API without Secrets: Introduction to Vulkan* Part 2

Table of Contents

Tutorial 2: Swap Chain – Integrating Vulkan with the OS

Welcome to the second Vulkan tutorial. In the first tutorial, I discussed basic Vulkan setup: function loading, instance creation, choosing a physical device and queues, and logical device creation. I'm sure you now want to draw something! Unfortunately we must wait until the next part. Why? Because if we draw something we'll want to see it. Similar to OpenGL*, we must integrate the Vulkan pipeline with the application and API that the OS provides. However, with Vulkan, this task unfortunately isn't simple and obvious. And as with all other thin APIs, it is done this way on purpose—for the sake of high performance and flexibility.

So how do you integrate Vulkan with the application's window? What are the differences compared to OpenGL? In OpenGL (on Microsoft Windows*) we acquire Device Context that is associated with the application's window. Using it we then have to define "how" to present images on the screen, "what" the format is of the application's window we will be drawing on, and what capabilities it should support. This is done through the pixel format. Most of the time we create a 32-bit color surface with a 24-bit depth buffer and a support for double buffering (this way we can draw something to a "hidden" back buffer, and after we're finished we can present it on the screen—swap front and back buffers). Only after these preparations can we create a Rendering Context and activate it. In OpenGL, all the rendering is directed to the default, back buffer.

In Vulkan there is no default frame buffer. We can create an application that displays nothing at all. This is a valid approach. But if we want to display something we can create a set of buffers to which we can render. These buffers along with their properties, similar to Direct3D*, are called a swap chain. A swap chain can contain many images. To display any of them we don't "swap" them—as the name suggests—but we present them, which means that we give them back to a presentation engine. So in OpenGL we first have to define the surface format and associate it with a window (at least on Windows) and after that we create Rendering Context. In Vulkan, we first create an instance, a device, and then we create a swap chain. But, what's interesting is that there will be situations where we will have to destroy this swap chain and recreate it. In the middle of a working application. From scratch!

Asking for a Swap Chain Extension

In Vulkan, a swap chain is an extension. Why? Isn't it obvious we want to display an image on the screen in our application's window?

Well, it's not so obvious. Vulkan can be used for many different purposes, including performing mathematical operations, boosting physics calculations, and processing a video stream. The results of these actions may not necessarily be displayed on a typical monitor, which is why the core API is OS-agnostic, similar to OpenGL.

If you want to create a game and display rendered images on a monitor, you can (and should) use a swap chain. But here is the second reason why a swap chain is an extension. Every OS displays images in a different way. The surface on which you can render may be implemented differently, can have a different format, and can be differently represented in the OS—there is no one universal way to do it. So in Vulkan a swap chain must also depend on the OS your application is written for.

These are the reasons a swap chain in Vulkan is treated as an extension: it provides render targets (buffers or images like FBOs in OpenGL) that integrates with OS specific code. It's something that core Vulkan (which is platform independent) can't do. So if swap chain creation and usage is an extension, we have to ask for the extension during both instance and device creation. The ability to create and use a swap chain requires us to enable extensions at two levels (at least on most operating systems, with Windows and Linux* among them). This means that we have to go back to the first tutorial and change it to request the proper swap-chain-related extensions. If a given instance and device doesn't support these extensions, the instance and/or device creation will fail. There are of course other ways through which we can display an image, like acquiring the pointer to a buffer's (texture's) memory (mapping it) and copying data from it to the OS-acquired window's surface pointer. This process is out of scope of this tutorial (though not really that hard). But fortunately it seems that swap chain extensions will be similar to OpenGL's core extensions: they will be something that's not in the core spec and that's not required to be implemented but they also are something that every hardware vendor will implement

anyway. I think all hardware vendors would like to show that they support Vulkan and that it gives impressive performance boost in games which are displayed on screen. And, what backs this theory, swap chain extensions are integrated into the main, "core" vulkan.h header.

In the case of swap-chain support, there are actually three extensions involved: two from an instance level and one from a device level. These extensions logically separate different functionalities. The first is the **VK_KHR_surface** extension defined at the instance level. It describes a "surface" object, which is a logical representation of an application's window. This extension allows us to check different parameters (that is, capabilities, supported formats, size) of a surface and to query whether the given physical device supports a swap chain (more precisely, whether the given queue family supports presenting an image to a given surface). This is useful information because we don't want to choose a physical device and try to create a logical device from it only to find out that it doesn't support swap chains. This extension also defines methods to destroy any such surface.

The second instance-level extension is OS-dependent: in the Windows OS family it is called **VK_KHR_win32_surface** and in Linux it is called **VK_KHR_xlib_surface** or **VK_KHR_xcb_surface**. This extension allows us to create a surface that represents the application's window in a given OS (and uses OS-specific parameters).

Checking Whether an Instance Extension Is Supported

Before we can enable the two instance-level extensions, we need to check whether they are available or supported. We are talking about instance extensions and we haven't created any instance yet. To determine whether our Vulkan instance supports these extensions, we use a global-level function called **vkEnumerateInstanceExtensionProperties()**. It enumerates all available instance general extensions, if its first parameter is null, or instance layer extensions (it seems that layers can also have extensions), if we set the first parameter to the name of any given layer. We aren't interested in layers so we leave the first parameter set to null. Again we call this function twice. For the first call, we want to acquire the total number of supported extensions so we leave the third argument nulled. Next we prepare storage for all these extensions and we call this function once again with the third parameter pointing to the allocated storage.

```
uint32_t extensions_count = 0;
if( (vkEnumerateInstanceExtensionProperties( nullptr, &extensions_count, nullptr ) !=
VK_SUCCESS) ||
    (extensions_count == 0) ) {
  printf( "Error occurred during instance extensions enumeration!\n" );
  return false;
}

std::vector<VkExtensionProperties> available_extensions( extensions_count );
if( vkEnumerateInstanceExtensionProperties( nullptr, &extensions_count,
&available_extensions[0] ) != VK_SUCCESS ) {
  printf( "Error occurred during instance extensions enumeration!\n" );
  return false;
}

std::vector<const char*> extensions = {
  VK_KHR_SURFACE_EXTENSION_NAME,
#if defined(VK_USE_PLATFORM_WIN32_KHR)
  VK_KHR_WIN32_SURFACE_EXTENSION_NAME
#elif defined(VK_USE_PLATFORM_XCB_KHR)
  VK_KHR_XCB_SURFACE_EXTENSION_NAME
#elif defined(VK_USE_PLATFORM_XLIB_KHR)
  VK_KHR_XLIB_SURFACE_EXTENSION_NAME
#endif
};

for( size_t i = 0; i < extensions.size(); ++i ) {
  if( !CheckExtensionAvailability( extensions[i], available_extensions ) ) {
    printf( "Could not find instance extension named \"%s\"!\n", extensions[i] );
    return false;
  }
```

```
    }
```

We can prepare a place for a smaller amount of extensions, but then **vkEnumerateInstanceExtensionProperties()** will return VK_INCOMPLETE to let us know we didn't acquire all the extensions.

Our array is now filled with all available (supported) instance-level extensions. Each element of our allocated space contains the name of the extension and its version. The second parameter probably won't be used too often, but it may be useful to check whether the hardware supports the given version of the extension. For example, we might be interested in some specific extension, and we downloaded an SDK for it that contains a set of header files. Each header file has its own version corresponding to the value returned by this query. If the hardware our application is executed on supports an older version of the extension (not the one we downloaded the SDK for) it may not support all the functions we are using from this specific extension. So sometimes it may be useful to also verify the version, but for a swap chain it doesn't matter—at least for now.

We can now search through all of the returned extensions and see whether the list contains the extensions we are looking for. Here I'm using two convenient definitions named VK_KHR_SURFACE_EXTENSION_NAME and VK_KHR_????_SURFACE_EXTENSION_NAME. They are defined inside a Vulkan header file and contain the names of the extensions so we don't have to copy or remember them. We just can use the definitions in our code, and if we make a mistake the compiler will tell us. I hope all extensions will come with a similar definition.

With the second definition comes a small trap. These two mentioned defines are placed in a vulkan.h header file. But isn't the second define specific for a given OS and isn't vulkan.h header OS independent? Both questions are true and quite valid. The vulkan.h file is OS-independent and it contains the definitions of OS-specific extensions. But these are enclosed inside #ifdef … #endif preprocessor directives. If we want to "enable" them we need to add a proper preprocessor directive somewhere in our project. For a Windows system, we need to add a VK_USE_PLATFORM_WIN32_KHR string. On Linux, we need to add VK_USE_PLATFORM_XCB_KHR or VK_USE_PLATFORM_XLIB_KHR depending on whether we want to use the X11 or XCB libraries. In the provided example project, these definitions are added by default through the CMakeLists.txt file.

But back to our source code. What does the CheckExtensionAvailability() function do? It loops over all available extensions and compares their names with the name of the provided extension. If a match is found, it just returns true.

```
for( size_t i = 0; i < available_extensions.size(); ++i ) {
  if( strcmp( available_extensions[i].extensionName, extension_name ) == 0 ) {
    return true;
  }
}
return false;
```

Enabling an Instance-Level Extension

Let's say we have verified that both extensions are supported. Instance-level extensions are requested (enabled) during instance creation—we create an instance with a list of extensions that should be enabled. Here's the code responsible for doing it:

```
VkApplicationInfo application_info = {
  VK_STRUCTURE_TYPE_APPLICATION_INFO,            // VkStructureType              sType
  nullptr,                                       // const void                 *pNext
  "API without Secrets: Introduction to Vulkan", // const char
*pApplicationName
  VK_MAKE_VERSION( 1, 0, 0 ),                    // uint32_t
applicationVersion
```

```
    "Vulkan Tutorial by Intel",                      // const char
*pEngineName
    VK_MAKE_VERSION( 1, 0, 0 ),                      // uint32_t
engineVersion
    VK_API_VERSION                                   // uint32_t
apiVersion
  };

  VkInstanceCreateInfo instance_create_info = {
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO,          // VkStructureType          sType
    nullptr,                                         // const void              *pNext
    0,                                               // VkInstanceCreateFlags     flags
    &application_info,                               // const VkApplicationInfo
*pApplicationInfo
    0,                                               // uint32_t
enabledLayerCount
    nullptr,                                         // const char * const
*ppEnabledLayerNames
    static_cast<uint32_t>(extensions.size()),        // uint32_t
enabledExtensionCount
    &extensions[0]                                   // const char * const
*ppEnabledExtensionNames
  };

  if( vkCreateInstance( &instance_create_info, nullptr, &Vulkan.Instance ) !=
VK_SUCCESS ) {
    printf( "Could not create Vulkan instance!\n" );
    return false;
  }
  return true;
```

3. Tutorial02.cpp, function CreateInstance()

This code is similar to the **CreateInstance()** function in the Tutorial01.cpp file. To request instance-level extensions we have to prepare an array with the names of all extensions we want to enable. Here I have used a standard vector with "const char*" elements and mentioned extension names in forms of defines.

In Tutorial 1 we declared zero extensions and placed a nullptr for the address of an array in a VkInstanceCreateInfo structure. This time we must provide an address of the first element of an array filled with the names of the requested extensions. And we must also specify how many elements the array contains (that's why I chose a vector: if I add or remove extensions in future tutorials, the vector's size will also change accordingly). Next we call the **vkCreateInstance()** function. If it doesn't return VK_SUCCESS it means that (in the case of this tutorial) extensions are not supported. If it does return successfully, we can load instance-level functions as previously, but this time also with some additional, extension-specific functions.

With these extensions come additional functions. And, as it is an instance-level extension, we must add them to our set of instance-level functions (so they will also be loaded at a proper moment and with a proper function). In this case we must add the following functions into a ListOfFunctions.inl wrapped into a VK_INSTANCE_LEVEL_FUNCTION() macro like this:

```
// From extensions
#if defined(USE_SWAPCHAIN_EXTENSIONS)
VK_INSTANCE_LEVEL_FUNCTION( vkDestroySurfaceKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfaceSupportKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfaceCapabilitiesKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfaceFormatsKHR )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceSurfacePresentModesKHR )
#if defined(VK_USE_PLATFORM_WIN32_KHR)
VK_INSTANCE_LEVEL_FUNCTION( vkCreateWin32SurfaceKHR )
```

```
  #elif defined(VK_USE_PLATFORM_XCB_KHR)
  VK_INSTANCE_LEVEL_FUNCTION( vkCreateXcbSurfaceKHR )
  #elif defined(VK_USE_PLATFORM_XLIB_KHR)
  VK_INSTANCE_LEVEL_FUNCTION( vkCreateXlibSurfaceKHR )
  #endif
  #endif
```

One more thing: I've wrapped all these swap-chain-related functions inside another #ifdef … #endif pair, which requires a USE_SWAPCHAIN_EXTENSIONS preprocessor directive to be defined. I've done this so Tutorial 1 would properly work. Without it, our first application (as it uses the same header files) would try to load all these functions. But we don't enable swap chain extensions in the first tutorial, so this operation would fail and the application would close without fully initializing Vulkan. If a given extension isn't enabled, functions from it may not be available.

Creating a Presentation Surface

We have created a Vulkan instance with two extensions enabled. We have loaded instance-level functions from a core Vulkan spec and from enabled extensions (this is done automatically thanks to our macros). To create a surface, we write code similar to the following:

```
  #if defined(VK_USE_PLATFORM_WIN32_KHR)
  VkWin32SurfaceCreateInfoKHR surface_create_info = {
    VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR,  // VkStructureType
sType
    nullptr,                                          // const void
*pNext
    0,                                                // VkWin32SurfaceCreateFlagsKHR
flags
    Window.Instance,                                  // HINSTANCE
hinstance
    Window.Handle                                     // HWND
hwnd
  };

  if( vkCreateWin32SurfaceKHR( Vulkan.Instance, &surface_create_info, nullptr,
&Vulkan.PresentationSurface ) == VK_SUCCESS ) {
    return true;
  }

  #elif defined(VK_USE_PLATFORM_XCB_KHR)
  VkXcbSurfaceCreateInfoKHR surface_create_info = {
    VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR,    // VkStructureType
sType
    nullptr,                                          // const void
*pNext
    0,                                                // VkXcbSurfaceCreateFlagsKHR
flags
    Window.Connection,                                // xcb_connection_t*
connection
    Window.Handle                                     // xcb_window_t
window
  };

  if( vkCreateXcbSurfaceKHR( Vulkan.Instance, &surface_create_info, nullptr,
&Vulkan.PresentationSurface ) == VK_SUCCESS ) {
    return true;
  }

  #elif defined(VK_USE_PLATFORM_XLIB_KHR)
  VkXlibSurfaceCreateInfoKHR surface_create_info = {
```

```
      VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR,    // VkStructureType
sType
      nullptr,                                           // const void
*pNext
      0,                                                 // VkXlibSurfaceCreateFlagsKHR
flags
      Window.DisplayPtr,                                 // Display
*dpy
      Window.Handle                                      // Window
window
   };
   if( vkCreateXlibSurfaceKHR( Vulkan.Instance, &surface_create_info, nullptr,
&Vulkan.PresentationSurface ) == VK_SUCCESS ) {
     return true;
   }

   #endif

   printf( "Could not create presentation surface!\n" );
   return false;
```

5. ListOfFunctions.inl

To create a presentation surface, we call the **vkCreate????SurfaceKHR()** function, which accepts Vulkan Instance (with enabled surface extensions), a pointer to a OS-specific structure, a pointer to optional memory allocation handling functions, and a pointer to a variable in which a handle to a created surface will be stored.

This OS-specific structure is called **Vk????SurfaceCreateInfoKHR** and it contains the following fields:

- sType – Standard type of structure that here should be equal to **VK_STRUCTURE_TYPE_????_SURFACE_CREATE_INFO_KHR** (where ???? can be WIN32, XCB, XLIB, or other)
- pNext – Standard pointer to some other structure
- flags – Parameter reserved for future use
- hinstance/connection/dpy – First OS-specific parameter
- hwnd/window – Handle to our application's window (also OS specific)

Checking Whether a Device Extension is Supported

We have created an instance and a surface. The next step is to create a logical device. But we want to create a device that supports a swap chain. So we also need to check whether a given physical device supports a swap chain extension, a device-level extension. This extension is called *VK_KHR_swapchain*, and it defines the actual support, implementation, and usage of a swap chain.

To check what extensions given physical device supports we must create code similar to the code prepared for instance-level extensions. This time we just use the **vkEnumerateDeviceExtensionProperties()** function. It behaves identically to the function querying instance extensions. The only difference is that it takes an additional physical device handle in the first argument. The code for this may look similar to the example below. It is a part of the CheckPhysicalDeviceProperties() function in our example source code.

```
   uint32_t extensions_count = 0;
   if( (vkEnumerateDeviceExtensionProperties( physical_device, nullptr,
&extensions_count, nullptr ) != VK_SUCCESS) ||
      (extensions_count == 0) ) {
     printf( "Error occurred during physical device %p extensions enumeration!\n",
physical_device );
     return false;
   }

  std::vector<VkExtensionProperties> available_extensions( extensions_count );
```

```
    if( vkEnumerateDeviceExtensionProperties( physical_device, nullptr,
&extensions_count, &available_extensions[0] ) != VK_SUCCESS ) {
      printf( "Error occurred during physical device %p extensions enumeration!\n",
physical_device );
      return false;
    }

    std::vector<const char*> device_extensions = {
      VK_KHR_SWAPCHAIN_EXTENSION_NAME
    };

    for( size_t i = 0; i < device_extensions.size(); ++i ) {
      if( !CheckExtensionAvailability( device_extensions[i], available_extensions ) ) {
        printf( "Physical device %p doesn't support extension named \"%s\"!\n",
physical_device, device_extensions[i] );
        return false;
      }
    }
```

6. Tutorial02.cpp, function CheckPhysicalDeviceProperties()

We first ask for the number of all extensions available on a given physical device. Next we get their names and look for the device-level swap-chain extension. If there is none there is no point in further checking the device's properties, features, and queue families' properties as a given device doesn't support swap chain at all.

Checking Whether Presentation to a Given Surface Is Supported

Let's go back to the CreateDevice() function. After creating an instance, in the first tutorial we looped through all available physical devices and queried their properties. Based on these properties we selected which device we want to use and which queue families we want to request. This query is done in a loop over all available physical devices. Now that we want to use swap chain I have to modify my CheckPhysicalDeviceProperties() function that is called inside a mentioned loop from CreateDevice() function like this:

```
    uint32_t selected_graphics_queue_family_index = UINT32_MAX;
    uint32_t selected_present_queue_family_index = UINT32_MAX;

    for( uint32_t i = 0; i < num_devices; ++i ) {
      if( CheckPhysicalDeviceProperties( physical_devices[i],
selected_graphics_queue_family_index, selected_present_queue_family_index ) ) {
        Vulkan.PhysicalDevice = physical_devices[i];
      }
    }
```

7. Tutorial02.cpp, function CreateDevice()

The only change is that I've added another variable that will contain an index of a queue family that supports a swap chain (more precisely image presentation). Unfortunately, just checking whether swap extension is supported is not enough because presentation support is a queue family property. A physical device may support swap chains, but that doesn't mean that all its queue families also support it. And do we really need another queue or queue family for displaying images? Can't we just use graphics queue that we'd selected in the first tutorial? Most of the time one queue family will probably be enough for our needs. This means that the selected queue family will support both graphics operations and a presentation. But, unfortunately, it is also possible that there will be devices that won't support graphics and presenting within a single queue family. In Vulkan we have to be flexible and prepared for any situation.

vkGetPhysicalDeviceSurfaceSupportKHR() function is used to check whether a given queue family from a given physical device supports a swap chain or, to be more precise, whether it supports presenting images to a given surface. That's why we needed to create a surface earlier.

So assume we have already checked whether a given physical device exposes a swap-chain extension and that we have already queried for a number of different queue families supported by a given physical device. We have also requested the properties of all queue families. Now we can check whether a given queue family supports presentation to our surface (window).

```cpp
    uint32_t graphics_queue_family_index = UINT32_MAX;
    uint32_t present_queue_family_index = UINT32_MAX;

    for( uint32_t i = 0; i < queue_families_count; ++i ) {
      vkGetPhysicalDeviceSurfaceSupportKHR( physical_device, i,
Vulkan.PresentationSurface, &queue_present_support[i] );

      if( (queue_family_properties[i].queueCount > 0) &&
          (queue_family_properties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) ) {
        // Select first queue that supports graphics
        if( graphics_queue_family_index == UINT32_MAX ) {
          graphics_queue_family_index = i;
        }

        // If there is queue that supports both graphics and present - prefer it
        if( queue_present_support[i] ) {
          selected_graphics_queue_family_index = i;
          selected_present_queue_family_index = i;
          return true;
        }
      }
    }

    // We don't have queue that supports both graphics and present so we have to use
separate queues
    for( uint32_t i = 0; i < queue_families_count; ++i ) {
      if( queue_present_support[i] ) {
        present_queue_family_index = i;
        break;
      }
    }

    // If this device doesn't support queues with graphics and present capabilities don't
use it
    if( (graphics_queue_family_index == UINT32_MAX) ||
        (present_queue_family_index == UINT32_MAX) ) {
      printf( "Could not find queue families with required properties on physical device
%p!\n", physical_device );
      return false;
    }

    selected_graphics_queue_family_index = graphics_queue_family_index;
    selected_present_queue_family_index = present_queue_family_index;
    return true;
```

8. Tutorial02.cpp, function CheckPhysicalDeviceProperties()

Here we are iterating over all available queue families. In each loop iteration, we are calling a function responsible for checking whether a given queue family supports presentation. **vkGetPhysicalDeviceSurfaceSupportKHR()** function requires us to provide a physical device handle, the queue family index we want to check, and the surface handle we want to render into (present an image). If support is available, VK_TRUE will be stored at a given address; otherwise VK_FALSE is stored.

Now we have the properties of all available queue families. We know which queue family supports graphics operations and which supports presentation. In our tutorial example I prefer families that support both. If I find one I store the family

index and exit immediately from CheckPhysicalDeviceProperties() function. If there is no such queue family I use the first queue family that supports graphics and a first family that supports presenting. Only then can I leave the function with a "success" return code.

A more advanced scenario may search through all available devices and try to find one with a queue family that supports both graphics and presentation operations. But I can also imagine situations when there will be no single device that supports both. Then we are forced to use one device for graphics calculations (maybe like the old "graphics accelerator") and another device for presenting results on the screen (connected with the "accelerator" and a monitor). Unfortunately in such case we must use "general" Vulkan functions from the Vulkan Runtime or we need to store device-level functions for each used device (each device may have a different implementation of Vulkan functions). But, hopefully, such situations will be uncommon.

Creating a Device with a Swap Chain Extension Enabled

Now we can return to the CreateDevice() function. We have found the physical device that supports both graphics and presenting but not necessarily in a single queue family. We now need to create a logical device.

```
   std::vector<VkDeviceQueueCreateInfo> queue_create_infos;
   std::vector<float> queue_priorities = { 1.0f };

   queue_create_infos.push_back( {
      VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,      // VkStructureType
sType
      nullptr,                                         // const void
*pNext
      0,                                               // VkDeviceQueueCreateFlags
flags
      selected_graphics_queue_family_index,            // uint32_t
queueFamilyIndex
      static_cast<uint32_t>(queue_priorities.size()),  // uint32_t
queueCount
      &queue_priorities[0]                             // const float
*pQueuePriorities
   } );

   if( selected_graphics_queue_family_index != selected_present_queue_family_index ) {
     queue_create_infos.push_back( {
      VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,      // VkStructureType
sType
      nullptr,                                         // const void
*pNext
      0,                                               // VkDeviceQueueCreateFlags
flags
      selected_present_queue_family_index,             // uint32_t
queueFamilyIndex
      static_cast<uint32_t>(queue_priorities.size()),  // uint32_t
queueCount
      &queue_priorities[0]                             // const float
*pQueuePriorities
     } );
   }

   std::vector<const char*> extensions = {
     VK_KHR_SWAPCHAIN_EXTENSION_NAME
   };

   VkDeviceCreateInfo device_create_info = {
     VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,             // VkStructureType
sType
     nullptr,                                          // const void
*pNext
```

```
       0,                                           // VkDeviceCreateFlags
flags
       1,                                           // uint32_t
queueCreateInfoCount
       &queue_create_infos[0],                      // const VkDeviceQueueCreateInfo
*pQueueCreateInfos
       0,                                           // uint32_t
enabledLayerCount
       nullptr,                                     // const char * const
*ppEnabledLayerNames
       static_cast<uint32_t>(extensions.size()),    // uint32_t
enabledExtensionCount
       &extensions[0],                              // const char * const
*ppEnabledExtensionNames
       nullptr                                      // const VkPhysicalDeviceFeatures
*pEnabledFeatures
    };

    if( vkCreateDevice( Vulkan.PhysicalDevice, &device_create_info, nullptr,
&Vulkan.Device ) != VK_SUCCESS ) {
      printf( "Could not create Vulkan device!\n" );
      return false;
    }

    Vulkan.GraphicsQueueFamilyIndex = selected_graphics_queue_family_index;
    Vulkan.PresentQueueFamilyIndex = selected_present_queue_family_index;
    return true;
```

9.   Tutorial02.cpp, function CreateDevice()

As before, we need to fill a variable of VkDeviceCreateInfo type. To do this, we need to declare the queue families and how many queues each we want to enable. We do this through a pointer to a separate array with VkDeviceQueueCreateInfo elements. Here I declare a vector and I add one element, which defines one queue from the queue family that supports graphics operations. We use a vector because if graphics and presenting aren't supported by a single family, we will need to define two separate families. If a single family supports both we just define one member and declare that only one family is needed. If the indices of graphics and presentation families are different we need to declare additional members for our vector with VkDeviceQueueCreateInfo elements. In this case the VkDeviceCreateInfo structure must provide info about two different families. That's why a vector once again comes in handy (with its size() member function).

But we are not finished with device creation yet. We have to ask for the third extension related to a swap chain—a device-level "***VK_KHR_swapchain***" extension. As mentioned earlier, this extensions defines the actual support, implementation, and usage of a swap chain.

To ask for this extension, similarly at an instance level, we define an array (or a vector) which contains all the names of device-level extensions we want to enable. We provide an address of a first element of this array and the number of extensions we want to use. This extension also contains a definition of its name in a form of a #define VK_KHR_SWAPCHAIN_EXTENSION_NAME. We can use it inside our array (vector), and we don't have to worry about any typos.

This third extension introduces additional functions used to actually create, destroy, or in general manage swap chains. Before we can use them, we of course need to load pointers to these functions. They are from the device level so we will place them in a ListOfFunctions.inl file using VK_DEVICE_LEVEL_FUNCTION() macro:

```
    // From extensions
    #if defined(USE_SWAPCHAIN_EXTENSIONS)
    VK_DEVICE_LEVEL_FUNCTION( vkCreateSwapchainKHR )
    VK_DEVICE_LEVEL_FUNCTION( vkDestroySwapchainKHR )
```

```
VK_DEVICE_LEVEL_FUNCTION( vkGetSwapchainImagesKHR )
VK_DEVICE_LEVEL_FUNCTION( vkAcquireNextImageKHR )
VK_DEVICE_LEVEL_FUNCTION( vkQueuePresentKHR )
#endif
```

10. ListOfFunctions.inl

You can once again see that I'm checking whether a USE_SWAPCHAIN_EXTENSIONS preprocessor directive is defined. I define it only in projects that enable swap-chain extensions.

Now that we have created a logical devices we need to receive handles of a graphics queue and (if separate) presentation queue. I'm using two separate queue variables for convenience, but they both may contain the same handle.

After loading the device-level functions we can read requested queue handles. Here's the code for it:

```
vkGetDeviceQueue( Vulkan.Device, Vulkan.GraphicsQueueFamilyIndex, 0,
&Vulkan.GraphicsQueue );
vkGetDeviceQueue( Vulkan.Device, Vulkan.PresentQueueFamilyIndex, 0,
&Vulkan.PresentQueue );
return true;
```

11. Tutorial02.cpp, function GetDeviceQueue()

Creating a Semaphore

One last step before we can move to swap chain creation and usage is to create a semaphore. Semaphores are objects used for queue synchronization. They may be signaled or unsignaled. One queue may signal a semaphore (change its state from unsignaled to signaled) when some operations are finished, and another queue may wait on the semaphore until it becomes signaled. After that, the queue resumes performing operations submitted through command buffers.

```
VkSemaphoreCreateInfo semaphore create info = {
  VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO,      // VkStructureType          sType
  nullptr,                                       // const void*              pNext
  0                                              // VkSemaphoreCreateFlags   flags
};

if( (vkCreateSemaphore( Vulkan.Device, &semaphore_create_info, nullptr,
&Vulkan.ImageAvailableSemaphore ) != VK_SUCCESS) ||
    (vkCreateSemaphore( Vulkan.Device, &semaphore_create_info, nullptr,
&Vulkan.RenderingFinishedSemaphore ) != VK_SUCCESS) ) {
  printf( "Could not create semaphores!\n" );
  return false;
}

return true;
```

12. Tutorial02.cpp, function CreateSemaphores()

To create a semaphore we call the **vkCreateSemaphore()** function. It requires us to provide create information with three fields:

- sType – Standard structure type that must be set to VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO in this example.
- pNext – Standard parameter reserved for future use.
- flags – Another parameter that is reserved for future use and must equal zero.

Semaphores are used during drawing (or during presentation if we want to be more precise). I will describe the details later.

Creating a Swap Chain

We have enabled support for a swap chain, but before we can render anything on screen we must first create a swap chain from which we can acquire images on which we can render (or to which we can copy anything if we have rendered something into another image).

To create a swap chain, we call the **vkCreateSwapchainKHR()** function. It requires us to provide an address of a variable of type VkSwapchainCreateInfoKHR, which informs the driver about the properties of a swap chain that is being created. To fill this structure with the proper values, we must determine what is possible on a given hardware and platform. To do this we query the platform's or window's properties about the availability of and compatibility with several different features, that is, supported image formats or present modes (how images are presented on screen). So before we can create a swap chain we must check what is possible with a given platform and how we can create a swap chain.

Acquiring Surface Capabilities

First we must query for surface capabilities. To do this, we call the **vkGetPhysicalDeviceSurfaceCapabilitiesKHR()** function like this:

```
   VkSurfaceCapabilitiesKHR surface_capabilities;
   if( vkGetPhysicalDeviceSurfaceCapabilitiesKHR( Vulkan.PhysicalDevice,
Vulkan.PresentationSurface, &surface_capabilities ) != VK_SUCCESS ) {
     printf( "Could not check presentation surface capabilities!\n" );
     return false;
   }
```

13. Tutorial02.cpp, function CreateSwapChain()

Acquired capabilities contain important information about ranges (limits) that are supported by the swap chain, that is, minimal and maximal number of images, minimal and maximal dimensions of images, or supported transforms (some platforms may require transformations applied to images before these images may be presented).

Acquiring Supported Surface Formats

Next, we need to query for supported surface formats. Not all platforms are compatible with typical image formats like non-linear 32-bit RGBA. Some platforms don't have any preferences, but other may only support a small range of formats. We can only select one of the available formats for a swap chain or its creation will fail.

To query for surface formats, we must call the **vkGetPhysicalDeviceSurfaceFormatsKHR()** function. We can do it, as usual, twice: the first time to acquire the number of supported formats and a second time to acquire supported formats in an array prepared for this purpose. It can be done like this:

```
   uint32_t formats_count;
   if( (vkGetPhysicalDeviceSurfaceFormatsKHR( Vulkan.PhysicalDevice,
Vulkan.PresentationSurface, &formats_count, nullptr ) != VK_SUCCESS) ||
       (formats_count == 0) ) {
     printf( "Error occurred during presentation surface formats enumeration!\n" );
     return false;
   }

   std::vector<VkSurfaceFormatKHR> surface_formats( formats_count );
   if( vkGetPhysicalDeviceSurfaceFormatsKHR( Vulkan.PhysicalDevice,
Vulkan.PresentationSurface, &formats_count, &surface_formats[0] ) != VK_SUCCESS ) {
     printf( "Error occurred during presentation surface formats enumeration!\n" );
     return false;
   }
```

14. Tutorial02.cpp, function CreateSwapChain()

Acquiring Supported Present Modes

We should also ask for the available present modes, which tell us how images are presented (displayed) on the screen. The present mode defines whether an application will wait for v-sync or whether it will display an image immediately when it is available (which will probably lead to image tearing). I describe different present modes later.

To query for present modes that are supported on a given platform, we call the **vkGetPhysicalDeviceSurfacePresentModesKHR()** function. We can create code similar to this one:

```
    uint32_t present_modes_count;
    if( (vkGetPhysicalDeviceSurfacePresentModesKHR( Vulkan.PhysicalDevice,
Vulkan.PresentationSurface, &present_modes_count, nullptr ) != VK_SUCCESS) ||
        (present_modes_count == 0) ) {
      printf( "Error occurred during presentation surface present modes enumeration!\n"
);
      return false;
    }

    std::vector<VkPresentModeKHR> present_modes( present_modes_count );
    if( vkGetPhysicalDeviceSurfacePresentModesKHR( Vulkan.PhysicalDevice,
Vulkan.PresentationSurface, &present_modes_count, &present_modes[0] ) != VK_SUCCESS ) {
        printf( "Error occurred during presentation surface present modes enumeration!\n"
);
      return false;
    }
```

15. Tutorial02.cpp, function CreateSwapChain()

We now have acquired all the data that will help us prepare the proper values for a swap chain creation.

Selecting the Number of Swap Chain Images

A swap chain consists of multiple images. Several images (typically more than one) are required for the presentation engine to work properly, that is, one image is presented on the screen, another image waits in a queue for the next v-sync, and a third image is available for the application to render into.

An application may request more images. If it wants to use multiple images at once it may do so, for example, when encoding a video stream where every fourth image is a key frame and the application needs it to prepare the remaining three frames. Such usage will determine the number of images that will be automatically created in a swap chain: how many images the application requires at once for processing and how many images the presentation engine requires to function properly.

But we must ensure that the requested number of swap chain images is not smaller than the minimal required number of images and not greater than the maximal supported number of images (if there is such a limitation). And too many images will require much more memory. On the other hand, too small a number of images may cause stalls in the application (more about this later).

The number of images that are required for a swap chain to work properly and for an application to be able to render to is defined in the surface capabilities. Here is some code that checks whether the number of images is between the allowable min and max values:

```
    // Set of images defined in a swap chain may not always be available for application
to render to:
    // One may be displayed and one may wait in a queue to be presented
    // If application wants to use more images at the same time it must ask for more
images
    uint32_t image_count = surface_capabilities.minImageCount + 1;
    if( (surface_capabilities.maxImageCount > 0) &&
        (image_count > surface_capabilities.maxImageCount) ) {
      image_count = surface_capabilities.maxImageCount;
```

```
    }
    return image_count;
```

The minImageCount value in the surface capabilities structure gives the required minimum number of images for the swap chain to work properly. Here I'm selecting one more image than is required, and I also check whether I'm asking for too much. One more image may be useful for triple buffering-like presentation mode (if it is available). In more advanced scenarios we would also be required to store the number of images we want to use at the same time (at once). Let's say we want to encode a mentioned video stream and we need a key frame (every forth image frame) and the other three images. But a swap chain doesn't allow the application to operate on four images at once—only on three. We need to know that because we can only prepare two frames from a key frame, then we need to release them (give them back to a presentation engine) and only then can we acquire the last, third, non-key frame. This will become clearer later.

Selecting a Format for Swap Chain Images

Choosing a format for the images depends on the type of processing/rendering we want to do, that is, if we want to blend the application window with the desktop contents, an alpha value may be required. We must also know what color space is available and if we operate on linear or sRGB colorspace.

Each platform may support a different number of format-colorspace pairs. If we want to use specific ones we must make sure that they are available.

```
// If the list contains only one entry with undefined format
// it mean that there are no preferred surface formats and any can be choosen
if( (surface_formats.size() == 1) &&
    (surface_formats[0].format == VK_FORMAT_UNDEFINED) ) {
  return{ VK_FORMAT_R8G8B8A8_UNORM, VK_COLORSPACE_SRGB_NONLINEAR_KHR };
}

// Check if list contains most widely used R8 G8 B8 A8 format
// with nonlinear color space
for( VkSurfaceFormatKHR &surface_format : surface_formats ) {
  if( surface_format.format == VK_FORMAT_R8G8B8A8_UNORM ) {
    return surface_format;
  }
}

// Return the first format from the list
return surface_formats[0];
```

Earlier we requested a supported format which was placed in an array (a vector in our case). If this array contains only one value with an undefined format, that platform doesn't have any preferences. We can use any image format we want.

In other cases, we can use only one of the available formats. Here I'm looking for any (linear or not) 32-bit RGBA format. If it is available I can choose it. If there is no such format I will use any format from the list (hoping that the first is also the best and contains the format with the most precision).

Selecting the Size of the Swap Chain Images

Typically the size of swap chain images will be identical to the window size. We can choose other sizes, but we must fit into image size constraints. The size of an image that would fit into the current application window's size is available in the surface capabilities structure, in "currentExtent" member.

One thing worth noting is that a special value of "-1" indicates that the application's window size will be determined by the swap chain size, so we can choose whatever dimension we want. But we must still make sure that the selected size is not smaller and not greater than the defined minimum and maximum constraints.

Selecting the swap chain size may (and probably usually will) look like this:

```
  // Special value of surface extent is width == height == -1
  // If this is so we define the size by ourselves but it must fit within defined
confines
  if( surface_capabilities.currentExtent.width == -1 ) {
    VkExtent2D swap_chain_extent = { 640, 480 };
    if( swap_chain_extent.width < surface_capabilities.minImageExtent.width ) {
      swap_chain_extent.width = surface_capabilities.minImageExtent.width;
    }
    if( swap_chain_extent.height < surface_capabilities.minImageExtent.height ) {
      swap_chain_extent.height = surface_capabilities.minImageExtent.height;
    }
    if( swap_chain_extent.width > surface_capabilities.maxImageExtent.width ) {
      swap_chain_extent.width = surface_capabilities.maxImageExtent.width;
    }
    if( swap_chain_extent.height > surface_capabilities.maxImageExtent.height ) {
      swap_chain_extent.height = surface_capabilities.maxImageExtent.height;
    }
    return swap_chain_extent;
  }

  // Most of the cases we define size of the swap_chain images equal to current
window's size
  return surface_capabilities.currentExtent;
```
18. Tutorial02.cpp, function GetSwapChainExtent()

Selecting Swap Chain Usage Flags

Usage flags define how a given image may be used in Vulkan. If we want an image to be sampled (used inside shaders) it must be created with "sampled" usage. If the image should be used as a depth render target, it must be created with "depth and stencil" usage. An image without proper usage "enabled" cannot be used for a given purpose or the results of such operations will be undefined.

For a swap chain we want to render (in most cases) into the image (use it as a render target), so we must specify "color attachment" usage with VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT enum. In Vulkan this usage is always available for swap chains, so we can always set it without any additional checking. But for any other usage we must ensure it is supported – we can do this through a "supportedUsageFlags" member of surface capabilities structure.

```
  // Color attachment flag must always be supported
  // We can define other usage flags but we always need to check if they are supported
  if( surface_capabilities.supportedUsageFlags & VK_IMAGE_USAGE_TRANSFER_DST_BIT ) {
    return VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT;
  }
  return 0;
```
19. Tutorial02.cpp, function GetSwapChainUsageFlags()

In this example we define additional "transfer destination" usage which is required for image clear operation.

Selecting Pre-Transformations

On some platforms we may want our image to be transformed. This is usually the case on tablets when they are oriented in a way other than their default orientation. During swap chain creation we must specify what transformations

should be applied to images prior to presenting. We can, of course, use only the supported transforms, which can be found in a "supportedTransforms" member of acquired surface capabilities.

If the selected pre-transform is other than the current transformation (also found in surface capabilities) the presentation engine will apply the selected transformation. On some platforms this may cause performance degradation (probably not noticeable but worth mentioning). In the sample code, I don't want any transformations but, of course, I must check whether it is supported. If not, I'm just using the same transformation that is currently used.

```
    // Sometimes images must be transformed before they are presented (i.e. due to
device's orienation
    // being other than default orientation)
    // If the specified transform is other than current transform, presentation engine
will transform image
    // during presentation operation; this operation may hit performance on some
platforms
    // Here we don't want any transformations to occur so if the identity transform is
supported use it
    // otherwise just use the same transform as current transform
    if( surface_capabilities.supportedTransforms & VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR
) {
      return VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
    } else {
      return surface_capabilities.currentTransform;
    }
```

20. Tutorial02.cpp, function GetSwapChainTransform()

Selecting Presentation Mode

Present modes determine the way images will be processed internally by the presentation engine and displayed on the screen. In the past, there was just a single buffer that was displayed all the time. If we were drawing anything on it the draw operations (whole process of image creation) were visible.

Double buffering was introduced to prevent the visibility of drawing operations: one image was displayed and the second was used to render into. During presentation, the contents of the second image were copied into the first image (earlier) or (later) the images were swapped (remember SwapBuffers() function used in OpenGL applications?) which means that their pointers were exchanged.

Tearing was another issue with displaying images, so the ability to wait for the vertical blank signal was introduced if we wanted to avoid it. But waiting introduced another problem: input lag. So double buffering was changed into triple buffering in which we were drawing into two back buffers interchangeably and during v-sync the most recent one was used for presentation.

This is exactly what presentation modes are for: how to deal with all these issues, how to present images on the screen and whether we want to use v-sync.

Currently there are four presentation modes:

- IMMEDIATE. Present requests are applied immediately and tearing may be observed (depending on the frames per second). Internally the presentation engine doesn't use any queue for holding swap chain images.
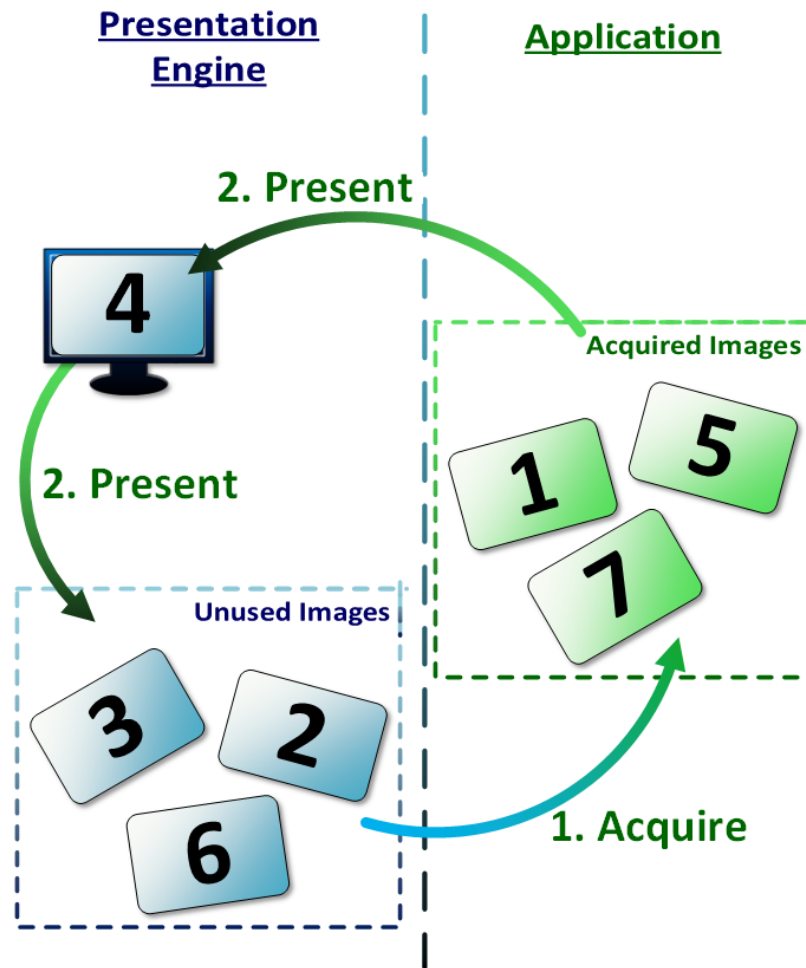
- FIFO. This mode is the most similar to OpenGL's buffer swapping with a swap interval set to 1. The image is displayed (replaces currently displayed image) only on vertical blanking periods, so no tearing should be visible. Internally, the presentation engine uses FIFO queue with "numSwapchainImages − 1" elements. Present requests are appended to the end of this queue. During blanking periods, the image from the beginning of the queue replaces the currently displayed image, which may become available to application. If all images are in the queue, the application has to wait until v-sync releases the currently displayed image. Only after that does it becomes available to the application and program may render image into it. This mode must always be available in all Vulkan implementations supporting swap chain extension.
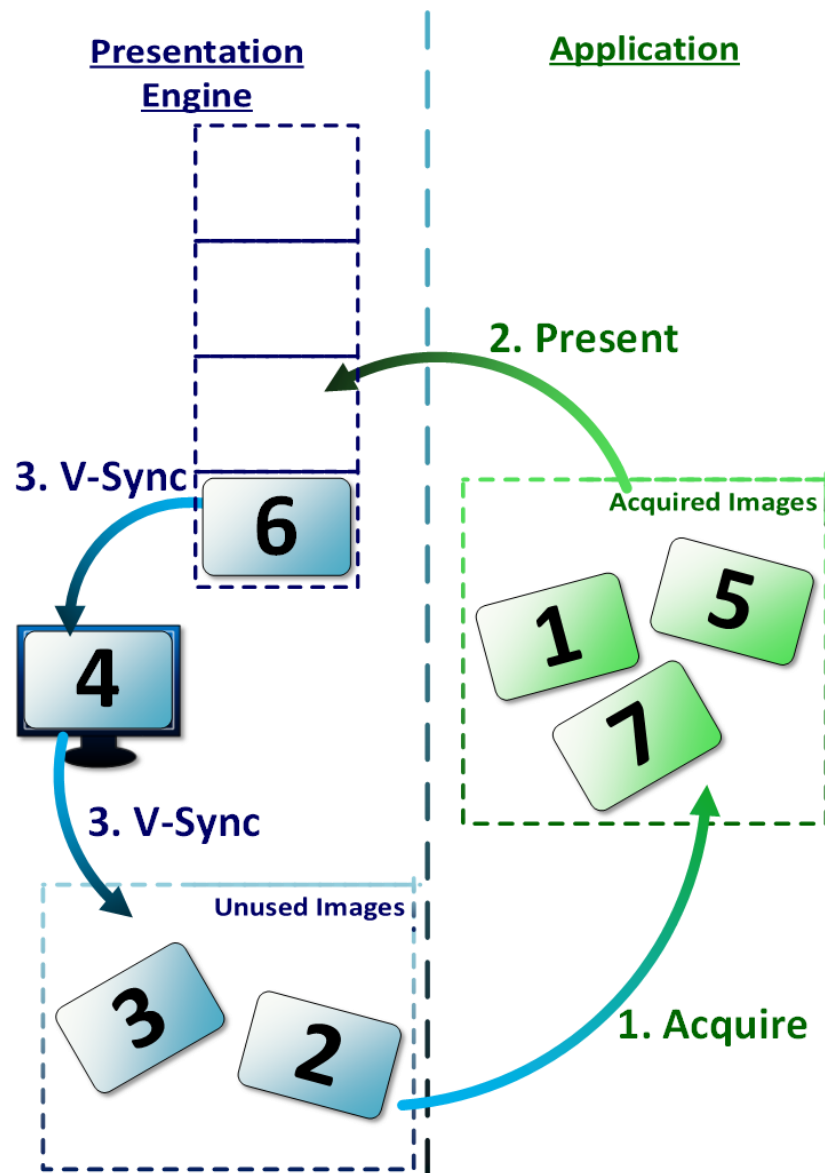
- FIFO RELAXED. This mode is similar to FIFO, but when the image is displayed longer than one blanking period it may be released immediately without waiting for another v-sync signal (so if we are rendering frames with lower frequency than screen's refresh rate, tearing may be visible)

- MAILBOX. In my opinion, this mode is the most similar to the mentioned triple buffering. The image is displayed only on vertical blanking periods and no tearing should be visible. But internally, the presentation engine uses the queue with only a single element. One image is displayed and one waits in the queue. If application wants to present another image it is not appended to the end of the queue but replaces the one that waits. So in the queue there is always the most recently generated image. This behavior is available if there are more than two images. For two images MAILBOX mode behaves similarly to FIFO (as we have to wait for the displayed image to be released, we don't have "spare" image which can be exchanged with the one that waits in the queue).
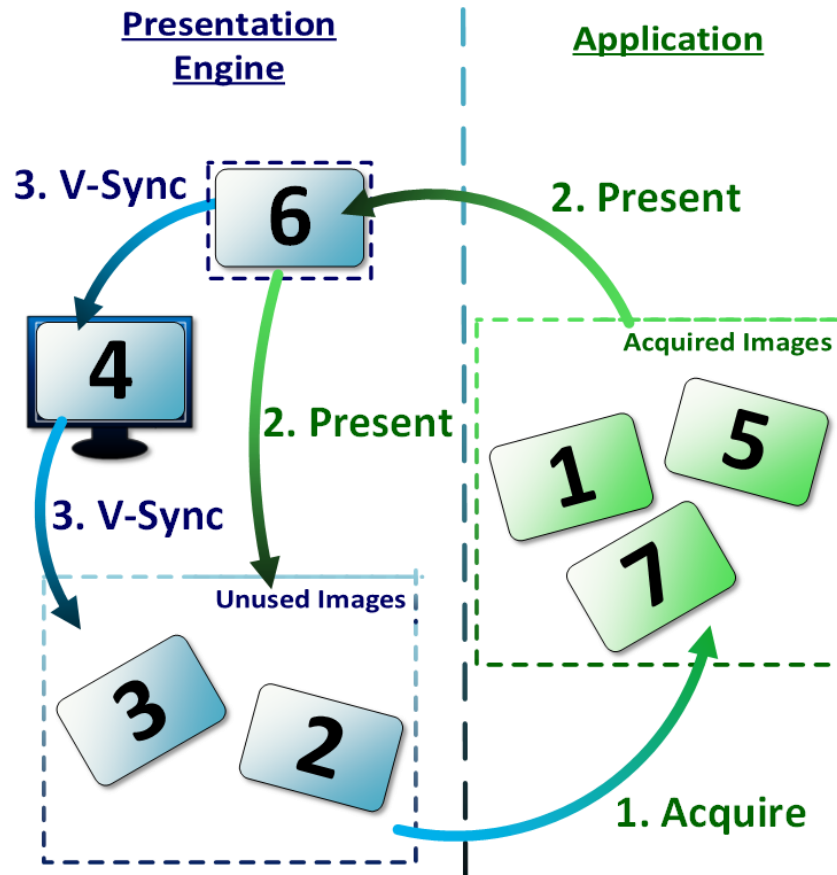
Deciding on which presentation mode to use depends on the type of operations we want to do. If we want to decode and display movies we want all frames to be displayed in a proper order. So the FIFO mode is in my opinion the best choice. But if we are creating a game, we usually want to display the most recently generated frame. In this case I suggest using MAILBOX because there is no tearing and input lag is minimized. The most recently generated image is displayed and the application doesn't need to wait for v-sync. But to achieve this behavior, at least three images must be created and this mode may not always be supported.

FIFO mode is always available and requires at least two images but causes application to wait for v-sync (no matter how many swap chain images were requested). Immediate mode is the fastest. As I understand it, it also requires two images but it doesn't make application wait for monitor refresh rate. On the downside it may cause image tearing. The choice is yours but, as always, we must make sure that the chosen presentation mode is supported.

Earlier we queried for available present modes, so now we must look for the one that best suits our needs. Here is the code in which I'm looking for MAILBOX mode:

```
    // FIFO present mode is always available
    // MAILBOX is the lowest latency V-Sync enabled mode (something like triple-
buffering) so use it if available
    for( VkPresentModeKHR &present_mode : present_modes ) {
      if( present_mode == VK_PRESENT_MODE_MAILBOX_KHR ) {
        return present_mode;
      }
    }
    return VK_PRESENT_MODE_FIFO_KHR;
```
21. Tutorial02.cpp, function GetSwapChainPresentMode()

Creating a Swap Chain

Now we have all the data necessary to create a swap chain. We have defined all the required values, and we are sure they fit into the given platform's constraints.

```
   uint32_t                         desired_number_of_images = GetSwapChainNumImages(
surface_capabilities );
   VkSurfaceFormatKHR           desired_format = GetSwapChainFormat( surface_formats );
   VkExtent2D                   desired_extent = GetSwapChainExtent(
surface_capabilities );
   VkImageUsageFlags            desired_usage = GetSwapChainUsageFlags(
surface_capabilities );
   VkSurfaceTransformFlagBitsKHR desired_transform = GetSwapChainTransform(
surface_capabilities );
   VkPresentModeKHR             desired_present_mode = GetSwapChainPresentMode(
present_modes );
   VkSwapchainKHR               old_swap_chain = Vulkan.SwapChain;

   if( static_cast<int>(desired_usage) == 0 ) {
     printf( "TRANSFER_DST image usage is not supported by the swap chain!" );
     return false;
   }

   VkSwapchainCreateInfoKHR swap_chain_create_info = {
     VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR,  // VkStructureType
sType
     nullptr,                                       // const void
*pNext
     0,                                             // VkSwapchainCreateFlagsKHR
flags
     Vulkan.PresentationSurface,                    // VkSurfaceKHR
surface
     desired_number_of_images,                      // uint32_t
minImageCount
     desired_format.format,                         // VkFormat
imageFormat
     desired_format.colorSpace,                     // VkColorSpaceKHR
imageColorSpace
     desired_extent,                                // VkExtent2D
imageExtent
     1,                                             // uint32_t
imageArrayLayers
     desired_usage,                                 // VkImageUsageFlags
imageUsage
     VK_SHARING_MODE_EXCLUSIVE,                     // VkSharingMode
imageSharingMode
     0,                                             // uint32_t
queueFamilyIndexCount
     nullptr,                                       // const uint32_t
*pQueueFamilyIndices
     desired_transform,                             // VkSurfaceTransformFlagBitsKHR
preTransform
     VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR,             // VkCompositeAlphaFlagBitsKHR
compositeAlpha
     desired_present_mode,                          // VkPresentModeKHR
presentMode
     VK_TRUE,                                       // VkBool32
clipped
     old_swap_chain                                 // VkSwapchainKHR
oldSwapchain
   };
```

```
  if( vkCreateSwapchainKHR( Vulkan.Device, &swap_chain_create_info, nullptr,
&Vulkan.SwapChain ) != VK_SUCCESS ) {
    printf( "Could not create swap chain!\n" );
    return false;
  }
  if( old_swap_chain != VK_NULL_HANDLE ) {
    vkDestroySwapchainKHR( Vulkan.Device, old_swap_chain, nullptr );
  }

  return true;
```
22. Tutorial02.cpp, function CreateSwapChain()

In this code example, at the beginning we gathered all the necessary data described earlier. Next we create a variable of type VkSwapchainCreateInfoKHR. It consists of the following members:

- sType – Normal structure type, which here must be a VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR.
- pNext – Pointer reserved for future use (for some extensions to this extension).
- flags – Value reserved for future use; currently must be set to zero.
- surface – A handle of a created surface that represents windowing system (our application's window).
- minImageCount – Minimal number of images the application requests for a swap chain (must fit into available constraints).
- imageFormat – Application-selected format for swap chain images; must be one of the supported surface formats.
- imageColorSpace – Colorspace for swap chain images; only enumerated values of format-colorspace pairs may be used for imageFormat and imageColorSpace (we can't use format from one pair and colorspace from another pair).
- imageExtent – Size (dimensions) of swap chain images defined in pixels; must fit into available constraints.
- imageArrayLayers – Defines the number of layers in a swap chain images (that is, views); typically this value will be one but if we want to create multiview or stereo (stereoscopic 3D) images, we can set it to some higher value.
- imageUsage – Defines how application wants to use images; it may contain only values of supported usages; color attachment usage is always supported.
- imageSharingMode – Describes image-sharing mode when multiple queues are referencing images (I will describe this in more detail later).
- queueFamilyIndexCount – The number of different queue families from which swap chain images will be referenced; this parameter matters only when VK_SHARING_MODE_CONCURRENT sharing mode is used.
- pQueueFamilyIndices – An array containing all the indices of queue families that will be referencing swap chain images; must contain at least queueFamilyIndexCount elements and as in queueFamilyIndexCount this parameter matters only when VK_SHARING_MODE_CONCURRENT sharing mode is used.
- preTransform – Transformations applied to the swap chain image before it can be presented; must be one of the supported values.
- compositeAlpha – This parameter is used to indicate how the surface (image) should be composited (blended?) with other surfaces on some windowing systems; this value must also be one of the possible values (bits) returned in surface capabilities, but it looks like opaque composition (no blending, alpha ignored) will be always supported (as most of the games will want to use this mode).
- presentMode – Presentation mode that will be used by a swap chain; only supported mode may be selected.
- clipped – Connected with ownership of pixels; in general it should be set to VK_TRUE if application doesn't want to read from swap chain images (like ReadPixels()) as it will allow some platforms to use more optimal

presentation methods; VK_FALSE value is used in some specific scenarios (if I learn more about these scenario I will write about them).

- oldSwapchain – If we are recreating a swap chain, this parameter defines an old swap chain that will be replaced by a newly created one.

So what's the matter with this sharing mode? Images in Vulkan can be referenced by queues. This means that we can create commands that use these images. These commands are stored in command buffers, and these command buffers are submitted to queues. Queues belong to different queue families. And Vulkan requires us to state how many different queue families and which of them are referencing these images through commands submitted with command buffers.

If we want to reference images from many different queue families at a time we can do so. In this case we must provide "concurrent" sharing mode. But this (probably) requires us to manage image data coherency by ourselves, that is, we must synchronize different queues in such a way that data in the images is proper and no hazards occur—some queues are reading data from images, but other queues haven't finished writing to them yet.

We may not specify these queue families and just tell Vulkan that only one queue family (queues from one family) will be referencing image at a time. This doesn't mean other queues can't reference these images. It just means they can't do it all at once, at the same time. So if we want to reference images from one family and then from another we must specifically tell Vulkan: "My image was used inside this queue family, but from now on another family, this one, will be referencing it." Such a transition is done using image memory barrier. When only one queue family uses a given image at a time, use the "exclusive" sharing mode.

If any of these requirements are not fulfilled, undefined behavior will probably occur and we may not rely on the image contents.

In this example we are using only one queue so we don't have to specify "concurrent" sharing mode and leave related parameters (queueFamilyCount and pQueueFamilyIndices) blank (or nulled, or zeroed).

So now we can call the **vkCreateSwapchainKHR()** function to create a swap chain and check whether this operation succeeded. After that (if we are recreating the swap chain, meaning this isn't the first time we are creating one) we should destroy the previous swap chain. I'll discuss this later.

Image Presentation

We now have a working swap chain that contains several images. To use these images as render targets, we can get handles to all images created with a swap chain, but we are not allowed to use them just like that. Swap chain images belong to and are owned by the swap chain. This means that the application cannot use these images until it asks for them. This also means that images are created and destroyed by the platform along with a swap chain (not by the application).

So when the application wants to render into a swap chain image or use it in any other way, it must first get access to it by asking a swap chain for it. If the swap chain makes us wait, we have to wait. And after the application finishes using the image it should "return" it by presenting it. If we forget about returning images to a swap chain, we will soon run out of images and nothing will display on the screen.

The application may also request access to more images at once but they must be available. Acquiring access may require waiting. In corner cases, when there are too few images in a swap chain and the application wants to access too many of them, or if we forget about returning images to a swap chain, the application may even wait an infinite amount of time.

Given that there are (usually) at least two images, it may sound strange that we have to wait, but it is quite reasonable. Not all images are available for the application because they are used by the presentation engine. Usually one image is displayed. Additional images may also be required for the presentation engine to work properly. So we can't use them because it could block the presentation engine in some way. We don't know its internal mechanisms and algorithms or the requirements of the OS the application is executed on. So the availability of images may depend on many factors:

internal implementation, OS, number of created images, number of images the application wants to use at a single time and on the selected presentation mode, which is the most important factor from the perspective of this tutorial.

In immediate mode, one image is always presented. Other images (at least one) are available for application. When the application posts a presentation request ("returns" an image), the image that was displayed is replaced with the new one. So if two images are created, only one image may be available for application at a single time. When the application asks for another image, it must "return" the previous one. If it wants two images at a time, it must create a swap chain with more images or it will wait forever. When we request more images, in immediate mode, the application can ask for (own) "imageCount – 1" images at a time.

In FIFO mode one image is displayed, and the rest are placed in a FIFO queue. The length of this queue is always equal to "imageCount – 1." At first, all images may be available to the application (because the queue is empty and no image is presented). When the application presents an image ("returns" it to a swap chain), it is appended to the end of the queue. So as soon as the queue fills, the application has to wait for another image until the displayed image is released during the vertical blanking period. Images are always displayed in the same order they were presented in by the application. When the v-sync signal appears, the first image from the queue replaces the image that was displayed. The previously displayed image (the released one) may become available to the application as it becomes unused (isn't presented and is not waiting in the queue). If all images are in the queue, the application will wait for the next blanking period to access another image. If rendering takes longer than the refresh rates, the application will not have to wait at all. This behavior doesn't change when there are more images. The internal swap chain queue has always "imageCount – 1" elements.

The last mode available for the time being is MAILBOX. As previously mentioned, this mode is most similar to the "traditional" triple buffering. One image is always displayed. The second image waits in a single-element queue (it always has place for only one element). The rest of the images may be available for the application. When the application presents an image, the image replaces the one waiting in the queue. The image in the queue gets displayed only during blanking periods, but the application doesn't need to wait for the next image (when there are more than two images). MAILBOX mode with only two images behaves identically to FIFO mode—the application must wait for the v-sync signal to acquire the next image. But with at least three images it immediately may acquire the image that was replaced by the "presented" image (the one waiting in the queue). That's why I requested one more image than the minimal number. If MAILBOX mode is available I want to use it in a manner similar to triple buffering (maybe the first thing to do is to check what mode is available and after that choose the number of swap chain images based on the selected presentation mode).

I hope these examples help you understand why the application must ask for an image if it wants to use any. In Vulkan we can only do what is allowed and required—not less and usually not too much more.

```
  uint32_t image_index;
  VkResult result = vkAcquireNextImageKHR( Vulkan.Device, Vulkan.SwapChain, UINT64_MAX,
Vulkan.ImageAvailableSemaphore, VK_NULL_HANDLE, &image_index );
  switch( result ) {
    case VK_SUCCESS:
    case VK_SUBOPTIMAL_KHR:
      break;
    case VK_ERROR_OUT_OF_DATE_KHR:
      return OnWindowSizeChanged();
    default:
      printf( "Problem occurred during swap chain image acquisition!\n" );
      return false;
  }
```

23. Tutorial02.cpp, function Draw()

To access an image, we must call the **vkAcquireNextImageKHR()** function. During the call we must specify (apart from the device handle like in almost all other functions) a swap chain from which we want to use an image, a timeout, a semaphore, and a fence object. A function, in case of a success, will store the image index in the variable we provided the address of. Why an index and not the (handle to) image itself? Such a behavior may be convenient (that is, during the

"preprocessing" phase when we want to prepare as much data needed for rendering as possible to not waste time during typical frame rendering) but I will describe it later. Just remember that we can check what images were created in a swap chain if we want (we just can't use them until we are allowed). An array of images will be provided upon such query. And the **vkAcquireNextImageKHR()** function stores an index into this very array.

We have to specify a timeout because sometimes images may not be immediately available. Trying to use an image before we are allowed to will cause an undefined behavior. Specifying a timeout gives the presentation engine time to react. If it needs to wait for the next vertical blanking period it can do so and we give it a time. So this function will block until the given time has passed. We can provide maximal available value so the function may even block indefinitely. If we provide 0 for the timeout, the function will return immediately. If any image was available at the time the call occurred it will be provided immediately. If there was no available image, an error will be returned stating that the image was not yet ready.

Once we have our image we can use it however we want. Images are processed or referenced by commands stored in command buffers. We can prepare command buffers earlier (to save as much processing time for rendering as we can) and use or submit them here. Or we can prepare the commands now and submit them when we're done. In Vulkan, creating command buffers and submitting them to queues is the only way to cause operations to be performed by the device.

When command buffers are submitted to queues, all their commands start being processed. But a queue cannot use an image until it is allowed to, and the semaphore we created earlier is for internal queue synchronization—before the queue starts processing commands that reference a given image, it should wait on this semaphore (until it gets signaled). But this wait doesn't block an application. There are two synchronization mechanisms for accessing swap chain images: (1) a timeout, which may block an application but doesn't stop queue processing, and (2) a semaphore, which doesn't block the application but blocks selected queues.

We now know (theoretically) how to render anything (through command buffers). So let's now imagine that inside a command buffer we are submitting some rendering operations take place. But before the processing will start, we should tell the queue (on which this rendering will occur) to wait. This all is done within one submit operation.

```
   VkPipelineStageFlags wait_dst_stage_mask = VK_PIPELINE_STAGE_TRANSFER_BIT;
   VkSubmitInfo submit_info = {
     VK_STRUCTURE_TYPE_SUBMIT_INFO,               // VkStructureType              sType
     nullptr,                                     // const void                  *pNext
     1,                                           // uint32_t
waitSemaphoreCount
     &Vulkan.ImageAvailableSemaphore,             // const VkSemaphore
*pWaitSemaphores
     &wait_dst_stage_mask,                        // const VkPipelineStageFlags
*pWaitDstStageMask;
     1,                                           // uint32_t
commandBufferCount
     &Vulkan.PresentQueueCmdBuffers[image_index], // const VkCommandBuffer
*pCommandBuffers
     1,                                           // uint32_t
signalSemaphoreCount
     &Vulkan.RenderingFinishedSemaphore           // const VkSemaphore
*pSignalSemaphores
   };

   if( vkQueueSubmit( Vulkan.PresentQueue, 1, &submit_info, VK_NULL_HANDLE ) !=
VK_SUCCESS ) {
     return false;
   }
```

24. Tutorial02.cpp, function Draw()

First we prepare a structure with information about the types of operations we want to submit to the queue. This is done through VkSubmitInfo structure. It contains the following fields:

- sType – Standard structure type; here it must be set to VK_STRUCTURE_TYPE_SUBMIT_INFO.
- pNext – Standard pointer reserved for future use.
- waitSemaphoreCount – Number of semaphores we want the queue to wait on before it starts processing commands from command buffers.
- pWaitSemaphores – Pointer to an array with semaphore handles on which queue should wait; this array must contain at least waitSemaphoreCount elements.
- pWaitDstStageMask – Pointer to an array with the same amount of elements as pWaitSemaphores array; it describes the pipeline stages at which each (corresponding) semaphore wait will occur; in our example, the queue may perform some operations before it starts using the image from the swap chain so there is no reason to block all of the operations; the queue may start processing some drawing commands and until pipeline gets to the stage in which the image is used, it will wait.
- commandBufferCount – Number of command buffers we are submitting for execution.
- pCommandBuffers – Pointer to an array with command buffers handles which must contain at least commandBufferCount elements.
- signalSemaphoreCount – Number of semaphores we want the queue to signal after processing all the submitted command buffers.
- pSignalSemaphores – Pointer to an array of at least signalSemaphoreCount elements with semaphore handles; these semaphores will be signaled after the queue has finished processing commands submitted within this submit information.

In this example we are telling the queue to wait only on one semaphore, which will be signaled by the presentation engine when the queue can safely start processing commands referencing the swap chain image.

We also submit just one simple command buffer. It was prepared earlier (I will describe how to do it later). It only clears the acquired image. But this is enough for us to see the selected color in our application's window and to see that the swap chain is working properly.

In the code above, the command buffers are arranged in an array (a vector, to be more precise). To make it easier to submit the proper command buffer—the one that references the currently acquired image—I prepared a separate command buffer for each swap chain image. The index of an image that the **vkAcquireNextImageKHR()** function provides can be used right here. Using image handles (in similar scenarios) would require creating maps that would translate the handle into a specific command buffer or index. On the other hand, normal numbers can be used to just select a specific array element. This is why this function gives us indices and not image handles.

After we have submitted a command buffer, all the processing starts in the background, on "hardware." Next, we want to present a rendered image. Presenting means that we want our image to be displayed and that we are "giving it back" to the swap chain. The code to do this might look like this:

```
   VkPresentInfoKHR present_info = {
     VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,               // VkStructureType              sType
     nullptr,                                          // const void              *pNext
     1,                                                // uint32_t
waitSemaphoreCount
     &Vulkan.RenderingFinishedSemaphore,               // const VkSemaphore
*pWaitSemaphores
     1,                                                // uint32_t
swapchainCount
     &Vulkan.SwapChain,                                // const VkSwapchainKHR
*pSwapchains
     &image_index,                                     // const uint32_t
*pImageIndices
```

```
      nullptr                                              // VkResult
*pResults
   };
   result = vkQueuePresentKHR( Vulkan.PresentQueue, &present_info );

   switch( result ) {
     case VK_SUCCESS:
       break;
     case VK_ERROR_OUT_OF_DATE_KHR:
     case VK_SUBOPTIMAL_KHR:
       return OnWindowSizeChanged();
     default:
       printf( "Problem occurred during image presentation!\n" );
       return false;
   }

   return true;
```

25. Tutorial02.cpp, function Draw()

An image (or images) is presented by calling the **vkQueuePresentKHR()** function. It may be perceived as submitting a command buffer with only operation: presentation.

To present an image we must specify what images should be presented from how many and from which swap chains. We can present many images from many swap chains at once (that is, to multiple windows) but only one image from a single swap chain can be presented at once. We provide this information through the VkPresentInfoKHR structure, which contains the following fields:

- sType – Standard structure type, it must be a VK_STRUCTURE_TYPE_PRESENT_INFO_KHR here.
- pNext – Parameter reserved for future use.
- waitSemaphoreCount – The number of semaphores we want the queue to wait on before it presents images.
- pWaitSemaphores – Pointer to an array with semaphore handles on which the queue should wait; this array must contain at least waitSemaphoreCount elements.
- swapchainCount – The number of swapchains to which we would like to present images.
- pSwapchains – An array with swapchainCount elements that contains handles of all the swap chains that we want to present images to; any single swap chain may only appear once in this array.
- imageIndices – An array with swapchainCount elements that contains indices of images that we want to present; each element of this array corresponds to a swap chain in a pSwapchains array; the image index is the index into the array of each swap chain's images (see the next section).
- pResults – A pointer to an array of at least swapchainCount element; this parameter is optional and can be set to null, but if we provide such an array, the result of the presenting operation will be stored in each of its elements, for each swap chain respectively; a single value returned by the whole function is the same as the worst result value from all swap chains.

Now that we have prepared this structure, we can use it to present an image. In this example I'm just presenting a single image from a single swap chain.

Each operation that is performed (or submitted) by calling **vkQueue…()** functions (this includes presenting) is appended to the end of the queue for processing. Operations are processed in the order in which they were submitted. For a presentation, we are presenting an image after submitting other command buffers. So the present queue will start presenting an image after the processing of all the command buffers is done. This ensures that the image will be presented after we are done using it (rendering into it) and an image with correct contents will be displayed on the screen. But in this example we submit drawing (clearing) operations and a present operation to the same queue: the PresentQueue. We are doing only simple operations that are allowed to be done on a present queue.

If we want to perform drawing operations on a queue that is different than the present operation, we need to synchronize the queues. This is done, again, with semaphores, which is the reason why we created two semaphores (the second one may not be necessary in this example, as we render and present using the same queue, but I wanted to show how it should be done in the correct way).

The first semaphore is for presentation engine to tell the queue that it can safely use (reference/render into) an image. The second semaphore is for us. It is signaled when the operations on the image (rendering into it) are done. The submit info structure has a field called pSignalSemaphores. It is an array of semaphore handles that will be signaled after processing of all of the submitted command buffers is finished. So we need to tell the second queue to wait on this second semaphore. We store the handle of our second semaphore in the pWaitSemaphores field of a VkPresentInfoKHR structure. And the queue to which we are submitting the present operation will wait, thanks to this second semaphore, until we are done rendering into a given image.

And that's it. We have displayed our first image using Vulkan!

Checking What Images Were Created in a Swap Chain

Previously I mentioned swap chain's image indices. Here in this code sample, I show you more specifically what I was talking about.

```
uint32_t image_count = 0;
if( (vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count, nullptr
) != VK_SUCCESS) ||
    (image_count == 0) ) {
  printf( "Could not get the number of swap chain images!\n" );
  return false;
}

std::vector<VkImage> swap_chain_images( image_count );
if( vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count,
&swap_chain_images[0] ) != VK_SUCCESS ) {
  printf( "Could not get swap chain images!\n" );
  return false;
}
```

26. -

This code sample is a fragment of an imaginary function that checks how many and what images were created inside a swap chain. It is done by a traditional "double-call," this time using a **vkGetSwapchainImagesKHR()** function. First we call it with the last parameter set to null. This way the number of all images created in a swap chain is stored in an "image_count" variable and we know how much storage we need to prepare for the handles of all images. The second time we call this function, we achieve the handles in the array we have provided the address of through the last parameter.

Now we know all the images that the swap chain is using. For the **vkAcquireNextImageKHR()** function and VkPresentInfoKHR structure, the indices I referred to are the indices into this array, an array "returned" by the **vkGetSwapchainImagesKHR()** function. It is called an array of a swap chain's presentable images. And if any function, in the case of a swap chain, wants us to provide an index or returns an index, it is the index of an image in this very array.

Recreating a Swap Chain

Previously, I mentioned that sometimes we must recreate a swap chain, and I also said that the old swap chain must be destroyed. The **vkAcquireNextImageKHR()** and **vkQueuePresentKHR()** functions return a result that sometimes causes the **OnWindowSizeChanged()** function to be called. This function recreates the swap chain.

Sometimes a swap chain gets old. This means that the properties of the surface, platform, or application window properties changed in such a way that the current swap chain cannot be used any more. The most obvious (and unfortunately not so good) example is when the window's size changed. We cannot create a swap chain image nor can

we change its size. The only possibility is to destroy and recreate a swap chain. There are also situations in which we can still use a swap chain, but it may no longer be optimal for surface it was created for.

These situations are notified by the return codes of the **vkAcquireNextImageKHR()** and **vkQueuePresentKHR()** functions.

When the VK_SUBOPTIMAL_KHR value is returned, we can still use the current swap chain for presentation. It will still work but not optimally (that is, color precision will be worse). It is advised to recreate swap chain when there is an opportunity. A good example is when we have performed performance-heavy rendering and after acquiring the image we are informed that our image is suboptimal. We don't want to waste all this processing and make the user wait much longer for another frame. We just present the image and recreate the swap chain as soon as there is an opportunity.

When VK_ERROR_OUT_OF_DATE_KHR is returned we cannot use current swap chain and we must recreate it immediately. We cannot present using the current swap chain; this operation will fail. We have to recreate a swap chain as soon as possible.

I have mentioned that changing the window size is the most obvious, but not so good, example of surface properties' changes after which we should recreate a swap chain. In this situation we should recreate a swap chain, but we may not be notified about it with the mentioned return codes. We should monitor the window size changes by ourselves using OS-specific code. And that's why the name of this function in our source is **OnWindowSizeChanged**. This function is called every time a window's size had changed. But as this function only recreates a swap chain (and command buffers) the same function can be called here.

Recreation is done the same way as creation. There is a structure member in which we provide a swap chain that the new one should replace. But we must implicitly destroy the old swap chain after we create the new one.

Quick Dive into Command Buffers

You now know a lot about swap chains, but there is still one important thing you need to know. To explain it, I will briefly show you how to prepare drawing commands. That one last important thing about swap chains is connected with drawing and preparing command buffers. I will present only information about how to clear images, but it is enough to check whether our swap chain is working as it should.

In the first tutorial, I described queues and queue families. If we want to execute commands on a device we submit them to queues through command buffers. To put it in other words: commands are encapsulated inside command buffers. Submitting such buffers to queues causes devices to start processing commands that were recorded in them. Do you remember OpenGL's drawing lists? We could prepare lists of commands that cause the geometry to be drawn in a form of a list of, well, drawing commands. The situation in Vulkan is similar, but far more flexible and advanced.

Creating Command Buffer Memory Pool

To store commands, a command buffer needs some storage. To prepare space for commands we create a pool from which the buffer can allocate its memory. We don't specify the amount of space—it is allocated dynamically when the buffer is built (recorded).

Remember that command buffers can be submitted only to proper queue families and only the types of operations compatible with a given family can be submitted to a given queue. Also, the command buffer itself is not connected with any queue or queue family, but the memory pool from which buffer allocates its memory is. So each command buffer that takes memory from a given pool can only be submitted to a queue from a proper queue family—a family from (inside?) which the memory pool was created. If there are more queues created from a given family, we can submit a command buffer to any one of them; the family index is the most important thing here.

```
VkCommandPoolCreateInfo cmd_pool_create_info = {
    VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO,      // VkStructureType
sType
```

```
        nullptr,                                        // const void*
pNext
        0,                                              // VkCommandPoolCreateFlags
flags
        Vulkan.PresentQueueFamilyIndex                  // uint32_t
queueFamilyIndex
    };

    if( vkCreateCommandPool( Vulkan.Device, &cmd_pool_create_info, nullptr,
&Vulkan.PresentQueueCmdPool ) != VK_SUCCESS ) {
        printf( "Could not create a command pool!\n" );
        return false;
    }
```

To create a pool for command buffer(s) we call a **vkCreateCommandPool()** function. It requires us to provide (an address of) a variable of structure type VkCommandPoolCreateInfo. It contains the following members:

- sType – A usual type of structure that must be equal to VK_STRUCTURE_TYPE_CMD_POOL_CREATE_INFO in this occasion.
- pNext – Pointer reserved for future use.
- flags – Value reserved for future use.
- queueFamilyIndex – Index of a queue family for which this pool is created.

For our test application, we use only one queue from a presentation family, so we should use its index. Now we can call the **vkCreateCommandPool()** function and check whether it succeeded. If yes, the handle to the command pool will be stored in a variable we have provided the address of.

Allocating Command Buffers

Next, we need to allocate the command buffer itself. Command buffers are not created in a typical way; they are allocated from pools. Other objects that take their memory from pool objects are also allocated (the pools themselves are created). That's why there is a separation in the names of the functions vkCreate…() and vkAllocate…().

As described earlier, I allocate more than one command buffer—one for each swap chain image that will be referenced by the drawing commands. So each time we acquire an image from a swap chain we can submit/use the proper command buffer.

```
    uint32_t image_count = 0;
    if( (vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count, nullptr
) != VK_SUCCESS) ||
        (image_count == 0) ) {
      printf( "Could not get the number of swap chain images!\n" );
      return false;
    }

    Vulkan.PresentQueueCmdBuffers.resize( image_count );

    VkCommandBufferAllocateInfo cmd_buffer_allocate_info = {
      VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO, // VkStructureType
sType
      nullptr,                                        // const void*
pNext
      Vulkan.PresentQueueCmdPool,                     // VkCommandPool
commandPool
      VK_COMMAND_BUFFER_LEVEL_PRIMARY,                // VkCommandBufferLevel
level
      image_count                                     // uint32_t
bufferCount
```

```
    };
    if( vkAllocateCommandBuffers( Vulkan.Device, &cmd_buffer_allocate_info,
&Vulkan.PresentQueueCmdBuffers[0] ) != VK_SUCCESS ) {
      printf( "Could not allocate command buffers!\n" );
      return false;
    }

    if( !RecordCommandBuffers() ) {
      printf( "Could not record command buffers!\n" );
      return false;
    }
    return true;
```
28. Tutorial02.cpp, function CreateCommandBuffers()

First we need to know how many swap chain images were created (a swap chain may create more images than we have specified). This was explained in an earlier section. We call the **vkGetSwapchainImagesKHR()** function with the last parameter set to null. Right now we don't need the handles of images, only their total number. After that we prepare an array (vector) for a proper number of command buffers and we can create a proper number of command buffers. To do this we call the **vkAllocateCommandBuffers()** function. It requires us to prepare a structured variable of type VkCommandBufferAllocateInfo, which contains the following fields:

- sType – Type of a structure, this time equal to VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO.
- pNext – Normal parameter reserved for future use.
- commandPool – Command pool from which the buffer will be allocating its memory during commands recording.
- level – Type (level) of command buffer. There are two levels: primary and secondary. Secondary command buffers may only be referenced (used) from primary command buffers. Because we don't have any other buffers, we need to create primary buffers here.
- bufferCount – The number of command buffers we want to create at once.

After calling the **vkAllocateCommandBuffers()** function, we need to check whether the buffer creations succeeded. If yes, we are done allocating command buffers and we are ready to record some (simple) commands.

Recording Command Buffers

Command recording is the most important operation we will be doing in Vulkan. The recording itself also requires us to provide a lot of information. The more information, the more complicated the drawing commands are.

Here is a set of variables required (in this tutorial) to record command buffers:

```
  uint32_t image_count = static_cast<uint32_t>(Vulkan.PresentQueueCmdBuffers.size());

  std::vector<VkImage> swap_chain_images( image_count );
  if( vkGetSwapchainImagesKHR( Vulkan.Device, Vulkan.SwapChain, &image_count,
&swap_chain_images[0] ) != VK_SUCCESS ) {
    printf( "Could not get swap chain images!\n" );
    return false;
  }

  VkCommandBufferBeginInfo cmd_buffer_begin_info = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,  // VkStructureType
sType
    nullptr,                                      // const void
*pNext
    VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, // VkCommandBufferUsageFlags
flags
    nullptr                                       // const
VkCommandBufferInheritanceInfo  *pInheritanceInfo
```

```
    };

  VkClearColorValue clear_color = {
    { 1.0f, 0.8f, 0.4f, 0.0f }
  };

  VkImageSubresourceRange image_subresource_range = {
    VK_IMAGE_ASPECT_COLOR_BIT,                      // VkImageAspectFlags
aspectMask
    0,                                              // uint32_t
baseMipLevel
    1,                                              // uint32_t
levelCount
    0,                                              // uint32_t
baseArrayLayer
    1                                               // uint32_t
layerCount
    };
```
<div align="right">

29. Tutorial02.cpp, function RecordCommandBuffers()

</div>

First we get the handles of all the swap chain images, which will be used in drawing commands (we will just clear them to one single color but nevertheless we will use them). We already know the number of images, so we don't have to ask for it again. The handles of images are stored in a vector after calling the **vkGetSwapchainImagesKHR()** function.

Next, we need to prepare a variable of structured type VkCommandBufferBeginInfo. It contains the information necessary in more typical rendering scenarios (like render passes). We won't be doing such operations here and that's why we can set almost all parameters to zeros or nulls. But, for clarity, the structure contains the following fields:

- sType – Structure type, this time it must be set to VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO.
- pNext – Pointer reserved for future use, leave it to null.
- flags – Parameter defining preferred usage of a command buffer.
- pInheritanceInfo – Parameter pointing to another structure that is used in more typical rendering scenarios.

Command buffers gather commands. To store commands in command buffers, we record them. The above structure provides some necessary information for the driver to prepare for and optimize the recording process.

In Vulkan, command buffers are divided into primary and secondary. Primary command buffers are typical command buffers similar to drawing lists. They are independent, individual "beings" and they (and only they) may be submitted to queues. Secondary command buffers can also store commands (we also record them), but they may only be referenced from within primary command buffers (we can call secondary command buffers from within primary command buffers like calling OpenGL's drawing lists from another drawing lists). We can't submit secondary command buffers directly to queues.

All of this information will be described in more detail in a forthcoming tutorial.

In this simple example we want to clear our images with one single value. So next we set up a color that will be used for clearing. You can pick any value you like. I used a light orange color.

The last variable in the code above specifies the parts of the image that our operations will be performed on. Our image consists of only one mipmap level and one array level (no stereoscopic buffers, and so on). We set values in the VkImageSubresourceRange structure accordingly. This structure contains the following fields:

- aspectMask – Depends on the image format as we are using images as color render targets (they have "color" format) so we specify "color aspect" here.
- baseMipLevel – First mipmap level that will be accessed (modified).
- levelCount – Number of mipmap levels on which operations will be performed (including the base level).

- baseArrayLayer – First array layer that will be accessed (modified).
- arraySize – Number of layers the operations will be performed on (including the base layer).

We are almost ready to record some buffers.

Image Layouts and Layout Transitions

The last variable required in the above code example (of type VkImageSubresourceRange) specifies the parts of the image that operations will be performed on. In this lesson we only clear an image. But we also need to perform *resource transitions*. Remember the code when we selected a use for a swap chain image before the swap chain itself was created? Images may be used for different purposes. They may be used as render targets, as textures that can be sampled from inside the shaders, or as a data source for copy/blit operations (data transfers). We must specify different usage flags during image creation for the different types of operations we want to perform with or on images. We can specify more usage flags if we want (if they are supported; "color attachment" usage is always available for swap chains). But image usage specification is not the only thing we need to do. Depending on the type of operation, images may be differently allocated or may have a different layout in memory. Each type of image operation may be connected with a different "image layout." We can use a general layout that is supported by all operations, but it may not provide the best performance. For specific usages we should always use dedicated layouts.

If we create an image with different usages in mind and we want to perform different operations on it, we must change the image's current layout before we can perform each type of operation. To do this, we must transition from the current layout to another layout that is compatible with the operations we are about to execute.

Each image we create is created (generally) with an undefined layout, and we must transition from it to another layout if want to use the image. But swap-chain-created images have VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR layouts. This layout, as the name suggests, is designed for the image to be used (presented) by the presentation engine (that is, displayed on the screen). So if we want to perform some operations on swap chain images, we need to change their layouts to ones compatible with the desired operations. And after we have finished with processing the images (that is, rendering into them) we need to transition their layouts back to the VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR. Otherwise, the presentation engine will not be able to use these images and undefined behavior may occur.

To transition from one layout to another one, *image memory barriers* are used. With them we can specify the old layout (current) we are transitioning from and the new layout we are transitioning to. The old layout must always be equal to the current or undefined layout. When we specify the old layout as undefined, image contents may be discarded during transition. This allows the driver to perform some optimizations. If we want to preserve image contents we must specify a layout that is equal to the current layout.

The last variable of type VkImageSubresourceRange in the code example above is also used for image transitions. It defines what "parts" of the image are changing their layout and is required when preparing an image memory barrier.

Recording Command Buffers

The last step is to record a command buffer for each swap chain image. We want to clear the image to some arbitrary color. But first we need to change the image layout and change it back after we are done. Here is the code that does that:

```
for( uint32_t i = 0; i < image_count; ++i ) {
  VkImageMemoryBarrier barrier_from_present_to_clear = {
    VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,      // VkStructureType
sType
    nullptr,                                     // const void
*pNext
    VK_ACCESS_MEMORY_READ_BIT,                   // VkAccessFlags
srcAccessMask
    VK_ACCESS_TRANSFER_WRITE_BIT,                // VkAccessFlags
dstAccessMask
    VK_IMAGE_LAYOUT_UNDEFINED,                   // VkImageLayout
oldLayout
```

```
        VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,        // VkImageLayout
newLayout
        Vulkan.PresentQueueFamilyIndex,              // uint32_t
srcQueueFamilyIndex
        Vulkan.PresentQueueFamilyIndex,              // uint32_t
dstQueueFamilyIndex
        swap_chain_images[i],                        // VkImage
image
        image_subresource_range                      // VkImageSubresourceRange
subresourceRange
    };

    VkImageMemoryBarrier barrier_from_clear_to_present = {
        VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,      // VkStructureType
sType
        nullptr,                                     // const void
*pNext
        VK_ACCESS_TRANSFER_WRITE_BIT,                // VkAccessFlags
srcAccessMask
        VK_ACCESS_MEMORY_READ_BIT,                   // VkAccessFlags
dstAccessMask
        VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,        // VkImageLayout
oldLayout
        VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,             // VkImageLayout
newLayout
        Vulkan.PresentQueueFamilyIndex,              // uint32_t
srcQueueFamilyIndex
        Vulkan.PresentQueueFamilyIndex,              // uint32_t
dstQueueFamilyIndex
        swap_chain_images[i],                        // VkImage
image
        image_subresource_range                      // VkImageSubresourceRange
subresourceRange
    };

    vkBeginCommandBuffer( Vulkan.PresentQueueCmdBuffers[i], &cmd_buffer_begin_info );
    vkCmdPipelineBarrier( Vulkan.PresentQueueCmdBuffers[i],
VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0, 0, nullptr, 0,
nullptr, 1, &barrier_from_present_to_clear );

    vkCmdClearColorImage( Vulkan.PresentQueueCmdBuffers[i], swap_chain_images[i],
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, &clear_color, 1, &image_subresource_range );

    vkCmdPipelineBarrier( Vulkan.PresentQueueCmdBuffers[i],
VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, 0, 0, nullptr, 0,
nullptr, 1, &barrier_from_clear_to_present );
    if( vkEndCommandBuffer( Vulkan.PresentQueueCmdBuffers[i] ) != VK_SUCCESS ) {
      printf( "Could not record command buffers!\n" );
      return false;
    }
  }

  return true;
```

30. Tutorial02.cpp, function RecordCommandBuffers()

This code is placed inside a loop. We are recording a command buffer for each swap chain image. That's why we needed a number of images. Image handles are also needed here. We need to specify them for image memory barriers and during image clearing. But recall that I said we can't use swap chain images until we are allowed to, until we acquire the image from the swap chain. That's true, but we aren't using them here. We are only preparing commands. The usage itself is performed when we submit operations (a command buffer) to the queue for execution. Here we are just telling

Vulkan that in the future, take this picture and do this with it, then that, and after that something more. This way we can prepare as much work as we can before we start the main rendering loop and we avoid switches, ifs, jumps, and other branches during the real rendering. This scenario won't be so simple in real life, but I hope the example is clear.

In the above code above, we are first preparing two image memory barriers. Memory barriers are used to change three different things in the case of images. From the tutorial point of view, only the layouts are interesting right now but we need to properly set all fields. To set up a memory barrier we need to prepare a variable of type VkImageMemoryBarrier, which contains the following fields:

- sType – Structure type which here must be set to VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER.
- pNext – Leave it null, pointer not used right now.
- srcAccessMask – Types of memory operations done on the image before the barrier.
- dstAccessMask – Types of memory operations that will take place after the barrier.
- oldLayout – Layout from which we are transitioning; it should always be equal to the current layout (which in this example, for a first barrier, would be VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR).Or we can use an undefined layout, which will let the driver perform some optimizations but the contents of the image may be discarded. Since we don't need the contents, we can use an undefined layout here.
- newLayout – A layout that is compatible with operations we will be performing after the barrier; we want to do image clears; to do that we need to specify VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL layout. We should always use a specific, dedicated layout.
- srcQueueFamilyIndex – A queue family index that was referencing the image previously.
- dstQueueFamilyIndex – A family index from which queues will be referencing images after the barrier (this refers to the swap chain sharing mode I was describing earlier).
- image – handle to image itself.
- subresourceRange – A structure describing parts of an image we want to perform transitions on; this is that last variable from the previous code example.

Some notes are necessary regarding access masks and family indices. In this example before the first barrier and after the second barrier only the presentation engine has access to the image. The presentation engine only reads from the image (it doesn't modify it) so we set srcAccessMask in the first barrier and dstAccessMask in the second barrier to VK_ACCESS_MEMORY_READ_BIT. This indicates that the memory associated with the image is read-only (image contents are not modified before the first barrier and after the second barrier). In our command buffer we will only clear an image. This operation belongs to the so-called "transfer" operations. That is why I've set the value of VK_ACCESS_TRANSFER_WRITE_BIT in the first barrier in dstAccessMask field and in the srcAccessMask field of the second barrier.

I won't go into more detail about queue family indices, but if a queue used for graphics operations and presentation are the same, srcQueueFamilyIndex and dstQueueFamilyIndex will be equal, and the hardware won't make any modifications regarding image access from the queues. But remember that we have specified that only one queue at a time will access/use the image. So if these queues are different, we inform the hardware here about the "ownership" change, that different queue will now access the image. And this is all the information you need right now to properly set up barriers.

We need to create two barriers: one that changes the layout from the "present source" (or undefined) to "transfer dst". This barrier is used at the beginning of a command buffer, when the previously presentation engine used an image and now we want to use it and modify it. The second barrier is used to change the layout back into the "present source" when we are done using the images and we can give them back to a swap chain. This barrier is set at the end of a command buffer.

Now we are ready to start recording our commands by calling the **vkBeginCommandBuffer()** function. We provide a handle to a command buffer and an address of a variable of type VkCommandBufferBeginInfo and we are ready to go. Next we set up a barrier to change the image layout. We call the **vkCmdPipelineBarrier()** function, which takes quite a

few parameters but in this example the only relevant ones are the first—the command buffer handle—and the last two: number of elements (barriers) of an array and a pointer to first element of an array containing the addresses of variables of type VkImageMemoryBarrier. Elements of this array describe images, their parts, and the types of transitions that should occur. After the barrier we can safely perform any operations on the swap chain image that are compatible with the layout we have transitioned images to. The general layout is compatible with all operations but with a (probably) reduced performance.

In the example we are only clearing images so we call the **vkCmdClearColorImage()** function. It takes a handle to a command buffer, handle to an image, current layout of an image, pointer to a variable with clear color value, number of subresources (number of elements in the array from the last parameter), and an array of pointers to variables of type VkImageSubresourceRange. Elements in the last array specify what parts of the image we want to clear (we don't have to clear all mipmaps or array levels of an image if we don't want to).

And at the end of our recording session we set up another barrier that transitions the image layout back to a "present source" layout. It is the only layout that is compatible with the present operations performed by the presentation engine.

Now we can call the **vkEndCommandBuffer()** function to inform that we have ended recording a command buffer. If something went wrong during recording we will be informed about it through the value returned by this function. If there were errors, we cannot use the command buffer, and we'll need to record it once again. If everything is fine we can use the command buffer later to tell our device to perform operations stored in it just by submitting the buffer to a queue.

Tutorial 2 Execution

In this example, if everything went fine, we should see a window with a light-orange color displayed inside it. The contents of a window should look similar to this:

Cleaning Up

Now you know how to create a swap chain, display images in a window and perform simple operations that are executed on a device. We have created command buffers, recorded them, and presented on the screen. Before we close the application, we need to clean up the resources we were using. In this tutorial I have divided cleaning into two functions. The first function clears (destroys) only those resources that should be recreated when the swap chain is recreated (that is, after the size of an application's window has changed).

```
if( Vulkan.Device != VK_NULL_HANDLE ) {
```

```
    vkDeviceWaitIdle( Vulkan.Device );

    if( (Vulkan.PresentQueueCmdBuffers.size() > 0) && (Vulkan.PresentQueueCmdBuffers[0]
!= VK_NULL_HANDLE) ) {
      vkFreeCommandBuffers( Vulkan.Device, Vulkan.PresentQueueCmdPool,
static_cast<uint32_t>(Vulkan.PresentQueueCmdBuffers.size()),
&Vulkan.PresentQueueCmdBuffers[0] );
      Vulkan.PresentQueueCmdBuffers.clear();
    }

    if( Vulkan.PresentQueueCmdPool != VK_NULL_HANDLE ) {
      vkDestroyCommandPool( Vulkan.Device, Vulkan.PresentQueueCmdPool, nullptr );
      Vulkan.PresentQueueCmdPool = VK_NULL_HANDLE;
    }
  }
```

31. Tutorial02.cpp, Clear()

First we must be sure that no operations are executed on the device's queues (we can't destroy a resource that is used by the currently processed commands). We can check it by calling **vkDeviceWaitIdle()** function. It will block until all operations are finished.

Next we free all the allocated command buffers. In fact this operation is not necessary here. Destroying a command pool implicitly frees all command buffers allocated from a given pool. But I want to show you how to explicitly free command buffers. Next we destroy the command pool itself.

Here is the code that is responsible for destroying all of the resources created in this lesson:

```
  Clear();

  if( Vulkan.Device != VK_NULL_HANDLE ) {
    vkDeviceWaitIdle( Vulkan.Device );

    if( Vulkan.ImageAvailableSemaphore != VK_NULL_HANDLE ) {
      vkDestroySemaphore( Vulkan.Device, Vulkan.ImageAvailableSemaphore, nullptr );
    }
    if( Vulkan.RenderingFinishedSemaphore != VK_NULL_HANDLE ) {
      vkDestroySemaphore( Vulkan.Device, Vulkan.RenderingFinishedSemaphore, nullptr );
    }
    if( Vulkan.SwapChain != VK_NULL_HANDLE ) {
      vkDestroySwapchainKHR( Vulkan.Device, Vulkan.SwapChain, nullptr );
    }
    vkDestroyDevice( Vulkan.Device, nullptr );
  }

  if( Vulkan.PresentationSurface != VK_NULL_HANDLE ) {
    vkDestroySurfaceKHR( Vulkan.Instance, Vulkan.PresentationSurface, nullptr );
  }

  if( Vulkan.Instance != VK_NULL_HANDLE ) {
    vkDestroyInstance( Vulkan.Instance, nullptr );
  }

  if( VulkanLibrary ) {
  #if defined(VK_USE_PLATFORM_WIN32_KHR)
    FreeLibrary( VulkanLibrary );
  #elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)
    dlclose( VulkanLibrary );
  #endif
  }
```

32. Tutorial02.cpp, destructor

First we destroy the semaphores (remember they cannot be destroyed when they are in use, that is, when a queue is waiting on a given semaphore). After that we destroy a swap chain. Images that were created along with it are automatically destroyed, and we don't need to do it by ourselves (we are even not allowed to). Next the device is destroyed. We also need to destroy the surface that represents our application's window. At the end, the Vulkan instance destruction takes place and the graphics driver's dynamic library is unloaded. Before we perform each step we also check whether a given resource was properly created. We can't destroy resources that weren't properly created.

Conclusion

In this tutorial you learned how to display on a screen anything that was created with Vulkan API. To brief review the steps: First we enabled the proper instance level extensions. Next we created an application window's Vulkan representation called a surface. Then we chose a device with a queue family that supported presentation and created a logical device (don't forget about enabling device-level extensions!)

After that we created a swap chain. To do that we first acquired a set of parameters describing our surface and then chose values for proper swap chain creation. Those values had to fit into a surface's supported constraints.

To draw something on the screen we learned how to create and record command buffers, which also included image's layout transitions for which image memory barriers (pipeline barriers) were used. We cleared images so we could see the selected color being displayed on screen.

And we also learned how to present a given image on the screen, which included acquiring an image, submitting a command buffer, and the presentation process itself.