**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Компьютерная графика»

**Лабораторная работа № 4-5**

Тема: Ознакомление с технологией OpenGL

Студент: Ильин Илья Олегович

Группа: 80-306

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2021

1. Постановка задачи

Создать графическое приложение с использованием OpenGL. Изобразить заданное тело с использованием средств OpenGL 2.1 (или выше).

Использовать буфер вершин. Точность аппроксимации тела задается пользователем. Обеспечить возможность вращения и масштабирования многогранника и удаление невидимых линий и поверхностей.

Реализовать простую модель освещения на GLSL. Параметры освещения и отражающие свойства материала задаются пользователем в диалоговом режиме.

1. Описание программы

Класс Renderer отвечает за рендер объекта в MTKView, использует Metal Kit. В Scene.swift реализованы классы и структуры, представляющие сцену, ее объекты, материалы объектов и освещение.

Модифицированный класс TiltCylinder теперь работает с Metal. Вместо вершин, он создаёт свой меш. Для уменьшения затрат по памяти и дальнейшего ускорения производительности будем переиспользовать вершины: меш будет содержать буфер вершин, а также сабмеши, которые содержат индексы вершин из буфера.

ViewController ответственен за создание класса Renderer и отрисовку на экране. Точность аппроксимации задается слайдером, при изменении его значения меняется значение точности отрисовки класса Renderer, за которым следует пересчёт мешей фигур сцены.

Для более наглядного эффекта было добавлено 3 источника света разных цветов, сам объект имеет металлический блеск. Используется модель освещения Фонга.

1. Результаты выполнения тестов

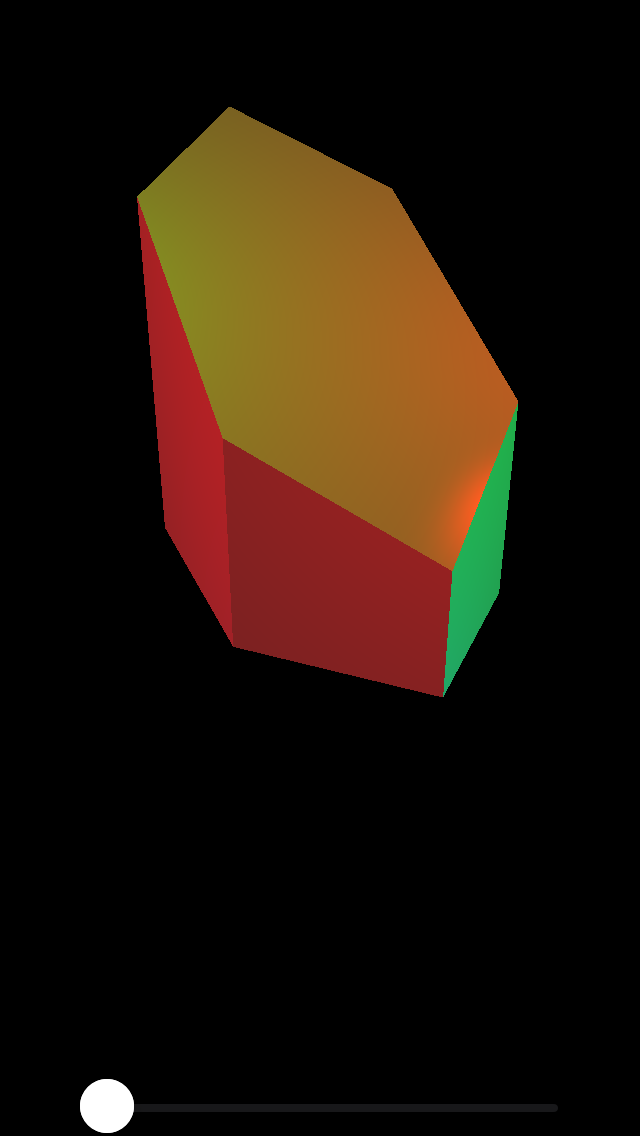


Рис. 1 Изначальный вид

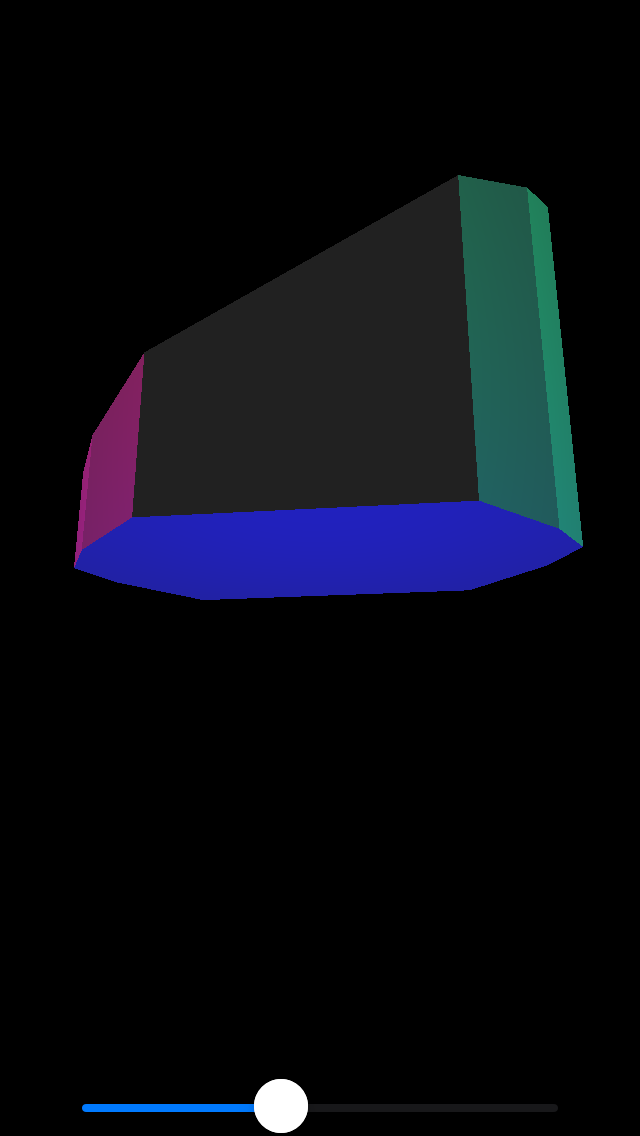
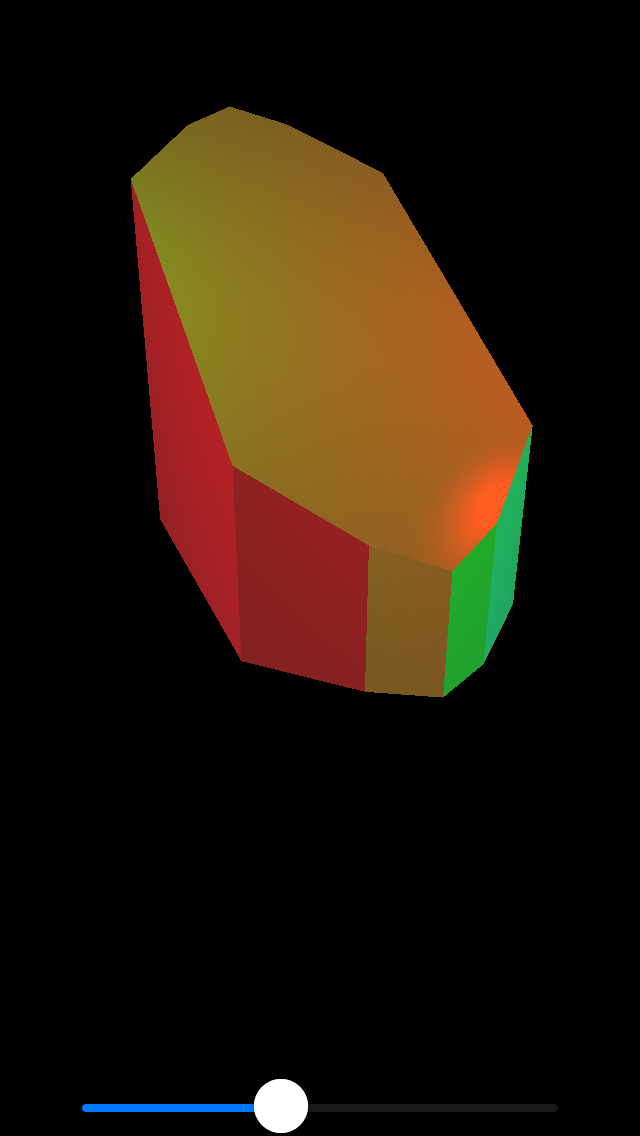


Рис. 2 Низкая точность аппроксимации, взгляд на освещение под разными углами

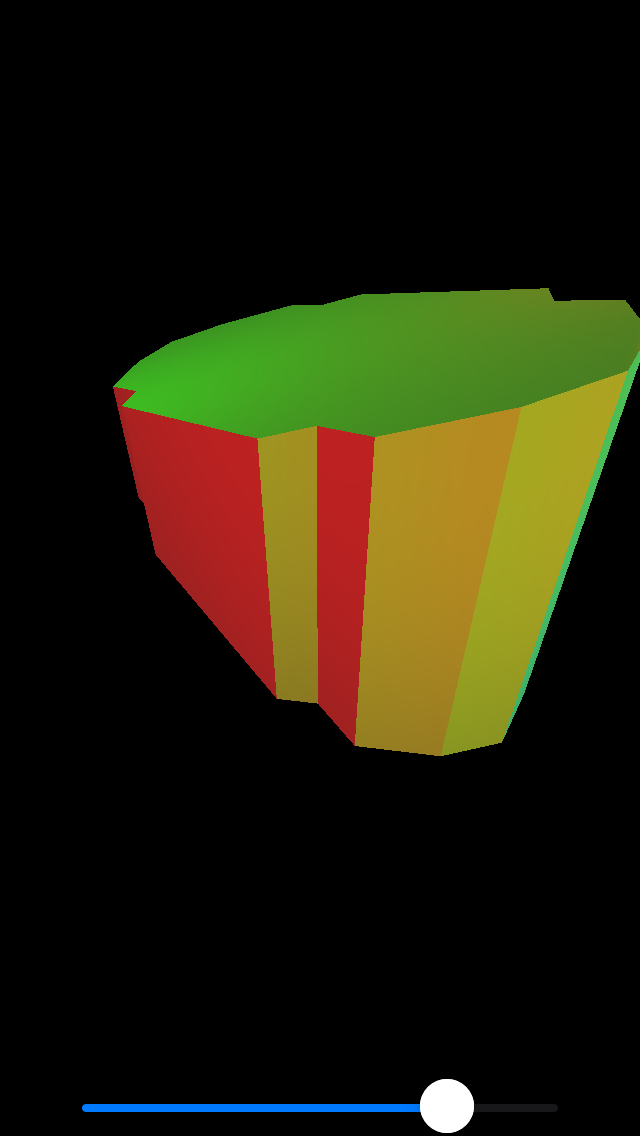
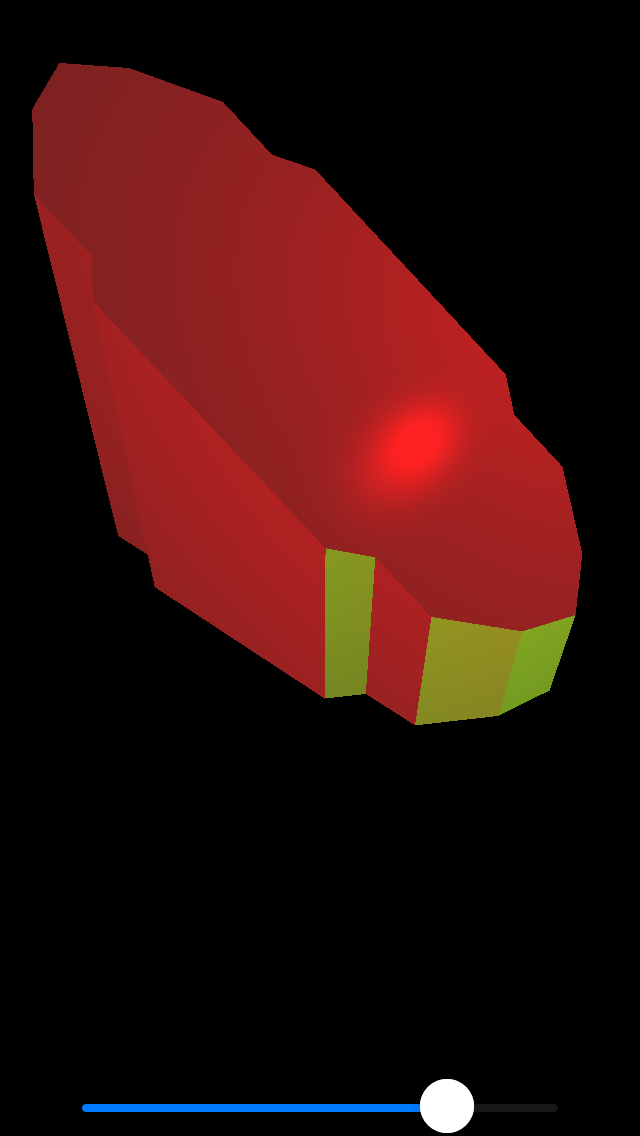


Рис. 3 Средняя точность апроксимации

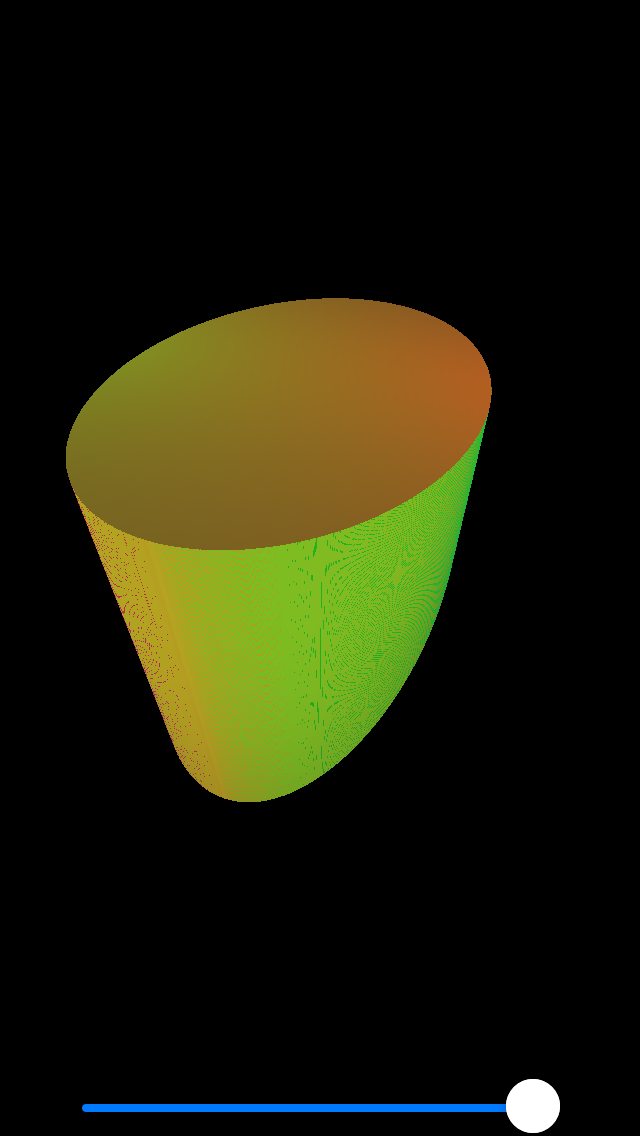
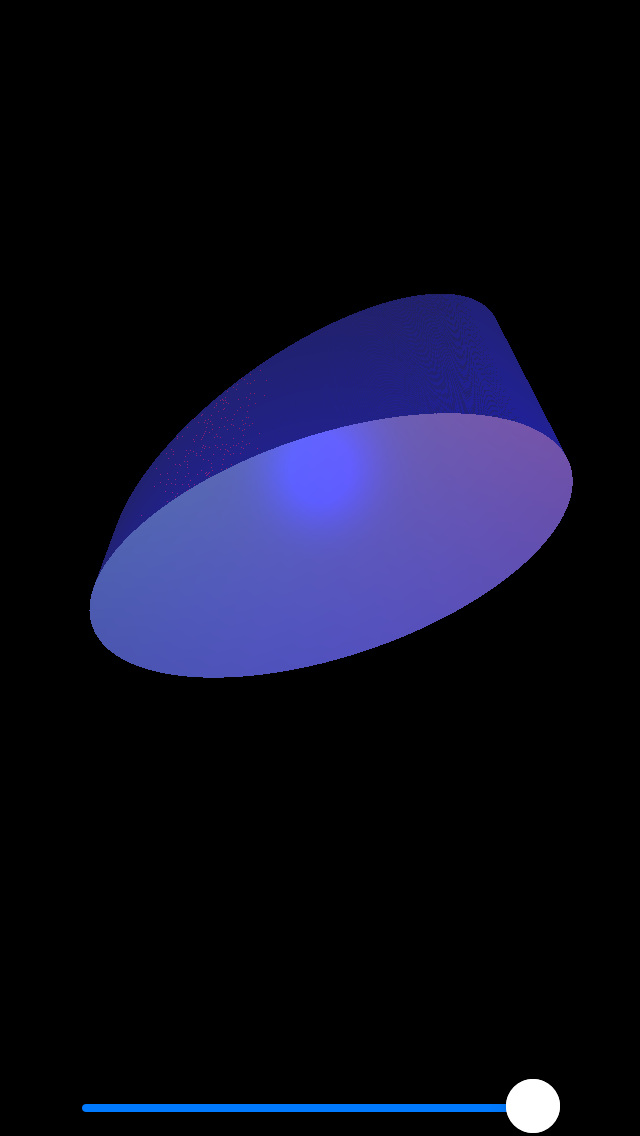
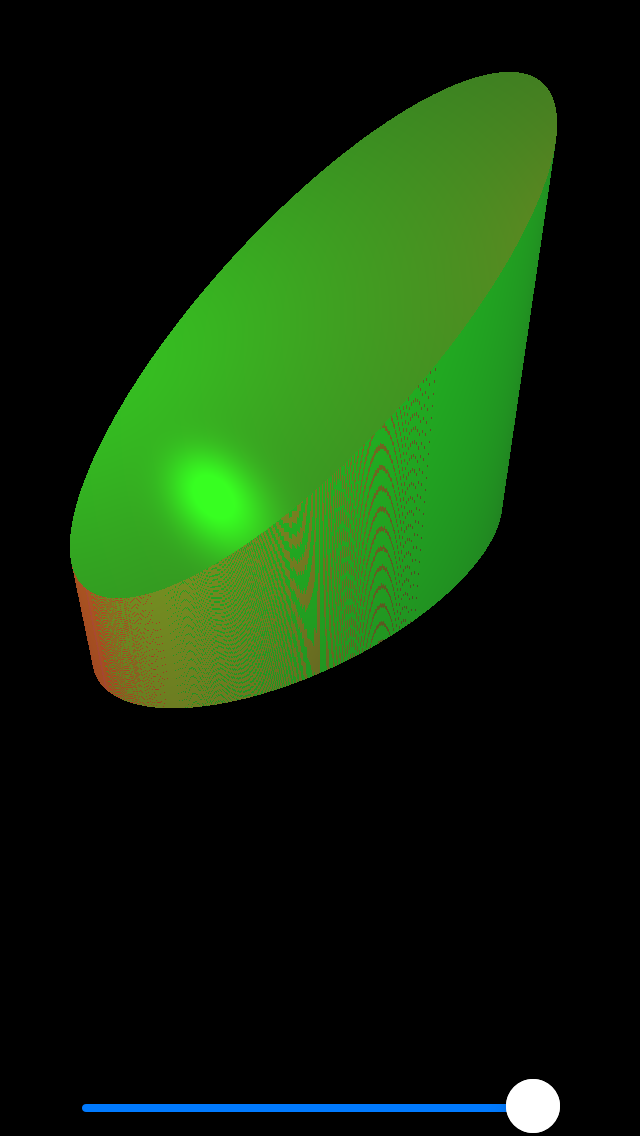


Рис. 4 Максимальная точность аппроксимации + вращение

1. Листинг программы

ViewController.swift

//

// ViewController.swift

// CG3-4

//

// Created by Илья Ильин on 01.11.2021.

//

/// ЛР 4

/// Создать графическое приложение с использованием OpenGL. Изобразить заданное тело с использованием средств OpenGL 2.1 (или выше).

/// Использовать буфер вершин. Точность аппроксимации тела задается пользователем. Обеспечить возможность вращения и масштабирования многогранника и удаление невидимых линий и поверхностей.

/// Реализовать простую модель освещения на GLSL. Параметры освещения и отражающие свойства материала задаются пользователем в диалоговом режиме.

**import** UIKit

**import** MetalKit

**import** ModelIO

**class** ViewController: UIViewController {

**var** mtkView: MTKView!

**var** renderer: Renderer!

**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() {

**super**.viewDidLoad()

setupViews()

**let** device = MTLCreateSystemDefaultDevice()!

mtkView.device = device

mtkView.colorPixelFormat = .bgra8Unorm\_srgb

mtkView.depthStencilPixelFormat = .depth32Float

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

mtkView.delegate = renderer

setupGestures()

}

// **MARK: - Views setup**

**func** setupViews() {

view.backgroundColor = .black

mtkView = MTKView()

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

}

}

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** 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

**var** vertexDescriptor: MDLVertexDescriptor

**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 {

**didSet** {

updateScene()

}

}

**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) {

**self**.device = device

commandQueue = device.makeCommandQueue()!

vertexDescriptor = Renderer.buildVertexDescriptor()

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

depthStencilState = Renderer.buildDepthStencilState(device: device)

scene = Renderer.buildScene(device: device, vertexDescriptor: vertexDescriptor, approximation: approximation)

**super**.init()

}

**static** **func** buildScene(device: MTLDevice, vertexDescriptor: MDLVertexDescriptor, approximation: Float) -> Scene {

**let** bufferAllocator = MTKMeshBufferAllocator(device: device)

**let** scene = Scene()

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))

**let** light1 = Light(worldPosition: simd\_float3(-2, 2, 2), color: simd\_float3(0, 1, 0))

**let** light2 = Light(worldPosition: simd\_float3( 0, -2, 2), color: simd\_float3(0, 0, 1))

scene.lights = [ light0, light1, light2 ]

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

cylinder.material.specularPower = 200

cylinder.material.specularColor = simd\_float3(0.8, 0.8, 0.8)

scene.rootNode.children.append(cylinder)

**return** scene

}

**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.attributes[2] = MDLVertexAttribute(name: MDLVertexAttributeTextureCoordinate,

format: .float2,

offset: MemoryLayout<Float>.stride \* 6,

bufferIndex: 0)

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

**return** vertexDescriptor

}

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

**let** depthStencilDescriptor = MTLDepthStencilDescriptor()

depthStencilDescriptor.depthCompareFunction = .less

depthStencilDescriptor.isDepthWriteEnabled = **true**

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

}

**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.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** updateScene() {

**let** bufferAllocator = MTKMeshBufferAllocator(device: device)

**if** **let** cylinder = scene.nodeNamed("Cylinder") **as**? TiltCylinder {

cylinder.recountMesh(a: 2, b: 1, delta: approximation, device: device, bufferAllocator: bufferAllocator, vertexDescriptor: vertexDescriptor)

}

}

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

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)

}

**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,

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: - 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) // ??!! to use

// commandEncoder.setCullMode(.back)

commandEncoder.setDepthStencilState(depthStencilState)

commandEncoder.setRenderPipelineState(renderPipeline)

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

commandEncoder.endEncoding()

commandBuffer.present(drawable)

commandBuffer.commit()

}

}

}

TiltCylinder.swift

**import** Foundation

**import** simd

**import** MetalKit

**class** TiltCylinder: Node {

**private** **var** a: Float /// semi-major axis

**private** **var** b: Float /// semi-minor axis

**private** **var** delta: Float /// approximation accuracy

**private** **var** height: Float = 2

**private** **var** upperAngle: Double = 30

**private** **var** verticesArray: [Vertex]

**private** **var** indicesArray: [[UInt16]]

**init**(name: String, a: Float, b: Float, delta: Float, device: MTLDevice, bufferAllocator: MTKMeshBufferAllocator, vertexDescriptor: MDLVertexDescriptor) {

**self**.a = a

**self**.b = b

**self**.delta = delta

(verticesArray, indicesArray) = TiltCylinder.countVertices(a: a, b: b, delta: delta, height: height, angle: upperAngle)

**super**.init(name: name)

mesh = createMesh(device: device, bufferAllocator: bufferAllocator, vertexDescriptor: vertexDescriptor)

}

**func** recountMesh(a: Float, b: Float, delta: Float, device: MTLDevice, bufferAllocator: MTKMeshBufferAllocator, vertexDescriptor: MDLVertexDescriptor) {

**self**.a = a

**self**.b = b

**self**.delta = delta

(verticesArray, indicesArray) = TiltCylinder.countVertices(a: a, b: b, delta: delta, height: height, angle: upperAngle)

mesh = createMesh(device: device, bufferAllocator: bufferAllocator, vertexDescriptor: vertexDescriptor)

}

**private** **func** createMesh(device: MTLDevice, bufferAllocator: MTKMeshBufferAllocator, vertexDescriptor: MDLVertexDescriptor) -> MTKMesh {

/// buffers and meshes

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

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

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

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

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

**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: verticesArray.count,

descriptor: vertexDescriptor,

submeshes: submeshes)

**let** mesh: MTKMesh

**do** {

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

} **catch** {

fatalError("Could create mesh")

}

**return** mesh

}

// **MARK: - Counting points**

**private** **class** **func** countVertices(a: Float, b: Float, delta: Float, height: Float, angle: Double) -> ([Vertex], [[UInt16]]) {

**var** resultVertices = [Vertex]()

**var** resultIndices = [[UInt16]]()

**var** vertices = [Vertex]()

**var** indices = [UInt16]()

**let** ellipsePoints = TiltCylinder.countEllipsePoints(a: a, b: b, delta: delta)

**var** shift: UInt16 = 0

/// bottom ellipse

**let** bottomEllipseMatrix = makeRotationMatrixAroundX(degrees: 90)

**let** bottomEllipse = ellipsePoints.map { bottomEllipseMatrix \* $0 }

(vertices, indices) = TiltCylinder.countEllipseVertices(ellipsePoints: TiltCylinder.reverseEllipsePoints(ellipsePoints: bottomEllipse))

resultVertices += vertices

resultIndices.append(indices.map{ $0 + shift })

shift += UInt16(vertices.count)

/// upper ellipse

**let** angleMeasurement = Measurement(value: angle, unit: UnitAngle.degrees)

**let** radians = Float(angleMeasurement.converted(to: .radians).value)

**let** stretchX: Float = 1/cos(radians)

**let** upperEllipseMatrix = makeTranslationMatrix(tx: 0, ty: height, tz: 0) \* makeRotationMatrixAroundZ(degrees: angle) \* makeRotationMatrixAroundX(degrees: 90) \* makeStretchMatrix(sx: stretchX, sy: 1, sz: 1)

**let** upperEllipse = ellipsePoints.map { upperEllipseMatrix \* $0 }

(vertices, indices) = TiltCylinder.countEllipseVertices(ellipsePoints: upperEllipse)

resultVertices += vertices

resultIndices.append(indices.map{ $0 + shift })

shift += UInt16(vertices.count)

/// count side surface

(vertices, indices) = TiltCylinder.countSideSurfaceVertices(upperEllipse: upperEllipse, bottomEllipse: bottomEllipse)

resultVertices += vertices

resultIndices.append(indices.map{ $0 + shift })

**return** (resultVertices, resultIndices)

}

**private** **class** **func** reverseEllipsePoints(ellipsePoints: [simd\_float4]) -> [simd\_float4] {

**return** ellipsePoints.isEmpty ? [] : [ellipsePoints.first!] + ellipsePoints.dropFirst().reversed()

}

**private** **class** **func** countSideSurfaceVertices(upperEllipse: [simd\_float4], bottomEllipse: [simd\_float4]) -> ([Vertex], [UInt16]) {

**if** upperEllipse.count < 3 || bottomEllipse.count < 3 || upperEllipse.count != bottomEllipse.count {

**return** ([], [])

}

**let** circledUpperEllipse = upperEllipse.dropFirst() + [upperEllipse[2]]

**let** circledBottomEllipse = bottomEllipse.dropFirst() + [bottomEllipse[2]]

**var** vertices = [Vertex]()

**var** indices = [UInt16]()

**var** prevPoints: (simd\_float4, simd\_float4)? = **nil**

**for** (upPoint, bottomPoint) **in** zip(circledUpperEllipse, circledBottomEllipse) {

**if** **let** prevPoints = prevPoints {

**let** prevUpPoint = prevPoints.0

**let** prevBottomPoint = prevPoints.1

**let** normal1 = countNormal(v1: prevBottomPoint, v2: prevUpPoint, v3: upPoint)

**let** normal2 = countNormal(v1: prevBottomPoint, v2: upPoint, v3: bottomPoint)

**let** v11 = Vertex(point: prevBottomPoint, normal: normal1)

**let** v12 = Vertex(point: prevUpPoint, normal: normal1)

**let** v22 = Vertex(point: upPoint, normal: normal2)

**let** v23 = Vertex(point: bottomPoint, normal: normal2)

**let** shift = UInt16(vertices.count)

vertices += [v11, v12, v22, v23]

indices += [shift, shift+1, shift+2]

indices += [shift, shift+2, shift+3]

/// triangles layout

/// [v11, v12, v13]

/// [v21, v22, v23]

}

prevPoints = (upPoint, bottomPoint)

}

**return** (vertices, indices)

}

**private** **class** **func** countEllipseVertices(ellipsePoints: [simd\_float4]) -> ([Vertex], [UInt16]) {

**var** verticesArray = [Vertex]()

**var** indicesArray = [UInt16]()

**if** ellipsePoints.count >= 3 {

**let** centerIdx = 0

**let** normal = countNormal(v1: ellipsePoints[0], v2: ellipsePoints[1], v3: ellipsePoints[2])

**for** (index, point) **in** ellipsePoints.enumerated() {

**if** index != 0 && index - 1 != centerIdx {

/// as triangles

indicesArray.append(UInt16(centerIdx))

indicesArray.append(UInt16(index - 1))

indicesArray.append(UInt16(index))

}

verticesArray.append(Vertex(point: point, normal: normal))

}

indicesArray.append(UInt16(centerIdx))

indicesArray.append(UInt16(ellipsePoints.count - 1))

indicesArray.append(UInt16(1))

}

**return** (verticesArray, indicesArray)

}

/// Count Ellipse points, located in Oxy plane

/// - **Parameters**:

/// - a: semi-major axis

/// - b: semi-minor axis

/// - delta: approximation accuracy

/// - **Returns**: array of ellipse points, first point is center

**private** **class** **func** countEllipsePoints(a: Float, b: Float, delta: Float) -> [simd\_float4] {

**let** a2: Float = a \* a

**let** b2: Float = b \* b

**var** x: Float = 0

**var** y: Float = b

**func** f(x: Float, y: Float) -> Float {

**return** b2\*x\*x + a2\*y\*y - a2\*b2

}

**var** points = [simd\_float4]()

/// upper arc

**while** (a2 \* (y - delta/2)) > (b2 \* (x+delta)) {

**let** vertex = simd\_float4(x: x, y: y, z: 0, w: 1)

points.append(vertex)

/// choose next point between (x + delta, y) and (x + delta, y - delta)

**if** (f(x: x + delta, y: y - delta/2) >= 0) {

y -= delta

}

x += delta

}

/// bottom arc

**while** y >= 0 && x <= a {

**let** vertex = simd\_float4(x: x, y: y, z: 0, w: 1)

points.append(vertex)

/// choose between (x + delta, y - delta) and (x, y - delta)

**if** (f(x: x + delta/2, y: y - delta) <= 0) {

x += delta

}

y -= delta

}

**if** **let** prevVertex = points.last {

**let** endVertex = simd\_float4(x: a, y: 0, z: 0, w: 1)

**if** endVertex != prevVertex {

points.append(endVertex)

}

}

/// plot other arcs using symmetry

**let** arc2: [simd\_float4]

**let** arc3: [simd\_float4]

**let** arc4: [simd\_float4]

**if** points.count > 2 {

arc2 = points[1..<points.count-1].map { (v) -> simd\_float4 **in**

**var** newV = v

newV.x = v.x \* -1

**return** newV

}.reversed()

} **else** {

arc2 = []

}

**if** points.count > 1 {

arc3 = points[1...].map { (v) -> simd\_float4 **in**

**var** newV = v

newV.x = v.x \* -1

newV.y = v.y \* -1

**return** newV

}

arc4 = points[0..<points.count-1].map { (v) -> simd\_float4 **in**

**var** newV = v

newV.y = v.y \* -1

**return** newV

}.reversed()

} **else** {

arc3 = []

arc4 = []

}

points += arc4 + arc3 + arc2

**let** center = simd\_float4(x: 0, y: 0, z: 0, w: 1)

**return** [center] + points

}

}

Shaders.metal

#include <metal\_stdlib>

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

**struct** VertexIn {

float3 position [[attribute(0)]];

float3 normal [[attribute(1)]];

float2 texCoords [[attribute(2)]];

};

**struct** VertexOut {

float4 position [[position]];

float3 worldNormal;

float3 worldPosition;

float2 texCoords;

};

**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;

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;

vertexOut.texCoords = vertexIn.texCoords;

**return** vertexOut;

}

**constant** float3 baseColor(0.5, 0.5, 0.5);

**fragment** float4 fragment\_main(VertexOut fragmentIn [[stage\_in]],

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

{

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** float4(finalColor, 1);

}

1. Выводы

В ходе выполнения данной работы познакомился с низкоуровневым фреймворком для работы с графикой - Metal Kit. Особенностью является работа с MTKMesh, что позволяет абстрагировать работу с памятью GPU и переиспользовать имеющиеся вершины. Specular составляющая освещения позволяет добиться более корректного представления металлических обьектов, а сочетание нескольких источников светов дает более красивую картинку

ЛИТЕРАТУРА

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

(Дата обращения: 03.11.2021).