1. Create vulkan instance,
   Select API extensions
   Select hardware (physical device)
2. Create logical device (???)
3. Create window
   Window surface
   Swap chain (double buffering)
4. Receive image from swap chain
   Wrap to image view (use specific part of an image)
   Wrap to frame buffer (used for color, depth and stencil targets)
5. Render passes
   What type of images to use during rendering
6. Graphics pipeline
   Describe configurable state of graphics card
   To change certex layout or shaders need new pipeline
7. Command pools and command buffers
8. Main loop
   Acquire image from swap chain
   Select command buffer
   Execute command buffer
   Return image to swap chain for presentation

- Create a VkInstance
- Select a supported graphics card (VkPhysicalDevice)
- Create a VkDevice and VkQueue for drawing and presentation
- Create a window, window surface and swap chain
- Wrap the swap chain images into VkImageView
- Create a render pass that specifies the render targets and usage
- Create framebuffers for the render pass
- Set up the graphics pipeline
- Allocate and record a command buffer with the draw commands for every possible swap chain image
- Draw frames by acquiring images, submitting the right draw command buffer and returning the images back to the swap chain

Check out
RAII, initializer lists

C:/"Program Files (x86)"/"Microsoft SDKs"/Windows/v10.0A/bin/"NETFX 4.7.2 Tools"/ResGen.exe
http://forums.codeblocks.org/index.php?topic=11128.0

shader compile auto
for /r %%i in (*.frag, *.vert) do %VULKAN_SDK%/Bin/glslangValidator.exe -V %%i

`VkInstance` - connection between program and Vulkan library

`vkInstanceCreateInfo` - create application
* vkApplicationInfo
* glfwGetRequiredInstanceExtensions
* validation layers
`vkPhysicalDevice` - select physical device
* find suitable device
* find queue families
`vkDevice` - create logical device

# General

Vulkan tries to remove guessing for driver.

# Structures

command buffer - buffer where you record commands ( `VkCommandBuffer` )
command pool - allocates command buffers, lightweight synchronization

render pass - provide information upfornt
descriptor layout - tell how to use resources, describes what shader can access

queue family - list of queues with the same functionality
queue - abstract mechanism to submit commands to GPU

swapchain - list of image buffers
imageView - wrapper around image, adds extra resources to image that describes how to use it

render pass - A render pass describes the scope of a rendering operation by specifying the collection of attachments, subpasses, and dependencies used during the rendering operation. A render pass consists of at least one subpass.
render pass is the set of attachments, the way they are used, and the rendering work that is performed using them

renderpass attachment - An attachment corresponds to a single Vulkan VkImageView. A description of the attachment is provided to the render pass creation, which allows the render pass to be configured

appropriately; the actual images to be used are provided when the render pass is used, via the VkFrameBuffer. It is possible to associate multiple attachments with a render pass;
More commonly, a color framebuffer and a depth buffer are separate attachments in Vulkan. Therefore the pAttachments member of VkRenderPassCreateInfo points to an array of attachmentCount elements.
A render pass represents a collection of attachments, subpasses, and dependencies between the subpasses, and describes how the attachments are used over the course of the subpasses. The use of a render pass in a command buffer is a render pass instance.

renderpass attachments are bound wy wrapping them into `frameBuffer`. buffer references all `imageViews`

## Depth buffer

Create the depth buffer image object
Allocate the depth buffer device memory
Bind the memory to the image object
Create the depth buffer image view

| this | is | table |
|---|---|---|
| 1 | 2 | 3 |
| `hola` | | |

a new line
another one