SAPIENZA
UNIVERSITÀ DI ROMA

ComputeSharp — A .NET library to run C# code in parallel on the GPU through DX12, D2D1, and dynamically generated HLSL compute and pixel shaders

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica (I3S)
Engineering in Computer Science

Sergio Pedri
ID number 1618800

Advisor
Prof. Massimo Mecella

Academic Year 2022/2023

Thesis not yet defended

ComputeSharp — A .NET library to run C# code in parallel on the GPU through DX12, D2D1, and dynamically generated HLSL compute and pixel shaders

Tesi di Laurea Magistrale. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: pedri.1618800@studenti.uniroma1.it

To my mom and all my friends
It did take a while, but I eventually made it

Abstract

Native applications on Windows can take advantage of built-in platform features to create highly efficient and visually engaging experiences, which might also require unique graphics effects to produce the intended aesthetics. While applications can leverage APIs such as DirectX and Direct2D (D2D) to implement all sorts of custom effects, doing so from scratch is extremely complex and error prone, especially if optimization is a concern. This can force developers to prefer sacrificing looks or performance in order to meet deadlines, or to avoid adding too much new code to maintain to their codebase. Often times, the main problem is that the team working on an application might just not be familiar with the underlying system APIs that would be needed in order to create custom graphics effects, and it might not be worth it to dedicate days or even weeks of engineering time to learning about all those technologies, just to implement a few custom effects. This puts a hard limit on creativity solely due to time or budget constraints, which can make the final product not look as good as it could have, and it can leave engineers frustrated for not being able to match the designs they were trying to follow when building their applications. This document introduces ComputeSharp, a .NET library for C# developers that abstracts over all of these graphics APIs built-in into Windows, and provides users an extremely easy and high level framework to create custom graphics effects and run parallel computation on the GPU, in a way that completely hides all of the implementation details of the underlying system APIs being used behind the scenes. This document will cover the main technologies used by the library and by modern Windows applications (COM, WinRT, UWP XAML), and it will detail how ComputeSharp.D2D1 in particular allows developers to create D2D pixel shaders entirely in C#. In doing so, it will also dive into the new changes being made into the Win2D library (a WinRT wrapper for D2D) to support external effects (such as those implemented by ComputeSharp), and finally it will provide two real world examples of how all of these building blocks have been leveraged in the Microsoft Store to create beautiful new UI modules used extensively in the application.

Contents

Chapter 1

Architectural foundations

Before going over the specifics of ComputeSharp and how it is leveraged in the Microsoft Store, let's introduce some fundamental concepts that the library is built on top of. ComputeSharp is powered by several system APIs on Windows, most importantly DirectX APIs [1], to execute its compute and pixel shaders on the GPU. Many of these APIs are based on the COM architecture and ABI [2], and on WinRT [3], which is an extension of COM for modern Windows apps. Lastly, ComputeSharp has built-in support for Win2D [4], a WinRT wrapper for Direct2D, and that's how it can be easily leveraged within UWP XAML [5] apps such as the Microsoft Store. We'll go into more details on all of these components, before covering how exactly does ComputeSharp utilize them.

1.1 COM

The Component Object Model (COM) is a binary-interface standard for software components, that was introduced by Microsoft in 1993 [6]. It is used to enable inter-process communication and to share object instances in a large range of programming languages, as well as to organize components through a language agnostic, object oriented architecture within the same process. COM is the foundation technology for Microsoft's OLE (compound documents), ActiveX (Internet-enabled components, now deprecated), as well as others [7].

COM is not an object oriented language, but a standard. It specifies an object model and programming requirements that enable COM objects (also called COM components, or sometimes simply objects) to interact with other objects. These objects can be within a single process, in other processes, and can even be on remote computers. They can be written in different languages, and they may be structurally quite dissimilar, which is why COM is referred to as a binary standard (ABI).

The essence of COM is a language-neutral way of implementing objects that can be used in environments different from the one in which they were created, even across machine boundaries. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it forces implemented components to provide well-defined rules for those components that are separate from the implementation. The different allocation semantics of languages are accommodated by making objects responsible for their own creation and destruction through reference counting. Type conversion casting between different interfaces of an object is achieved through the `QueryInterface` [8] method. The preferred method of "inheritance" within COM is the creation of sub-objects to which method "calls" are delegated [9].

COM defines the essential nature of a COM object. In general, a software object is made up of a set of data and the functions that manipulate the data. A COM object is one in which access to an object's data is achieved exclusively through one or more sets of related functions. These function sets are called interfaces, and the functions of an interface are called methods. Additionally, COM requires that the only way to gain access to the methods of an interface is through a pointer to the interface. This closely mirrors the standard way of defining classes and inheritance in C++. In fact, the COM ABI very closely mirrors that of C++ inheritance, making it particularly convenient to author COM types via C++[1].

1.1.1 COM ABI and `IUnknown`

The COM Application Binary Interface (ABI) defines how programs running under separate processes can communicate with each other using COM. The ABI defines how data structures are laid out in memory and how function calls are made between processes. The ABI also defines how interfaces are identified and how they are queried for by clients. This ABI is leveraged in several components in ComputeSharp, due to its language agnostic characteristics (and also because ComputeSharp itself is written in C#, whereas all the native Windows components are not, and this approach allows them to seamlessly communicate with each other). The fundamental interface for all COM objects is `IUnknown` [12], which provides the base for the COM ABI. It is defined as follows:

[1] Crucially, C# also has built-in support for COM (both in terms of runtime powered COM interop, as well as via source generators and new APIs, on modern .NET[10]). For scenarios where performance is critical (such as in ComputeSharp), it is now common to leverage function pointers[11] to declare blittable `struct` types that directly map over native COM objects. This requires more care on the consuming side to correctly manage the lifetime of these native objects, but in turn gives an almost exact 1:1 match with the corresponding C++ APIs, and the best performance possible. This specific style of COM interop is used throughout ComputeSharp.

```
[uuid(00000000-0000-0000-C000-000000000046)]
class IUnknown
{
    HRESULT QueryInterface(
            [in] REFIID riid,
            [out] void** ppvObject);

    ULONG AddRef();

    ULONG Release();
};
```

These methods provide all the necessary infrastructure for basic COM objects. For the purposes of this document, this is all that's needed with respect to ComputeSharp (as additional COM features such as apartments [13] are not used), and it's what's commonly referred to as "nano COM", or simply "COM ABI" or "`IUnknown` ABI". Here is how these various interface methods are used, what their semantics are, and how they make up the foundation of all COM objects:

- `QueryInterface`: as the name suggests, this method performs a query for a given interface on the target object, and it's the COM way to perform safe type casts. Callers can invoke this method by passing the `IID` of the interface to look for, and a pointer to the COM object to receive. If the target object does implement the requested interface, its reference count is incremented and "ppvObject" is set to that object after the interface cast (note that the address might not be the same as the target object, as it might be pointing to a different vtable or to a different object entirely). If the interface is not implemented, the method will return `E_NOINTERFACE` instead to signal an error, and "ppvObject" will be set to `null`. Calling `QueryInterface` is the only way to safely cast COM objects to different interfaces, and to check whether they do implement specific interfaces. The one exception is upcasting an interface pointer to a base interface, which is always valid and does not need to go through `QueryInterface`.

- `AddRef`: increments the reference count for the current object, and returns the incremented count. This method is usually implemented with atomic operations, though that is not required (in fact, several D2D objects do not use atomic operations, as they rely on these objects only ever being used while other internal D2D locks are being held by a given thread). `AddRef` is usually invoked when a given COM object is passed to a method, which then needs to store it for later use (for instance, in some class field). Incrementing the

reference count ensures that the object will remain alive even if the caller will later release the object.

- `Release`: conceptually the opposite of `AddRef`, this method decrements the reference count on the current COM object, and returns the decremented count. Crucially, if the count is 0, this method is also responsible for deleting the COM object before returning. This is how memory is managed in the COM world: callers simply release references when they're no longer needed, and all objects are automatically freed when their reference count reaches 0. This is particularly convenient in C++, as it makes working with COM object feel quite natural, especially when using types such as `ComPtr<T>` [14], which implement automatic support for RAII-style [15] programming. That is, a `ComPtr<T>` value can automatically call `AddRef` whenever it's copied, and automatically call `Release` whenever it goes out of scope, making all the lifetime management required by COM objects transparent to developers and not error prone.

Here's a simple example that showcases how these methods can be used for basic operations (note that this sample is using raw pointers merely to illustrate the various `IUnknown` APIs more clearly, but real world code would not interact with COM objects this way, but as mentioned above it'd rather do so through `ComPtr<T>` or other higher level wrappers and projections):

```
ID2D1Image* image = nullptr;

// Query interface for ID2D1Effect on a new COM object
ID2D1Effect* effect = nullptr;
HRESULT hresult = image->QueryInterface(__uuidof(ID2D1Effect), (void**)&effect);

// Increments the ref count
image->AddRef();

// Decrements the ref count. The second call will actually cause
// the object to be destroyed, as it'd be the last active reference.
effect->Release();
image->Release();
```

1.1.2 COM composition

One last thing that's worth mentioning about COM, as it's directly leveraged by ComputeSharp to implement several of the components outlined below, is how can COM objects support implementing multiple "disjointed" interfaces, ie. interfaces that don't share the same inheritance chain up to `IUnknown`. There are several

possible approaches for COM objects to handle inheritance (eg. COM aggregation [16] is another one), but they're not directly used by ComputeSharp. The technique that is used is referred to as "COM composition", and it involves objects having separate vtables for each disjointed interface they implement. Then, `QueryInterface` calls can just return the address of the correct vtable for the requested interface. All APIs for interfaces following the main one (ie. the one at offset 0) will simply need all methods to offset the input address they receive for the target COM object before touching any instance state, to restore the correct object address. This approach also allows the shared methods (ie. the ones from `IUnknown`) to be the same across all interfaces, with no duplication. Here's what the class diagram looks like for a COM object implementing two separate disjointed interfaces:

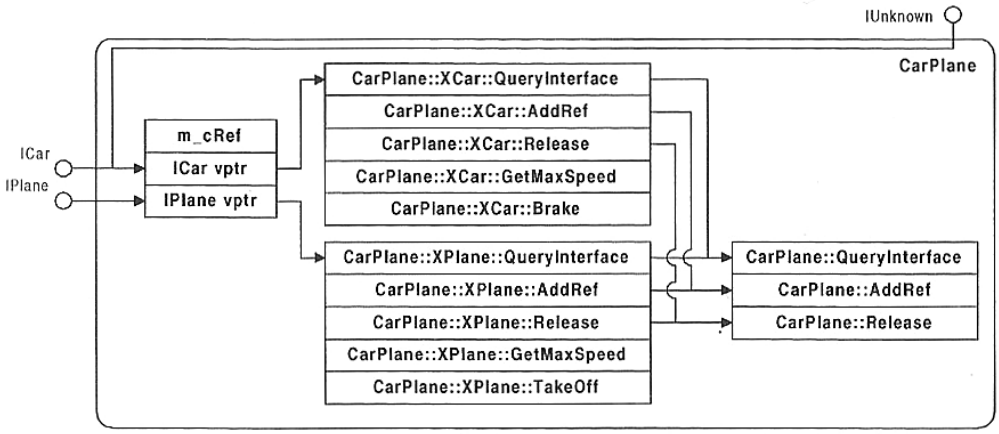


Figure 1.1. COM composition

As we can see, all `IUnknown` methods are shared for the whole type, and the state has separate vtables for each interface, each pointing to the implemented methods for that hierarchy chain. To clarify, here's an example of what the implementation looks like for an interface method from one of these "shifted" interfaces. This implementation is in C#, as it's from one of the ComputeSharp types that will be explained below:

```
[UnmanagedCallersOnly]
public static int QueryInterface(
    PixelShaderEffect* @this,
    Guid* riid,
    void** ppvObject)
{
    @this = (PixelShaderEffect*)&((void**)@this)[-1];

    return @this->QueryInterface(riid, ppvObject);
}
```

```
[UnmanagedCallersOnly]
public static int MapOutputRectToInputRects(
    PixelShaderEffect* @this,
    RECT* outputRect,
    RECT* inputRects,
    uint inputRectsCount)
{
    @this = (PixelShaderEffect*)&((void**)@this)[-1];

    // Implementation here...
}
```

These are two methods from the `ID2D1EffectImpl` implementation that's built-in into ComputeSharp.D2D1. As we will see in more detail below, this type also implements the `ID2D1DrawTransform` interface, which has a different hierarchy chain than `ID2D1EffectImpl`. So, ComputeSharp.D2D1 leverages COM composition to implement both interfaces on the same object, with minimal overhead. Here we can see two different cases: for `QueryInterface`, the implementation is simply forwarded to the shared implementation for the `PixelShaderEffect` type (which will also handle queries for `ID2D1EffectImpl`), and for `MapOutputRectToInputRects` the implementation is directly provided here (omitted for brevity in this snippet). In both cases, we can see how the current COM object address (ie. the "this" parameter) is first shifted back by 1 pointer, as the `ID2D1DrawTransform` interface holds the second vtable in the type layout. As we mentioned, this restores the real address of the COM object, so that accessing instance state will work correctly.

As a side note, since this snippet is from C# code, we can also see how both methods are annotated with `[UnmanagedCallersOnly]` [17], as they're directly exposed as unmanaged entry points in the vtable for the native object being constructed from managed code. The attribute ensures that these methods will use the unmanaged calling convention and will insert the necessary GC transitions for the unmanaged to managed and managed to unmanaged context switches at the start and end of each invocation.

1.2 WinRT

WinRT is a platform-agnostic component and application architecture first introduced in Windows 8 and Windows Server 2012 [18]. It is an unmanaged application binary interface based on COM that allows interfacing from multiple languages, just like COM does as well. WinRT is not a runtime in a traditional sense, but rather a language-independent application binary interface based on COM to allow APIs

to be consumed from multiple languages. WinRT components are designed with interoperability among multiple languages and APIs in mind, including native, managed and scripting languages. Built-in APIs provided by Windows which use the WinRT ABI are commonly known as WinRT APIs, though anyone can use the WinRT ABI for their own APIs as well, once again just like COM.

Windows Runtime (WinRT) APIs are described in machine-readable metadata files with the extension .winmd (also known as Windows Metadata) [19]. These metadata files are used by tools and language projections in order to enable language projection. For instance, for C#, language projections are either generated automatically via built-in WinRT interop (eg. on UWP and .NET Framework), or via an external tool to generate these projections, such as CsWinRT [20]. ComputeSharp leverages both of these approaches, as it supports both UWP as well as modern .NET, which utilizes CsWinRT instead. The .winmd file format is extremely important, as that's how all of these WinRT components are shipped (as a combination of a .dll and a .winmd file). For instance, Win2D also ships this way (and we'll see more about this library below).

1.2.1 WinRT ABI and `IInspectable`

The foundational interface for WinRT is `IInspectable`, defined as follows:

```
[uuid(AF86E2E0-B12D-4C6A-9C5A-D7AA65101E90)]
class IInspectable : IUnknown
{
    HRESULT GetIids(
        [out] ULONG* iidCount,
        [out] IID** iids) = 0;

    HRESULT GetRuntimeClassName([out] HSTRING* className);

    HRESULT GetTrustLevel([out] TrustLevel* trustLevel);
};
```

We can immediately notice how the relationship with COM is very clear: every WinRT object must derive from `IInspectable`, which itself derives from `IUnknown`. That is, every WinRT object is also a COM object. In addition to the `IUnknown` APIs, we can see how WinRT also includes several new methods available on all types, which makes them more difficult to manually implement, but at the same time offering more flexibility to consumers. `GetIids` and `GetRuntimeClassName` implement a form of introspection into WinRT types, allowing developers to query information about them in a more advanced way than just calling `QueryInterface` with a well

known identifier. `GetTrustLevel`, on the other hand, returns the trust level of a given WinRT object, which is a concept that directly ties into the AppContainer isolation system [21] that modern packaged [22] Windows apps can benefit from. For instance, UWP apps (which we'll see below) run under AppContainer in base trust mode, offering the highest level of security possible with this model.

Without going into the technical implementation of each additional feature that WinRT adds compared to just using COM, it's worth noting how the WinRT ABI also includes the ability to have generic types, asynchronous methods, it has rich support for versioning and metadata contracts and annotations, and it supports delegate types (ie. analogous to a function pointer, but with optional instance state attached to them as well). All of these features make it significantly more powerful than just COM, and especially well suited for being projected into languages such as C#, which have many of these concepts available as first class citizens (such as asynchronous methods, which can be seamlessly projected and used as if users were working with normal C# code instead).

1.3 UWP XAML and Composition

The Universal Windows Platform (UWP) is a device-agnostic application platform created by Microsoft and first introduced in Windows 10, and supported by all devices running that version of Windows or above (ie. Windows Desktop, Xbox, Windows 10 Mobile, HoloLens, etc.). The UWP XAML framework is a built-in retained mode UI framework for building native GUI apps on Windows, which is built on top of DirectX, COM and WinRT, and leverages the XAML language [23] to define UI components.

The UWP framework is has several key characteristics:

- It requires apps to be packaged and to run in base trust mode by default, and declaring all needed app capabilities in the application manifest. This ensures that applications can only interact with the resources they strictly need to, and gives users a centralized way to verify all those permissions, and optionally to revoke them at any time. This is in stark contrast with how Windows applications have traditionally run in full trust mode instead, allowing them unrestricted access to all system functionality.

- It runs over a shared set of functionality that all Windows 10 devices implement (regardless of the device family). This is what allows UWP applications to run on different devices (eg. a Windows Desktop and an Xbox) without any changes to the binary. Developers can (and are welcome to) create dynamic

UI experiences that automatically adapt to each device form factor to provide the best UX possible to users.

- Because UWP apps are packaged and distributed via MSIX, they can leverage additional functionality such as tighter integration with the OS, and support for system APIs that require callers to have a package identity in order to be used (eg. the `TaskbarManager` APIs [24]).

Most importantly, for the purposes of this document, UWP XAML applications run on top of DirectX 11 APIs. This is particularly important because as we will see later on, ComputeSharp.D2D1 is backed by D2D pixel shaders, and D2D itself is built on top of DirectX 11, just like UWP XAML. This naturally gives the two the ability to interoperate, which is what the Microsoft Store is also leveraging to integrate custom effects into its UI. Here we can see the layering of the architecture of UWP XAML applications, at a very high level:

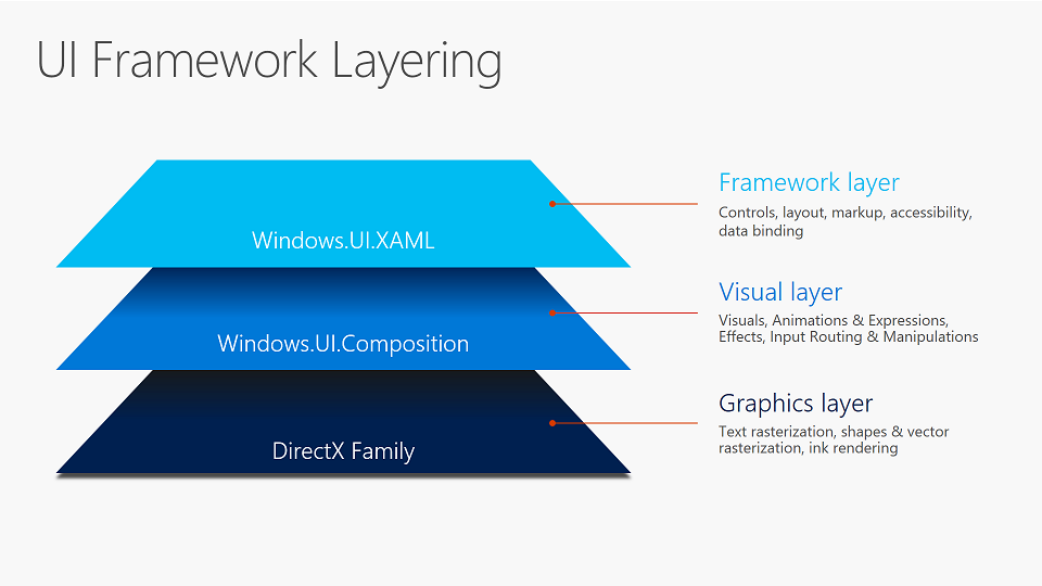


Figure 1.2. UWP XAML visual layers

At the base level is DirectX, which is responsible for sending the actual drawing commands produced by the rest of the framework to the GUI, to perform the final rendering of each frame of the application. This is where D2D also comes into play: it can be used to render to intermediate surfaces which are then composited with the rest of the native content of a UWP XAML application, and then displayed next to other XAML content.

1.4 Win2D

As briefly mentioned above, Win2D is an easy-to-use WinRT library for immediate mode 2D graphics rendering with GPU acceleration. It's powered by Direct2D and DirectX 11, and also provides seamless integration with XAML components. Because it is a WinRT library (that is, it ships as a .dll and accompanying .winmd file), it can be used by both C++ as well as any other language with WinRT projections (such as C#).

To better understand how ComputeSharp integrates with Win2D, it is worth going over what the relationship between Win2D and D2D is. Win2D itself is fundamentally a thin WinRT wrapper around D2D. For many of its components, there is often a 1:1 matching between Win2D APIs and underlying D2D APIs [25]. For instance, Win2D's `CanvasDrawingSession` maps to D2D's `ID2D1DeviceContext1` type, Win2D's `CanvasDevice` maps to D2D's `ID2D1Device1`, etc. This is also the case for all Win2D effects, which are all direct WinRT wrappers around the built-in D2D effects that ship into Windows. This equivalence between Win2D and D2D will especially be important in the context of custom effects, as we'll see later on, as effect authors will have to be extra careful about ensuring that the relationship between wrapping WinRT types and underlying D2D objects is structured as expected, and following the same rules that Win2D and D2D expect in order for all components to work correctly together. This complexity is one of the reasons why all the work to set this part of the infrastructure up is also taken care of by ComputeSharp, with users only have to worry about implementing D2D pixel shaders and using them to draw visual components in their applications.

Chapter 2

ComputeSharp

As mentioned in the abstract, ComputeSharp is a .NET library to run C# code in parallel on the GPU through DX12, D2D1, and dynamically generated HLSL compute and pixel shaders, with the goal of making GPU computing easy to use for all .NET developers. The basic premise is quite simple: leveraging the GPU to accerate workloads that can benefit from its highly parallel architecture can be very advantageous, but it is also something that is extremely difficult to do, especially for developers that are not used to working with ”lowlevel” code and to interacting with system libraries. This is where ComputeSharp comes in: it provides a framework for developers to easily prototype compute and pixel shaders, and run the on the GPU, while the library takes care of all the necessary setup behind the scenes, in a way that's completely transparent to users (it's magic ✨). It also offers built-in APIs to interoperate with Win2D, as well as with UWP and WinUI 3 XAML.

2.1 Infrastructure and DirectX support

Let's first do an overview of ComputeSharp's architecture, to see what individual packages it consists of, and how they're all linked to one another. Worth noting that the whole project could've also been shipped as a single package, but that hasn't been done to make it more modular and to reduce the binary size impact on consumers. Effectively, breaking the package up like this is a great way to make the whole library more ”pay for play”, and to only take in additional dependencies when really needed.

Here's the various packages that ComputeSharp offers:

- **ComputeSharp.Core**: a library containing basic primitives to support ComputeSharp. Most notably, this includes all projections for HLSL functions (we'll go into more detail on this below), as well as for HLSL primitive types.

This package is not meant to be used directl by developers, but rather it's a transitive dependency of the other ComputeSharp packages.

- **ComputeSharp**: a library to run C# code in parallel on the GPU through DX12 and dynamically generated HLSL compute shaders. This is the "main" library (and the first one to be published), and it includes the DirectX 12 compute shader backend.
- **ComputeSharp.D3D12MemoryAllocator**: an extension library for ComputeSharp to enable using D3D12MA [26] as the memory allocator for graphics resources. This is an open-source, high performance memory allocator for DirectX 12, developed by AMD. This package can optionally be referenced and enabled by developers in case their applications are making heavy use of memory allocations, and the extra performance for this allocator was needed. This is particularly crucial on DirectX 12, where its "lowlevel" design makes it so that developers are responsible for allocating resources in an efficient way, and the driver itself is not doing much to help in this area, contrary to how it instead does on DirectX 11 and earlier versions. This is by design, as it allows advanced developers to further optimize their code to fit exactly their needs, but it can result in lower performance when not being particularly careful with memory allocation patterns.
- **ComputeSharp.Dynamic**: an extension library for ComputeSharp to enable dynamic compilation of shaders at runtime. By default, ComputeSharp only supports compiling shaders at build time, which is an intentional limitation that is in place to avoid having to bundle the native DirectX shader compiler in all cases (as it increases the binary size by a significant amount). This package includes that redistributable shader compiler, and exposes APIs that allow the rest of the library backend to hook into it when needed. This functionality being in a separate package is another example of how this subdivision makes the whole architecture more pay for play[1].
- **ComputeSharp.Pix**: an extension library for ComputeSharp to enable PIX

[1]This package is being slightly altered in the upcoming 3.0 release. Specifically, support for runtime shader compilation has been removed, as the recommendation is to always compile shaders ahead of time. The new version of ComputeSharp will also rename this package to **ComputeSharp.Dxc** (to match the name of the DXC compiler being used, for clarity), and it will only include APIs to perform shader reflection). This change also allowed the logic to compile shaders to be entirely moved to just the source generator, which further simplified the codebase as a whole. Lastly, changing the default behavior (or in fact, the only supported behavior) to be build time shader compilation also unlocks new optimizations for the compute shaders produced by ComputeSharp. For instance, it will be possible to enable the DXC flag to strip reflection data by default, which significantly reduces the size of compiled shaders (which in turn reduces binary size in consuming applications and libraries).

[27] support to produce debugging information. This library bundles the redistributable PIX native binary and exposes some extension APIs that allow developers to queue log messages of various formats into their command queues. These can then be visualized using the PIX debugging tool to investigate all sorts of performance problems.

- **ComputeSharp.Uwp**: a UWP library with controls to render DX12 shaders powered by ComputeSharp. Specifically, this library bundles two custom `SwapChainPanel` [28] controls: `ComputeShaderPanel` and `AnimatedComputeShaderPanel`. These allow developers to very easily integrating a DirectX 12 compute shader powered by ComputeSharp into their UWP XAML applications, with optional animation support. The controls also offer some additional options such as dynamic resolution scaling, play/pause, and more.

- **ComputeSharp.WinUI**: a WinUI 3 library that mirrors ComputeSharp.Uwp, just for WinUI 3 instead of UWP XAML.

- **ComputeSharp.D2D1**: a library to write D2D1 pixel shaders entirely with C# code, and to easily register and create `ID2D1Effect`-s from them. This library is a "parallel" library to ComputeSharp, which bundles a D2D backend instead of a DirectX 12 one. Similarly, it also primarily supports D2D pixel shaders, as opposed to DX12 compute shaders. Another characteristic is that compared to ComputeSharp, this library is meant to be much more lowlevel, and not exposing the same kind of user friendly, high level APIs that ComputeSharp instead offers. The reason for this is that D2D itself is quite an advanced library that you wouldn't be able to use directly from C# anyway, so this library only provides foundational support for it. The high level wrappers built on top of this are provided by the package referencing Win2D instead, which also mirrors the D2D-Win2D relationship that already existed.

- **ComputeSharp.D2D1.Uwp**: a UWP library with APIs to leverage D2D1 functionality with D2D1 pixel shaders powered by ComputeSharp.D2D1. This is the "high level wrapper" for ComputeSharp.D2D1 mentioned above: this library references Win2D and exposes easy to use APIs for developers to plug their custom D2D pixel shaders into a Win2D drawing pipeline and use them in their own applications.

- **ComputeSharp.D2D1.WinUI**: a WinUI 3 library that mirrors ComputeSharp.D2D1.Uwp. This is the same kind of relationship as with ComputeSharp.WinUI and ComputeSharp.Uwp: same public APIs, and just targeting a different UI framework (WinUI 3 instead of UWP XAML).

Here we can also see the dependency graph of all these various packages:

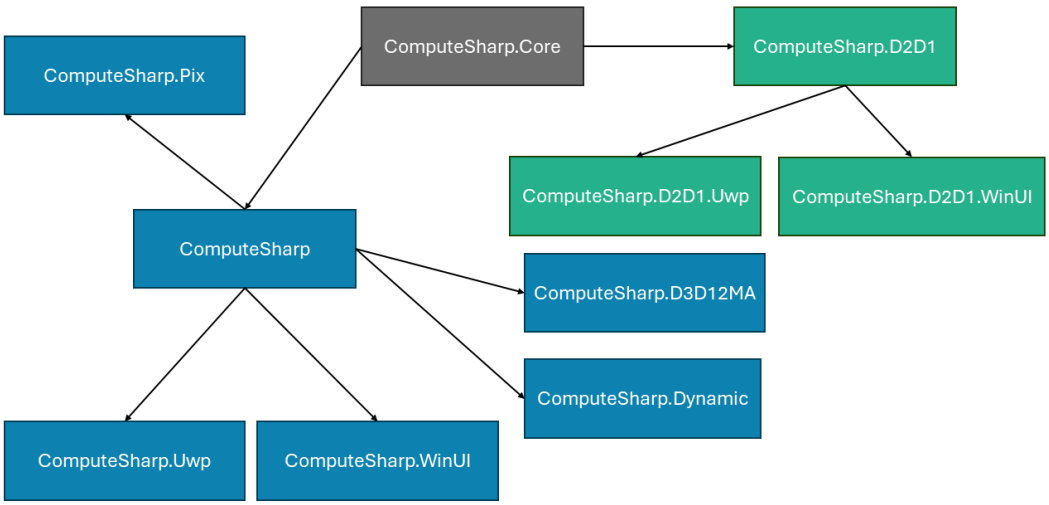


Figure 2.1. ComputeSharp dependency graph

2.2 Roslyn source generators

One of the most important characteristics of ComputeSharp is that it's a purely C# library, and it allows users to just write C# code to implement their shaders. Of course, DirectX shaders (of all kinds, so both compute shaders and pixel shaders) actually need to be written in HLSL [29] (High-level shader language), which is a C-like proprietary programming language designed to author programmable DirectX shader. Requiring ComputeSharp users (or, C# developers in general) to learn and write HLSL would have significantly made the whole experience much more difficult and less accessible for beginners, not to mention it would've also been more complicated due to the extra work needed to configure and use multiple programming languages in the same project.

To solve this, ComputeSharp bundles some custom Roslyn [30] source generators [31], which are responsible for analyzing and transpiling the C# code representing shaders to run into HLSL, as well as optionally precompiling it at build time and embedding the resulting bytecode straight into the final assembly, along with additional metadata information that's necessary to then load and dispatch these shaders. This makes the whole process of running code on the GPU completely transparent to users, which can just focus on learning the various APIs exposed by ComputeSharp, and then using them in custom shader types which are then executed on the GPU.

Before going into the details of the D2D1 source generator, he's a general overview

of the architecture of all Roslyn source generators. These are essentially plugins for the C# compiler (ie. Roslyn), which are executed both when building a project as well as during normal code edits, allowing them to generate additional source files that also benefit from IntelliSense [32] and code completion support within the IDE being used. Here's what the general architecture of a source generator looks like:

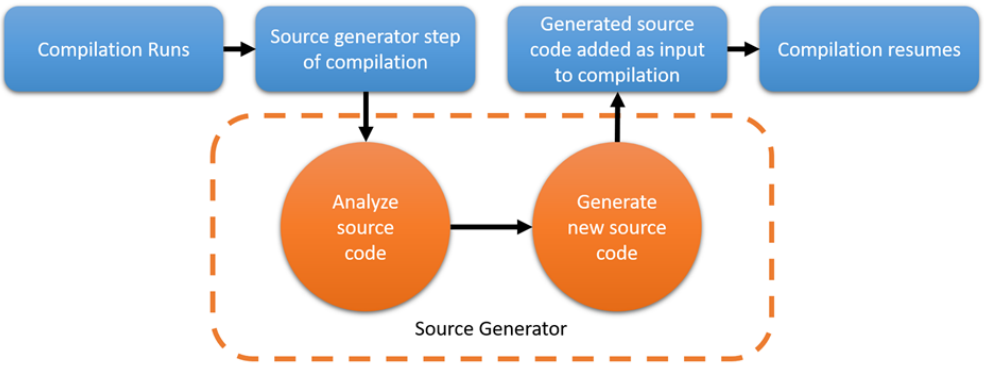


Figure 2.2. Roslyn source generator pipeline

As we can see, source generators receive an initial compilation object as input, allowing them to perform all necessary analysis on the source code being compiled, both in terms of syntax nodes [33] as well as symbol objects [34]. This is how the ComputeSharp source generators are able to inspect all code related to shaders being used, and then perform the necessary steps to generate all supporting code that will power their execution at runtime.

2.3 D2D1 source generator

For the purposes of this document, we'll only cover the D2D1 source generator, which is bundled with ComputeSharp.D2D1. There is also another source generator, bundled with ComputeSharp, which is responsible for processing DirectX 12 compute shaders. While it does have some similarities with the D2D1 source generator, it has several key differences, which are not relevant here due to the focus on D2D pixel shaders.

As mentioned above, the D2D1 source generator has several tasks:

- Analyzing and transpiling the C# source into HLSL source
- Processing all intrinsic APIs being used (ie. HLSL projections)
- Generating all marshalling code for the managed/unmanaged transitions

- Generating loading code for all additional metadata for the shader

In short, the main advantage of this is that all this work is done upfront while compiling a project, and no introspection at all is needed at runtime. This significantly makes the execution of shaders much faster, and it also makes the code trim-friendly and AOT-friendly. Older versions of ComputeSharp relied instead on runtime reflection and even decompiled the IL code to extract the C# code to transpile, but that was of course slower, and also not viable at all in AOT scenarios (where there is no IL to decompile).

2.3.1 `ID2D1PixelShader` interface

To start, let's look at the public interface for a D2D1 pixel shader. This is the interface that users of ComputeSharp.D2D1 are meant to interact with directly: they'll be implementing this interface in each custom shader, and they'll also see this interface present in all signatures of public APIs that are working with pixel shaders in some capacity.

```
namespace ComputeSharp.D2D1;

public interface ID2D1PixelShader : ID2D1Shader
{
    Float4 Execute();
}
```

The interface is extremely simple, and just exposes that `Execute` entry point. This matches the entry point method in the final HLSL code, which is the one that will be executed on the GPU. The resulting value is a `Float4` value, representing a pixel in 32 bits per channel precision (which is automatically converted by the GPU if necessary). All that users have to do is to implement a `partial struct` type with this interface, and write the logic for the pixel shader in that `Execute` method. From there, the D2D1 source generator is triggered and starts doing all of its work. To see exactly what code it will generate, let's go over each method inherited by that base `ID2D1Shader` interface. This is an "implementation detail" interface that's only meant to be used to support the source generator, and to give the library code a way to hook into the generated methods from arbitrary user types. We can go over each declared method and an example implementation to see what the generated code looks like exactly[2].

[2]As we'll also see in the last chapter, this API shape has been modified in newer version of ComputeSharp, with all the logic with for generated methods being moved to a new `D2D1PixelShaderDescriptor<T>` interface. This document will outline the API shape as of version 2.1 of the library. There are no major conceptual differences, so most of what is described below still generally applies, just in a slightly different form.

2.3.2 Implementing a D2D shader

First, let's look at the sample shader we'll use to inspect its corresponding generated code. This will make it easier to understand how all the various components fit together, as they'll match the code from the shader type itself:

```
[D2DInputCount(2)]
[D2DInputSimple(0)]
[D2DInputComplex(1)]
[D2DInputDescription(0, D2D1Filter.MinMagMipPoint)]
[D2DInputDescription(1, D2D1Filter.MinMagMipLinear)]
[D2DOutputBuffer(D2D1BufferPrecision.Float32, D2D1ChannelDepth.Four)]
[D2DShaderProfile(D2D1ShaderProfile.PixelShader50)]
[D2DPixelOptions(D2D1PixelOptions.TrivialSampling)]
[D2DCompileOptions(D2D1CompileOptions.Default | D2D1CompileOptions.IeeeStrictness)]
[D2DRequiresScenePosition]
[AutoConstructor]
public readonly partial struct DemoShader : ID2D1PixelShader
{
    private readonly float offsetX;
    private readonly float offsetY;
    private readonly int length;
    private readonly int2 size;

    [D2DResourceTextureIndex(2)]
    private readonly D2D1ResourceTexture1D<float> factors;

    public float4 Execute()
    {
        float result = 0;

        for (int i = 0; i < this.length; i++)
        {
            float4 a = D2D.SampleInputAtOffset(0, new float2(i - this.offsetX, this.size.Y));
            float4 b = D2D.SampleInputAtOffset(1, new float2(this.size.X, i - this.offsetY));

            result += Hlsl.Dot(a, b);
        }

        return this.factors[(int)Hlsl.Lerp(result, 0, this.length)];
    }
}
```

In this shader, we can see almost all of the ComputeSharp.D2D1 features at play. We have a shader with two inputs, where one is marked as using simple sampling, and one using complex sampling. The shader also indicates custom sampling modes for each input, and it requires the output buffer to use full 32 bit precision for each

channel. The shader is being precompiled with the 5.0 profile, and the additional "trivial sampling" option is also specified. Lastly, the shader is compiled with the additional IEEE strictness option enabled, and it requires access to the current position within each dispatched thread (as that is used within the shader code). We can also see how the shader is not only using two input textures, but it's also capturing one D2D resource texture to provide additional values loaded at runtime.

2.3.3 `InitializeFromDispatchData` method

The first method being generated is `InitializeFromDispatchData`:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
void ID2D1Shader.InitializeFromDispatchData(ReadOnlySpan<byte> data)
{
    if (data.IsEmpty)
    {
        return;
    }

    ref byte r0 = ref MemoryMarshal.GetReference(data);
    offsetX = Unsafe.As<byte, float>(ref Unsafe.AddByteOffset(ref r0, (nint)0));
    offsetY = Unsafe.As<byte, float>(ref Unsafe.AddByteOffset(ref r0, (nint)4));
    length = Unsafe.As<byte, int>(ref Unsafe.AddByteOffset(ref r0, (nint)8));
    size = Unsafe.As<byte, Int2>(ref Unsafe.AddByteOffset(ref r0, (nint)16));
}
```

This method is responsible for marshalling data back from a D2D constant buffer to a managed instance of a shader type. The data from either end cannot just be directly copied, as the D2D constant buffer has some very specific alignment requirements, meaning that the marshalled data might not have the same layout of the fields in the managed shader type. This method allows users to easily get a managed instance of their shader type back from a given D2D constant buffer, for instance from an instantiated Win2D effect type (more on this below). Worth noting that just like all other generated methods in this interface, and as the attributes over the generated code suggest, none of these methods are meant to be called directly by users. Rather, users would indirectly invoke these via the public APIs exposed

by ComputeSharp.D2D1, such as `PixelShader` and `PixelShaderEffect`[34].

2.3.4 `GetPixelOptions` method

Next, we have `GetPixelOptions`:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
readonly uint ID2D1Shader.GetPixelOptions()
{
    return 1;
}
```

This method exposes the pixel options annotated over the shader type, if any. In this case, because the shader is indicating the `D2D1PixelOptions.TrivialSampling` option, this method is returning 1, which is simply the raw hardcoded value corresponding to that native option (ie. the `D2D1_PIXEL_OPTIONS` [35] value). We can see how unlike the previous generated method, and like all other generated methods, `GetPixelOptions` is also marked as `readonly`, as it doesn't modify any instance state on the target pixel shader value. This allows the compiler to opportunistically remove some safety copies, which can further improve performance.

2.3.5 `GetInputCount` method

Similar to the previous method, the next one is `GetInputCount`:

[3] In newer versions of the library, this code is replaced by a new generated `ConstantBuffer` type mapping the native layout of the constant buffer for the shader. Marshalling data back and forth is then lowered as just copying all fields of the shader types to and from these generated native layout types. Doing so results in much less verbose and easier to understand code. The specific logic to handle marshalling is emitted in a separate new type, `ConstantBufferMarshaller`.

[4] For both this and all other interface methods on the new `ID2D1PixelShaderDescriptor<T>` interface, future work will also include switching all APIs to leverage static abstract interface methods. This will both make them easier to read (due to no longer looking like they might be accessing instance state), and it will allow them to result in better codegen. This is because each API will be called directly and will no longer have to go through an uninitialized dummy instance of a shader type used as a local to invoke the instance API on. This can particularly be beneficial in cases where the compiler isn't able to fully elide those locals, and when the shader type is sufficiently large in size.

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
readonly uint ID2D1Shader.GetInputCount()
{
    return 2;
}
```

As the name suggests, this method simply exposes the number of inputs for the current shader. This is used in several places inside ComputeSharp.D2D1, such as when configuring the internal D2D effect implementation and its effect graph, or when validating inputs being set by the user on effect instances. The returned value simply matches the value provided to `[D2DInputCount]` on the shader type[5].

2.3.6 `GetInputType` method

Next, we have `GetInputType`:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
readonly uint ID2D1Shader.GetInputType(uint index)
{
    return (2U >> (int)index) & 1;
}
```

This method is responsible for providing the backend an easy and fast way of checking the type of each declared input for the shader. There are only two possible values: simple or complex. The distinction has very important semantics differences, and it can also affect the performance of a given effect graph, as it directly impacts whether effects can be linked together or not [36]. The method is returning a value extracted from a bitmask that's constructed by the source generator, which has a 1 in each position where a complex input is located. For this shader, there are two inputs, and only the second one is complex, so the bitmask has a 0 for the first input, and 1 for the second input, as expected.

[5] In newer versions of the library, this method and all other ones simply returning a value have been converted to properties. These APIs being methods was mostly a style remnant from earlier releases, with the idea being that using methods would better signal how these APIs were only meant to be used to support the infrastructure, and not something for general users to ever look at. With the new `ID2D1PixelShaderDescriptor<T>` interface, this is now no longer needed.

2.3.7 `LoadInputDescriptions` method

Continuing on with metadata related methods, we also have `LoadInputDescriptions`:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
[SkipLocalsInit]
readonly void ID2D1Shader.LoadInputDescriptions<TLoader>(ref TLoader loader)
{
    Span<byte> data = stackalloc byte[24];
    ref byte r0 = ref data[0];
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)0)) = 0;
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)4)) = 0;
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)8)) = 0;
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)12)) = 1;
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)16)) = 21;
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)20)) = 0;
    loader.LoadInputDescriptions(data);
}
```

This method is responsible for loading all the input descriptions specified on the shader type, if any. The code seems particularly obscure because the resulting metadata information is serialized into a binary buffer (which is done on purpose to avoid leaking internal implementation types on the API surface only meant to be used by the source generator), but what it's doing is simply supplying data to construct a sequence of D2D1 input description values. Each value matches a type with the following definition:

```
struct InputDescription
{
    int index;
    D2D1_FILTER filter,
    int levelOfDetailCount;
};
```

These values are then internally marshalled back by ComputeSharp.D2D1 and they are used to return a higher level `D2D1InputDescription` value to users, when requested. Once again, this is both used internally by the effect implementation to correctly configure the effect graph and all related properties, as well by public APIs that allow users to easily do introspection on shader types and resolve their

metadata information at runtime, if needed[6].

2.3.8 `LoadResourceTextureDescriptions` method

Very similarly to `LoadInputDescriptions`, `LoadResourceTextureDescriptions` is also generated[7], which does conceptually the same work as the previous method, but gathering metadata for resource textures instead:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
readonly void ID2D1Shader.LoadResourceTextureDescriptions<TLoader>(ref TLoader loader)
{
    Span<byte> data = stackalloc byte[8];
    ref byte r0 = ref data[0];
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)0)) = 2;
    Unsafe.As<byte, uint>(ref Unsafe.AddByteOffset(ref r0, (nint)4)) = 1;
    loader.LoadResourceTextureDescriptions(data);
}
```

2.3.9 `GetOutputBuffer` method

Next, `GetOutputBuffer` is also generated:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
readonly void ID2D1Shader.GetOutputBuffer(out uint precision, out uint depth)
{
    precision = 5;
    depth = 4;
}
```

This method is responsible for exposing the values passed through `[D2DOutputBuffer]`. Like mentioned previously, the intent here is once again not to leak any implementation types from the API surface of these generated methods, so

[6] In newer versions of the library, this method is a property, which returns a `ReadOnlyMemory<D2D1InputDescription>` value. This is lowered to a cached static array in the generated code. Doing so results once again in much more readable code.

[7] Just like with `LoadInputDescriptions`, in newer versions of the library this method is also converted to a property, which simply returns a `ReadOnlyMemory<D2D1ResourceTextureDescription>` value. This also allows removing the supporting `ID2D1ResourceTextureDescriptionsLoader` interface entirely, which further simplifies both the public API surface and the generated code.

these values are simply returned back as `uint` values, and then converted internally by ComputeSharp.D2D1. In reality, they are `D2D1_BUFFER_PREVISION` [37] and `D2D1_CHANNEL_DEPTH` [38] enum values, respectively. For instance, as we can see in this method, the returned values are actually `D2D1_BUFFER_PRECISION_32BPC_FLOAT` and `D2D1_CHANNEL_DEPTH_4`.

2.3.10 `LoadDispatchData` method

Next, we have `LoadDispatchData`. This is conceptually the opposite of `InitializeFromDispatchData`: rather than setting all fields of the shader value from a given D2D constant buffer, it instead takes a shader value as input and creates a D2D constant buffer with its dispatch data, making sure to also respect all D2D alignment rules:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
[SkipLocalsInit]
readonly void ID2D1Shader.LoadDispatchData<TLoader>(ref TLoader loader)
{
    Span<byte> data = stackalloc byte[24];
    ref byte r0 = ref data[0];
    Unsafe.As<byte, float>(ref Unsafe.AddByteOffset(ref r0, (nint)0)) = offsetX;
    Unsafe.As<byte, float>(ref Unsafe.AddByteOffset(ref r0, (nint)4)) = offsetY;
    Unsafe.As<byte, int>(ref Unsafe.AddByteOffset(ref r0, (nint)8)) = length;
    Unsafe.As<byte, Int2>(ref Unsafe.AddByteOffset(ref r0, (nint)16)) = size;
    loader.LoadConstantBuffer(data);
}
```

In this case, we can see that there are 4 bytes of padding between ”length” and ”size” in the resulting buffer. This is because the alignment rules require vector types to be aligned to 16 bytes boundary. All this additional complexity is an example of the kind of work that the source generator takes care of automatically, behind the scenes, so that users don’t have to think about any of this.

2.3.11 `BuildHlslSource` method

Finally, we have the transpiled HLSL source, which is produced by `BuildHlslSource`:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(global::System.ComponentModel.EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
readonly void ID2D1Shader.BuildHlslSource(out string hlslSource)
{
    hlslSource = """
        // ================================================
        //                   AUTO GENERATED
        // ================================================
        // This shader was created by ComputeSharp.
        // See: https://github.com/Sergio0694/ComputeSharp.

        #define D2D_INPUT_COUNT 2
        #define D2D_INPUT0_SIMPLE
        #define D2D_INPUT1_COMPLEX
        #define D2D_REQUIRES_SCENE_POSITION

        #include "d2d1effecthelpers.hlsli"

        float offsetX;
        float offsetY;
        int __reserved__length;
        int2 size;

        Texture1D<float> factors : register(t2);
        SamplerState __sampler__factors : register(s2);

        D2D_PS_ENTRY(Execute)
        {
            float result = 0;
            for (int i = 0; i < __reserved__length; i++)
            {
                float4 a = D2DSampleInputAtOffset(0, float2(i - offsetX, size.y));
                float4 b = D2DSampleInputAtOffset(1, float2(size.x, i - offsetY));
                result += dot(a, b);
            }

            return factors[(int)lerp(result, 0, __reserved__length)];
        }
        """;
}
```

Here we can see not only how all the shader code has been correctly transpiled to HLSL, but also how the final shader code has some additional code inserted into it to correctly forward all the info that the shader compiler needs in order

to compile the shader successfully, such as input descriptions. Additionally, the "d2d1effecthelpers.hlsli" header is also imported, as that's what exposes all the D2D helper APIs for HLSL [39]. Note that in case the shader is precompiled at build time, this embedded source is not actually used by the ComputeSharp.D2D1 backend when dispatching shaders. However, it's still embedded in the final assembly so that users can also retrieve it at runtime in case they wanted to do further introspection on the generated code while running their applications, or for debugging purposes.

2.3.12 `LoadBytecode` method

And to complete the interface, here's the last generated method, `LoadBytecode`:

```
[GeneratedCode("ComputeSharp.D2D1.SourceGenerators.ID2D1ShaderGenerator", "2.1.0.0")]
[DebuggerNonUserCode]
[ExcludeFromCodeCoverage]
[EditorBrowsable(EditorBrowsableState.Never)]
[Obsolete("This method is not intended to be used directly by user code")]
readonly void ID2D1Shader.LoadBytecode<TLoader>(
    ref TLoader loader,
    ref D2D1ShaderProfile? shaderProfile,
    ref D2D1CompileOptions? compileOptions)
{
    if (shaderProfile is null or D2D1ShaderProfile.PixelShader50 &&
        compileOptions is null or (D2D1CompileOptions.IeeeStrictness | D2D1CompileOptions.Default))
    {
        ReadOnlySpan<byte> bytecode = new byte[]
        {
            0x44, 0x58, 0x42, 0x43, 0xE1, 0xE6, 0x59, 0xAF, 0xE9, 0xEB, 0xAA, 0x52, 0x12, 0xA8, ...
        };
        loader.LoadEmbeddedBytecode(bytecode);
        shaderProfile = D2D1ShaderProfile.PixelShader50;
        compileOptions = (D2D1CompileOptions.IeeeStrictness | D2D1CompileOptions.Default);
    }
    else
    {
        D2D1ShaderProfile effectiveShaderProfile = shaderProfile ??= D2D1ShaderProfile.PixelShader50;
        D2D1CompileOptions effectiveCompileOptions = compileOptions ??=
            (D2D1CompileOptions.IeeeStrictness | D2D1CompileOptions.Default);

        D2D1ShaderCompiler.LoadDynamicBytecode(
            ref loader,
            in this,
            effectiveShaderProfile,
            effectiveCompileOptions);
    }
}
```

As the name suggestes, this method is responsible for retrieving the precompiled bytecode, or compiling the bytecode on the fly. As we can see, the generated method already has the whole precompiled shader bytecode embedded into it (this gets lowered to an RVA field, ie. a static data segment in the loaded PE file, which is extremely fast to access). This is directly returned to the caller in case the bytecode is requested with the same options that were used to generate the bytecode during compile time (or if no options are specified, which means the default ones are used). Otherwise, if that's not the case, the shader is just compiled on the fly using the transpiled HLSL source we saw previously. This ensures that all cases are covered, even though of course under normal use, it is recommended to just leverage the precompiled bytecode, as that significantly improves the dispatch time for the first shader invocation[8].

2.4 Intrinsics support

As mentioned above as well, ComputeSharp also includes several intrinsic APIs, ie. projections for HLSL and D2D methods and types. These are exposed to make it more convenient to write C# shaders, using a syntax that closely matches that of HLSL. It also has the additional benefit that allows shaders to express logic that's normally only available in HLSL, such as swizzled accesses to HLSL vector and matrix types.

2.4.1 Intrinsics types

There are 3 categories of intrinsics exposed by ComputeSharp.D2D1 (some also being shared with ComputeSharp):

- **HLSL intrinsics**: these are special HLSL built-in functions [40] that can be used to perform mathematical and logical operations in a very efficient way. They are all hardware accelerated, and much more efficient than what the same operation would be if it was implemented from scratch in a shader. All these functions are exposed in the `Hlsl` class.

- **HLSL types**: these are projections for all built-in HLSL vector and matrix types [41]. The advantage of being usable from C#, as mentioned above, is that they can both be used along with HLSL intrinsic APIs, as well as within

[8] In newer versions of the library, the shader bytecode (along with the input types), are instead lowered to `ReadOnlyMemory<T>` properties, which are backed by a generated `MemoryManager<T>` type using an RVA field for storage. This provides extremely fast performance while still exposing a very user friendly API that is easy to read and consume.

normal methods, and they also expose the same additional APIs available when authoring HLSL shaders manually.

- **D2D helpers**: these are additional APIs [39] exposed by the D2D header that's referenced by all D2D custom effects. They include methods that can index from textures, perform interpolation, get the current dispatch position, and more.

2.4.2 Intrinsics implementation

It's worth noting that all these intrinsic APIs are really just projections, and they're not implemented in C#. As such, they're not usable unless they're actually being executed within a shader. To illustrate this, let's look at the implementation of one of these APIs, specifically one HLSL intrinsic method:

```
namespace ComputeSharp;

public static partial class Hlsl
{
    public static void AllMemoryBarrier()
    {
        throw new InvalidExecutionContextException("Hlsl.AllMemoryBarrier()");
    }

    [HlslIntrinsicName("abs")]
    public static Int4 Abs(Int4 x)
    {
        return default;
    }
}
```

Here we can see two different types of HLSL intrinsic APIs:

- `AllMemoryBarrier` returns `void`, and as such it just never makes sense to be called from outside a shader. For this reason, the implementation simply always throws an exception to let developers know that they're calling the API from the wrong context.
- `Abs` returns a value, so developers might also want to use it from a `static readonly` field initializer within a shader. To allow this, all these methods returning a value are simply returning `default` when executed on the CPU, which means using them in a field initialier wouldn't crash the process. Other than that, this method is effectively just as unusable as the `void` returning one.

2.4.3 Intrinsics rewriting

The way all these methods are actually enabled is by having the source generator replace them with the correct invocation during the HLSL source code rewriting (in fact, we can see that `[HlslIntrinsicName]` attribute too to inform the generator that it should be using a particular name for that intrinsic API). This way, the resulting HLSL source only contains calls to valid HLSL intrinsic methods, which can then be executed normally on the GPU.

Let's also look at an example of a swizzled access API on a vector type:

```
namespace ComputeSharp;

[StructLayout(LayoutKind.Explicit, Size = 16, Pack = 4)]
public unsafe partial struct Float4
{
    private static readonly void* UndefinedData =
        (void*)RuntimeHelpers.AllocateTypeAssociatedMemory(typeof(Float4), sizeof(Float4));

    [UnscopedRef]
    public readonly ref readonly Float4 XZWX => ref *(Float4*)UndefinedData;

    [UnscopedRef]
    public ref Float4 XZWY => ref *(Float4*)UndefinedData;
}
```

This is just a small fragment of the `Float4` type, but we can see how it has explicit layout to match the size it would have in HLSL (to make marshalling operations simpler and faster), and it exposes all possible combinations of swizzled properties (these are just two of them, but all combinations are present). Of course, given C# doesn't actually support swizzled memory indexing, these just do nothing, and return the address of some undefined data. This is done once again to make these properties still be usable from field initializers of static shader fields. We can also see how the return type changes from `ref` to `ref readonly` depending on whether any item in the swizzled vector is repeated to match the rules for HLSL. This further improves the user experience by providing additional compile time checks when writing shader code, just like the use of `[UnscopedRef]` also does, by allowing the compiler to verify that no instance references escape to an invalid scope.

2.5 Interop API surface

Now that all the generated code and architecture for HLSL intrinsic APIs has been laid out, we can look at the actual public API surface that ComputeSharp.D2D1

exposes to register effects and instantiate them. The core of the functionality is exposed through the `D2D1PixelShaderEffect` type, which has the following APIs:

```
namespace ComputeSharp.D2D1.Interop;

public static unsafe class D2D1PixelShaderEffect
{
    public static void RegisterForD2D1Factory1<T>(void* d2D1Factory1, out Guid effectId)
        where T : unmanaged, ID2D1PixelShader;

    public static ReadOnlyMemory<byte> GetRegistrationBlob<T>(out Guid effectId)
        where T : unmanaged, ID2D1PixelShader;

    public static void CreateFromD2D1DeviceContext<T>(void* d2D1DeviceContext, void** d2D1Effect)
        where T : unmanaged, ID2D1PixelShader;

    public static void SetConstantBufferForD2D1Effect<T>(void* d2D1Effect, in T shader)
        where T : unmanaged, ID2D1PixelShader;

    public static void SetResourceTextureManagerForD2D1Effect(
        void* d2D1Effect,
        void* resourceTextureManager,
        int resourceTextureIndex);

    public static void SetResourceTextureManagerForD2D1Effect(
        void* d2D1Effect,
        D2D1ResourceTextureManager resourceTextureManager,
        int resourceTextureIndex);

    public static void SetTransformMapperForD2D1Effect(void* d2D1Effect, void* transformMapper);

    public static void SetTransformMapperForD2D1Effect<T>(
        void* d2D1Effect,
        D2D1TransformMapper<T> transformMapper)
        where T : unmanaged, ID2D1PixelShader;
}
```

As we can see, this matches what we previously said about this library being more lowlevel compared to eg. ComputeSharp and ComputeSharp.D2D1.Uwp, in that it directly exposes APIs that are meant to interact with native D2D objects. But, this is all that's needed to provide the foundation for other higher level wrappers. For instance, Paint.NET [42] is also leveraging ComputeSharp.D2D1 to power all of its GPU accelerated effects, which are also written in C# as ComputeSharp.D2D1 shaders. As a side note, this is why that `GetRegistrationBlob` method exists, as it allows custom plugins for Paint.NET, which might use a different version of ComputeSharp.D2D1 than the main application, to pass all necessary information

to register a D2D effect, without any public APIs from Paint.NET's plugin SDK needing to expose any ComputeSharp specific type, which could cause problems in case there was a version mismatch between either side.

Figure 2.3. Paint.NET running a ComputeSharp.D2D1 shader

Going back to the `D2D1PixelShaderEffect` type, we can see how the public APIs are conceptually grouped into different categories: registering and instantiating an effect, and interacting with an instantiated effect. The entry point is that `RegisterForD2D1Factory1` method, which is responsible for registering a new D2D effect using the input shader type on the target D2D factory. This leverages a custom `ID2D1Effect` implementation, provided by ComputeSharp.D2D1, which is effectively a combination of the `ID2D1EffectImpl` and `ID2D1DrawTransform` interfaces. These are attached to a custom type (which employs COM composition to implement them on the same object). This type is only an implementation detail, and it is never exposed to users, neither from ComputeSharp.D2D1 nor through D2D itself (which in fact completely hides concrete effect instances behind the higher level `ID2D1Effect` interface).

It's worth going into more detail on the way ComputeSharp.D2D1 allows users to manage D2D resource textures, as well as custom transform mapping. These may be useful when implementing advanced effects and when being particularly careful about performance (for instance, a convolution effect may have an input

with complex sampling, but it will only ever need to access pixels from a given area relative to each output pixel, and D2D can leverage this information to further optimize the tiles subdivision during rendering). ComputeSharp.D2D1 exposes a set of high level C# APIs to manage both resource textures and to implement transform mappers, as well as some custom COM interfaces that these wrappers implement. These interfaces serve two purposes: they are the mechanism with which these high level C# wrappers interact with ComputeSharp.D2D1's `ID2D1EffectImpl` object, and they also allow consumers to implement those interfaces from scratch if they didn't want (or couldn't) use the built-in high level wrappers for any reason.

2.5.1 D2D resource textures

First, let's go over the COM interfaces for D2D resource textures:

```
[uuid(3C4FC7E4-A419-46CA-B5F6-66EB4FF18D64)]
class ID2D1ResourceTextureManager : IUnknown
{
    HRESULT Initialize(
        [in, optional] const GUID                              *resourceId,
        [in]           const D2D1_RESOURCE_TEXTURE_PROPERTIES *resourceTextureProperties,
        [in, optional] const BYTE                              *data,
        [in, optional] const UINT32                            *strides,
                       UINT32                                  dataSize);

    HRESULT Update(
        [in, optional] const UINT32 *minimumExtents,
        [in, optional] const UINT32 *maximimumExtents,
        [in]           const UINT32 *strides,
                       UINT32       dimensions,
        [in]           const BYTE   *data,
                       UINT32       dataCount);
};
```

This interface represents a manager object for D2D resource textures, which can then be used within effects and shares among them. For details of how these work, `Initialize` maps to `ID2D1EffectContext::CreateResourceTexture` [43], whereas `Update` maps to `ID2D1ResourceTexture::Update` [44]. The same behavior and parameters as those methods is used. This interface is implemented by ComputeSharp.D2D1, and it can be used through the APIs in `D2D1ResourceTextureManager`. That is, `D2D1ResourceTextureManager.Create` is first used to create an `ID2D1ResourceTextureManager` instance. Then, `Initialize` can be used to initialize the resource texture held by the manager. The manager can also be assigned to an effect at any time, using the available property indices from

`D2D1PixelShaderEffectProperty`. To update texture data after its initial creation, `Update` can be used. An RCW (Runtime Callable Wrapper [45]) is also available for all of these APIs, implemented by the same `D2D1PixelShaderEffect` type. In this case, the constructor can be used to create and initialize an instance (equivalent to calling `D2D1ResourceTextureManager.Create` and `Initialize`), and then `Update` is available for texture data updates. The instance implements `ICustomQueryInterface`, which can be used in advanced scenarios to retrieve the underlying COM object.

The `ID2D1ResourceTextureManager` contract allows callers to create a resource texture at any time even before the effect has been initialized and the manager assigned to it. In that case, the manager will buffer the data internally, and will defer the creation of the actual resource texture until it's possible for it to do so, at which point it will also free the buffer.

If implementing a custom resource manager is needed, in order for external implementations to be accepted by the effect objects created by `D2D1PixelShaderEffect`, the `ID2D1ResourceTextureManagerInternal` interface also needs to be implemented:

```
[uuid(5CBB1024-8EA1-4689-81BF-8AD190B5EF5D)]
class ID2D1ResourceTextureManagerInternal : IUnknown
{
    HRESULT Initialize(
        [in]           const ID2D1EffectContext *effectContext,
        [in, optional] const UINT32             *dimensions);

    HRESULT GetResourceTexture([out] ID2D1ResourceTexture **resourceTexture);
};
```

Note that due to the fact `ID2D1EffectContext` and `ID2D1ResourceTexture` are not thread safe, they are only safe to use without additional synchronization when `Initialize` and `GetResourceTexture` are called. This happens from the internal `ID2D1EffectImpl` object, which is invoked by D2D, which will handle taking the necessary lock before invoking these APIs. Due to the fact the context has to be stored internally, custom implementations have to make sure to retrieve the right `ID2D1Multithread` instance from the effect context and use that to synchronize all accesses to those two objects from all APIs (including those exposed by `ID2D1ResourceTextureManager`). The built-in implementation of these two interfaces provided by `D2D1ResourceTextureManager` takes care of all this.

The "dimensions" parameter in `ID2D1ResourceTextureManagerInternal::Initialize` is optional, and can be passed by an `ID2D1Effect` when initializing an input resource texture manager, if the dimensions of the resource texture at the target

index are known in advance. This can allow the manager to perform additional validation when the resource texture is initialized. Not providing a value is not an error, but if a resource texture with invalid size is used the effect might fail to render later on and be more difficult to troubleshoot.

2.5.2 D2D transform mappers

Next, let's go over the COM interfaces for custom transform mappers as well:

```
[uuid(02E6D48D-B892-4FBC-AA54-119203BAB802)]
class ID2D1TransformMapper : IUnknown
{
    HRESULT MapInputRectsToOutputRect(
        [in]  const ID2D1DrawInfoUpdateContext* updateContext,
        [in]  const RECT*                       inputRects,
        [in]  const RECT*                       inputOpaqueSubRects,
              UINT32                            inputRectCount,
        [out] RECT*                             outputRect,
        [out] RECT*                             outputOpaqueSubRect);

    HRESULT MapOutputRectToInputRects(
        [in]  const RECT* outputRect,
        [out] RECT*       inputRects,
              UINT32      inputRectsCount);

    HRESULT MapInvalidRect(
              UINT32 inputIndex,
              RECT   invalidInputRect,
        [out] RECT*  invalidOutputRect);
};
```

These can be used to implement custom draw transform logic, assign it to an effect and also share these transforms over multiple effects. For details of how these work, `MapInputRectsToOutputRect` maps to `ID2D1Transform::MapInputRectsToOutputRect` [46], `MapOutputRectToInputRects` maps to `ID2D1Transform::MapOutputRectToInputRects` [47], and `MapInvalidRect` maps to `ID2D1Transform::MapInvalidRect` [48]. The main difference between these APIs and the ones in `ID2D1Transform` is the fact that this interface is standalone (ie. it doesn't inherit from `ID2D1TransformNode`), and that it allows transform mappers to also read and update additional data tied to an effect instance (such as the shader constant buffer). This is done through the `ID2D1DrawInfoUpdateContext` interface, that is passed to `MapInputRectsToOutputRect`. This interface is defined as follows:

```
[uuid(430C5B40-AE16-485F-90E6-4FA4915144B6)]
class ID2D1DrawInfoUpdateContext : IUnknown
{
    HRESULT GetConstantBufferSize([out] UINT32 *size);

    HRESULT GetConstantBuffer(
        [out] BYTE   *buffer,
              UINT32 bufferCount);

    HRESULT SetConstantBuffer(
        [in] const BYTE *buffer,
             UINT32     bufferCount);
};
```

That is, `ID2D1DrawInfoUpdateContext` allows a custom transform (an `ID2D1TransformMapper` instance) to interact with the underlying `ID2D1DrawInfo` object that is owned by the effect being used, in a safe way. For instance, it allows a transform to read and update the constant buffer, which can be used to allow a transform to pass the exact dispatch area size to an input shader, without the consumer having to manually query that information beforehand (which might not be available either). This interface is implemented by ComputeSharp.D2D1, and it can be used through the APIs in `D2D1TransformMapper<T>`, in several ways. That is, consumers can either implement a type inheriting from `D2D1TransformMapper<T>` to implement their own fully customized transform mapping logic, or they can use the helper methods exposed by `D2D1TransformMapperFactory<T>` to easily retrieve ready to use transforms. A CCW (COM callable wrapper [49]) is also available for all of these APIs, implemented via the same `D2D1TransformMapper<T>` type. That is, a given instance can expose its underlying CCW through the `ICustomQueryInterface` interface, and this can then be passed to an existing D2D1 effect instance.

Chapter 3

Extending Win2D for custom effects

As we briefly mentioned above, Win2D is a lightweight WinRT library for hardware accelerated 2D graphics rendering, powered by D2D[1]. It provides several APIs to represent objects that can be drawn, which similarly to D2D, are divided into two categories: images and effects. Images, represented by the `ICanvasImage` interface, have no inputs and can be directly drawn on a given surface. For example, `CanvasBitmap`, `VirtualizedCanvasBitmap` and `CanvasRenderTarget` are examples of image types. Effects, on the other hand, are represented by the `ICanvasEffect` interface. They can have inputs as well as additional resources, and can apply arbitrary logic to produce their outputs (as an effect is also an image). Win2D includes effects wrapping most D2D effects [50], such as `GaussianBlurEffect`, `TintEffect` and `LuminanceToAlphaEffect`.

Images and effects can also be chained together, to create arbitrary graphs which can then be further optimized [36] and displayed in your application. Together, they provide an extremely flexible system to author complex graphics in an efficient manner. However, there are cases where the built-in effects are not sufficient, and you might want to build your very own Win2D effect[2]. To support this, we added a

[1]Win2D uses D2D internally, and it abstracts all the complexity of initializing and managing DirectX 11 and D2D devices, only exposing higher-level WinRT APIs to perform drawing operations. As we mentioned, it also includes some XAML controls, compatible with either UWP XAML or the WinUI 3 framework, that make it easy to insert drawn content into the visual tree. For instance, as we will see in the rest of this document as well, the Microsoft Store app leverages the `CanvasImageSource` control to draw content on screen. This is a Win2D type that can be inserted into XAML and that provides a surface to draw on via D2D, which is then applied to the visual tree via a XAML brush, which Win2D manages automatically.

[2]Worth noting, Win2D also has a `PixelShaderEffect` type, which represents a generic D2D pixel shader. It accepts compiled DXIL code (the bytecode for DirectX shaders) and then exposes properties to set inputs and constant buffer values in a generic way. It also features some options to customize how the input/output mapping is performed. This was evaluated as one of the options to

new set of powerful interop APIs to Win2D, with the goal to allow defining custom images and effects that can seamlessly integrate with Win2D.

3.1 Implementing `ICanvasImage`

The simplest scenario to support is creating a custom `ICanvasImage`. As we mentioned, this is the WinRT interface defined by Win2D which represents all kinds of images that Win2D can interop with. This interface only exposes two `GetBounds` methods, and extends `IGraphicsEffectSource` [52], which is a marker interface representing "some effect source":

```
[version(NTDDI_WINTHRESHOLD), uuid(794966D3-6A64-47E9-8DA8-B46AAA24D53B)]
interface ICanvasImage : IInspectable
    requires Windows.Graphics.Effects.IGraphicsEffectSource, Windows.Foundation.IClosable
{
    [overload("GetBounds")]
    HRESULT GetBounds(
        [in] Microsoft.Graphics.Canvas.ICanvasResourceCreator* resourceCreator,
        [out, retval] Windows.Foundation.Rect* bounds);

    [overload("GetBounds")]
    HRESULT GetBoundsWithTransform(
        [in] Microsoft.Graphics.Canvas.ICanvasResourceCreator* resourceCreator,
        [in] NUMERICS.Matrix3x2 transform,
        [out, retval] Windows.Foundation.Rect* bounds);
}
```

As we can see, there are no "functional" APIs exposed by this interface to actually perform any drawing. In order to implement a custom `ICanvasImage` object, we'll need to also implement the `ICanvasImageInterop` interface, which exposes all the necessary logic for Win2D to draw the image. This is a COM interface defined in the public `Microsoft.Graphics.Canvas.native.h` header, that ships with Win2D.

The interface is defined as follows:

integrate ComputeSharp into the Microsoft Store as well, but it resulted being too limiting, and without the ability of being extended without constituting a breaking change for existing users of this API. The main problems were that the input shaders were artificially being restricted to using the pixel shader profile[51] version 4.1 or lower, that there was no compile time validation nor a strongly typed API to set the constant buffer value of a shader, and that there was no way to customize the input/output transform logic directly, which can be critical to oprimize performance in some scenarios. Because of this, and consulting with the Win2D team, we made the decision to keep this API as is, and instead build a new effect type directly into ComputeSharp that would be designed from the ground up with all of these additional features in mind. The work to add the new interop APIs into Win2D was directly driven by the desire to implement this new `PixelShaderEffect<T>` type, which will be described in detail below.

```
[uuid(E042D1F7-F9AD-4479-A713-67627EA31863)]
class ICanvasImageInterop : IUnknown
{
    HRESULT GetDevice(
        ICanvasDevice** device,
        WIN2D_GET_DEVICE_ASSOCIATION_TYPE* type);

    HRESULT GetD2DImage(
        ICanvasDevice* device,
        ID2D1DeviceContext* deviceContext,
        WIN2D_GET_D2D_IMAGE_FLAGS flags,
        float targetDpi,
        float* realizeDpi,
        ID2D1Image** ppImage);
};
```

And it also relies on these two enumeration types, from the same header. The first one, `WIN2D_GET_DEVICE_ASSOCIATION_TYPE` is used by custom effects to indicate the kind of D2D device that they're bound to, and is defined as follows:

```
enum WIN2D_GET_DEVICE_ASSOCIATION_TYPE
{
    WIN2D_GET_DEVICE_ASSOCIATION_TYPE_UNSPECIFIED,
    WIN2D_GET_DEVICE_ASSOCIATION_TYPE_REALIZATION_DEVICE,
    WIN2D_GET_DEVICE_ASSOCIATION_TYPE_CREATION_DEVICE
}
```

The second one, `WIN2D_GET_D2D_IMAGE_FLAGS` is passed as input by D2D to custom effects and provides additional information for them on how to correctly realize their internal effect graph and produce the resulting output image for D2D to continue performing the requested drawing operation. It is defined as follows:

```
enum WIN2D_GET_D2D_IMAGE_FLAGS
{
    WIN2D_GET_D2D_IMAGE_FLAGS_NONE,
    WIN2D_GET_D2D_IMAGE_FLAGS_READ_DPI_FROM_DEVICE_CONTEXT,
    WIN2D_GET_D2D_IMAGE_FLAGS_ALWAYS_INSERT_DPI_COMPENSATION,
    WIN2D_GET_D2D_IMAGE_FLAGS_NEVER_INSERT_DPI_COMPENSATION,
    WIN2D_GET_D2D_IMAGE_FLAGS_MINIMAL_REALIZATION,
    WIN2D_GET_D2D_IMAGE_FLAGS_ALLOW_NULL_EFFECT_INPUTS,
    WIN2D_GET_D2D_IMAGE_FLAGS_UNREALIZE_ON_FAILURE
}
```

Both of these enum types will be explained in more detail below, along with all the specifics of how they should be used by custom effects, and how different values should influence the internal functionality of such effects.

The two `GetDevice` and `GetD2DImage` methods are all that's needed to implement custom images (or effects), as they provide Win2D with the extensibility points to initialize them on a given device and retrieve the underlying D2D image to draw. Implementing these methods correctly is critical to ensure things will work properly in all supported scenarios. Win2D will leverage this interface by doing a `QueryInterface` for it on an input `ICanvasImage` object when trying to draw it, in case it doesn't already implement Win2D's own internal interface for built-in effects. This is how external effects can plug into its rendering pipeline.

3.1.1 Implementing `GetDevice`

The `GetDevice` method is the simplest of the two. What it does is it retrieves the canvas device associated with the effect, so that Win2D can inspect it if necessary (for instance, to ensure it matches the device in use). The "type" parameter indicates the "association type" for the returned device. There are two main possible cases:

- If the image is an effect, it should support being "realized" and "unrealized" on multiple devices. What this means is: a given effect is created in an uninitialized state, then it can be realized when a device is passed while drawing, and after that it can keep being used with that device, or it can be moved to a different device. In that case, the effect will reset its internal state and then realize itself again on the new device. This means that the associated canvas device can change over time, and it can also be `null`. Because of this, "type" should be set to `WIN2D_GET_DEVICE_ASSOCIATION_TYPE_REALIZATION_DEVICE`, and the returned device should be set to the current realization device, if one is available[3].
- Some images have a single "owning device" which is assigned at creation time and can never change. For instance, this would be the case for an image representing a texture, as that is allocated on a specific device and cannot be moved. When `GetDevice` is called, it should return the creation device and set "type" to `WIN2D_GET_DEVICE_ASSOCIATION_TYPE_CREATION_DEVICE`. Note that when this type is specified, the returned device should not be `null`.

[3]The `GetDevice` and its associated `WIN2D_GET_DEVICE_ASSOCIATION_TYPE` type were not present in the first version of these new APIs in Win2D, and they are a result of how useful it is to test early previews in real world scenarios. We started testing a preview of Win2D featuring `ICanvasImageInterop` (along with a preview of ComputeSharp) in the Microsoft Store, and only during testing we realized that with the current logic, Win2D would crash if a device was lost while the application was running. This was because Win2D was assuming that devices on external effects could not change after realization (as it could not make any assumptions on how external effects were implemented), and as such it would throw when trying to draw an image on the new device. We noticed this, and designed `GetDevice` as a response to that, which finally allowed external effects to have the same flexibility of built-in Win2D effects in terms of device association and effect realization logic.

To clarify, here's a diagram that further shows the Win2D/D2D realization logic:

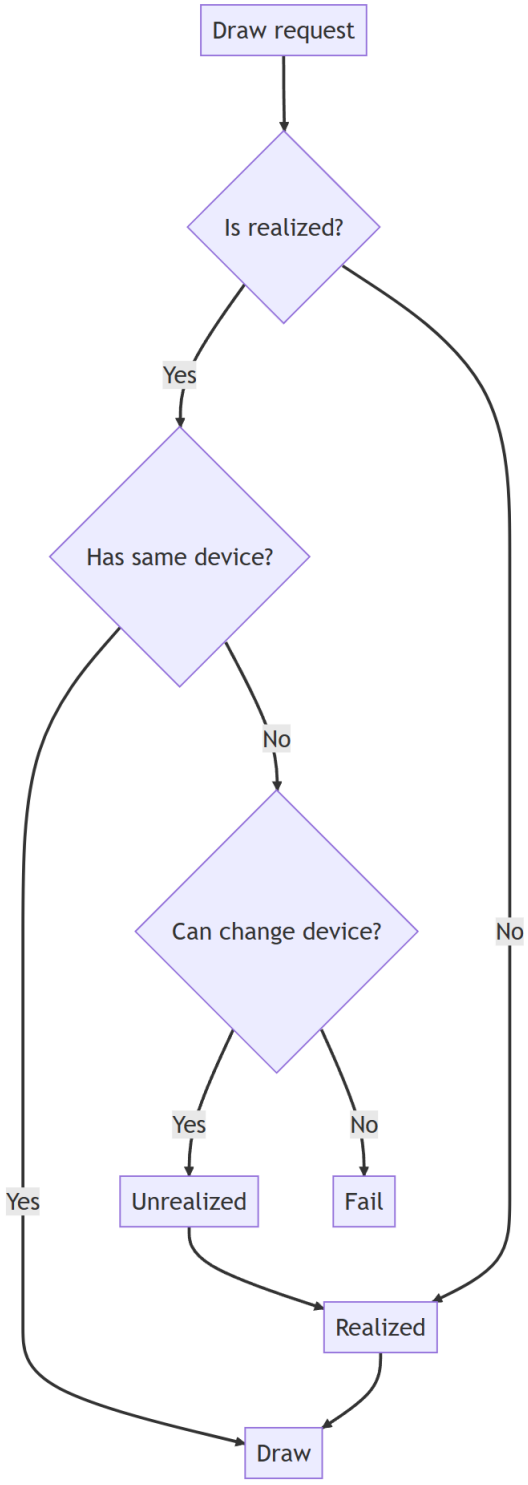


Figure 3.1. Win2D effect lifecycle

Worth noting that Win2D can call `GetDevice` while recursively traversing an effect graph, meaning there might be multiple active calls to `GetD2DImage` in the stack. Because of this, `GetDevice` should not take a blocking lock on the current image, as that could potentially deadlock. Rather, it should use a re-entrant lock in a non-blocking manner, and return an error if it cannot be acquired. This ensures that the same thread recursively calling it will successfully acquire it, whereas concurrent threads doing the same will fail gracefully.

3.1.2 Implementing `GetD2DImage`

`GetD2DImage` is where most of the work takes place. This method is responsible for retrieving the `ID2D1Image` object that Win2D can draw, optionally realizing the current effect if needed. This also includes recursively traversing and realizing the effect graph for all sources, if any, as well as initializing any state that the image might need (eg. constant buffers and other properties, resource textures, etc.).

The exact implementation of this method is highly dependent on the image type and it can vary a lot, but generally speaking for an arbitrary effect you can expect the method to perform the following steps:

- Check whether the call was recursive on the same instance, and fail if so. This is needed to detect cycles in an effect graph (eg. effect A has effect B as source, and effect B has effect A as source).
- Acquire a lock on the image instance to protect against concurrent access.
- Handle the target DPIs according to the input flags
- Validate whether the input device matches the one in use, if any. If it does not match and the current effect supports realization, unrealize the effect.
- Realize the effect on the input device. This can include registering the D2D effect on the `ID2D1Factory1` object retrieved from the input device or device context, if needed. Additionally, all necessary state should be set on the D2D effect instance being created.
- Recursively traverse any sources and bind them to the D2D effect.

With respect to the input flags, there are several possible cases that custom effects should properly handle, to ensure compatibility with all other Win2D effects. Excluding `WIN2D_GET_D2D_IMAGE_FLAGS_NONE`, the flags to handle are the following:

- `WIN2D_GET_D2D_IMAGE_FLAGS_READ_DPI_FROM_DEVICE_CONTEXT`: in this case, ”device” is guaranteed to not be `null`. The effect should check whether the

device context target is an ID2D1CommandList, and if so, add the WIN2D_GET_D2D_IMAGE_FLAGS_ALWAYS_INSERT_DPI_COMPENSATION flag. Otherwise, it should set "targetDpi" (which is also guaranteed to not be null) to the DPIs retrieved from the input context. Then, it should remove WIN2D_GET_D2D_IMAGE_FLAGS_READ_DPI_FROM_DEVICE_CONTEXT from the flags.

- WIN2D_GET_D2D_IMAGE_FLAGS_ALWAYS_INSERT_DPI_COMPENSATION and WIN2D_GET_D2D_IMAGE_FLAGS_NEVER_INSERT_DPI_COMPENSATION: used when setting effect sources (see notes below).

- WIN2D_GET_D2D_IMAGE_FLAGS_MINIMAL_REALIZATION: if set, skips recursively realizing the sources of the effect, and just returns the realized effect with no other changes.

- WIN2D_GET_D2D_IMAGE_FLAGS_ALLOW_NULL_EFFECT_INPUTS: if set, effect sources being realized are allowed to be null, if the user has not set them to an existing source yet.

- WIN2D_GET_D2D_IMAGE_FLAGS_UNREALIZE_ON_FAILURE: if set, and an effect source being set is not valid, the effect should unrealize before failing. That is, if the error occurred while resolving the effect sources after realizing the effect, the effect should unrealize itself before returning the error to the caller.

With respect to the DPI-related flags, these control how effect sources are set. To ensure compatibility with Win2D, effects should automatically add DPI compensation effects to their inputs when needed. They can control whether that is the case like so:

- If WIN2D_GET_D2D_IMAGE_FLAGS_MINIMAL_REALIZATION is set, a DPI compensation effect is needed whenever the "inputDpi" parameter is not 0.

- Otherwise, DPI compensation is needed if "inputDpi" is not 0, WIN2D_GET_D2D_IMAGE_FLAGS_NEVER_INSERT_DPI_COMPENSATION is not set, and either WIN2D_GET_D2D_IMAGE_FLAGS_ALWAYS_INSERT_DPI_COMPENSATION is set, or the input DPI and the target DPI values don't match.

This logic should be applied whenever a source is being realized and bound to an input of the current effect. Note that if a DPI compensation effect is added, that should be the input set to the underlying D2D image. But, if the user tries to retrieve the WinRT wrapper for that source, the effect should take care to detect whether a DPI effect was used, and return a wrapper for the original source object

instead. That is, DPI compensation effects should be transparent to users of the effect.

After all initialization logic is done, the resulting `ID2D1Image` (just like with Win2D objects, a D2D effect is also an image) should be ready to be drawn by Win2D on the target context, which is not yet known by the callee at this time. All this infrastructure is needed to ensure effects can correctly be realized and drawn on arbitrary drawing contexts at any time.

3.1.3 Implementing `GetBounds`

Finally, the last missing component to fully implement a custom `ID2D1Image` object is to support the two `GetBounds` overloads. To make this easy, Win2D exposes a C export which can be used to leverage the existing logic for this from Win2D on any custom image. The export is as follows:

```
HRESULT GetBoundsForICanvasImageInterop(
    ICanvasResourceCreator* resourceCreator,
    ICanvasImageInterop* image,
    Numerics::Matrix3x2 const* transform,
    Rect* rect);
```

Custom images can invoke this API and pass themselves as the ”image” parameter, and then simply return the result to their callers. The ”transform” parameter is optional and it can be `null`, if no transform is available.

3.2 Optimizing `ID2D1DeviceContext` accesses

The ”deviceContext” parameter in `ICanvasImageInterop::GetD2DImage` can sometimes be `null`, if a context is not immediately available before the invocation. This is done on purpose, so that a context is only created lazily when it’s actually needed. That is, if a context is available, Win2D will pass it to the `GetD2DImage` invocation, otherwise it will let callees retrieve one on their own if necessary.

Creating a device context is relatively expensive, so to make retrieving one faster Win2D exposes APIs to access its internal device context pool. This allows custom effects to rent and return device contexts associated with a given canvas device in an efficient manner. The device context lease APIs are defined as follows:

```
[uuid(A0928F38-F7D5-44DD-A5C9-E23D94734BBB)]
class ID2D1DeviceContextLease : IUnknown
{
    HRESULT GetD2DDeviceContext(ID2D1DeviceContext** deviceContext);
};

[uuid(454A82A1-F024-40DB-BD5B-8F527FD58AD0)]
class ID2D1DeviceContextPool : IUnknown
{
    HRESULT GetDeviceContextLease(ID2D1DeviceContextLease** lease);
};
```

The `ID2D1DeviceContextPool` interface is implemented by `CanvasDevice`, which is the Win2D type implementing the `ICanvasDevice` interface. To use the pool, use `QueryInterface` on the device interface to obtain an `ID2D1DeviceContextPool` reference, and then call `ID2D1DeviceContextPool::GetDeviceContextLease` to obtain an `ID2D1DeviceContextLease` object to access the device context. Once that's no longer needed, release the lease. Make sure to not touch the device context after the lease has been released, as it might be used concurrently by other threads[4].

3.3 Supporting WinRT wrapper lookups

As seen in the Win2D interop docs [53], the Win2D public header also exposes a `GetOrCreate` method (accessible from the `ICanvasFactoryNative` activation factory for `CanvasDevice`, or through the `GetOrCreate` C++/CX helpers defined in the same header). This allows retrieving a WinRT wrapper from a given native resource. For instance, it lets you retrieve or create a `CanvasDevice` instance from an `ID2D1Device1` object, a `CanvasBitmap` from an `ID2D1Bitmap`, etc.

This method also works for all built-in Win2D effects: retrieving the native resource for a given effect and then using that to retrieve the corresponding Win2D wrapper will correctly return the owning Win2D effect for it. In order for custom effects to also benefit from the same mapping system, Win2D exposes several APIs in the interop interface for the activation factory for `CanvasDevice`, which is the `ICanvasFactoryNative` type, as well as an additional effect factory interface, `ICanvasEffectFactoryNative`:

[4]This is also an example of an API that was designed while dogfooding early previews. The idea came from noticing how inconvenient (and inefficient) it was for ComputeSharp to obtain `ID2D1DeviceContext` instances when necessary (almost exactly having to mirror what Win2D was already doing, or just giving up on the performance improvements of pooling contexts). To avoid this, we proposed and implement this new pair of APIs, which allow external effects to simply access the same internal cache of device contexts that Win2D handles. This greatly simplifies the code related to this in external effects, while also improving performance.

```
[uuid(29BA1A1F-1CFE-44C3-984D-426D61B51427)]
class ICanvasEffectFactoryNative : IUnknown
{
    HRESULT CreateWrapper(
        ICanvasDevice* device,
        ID2D1Effect* resource,
        float dpi,
        IInspectable** wrapper);
};

[uuid(695C440D-04B3-4EDD-BFD9-63E51E9F7202)]
class ICanvasFactoryNative : IInspectable
{
    HRESULT GetOrCreate(
        ICanvasDevice* device,
        IUnknown* resource,
        float dpi,
        IInspectable** wrapper);

    HRESULT RegisterWrapper(IUnknown* resource, IInspectable* wrapper);

    HRESULT UnregisterWrapper(IUnknown* resource);

    HRESULT RegisterEffectFactory(
        REFIID effectId,
        ICanvasEffectFactoryNative* factory);

    HRESULT UnregisterEffectFactory(REFIID effectId);
};
```

There are several APIs to consider here, as they're needed to support all the various scenarios where Win2D effects can be used, as well as how developers could do interop with the D2D layer and then try to resolve wrappers for them. Let's go over each of these APIs.

The `RegisterWrapper` and `UnregisterWrapper` methods are meant to be invoked by custom effects to add themselves into the internal Win2D cache:

- `RegisterWrapper`: registers a native resource and its owning WinRT wrapper. The "wrapper" parameter is required to also implemement `IWeakReferenceSource` [54], so that it can be cached correctly without causing reference cycles which would lead to memory leaks. The method returns `S_OK` if the native resource could be added to the cache, `S_FALSE` if there was already a registered wrapper for "resource", and an error code if an error occurs.

- `UnregisterWrapper`: unregisters a native resource and its wrapper. Returns

`S_OK` if the resource could be removed, `S_FALSE` if ”resource” was not already registered, and an error code if another error occurrs.

Custom effects should call `RegisterWrapper` and `UnregisterWrapper` whenever they are realized and unrealized, ie. when a new native resource is created and associated with them. Custom effects that do not support realization (eg. those having a fixed associated device) can call `RegisterWrapper` and `UnregisterWrapper` when they are created and destroyed. Custom effects should make sure to correctly unregister themselves from all possible code paths that would cause the wrapper to become invalid (eg. including when the object is finalized, in case it's implemented in a managed language).

The `RegisterEffectFactory` and `UnregisterEffectFactory` methods are also meant to be used by custom effects, so that they can also register a callback to create a new wrapper in case a developer tries to resolve one for an ”orphaned” D2D resource:

- `RegisterEffectFactory`: register a callback that takes in input the same parameters that a developer passed to `GetOrCreate`, and creates a new inspectable wrapper for the input effect. The effect id is used as key, so that each custom effect can register a factory for it when it's first loaded. Of course, this should only be done once per effect type, and not every time the effect is realized. The ”device”, ”resource” and ”wrapper” parameters are checked by Win2D before invoking any registered callback, so they are guaranteed to not be `null` when `CreateWrapper` is invoked. The ”dpi” is considered optional, and can be ignored in case the effect type doesn't have a specific use for it. Note that when a new wrapper is created from a registered factory, that factory should also make sure that the new wrapper is registered in the cache (Win2D will not automatically add wrappers produced by external factories to the cache).

- `UnregisterEffectFactory`: removes a previously register callback. For instance, this could be used if an effect wrapper is implemented in a managed assembly which is being unloaded.

Also worth noting that as mentioned above, `ICanvasFactoryNative` is implemented by the activation factory for `CanvasDevice`, which you can retrieve by either manually calling `RoGetActivationFactory`, or using helper APIs from the language extensions you're using (eg. `winrt::get_activation_factory` [55] in C++/WinRT). For more info, see WinRT type system [3] for more information on how this works.

For a practical example of where this mapping comes into play, consider how built-in Win2D effects work. If they are not realized, all state (eg. properties, sources, etc.) is stored in an internal cache in each effect instance. When they are realized, all state is transferred to the native resource (eg. properties are set on the D2D effect, all sources are resolved and mapped to effect inputs, etc.), and as long as the effect is realized it will act as the authority on the state of the wrapper. That is, if the value of any property is fetched from the wrapper, it will retrieve the updated value for it from the native D2D resource associated with it.

This ensures that if any changes are made directly to the D2D resource, those will be visible on the outer wrapper as well, and the two will never be "out of sync". When the effect is unrealized, all state is transferred back from the native resource to the wrapper state, before the resource is released. It will be kept and updated there until the next time the effect is realized. Now, consider this sequence of events:

- You have some Win2D effect (either built-in, or custom).
- You get the `ID2D1Image` from it (which is an `ID2D1Effect`).
- You create an instance of a custom effect.
- You also get the `ID2D1Image` from that.
- You manually set this image as input for the previous effect (via `ID2D1Effec⌋ t::SetInput` [56]).
- You then ask that first effect for the WinRT wrapper for that input.

Since the effect is realized (it was realized when the native resource was requested), it will use the native resource as the source of truth. As such, it will get the `ID2D1Image` corresponding to the requested source, and try to retrieve the WinRT wrapper for it. If the effect this input was retrieved from has correctly added its own pair of native resource and WinRT wrapper to Win2D's cache, the wrapper will be resolved and returned to callers. If not, that property access will fail, as Win2D can't resolve WinRT wrappers for effects it does not own, as it doesn't know how to instantiate them.

This is where `RegisterWrapper` and `UnregisterWrapper` help, as they allow custom effects to seamlessly participate in Win2D's wrapper resolution logic, so that the correct wrapper can always be retrieved for any effect source, regardless of whether it was set from WinRT APIs, or directly from the underlying D2D layer. To explain how the effect factories also come into play, consider this scenario:

- A user creates an instance of a custom wrapper and realizes it

- They then gets a reference to the underlying D2D effect and keeps it.

- Then, the effect is realized on a different device. The effect will unrealize and re-realize, and in doing so it will create a new D2D effect. The previous D2D effect no longer as an associated inspectable wrapper at this point.

- The user then calls `GetOrCreate` on the first D2D effect.

Without a callback, Win2D would just fail to resolve a wrapper, as there's no registered wrapper for it[5]. If a factory is registered instead, a new wrapper for that D2D effect can be created and returned, so the scenario just keeps working seamlessly for the user[6].

3.4 Implementing `ICanvasEffect`

The Win2D `ICanvasEffect` interface extends `ICanvasImage`, so all the previous points apply to custom effects as well. The only difference is the fact that `ICanvasEffect` also implements additional methods specific to effects, such as invalidating a source rectangle, getting the required rectangles and so on.

To support this, Win2D exposes C exports that authors of custom effects can use, so that they won't have to reimplement all this extra logic from scratch. This works in the same way as the C export for `GetBounds`. Here are the available exports for effects:

[5]While working on adding support for external factories, we also discovered a long standing bug in Win2D around how the interop APIs allowed users to create Win2D wrappers around D2D objects they owned[57], which originated by the fact that Win2D was not validating that the D2D factory for the D2D device in use and the D2D resource being wrapped were a match (they have to be, or the two D2D objects would not be able to work together at all). This was also fixed as part of the new major update for Win2D adding the external effects functionality.

[6]Along with all other nuances mentioned in the rest of this chapter, this is a good example of the complexity involved in creating custom Win2D effects from scratch. It is truly a very tricky and delicate implementation, which must consider several different aspects of how Win2D and D2D are intertwined and working together in order to correctly support all the expected scenarios. This is why the new `ICanvasImageInterop` infrastructure is only meant for advanced users (and it's primarily only meant for ComputeSharp), and why the official Microsoft documentation on Win2D also recommends using ComputeSharp to author new D2D pixel shader effects to wire up through Win2D, rather than trying to implement a Win2D effect on its own. The same goes for C++ users, which might also want to consider using the same ComputeSharp APIs through a C# WinRT component that can be natively compiled (using either the .NET Native or NativeAOT runtimes for .NET) and then directly referenced (via the resulting .dll and .winmd files).

```
HRESULT InvalidateSourceRectangleForICanvasImageInterop(
    ICanvasResourceCreatorWithDpi* resourceCreator,
    ICanvasImageInterop* image,
    uint32_t sourceIndex,
    Rect const* invalidRectangle);

HRESULT GetInvalidRectanglesForICanvasImageInterop(
    ICanvasResourceCreatorWithDpi* resourceCreator,
    ICanvasImageInterop* image,
    uint32_t* valueCount,
    Rect** valueElements);

HRESULT GetRequiredSourceRectanglesForICanvasImageInterop(
    ICanvasResourceCreatorWithDpi* resourceCreator,
    ICanvasImageInterop* image,
    Rect const* outputRectangle,
    uint32_t sourceEffectCount,
    ICanvasEffect* const* sourceEffects,
    uint32_t sourceIndexCount,
    uint32_t const* sourceIndices,
    uint32_t sourceBoundsCount,
    Rect const* sourceBounds,
    uint32_t valueCount,
    Rect* valueElements);
```

Let's go over how they can be used:

- `InvalidateSourceRectangleForICanvasImageInterop` is meant to support `InvalidateSourceRectangle`. Simply marshal the input parameters and invoke it directly, and it'll take care of all the necessary work. Note that the "image" parameter is the current effect instance being implemented.
- `GetInvalidRectanglesForICanvasImageInterop` supports `GetInvalidRectangles`. This also requires no special consideration, other than needing to dispose the returned COM array once it's no longer needed.
- `GetRequiredSourceRectanglesForICanvasImageInterop` is a shared method that can support both `GetRequiredSourceRectangle` and `GetRequiredSourceRectangles`. That is, it takes a pointer to an existing array of values to populate, so callers can either pass a pointer to a single value (which can also be on the stack, to avoid one allocation), or to an array of values. The implementation is the same in both cases, so a single C export is enough to power both of them.

With this interface also implemented, custom effects for Win2D can reach perfect feature parity with all of Win2D's built-in effects, and they can seamlessly be used

in all scenarios where Win2D effects are supported (with the only exception being Composition interop, which is not supported due to OS limitations that are outside the scope of Win2D).

3.5 Drawing shaders via ComputeSharp

As we could see, implementing a custom effect from scratch is extremely complicated and requires carefully integrating functionality from several different interfaces. To help with this, the officially recommended approach for C# developers is to instead leverage ComputeSharp, and its Win2D-related APIs to easily build custom effects either by combining existing effects together, or by authoring entirely new effects via D2D pixel shaders also written in C#. These effects can then easily be integrated with Win2D due to their built-in support for interop with the library. This approach is the same one also used by the Microsoft Store to power several custom graphics effects, as we'll see in more detail below.

There are two main components in ComputeSharp to interop with Win2D[7]:

- `PixelShaderEffect<T>`: a Win2D effect that is powered by a D2D1 pixel shader. The shader itself is written in C# using the APIs provided by ComputeSharp. This class also provides properties to set effect sources, constant values, and more.

- `CanvasEffect`: a base class for custom Win2D effects that wraps an arbitrary effect graph. It can be used to "package" complex effects into an easy to use object that can be reused in several parts of an application.

Here is an example of a custom pixel shader (ported from shadertoy [58]), used with `PixelShaderEffect<T>` and then draw onto a Win2D `CanvasControl` (note that `PixelShaderEffect<T>` implements `ICanvasImage`, so it can directly be drawn with Win2D):

[7]This ComputeSharp package is a .NET library and not a WinRT component. This is due to some of the feature requirements and the shape of the public APIs being exposed (for instance, WinRT components cannot expose public abstract types). But, as also mentioned above, it is still possible to simply wrap code using these APIs in a small WinRT component, which can then be consumed by native applications as well, if needed.

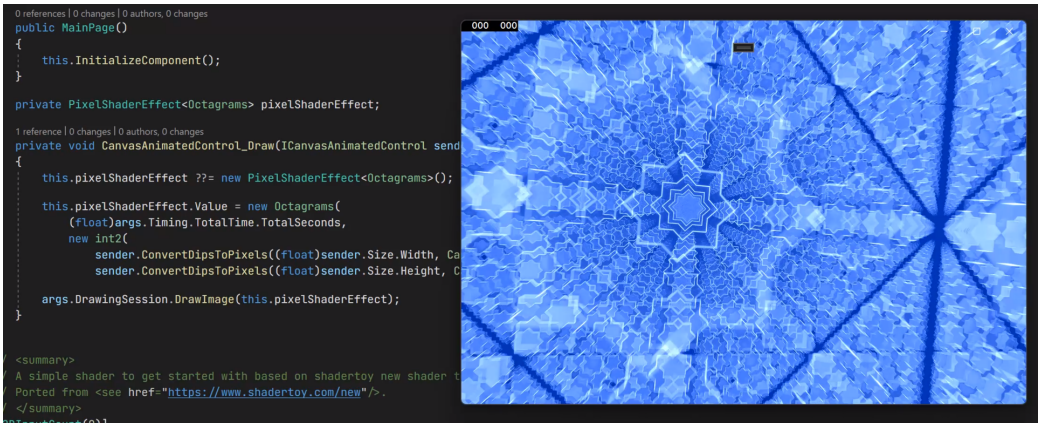


Figure 3.2. Octagram D2D pixel shader with ComputeSharp

In the figure above, we can see how in just two lines of code we can create an effect and draw it via Win2D. ComputeSharp takes care of all the work necessary to compile the shader, register it, and manage the complex lifetime of a Win2D-compatible effect. Next, we will look at a step by step guide on how to create a custom Win2D effect that also uses a custom D2D1 pixel shader. We'll go over how to author a shader with ComputeSharp and setup its properties, and then how to create a custom effect graph packaged into a `CanvasEffect` type that can easily be reused in your application.

3.6 ComputeSharp wrappers for Win2D

Before diving into the end to end sample, it is necessary to do an overview on the public APIs that ComputeSharp.D2D1.Uwp exposes (same as ComputeSharp.D2D1.WinUI), to make it clearer how all the various moving parts are connected. As we mentioned, these two packages are built on top of the base interop logic functionality provided by ComputeSharp.D2D1, and they also bring in Win2D as a transitive dependency, and leverage the new `ICanvasImageInterop` interface to implement Win2D-compatible effects that can easily interoperate with that library.

3.6.1 `PixelShaderEffect<T>` API surface

Let's start from the API surface of the base effect type, ie. `PixelShaderEffect<T>`. This type is an `ICanvasImage` implementation (also implementing `ICanvasImageI` `nterop`), which will execute a given shader type in its transform graph. Essentially, it's a Win2D wrapper for ComputeSharp.D2D1's `ID2D1Effect` implementation. Here is its full API surface:

```
public sealed class PixelShaderEffect<T> : ICanvasEffect
    where T : unmanaged, ID2D1PixelShader
{
    public T ConstantBuffer { get; set; }

    [DisallowNull]
    public D2D1TransformMapper<T>? TransformMapper  { get; set; }

    public SourceCollection Sources { get; }

    public ResourceTextureManagerCollection ResourceTextureManagers { get; }

    public sealed class SourceCollection :
        IList<IGraphicsEffectSource?>,
        IReadOnlyList<IGraphicsEffectSource?>,
        IList
    {
    }

    public sealed class ResourceTextureManagerCollection :
        IList<D2D1ResourceTextureManager?>,
        IReadOnlyList<D2D1ResourceTextureManager?>,
        IList
    {
    }
}
```

The API surface is actually larger than this, but all inherited members from `ICanvasImage` and from the various interfaces that `SourceCollection` and `ResourceTextureManagerCollection` implement have been omitted for brevity. The key components for this wrapper are the fact that it internally manages all the necessary state to correctly implement a Win2D effect (with the realization and Win2D wrapper caching logic mentioned above as well), and provides high level APIs to easily set and retrieve the constant buffer, the transform mapper, as well as any sources and D2D resource textures that might be used. Of course, since the same `PixelShaderEffect<T>` class is used for all shader types, some validation (eg. the index of sources being set) is only done at runtime, which allows this class to be particularly flexible and usable with any pixel shader.

For instance, here's how this class can be used on its own to draw a given shader (this snippet assumes that the code is within a given `Draw` handler for the `CanvasControl.Draw` event):

```
PixelShaderEffect<MyShader> effect = new()
{
    ConstantBuffer = new MyShader(...),
    TransformMapper = D2D1TransformMapperFactory<MyShader>.Inflate(5),
    Sources =
    {
        [0] = texture0,
        [1] = texture1,
    },
    ResourceTextureManagers =
    {
        [0] = resourceTexture0
    }
};

args.DrawingSession.DrawImage(effect);
```

In this example, we can see how the entire drawing process has been abstracted through Win2D, with users only having to deal with ComputeSharp shaders and APIs, without needing to worry at all about any of the underlying D2D implemementation details[8].

3.6.2 CanvasEffect API surface

At an even higher level level, we find the `CanvasEffect` class, which acts as a base type for all ”packaged” Win2D effects. The meaning of ”packaged” in this context is that these effects can include arbitrary effect graphs as their implementation, while exposing a simple API surface for consumers. That is, library authors can use this base class to implement all sorts of custom effects, and then only expose the actual APIs that they want users of those effects to see and set. This is the API surface of the base class (once again excluding methods that are inherited by implemented interfaces, for brevity):

[8]This minimal sample also shows some of the new features exposed by `PixelShaderEffect<T>` that were not possible in the built-in `PixelShaderEffect` type bundled with Win2D. Specifically, we can see how the constant buffer is strongly typed and directly loaded by assigning a new instance of the shader type (rather as an untyped, unstructured map of boxed values tied to named keys, like Win2D does). Additionally, we also have the ability to plug in a D2D transform mapper (which also allows accessing the constant buffer and mutating it from one of its callbacks, which can be useful in very advanced scenarios). Lastly, `PixelShaderEffect<T>` also supports D2D resource textures, which can be extremely useful when shaders need additional constant data that can be read or sampled (such as a convolutional kernel, in a gaussian blur effect). The added benefit of using resource textures is that they don’t factor into the input/output mapping logic (as they are not standard shader inputs), which can greatly simplify authoring effects, as it removes the need to worry about these additional inputs when calculating the tiling subdivision of a drawing operation.

```
public abstract class CanvasEffect : ICanvasImage
{
    protected abstract void BuildEffectGraph(EffectGraph effectGraph);

    protected abstract void ConfigureEffectGraph(EffectGraph effectGraph);

    protected void InvalidateEffectGraph(InvalidationType invalidationType);

    protected void SetAndInvalidateEffectGraph<T>(
        [NotNullIfNotNull(nameof(value))] ref T storage,
        T value,
        InvalidationType invalidationType = InvalidationType.Update);

    protected virtual void Dispose(bool disposing);

    protected enum InvalidationType : byte
    {
        Update,
        Creation
    }
}
```

For the purposes of authoring effects, those `protected` methods are the entry points that deriving types can override to inject their own functionality, such as building their own effect graph and setting the effect state before drawing. Let's look at what the role of each of those methods is in particular:

- `BuildEffectGraph`: builds the effect graph for the current `CanvasEffect` instance, and configures all effect nodes, as well as the output node for the graph. That `ICanvasImage` instance will then be passed to Win2D to perform the actual drawing, when needed. This method is called once before the current effect is drawn, and the resulting image is automatically cached and reused. It will remain in use until `InvalidateEffectGraph` is called with `InvalidationType.Creation`. If the effect is invalidated with `InvalidationType.Update`, only `ConfigureEffectGraph` will be called. As such, derived types should save any effect graph nodes that might need updates into instance fields, for later use. For instance, consider a `FrostedGlassEffect` type deriving from `CanvasEffect`, with these effects and nodes in sequence: source, blur, tint, noise, output. In this example, the blur is a `GaussianBlur」Effect` instance, the tint is a `TintEffect`. and the noise is some other custom effect like the one we described earlier. In this case, the output `ICanvasImage` node configured by will be the noise effect, as that's the output node for the effect graph. At the same time, both the blur and the tint nodes will also need

to be registered into the effect graph, so that `ConfigureEffectGraph` will be able to set properties on them when needed. For instance, the effect might expose a property to control the blur amount, as well as the tint color and opacity. Generally speaking, an implementation of `BuildEffectGraph` will consist of these steps: creating an instance of all necessary nodes in the effect graph, connecting the effect nodes as needed to build the connected graph, and registering all effects as effect nodes in the graph, including the output node. This method should never be called directly. It is automatically invoked when an effect graph is needed.

- `ConfigureEffectGraph`: configures the current effect graph whenever it is invalidated. For instance, if a node is a `PixelShaderEffect<T>` type, it can set its constant buffer from a property exposed by the effect instance. This method is guaranteed to be called after `BuildEffectGraph` has been invoked already. As such, any nodes that are registered by `BuildEffectGraph` can be assumed to never be `null` when this method runs. This method should never be called directly. It is used internally to configure the current effect graph.

- `InvalidateEffectGraph`: invalidates the last cached output node retrieved when drawing the effect. This method is used to signal when the effect graph should be updated, and it can indicate that either the entire graph should be created again or whether it simply needs to refresh its internal state (by calling `ConfigureEffectGraph`). If `InvalidateEffectGraph` is called with `InvalidationType.Update`, the effect graph will only be refreshed the next time the image is actually requested. That is, repeated requests for updates do not result in unnecessarily calls to `ConfigureEffectGraph`. This allows effect authors to further optimize the drawing speed of custom effects, by skipping unnecessary work when some properties are changed that are known not to affect the actual layout of the effect graph in use.

- `SetAndInvalidateEffectGraph`: updates the backing storage for an effect property and checks if it has changed. If so, it invalidates the effect graph as well with the requested invalidation type. This is mostly a convenience helper for effect properties, that simply sets a property value and calls `InvalidateEffectGraph` if needed.

3.6.3 CanvasEffect graph APIs

The `CanvasEffect` type also defines several `protected` types (some of which were also visible in the API surface just shown above). These are used by derived types

to access the internal effect graph and register and retrieve nodes. Let's go through the API surface for these types as well:

```
protected readonly ref struct EffectGraph
{
    public ICanvasImage GetNode(IEffectNode effectNode);

    public T GetNode<T>(IEffectNode<T> effectNode)
        where T : class, ICanvasImage;

    public void RegisterNode(ICanvasImage canvasImage);

    public void RegisterNode<T>(EffectNode<T> effectNode, T canvasImage)
        where T : class, ICanvasImage;

    public void RegisterOutputNode(ICanvasImage canvasImage);

    public void RegisterOutputNode<T>(EffectNode<T> effectNode, T canvasImage)
        where T : class, ICanvasImage;

    public void SetOutputNode(IEffectNode effectNode);
}

protected interface IEffectNode
{
}

protected interface IEffectNode<out T> : IEffectNode
    where T : ICanvasImage
{
}

protected sealed class EffectNode<T> : IEffectNode<T>
    where T : class, ICanvasImage
{
}
```

As we can see, these APIs are split across two groups:

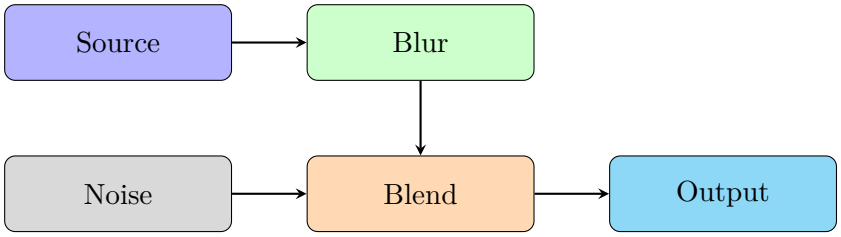
- `EffectGraph` is a type that provides direct access to the internal effect graph for the current effect instance. It exposes several `GetNode`, `RegisterNode`, `RegisterOutputNode` and `SetOutputNode` methods that derived types can use both to configure an effect graph, as well as to retrieve registered nodes later on, to change their properties or to configure them as output nodes in case an effect graph layout was needed.

- `EffectNode<T>` and its base interfaces are used to provide a type safe way to declaring effect nodes that can then be used for lookup. That is, instead of using some other value (eg. placeholder objects, or strings, or something else) for this, derived types can instead declare `static readonly` fields of some `EffectNode<T>` type, which allows them to then look those nodes up and retrieve the instantiated effects, without having to manually do type casts, which might be error prone. We'll also see how these types can be used in practice in the full end to end demo below.

This is the full API surface that ComputeSharp.D2D1.Uwp and ComputeSharp.D2D1.WinUI expose. With this, developers can now easily leverage the full power of ComputeSharp.D2D1 into their UWP XAML and WinUI 3 applications, and plug those effects through Win2D without having to deal with any of the complexity that the underlying implementation requires.

3.7 End to end ComputeSharp sample

For this demo, we want to create a simple frosted glass effect. This will have the following effect graph:



We'll also want to expose properties to control the blur and noise amount (the noise itself will be procedurally generated via a custom shader, as we'll see below). The final effect will contain a "packaged" version of this effect graph and be easy to use by just creating an instance, setting those properties, connecting a source image, and then drawing it.

3.7.1 Designing a D2D pixel shader

For the noise on top of the effect, we can use a simple D2D1 pixel shader. The shader will compute a random value based on its coordinates (which will act as a "seed" for the random number), and then it will use that noise value to compute the RGB amount for that pixel. We can then blend this noise on top of the resulting image. To write the shader with ComputeSharp, we just need to define a `partial struct`

type implementing the `ID2D1PixelShader` interface, and then write our logic in the `Execute` method. For this noise shader, we can write something like this:

```
using ComputeSharp;
using ComputeSharp.D2D1;

[D2DInputCount(0)]
[D2DRequiresScenePosition]
[D2DShaderProfile(D2D1ShaderProfile.PixelShader40)]
[AutoConstructor]
public readonly partial struct NoiseShader : ID2D1PixelShader
{
    private readonly float _amount;

    public float4 Execute()
    {
        // Get the current pixel coordinate (in pixels)
        int2 position = (int2)D2D.GetScenePosition().XY;

        // Compute a random value in the [0, 1] range for each target pixel. This line just
        // calculates a hash from the current position and maps it into the [0, 1] range.
        // This effectively provides a "random looking" value for each pixel.
        float hash = Hlsl.Frac(Hlsl.Sin(Hlsl.Dot(position, new float2(41, 289))) * 45758.5453f);

        // Map the random value in the [0, amount] range, to control the strength of the noise
        float alpha = Hlsl.Lerp(0, _amount, hash);

        // Return a white pixel with the random value modulating the opacity
        return new(1, 1, 1, alpha);
    }
}
```

Let's go over this shader in detail:

- The shader has no inputs, it just produces an infinite image with random grayscale noise.
- The shader requires access to the current pixel coordinate.
- The shader is precompiled at build time (using the `PixelShader40` profile, which is guaranteed to be available on any GPU where the application could be running).
- The `[AutoConstructor]` attribute is part of ComputeSharp, and it will just generate a constructor automatically for us to set that `_amount` field[9].

[9]This is planned to be removed in ComputeSharp 3.0, and replaced by built-in support for

This shader will generate our custom noise texture whenever needed. Next, we need to create our packaged effect with the effect graph connecting all our effects together. For our easy to use, packaged effect, we can use the `CanvasEffect` type from ComputeSharp. This type provides a straightforward way to setup all the necessary logic to create an effect graph and update it via public properties that users of the effect can interact with. There are two main methods we'll need to implement:

- `BuildEffectGraph`: this method is responsible for building the effect graph that we want to draw. That is, it needs to create all effects we need, and register the output node for the graph. For effects that can be updated at a later time, the registration is done with an associated `EffectNode<T>` value, which acts as lookup key to retrieve the effects from the graph when needed.

- `ConfigureEffectGraph`: this method refreshes the effect graph by applying the settings that the user has configured. This method is automatically invoked when needed, right before drawing the effect, and only if at least one effect property has been modified since the last time the effect was used. Changes applied at this stage can include both updating the constant buffer of any underlying `PixelShaderEffect<T>` node, modifying any properties of arbitrary nodes in the effect graph, as well as selecting a different output node for the effect graph itself (even with no other changes in any effect property).

3.7.2 Defining the high level effect

Our custom effect can be defined by combining the various building blocks that are necessary. We can conceptually subdivide the effect implementation into exposed properties, effect graph building logic and effect graph configuration logic, through the `BuildEffectGraph` and `ConfigureEffectGraph` effects we just mentioned. This is what the internal state and public properties look like for this specific effect:

primary constructors in C# 12[59]. This is one of the upcoming features of ComputeSharp that requires .NET 8: because captured primary constructors result in compiler generated fields with an unspeakable name (eg. °<name>P") and that are not publicly accessible, the source generator will handle this by emitting a series of field accessors using the new `[UnsafeAccessor]` API included in .NET 8[60]. It is worth mentioning that due to special runtime support, these unsafe accessors do not incur any runtime overhead, as the entire calls are completely elided at compile time. As such, the new version of the ComputeSharp source generator is always using them in all cases, which both simplifies the generator logic, as well as fully takes care of both primary constructor fields, as well as fields that are not publicly accessible (ie. shader types are free to only have private fields, and the constant buffer marshalling will still work just fine even in that scenario).

```
using ComputeSharp.D2D1.WinUI;
using Microsoft.Graphics.Canvas;
using Microsoft.Graphics.Canvas.Effects;

public sealed partial class FrostedGlassEffect : CanvasEffect
{
    private static readonly EffectNode<GaussianBlurEffect> BlurNode = new();
    private static readonly EffectNode<PixelShaderEffect<NoiseShader>> NoiseNode = new();

    private ICanvasImage? _source;
    private double _blurAmount;
    private double _noiseAmount;

    public ICanvasImage? Source
    {
        get => _source;
        set => SetAndInvalidateEffectGraph(ref _source, value);
    }

    public double BlurAmount
    {
        get => _blurAmount;
        set => SetAndInvalidateEffectGraph(ref _blurAmount, value);
    }

    public double NoiseAmount
    {
        get => _noiseAmount;
        set => SetAndInvalidateEffectGraph(ref _noiseAmount, value);
    }
}
```

As we can see, we expose properties[10] to allow consumers of this packaged effect to configure the source image, the blur amount and the noise amount. We also have to markers to perform lookup operations on the blur and noise effect in the internal effect graph, so we can update their values before drawing. This is how we can then configure the effect graph during initialization (omitting the `using` directives from now on, for brevity):

[10] As part of the future work, as it needs support for `partial` properties in C#, ComputeSharp is planned to also bundle a source generator to make it easier creating packaged effects. It will work by having the runtime library include an attribute (such as `[CanvasEffectProperty]`) which can be applied on auto-properties defining properties of an effect, and a source generator which will implement the properties by emitting the same calls to `SetAndInvalidateEffectGraph` (with the requested invalidation mode as well). This will remove the need to manually define backing fields for the storage and both property accessors, and it will greatly reduce the verbosity currently required to author such packaged effects.

```
partial class FrostedGlassEffect
{
    protected override void BuildEffectGraph(EffectGraph effectGraph)
    {
        // Create the effect graph
        GaussianBlurEffect gaussianBlurEffect = new();
        BlendEffect blendEffect = new() { Mode = BlendEffectMode.Overlay };
        PixelShaderEffect<NoiseShader> noiseEffect = new();
        PremultiplyEffect premultiplyEffect = new();

        // Connect the effect graph
        premultiplyEffect.Source = noiseEffect;
        blendEffect.Background = gaussianBlurEffect;
        blendEffect.Foreground = premultiplyEffect;

        // Register all effects. For those that need to be referenced later (ie. the ones with
        // properties that can change), we use a node as a key, so we can perform lookup on
        // them later. For others, we register them anonymously. This allows the effect
        // to autommatically and correctly handle disposal for all effects in the graph.
        effectGraph.RegisterNode(BlurNode, gaussianBlurEffect);
        effectGraph.RegisterNode(NoiseNode, noiseEffect);
        effectGraph.RegisterNode(premultiplyEffect);
        effectGraph.RegisterOutputNode(blendEffect);
    }
}
```

Finally, here's the logic to update our internal state following property changes:

```
partial class FrostedGlassEffect
{
    protected override void ConfigureEffectGraph(EffectGraph effectGraph)
    {
        // Set the effect source
        effectGraph.GetNode(BlurNode).Source = Source;

        // Configure the blur amount
        effectGraph.GetNode(BlurNode).BlurAmount = (float)BlurAmount;

        // Set the constant buffer of the shader
        effectGraph.GetNode(NoiseNode).ConstantBuffer = new NoiseShader((float)NoiseAmount);
    }
}
```

To recap the general structure we talked abouve, we can see there are four sections in this class:

- First, we have fields to track all mutable state, such as the effects that can be

updated as well as the backing fields for all the effect properties that we want to expose to users of the effect.

- Next, we have properties to configure the effect. The setter of each property uses the `SetAndInvalidateEffectGraph` method exposed by `CanvasEffect`, which will automatically invalidate the effect if the value being set is different than the current one. This ensures the effect is only configured again when really necessary.

- Lastly, we have the `BuildEffectGraph` and `ConfigureEffectGraph` methods we mentioned above.

As a small note, the `PremultiplyEffect` node after the noise effect is very important. This is because Win2D effects assume that the output is premultiplied, whereas pixel shaders generally work with unpremultiplied pixels. As such, remember to manually insert premultiply/unpremultiply nodes before and after custom shaders, to ensure colors are correctly preserved.

3.7.3 Drawing the effect

And with this, our custom frosted glass effect is ready. We can easily draw it as follows:

```
private void CanvasControl_Draw(CanvasControl sender, CanvasDrawEventArgs args)
{
    FrostedGlassEffect effect = new()
    {
        Source = _canvasBitmap,
        BlurAmount = 12,
        NoiseAmount = 0.1
    };

    args.DrawingSession.DrawImage(effect);
}
```

In this example, we're drawing the effect from the `Draw` handler of a `CanvasControl`, using a `CanvasBitmap` which we previously loaded as source. This is the input image we'll use to test the effect:

Figure 3.3. Original mountains image

And here is the result:

Figure 3.4. Processed mountains image

Chapter 4

Microsoft Store integration

As mentioned above, the Microsoft Store is also using ComputeSharp to implement custom effects that can then be plugged into Win2D. There are several reasons why we decided to use this approach. For one, leveraging Win2D allows effects to easily be chained together and be hardware accelerated, which results in very good performance even on lower end devices. This is further improved by the fact that we're manually controlling when effects are drawn to the screen and only doing that once when the control is loaded (and again when needed to handle special cases like handling device lost events [61]). This means that each effect is just rendered offline to a D2D bitmap, which is then displayed to the user without requiring any additional processing until the contents of the image change. Using this approach, as opposed to keeping a live Composition brush in the visual tree, allowed us to both achieve a much smoother experience as well as being able to create much more advanced and unique effects than what would have been possible via Composition only.

Figure 4.1. The Microsoft Store

As of today, the Microsoft Store is using two fully custom Win2D effects, which also include two different D2D pixel shaders, written entirely in C# via ComputeSharp. They're used in several core UI modules in the application, making it so that a significant part of the Microsoft Store UI today is effectively rendered through ComputeSharp as well. We'll over both of those effects in more details in the sections below, to show how the shaders are implemented and how both effects are structured and used within the Microsoft Store.

4.1 Case study: app cards

The first custom effect we introduced is a "background blur" effect, which receives an input image, aligns it to the center of the output area, applies a relatively strong gaussian blur effect to it and finally overlays a theme-aware procedural noise texture on top. This results in a soft blur image that makes for a great material to use as background for cards and modules in the Microsoft Store. For instance, it is used in the app cards in the home page and in several collection pages, as well as in the new "App of the day" card.

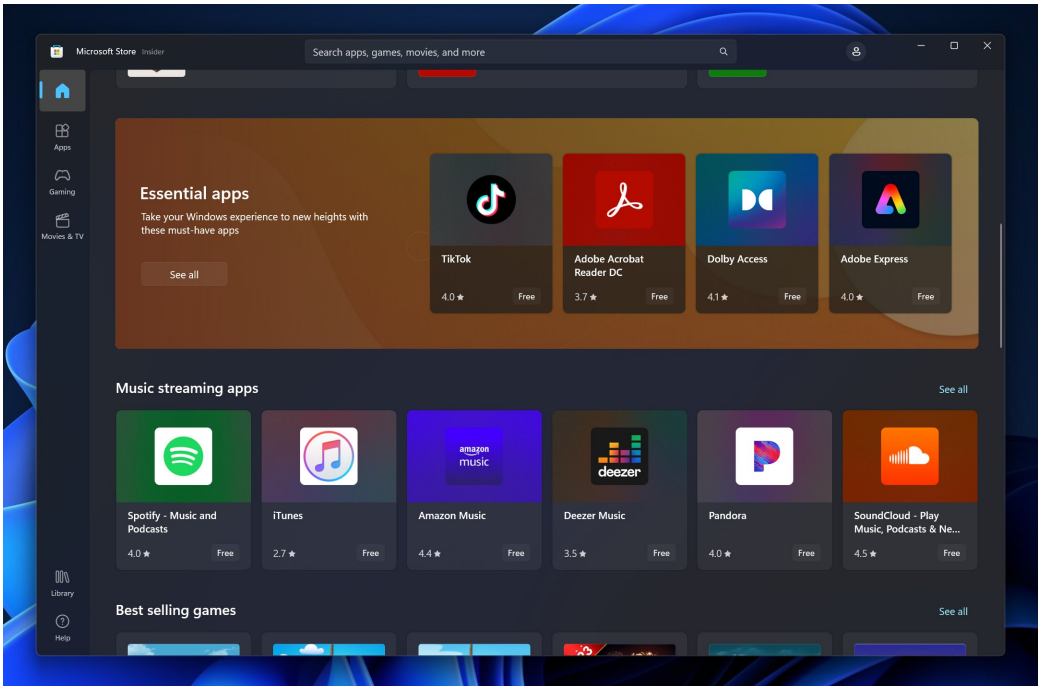
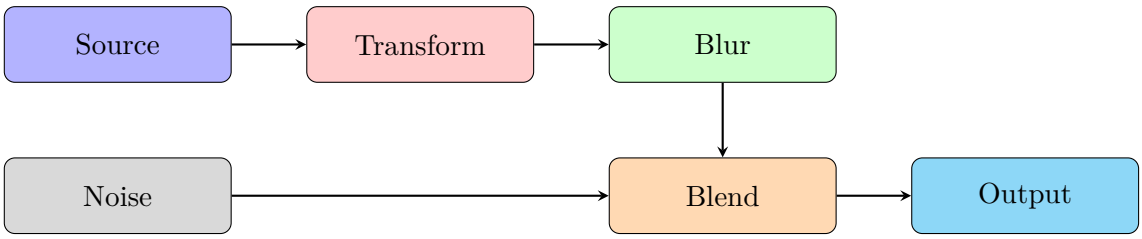


Figure 4.2. App cards in the Microsoft Store

Let's first look at the effect graph for this custom material. It's a variation of the frosted glass effect described earlier, but with an additional transform effect as well as a different noise shader. The effect graph is as follows:



That transform node plays an important role on the output, as it ensures that the source image (which is usually the icon of a given application) is properly centered before being blurred, so that the resulting area doesn't end up looking "empty" or misaligned compared to the original icon which is then overlayed on top of that background brush. The exact translation coordinates are computed when loading the image, and then just configured on the effect itself along with the other parameters. The transform is a `Transform2DEffect` object, which is Win2D's wrapper for D2D's affine transform effect [62], and allows for easy and efficient D2D affine transform manipulations on input images.

4.1.1 Noise shader

Let's go over the theme-aware noise shader that's used in this background blur effect. Here is its implementation:

```
[D2DInputCount(0)]
[D2DOutputBuffer(D2D1BufferPrecision.UInt8Normalized)]
[D2DRequiresScenePosition]
[D2DShaderProfile(D2D1ShaderProfile.PixelShader40)]
[D2DCompileOptions(D2D1CompileOptions.Default |
    D2D1CompileOptions.EnableLinking | D2D1CompileOptions.PartialPrecision)]
[AutoConstructor]
public readonly partial struct NoiseShader : ID2D1PixelShader
{
    private readonly float _alpha;
    private readonly float _minimum;
    private readonly float _maximum;

    public NoiseShader(byte alpha, byte minimum, byte maximum)
        : this(alpha / 255.0f, minimum / 255.0f, maximum / 255.0f)
    {
    }

    public float4 Execute()
    {
        int2 position = (int2)D2D.GetScenePosition().XY;

        // Compute a random value in the [0, 1] range for each target pixel
        float hash = Hash21(position);

        // The noise value is the linear interpolation between the given range
        float color = Hlsl.Lerp(_minimum, _maximum, hash);

        return new(color, color, color, _alpha);
    }

    // Computes a pseudorandom hash of a float2 value in the [0, 1] range.
    private static float Hash21(float2 x)
    {
        // These operations below don't have any specific meaning, they're just there as they
        // result in a uniform noise distribution given any input values. The numeric constants
        // also use several prime numbers to help prevent repeated patterns from showing up
        // in the output. Other than that, they're just arbitrary values and operations that
        // in practice end up producing a noise texture that is visually appealing.
        return Hlsl.Frac(Hlsl.Sin(Hlsl.Dot(x.XY, new float2(27.619f, 57.583f))) * 43758.5453f);
    }
}
```

The shader is relatively similar to the noise shader also used in the frosted glass effect, but it has a few key differences. The most important one is that rather than always using saturated RGB colors, and only modulating the alpha channel, it instead applies the noise on the RGB channels directly, along with a fixed alpha value that's configurable as well. This results in the noise also having a distinct tint color depending on the range of noise being requested. The reason for this is that the Microsoft Store supports both light and dark themes (along with high contrast themes, which don't use this effect at all to improve contrast though). Having the noise modulate the RGB channels allows the background blur to be darker or lighter as needed to properly match the theme color and to make the text displayed above it to always be readable.

Another difference compared to the frosted glass noise shader is also the use of more D2D attributes that in this case are used to further improve performance. The `D2D1BufferPrecision.UInt8Normalized` buffer option is forwarded to D2D and instructs it to only use 8 bits per channel in the intermediate buffer for the effect, if needed. This saves memory use, since the shader is only producing noise colors normalized in the 0-255 range, so it doesn't need any more precision than that (which means that using 32 bits per channel would just unnecessarily use more memory and have lower performance during buffer conversions). The buffer options are also customized, with the addition of `D2D1CompileOptions.PartialPrecision` (compared to the options that would've otherwise been applied by default). This option simply tells D2D that the shader can be evaluated with partial precision if possible, which can again result in a little performance boost, given the shader doesn't really need extra precision for the operations it needs to do.

The noise effect itself also serves another important role other than styling the final image: it also helps avoid color banding. This is a type of visual artifact that can happen due to most displays only using 8 bits per channel, meaning there's a relatively small range of colors that can be displayed per channel. This is not generally noticeable for most content, but it can become apparent when displaying color gradients or particularly blurry images: these can often result in visible "bands" of colors rather then a smooth transition. To mitigate this issue, noise can be applied on top, as it "breaks" these linear steps between shades and makes the image appear more smooth again, emulating what it would ideally look like if it was to be computed and displayed over HDR content and displays.

Figure 4.3. Noise texture comparison

In the image above, we can see a very blurred wallpaper that has a noise texture overlayed on top of it over the right 2/3rds of the image. Especially in the upper portion of the image, we can observe some color bands on the left side, which are instead mitigated and almost gone entirely in the right half of the image, where the noise is hiding this visual artifacts. This is the same benefit that we also get in the Microsoft Store when using this noise texture over blurred content, especially when using a very high blur radius, which would otherwise make the effect even more noticeable if noise wasn't also used.

4.2 Case study: game, movies and TV show cards

The second effect leveraging ComputeSharp that is available in the Microsoft Store is a "vertical blur crossfade" effect. This is a complex material that we adopted for the new game, movies and TV series cards, which are displayed throughout the application. It includes two custom D2D pixel shaders (once again implemented in C# via ComputeSharp) and a relatively large effect graph, also featuring multiple output nodes depending on the scenario where the effect is used. This is also a good showcase of the good performance that this whole approach enables: we can use even large and potentially expensive effects like this without sacrificing user experience and without making the application slow, because they're all hardware accelerated, highly optimized by the D2D effect architecture, and only rendered lazily and once

per card, rather than in real time.

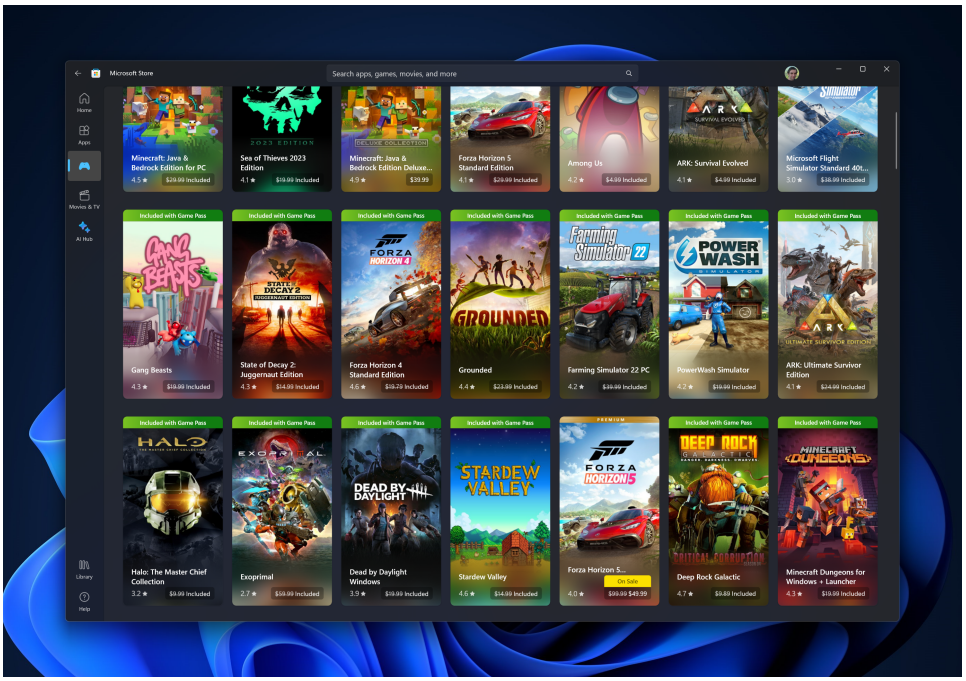
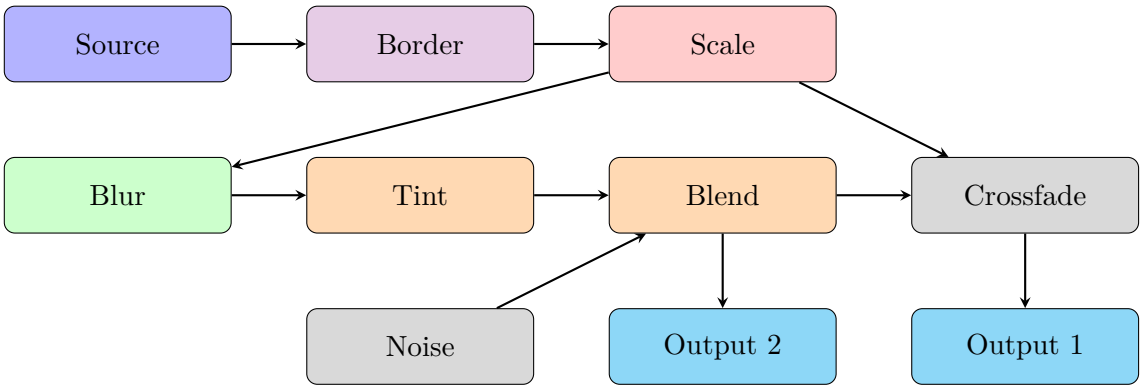


Figure 4.4. Game cards in the Microsoft Store

Like with the background blur effect, let's start by looking at the effect graph for this new material as well:



There are several moving parts here:

- The border effect [63] is used to extend the source image to cover the entire output area. Specifically, the output area has an aspect ratio around 9:16, but the source image is not as tall. The border effect is therefore used, in "mirror" mode, to fill in the bottom of the image with a flipped version of the source image. This acts as the base for the "reflection" at the bottom of each card.

- Just like the transform effect in the background blur material, the scale effect

here is also used to ensure the source image is correctly postioned. In this case, it's used to make sure that images even of slightly different size are correctly stretched to fit the width of the output area, before being mirrored. This avoids the border effect also creating a reflection on the left and right edges of the image.

- The blur, tint and noise effects are applied on the mirrored image, and they are what creates the final look of the reflection at the bottom of the card. Note that the effects are applied to the whole area first, and then just clipped aferwards.

- The crossfade effect is a custom pixel that we'll describe below. What this does is it blends these two versions of the source image (the original, stretched one, and the mirrored one with the other effects on top of it), with a vertical blur gradient. This hides the reflection point and makes it look like the bottom of the image just gradually morphs into its own reflection, which makes it for a much more visually appealing look (as opposed to the seam being clearly visible).

- Lastly, there are two output nodes, which are selected based on the size of the source image. In most scenarios, "Output 1" is selected, which is the one with the look just described. But, in some cases the source image is actually a square, and it wouldn't look right if it was processed with the same pipeline (as it would end up being too stretched and with the wrong aspect ratio). In those cases, we no longer use a reflection, but rather make the entire background of the card blurred (therefore hiding the original image entirely), and then we just overlay the original image as a separate step on top of this (in a way not unlike the app cards using the background blur effect)[1].

4.2.1 Crossfade shader

As we mentioned, the noise shader in this material is the same as the one also used by the background blur effect, but the crossfade effect is instead another custom D2D pixel shader. Its implementation is as follows:

[1] As a last example of just how useful dogfooding new APIs in real world scenarios is, this effect is what prompted the addition of the new APIs in `CanvasEffect` to register and switch between multiple output nodes. The first version of that base effect type didn't allow changing the output image, which was fixed after the first realization. It quickly became obvious how that design was making it more difficult to implement this effect in the Microsoft Store in an efficient manner, as it required us to just throw away the previous instance entirely and construct a new one (which then had to be realized) just to be able to set a different image as output node for the effect graph. Adding `EffectGraph.RegisterOutputNode` solved this problem entirely and allowed our packaged effect to be much more efficient, and with less complexity around its state management as well.

```
[D2DInputCount(2)]
[D2DInputSimple(0)]
[D2DInputSimple(1)]
[D2DInputDescription(0, D2D1Filter.MinMagMipPoint)]
[D2DInputDescription(1, D2D1Filter.MinMagMipPoint)]
[D2DPixelOptions(D2D1PixelOptions.TrivialSampling)]
[D2DRequiresScenePosition]
[D2DShaderProfile(D2D1ShaderProfile.PixelShader40)]
[AutoConstructor]
public readonly partial struct PerPixelCrossFadeShader : ID2D1PixelShader
{
    private readonly int _offsetStartY;
    private readonly int _offsetLengthY;

    public float4 Execute()
    {
        // Get the current vertical offset within the output image
        int offsetY = (int)D2D.GetScenePosition().Y;

        // Calculate the blend amount between the two inputs. This is done by calculating the
        // relative "progress" of the current Y offset within the parameters specified in the
        // shader constant buffer. That is: if Y is below the initial offset, only input 0 is
        // displayed. The input then linearly crossfades into the second one for the specified
        // amount (ie. vertical length), and then only input 1 is displayed after that point.
        float factor = Hlsl.Saturate((offsetY - _offsetStartY) / (float)_offsetLengthY);

        // Add a fast ease-out easing function to the fade factor. The input is normalized in the
        // [0, 1] range, and sin(x * C) where C > 1 increases the slope of the function, making the
        // transition quicker at first, and then gradually converging back to 1 by slowing down at
        // the end. This makes the transition more visually pleasing (the 1.57 coefficient is arbitrary).
        float easing = Hlsl.Sin(factor * 1.57f);

        // Calculate the final color by blending inputs 0 and 1 with the computed factor
        float3 blend = Hlsl.Lerp(D2D.GetInput(0).XYZ, D2D.GetInput(1).XYZ, easing);

        return new(blend, 1);
    }
}
```

Its implementation is rather simple, but it was the result of plenty of iteration to get the exact visual result we were looking for. For instance, that easing function is also arbitrary, and just the result of lots of trial and error trying to find the best possible transition between the source image and the reflection at the bottom of each card.

Chapter 5

Conclusions

Throughout this document, we saw how native DirectX and D2D functionality can be leveraged from modern Windows app using the UWP XAML framework to create all sorts of advanced graphics effects, and how ComputeSharp manages all of the complexity involved to allow developers to easily author D2D pixel shaders and custom effects without having to manually interact with any lowlebel system APIs. Furthermore, we saw real examples of how a production app like the Microsoft Store, which runs on over 1 billion devices, is taking advantage of all these features to build engaging user experiences for its users. This document covered two custom materials in particular, but we're constantly looking at new ways to push the boundary of what we can do with respect to visual effects in the Microsoft Store, and ComputeSharp has become one of the core components that we're leveraging to achieve this.

5.1 Future work

All the functionality from ComputeSharp that was covered is part of the new 2.1 release of the library, which is the first one to introduce built-in support for Win2D through all the new interop APIs that have been added to it. This is also the first stable release that the Microsoft Store has switched to in order to continue using all the ComputeSharp-powered D2D effects mentioned above. The work on ComputeSharp as a whole hasn't stopped though, and there's several new areas that are being explored to extend the scope of the project and improving its performance and usefulness. Here are some of the work items that are planned for future releases:

- Investigating D2D compute shaders (both via interop APIs in ComputeSharp.D2D1, as well as high level Win2D wrappers). This has been done through the Microsoft Global Hackathon 2023 (in September 2023), with successful results. It was possible to define and run D2D compute shaders

and to wire them up through the existing ComputeSharp infrastructure. This might be integrated into a stable release in the future if needed, pending more use case scenarios to justify further investment in this area.

- Refactor the source generator architecture to be simpler, faster, generate single files per shader type. This work has been done in https://github.com/Sergio0694/ComputeSharp/pull/578, https://github.com/Sergio0694/ComputeSharp/pull/579, https://github.com/Sergio0694/ComputeSharp/pull/580 and https://github.com/Sergio0694/ComputeSharp/pull/587.
- Restructure the API surface for supporting code for shader types. Currently, as shown in this document, the interface methods were specifically considered an implementation detail and hidden from users. This led to an unnecessarily odd API surface, and it also meant that there was no way for users to opt-out from the source generation, if they wanted to manually implement that code. This has been changed in https://github.com/Sergio0694/ComputeSharp/pull/582, through a new `ID2D1PixelShaderDescriptor<T>` interface, which has a much more streamlined API surface and is publicly documented. Together with the improvements in generated code, this also makes all the supporting infrastructure much easier to read, understand and debug, for consumers of the library.
- Targeting .NET 8 and C# 12 and taking advantage of all new features there.
- Ensuring that all WinRT-based projects also support AOT, via new CsWinRT versions.

Support for D2D compute shaders in particular is by far the most complex new feature in that list, but it would also significantly improve the flexibility of the library as a whole, allowing developers to implement even more advanced graphics effects that can then be plugged into Win2D, which would no longer be constrained to the memory access limitations that D2D pixel shaders have. As mentioned, this has already been prototyped, and we'll also be exploring new ways to leverage this in the Microsoft Store in the future, and publish our findings then.

Bibliography

[1] *DirectX graphics*. URL: https://learn.microsoft.com/en-us/windows/win32/getting-started-with-directx-graphics.

[2] *Component Object Model (COM)*. URL: https://learn.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal.

[3] *WinRT type system*. URL: https://learn.microsoft.com/en-us/uwp/winrt-cref/winrt-type-system.

[4] *Win2D*. URL: https://learn.microsoft.com/en-us/windows/apps/develop/win2d/.

[5] *UWP XAML*. URL: https://learn.microsoft.com/en-us/windows/uwp/xaml-platform/.

[6] *Component Object Model*. URL: https://en.wikipedia.org/wiki/Component_Object_Model.

[7] *The Component Object Model*. URL: https://learn.microsoft.com/en-us/windows/win32/com/the-component-object-model.

[8] *QueryInterface*. URL: https://learn.microsoft.com/en-us/cpp/atl/queryinterface.

[9] *COM technical overview*. URL: https://learn.microsoft.com/en-us/windows/win32/com/com-technical-overview.

[10] *ComWrappers*. URL: https://learn.microsoft.com/en-us/dotnet/standard/native-interop/tutorial-comwrappers.

[11] *C# function pointers*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/function-pointers.

[12] *IUnknown*. URL: https://learn.microsoft.com/en-us/windows/win32/directshow/how-iunknown-works.

[13] Raymond Chen. *A slightly less brief introduction to COM apartments*. URL: https://devblogs.microsoft.com/oldnewthing/20191125-00/?p=103135.

[14] *ComPtr<T>*. URL: https://learn.microsoft.com/en-us/cpp/cppcx/wrl/comptr-class.

[15] *Resource Acquisition Is Initialization*. URL: https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization.

[16] *COM aggregation*. URL: https://learn.microsoft.com/en-us/windows/win32/com/aggregation.

[17] *UnmanagedCallersOnlyAttribute*. URL: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.unmanagedcallersonlyattribute.

[18] *Windows Runtime*. URL: https://en.wikipedia.org/wiki/Windows_Runtime.

[19] *Windows Metadata (WinMD) file format*. URL: https://learn.microsoft.com/en-us/uwp/winrt-cref/winmd-files.

[20] *CsWinRT*. URL: https://learn.microsoft.com/en-us/windows/apps/develop/platform/csharp-winrt/.

[21] *AppContainer isolation*. URL: https://learn.microsoft.com/en-us/windows/win32/secauthz/appcontainer-isolation.

[22] *What is MSIX?* URL: https://learn.microsoft.com/en-us/windows/msix/overview.

[23] *Extensible Application Markup Language*. URL: https://en.wikipedia.org/wiki/Extensible_Application_Markup_Language.

[24] *TaskbarManager API*. URL: https://learn.microsoft.com/en-us/uwp/api/windows.ui.shell.taskbarmanager.

[25] *Win2D interop with D2D*. URL: https://learn.microsoft.com/it-it/windows/apps/develop/win2d/interop.

[26] *D3D12MA*. URL: https://gpuopen.com/d3d12-memory-allocator/.

[27] *PIX on Windows*. URL: https://devblogs.microsoft.com/pix/introduction/.

[28] *SwapChainPanel*. URL: https://learn.microsoft.com/en-us/uwp/api/windows.ui.xaml.controls.swapchainpanel.

[29] *HLSL*. URL: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl.

[30] *The .NET Compiler Platform SDK*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/.

[31] *Roslyn source generators*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview.

[32] *IntelliSense*. URL: https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense.

[33] *Roslyn syntax analysis*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/syntax-analysis.

[34] *Roslyn semantic analysis*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/semantic-analysis.

[35] *D2D1_PIXEL_OPTIONS*. URL: https://learn.microsoft.com/en-us/windows/win32/api/d2d1effectauthor/ne-d2d1effectauthor-d2d1_pixel_options.

[36] *D2D effects linking*. URL: https://learn.microsoft.com/en-us/windows/win32/direct2d/effect-shader-linking.

[37] *D2D1_BUFFER_PRECISION*. URL: https://learn.microsoft.com/en-us/windows/win32/api/d2d1_1/ne-d2d1_1-d2d1_buffer_precision.

[38] *D2D1_CHANNEL_DEPTH*. URL: https://learn.microsoft.com/en-us/windows/win32/api/d2d1effectauthor/ne-d2d1effectauthor-d2d1_channel_depth.

[39] *HLSL helpers*. URL: https://learn.microsoft.com/en-us/windows/win32/direct2d/hlsl-helpers.

[40] *HLSL intrinsics*. URL: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-intrinsic-functions.

[41] *HLSL data types*. URL: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-data-types.

[42] *Paint.NET*. URL: https://www.getpaint.net/download.html.

[43] *ID2D1EffectContext::CreateResourceTexture*. URL: https://learn.microsoft.com/en-us/windows/win32/api/d2d1effectauthor/nf-d2d1effectauthor-id2d1effectcontext-createresourcetexture.

[44] *ID2D1ResourceTexture::Update*. URL: https://learn.microsoft.com/en-us/windows/win32/api/d2d1effectauthor/nf-d2d1effectauthor-id2d1resourcetexture-update.

[45] *Runtime Callable Wrapper*. URL: https://docs.microsoft.com/dotnet/standard/native-interop/runtime-callable-wrapper.

[46] *ID2D1Transform::MapInputRectsToOutputRect*. URL: https://learn.microsoft.com/windows/win32/api/d2d1effectauthor/nf-d2d1effectauthor-id2d1transform-mapinputrectstooutputrect.

[47] *ID2D1Transform::MapOutputRectToInputRects*. URL: https://learn.microsoft.com/windows/win32/api/d2d1effectauthor/nf-d2d1effectauthor-id2d1transform-mapoutputrecttoinputrects.

[48] *ID2D1Transform::MapInvalidRect*. URL: https://learn.microsoft.com/windows/win32/api/d2d1effectauthor/nf-d2d1effectauthor-id2d1transform-mapinvalidrect.

[49] *COM Callable Wrapper*. URL: https://learn.microsoft.com/dotnet/standard/native-interop/com-callable-wrapper.

[50] Sergio Pedri. *D2D effects overview*. URL: https://learn.microsoft.com/en-us/windows/win32/direct2d/effects-overview.

[51] *D2D shader profile*. URL: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/specifying-compiler-targets.

[52] *IGraphicsEffectSource*. URL: https://learn.microsoft.com/en-us/uwp/api/windows.graphics.effects.igraphicseffectsource.

[53] *Win2D interop*. URL: https://microsoft.github.io/Win2D/WinUI2/html/Interop.htm.

[54] *IWeakReferenceSource*. URL: https://learn.microsoft.com/en-us/windows/win32/api/weakreference/nn-weakreference-iweakreferencesource.

[55] *winrt::get$_{a}ctivation_{f}actory_{f}unction$*. URL: https://learn.microsoft.com/en-us/uwp/cpp-ref-for-winrt/get-activation-factory.

[56] *ID2D1Effect::SetInput*. URL: https://learn.microsoft.com/en-us/windows/win32/api/d2d1_1/nf-d2d1_1-id2d1effect-setinput.

[57] *D2D device mismatch*. URL: https://github.com/microsoft/Win2D/issues/913.

[58] *Octagrams*. URL: https://www.shadertoy.com/view/tlVGDt.

[59] *C# 12 primary constructors*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/primary-constructors.

[60] *[UnsafeAccessor] attribute*. URL: https://github.com/dotnet/runtime/issues/81741.

[61] *Handle device removed scenarios in Direct3D 11*. URL: https://learn.microsoft.com/en-us/windows/uwp/gaming/handling-device-lost-scenarios.

[62] *D2D spatial transforms*. URL: https://learn.microsoft.com/en-us/windows/win32/direct2d/effects-overview#spatial-transforms.

[63] *D2D border effect*. URL: https://learn.microsoft.com/en-us/windows/win32/direct2d/border.