4

Moving from 2D to 3D World

With the opening of chapter, we have entered into an intermediate phase of learning Vulkan. This chapter would take a big leap and delve into 3D graphics basics concepts in practical step-by-step in an incremental manner. In this chapter, we will pull the 3rd dimension (z) of coordinate system in and build a simple application to demonstrate array and indexed based drawing APIs in Vulkan. We will add 3D transformation with simple Model, View, and Projection analogy and learn how to render perspective and orthographic views.

A 3D graphics application simulates the perception of depth where the 3D objects appear in correct order from the viewer’s sight, this is achieved using the Depth buffer. In this chapter, we will learns the steps to implement depth buffer in Vulkan and see the difference it makes. This chapter will also introduce Assimp library to read 3D mesh object from a variety of formats. We will learn to use multiple pipeline objects in Vulkan and render our meshes into wireframe and solid geometry. Later we will bring realism by adding light shading with the physics of lighting and material.

As we approach the closing, we learn build a procedural terrain model with the help of perlin noise and simulate a fly through application with proper cube mapping.

In this chapter, we will cover the following topics and by the end of the chapter you should be able to run your first Vulkan application to render a triangle on your system.

* Getting started with 3D Programming in Vulkan
* Working with Meshing with Assimp
* Let there be a light – Bringing Light Shading
* Spherical mapping reflect
* Fly through procedural terrains
* Summary

# Getting started with 3D Programming in Vulkan

Building a 3D application is simple. In this section, we will build our first 3D application in Vulkan step-by-step as follows:-

1. **Define Geometry:** Define the object coordinates to provide the basic skeleton of the 3D object geometry.
2. **Add 3D Transformation:** Position the rendered object in 3D space and apply 3D transformation on it.
3. **Enable the Depth buffer:** The depth buffer allows various geometry faces to appear in correct order from the viewers point in the scene.
4. 1. Introduction to uniforms
   2. Prespective view setting the aspect.

## Define Geometry

In chapter 2, under the example *Ch2\_03\_VulkanBufferObjects,* we build a simple 2D triangle using two vertex attributes position and color in 2D coordinate system. For this example, we are taking that example as the base and introduce change to render a simple cube in 3D coordinate system.

struct Vertex

{

glm::vec3 m\_Position; **// Vertex Position => x, y, z**

glm::vec3 m\_Color; **// Color format => r, g, b**

};

The geometry data consists of 6 square cube faces where each square face comprises of two triangles faces. Each vertex is stored into an interleaved form storing the position followed by color information. The below code snippet shows one of square face as an illustration.

const float Dim = 0.5f;

**/\* 6 faces each containing two triangles \*/**

static const Vertex cubeVertices[] =

{

{ glm::vec3( Dim, -Dim, -Dim), glm::vec3(0.f, 0.f, 0.f) },

{ glm::vec3(-Dim, -Dim, -Dim), glm::vec3(1.f, 0.f, 0.f) },

{ glm::vec3( Dim, Dim, -Dim), glm::vec3(0.f, 1.f, 0.f) },

{ glm::vec3( Dim, Dim, -Dim), glm::vec3(0.f, 1.f, 0.f) },

{ glm::vec3(-Dim, -Dim, -Dim), glm::vec3(1.f, 0.f, 0.f) },

{ glm::vec3(-Dim, Dim, -Dim), glm::vec3(1.f, 1.f, 0.f) },

. . . . .

};

While setup, use the cube geometry information and create the device buffer object.

void Cube::Setup()

{

uint32\_t dataSize = sizeof(cubeVertices);

uint32\_t dataStride = sizeof(cubeVertices[0]);

CreateVertexBuffer(cubeVertices, dataSize, dataStride);

. . . . .

}

The cube geometry is allocated on the device using helper function CreateBuffer() from VulkanHelper. The vertex buffer will be used directly by the graphics pipeline for various shader stages in order to directly access and execute it on the GPU.

Uploading the vertex buffer data on the device is just not sufficient, we also need to indicate the graphics pipeline how to interpret this data. For example, where the data can be read from, how many attribute it contains, what is the size of an individual vertex information. This is done with the help of vertex input binding (VkVertexInputBindingDescription) and input attribute (VkVertexInputAttributeDescription). For more information on buffer resource, how to create buffer resource in Vulkan please refer to chapter 2, Getting Started with Vulkan.

**// Helper Vulkan buffer resource structure defined in VulkanHelper.h**

struct VulkanBuffer

{

VkBuffer m\_Buffer; **// Buffer resource object**

uint64\_t m\_DataSize; **// Actual data size**

VkDeviceMemory m\_Memory; **// Allocated device memory**

VkMemoryRequirements m\_MemRqrmnt; **// Memory requirement**

VkMemoryPropertyFlags m\_MemoryFlags;**// Memory properties flags**

};

**// Cube.h**

struct { VulkanBuffer m\_BufObj; } VertexBuffer;

void Cube::CreateVertexBuffer(const void \* vertexData,

uint32\_t dataSize, uint32\_t dataStride)

{

VertexBuffer.m\_BufObj.m\_DataSize = dataSize;

VertexBuffer.m\_BufObj.m\_MemoryFlags =

VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT | VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT;

const VkPhysicalDeviceMemoryProperties& memProp =

m\_VulkanApplication->m\_physicalDeviceInfo.memProp;

const VkDevice& device = m\_VulkanApplication->m\_hDevice;

**// Create a vertex buffer resource & write data**

VulkanHelper::CreateBuffer(device, memProp, VertexBuffer.m\_BufObj,

VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT, vertexData);

**// Indicates the rate at which the information will be**

**// injected for vertex input.**

m\_VertexInputBinding.binding = 0;

m\_VertexInputBinding.inputRate = VK\_VERTEX\_INPUT\_RATE\_VERTEX;

m\_VertexInputBinding.stride = **dataStride**;

**// The VkVertexInputAttribute interpreting the data.**

m\_VertexInputAttribute[0].binding = 0;

m\_VertexInputAttribute[0].location = 0;

m\_VertexInputAttribute[0].format = **VK\_FORMAT\_R32G32B32\_SFLOAT**;

m\_VertexInputAttribute[0].offset =

**offsetof(struct Vertex, m\_Position);**

m\_VertexInputAttribute[1].binding = 0;

m\_VertexInputAttribute[1].location = 1;

m\_VertexInputAttribute[1].format = **VK\_FORMAT\_R32G32B32\_SFLOAT**;

m\_VertexInputAttribute[1].offset =

**offsetof(struct Vertex, m\_Color);**

}

The last thing needs to update is the draw command vkCmdDraw specifying the geometry specification.

const int vertexCount = sizeof(cubeVertices) / sizeof(Vertex);

vkCmdDraw(m\_VulkanApplication->m\_hCommandBufferList[i], vertexCount,1,0,0);

The Cube’s object m\_Cube is created in MyFirst3DApp class (derived from VulkanApp)

class MyFirst3DApp : public VulkanApp

{

. . .

virtual void Configure(); **// Set dimensions, layer & extension**

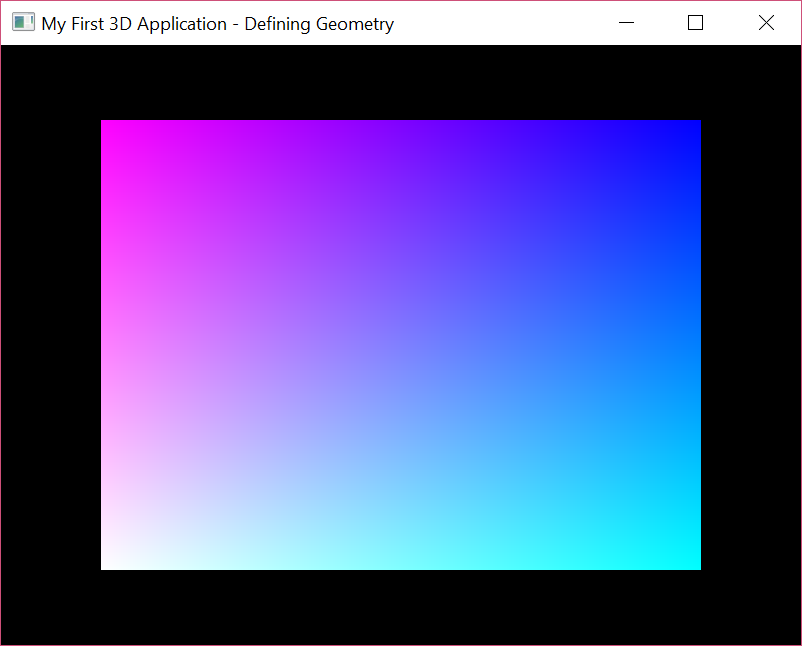
virtual void Setup(); **// Calls m\_Cube->Setup();**

virtual void Update(){} **// This will be used in next section**

Cube\* m\_Cube; // Object is create in Constructor of this class.

};

The below output shows rendered colored cube. Well! it appears to be more like a 2D flatten rectangle rather a 3D cube. This is because only one face of the cube displaying, the cube also have incorrect dimensions due to the fact that we have not addressed the screen aspect ratio into account. Let’s move to next section and fix these issues.



## Add 3D Transformation

In this section, we will add some 3D transformation to our existing application using *Model-View-Projection* (MVP) analogy. In this, we will rotate our cube in the 3D space (*Modeling*), the rotating cube will be observed from some location in the 3D space (*Viewing*) and also define the projection system (*Projection* like perspective or orthographic view).

The rotation will be performed using DrawableInterface class Rotate() api. This class provide some generic interfaces that we used throughout the book in order to implement a drawing item class like Cube, Quad, Rectangle etc. The below snippet shows a snapshot of this interface, for detailed information we encourage our readers to visit VulkanApp.h.

class DrawableInterface

{

. . . . .

**// Life Cycle**

virtual void Setup() UNIMPLEMENTED\_INTEFACE

virtual void Update() UNIMPLEMENTED\_INTEFACE

**// 3D Transformation**

void Rotate(...) {...}

void Translate(...) {...}

void Scale(...) {...}

. . . . .

};

In order to utilize the 3D transformation API’s we must derived the cube class from this interface.

class Cube : public DrawableInterface { ... };

First, specify the view and projection information from the application in the setup() function as shown below.

void MyFirst3DApp::Setup()

{

static glm::mat4 Projection =

glm::perspective(glm::radians(45.0f), **// FOV**

800.0f/600.0f, **// Aspect Ratio**

0.1f, **// Near plane**

100); **// Far plane**

m\_Cube->**SetProjection**(&Projection);

static glm::mat4 View = glm::lookAt(

glm::vec3(0, 0, 5), **// Eye location**

glm::vec3(0, 0, 0), **// Center location**

glm::vec3(0, 1, 0)); **// Up direction**

m\_Cube->**SetView**(&View);

m\_Cube->Setup();

}

Next, the specify the 3D transformation where the below code shows how the rotation is applied on the Cube object m\_Cube in the update() function from the our custom application class MyFirst3DApp (derived from VulkanApp).

void MyFirst3DApp::Update()

{

static float rot = 0;

m\_Cube->Rotate((rot += .005f), 1.0f, 1.0f, 1.0f);

m\_Cube->Update();

}

The transformation information is shared with vertex shader each time a frame is rendered. The transformation keeps on changing therefore we must store it as a uniform buffer. In the current implementation, we created a uniform buffer object of size equal to 4x4 float matrix in the CreateUniformBuffer().

struct {

VulkanBuffer m\_BufObj;**// Contains Vulkan buffer resource metadata**

VkDescriptorBufferInfo m\_BufferInfo; **// Buffer info**

std::vector<VkMappedMemoryRange>m\_MappedRange;

**// Metadata of memory mapped objects**

uint8\_t\* m\_Data; **// Host pointer containing the mapped device**

**// address which is used to write data into.**

size\_t m\_DataSize; **// Data size.**

} UniformBuffer;

The UniformBuffer is the helper structure aggregating all the information for uniform buffer object, including the necessary information required to create the descriptors. In the below code the uniform buffer is create with host visibility memory property flag, this is because we are interested to update this memory from the host side.

One the Vulkan buffer object is create using VulkanHelper::CreateBuffer(), we use vkMapMemory(..) api to map the device memory virtual address into UniformBuffer.m\_Data pointer at the host side and will use this pointer to populate the transformation information in the Cube::Update() function. The mapping metadata is important to populate in the UniformBuffer.m\_MappedRange indicating where to map the physical backing from and the start and range of mapped memory.

void Cube::CreateUniformBuffer()

{

UniformBuffer.m\_BufObj.m\_MemoryFlags =

**VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT**;

UniformBuffer.m\_BufObj.m\_DataSize **= sizeof(glm::mat4)**;

**// Create buffer resource states using VkBufferCreateInfo**

VulkanHelper::**CreateBuffer**(m\_VulkanApplication->m\_hDevice,

m\_VulkanApplication->m\_physicalDeviceInfo.memProp,

UniformBuffer.m\_BufObj, **VK\_BUFFER\_USAGE\_UNIFORM\_BUFFER\_BIT**);

**// Map the GPU memory on to local host**

VulkanHelper::MapMemory(m\_VulkanApplication->m\_hDevice,

UniformBuffer.m\_BufObj.m\_Memory, 0, UniformBuffer.m\_BufObj.m\_MemRqrmnt.size, 0, UniformBuffer.m\_Data);

**// We have only one Uniform buffer object to update**

UniformBuffer.m\_MappedRange.resize(1);

**// Populate the VkMappedMemoryRange data structure**

UniformBuffer.m\_MappedRange[0].sType =

VK\_STRUCTURE\_TYPE\_MAPPED\_MEMORY\_RANGE;

UniformBuffer.m\_MappedRange[0].memory =

UniformBuffer.m\_BufObj.m\_Memory;

UniformBuffer.m\_MappedRange[0].offset = 0;

UniformBuffer.m\_MappedRange[0].size =

UniformBuffer.m\_BufObj.m\_MemRqrmnt.size;

**// Update descriptor buffer info in order to write descriptors**

UniformBuffer.m\_DescriptorBufInfo.buffer =

UniformBuffer.m\_BufObj.m\_Buffer;

UniformBuffer.m\_DescriptorBufInfo.offset = 0;

UniformBuffer.m\_DescriptorBufInfo.range =

UniformBuffer.m\_BufObj.m\_DataSize;

}

The uniform buffer created on the GPU memory are shared with the pipeline shader stages and those stage need to know about the nature of this data. Such as where the inputs will be read from, how many attributes does each component have with it respective location. This interpretation is done through descriptor and descriptor sets. Kindly, browse the chapter 4 <recipe name> source code for the detailed implementation of Descriptor and descriptor layout in CreateDescriptor().

void Cube::CreateDescriptor()

{

CreateDescriptorSetLayout();

CreateUniformBuffer();

CreateDescriptorPool();

CreateDescriptorSet();

}

For more detail information on Uniform, Descriptors and Descriptor sets please refer to <Provide references to Uniform>. For indepth API detail on Descriptor and descriptor for the usage of uniform in Vulkan API and Push Constants, please refer to chapter 10, Descriptors and Push Constant in one of our another Packt publication, *Learning Vulkan* from the same author.

The transformation on the Cube object is stored in the form of 4x4 matrix MVP containing model (m\_Model), view (m\_View) and projection (m\_Projection) information. With the below code, you must be wondering why the projection and view are pointers, this is because several drawing object can be displayed with same projection and view information therefore we don’t want to spend our previous resources with redundant information. Also, the m\_Model simply contains the transformation of drawing object itself and there might be several objects with each one with varying transformation attributes.

In the Update() function the uniform buffer invalidated before using it in order to make it visible to the host, this is done using vkInvalidateMappedMemoryRanges() api. Use the mapped UniformBuffer.m\_Data pointer and copy the data. Lastly call the vkInvalidateMappedMemoryRanges() in order to ensure host write are available updated into device memory.

This write data is refected to on the device if the memory proper is specified with VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT automatically by the driver. However, in the absence of this flag must be called explicitly.

void Cube::Update()

{

glm::mat4 MVP = (\*m\_Projection) \* (\*m\_View) \* m\_Model;

vkInvalidateMappedMemoryRanges(m\_VulkanApplication->

m\_hDevice, 1, &UniformBuffer.m\_MappedRange[0]);

**// Copy updated data into the mapped memory**

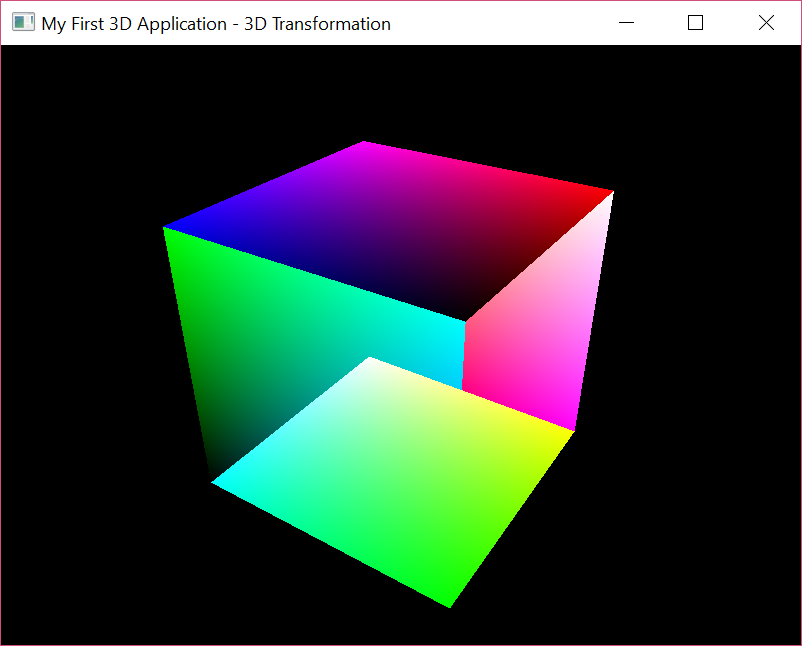
memcpy(UniformBuffer.m\_Data, &MVP, sizeof(MVP));

vkFlushMappedMemoryRanges(m\_VulkanApplication->m\_hDevice,

1, &UniformBuffer.m\_MappedRange[0]);

}

The below image shows the image of the 3D cube after applying the 3D transformation. You must have notices the cube is somewhat appearing as the faces are not rendering in correct order. This is because the graphics pipeline does not have the rule which face to display correctly based of the depth of faces from the viewer location. In the next section, we will fix this by introducing depth buffer.



## Bring depth perception with Depth buffer

The third thing required to implement a 3D scene is the perception of depth which is done with the help of *depth buffer* or *Z-buffering* in the computer graphics. In depth testing, a 2D array is used to store the depth information of fragment that is competing to acquire the screen space. Fragments that belongs to the same screen space are compared based on the predefined depth comparison rules (like chose closer fragment to the viewer). Only those fragment are display on the screen which passes this depth test rule.

In this section, we will implement the depth buffer in Vulkan. The depth testing is an implicit operation of the graphics pipeline, however simply turning on the flags would not make it work. We need to full fill the prerequisites in order to utilize it. Follow are the steps needed in order to allow a graphics pipeline to use depth testing.

1. Create a depth image of size same as swapchain color image view size. Remember, in our first chapter, we acquire the color image views from the swapchain in order to store the render scene’s color information into these image views. These image views are later specified into the render pass and framebuffer. Similarly, the depth image we are using will be specified in the render pass and frame buffer so that it is available to the graphics pipeline. The Depth image are created in VulkanApp::CreateDepthImage() function.

struct {

VkFormat m\_Format;

VulkanImageView m\_ImageView;

VulkanImage m\_Image;

}DepthImage;

void VulkanApp::CreateDepthImage()

{

DepthImage.m\_Format = VK\_FORMAT\_D16\_UNORM;

DepthImage.m\_Image.extent =

{ p\_pWindow->width()), m\_pWindow->height()), 1 };

VkImageCreateInfo imageInfo = {};

imageInfo.sType = VK\_STRUCTURE\_TYPE\_IMAGE\_CREATE\_INFO;

imageInfo.imageType = VK\_IMAGE\_TYPE\_2D;

imageInfo.format=DepthImage.m\_Format;**// VK\_FORMAT\_D16\_UNORM**

imageInfo.extent = DepthImage.m\_Image.extent;

imageInfo.samples = NUM\_SAMPLES; **// VK\_SAMPLE\_COUNT\_1\_BIT**

imageInfo.tiling = VK\_IMAGE\_TILING\_OPTIMAL;

imageInfo.usage =

VK\_IMAGE\_USAGE\_DEPTH\_STENCIL\_ATTACHMENT\_BIT;

VulkanHelper::CreateImage(m\_hDevice,

m\_physicalDeviceInfo.memProp,

DepthImage.m\_Image, &imageInfo);

VkImageAspectFlags aspectMask=VK\_IMAGE\_ASPECT\_DEPTH\_BIT;

. . .

**// Set image layout for depth stencil image**

if (!m\_hCommandPool) {

VulkanHelper::CreateCommandPool(m\_hDevice,

m\_hCommandPool, m\_physicalDeviceInfo); }

VulkanHelper::AllocateCommandBuffer(m\_hDevice,

m\_hCommandPool, &cmdBufferDepthImage);

VulkanHelper::BeginCommandBuffer(cmdBufferDepthImage);

VulkanHelper::SetImageLayout(DepthImage.m\_Image.image,

aspectMask, VK\_IMAGE\_LAYOUT\_UNDEFINED, VK\_IMAGE\_LAYOUT\_DEPTH\_STENCIL\_ATTACHMENT\_OPTIMAL, (VkAccessFlagBits)0, cmdBufferDepthImage);

VulkanHelper::EndCommandBuffer(cmdBufferDepthImage);

VulkanHelper::SubmitCommandBuffer(m\_hGraphicsQueue,

cmdBufferDepthImage);

**// Create the image view**

VkImageViewCreateInfo imgViewInfo = {};

imgViewInfo.sType =

VK\_STRUCTURE\_TYPE\_IMAGE\_VIEW\_CREATE\_INFO;

imgViewInfo.image = DepthImage.m\_Image.image;

imgViewInfo.format = VK\_FORMAT\_D16\_UNORM;

imgViewInfo.components =

{VK\_COMPONENT\_SWIZZLE\_IDENTITY};

imgViewInfo.subresourceRange.aspectMask = aspectMask;

imgViewInfo.viewType = VK\_IMAGE\_VIEW\_TYPE\_2D;

VulkanHelper::CreateImageView(m\_hDevice,

**DepthImage.m\_ImageView**, &imgViewInfo);

}

1. The created depth image view will be used to create a depth attachment, this attachment will be used in the render pass to create render pass instance.

void VulkanApp::CreateRenderPass()

{

VkAttachmentDescription attachments[2] = {};

. . .

attachments[0].format = m\_hSwapChainImageFormat;

attachments[0].samples = NUM\_SAMPLES;

attachments[0].loadOp = VK\_ATTACHMENT\_LOAD\_OP\_CLEAR;

attachments[0].storeOp = VK\_ATTACHMENT\_STORE\_OP\_STORE;

attachments[0].initialLayout = VK\_IMAGE\_LAYOUT\_UNDEFINED;

attachments[0].finalLayout =

VK\_IMAGE\_LAYOUT\_PRESENT\_SRC\_KHR;

**attachments[1].format = DepthImage.m\_Format;**

**attachments[1].samples = NUM\_SAMPLES;**

**attachments[1].loadOp = VK\_ATTACHMENT\_LOAD\_OP\_CLEAR;**

**attachments[1].storeOp = VK\_ATTACHMENT\_STORE\_OP\_STORE;**

**attachments[1].stencilLoadOp = VK\_ATTACHMENT\_LOAD\_OP\_LOAD;**

**attachments[1].stencilStoreOp =**

**VK\_ATTACHMENT\_STORE\_OP\_STORE;**

**attachments[1].initialLayout = VK\_IMAGE\_LAYOUT\_UNDEFINED;**

**attachments[1].finalLayout =**

**VK\_IMAGE\_LAYOUT\_DEPTH\_STENCIL\_ATTACHMENT\_OPTIMAL;**

**attachments[1].flags =**

**VK\_ATTACHMENT\_DESCRIPTION\_MAY\_ALIAS\_BIT;**

VkAttachmentReference attachmentRef[2] = {};

attachmentRef[0] = { 0, VK\_IMAGE\_LAYOUT\_COLOR-

\_ATTACHMENT\_OPTIMAL }; // Color attachment

**attachmentRef[1] = { 1, VK\_IMAGE\_LAYOUT\_DEPTH\_-**

**STENCIL\_ATTACHMENT\_OPTIMAL };** **// Depth attachment**

**// Fill in the sub pass with color and depth attachment**

VkSubpassDescription subpass = {};

subpass.pColorAttachments = &attachmentRef[0];

subpass.pDepthStencilAttachment = &attachmentRef[1];

**// Create the render pass**

VkRenderPassCreateInfo renderPassInfo = {};

renderPassInfo.sType = VK\_STRUCTURE\_TYPE\_-

RENDER\_PASS\_CREATE\_INFO;

renderPassInfo.attachmentCount = 2; **// Color + Depth**

vkCreateRenderPass(renderPassInfo, ...);

}

1. Frame buffer also need to specify with depth image view as an attachment. The following show the changes indicate in bold in the CreateFramebuffers() function.

void VulkanApp::CreateFramebuffers()

{

**// Resize the list based on swap chain image view count**

m\_hFramebuffers.resize(m\_hSwapChainImageViewList.size());

VkImageView attachments[2];

**attachments[1] = DepthImage.m\_ImageView.imageView;**

**// Setup VkFramebufferCreateInfo for frame buffer object**

VkFramebufferCreateInfo framebufferInfo = {};

framebufferInfo.sType =

VK\_STRUCTURE\_TYPE\_FRAMEBUFFER\_CREATE\_INFO;

framebufferInfo.renderPass = m\_hRenderPass;

**framebufferInfo.attachmentCount = 2; // Color and Depth**

framebufferInfo.pAttachments = attachments;

. . .

**// For each swapchain image view create a framebuffer**

for(int i=0; i < m\_hSwapChainImageViewList.size(); i++)

{

attachments[0] = m\_hSwapChainImageViewList[i];

**// Create frame buffer object**

vkCreateFramebuffer(m\_hDevice, &framebufferInfo,

nullptr, &m\_hFramebuffers[i]);

}

}

1. The depth image must be created before the render pass is created.

void VulkanApp::InitializeVulkan()

{

. . .

CreateSwapChain(); // Create Swap chain

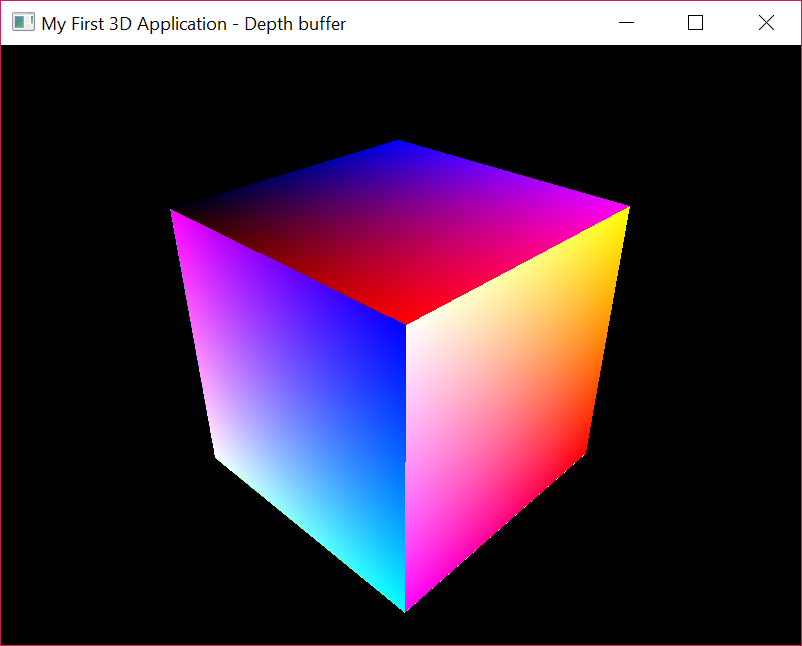
**CreateDepthImage(); // Create the depth image**

CreateRenderPass(); // Create Render Pass

. . .

}

Following is the output of the same



1. color attachment and we indicated to the These color image.Unlike the color image that stores the color information, the depth image stores depth information of the  
   primitive's corresponding fragment from the camera view. The depth image's dimension is  
   usually the same as the color image. Not a hard-and-fast rule, but in general, the depth  
   image stores the depth information as 16-, 24-, or 32-bit float values
2. 3D transformation with the help of Model-View-Projection analogy.
   1. IT’S A CUBE BUT DOES NOT SEEMS TO BE A CUBE.
   2. IT RECTANGLE NOT A SQUARE

# Drawing Mesh with Assimp

A mesh is a collection of various geometrical attributes like vertex, normal, texture coordinate and texture assets like norma maps, albedo etc. In this section, we will learn to draw a simple 3D Mesh model using Assimp library. Assimp can be downloaded from [http://assimp.org](http://assimp.org/), it simplifies programmer efforts without going into the details of the formatted mesh type parsing and loading. Let’s proceed to data structure, implementation and rendering section of mesh using assimp.

## Data structures

**CPU data structure:** The Vertex data structure contains various attribute like position, normal, texture coordinates etc. In this example, we will only use the position attribute and to keep the example simple and easy to understand. In the next example, we will use the vertex normal attribute and show a simple diffuse light example.

struct Vertex {

Vertex(const glm::vec3& p\_Pos) { m\_Pos = p\_Pos; }

glm::vec3 m\_Pos;

};

The MeshNode stores the list vertices in a mesh, each vertex is indicated by an index number. The indices are used to define geometry faces, a face comprises of three vertices where each vertex can be indicated through a 16 or 32 bit index.

struct MeshNode {

std::vector<Vertex> Vertices;

std::vector<unsigned int> Indices;

};

A Mesh self-comprises of many individual mesh nodes and represented by m\_Nodes.

std::vector<MeshNode> m\_Nodes;

**GPU Data structures:**

The VulkanBuffer represents a data structure the stores GPU data structure, it contains the device buffer (buffer) containing mesh information stored on the GPU. The m\_DataSize is actual size of the buffer, however, the size allocated on the GPU could be aligned to the gpu memory requirements and hence may be equal or bigger to the m\_DataSize. The physical backing is made on the m\_Memory as per the requirement provided specified by the m\_MemRqrmnt and memory property flags m\_MemoryFlags.

struct VulkanBuffer

{

VkBuffer m\_Buffer; **// Buffer resource object**

uint64\_t m\_DataSize; **// Actual data size request for, use**

**// m\_MemRqrmnt.size for actual backing size**

VkDeviceMemory m\_Memory; **// Buffer resource object's**

**// allocated device memory**

VkMemoryRequirements m\_MemRqrmnt; **// Memory requirement for**

**// the allocation buffer,**

**// useful in mapping/unmapping**

VkMemoryPropertyFlags m\_MemoryFlags; **// Memory properties flags**

};

The GPU buffer for vertex and indices as stored in the Mesh data structure, it also stores the count of indices.

struct Mesh {

VulkanBuffer vertexBuffer;

VulkanBuffer indexBuffer;

uint32\_t indexCount;

} m\_Mesh;

**Assimp data structures:** We required two main data structure from Assimp – Importer and aiScene. The importer load the formatted mesh files (like .dae, .obj, .3ds, .fbx) into assimp’s aiScene. The aiScene provide a scene information of the mesh like how many mesh and submesh it contains and the information of each mesh including geometry, light and material etc.

Assimp::Importer m\_AssimpImporter;

const aiScene\* m\_pMeshScene;

## Implementation

The Load function accepts a mesh file name and load the formatted mesh using importer’s ReadFile() function, it import the mesh information into aiScene object, it also accept two additional attributes, the aiProcess\_Triangulate specifying triangulation of the geometry. When triangulation is done, each face is represented by three vertices and the information of these 3 vertices are stored in the indices. The another parameter aiProcess\_PreTransformVertices pre-transforms all vertices with the local transformation matrices of their nodes.

After reading the file the mesh information is loaded into m\_pMeshScene, containing the geometry and related attribute for each node. These nodes are further loaded into CPU data structures describes above using the LoadNode() function.

bool SimpleMesh::Load(const char\* p\_Filename)

{

m\_pMeshScene = m\_AssimpImporter.ReadFile(p\_Filename,

aiProcess\_Triangulate | aiProcess\_PreTransformVertices);

uint32\_t vertexCount = 0;

if (!m\_pMeshScene){ return false; **// Error** }

m\_Nodes.clear();

m\_Nodes.resize(m\_pMeshScene->mNumMeshes);

for (size\_t i = 0; i < m\_Nodes.size(); i++){

vertexCount += m\_pMeshScene->mMeshes[i]->mNumVertices;

LoadNode(&m\_Nodes[i], m\_pMeshScene->mMeshes[i]);

}

return true;

}

The load node function loads the vertex and indices into local MeshNode data structure object as shown below:-

void SimpleMesh::LoadNode(MeshNode\* p\_MeshNode, const aiMesh\* p\_pAiMesh)

{

**// Store mesh vertices**

p\_MeshNode->Vertices.reserve(p\_pAiMesh->mNumVertices);

for (unsigned int i = 0; i < p\_pAiMesh->mNumVertices; ++i){

const aiVector3D& position = p\_pAiMesh->mVertices[i];

p\_MeshNode->Vertices.push\_back

(Vertex(glm::vec3(position.x, -position.y, position.z)));

}

**// Store mesh indices**

p\_MeshNode->Indices.resize(p\_pAiMesh->mNumFaces \* 3);

for (unsigned int i = 0; i < p\_pAiMesh->mNumFaces; ++i){

const aiFace& Face = p\_pAiMesh->mFaces[i];

if (Face.mNumIndices != 3) continue;

p\_MeshNode->Indices[i \* 3 + 0] = Face.mIndices[0];

p\_MeshNode->Indices[i \* 3 + 1] = Face.mIndices[1];

p\_MeshNode->Indices[i \* 3 + 2] = Face.mIndices[2];

}

}

Now, we will load the information of each node from MeshNode list into the Mesh object m\_Mesh. This is done using LoadMesh() function as shown below, the vertex and index information is populated locally into vertexBuffer and indexBuffer, these are used to create GPU device local buffer m\_Mesh.vertexBuffer and m\_Mesh.indexBuffer.

The device local memory can be created in two ways – with or without staging. The former is an optimize way for buffer creation for use cases where the buffer does not change or changes very less frequently.

* When staging is specified (former way) two buffers are created, one on device and another on the host. The Vulkan command buffer are used to copy buffer of host buffer into device local buffer. For more information on source code kindly refer to the VulkanHelper::CreateStagingBuffer().
* In the latter way the created device local buffer is host visible, the host visible buffer allows to copy data from host to device using simple memcpy(). For this a mapping is made between the host and device, using the device mapped pointer the coping of buffer is performed. For more information on source code kindly refer to the VulkanHelper::CreateBuffer().

void SimpleMesh::LoadMesh(const char\* p\_Filename, bool p\_UseStaging)

{

Load(p\_Filename);

**// Populate vertex buffer**

std::vector<Vertex> vertexBuffer;

for (uint32\_t m = 0; m < m\_Nodes.size(); m++){

for (uint32\_t i = 0; i < m\_Nodes[m].Vertices.size(); i++){

Vertex vertex(m\_Nodes[m].Vertices[i].m\_Pos,

m\_Nodes[m].Vertices[i].m\_Normals);

vertexBuffer.push\_back(vertex);

}

}

**// Populate index buffer**

std::vector<uint32\_t> indexBuffer;

for (uint32\_t m = 0; m < m\_Nodes.size(); m++) {

for (size\_t i = 0; i < m\_Nodes[m].Indices.size(); i++) {

indexBuffer.push\_back(m\_Nodes[m].Indices[i]);

}

}

m\_Mesh.indexCount = static\_cast<uint32\_t>(indexBuffer.size());

const VkDevice device = m\_VulkanApplication->m\_hDevice;

VkPhysicalDeviceMemoryProperties memProp = m\_VulkanApplication->

m\_physicalDeviceInfo.memProp;

m\_Mesh.vertexBuffer.m\_DataSize =

vertexBuffer.size() \* sizeof(Vertex); **// Vertex buffer size**

m\_Mesh.indexBuffer.m\_DataSize =

indexBuffer.size() \* sizeof(uint32\_t); **// Index buffer size**

VkMemoryPropertyFlags memoryProperty = (p\_UseStaging ?

VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT :

VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT);

m\_Mesh.vertexBuffer.m\_MemoryFlags = memoryProperty;

m\_Mesh.indexBuffer.m\_MemoryFlags = memoryProperty;

**// Create vertex and index gpu buffers**

if (p\_UseStaging)

{

if (!m\_VulkanApplication->m\_hCommandPool) {

VulkanHelper::CreateCommandPool(device, m\_VulkanApplication->

m\_hCommandPool, m\_VulkanApplication->m\_physicalDeviceInfo);

}

VkCommandPool cmdPool = m\_VulkanApplication->m\_hCommandPool;

VkQueue queue = m\_VulkanApplication->m\_hGraphicsQueue;

**// Create vertex buffer using staging**

VulkanHelper::CreateStagingBuffer(device, memProp, cmdPool, queue,

m\_Mesh.vertexBuffer, VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT | VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT, vertexBuffer.data());

**// Create index buffer using staging**

VulkanHelper::CreateStagingBuffer(device, memProp, cmdPool, queue,

m\_Mesh.indexBuffer, VK\_BUFFER\_USAGE\_INDEX\_BUFFER\_BIT | VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT, indexBuffer.data());

}

else

{

**// Create vertex buffer**

VulkanHelper::CreateBuffer(device, memProp, m\_Mesh.vertexBuffer,

VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT, vertexBuffer.data());

**// Create index buffer**

VulkanHelper::CreateBuffer(device, memProp, m\_Mesh.indexBuffer,

VK\_BUFFER\_USAGE\_INDEX\_BUFFER\_BIT, indexBuffer.data());

}

**// Indicates the rate at which the information will be**

**// injected for vertex input.**

m\_VertexInputBinding.binding = 0;

m\_VertexInputBinding.inputRate = VK\_VERTEX\_INPUT\_RATE\_VERTEX;

m\_VertexInputBinding.stride = sizeof(Vertex);

**// The VkVertexInputAttribute interpreting the data.**

m\_VertexInputAttribute[0].binding = 0;

m\_VertexInputAttribute[0].location = 0;

m\_VertexInputAttribute[0].format = VK\_FORMAT\_R32G32B32\_SFLOAT;

m\_VertexInputAttribute[0].offset = offsetof(struct Vertex, m\_Pos);

}

Specify the vertex input binding information indicating the stride of each vertex so that the GPU can interpret the data from each vertex correctly. Also, we like to mention how the vertices would be consume whether it would per vertex or instance based.

In the vertex attributes, specify the binding, location (where to read this attribute from vertex), format of the vertex (RGB, RGBA etc.) and offset depicting where data reside within the each vertex from the start.

## Drawing index based Geometry

So far in this book we render geometry that contains vertex information using Vulkan drawing API vkCmdDraw(). In this example, we will be using vertex and indices and draw the mesh object using vkCmdDrawIndexed() API.

For index based geometry, when the command buffer are recorded in the render pass, the vertex and index buffer information are binded first with vkCmdBindVertexBuffers() and vkCmdBindIndexBuffers() respectively.

const VkDeviceSize offsets[1] = { 0 };

**// Bind mesh vertex buffer**

vkCmdBindVertexBuffers(m\_VulkanApplication->

m\_hCommandBufferList[i], 0, 1,

&m\_Mesh.vertexBuffer.m\_Buffer, offsets);

**// Bind mesh index buffer**

vkCmdBindIndexBuffer(m\_VulkanApplication->

m\_hCommandBufferList[i], m\_Mesh.indexBuffer.m\_Buffer,

0, VK\_INDEX\_TYPE\_UINT32);

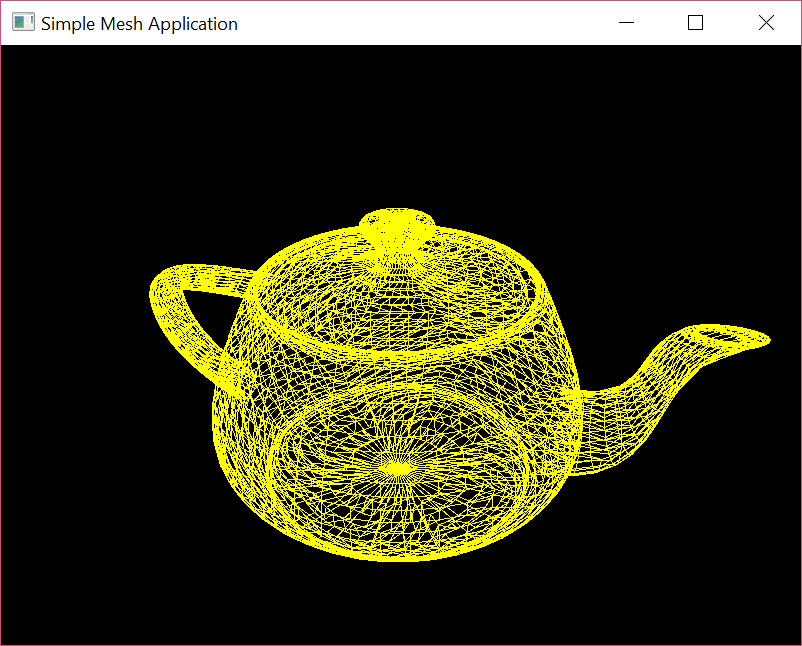
Next, the drawing is performed using vkCmdDrawIndexed, the first parameter is command buffer into which the recording is being performed. The second parameter specify the handle of the index buffer (VkBuffer) that will be bound to this API. The third parameter is an offset depicts start offset in bytes in the index buffer. The last field indicates whether the indices are 16-bits (VK\_INDEX\_TYPE\_UINT16) or 32-bits (VK\_INDEX\_TYPE\_UINT32) wide.

**// Render mesh vertex buffer using it's indices**

vkCmdDrawIndexed(m\_VulkanApplication->m\_hCommandBufferList[i],

m\_Mesh.indexCount, 1, 0, 0, 0);

Below is the output of the rendered mesh, we used VK\_PRIMITIVE\_-TOPOLOGY\_LINE\_LIST in the VkPipelineInputAssemblyStateCreateInfo of the graphics pipeline in order to render the geometry with line list topology.

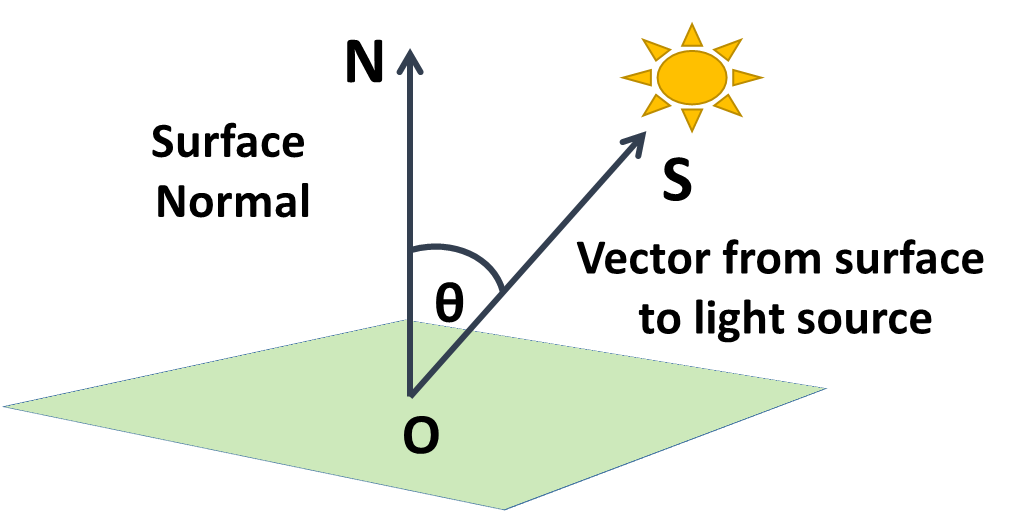


# Light shading

This section will take our previous implementation ahead bring little realism in the mesh application by adding a diffuse light shading on it. In the fourth coming examples, we will also show example with phong light model. Light and material are both colors. The color intensity associated with an object is called material and the color associated with illumination is called light. Mathematically, the reﬂected or resultant color is the product of light and material color.

In the diffuse light, the light rays come from a given source direction and after hitting through the surface it deflects with variable intensity based on the angle between the light direction and surface normal. The angle between the surface normal and the incident ray is calculated using lambert law.

*Lambert cosine law* states that the intensity of illumination on a diffuse surface is directly proportional to the cosine of the angle made by the surface normal vector and the direction of light.



The general equation for calculating the diffuse light is

Diffuse = Diffuse Light Intensity \* Diffuse Material color \* Light normal

The Light normal is the dot product between the surface normal and incident light vector and it gives the cosine of the angle between them.

## Extracting and using normals from mesh

In order to, use the normal in the mesh model, the ReadFile() function of Assimp’s Importer class needs to be specify with aiProcess\_GenSmoothNormalsparameter. This will make the aiScene contain smooth normals.

bool SimpleMesh::Load(const char\* p\_Filename) {

m\_pMeshScene = m\_AssimpImporter.ReadFile(p\_Filename,

aiProcess\_Triangulate | **aiProcess\_GenSmoothNormals**);

}

These normals can be store in the Vertex data structure which is now introduced with a new glm::vec3 element called m\_Normals.

struct Vertex{

glm::vec3 m\_Pos;

**glm::vec3 m\_Normals;**

};

These normals are populate in the LoadNode() similar to as we did for vertices in the last example as shown below:-

void SimpleMesh::LoadNode(MeshNode\* p\_MeshNode, aiMesh\* p\_pAiMesh) {

p\_MeshNode->Vertices.reserve(p\_pAiMesh->mNumVertices);

for (unsigned int i = 0; i < p\_pAiMesh->mNumVertices; ++i) {

const aiVector3D& position = p\_pAiMesh->mVertices[i];

**aiVector3D& normals = p\_pAiMesh->mNormals[i];**

p\_MeshNode->Vertices.push\_back(

Vertex(glm::vec3(position.x, -position.y, position.z),

**glm::vec3(normals.x, -normals.y, normals.z)**));

}

. . .

}

## Updating vertex shader

For computing the diffuse light shading in the vertex shader we use vertex position, vertex normal, light position and normal matrix. The inPosition is transformed into eye coordinate (by multiplying it with model-view matrix), similarly by taking the product of normal matrix (UBO.normalMatrix) and each normal inNormal we convert it into the eye coordinate. Like vertex coordinates are are transformed into eye coordinates using the ModelView matrix, the normals are transformed using Normal matrix.

The light vector nLight is calculated by subtracting eye coordinates of vertex position eyeCoord from UBO.lightPosition, the direction nLight is from the surface to the light source. The nLight and nNormal must normalized before taking dot product in order to find the cosine angle between them.

The light intensity is stored as the cosine angle between surface normal vector and light vector. The color information of the material and light is specified in global variables MaterialColor and LightColor, the product of these two variables is stored in the new variable diffuse. The cosine angle is calculated as the dot product of the nLight and nNormal and stored in the cosAngle variable.

We do not want the negative light intensities, as they do not make sense. Therefore, we should only consider light intensity within the range 0.0 and 1.0, for this reason the max() function is used in resultant light intensity (max( 0.0, dot( nNormal, nLight ));).

**// Filename: Light.vert**

#version 450

#extension GL\_ARB\_separate\_shader\_objects : enable

layout (std140, binding = 0) uniform bufferVals {

mat4 normalMatrix; **// Normal matrix**

mat4 projection; **// Projection matrix**

mat4 view; **// View matrix**

mat4 model; **// Model matrix**

vec4 lightPosition; **// Light position**

} UBO;

layout (location = 0) in vec4 inPosition; **// Position attribute**

layout (location = 1) in vec3 inNormal; **// Normals attribute**

layout (location = 0) out vec4 fragColor; **// Color attribute**

out gl\_PerVertex { vec4 gl\_Position; };

vec3 LightColor = vec3(0.6, 0.7, 0.4); **// Light Color**

vec3 MaterialColor = vec3(0.3, 0.6, 0.5); **// Material Color**

void main() {

vec4 pos = UBO.model \* inPosition;

vec3 nNormal = normalize(mat3(UBO.normalMatrix) \* inNormal);

vec3 eyeCoord= vec3(myBufferVals.view \* inPosition);

vec3 nLight = normalize(UBO.lightPosition.xyz - eyeCoord);

float cosAngle = max( 0.0, dot( nNormal, nLight ));

vec3 diffuse = MaterialColor \* LightColor;

fragColor = vec4(cosAngle \* diffuse, 1);

gl\_Position = UBO.projection \* UBO.view \*

UBO.model \* inPosition;

}

## Updating Uniforms

The uniform buffer object in this example consists of model, view, projection matrix. In addition to that it contains the light position and normal matrix.

The normal matrix is a sub matrix of the model view matrix, it preserves the normal of the geometry when affine transformation is applied. Mathematically, Normal matrix is the inverse transpose upper left 3x3 matrix of the model view matrix.

struct {

glm::mat4 NormalMatrix;

glm::mat4 Projection;

glm::mat4 View;

glm::mat4 Model;

glm::vec4 LightPosition;

} TransformationUniforms;

In the update() function, we will populate the uniform buffer per frame. The uniform structure object is filled with relavant field value and uploaded to the GPU memory with VulkanHelper’s WriteMemory() function.

void SimpleMesh::Update()

{

glm::mat4 normalMatrix = m\_Model \* (\*m\_View);

TransformationUniforms.NormalMatrix = glm::mat3x3(

glm::vec3(normalMatrix[0]),

glm::vec3(normalMatrix[1]),

glm::vec3(normalMatrix[2]));

TransformationUniforms.Projection = (\*m\_Projection);

TransformationUniforms.View = (\*m\_View);

TransformationUniforms.Model = m\_Model;

static int radius = 100;

static int numSegments = 500;

static float i = 0;

float theta = 2.0f \* 3.14 \* ((i > numSegments) ? i = 0 :

i+=0.4) / numSegments; **// Get the current angle**

TransformationUniforms.LightPosition.x = radius \* cosf(theta); **// Calculate the X component**

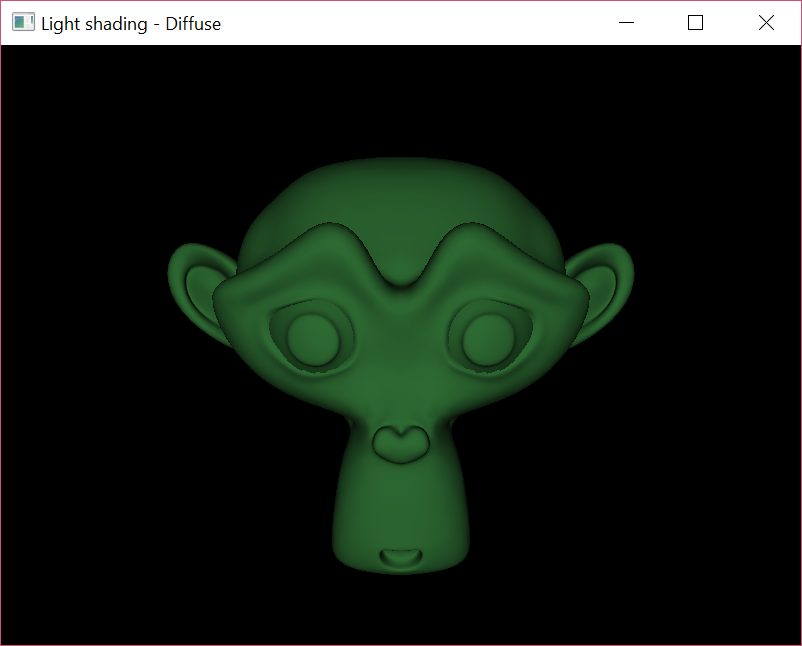
TransformationUniforms.LightPosition.z = radius \* sinf(theta); **// Calculate the Z component**

VulkanHelper::WriteMemory(m\_VulkanApplication->m\_hDevice,

UniformBuffer.m\_MappedMemory, UniformBuffer.m\_MappedRange, UniformBuffer.m\_BufObj.m\_MemoryFlags, &TransformationUniforms, sizeof(TransformationUniforms));

}

Below is the output of the diffuse light shading:-



# Summary

In this chapter, we introduced our readers with a number of basic 3D graphics example in Vulkan. We kicked off with building a simple 3D cube example and added 3D transformation on it. We demonstrated the artefacts of rendering 3D geometry object in the absence to depth testing and fix it with implementation of depth testing in Vulkan.

Next, we dive deep into render 3D mesh models and build a minimal example using Assimp library, we took this example further and bring life into it by adding light shading.

<Todo for Selva> Give some preface to next chapter.