Simpler Vulkan version 2

Background and documentation (draft)

by Ingemar Ragnemalm 2026

Introduction

2016, the Vulkan framework was launched as a more modern API for graphics programming, promoting better performance and flexibility than OpenGL. The API is very verbose, with detailed settings needed for every step. Using that as is for course work was entirely unfeasable, but a few years ago, David Berntsson took on my bachelor thesis proposal to create a simplified API on top of Vulkan, called *Simpler Vulkan*. This was launched as an alternative to OpenGL in 2025.

This was a huge improvement, but I could also see that it was still clearly harder to use than OpenGL, and as expected the simplicity comes for a cost in features. I have worked through the resulting labs and I could see further possibilities.

Spring 2026, I developed these new additions, resulting in more simplifications for labs for TSBK07/TSBK11 labs, closer to the original OpenGL labs, plus additions for making the API feasible for the advanced course TSBK03. This is what I call version 2.

Installing SVK on Linux

As of 2026, Vulkan repositories are gone and we must rely on downloading a ”tarball” with many more steps. This what I did. It may not be perfect or optimal (or even complete, see vulkan.lunarg.com) but it worked for me.

Go to

https://vulkan.lunarg.com/sdk/home#linux

and download the version you want.

Go to

https://vulkan.lunarg.com/doc/view/latest/linux/getting_started.html#use-the-vulkan-sdk

sudo apt update
sudo apt upgrade
sudo apt install xz-utils

and then scroll down to "Install the SDK"

You will now do these steps:
Create the "vulkan" directory in your home directory

Generate the sha256sum:
copy in sha256sum $HOME/Downloads/vulkansdk-linux-x86_64 and hit tab to get your version

Extract the package
copy in tar xf $HOME/Downloads/vulkansdk-linux-x86_64 and hit tab again

Install more stuff:
sudo apt install libxcb-xinput0 libxcb-xinerama0 libxcb-cursor-dev

This didn't seem to do anything, but I ran it anyway:
source ~/vulkan/1.x.yy.z/setup-env.sh
which means
source ~/vulkan/1
hit tab, then s, then tab

Finally, I also installed
libsdl2-dev
libvulkan-dev

and then things runs just fine!

This is only for Linux. The system should be possible to use from Windows and Mac but we have not made makefiles for those.

Changes from version 1

As of summer 2026, many changes and additions are in beta stages, so you will find weaknesses and omissions. How much of what I did help, or am I abstracting entirely too much? Remember, my goal is *not* to teach you Vulkan as such, but to make a tool that uses Vulkan to teach graphics as a subject.

VectorUtils has been rewritten to remove some unnecessary additions (intentionally left out in the old VectorUtils as something the students are supposed to add as needed) as well as restore some calls that conform to GLSL and others.

A new pipeline builder now includes a parser for GLSL code to find sizes and bindings, reducing the sources for errors and simplifying the code. Layouts and descriptor sets are now hidden.

New calls to the loader include the old LittleOBJLoader calls LoadModel, DrawModel, LoadModelData and LoadDataToModel, restoring the Model concept as a limited holder for geometry which is intended for the laborants to build their own containers around. It allows the labs to be more streamlined, especially TSBK07/11 lab 1.

Texture loading is now external from the new Model. They are installed into the pipeline by name. Similarily, uniforms are installed by name.

New calls were added in order to create floating-point textures, inline textures, enabling transparency ETC.

A number of example programs have been written, some based on old OpenGL demos.

New labs

My labs are now closer to the OpenGL labs, not least by using the shader parser but also by using the simpler model loader, using LoadModelData to multiple buffers the same way that earlier parts of the lab does.

In TSBK03, Lab 1a needs multi-pass rendering, which is at this time solved by copying to texture. Lab 1b needs additional buffers. Lab 2 needs to update buffers on the fly. Lab 3 needed to switch textures on the fly and required a bug fix for texture loading.

General usage

In this section I make an overview on how to use SVK with the version 2 additions.

Initialization

You always start with
`svk.init();`
optionally preceded with initialization options like window title or window size.

You install callbacks. You always need to tell where you put the drawing code:
`svk.bindDrawingFunction(display);`
There are also callbacks for other events like mousedowns and keydowns.

You also load geometry and textures in the initialization part (usually in a separate init function but that is just a convention).

You create pipeline(s), once per set of shaders. For our purposes, the call
`svkBuildPipeline()`
loads the shaders. It parses the shaders in order to initialize the pipeline with informations from the shaders, but that is hidden from you.

Geometry is loaded to the Model struct using
`LoadModel`
which loads a model from disc, or
`LoadDataToModel.`
Which loads from inline data.

Textures are loaded from disc with
`svk.loadTGATexture`
and then installed in pipelines using
`svkInstallTexture.`

Other uniforms (matrices, etc) need to be packed into a struct. You inform the pipeline about that struct using
`svkInstallUniform.`

Drawing

Drawing always starts with
`svk.beginRendering(cmd, true);`
and ends with
`vkCmdEndRendering(cmd);`

Depending on what you do, there may be just a single beginRendering/vkCmdEndRendering pair, but there can also be several.

Now let us consider what to put between the begin and end calls. You bind the pipeline you want using
`svkBindPipeline`
which binds the pipeline as well as the data you have installed for it.
It is really only two calls, vkCmdBindPipeline() and svkBindPipelineData().

You switch between pipelines in order to change shader.
You may also want to call
`svkBindPipelineData()`
again since it enables the textures that you loaded before. It is vital if you change textures during a drawing pass.

You often want to change shader data, most notably transformation matrices (player movement and other animations) and time, but there may be others. We return to animations below.

You draw models with DrawModel(). DrawModel() exists in several versions, but I think you usually want the minimal
`DrawModel(myModel, cmd);`

which assumes that your attributes for position, normals and texture coordinates have locations 0, 1, 2.

Uniforms and data alignment

In OpenGL, uniforms are global within a model, a drawing call. In Vulkan, uniforms are global *within a render pass*.

In the shader, globals must be declared in a struct. At the host, this is not strictly necessary for a single variable but should be used if you have several uniform variables.

An important issue for uniforms is the question of *data alignment*. Vulkan enforces a data alignment that can be different from the host. Generally speaking, Vulkan loves power of 2 sizes, which means that a vec3 can take the space of four floats! For this reason, declaring a vec3 before a mat4 can cause strange errors.

The ”confused polygon” example shows how this works, and does not, with three variants. It displays a single triangle, with a purple color delivered in a uniform together with the rotation matrix. They are declared in a uniform struct, identical between CPU and shaders. We start like this:

```
struct uniforms
{
      vec3 color;
      mat4 theMatrix;
} uniforms;
```

and similarily in the shaders:

```
layout(binding = 2) uniform uniforms1
{
      vec3 color;
      mat4 theMatrix;
} uniforms;
```

The uniform struct is uploaded and installed in the pipiline data as:

```
        svk.uploadToUniformBuffer(uniformsBuffer, &uniforms,
sizeof(uniforms)); //
         svkInstallUniform(pld, "uniforms1", uniformsBuffer);
```

This code looks fine, compiles, but produces a blank screen! The reason is that data are misaligned. If I change the order like this:

```
struct uniforms
{
      mat4 theMatrix;
      vec3 color;
} uniforms;
```

it works, but causes validation warnings since the data and the buffer size don’t match!
But if I change the vec3 to a vec4

```
struct uniforms
{
      mat4 theMatrix;
      vec4 color;
} uniforms;
```

Then all is well! Simple conclusion: mat4 and vec4 are happy in uniforms, vec3 and scalars are not. The less simple solution is to make the CPU align like Vulkan does.

Push constants

Push constants is a kind of uniforms that are recommended for data that changes often, especially more than once per render pass. Note that ordinary uniforms can *not* change within a render pass! However, the size is limited. On some GPUs, the push constant memory can be as small as a single mat4! On the positive side, you don't have to allocate buffers as with uniforms. The shader parser calculates the size and initializes the pipeline with that size. Let's say we want to update a matrix m which goes to the vertex shader. It can look like this:

Declare it in the shader:

```
layout(push_constant) uniform constants {mat4 m; };
```

In your display function, do this:

```
mat4 m = Rz(a);
vkCmdPushConstants(cmd, pld.pipelineLayout, VK_SHADER_STAGE_VERTEX_BIT, 0,
sizeof(m), &m);
```

It gets more complicated with several push constants. It is possible to update only some variables, but then you must specify exactly where in the push constant layout it is located, by an offset:

A somewhat more elaborate example is found in the psychedelic teapot example. It includes a matrix and one float, the time.

Vertex, the matrix:

```
layout(push_constant) uniform constants
{
	mat4 modelToWorldMatrix;
};
```

Fragment. Here we only see the time. Notice the offset.

```
layout(push_constant) uniform constants
{
	layout(offset = 64) float time; // Horrible syntax! sizeof(mat4)
};
```

On the CPU, the data is updated like this:

```
		vkCmdPushConstants(cmd, pld.pipelineLayout, VK_SHADER_STAGE_VERTEX_BIT,
0, sizeof(mat4), (total).m);
		vkCmdPushConstants(cmd, pld.pipelineLayout,
VK_SHADER_STAGE_FRAGMENT_BIT, sizeof(mat4), sizeof(time), &time);
```

Animation

Animation is a surprisingly complex topic. To move things round in the scene, you have several options, and some that will not work.

If all you want is a single object moving, you can store its position in a uniform and change it once per frame. However, *uniforms can not change within a render pass*! Push constants, however, can be changed any time. This leaves you with the following options:

1. Push constants for matrices or positions.
2. Uniforms that change between render passes.
3. Instancing with a position array.
4. Indexing with a position array, with the index given as push constants.
5. Time-based animations, where the time is applied to movement or textures, can store the time as a uniform.

As you can see, push constants is the recommended option. However, the small maximum size of push constants is sometimes a problem.

EXAMPLES

The y axis and the column-wise matrices

An irritating issue with Vulkan is that the Y axis goes down, not up. Because of this, my vertex shaders tend to end with this:

```
	// Vulkan Y axis fix. Always put this at end until the global fix can be used.
	gl_Position.y = -gl_Position.y;
```

To make matters worse, this gets harder when you do multi-pass rendering. Generally speaking, you may only want to do this at the last pass.

It is notable that Vulkan has a setting for using the Y upwards, but as fas as I can tell, this only works above some (to me not known) GPU generation. Anyway, I have failed to use it. Feel free to use it.

Another issue is that matrices are stored column-wise, which makes them impractical to express as arrrays. This was actually the case in OpenGL as well, but OpenGL has options to auto-transpose so we usually didn't notice. For Simpler Vulkan, VectorUtils will transpose the matrices for you so you usually don't notice.

Textures

A texture is basically just an image, a VkImage, same type as the frame buffer, but it is handled in some interesting ways. The VkImage is generally packagaged into a custom struct called a AllocatedImage.

You load them like this:

```
svk.loadTGATexture(blackTex, "black.tga");
```

This loads it but says nothing of how it is supposed to be used, in what pipeline(s). You inform SVK pipeline about the variable as:

```
svkInstallTexture(shader, "tex", blackTex);
```

In the shader (usually the fragment shader) you declare it as:

```
layout(binding = 1) uniform sampler2D tex;
```

and use it by sampling the data with:

```
vec4 t = texture(tex, texCoord);
```

If you want to change between textures for the same variable, like drawing with the same shader but another texture, you must also add

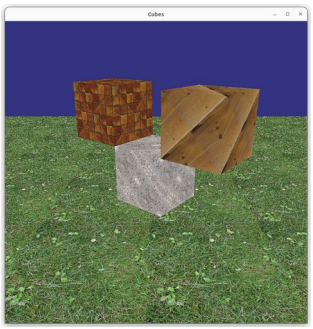
```
svkBindPipelineData(cmd, shader);
```

each time in order to update the pipeline information. The example "cubes" shows three cubes plus a ground plane with different textures. This code changes texture and location of one of the cubes:

```
	svkInstallTexture(pld, "tex", rock);
	svkBindPipelineData(cmd, pld);

		myMatrix = worldToView * T(0,0,0) * Ry(time) * T(0, 2, 0);
		vkCmdPushConstants(cmd, pld.pipelineLayout, VK_SHADER_STAGE_VERTEX_BIT, 0, sizeof(myMatrix), (myMatrix).m);
		DrawModel(cube, cmd);
```

Change texture, update, change the location by building a matrix, upload and draw.



Cubes, drawing a single shape with different locations and textures

SimplerVulkan currently only supports TGA files. These textures are loaded in 8888 format, but SimplerVulkan v2 introduces calls for creating floating-point textures.

Texture samplers

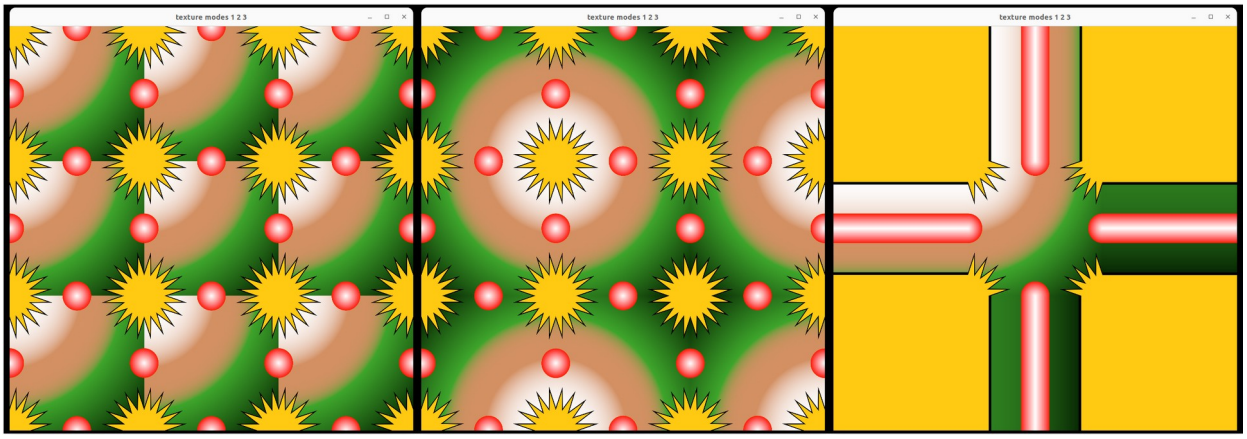
An important issue is to configure how the texture acccess is performed. The default is a repeated texture, nearest neighbor, but you can configure it by creating a *texture sampler*. Here are some examples from the ”texture modes” demo, using ”createTextureSampler”, one of the v2 additions:

```
      svk.createTextureSampler(sampler1, VK_FILTER_LINEAR, VK_FILTER_LINEAR,
VK_SAMPLER_ADDRESS_MODE_REPEAT);
      svk.createTextureSampler(sampler2, VK_FILTER_LINEAR, VK_FILTER_LINEAR,
VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT);
      svk.createTextureSampler(sampler3, VK_FILTER_LINEAR, VK_FILTER_LINEAR,
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE);
```

You use the samplers by inserting the selected sampler into the AllocatedImage texture like this:

```
tex.textureSampler = &sampler1;
```

The picture below show the result. Note that I have used a picture that is intended for mirrored repeat. This may not be what you want for all cases.



The ”texture modes” demo showing repeat, mirrored repeat and clamp to edge.

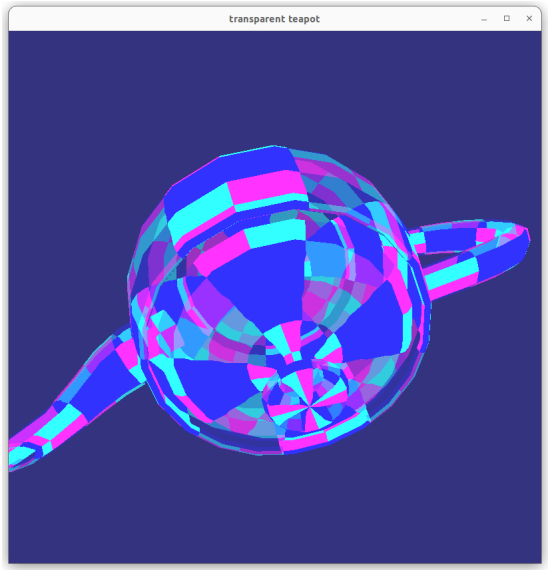
The code also includes the choice between linear or nearest neighbor sampling. A number of more modes are available. One other kind of sampling that is very desirable is mip mapping but that is not yet supported.

Transparency

Transparency is supported, but must be enabled *before* the pipeline using it is created, with the following call:

```
svk.setBlendingEnabled(true);
```

Then you can use textures with transparency, like TGA files with transparency, or PNG files (not supported by the package but feel free to add it). The demo ”transparent teapot” below (based on the OpenGL demo by the same name) uses an inline texture with varying alpha value. The demo also plays with back-face culling using the calls svk.cullFront();, svk.cullBack(); and svk.cullNone(); in order how transparency can be done right (back first) or wrong. So activating transparency is easy, doing it right is harder.



Transparent teapot

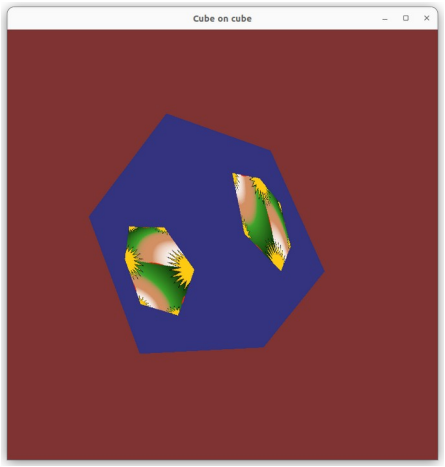
Render to texture and copying to texture

As of august 2026, I have not yet been able to render to texture. However, we can copy the framebuffer to a texture which is a viable alternative for the same effect.

Working with images in Vulkan is very much a matter of image formats, and converting between them. This is called an image *transition*. Images can be stored in formats intended for different uses.

SVK has relelevant helper functions since v1, transitionImage() and copyImageToImage(). Based on these, version 2 includes svk.copyFramebufferToTexture, which performs the needed transitions and copy.

The demo ”*cube on cube*” shows how this can be used.



Cube on cube

The ”cube on cube” demo renders the cube, then copies the framebuffer to a texture and renders again. This can be varied in many ways including applying filters and combining images.

Other events

The event loop will call a few other callbacks when some events occur. In case you want to modify it, the event loop is named run() and is located in svk.cpp around line 675. It handles mouse button pushes, mouse movements, keydowns, timing and window resizing.

In version 1, ESC changed the ”mouse mode”, and the pointer was by default hidden, but from version 1, a change is that ESC by default quits, unless you define your own keydown handling and then you decide.

svk.quit() tells Simpler Vulkan to quit the running program. With a custom keyboard event, this is no longer triggered by ESC you you should decide if you want to trigger that.

The calls svk.hideMouse() and svk.showMouse() can be used to get the old behavior if you prefer that.

Window resizing should affect your frustum to avoid stretched graphics. With resize events, you get the new size and can adapt the frustum accordingly. You can also call svk.getAspectRatio(). The demo ”proportional teapot” show how this can be done.

Data queries

Some data can be requested at any time. You can request the current time by
`svk.getTimeSinceStart(),`
which gives you the time in seconds as a double.
`double svk.getDelta()`
gets the time between frames.

You can request window size through SDL as SDL_GetWindowSize(_window, &w, &h); There is also an internal variable called _drawExtent which is not currently public. (Change that if you like.)
`void svk.getMouse(int *x, int *y)`
gets the current mouse position.
`float svk.getAspectRatio()`
gets the aspect ratio of the window. (Use that to adjust your frustum dimensions.)

You get the dimensions of an AllocatedImage as the VkExtent3D size, i.e. myImage.size.width and myImage.size.height.

Summary of important calls

Even after my additions, to simplify the system even more, there are still multiple important calls. We still need to intialize and load all our assets, textures, transformations, geometry, shaders. Here are a summary of the most vital calls:

```
svk.init();
svkBuildPipeline(pld, "phong.vert", "phong.frag");
svk.allocateVertexBuffer(buffer, bufferSize);
svk.uploadToGPUBuffer(buffer, bufferData, bufferSize);
teapot = LoadModel("teapot.obj");
svk.allocateUniformBuffer(projectionMatrixBuffer, sizeof(mat4));
svk.uploadToUniformBuffer(projectionMatrixBuffer, projectionMatrix.m,
sizeof(mat4));
svkInstallUniform(pld, "proj", projectionMatrixBuffer, sizeof(mat4));
svk.loadTGATexture(tex, "maskros512.tga");
svkInstallTexture(pld, "tex", tex);

svk.beginRendering(cmd, true);
vkCmdEndRendering(cmd);
vkCmdBindPipeline(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS, pld.pipeline);
DrawModel(myModel, cmd);
svkBindPipelineData(cmd, pld);
```

API documentation

The following section is the documentation of the parts of the API that students are expected to use. It can not be considered complete, but is focused on the parts that are essential.

In the SVK class:

`void clearDepth(VkCommandBuffer cmd);`
This call clears the depth buffer. It works but causes warnings if the image formats are not just right. WILL BE REVISED.

`void repeatingTimer(double time);`
Sets the minimal time between each frame.

`void copyFramebufferToTexture(VkCommandBuffer cmd, AllocatedImage texture);`
Based on copyImageToImage in svk_images, this copies explicitly from the frame buffer (_drawImage) to an image (typically a texture). This should not be used inside drawing passes (bracketed by svk.beginRendering() and vkCmdEndRendering()).

```
void hideMouse();
void showMouse();
```
They do what they say.

`void run();`
Starts the event loop. This will make the display function get called repeatedly.

`void quit();`
Signals the event loop to quit.

`void beginRendering()`
This is a simple wrapper of the two calls svk.getRenderingInfo(); and vkCmdBeginRendering()

```
transparency
turning off or erasing the Z buffer
```

In svk_images:

`void transitionImage(VkCommandBuffer cmd, VkImage image, VkImageLayout currentLayout, VkImageLayout newLayout);`
This call recodes a VkImage to a different format/usage.

`void copyImageToImage(VkCommandBuffer cmd, VkImage source, VkImage destination, VkExtent2D srcSize, VkExtent2D dstSize);`
This call copies one VkImage to another. You need to use transitionImage to make them compatible. It should not be called within a beginRendering/vkCmdEndRendering pair.

In svk_loader:

svk_loader is a rather big unit which handles both building the pipeline, parsing the shaders, loading OBJ files, and more.

`void svkBuildPipeline(PipelineData &p, const char *vertexShader, const char *fragmentShader);`
This is my smarter pipeline builder, based on parsing the shaders. For the programmer, it just looks like it loads the shaders, but it does more.

`void svkBindPipelineData(VkCommandBuffer cmd, PipelineData &p);`

Select the desired pipeline (shaders) for drawing.

```
void svkValidator(PipelineData &p);
void printPipelineData(PipelineData p);
```
These calls do printouts for debugging.

```
void svkInstallUniform(PipelineData &p, const char *name, AllocatedBuffer
buffer, int bufferSize);
void svkInstallTexture(PipelineData &p, const char *name, AllocatedImage
texture);
```
Assign buffers and images to uniforms in the shaders.

Lower-level attribute calls:

```
void svkAllocateAndUploadVertexBuffer(AllocatedBuffer &buffer, float
*bufferData, int bufferSize);
void svkAllocateAndUploadIndexBuffer(AllocatedBuffer &buffer, uint32_t
*bufferData, int bufferSize);
```
These are simple calls reducing 2 lines to 1.

```
void svkBindVertexBuffer(PipelineData pld, VkCommandBuffer cmd, AllocatedBuffer
&buffer, const char *name);
```
Binds a vertex buffer by name. Unnecessary if you use LoadModel.

OBJ model loading and drawing in svk_loader:

```
Model *LoadModel(const char *filename);
```
Loads a model in OBJ format from disc

```
Model *LoadDataToModel(vec3 *vertices, vec3 *normals, vec2 *texCoords, uint
*indices, int numVert, int numIndices);
```
Loads a model from your own buffers.

```
void DrawModel(Model *model, VkCommandBuffer cmd, int b0, int b1, int b2);
```
Draws the model using explicit location numbers. It is up to you to make sure that they match the shader.

```
void DrawModel(Model *model, VkCommandBuffer cmd);
```
Draws the model assuming that you are using the locations 0, 1, 2.

```
void DrawModel(Model *model, VkCommandBuffer cmd, PipelineData pld, const char
*vertex, const char *normal, const char *texCoord);
```
Draws the model using the provided attribute names. This is not fast but it is pretty easy.

Optimizations

Several optimizations are initially left out. That was for simplicity, to keep the number of concepts and problems down while we are only trying to make it work, and I do not want to keep these from you once you are ready to optimize. It can be irrelevant during the course, but if you work professionally with Vulkan, you should know about them, and of course you are encouraged to try them and see if you can see any difference. Low-level Vulkan goes much further so you may want to look into the SVK source-code for more detail.

Avoid looking up data by name after the initialization

It is possible to ask for locations and bindings by name, if you need these, but if you do, it is advisable to do that only in the initalization stage, not every frame. The lookup takes time, and the current implementation only does a linear search which takes time for large amounts of data.

Interleaved geometry

Instead of having separate buffers for vertices, normal vectors and texture coordinates, it is possible to pack them all into a single buffer, with all parts for a vertex being in a single place, like VNTVNTVNT... This has advantages for memory access, grabbing all three in one access or at least in adjacent memory. This is not (yet) supported by the new model handling.

Make all vec3 into vec4

The 4-component vec4 makes memory access more efficient, so even if we only need three components, adding a dummy coordinate can be faster.

Multi-threading on the CPU

One important feature of Vulkan, maybe the most important, is the ability to use the API from multiple CPU threads. This is not part of the labs and I honestly have no idea how hard it is to work multi-threaded with Simpler Vulkan. I expect there to be parts that need to be modified, and it is obvious that we need to think much more about synchronization.

The new labs

The labs are now modified with the new additions.

You must assign location and binding numbers in the shaders, but the host will find these so you don't have to give matching numbers there.

The old Lab 1a is using FBO's to render to texture. Rendering to texture is not yet supported by Simpler Vulkan but instead you are can compy from frame buffer to texture.

IMPORTANT: The current version will only recognise the following types when parsing your shaders. If you use other types, the pipeline builder will fail. You need to add them to the function. These are the supported types:

```
int, float, mat4, mat3, vec3, vec2, vec4
```

Known weaknesses and omissions

A big weakness today is that there is no support/demos for multi-threading, which is sad because it is an important strength of Vulkan. However, I believe that it would complicate the code a lot with synchronizations between threads, so it is not truly a needed focus for a graphics course. Still, it would be desirable to make the package more usable as platform for performance heavy projects.

There is currently no support for rendering to texture, and the support for stencil buffers does not yet work.

The current version is written for Vulkan 1.3. With 1.4, it works but raises warnings. I have not been able to solve these yet.