SimpleShader

Helper classes for loading & managing
Direct3D shaders and their constant buffers

Constant buffers *everywhere*

- Each shader will eventually need one *or more* constant buffers
  - Vertex shader needs matrices
  - Pixel shader needs lights, camera info, color tint, etc.

- This means:
  - Multiple C++ buffer structs
  - Multiple Map/memcpy/Unmap steps per object
  - Updating C++ with each new shader variable

More like constant BUMMER

- I just wanted some dang variables
- Why is this so complex?
- Tradeoff is efficiency
  - Grouping data for a single transfer is faster
  - Requires a lot of manual setup
  - Must keep C++ & HLSL in sync
- Couldn’t we automate some of this? 🤔

Constant buffer management got you down?

You need SimpleShader!

Introducing SimpleShader

- Helper classes to ease:
  - Shader loading
  - C++ & shader interaction (constant buffers)
  - Binding of resources (textures & sampler)
  - Input Layout creation
- Less worrying about cbuffers in C++
- Less C++ code to do the same thing!
- Available at https://github.com/vixorien/SimpleShader

Is SimpleShader right for me?

- Turn these:

```
struct VertexShaderExternalData
{
    DirectX::XMFLOAT4 colorTint;
    DirectX::XMFLOAT4X4 worldMatrix;
    DirectX::XMFLOAT4X4 viewMatrix;
    DirectX::XMFLOAT4X4 projectionMatrix;
};
```

```
VertexShaderExternalData vsData;
vsData.colorTint = material->GetColorTint();
vsData.worldMatrix = transform.GetWorldMatrix();
vsData.viewMatrix = camera->GetView();
vsData.projectionMatrix = camera->GetProjection();

D3D11_MAPPED_SUBRESOURCE mappedBuffer = {};
context->Map(vsConstantBuffer.Get(), 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedBuffer);
memcpy(mappedBuffer.pData, &vsData, sizeof(vsData));
context->Unmap(vsConstantBuffer.Get(), 0);
```

- Into this:

```
// Set variables using their actual names!
// Data is collected locally (in C++) until copy step below
simpleVertShader->SetFloat4("colorTint", material->GetColorTint());
simpleVertShader->SetMatrix4x4("world", transform.GetWorldMatrix());
simpleVertShader->SetMatrix4x4("view", camera->GetView());
simpleVertShader->SetMatrix4x4("proj", camera->GetProjection());

// Actually copy the entire buffer of data to the GPU
simpleVertShader->CopyAllBufferData();
```

Look ma’, no Buffer Struct!

What does SimpleShader do?

- Allows setting shader variable data by name – FROM C++!
- Control exactly when the CPU->GPU copy occurs
- Auto binding of constant buffer resources
- Creates an ID3D11InputLayout for you
- Simplifies texture and sampler binding

General usage

- Replace D3D shaders w/ SimpleShader objects
  - ID3D11PixelShader → SimplePixelShader
  - ID3D11VertexShader → SimpleVertexShader

- Load shaders at startup

```
basicVertexShader = std::make_shared<SimpleVertexShader>(
    device, context, FixPath(L"VertexShader.cso").c_str());
basicPixelShader = std::make_shared<SimplePixelShader>(
    device, context, FixPath(L"PixelShader.cso").c_str());
```

- Bind shader & set data before drawing

```
// Turn on these shaders
vs->SetShader();
ps->SetShader();

// Send data to the vertex shader
vs->SetMatrix4x4("world", transform->GetWorldMatrix());
vs->SetMatrix4x4("view", camera->GetView());
vs->SetMatrix4x4("projection", camera->GetProjection());
vs->CopyAllBufferData();

// Send data to the pixel shader
ps->SetFloat3("colorTint", colorTint);
ps->CopyAllBufferData();
```

How does SimpleShader work?

- It uses *reflection* to get information about a compiled shader and its cbuffers/resources/etc.

- For each constant buffer it finds:
  - Stores that buffer's info in a hash table
  - Creates corresponding "local data buffer" (unsigned char*)

- For each variable in each cbuffer:
  - Stores the size and byte offset of the variable in another hash table

Setting a variable

- When you use SimpleShader to set a variable:

```
simpleVertShader->SetFloat4("colorTint", material->GetColorTint());
```

  - It looks up that variable's info by name
  - Then looks up the corresponding constant buffer's information
  - Lastly, memcpy's your data into that cbuffer's local data buffer

- Nothing is copied to the GPU yet! (This is good)

Methods for Setting Variables

- `SetInt()`
- `SetFloat(), SetFloat2(), SetFloat3(), SetFloat4()`
- `SetMatrix4x4()`

- Each takes two parameters:
  - Name of variable to set
  - The actual data to copy

- `SetData()` also exists
  - To copy arbitrary data (including structs of data)
  - Useful for things like arrays of light data
  - All other Set() methods really just call this

Binding (setting) a shader

- When you're ready to "activate" a shader…
- Call its SetShader() method
  - This will bind the shader in Direct3D
  - This shader becomes the active one
- Setting SimpleVertexShader automatically binds InputLayout
  - No need to make your own InputLayouts anymore

Controlling data copies

- You can control exactly when data is copied

- `CopyBufferData(string bufferName)`
  - Copies a single buffer's worth of data to the GPU
  - Great if you only need to update one cbuffer

- `CopyAllBufferData()`
  - Copes all of this shader's buffers' data to the GPU
  - Loops and copies each buffer, one by one

Types of SimpleShaders

- The following classes are included

- SimpleVertexShader
- SimplePixelShader
- SimpleGeometryShader
- SimpleHullShader
- SimpleDomainShader
- SimpleComputeShader

- We'll be focusing on Vertex and Pixel

Odds & ends – Object creation

- Deleting a SimpleShader will release any resources it created
  - The shader itself
  - All constant buffers it made
  - All C++ arrays (local data buffers)
  - Input layouts for vertex shaders

- Works with smart pointers
  - Use `shared_ptr` to store SimpleShaders
  - No need to manually delete then
  - Internally uses ComPtrs for D3D objects as necessary

Odds & ends – Extra features

- SimpleVertexShader handles input layouts
  - Builds a description based on shader reflection
  - Creates ID3D11InputLayout for you
  - Automatically set when you set the shader
- SimpleShader also simplifies setting:
  - Textures
  - SamplerStates
- Many other functions exist for retrieving shader info

Odds & Ends – Error & warning reporting

- Be default, SimpleShader fails silently
  - Errors are ignored as much as possible
  - Example: if setting a variable that doesn't exist, nothing happens

- Can turn on error and/or warning reporting
  - Will appear in console window and Visual Studio's output window

  - Set one or both of these before loading any shaders:
  - `ISimpleShader::ReportErrors = true;`
  - `ISimpleShader::ReportWarnings = true;`