Vanilla Engine Developer Manual

Table of Contents

1. Introduction

Welcome to the Vanilla Engine Developer Manual. This documentation aims to make you fully understand the inner workings of a modern graphical rendering engine — specifically one specialised for the development of a 2D game.

Think of this manual as a "System Bible". In here, you should find 3 answers for every component in the codebase:

- **How** does the engine work?
- **What** makes it work that way?
- **Why** was designed it that way?

Keep in mind that, despite its niche use-case and apparent lack of features when compared to other mainstream game engines, Vanilla is anything *but* a simple program. Building a game is the main idea behind the programming classes; understanding Vanilla's architecture is a bonus—though something that would surely improve your ability to understand code and complex systems programming.

1.1. Why C?

Vanilla is being developed specifically to provide a easy-to-use enviroment for game development in C. Most game engines are designed around a paradigm known as "Object-Oriented Programming"—also known as OOP—which has become very popular since the introduction of programming languages such as Java, C++ and C#.

While OOP does lower the knowledge barrier needed to develop a functional game, it isn't quite as interesting from a learning perspective. OOP is easier because it hides away a lot of the details that are needed to make a computer program run; but that same reason is exactly why it isn't quite as good for learning. You will learn the high-level concepts, but you will likely develop bad practices since you don't deeply understand what it is you're asking your computer to do.

In summary: It is easier to write bad OOP code, because you don't need to know as much to write valid OOP code ("valid" here meaning "it will compile without errors"). It is more dificult to write valid C code—but when you do, you're more likely to understand the details of *why* your code is valid.

1.2. Why Vulkan?

To answer this question, first we must understand what Vulkan *is*.

Vulkan is, in technical terms, a **low-overhead, cross-platform graphical API**. Now let's break that word soup down:

- **API**: the acronym stands for "Application Programming Interface". A "Programming Interface" is, in essence, a standardised language used by an application—hence the name. An API is a standardised way that your computer talks to a component—in this case, your graphics card.
- **Graphical API**: If the communication between your CPU and GPU wasn't standardised, you would have to program a version of your game for every single graphics card model out

on the market. That's why the Khronos Group created OpenGL (and Vulkan much later). They defined the rules for a standard language, and then told the GPU manufacturers "here, make your GPU speak this language." That way, we only need to learn how to use one API; It is the manufacturer's responsibility to make the graphics card speak our language. This language is what Vulkan is.

- **Cross-Platform**: This one is pretty self-explanatory; it means Vulkan works in any platform, assuming its components can "speak" Vulkan—which is basically everything these days.
- **Low-Overhead**: Here's where the issue of using Vulkan lives. Most software you use tries to be "helpful": if you make a mistake, it either fixes it or steers you in the right direction. The issue is, checking for your mistakes costs precious CPU cycles. Every time you want to render something on the screen, the CPU must send that information out to the GPU, which takes *ages* in computer-time. Seriously: If 1 CPU clock "tick" lasted 1 second, it would take around 2 and a half hours for your information to get there on the average modern computer. To add error checking, you'd need to delay the travel of information further to validate the commands—and this extra processing is called an "overhead". Therefore "low-overhead" means exactly what it says on the tin: it is blazing-fast, but it will do *exactly* you what you tell it to—just like C!

While OpenGL still works just fine, it is being deprecated, as it was created in 1992 and received its last update in 2017. Vulkan was created by the same group (Khronos) as a sucessor to OpenGL, but with modern computers in mind. It is much faster, much more feature rich, and far, far more complex to use. But don't worry—you'll likely never have to interact with it yourself throughout this project. In fact, this takes us to our next question:

1.3. What is Vanilla?

This game engine is exactly the reason you won't need to touch Vulkan, and most of the reason game engines exist in the first place. Vanilla is a set of functions and structures that will allow you to focus on the game's logic, not on the complex handling of data between your game and the components on the computer.

For example, let's write some example code to render a single sprite with texture on the screen, but using Vulkan directly (simplified so it fits in less than 5 pages):

```
// First, we wait for the previous frame to finish rendering.
vkWaitForFences(device, 1,
                &in_flight_fences[current_frame],
                VK_TRUE, UINT64_MAX);

// Then, we start the next frame
uint32_t image_index;
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX,
                      image_available_semaphore,
                      VK_NULL_HANDLE, &image_index);


// (Continues on the next page...)
```

```
// Now we unlock the frame so we can draw on it (this lock is called a "fence")
vkResetFences(device, 1,
              &in_flight_fences[current_frame]);

// Only now we listen for the commands needed to render the image
vkResetCommandBuffer(command_buffers[current_frame], 0);
VkCommandBufferBeginInfo begin_info = {
  .sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
  .flags = 0
};

if (vkBeginCommandBuffer(command_buffer[current_frame],
                         &begin_info) == VK_SUCCESS) {
    // Now we define where to draw things and how to clear the screen
    VkRenderPassBeginInfo render_pass_info = {
      .sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO,
      .renderPass = render_pass,
      .framebuffer = swapchain_framebuffers[image_index],
      .renderArea.offset = {0, 0},
      .renderArea.extent = swapchain_extent,
      .clearValueCount = 1,
      .pClearValues = &clear_colour
    };

  // Now we include the instructions to the GPU of how to render what we want
(these are called a "rendering pipeline")
  vkCmdBindPipeline(command_buffers[current_frame],
                    VK_PIPELINE_BIND_POINT_GRAPHICS,
                    graphics_pipeline);

  // Now we send the texture and other data to the GPU
  memcpy(uniform_buffer_mapped[current_frame], &ubi, sizeof(ubo));
  vkCmdBindDescriptorSets(command_buffers[current_frame],
                          VK-PIPELINE_BIND_POINT_GRAPHICS,
                          pipeline_layout, 0, 1,
                          &descriptor_sets[current_frame],
                          0, NULL);

  // Now we send to the GPU the vertices of the object to be drawn, alongide
texture coordinates and any other data we may have
  VkBuffer vertex_buffers[] = {vertex_buffer};
  VkDeviceSize offsets[] = {0};
  vkCmdBindVertexBuffers(command_buffers[current_frame],
                         0, 1, vertex_buffers, offsets);
  vkCmdDraw(command_buffers[current_frame], 6, 1, 0, 0); // drawing 2 triangles
(6 vertices) into 1 square.
  vkCmdEndRenderPass(command_buffers[current_frame]);
  vkEndCommandBuffer(command_buffers[current_frame]);
}

// Now we submit our commands to the graphics queue
VkSubmitInfo submit_into = {
```

```
  .sType = VK_STRUCTURE_SUBMIT_INFO,
  .waitSemaphoreCount = 1,
  .pWaitSemaphores = &image_available_semaphores[current_frame],
  .pWaitDstStageMask = (VkPipelineStageFlags[])
{VK_PPIPELINE_STAGE_COLOR_ATTATCHMENT_OUTPUT_BIT},
  .commandBufferCount = 1,
  .pCommandBuffers = &command_buffers[current_frame],
  .signalSemaphoreCount = 1,
  .pSignalSemaphores = &render_finished_semaphores[current_frame]
};

vkQueueSubmit(graphics_queue, 1, &submit_info,
              &in_flight_fences[current_frame]);

// And finally, we present the finished frame to the screen
VKPresentInfoKHR present_info = {
  .sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,
  .waitSemaphoreCOunt = 1,
  .pWaitSemaphores = &render_finished_semaphores[current_frame],
  .swapchainCount = 1,
  .pSwapchains = &swapchain,
  .pImageIndices = &image_index
};

vkQueuePresentKHR(present_queue, &present_info);
current_frame = (current_frame + 1) % MAX_FRAMES_IN_FLIGHT;
```

And, mind you, this is just to get a single static image on the screen; you can't interact with it in any way yet.

Now, let's see how you'd do the same thing using Vanilla:

```
// First, let's define a player object
Vnl_Sprite player = {
  .texture  = player_tex,
  .position = {100.0f, 250.0f},
  .scale    = {1.0f, 1.0f},
  .rotation = 0.0f
};

// And now we tell Vanilla to draw it
while (vnl_game_open()) {
  vnl_begin_frame();

  // Let's make our player move right when we press 'D'
  if (vnl_key_is_down(VNL_KEY_D)) {
    player.position.x += 5.0f;
  }


  // (Continues on the next page...)
```

```
  // Now, we render the sprite to the screen
  vnl_draw(&player);
  vnl_end_frame();
}
```

And we're done. Yes, Vanilla uses Vulkan—but on the back-end. You'll never have to see it if you don't look on its insides. That is the purpose of a game engine (and, more generally, of systems programming): to take some of the technical complexity away from the code.

Now that we've cleared what the idea behind Vanilla is, let's take a look at how it works.

2. The Achitecture

2.1. Game Lifecycle

2.2. Abstraction Layers

2.3. Handling Memory

3. The Developer's Interface

3.1. Quickstart

3.1.1. Adding Vanilla Into Your Project

3.1.2. Configuring Vanilla

Vanilla needs to know some basic information about your game before starting up. This information should be provided to the engine in the form of a configuration struct of type `VnlConfig`. You can see its definition in Section 4.1.

The user must create and provide a VnlConfig struct to initialise the engine. To speed up configuration, you can use the engine's default config values (`VNL_DEFAULT_CONFIG`), and then change the ones you need to:

```
VnlConfig config = VNL_DEFAULT_CONFIG // defined in vnl_macros.h
config.title = "My New Cool Game";
config.window.width = 1920;
config.window.height = 720;
```

3.2. Common Game Structures/Objects

3.3. Using Vanilla Data Structures

3.4. Resource Management

3.5. Input & Timing

4. Vanilla Struct Reference

4.1. VnlConfig

Vanilla's configuration struct is defined as:

```
typedef struct VnlConfig {
  VnlWindow window;
  VnlGameVersion version;
  const char* title;
} VnlConfig;
```

See also: VnlWindow and VnlGameVersion in Sections 4.2 and 4.3

- `window`: Through it, you may control the game window's size.
- `version`: Use this to define your game's version.
- `title`: This is the title of your game's window.

4.2. VnlWindow

4.3. VnlGameVersion

5. Vanilla Function Reference

6. The Backend Implementation

6.1. Fixed Function Pipeline

6.2. Synchronization

6.3. Vertex Layout