*World of Tanks** 1.0+: Enrich User Experience with CPU-Optimized Graphics and Physics

Authors: Philipp Gerasimov, Senior Game / Graphics Application Engineer, Intel; Bronislav Sviglo, World of Tanks Rendering Team Lead, World of Tanks Team, Wargaming.net; Aleksei Fedotov, Senior Software Engineer, TBB Team, Intel

Introduction

During the last decade, computer games have significantly improved visual quality by utilizing continuously developing GPUs. Games now employ real-time graphics that rival the kind of cinematic rendering typical of Hollywood films, and high-performance GPUs have become commonplace in end-user PC platforms. With games coming closer to photo-realistic GPU-accelerated graphics, developers are now seeking the next frontier: the next step forward that will enrich the gamer's experience, improve performance, and create more immersive and dynamic worlds.

To achieve this, developers should look at modern CPUs. This article (based on the joint Intel/Wargaming* presentation at GDC 2018 in San Francisco) will explore how one of the biggest game developers, Wargaming, uses the innovative features of modern CPUs to re-architecture their engine and attain new levels of performance and gaming experience. With special focus on *World of Tanks**, we'll look at the creative ways in which Wargaming employs CPU multi-core and CPU single instruction, multiple data (SIMD) capabilities to significantly enhance the immersive experience of the game. Using the Intel® Threading Building Blocks (Intel® TBB) tasking system as a foundation, we discuss how it supports the game engine's multi-threaded foundation, use of Havok Destruction*, novel and physically correct tank tread simulation and concurrent rendering. In this article, we will concentrate the discussion on the ways in which CPU multi-core threading has been used to optimize *World of Tanks*.

Wargaming* and *World of Tanks**

Founded in Minsk, Belarus in 1998, Wargaming grew to become one of the biggest game developers in the world, mostly thanks to *World of Tanks*. The free-to-play, online multiplayer wargame allows users to battle each other with 20thcentury tanks, tank destroyers and other heavy armored vehicles. Since its initial, wide release in 2010, the game has attracted over 140 million registered users worldwide, and the company keeps bringing in new features, maps, tanks and other in-game content with regular updates, making the game one of the top massively multiplayer online (MMO) games on the market. This March, Wargaming released the biggest update to *World of Tanks* since its first release—*World of Tanks 1.0*. This brought a new engine, completely new graphics renderer, fully redesigned high-resolution maps, and many other great new features.

Intel and Wargaming have been working closely together for some years, on issues ranging from the engine architecture to day-to-day optimizations, making sure the new version of *World of Tanks* is optimized for a broad range of computers and serves as a solid foundation for future enhancements.

World of Tanks 1.0

The *World of Tanks 1.0* update took around four years of development, with the engineering team taking a giant step forward in re-writing the BigWorld engine into their new proprietary Core engine, producing a modern, powerful and flexible solution to run the most popular games. The new Core engine includes significant architectural changes in rendering and physics engines to allow designers to dramatically improve visual quality with more detailed maps, add new realism in gameplay with water simulation, object destructions and be prepared for even more features, and all without a dramatic performance drop. All of that required significant work on GPU and CPU optimizations.

Figure 1: *Same in-game location in World of Tanks 0.9x and 1.0.*

Modern CPUs

The CPU is the brain, and heart, of every computing device. Since their introduction, CPUs have been constantly evolving—and this process will never stop. Generations of Intel® processors have brought new features and improvements in performance, memory, cache, latencies, instruction sets, etc.

One of the major areas of improvement is related to Parallel Computing, and consists of two major parts: Multi-Core, and Vector Instruction sets.

Multi-Core

Almost every computing device nowadays has a multi-core CPU—mobile phones, gaming consoles, desktops, and notebooks. There is a big reason for that: adding more cores is the most efficient way to increase performance with current technological processes. And this will only continue in the future. So, it's critical for programmers, including game developers, to utilize all the available CPU cores to get access to all the computational resources of modern CPUs.

Vector Instructions

Vector instruction, or SIMD, allow developers to process multiple data in a single instruction, like vector addition or subtraction. There are multiple vector instruction sets on PC platforms, including Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX, Intel® AVX2), which are also available on the latest gaming consoles. These instruction sets can be widely used in game development, significantly improving performance in physics, rendering, and AI engine sub-systems.

Let's look at the average computer system from 2017 (left) and 2016 to see what changes have happened, and what that means for developers and games.

CPU	Intel® Core™ i7-4790K 2017	Intel® Core™ i7-4650U 2016
# Cores	4	2
# Threads	8	4
Base Frequency	3.60 GHz	1.70 GHz
Max Frequency	4.00 GHz	3.30 GHz
Instruction Set Extensions	Intel® SSE4.1 Intel® SSE4.2 Intel® AVX2	Intel® SSE4.1 Intel® SSE4.2 Intel® AVX2

Table 1: *Desktop processor SKUs.*

For quite a long time, CPUs in notebooks predominantly had two physical cores (four threads with Hyper Threading). Starting from late 2017, many mobile SKUs have four physical cores (eight threads with Hyper Threading).

CPU	Intel® Core™ i7-8700K 2017	Intel® Core™ i7-8650U 2016
# Cores	6	4
# Threads	12	8
Base Frequency	3.70 GHz	1.90 GHz
Max Frequency	4.70 GHz	4.20 GHz
Instruction Set Extensions	Intel® SSE4.1 Intel® SSE4.2 Intel® AVX2	Intel® SSE4.1 Intel® SSE4.2 Intel® AVX2

Table 2: *Notebook processor SKUs.*

On desktop PC SKUs, late 2017 also represents an increase in the most popular gaming SKUs. Intel® Core™ i7 processors now have six cores.

This increase in the number of cores gives developers opportunities to significantly improve performance in their application, and add functionality—including new visual effects, physics, and sound.

New Versus Old Hardware Configurations

While new hardware keeps moving forward with better performance and added capabilities, gamers around the world are using an extremely wide spectrum of computer configurations of varying ages. This creates a challenge for game developers to enable new features, while still being able to support users with older PCs.

With developers starting to architect their engines to take advantage of new hardware with multiple cores, while still being able to support existing configurations, it's important to develop or select the right threading framework. This subsystem will make the utilization of available computing devices easier and efficient.

Intel® TBB is a programming API that helps developers to take advantage of multithreading, and even a heterogeneous environment, in their applications by efficiently exploiting available computing resources. Through its 10+ years of evolution, the Intel TBB library introduced several high-level interfaces, which were specifically designed to program software executing in parallel. See figure 2 for a brief overview of the Intel TBB hierarchy.

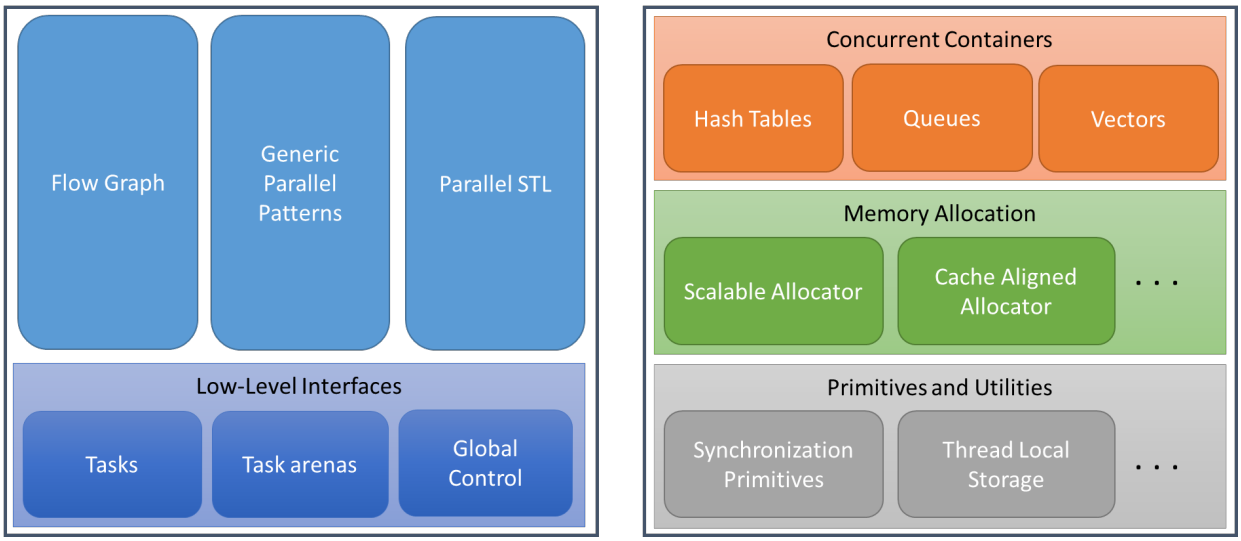

Figure 2: *Intel® Threading Building Blocks library components. Left: high-level and low-level tasking API; right: various concurrent containers and synchronization primitives.*

Besides tasking API, which allows executing arbitrary functions concurrently, there are several useful entities for concurrent environments, such as containers, synchronization primitives, and memory allocators, which do not require linking with the Intel TBB binaries. Those are shown on the right half of figure 2. Perhaps, one of the most useful features of the Intel TBB library is its support for nesting of parallel constructs. The library was designed with nesting support in mind from the very beginning, and hence does it efficiently. For example, one can call a generic parallel algorithm such as "*parallel_for*" from inside another invocation of generic parallel algorithm, or flow graph node, or whatever other construct the library has—without being afraid of oversubscribing the system. Figure 3 demonstrates an example where nesting support could be beneficial.

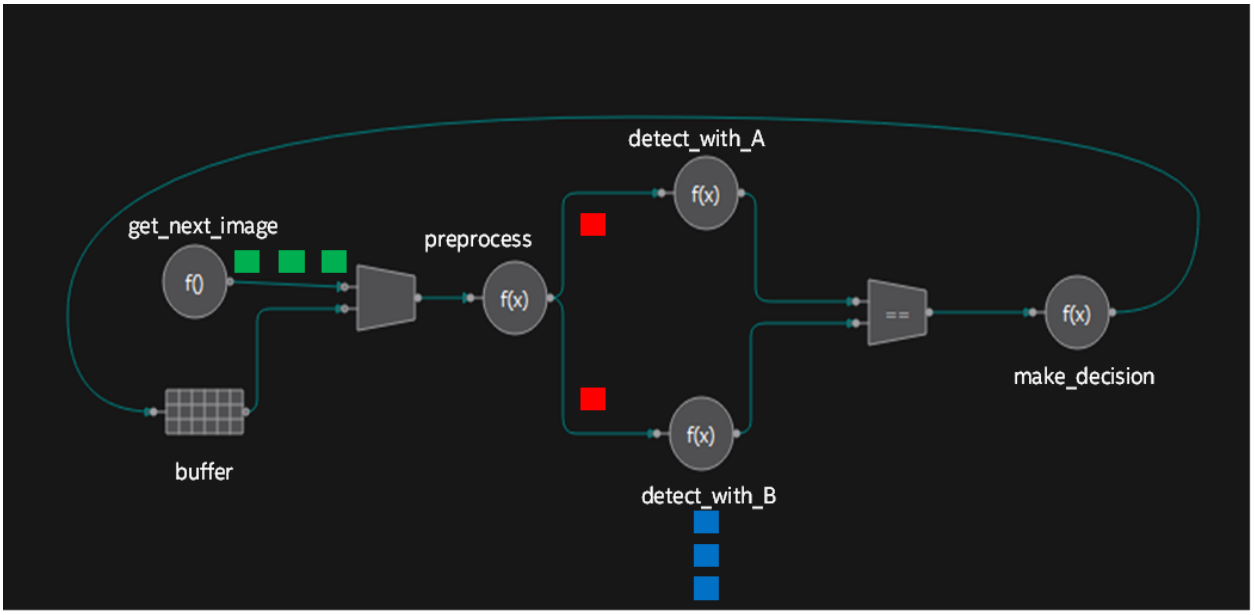

Figure 3: *Feature detection Intel® TBB Flow Graph that expresses three patterns to parallelism: pipelining (green boxes), functional parallelism (red boxes), and data parallelism (blue boxes).*

Readers who want to explore more information about Intel TBB—including product news, the library forum and history—can go to the product page at Intel.com, or visit the dedicated site threadingbuildingblocks.org. Articles about usages of the library, and how it helped in resolving various multi-core programming problems can be found in The Parallel Universe magazine. The magazine's special edition, issued in June 2016, was devoted to the Intel TBB library and its evolution.

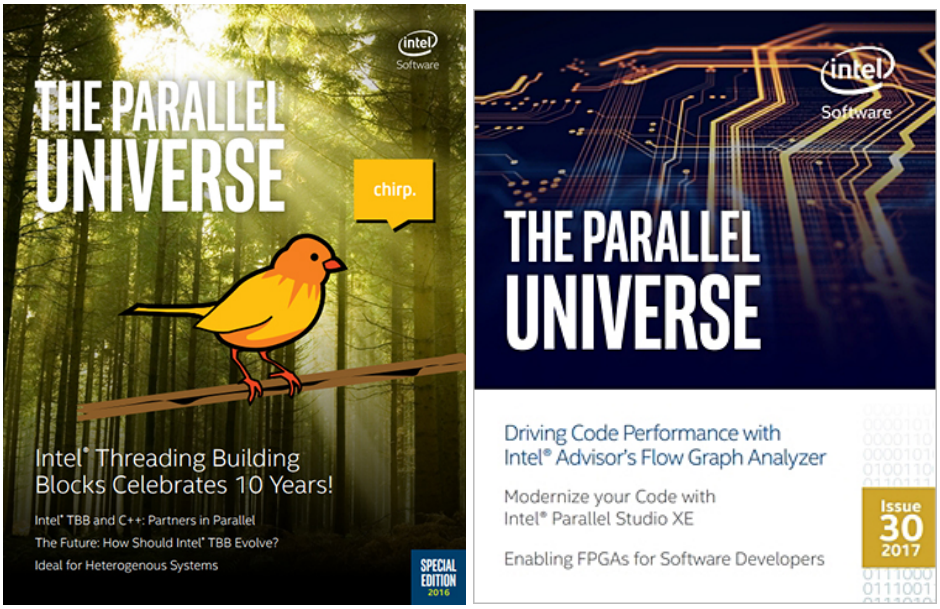

Figure 4: *The Parallel Universe digital magazine.*

For Wargaming, engineering team power, flexibility, ease-of-use and maintenance were the key factors for choosing Intel TBB as the primary threading API for *World of Tanks 1.0,* and projected future versions.

Since 2016, Intel and Wargaming have been working together to integrate the API into *World of Tanks*' core engine, by re-thinking engine structure and execution models, choosing the right algorithm to have scalable and efficient solutions for future game versions, and the hardware platforms that will support them.

Havok Destruction*

The creation of more dynamic in-game worlds is one of the most important tasks for game developers. Even with photo-realistic rendering, the game can still feel unnatural if it's not dynamic—if physics effects and the environment look static. Destructions can add dynamism to game environments, so *World of Tanks* is the ideal case for examining such functionality.

Before update 1.0, destructible physics were quite simple, realized via simple particle system with following model change. The destruction wrought in *World of Tanks 1.0* is powered by Havok Destruction is part of the powerful and popular game physics middleware, Havok Physics*.

The team started integration with the default Havok* threading solution—Havok's own job manager:

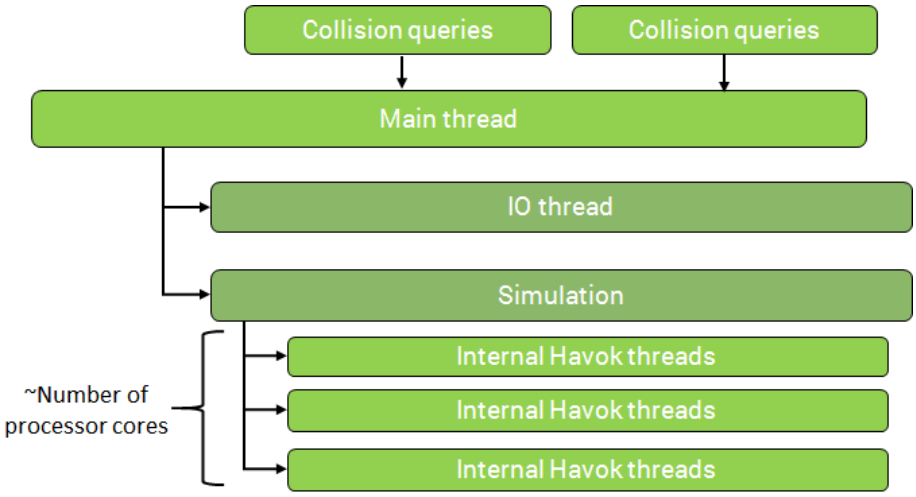

Figure 5: *Havok Destruction threading.*

The game also has special set of collision geometry, to improve performance:

Figure 6: *Different geometry sets for Collision and Destruction.*

That solution worked pretty well, and this scheme was released in *World of Tanks 1.0*. Adding more multi-core oriented features threw up one potential issue: having two different multi-threading job managers. Havok Destruction's internal system requires its own thread-pool allocation, and these threads could compete with Intel TBB's thread pool, effectively oversubscribing the system.

As Intel TBB was selected as the primary threading framework, it would be logical to wrap all the effects and subsystems in the game into Intel TBB entities, so that Intel TBB manages their scheduling on available cores. That resulted in the collaboration between Wargaming, Intel and Havok to enable Intel TBB as an alternative to Havok's threading layer. Which allows the Intel TBB to map Havok Destruction jobs onto Intel TBB threads, without causing any composability problems such as system oversubscription—particularly important on PCs with a relatively small number of CPU cores.

Enhanced Tank Treads

Tanks are heavy motorized vehicles that share one distinctive feature—caterpillar treads. Treads are a continuous track driven by two or more wheels, which makes them a great subject for physical simulation. If done right, this improves immersion and experience for the gamer—particularly hardcore fans who have educated themselves on how each part of the tank should look and behave.

The *World of Tanks* team decided to develop their own tread simulation technique—sophisticated, scalable, powered by the spring chain physics model and producing realistic visual results (see figure 7).

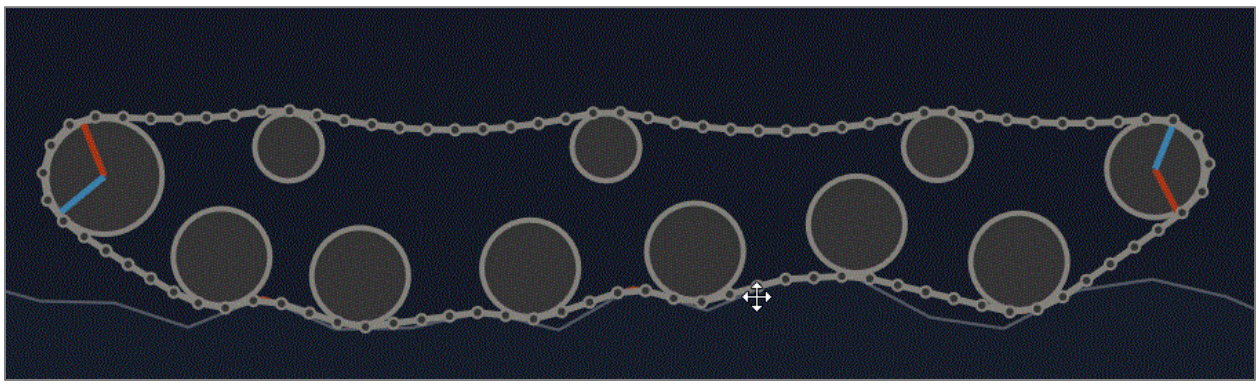

Figure 7: *2D tank tread simulation.*

Previously, *World of Tanks* had two different varieties of simulating treads:

- Skinned mesh. The simplest and fastest technique, but visually static. The dynamic effect was created by moving textures.
- Spline simulation, where movements and integration was achieved by simulating tread with spline control points, and moving them to simulate interaction with terrain. The visual result was good, but the process was very complex for artists to set up for all the different tank models. In addition, collisions with the terrain and in-game objects were not realistically simulated.

The new spring-chain physics solves most of these problems, and gives outstanding visual results:

Figure 8: *Tank tread simulation: screenshots showing detailed terrain collisions (left) and gravity (right).*

Now the tread is divided into four parts: front and back, top and bottom:

Figure 9: *Tank tread simulation: four-part setup.*

The engine ray-casts the area underneath the tank, and simulates collision with a dynamically created height field.

Physical simulation of each tread segment allows a significantly improved visual result, where simulated tread correctly interacts with tracks, terrain and tank geometry. Treads are correctly reacting to tank movements, letting gamers feel the physical material and mechanics. It also significantly reduces artist work on setting up each tank model, as new threads work “out of the box” for every tank without manual hard-tuning.

For each tread, Intel TBB creates its own task, which is later executed on one of the available worker threads:

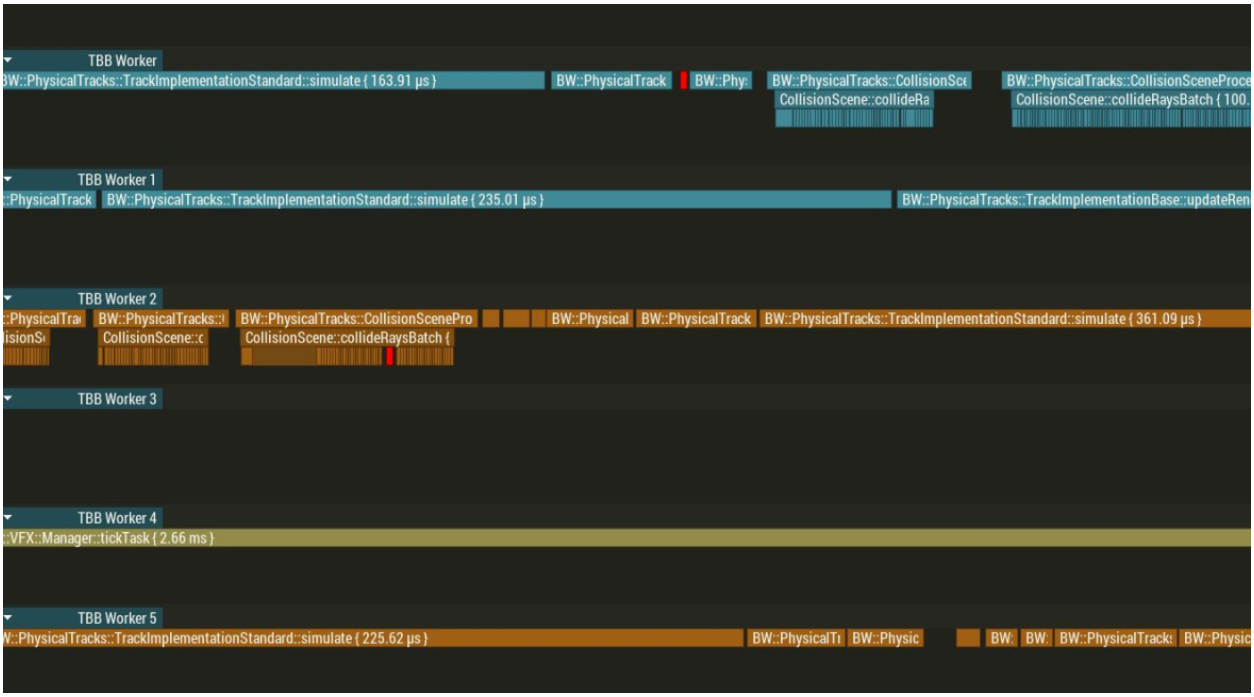

Figure 10: *Mapping of simulation tasks onto Intel TBB worker threads.*

Figure 10 shows that Intel TBB tasks take around 250 microseconds, which is a huge amount of work per task. Therefore, depending on the algorithm scalability, such tasks could be split even more, potentially giving better CPU utilization, and improving overall simulation performance.

Concurrent Rendering

One of the most important performance issues on the CPU side is the limitation of engine performance on the main thread (which does main game logic, prepares work for the rendering thread, or issues rendering calls itself). The result is that even when CPU has multiple cores, and the game is able to create multiple threads for various subsystems or jobs, the main bottleneck is the single core frequency where the most work happens.

That's why modern rendering APIs such as DirectX* 12 and Vulkan* were architected with multi-core nature in mind. That allows 3D applications to prepare and issue rendering calls from multiple threads, effectively distributing the work across available CPU cores, minimizing the bottleneck on the main thread. As *World of Tanks* currently uses DirectX* 11 API, however, the engineering team came up with a creative approach that allowed them to accomplish this with the existing API.

The initial version of *World of Tanks* used the BigWorld engine, and it's rendering pipeline—which was a single-thread DirectX 3D* 9 application:

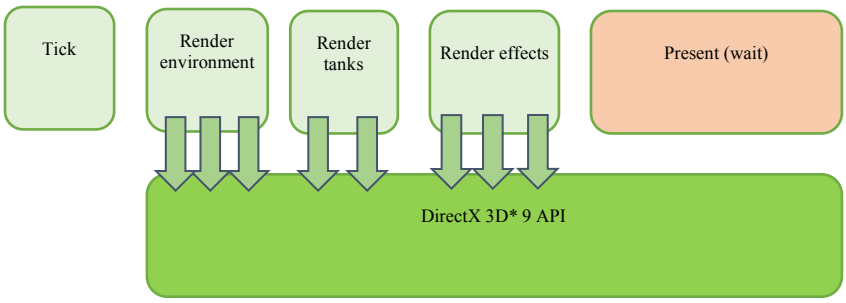

Figure 11: *Initial rendering pipeline, c.2010.*

The first step was to de-couple rendering from the rest of the game and introduce threading. The team came up with abstract rendering interface (ARI) methodology, and separated all API-specific rendering code into a separate rendering thread. Patch 0.9.15 and Core Engine 3.0 (August 2016):

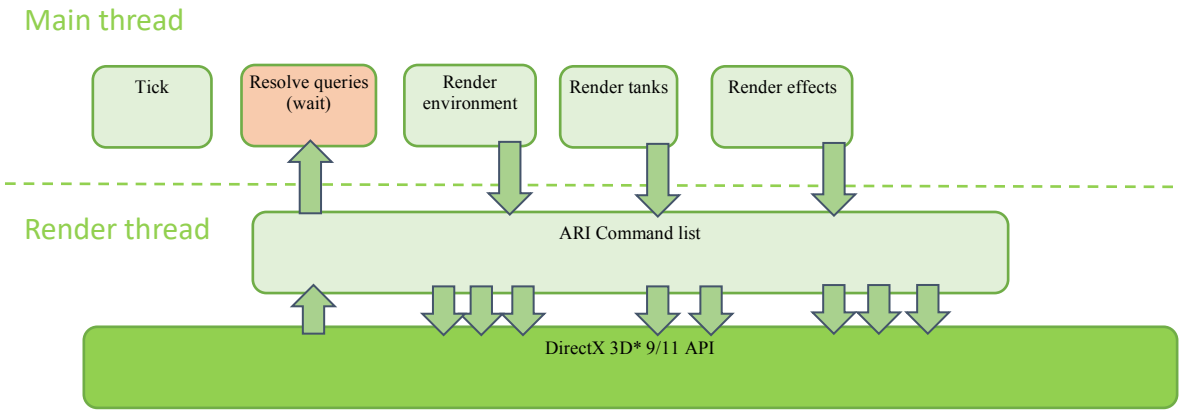

Figure 12: *Rendering pipeline v0.9.15, August 2016.*

Around the time this work was underway, new rendering APIs—such as Mantle, DirectX 12, and Vulkan—started to appear, and the team initially architected ARI with the same metaphor. Basically, ARI gathers the render commands into the software command buffers. Then, those gathered command buffers were submitted to a render thread, which in turn was in charge of processing them and making direct calls to an appropriate graphics API (at that moment it was either DirectX* 9 or DirectX 11).

The important part of ARI is the Wgfx intermediate compiler—a faster, multicore aware alternative to DirectX 3D effect framework, with lots of under-the-hood optimizations, and a customized API. Despite the engine now having a separate render thread for processing software command buffers, ARI render commands were only issuing in the main thread, however. But even that gave up to a 30% boost in performance.

ARI can be quickly explained in the following terms:

- **Command list.** Explicitly describes what should be done by the rendering thread. The command list contains a list of render commands—such as draw, clear, update, copy-resource—that are supposed to be executed later on the render thread.

- **Device interface**. The “driver” between ARI frontend and graphics API (such as DirectX 9, DirectX 11 or DirectX 12).
- **Resource**. Any resource-like buffer: texture, query, graphics pipeline state, etc.

ARI made it possible to start thinking about concurrent rendering and where the submission of work is distributed across multiple threads, minimizing the amount of work on the main thread.

The command list must contain plain, old data (POD) commands, independent from each other and with all the information required to be issued later in the render thread. Since command lists do not depend on each other, their preparation can be done in parallel. The correct result can be guaranteed by the order of submitting the command list into the separate render thread.

Examples of commands are: draw, clear or query, and so on.

However, there are some kinds of commands—such as update or copy-resource—that require having manual memory management and lifetime control of the content the user provides to be a part of these commands.

So, when uploading a texture on GPU, developers usually don’t want to copy the whole texture content right into the command. Instead, they provide just a pointer or link on the memory to copy from—and guarantee that this memory remains valid till the end of this frame.

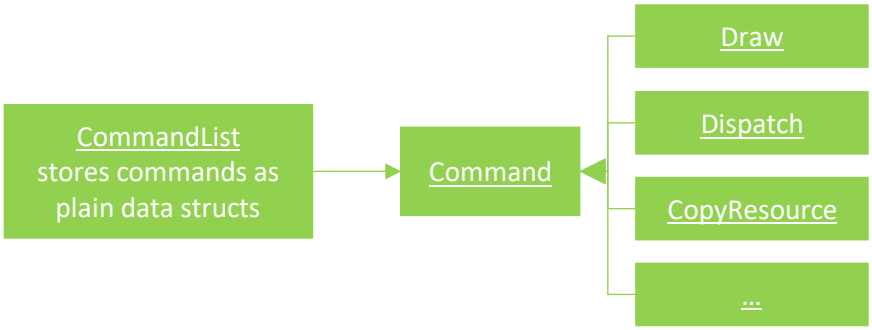

Figure 13: *Software “Command Lists” structure.*

The ARI interface for device access is divided into three categories based on the access pattern:

- Free-threaded, or thread-independent interface, is used for operations which may be executed at any time from any thread. With this interface we can perform resource creation and deletion, receiving of an adapter state, and so on.
- Single-threaded interface is mostly related to the command list operations, such as filling list with commands, compiling it to native commands in case of DirectX 12, and so on.
- Creation thread-only interface is a legacy item, and is about limitations of existing old graphics API—DirectX 9, DirectX 11 for example—to perform special operations, like present or reset on a render device.

With the already integrated Intel TBB API, the threading model looks like this:

- Intel TBB flow graph, which manages dependencies between render tasks and defines the actual order of submission of gathered command lists into the render thread.
- Render tasks, which represent independent command lists for each render sub-system of the engine.
- Separate render thread that consumes software command buffers and makes native calls to graphics APIs.

On the high-level, render frame is managed by Intel TBB flow graph which acts as a render task scheduler.

1) Flow graph consists of number of nodes and edges.
2) Each node of the graph is a big chunk of render work within one sub-system.
3) Edges form dependencies between different sub-systems or key renderer stages.

Figure 14 shows various examples of the render nodes, so you can have an idea what they might be.

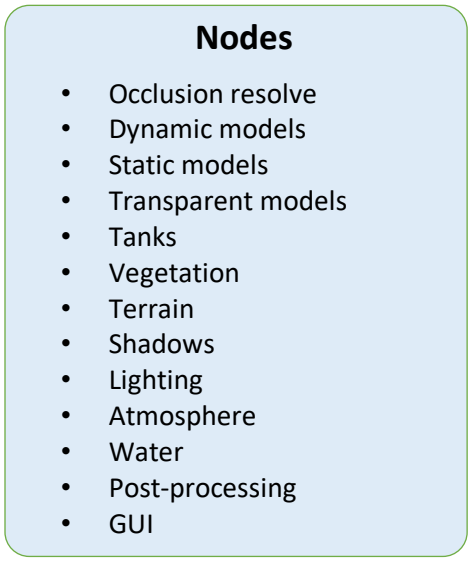

Figure 14: *Render Nodes*

The illustration below shows an example of how a high-level frame render graph may look. Having such high-level architecture of the frame is very helpful for everyone in the team, so one can quickly understand what the actual flow or order of execution is and what sub-systems depend on each other. Moreover, this code is easy to modify and optimize because all high-level information about coarse-grained tasks are located in one place. Task-based or functional parallelism is easy to express using the Intel TBB flow graph because all one needs to do is to merely wrap up each task into a flow-graph node and add dependency between them.

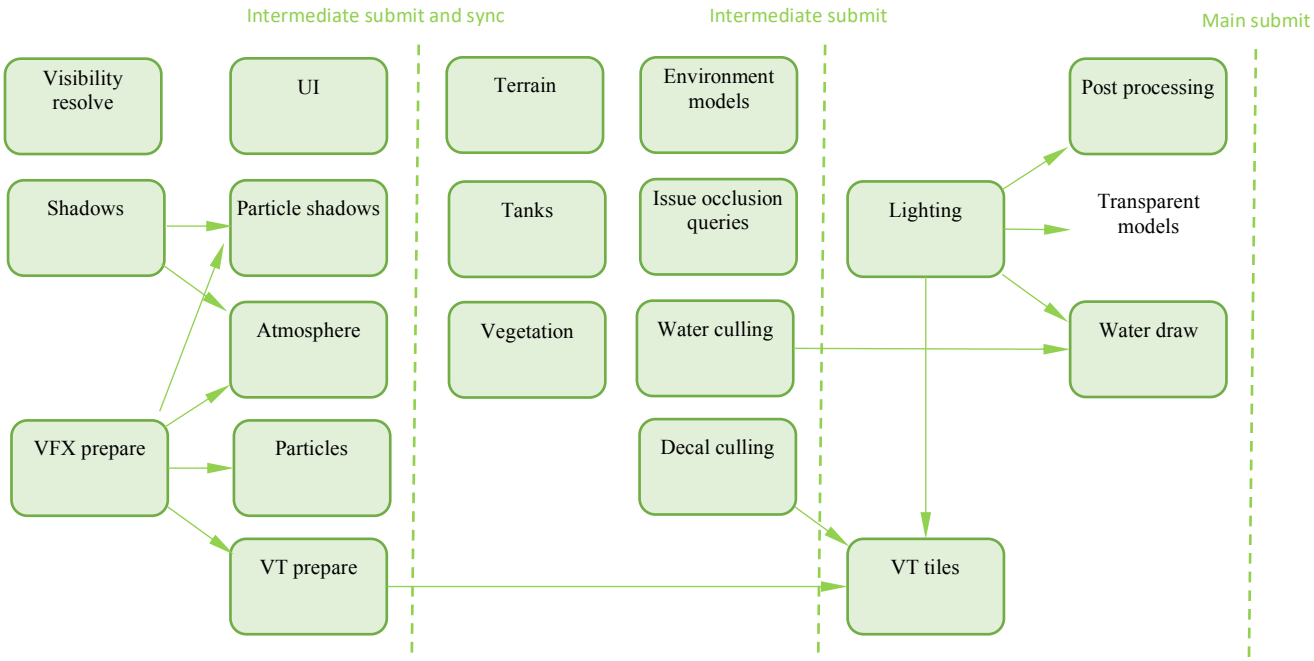

Figure 15: *High-level frame render graph.*

As previously mentioned, the current core engine uses Intel TBB flow graph to gather command lists for various sub-systems in parallel, and then flush them in a predefined order into the render thread. Unfortunately, if we do only one flush to the render thread per frame, it will make the render thread stay idle. So, we need to perform intermediate flushes to give the render thread some work.

So, after describing ARI and concurrent rendering models let's look how it works in the game and how it changes the performance.

The original frame looks like (this test was done on 4-core Intel® Core™ i5 processor) figure 16.

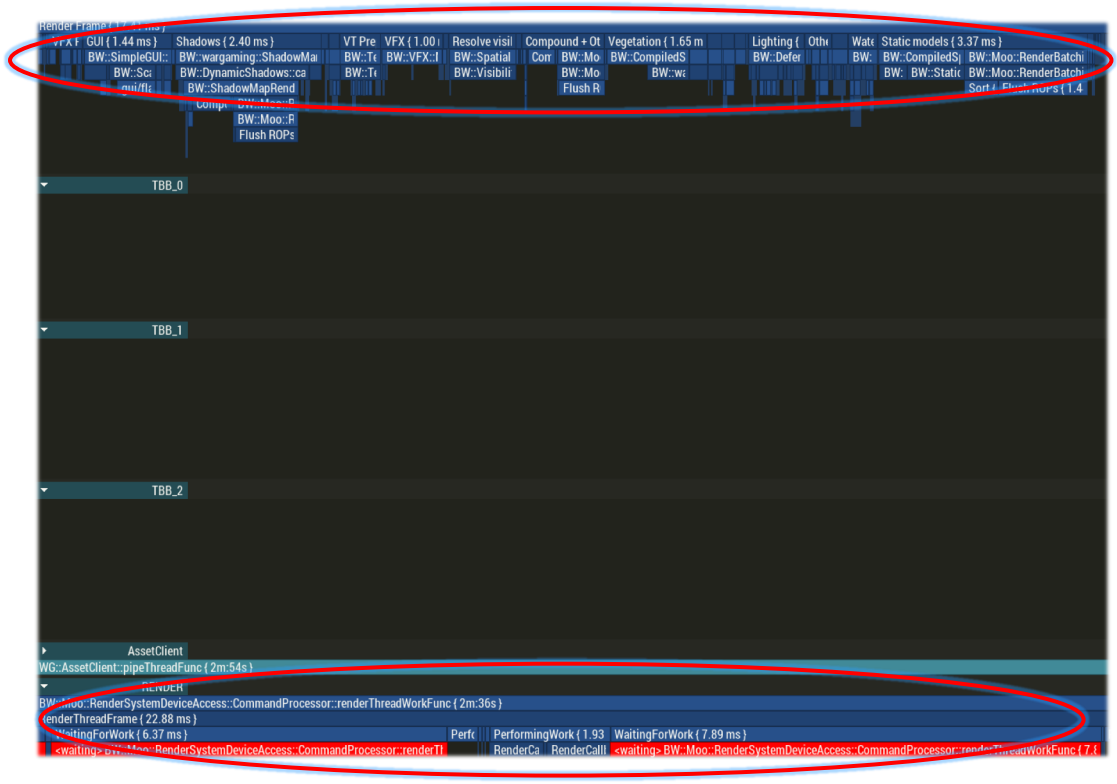

Figure 16: *Telemetry*- rendering frame with no concurrent rendering, 4-core Intel® Core™ i5 processor.*

Figure 17 shows the main thread that gathers all the commands into the number of independent command lists, one-by-one. And it takes a lot of time to do it (about 17ms). Figure 17 also shows the render thread, which is actually processing the previous render frame. You may also notice that render thread is idle for a large amount of time, which is due to the main thread not being able to generate render commands fast enough. Remember to feed the render thread with work in a timely fashion.

After enabling concurrent rendering, and adding parallel execution for resolve visibility, static models, tanks, lighting, water, vegetation, post-processing, particles and shadows, it looks like this:

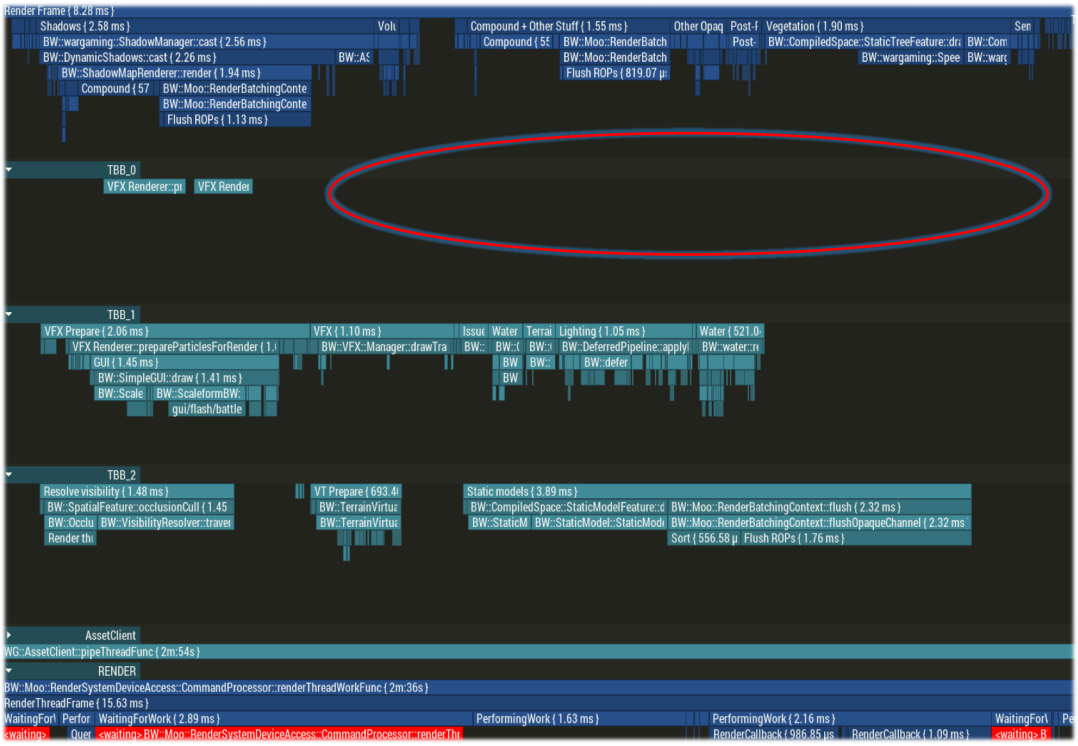

Figure 17: *Telemetry* - rendering frame with concurrent rendering enabled, 4-core Intel® Core™ i5 processor.*

Overall frame-rate is doubled, and it now takes just 8ms per frame. But there is something curious about this picture: a bubble where no work happens on one of the worker threads. This bubble shows that we have to be careful with two things:

1. The size of each individual render task
2. The set of dependencies between this task and others

Neglecting these could lead to big inefficiencies.

In general, as we can see, functional parallelism is easy to implement. It requires minimal effort from the engineering team to write and support such code, and because they can take a high-level view of our rendering pipeline, they can easily make modifications to increase parallelism in our engine.

Unfortunately, high-level frame render graph requires us to have big chunks of work which is not always optimal because not all tasks have the same size or the same execution time. Not all tasks may be executed in parallel because some of them may have dependencies on each other. So this approach quickly leads to the critical execution path. You may think about minimizing the amount of dependencies between tasks, or subdividing the task into smaller ones—but that may significantly reduce the readability of the code. How can we improve performance while still keeping our code readable?

Before, we were using only functional parallelism, but there is also data parallelism to consider. What we can do is exploit data parallelism inside each sub-system independently, and be sure that Intel TBB is

going to do the rest. Thus, we still have an easy to read and maintain frame render graph on the outer-level, and each sub-system may produce additional tasks on the inner-level to better utilize all available CPU cores.

For example, let's look how concurrent rendering works on more powerful CPUs, like an Intel® Core™ i7 processor with 6 physical cores.

Here we have concurrent rendering off with average frame time 12ms:

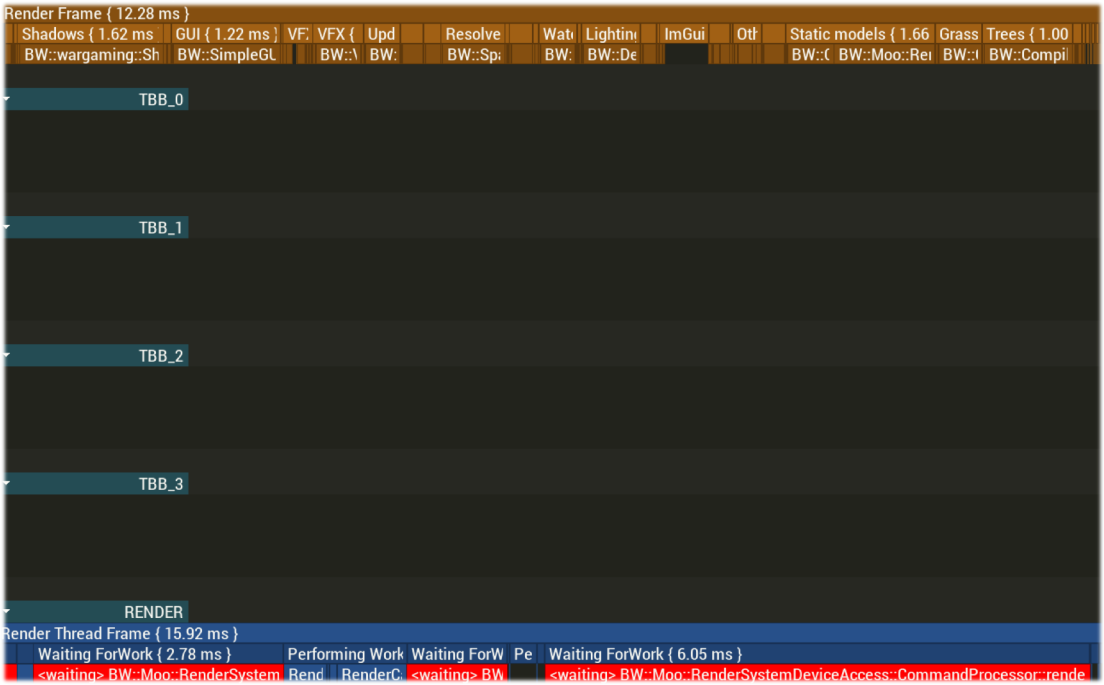

Figure 18.1: *Telemetry* - rendering frame with no concurrent rendering on a six-core Intel® Core™ i7 processor.*

And with concurrent rendering enabled:

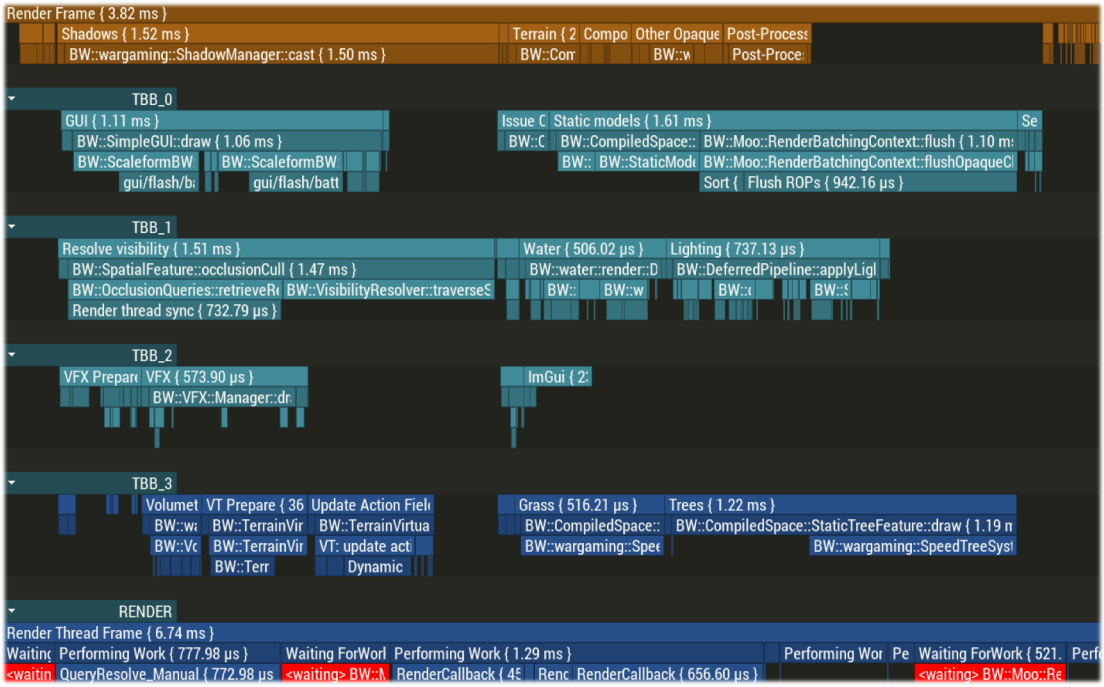

Figure 18.2: *Telemetry* - rendering frame with concurrent rendering enabled on a six-core Intel® Core™ i7 processor.*

The frame time now is only 4ms, which represents almost x3 performance gain! There are still bubbles on some threads, but CPU utilization is much better, so is the GPU utilization and overall application performance.

In summary, concurrent rendering allows for significant reductions in time spent on command generation. Despite the fact that a parallel command generation was added, the code is still simple, and easy to read and modify. Released/freed CPU time may be used for other awesome features like tank treads, more objects on the scene, or even more destructions.

There is still a lot of work to be done, the renderer may be subdivided into even smaller tasks, and each task in turn may be split into sub-tasks by using parallel algorithms. In addition, be very careful with the command submission pattern on outdated graphics APIs, and always have enough work for the render thread, so it does not stay idle.

DirectX 12 and Vulkan allow for the processing of commands right inside the generation thread, which the *World of Tanks* team plans to exploit in the future, to reduce pressure on the render thread.

Conclusion

This article shows multiple examples of a smart approach to using both CPU and GPU enables improved visual fidelity, with new and visually interesting features that will be enjoyed by gamers.

The CPU and GPU are both evolving to provide more computational resources for developers. It is important for developers to consider both the CPU and the GPU when seeking to improve visuals and improve rendering performance. The *World of Tanks* engineering team continues to explore and develop new features. Havok Destruction and Tank Treads features are already available in the released version of the game, and the concurrent rendering is in its final stages of development.