Investigating real-time rendering techniques approaching realism using the Vulkan API

Sandro Weber*
Technische Universitaet Muenchen
webers@in.tum.de

Lorenzo La Spina
Technische Universitaet Muenchen
lorenzo.la-spina@tum.de

October 20, 2016

Abstract

Modern Computer Graphics require heavy drawing routines to synthesize photo-realistic/artistic pictures of simulated 3D environments. Because of simulation and advanced effects rendering costs, only a small share of gamers, equipped with high-end machines, can enjoy the full set of visual features in modern video-games. Earlier this year, we've seen the birth of new, low-level, graphics APIs such as Vulkan and Direct3D 12. Using the aforementioned APIs, programmers can manually administer GPUs memory and exploit the capabilities of modern GPUs architectures. In this survey we try to understand if such APIs are able to make rendering faster, we make an implementation comparison between state-of-the art volume rendering in Direct3D 11 and Vulkan to investigate what happens under the hood of a modern and demanding drawing task, widely used to simulate atmospheric effects.

I. Introduction

In this survey, we want to understand whether a better usage of the GPU[1] and a smarter administration of its memory can lead to an improvement in frame rate of a compute intensive algorithm. To achieve this goal, we will compare a Direct3D11 with a Vulkan program, highlighting differences as we implement a volume ray tracer to simulate fog, using compute functionality available on the majority of modern GPU architectures.
Throughout this paper we will also show the fundamentals of Vulkan programming to better understand why new generation APIs offer better performance over classic APIs[2] such as Direct3D11 and OpenGL4.xx. The machine, on which the comparison has been made, uses an Nvidia GeForce GTX 970.

II. Background

In this section we will give some fundamental concepts explaining Vulkan philosophies and basic components.
Each part will be explained via simple examples of common *mistakes* or *best-practice* situations guiding the reader to a solution achieved by modern APIs techniques. Although we will present C++ code, the reader does not require an in-depth knowledge of the language to understand the key concepts behind the matter at hand. Let's start with a summary of how a Vulkan program works.
A typical Vulkan programs begins with platform-specific calls to open a windows or otherwise prepare a display device onto which the program will draw. Then, calls are made to open *queues* to which *command buffers* are submitted. The command buffer contain lists of commands which will be executed by the underlying hardware. The application can also allocate device memory, associate *resources* with memory and refer to these resources from within command buffers. Drawing commands cause application-defined shader programs to be invoked, which can consume the data in the resources and use them to produce images. To display resulting images, further platform-specific commands are made to transfer the resulting image to a display device or window.

*Project Supervisor
[1]Graphics Processing Unit
[2]Application Programming Interface

i. Philosophies

To the programmer, Vulkan is a set of commands that allow the specification of *shader programs*, *kernels*, data used by the kernels or shaders, and states controlling aspects of Vulkan outside the scope of shaders. Typically the data represents geometry in two or three dimensions and texture images, while the shaders and kernels control the processing of the data, rasterization of the geometry, and the lighting and shading of *fragments* generated by rasterization, resulting in the rendering of geometry into the framebuffer.
A keyword usually found in Vulkan context is **re-use**. The idea is to keep resources available on VRAM when needed to avoid useless PCI traffic spawned by the continuous load and unload operation generated by a draw call. For example, consider a program that renders five different objects using a shading technique for each of them, for the sake of the example, imagine that each shading technique uses a different texture and that those five textures are not collected in an atlas. Each time we instruct the context to execute a different shader program we generate a draw call, i.e. we bind programs and data to the graphics pipeline to actually draw the result on the currently bound frame buffer, submitting a series of draw commands. What we are interested in, at this point, is the **context switch**, during this phase, in the example above, we bind a different texture to the pipeline meaning that we probably invalidate previous texture memory and request new data to the host, this is likely to generate a memory transfer from host memory to GPU memory wasting cycles and resulting in poor performance.
The reason is that classic APIs don't have explicit control on what to keep in memory and what to discard, it is automatically inferred by the driver.

ii. Pipelines

A pipeline is controlled by a monolithic object created from a description of all of the shader stages and any relevant fixed-function stages. Linking the whole pipeline together allows for the optimization of shaders based on their input/outputs and eliminates expensive draw time state validation e.g. in the degenerate situation presented above. A pipeline object is bound to the device state in command buffers. Pipelines consist of shader stages, their resources and the pipeline layout. The pipeline layout describe the complete set of resources that can be accessed by a pipeline, more specifically, it represents the a sequence of *descriptor sets* with each having a specific layout. We will explain what descriptor sets and their layouts later, for now, let's just understand that this sequence of layouts is used to determine the interface between shader stages and shader resources.

iii. Command Buffers

Command Buffers are objects used to record commands which can be subsequently submitted to a device queue for execution.
Recorded commands include commands to bind pipelines and descriptor sets to the command buffer, commands to modify dynamic state, commands to draw (for graphics rendering), commands to dispatch (for compute kernels), commands to copy buffers and images, and other commands.
Unless specified otherwise, and without explicit synchronization, the various commands submitted to a queue via command buffers may execute in arbitrary order relative to each other, and/or concurrently. Also, the memory side-effects of those commands may not be directly visible to other commands without memory barriers. This is true within a command buffer, and across command buffers submitted to a given queue.

iv. Memory

The most important concept, to understand what we are doing, is *memory allocation*. Usually, in classic APIs, programmers have only the possibility to create resource views or uniform buffers using a few options just to describe the resource type. In Vulkan, memory

allocation is broken up into two categories, *host memory* and *device memory*.
Host memory is memory needed by the Vulkan implementation for non-device-visible storage. This storage may be used for e.g. internal software structures. Vulkan provides applications the opportunity to perform host memory allocations on behalf of the Vulkan implementation. This feature is not used in our code so we will only explain it briefly. In this case, the implementation perform its own memory allocations. Since most memory allocations are off the critical path, this is not meant as a performance feature.
Device memory is memory that is visible to the device, for example the contents of textures that can be natively used by the device, or uniform buffer objects that reside in on-device memory, as introduced in the example above. A keyword to remind when speaking of device memory is ***memory properties***. Memory properties of a physical device describe the memory heaps and memory types available on the board. Properties can be queried via physical devices API calls and the retrieved structure describes a number of *memory heaps* as well as a number of *memory types* (representing the type of resource e.g. textures, buffer, and the scope or visibility of this resource) that can be used to access memory allocated in those heaps. Each heap describes a memory resource of a particular size, and each memory type describes a set of memory properties (e.g. host cached vs uncached) that can be used with a given memory heap. Allocations using a particular memory type will consume resources form the heap indicated by that memory type's heap index. More than one memory type may share each heap, and the heaps and memory types provide a mechanism to advertise an accurate size of the physical memory resources while allowing the memory to be used with a variety of different properties.

v. Resources

Vulkan supports two primary resource types: *buffers* and *images*. Resources are views of memory with associated formatting and dimensionality. Buffers are essentially unformatted arrays of bytes whereas images contain format information, can be multidimensional and my have associated metadata. They are both accessed via *views* to determine how data should be interpreted.
Buffers are used for various purposes by binding them to a graphics or compute pipeline via descriptor sets or by directly specifying them as parameters to certain commands. For example vertex/index buffers bound to the input assembler of a graphics pipeline.
Images represent multidimensional (up to 3) arrays of data which can be used for various purposes (e.g. color/depth attachments, textures or, in our case, density data for a volume), by binding them to a graphics or compute pipeline via descriptor sets, or by directly specifying them as parameters to certain commands. For example an image blit operation to copy a portion of an image into another. Images have an additional peculiar functionality, *image layouts*. Images are stored in implementation-dependent opaque layouts in memory. Implementations may support several opaque layouts, and the layout used at any given time is determined by the image layout state of the image subresource. Each layout has limitations on what kinds of operations are supported for image subresources using the layout. Applications have control over which layout each image subresource uses, and can transition an image subresource from one layout to another. Transitions can happen with an image memory barrier, included as part of a pipeline barrier or a command buffer event, or as part of a subpass dependency within a render pass, as we will see how we use the ray tracer texture output as KHR surface for window display. The image layout is per-image subresource, and separate image subresources of the same image can be in different layouts at the same time with one exception, depth and stencil aspects of a given image subresource must always be in the same layout.
To actually read image data in an implementation, the programmer has two different alterna-

tives. We can use a *sampler* or a raw *imageLoad*. The latter is a pretty straightforward operation, given a 2D or 3D coordinate, the pixel value is returned. A sampler, still requires a coordinate but, this coordinate may not correspond to an unsigned integer defining a position in an array of pixels, in fact this coordinate is usually a floating point because samplers apply filtering and other transformations for the shader. A quick example is the type of interpolation used to retrieve a color value from a diffuse texture in a shader. We will discuss this matter in detail when we will describe the implementation details of the ray tracer.

vi. Resource Descriptors

Finally, we have all the elements to talk about *descriptors*. Shaders access buffer and image resources by using special shader variables which are indirectly bound to buffer and image views via the API. These variables are organized into sets, where each set of bindings is represented by a *descriptor set* object in the API and a descriptor set is bound all at once. A *descriptor* is an opaque data structure representing a shader resource such as a buffer view, image view, sampler or combined image sampler. The content of each set is determined by its *descriptor set layout* and, the sequence of set layouts that can be used by resource variables in shaders within a pipeline, is specified in a *pipeline layout*. Each shader can use up to a device specified value of descriptor sets, each descriptor set can include bindings for descriptors of all descriptor types. Each shader resource variable is assigned to a tuple of (set numbers, binding number, array element) that defines its location within a descriptor set layout. In GLSL, the set number and binding number are assigned via layout qualifiers, and the array element is implicitly assigned consecutively starting with index equal zero for the first element of an array (and array element is zero for non-array variables).

vii. Shaders

A shader specifies programmable operations that execute for each vertes, control point, tessellated vertex, primitive, fragment, or workgroup in corresponding stage(s) of the graphics and compute pipelines. Graphics pipelines include vertex shader execution as a result of primitive assembly, followed, if enabled, by tessellation control and evaluation shaders operating on patches, geometry shaders, if enabled, operating on primitives, and fragment shaders, if present, operating on fragments generated by rasterization. From now on we will refer to vertex, tessellation control, tessellation evaluation and geometry shaders as *vertex processing stages* as they occure in the logical pipeline before rasterization. Only compute shader stage is included in a compute pipeline. Compute shaders operate on compute invocations in a workgroup.
Shaders can read from input variables, and read from or and write to output variables. Input and output variables can be used to transfer data between shader stages, or to allow the shader to interact with values that exist in the execution environment. Similarly, the execution environment provides constants that describe capabilities. Shader variables are associated with execution environment-provided inputs and outputs using *built-in* decorations in the shader.
At each stage of the pipeline, multiple invocations of a shader may execute simultaneously. Further, invocations of a single shader produced as the result of different commands may execute simultaneously. The relative execution order of invocations of the same shader type is undefined. Shader invocations may complete in a different order than that in which the primitives they originated from were drawn or dispatched by the application. However fragment shader outputs are written to attachments in rasterization order. The relative order of invocations of different shader types is largely undefined. However, when invoking a shader whose inputs are generated from a previous pipeline stage, the

shader invocations from the previous stage are **guaranteed** to have executed far enough to generate input values for all required inputs.

We now have all the basic elements needed to go through the implementation details.

III. Implementation

In this section we will examine the most critical, important and interesting details of the volume ray tracing implementation in Vulkan. We will first talk about API initialization and swapchain creation, then we will apply the knowledge acquired in the previous section to create and bind the required resources to our pipelines. After that we will examine shader programs with a little insight on the used ray casting algorithm and the key innovations over classic approaches. Finally we will compare the two generations of APIs highlighting important differences and peculiarity in the program logic.

i. Initialization

Vulkan initialization is a little bit more involved that for classic APIs. The real difference is not the initialization process per se, but in the level of detail of the information that the programmer must provide to initialize devices and devices context. Device and device context are structures that graphics APIs usually expose, they have 2 different and distinct roles:

Device used for device memory management. Functionalities include allocation, release and others.

Device context used to submit rendering commands to the pipeline. Functionalities include setting shaders for a specific material, setting textures to be used in the pipeline and others.

Vulkan doesn't explicitly offer such structures because, as shown in the previous section, Vulkan has a different execution model. Still it provide two different abstractions of a device:

- vkPhysicalDevice
- vkDevice

vkPhysicalDevice represents the physical layer, it offers functions to query hardware properties, memory heaps and their limits. *vkDevice* is an abstraction that allows for device memory management as they represent logical connections to physical devices. It is trivial that, in order to create a device that is able to allocate and release resources in our application, we will query hardware capacities retrieving all needed information about queues, memory heaps and their available types and finally we will use the Vulkan instance to create a *vkDevice*. Vulkan doesn't have a global state, all per-application data is stored in a *VkInstance* object. Creating such object initializes the Vulkan library and allows the application to pass information about itself to the implementation.
Once devices have all been initialized we have access to their queues. An important property is the *queue family index*, it is used in Vulkan in order to tie operations to a specific family of queues. There are three key situations where this index is required:

- When creating a *command pool*, a queue family index is specified so that *command buffers* allocated from this pool can only be submitted on queues corresponding to this family queue.
- When creating *image* and *buffer* resources, a set of queue families is included in their initialization structures to specify the queue families that can access the resource.
- When inserting *buffer memory barriers* or *image memory barriers* a source and destination queue family index is specified to allow ownership of a buffer or image to be transferred from one queue family to another.

Next task in a Vulkan program initialization is the initialization of a *swapchain*. Swapchains are structures in charge of keeping a collection of framebuffers and presenting them to the windowing system, if a swapchain holds on to two buffers and always updates the least frequently used we say that our system uses *double buffering*. Swapchain creation is essentially made of two operations, query the device queue for supported image formats and, finally, swapchain and framebuffers allocation. Present operations are explicitly synchronized using semaphores that signal on render complete and on present complete to implement the framebuffer swap logic. Framebuffer allocation requires a step in between, to initialize a framebuffer we first need to define a *render pass*. A render pass represents a collection of attachments, subpasses, and dependencies between the subpasses, and describes how the attachments are used over the course of the subpasses.

Attachment describes the properties of an attachment including its format, sample count, and how its contents are treated at the beginning and end of each render pass instance.

Subpass represents a phase of rendering that reads and writes a subset of the attachments in a render pass. Rendering commands are recorded into a particular subpass of a render pass instance.

Subpass description describes the subset of attachments that is involved in the execution of a subpass. Each subpass can read from some attachments as *input attachments*, write to some as *color attachments* or *depth/stencil attachments*. A subpass description can also include a set of *preserve attachments*, which are attachments that are not read or written by the subpass but whose contents must be preserved throughout the subpass.

The subpasses in a render pass all render to the same dimensions, and fragments for pixel (x, y, layer) in one subpass can only read attachment contents written by previous subpasses at the same (x, y, layer) location.
It is worth noting that, by describing a complete set of subpasses in advance, render passes provide the implementation an opportunity to optimize the storage and transfer of an attachment data between subpasses. In practice, this means that our subpasses with a simple framebuffer-space dependency may be merged into a single tiled rendering pass, keeping the attachment data on-chip for the duration of a render pass instance. However, it is also quite common for a render pass to only contain a single subpass.

ii. Implementation Flow

Now that we have created and allocated all underlying structures of our Vulkan implementation, we can describe the steps that the application will do in order to render a picture. The application has two pipelines, specifically one compute pipeline to do ray tracing computations and another to show the result on screen. The compute pipeline has a texture buffer for result output, a sampler 3D for volume reads and a uniform buffer for implementation data.

```
struct uniformBuffer
{
        // view matrix for camera
        mat4 camView;
        // unsigned extent of volume
        ivec3 volumeExtent;
};
```

We would like to bind these objects to the a compute pipeline. As per subsections ii, v and vi we need to create and allocate resources, allocate a descriptor pool, define descriptor set layout and finally create a descriptor set. Allocating a descriptor pool is a critical operation, by allocating a pool we tell the API the maximum number of requested descriptors per type. Descriptor pool allocation requires a list of *VkDescriptorPoolSize* structures, each containing a descriptor type and the number of maximum requested descriptors of that type. Allocation of the pool is done via a logical device call. Note that if the implementation requests more descriptors than maximum available, it will return an error. In order to create a descriptor set layout a collection of *VkDescriptorSetLayoutBinding* structures must be allocated and filled. Relevant information contained is this structure are:

Descriptor Type describing the type of the descriptor, in our case, the compute pipeline will need *storage image*, *uniform buffer*, *image sampler*.

Shader stage flag describing the pipeline stage where the resource will be accessed. According to ii, we can only specify *compute*.

Binding describing the bind point of the resource in the shader program.

Descriptor Count describing the number of descriptors contained in the binding, accessed in a shader as an array. For example, if the count is zero this binding entry is reserved and the resource must not be accessed from any stage via this binding within any pipeline using the set layout.

With these data in place, we can create descriptor set layout, pipeline layout and descriptor set via logical device calls. The implementation is still unaware of the actual data we need to use in the shader program, it is possible to link actual data to a shader program using *VkWriteDescriptorSet* structures. Each *VkWriteDescriptorSet* structure contains again the descriptor type, the binding of the resource and a pointer to the descriptor itself. Data is linked via a logical device call.
Pipeline allocation requires a pointer to a list of shader modules, in this case just the compute shader, and a pointer to a pipeline layout, again, a logical device call creates the pipeline object. The second part of the rendering loop uses a fullscreen quad and just renders the output texture on this quad, the process of resources and pipeline allocation is similar to the one described so far for the compute shader and we will not discuss it further.
In subsection iii we mentioned that rendering commands are submitted to a queue via its registration to a command buffer, this is also valid for compute shader dispatch operation. The last critical step is rendering synchronization, we want to make sure that the compute stage is finished rendering before the graphics pipeline asks for the texture output. To ensure this, we create a *signaled fence* and, for each frame, we reset it after every signal for the next frame.

```
// Swap buffer
mSwapchain.GetNextImage(mSemaphores.presentComplete,
    &mCurrentBuffer);

// Command Buffer to be
// submitted to the queue
mSubmitInfo.commandBufferCount = 1;
mSubmitInfo.pCommandBuffers =
    &mDrawBuffers[mCurrentBuffer];

// Submit graphics pipeline to draw frame
mMainDevice->QueueSubmit(QUEUE_TYPE_GRAPHICS,
    1, mSubmitInfo, VK_NULL_HANDLE);

// Wait for graphics pipeline to finish
    render
mSwapchain.QueuePresent(mCurrentBuffer,
    mSemaphores.renderComplete);
mMainDevice->QueueWaitIdle(QUEUE_TYPE_GRAPHICS);

// Compute synchronization and fence reset
mMainDevice->QueueWaitForFence(mCompute.fence);
mMainDevice->ResetFences(&mCompute.fence,
    1);

// Compute
VkSubmitInfo computeSubmitInfo = {};
computeSubmitInfo.sType =
    VK_STRUCTURE_TYPE_SUBMIT_INFO;
computeSubmitInfo.commandBufferCount = 1;
computeSubmitInfo.pCommandBuffers =
    &mCompute.cmdBuffer;

// Call compute pipeline
mMainDevice->QueueSubmit(QUEUE_TYPE_COMPUTE,
    1, computeSubmitInfo, mCompute.fence);
```

All things we've seen so far in the current section would not have been necessary using classic APIs. Whenever the implementation requests memory allocation, the driver chooses the best allocation he sees fit based on the request, rendering all steps above unnecessary. This is in general a bad thing because the driver could assume a different usage of a resource and end up with slowing down the application because of subsequent requests of data generating unnecessary traffic.

iii. Volume Ray Casting Notes

We will give a brief introduction to volume ray casting. Volume ray casting, is an image-based volume rendering technique. It computes 2D images from 3D volumetric data sets (3D scalar fields). Volume ray casting, which processes volume data, must not be mistaken with ray casting in the sense used in ray tracing, which processes surface data. In the volumetric variant, the computation doesn't stop at the surface but "pushes through" the object, sampling the object along the ray. Unlike ray tracing, volume ray casting does not spawn secondary rays.
Our basic fog simulator based on volume ray casting is made of three steps:

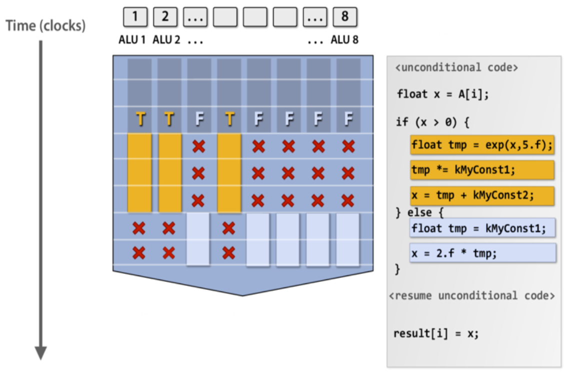
1. **Ray Casting**. For each pixel of the image plane, a ray is shot through the volume. In our case, the volume is "tiled" through an infinite space, so we don't need intersections test with bounding volumes which is usually necessary in order to avoid wasting ray steps in empty space areas.

2. **Sampling**. Along the part of the ray of sight that lies within the volume, equidistant *sampling points* or *samples* are selected. In general, the volume is not aligned with the ray of sight, and sampling points will usually be located in between voxels. Because of that, it is necessary to interpolate the values of the samples from its surrounding voxels (commonly used trilinear interpolation).

3. **Compositing**. While sampling the volume along the ray, the density values are composited using the *Over* operator proposed by [Porter, Duff et al. 1984], resulting in the final color value for the pixel that is currently being processed. The ray is marched in a front-to-back fashion allowing early ray termination when accumulated density is ≥ 1.0

iv. Shader Notes

In this section we will present the pseudo-code of our fog simulator, what we are interested in here is a curious compiler optimization difference between HLSL and GLSL to SPIR-V compiler, in our case, this difference is critical and reveals to be a major bottleneck in the final result. In Vulkan, shader modules require shader programs and compute kernels into the *SPIR-V* format. SPIR-V is a binary intermediate language for shader programs and compute kernels, shader programs and compute kernels can be pre-compiled or compiled at runtime using a third party library. To understand the matter at hand we need to introduce the execution model of a GPU. GPU architectures are called *SIMT*, SIMT stands for **S**ingle **I**nstruction **M**ultiple **T**hread, that is the reason why we say GPUs execute the code in *lockstep*, each thread execute the same instruction at the same time. It is trivial that, in such systems, branching is a really expensive operation and, if not optimized by the compiler and/or the programmer, can lead to a massive loss of performance. Graphics APIs offer a simple



```
<unconditional code>

float x = A[i];

if (x > 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Figure 1: *Branching in lockstep execution. Worst case scenario yields 1/8 of total performance.*

interface to run compute kernels, users only need to specify the size of a work group. Inside the kernel, global variables for thread identification are available so we can easily calculate the pixel coordinate from local thread id. In general, available variables are:

Num Threads (x, y, z) Defines the number of threads to be executed in a single thread group when a compute shader is dispatched. The x, y and z values indicate the size of the thread group in a particular direction and the total of x*y*z gives the number of threads in the group. The ability to specify the size of the thread group across three dimensions allows individual threads to be accessed in a manner that can be logically mapped to 2D and 3D data structures.

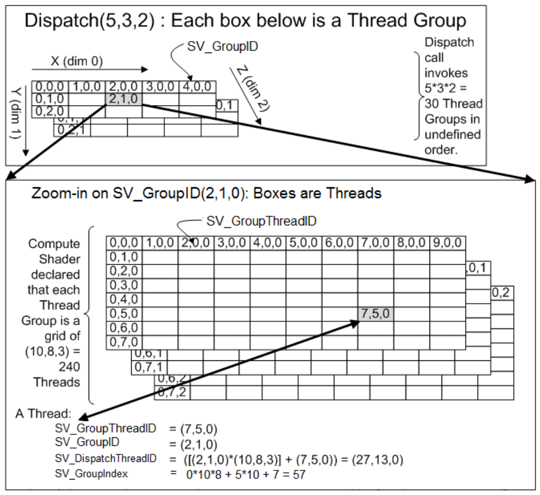
Group Thread ID Indices for which an individual thread within a thread group are bound to a particular instance of a compute shader. *Group Thread ID* varies across the range specified for the compute shader in the *numthreads* attribute.

Group Index The "flattened" index of a compute shader thread group, which turns the multi-dimensional *Group Thread ID* into a 1D value. *Group Index* varies from 0 to (numThreadsX * numThreadsY * numThreadsZ) - 1.

Group ID Indices for which thread group a compute shader is executing in. The indices are the whole group and not an individual thread. Possible values vary across the range passed as parameter to *Dispatch*.

Dispatch Thread ID Indices for which combined thread and thread group a compute shader is executing in. *Dispatch Thread ID* is the sum of *Group ID* * *numthreads* and *Group Thread ID*. It varies across the range specified in *Dispatch* and *numthreads*.

Let us sketch the fog simulator code:



Dispatch(5,3,2) : Each box below is a Thread Group
SV_GroupID
X (dim 0)
Y (dim 1)
Z (dim 2)
0,0,0
1,0,0
2,0,0
3,0,0
4,0,0
0,1,0
2,1,0
0,2,0
0,1
0,2,1
Dispatch call invokes 5*3*2 = 30 Thread Groups in undefined order.
Zoom-in on SV_GroupID(2,1,0): Boxes are Threads
SV_GroupThreadID
Compute Shader declared that each Thread Group is a grid of (10,8,3) = 240 Threads
0,0,0
1,0,0
2,0,0
3,0,0
4,0,0
5,0,0
6,0,0
7,0,0
8,0,0
9,0,0
0,1,0
0,2,0
0,3,0
0,4,0
0,5,0
0,6,0
0,7,0
7,5,0
0,1
0,2
0,6,1
0,7,1
0,6,2
0,7,2
A Thread:
SV_GroupThreadID = (7,5,0)
SV_GroupID = (2,1,0)
SV_DispatchThreadID = ([(2,1,0)*(10,8,3)] + (7,5,0)) = (27,13,0)
SV_GroupIndex = 0*10*8 + 5*10 + 7 = 57

Figure 2: *Example Dispatch showing the relationship between attributes and system values.*

```
float RescaleNAdd(...)
{
    // Used as an optimization,
    // instead of adding up all octaves
         for each step,
    // lerp the distance to the eye and
         decrease
    // octaves sampled
    uint octaves =
         OctavesFromDistance(ray);

    // ...

    float density = 0.0f;
    for(uint oc = 1; oc < octaves; oc++)
    {
        density += sampleVolume(...);
    }

    return density;
}

void main()
{
    // Compute image plane pixel position
         in world space
    // using attributes and system values
    // ...

    // Initialize Ray
    // ...

    // Ray marching loop
    for(uint i = 0; i < maxRaySteps; i++)
    {
        float d = RescaleNAdd(...);

        finalColor = finalAlpha *
             finalColor + (1.0f -
            finalAlpha) * d * fogColor;
        finalAlpha = finalAlpha + (1.0f -
            finalAlpha) * d;

        if(finalAlpha >= 1.0f)
            break;

        ray.pos += ray.direction * step;
    }

    outputTex[DTid.xy] =
        float4(finalColor, 1.0f);
}
```

Our optimization of the number of octaves is justified by the fact that, the further we are from the eye the lower is the resolution required. Although this calculation optimizes the code in D3D11, it results in a bottleneck in the GLSL implementation. It is obvious that, if the number of iterations is known at compile time, the compiler can automatically unroll the loop removing branching operations from the code. At this time, is not completely clear the reason why it happens. Maintainers of Vulkan and SPIR-V could not provide a concrete and complete answer. Therefore, we can only conclude that, at this time, some kernel programs might perform better on D3D11 than SPIR-V.

IV. Results

In this section we will benchmark our volume ray caster. We used two different systems for the performance evaluation. An MSI gaming laptop GS60 4k Intel i7-4720HQ@2.6GHz equipped with an Nvidia GTX 970M 3Gb VRAM, the second is a desktop machine with two Intel Xeon 2.53GHz and an Nvidia GTX 970 4Gb VRAM.
Table 1 shows that volume ray casting compiled as SPIR-V is slower than HLSL. Although results are clear we would like to present a second benchmark to have more data to analyse. Because results of the volume ray casting are astonishing, we wanted to give it another chance with a simpler and standard algorithm that can be found in literature, ray tracing. In this algorithm we use a bunch of spheres and

System			
CPU	GPU	API	Frame avg.
i7-4720HQ@2.6GHz	GTX 970M	Vulkan	500*ms*
2x Xeon e5630@2.53GHz	GTX 970	Vulkan	450*ms*
i7-4720HQ@2.6GHz	GTX 970M	D3D11	25*ms*
2x Xeon e5630@2.53GHz	GTX 970	D3D11	20*ms*

Table 1: *Volume ray caster benchmark*

System			
CPU	GPU	API	Frame avg.
i7-4720HQ@2.6GHz	GTX 970M	Vulkan	0.4193*ms*
2x Xeon e5630@2.53GHz	GTX 970	Vulkan	0.3353*ms*
i7-4720HQ@2.6GHz	GTX 970M	D3D11	2.11*ms*
2x Xeon e5630@2.53GHz	GTX 970	D3D11	1.23*ms*

Table 2: *Simple ray tracing benchmark*

trace diffuse, specular and shadow rays, results are summed in Table 2. This time we have a much better results, let's try to understand what happens there. We expect that the dynamic number of octaves is the cause of the high frame time in the application so, to validate our hypothesis, we enforced the maximum number of octaves yielding 500ms to render on D3D11. We can safely assume that D3D11 shader compiler optimize the code in a better manner than the other.
Let us now examine ray tracing performance, this shader does not contain dynamic loops and, as data suggests, SPIR-V is much faster than in the other experiment.

V. Conclusions

HLSL performance seems more consistent, a reason why, could be that D3D11 and HLSL are much more tightly coupled than Vulkan and SPIR-V, also, the latter is still a pretty new technology with a still non-definitive standard. During our journey through graphics APIs we have implemented other algorithm and, even though it is not the matter at hand in this survey, polygon performance is much better in

Vulkan. Furthermore, heavy polygon drawing routines involving around 300k triangles, multi-texturing and complex lighting can be rendered at a stable 0.3ms per frame. We are very excited about this new technology and we strongly believe that, in the future, gaming industry will adopt it as a standard.

References

[Real Time Rendering, 2008] Real Time Rendering, Third Edition, T. Akenine-Moeller, E. Haines, N. Hoffman. (2008). Standard textbook collecting modern approaches to computer graphics. *Computer Science*, ISBN 987-1-56881-424-7.

[Vulkan Specification, 2016] Vulkan 1.0.26, A Specification, Khronos Group Inc. (2016).

[GLSL Specification, 2016] GLSL 4.50, A Specification, Khronos Group Inc. (2016).

[Direct3D Documentation, 2010] Direct3D 11 Documentation, Microsoft. (2010).