

# Space and Ships and PewPew

## Implemented Effects

### Particles

Particles are implemented using instancing. I create 200 particles when to display the explosion. I have 32 textures which are chosen at random for each particle. Each particle has a time-to-live, which is also randomly distributed. They get ejected from the center in a random direction with a random speed. Each particle is a billboard which is always camera-facing.

Sources:

- Particle-generation: Vulkan/vulkanmain.cpp:152
- workingdir/shader/billboard.vert
- workingdir/shader/billboard.frag

### Rendering

The renderer consists of 4 stages. The first stage calculates the shadow maps. The second stage is the normal rendering and lighting stage. The third stage does bloom and the last stage composites bloom and lighting to the final image.

### Shadow Mapping

We generate a cascading shadow map with 3 cascades. We use a rather naive and non-optimal approach in generating the light-view-frustums. First we duplicate the view camera but we change the near and far planes. They should represent the space which one shadow map is responsible for. Then we take 4 points from the view-frustum. If we imagine the view-frustum as a deformed cube, each point is one vertex. We chose the vertices so that no 2 vertices are neighbours in the cube. This allows us to generate a sphere out of the 4 points, where the center is equidistant to each vertex of the view-frustum and the range is the distance. From this we can generate an orthogonal camera which looks at the center of the sphere, where the left, right, top and bottom planes are each the radius of the sphere.

Sources:

- Sphere calculation: Vulkan/Camera.cpp:31
- Shadowmap camera generation: Vulkan/Camera.cpp:83

After this we create instances of all solid objects with the model-view matrix with the new cameras. Then we render each object into 2k depth textures.

Sources:

- Sphere calculation: Vulkan/Camera.cpp:31
- Pipeline setup: Vulkan/Vulkan/Specialization/VShadowMapGeneration.cpp

Finally we render a fullscreen quad and do shading. To do the depth test against the depth textures we first have to bring the position into light-screen-space. We do shading in view

space, therefore we have to calculate a view to light-space matrix for each shadowmap. This matrix consists of the inverse world to view matrix, the world to light-view matrix and finally the light-view to light-screen matrix. This can then be used to bring the view-space position into light-screen-space, where we sample the shadowmap and do lighting calculations according to the result.

Sources:

- `workingdir/shader/dirlight.frag`
- View to Light-Space calculation: `Vulkan/World.cpp:408`

## Rendering and Deferred Shading

We start off by rendering all solid objects normally. This first rendering fills 5 GBuffers.

- Diffuse color + glow-intensity: A 32 bit RGBA image. RGB is the diffuse color and the A value is the glow-intensity. The idea for glow-intensity was to make the area light up without a lightsource. We only use the diffuse color at the moment.
- Normals. A 32-bit RG image which holds x and y of the normalized normal-vector in view-space. The third component can be reconstructed if needed.  
(`workingdir/shader/dirlight.frag:38`)
- Material-info: A 32-bit RGBA image. We do not use this image but we planned to use additional material information like specular-power.
- Depth-Stencil: A 24-bit unorm depth and 8-bit stencil image. The depth is pretty self explanatory. The stencil value stores 1 if a solid object was rendered, which can reflect light. If it is 0 then the fragment does not need to be shaded. We do this because the skybox occupies a significant amount of space and we do not want to shade those fragments.

After each solid object is rendered we have 2 light passes and 2 additional non-shaded types of elements to render. The light passes are the global and local lights. The non-shaded elements are projectiles and explosion-particles.

These stages can use the previous GBuffers and write into one light-accumulation-texture. This is a 64-bit RGBA texture where each color value is a 16-bit floating point. Blending is done additively so each calculation adds light to the screen.

The idea of the different GBuffers came from this presentation:

<https://de.slideshare.net/querrillagames/deferred-rendering-in-killzone-2-9691589>

Shadercode:

- `workingdir/shader/dirlight.frag`
- `workingdir/shader/locallight.frag`

## Bloom

For bloom we first generate mipmaps for the light-accumulation-texture which was used in the deferred shader. We use a total of 7 mipmap-layers. Then we render the last 5 layers to a different offscreen image. This image is only 1/4th of the light-accumulation-image in size

and has 5 layers. We use the ping-pong approach to first to do a brightness-pass, then vertical blur and finally a horizontal blur for each 5 mipmap-layers.

The idea to use multiple layers came from this website:

<http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/>

Sources:

- Pipeline setup: Vulkan/Vulkan/Specialization/VBloomRenderStage.cpp
- workingdir/shader/brightness.frag
- workingdir/shader/vbloom.frag
- workingdir/shader/hbloom.frag

## HDR

Finally we perform a simple tone-mapping on the whole scene to smooth out the colors a bit.

Idea from:

<https://learnopengl.com/Advanced-Lighting/HDR>

Sources:

- Pipeline setup: Vulkan/Vulkan/Specialization/VHDRRenderStage.cpp
- workingdir/shader/hdr.frag

## Details of the Abstraction (it's a bit complicated)

I abstracted renderings into Bundles. Each Bundle has an InstanceGroup and a ContextGroup, which in turn group together Instances and Contexts.

A Context is basically a bit of memory, textures and samplers. In most cases they refer to uniform blocks or uniform variables. They are defined by a ContextBase.

A Model consists of a ModelBase and a list of ContextBases. A ModelBase defines how the per-vertex data look like. The ContextBases are data, images and samplers that are model-specific. (e.g. textures, materialinfo, ... setting up a model-specific context: Vulkan/Initializations.cpp)

Instances are registered with an InstanceBase and a Model. Additionally optional per-Instance data needs to be passed. (e.g. vulkanmain.cpp:646)

Rendering with a pipeline requires a list of Contexts, which are identified by their ContextBases, a Model which is identified by the ModelBase and a list of Instances, which are identified by the InstanceBase.

## Memory/Object Management

Uniform-buffers are all stored in a single big buffer. Each Context(which corresponds to a Uniform Buffer) uses a part of the big buffer. For each frame we need to update the Uniforms. We do this by getting a staging buffer each frame, filling it up with all Uniform-data and transferring everything once.

InstanceGroups do something similar. They allocate one big buffer and dump all per-instance data into this buffer. Upload to the GPU works the same as with Uniform-buffers.

Staging buffers come from a central pool which also has a big memory chunk, which smaller buffers are taken from if needed. The implementation is not ideal as smaller buffer objects are allocated instead of getting one big buffer and using only parts.

Sources:

- Vulkan/Vulkan/VBufferStorage.h
- Vulkan/Vulkan/VBufferStorage.cpp

Staging-buffers, fences and transfer-command-buffers are cached and reset when they are no longer in use.

Sources:

- Make Objects which are no longer needed available  
Vulkan/Vulkan/VInstance\_Transfer.cpp:100

Descriptor sets are currently only created once. This means updates need complete pipeline flush, which is slow. The reason for this is updates invalidate all command buffers where the descriptor set is bound. The goal would be to create at least 2 descriptor sets for each Context so that updates only need a rebuild of the command buffers and not a complete GPU flush. Currently we only update descriptor sets when the framebuffer changes, which automatically needs a flush therefore this was not important and is left sub-optimal.

## Debug-Controls

The scene can be stopped and the debug-camera can be enabled via the “i”-Key. The camera can be controlled with WASD for moving in the 2D plane. SPACE can be used to move upwards and LEFT CONTROL can be used to move downwards.

With the mousewheel it is possible to zoom out. Zooming out also increases the move speed for the other controls.

The PLUS key on the Keypad can be used to speed up the automatic camera by a factor of 10 only if the debug-camera is disabled.

## Libraries

I use 3 external libraries for my implementation:

GLFW for window creation, input handling and so forth. <https://www.glfw.org/>

glm for vector, matrix and quaternion math. <https://glm.g-truc.net/0.9.9/index.html>

stb for image loading <https://github.com/nothings/stb>

## Models

I got the X-Wing and Tie-Fighter models from

<https://www.videocopilot.net/blog/2016/05/free-star-wars-model-pack/>

And the Gallofree-frigate from

<https://sketchfab.com/models/071b158d02c044ee9b431aeb28b85b6a>

## Changes for Final Version

I implemented a simple tone mapping algorithm to get the HDR image back to a SDR image, but I am not totally impressed with the result.

For the shadow mapping I additionally fitted the camera so that it exactly matches the view-frustum and not the sphere that we create.

Additionally I adapted the distances for each shadow cascade.

In the first implementation I created instances for each rendered object in each shadow-map. I changed it so that the shadow-casting cameras are now calculated from view-space. With this I can simply use the pre-existing per-instance model-to-view matrix. This allows me to reuse the instances and simply multiply the model-positions with the model-to-view matrix of the object with the world-to-view matrix and the view-to-screen matrix of the shadow-map camera.

For the bloom effect I changed the bloom-texture generation to use 7 images instead of 5 and I changed the size of the image to not be scaled in proportion to the rendered image, but additionally it is rounded to the next power of 2. This gives better results when downscaling as otherwise the linear filtering would sometimes skip lines if the height or width is not divisible by 2.

Descriptor set management was also changed so that I use 2 descriptor-sets for each context. That means if an image or a sampler changes then I switch to the second descriptor set. This allows the image/sampler to be changed without stalling the graphics pipeline. I now also support combined images and samplers.