

Metal_PGRF3

Tento tutoriál vznikl jako ukázka programování grafické karty na zařízeních z ekosystému Apple (Mac, iPad, iPhone, Apple TV), za použití knihovny Metal pro účely výuky PGRF3 na FIM UHK. Tutoriál je dostupný na adrese: https://github.com/RomanAuersvald/Metal_PGRF3

Od příchodu macOS Mojave 10.14 označuje Apple OpenGL a OpenCL jako zastaralé API a doporučuje vývoj grafických a výpočetních aplikací s využitím API Metal nebo MetalKit, které jsou vyvíjené od roku 2014 a běží na strojích s procesorem A7 a vyšším.

Pro zjednodušení práce s maticemi je využívána třída `float4x4+Extensions` z portálu raywenderlich.com.

Repozitář se skládá te tří projektů:

1. `PGRF3_metal_starter` - Založený projekt připravený pro postupné doplnění kódu do projektu.
2. `PGRF3_metal_color` - Projekt, kde se po spuštění zobrazí trojúhelník vyplněný interpolovanými barvami.
3. `PGRF3_metal_texture` - Projekt, kde je výstupem trojúhelník vybarvený vzorem textury.

Pokud v tutoriálu naleznete jakoukoli chybu, provedte příslušné akce pro opravu zde na GitHubu. Účelem tohoto tutoriálu je poskytnout informační základ pro samostatné vytvoření úloh pro předmět. Vylepšením tutoriálu proto pomůžeme dalším ročníkům.

Příprava prostředí

Pro vývoj aplikací pro Apple zařízení je zapotřebí instalace vývojového prostředí `XCode` - dostupné z `app Store` nebo betaverze z `developer.apple.com`. Samotný XCode nabízí prostředky pro vývoj, testování, debugging, publikaci aplikací a mnoho dalšího. Není potřebné instalovat žádné další programy ani knihovny - vše potřebné pro tento tutoriál již XCode obsahuje.

Vytvoření projektu

Po instalaci a spuštění XCode se zobrazí nabídka s možností vytvoření nových projektů, nebo naklonování existujícího z Gitu.

Zvolíme pro jaký OS budeme aplikaci vyvíjet. Pro účely tohoto tutoriálu zvolíme `macOS` a `App`, potvrďme.

Choose a template for your new project:

iOS watchOS tvOS macOS Cross-platform

Filter

Application



App



Game



Command
Line Tool

Framework & Library



Framework



Library



Metal Library



XPC Service



Bundle

Other



AppleScript App



Safari Extension



Automator Action



Contacts Action



Generic Kernel

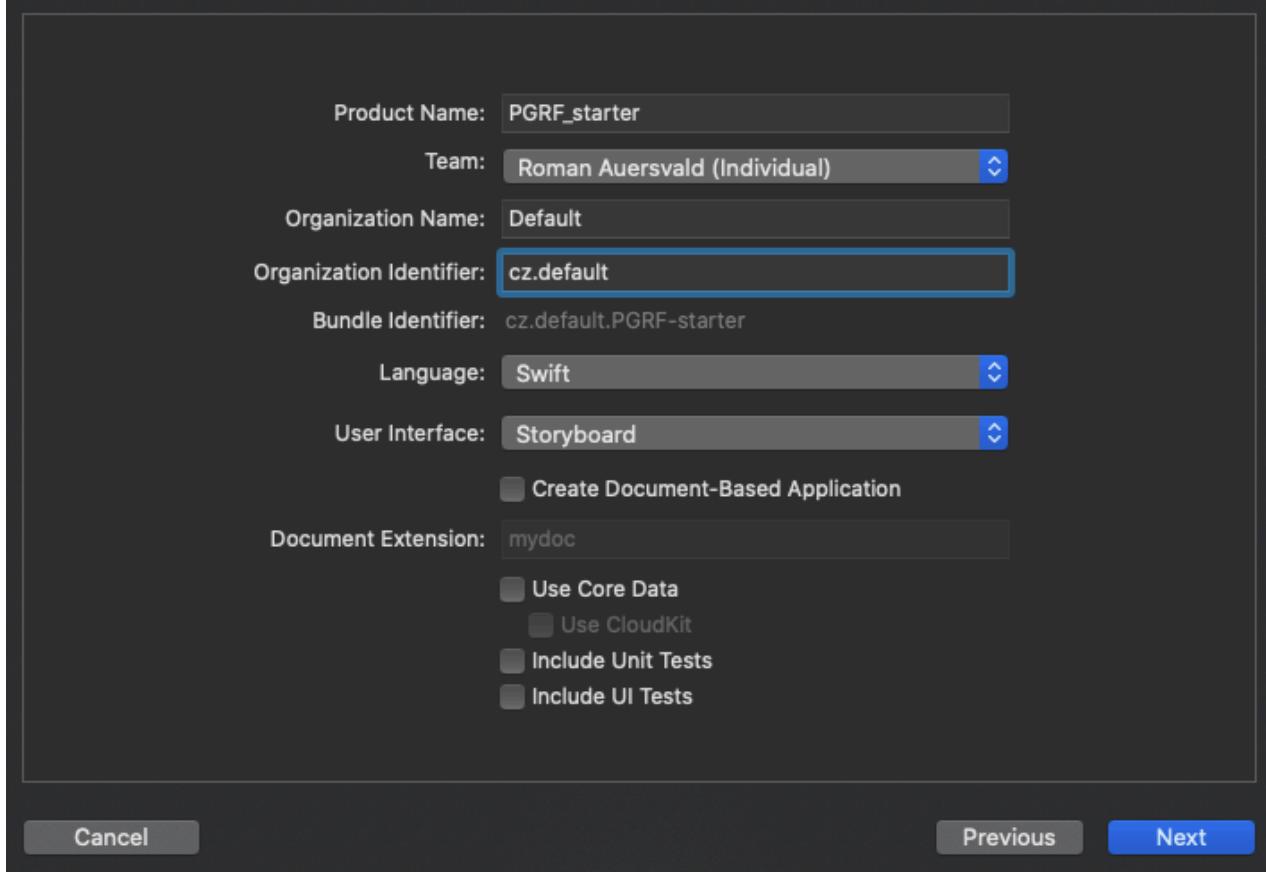
Cancel

Previous

Next

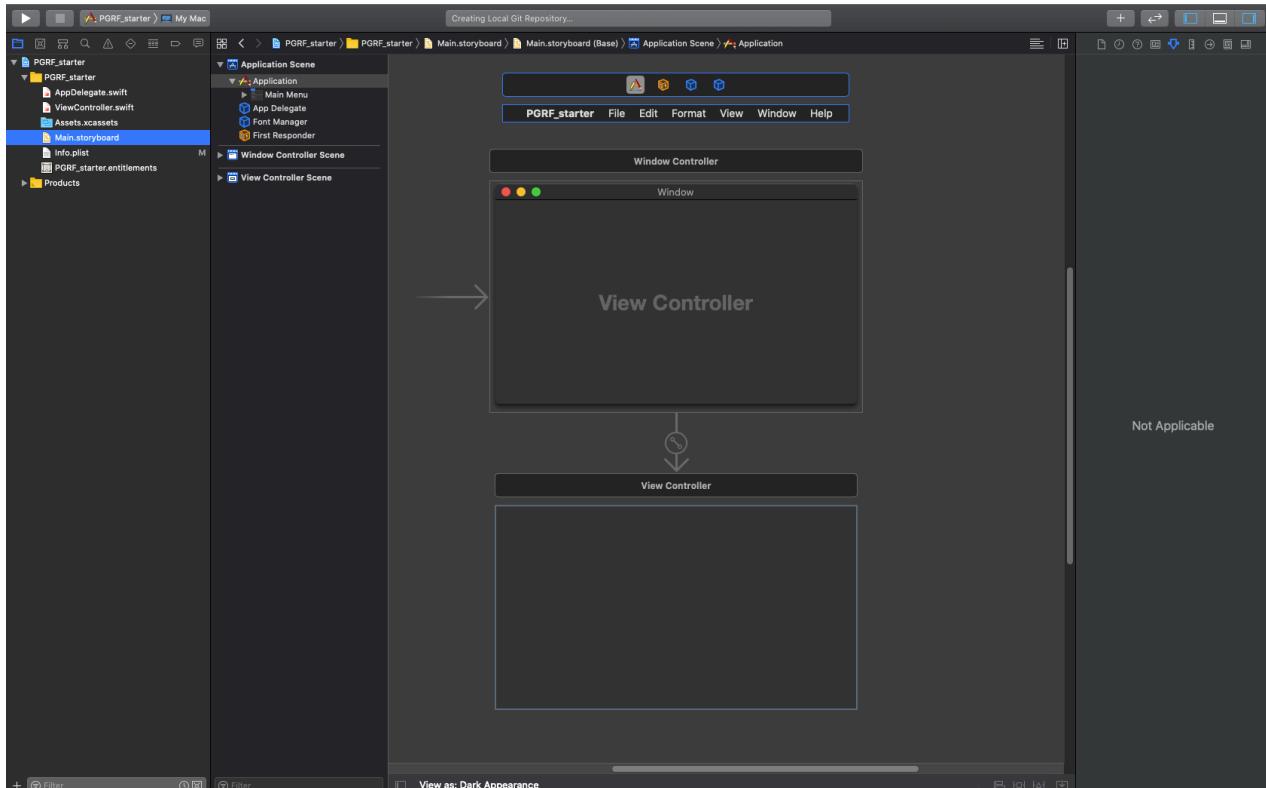
V následujícím kroku vyplníme údaje o aplikaci. Jako tým zvolíme účet, kde je v závorce uvedeno (individual), tím se vyhneme nutnosti vytváření certifikátů pro vývoj. Ty budou vytvořeny automaticky. Jazyk nastavíme na `Swift` a potvrďme.

Choose options for your new project:



V následujícím kroku stačí pouze vybrat umístění aplikace a potvrdit tlačítkem **Create**.

Po vytvoření nám bude zobrazeno výchozí rozvržení aplikace a otevře se námi vytvořený projekt, jehož soubory uvidíme v navigačním sloupci vlevo.



Základní pojmy

`ViewController` - Hlavní View Controller, který řídí obsah a akce v zobrazeném okně.

`MTKViewDelegate` - Delegát MTKView, stará se o překreslování scény a adaptaci při změně velikosti plátna.

`pipelineState` - Nastavení renderovací pipeline - formáty barev, blending, reference na VS a FS funkce.

`commandQueue` - Posloupnost příkazů ke spuštění na GPU.

`simd_float 3 - 4` - Datový typ pro definici vektorů.

`MTLDevice` - Reference na aktuální zařízení.

`renderEncoder` - Obsahuje nastavení pro renderování aktuálního snímku.

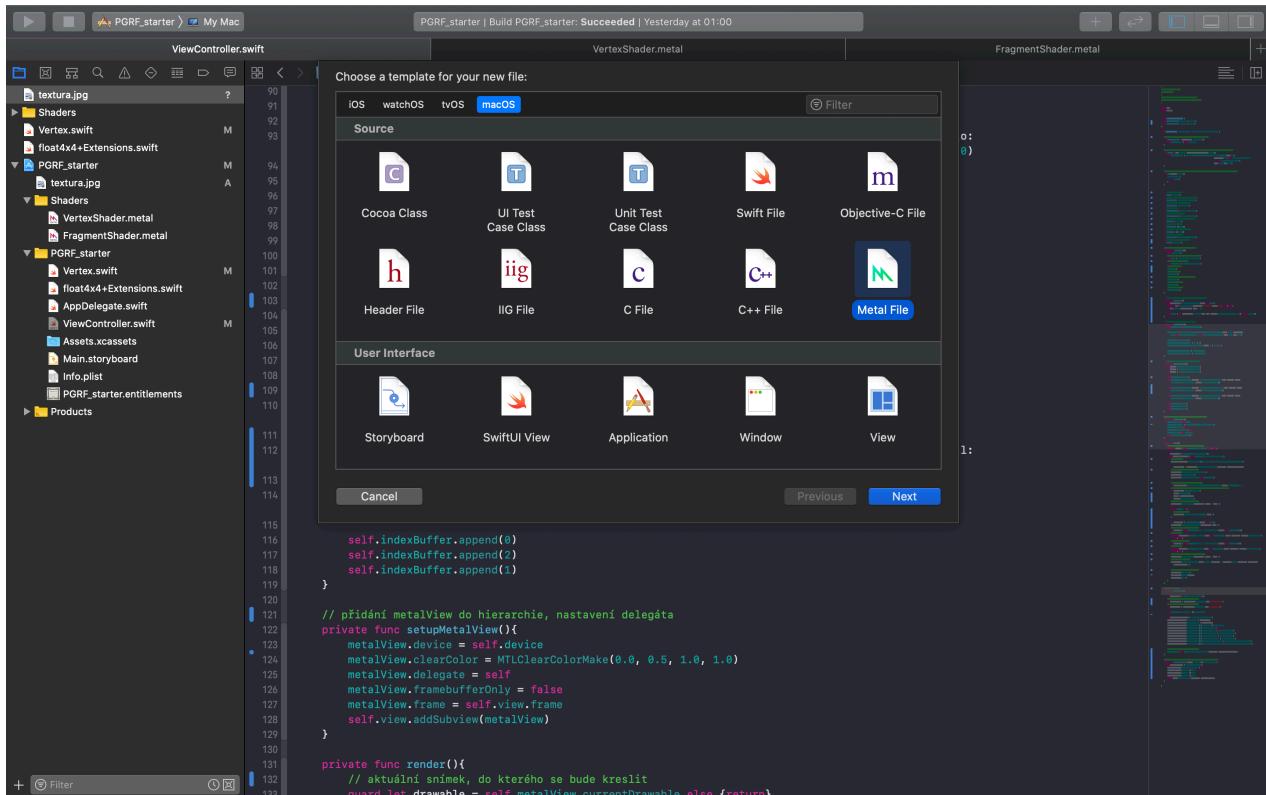
`renderPassDescriptor` - Nastavení renderEncoderu.

`drawIndexedPrimitives` - Metoda renderEncoderu, která zajistí vykreslení nastavené topologie (.triangle, .triangleStrip, .line, .lineStrip, .point, ...).

Vykreslení objektu

Vertex Shader (VS)

Pro vytvoření VS vytvoříme příslušný soubor `cmd + N` nebo `file -> New File` a z nabídky vybereme Metal file.



Po vytvoření definujeme struktury, které budou do shaderu vstupovat a vystupovat. Jedná se o `VertexIn` a `VertexOut`. `VertexOut` bude sloužit jako struktura pro předávání dat mezi shadery. V tomto případě z VS do FS.

```
// vertex vstupující do vs
struct VertexIn {
    float3 position;
    float4 color;
    float3 normal;
    float2 texCoord;
};

// vertex pro fs
struct VertexOut {
    // vypočtená pozice ve scéně
    float4 computedPosition [[position]];
    // barva
    float4 color;
    // souřadnice do textury
    float2 texCoord;
    // normála
    float3 normal;
    // velikost bodu
    float pointsize[[point_size]];
};
```

Do shaderu vstupují také matice. Projekční a modelová.

```
// vstupující matice
struct SceneMatrices {
    float4x4 projectionMatrix;
    float4x4 viewModelMatrix;
};
```

Nyní definujeme samotnou hlavní funkci shaderu.

```
// hlavní metoda shaderu
vertex VertexOut basic_vertex(
    // pole vertexů vstupujících
    const device VertexIn* vertex_array [[ buffer(0) ]],
    // matice
    const device SceneMatrices& scene_matrices [[ buffer(1) ]],
    // id vertexu
    unsigned int vid [[ vertex_id ]]) {

    float4x4 projectionMatrix = scene_matrices.projectionMatrix;

    // vytvoření výstupního vertexu a naplnění parametrů
    VertexOut outVertex = VertexOut();
    VertexIn v = vertex_array[vid];
    outVertex.computedPosition = projectionMatrix * float4(v.position, 1.0);

    outVertex.normal = v.normal;
    outVertex.color = v.color;
    outVertex.texCoord = v.texCoord;
    outVertex.pointsize = 4;
    return outVertex;
}
```

Fragment Shader (FS)

Pro FS vytvoříme příslušný soubor stejným způsobem, jako při vytváření souboru pro VS. Je také možné oba shadery umístit do jednoho souboru, jedná se pouze o preferenci.

Definujeme vstupní strukturu vertexu.

```
// vstupující vertex
struct VertexIn {
    // vypočtená pozice ve scéně
    float4 computedPosition [[position]];
    // barva
    float4 color;
    // souřadnice do textury
    float2 texCoord;
    // normála
    float3 normal;
    // velikost bodu pro bodové zobrazení
    float pointsize[[point_size]];
};
```

Vytvoříme hlavní funkci fragment shaderu, která obarví příslušný pixel podle barvy vertexu.

```
// hlavní metoda fs
fragment float4 basic_fragment(
    // vstupující vertex
    VertexIn interpolated [[stage_in]]) {
    return interpolated.color;
}
```

Příprava renderovací pipeline

Následující kód řeší inicializaci `MTLRenderPipelineDescriptor` u, kterému nastavíme vertex a fragment shader a určíme další parametry. Výsledkem tohoto kódu je vytvořený `PipelineState` s nastavením z `MTLRenderPipelineDescriptor` u.

```
// nastavení shaderů
private func setupShaders(){

    let defaultLibrary = device.makeDefaultLibrary()!
    // inicializace fs a jeho hlavní metody
    let fragmentProgram = defaultLibrary.makeFunction(name: "basic_fragment")
    // inicializace vs a jeho hlavní metody
    let vertexProgram = defaultLibrary.makeFunction(name: "basic_vertex")

    self.metalView.colorPixelFormat = .bgra8Unorm_srgb

    let pipelineStateDescriptor = MTLRenderPipelineDescriptor()
    pipelineStateDescriptor.vertexFunction = vertexProgram
    pipelineStateDescriptor.fragmentFunction = fragmentProgram
    pipelineStateDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm_srgb
    pipelineStateDescriptor.colorAttachments[0].isBlendingEnabled = true
    pipelineStateDescriptor.colorAttachments[0].rgbBlendOperation = MTLBlendOp
    pipelineStateDescriptor.colorAttachments[0].alphaBlendOperation = MTLBlendOp
    pipelineStateDescriptor.colorAttachments[0].sourceRGBBlendFactor = MTLBlendF
    pipelineStateDescriptor.colorAttachments[0].sourceAlphaBlendFactor = MTLBlende
    pipelineStateDescriptor.colorAttachments[0].destinationRGBBlendFactor = MTLBla
    pipelineStateDescriptor.colorAttachments[0].destinationAlphaBlendFactor = MTLBla

    // komplikace pipeline do pipeline state
    pipelineState = try! device.makeRenderPipelineState(descriptor: pipelineSta
}
```

Tuto funkci zavoláme ve `viewDidLoad()`.

Definice tělesa

K definici bodů ve scéně budeme využívat datovou strukturu `Vertex`, kterou vytvoříme, a bude obsahovat potřebné parametry.

```
// Vrchol tělesa
struct Vertex {
    var position : vector_float3
    var color : vector_float4
    var normal: vector_float3
    var texcord: vector_float2
}
```

Vytvoříme globální proměnnou `vertexBuffer` v našem `ViewControlleru`.

```
/// pole vrcholů objektu
var vertexBuffer = [Vertex]()
```

Tuto strukturu budeme také kódovat pro použití v našem VS, proto musí obsahovat všechny potřebné parametry.

Pro naši ukázkou zobrazíme trojúhelník, který bude definován Vertexy, kde každý bude mít jinou barvu a po vykreslení bude trojúhelník vybarven jejich interpolací.

Zadefinujeme proto novou metodu `createPrimitives()`, která nám daný trojúhelník vytvoří.

```
/// vytvoření vzorového trojúhelníku - vb a ib
private func createPrimitives(){
    let vertCol1 = SIMD_float4.init(1, 0, 0, 1)
    let vertCol2 = SIMD_float4.init(0, 1, 0, 1)
    let vertCol3 = SIMD_float4.init(0, 0, 1, 1)

    self.vertexBuffer.append(Vertex(position: SIMD_float3.init(0, 0, -2), color: SIMD_float4.init(1, 0, 0, 1)))
    self.vertexBuffer.append(Vertex(position: SIMD_float3.init(0, 0.5, -2), color: SIMD_float4.init(0, 1, 0, 1)))
    self.vertexBuffer.append(Vertex(position: SIMD_float3.init(1, 1, -2), color: SIMD_float4.init(0, 0, 1, 1)))

    self.indexBuffer.append(0)
    self.indexBuffer.append(2)
    self.indexBuffer.append(1)
}
```

Výsledkem je naplnění proměnné `vertexBuffer` daty o Vertexech a proměnnou `indexBuffer` daty o topologii. Tuto funkci zavoláme ve `viewDidLoad()`.

Matice

Pro korektní zobrazení vzorového trojúhelníku a případnou další transformaci vrcholů definujeme globální strukturu `SceneMatrices`, kde budou ukládány projekční a modelView matice.

```
struct SceneMatrices{
    var projectionMatrix: float4x4 = float4x4()
    var modelviewMatrix: float4x4 = float4x4()
}
```

Vytvoříme globální proměnnou `sceneMatrices` v našem `ViewControlleru`.

```
/// globální matice scény
var sceneMatrices : SceneMatricesfloat4x4!
```

Obě matice jsou formátu `float4x4`, což označuje matici 4×4 typu Float. Matice inicializujeme přidáním následující funkce do `ViewController.swift`.

```
/// nastaví projekční a modelovou matici
private func setupMatrices(){
    self.sceneMatrices = SceneMatricesfloat4x4()

    projectionMatrix = float4x4.makePerspectiveViewAngle(float4x4.degrees(toRad: 45))
    worldModelMatrix = float4x4()
    worldModelMatrix.translate(0.0, y: 0.0, z: 4)
    worldModelMatrix.rotateAroundX(float4x4.degrees(toRad: 0), y: 0.0, z: 0.0)

    sceneMatrices.modelviewMatrix = worldModelMatrix
    sceneMatrices.projectionMatrix = projectionMatrix
}
```

Tuto funkci zavoláme ve `viewDidLoad()`.

Rendering

Nyní je na čase definovat metodu, která bude obsluhovat renderování trojúhelníku.

```

private func render(){
    // aktuální snímek, do kterého se bude kreslit
    guard let drawable = self.metalView.currentDrawable else {return}

    let commandBuffer = commandQueue.makeCommandBuffer()!
    if let renderPassDescriptor = self.metalView.currentRenderPassDescriptor{
        // pozadí smínku
        renderPassDescriptor.colorAttachments[0]. clearColor = MTLClearColorMake(0.0, 0.0, 0.0, 1.0)

        let renderEncoder = commandBuffer.makeRenderCommandEncoder(descriptor:
            // obecné nastavení render encoderu
            renderEncoder.setFrontFacing(.counterClockwise)
            renderEncoder.setCullMode(.back)
            renderEncoder.setRenderPipelineState(self.pipelineState)

            // výpočet velikosti pro matice v paměti
        let uniformBufferSize = MemoryLayout<SceneMatrices<float4x4>>.size(ofValue: 1)
        // vytvoření bufferu s maticemi o velikosti vypočtené v předchozím kroku
        let uniformBuffer = device.makeBuffer(
            bytes: &sceneMatrices,
            length: uniformBufferSize,
            options: .storageModeShared)
        // scene matrices pro vertex shader
        renderEncoder.setVertexBuffer(uniformBuffer, offset: 0, index: 1)

        // výpočet velikosti vertex bufferu
        let vbDataSize = self.vertexBuffer.count * MemoryLayout.size(ofValue: 1)
        // vytvoření vb
        guard let vertexBuffer = device.makeBuffer(bytes: self.vertexBuffer, length: vbDataSize) else {
            return
        }

        // výpočet velikosti ib
        let ibDataSize = self.indexBuffer.count * MemoryLayout.size(ofValue: 1)
        // vytvoření ib
        guard let indexBuffer = device.makeBuffer(bytes: self.indexBuffer, length: ibDataSize) else {
            return
        }

        // přiřazení vb do render encoderu - pro vs
        renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, index: 0)

        // kreslení trojúhelníku
        renderEncoder.drawIndexedPrimitives(type: .triangle, indexCount: 3, indexType: .uint16,
            vertexFormat: .float32,
            vertexBuffer: vertexBuffer,
            vertexOffset: 0,
            indexBuffer: indexBuffer,
            indexOffset: 0)

        // konec kódování aktuálního snímku
        renderEncoder.endEncoding()
        commandBuffer.present(drawable)
        commandBuffer.commit()
    }
}

```

Tuto funkci přidáme do

```
// volané delegátní metodou MTKViewDelegate - pro renderování, volá dále naši metody
func draw(in view: MTKView) {
    autoreleasepool {
        * zde
    }
}
```

kde bude volaná při každém překreslení našeho `metalView`. Ve výchozím nastavení se scéna překresluje rychlostí až 120 FPS, tato rychlosť závisí také na složitosti zobrazované scény.

V tento okamžik po spuštění aplikace bychom měli spatřit vyrenderovaný trojúhelník podobný tomuto.

Textura

Pro implementaci oboření texturou jsou zapotřebí následující objekty: `obrázek textury`, `SamplerState` a `MTKTextureLoader`.

Nejdříve je zapotřebí texturu globálně definovat v našem ViewControllereru.

```
/// textura pro aplikaci na těleso
var texture: MTLTexture?
```

Poté samotnou texturu načíst ze souboru umístěného v našem projektu. (Přetažením zdrojového obrázku do levého sloupce se zdrojovými soubory se textura vloží do projektu.) Samotné načtení zajistí následující funkce.

```
/// načte texturu ze souboru umístěného v root adresáři
private func loadTexture(){
    let textureLoader = MTKTextureLoader.init(device: self.device)
    guard let path = Bundle.main.path(forResource: "textura", ofType: "jpg") else { return }
    let data = NSData(contentsOfFile: path)! as Data

    self.texture = try! textureLoader.newTexture(data: data, options: [MTKTextureLoader.Option.wrapMode: .repeat])
}
```

Definovanou funkci zavoláme ve `viewDidLoad()`. Nyní přiřadíme texturu našemu `renderEncoder` u. Vložením následujícího kódu pod přízezení vertexBufferu v metodě `render()`.

```
if self.texture != nil{
    // nastavení textury
    renderEncoder.setFragmentTexture(texture!, index: 0)
}
```

Pro vzorkování textury je nutné vytvořit `SamplerState`, který se o tuto činnost v našem FS postará.

```
// specifikace a vytvoření sampler state pro vzorkování textury v fs
private func buildSamplerState(device: MTLDevice) -> MTLSamplerState {
    let samplerDescriptor = MTLSamplerDescriptor()
    samplerDescriptor.normalizedCoordinates = true
    samplerDescriptor.minFilter = .linear
    samplerDescriptor.magFilter = .linear
    samplerDescriptor.mipFilter = .linear
    return device.makeSamplerState(descriptor: samplerDescriptor)!
}
```

S definovanou metodou, která nám vrátí nastavený SamplerState jej přidáme k našemu `renderEncoder` už hned pod přiřazením textury následujícím kódem.

```
let samplerState = buildSamplerState(device: self.device)
renderEncoder.setFragmentSamplerState(samplerState, index: 0)
```

Celá metoda `render()` nyní vypadá takto:

```

private func render(){
    // aktuální snímek, do kterého se bude kreslit
    guard let drawable = self.metalView.currentDrawable else {return}

    let commandBuffer = commandQueue.makeCommandBuffer()!
    if let renderPassDescriptor = self.metalView.currentRenderPassDescriptor{
        // pozadí smínku
        renderPassDescriptor.colorAttachments[0]. clearColor = MTLClearColorMake(0.0, 0.0, 0.0, 1.0)

        let renderEncoder = commandBuffer.makeRenderCommandEncoder(descriptor:
            // obecné nastavení render encoderu
            renderEncoder.setFrontFacing(.counterClockwise)
            renderEncoder.setCullMode(.back)
            renderEncoder.setRenderPipelineState(self.pipelineState)

            // výpočet velikosti pro matice v paměti
            let uniformBufferSize = MemoryLayout<SceneMatrices<float4x4>>.size(ofValue: 1)
            // vytvoření bufferu s maticemi o velikosti vypočtené v předchozím kroku
            let uniformBuffer = device.makeBuffer(
                bytes: &sceneMatrices,
                length: uniformBufferSize,
                options: .storageModeShared)
            // scene matrices pro vertex shader
            renderEncoder.setVertexBuffer(uniformBuffer, offset: 0, index: 1)

            if self.texture != nil{
                // nastavení textury
                renderEncoder.setFragmentTexture(texture!, index: 0)
            }

            let samplerState = buildSamplerState(device: self.device)
            renderEncoder.setFragmentSamplerState(samplerState, index: 0)
            // výpočet velikosti vertex bufferu
            let vbDataSize = self.vertexBuffer.count * MemoryLayout.size(ofValue: 1)
            // vytvoření vb
            guard let vertexBuffer = device.makeBuffer(bytes: self.vertexBuffer, length: vbDataSize) else {
                // výpočet velikosti ib
                let ibDataSize = self.vertexBuffer.count * MemoryLayout.size(ofValue: 1)
                // vytvoření ib
                guard let indexBuffer = device.makeBuffer(bytes: self.indexBuffer, length: ibDataSize) else {
                    // přiřazení vb do render encoderu - pro vs
                    renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, index: 0)
                    // kreslení trojúhelníku
                    renderEncoder.drawIndexedPrimitives(type: .triangle, indexCount: 3, indexType: .uint16)
                }
            }
            // konec kódování aktuálního snímku
            renderEncoder.endEncoding()
            commandBuffer.present(drawable)
            commandBuffer.commit()
        }
    }
}

```

FS

Poslední krok, který proobarvení tělesa texturou zbývá, je nastavení FS na využití našeho sampleru. Proto přejdeme do souboru s naším FS a přidáme vstupní parametry.

```
// obrázek textury
texture2d<float, access::sample> tex2D [[ texture(0)
// sampler pro texturu
sampler sampler2D [[ sampler(0) ]]]
```

Výpočet barvy nyní změníme z:

```
return interpolated.color;
```

na

```
float3 color = tex2D.sample(sampler2D, interpolated.texCoord).rgb;
return float4(color, 1);
```

Nyní po spuštění aplikace uvidíme náš trojúhelník obořený podle naší textury.

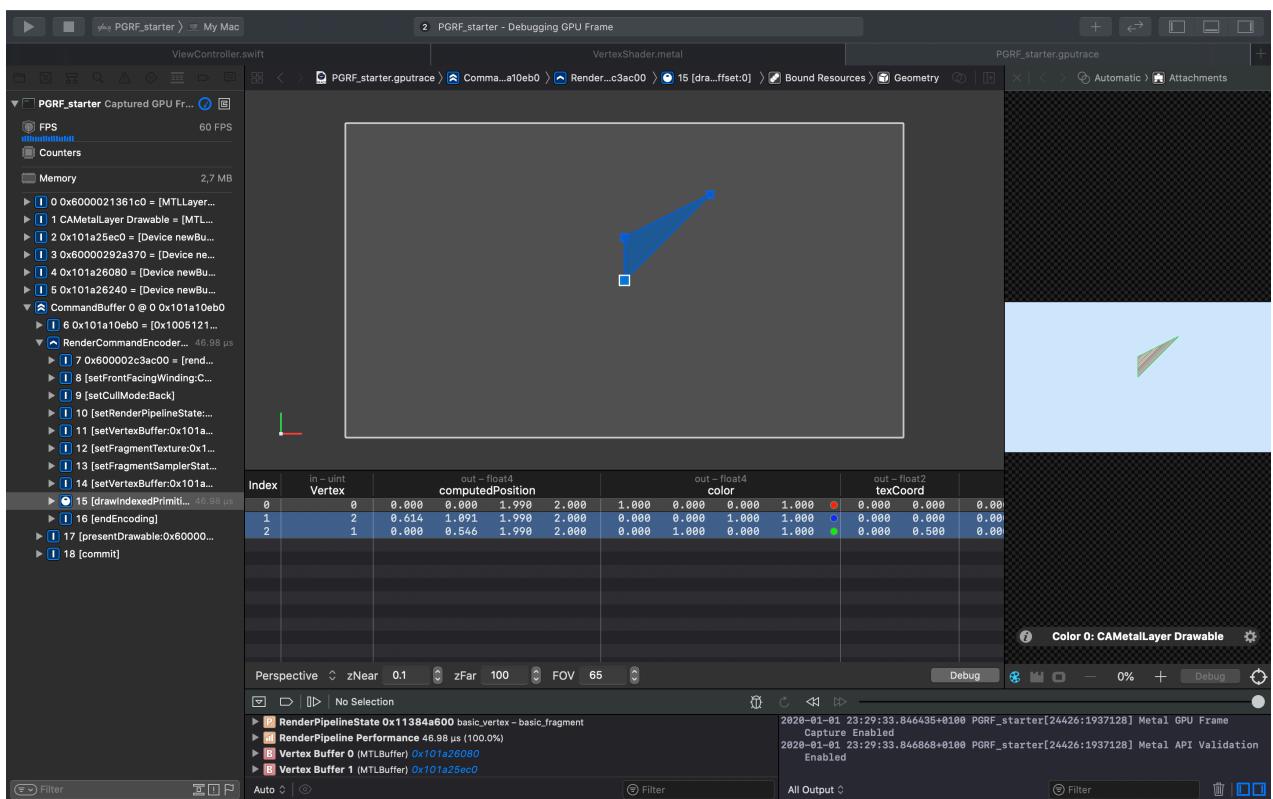
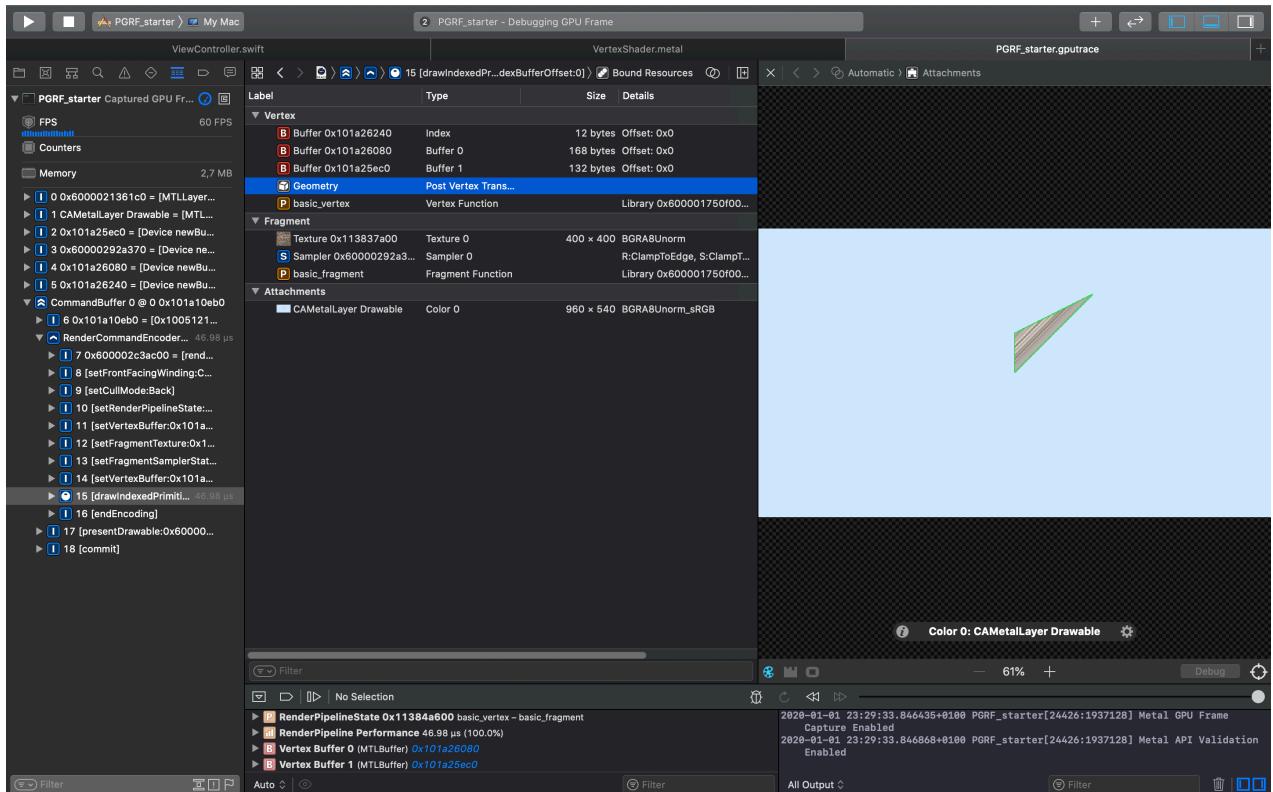
Tipy a triky

Debugging

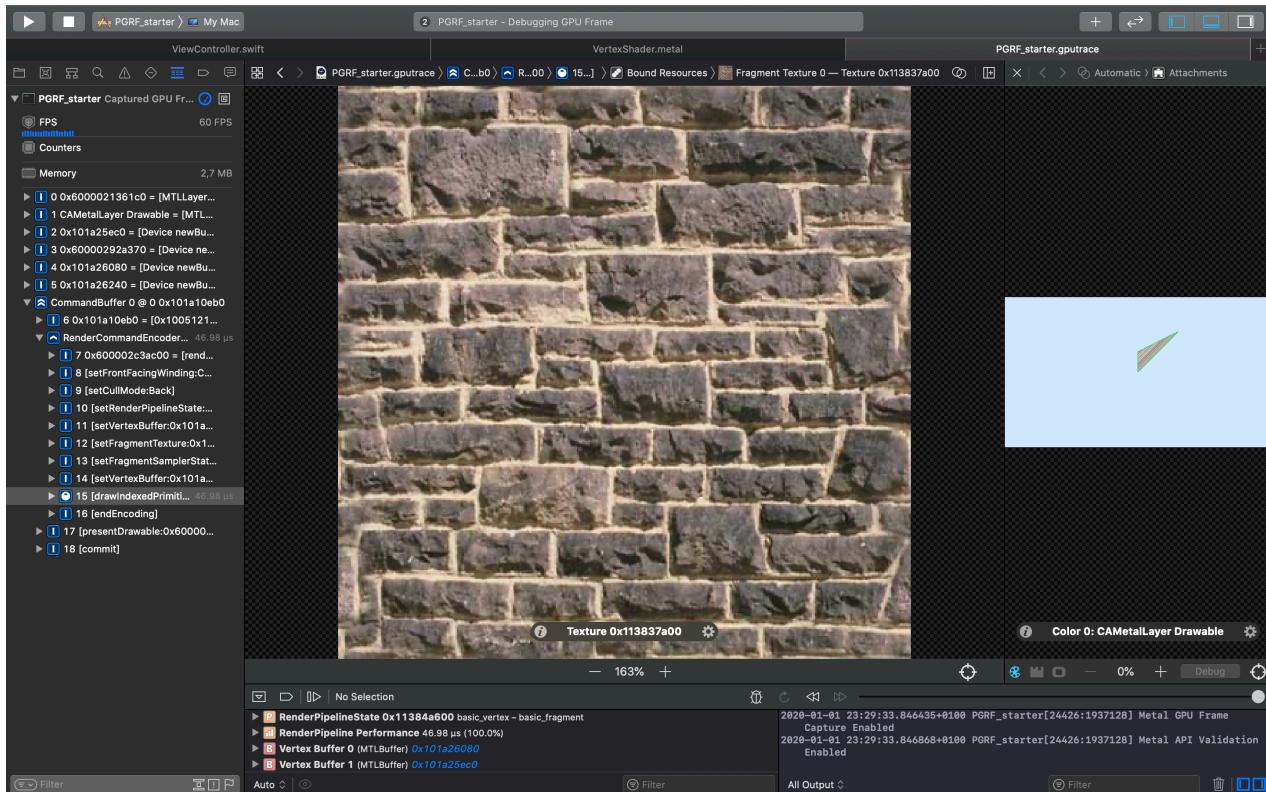
Pro snadnější debugging samotného vykreslování nabízí XCode možnost zaznamenat aktuální snímek a zobrazit podrobnosti o tom, co se odehrává na grafické kartě a jak jsou nastaveny buffery.

Pro zaznamenání obrazovky slouží ikona kamery na spodní liště.

Po přepnutí do režimu lazení se změní rozložení oken a zobrazí se seznam vláken s parametry a položkami, které je možné dále rozkliknout. Například můžeme po otevření položky **Geometry** zobrazit aktuální informace o jednotlivých vrcholech trojúhelníku.



Nebo zobrazit námi načtenou texturu, která se předala našemu FS.



Z ladícího režimu se přepneme zpět kliknutím na šipku na spodní liště, nebo ukončením běhu aplikace tlačítkem stop.

Užitečné odkazy a zdroje

Tutoriály

Série tutoriálů začínající:

<https://www.raywenderlich.com/7475-metal-tutorial-getting-started>

<https://www.raywenderlich.com/9211-moving-from-opengl-to-metal> <https://academy.realm.io/posts/3d-graphics-metal-swift/> <https://www.clientresourcesinc.com/2018/07/27/rendering-graphics-with-metalkit-swift-4-part-2/> <http://metalkit.org> <https://metalbyexample.com/modern-metal-1/>

Užitečné zdroje

<https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf>

<https://developer.apple.com/library/archive/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Render-Ctx/Render-Ctx.html>

https://developer.apple.com/documentation/metal/shaderauthoring/developinganddebuggingmetal_shaders

Zdroje

<https://www.root.cz/zpravicky/apple-macos-10-14-jiz-oznacuje-opengl-a-opencl-za-zastarale/>

