Universitat Autònoma de Barcelona

Dipòsit digital de documents de la UAB

This is the **published version** of the bachelor thesis:

Montoto González, Manuel; Moure Lopez, Juan Carlos, dir. Hands-on study on Vulkan and the hardware ray-tracing extensions. 2021. (958 Enginyeria Informàtica)

This version is available at https://ddd.uab.cat/record/248457

under the terms of the license

Hands-on study on Vulkan and the hardware ray-tracing extensions

Manuel Montoto González

Abstract– The new GPUs from Nvidia and AMD include hardware ray-tracing acceleration units that enable regular consumer computers to be capable of drawing 3D scenes in a more realistic way than by simple rasterization. Rasterization is limited among other things by the fact that drawing is always made in the local context of every object in the scene, while ray-tracing is performed on the whole scene. The access to these hardware ray-tracing units by the programmer is by means of the new ”RTX” extensions that have been released by the Khronos Group for their Vulkan API, the lower level successor to the industry-standard OpenGL. So it may be time to bite the bullet, leave the comfort of OpenGL behind, and start learning the API of it's successor and how it engages with the new RTX extensions. This text attempts to give you some general guidance to aid in learning the Vulkan API, a small introduction in how ray-tracing works, how this new hardware handles it, and the new type of shaders required to be supplied to the GPU for the calculations and drawing. Then, finally, a small ray-tracer is put together in order to create some 3D scenes showing some of the interesting new capabilities that are difficult or impossible to achieve by means of rasterization: Shadows, reflections, and refraction of the light rays, along with measuring and providing the frames per second achieved with the employed hardware configuration while drawing each of them.

Keywords– Vulkan, OpenGL, RTX, Ray-tracing, Rasterization, GLSL, Shaders, Nvidia, AMD, Intel, Khronos Group.

1 Introduction and Context

Hardware acceleration of tasks has been going on for decades, with the CPU acting as an orchestra director for all of the coprocessors. The 1985 Commodore Amiga is a prime example on this, where it took the IBM PC ten years to start getting close to it. From basic math functions to complex video decoding, in the end it's usually better to free the CPU from tasks not well suited for generic processing. In the 90s the PC architecture took the world by storm by being cheaper by doing nearly everything in software thanks to riding the, at the moment, ever increasing CPU speed wave, but when CPUs started hitting the wall of the 4.0 ghz frontier focus on hardware acceleration got back. Nowadays we have really interesting examples of coprocessors for many tasks and even a new computer architecture by Apple trouncing the speed of their last year Intel computers with an ARM CPU with integrated AI and video processing capabilities integrated in the silicon.

Among the many examples on hardware acceleration of the past two decades undoubtely the stars of the show have been the GPUs. Cheap computers of today exercise the power of past super computers and generation after generation the architectures have been streamlined, optimized and super charged with new abilities.

Now, the rage is all on the hardware acceleration of ray-tracing. Nvidia is leading the way but AMD is getting closer. The first hardware with modern ray-tracing capabilities was released by Nvidia in 2019, but it's been in 2020 with the Nvidia Ampere RTX 30 series and AMD Radeon RX 6000 series generation and the release of the Sony PlayStation 5 and Microsoft Series X video game consoles that ray-tracing has been kickstarted to the mainstream adoption.

1.1 Rasterization vs ray-tracing, in hardware

The traditional process of drawing on screen is called “rasterization”, while the object of this work is “ray-tracing”. Drawing, in both techniques, builds on top of the concept of “shaders”, small routines delegated to be processed by the GPU instead of the CPU, usually written in GLSL and HLSL languages that are similar to C and C++, that get some standard input in variables with standarized names and then output a result in some other variable or variables with standard names too. Vertex shaders get a vertex as input and expect a (usually processed) vertex as an output. Pixel shaders get the values of the three vertices from the current triangle being drawn interpolated to the current position inside the triangle and it's expected to output a RGBA value to be stored in the image buffer.

1.1.1 Rasterization

In rasterization the objects from a 3D scene are processed one by one, vertices projected onto the camera's 2D plane by using vertex shaders, code written by the programmer to be executed on the GPU, pixels drawn into a buffer by using fragment shaders (in OpenGL / Vulkan, pixel shaders in Direct 3D terminology). The order of drawing into the buffer is pretty random, so the same pixel may be overwritten many times until the final image is done. To avoid sorting problems with objects projected onto the same pixels, and get a speed boost by culling the draw call if it's not going to affect the final image, a distance from camera buffer is used, the "z-buffer".

Nowadays thanks to the powerful GPUs and a variety of ever improving techniques, rasterization can give very convincing and quality results, but still falls short due to the implicit limitations with lightning and how light interacts between objects on the scene.

An important detail in rasterization is that objects are drawn in isolation from the other objects. Scene data is on the CPU memory / side, and while you could technically send this information to the GPU for the shaders consumption, in practice this is not feasible due to, among other factors, the small memory available (a few kilobytes, usually).

1.1.2 Brief introduction to ray-tracing

On the other side, ray-tracing is, interestingly, a simpler concept. Instead of iterating through the objects drawing them, the GPU goes directly to iterate all over the final display surface pixels by creating "rays" that enter the scene, simulating the path back from the sources to the camera. The scene is uploaded in full to the GPU memory and each of these rays check intersections with all the scene objects and execute new types of shaders in case of no collision (so as to draw a background, for example), in the case of any surface intersection, and in the case of the intersection point that lies the nearest to the camera.

The programmer can leverage this information when writing the intersection shaders to direct the GPU to colorize the buffer pixels as required, even by instancing new rays to get more information from the scene by knowing where the light hitting the current point was bouncing from.

To try to make this concept clearer let's check the drawing at figure (1) . Here we can see rays being generated at the left going to the right.

Some of the rays intersect an object. The surface normal is taken in consideration to calculate a new exiting ray that is symmetrical to the arriving ray with the normal axis. Some of these rays go to the infinity now. One ray is hitting a different object. Then, again, a new ray is calculated then reaches, finally, the infinity.

Rays return backwards reporting a color. Then the intersection shader uses this information and the surface properties to calculate a color value. This doesn't have to be

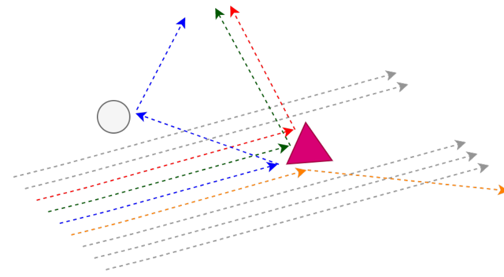

Fig. 1: Rays bounce on objects, or miss and reach the infinite

photorealistic at all. The programmer is free to do whatever is necessary with this information. It's worth noticing here that this is only possible by means of having the full scene information available and allows for truly mind blowing capabilities that are near to impossible to achieve by rasterization, such as semi transparent objects, true reflections, "free" realistic shadows and much more. Each technique may be implemented in simple ways for basic results:

- Rays that intersect a light source can return a bright color to the exit point of a surface. Rays not intersecting a light source can return a dark color, black or a base level color acting as the diffuse light level. In this way the renderer is already achieving "free" realistic shadows.

- Materials are simulated by altering how the rays bounce and what is done with the returned values. Metallic surfaces or mirrors get to carry a lot of the returning color values. Non reflective surfaces will mainly ignore the arriving color and returning theirs.

- Rays that reach a non-opaque object can get through it but it's path and color values be altered to simulate semi transparency and light refraction.

- Rays reaching the infinitum may return a background image, create a design pattern by means of the coordinate values, a simple fill color, etc.

One could say the only disadvantadge is the stagering amount of computing power required, but that is now starting to be addressed in the new consumer hardware, and no doubt with each new generation the number of rays per seconds will only get higher.

2 Goals of this work

The main goals are understanding how Vulkan works and getting a Vulkan code base up and working; initializing the RTX hardware to be able to load and launch ray-tracing shaders, then proceed to build a ray-tracer to experiment with 3D models and the GLSL shader language in order to learn more about what ray-tracing can do.

Once this is all in working order, try to get a glance of the speed of the hardware by measuring the frames per second obtained in the test scenes, to finally proceed to work in optimizing the software as much as possible during the remaining days of the semester, until the moment of releasing this document.

3 METHODOLOGY

A lot of reading and some planning preceded the start of the programming work. In this way some basic errors were avoided in the management of time and fewer tries were required to understand and put the Vulkan code base to work. In this phase, the Vulkan SDK was installed, and code examples were downloaded and tested. Windows and Vulkan boilerplate code was added for the detection of the environment, window and event management and draw calls initialization, but ray-tracing tests were still not possible, as there's currently no software fallback provided by Vulkan, and writing an equivalent software ray-tracer, as fun as it can be, was out of the planned scope for this project.

The compiler installed in the development machine was Microsoft Visual Studio 2019, and the code has been tested in 64 bits from the start. Development was helped by the use of a Subversion code repository. That allowed for many easy error catches, as comparing changes in the source code with last known working versions was made trivial in this way. The operating system was the latest revision of Windows 10, being it the current standard for PC video games until Linux gaming catches up.

A proper GPU was finally acquired and connected to the computer, a more difficult step than it should have been as the market conditions during the start of 2021 were incredibly tough (three months of active search were required until I got my hands on a Nvidia RTX 3060ti).

Now, with a capable system, ray-tracing examples were studied and work on the ray-tracer started. Many of the examples didn't compile or work properly due to the experimental nature of the RTX extensions before the specifications were closed in December of 2020. Some examples were able to work after renaming some of the definitions and function calls, but in the end examples were not that useful, and this became even worse after the Nvidia software autoinstalled an update without warning first, rendering my first month of work obsolete by making the executables compiled with the old SDK incompatible, forcing me to upgrade to the newer Vulkan SDK and spending a few days fixing the functionality.

Annoying as it can be, this wasn't all that unexpected as we are still in what I call the "Wild Western" stage of this technology. I have had to fight with equivalent problems with many pioneering technologies before, and anyone who has had contact with pre-release game system devkits, the first years of 3D graphics acceleration or the first modern virtual reality wave, can surely relate and can remember similar experiences.

As it will be explained later, testing of a handful of ray-tracing techniques were then put into test. Sometimes the same image has been drawn by rasterization and ray-tracing and compared for differences. The frames per second achieved have also been noted as a way to hypothesize how more complex scenes could be.

4 THE VULKAN GRAPHICS API

4.1 The state of the art in graphics APIs

Vulkan is the successor of OpenGL as the industry standard for 3D graphics, and brings more control to the programmer allowing for the use of multiple screens, multiple hardware devices and so on.

While the concept of ray-tracing is in theory not tied up to any API or library at all, the reality is that nowadays standarized ray-tracing hardware acceleration is limited to only two choices: The Microsoft Direct 3D 12 API, itself limited to target the Windows operating system and the Xbox Series X and S game consoles, and Vulkan, that besides Windows can also target Linux, the PlayStation 5 console and (very) modern mobile phones and tablets. Vulkan is also interesting as it's the API of choice for the Nintendo Switch games console, but unfortunately the Switch does not have any hardware ray-tracing capabilities.

It's worth noting the abscence of Apple's Mac OS X in the list, but it can still be targeted by means of an API wrapper. Apple has traditionally been hostile to video games in general and when DirectX 12 and Vulkan were developed Apple released their own proprietary API "Metal" as a lock-in for iOS developers. Metal is not a bad API but it's usefulness from a market share perspective is truly limited. Thankfully there's an official wrapper developed, MoltenVK (`github.com/KhronosGroup/MoltenVK`), allowing for Vulkan code to run unmodified in OSX and iOS and already supports even the latest Apple ARM M1 devices.

The Khronos Group has not developed any ray-tracing extensions for OpenGL, so if one doesn't want to get tangled to the Microsoft proprietary ecosystem and wants realistic cross platform possibilities Vulkan is the natural choice here.

4.2 Fast introduction to Vulkan

4.2.1 Before starting

The official Vulkan SDK can be downloaded from `www.lunarg.com/vulkan-sdk/`

Just like OpenGL, Vulkan works around the concept of handles, where the programmer asks for access to resources and the drivers return an ID associated to them. With Vulkan residing in a lower-level layer than OpenGL pointers to the hardware memory buffers aren't out of the question and it's possible to access VRAM from the CPU

side, but using handles allow for the drivers extra freedom to rearrange things when necessary, like when a surface gets invalidated, without requiring handles to be reacquired again.

Of course a high degree of control requires dealing with an equally high amount of details. Vulkan calls require a lot of information to be supplied by the programmer and the way this is done is by passing structures of data instead of having functions with many parameters.

Tastefully Vulkan is still a C API; C++ can be used to target the C API but it's not a requirement. By doing it this way, creating bindings for higher-level languages for people who use them, like the scripting parts of video games, becomes easier. But this time, there's also a C++ version of the API, supplied as a one file C++ include, taking over the tasks of some petty initialization and adding some extra conventions resulting in smaller code that is a bit more robust. This header can be downloaded from it's official Github repository at `github.com/KhronosGroup/Vulkan-Hpp`

My tests have revolved mainly around the plain C API although some tests have also been conducted with the C++ header as to gain insight on it.

4.2.2 Vulkan initialization

Compared with OpenGL, initializing Vulkan is a long process with a steep learning slope, no matter if the C or the C++ version of the API is being used. But the general steps are logical, indeed.

You start by creating a **VkInstance** to search for available GPUs with Vulkan support. Usually a personal computer is going to have just one, but this allows for the creation of heterogeneous multi GPU systems and gives the programmer control over what to do with each one. You then loop through them, read the device properties through **VkPhysicalDevice** and when you are ready to select one of the available hardware devices a **VkDevice** is created to work with it.

Next steps involve allocating the VRAM required for work. Creating a **VkBuffer** on the **VkDeviceMemory** with **vkAllocateMemory**. The ability of deciding how and where to store objects in this way is what I consider some of the star differences of Vulkan over OpenGL.

Resources get binded to the **VkDevice**, a **VkCommandBuffer** gets created from the **VkCommandPool** so we later can use it to schedule work **VkQueue** queues from the device. There are different types of queues meant for drawing, GPGPU work and more. A fun fact is that no Vulkan device has a hard requirement for implementing drawing, something surprising from a video games developer but probably obvious for people used to HPC (High Performance Computing).

Now a **VkPipeline** gets created from **vkCreateGraphicsPipelines** and it's up to the programmer to configure how drawing will be performed by it. The layout may include rasterization, calls to shaders and much more. The pipeline is one of the most complex objects in Vulkan.

4.2.3 Shaders

Shaders are handled within **VkShaderModule** where SPIR-V compiled files are loaded.

One big historic limitation of OpenGL version Direct3D has always been the fact that shaders had to be loaded in source code form and compiled by the drivers. There were some synthax differences among GPU vendors or even driver versions. Process was slow, and shader code was there for anyone to look into, something that many companies just don't like. It was possible to store the compiled binary representation of the driver to disk to cache the compilation results, but this was standard nor good practice as it could make the application crash when the user updated the drivers or upgraded the hardware.

Shaders in Vulkan are not fundamentally tied to any language. One can use GLSL, HLSL or even create own languages if such a need arises. The programmer then uses a SPIR-V compiler to store a binary containing SPIR-V bytecode ready for the GPU drivers to be loaded and executed.

Storing the shaders in SPIR-V representation allows for simpler drivers and much faster loading speed.

The Vulkan SDK contains the glslangValidator.exe compiler whose invocation could for example be done in this way:

glslangValidator.exe –target-env vulkan1.2 -V -S rgen INPUT_FILE -o OUTPUT_FILE

This small example shows how a minimum Vulkan version can be specified, and the type of shader being requested, a ray generation shader in this case (more on this type of shader later).

To tell Vulkan what shaders to use and their user data there's a buffer of the shader handles and the user values specified by the programmer.

4.2.4 Draw loop

vkAcquireNextImageKHR gets an image handle where drawing can take place. This doubles also as the synchronization mechanism as it will wait until the time is correct for a new frame, with a maximum of one second.

Vulkan is parallel by nature so to ensure the GPU has finished all the required tasks before trying to render a scene semaphores and fences primitives are supplied by the API (**VkSemaphore** and **VkFence**) and created by using the functions **vkCreateSemaphore** and **vkCreateFence** respectively. Then **vkWaitForFences** pauses execution and vkResetFences resets it after being allowed to pass so

to be able to use it on next frame.

At this point the **VkCommandBuffer** we initialized before can be used to start issuing new work for the GPU to execute. First by resetting it with **vkBeginCommandBuffer**, then adding commands to the queue with **vkCmdBeginRenderPass / vkCmdEndRenderPass** and closing it with **vkEndCommandBuffer** and submitting everything for GPU processing with **vkQueueSubmit**

Finally **vkQueuePresentKHR** is called to display the frame we generated.

5 THE VULKAN RTX EXTENSIONS

5.1 Getting ray-tracing to work with Vulkan

The flexibility of Vulkan starts to pay off now, as adding a ray-tracing step in the draw process is like rasterizing but by calling the function **vkCreateRayTracingPipelinesKHR**.

In the information creation structure the number of ray bounces is stated by the recursion level. One means no bounces after reaching a surface for the rasterizing-like aesthetic. The real magic here lies on upping that limit, sending the scene data to the GPU and the new shaders that are going to be required by the GPU during the ray-tracing process.

The command vkCmdTraceRaysNVX is issued to initiate the process of launching a 2D grid of rays from the camera plane, one per each pixel that will get on the image buffer.

5.2 The acceleration structure

The acceleration structure purpose is to store the scene information needed to render the image seen from the camera location, spatially sorted, and in a way that is efficient to traverse so as to calculate the ray intersections.

One can talk about acceleration structures, in fact, as there are two types: The bottom level and top level.

- Bottom level acceleration structures

  For practical purposes, the bottom level acceleration structure is the representation of the geometry of our objects. In a tree-like fashion other bottom level acceleration structures can hang from it as children nodes. These are the objects that will be evaluated for intersection detection with the rays. There's also information about how the object is transformed and the materials to use.

- Top level acceleration structure

  The top level acceleration structure acts as the root for the scene tree comprised of the bottom level structures and allows the GPU to traverse the data required during the ray-tracing process.

The node contents are basically AABB bounding boxes for fast broad tests and triangle sets for detailed intersection calculation, but the internal organization and data of an acceleration structure is opaque to the programmer because this enables the drivers to customize it in the most efficient way for each GPU type.

In order to feed the required data to the GPU, geometry is summited in a standarized way by using the **VkAccelerationStructureGeometryKHR** structure. This structure contains fields for specifying triangles, AABB bounding boxes or children acceleration structures.

5.3 Ray-tracing shaders

There are five types of ray-tracing shaders:

- Ray Generation

  These represent the starting point for the ray-tracing calculation. A ray is instanced and it's output gets written to the output buffer. There's an example at the appendix A.5.1

- Intersection

  These are used for non-triangle shapes hit detection and are completely determined by the programmer who can use these to implement sphere collision detection, for example.

- Hit (Any)

  Any hit shaders are called on each instance of the ray hitting enter and exit points for all meshes inside the scene. Useful for some effects but not the fastest way to draw.

- Hit (Closest)

  This is the "true" hit shader that is usually required for ray-tracing. It's called when the ray intersects a mesh for the closest position from the ray starting point. The closest point to the camera plane in the case of the initial ray instance, then the closest point after hitting the last surface on the ongoing process of recursing the ray bounces. An example is at the appendix A.5.2

- Miss

  If the ray does not intersect anything this shader gets called and the programmer can decide what to display in that case. A solid background color, a gradient, an image, etc. This can be a really simple shader as can be observed in appendix A.5.3

6 GETTING ONE'S HANDS DIRTY

Up to this point there has been basically a lot of theory. Now, in order to put these concepts to work, and learn about how all of this really fits together in practice, let's talk about the the very simple ray-tracer that has been developed by using the Vulkan API. The test scenes have been purposely designed for testing features that are not easy to simulate in rasterization: Shadows, transparencies and reflections have been generated.

6.1 Getting prettier scenes: Loading 3D models

With not much time or abilities to model 3D objects at first I have been using simple models created by hand by some trial and error. I later used simple loops to create improved and more complex objects, but nothing beats real models crafted by real artists. Thankfully the Khronos Group aknowledges this may be important for people starting with the API and there's a small repository of sample 3D models that allow for nicer tests. This repository can be found in this URL:

`github.com/KhronosGroup/glTF-Sample-Models`

The models are stored in the glTF format, and for the purposes and scope of this project, it's basically a list of vertex positions plus a list of triangles created by listing the number of primitives and listing the indices in the vertex list, plus texture information. Just as with the hand created models the vertices are stored inside acceleration structures then feed to the GPU.

The tiny glTF library has been used to help with the task of loading the required vertex and transform data.

6.2 Shadows

Shadows can be described as light attenuation by means of a barrier in the path between a surface and a light source. There isn't a unique way to simulate shadows in rasterization. A variety of techniques have been designed over the years but in the end shadows are usually reserved to some objects interacting with some light sources projecting an approximation of the shadow on some specific surfaces. All in all a very poor approximation which is a lot of work and not appropiate in many situations. Making things even worse, self shadowing of objects, where the own surface features project shadows in the own object require even more intensive work with many drawing passes.

Meanwhile, in raytracing perfectly formed and complete shadows come "for free" as the color calculated simply is not the same if there's a clear path to a light source or there isn't. Shadows also can be colorized by transparent objects.

Figure (2) shows an object with not only shadow and self shadowing but also self reflections. More on reflections on the next subsection.

Shadows are difficult to calculate when not using raytracing as to be accurate and complete they require interacting with the whole scene geometry to render properly, but this data is not present in the usual rasterizing pipeline. Objects drawn by rasterization can't read the world around them: remember objects are drawn in isolation.

In ray tracing shadows get calculated by means of intersecting objects from the current location to all of the light positions. If an object gets in the way the point will get darker, if not, light will add to the overall brightness.

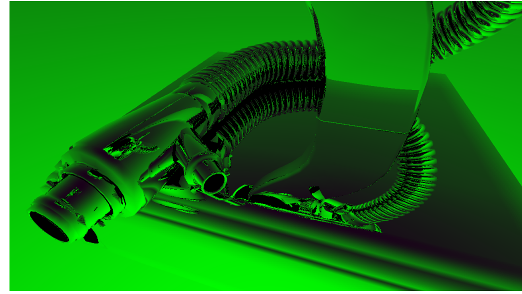

Fig. 2: Self reflections and shadowing

6.3 Reflections

The same set of problems happen when trying to render reflections: This makes impossible to draw correct representation of the reflections of nearby objects.

In ray tracing reflection comes as a natural feature: Similarly to shadows a new ray is calculated from the nearest intersection point surface, but this time it's aligned to the barycentric coordinates normal of the vertices to check intersections with other objects or other parts of the current one. The returned value is then used to form the final pixel value. This is an iterative process, new intersections found can also launch new rays. A depth limit is usually set so as to not get stuck in a loop and limit the processing and time required to form the final scene. In my tests looks like this limit is 31 extra bounces besides the initial ray from the camera plane into the scene.

This allows for some interesting effects. Not allowing for the original rays to bounce gives a very raster-like image look. In videogames a few bounces, for example 4, give good enough results. In practice, even high quality images get good enough from 10 or more bounces, although this is very dependent on every image and their specific details.

As an example, in figure (3) a bunch of spheres are reflected on the curved surface of another sphere. This image had a bounce limit of 31 times, and as such features very detailed reflections of other reflections with all the scene spheres visible.

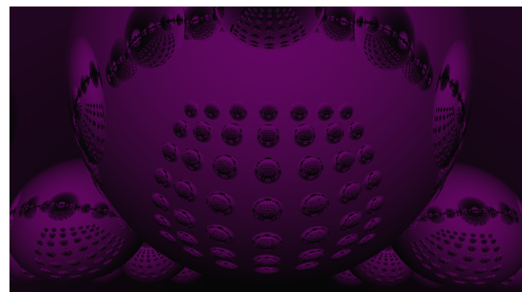

Fig. 3: Reflection of nearby objects

6.4 Transparencies and light refraction

Transparencies have always been possible to simulate in rasterization by means of having two draw queues, one

for regular opaque objects, unsorted relying on z-buffer, and other for ordering the draw calls from far to near and reading the z-buffer but not writing to it. Still, this has always been only an approximation as the objects local information limits the precision of the calculated color if multiple transparent surfaces intersect. HDR drawing pipeline where values could extend beyond the regular values helped but in the end the color mix was always off because interactions among intersecting surfaces. Self transparency was problematic because of that, too. Objects with no clear sorting order, like two objects interlaced, had even more problems because different methods of assessing the object distance from the camera gave different results. And besides being impossible to calculate a really accurate color value for the transparency itself, the biggest deficiency was in simulating transparency on curved surfaces where light would experience refraction. Shaders tailored for these special surfaces were used to simulate a fake approximation of what could be expected instead. Some programmers opt for not featuring transparencies in their game engines because they are too much trouble for far from perfect and even unpredictable results.

Due to all of these problems, a scene comprised of crystal balls is better reimagined as opaque spheres when using rasterization in figure (4), but looks gorgeous in ray-tracing in figure (5) .

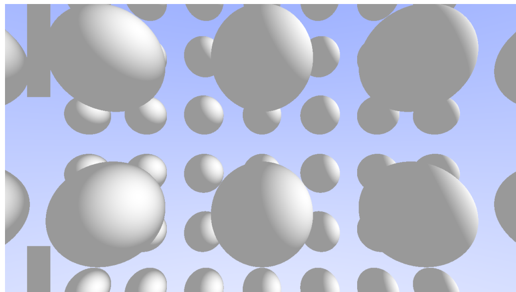

Fig. 4: Dull raster spheres

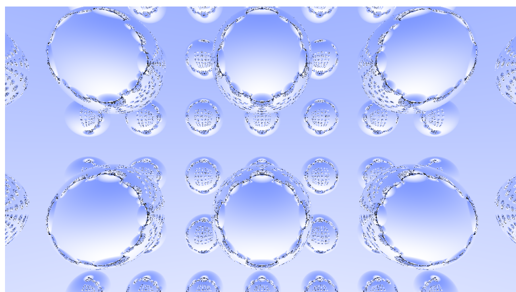

Fig. 5: Same scene with realistic looking light refraction on crystal spheres

6.5 Performance of generating simple scenes

Showing images is visually interesting, but what about the performance? A still image is going to look pretty no matter if it's requiring a few milliseconds to be drawn or a few minutes. So the following table gives information on how many frames per second these scenes got in the test machine in order to help the reader to get a more complete understanding. The original image resolution is the current gaming standard,1920x1080 pixels (Full HD), at 32 bits per pixel (R, G, B, A). At first these scenes were working in the order of 200 - 300 frames per second, but once the bulk of the work was done a few extra days were spent in simplifying the CPU side of the code and, more importantly, the closest-hit shader.

Scene	Frames per second
Self reflections and shadows on a complex 3D object (raytracing) Figure 2	3221
Reflecting objects (raytracing) Figure 3	2840
100 spheres (raytracing) - Figure 5	3162

As it can be seen in the table, thanks to this last effort, and very much to my surprise and amusement, speeds in the order of the thousands of frames per second have been achieved with these scenes.

The exact hardware specifications giving these results can be checked at appendix A.1

7 Conclusions

7.1 Thoughts about Vulkan

After one manages to get through the initial learning curve Vulkan gets more predictable than OpenGL, gets parallel processing capabilities, multi-GPU and multi-screen management, a much leaner layer interfacing to the hardware (that I would not really call "low level", but it's better than OpenGL) and acces to the fascinating world of hardware ray-tracing.

The learning curve is high. Vulkan is not meant for direct usage, but for the programmer to build a wrapper around it or use an existing 3D engine. Both the C and the C++ APIs feel tedious to use, with lots of repetitive initialization of structures, very long function names and namespaces, and a lot of intermediate steps until one is able to start drawing. Those who elect to use it directly will need to remember at times that in the end it's been designed for engine creators, and not to be used directly by application and game programmers.

This situation remembers me of comparing assembler code of the CPUs of the 1970s and 1980s with the CPUs from the 1990s and beyond that targeted compiler writers instead of assembler programmers. The increased control over the memory allocation on the GPU, being able to manage multiple devices with multiple capabilities and address multiple screens or even not drawing anything to any screen makes the code verbose, long and tedious even for the smallest of tasks.

Summing it up, I would say that this is a solid case for using some lean third party wrapper to ease the work, something I've always argued against in OpenGL, but it's a

worthwile and welcome step forward from OpenGL and a solid contender for Microsoft's Direct3D 12.

I'm not very satisfied with the existing documentation. I have liked just one book on Vulkan (none on the RTX extensions),

7.2 My thoughts on the hardware ray-tracing and the RTX extensions

The ray-tracing hardware is amazing. It's fast, it's fun. This hardware feels like once in a decade step forward kickstarting a new generation of 3D graphics, and I can't wait to see what new GPUs are going to bring in the future. About the RTX extensions, taking into account the lenghthy Vulkan initialization process, in the end using the RTX extensions and shaders feel like an easy endeavour in comparison. In this regard it's a pity the RTX pipeline has not been backported to OpenGL for simpler programs that don't require the extra control in Vulkan.

Ray-tracing is slower than traditional rasterizing but in the end it's the programmer who decides how to draw and both techniques can be mixed. This allows for increased realism when possible but without sacrificing that much speed. This is especially true for visualizations that do not require as much speed as video games. Ray-traced scenes look beautiful and the impact realistic light and the combined effect of refractions, shadows and reflections is big and transforms otherwise dull images into satisfying to watch scenes that many years ago would have required many hours each to be rendered.

I'd say documentation is still a problem. While there are a lot of websites with tutorials and tons of code all over Github that presumably should make the programmer's life easier, I personally haven't found these very useful, especially true with ray-tracing examples with many requiring old Vulkan SDK versions. There aren't that many books for Vulkan, but I haven't managed to find a book on the RTX extensions that is worth buying, which is a shame because it would have made my life much easier during the course of the semester I worked on this project.

The super-high frames per second counts achieved make me hopeful that fully ray-traced games are already at the corner. In the meanwhile, a mix of rasterization with improved shadows, reflections and light refraction effects will become the norm.

7.3 Continuing the work

The time alloted for the final year project on a engineering degree is limited and I'm pretty satisfied with the results I have achieved. That said, it's not my intention to stop after finishing this essay.

My original intent was to compare Nvidia vs AMD performance and compare the same images rendered on the cards trying to find differences. Hardware availability during this work has been unfortunately crippled due to the ongoing semiconductor crisis that started in 2020, the cryptocurrency miner craze and the wild speculation by organized groups using bot farms to automate buying GPUs and other high demand items for later higher price resale. So while I should probably be thankful that I was lucky enough to even manage to get one card, I have been unable to obtain any other GPU for comparison. I'll eagerly get to this when the stock problems get away.

Once I have an AMD card I'm also willing to measure the power draw of both cards while rendering the scenes.

Feature-wise, following up this small research work I expect next to add support for textures. I deliberately ommited it from my tests as I didn't want to get visually distracted from pure color values obtained by the ray-tracing process to better assess reflections, but in real world applications textures are a must have.

I would also like to put more effort on trying to optimize the shaders and see how far I can push my graphics card. While I have spent a good deal of time trying to improve the code done these months in the end now with the newly acquired insight I should be able to more readily get to the point and make smaller and faster implementations of everything.

On a more general level I also want to continue this line of work by including support for ray-tracing in my personal library of base code and using the RTX extensions at work. Surely starting the code from scratch as now I have a much deeper understanding of Vulkan and how the RTX extensions work so I'll be able to make a smaller and more to the point implementation. A down side on Vulkan has been that my expectations of being able to simplify the current code have gotten crushed by the extra effort required with Vulkan, versus the simpler and much easier OpenGL wrapper I managed to fit in less than 2500 lines of C code, but the added capabilities of the RTX extensions make this effort worthwhile.

ACKNOWLEDGMENTS

I would like to give thanks to my wife Montse Estadella for her support and help over these years that I've become a student again. She has helped me to endure through the most difficult times until I've finally been able to get to the end, and my daughter Ada who may one day be able to write her own ray-tracer too ;-)

Juan Carlos Moure for his ongoing patience with whom I suspect may be a somewhat difficult student.

And all the teachers from the Computer Architecture and Operating Systems (CAOS) department, for hoarding nearly all of the fun subjects in Computer Engineering in the UAB :-)

Finally, this work is also dedicated to the memory of Byuu, a giant on the video game emulation and preservation scene, who tragically passed away during the course of this work. Godspeed Dave.

References

1. "Vulkan Programming Guide" Graham Sellers, John Kessenich, Addison Wesley, 2017. ISBN 9780134464541

2. "Object-Oriented Ray Tracing in C++" Wilt, Nicholas, Wiley Editorial, 1994. ISBN 0-471-30414-X

3. Official website of the book series"Ray tracing in one weekend" [Online]. raytracing.github.io.

4. "Vulkan official website" [Online]. www.khronos.org/vulkan.

5. "Khronos Vulkan Resources Archive" [Online]. github.com/KhronosGroup/Khronosdotorg/blob/main/api/vulkan/resources.md.

6. "Vulkan - Nvidia Developer" [Online]. developer.nvidia.com/vulkan.

7. "NVIDIA Vulkan Ray Tracing Tutorial" [Online]. developer.nvidia.com/rtx/raytracing/vkray.

8. "Introduction to Real-Time Ray Tracing with Vulkan", NVIDIA Developer blog [Online]. developer.nvidia.com/blog/vulkan-raytracing/.

9. "GLTF Sample Models" [Online]. github.com/KhronosGroup/glTF-Sample-Models.

Appendix

A.1 The hardware employed for this work

On the CPU side, there's a 3.3 GHz water cooled Intel i7 5820K that has been overclocked to 4.2 GHz, with 32 GB of DDR4 RAM.

On the GPU side, the computer mounts an Nvidia RTX 3060ti graphics card. This card was only released on December 2021, and it's basic specifications are:

- Base clock speed of up to 1.665 GHz, overclocked to 2.1 Ghz
- 8 GB of GDDR6 VRAM
- 4864 CUDA Cores
- 38 Ray-tracing cores
- Memory bandwith: 448 GB/s

A.2 Open source code and libraries used

- Vulkan SDK

  The SDK is required for Vulkan usage. Some examples have been used as a guide on how to correctly set-up a Vulkan program to work.

- tiny glTF

  Required for 3D model data loading. Check appendix XX for some more information.

- stb_image

  This library is required by tinygltf, but not really used, as the ray-tracer developed did not support textures at the time of this writing.

- ktx

  Required by the Vulkan base code, simplifies usage of Vulkan images.

- glm gl math

  Math library for vectors.

Some open source example code from Sascha Willems (Khronos Group) was also of help in getting started.

A.3 3D Model store format: glTF and the tiny glTF library

Although I used a custom format for 3D model data loading in the 3D engine I wrote a few courses ago in the graphics topic, this time and out of curiosity for the endorsement of the Khronos Group, I wanted to try glTF (GL Transmission Format). The glTF specification is royalty-free, and the 2.0 version can be found at www.khronos.org/gltf/. Many objects are available for free by searching the Internet, allowing for more beautiful tests. The Blender 3D modeler also supports this format, so it was ideal for my tests. glTF can be used in text and binary formats, and when using the text one it's internally organized in JSON format.

To aid with the loading of the glTF models the free tinygltf library has been used. This library can be found at github.com/syoyo/tinygltf and I appreciate the fact it's composed by a very few header files and compiles at the very first try, something that sadly in my personal experience is not that usual.

A.4 Small survey of current RTX hardware

At the date of this writing, June 2021, RTX hardware is available in the form of the Sony PlayStation 5 and Microsoft Series consoles, sadly not fully open to the public for tinkering, the Turing and Ampere graphics cards from nVidia, and the Radeon RX 6000 series from AMD.

Nvidia is currently offering a chaotic line of cards ranging from some RTX 2070 and RTX 2080 models from the past generation to an unending supply of new Ampere chips that are instantly sold out when available at online

stores. Models supposedly available are the RTX 3060, RTX 3060Ti (the one used in this work), RTX 3070, RTX 3070Ti, RTX 3080, 3080Ti and the RTX 3090, and five more models for laptops. These share the same basic architecture and vary on the amount and type of VRAM employed and the number of the processing units.

Similarly, AMD offers a more streamlined range with the RX 6800 and 6800 XT, the high end RX 6900 XT and the cheaper RX 6700 XT. The RX 6000M series for laptops have been introduced less than a month ago.

AMD is putting a tough battle to Nvidia with the RX 6000 series, but in this generation these cards are not nearly as fast in hardware ray-tracing as the ones from Nvidia. If this is really that troublesome for gamers, taking into account that not many video games support ray-tracing and rely on the rasterizing speed, very good with both manufacturers, then the answer is probably not, but for development purposes the situation may be different.

A.5 Shader source code examples

A.5.1 A ray generation shader

```
#version 460
#extension GL_EXT_ray_tracing : require

layout(binding = 0, set = 0) uniform
    accelerationStructureEXT topLevelAS;
layout(binding = 1, set = 0, rgba8) uniform image2D
    image;
layout(binding = 2, set = 0) uniform CameraProperties
{
  mat4 viewInverse;
  mat4 projInverse;
  vec4 lightPos;
} cam;


struct RayPayload {
  vec3 color;
  float distance;
  vec3 normal;
  float reflector;
};

layout(location = 0) rayPayloadEXT RayPayload rayPayload
    ;

layout (constant_id = 0) const int MAX_RECURSION = 0;

void main()
{
  const vec2 pixelCenter = vec2(gl_LaunchIDEXT.xy) +
    vec2(0.5);
  const vec2 inUV = pixelCenter/vec2(gl_LaunchSizeEXT.xy
    );
  vec2 d = inUV * 2.0 - 1.0;

  vec4 origin = cam.viewInverse * vec4(0,0,0,1);
  vec4 target = cam.projInverse * vec4(d.x, d.y, 1, 1) ;
  vec4 direction = cam.viewInverse*vec4(normalize(target
    .xyz / target.w), 0);

  uint rayFlags = gl_RayFlagsOpaqueEXT;
  uint cullMask = 0xff;
  float tmin = 0.001;
  float tmax = 10000.0;

  vec3 color = vec3(0.0);

  for (int i = 0; i < MAX_RECURSION; i++) {

    // Lanzar el rayo
    traceRayEXT(topLevelAS, rayFlags, cullMask, 0, 0, 0,
      origin.xyz, tmin, direction.xyz, tmax, 0);

    // Recoger el color
    vec3 hitColor = rayPayload.color;

    if (rayPayload.distance < 0.0f) {

      color += hitColor ;
      break; // Rompemos el bucle
    }
    else {

      const vec4 hitPos = origin + direction *
     rayPayload.distance;
      origin.xyz = hitPos.xyz + rayPayload.normal *
     0.001f;
      direction.xyz = reflect(direction.xyz, rayPayload.
     normal);
    }

  }

  imageStore(image, ivec2(gl_LaunchIDEXT.xy), vec4(color
    , 0.0f)); // Guardamos el valor
}
```

src/raygen.glsl

A.5.2 A closest-hit shader

```
#version 460

#extension GL_EXT_ray_tracing : require
#extension GL_EXT_nonuniform_qualifier : enable


struct payload {
  vec3 color;
  float distance;
  vec3 normal;
  float reflector;
};

layout(location = 0) rayPayloadInEXT payload ray_payload
    ;

hitAttributeEXT vec3 attribs;

layout(binding = 0, set = 0) uniform
    accelerationStructureEXT topLevelAS;
layout(binding = 2, set = 0) uniform UBO
{
  mat4 viewInverse;
  mat4 projInverse;
  vec4 pos_luz;
  int tam_vertice;
} ubo;

layout(binding = 3, set = 0) buffer Vertices { vec4 v[];
     } vertices;
layout(binding = 4, set = 0) buffer Indices { uint i[];
    } indices;



struct T_VERTICE
{
  vec3 pos ;
  vec3 normal ;
  vec2 uv ;
  vec4 color ;
};


T_VERTICE leer_vertice(uint num_triangulo)
{
  const int m = ubo.tam_vertice / 16;

  vec4 d0 = vertices.v[m * num_triangulo + 0];
  vec4 d1 = vertices.v[m * num_triangulo + 1];
  vec4 d2 = vertices.v[m * num_triangulo + 2];

  T_VERTICE v;
  v.pos = d0.xyz;
  v.normal = vec3(d0.w, d1.x, d1.y);
  v.color = vec4(d2.x, d2.y, d2.z, 1.0);

  return v;
}


void main()
{
  // Lectura de la posicion de los vertices
```

```
62    ivec3 num_triangulo = ivec3(indices.i[3 *
         gl_PrimitiveID], indices.i[3 * gl_PrimitiveID + 1],
          indices.i[3 * gl_PrimitiveID + 2]);
63
64    T_VERTICE v0 = leer_vertice(num_triangulo.x);
65    T_VERTICE v1 = leer_vertice(num_triangulo.y);
66    T_VERTICE v2 = leer_vertice(num_triangulo.z);
67
68    // Calculo de la normal del punto
69    const vec3 barycentricCoords = vec3(1.0f - attribs.x -
          attribs.y, attribs.x, attribs.y);
70    vec3 normal = normalize(v0.normal * barycentricCoords.
         x + v1.normal * barycentricCoords.y + v2.normal *
         barycentricCoords.z); // Mejor asi?
71
72    // Iluminacion
73    vec3 lightVector = normalize(ubo.pos_luz.xyz);
74    float dot_product = max(dot(lightVector, normal), 0.6)
         ;
75    //ray_payload.color = v0.color.rgb * vec3(dot_product)
         ;
76    ray_payload.color = ((v0.color.rgb + v1.color.rgb + v2
         .color.rgb) / 3.0f) * vec3(dot_product); // Mejor
         asi?
77    ray_payload.distance = gl_RayTmaxEXT;
78    ray_payload.normal = normal ;
79
80    // Marcamos el objeto como reflectivo
81    ray_payload.reflector = 1.0f ;
82 }
```

src/closesthit.glsl

A.5.3 A simple miss shader

```
1  #version 460
2  #extension GL_EXT_ray_tracing : require
3
4
5  struct RayPayload {
6    vec3 color;
7    float distance;
8    vec3 normal;
9    float reflector;
10 };
11
12 layout(location = 0) rayPayloadInEXT RayPayload
        rayPayload;
13
14
15 void main()
16 {
17   // Generamos un fondo con un degradado usando los tres
          ejes de coordenadas
18
19   vec3 unitDir = normalize(gl_WorldRayDirectionEXT);
20   rayPayload.color = vec3(unitDir.x, unitDir.y, unitDir.
        z) ;
21
22   rayPayload.distance = -1.0f;
23   rayPayload.normal = vec3(0.0f);
24   rayPayload.reflector = 0.0f;
25
26   //rayPayload.color = vec3(1.0, 0.1, 0.2) ;
27   //rayPayload.color = vec3(0.1, 0.1, 0.1) ;
28 }
```

src/miss.glsl