**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Компьютерная графика»

**Лабораторная работа № 6**

Тема: Создание шейдерных анимационных эффектов в OpenGL

Студент: Ильин Илья Олегович

Группа: 80-306

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2021

1. Постановка задачи

Для поверхности, созданной в л.р. №4-5, обеспечить выполнение следующего шейдерного эффекта:

**Вариант 13:**  Анимация. Прозрачность изменяется по синусоидальному закону

1. Описание программы

Для реализации анимации был написан класс *NodeAnimation*, который хранит два замыкания: одно для анимации материала *Node*, второе для анимации *modelMatrix* т.е различных вращений в собственной системе координат объекта. Шейдер фрагментов был модифицирован, чтобы работать с 4 составляющей цвета - прозрачностью. Класс *Renderer* при каждом цикле отрисовки фигуры вызывает метод *update* объекта *Node*, объект по необходимости пересчитывает собственные точки и применяет заданные ранее анимации.

1. Набор тестов

Тест 1: изначальное положение фигуры, начало анимации

Тест 2: сдвиг фигуры, конец анимации появления

1. Результаты выполнения тестов

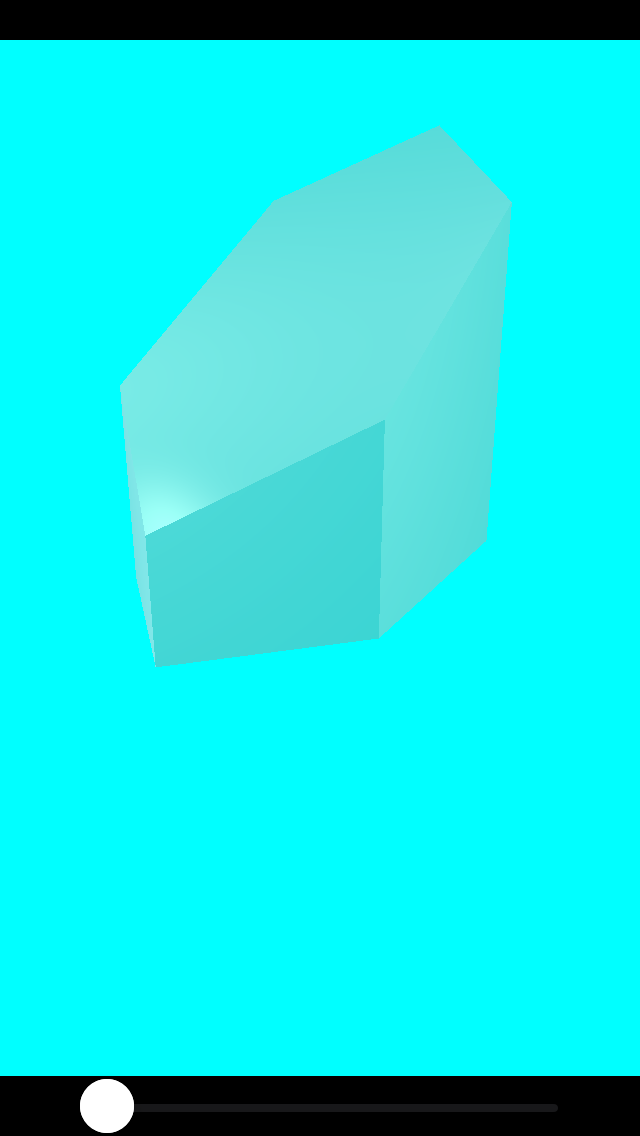
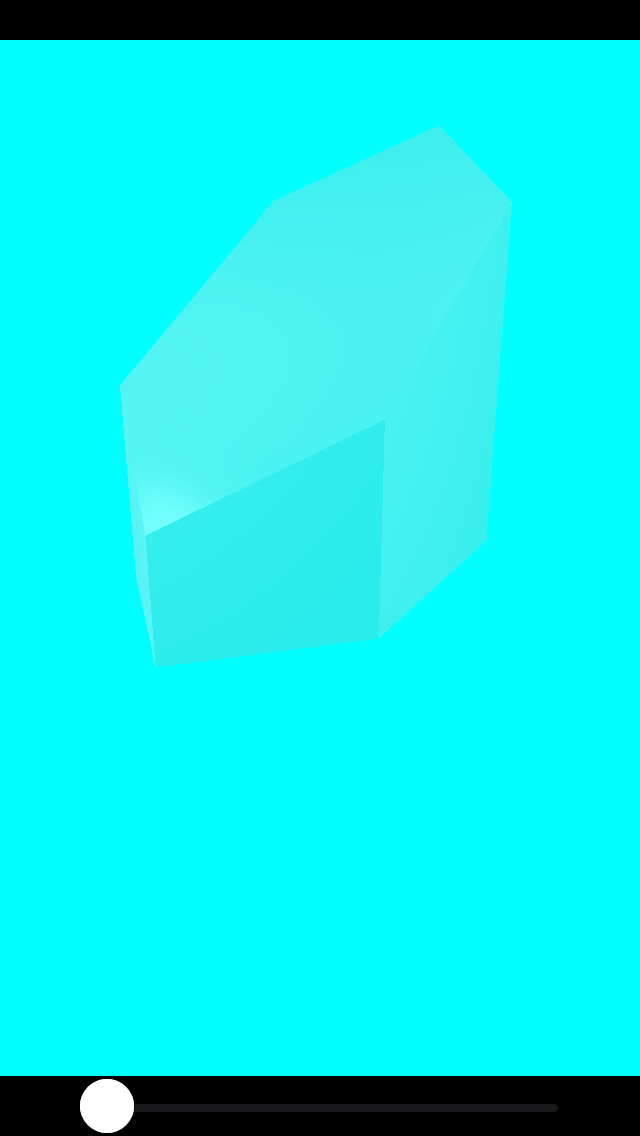


Рис. 1. Тест 1.

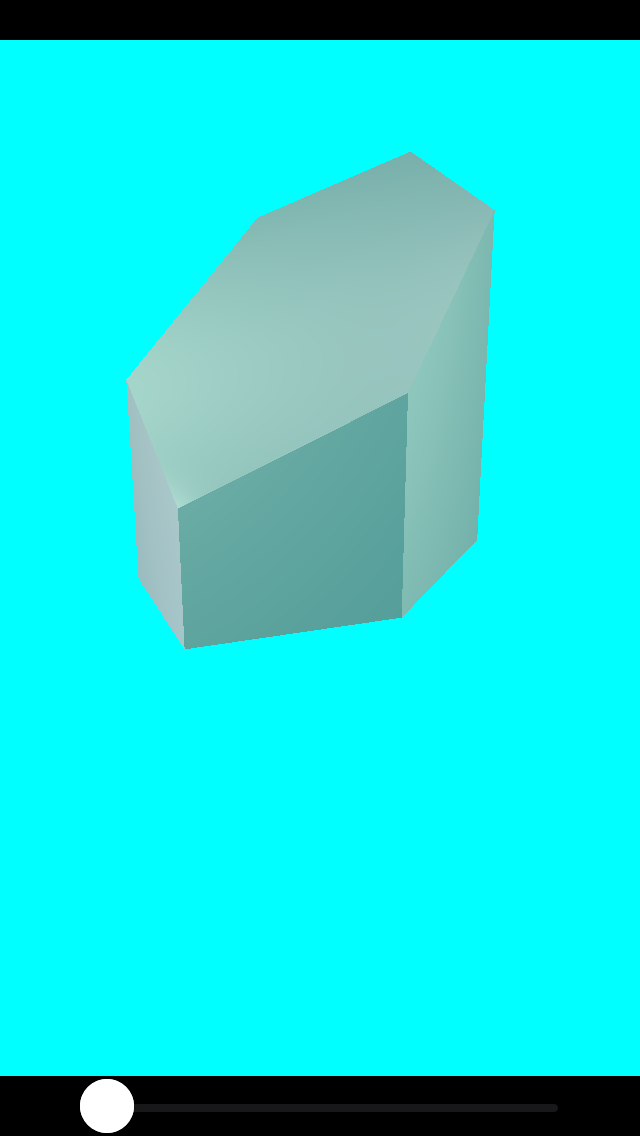
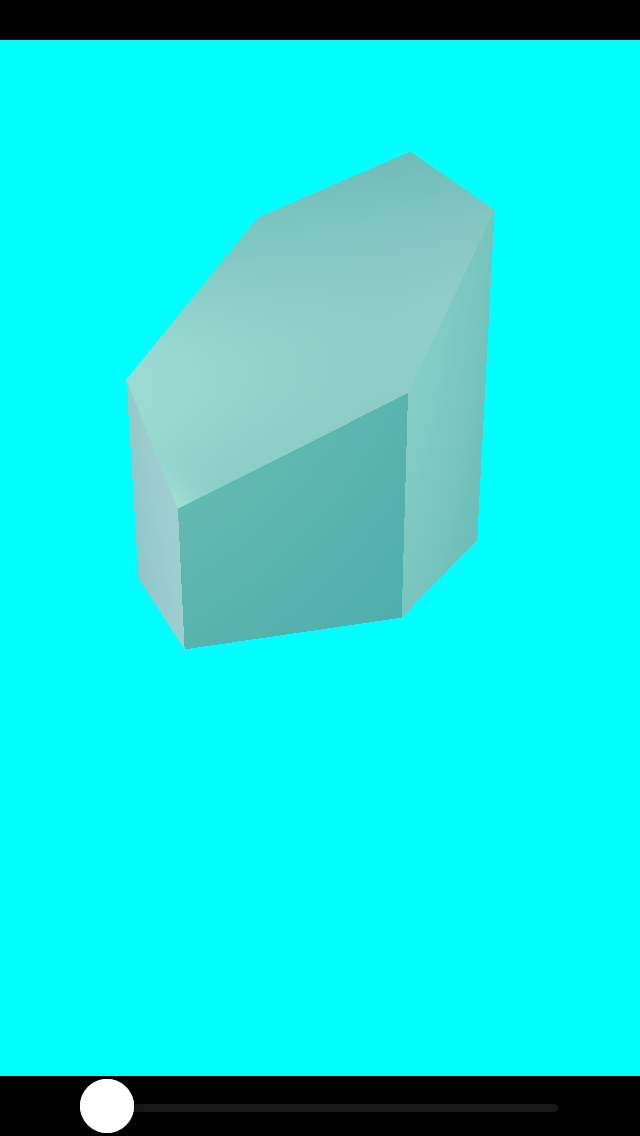


Рис. 2. Тест 2.

1. Листинг программы

ViewController.swift

//

// ViewController.swift

// CG6

//

// Created by Илья Ильин on 16.12.2021.

//

/\*

Для поверхности, созданной в л.р. №4-5, обеспечить выполнение следующего шейдерного эффекта:

Вариант 13: Анимация. Прозрачность изменяется по синусоидальному закону

\*/

**import** UIKit

**import** MetalKit

**class** ViewController: MetalViewController {

**var** approximation: Float = 1.0 {

**didSet** {

renderer.approximation = **self**.approximation

}

}

// **MARK: Sliders and gestures**

**var** deltaSlider: UISlider! /// from 0.001 to 1

**let** panSensivity: Float = 5.0

**let** pinchSensivity: Float = 0.5

**var** lastPanLocation: CGPoint!

// **MARK: - UIViewController**

**override** **func** viewDidLoad() {

setupViews()

setupScene()

**super**.viewDidLoad()

setupGestures()

}

// **MARK: - Scene setup**

**func** setupScene() {

**self**.scene = Scene()

**self**.scene.ambientLightColor = simd\_float3(0.01, 0.01, 0.01)

**let** light0 = Light(worldPosition: simd\_float3( 2, 2, 2), color: simd\_float3(1.0, 0.8, 0.8))

**let** light1 = Light(worldPosition: simd\_float3(-2, 2, 2), color: simd\_float3(0.8, 1.0, 0.8))

**let** light2 = Light(worldPosition: simd\_float3( 0, -2, 2), color: simd\_float3(0.8, 0.8, 1.0))

**self**.scene.lights = [ light0, light1, light2 ]

**let** animation = NodeAnimation(materialAnimation: { material, time **in**

**var** newMaterial = material

newMaterial.color.w = sin(time)

**return** newMaterial

}, modelAnimation: **nil**)

**let** material = Material(specularColor: simd\_float3(0.8, 0.8, 0.8),

specularPower: 100,

color: simd\_float4(0.5, 0.5, 0.5, 1.0))

**let** cylinder = TiltCylinder(name: "Cylinder", material: material, a: 2, b: 1, delta: approximation)

cylinder.animation = animation

**self**.scene.rootNode.children.append(cylinder)

}

// **MARK: - Views setup**

**func** setupViews() {

view.backgroundColor = .black

mtkView = MTKView()

mtkView.clearColor = MTLClearColor(red: 0, green: 1, blue: 1, alpha: 1)

mtkView.isOpaque = **false**

mtkView.translatesAutoresizingMaskIntoConstraints = **false**

view.addSubview(mtkView)

deltaSlider = UISlider()

deltaSlider.value = 1

deltaSlider.maximumValue = 1000

deltaSlider.minimumValue = 1

deltaSlider.addTarget(**self**, action: **#selector**(**self**.deltaValueChanged(\_:)), for: .valueChanged)

deltaSlider.translatesAutoresizingMaskIntoConstraints = **false**

view.addSubview(deltaSlider)

addConstraints()

}

**func** addConstraints() {

NSLayoutConstraint.activate([

mtkView.topAnchor.constraint(equalTo: view.safeAreaLayoutGuide.topAnchor),

mtkView.leadingAnchor.constraint(equalTo: view.safeAreaLayoutGuide.leadingAnchor),

mtkView.trailingAnchor.constraint(equalTo: view.safeAreaLayoutGuide.trailingAnchor),

mtkView.bottomAnchor.constraint(equalTo: deltaSlider.topAnchor),

deltaSlider.leadingAnchor.constraint(equalTo: view.safeAreaLayoutGuide.leadingAnchor, constant: 40),

deltaSlider.trailingAnchor.constraint(equalTo: view.safeAreaLayoutGuide.trailingAnchor, constant: -40),

deltaSlider.bottomAnchor.constraint(equalTo: view.safeAreaLayoutGuide.bottomAnchor),

])

}

**@objc** **func** deltaValueChanged(\_ sender: UISlider!) {

approximation = (sender.maximumValue + 1 - sender.value) / sender.maximumValue

}

//**MARK: - Gesture related**

**func** setupGestures() {

**let** pan = UIPanGestureRecognizer(target: **self**, action: **#selector**(ViewController.pan(\_:)))

**let** pinch = UIPinchGestureRecognizer(target: **self**, action: **#selector**(ViewController.pinch(\_:)))

**self**.view.addGestureRecognizer(pan)

**self**.view.addGestureRecognizer(pinch)

}

**@objc** **func** pan(\_ sender: UIPanGestureRecognizer) {

**if** sender.state == UIGestureRecognizer.State.changed {

**let** pointInView = sender.location(in: **self**.view)

**let** xDelta = Float((lastPanLocation.x - pointInView.x)/**self**.view.bounds.width) \* panSensivity

**let** yDelta = Float((lastPanLocation.y - pointInView.y)/**self**.view.bounds.height) \* panSensivity

renderer.rotationX -= xDelta

renderer.rotationY -= yDelta

lastPanLocation = pointInView

} **else** **if** sender.state == UIGestureRecognizer.State.began {

lastPanLocation = sender.location(in: **self**.view)

}

}

**@objc** **func** pinch(\_ sender: UIPinchGestureRecognizer) {

**let** newScale = Float(sender.scale) \* pinchSensivity

renderer.scale = newScale

}

**@objc** **private** **func** startZooming(\_ sender: UIPinchGestureRecognizer) {

**let** scaleResult = sender.view?.transform.scaledBy(x: sender.scale, y: sender.scale)

**guard** **let** scale = scaleResult, scale.a > 1, scale.d > 1 **else** { **return** }

sender.view?.transform = scale

sender.scale = 1

}

}

MetalViewController.swift

**import** UIKit

**import** MetalKit

**class** MetalViewController: UIViewController {

**var** mtkView: MTKView!

**var** renderer: Renderer!

**var** scene: Scene!

**override** **func** viewDidLoad() {

**super**.viewDidLoad()

**let** device = MTLCreateSystemDefaultDevice()!

mtkView.device = device

mtkView.colorPixelFormat = .bgra8Unorm\_srgb

mtkView.depthStencilPixelFormat = .depth32Float

renderer = Renderer(view: mtkView, device: device, scene: scene)

mtkView.delegate = renderer

}

}

Node.swift

**import** Foundation

**import** MetalKit

**import** simd

**struct** Material {

**var** specularColor = simd\_float3(1, 1, 1)

**var** specularPower = Float(1)

**var** color: simd\_float4 = simd\_float4(0, 0, 0, 1)

}

**struct** NodeAnimation {

**var** materialAnimation: ( (Material, Float) -> (Material) )?

**var** modelAnimation: ( (float4x4, Float) -> (float4x4) )?

}

**protocol** NodeDelegate : AnyObject {

**func** createMesh(node: String, vertices: [Vertex], indices: [[UInt16]]) -> MTKMesh?

}

**class** Node {

// **MARK: Node structure**

**private**(**set**) **var** name: String

**weak** **var** parent: Node?

**var** children = [Node]()

// **MARK: Delegate**

**weak** **var** delegate: NodeDelegate?

// **MARK: Node drawing properties**

**private** **var** needRecount: Bool = **true**

**private**(**set**) **var** time: Float = 0.0

**private**(**set**) **var** approximation: Float {

**didSet** {

**if** oldValue != approximation {

needRecount = **true**

}

}

}

**public** **var** modelMatrix = matrix\_identity\_float4x4

**private**(**set**) **var** mesh: MTKMesh?

**public** **internal**(**set**) **var** material: Material

**public** **internal**(**set**) **var** vertices = [Vertex]()

**public** **internal**(**set**) **var** indices = [[UInt16]]()

**var** animation: NodeAnimation?

// **MARK: Initialization**

**init**(name: String, material: Material, approximation: Float) {

**self**.material = material

**self**.approximation = approximation

**self**.name = name

}

**convenience** **init**(name: String) {

**self**.**init**(name: name, material: Material(), approximation: 1.0)

}

// **MARK: Interface**

**final** **func** update(time: Float, approximation: Float) {

**self**.time = time

**self**.approximation = approximation

updateLogic()

}

**final** **func** nodeNamedRecursive(\_ name: String) -> Node? {

**for** node **in** children {

**if** node.name == name {

**return** node

} **else** **if** **let** matchingGrandchild = node.nodeNamedRecursive(name) {

**return** matchingGrandchild

}

}

**return** **nil**

}

**final** **func** getChildrenNodes() -> [Node] {

**var** nodes = [**self**]

**for** child **in** children {

nodes += child.getChildrenNodes()

}

**return** nodes

}

// **MARK: Internal methods**

/// Counts nodes vertices, needs to be overriden and set verices and indices

**internal** **func** countVertices() {}

// **MARK: Private methods**

**private** **func** updateMesh() {

mesh = delegate?.createMesh(node: name, vertices: vertices, indices: indices)

}

/// Called when we need to update figure mesh due to approximation changes

**private** **func** updateLogic(){

**var** needMeshUpdate = **false**

**if** needRecount {

countVertices()

needRecount = **false**

needMeshUpdate = **true**

}

material = animation?.materialAnimation?(material, time) ?? material

modelMatrix = animation?.modelAnimation?(modelMatrix, time) ?? modelMatrix

**if** needMeshUpdate {

updateMesh()

}

}

}

Renderer.swift

**import** Foundation

**import** MetalKit

**import** simd

**struct** VertexUniforms {

**var** viewProjectionMatrix: float4x4

**var** modelMatrix: float4x4

**var** normalMatrix: float3x3

}

**struct** FragmentUniforms {

**var** cameraWorldPosition = simd\_float3(0, 0, 0)

**var** ambientLightColor = simd\_float3(0, 0, 0)

**var** specularColor = simd\_float3(1, 1, 1)

**var** specularPower = Float(1)

**var** materialColor = simd\_float4(1, 1, 1, 1)

**var** light0 = Light()

**var** light1 = Light()

**var** light2 = Light()

}

// **MARK: - Renderer**

**class** Renderer: NSObject {

**let** device: MTLDevice

**let** commandQueue: MTLCommandQueue

**var** renderPipeline: MTLRenderPipelineState

**let** depthStencilState: MTLDepthStencilState

**let** vertexDescriptor: MDLVertexDescriptor

**let** bufferAllocator: MTKMeshBufferAllocator

**var** time: Float = 0

**let** scene: Scene

**var** cameraWorldPosition = simd\_float3(0, 0, 2)

**var** viewMatrix = matrix\_identity\_float4x4

**var** projectionMatrix = matrix\_identity\_float4x4

**var** approximation: Float = 1.0

**var** rotationX: Float = 0.0

**var** rotationY: Float = 0.0

**var** rotationZ: Float = 0.0

**var** scale: Float = 1.0

// **MARK: - Initialization**

**init**(view: MTKView, device: MTLDevice, scene: Scene) {

**self**.device = device

commandQueue = device.makeCommandQueue()!

vertexDescriptor = Renderer.buildVertexDescriptor()

renderPipeline = Renderer.buildPipeline(device: device, view: view, vertexDescriptor: vertexDescriptor)

depthStencilState = Renderer.buildDepthStencilState(device: device)

**self**.bufferAllocator = MTKMeshBufferAllocator(device: device)

**self**.scene = scene

**super**.init()

setupNodes()

}

**private** **func** setupNodes() {

**for** node **in** scene.nodes {

node.delegate = **self**

node.update(time: time, approximation: approximation)

}

}

**private** **static** **func** buildVertexDescriptor() -> MDLVertexDescriptor {

**let** vertexDescriptor = MDLVertexDescriptor()

vertexDescriptor.attributes[0] = MDLVertexAttribute(name: MDLVertexAttributePosition,

format: .float3,

offset: 0,

bufferIndex: 0)

vertexDescriptor.attributes[1] = MDLVertexAttribute(name: MDLVertexAttributeNormal,

format: .float3,

offset: MemoryLayout<Float>.stride \* 3,

bufferIndex: 0)

vertexDescriptor.layouts[0] = MDLVertexBufferLayout(stride: MemoryLayout<Vertex>.stride)

**return** vertexDescriptor

}

**private** **static** **func** buildDepthStencilState(device: MTLDevice) -> MTLDepthStencilState {

**let** depthStencilDescriptor = MTLDepthStencilDescriptor()

depthStencilDescriptor.depthCompareFunction = .less

depthStencilDescriptor.isDepthWriteEnabled = **true**

**return** device.makeDepthStencilState(descriptor: depthStencilDescriptor)!

}

**private** **static** **func** buildPipeline(device: MTLDevice, view: MTKView, vertexDescriptor: MDLVertexDescriptor) -> MTLRenderPipelineState {

**guard** **let** library = device.makeDefaultLibrary() **else** {

fatalError("Could not load default library from main bundle")

}

**let** vertexFunction = library.makeFunction(name: "vertex\_main")

**let** fragmentFunction = library.makeFunction(name: "fragment\_main")

**let** pipelineDescriptor = MTLRenderPipelineDescriptor()

pipelineDescriptor.vertexFunction = vertexFunction

pipelineDescriptor.fragmentFunction = fragmentFunction

pipelineDescriptor.colorAttachments[0].pixelFormat = view.colorPixelFormat

pipelineDescriptor.colorAttachments[0].isBlendingEnabled = **true**

pipelineDescriptor.colorAttachments[0].rgbBlendOperation = .add

pipelineDescriptor.colorAttachments[0].alphaBlendOperation = .add

pipelineDescriptor.colorAttachments[0].sourceRGBBlendFactor = .sourceAlpha

pipelineDescriptor.colorAttachments[0].sourceAlphaBlendFactor = .sourceAlpha

pipelineDescriptor.colorAttachments[0].destinationRGBBlendFactor = .oneMinusSourceAlpha

pipelineDescriptor.colorAttachments[0].destinationAlphaBlendFactor = .oneMinusSourceAlpha

pipelineDescriptor.depthAttachmentPixelFormat = view.depthStencilPixelFormat

**let** mtlVertexDescriptor = MTKMetalVertexDescriptorFromModelIO(vertexDescriptor)

pipelineDescriptor.vertexDescriptor = mtlVertexDescriptor

**do** {

**return** **try** device.makeRenderPipelineState(descriptor: pipelineDescriptor)

} **catch** {

fatalError("Could not create render pipeline state object: \(error)")

}

}

// **MARK: - Drawing**

**func** updateSceneLogic() {

**for** node **in** scene.nodes {

node.update(time: time, approximation: approximation)

}

}

**func** update(\_ view: MTKView) {

time += 1 / Float(view.preferredFramesPerSecond)

cameraWorldPosition = simd\_float3(0, 0, 6)

viewMatrix = float4x4(translationBy: -cameraWorldPosition)

**let** aspectRatio = Float(view.drawableSize.width / view.drawableSize.height)

projectionMatrix = float4x4(perspectiveProjectionFov: Float.pi / 3, aspectRatio: aspectRatio, nearZ: 0.1, farZ: 100)

scene.rootNode.modelMatrix = float4x4(rotationAbout: simd\_float3(1, 0, 0), by: rotationY) \* float4x4(rotationAbout: simd\_float3(0, 1, 0), by: rotationX) \* float4x4(scaleBy: scale)

updateSceneLogic()

}

**func** drawScene(commandEncoder: MTLRenderCommandEncoder) {

drawNodeRecursive(scene.rootNode, parentTransform: matrix\_identity\_float4x4, commandEncoder: commandEncoder)

}

**func** drawNodeRecursive(\_ node: Node, parentTransform: float4x4, commandEncoder: MTLRenderCommandEncoder) {

**let** modelMatrix = parentTransform \* node.modelMatrix

**if** **let** mesh = node.mesh {

**let** viewProjectionMatrix = projectionMatrix \* viewMatrix

**var** vertexUniforms = VertexUniforms(viewProjectionMatrix: viewProjectionMatrix,

modelMatrix: modelMatrix,

normalMatrix: modelMatrix.normalMatrix)

commandEncoder.setVertexBytes(&vertexUniforms, length: MemoryLayout<VertexUniforms>.size, index: 1)

**var** fragmentUniforms = FragmentUniforms(cameraWorldPosition: cameraWorldPosition,

ambientLightColor: scene.ambientLightColor,

specularColor: node.material.specularColor,

specularPower: node.material.specularPower,

materialColor: node.material.color,

light0: scene.lights[0],

light1: scene.lights[1],

light2: scene.lights[2])

commandEncoder.setFragmentBytes(&fragmentUniforms, length: MemoryLayout<FragmentUniforms>.size, index: 0)

**let** vertexBuffer = mesh.vertexBuffers.first!

commandEncoder.setVertexBuffer(vertexBuffer.buffer, offset: vertexBuffer.offset, index: 0)

**for** submesh **in** mesh.submeshes {

**let** indexBuffer = submesh.indexBuffer

commandEncoder.drawIndexedPrimitives(type: submesh.primitiveType,

indexCount: submesh.indexCount,

indexType: submesh.indexType,

indexBuffer: indexBuffer.buffer,

indexBufferOffset: indexBuffer.offset)

}

}

**for** child **in** node.children {

drawNodeRecursive(child, parentTransform: modelMatrix, commandEncoder: commandEncoder)

}

}

}

// **MARK: - NodeDelegate**

**extension** Renderer : NodeDelegate {

**internal** **func** createMesh(node: String, vertices: [Vertex], indices: [[UInt16]]) -> MTKMesh? {

**if** vertices.isEmpty || indices.isEmpty { **return** **nil** }

**let** vertexBuffer = bufferAllocator.newBuffer(MemoryLayout<Vertex>.stride \* vertices.count, type: .vertex)

**let** vertexMap = vertexBuffer.map()

vertexMap.bytes.assumingMemoryBound(to: Vertex.**self**).assign(from: vertices, count: vertices.count)

**var** submeshes = [MDLSubmesh]()

**for** indices **in** indices {

**let** indexBuffer = bufferAllocator.newBuffer(MemoryLayout<UInt16>.stride \* indices.count, type: .index)

**let** indexMap = indexBuffer.map()

indexMap.bytes.assumingMemoryBound(to: UInt16.**self**).assign(from: indices, count: indices.count)

**let** submesh = MDLSubmesh(indexBuffer: indexBuffer,

indexCount: indices.count,

indexType: .uInt16,

geometryType: .triangles,

material: **nil**)

submeshes.append(submesh)

}

**let** mdlMesh = MDLMesh(vertexBuffer: vertexBuffer,

vertexCount: vertices.count,

descriptor: vertexDescriptor,

submeshes: submeshes)

**let** mesh: MTKMesh?

**do** {

mesh = **try** MTKMesh(mesh: mdlMesh, device: device)

} **catch** {

mesh = **nil**

NSLog("Couldn't create mesh of node %s", node)

}

**return** mesh

}

}

// **MARK: - MTKViewDelegate**

**extension** Renderer: MTKViewDelegate {

**func** mtkView(\_ view: MTKView, drawableSizeWillChange size: CGSize) {

}

**func** draw(in view: MTKView) {

update(view)

**let** commandBuffer = commandQueue.makeCommandBuffer()!

**if** **let** renderPassDescriptor = view.currentRenderPassDescriptor, **let** drawable = view.currentDrawable {

**let** commandEncoder = commandBuffer.makeRenderCommandEncoder(descriptor: renderPassDescriptor)!

commandEncoder.setFrontFacing(.counterClockwise)

commandEncoder.setCullMode(.back)

commandEncoder.setDepthStencilState(depthStencilState)

commandEncoder.setRenderPipelineState(renderPipeline)

drawScene(commandEncoder: commandEncoder)

commandEncoder.endEncoding()

commandBuffer.present(drawable)

commandBuffer.commit()

}

}

}

Shaders.swift

//

// Shaders.metal

// CG3-4

//

// Created by Илья Ильин on 01.11.2021.

//

#include <metal\_stdlib>

**using** **namespace** metal;

**constant** **float** alphaTestReferenceValue = 0.0;

**struct** VertexIn {

float3 position [[attribute(0)]];

float3 normal [[attribute(1)]];

};

**struct** VertexOut {

float4 position [[position]];

float3 worldNormal;

float3 worldPosition;

};

**struct** Light {

float3 worldPosition;

float3 color;

};

**struct** VertexUniforms {

float4x4 viewProjectionMatrix;

float4x4 modelMatrix;

float3x3 normalMatrix;

};

#define LightCount 3

**struct** FragmentUniforms {

float3 cameraWorldPosition;

float3 ambientLightColor;

float3 specularColor;

**float** specularPower;

float4 materialColor;

Light lights[LightCount];

};

**vertex** VertexOut vertex\_main(VertexIn vertexIn [[stage\_in]],

**constant** VertexUniforms &uniforms [[buffer(1)]])

{

VertexOut vertexOut;

float4 worldPosition = uniforms.modelMatrix \* float4(vertexIn.position, 1);

vertexOut.position = uniforms.viewProjectionMatrix \* worldPosition;

vertexOut.worldPosition = worldPosition.xyz;

vertexOut.worldNormal = uniforms.normalMatrix \* vertexIn.normal;

**return** vertexOut;

}

**fragment** half4 fragment\_main(VertexOut fragmentIn [[stage\_in]],

**constant** FragmentUniforms &uniforms [[buffer(0)]])

{

float3 baseColor = float3(uniforms.materialColor);

**float** alpha = uniforms.materialColor.w;

**if** (alpha < alphaTestReferenceValue) {

discard\_fragment();

}

float3 specularColor = uniforms.specularColor;

float3 N = normalize(fragmentIn.worldNormal);

float3 V = normalize(uniforms.cameraWorldPosition - fragmentIn.worldPosition);

float3 finalColor(0, 0, 0);

**for** (**int** i = 0; i < LightCount; ++i) {

float3 L = normalize(uniforms.lights[i].worldPosition - fragmentIn.worldPosition.xyz);

float3 diffuseIntensity = saturate(dot(N, L));

float3 H = normalize(L + V);

**float** specularBase = saturate(dot(N, H));

**float** specularIntensity = powr(specularBase, uniforms.specularPower);

float3 lightColor = uniforms.lights[i].color;

finalColor += uniforms.ambientLightColor \* baseColor +

diffuseIntensity \* lightColor \* baseColor +

specularIntensity \* lightColor \* specularColor;

}

**return** half4(float4(finalColor, alpha));

}

1. Выводы

В ходе выполнения данной ЛР я глубже познакомился с шейдерами, научился работать с цветом и модифицировал существующий проект, теперь анимации материала и вращения можно применять к любому объекту наследованному от *Node*.

ЛИТЕРАТУРА

1. Справка по фреймворку Metal [Электронный ресурс]URL: <https://metalbyexample.com>

(Дата обращения: 03.11.2021).