Vulkan Shadow Mapping

Author: Caio Valente (https://github.com/valentecaio/vk)

Date: 2024-07-06

Introduction

The goal of this project is to study computer graphics using the Vulkan API and to write an application able to render a 3D scene applying the shadow mapping technique to generate shadows in real time using shader programming to access available graphics card functionalities.

Repository structure

```
├── build/                    # output for make - binaries and compiled code (C++ and GLSL)
├── include/                  # header of libraries used in the project
│   ├── base/                 # namespace vks - vulkan abstractions by Sascha Willems [1]
│   ├── glm/                  # geometry library
│   ├── ktx/                  # used by base library to read GLTF models
│   ├── tiny_gltf.h           # used by base library to read GLTF models
│   ├── stb_image.h           # image library used to read textures
│   └── tiny_obj_loader.h     # library used to load obj models
├── lib/
│   └── libbase.a             # compiled base library (only necessary to read GLTF models)
├── models/
│   ├── samplescene.gltf      # model used by shadow mapping example
│   └── viking_room.obj       # model used by tutorial example
├── shaders/                  # GLSL shaders used by both examples
│   └── compile.sh            # bash script to compile GLSL shaders to HLSL
├── src/                      # code written in this project
│   ├── utils/
│   ├── vk/                   # namespace vk - my own vulkan abstractions
│   ├── shadow_mapping.cpp    # main for shadow mapping example
│   └── tutorial.cpp          # main for tutorial example
├── textures/                 # images used as textures
├── Makefile                  # build scripts
└── VulkanShadowMapping.pdf   # this file
```

How to run code

This project uses some header-only libraries that are included in the lib/ directory and should work out-of-the-box.

The static library libbase.a included in the lib/ directory is used to open GLTF models. It should work in unix-like systems, but it can also be re-generated for many other systems by building Sascha Willems' Vulkan repository [1]. The library originally provided many other functionalities in the vks namespace, but I transformed these into header-only files and added them to the include/base/ directory.

The project needs Vulkan tools and GLFW to run:

```
# vulkan
sudo apt install vulkan-tools libvulkan-dev vulkan-validationlayers-dev spirv-tools
libxxf86vm-dev libxi-dev

# glfw
sudo apt install libglfw3-dev
```

The compilation is done with g++ and the build is handled by make, using the Makefile provided in the root directory.

```
make                        # compile code (G++ and GLSL) and generate binaries in build/
./build/shadow_mapping      # run shadow mapping example
./build/shadow_mapping -d   # visualise the shadow mapping in the shadow mapping example
./build/tutorial            # run tutorial example (a simple scene with texture)
```

In the shadow mapping example, the camera can be controlled with WASD keys and mouse (left and right buttons). Pressing space will pause the light animation.

The following parameters can be adjusted in the file shadow_mapping.cpp before initiating vulkan objects:

```
   /************************ public settings ************************/
   bool paused = false;                // flag to pause animations (movement still allowed)
   bool displayShadowMap = false;      // display the shadow map (debug)
   uint32_t shadowMapize = 2048;       // size of the shadow map buffer
   uint32_t gpu_id = 0;                // change gpu here
   float zNear = 1.0f;                 // near plane for the shadow map
   float zFar = 96.0f;                 // far plane for the shadow map
   float lightFOV = 45.0f;             // field of view for the light source
   uint32_t width = 800;               // surface window width
   uint32_t height = 600;              // surface window height
   float timerSpeed = 0.20f;           // multiplier to control the speed of light animation
   glm::vec3 lightPos = glm::vec3();  // light position
   VkClearColorValue bgColor = {0.01f, 0.01f, 0.21f, 1.0f}; // background color
```

Architecture overview

This implementation is strongly based on the examples available at Sascha Willems repository [1].

I used many of their abstractions for vulkan objects, such as:

- vulkanDevice - Contains physical and logical devices and a command pool.
- vulkanSwapChain - Encapsulates swap chain buffers, images and a GLFW surface.
- vulkanBuffer - Encapsulates a VkBuffer and its memory with auxiliary methods to handle memory mapping and unmapping.
- Camera - A first-person camera class that can be positioned and rotated in the scene.

The code handles two main render passes: the shadow map generation pass and the scene rendering pass. Below is a detailed explanation of the architecture and its components.

```
   /************************ vulkan objects ************************/

   VkInstance instance;                // connection between application and vulkan library
   VkDebugUtilsMessengerEXT debugMsgr; // used to report validation layer errors
   vks::VulkanDevice *vulkanDevice;    // wrapper for vulkan device (logical and physical)
   VkDevice device;                    // pointer to vulkanDevice->logicalDevice
   VkPhysicalDevice physicalDevice;    // pointer to vulkanDevice->physicalDevice
   VkCommandPool commandPool;          // pointer to vulkanDevice->commandPool. A pool for submitting
command buffers
   VkQueue queue;                      // graphics queue
   VulkanSwapChainGLFW swapChain;      // wrapper for swap chain
   VkSemaphore semaphPresentComplete;  // semaphore for swap chain image presentation
   VkSemaphore semaphRenderComplete;   // semaphore for command buffer submission and execution

   std::vector<VkCommandBuffer> drawCmdBuffers; // command buffers (one per swap chain image)
   std::vector<VkFence> waitFences;             // wait fences     (one per swap chain image)
   std::vector<VkShaderModule> shaderModules;   // shader modules  (one per shader)

   // information about a queue submit operation
   struct {
     VkSubmitInfo info;
     VkPipelineStageFlags stageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
   } submit;

   // render pass of main scene
   struct ScenePass {
     std::vector<VkFramebuffer> frameBuffers;    // frame buffers for the scene rendering (one per
swap chain image)
     vk::FrameBufferAttachment depth;            // depth attachments
     VkFormat depthFormat;
     vks::Buffer uniformBuffer;                  // uniform buffer for the scene rendering
     VkRenderPass renderPass;
   } scenePass{};
   // offscreen pass for shadow map rendering
   struct OffscreenPass {
     uint32_t width, height;                     // fixed size equal to shadowMapize
     VkFramebuffer frameBuffer;                  // only one because we render to the whole image
     vk::FrameBufferAttachment depth;            // depth attachment (shadow map)
     VkFormat depthFormat = VK_FORMAT_D16_UNORM; // 16 bits is enough for the shadow map
     VkSampler depthSampler;                     // we use this sampler in the fragment shader of the
scene
     vks::Buffer uniformBuffer;                  // uniform buffer for the shadow map rendering
     VkRenderPass renderPass;
   } offscreenPass{};

   // uniform buffer data for the offscreen shadow map rendering (offscreen.vert)
   struct UniformDataOffscreen {
     glm::mat4 depthMVP;
   } uniformDataOffscreen;

   // uniform buffer data for the scene rendering or shadow map visualisation (scene.frag &
debug.frag)
   struct UniformDataScene {
     // variables for scene rendering (scene.frag)
     glm::mat4 projection;    // projection matrix
     glm::mat4 view;          // view matrix
     glm::mat4 model;         // model matrix
     glm::mat4 lightSpace;    // MVP matrix from light's point of view
     glm::vec4 lightPos;      // light position in view space

     // variables for shadow map visualisation (debug.frag)
     float zNear;             // near plane for the shadow map
     float zFar;              //  far plane for the shadow map
   } uniformDataScene;

   // pipelines for each render
   struct Pipelines {
     VkPipeline offscreen;    // pipeline for the offscreen rendering (create the shadow map)
     VkPipeline sceneShadow;  // pipeline for the scene rendering (uses the shadow map)
     VkPipeline debug;        // pipeline for the shadow map visualisation (debug)
     VkPipelineLayout layout; // common uniform layout for all pipelines
     VkPipelineCache cache;   // common cache for the pipelines
   } pipelines;

   // descriptor sets for each render
   struct Descriptors {
     VkDescriptorSet offscreen;    // descriptor set for the offscreen rendering
     VkDescriptorSet scene;        // descriptor set for the scene rendering
     VkDescriptorSet debug;        // descriptor set for the shadow map visualisation
     VkDescriptorSetLayout layout; // common layout for all descriptor sets
     VkDescriptorPool pool;        // common pool for submitting descriptor sets (uniform buffers)
   } descriptors;
```

The Vulkan objects of this application

Vulkan Instance and Devices

The `ShadowMapping` class initialises a Vulkan instance to interface with the Vulkan library and sets up a physical device (GPU) and a logical device. The logical device handles command buffers and queues required for rendering operations.

- Vulkan Instance: The entry point for all Vulkan applications, managing the connection to the Vulkan library.
- Physical Device: Represents the GPU, selected from available devices based on specific criteria.
- Logical Device: Provides an abstraction to interact with the physical device, managing resources like command buffers and queues.

Command Buffers and Pools

Command buffers are allocated from a command pool and are used to record rendering commands. The system uses multiple command buffers, one for each image in the swap chain, to manage rendering operations efficiently.

- Command Pool: A memory pool from which command buffers are allocated.
- Command Buffers: Record rendering commands to be submitted to the GPU for execution.

Swap Chain

The swap chain manages the presentation of rendered images to the screen. It consists of multiple image buffers (color attachments) that are swapped during rendering to achieve smooth frame transitions. It contains the dimensions of the screen and must be recreated if the surface window is resized.

Render Passes

The system uses two main render passes:

1. Shadow Map Pass: This pass renders the scene from the light's perspective to generate a depth map (shadow map). This map is later used to determine shadowed areas in the scene.
2. Scene Render Pass: This pass renders the scene from the camera's perspective, applying shadows using the previously generated shadow map.

Framebuffers and Attachments

Framebuffers are used to hold the render targets during a render pass. The system sets up framebuffers for both the shadow map and the scene render passes, with appropriate depth and color attachments.

- FrameBufferAttachment: Represents an attachment (image, memory, and view) used in framebuffers.
- ScenePass: Holds the framebuffer and attachments for the scene rendering.
- OffscreenPass: Holds the framebuffer and attachments for the offscreen shadow map rendering.

Pipelines and Shaders

We use multiple graphics pipelines for different rendering stages. Each pipeline is configured with its own shaders.

- Offscreen Pipeline: Renders the scene from the light's perspective to generate the shadow map.
- Scene Pipeline: Renders the scene with shadows applied.
- Debug Pipeline: Optionally visualises the shadow map for debugging purposes.

Descriptor Sets and Uniform Buffers

Descriptor sets are used to bind resources like uniform buffers and textures to the shaders. Uniform buffers hold transformation matrices and other parameters required by the shaders.

- Descriptors: Manage the binding of resources to shaders.
- Uniform Buffers: Hold data such as transformation matrices for the light and camera.

Semaphores and Fences

Synchronisation primitives ensure that rendering operations are properly ordered and that the GPU and CPU do not interfere with each other.

- Semaphores: Used to synchronise operations within the GPU.
- Fences: Used to synchronise operations between the CPU and GPU, ensuring that the CPU waits for the GPU to finish rendering before proceeding.

Initialization and Cleanup

The `init` method sets up all Vulkan resources and states required for rendering. This includes creating the Vulkan instance, devices, command buffers, swap chain, framebuffers, and pipelines. The `cleanup` method ensures that all resources are properly released to avoid memory leaks.

- Initialization: Sets up Vulkan instance, physical and logical devices, swap chain, framebuffers, pipelines, and descriptor sets.
- Cleanup: Releases all Vulkan resources, including images, framebuffers, buffers, semaphores, and the Vulkan instance.

Rendering Loop

The main rendering loop handles input events, updates the scene, and renders frames. The `tick` method updates the scene and renders a frame, while the GLFW callbacks handle user input to control the camera.

- Main Loop: Continuously polls for events, updates the scene, and renders frames.
- Input Handling: Uses GLFW callbacks to manage keyboard and mouse input for camera control.

Shadow Mapping Algorithm

Shadow mapping is a technique in computer graphics used to create shadows in 3D scenes. The basic idea is to determine which parts of the scene are visible from the light's perspective and which are not, and then use this information to add shadows when rendering the scene from the camera's point of view.

Shadow pass

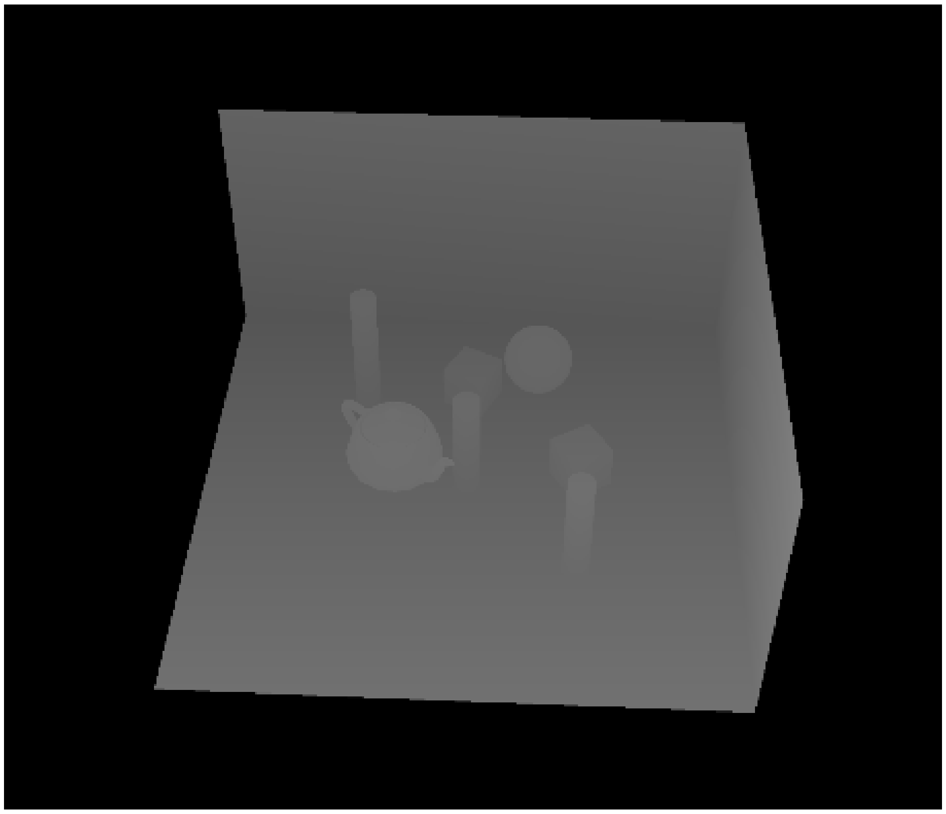
Since there are two steps in the rendering process, we have two render passes. The first one renders the shadow map. It is a simple pass that renders the scene from the light's perspective and stores the depth information (distance from the light to the objects) in a depth map, also called a shadow map. This depth map represents the distances from the light to the closest surfaces in the scene and is used in the next step.

The shadow map does not need to be rendered to screen. Instead, it is stored in a depth buffer. For this reason, we do not need to calculate colours for the map, and we don't need fragment shaders. Only a simple vertex shader is used:

```glsl
#version 450

layout (binding = 0) uniform UBO {
 mat4 depthMVP;
} ubo;

layout (location = 0) in vec3 inPos;

out gl_PerVertex {
 vec4 gl_Position;
};

void main() {
 gl_Position =  ubo.depthMVP * vec4(inPos, 1.0);
}
```

offscreen.vert vertex shader

For debug purposes, we can draw the buffer to the screen, resulting in a black and white scene from the light's point of view. In this image, the points that are closer to the light are brighter.


(A) Result of the first render pass (depth map visualisation).

Scene pass

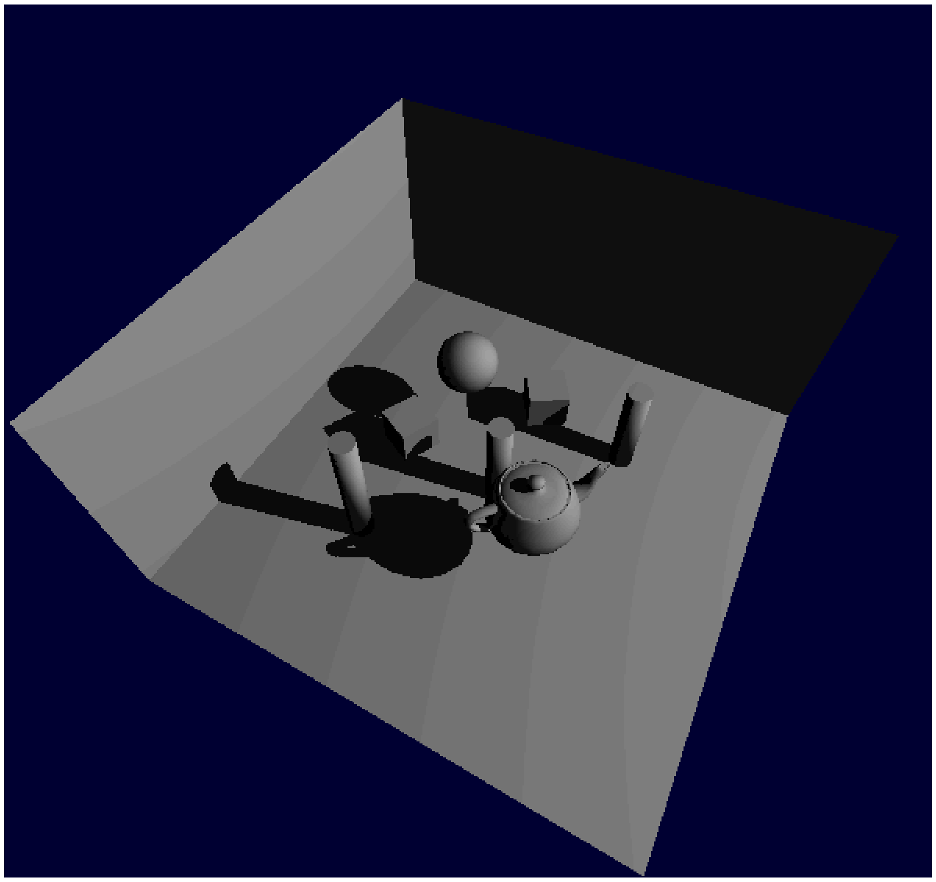
The second pass renders the scene normally from the camera's perspective, but for each pixel, transform its position into the light's coordinate space to determine its position relative to the light source. In the fragment shader, the obtained depth is compared with its corresponding depth value in the pre-calculated shadow map. If the depth value from the camera's perspective is greater than the depth value in the shadow map, it means the pixel is occluded by another object from the light's perspective, and so must be in shadow. In this case, the pixel is darkened.

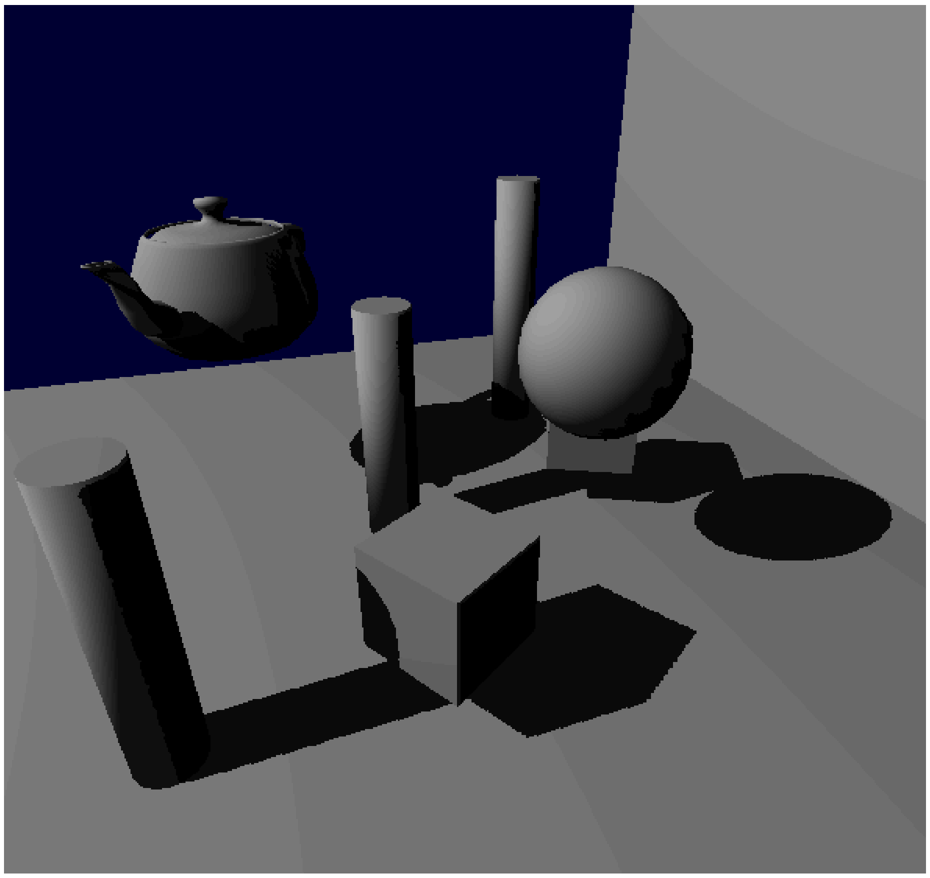
```
#version 450

layout (binding = 0) uniform UBO {
 mat4 projection;
 mat4 view;
 mat4 model;
 mat4 lightSpace;
 vec4 lightPos;
} ubo;

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec2 inUV;
layout (location = 2) in vec3 inColor;
layout (location = 3) in vec3 inNormal;

layout (location = 0) out vec3 outNormal;
layout (location = 1) out vec3 outColor;
layout (location = 2) out vec3 outViewVec;
layout (location = 3) out vec3 outLightVec;
layout (location = 4) out vec4 outShadowCoord;

// matrix to convert coordinates from [-1, 1] to [0, 1]
// we need this because the shadow map is in [0, 1] range
// but the shadow coord is in [-1, 1] range
const mat4 biasMat = mat4(
 0.5, 0.0, 0.0, 0.0,
 0.0, 0.5, 0.0, 0.0,
 0.0, 0.0, 1.0, 0.0,
 0.5, 0.5, 0.0, 1.0
);

void main() {
  vec4 pos = ubo.model * vec4(inPos, 1.0);

  outColor = inColor;
  outNormal = mat3(ubo.model) * inNormal;
  outViewVec = -pos.xyz;
  outLightVec = normalize(ubo.lightPos.xyz - inPos);
  outShadowCoord = (biasMat * ubo.lightSpace * ubo.model) * vec4(inPos, 1.0);
  gl_Position = ubo.projection * ubo.view * ubo.model * vec4(inPos.xyz, 1.0);
}
```

scene.vert vertex shader

```glsl
#version 450

layout (binding = 1) uniform sampler2D shadowMap;

layout (location = 0) in vec3 inNormal;
layout (location = 1) in vec3 inColor;
layout (location = 2) in vec3 inViewVec;
layout (location = 3) in vec3 inLightVec;
layout (location = 4) in vec4 inShadowCoord;

layout (location = 0) out vec4 outFragColor;

// ambient light intensity
#define ambient 0.1

// samples the shadow map to determine if the fragment is in shadow
float textureProj(vec4 shadowCoord) {
 float shadow = 1.0;
 if (shadowCoord.z > -1.0 && shadowCoord.z < 1.0) {
   float dist = texture(shadowMap, shadowCoord.st).r;
   if (shadowCoord.w > 0.0 && dist < shadowCoord.z) {
     shadow = ambient;
   }
 }
 return shadow;
}

void main() {
 float shadow = textureProj(inShadowCoord / inShadowCoord.w);

 vec3 N = normalize(inNormal);
 vec3 L = normalize(inLightVec);
 vec3 diffuse = max(dot(N, L), ambient) * inColor;

 outFragColor = vec4(diffuse * shadow, 1.0);
}
```

scene.frag fragment shader

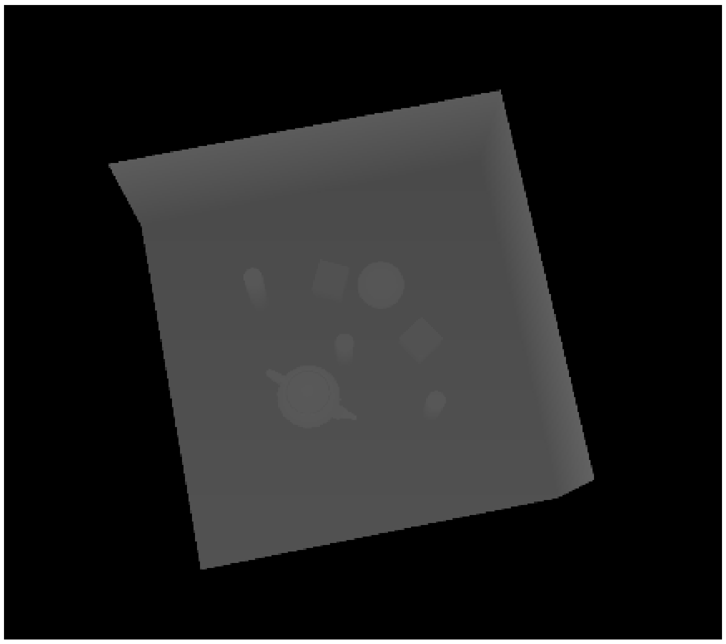

(B) The result of the second pass.

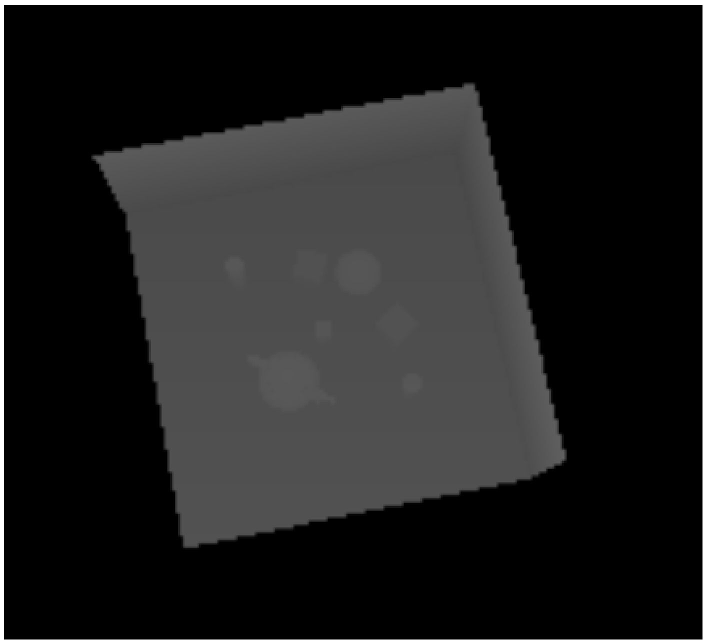

(C) Same scene from (B) rendered from a different point of view.

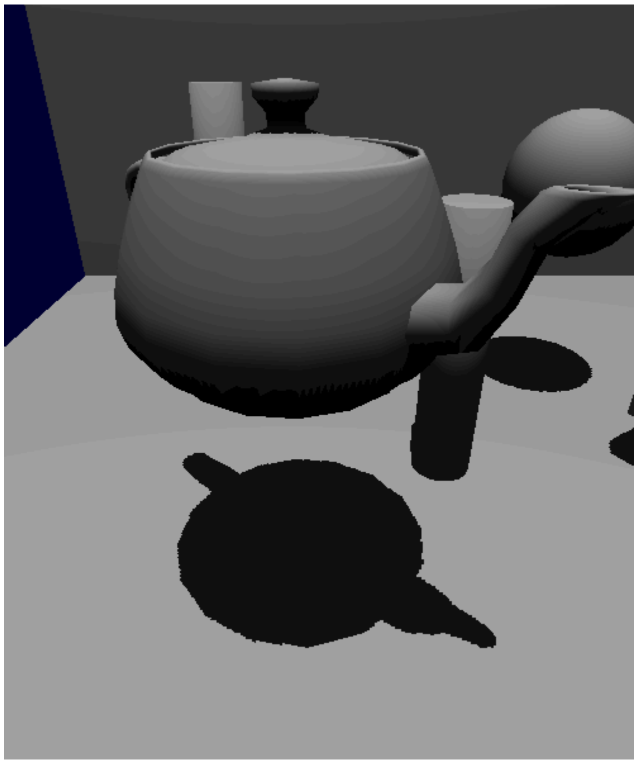
Shadow map buffer size

Since the shadow map is not rendered to the screen, the size of its buffer can vary. A big buffer will offer better shadow details, at the cost of performance and memory usage. Too small buffers, on the other hand, may contain aliases, which are visible jagged edges or "stairs" in the shadow boundaries.

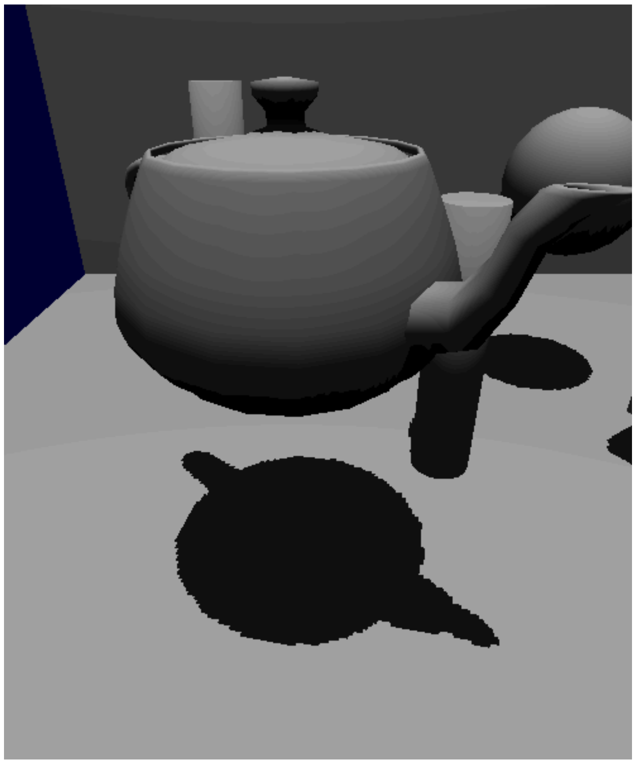
Finding a middle ground that balances quality and performance can be challenging, especially on low-end hardware. Techniques such as Dynamic Resolution Adjustment, which dynamically changes the resolution of the shadow map based on the camera's distance from the shadows, Cascaded Shadow Maps, which use multiple shadow maps with varying resolutions for different scene regions, and Filtering, which smooths out shadow edges and reduces aliasing, can significantly help in achieving this balance.


(A) Visualisation of a 2048 x 2048 shadow map.

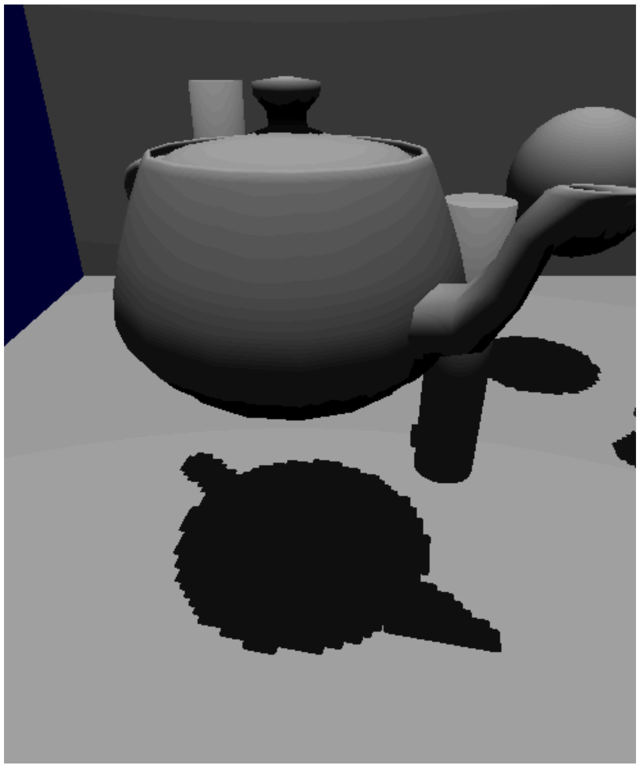

(B) Visualisation of a 256 x 256 shadow map.



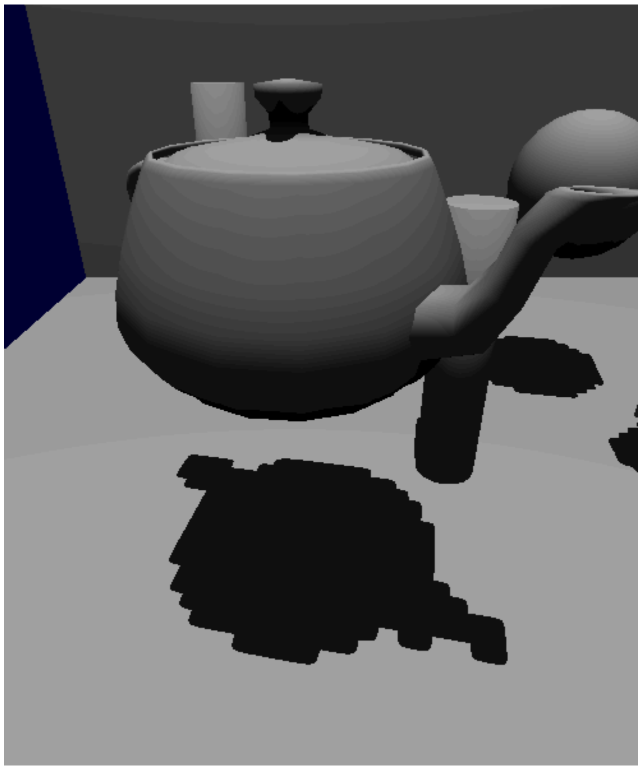
(C) Scene using a 2048 x 2048 shadow map.



(D) Scene using a 1024 x 1024 shadow map.

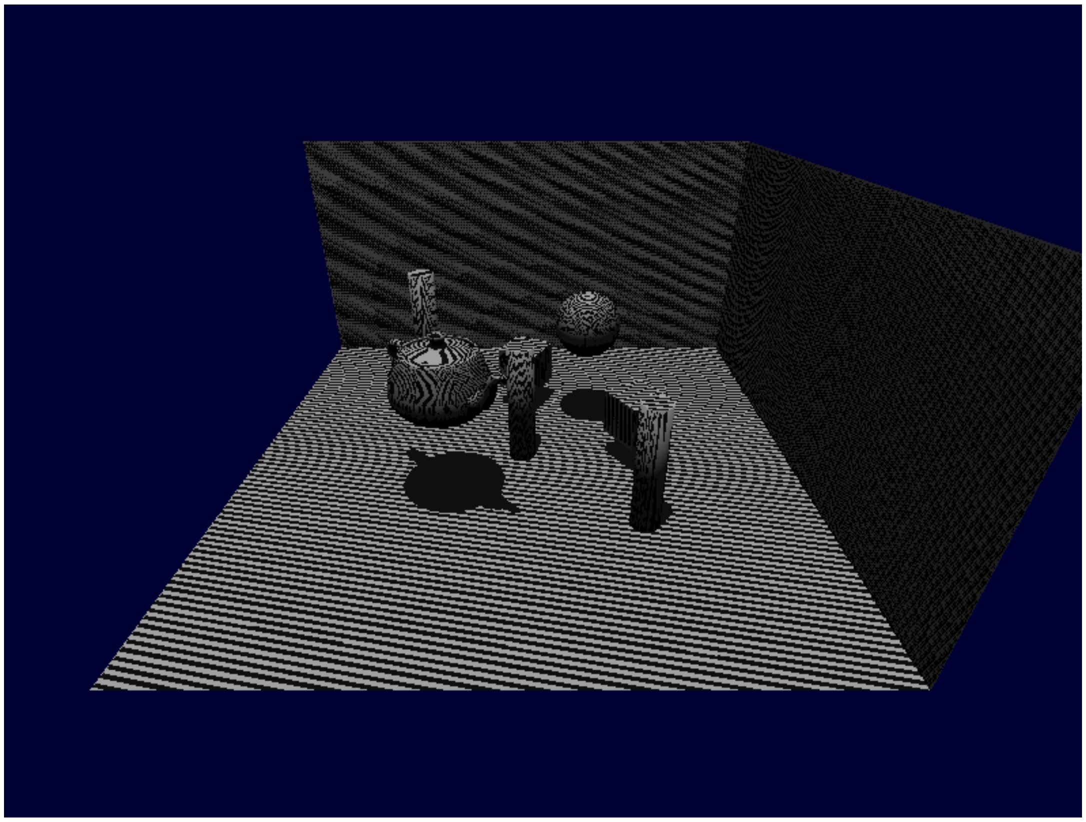


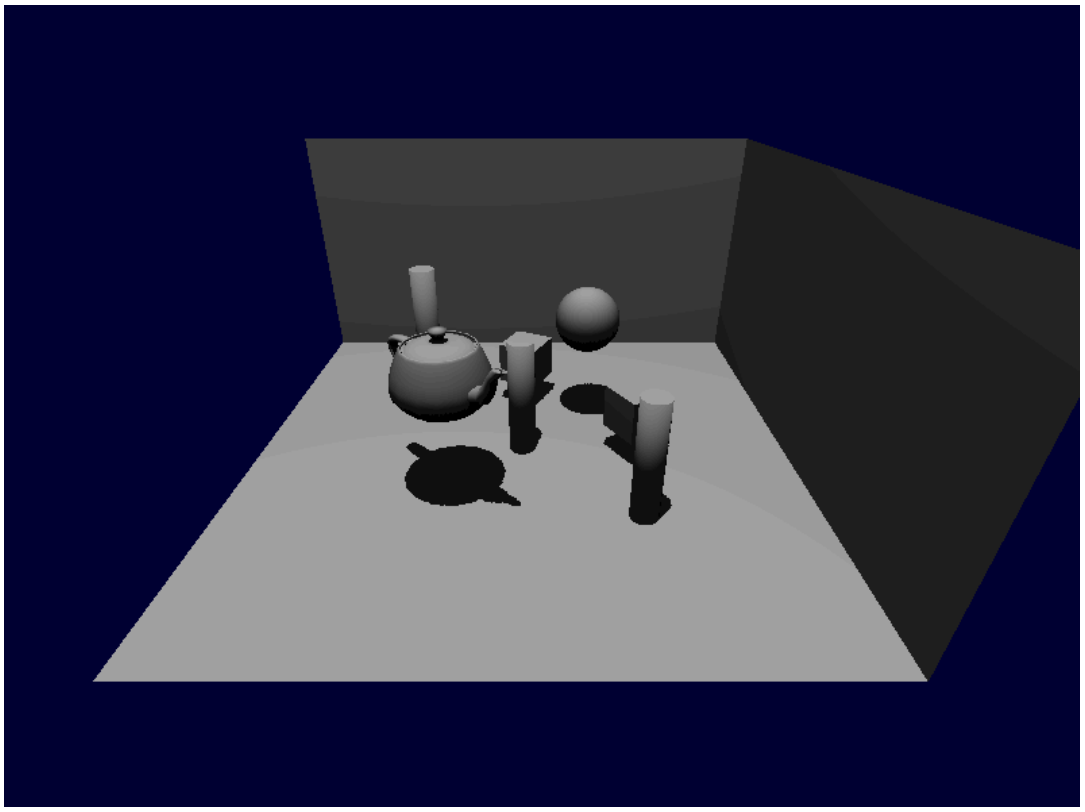
(E) Scene using a 512 x 512 shadow map.



(F) Scene using a 256 x 256 shadow map.

Shadow Acne and Biassing


(A) Scene without shadow biassing


(B) Scene with shadow biassing

```
    // Set depth bias (aka "Polygon offset") to avoid shadow mapping artefacts
    float depthBiasConstant = 1.25f; // constant factor (always applied)
    float depthBiasSlope = 1.75f;    // slope factor (applied depending on polygon's slope)
    vkCmdSetDepthBias(drawCmdBuffers[i], depthBiasConstant, 0.0f, depthBiasSlope);
```

(C) Shadow biassing with Vulkan

Shadow biassing is a crucial technique in shadow mapping to address common artefacts such as shadow acne. Shadow acne occurs due to precision errors when comparing depths in the shadow map, causing surfaces to incorrectly self-shadow and produce a dotted or striped pattern on the surface. This artefact is typically seen when the depth values stored in the shadow map are very close to the actual surface depth values during the rendering pass. To combat this, a small depth bias is applied to the shadow map values, slightly offsetting the depth comparisons to ensure that surfaces do not incorrectly shadow themselves. This bias can be applied as a constant offset or adjusted dynamically based on the slope of the surface relative to the light source, known as slope-scale depth bias. By carefully tuning these bias values, developers can significantly reduce shadow acne, producing cleaner and more visually accurate shadows in rendered scenes.

Vulkan handles depth bias through the vkCmdSetDepthBias command, which allows specifying how depth bias is applied during rendering. This command takes three parameters: a constant factor, a clamp value, and a slope factor. The constant factor (depthBiasConstant) uniformly offsets the depth values, while the slope factor (depthBiasSlope) adjusts the bias based on the angle of the polygon relative to the light, thereby addressing the varying depth values on sloped surfaces. The clamp value can be used to set a maximum allowable depth bias.

References

[1] "Vulkan" repository containing C++ Vulkan examples by Sascha Willems.
https://github.com/SaschaWillems/Vulkan/

[2] "Interactive Computer Graphics" classes from The University of Utah, taught by Cem Yuksel and available on their youtube channel.
https://youtu.be/UVCuWQV_-Es?list=PLplnkTzzqsZS3R5DjmCQsqupu43oS9CFN
https://graphics.cs.utah.edu/courses/cs6610/spring2021/

[3] Shadow Mapping OpenGL tutorials, by ogldev (Etay Meiri), learnopengl (Joey de Vries), opengl-tutorials and thebennybox.
https://ogldev.org/www/tutorial23/tutorial23.html
https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/
https://www.youtube.com/watch?v=EsccgeUpdsM&ab_channel=thebennybox

[4] "Vulkan tutorial" by Alexander Overvoorde.
https://docs.vulkan.org/tutorial/latest/00_Introduction.html

[5] "Vulkan specification" by The Khronos Vulkan Working Group.
https://registry.khronos.org/vulkan/specs/1.3/html/

[6] "Vulkan-Samples" repository by The Khronos Vulkan Working Group.
https://github.com/KhronosGroup/Vulkan-Samples

[7] "GLFW Vulkan guide".
https://www.glfw.org/docs/3.3/vulkan_guide.html

[8] "Shadow Acne" explanation by DigitalRune.
https://digitalrune.github.io/DigitalRune-Documentation/html/3f4d959e-9c98-4a97-8d85-7a73c26145d7.htm