer(stageIdx, i); break;
        case ResKind::Sampler: FillSampler(stageIdx, i); break;
        case ResKind::SampledImage: FillTexture(stageIdx, i); break;
        case ResKind::InputAttachement: FillTexture(stageIdx, i, false, true); break;
        case ResKind::StorageImage: FillTexture(stageIdx, i, true); break;
        default: LS_ABORT_ON_REACHED();
        }
    }
};

const auto shaderResources = internalStageData.Compiler->get_shader_resources();
processResources(shaderResources.uniform_buffers, ResKind::UBO);
processResources(shaderResources.storage_buffers, ResKind::SBO);
/* ... */
```

Reflection

Save it to simple .xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <UniformBuffer name="TexParams" used_in="LS_VULKAN" global_in="" size="48">
        <Binding descriptor_set_index="1" descriptor_set_binding="0"/>
        <Members>
            <Member name="scaleBias" type="Float4" array_len="1" offset="0"/>
            <Member name="color" type="Float4" array_len="1" offset="16"/>
            <Member name="offset" type="Float2" array_len="1" offset="32"/>
            <Member name="PADDING" type="Float2" array_len="1" offset="40"/>
        </Members>
    </UniformBuffer>
    <Sampler name="BaseSampler" used_in="LS_VULKAN" global_in="">
        <Binding descriptor_set_index="1" descriptor_set_binding="2"/>
    </Sampler>
    <Texture name="Base" used_in="LS_VULKAN" global_in="" type="Texture2D">
        <Binding descriptor_set_index="1" descriptor_set_binding="1"/>
    </Texture>
</data>
```

Reflection

Using simple 500 LoC python script generate .h (to create constexpr bindings) and .cpp (to validate sizeof and layout) files

```
struct Tex
{
    static constexpr rf::sb::Binding Base = {/* ... */, rf::sb::DescriptorSet(1), 1};
    static constexpr rf::sb::Binding BaseSampler = {/* ... */, rf::sb::DescriptorSet(1), 2};
    struct TexParams : rf::sb::BindIndex</* ... */, rf::sb::DescriptorSet(1), 0>
    {
        float4 scaleBias;
        float4 color;
        float2 offset;
        float2 PADDING;

        TexParams() noexcept
            : scaleBias{float4(0.0f)}
            , color{float4(0.0f)}
            , offset{float2(0.0f)}
            , PADDING{float2(0.0f)}
        {}
        /* copy/move c-tors, copy/move operators, operators ==/!= */
};
```

Using C++ reflection deserializer embed same thing in material files

Reflection

Now resources for the engine shaders are bound like that:

```
auto cb = commandBuffer.MapTempConstantBuffer<sb::Tex::TexParams>();
cb->scaleBias = m_ScaleAndBias2D;
cb->offset = stringOffset;
cb->color = color;
commandBuffer.UnmapAndBindTempConstantBuffer(cb);

commandBuffer.SetTexture(sb::Tex::Base, texture);
commandBuffer.SetSamplerState(sb::Tex::BaseSampler, samplerState);
```

If it compiles it [probably] works.

And materials binding is simple iteration over resources – no lookups involved:

```
for (const auto& i : data.Textures)
{
    commandBuffer.SetTexture(i.Binding, i.Texture);
}
```

Few words on validation

Vulkan's validation layers are great but it's too late/heavy.
Example cases:

- bindings collisions between the stages
- IO interface between the stages
- vertex format mismatch
- If all required resources was bound before the draw/dispatch call

```
[Vertex shader]
VK_DESCRIPTOR(DS_Shader, 0) cbuffer CB : register (b0)
{
    float4x4 WorldMatrix;
    float4 Color;
}
VK_DESCRIPTOR(DS_Shader, 1) Texture2D Tex;

[Fragment shader]
VK_DESCRIPTOR(DS_Shader, 0) cbuffer CB : register (b0)
{
    float4x4 WorldMatrix;
    float4 Color;
}
VK_DESCRIPTOR(DS_Shader, 1) SamplerState SS; // compile time error
```

```
[Vertex shader]
struct VsOut
{
    float4 position : SV_Position0;
    float2 texCoord : TexCoord0;
};

[Fragment shader]
struct PsIn
{
    float4 position : SV_Position;
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
};

// texCoord0 – not wide enough type in VS
// texCoord1 – not present in VS
```