SharpMedia Graphics Design

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

Table Of Contents

[About 3](#_Toc166947994)

[Goals 3](#_Toc166947995)

[Deployment Structure 4](#_Toc166947996)

[Overview 4](#_Toc166947997)

[Namespace Structure 5](#_Toc166947998)

[*Namespace* Graphics 5](#_Toc166947999)

[*Class* AdapterFactory 5](#_Toc166948000)

[*Class* Device 6](#_Toc166948001)

[*Class* PixelFormat 6](#_Toc166948002)

[*Class* Image 7](#_Toc166948003)

[*Class* RenderTarget 7](#_Toc166948004)

[*Class* VertexFormat 7](#_Toc166948005)

[*Class* Vertices 8](#_Toc166948006)

[*Class* IndexFormat 8](#_Toc166948007)

[*Class* Indices 8](#_Toc166948008)

[Textures 8](#_Toc166948009)

[*Namespace* Driver 9](#_Toc166948010)

[Required Interfaces 9](#_Toc166948011)

[Extended Interfaces 9](#_Toc166948012)

[*Namespace* Shader 10](#_Toc166948013)

[DAG 10](#_Toc166948014)

[Metadata 14](#_Toc166948015)

[Debugging 14](#_Toc166948016)

[Assembling 15](#_Toc166948017)

[*Namespace* Image 17](#_Toc166948018)

[Image Reference 17](#_Toc166948019)

[Composition 17](#_Toc166948020)

[Effects 17](#_Toc166948021)

[Loaders 18](#_Toc166948022)

[*Namespace* GeneralPurpuse 19](#_Toc166948023)

[Kernel 19](#_Toc166948024)

[Arrays 19](#_Toc166948025)

[Reduction 19](#_Toc166948026)

[*Class* Algorithm 20](#_Toc166948027)

[*Namespace* Numerics 20](#_Toc166948028)

[Usage scenarios 21](#_Toc166948029)

[Creating a device 21](#_Toc166948030)

[Image manipulation and references 22](#_Toc166948031)

[Procedural images 23](#_Toc166948032)

[Textures and device binding 23](#_Toc166948033)

[MRT 23](#_Toc166948034)

[DAG Metadata Creation 23](#_Toc166948035)

[DAG Editor Creation 23](#_Toc166948036)

[DAG usage and specifications 23](#_Toc166948037)

[Registering a function to Assembler 23](#_Toc166948038)

[DAG to CIL conversion 23](#_Toc166948039)

[CIL to DAG conversion 24](#_Toc166948040)

[Implementation Notes 25](#_Toc166948041)

# About

Graphics library provides abstraction when performing operations using graphics hardware. The main aim is realistic real-time drawing. Note that graphics is not responsible for scene organization, culling, Z-order drawing, it only utilizes what the hardware allows. As such, it can be used for other (possibly non graphics) calculations, more widely known as general purpose GPU (GPGPU).

# Goals

Graphics library has many goals that it must maintain. Being one of the fundamental parts of SharpMedia, it must be designed with care.

One of the main problems of this library is that it should address features that are and also features that will come. The architecture must be extendable and must provide fallbacks when certain features are not available. To address those issues, graphics defines a great set of functionality that each driver must provide. This means that older graphics cards cannot run SharpMedia Graphics, but we think that they also couldn’t run it because of speed requirements. In some cases, abstraction cannot take into account all fallbacks, such as fallback when geometry shader (DX10 class feature) is not provided and higher level abstractions (in scene library or somewhere else) must address such issues by providing different algorithms for different capabilities.

Graphics must also allow runtime shader manipulation. As runtime information to code is useful to CLR, the abstraction of shader code is useful to graphics. We can perform optimizations on such code, analyze code, compose code from different sources or emit such code at runtime. Apart from that, shader system must provide some sort of dynamic “code” linking (as interfaces in C#) and variable number of elements in array (lists). This allows description of shaders in independent way of number of lights, animations, texturing and smart merging operations.

The goal of graphics is also performance. Because graphics hardware usually performs parallel tasks much faster than CPU, we want the best of graphics. This does not imply that graphics needs higher level algorithms but it means that it must use hardware whenever possible.

One of the serious drawbacks of graphics systems is that drivers must be loaded and initialized (sometimes, even window is required) before we can load any specific resources. SharpMedia Graphics system must overcome these limitations and allow loading of resources into RAM and fetching them to GPU’s memory when required or specifically asked to. For some simple operations, such as image manipulation, driver part may not even be required.

The system should also be stable by tracking events such as lost device and automatically recovering from invalid states. Driver system should allow various types of devices to be plugged, even software devices.

Unresolved issues:

* Resource caching

# Deployment Structure

Graphics library is distributed as SharpMedia library assembly. The driver, however, is a service and always lives inside SharpMedia OS Kernel.

# Overview

SharpMedia graphics is abstraction of graphics device together with many helper functionalities. It is built upon latest DirectX Graphics 10 arhitecture.

The graphics pipeline defined by graphics system is as follows:

Graphics let you costumize all shaders and inputs (textures, vertex/index buffers). It also provides software equivalents of textures (images) and abstraction of shaders.

# Namespace Structure

Basic namespace structure reflects components the graphics system is composed of. The system is highly sectionized so classes you need can be found quickly. Apart from that, system has a lot of links between each other; for example, Image system can use metadata programming to implement image filters (especially on texture images), GPGPU can also use a lot of shader functionality to work properly …

## *Namespace* Graphics

Namespace graphics provides base abstractions of graphics elements.

### *Class* GraphicsService

Graphics initialization is the step that must be taken in order to obtain a valid device. The following rules apply for graphics initialization:

* There is always only **one** active graphics device;
* Device can be **shared** between processes or owned by **one** process only (creator process);
* If device is shared, only one process (creator process) can use swap chain, manipulate OS window and present the contents of swap chain, all other processes can only use render targets.
* Only creator process can request reset of device (this fires events on all using device objects).

Initialization actually uses the **Driver.IGraphicsService** interface to initialize a device, or share one. The service is queried through OS functionality.

### *Class* Device

Device is a rendering API. It mimics (in many ways) the DirectX10 device interface but is a little bit more high level and more flexible. It can bind software resources to hardware on request or when first rendering using them is issued. Driver can create other resources.

Before device changes any state for rendering, it must be locked. After you stop working with device, you must unlock it. Runtime checks must be made to ensure such usage.

Device must expose all capabilities. If something cannot be done it should throw an exception.

You can obtain device using **GraphicsService**(if you are the first one using it) or through **Device.GetDefaultDevice()**. The later option is only available if device is already constructed and will share the device.

**All SharpMedia processes share the same driver device (e.g. a service) but each one has it's own Device object which contains it's state.**

#### Texture Sharing

Because processes need to affect the actual graphics and non-owner proccesses have no access to the actual swap chain, they cannot directly affect the rendering. To allow this, device sharing is implemented. A texture can be shared by many processes. The owner has full access to such render target. Other processes have **read-only** access to such render targets.

Such render target is created through **Device** class. Render targets can be queried by their asociated GUID[[1]](#footnote-2). When such render target is registered, unique ID is assigned (and such ID cannot be guessed, so there is no misusage). This ID can than be sent to other processes and those processes can obtain registered render target and access it inside Lock/Unlock blocks.

### *Class* PixelFormat

Pixel format is very extandable and is not limited to hardware supported formats only. All pixel formats are described by string description. Description is then parsed and pixel format is created from it. Common formats are also available (usually those supported by hardware) and they are detected from generic formats.

The format system allows accessing elements by their formats and offsets so algorithms can then be written quite easily.

This way, we can easily extract, isolate or manipulate only certain patterns inside image. In future, image formats will also be more widely supported by GPU and we could utilize full the power of this system.

Another important feature are typeless formats. Typeless formats can include not fully typed component formats, such as 32-bit value. This value can be (later in the process) interpret as float, signed integer, unsigned integer, uniform float (maps unsigned integer to range [0,1]) or signed uniform float (maps signed integer to range [-1,1]).

### *Class* VertexFormat

Vertex format allows very generic layout of elements. It is essentially almost the same as pixel format (it does not contain common formats). It allows per component format and offset. It is also described as a string that is than parsed.

Good think about this format is that hardware supports (almost) any type of format. Using such format, we can easily search for required components. All bindings in device are automatically mapped from vertex buffers based on vertex formats via **Geometry** class to shaders.

### *Class* IndexFormat

Index format describes the index buffer layout. Since only 16- or 32-bit indices are supported, the specification is fairly simple, one boolean can hold all information what kind of format we are using.

### Buffers

In SharpMedia, buffers are typeless. A buffer is a region of GPU memory with ability to be bound to the pipeline. It has restrictions about it's location, usage, binding points and CPU access. All buffers also implement a **IGraphicsLocality** interface that controls when (or if) resource is bound to GPU and if CPU copy is desirable.

Buffers can be (in general) bound to the following slots of pipeline:

* Geometry (index and vertex buffer);
* Constant buffer;
* Geometry-output;
* Texture;
* Render target.

Buffer is represented by **TypelessBuffer** class. At it's construction, you must specify all the usages, locality, CPU access and it's size in bytes. When you want to bind a buffer, you must create an appropriate view – make the buffer typed.

#### Vertex And Index Buffers

Vertex and Index buffers cannot be directly bound to pipeline; you must use the **Geometry** class. A geometry class allows typesafe binding of buffers with abilities to iterate through shapes and more. It abstract out the input layout issues; that is showing where each element is. **Geometry** also caches input layout.

A vertex buffer can be create using a **CreateVertexBuffer** method. A buffer is created with **VertexFormat** and optional offset from the beginning of typeless buffer. Index buffer is created similarly, with **IndexFormat** and optional offset.

#### Constant Buffers

Constant buffers are used to provide shaders with constant data, such as light colours, positions, animation data ... Constant buffers contain data as described by ShaderParameters. Using of constant buffers must always be asociated with actual shader parameters and their offsets and types.

#### Geometry Output

Geometry output stage is described by **Geometry** class. Only vertex buffers can be bound. Their are filled with apropriate data output by geometry shader.

#### Texture

A texture view is boundable as shader texture. Only 1D textures are supported. Buffer textures also cannot be sampled; only Load operation is permitted.

#### Render Target

Binding buffer as render target is sometimes desirable. This is the same as render to texture. Some restrictions may apply upon buffer format (especially floating point formats).

### Textures

In SharpMedia, the following textures exist:

* **TypelessTexture1D;**
* **TypelessTexture1DArray;**
* **TypelessTexture2D;**
* **TypelessTexture2DArray;**
* **TypelessTexture3D;**

The difference between buffers and textures is, that textures:

* Can contain mipmaps;
* Have dimensions and allow sampling instructions;
* Are not fully typeless; texel is fixed size and components have their fixed locations (but not formats).

For example, a typeless texture is of format »R8G8B8\_TYPELESS«, while buffer has no format. No that typeless texture can be viewed as a **Texture** or as a **RenderTarget.** No other options exist for textures. As with buffers, all usages must be enumerated at resource's creation.

When typeless texture is viewed as texture, it can specify the LOD[[2]](#footnote-3), typed format (»R8G8B8\_UINT«) and also subresource range (such as textures in array from 2-5).

## *Namespace* Driver

The driver namespace is collection of interfaces that driver must implement in order to be catagorized as SharpMedia Graphics Driver. A driver acts as a service which can either inject itself into the process (exclusive mode) or acts as a service that works with processess and provides some special treatment to owning process.

The central object is **IGraphicsService,** which is, as it's name suggests, a service. Using this object, we can create a **IGraphicsDevice** and asociated **ISwapChain** and **IWindowBackend**. The other way is to gain access to already craeted device (device sharing). The client side **GraphicsService** class actually wraps all of those classes into client-friendly classes and you should always use this class.

Through **IGraphicsDevice**, state of rendering can be controlled and resources (hardware) can be created. The API is preaty small and dumb. The actual client friendly classes are implemented on client side, where everything is cached (formats, size, state). Note that those classes are not strictly driver wrappers, but more (can be independant of wrappers, allow events ...).

## *Namespace* Shader

Shader abstraction is one of the foundamental advantages of SharpMedia system. We would like to use shaders for various tasks, such as:

* Rendering, material description must be light/animation/… independant;
* Image filters, compositing operations;
* Global illumination (radiosity, raytracing);
* Matrix system solvings, FFT, differential equations simulations;
* Other GPGPU algorithms.

With all those goals in mind, we have chosen a shader that can be either drawn by connecting shader operations with pins or programmed inside C# code using metadata language. The first is the option for artists creating (or modifying shaders) while the other option is for programmers using shaders as part of their GPGPU calculations. A more excotic option is using Assembler, which converts C# code (with many restrictions) into shader code. There is also an option that converts only efficient patters of C# code to GPU code (loops, etc.), resulting in mixed C#/GPU code. This allows a lot of future research for better efficiency.

### ShaderCode

ShaderCode is directed acyclic graph (DAG) and represents a family of shaders. As the acyclic states, there are no cycles in the graph. Each operation can have one or more inputs and one or more outputs. Operations are connected by means on typed pins. ShaderCode maps to family of either pixel, vertex or geometry shaders. Some operations are specific to different shaders.

Each DAG can begin with many operations that contain only outputs (constant, uniforms). There are only two special operations:

* **InputOperation** – all per element (per vertex, per pixel) attributes begin life as pins here;
* **OutputOperation** – results of DAG are written here.

Only operations that affect output operation are considered relavant to DAG and only those are executed, other are garbage collected (only backlinks are tracked by the system).

#### Operations

Each operation extends the ***IOperation*** interface. This interface specifies that operation can:

* Compile itself by means of ShaderCompiler;
* Describe it's inputs and outputs;
* Bind inputs, generate output pins (after all input pins are bound);
* Enumerate all inputs and outputs.

Operations represent some logical functionality, usually some sort of transformation. Because we wanted to represent shaders in animation/lighting/postrendering effects invariant way, we introduced the concepts of interfaces and dynamic arrays.

##### Arithmetic

Arithmetic operations are addition, multiplication, cross product of vectors, sinus, cosinus … Whenever possible, operations are defined for all pin types. For example, sinus is defined for vector types too and computes sinus of each component.

Most arithmetic operations map directly to shader instructions and are considered fast operations.

##### Texture Lookups

Texture lookups are provide for different texture types (1D, 2D, 3D, Cube, Array1D, Array2D). Mipmapping and other properties of sampler are controlled via samplers objects or manually. A texture can be accessed through samplers (using filtering and mipmapping) or through reading data from textures directly (loading).

##### If operation

If operation is essentially a flow operation but in DAG, it is not represented as such. It is an operation that is given 3 inputs:

* A boolean that is either true or false;
* A true branch value;
* A false branch value;

The result of operation is true branch pin if value is true and false branch value if pin is false. Note that DAG will optimize this operation is such way, that the whole instruction set that computed the value will not be executed if value is not needed:

public void Func(bool b, float y)

{

float x = Math.MathHelper.Sqrt2 \* y;

float z = y \* y \* y;

float r = b ? x : z;

}

will be converted to:

public void Func(bool b, float y)

{

float r;

if(b)

{

r = Math.MathHelper.Sqrt2 \* y;

} else {

r = y \* y \* y;

}

}

##### Flow

Flow controls are preaty slow on graphics, many require more than one cycle to execute and introduce additional penalty if neigbour shaders execute other branch, so we should avoid them whenever applicable.

Flow operations are also a little bit different because they can be cyclic (loops). We avoid loops by making flow operations parent operation to all operations that execute within it. Flow operations, such as loop statement, can emit special pins that can take many paths (are input and output) or go to different branches.

All elements within flow operation are isolated from other elements. Isolation means that there are no output pins that leave flow operation's region. The restriction does not apply for input pins which can be always linked to flow operation block.

Loop

Operation 3

Loop Pin 1 (in)

Loop Pin 1 (out)

Constant

Operation 4

Operation 0

Operation 2

Operation 1

#### Parameters

We know two types of parameters. The so called fixed parameters and DAG together define a hardware shader. Based on those, additional dynamic parameters must be specified (they can change and more of them may be created). Using all those, shader is fully defined.

Fixed parameters must provide all interfaces, dynamic arrays (size at least) and may fix any other parameter that can then be further optimized (loop unrolling to avoid dynamic branching, static if branching, constant precalculation …).

#### Interfaces

Interfaces are the dynamically pluggable elements. **Interfaces are pins, they actually hold data and code.**

When interface is bound, the operation can ask it to emit code (or some part of code) and to require additional uniform parameters. These parameters must be set by dynamic shader parameters.

Interface is defined by it's type, not the object. This means that the shader may be reused if interface changes, but type of interface does not. This also implies that interfaces must provide the same code for all scenario, only additional parameters may differ in value.

Interface operation can expose different operation types (e.g. multiple binding). For example, Lighting operation will expose ILight (array of lights, actually), IBRDF and IColourMixer interfaces. Each of interface types can support arrays.

#### Dynamic Arrays

Dynamic arrays allow array abstraction where the size of array is specified by fixed parameters. Some special operations are provided that work on dynamic arrays, such as ***AddArray***, ***IndexAtPosition*** and ***MulArray*** that translate the whole array into single pin. Dynamic array indexing is also provided. All such methods also work with static arrays.

### Metadata

Metadata programming can be especially useful because it maps directly to C#-like programming. This means that some special classes that represent data exist and they have overloaded operators. Most of the time, you can write shader code as if it was executing at the same time.

What is actually going on, the operations that are executed are translated to DAG. After the whole algorithm is written, we can compile it (with parameters) to hardware shader and execute it there.

### Debugging

Debugging is essential for testing of shaders as well as GPGPU programs. Because of our shader arhitecture using DAGs, we can analyze outputs incrementally, by executing a few operations and then checking if output is desired.

Debugging should use MRT (multi render targets) to execute as many passes in one step as possible. The results can than be viewed as:

* Graphical image, possibly for every operation;
* Numerical results, by transfering render target texture to RAM;
* Filtered results (minimum, maximum, other DAG based conditions).

### Assembling

We can assembly CIL code to DAGs and vice versa. This implies that any code that runs on GPU can also run on CPU but vice versa is not garantied due to conversion limitations (and GPU limits). Assembling to GPU code needs additional information that are provided through attributes.

**Note: Shaders can be debugged using C# debugger if written as methods and run with GPUEmulator (or EnableGPUEmulation flag on GraphicsDevice set to true).**

#### Specifications

Limitations are imposed only on CIL -> GPU conversion. The following must be true when compiling a method to DAG:

1. It has **GPUEntryPoint** attribute;
2. It is static;
3. It returns void;
4. All parameters have either **In**, **Out**, **Resource** or **Constant** attribute (only one of them);
5. All Out parameters must also be marked with out usage
6. The types used can be only:
   1. Built-in types: **Bool, Single, Double, Int32, Uint32**
   2. Vector types: **float, double, int, uint, bool** vectors of 2-4 dimensions.
   3. Rectangular matrix types : **Matrix2x2f**, **Matrix2x2d** ...
   4. **Texture**\*D and **Sampler** types, only as [Resource].
   5. Types extending **IInterface**, serving only as interface »references«. Can be only used as [**Constant**]
   6. All arrays of types presented above.
7. Calls only valid **GPUCallable**[[3]](#footnote-4) static methods (**InterfaceCall** and **GPUEntryPoint** included).

When converting to DAG, double precission is perserved, but conversion to shader is free to use only single precission floating numbers (since double precission are usually not available).

## *Namespace* Image

Image processing namespace allows various operations on images, be it hardware accelerated (textures) or not. Image namespace is provided for realtime usage (HDR – high dynamic range, composition of two textures, image analysis from webcam for head movement …) and also for offline/tools usage.

### Image Reference

Image reference is a *resource* reference. As such, it can point to any image, even on not-loaded images, or procedural (possibly non-constructed) images. More references can exist on the same image resource. Reference allows dynamic persistency to be tracked, recompution of certain images based on other images and entirelly remote access to resources (since they need not to exist in our RAM).

### Composition

Composition framework allows adding image(s) together using blending, filters or any other operation. When the image is hardware accelerated and resides in hardware memory, composition framework can detect that and make composition using shaders. Composition framework works on top of image references.

One of the most powerful features of composition library is that you can store an image as series of composition operations on some other images (you have file references on them) and the image can be reconstructed at runtime. The cost of loading an image from file is reduced and the cost of composition is usually relativelly low (especially if done using shaders). What is more, change to previous images will effect composed image too.

When resulting image is requested, code can be generated on the fly (either DAG or JIT code) and image everything can be processed very quickly. We expect the image to be generated faster then if it was transfered from filesystem.

#### Blending

Blending of images is operation that can combine image pixels based on certain constant or per pixel value, usually coded in alpha channel.

#### Custom operations

Custom operations can be formed as delegates. If you want hardware acceleration, the delegate must be either compatible with specifications that allow assembling them to DAGs or must provide a seperate DAG.

### Effects

Effects framework works on one image only and changes it in a way. Effects can also be hardware accelerated or they can work by emitting code that is then JIT-ed. Effects can also be applied during image composition.

#### HDR

High dynamic range effect transforms high dynamic range image to display compatible image. The HDR effect can have many parameters, such as mapping function (it may not be linear) and previous mapping parameters to apply effect as dynamic eye sensitive adjustment. At low intensities, you can combine HDR effect with grayscale effect (our eyes do not see colours at low intensities).

#### Grayscale

Grayscale effect converts RGB(A) image to grayscale image, usually also in RGB(A) format.

#### Sharpen

Sharpen effect sharpens edges on image.

#### Clamping

Image is clamped to range, e.g. not selected element exceeds the [min,max] range.

### Loaders

Most of the time, images are stored in hardware accelerated compression formats. When images reside on disc in our database system, they should be in raw form and the stream can be compressed to save some storage. However, we should also support lossy formats, such as jpg and png and different encoding schemes, such as RLE (run length encoding).

Loaders are also needed because many image formats will be imported. Loaders infrastructure must be dynamic.

## *Namespace* GeneralPurpuse

General purpuse computations are becoming more and more popular because of the increase of computation power and flexibility of pipeline. The main goal of to provide simple to use and simple to debug functionality for C# code that will be partly assembled to GPU code.

The **GPExecutor** will assemble whole programs, not just shaders. Inside those programs, you can do any type of manipulation. The core of computation should be done in specific kernels or GPU shaders themselves. This is the only part GPAssembler will replace when GPU device is available.

**GPExecutor** tries to to put almost no limitations on the code written. If the code is not convertible to GPU, it is run by CPU. This means that any program will run using **GPExecutor**. However, to use the **GraphicsDevice**, you must issue calls to shaders. A call can be a direct one (calling static method with **GPUEntryPoint** attribute) or through the **GPGPU** static class. The later gives you more freedom since you can configure the pipeline better but the first one is more elegant. The call through **GPGPU** also makes sure that it will not be emulated by CPU.

### *Class* CPUEmulationAttribute

This attribute can be appended before the method that is executable by **GPExecutor**. If **GPExecutor** is running is GPU emulation mode (CPU), this attribute states if this is permitted, how and in which situations it can perform emulation. The properties define if the method can be »changed« in order to allow multithreading (this is not desirable if debugging) and number of threads that can be used.

If the attribute is not present, the default (no threading) emulation is used.

### *Namespace* Numerics

Some numerical (sub)algorithms are implemented here. They can be useful in variety of other GPGPU algorithms and are designed to be efficiently mixed with other algorithms.

# Usage scenarios

## Creating a device

## Device locking

## Image manipulation and references

Image manipulation should be easy and references should allow very flexible usage:

public void Image()

{

Image image = new Texture2D(Usage.Default, TextureUsage.Texture,

PixelFormat.FromCommon(CommonPixelFormat.R8G8B8), 512, 512);

InitializeImage(image);

// Obtain a reference on this image.

ImageRef reference = new ImageRef(image);

// This reference can be sent via any channel and will point to

// resource in this AppDomain, copy is made when image is explicitly requested.

// We can also use this reference for compositor.

Compositor c = GetCompositor();

// Reference 2 points to non-existant resource.

ImageRef reference2 = c.Blend(reference, 0.5f);

// Refence 3 also point to non-existant resource.

ImageRef reference3 = c.Add(reference2, reference2);

// The image is calculated here, on demand, using a single pass (merged DAGs) on GPU

// pr by CPU if Device not available.

Image output = reference2.Image;

}

## Procedural images

## Textures and device binding

## MRT

## DAG Metadata Creation

## DAG Editor Creation

## DAG usage and specifications

## Registering a function to Assembler

## DAG to CIL conversion

## CIL to ShaderCode conversion

We can easily convert (compatible) C# code into DAG:

void CILToShaderCode()

{

DAG dag = ShaderAssembler.Assemble(

delegate([In(PinComponent.Position)] Vector4d position,

[Out(PinComponent.Position)] Vector4d outPosition,

[Constant] Math.Matrix.Matrix4x4d mvp)

{

outPosition = position \* mvp;

}

);

}

# Implementation Notes

Implementing graphics is hard because a lot of functionality must be included and the programmer must know hardware and usages of component.

1. Global Unique Identifier [↑](#footnote-ref-2)
2. Level Of Detail, e.g. which mipmaps can it use, 0 means base mipmap. [↑](#footnote-ref-3)
3. **GPUCallable** is any method that obeys **GPUEntryPoint** specifications without the need to mark parameters with attributes. [↑](#footnote-ref-4)