SharpMedia Graphics Guidelines

© 2007Žiga Osolin – SharpMedia project, document version: DRAFT

Table of Contents

[About 4](#_Toc170220373)

[Prerequisites 4](#_Toc170220374)

[Goals 4](#_Toc170220375)

[Architecture 5](#_Toc170220376)

[Pipeline 6](#_Toc170220377)

[Input Stage 6](#_Toc170220378)

[Vertex Shader 7](#_Toc170220379)

[Geometry Shader and Geometry Output 7](#_Toc170220380)

[Rasterization 7](#_Toc170220381)

[Pixel Shader 7](#_Toc170220382)

[Output Stage 7](#_Toc170220383)

[Typeless Resources and Views 8](#_Toc170220384)

[Buffers 9](#_Toc170220385)

[Late Device Binding and RAM Access 10](#_Toc170220386)

[Geometry Usage 10](#_Toc170220387)

[Vertex Format 10](#_Toc170220388)

[Index Format 11](#_Toc170220389)

[Instancing 11](#_Toc170220390)

[Texture and Render Target Usage 11](#_Toc170220391)

[Constant Buffers 11](#_Toc170220392)

[Textures 12](#_Toc170220393)

[Pixel Format 13](#_Toc170220394)

[Typeless Formats 13](#_Toc170220395)

[Compression Formats 13](#_Toc170220396)

[Depth Stencil Formats 14](#_Toc170220397)

[Texture Usage 14](#_Toc170220398)

[Render Target and Depth Stencil Target Usage 14](#_Toc170220399)

[Multisampling 14](#_Toc170220400)

[DAG 15](#_Toc170220401)

[Pins 15](#_Toc170220402)

[Arrays 16](#_Toc170220403)

[Operations 17](#_Toc170220404)

[Scoping 18](#_Toc170220405)

[Functions 19](#_Toc170220406)

[Interfaces 20](#_Toc170220407)

[Interface and Interface Operation 20](#_Toc170220408)

[Reusing Shaders 20](#_Toc170220409)

[CPU resolving 21](#_Toc170220410)

[Fixed and Non-Fixed Parameters 22](#_Toc170220411)

[Shared Constant Buffers 22](#_Toc170220412)

[Constant Buffer Grouping 22](#_Toc170220413)

[Interfaces and Constants 23](#_Toc170220414)

[Fixed Optimizations 24](#_Toc170220415)

[Textures and Samplers 24](#_Toc170220416)

[Metadata Programming 25](#_Toc170220417)

[Pin Binder 25](#_Toc170220418)

[DAG Generator 25](#_Toc170220419)

[Assembling 26](#_Toc170220420)

[Debugging 26](#_Toc170220421)

[State Objects 27](#_Toc170220422)

[State Pool 27](#_Toc170220423)

[Graphics Device 28](#_Toc170220424)

[Rendering State 28](#_Toc170220425)

[Rendering 28](#_Toc170220426)

[TypelessTexture Sharing 28](#_Toc170220427)

[Graphics Service 29](#_Toc170220428)

[Explicit Mode 29](#_Toc170220429)

[Shared Mode 29](#_Toc170220430)

[How To 31](#_Toc170220431)

[Questions 32](#_Toc170220432)

# About

These guidelines will outline the coding guidelines for SharpMedia Graphics library. Graphics library gives you control to GPU. With efficient usage of library, you can exploit this library to speed up many tasks, also those not related to computer graphics.

# Prerequisites

Reader should be familiar with SharpMedia Services architecture and have basic knowledge of DirectX/OpenGL pipelines.

# Goals

By the end of this document, you should be able to:

* Obtains a graphics service object and create a device is different modes;
* Share devices and textures;
* Understand typeless and typed concepts of resources;
* Create different buffers and textures;
* Load textures or buffers using resource loader;
* Create a DAG manually;
* Create a DAG using metadata;
* Create a DAG using assembler;
* Compile a DAG to shader;
* Create and bind state objects;
* Render the scene;
* Provide constant buffers and optimize DAG for packed constant buffers;
* Use typeless textures as render targets and textures;
* Write geometry shader and bind it to GO buffer;
* Utilize DAG interfaces and arrays.

Note that this is not a reference but a guide through Graphics library. You may still need to check actual classes and methods after you finish reading. This guide will however explain to you more advanced concepts. When you read through the whole guide, you should be able to quickly understand how things work but not necessary how to do it. You will be however able to quickly find what you need in API reference.

# Architecture

Graphics is a complex library and service in one. To API users, it is mostly a library, since all service functionality is abstracted away by using client side classes. In its heart, the graphics is a service; the driver usually resides in System process and acts as a service to many client side devices and objects. There is also a special exclusive mode, where driver’s code is injected into process’ application domain and used without remoting.

Figure - Graphics as Service

Client side objects are not just wrappers around driver objects. They provide a rich API for working with specific classes. Besides all that, data holding resources (such as textures and buffers) can exist **without** device and are automatically bound to device when this is required.

## Pipeline

Graphics library is based on graphics pipeline. Pipeline is becoming more and more programmable but there are still many parts where fixed function is used.

Figure – A simplified graphics pipeline

Pipeline consists of many stages. The user can configure the pipeline through input, state objects, shaders, shader resources, geometry output targets and render targets. Fixed parts are rasterization and interpolation, clipping, depth and stencil tests and other data manipulation steps.

Pipeline is highly parallel element; vertices, geometry and pixels are all processed by separate units.

### Input Stage

The input stage of pipeline is responsible for describing input. In SharpMedia, all input describing task is abstracted by **Geometry** class. A set of **VertexBuffer**s[[1]](#footnote-2) and an optional index buffer are given in order to describe an input required by the vertex shader.

### Vertex Shader

A vertex shader executes once for each vertex. If geometry is indexed, vertices may be cached if they can be reused; otherwise they are recalculated. A vertex shader input signature must match the geometry signature.

Vertex shader can also use other resources not specified per vertex. The constant buffer[[2]](#footnote-3) can be used to provide global constants that do not change between vertices. This can include global time, camera matrix, animation matrices and more. Up to 16 constant buffers may be used. Data is usually packed in the same buffer by the frequency of update (once per frame, once per primitive …).

As a result, vertex shader outputs a set of outputs that are used by next pipeline element (can be geometry shader or rasterization).

### Geometry Shader and Geometry Output

Geometry shader is optional. Geometry output is also optional. This type of shader works upon geometry primitive; that can be either a point, line, triangle or any of the enumerated with adjacencies. Given a primitive, shader can output any number of generated primitives that are than output to geometry bound as geometry output (if present) and further processed by next stages.

Shader can use the same resources as vertex shader to accomplish its task.

### Rasterization

Rasterization converts position into screen space by dividing position by its homogenous components. After that, it converts primitive to pixels. For each pixel, all per-vertex components are interpolated to per-pixel components. All pixels are than sent to pixel shaders (along with their per-pixel components).

An early Z-test can happen after rasterization if pixel shader does not write to depth value. If Z-test fails, pixel is never processed. Scissor test is also executed after rasterization.

### Pixel Shader

Pixel shader does processing per pixel. As any shader, it can use all the resources available to shaders. As an output, pixel shader outputs so many colours as there are render targets. Optionally, it can also output the depth value.

### Output Stage

Output stage takes care for Z-test if not previously tested before pixel shader. Stencil test is also executed and stencil buffer may be updated (if present). After that, blending of incoming pixel colour and existing colour is done for each render target. All render targets **must** have the same:

* Format;
* Dimensions;
* Sampling.

## Typeless Resources and Views

For flexibility, all resources are typeless. This means that their internal format is not known at their creation. A resource can be viewed using a certain format and offset. Only views of resources are boundable as stages in graphics pipeline. A C++ equivalent example of this is a void\* pointer. As such, it holds only a raw data with no format. Once such pointer is converted to some useful format it can be used for manipulation.

We know two types of graphical resources: buffers and textures. The difference between the two are that buffers are more generic and can be bound almost everywhere. On the other hand, textures are optimized for sampling, allow mipmaps and can have multiple dimensions.

Typeless resources give us a lot of flexibility. For instance, a buffer can be viewed as a vertex buffer or as a render target. Most of the time, you will only use the views. However, sometimes you need more flexibility; for example, you may need your buffer to be updated by geometry output stage and then bound as geometry, or even as constants buffer. This can be easily accomplished with resource views.

# Buffers

Buffers are most generic elements of the GPU. They are defined by size in bytes. Because communication with GPU is not trivial and GPU is highly specialized unit, additional properties must be set so the GPU knows more about your buffer’s intended use and its frequency, CPU management and more. Each buffer has the following properties:

* The size in bytes;
* Resource usage – specifies what operations are permitted by GPU (read, write or both) and how frequent will CPU read or write to resource;
* Buffer usage – enumerates all possible buffer usages, such as using buffer as vertex buffer, index buffer, texture, render target or geometry output. Always use most restrictive combination;
* CPU access – which access is permitted by CPU. This must be compatible with resource usage. Even if resource usage allows CPU read back, not setting CPU reading capability does not allow this operation;
* Locality – locality specifies where buffer is placed. You can specify the buffer to always reside in RAM, to always reside in GPU memory or to reside in both memories. There is also an option the buffer to reside in RAM when not bound to device and to reside in GPU when bound to device;

If RAM copy of buffer exists and the contents are not modified by GPU, reading of buffer is free. It is possible even when buffer does not have CPU read access (this applies for GPU part of buffer only). This may however become restricted if buffer is changed by the GPU operations and read back is not permitted.

Figure – Buffer and its usages hierarchy

## Late Device Binding and RAM Access

Buffers can be late-bound to device. Even buffer views can be late bound. This enables buffers to exist before actual device is created. Sometimes, device may even not need to be created if you manipulate buffer is software. Late binding also allows for buffer serialization and deserialization[[3]](#footnote-4).

Late binding of buffers happens when buffer is explicitly bound to device using a **BindToDevice** method. You can also explicitly free the buffer from device using **UnBindFromDevice** method. The buffer is also bound to hardware if any of its views are bound to pipeline. A buffer is never automatically freed from device. On device releasing, each buffer that is bound to device must be either disposed or unbound from device. If you do not unbind them, exception is thrown.

## Geometry Usage

Vertex and index buffer views can be simply created using **Buffer**’s **Create\*** methods. Upon creation, vertex format must be supplied for vertex buffer and index format for index buffer. Format already includes the element width. You can optionally provide an offset from the beginning of the buffer.

In order to use such buffers as geometry, they must be bound to **Geometry** class. Geometry is fed by (possibly) many vertex buffers and an optional index buffer. If index buffer is not present, vertices are processed in ascending order. Each vertex buffer can bind only specific components of vertex format, such as only *Position* in “P.F3 N.F3”[[4]](#footnote-5) format. After all buffers are correctly bound, such geometry can be bound to pipeline. We can bind it as input for vertex shader or as output of geometry shader. Geometry shader output geometry may not contain an index buffer and can contain maximum of 4 buffers.

### Vertex Format

A vertex format defines the layout of elements in buffer. The format describes how to interpret data in certain buffer to be bound to **Geometry**. **VertexFormat** consists of *components*. Each component contains information about its offset, pin component and component format. The pin component maps to shader’s component; most useful are *Position, Normal, TexCoord0* etc. A format can be used to interpret data in appropriate way. Most common component formats are vectors and matrices of different floating point precisions. Integer types are also supported.

Vertex format should be created using a **VertexFormat.FromString(string)** static method. When created, it can be freely shared because it is immutable and thus thread-safe. The string description consists of component – component format pairs, separated by spaces:

* “P.F2 N.F3” – position of floating point 2D vector followed by normal of 3D floating point vector;
* “T0.H2 X.F T1.H2” – texture coordinate 0 of half 2D vector, followed by unused 4 bytes (size of ‘F’) and followed by another texture coordinate.

### Index Format

Index format is pretty simple and describes the **IndexBuffer** layout. Index buffer contains only indices that are tightly packed. Each index can be only 2 or 4 bytes long.

### Instancing

Geometry also provided vertex buffer binding per instance, not per vertex. This means that the data will be common to all elements in the same instance, not only vertex. There is also an option to bind data per multi-instance (updates every N instances). We call it multi instancing. This call be useful to limit number of batches (actual render calls) to improve performance.

## Texture and Render Target Usage

A **Texture** class is typed view of either **Buffer** or **TypelessTexture\***. It can be bound to any shader. Up to 128 textures can be bound to each shader. A texture is described by offset, element stride and **PixelFormat[[5]](#footnote-6)** in buffer’s case. A **RenderTarget** is similar to texture, except that data is written to it, not read from it.

## Constant Buffers

Constants buffers are views of **Buffer**s. Each constant buffer is limited in size; it is maximum 65536 bytes long. This is enough to put inside up to 4096 4-dimensional floating point vectors. This buffer is special because it is not described by any special format but description is given by actual shader. A shader can bind up to 16 constant buffers at the same time.

It is good if you sectionize data to buffers by frequency of update and relations. For example, there must be a big chance that if some part of buffer changes, that other parts of it change too. An example would be an object that has animation, lights and uses shared constants. Animation constants can be packed into one constant buffer, lights into another and shared constant into another. Changing animation usually implies only buffer changing (and *no* buffer update). Lights parameters are not changed frequently so this kind of packing is fast. Shared constants are never changed. A DAG API takes care of data sectionizing.

# Textures

Typeless textures are other type of resources available. They are specialized resources that can be sampled. A texture can be used as a **Texture[[6]](#footnote-7)**, **RenderTarget** or **DepthStencilTarget** through the views mechanism.

When texture is created, the following properties must be provided:

* Resource usage – where to place such texture and what kind of access is available by GPU (same as described by buffer);
* Texture usage – can or *Texture*, *Render Target* and *Depth Stencil* usage. A special *Cube Map* usage can also be applied for arrays of 6 2D textures;
* Locality – how RAM backups are handled if necessary;
* Format layout – may be typeless[[7]](#footnote-8);
* Dimensions and face count;
* Mipmaps – can specify 1 to not use mipmaps. Specifying 0 means full mipmaps chain.

As for buffer, late binding is also practiced. If textures are immutable, we however suggest filling them at creation and avoid additional resource copy in RAM.

Figure – Typeless Texture Views and Usages

## Pixel Format

A pixel format is a bit more complex than vertex format. This is mainly due to **compression formats** and **hardware restrictions**. A format is described similarly to vertex format. There are components (*Red, Green, Blue, Alpha, Bump, Refraction …*) that have component formats (*UInt8, UNorm8[[8]](#footnote-9), Float16, SNorm16[[9]](#footnote-10), SInt32 …*). Each component also owns in offset in format.

A pixel format has associated **common format layout**. This format describes how components are packed together. It is automatically calculated from format’s description and can be non-standard. Only if *common format layout* is standard can the format be bound to hardware. An example of such format is “R.UI8 G.UI8”[[10]](#footnote-11). It has common format layout X8Y8\_UINT, meaning that there are two components mapped to X (Red) and Y (Blue) when accessing texture form shader, both treated as 8-bit unsigned integers.

Note that you **can** freely use non-standard formats but they can’t be bound to hardware. They may however be useful for some image resources. For example, you are not limited to use only 4 components in your format; you can pack much information into one element of texture/image. Such formats can be *split* or *joined* using Images library. Only hardware requires special format layout, image library can work with any valid format.

### Typeless Formats

To make things even more complicated, typeless formats are introduced. It is sometimes desirable to not describe format fully but only provide details about its size and components, but not fully determine component formats. The component must specify the size of format but not the actual format. For example **Typeless16** format may be used instead of all 16-bit formats such as **UInt16, SInt16, Float16, UNorm16** and **SNorm16**. The common format layout of such format also becomes typeless.

Typeless textures disable some operations since actual format is not known. Size of format, components and typeless layout is however known and this is enough to create hardware resource if this is possible. When you typed view of **TypelessTexture** is created, the full format specifications must be known.

### Compression Formats

Compression formats are just *modifiers* of formats. All compression is done behind the scenes if not explicitly requested otherwise. Each compression format usually maps to one or more pixel formats. Compressions are usually quite specialized and require specific formats. If format describes compression, it is **Image/Texture**’s responsibility to decompress such format upon mapping and compress it upon unmapping of texture. There is also a **no-auto-compression/decompression** option when mapping images.

### Depth Stencil Formats

Depth stencil formats are somewhat special. They contain their own *common format layout*s. This is because hardware does not hold depth stencil textures in ordinary way, such as defined by pixel format. Depth stencil formats are usually optimized for depth stencil operations and are in hierarchical, compressed form.

This means that *“D.F32”* will result in different in different *common format layout* than “R.F32”, even though they may seem the same.Depth and stencil components are treated differently than other components.

## Texture Usage

The most *popular* usage for **TypelessTexture** is usage as texture. It can be bound to any shader, as buffer textures. Texture views of **TypelessTexture** support sampling and can use mipmaps if required. Apart from that, they support multiple dimensions (buffers don’t). Only such textures can be bound as 2D, 3D, Cube, 1D array and 2D array textures.

Figure – texture and it’s “in-shader” types.

## Render Target and Depth Stencil Target Usage

A texture can also be used as **RenderTarget** or **DepthStencilTarget**. Usually, depth stencil formats are too restrictive to enable any other usage but depth stencil usage. In future, we expect this to change. A **TypelessTexture** can however be used as render target and texture simultaneously. Even array textures can be bound as render targets. In such case, the pixel shader must choose the array index where to write result.

## Multisampling

**TypelessTexture**s can have multisampling applied. Multisampling consists of number of passes and quality of sampling. Textures with multisampling cannot have mipmaps. If it is bound as render target, all other targets must match in multisampling passes and their quality.

# DAG

A DAG (directed acyclic graph) represents a group of shaders. There must be no cycles in the graph. Graph consists of operations, connected by pins. An operation represents some sort of pin processing while pins represent the data flow. Each pin is fully typed. Operations can take varying number of pins as arguments and output varying number of pins. Pin can only be created by operation. There is always the **InputOperation** and the **OutputOperation** associated with graph. The input operation provides per-element data (per-vertex, per-geometry, per-pixel) while output operation outputs data for each element.

DAG qualifies as mathematical graph and graph operations can be applied to it. Those are used for different analyses. DAG is transferred back to forward because only operations that affect the output operation are required. This effectively implements dead code elimination. This means that **only** operations that can be reached from output operation are part of DAG, all others are discarded[[11]](#footnote-12).

## Pins

A pin has type. The owner of the pin is the operation that created it. Each pin is immutable and represents data flow. It can be bound to one or more operations as input.

Figure – Owner operation’s output pins are used as input pins of other operations

Pins are never created manually; they are always created by operations. This means that some operations do not require an input, only produce outputs. One of such operations is **InputOperation**. There are also others, such **ConstantOperation** (provides constants).

We divide pins in several categories, based on their usages. We know data pins. They can be integer or floating point based. The other kind of dividing is into scalars, vectors and matrices.

Figure – Pins categorization.

Texture pins support random access to data stored in textures with special operations that allow that. Textures and samplers can be used together to define data filtering. Interfaces pins are described later in the Interfaces section. They are a way to describe DAG without specifying everything at its creation but at runtime (compile time, actually). Dummy pins are used to link certain operations that provide no output but affect the shader. One of such operations is **Discard**. It can discard a pixel if input value is true. However, if it were not linked with **OutputOperation** using dummy pin, it would be dead-code eliminated and not included in shader when compiled.

### Arrays

A pin can also represent an array of pins of the same type. Array can have static size or size specified at runtime when DAG is compiled to shader. There are also special operations working on arrays, such as **AddArray** (adds all elements of arrays into one pin) or **MulArray** (multiplies all elements of array into one pin). All such operations work with both dynamically or statically specified array sizes.

## Operations

Operations are vertices in graph. Each operation may require certain pins (with meaning and requested) to be bound to them. They perform some sort of operation on those pins and output resulting pins. Operations are actually some sort of transformation element. Note that operation **do not** change pins, they create new immutable pins as outputs. Before operation can create output pins, **all** input pins must be provided. This in essence means that there can be no cycles in the graph.

Each operation is able to describe input pins it requires and output pins in produces. Because many times, not only one typed pin is acceptable, the description is more generic. It consists of pin description and or-ed formats it can have. This is done using **PinDescription** class. The **PinRelation** class contains the relation between two pins. For example, **AddOperation** has two input pins that can have any arithmetic type. However, they are related – they must both be of the same type. A relation between descriptions is created that claims that both formats *must* be the same.

Figure – Pin description and relations. For output pins, there are no relations because the format is perfectly defined when both input pins are bound.

Such format description allows for more generic validation and gives the editor more ideas about what the operation needs. It can automatically perform conversions and help the use to interact faster with DAG.

Common operations are addition, multiplication, subtraction, sampling, loading etc. When shader is compiled, all operations must be able to describe themselves to **ShaderCompiler**. The same operation must always produce the same code if fixed parameters are not changed.

## Scoping

Scoping is also an important aspect of DAG. A scoped operation is a special operation that contains internal operations. Internal operations are either fully isolated from graph (plug-in procedures) or contain inputs from higher scopes of DAG. In both cases, internal operations’ pins **cannot** cross the scoping operation’s boundary.

Operation 0

Input Pin 0

Output Pin 0

Internal Operation 1

Operation 1

Scoped Operation  
  
  
  
  
  
  
Internal Input Pin 0 Internal Output Pin 0

The figure above shows some of possible scenarios. The scoped operation takes one input pin (from *Operation 1*) and returns one output pin. This scoped operation has only one internal operation; this operation takes an internal input pin (*Internal Input Pin 0*) and one pin from operation (*Operation 0*) from higher scope. The internal operation writes result to *Internal Output Pin 0*. The scoped operation can define mechanisms how (and if) *Input Pin 0* and *Internal Input Pin 0* are related. The same is true for output pins.

Loops can be implemented using scoped operation. Let’s say that Input Pin 0 defines number of iterations. Internal Input Pin 0 is initially set to default value (0 or null vector). When data is written to Internal Output Pin 0, Internal Input Pin 0 can be updated with new value of Internal Output Pin 0. When iteration is over (depends on Input Pin 0), the Output Pin 0 has the value of Internal Output Pin 0 after those iterations. This is translation of cyclic structure (loop) to acyclic (DAG).

Operations are automatically assigned to scopes. There is **no** explicit control to place an operation in certain scope. For each operation, we check all operations defined by their input pins (owners of those pins):

* If pin is Internal Input Pin, we assign scope to the scoped operation that defines that pin. This is done only if no deeper scope is already assigned to operation;
* If pin is linked to operation that has deeper scope than our current scope, we correct the scope to that operation’s scope.

### Functions

Scoped operations where internal operations that don’t depend directly on any other operation outside operation’s scope are functions. A delegate has fixed input and output pins. A user or API can define a set of functions that can be used in DAGs. For instance, the GPGPU API describes some algorithms as such scoped operations.

## Interfaces

Interfaces are pins that hold not only all relevant data but also procedure how something is accomplished. All interfaces are applied at **runtime** in the same way as setting constants.

An example of interface is a BDRF[[12]](#footnote-13). A simple albedo[[13]](#footnote-14) BDRF will ignore input and output ray directions and only sample a texture at certain position. It will define a texture to be associated with such interface pin. Another version of BDRF is diffuse material. It computes the colour based on light direction. As such, it has only material diffuse colour associated.

The power of interfaces is that everything is described generically (without actual implementation). Interface pins are defined at runtime, together with parameters they need. Then, DAG is compiled to shader. For example, a light is an interface. The Scene library decides which lights are influencing object and only those lights are specified as interfaces to DAG. If required[[14]](#footnote-15), DAG is compiled to describe the correct shader and shader is executed for the object.

### Interface and Interface Operation

Specific interfaces can only be used with specific operations that “know” them. The **LightingOperation[[15]](#footnote-16)** defines many inputs. Among interface pin inputs are array of lights, a BDRF and a lighting mixer interface. Each light defines its own parameters and procedure how to compute lighting. A BDRF describes material – how light interacts with material. The final, lighting mixer defines how lighting data from various lights in mixed together into one output.

In C#, each interface element must implement **IInterface** interface. Operations can define specific interface types that must be bound to locations. For example, **LightingOperation** requires an array of **ILight** interface pins to be bound to certain input slot. At runtime, the user can only specify classes that implement **ILight** for parameter, or subscription will fail.

### Reusing Shaders

Interfaces are very generic and with no optimizations, we would need to recompile the DAG for every rendering. This is not acceptable due to slow compile time. A DAG allows shaders to be reused by asserting the following:

* The same interface type **must always** provide the same implementation – this means that implementation must not be based on any outside variable (such as *optimizing* the shader from one multiplication if diffuse colour of light is white);
* In interface array, interfaces must be pushed always in the same order. For example, lights are always in array in the same order. If we would reorder them, this would result in recompilation of shader (because shader changes);
* All fixed values[[16]](#footnote-17) must stay the same. We suggest you omit using fixed values if they change frequently or if they do not contribute to any important optimizations (such as loop unrolling).

### CPU resolving

Some parts of the DAG must be resolved at compile time. This is CPU resolving. Without interfaces, this is never required. DAG is simply converted to shader code. With interfaces, some tasks cannot be executed on GPU. For instance, if we have an array of interfaces, we can manipulate such array by adding new interfaces to it or by removing some interfaces. The analysis must be done at CPU. What is more, adding interfaces may be conditional which makes CPU analysis a bit more complicated. We may actually need to create branches – two or more different ways to execute shader based on interfaces.

CPU resolving can be more efficient if conditions are fixed parameters because this means that everything will be evaluated at CPU and compiled to GPU code without branches.

## Fixed and Non-Fixed Parameters

Constant parameters can be defined using **ConstantOperation**. This operation creates a named pin that contains value defined at shader compiling or shader using stage. Shader compilation goes through the following stages:

1. Enumerate all operations that affect DAG in correct order;
2. Go through all constant operations and extract parameters;
3. Create **FixedShaderParameters** from DAG. It prepares DAG for compilation by providing all necessary fixed data (interface types must be provided here, some may be optionally provided);
4. Compile shader by translating operations first to driver code (HLSL, GLSL …) and then running driver compiler. Shader object is available at this stage;
5. Create constant buffers:
   1. Specify **ShaderParameters** (created from **FixedShaderParameters**) to pack additional constant parameters to constant buffers as required by shader. Use **Pack()** method on **ShaderParameters** to pack data into buffers and obtain buffers;
   2. Use **ConstantBufferLayout** array to manually create and provide all buffers;
   3. Use combination of **ShaderParameters** and **ConstantBufferLayout** to describe buffer (by manually editing some buffers and editing some buffers through shader parameters).
6. Binds constant buffers and shader.

### Shared Constant Buffers

Shader parameters are packed in **TypelessBuffer**s with **ConstantBuffer** view. Through buffers, we can provide a big amount of data. The data layout is defined by shader and **ShaderParameters** abstract this away from user. This means that user will usually use named binding.

Sometimes, it is preferable that some constants are shared. For instance, there may be some constants defined for the whole scene. It may include data such as camera position, camera projection matrix, current time, delta time etc. We certainly would not like to update this buffer for each object. It could be shared between all objects that require it.

We allow this by adding use defined **ConstantBufferLayout** class when compiling the shader (provided to **FixedShaderParameters**). This description holds named parameters with their offset and types. Names must be globally unique. When description is bound to such buffer, all named constants with the same name as any of element described in the layout are automatically bound to this buffer (under certain index) with appropriate offsets and types. Type mismatch throws an exception.

### Constant Buffer Grouping

Apart from specifying shared constant buffer descriptions, we can also group constants, usually by frequency of update. For example, we have a buffer that changes once or more per frame (constants that are recalculated every time), a buffer that changes every second (animation), a buffer that almost never changes (immutable constants, such as diffuse colour of material …) and also some shared constants as described in previous section.

Figure – parameters can be grouped into constant buffers by frequency of update. If group is not provided, default group is always used.

Such grouping is very effective because changes are usually applied to only one buffer (default constants buffer). This requires less buffer mappings.

Grouping is done by specifying a *group object*. If null is used, default group is applied. Otherwise, two objects are in the same group if their group object is the same. Only up to 15 different group object (and null object) can be applied. The position (constant buffer index) in group is not defined until shader is compiled.

**ShaderParameters** is actually a wrapper around multiple **ConstantBufferLayout** objects. It updates only those that need updating and only those that are not shared. You can swap different constant buffers (as long as they contain the same description). This allows for pre-creation all the necessary buffers.

### Interfaces and Constants

Each interface defines a set of constant associated with it. For example, a directional defines light’s colour and direction. This is the reason why it has to be provided at fixed shader parameters. Those constants are than set using **ShaderParameters** class. There is also an option to provide constants using **ConstantBufferLayout** class or to group parameters. For instance, all lights can be grouped in the same constants buffer.

The name of interface defined data consists of interface binding name, a dot, followed by the actual parameter. For example, “BDRF.diffuse” means that interface is bound to “BDRF” name and “diffuse” element is defined for it. Another example is “lights[2].direction”, where we use array accessor into pin array.

### Fixed Optimizations

There are some optimizations done for fixed variables. All evaluations that can be done at CPU are always performed there with the same precision as they would be performed on GPU.   
  
  
  
  
  
  
  
  
   
  
  
This DAG is optimized to one fixed providing operation if both two constants are fixed.

Output Pin 0

Input Pin 1

Input Pin 0

Add

Constant Operation 1

Constant Operation 0

Another important optimization that is performed is loop unrolling. All loops are unrolled if number of iterations can be “guessed” from pins and the iteration count is not too large (maximum loop unrolling can be used specified).

## Textures and Samplers

DAG also contains textures and samplers. **FixedShaderParameters** define name-index/format pairs for both textures and samplers. For example, if you use a 2D texture named “*myTexture*”, shader compilation will assign an index where it must be bound. The same rule applies for samplers; fixed shader parameters assign each sampler name an index.

*Texture and Samplers indices are always sequential – this means that index 0 will always be present if index 1 exists.*

## Metadata Programming

Metadata programming library is implemented on-top of DAG. This allows full DAG optimizations while making the *language* better suited for inside C# programming. The term metadata programming means that you write code with special classes. The code is not executed when this C# code is run but is only recorded and converted to DAG at that time. DAG is than compiled and code is executed as GPU shader.

### Pin Binder

A pin binder binds to pin. Each pin type has its own pin binder class, for example **Floatx1, Doublex3, Integer4x4** etc. They all give access to underlying pin as well as their DAG generator. All those classes are smart and provide different methods, properties and operators to define operations.

For instance, take a look at **Floatx1**’s *operator +*. Operator is overloaded for all possible pin binders that support such operation. Those are **Floatx1** and **Integerx1**. Besides that, it also allows **float** and **int** types. They are automatically converted to fixed pins.

When such operation or method is executed, DAG operation is added (in this case **AddOperation**) and appropriate pins are added as inputs to operation. A new pin binder is returned that represent the output pin. It can be used in further processes.

*Note that operator= is not overloaded. This means that references are copied. There is no explicit move instruction.*

Pin binders can be created using static constructors; **Floatx1.CreateFixed(float)** or **Float1.CreateConstant(string)**.

### DAG Generator

A DAG generator can provide pin binder for pins defined in **InputOperation** and also linking pin binders to **OutputOperation**. When you finish describing shader in metadata, you can extract the DAG you are working with and use it for your operations.

## Assembling

TODO:

## Debugging

TODO:

# State Objects

State object controls fixed states of pipeline. There are 4 state objects in SharpMedia Graphics Library:

* Sampler state – can be bound to any sampler index to any shader;
* Rasterization state – controls rasterization of primitives such as culling and orientation;
* Blend state – controls render target blending operations;
* Depth-stencil state – controls depth and stencil tests.

State objects are mutable and can live without graphics device. You can change them simply by setting them properties. Before applying them, you must *intern* them to state pool where they are also made immutable.

## State Pool

A **StatePool** is associated with **GraphicsDevice** when device exist. Until then, it can work correctly and interns all states when graphics device is first bound to pool. When state is interned, it is made sure that only one instance with such configuration exists. If such state already exists, existing is returned; otherwise we return the state that was interned.

When states are interned, they are made immutable. **StatePool** can also decide if driver part of state should be created. It is responsible for state caching – there may be hardware limitation on maximum number of state objects. **StatePool** must make sure there are not too many used at the same time.

**StatePool** is free to pre-create all states that are required.

# Graphics Device

A **GraphicsDevice** is a state machine. It is a wrapper around actual **Driver.IGraphicsDevice** and implements device sharing. This means that device can be shared between many processes. Even some resources can be shared. The only restriction is that you do not use the driver device at the same time and **GraphicsDevice** takes care of this using locking.

A device is locked when rendering is applied. There is no need for locking when resources are created, mapped or unmapped. A device supports hierarchal locking. This means that you can lock many times if you also unlock the same number of times.

Note that lock on graphics device *must not* last long. Some process may have a limited device lock time and if it is exceeded, special **LockTimeoutException** is thrown upon first access to states that require the device in locked state.

## Rendering State

Rendering state is controlled through **GraphicsDevice**. It consists of 3 programmable stages (vertex, pixel and geometry shader) and 3 configurable stages (blend, rasterization and depth-stencil). Programmable states are described with the actual shader, together with samplers, textures and constant buffers (they can be extracted from **ShaderParameters**). Vertex programmable also allows input geometry, geometry allows output geometry and pixel state requires render targets and depth-stencil target.

All states can be set using **Set\*** methods. We group all data relevant to specific stages to enable efficient validation. Properties and **Get\*** methods are provided to manipulate states.

The state of rendering won’t be affected if you unlock and relock it. This must be ensured otherwise someone could potentially access other processes’ resources.

## Rendering

Rendering is done using **Draw\*** calls. Stage validation is done on all draw calls. We support rendering using offsets, auto rendering (supported for simple geometry) and instanced rendering.

## TypelessTexture Sharing

We allow texture resource sharing. This is a very restricted process-process communication. One process can register a texture. It is the owner of this texture. Upon registration, a GUID[[17]](#footnote-18) is returned. This is to ensure security (one without GUID cannot access a texture because there are too many combinations). Only if you send a GUID to other process is it possible that this process gains access to texture.

Furthermore, process cannot write to texture if it is not an owner process. It can only read from texture and use it as a source operand for its own operations.

# Graphics Service

A **GraphicsService** class can be created from any graphics driver service.

Figure – A device creation process with side-effects.

## Explicit Mode

You can create **GraphicsDevice** is explicit mode. This means that you are the only one using the GPU. No other process[[18]](#footnote-19) can gain access to the device if you have obtained it in explicit mode. When you create device in explicit mode, you are also given a **SwapChain** and **Window**. Using swap chain, you can render to screen. Swap chain can be used as a render target. You must use **Present** method to show contents of swap chain.

Window can be used for event response tasks. This includes proper close, minimized, maximized, focused and so on event handling.

*Explicit mode should be used only for some specific applications that are guaranteed to run without any other graphical process.*

## Shared Mode

A shared mode allows processes to share the same device. Actually, they share the same driver part while the client side wrapper is not shared.

To enable shared mode, the device owner process must first create the device. It is also given control over **SwapChain** and **Window**. Not device owning processes cannot use this objects. This means that only one process can interact with window and render to screen.

After the device has been created in shared mode, other processes can freely obtain wrappers around the device. Synchronization between device wrappers is done using locking as described in Graphics Device section. This means that two processes cannot use device simultaneously.

The owner process is usually very intelligent and communicated with other processes that wish to render to screen. This is usually done through shared textures. The owner process simply copies portion of shared texture to swap chain (and applies some effects).

# How To

How To section answers some interesting and/or confusing aspects of working with graphics library.

# Questions

This is a questions section. Any relevant question sent to our e-mail or asked by any team members is placed here and answered.

1. VertexBuffer is a view of Buffer. [↑](#footnote-ref-2)
2. ConstantBuffer is a view of Buffer. More on this in following sections. [↑](#footnote-ref-3)
3. If buffers can reside in GPU only, serialization and deserialization will fail. [↑](#footnote-ref-4)
4. P means position, F3 means Float3 vector and N means normal. [↑](#footnote-ref-5)
5. More about pixel format in Textures section. [↑](#footnote-ref-6)
6. Do not confuse TypelessTexture with Texture; Typeless texture is generic data storage while Texture is shader binding object. [↑](#footnote-ref-7)
7. More about typeless format in PixelFormat section. [↑](#footnote-ref-8)
8. Unsigned uniform format – it is based on 8-bit unsigned integer format (range 0 – 255) that is uniformly mapped to [0,1] interval. [↑](#footnote-ref-9)
9. Signed uniform format – it is based on 8-bit signed integer format (range -128 – 127) that is uniformly mapped to [-1,1] interval. [↑](#footnote-ref-10)
10. R stands for Red, asociated with UI8 (unsigned 8-bit integer). G stands for green. All offsets are calculated from components. [↑](#footnote-ref-11)
11. Only InputOperation is never discarded because it must still exists. It describes the input layout. Note that garbage collection will delete all operations that use InputOperation's pins and do not contribute to output. [↑](#footnote-ref-12)
12. Bidirectional Reflection Function – describes the amount of light from given input direction and output ray direction for all three colour channels. [↑](#footnote-ref-13)
13. Albedo material is non-realistic and only provides colour not taking any input or output ray direction into account. [↑](#footnote-ref-14)
14. DAG is recompiled only if interface types change, not the data. This is described in Reusing Shaders section. [↑](#footnote-ref-15)
15. It is defined in Scene library. [↑](#footnote-ref-16)
16. Fixed values are constants that are specified at compile time, not use time. More about them in fixed and non-fixed parameters. [↑](#footnote-ref-17)
17. Global Unique Identifier. [↑](#footnote-ref-18)
18. Some processes, such as System Process, have rights to remove device from processes. [↑](#footnote-ref-19)