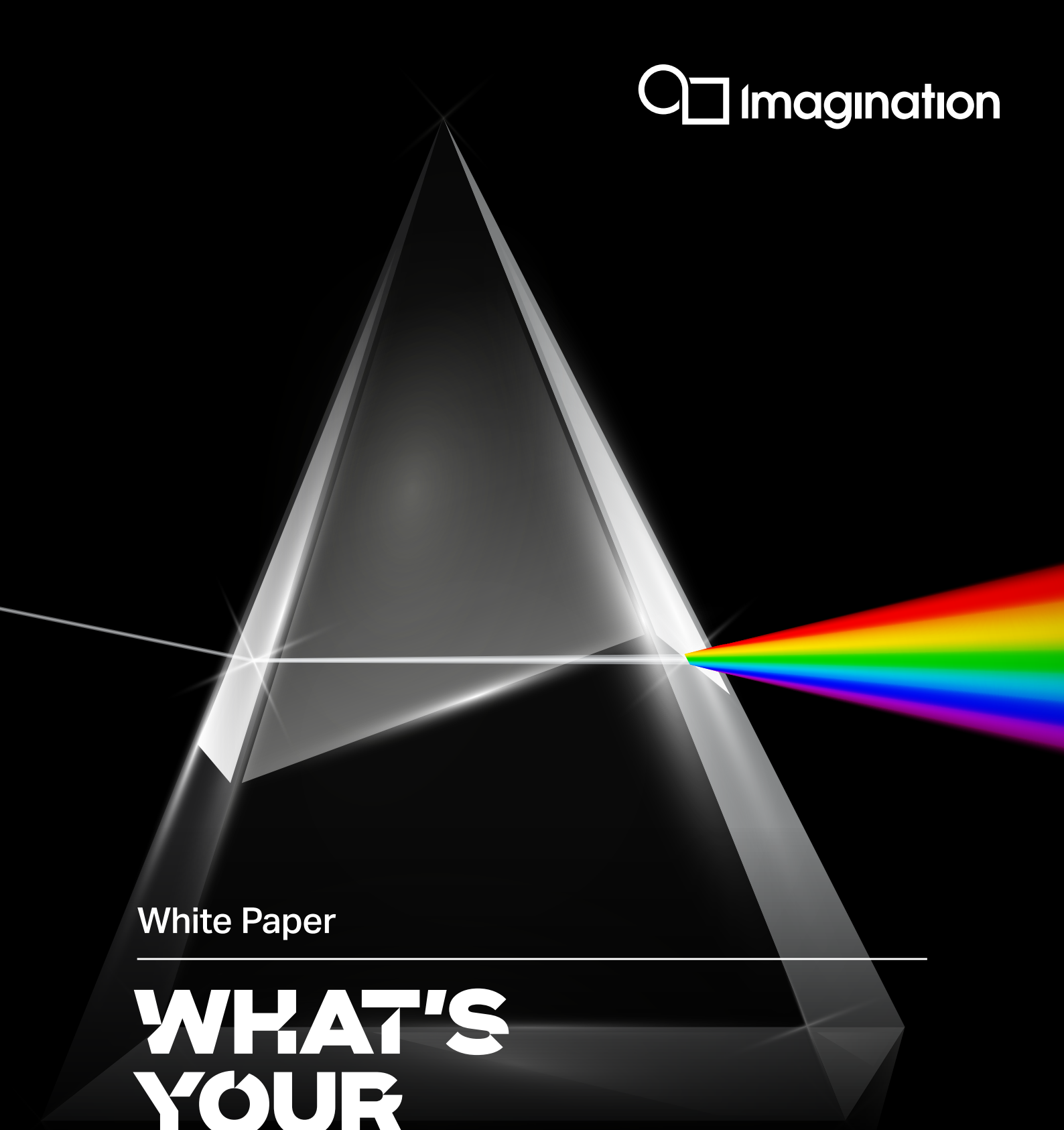
Imagination



White Paper

WHAT'S YOUR LEVEL?

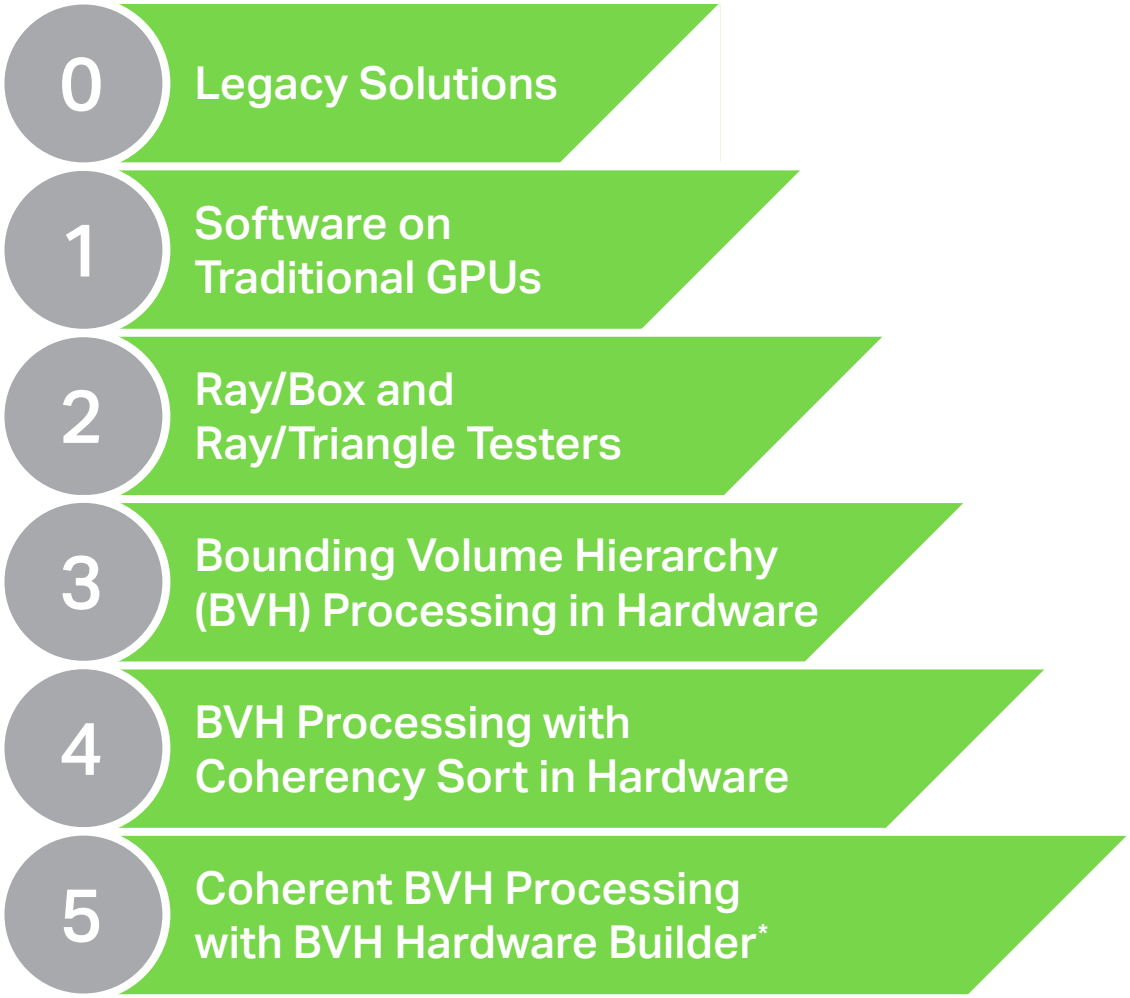
The six levels of ray tracing acceleration

By Kristof Beets, Senior Director of Technical Product Management, Imagination Technologies

The six levels of ray tracing acceleration

Ray tracing is a technology that is currently dominating headlines as the next step in graphics technology and by the end of the year will be widely available to consumers across desktop personal computers (PCs) and consoles. This primer will introduce the concept of ray tracing levels to make clear that not all ray tracing solutions are created equally, and that higher-level ray tracing is more capable and feature-rich than the lower levels. As the levels are incremental this primer introduces the architectural changes and capabilities as we build up from Level 0 to Level 5.

This paper assumes a basic knowledge and understanding of the fundamentals of ray tracing which you can find out more about by reading our **ray tracing primer**.



* A BVH Builder (SHG) can also be added to lower efficiency levels where this would be indicated by the addition of "plus", e.g. a "Level 2 Plus" solution.

Ray Tracing Level 0

Legacy Solutions

Ray tracing is not a new approach to rendering graphics. The concept has been around longer than today's traditional, mainstream GPU rendering, and has been in use for quite some years as the rendering approach of choice for product design, architectural work, visual effects and movie rendering. Ray tracing is also fundamental to artwork creation for games where it is commonly used offline to bake light and shadow information into the artwork which is used in real-time game engines. However, once "baked", this light information is static and cannot change.

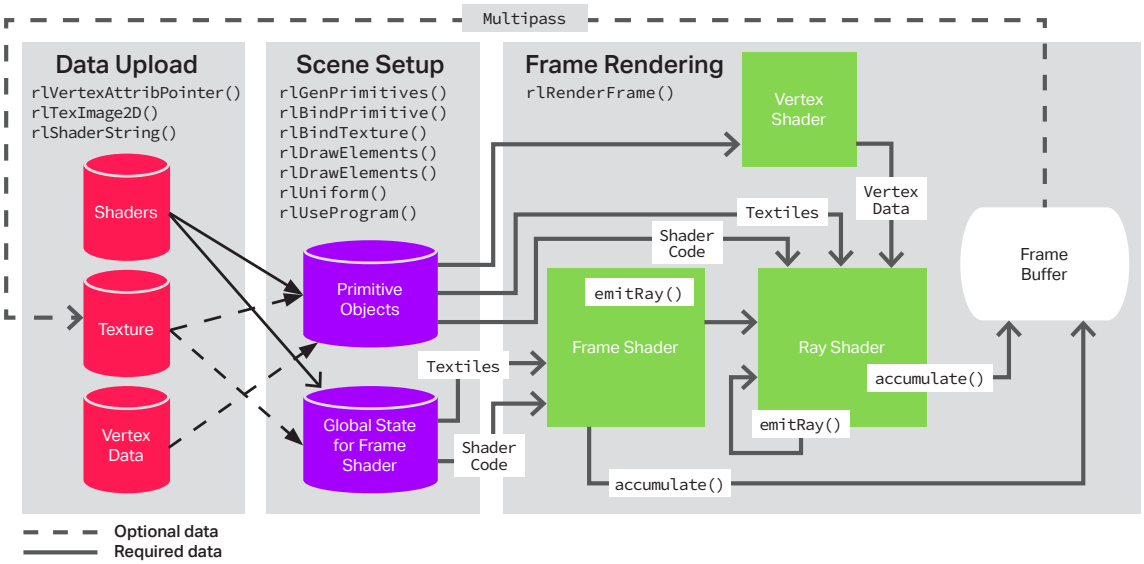
As ray tracing is already a building block of today's game content, hardware acceleration to enable ray tracing with full dynamics and in real-time, has been a long-standing goal for many companies. Indeed, many have tried to accomplish this and, until very recently, most have failed.

Typically, these early attempts at accelerating ray tracing focused on new and very different "GPU" architectures, which had to be matched with equally new and non-standard proprietary APIs. This divergence was a major stumbling block for market adoption, as switching from, say, an OpenGL® (or OpenGL ES, Vulkan® or DirectX®) ecosystem to something completely different is expensive and difficult.

Intel® Embree kernels and Caustic's OpenRL are some of the historical examples where a custom API is presented which has no easy continuity with the dominant traditional graphics APIs. Caustics OpenRL was, in many ways, a first step in the right direction, as it started from something which looked like OpenGL and used some of the known concepts and language, which helped remove some of the barriers to entry. Ultimately though, it was still a new and divergent API. Some ray tracing solutions do not even consider an API and instead offer a full rendering engine which you have to use, which of course is an even bigger hurdle to adoption by developers.

While without a doubt these early attempts have helped pave the way to enabling today's real-time ray tracing solutions, they have simply not been successful and hence are Level 0 solutions.

Open RL Data Flow



Ray Tracing Level 1

Software on Traditional GPUs

As the main downfall for Level 0 solutions was ecosystem and compatibility the most obvious way to enable a transition path is to implement ray tracing using the existing graphics and compute APIs. Indeed, many software solutions have been offered which did exactly that, especially in the PC add-in graphics board market, with many proprietary effects and implementations deeply embedded into game engines.

While most of these solutions are not really an "API" and definitely not a "standard API" these solutions do broaden the scope and usability for ray tracing and have been widespread for many years. However, most focused on either very high-end cinematic rendering at slow speeds (think hours per frame, not frames per second) or applied to very limited and selected effects enabled inside a game engine. In many cases these effects have adopted the name ray tracing but often they only vaguely used a simplified version of the concept.

The main issue with pure software for ray tracing is that ray tracing is inherently computationally expensive and too complex to handle it in a generic API-type way. This means that it allows for very few shortcuts and these only work if you can keep the problem-space narrow allowing you to use lots of tricks. However, with generic API models that is not possible. Hence while software ray tracing on GPUs is usually faster than on CPUs the framerate always remained low and not truly real-time. Some "ray tracing" software solutions offer higher speeds – even real-time – but typically this is achieved by cutting corners and optimising for those specific subsets of functionality, and many of these techniques depend on (pre-calculated) look-up tables and structures that have been highly tuned and optimised within the capability boundaries of the algorithms.

Software solutions play a key role in helping build up demand for functionality but are never a long-term solution as software will inherently rapidly fall orders of magnitude behind dedicated and tuned hardware solutions with regards to performance, power efficiency and bandwidth efficiency; hence these are Level 1 solutions. They can support and extend the ecosystem of higher-level solutions though – for example, Microsoft offers DXR emulation, which is very much a Level 1 solution. It allows content to run on a wider range of platforms but likely at significantly reduced performance and/or quality.

Ray Tracing Level 2

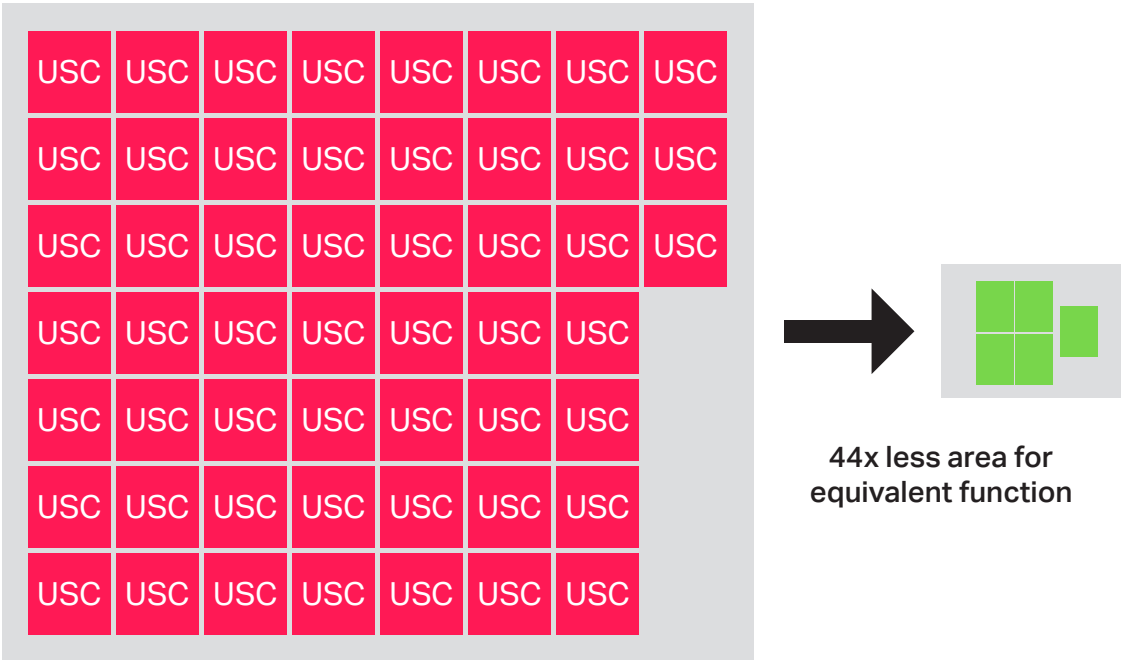
Ray/Box and Ray/Tri Testers

When looking at ray tracing processing on a GPU in software it quickly becomes clear that most of the cycles invested are in the processing of the ray-box and ray-triangle intersections.

Essentially, just doing a ray-box intersection test using programmable GPU logic, basically fused multiple adds (FMA), requires 44x more silicon area than what can be achieved in a fixed-function block which offers the same capability. At GDC in 2014, Imagination illustrated this in an effective visual way (see below).

It should come as no surprise that using a much smaller fixed-function block is also much more power and bandwidth-efficient and also releases a lot of ALU cycles which can be used for other shader processing.

A similar benefit can be seen for ray-triangle testing and it should thus come as no surprise that adding both of these types of fixed functionality into a GPU significantly speeds up the ray tracing capabilities of GPUs. In its simplest form, this is the foundation which makes real-time ray tracing possible. These fixed-function operations can be exposed as new instructions within the shader programs and this approach forms the baseline of adding ray tracing acceleration into the PC and console graphics solutions launched in 2020, which we describe as a Level 2 solution.



Ray Tracing Level 3

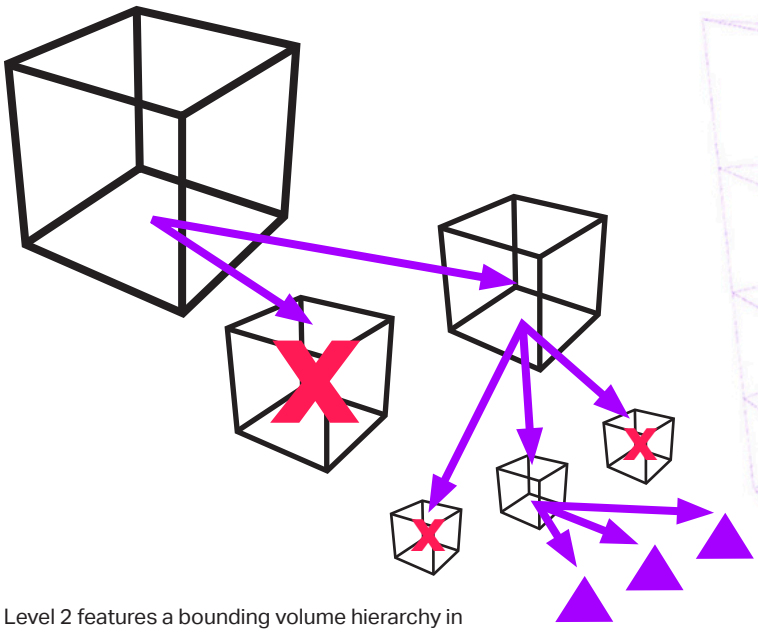
BVH Processing in Hardware

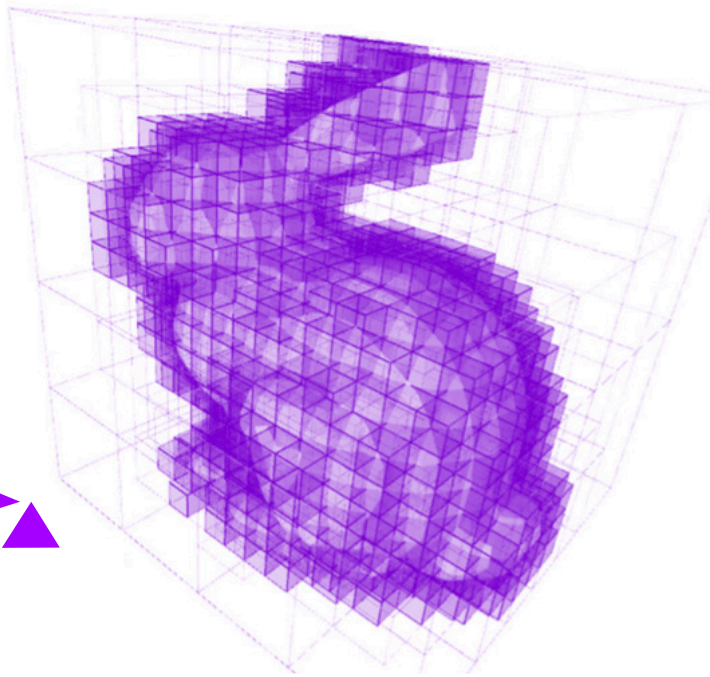
Level 2 solutions have the box and triangle testers, but all other processing remains in the shader code. Specifically, what remains is the traversal of the Bounding Volume Hierarchy (BVH).

The BVH is a hierarchy of bounding volumes (conceptually usually cubes), which subdivide the scene in an aim to cut back the amount of intersection work that has to be done by quickly culling large parts of the box and triangle hierarchy. The BVH means that we avoid testing every ray with every triangle in the scene which would be excessive. Instead, we build a hierarchical structure of volumes which are subdivided into smaller volumes and which ultimately contain triangles, the standard primitive for all GPU rendering. By processing a ray versus this hierarchy, we can quickly reject large amounts of work since if we do not hit a bounding volume, we can also reject the full hierarchy of smaller volumes and triangles within it.

The traversal code used for this is not a great fit for the parallel processing nature of GPUs. As a ray is traced through the BVH data structure the code has to make a lot of decisions (branches/divergence) along the way – e.g. you start at the top and do a box check, based on that check you trigger more box checks until ultimately you hit the triangle level of checks. Making these decisions and triggering instructions from the shader code is possible but not efficient. Ultimately, it is a poor use of valuable shader processing capabilities and is also not a very good fit for the SIMD nature of all GPUs as it's inherently full of branches and decisions, which are never a good fit for a parallel processing architecture.

Hence, a logical next step is to extend the ray tracing hardware to handle the full ray BVH processing workflow, thus offloading ever more cycles from the shader code into dedicated tuned hardware. In this scenario, the walking through the BVH for each ray is now fully managed by dedicated logic which includes the usage of the ray-box and ray-triangle testing units from the Level 2 hardware. In addition to offloading more work from the ALUs, this also adds the benefit that caching and data flow can be much more optimised and wider-parallelism efficiency can be achieved by processing more rays together as they run through the BVH structure.





Level 2 features a bounding volume hierarchy in hardware to speed up ray/box, ray/triangle testing.

Inline ray tracing versus full ray tracing

While this primer is focused on hardware architecture levels there are also differences on the software side with two different levels of ray tracing API functionality. The first, and simpler level, is "inline ray tracing" also known as "ray queries". Fundamentally, as the name implies, this approach to ray tracing sets up a ray with an associated state and then proceeds with a ray query: the ray is traced through the BVH and will report back its result. Essentially, this is exactly what the Level 2 and Level 3 solutions do in hardware - you loop through box-ray tests and ultimately ray-triangle tests and then report back the desired result, which is simplified as a hit or a miss. A simple example of this would be to send a ray query to a light source; if the ray hits an opaque object we know we are in shadow but if the ray reaches the light source we know the pixel is lit, and thus, based on this simple ray query, we have now implemented ray traced shadows:

```
Setup Ray (myRay)
AnyHit = RayQuery (myRay)
If (AnyHit = TRUE)
        Execute Shadow code
Else
        Execute Lit code
```

While the above is simple it's not what most people recognise as ray tracing – i.e. rays bouncing around within the environment to create complex effects as a result of these multiple light bounces. This much more complex form of ray tracing is known as "full ray tracing" and it does exactly that; as a ray hits/misses objects it creates new shader programs which execute.

This concept of a ray bouncing around and thus emitting from a shader program, which hits another object, which in its turn launches another shader program, which in its turn launches another ray which then launches another shader program, and so on, is a concept known as recursion. The result of recursion is a stack of shader programs, and the process continues until somewhere your ray stops bouncing and you wind back your stack to collect all the processing stages of tracing the ray around the scene.

Conceptually it's like this:

```
EmitRay (Ray1)
  Hit Object which EmitRay (Ray2)
        Hit Object which EmitRay (Ray3)
                Hit Object – execute shader
                program
        Execute shader program which takes
        previous hit data into account
  Execute shader program taking both the previous
  two hits data into account
Original program which takes the whole ray data
flow into account
```

This type of recursion is complex since, as you can see from above, the stack of stages has an unknown depth ahead of time as you do not know what objects the ray will or will not hit. This makes it a very dynamic and multi-staged process and each stage requires storage and resources. This is what is known as "full ray tracing" and is significantly more capable, but also more complex, than ray queries. Imagination Technologies, as one of the ray tracing pioneers, enables an architecture which supports both inline ray tracing (ray queries) as well as full ray tracing.

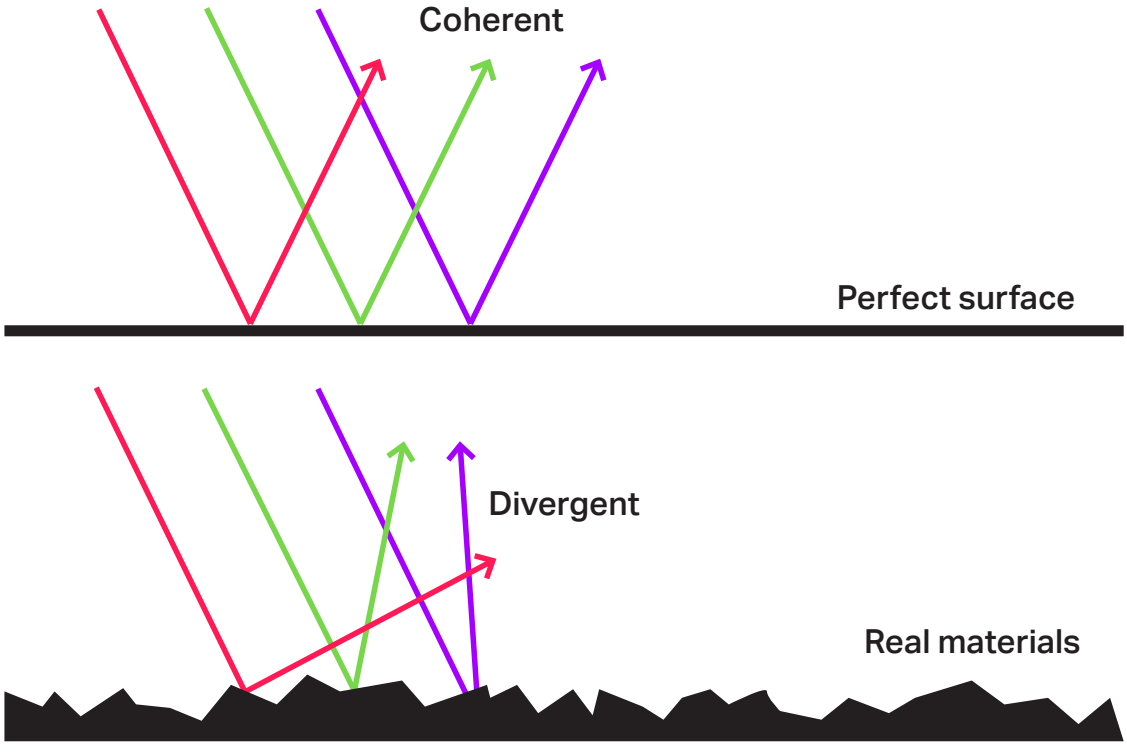
Ray Tracing Level 4
BVH Processing with Coherency Sorting in Hardware

While ray tracing is "embarrassingly parallel" in nature, one of the reasons why real-time ray tracing has taken so long to become practical is that the parallelism is there but it's very often divergent and non-coherent in nature. This can be understood from the below illustration.

In the real world, materials have different properties – some are smooth, but most are rough – and therefore, for realistic surfaces, rays will not be reflected in exactly the same way, but rather bounce in a variety of directions. This results in divergence; e.g. the ray bounces from one pixel to the next pixel resulting in rays going in very different directions. Consequently, the ray will cross the BVH boxes along different paths – thus causing divergent memory accesses – and, logically, rays travelling in different directions will also intersect with different triangles, triggering different shader programs – thus causing divergence in the shader execution.

Divergence is bad for GPUs as while they are great at processing highly parallel workloads their SIMD architectures only makes sense if those workloads are coherent and similar. If each pixel wants to do something different, the tricks upon which GPUs depend for high execution and bandwidth efficiency fail. This means you end up with a brute force approach (i.e. the use of large amounts of ALUs and ray tracing units), is required to compensate as the processing flow struggles to use them efficiently (namely high peak throughput count on paper, but poor utilisation and thus low throughput numbers in real-world use).

Now, while rays from one pixel to the next may be divergent this does not mean that there is no "coherency" among the soup of rays that are bouncing around. Again, this is best illustrated in the image on the next page.



Ray Tracing Level 4
Continued...

The reflective shape below shows hidden coherency in the rays, which reflect from this object e.g. you can see that the person wearing yellow is reflected many times, meaning those rays go into the same direction and are, indeed, coherent. Even more, if we can group those rays, they will follow a similar path through the BVH providing us with a high rate of cache hits and data re-use. They will also ultimately hit and intersect with the same triangles and would likely also execute the same or similar shader programs, consequently delivering high efficiency in traditional parallel GPU ALU pipelines.

What we need therefore is a way of capturing this "hidden" coherency to deliver this efficiency improvement. Imagination did so with its 2014 PowerVR Wizard GPU architecture, which pioneered real-time ray tracing within a modern GPU architecture and introduced concepts such as hybrid rendering (mixing traditional and ray traced rendering), by included a coherency sorting engine.

In terms of innovation, the Coherency Engine is, in many ways, the equal/sequel to tile-based rendering, which Imagination pioneered in the late 1990s in its PowerVR GPUs, which today is embraced by all modern GPU architectures.

The Coherency Engine finds and sorts coherent rays in a scene and then packages them up for efficient processing on the GPU.

Ray Tracing Level 4
Continued...

Tile-based rendering also does a coherency sort: following geometry/vertex shader processing the triangles are sorted into tiles, which later ensures that each triangle's pixels are processed in coherent pixel groups per tile. This means that all processing can stay in tile memory on chip, thus reducing bandwidth and improving efficiency

The Coherency Engine for rays achieves the same, sorting rays into coherent bundles which can share memory accesses effectively guaranteeing perfect cache hits and thus using less bandwidth. Of course, this also helps with power efficiency as data movement is a big consumer of power. The hardware coherency gatherer also helps achieve higher performance efficiency as the SIMD engines will see high execution efficiency, as these bundles will hit similar triangle/objects. It is this massive leap in efficiency that makes this type of architecture a Level 4 ray tracing solution. For power and bandwidth-limited designs, for example in smartphones, this will be absolutely essential to make real-time ray tracing a practical reality.

What about MIMD architectures?

Some more exotic and mostly historical ray tracing "solutions" have been based on a MIMD (Multiple Instruction/Multiple Data) architecture and they claim this solves the coherency issue with ray tracing. However, MIMD is not really a solution – it's a form of accepting defeat that there is no coherency in the processing and adapting the ALUs to be more effective in executing such divergent workloads. The problem with MIMD, and its universal lack of success as a processing solution, is that MIMD is expensive in silicon complexity and size. With MIMD, you need to generate ALUs that are all able to execute different instructions and which also share no data with any other ALUs. It's akin to creating a single-threaded GPU where each ALU carries all the overhead to execute unique instructions with unique data fetches.

As such, MIMD is not an elegant solution, it's a very expensive way of handling a lack of coherency. Furthermore, while a MIMD approach can handle the divergence in execution it does not solve the fact that the data is still divergent – rather now the problem of divergence has moved from the ALU into the memory subsystem. While modern memory technologies are very fast and sustain very high data rates, they again only manage this when the data fetched is coherent; e.g. you read continuous data. When it is random – as per MIMD – fetching data from memory is orders of magnitude slower. Hence MIMD is not a real solution for real-time ray tracing.

Ray Tracing Level 5

Coherent BVH Processing with Scene Hierarchy Generator in Hardware

Up to now most of our focus has been on accelerating the tracing of rays through the BVH structure but we have ignored the actual creation of the BVH itself. The actual approach and decisions on how to create this structure can be varied: e.g. how deep is the hierarchy, and when do you split or subdivide further, etc.

The process and decisions can even be dynamic and based on heuristics derived from the workload. Also, the bandwidth consumption and accuracy of the structure can be traded against each other. Up to Level 4, the creation of this structure is done in software, meaning that it is either done using the CPU and/or using a GPU compute path. With a Level 5 solution, we move this BVH creation process into dedicated hardware, which is optimised to work with the BVH traversal approach enabled in Level 3 and 4 solutions.

By building the BVH using dedicated hardware we are further offloading the shader core of work and executing this work using more power and processing efficient logic. Additionally, for PowerVR architecture, the Scene Hierarchy Generator (SHG) is integrated inside the GPU, which means the data flow can go direct from the traditional vertex and geometry processing phases into the SHG block to generate the BVH in memory. This process can even be coupled with traditional geometry outputs such as a tile pointer list for tile-based rendering, thus enabling hybrid rendering in the most efficient way.

A dedicated fine-tuned block is fast and efficient, and it means this level of ray tracing accelerator can handle much higher numbers of dynamic geometry, thus higher scene complexities including fully dynamic scene including many complex animated highly detailed objects.

This BVH hardware builder, such as the Imagination Scene Hierarchy Generation block, could theoretically be added to lower efficiency ray tracing levels. If this was done, in terms of the levels it would there be indicated by a “plus”. In this scenario, a Level 2 ray tracing solution with a BVH builder in hardware, would, for example, be a “Level 2 Plus” solution and, of course, not a level 5 solution.

Images such as these, featuring realistic, dynamic shadows, could be possible in real-time on Level 4 ray tracing hardware.

Summary

While it is early days for ray tracing, the realism it provides and the development efficiencies it brings means that eventually an increasing percentage of graphics processing will be done this way. It will surely revolutionise multiple fields, from architectural visualisations and engineering prototyping to TV and movie animation, and of course, for gaming. The technique has even been touted for uses other than just traditional graphics such as for collision detection, physics acceleration, sound processing or volume rendering.

However, the fact is that Moore's Law is coming to an end and the industry can no longer rely on hardware becoming exponentially more powerful every two years. Therefore, to continue to advance ray tracing, particularly for power-constrained mobile platforms, efficiently designed fixed-function accelerator solutions will be increasingly essential.

Imagination Technologies is one of the pioneers of ray tracing and in 2016 demonstrated PowerVR GR6500, a Level 5 real-time ray tracing test board. In the near future, we will also be launching PowerVR architecture-based GPUs featuring high-level hardware ray tracing acceleration, so whether for mobile platforms or high-power, desktop/server or Cloud, you will have access to an advanced, efficient solution that will give your offering a true competitive edge.

Global illumination using ray tracing within a mobile power budget

www.imaginationtech.com