Parallel Techniques in Modeling Particle Systems Using Vulkan* API

Example of using GPU- and CPU-based solutions utilizing Vulkan graphics and compute API

Tomasz Chadzynski
Integrated Computer Solutions Inc.

Table of Contents

1 Introduction

Parallel processing has become one of the most important aspects of today's programming techniques. The shift in paradigm forced by the CPUs hitting the power wall enforced programming techniques that emphasized the spreading of computations over multiple cores/processors. As it turns out, many of the computation tasks that software developers face are parallelizable. One example is the case of modeling particle systems. Such systems are widely used in many fields from physical simulation to computer games.

The Vulkan* API is a collaborative effort by the industry to meet current demands of computer graphics. It is a new approach that emphasizes hiding the CPU bottleneck through parallelism, allowing much more flexibility in application structure. Aside from components related only to graphics, the Vulkan API also defines the compute pipeline for numerical computation.

This paper discusses and compares aspects of the implementation of a particle system using CPUs and GPUs supported by a Vulkan-based renderer example.

1.1 Assumptions and expectations

With all the performance gains that Vulkan has to offer, there is a downside in the form of a rather steep learning curve. Drawing a triangle, the simplest example available on the Internet can take around six hundred lines of code. This is because Vulkan API requires the developer to specify nearly all aspects of the rendering mechanism. Still, given some effort, Vulkan is a clear and straightforward example for using API.

This document does not intend to teach Vulkan basics. Some basic knowledge of Vulkan is required, however. You should already know how to set up simple Vulkan applications. If you are new to Vulkan, then a good start would be to work through the LunarG tutorial.

The example code accompanying this paper is organized to emphasize presenting architectural concepts which lead to two important assumptions: 1) The return status from Vulkan function calls is in most cases ignored to avoid overly expanding the code base. In production, every return status should be checked and reacted on; 2) Conceptually close topics are kept within a single code flow. This reduces the need for jumping through the sources, but sometimes leads to rather long functions.

This paper is recommended to be read while looking at the accompanying code. Make sure you have the examples downloaded and use your favorite code browsing tool. To avoid long code listings, this paper references the code through a series of tags. For example, GPU_TP.15 references point fifteen in the Vulkan compute version of the code base. Each tag is unique and easily searchable within the code base.

Lastly, one of the major design goals of Vulkan is to give as much freedom to the application designer as possible. Therefore, there are many ways of implementing a given task depending on the design goals, target hardware, scalability, and so on. This paper's goal is to introduce and spark some creative thinking, rather than provide a final approach.

1.2 Additional resources

Vulkan has received much attention, which has resulted in many good information sources. Here are few recommended reference materials:

1) Vulkan specification and quick reference: https://www.khronos.org/registry/vulkan/
2) OpenGL and GLSL specification. We are mostly interested in GLSL as we use it as a shader programming language: https://www.khronos.org/registry/OpenGL/index_gl.php
3) Vulkan SDK and an excellent tutorial covering Vulkan basics: https://vulkan.lunarg.com/
4) Another Vulkan tutorial goes beyond basics, covering more advanced memory handling and textures: https://vulkan-tutorial.com/
5) Database of hardware supporting Vulkan. A great source for capabilities that are offered by the variety of hardware: http://vulkan.gpuinfo.org/
6) A large set of examples in Vulkan from basic triangle to compute shader: https://github.com/SaschaWillems/Vulkan

1.3 Building examples

The project build is organized using Visual Studio* 2017. Overall, it uses three dependencies: GLM, GLFW, and Vulkan SDK. The first two are installed automatically through NuGet. The Vulkan SDK should be downloaded and installed. After opening the project, select the "Property Manager" tab. From there, unfold the "Debug|x64" and open Vulkan Dir. Then go to "User Macros" and set the VK_DIR property to point to the main Vulkan SDK directory. For example, "C:\VulkanSDK\1.0.51.0." The Debug and Release both use the same property, so one change will affect both. From there just click "Build."

2 Concept of a Simulated Particle System

Our goal is to model the movement of multiple objects (particles) through space. To make things more interesting, the scene will contain entities that would alter the path through which the particle is moving.

A single particle is defined as a set of two vectors, position in space, velocity, and an additional scalar representing mass. These set of properties describe the state of each particle:

$$P = \{\overline{\boldsymbol{x}},\ \overline{\boldsymbol{v}},\ \mathrm{m}\} \tag{1}$$

We can look at the way a particle moves through space with displacement given by the differential equation:

$$\frac{\boldsymbol{d}\overline{\boldsymbol{x}}}{\boldsymbol{dt}} = \overline{\boldsymbol{v}} \tag{2}$$

For now, we will treat the velocity as a constant and solve for displacement:

$$\int \boldsymbol{d}\overline{\boldsymbol{x}} = \int \overline{\boldsymbol{v}}\mathbf{dt}$$

$$\overline{\boldsymbol{x}} = \overline{\boldsymbol{v}}\ t + C \tag{3}$$

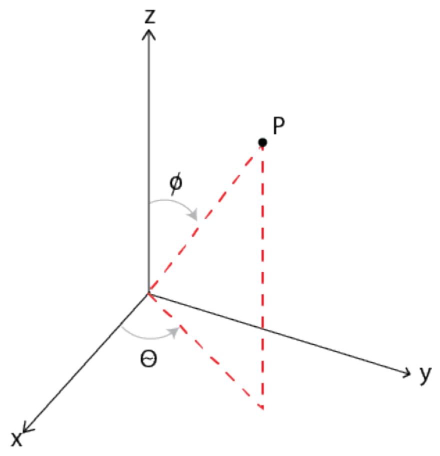
We arrive at a general formula that models how a particle is moving through space. Constant velocity is assumed (but will change soon). Also, we must have constant C, a value which must be known, in order to describe our system completely. The constant C is related to the initial conditions of this differential equation. In other words, the particle at the beginning of its life has to have a predefined state. This requires using generators.

2.1 Generators

The concept of a generator is to be a function which fulfills the task of "placing" a new particle into the scene in some predefined manner. This demonstration defines a single sprinkler generator with functionality consisting of spawning particles at a predefined point in space with given velocity and mass.

The position of the point of origin is given directly as a parameter. The direction is defined using spherical coordinates which are then internally converted into the Euclidean coordinate system. The angles are defined as follows:

1) θ (theta): Angle starts at x-axis and rotates counterclockwise.
2) ϕ (phi): Angle starts from z-axis towards xy-plane.
3) The value of speed is given as a separate argument.



To convert into Euclidean coordinates, we use the following set of equations:

$$x=\sin(\phi)\cos(\boldsymbol{\theta}),\ y=\sin(\phi)\sin(\boldsymbol{\theta}),\ z=\cos(\phi) \tag{4}$$

Going back to our general equation for particle motion and factoring in the generator, we can determine that at time t=0 the position of a particle becomes x_0:

$$\overline{\boldsymbol{x}}_0 = \overline{\boldsymbol{v}}_0 \cdot 0 + C \longrightarrow C = \overline{\boldsymbol{x}}_0$$

$$\overline{\boldsymbol{x}} = \overline{\boldsymbol{v}}t + \overline{\boldsymbol{x}}_0 \tag{5}$$

At this point, we have a formula that we can use to describe the motion of a particle. However, our scene will not look that interesting as all of the particles would only move in a straight line and at a constant speed. To give our scene more dynamic behavior, we introduce the concept of interactors.

2.2 Interactors

To understand how we can consistently influence the motion of a particle, we start treating velocity as a variable instead of constant. The change of velocity is given by the following formula:

$$\frac{d\overline{\boldsymbol{v}}}{dt} = \overline{\boldsymbol{a}} \tag{6}$$

This equation looks similar to the previous formula for displacement. By solving this differential equation, we arrive at the formula for velocity:

$$\overline{\boldsymbol{v}} = \overline{\boldsymbol{v}}_0 + \overline{\boldsymbol{a}}t \tag{7}$$

Therefore, to consistently change particle motion we influence the velocity indirectly through changing acceleration. We define the interactor as a function that, for the given particle state and given interactor properties, returns the acceleration vector:

$$f\,(p,\, i_n) \rightarrow \overline{\boldsymbol{a}} \tag{8}$$

Unfortunately, defining the interactor as a function becomes problematic. When we substitute back into the velocity equation, we see that the differential equation has a potential to become unsolvable:

$$\frac{d\overline{\boldsymbol{v}}}{dt} = f\,(p,\, i_n) \tag{9}$$

This is not what we want. Our goal is to have the ability to introduce different types of interactors without being worried about breaking our model. To do that, we need to implement the Euler method of solving differential equations to approximate the particle movement numerically.

2.3 Euler method in numerical simulation of particle movement

The Euler method working principle is to step through the differential equation instead of solving across the entire domain. If we assume a sufficiently small time step, we can approximate the velocity (2) and acceleration (6) to be constants within the span of a single step. This allows us to fall back to the equations (5) and (7), and discretize them so they can be programmed. Therefore, the final set of equations to be used is as follows:

$$\overline{\boldsymbol{x}}[n] = \overline{\boldsymbol{x}}[n - 1] + \overline{\boldsymbol{v}}[n] * \Delta t$$

$$\overline{\boldsymbol{v}}[n] = \overline{\boldsymbol{v}}[n - 1] + \overline{\boldsymbol{a}}[n] * \Delta t \tag{10}$$

If the math so far in this chapter has given you trouble, don't get discouraged. It is enough to understand the final equations (10) to work with the rest of material. From now on we will treat the acceleration vector as a constant within the single iteration. Next, we dive deeper into interactors that affect the motion of a particle.

2.4 More on interactors

We established earlier that the particle motion within the scene would be influenced by changing acceleration, using Newton's second law:

$$\overline{\boldsymbol{a}} = \frac{\overline{\boldsymbol{F}}}{\boldsymbol{m}} \tag{11}$$

We see that the task of an interactor should be to generate some force and convert it into acceleration using the mass of the particle. Moreover, we can sum up the forces coming from multiple interactors together:

$$\overline{\boldsymbol{F}}_{\boldsymbol{NET}} = \sum_{\boldsymbol{i}} \overline{\boldsymbol{F}}_{\boldsymbol{i}} \tag{12}$$

Therefore, our final approach is as follows:

$$\overline{\boldsymbol{a}}_{NET} = \sum_{i} \overline{\boldsymbol{a}}_i = \sum_{i} \frac{\overline{\boldsymbol{F}}_i}{\boldsymbol{m}} \tag{13}$$

The main reason to not multiply out the mass is that interactors that simulate gravity don't use it. In other words, the acceleration due to gravity does not depend on the mass of an object on which gravitational pull is exerted. It is also possible that some interactors which don't model physically accurate behavior could simply ignore mass as well. This is an optimization step which could save a significant number of cycles spent on multiplication.

2.4.1 Constant force interactor

This is the simplest form of an interactor. Its function is to exert the force on a particle regardless of any other properties. It could be used to make a particle go towards some predefined direction:

$$\overline{\boldsymbol{a}}_i = \frac{\overline{\boldsymbol{F}}_i}{\boldsymbol{m}} \tag{14}$$

2.4.2 Gravity point interactor

This is the second type of interactor. It represents a single point that exerts the force of gravity on particles in the scene. Such a point has mass and position, but in our example it does not have volume. It uses the standard gravity equation:

$$\overline{\boldsymbol{F}} = \frac{\boldsymbol{GMm}}{\boldsymbol{d}^2}\hat{\boldsymbol{r}} \tag{15}$$

Now we can see why we chose not to include the mass of a particle within each interactor. When deriving the formula for acceleration, we end up with:

$$\overline{\boldsymbol{a}} = \frac{\boldsymbol{GM}}{\boldsymbol{d}^2}\hat{\boldsymbol{r}} \tag{16}$$

In this equation, G is the gravitational constant, M is the mass of the gravity point, d represents the distance, and r is the unit vector pointing toward the gravity point interactor from the location of the particle.

2.4.3 Gravity planar interactor

This is the last type of interactor implemented for this example. It is modeled as an infinite plane exerting gravitational force on the particles in the scene. In this case, the r vector is always perpendicular to the plane. Therefore, for any given particle location, the distance has to be calculated, as such: Let P_1={x1,y1,z1} represent a particle in space. The plane is described by its normal vector n={a,b,c}, and a point on that plane, P={x,y,z}. The distance D is given by:

$$D = \frac{\boldsymbol{abs}(\boldsymbol{dot}(\overline{\boldsymbol{n}},\overline{\boldsymbol{P}}_{\boldsymbol{1}})-\boldsymbol{d}}{\boldsymbol{length}(\overline{\boldsymbol{n}})} \tag{17}$$

$$d = -\,dot(\overline{\boldsymbol{n}}, \overline{\boldsymbol{P}}) \tag{18}$$

This type of interactor can be used to model large objects compared to scene size such as the Earth.

2.4.4 Additional thoughts

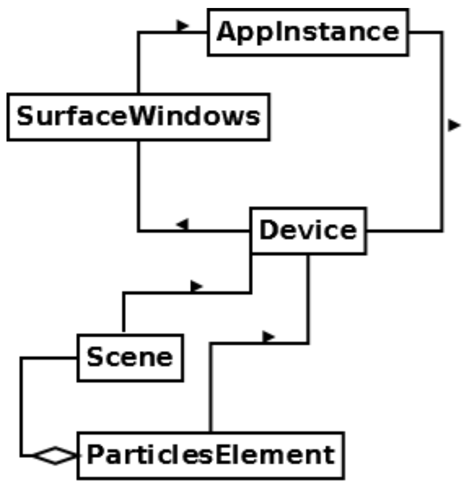
Simulating physically correct behavior is not the only way to build interesting particle systems. In fact, setting up a physically accurate system in equilibrium could be very hard. A similar effect could be achieved by using non-physically accurate interactors. Such functions can look into the entire state of a particle and generate acceleration that will attempt to force certain behavior.

3 Vulkan Renderer for CPU-Based Simulator

The design of the renderer component used in visualizing the CPU version of the particle simulator will be described first. If you are not familiar with Vulkan, it is a good idea to pause here and review one of the tutorials. The LunarG tutorial covers sufficient material to understand the rest of this paper. Also, remember that Vulkan leaves the application design to the programmer and allows for a variety of ways to approach a task. Take this guide as one of the options available rather than the definitive solution.

At a high level, when programming using Vulkan, the goal is to construct a virtual device to which drawing commands will be submitted. The draw commands are submitted to constructs called “queues.” The number of queues available and their capabilities depends upon how they were selected during construction of the virtual device, and the actual capabilities of the hardware. The power of Vulkan lies in the fact that the workload submitted to queues could be assembled and sent in parallel to already executing tasks. Vulkan offers functionality to coherently maintain the resources and perform synchronization.

In our application, we took the approach of breaking up the Vulkan renderer setup into five components (see graphic below). The goal is to present an easy-to-understand codebase for the Vulkan application.



3.1 Physical device selection

The first step in every Vulkan application is to create the Instance Object. This functionality is delivered through VkInstance. It is required that an application has a VkInstance created before starting anything else. The VkInstance is the interface used to inspect the hardware capabilities of a computer system. This functionality is encapsulated in the AppInstance object (CPU_TP1). The AppInstance object is essentially a wrapper exposing the interface for convenient hardware device enumeration.

3.1.1 Validation layers: A crucial step in Vulkan application development

One of the most critical steps when developing with Vulkan is to enable the validation layers (CPU_TP2). By design, for performance reasons, Vulkan does not check the validity of the application code. When used incorrectly, Vulkan calls would simply fail or crash with virtually no traces that would allow for debugging. With validation layers enabled, the Vulkan loader injects a layer of code for verification. The enabling of validation layers is a critical step that every developer should remember. However, that layer affects performance and should be disabled in production.

3.2 Vulkan virtual device and swapchain

The next step is to create the actual virtual device that will expose the command queue used to render the scene. First, a surface object must be instantiated. (The topic of the surface is out of the scope of this article, but it is covered in detail in the LunarG tutorial.) However, in this case, the demo code supports Windows* only, and the surface is created and exposed by the SurfaceWindows class (CPU_TP3).

Next is creating the Device object (CPU_TP4). Note that the Device class encapsulates much more than VkDevice. Starting with the constructor, the device class exposes multiple versions, which allows more flexibility in initialization (CPU_TP5). The physical device used to create the virtual device can be "first fit" or it can be specified using a VendorID number, if the software is to use specific hardware. End-user systems can have multiple devices that support Vulkan, and the first one on the list is not always the device desired.

The default constructor (CPU_TP6) obtains the list of available physical devices and then runs the createDevice method. If the creation was successful, this physical device is then used. If the creation was not successful, the constructor tries the next physical device. The actual device creation is done in the createDevice method (CPU_TP7). The first step is to determine if the physical device can deliver queues with the required functionality (CPU_TP8). Devices can deliver many types of queues. Some queues may support all of the required operations. Some queues may be highly specialized for the specific tasks, such as compute or transfer. In this case, the application expects the queue to allow for transfer and drawing graphics. The application will iterate over available queues until an index is found that matches the expected criteria.

Once completed, the virtual device is created. Besides VkDevice, the object supplies the queue handlers and swapchain that correspond to the virtual device. (For more details, please check the Vulkan tutorials and reference.)

3.3 Scene setup

The ParticlesElement (CPU_TP9) task is to draw the particle fountain. For it to be drawn, it has to belong to the scene. The *Scene* object role is to aggregate all the scene elements, maintain command buffer, and the render pass (CPU_TP10). When the simulator is ready to render, it calls the render function (CPU_TP11). This function is responsible first for initiating the command buffer for recording, obtaining the current render target, and finally starting the render pass (CPU_TP12). Once that is done, the renderer begins iterating over every element in the scene (CPU_TP13), allowing it to record drawing commands into the command buffer. In this case, there is a single scene element that draws the particle fountain. After all scene elements finish recording, the render function submits the command buffer into the queue and presents the results on the screen.

3.4 Scene Element

The SceneElement (CPU_TP14) represents the single element or entity within the scene. The expected scope of scene element objects is to handle everything related to rendering a single entity and maintaining required resources, like the pipeline and buffers. The ParticlesElement focuses its entire operation around two methods. First is the constructor (CPU_TP15) that initializes the rendering pipeline. Second is the recordToCmdBuffer (CPU_TP16) that is invoked by the Scene class to record element-specific draw calls into the command buffer.

3.5 Graphics memory allocation and performance

There is one important aspect that has to be considered when the memory for the buffers is allocated. The Vulkan device can expose different memory types that it can use. The Memory tab in the hardware database shows that devices expose multiple heaps from which memory could be allocated. Looking at the list of memory types, each type represents two values: the memory type flags that describe properties and the heap that is associated with them. (See hardware database referenced in section 1.2.)

The type of heap selected could significantly affect performance. For example, some devices, mainly discrete graphics cards, expose one heap with only DEVICE_LOCAL_BIT supported, which is the fastest memory available for that device, but does not allow direct access from the CPU (more on that in a moment). Another heap with the HOST_VISIBLE_BIT allows direct access, but has potentially longer access time.

As of today, most of Intel's integrated GPUs expose heaps with both DEVICE_LOCAL_BIT and HOST_VISIBLE_BIT. The presence of the first one ensures optimal access time for the platform. The second allows the memory mapping of the allocated buffer into the CPU address space and direct access (CPU_TP17, CPU_TP18).

When selecting a memory heap there is a three-step process. The first step is to specify and create the buffer (CPU_TP19) (note that vkCreateBuffer does not allocate memory for the buffer yet!). The second step is to use the Vulkan API to obtain the list of memory types that can support the defined buffer (CPU_TP20). The third step is to use the selected memory index to allocate the buffer memory. Of particular interest is the memoryTypeBits field in the VkMemoryRequirements. This field is a bit string that has a bit set to 1 at the position of the bit that corresponds to the memory type index of the memory type that can be used to allocate buffer memory. If the bit contains a value of 0, then a memory heap index cannot be used with the particular object of interest.

The first step above will likely provide multiple options. Therefore, in the second step, the exact memory type desired must be chosen. In this case, on Intel's GPU, all of the memory types have DEVICE_LOCAL_BIT set. HOST_VISIBLE_BIT and HOST_COHERENT_BIT also must be set. The HOST_COHERENT_BIT guarantees that memory operation is observed without the need to manually flush the buffers. If the memoryTypeBit is set and the memory type at the given index supports the requested flags, then the specific index is selected to be used (CPU_TP21). In the third step, that index is used when allocating the actual memory on the heap (CPU_TP22).

This memory allocation topic is quite important, and you are encouraged to look further into it. Topics like staging buffer or Vulkan buffer copy operations would be a good next step, especially for devices with dedicated memory.

4 CPU-Based Particle Simulator

Section 2 describes the model of the particle simulator that was implemented for this project. The implementation is based on a real-time infinite loop. The simulation loop contains three main operations:

1) Obtain the time difference between the current and previous iteration (CPU_TP23).
2) Compute the next Euler iteration for the particle model (CPU_TP24).
3) Upload and render a new frame using the Vulkan renderer (CPU_TP25).

The entire physical simulation happens within the progress method (CPU_TP26) within the algorithm, and it is composed of three major steps:

4) For every particle, the algorithm launches the interactors and adds up the acceleration vectors (CPU_TP27).
5) The algorithm sorts the particle list in such a way that all of the active particles (TTL>0) occupy the left side of the list, and all of the inactive particles occupy the right side of the list (CPU_TP28).
6) Launches particle generator(s) in order to add new particles to the list if the generator's conditions have been met, and there is space left in the buffer (CPU_TP29).

With this simple model, many interesting effects may be generated. Although this implementation uses a real-time approach (i.e., the actual hardware clock is used to obtain the time difference), the code can easily be modified to use a stepwise approach. An important point to remember is that the time difference has to be sufficiently small, otherwise it will break the model. Knowing that, it can be determined whether or not this approach is a good choice for a system that is capable of spinning the main loop at a sufficient speed. If that is not the case, consider switching from a real-time approach to an arbitrarily determined time-step, which will not use the system clock.

4.1 Particle model

To represent the set of particles in a scene, the Buffer class (CPU_TP30) is used. It is a relatively simple class that encapsulates a memory array of particle structures (CPU_TP31). The class also delivers a few convenience functions and the array sorting algorithm.

The purpose of the sorting algorithm is to place particles into two groups within the particle array. The active particles are placed on the left side and the inactive on right side. It is implemented as follows:

1) Initialize L index at the first element and P index at the last element.
2) Progress L toward the end of the array until a particle with TTL<=0 is encountered.
3) Progress P toward beginning the array until a particle with TTL>0 is encountered.
4) Swap tab[L] with tab[P].
5) Repeat with L+1.

If at any time L==P, the finish condition is reached.

4.2 Scene interactors and generators

Both interactors and particle generators influence the scene during the simulation. Within the model, both are represented by their corresponding abstract interfaces. The interactors use BaseInteractor (CPU_TP32) and generators, BaseGenerator (CPU_TP33). Both need to be defined before the Model object is instantiated (CPU_TP34).

In the case of the interactor, all that the model requires is to obtain the acceleration vector given the particle state and time difference. Internally, the interactor can perform arbitrary operations, and interactors can be fundamentally different from each other. For example, the ConstForceInteractor (CPU_TP35) uses only force magnitude, direction vector, and particle data.

The BaseGenerator, aside from a common interface, delivers functions to generate particle properties like speed with randomized deviations (CPU_TP36). However, it leaves the exact particle location, the rate of appearance, and other details to the derived classes. In this example, point generator (CPU_TP37) is used. The point generator is set in such a way that it acts as particle sprinkler (CPU_TP38). It is given direction in polar coordinates specified in degrees, and emits particles according to a defined rate.

4.2.1 Performance vs. maintainability

There exists a performance bottleneck in the implementation. A specific design choice was made to prioritize maintainability over performance. The problem originates from the polymorphic call into the computeAcceleration method (CPU_TP39).

Polymorphic calls tend to be slower than regular function calls, and every interactor is executed for every particle. It is clear how this could become a significant factor for a large number of particles. However, the advantage in this case is that there is an easy way to implement new interactors into the mix with no need make changes to the Model's code.

The alternative here could be to implement some mechanism that would detect the type of interactor and execute its procedures using a conditional statement. This will improve performance, but also increase the complexity of the code.

Which approach to use is a case-by-case decision that depends on project goals. When a small number of particles is expected with a large variety of interactors present, then the virtual method approach is a good choice. However, if the number of particles is large, then switching to other solutions could be a good way to improve performance.

4.3 Lifetime of the particle and the importance of sorting

There is one last topic related to particle generation and particle lifetime. Once a particle is moving through the scene, it can easily fly outside of the visible area. Since it is not visible, there is virtually no purpose in modeling it. Sometimes the goal is to model effects like a flame with a specific height. To implement such cases, every particle must have a property that determines its lifespan called “Time To Live” (TTL). At each iteration, the time difference is subtracted, and if the value is equal to or below zero, the particle is considered inactive. When particles become inactive, no operations are performed on them and they are no longer drawn.

There are two issues related to the case of inactive particles. First, is that a way is needed to free the space up for new particles. Second, is that inactive particles should no longer be drawn on the screen. This project example uses a simple rendering mechanism. However, in the case of a more complicated multi-pass renderer with more computation-heavy shaders, it is a good idea to avoid processing elements that would not be visible on the screen. This is where sorting comes in. Sorting the particle array allows for submitting draw calls that specify only the number of particles that should appear on the screen. It also simplifies the work of generators as they do not have to search through the buffer for free slots to fill.

There is a possible alternative approach. Instead of the contiguous array, a linked list could be used. This would eliminate the need for sorting, but, on the other hand, it would slow down the upload of data to the GPU buffers because of the need to convert the linked list structure into a contiguous array.

5 Multithreaded Approach to Computing Particle System

The speed of the simulation could be improved by parallelizing the workload. There are a few cases to consider:

1) Parallelize the workload across the array of particles, calculating the interactors' contribution and Euler step as a single work unit.
2a) Parallelize the workload across interactors, calculating the effect of each interactor at every particle as a single work unit.
2b) Further parallelize Option 1, above, by splitting interactor contributions into separate work units.
3) Parallelize the sorting algorithm.
4) Parallelize the generators' contribution.

Option 1, above, is the recommended approach. For each particle, the algorithm computes, as a single task, the contribution from each interactor, summing the acceleration vectors and then performing the Euler step (CPU_TP40). Since each particle resides in a different memory region, and interactors usually do not modify their internal data when computing particles, this method provides the most optimal solution. The OpenMP* framework was used to provide multithreading. OpenMP is an API that is based primarily on compiler directives and used for multithreaded-oriented development. It has lower overhead regarding required code to be written. It could be easily used to parallelize loops, but has much more to offer. However, sometimes it might not deliver the level of detail as some other available libraries.

Option 2a, is probably not recommended. Some might find that this option is appealing from the perspective that the data that defines the properties of an interactor would have to be loaded once, and then used extensively while computing each particle. This is something that will be supported by the CPU cache, significantly speeding up the process. However, there are issues with this approach. For example, acceleration must be summed up, which means there will be a need to synchronize access between threads executing the interactors and then store the acceleration vector for every particle in an additional memory region. Moreover, the entire array will require an additional pass performed to calculate the Euler step. These two disadvantages would produce significant synchronization overhead, likely rendering this strategy impractical.

Option 2b is an extension of Option 1. Hypothetically, if there are computationally demanding interactors, it could be beneficial to split their operations. However, the goal is to achieve minimal context switching. If every particle computation and every interactor contribution is parallelized, it will most likely lead to an explosion of threads. Switching between those threads would further decrease performance.

Option 3 suggests improving the sorting algorithm through parallelization. This strategy might improve sorting speed; however, the type of data must be considered. In this particular case, a particle is in an active or inactive state. The order of the particles that are in the same state is not important, as long as they are all placed on the correct side of the sorted array. This allows for the array to be sorted in a single pass. If a parallel sorting algorithm is used instead, it is possible that performance could be improved. However, such improvement is not guaranteed. Possible improvements may be determined through experimentation.

In Option 4, parallelizing the generation of particles has a high potential of improving performance, but only under certain conditions. The first consideration is to determine the workload of all of the generators. If the memory buffer is almost always full, there won't be any space for the generators to work. Therefore, spawning additional threads to execute the generators will only create unnecessary overhead. Every generator takes as much space as needed, and what is left is assigned to the next generator in the list. This could result in some threads executing generators just to find out that there is nothing for them to do.

5.1 Going further with parallelization

There are few more things worth mentioning that could affect performance. First is the issue of false sharing. A single particle structure size was 44 bytes. Given that the cache line size was 64 bytes, what was fetched from lower level memory was an entire particle structure, or part of a particle structure, that the current thread was working on, as well as part of a particle structure located right next to it. If two parallel threads each work on a single particle located next to each other, the modification of the acceleration vector inside one thread will invalidate the entire cache line. Essentially, this will cause cache miss on the other thread, even though the threads were assigned to different particles. This situation can be alleviated by aligning the particle structures to the 64-bytes boundary. However, this will also increase memory waste.

The second and preferred solution is to group multiple particle structures together and direct them to be processed as a single unit. This is what OpenMP does by default in the "parallel for" directive.

OpenMP is a very good framework, but it has disadvantages. OpenMP is built into a number of compilers that support it. However, not all compilers support OpenMP, or support an outdated version of OpenMP. A great alternative to OpenMP is the thread support library that was introduced with C++11 and should be supported in almost all C++ compilers currently available.

6 Vulkan Compute in a Modeling Particle System

This section addresses the topic of porting a CPU-based particle simulator onto the GPU using Vulkan compute. Vulkan compute is an integral part of the Vulkan API. Although most of the simulation will be moved onto the GPU, the sorting and generation of particles will still be kept on the CPU.

The goal of this section is to present how to set up the compute pipeline, instantiate more advanced shader input constructs, and show how to send frequently changing, but small, portions of data onto the GPU without using memory mapping.

This is a good moment to mention a tool that is extremely valuable when debugging Vulkan applications. The tool, RenderDoc, can be used to capture the current application state for inspection. The tool can be used to examine buffers content, shader input, textures, and much more. RenderDoc is automatically installed with VulkanSDK. Although not discussed in this document, it is highly recommended for readers to become familiar with this tool.

6.1 Using GPU memory buffer for computation and visualization

The first step is to repurpose the buffer structure that holds particle data. Among the buffer types available to use on a GPU is a buffer type called the shader storage buffer object (SSBO). This is a type of buffer that is similar to the uniform buffer. However, SSBOs are much larger in size than the uniform buffer and are writable from within the shader.

The internal storage, which was the standard C-style array contained by the Buffer class, is now replaced with an SSBO. Moreover, the same buffer memory region will be reused as a vertex buffer for rendering. The compute shader will modify the SSBO content, and then the renderer will use it as a vertex buffer object (VBO) to render the frame.

Up until now, the Buffer class was independent of Vulkan. Now, however, the entire application is dependent of Vulkan. This means that before the Model class can be instantiated, the Buffer class must be constructed. And before the Buffer class is constructed, the Vulkan virtual device must be created.

To make things appear in a more organized order, the buffer, the device, the AppInstance, and the rest of supporting code was moved into a new namespace called "base." Now, instead of creating the Model first, the application starts from creating the AppInstance, device, and buffer objects (GPU_TP41). The code creating the vertex buffer has been moved out of ParticleElement into buffer (GPU_TP42). Note the change in the usage field: it now has two flags assigned, VERTEX_BUFFER_BIT and STORAGE_BUFFER_BIT. The rest of the procedure remains the same. The Buffer class now delivers the VkBuffer handle and interface to access the buffer on the CPU side (GPU_TP43). The constructed buffer is then used in instantiation of both the Model (GPU_TP44) and the ParticlesElement(GPU_TP45) objects.

6.2 Moving computation from the CPU to compute shader on the GPU

The Model class is where most of the changes will occur. In this version, the Model class will contain another pipeline which will execute the compute shader.

First, one of the biggest changes is in how the interactors are generally implemented. Although the interactor logic was moved into the compute shader, there is still a need for the mechanism to allow for the ability to dynamically specify the properties of each interactor in the application logic. Because of the structure of GLSL language, it is no longer possible to implement interactors using an abstract interface. This requires the interactors to be split by their type and provide specific data for all of them. To do this, we define a structure (GPU_TP46) called Setup to contain arrays for each type of interactor. Each array defines properties for an interactor and allows an arbitrary number of interactors of a given type to be present up to a specified upper boundary. The count variables specify the actual number of active interactors.

When submitting complex structures to buffer objects, it is critical to adhere to the layout rules that apply for the given type of buffer (Note the alignment statement in the structure field's declaration (GPU_TP47)). For uniform or shader storage buffer objects, these layouts are std140 and std430. Their full specification can be found in the Vulkan specification and GLSL language specification.

As in the CPU version, interactors were initialized before constructing the Model object (GPU_TP48). However, this time the initialization is based on populating the Setup structure. Later, the entire structure will be uploaded into the GPU during construction (GPU_TP49), making it available for the compute shader to access.

In the Vulkan API the compute stage and graphics stage cannot be bound into a single pipeline. As a result, two pipeline objects are available. Compared to the graphics pipeline, the compute pipeline (GPU_TP50) is less complicated. The primary concern is to define the proper pipeline layout. Beyond that, the process of creating the compute pipeline is nearly the same as creating the graphics pipeline.

Once the pipeline is established and the necessary data is ready to execute, the procedure starts recording commands into the command buffer for the compute pipeline (GPU_TP51). However, there are two concepts that are worth examining.

The first concept is "push constants." Push constants are small regions of memory residing internally in the GPU designated for fast access. The advantage of using push constants is that data delivery can be scheduled right within the command buffer (GPU_TP52) without the need for memory mapping. The memory layout of push constants needs to be defined during pipeline construction, but the process is much simpler compared to the descriptor set (GPU_TP53).

The second concept is related to the division of workload. The number of elements is split into work groups that are scheduled to execute on the compute device. The size of the work group is specified within the compute shader (GPU_TP54). Then the API is given the number of work groups scheduled (GPU_TP55) for execution of the compute job. The sizes and number of workgroups are both limited by the hardware. Those limits can be checked by looking at the maxComputeWorkGroupCount and maxComputeWorkGroupSize fields, which are part of VkPhysicalDeviceLimits structure (which is a part of the VkPhysicalDeviceProperties structure that is obtained through calling vkGetPhysicalDeviceProperties). It is imperative to schedule a sufficient number work groups with appropriate size to cover the entire set of particles. The optimal choice of sizes is dictated by the specific hardware architecture and influenced by the compute shader code (e.g., the internal memory requirements).

6.3 Structure of the compute shader-oriented simulation

With the transition to the compute shader, the interactors take the form of two components. First are the interactor parameters which are delivered through the uniform buffer (GPU_TP56). The second is the function set wherein each function corresponds to a specific interactor and calculates the acceleration vector for the processed particle (GPU_TP57).

The function per iteration of the compute shader is to obtain a particle from SSBO using the global id index (GPU_TP58). The compute shader iterates through each interactor descriptor and invokes its function as many times as the count field indicates (GPU_TP59). Once all interactors have been executed, the acceleration vectors are summed and the compute shader invokes the Euler step function (GPU_TP60). The particle is then saved back to SSBO.

After the computing step is done, the renderer is invoked. The renderer uses the same buffer, but accesses memory through the vertex buffer interface, instead of SSBO, to render the scene.

7 Going Further

Building on this work, there are few options to consider that may improve performance.

7.1 Eliminate the CPU role in computations

Currently, the GPU compute shader only calculates the interactor’s influence on the particle. All sorting and generating of the particles were left to the CPU (see section 5). In the case of Intel's GPU, this approach is not penalized at all due to the uniform access to the memory. However, for other platforms, especially a discrete GPU, allocating buffers using the memory heap with most efficient access time is a must. This could mean that address space in such a heap might not be accessible from the CPU through memory mapping. In that case, the only way to access the data would be by copying buffers internally between heaps back and forth, or accessing it through the compute shader.

An approach in which buffers are copied between each other will almost certainly create a performance bottleneck. Therefore, the compute shader option seems more appealing. Initially, the sorting was left in its sequential form because of the unclear gains of parallel algorithms for this specific scenario. However, a GPU offers many more computing units which could make parallel sorting algorithms, like an “odd-even sort,” deliver better results.

There is still the issue of synchronization. An instance of the compute shader cannot start the next sorting pass before the previous one has completed. However, if only a single local work group is used, then the GLSL language delivers a set of Shader Invocation Control Functions (see GLSL specification) that will deliver the capability of setting up proper memory barriers.

Similarly, in the case of the generators, the biggest problem was the random number generator not being thread-safe and its dependence on the generator’s internal state. The randomize function can be ported into GLSL, and the coherence maintained using the GLSL atomic memory functions.

7.2 Parallelizing the performance path by double buffering

For this project, the approach was to first compute the scene and then render it sequentially. The complexity of the rendering in this case was fairly basic, and, therefore, its impact on performance was negligible. However, this will not always be the case. With more involved renderers it is necessary to attempt to perform computations and rendering in parallel. The Vulkan API was designed with this goal in mind and provides the means to synchronize program flow, but does not enforce it. For example, looking at where the program loop is waiting for the rendering function to finish (CPU_TP61). The part of the code responsible for presentation obviously has to wait, but the model does not. In fact, the model could start computing the next step as soon as the vertex data is delivered and only need to wait before entering the setVertexBufferData function. From here the application can be split into two threads and use a producer-consumer setup.

In the GPU version, instead of using a single SSBO, double buffering can be introduced. One buffer is used both as a render-input and compute-input. At the same time, the second buffer is populated by the compute shader with the Euler function output.

It is important to remember that there is an upper threshold to what can be achieved, which is limited by the capabilities of the hardware. The compute and render tasks executed on the same GPU will not always provide performance gains because of the finite number of compute units available.

8 Conclusion

Vulkan offers the ability to bypass the CPU bottleneck that exists in previous generations of graphics APIs. However, Vulkan has a steep learning curve and is more demanding when it comes to application design. Nonetheless, despite being very explicit, it is a very consistent and straightforward API.

It is worth remembering these few points which will help with application development, as well as allow achievement of better application performance:

1) Start your development with the validation layers enabled and become familiar with tools like RenderDoc.
2) Carefully design your application as you will have to make many decisions about various details.
3) Pay attention to where your buffer memory is allocated as it might have significant performance implications.
4) Understand the hardware you are working with. Not all the techniques will work equally well on every type of GPU. Although all Vulkan devices expose the same API, their properties and limitations might be fundamentally different from each other.