The Forge - Metal2 Implementation

Revison 1.0

*Overview*

Metal2 it's an interesting and powerful API (with its own strengths and weaknesses), but it is certainly different in many aspects to Vulkan and Direct3D12, falling somewhere in the middle between them and their previous counterparts, OpenGL and Direct3D11. This can make approaching TheForge's Metal2 implementation problems challenging, and that's why we expect this document helps our engineers in the process.

This document aims to explain the main between Metal2 and Vulkan/Direct3D12, the limitations present when developing for iOS as well as some important details that we need to take into account when writing metal shaders for TheForge. It also introduces a few XCode development important topics and presents current aspects of the implementation that can be improved.

*Metal2 API Differences*

* **No GPU Queries:** this means no timestamps, occlusion or pipeline statistics queries. The most noticeable problem that comes out of this is no real-time GPU profiling, which means that we can only get CPU timings for our samples. Despite this, GPU times can still be retrieved by using the XCode Graphics Debugger.
* **Different resource binding model:** in Metal2 there isn't a direct equivalent to D3D12's rootSignatures or VK's pipelineLayout, but instead resources are bound to specific binding locations in a similar fashion to D3D11 and OpenGL. This means that the descriptor's update frequency is ignored on Metal2, and that binding multiple resource the same descriptor (mCount > 1) is not directly supported by the API (we use argument buffers, metal buffers containing pointers to resources, to support this last feature).
* **No Geometry/Hull/Domain stages:** in Metal2, only vertex, fragment and compute shader stages are supported. There are ways to support the missing stages though (see sample *07\_Tessellation* for reference), like:
  + ***Geometry:*** compute shaders.
  + ***Hull****:* compute shader that stores both the tessellation factors and control points on writeable buffers.
  + ***Domain:*** vertex shader that take the control points buffer as the [[stage\_in]] attribute.
* **No multi-draw indirect:** this is emulated in our implementation by recording multiple indirect draw commands one by one. This should be considered on code that rely on this feature heavily (such as 04\_ExecuteIndirect or VisibilityBuffer samples).
* **Direct cmds recording:** MTLCommandQueues are not typed-limited, but instead different types of command lists can be submitted to the same queue. This by itself is not a problem, but combined to the fact that there is no direct command lists on Metal2 (they can only execute either render, compute or blit operations), means that if we ever record multiple types of commands into a direct command list, internally we will have more than a single MTLCommandEncoder (the Metal2 equivalent to ID3D12CommandList and VkCommandBuffer) executing asynchronously. To ensure the same behaviour that we have in DX12/VK where commands in a direct command list follow the order of recording, we use the **util\_sync\_encoders** (which signal a fence at the of the current encoder and makes the new encoder wait on it) function every time we change the MTLCommandEncoder we're recording commands to.
* **Fencing:** MTLFences status or value cannot be checked from the CPU side, in contrast to DX12/VK. Because of this, we cannot use MTLFence to wait for queue completion (they can still be used, i.e. to sync different encoders like in the previous issue), but we can instead use a dispatch\_semaphore signaled from the CPU using the MTLCommandBuffer **addCompletedHandler** function.
* **Number of threads in a group:** in Metal2, the number of threads per group in a compute shader cannot be obtained through reflection, but has to be specified from the application code. To solve this, when performing reflection on a compute shader, we look for a comment above the ( //[numthreads(x, y, z)] ) stage function that specifies it.
* **Rendering point primitives:** when rendering point primitives, make sure to specify the [[point\_size]] attribute from the vertex shader to avoid undefined behaviour (which could hurt performance).
* **ScissorRect dimensions:** in contrast to DX12 and VK, the bounds of the scissorRects set by the application have to fall inside the range [0, width] and [0, height] and have a minimum width and height of 1 (which can create problems if not taken into account, i.e. NuklearUI driver).
* **Drawable's dimensions:** on Retina displays, the drawable's (or surface's) size is doubled than the window size specified for the application. To have consistent RT size between the different implementation of TheForge, we calculate the retina scaling factor on initialization and divide the window's dimensions by it, contraresting the scaling and ending up having the desired resolution. The retina scaling factor has to be taken into account when referencing window dimensions or location, such as when resizing the window or calculating the mouse location.
* **No resource barriers:** resource state changes are automatically managed by the API.
* **No buffer memory mapping:** host visible buffers can be accessed by using the MTLBuffer's **contents** function.
* **No 3-Component color formats** (i.e. RGB8, needs to be RGBA8).
* **No render/write to mip support on macOS** (it does work apparently on iOS).
* **Shaders must have the .metal extension to compile.**

*iOS-specific Limitations*

* **Resource bundling:** to be able to load resources when deploying an iOS app, resources (fonts, textures, shaders, etc.) have to be bundled with the application. To do this, you can go to your *project* configuration, select the *iOS scheme*, and drag your resources to the *Copy Bundle Resources* panel under the *Build Phases* tab.
* **Separated depth/stencil RTs:** when creating a render target with the D24S8 pixel format on iOS, two different render targets will be created instead (a depth one with a D32 format and a stencil one with a S8 format), as neither this pixel format nor combined depth/stencil attachments are supported on iOS.
* **Only D32 depth format is natively supported on iOS** (other formats will get interpreted as this one).
* **No block-compressed formats are supported on iOS.**
* **Cube Array textures aren't supported on iOS.**
* **Tessellated indirect draw is not supported on iOS** (instead is simulated recording regular draw commands with the arguments read from the indirect buffer).
* **Shader binding limits:** on iOS, there is a limit of 31 buffers, 31 textures and 16 samplers per shader stage. This limitation makes Visibility Buffer rendering path virtually unachievable (unless we are rendering a really simple scene that uses less than 31 textures).

*Metal Shading Language Notes*

* The entry point for all shader stages has to be called **stageMain** (main is reserved on Metal).
* When selecting your buffers binding locations for a shader stage function, take into account that if you use the **[[stage\_in]]** attribute for one of the functions' argument, the **[[buffer(0)]]** binding location will be reserved by that argument.
* When sharing a root signature between multiple shaders, make sure you have **coherent binding locations** for all your resources (buffers, textures and samplers) **across said shaders** to avoid any binding problems.
* The **device** address space attribute identifies stage function's arguments which are both readable and writable (in contrast to the constant address space).

*XCode Notes*

* **Line endings**

To avoid conflicts between the files shared between XCode and Visual Studio, make sure your XCode settings have always **Windows (CRLF)** as the default line ending setting (and disable the **Convert existing files on save** box).

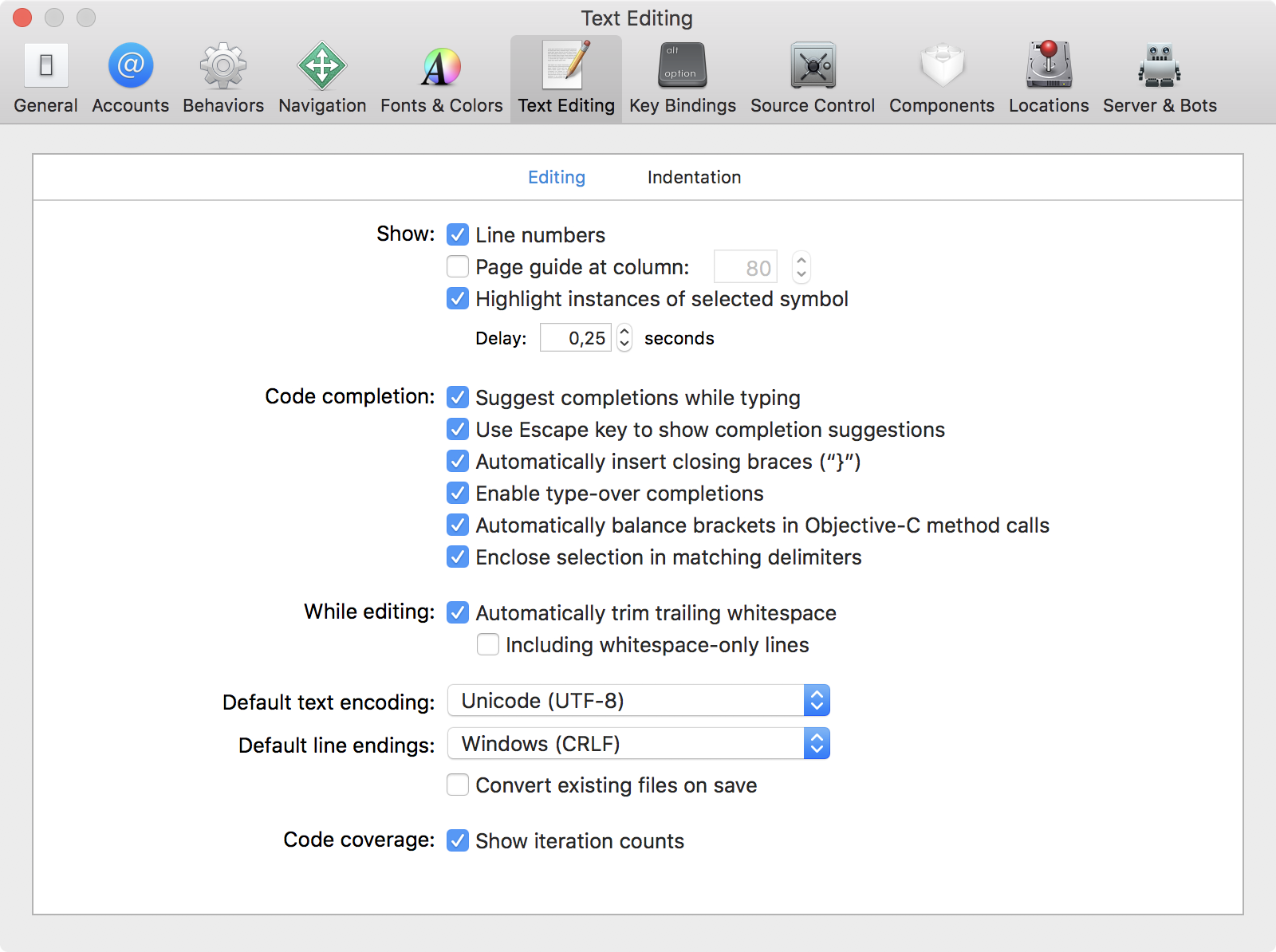


Figure 1: XCode preference panel with correct line endings (CRLF) selected.

* **Metal Shader File Type**

XCode automatically compiles all metal shaders added to a project into a default *MTLLibrary*, which it is then stored into the root of the application bundle and can be used by the app to retrieve the *MTLFunction* used for the different stages without the need to compile the shader code at runtime. Our application don't take any advantage out of this (we're either compiling each shader separately into binaries or compiling at runtime) and this will most likely result in compilation errors (as no two functions are allowed to share the same name, which is a problem as all our shaders share the same entry point name).

In order avoid XCode from compiling any new shaders you add to the project, set the file's type to **Plain Text** in the *Utilities* panel (right). Also make sure that the *Target Membership* is only checked for the iOS scheme, as there is no need to add these resources to the macOS bundle.

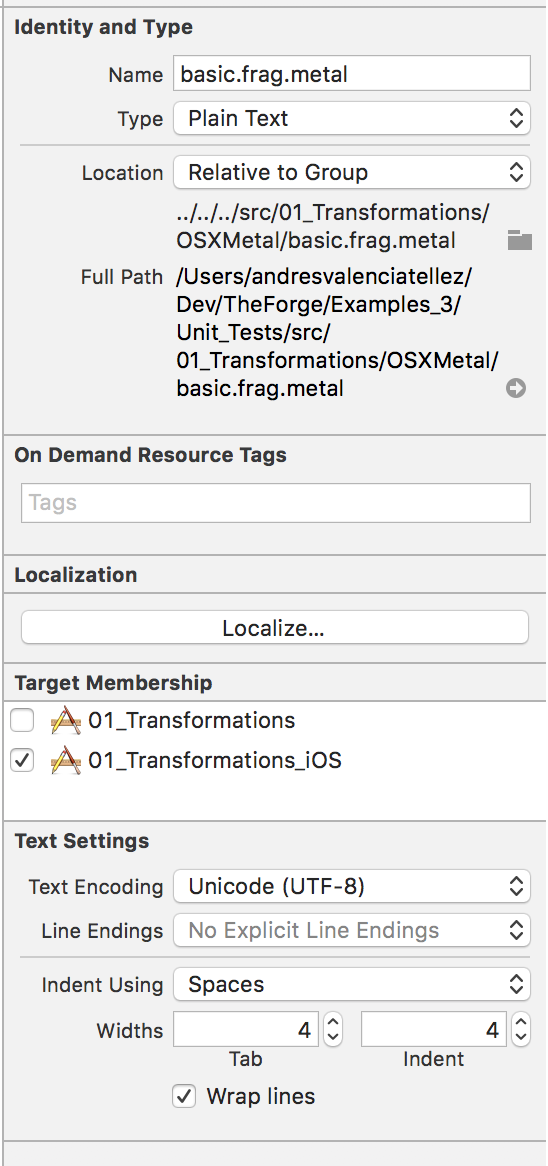


Figure 2: Example of a metal shader's utility panel properties.

* **Scheme Settings**

Projects in XCode can be build and run in various different ways. The scheme settings (accessible through *Product->Scheme->Edit Scheme* or ⌘<) allow you to tweak the settings for this different configurations, allowing you, for example, to choose the *Run* (⌘R) build setting (Debug or Release) or to enable/disable the Metal validation layer.

*Missing features*

Despite having achieved a very robust Metal2 implementation, there are still some details missing or that could be improved to achieve a similar quality level on these platforms.

* **PreBuild script to compile and bundle binary shaders on iOS builds:** we cannot run the xcrun command line tool from our iOS applications, so having a pre build script that compiles them one by one before deploying and copy the results into the app's bundle would allow us to avoid runtime compilation on iOS.
* **NEON support on iOS:** our math library (*ModifiedSonyMath)* doesn't have support for ARM SIMD intrinsics, which could greatly improve performance on the iOS platform.
* **TouchUI for iOS:** NuklearUI can be rendered for iOS, but it's unusable due to relying on pixel perfect input to control the different buttons and sliders, which is just not doable on touch screens. Having a UI system designed for touch interactions could greatly improve the usability of our samples on iOS (as we've seen with the virtual joysticks implementation).
* **Explicit Window handling on macOS:** at the moment, our Metal2 implementation heavily relies on the MetalKit API to create, drive and manage our windows and surfaces. This is ok for iOS, but on macOS (where at some point we would like to deal with more than a single window), we should be explicitly creating and managing our windows as we do on Windows and stop using MetalKit completely.
* **Monitor handling functionality on macOS.**
* **Gamepad support on macOS.**

*Useful documentation*

* [Apple Developer Documentation](https://developer.apple.com/documentation/)
* [Metal Best Practices](https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/MTLBestPracticesGuide/index.html)
* [Understanding Argument Buffers](https://developer.apple.com/documentation/metal/fundamental_components/gpu_resources/understanding_argument_buffers)
* [Metal Shading Language Specification](https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf)
* [Metal Feature Set Tables](https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf)

*Revision Story*

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Author | Date | Changes |
| 1.0 | Andrés Valencia Téllez | 22/02/2018 | First version of the document. |