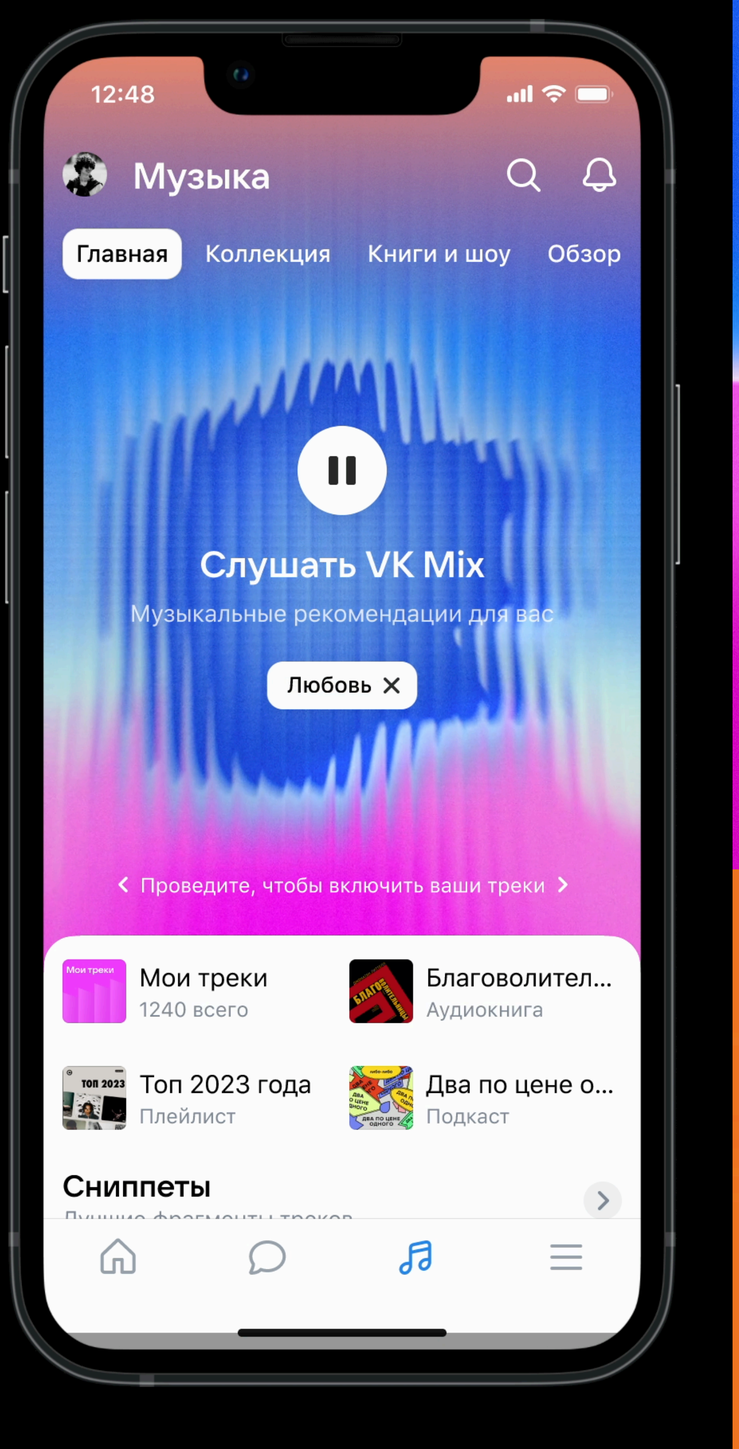
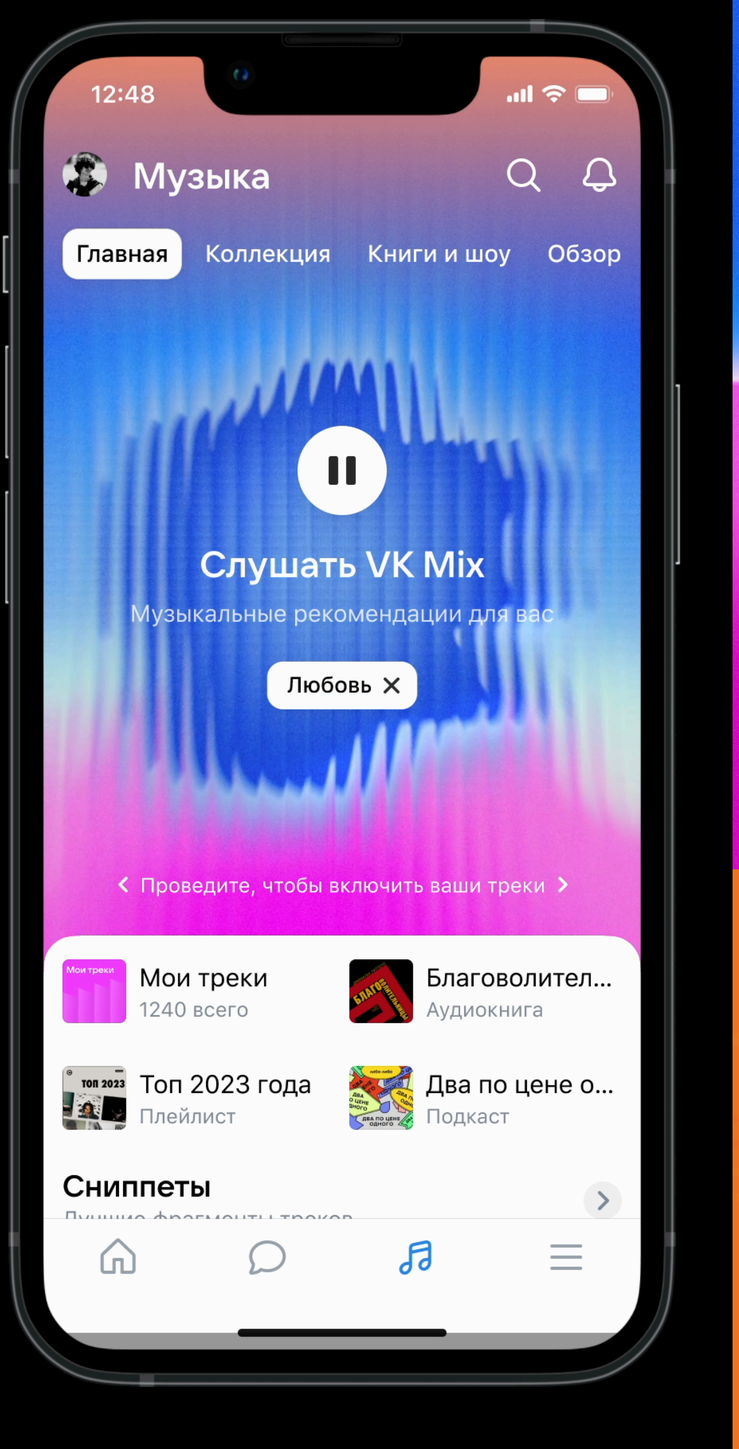
Здесь куют Metal

Тёма Нестеренко

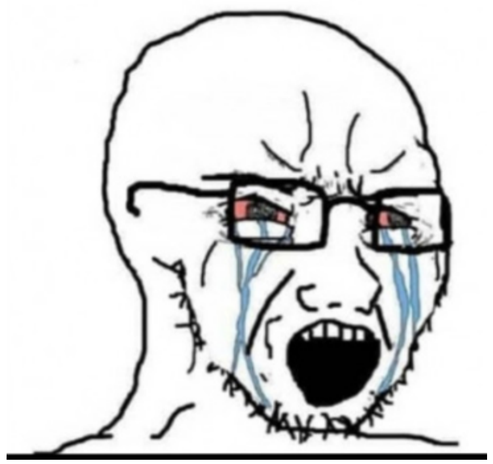
vk.com/music


VKontakte Core iOS

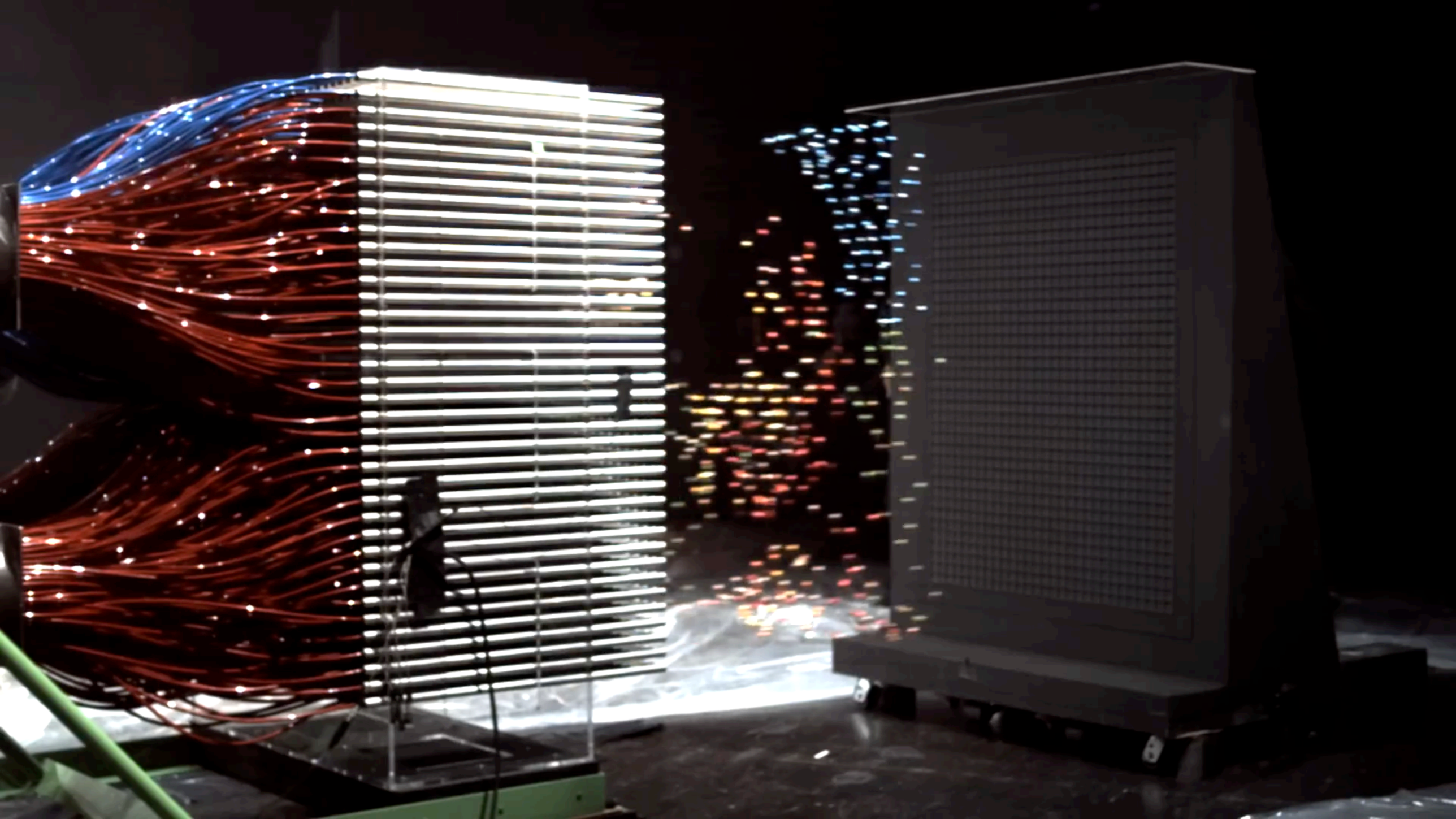

12:48
Музыка
Главная
Коллекция
Книги и шоу
Обзор
Слушать VK Mix
Музыкальные рекомендации для вас
Любовь
Проведите, чтобы включить ваши треки
Мои треки
1240 всего
Благоволител...
Аудиокнига
Топ 2023 года
Плейлист
Два по цене о...
Подкаст
Сниппеты

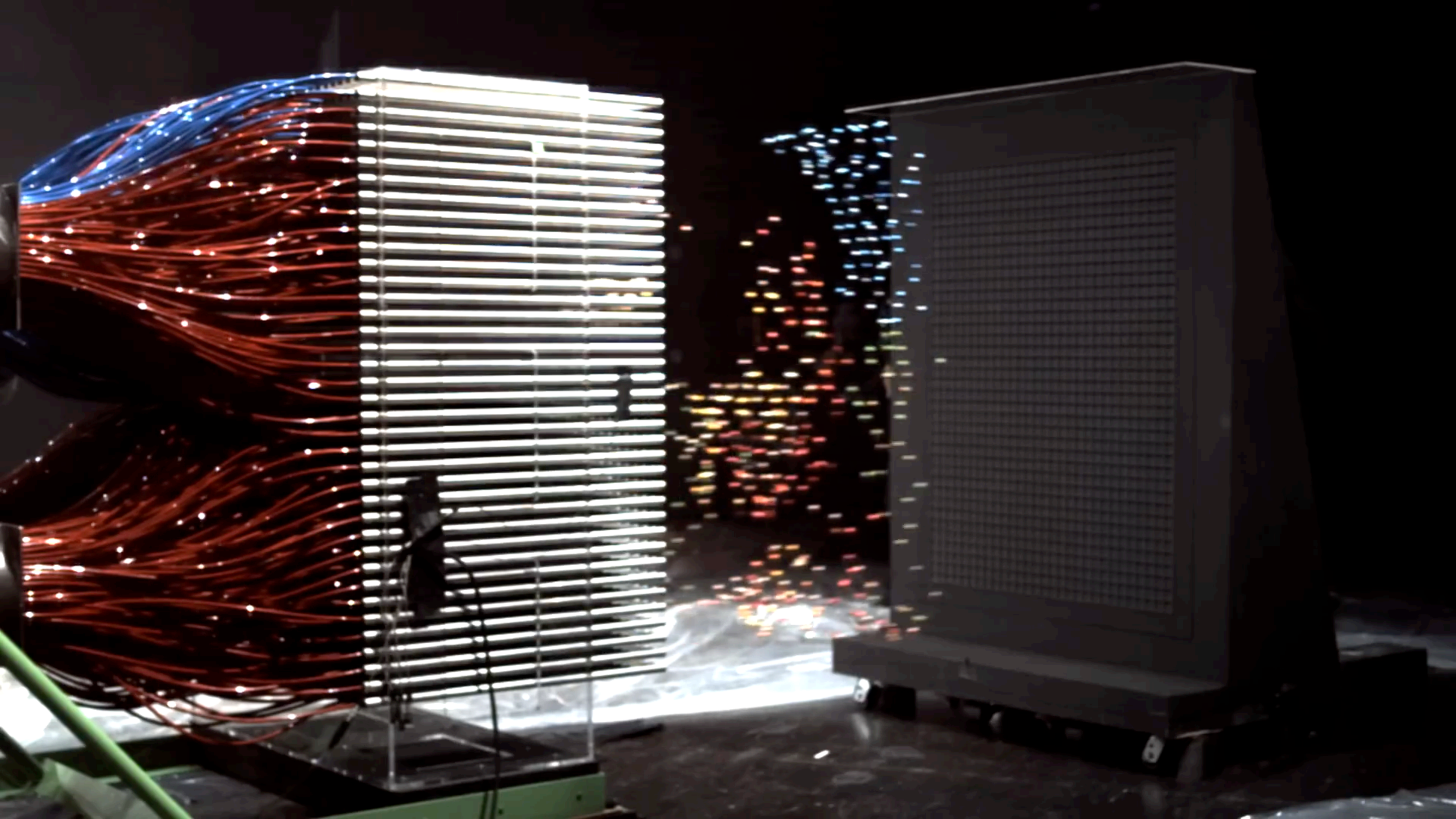

12:48
Музыка
Главная
Коллекция
Книги и шоу
Обзор
Слушать VK Mix
Музыкальные рекомендации для вас
Любовь
Проведите, чтобы включить ваши треки
Мои треки
1240 всего
Благоволител...
Аудиокнига
Топ 2023 года
Плейлист
Два по цене о...
Подкаст
Сниппеты

QUAKE 2, 1997, CPU SHADING

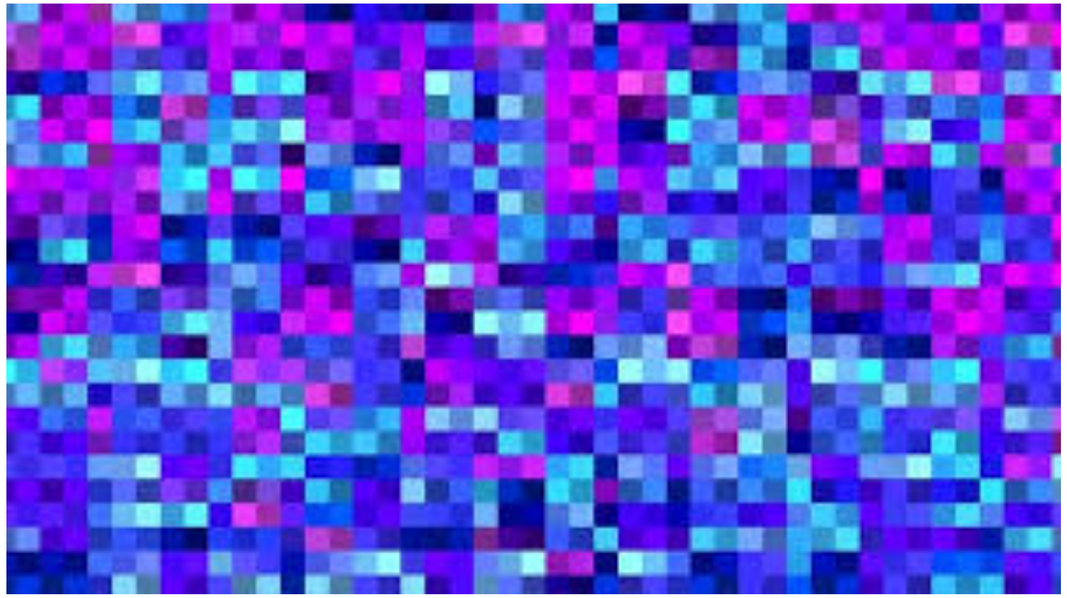


QUAKE 3, 1999, GPU SHADING, JAPAN



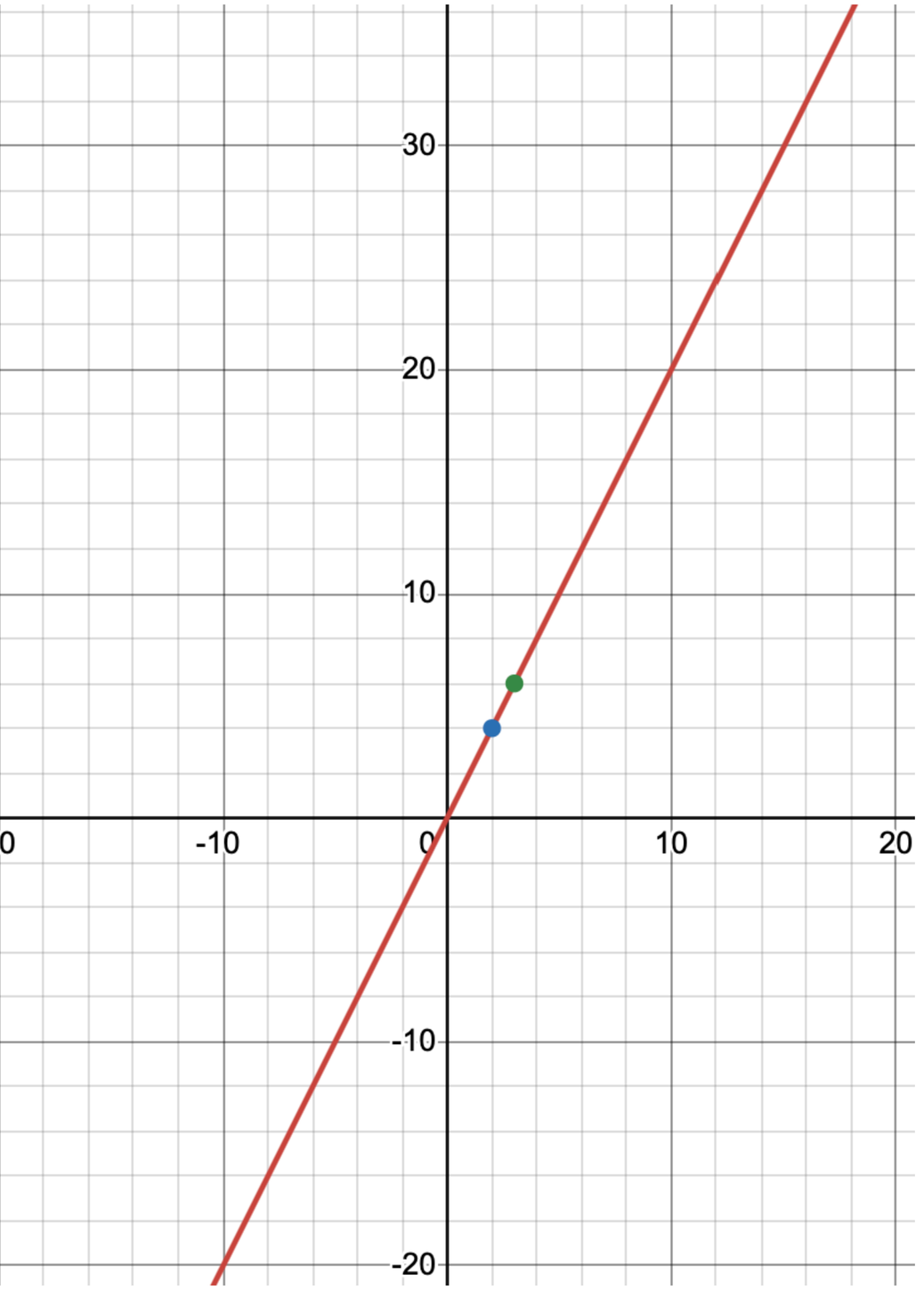


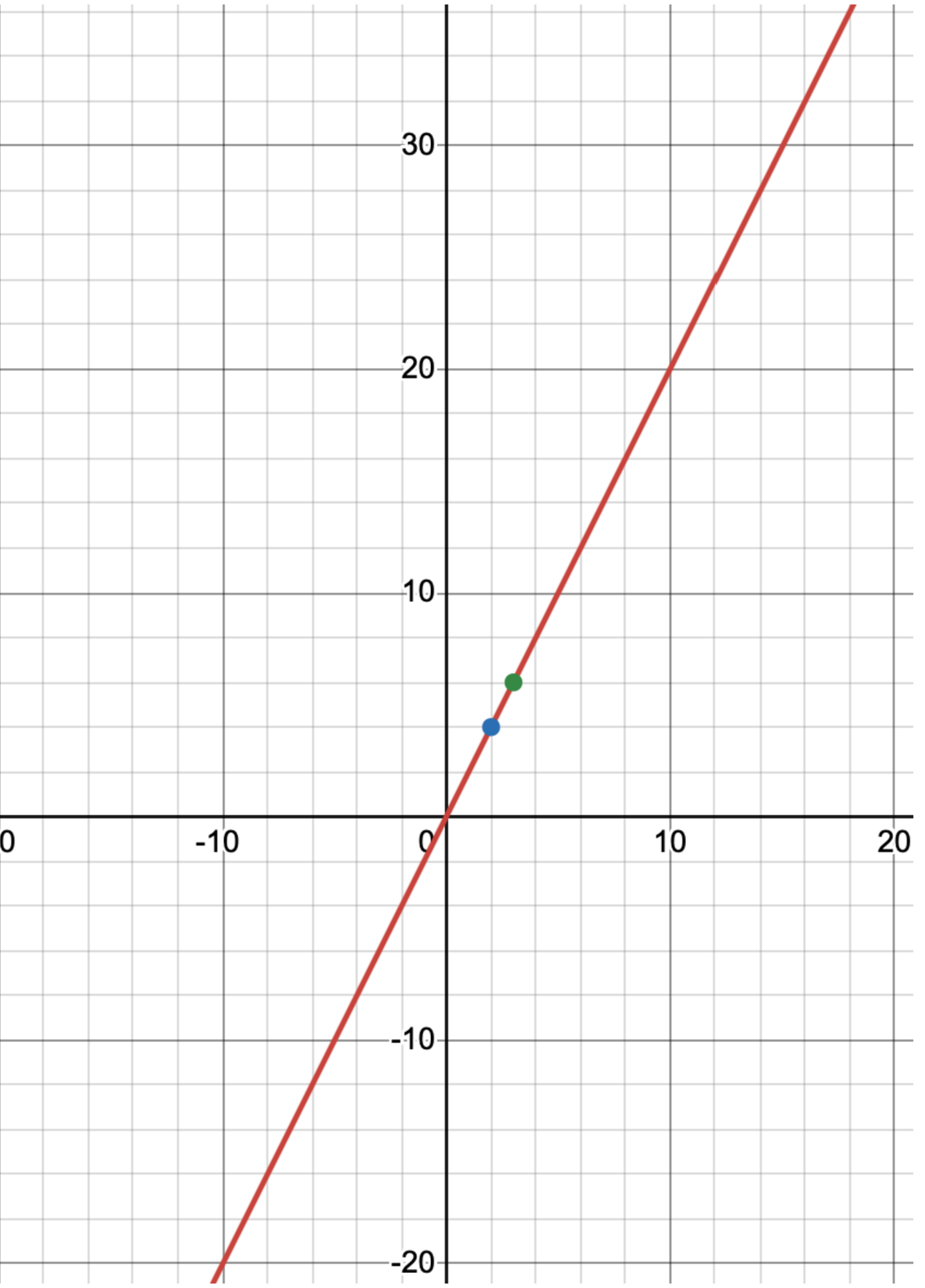
MTLTexture

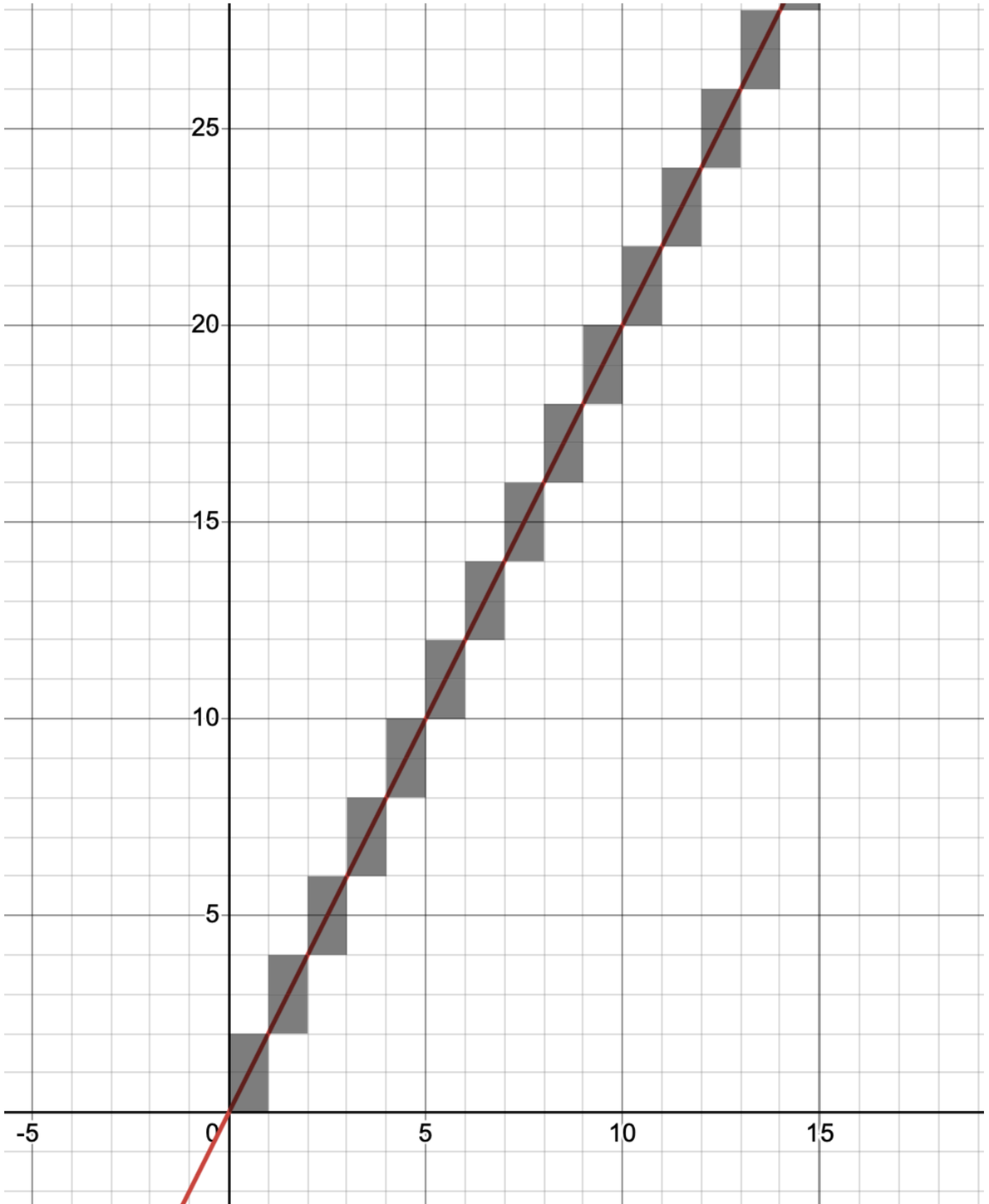


Shader

```
kernel void shader(texture2d output [[texture(0)]],
                   ...
                   uint2 gid [[thread_position_in_grid]])
{
    ...
}
```


30
20
10
0
-10
0
10
20
-10
-20


30
20
10
0
-10
20
-10
-20


25
20
15
10
5
-5
0
5
10
15

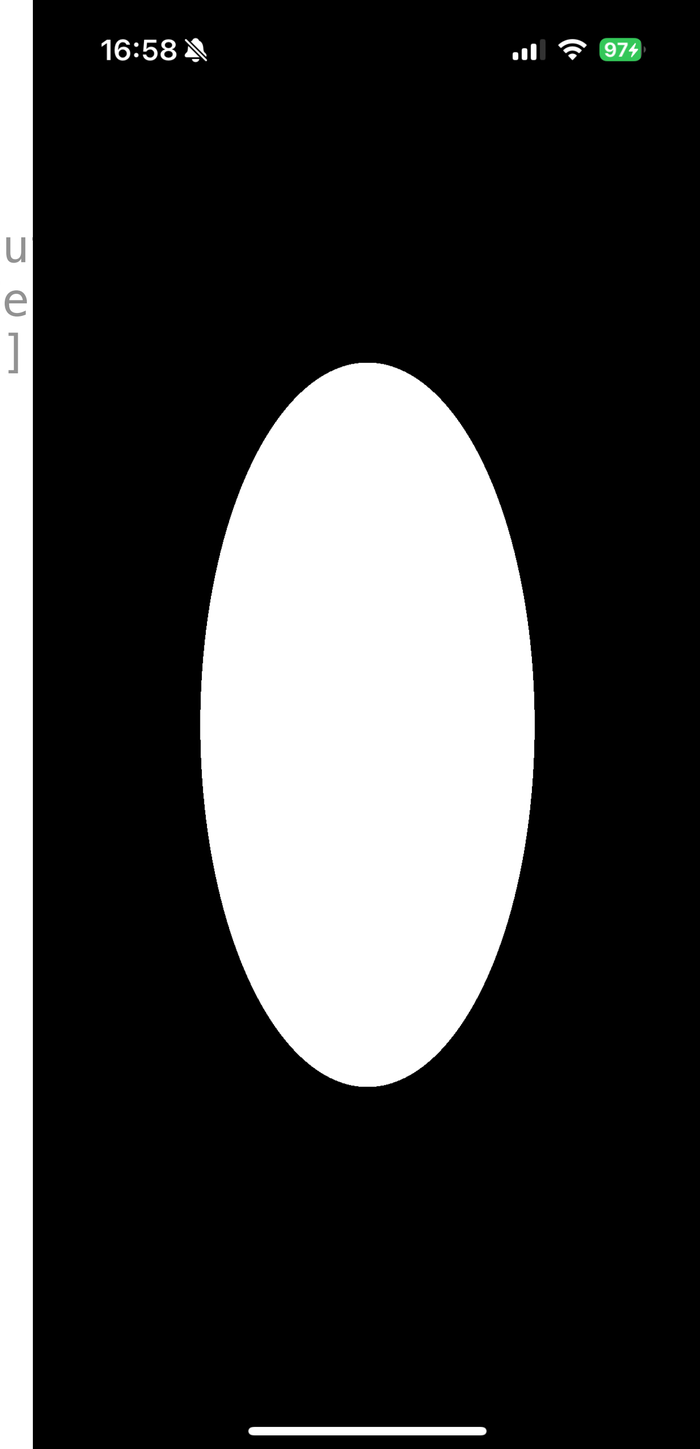
```
kernel void computeShape(texture2d<float, access::write> output [[texture(0)]],
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);
```

```
kernel void computeShape(texture2d<float, access::write> output [[texture(0)]],
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);

    float2 center = float2(0.5, 0.5);
    float rad = 0.25;

    float2 vec = uv - center;
    float dist = sqrt(vec.x * vec.x + vec.y * vec.y);

    float outputColor;

    if (dist > rad) {
        outputColor = 0;
    } else {
        outputColor = 1;
    }
```

```
kernel void computeShape(texture2d<float, access::write> output [[texture(0)]],
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);

    float2 center = float2(0.5, 0.5);
    float rad = 0.25;

    float2 vec = uv - center;
    float dist = length(vec);

    float outputColor = step(rad, dist);
```

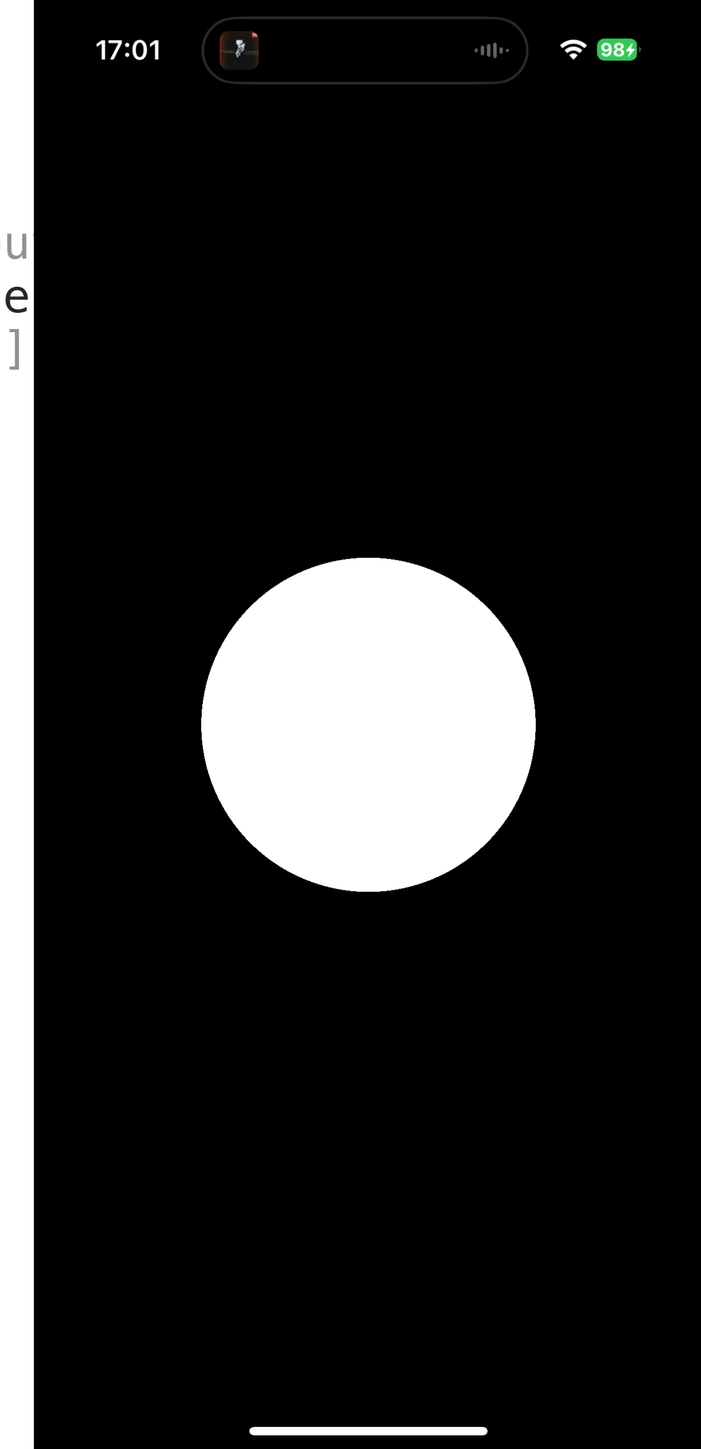
```
kernel void computeShape(texture2d<float, access::write> output [[texture(0)]],
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);

    float2 center = float2(0.5, 0.5);
    float rad = 0.25;

    float2 vec = uv - center;
    float dist = length(vec);

    float outputColor = step(rad, dist);

    output.write(outputColor, gid);
}
```

```
kernel void computeShape(texture2d<float, access::write> outpu
                         constant Uniforms & uniforms [[ buffe
                         uint2 gid [[thread_position_in_grid]]
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);

    float2 center = float2(0.5, 0.5);
    float rad = 0.25;

    float2 vec = uv - center;
    float dist = length(vec);

    float outputColor = step(rad, dist);

    output.write(outputColor, gid);
}
```



```
kernel void computeShape(texture2d<float, access::write> output [[texture(0)]],
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);

    uv = squarifyUV(uv, uniforms.aspect);
}



float2 squarifyUV(float2 uv, float aspect) {
    uv.y -= 0.5;
    uv.y /= aspect;
    uv.y += 0.5;
    return uv;
}
```

```
kernel void computeShape(texture2d<float, access::write> outpu
                         constant Uniforms & uniforms [[ buffe
                         uint2 gid [[thread_position_in_grid]]
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);

    uv = squarifyUV(uv, uniforms.aspect);
}



float2 squarifyUV(float2 uv, float aspect) {
    uv.y -= 0.5;
    uv.y /= aspect;
    uv.y += 0.5;
    return uv;
}
```



```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
typedef struct {
    float aspect;
} Uniforms;
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```swift
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

```
class MetalView: MTKView, MTKViewDelegate {
    var uniforms = Uniforms()
    var commandQueue: MTLCommandQueue
    var pipelineState: MTLComputePipelineState!

    override init(frame frameRect: CGRect, device: (any MTLDevice)?) {
        self.commandQueue = device!.makeCommandQueue()!
        super.init(frame: frameRect, device: device)
        self.pipelineState = self.makePipelineState("computeShape")
        self.delegate = self
        self.framebufferOnly = false
    }

    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
        uniforms.aspect = Float(size.width / size.height)
    }

    func draw(in view: MTKView) {
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
    }
```

Uniforms

```
typedef struct {
    float aspect;
} Uniforms;
```

or

```
struct Uniforms {
    var aspect: Float
}
```

Uniforms

```
typedef struct {
    float aspect;
} Uniforms;
```

or

```
struct Uniforms {
    var aspect: Float
}
```

Texture

```
class MTLTexture {
    var _bytes: SomePixelData
}
```

Uniforms

```
typedef struct {
    float aspect;
} Uniforms;
```

or

```
struct Uniforms {
    var aspect: Float
}
```

Texture

```
class MTLTexture {
    var _bytes: SomePixelData
}
```

Shader

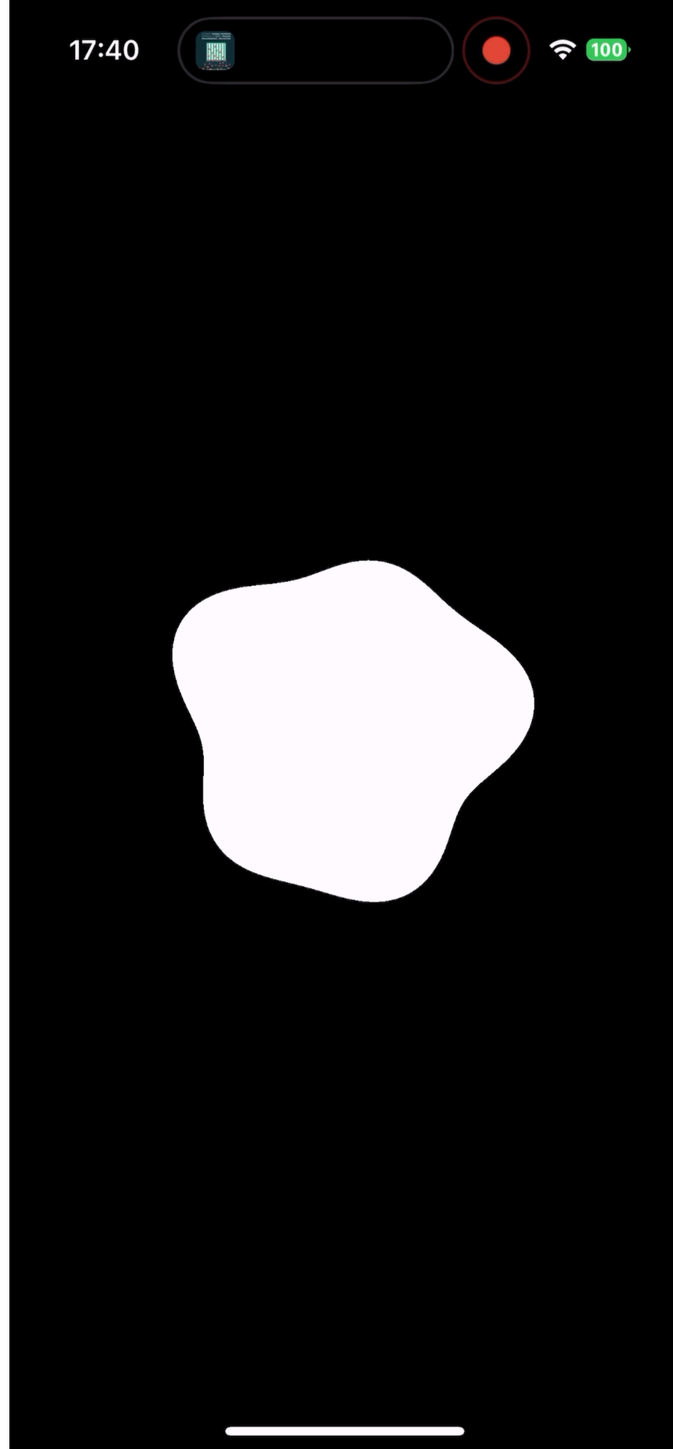
```
kernel void computeShape(texture2d outputTexture,
                         Uniforms uniforms,
                         uint2 position) {
   some shading code
}
```

```
func draw(in view: MTKView) {
        uniforms.time += 1 / preferredFramesPerSecond
        let buf = commandQueue.makeCommandBuffer()!
        let drawable = view.currentDrawable!

        let command = buf.makeComputeCommandEncoder()!
        command.setComputePipelineState(pipelineState)
        command.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
        command.setTexture(drawable.texture, index: 0)
        let size = MTLSize(width: drawable.texture.width, height: drawable.texture.height, depth: 1)
        let width = pipelineState.threadExecutionWidth
        let height = pipelineState.maxTotalThreadsPerThreadgroup / width
        let threadsPerGroup = MTLSizeMake(width, height, 1)

        command.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
        command.endEncoding()

        buf.present(drawable)
        buf.commit()
}
```

```
kernel void computeShape(texture2d<float, access::write> output,
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);
    uv = squarifyUV(uv, uniforms.aspect);

    float2 center = float2(0.5, 0.5);
    float rad = 0.25;

    float2 vec = uv - center;
    float dist = length(vec);

    float angle = atan2(vec.x, vec.y);
    float distortion = sin(angle * 5 + uniforms.time);
    distortion += cos(angle * 3 - uniforms.time);
    rad += distortion * 0.02;

    float outputColor = step(dist, rad);

    output.write(outputColor, gid);
}
```

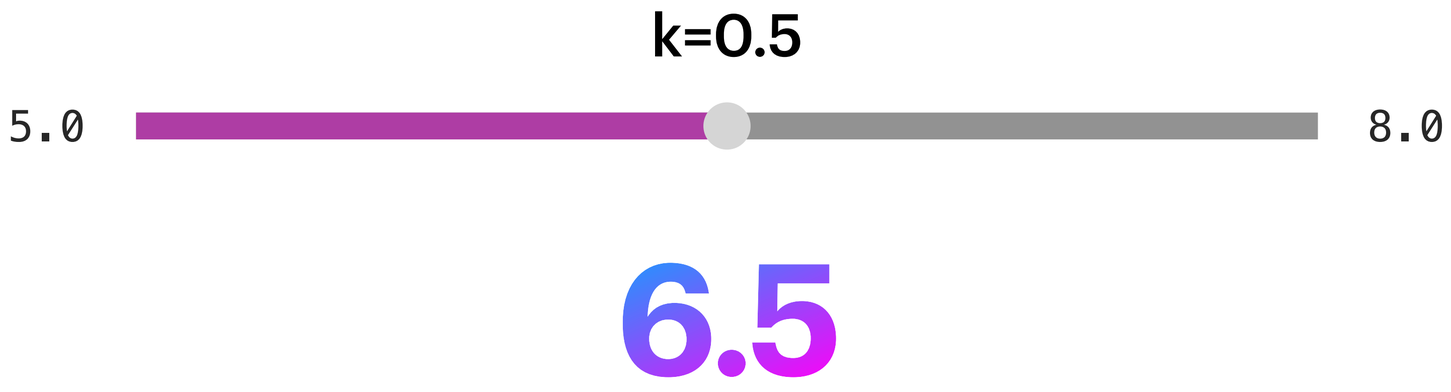
```
kernel void computeShape(texture2d<float, access::write> output,
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);
    uv = squarifyUV(uv, uniforms.aspect);

    float2 center = float2(0.5, 0.5);
    float rad = 0.25;

    float2 vec = uv - center;
    float dist = length(vec);

    float angle = atan2(vec.x, vec.y);
    float distortion = sin(angle * 5 + uniforms.time);
    distortion += cos(angle * 3 - uniforms.time);
    rad += distortion * 0.02;

    float outputColor = step(dist, rad);

    output.write(outputColor, gid);
}
```

```
kernel void computeShape(texture2d<float, access::write> output,
                         constant Uniforms & uniforms [[ buffer(0) ]],
                         uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 pos = float2(gid);
    float2 uv = pos / float2(width, height);
    uv = squarifyUV(uv, uniforms.aspect);

    float2 center = float2(0.5, 0.5);
    float rad = 0.25;

    float2 vec = uv - center;
    float dist = length(vec);

    float angle = atan2(vec.x, vec.y);
    float distortion = sin(angle * 5 + uniforms.time);
    distortion += cos(angle * 3 - uniforms.time);
    rad += distortion * 0.02;

    float outputColor = step(dist, rad);

    output.write(outputColor, gid);
}
```



```
extension Float: Interpolatable {
    public static func lerp(_ l: Float, _ r: Float, _ k: Float) -> Float {
        l + k * (r - l)
    }
}
```

```
extension Float: Interpolatable {
    public static func lerp(_ l: Float, _ r: Float, _ k: Float) -> Float {
        l + k * (r - l)
    }
}
```



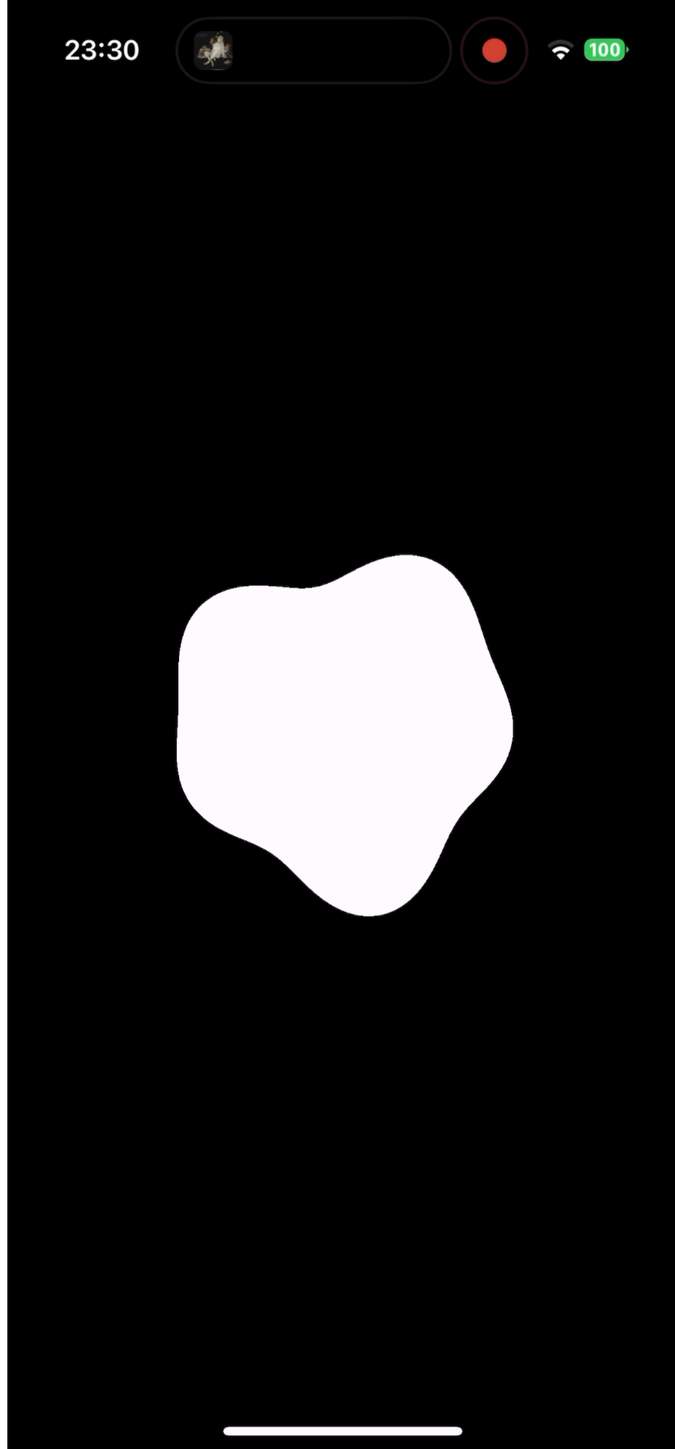
```
extension Float: Interpolatable {
    public static func lerp(_ l: Float, _ r: Float, _ k: Float) -> Float {
        l + k * (r - l)
    }
}
```

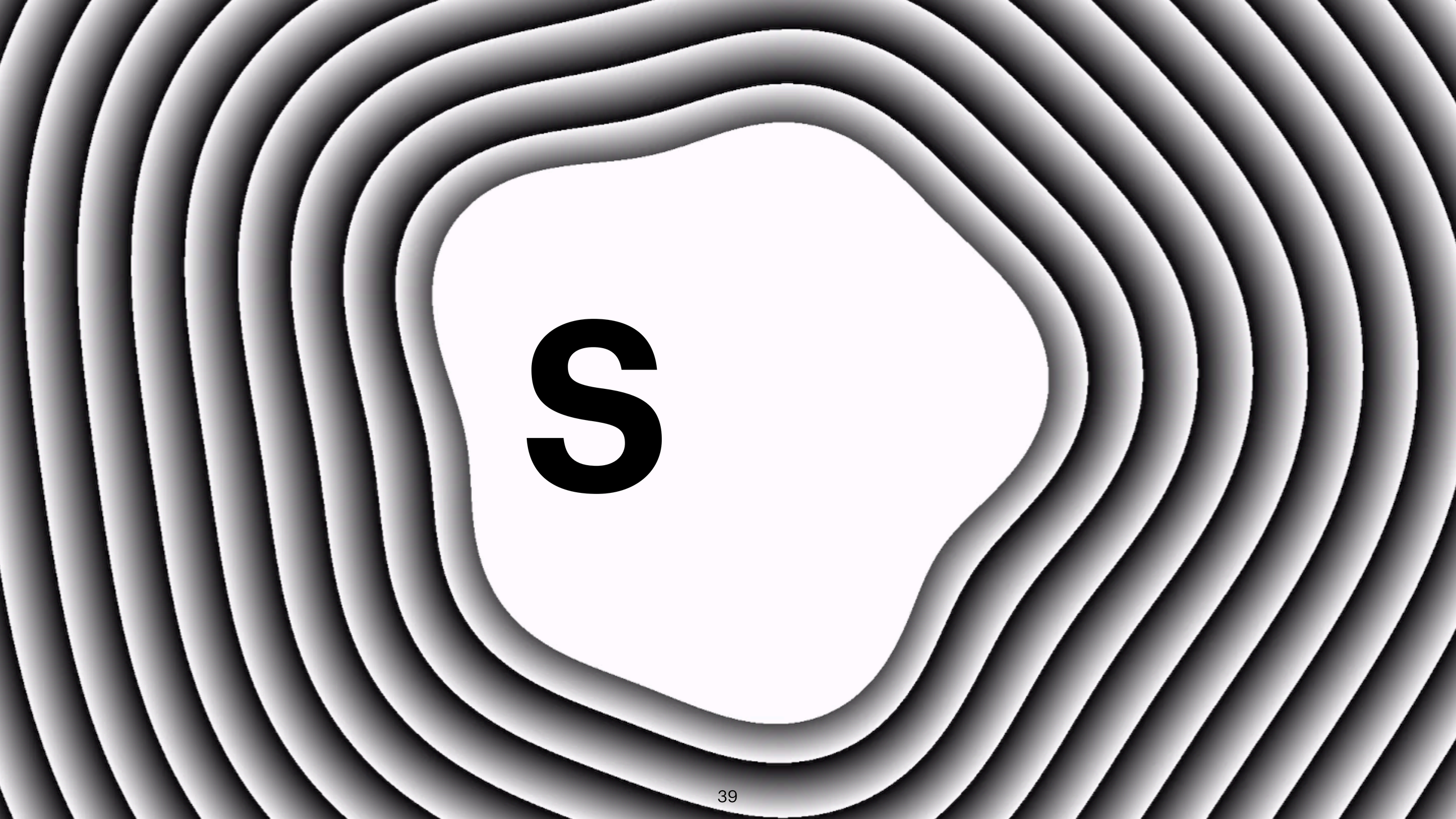
```swift
// MARK: MetalView.swift
    var animationStartTime: Float = -Float.greatestFiniteMagnitude // initial low value
    var animationDuration: Float = 1 // 1 second

    func startAnimation() {
        animationStartTime = uniforms.time
    }

    func draw(in view: MTKView) {
        uniforms.time += 1 / Float(preferredFramesPerSecond)

        let deltaTime = uniforms.time - animationStartTime
        let progress = Float(deltaTime / animationDuration)
        if progress < 1 {
            uniforms.rad = Float.lerp(0.25, 0.5, progress)
        }
```

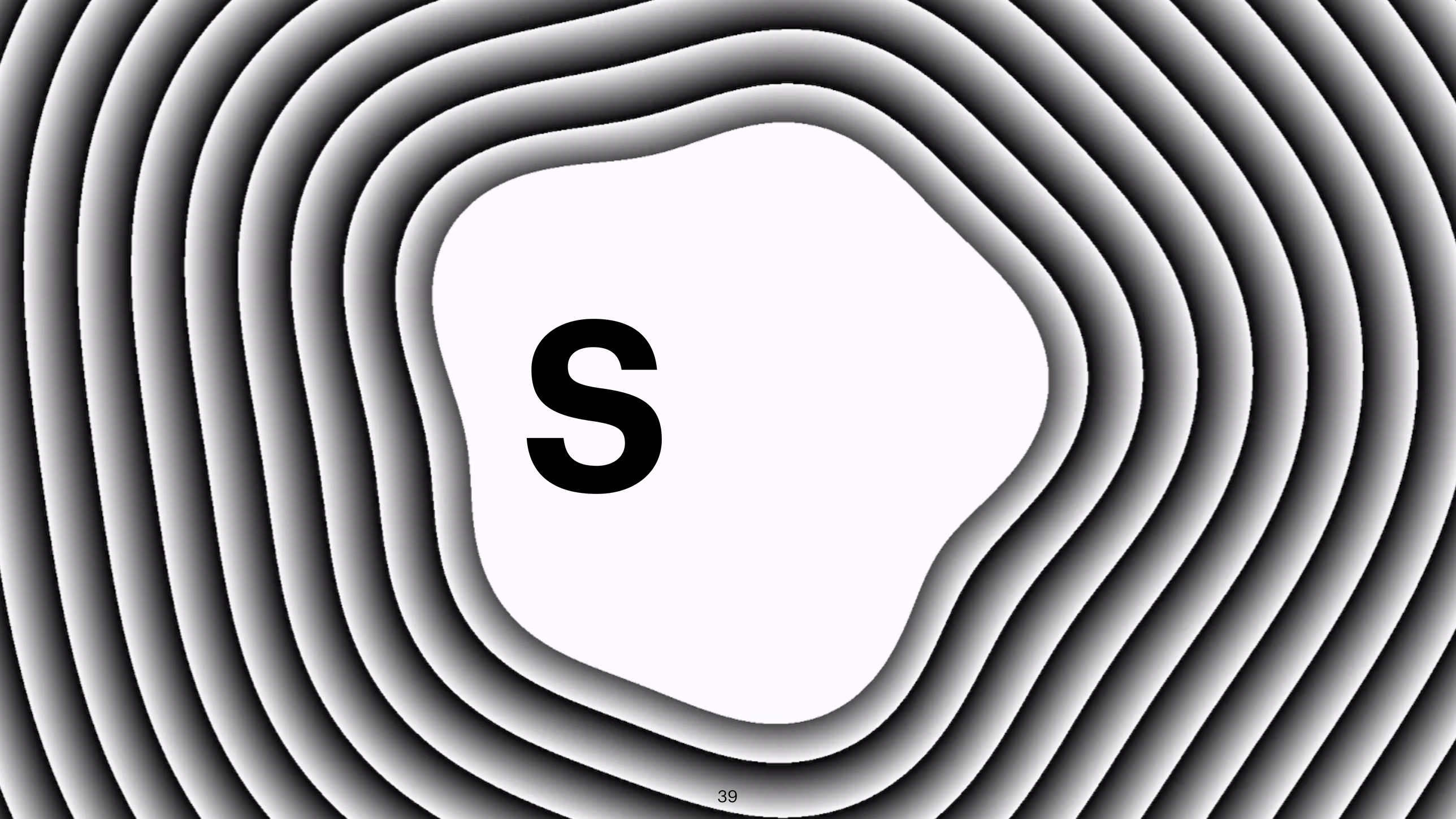
```
    kernel void computeShape(...) {
        ...

        float2 center = float2(0.5, 0.5);

    //    float rad = 0.25;
        float rad = uniforms.rad;

        ...

        float outputColor = step(dist, rad);
        output.write(outputColor, gid);
    }
```

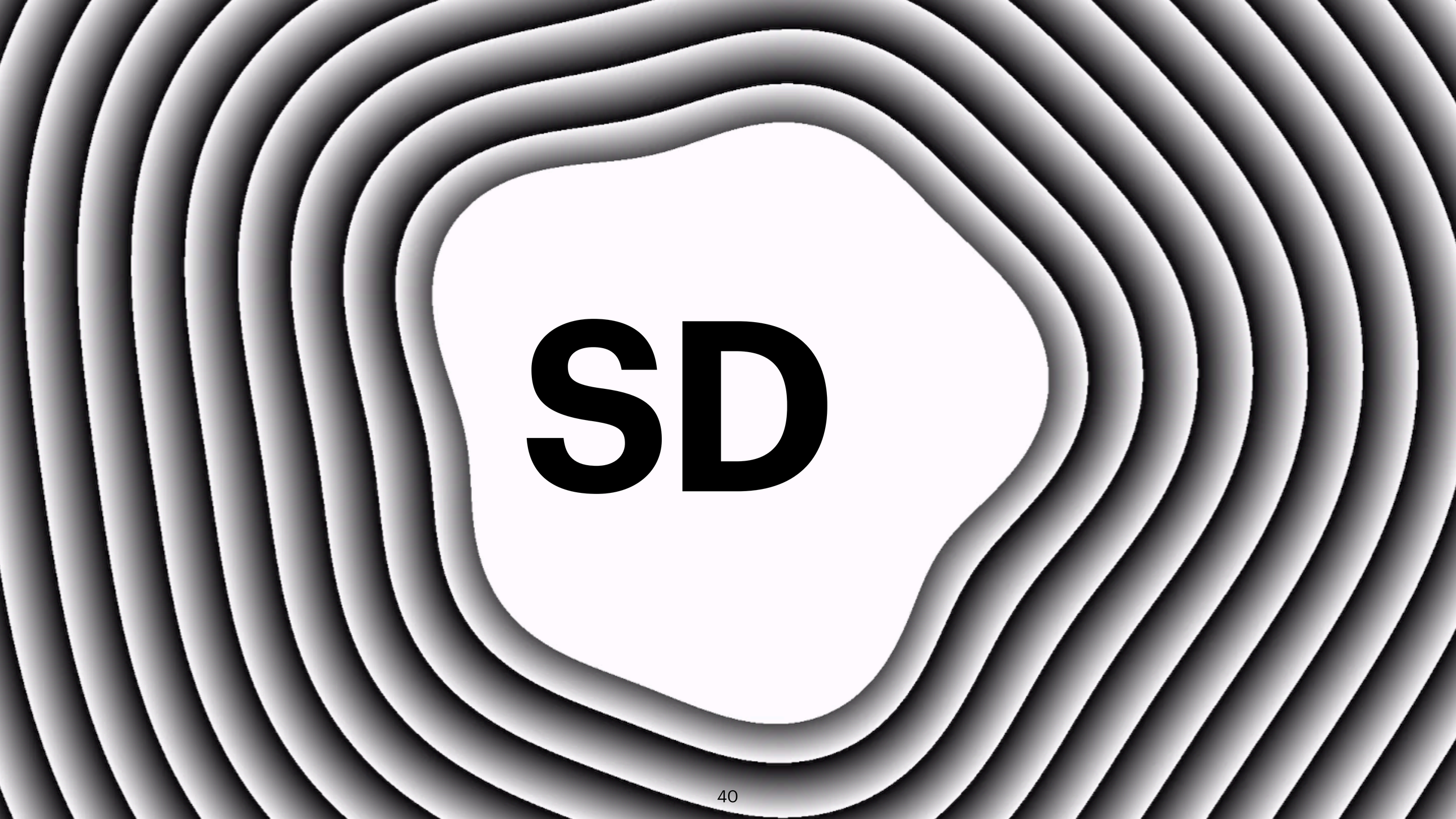
```swift
// MARK: MetalView.swift
    var animationStartTime: Float = -Float.greatestFiniteMagnitude // initial low value
    var animationDuration: Float = 1 // 1 second

    func startAnimation() {
        animationStartTime = uniforms.time
    }

    func draw(in view: MTKView) {
        uniforms.time += 1 / Float(preferredFramesPerSecond)

        let deltaTime = uniforms.time - animationStartTime
        let progress = Float(deltaTime / animationDuration)
        if progress < 1 {
            uniforms.rad = Float.lerp(0.25, 0.5, progress)
        }
```

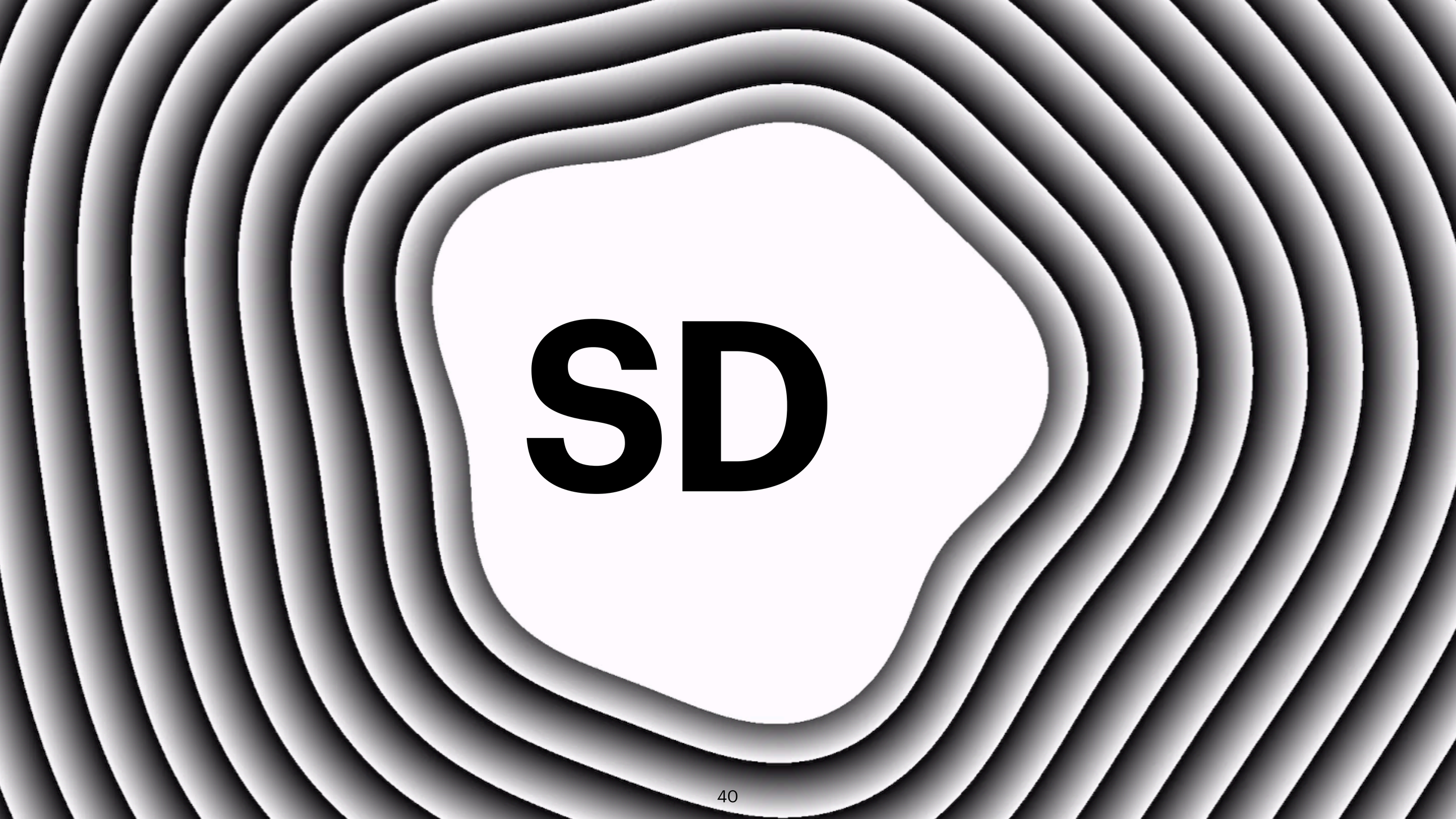
```swift
    kernel void computeShape(...) {
        ...

        float2 center = float2(0.5, 0.5);

    //    float rad = 0.25;
        float rad = uniforms.rad;

        ...

        float outputColor = step(dist, rad);
        output.write(outputColor, gid);
    }
```




S


S


SD


SD


SDF


SDF

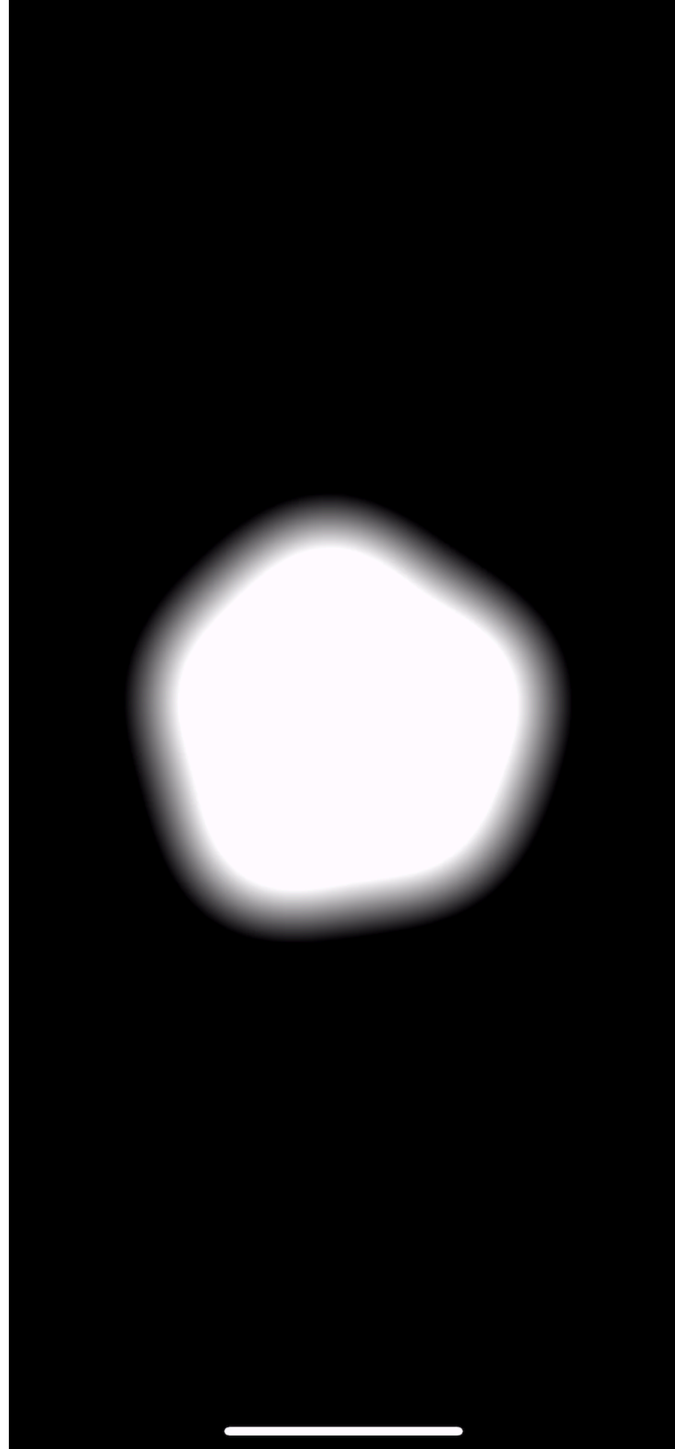

SDF
Signed Distance Field


SDF
Signed Distance Field

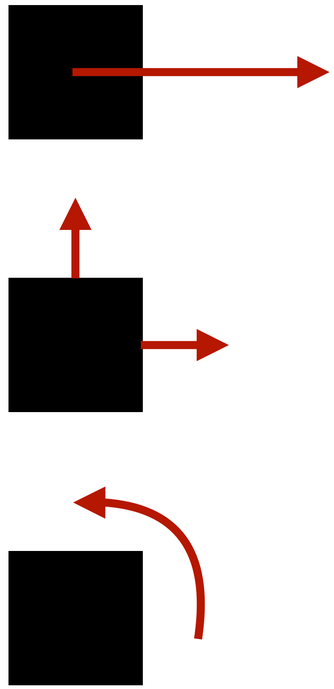
```
float circleSDF(float2 vec, float rad, float time) {
    float angle = atan2(vec.x, vec.y);
    float distortion = sin(angle * 5 + time);
    distortion += cos(angle * 3 - time);
    rad += distortion * 0.01;
    return length(vec) - rad;
}

kernel void computeShape(...) {
    ...
    float2 center = float2(0.5, 0.5);
    float2 vec = uv - center;

    float dist = circleSDF(vec, uniforms.rad, uniforms.time);

    if (dist > 0) {
        float glow = dist / (uniforms.rad / 3);
        glow = 1 - glow; // invert
        outputColor = mix(0, 1, glow);
    } else {
        outputColor = 1;
    }

    output.write(outputColor, gid);
}
```

```
float circleSDF(float2 vec, float rad, float time) {
    float angle = atan2(vec.x, vec.y);
    float distortion = sin(angle * 5 + time);
    distortion += cos(angle * 3 - time);
    rad += distortion * 0.01;
    return length(vec) - rad;
}

kernel void computeShape(...) {
    ...
    float2 center = float2(0.5, 0.5);
    float2 vec = uv - center;

    float dist = circleSDF(vec, uniforms.rad, uniforms.time);

    if (dist > 0) {
        float glow = dist / (uniforms.rad / 3);
        glow = 1 - glow; // invert
        outputColor = mix(0, 1, glow);
    } else {
        outputColor = 1;
    }

    output.write(outputColor, gid);
}
```

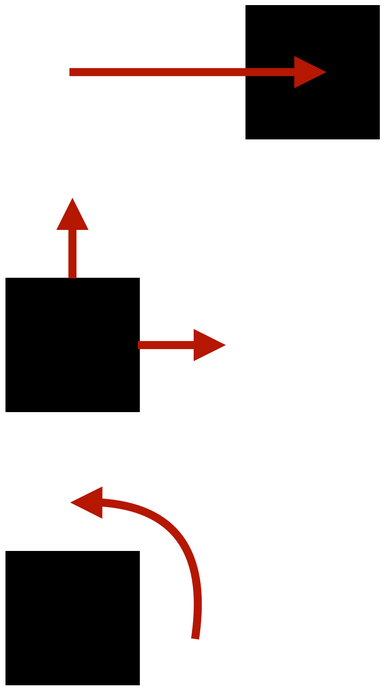
```
float circleSDF(float2 vec, float rad, float time) {
    float angle = atan2(vec.x, vec.y);
    float distortion = sin(angle * 5 + time);
    distortion += cos(angle * 3 - time);
    rad += distortion * 0.01;
    return length(vec) - rad;
}

kernel void computeShape(...) {
    ...
    float2 center = float2(0.5, 0.5);
    float2 vec = uv - center;

    float dist = circleSDF(vec, uniforms.rad, uniforms.time);

    if (dist > 0) {
        float glow = dist / (uniforms.rad / 3);
        glow = 1 - glow; // invert
        outputColor = mix(0, 1, glow);
    } else {
        outputColor = 1;
    }

    output.write(outputColor, gid);
}
```



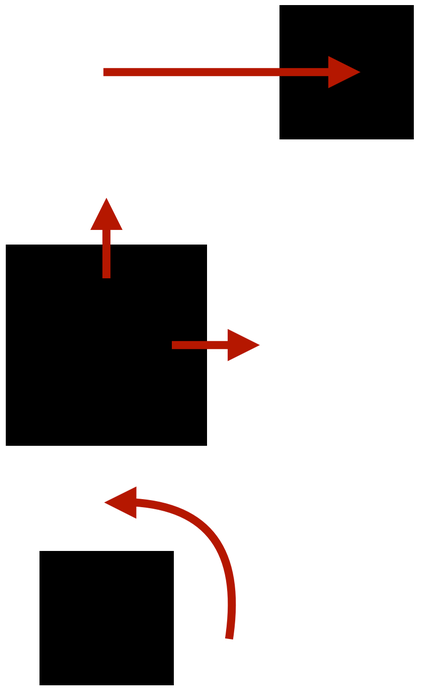
```
float2 translate(float2 pos, float2 translation) {
    return pos + translation;
}

float2 scale(float2 pos, half scale) {
    return pos / scale;
}

float2 rot(float2 pos, float angle) {
    half sine = sin(angle);
    half cosine = cos(angle);
    return float2(cosine * pos.x + sine * pos.y,
                  cosine * pos.y - sine * pos.x);
}
```



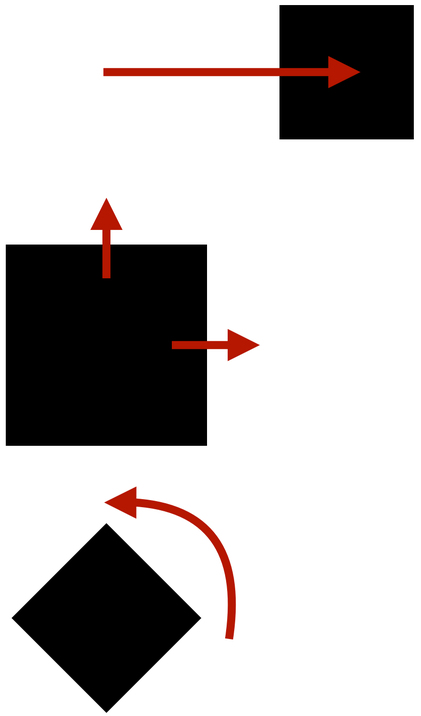
```
float2 translate(float2 pos, float2 translation) {
    return pos + translation;
}

float2 scale(float2 pos, half scale) {
    return pos / scale;
}

float2 rot(float2 pos, float angle) {
    half sine = sin(angle);
    half cosine = cos(angle);
    return float2(cosine * pos.x + sine * pos.y,
                  cosine * pos.y - sine * pos.x);
}
```



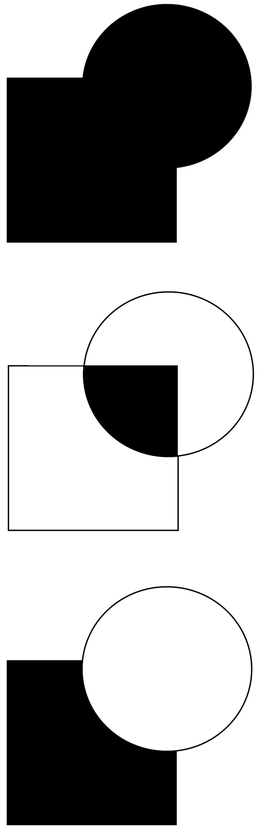
```
float2 translate(float2 pos, float2 translation) {
    return pos + translation;
}

float2 scale(float2 pos, half scale) {
    return pos / scale;
}

float2 rot(float2 pos, float angle) {
    half sine = sin(angle);
    half cosine = cos(angle);
    return float2(cosine * pos.x + sine * pos.y,
                  cosine * pos.y - sine * pos.x);
}
```



```
float2 translate(float2 pos, float2 translation) {
    return pos + translation;
}

float2 scale(float2 pos, half scale) {
    return pos / scale;
}

float2 rot(float2 pos, float angle) {
    half sine = sin(angle);
    half cosine = cos(angle);
    return float2(cosine * pos.x + sine * pos.y,
                  cosine * pos.y - sine * pos.x);
}
```



```
float unionSDF(float d1, float d2) {
    return min(d1, d2);
}


float intersectionSDF(float d1, float d2) {
    return max(d1, d2);
}


float differenceSDF(float d1, float d2) {
    return max(d1, -d2);
}
```



```
float fireTopSDF(float2 pos, float rad, float time, float direction) {
    float trianglesCount = 6;

    // определяем, насколько мы далеко от горизонтального центра
    float distFromCenter = (pos.x * pos.x) / rad * 1.5;
    pos.y -= distFromCenter / 2;

    // добавляем немного асимметрии
    if (direction > 0) { pos.x += sins(time) * 0.05; }

    // поворачиваем треугольники наружу от центра
    pos = rot(pos, -pos.x * 0.1);

    // двигаем координату X в центр огня
    pos.x += sign(pos.x) * fract(time) / trianglesCount;

    // задаём несколько треугольников
    pos.x = fract(pos.x * trianglesCount) - 0.5;

    // смещаем треугольники выше
    pos.y += rad;

    // определяем ширину основания треугольника, чем дальше от центра — тем меньше
    float height = 1 - distFromCenter;

    float width = 0.5 / rad;
    return triangleSDF(pos, float2(width, height));
}
```

```
float fireTopSDF(float2 pos, float rad, float time, float direction) {
    float trianglesCount = 6;

    // определяем, насколько мы далеко от горизонтального центра
    float distFromCenter = (pos.x * pos.x) / rad * 1.5;
    pos.y -= distFromCenter / 2;

    // добавляем немного асимметрии
    if (direction > 0) { pos.x += sins(time) * 0.05; }

    // поворачиваем треугольники наружу от центра
    pos = rot(pos, -pos.x * 0.1);

    // двигаем координату X в центр огня
    pos.x += sign(pos.x) * fract(time) / trianglesCount;

    // задаём несколько треугольников
    pos.x = fract(pos.x * trianglesCount) - 0.5;

    // смещаем треугольники выше
    pos.y += rad;

    // определяем ширину основания треугольника, чем дальше от центра — тем меньше
    float height = 1 - distFromCenter;

    float width = 0.5 / rad;
    return triangleSDF(pos, float2(width, height));
}
```

```
float fireTopSDF(float2 pos, float rad, float time, float direction) { ... }

float fireSDF(float2 pos, float rad, float time) {
    // добавляем distortion
    pos.x += sins(pos.y * 10.0 + time) * 0.015;
    pos.y += coss(pos.x * 10.0 + time * 0.7) * 0.01;

    float fireTopDist = fireTopSDF(pos, rad, time, 1);;

    // модифицируем координаты для параболы
    float2 parabolaPos = pos;
    parabolaPos.y *= -1;
    parabolaPos.y += rad * 1.25;

    // 2.5 — коэффициент кривизны параболы
    float parabolaDist = parabolaSDF(parabolaPos, 2.5);

    return smoothIntersection(parabolaDist, fireTopDist, 0.1);
}
```

```
float fireTopSDF(float2 pos, float rad, float time, float direction) { ... }

float fireSDF(float2 pos, float rad, float time) {
    // добавляем distortion
    pos.x += sins(pos.y * 10.0 + time) * 0.015;
    pos.y += coss(pos.x * 10.0 + time * 0.7) * 0.01;

    float fireTopDist = fireTopSDF(pos, rad, time, 1);;

    // модифицируем координаты для параболы
    float2 parabolaPos = pos;
    parabolaPos.y *= -1;
    parabolaPos.y += rad * 1.25;

    // 2.5 — коэффициент кривизны параболы
    float parabolaDist = parabolaSDF(parabolaPos, 2.5);

    return smoothIntersection(parabolaDist, fireTopDist, 0.1);
}
```

```swift
// MARK: MetalView.swift
    var currentShapeIsCircle: Bool = true
    var animationStartTime: Float = -Float.greatestFiniteMagnitude // initial low value
    var animationDuration: Float = 1 // 1 second

    func swapShape() {
        currentShapeIsCircle.toggle()
        startAnimation()
    }

    func draw(in view: MTKView) {
        uniforms.time += 1 / Float(preferredFramesPerSecond)

        let deltaTime = uniforms.time - animationStartTime
        let progress = Float(deltaTime / animationDuration)
        if progress < 1 {
            let fromMul = Float.lerp(1, 0, progress)
            let toMul = Float.lerp(0, 1, progress)

            if currentShapeIsCircle {
                uniforms.circleMul = fromMul
                uniforms.fireMul = toMul
            } else {
                uniforms.fireMul = fromMul
                uniforms.circleMul = toMul
            }
        }
```

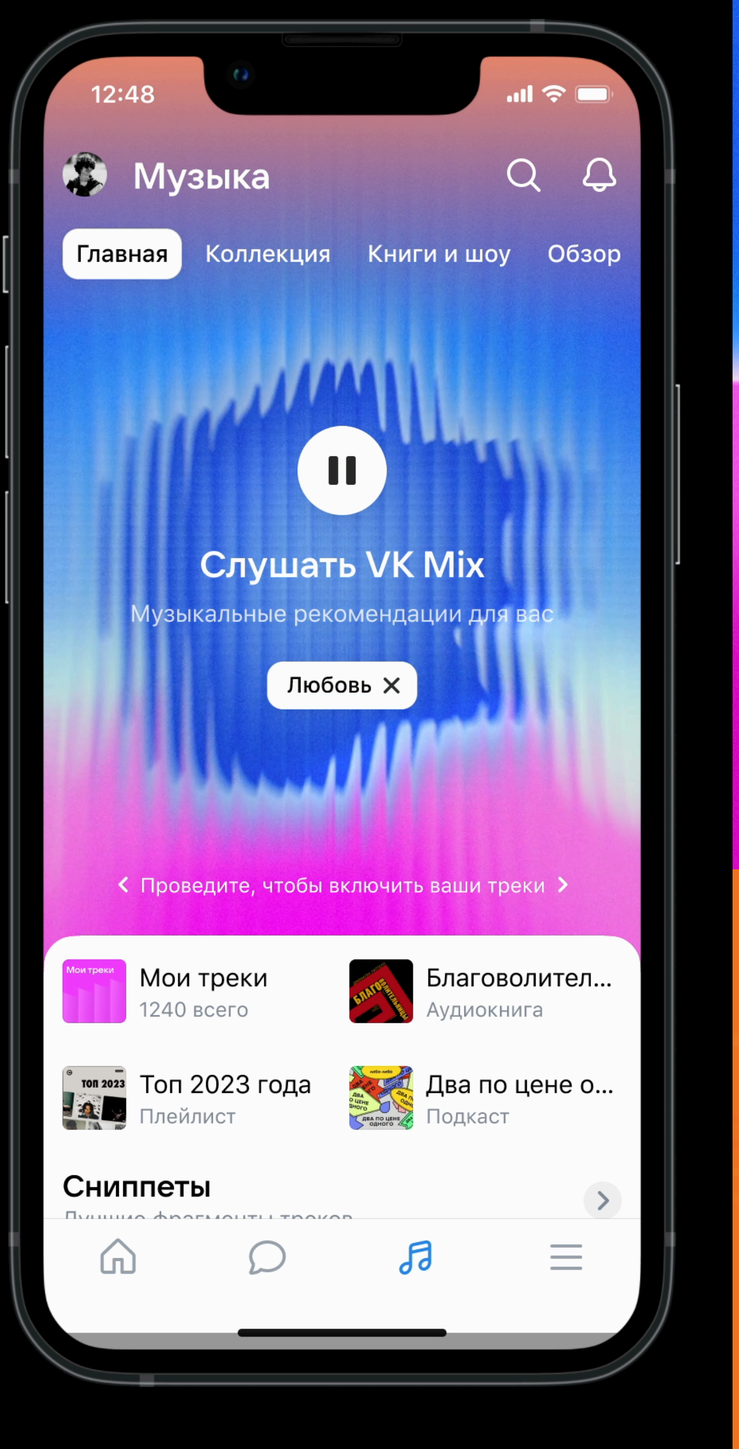
```
typedef struct
{
    float aspect;
    float time;
    float rad;
    float circleMul;
    float fireMul;
} Uniforms;
```

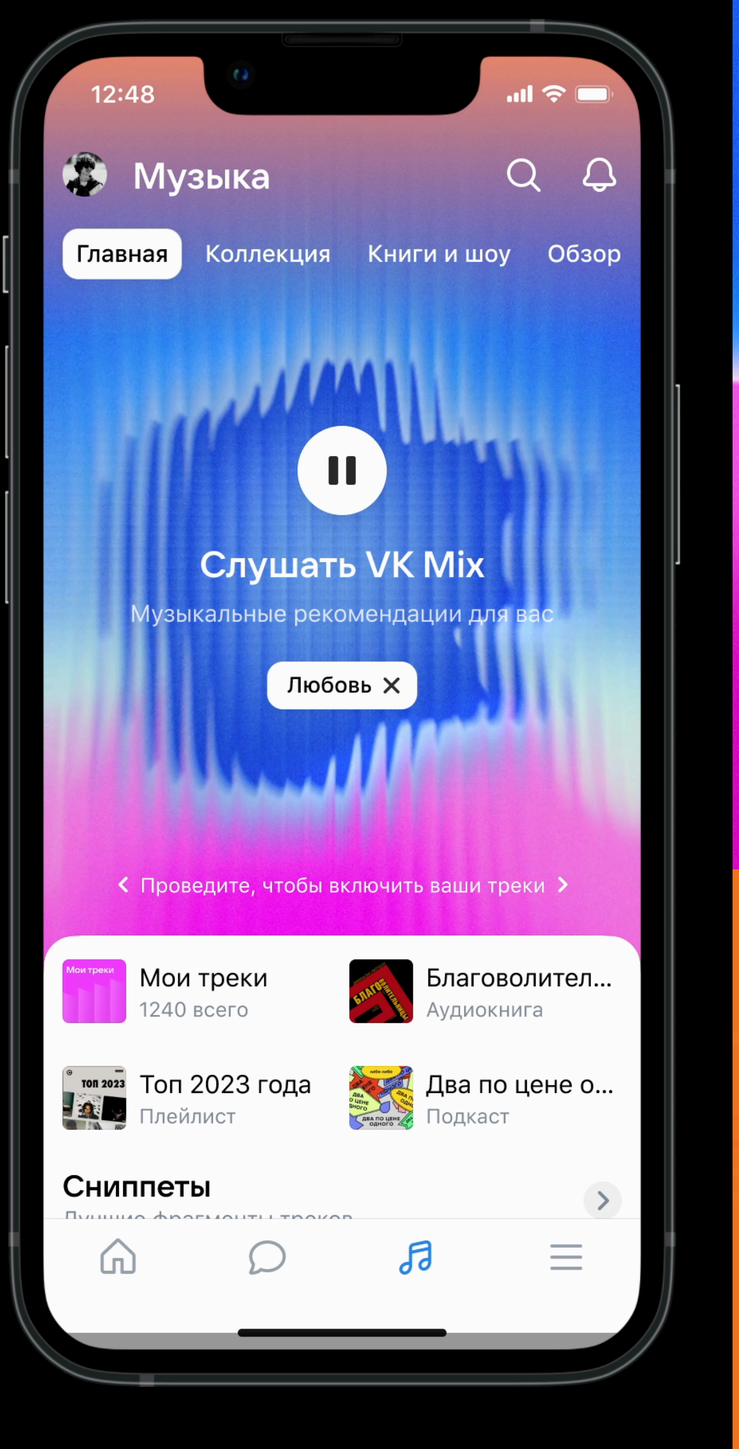
```
kernel void computeShape(...) {
    ...
    float dist = 0;

    if (uniforms.circleMul > 0) {
        dist += circleSDF(vec, uniforms.rad, uniforms.time) * uniforms.circleMul;
    }
    if (uniforms.fireMul > 0) {
        dist += fireSDF(vec, uniforms.rad, uniforms.time) * uniforms.fireMul;
    }

    float outputColor = step(dist, 0);
    ...
    output.write(outputColor, gid);
}
```

```swift
// MARK: MetalView.swift
    var currentShapeIsCircle: Bool = true
    var animationStartTime: Float = -Float.greatestFiniteMagnitude // initial low value
    var animationDuration: Float = 1 // 1 second

    func swapShape() {
        currentShapeIsCircle.toggle()
        startAnimation()
    }

    func draw(in view: MTKView) {
        uniforms.time += 1 / Float(preferredFramesPerSecond)

        let deltaTime = uniforms.time - animationStartTime
        let progress = Float(deltaTime / animationDuration)
        if progress < 1 {
            let fromMul = Float.lerp(1, 0, progress)
            let toMul = Float.lerp(0, 1, progress)

            if currentShapeIsCircle {
                uniforms.circleMul = fromMul
                uniforms.fireMul = toMul
            } else {
                uniforms.fireMul = fromMul
                uniforms.circleMul = toMul
            }
        }
```

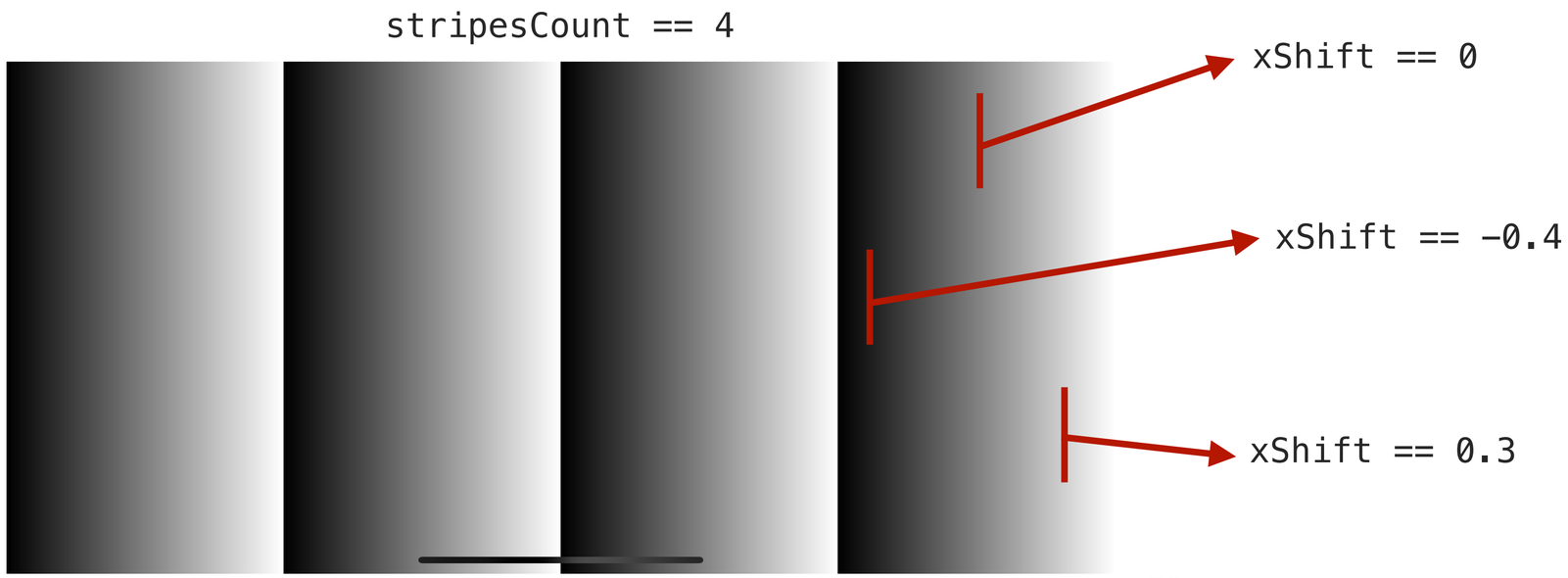
```
typedef struct
{
    float aspect;
    float time;
    float rad;
    float circleMul;
    float fireMul;
} Uniforms;
```

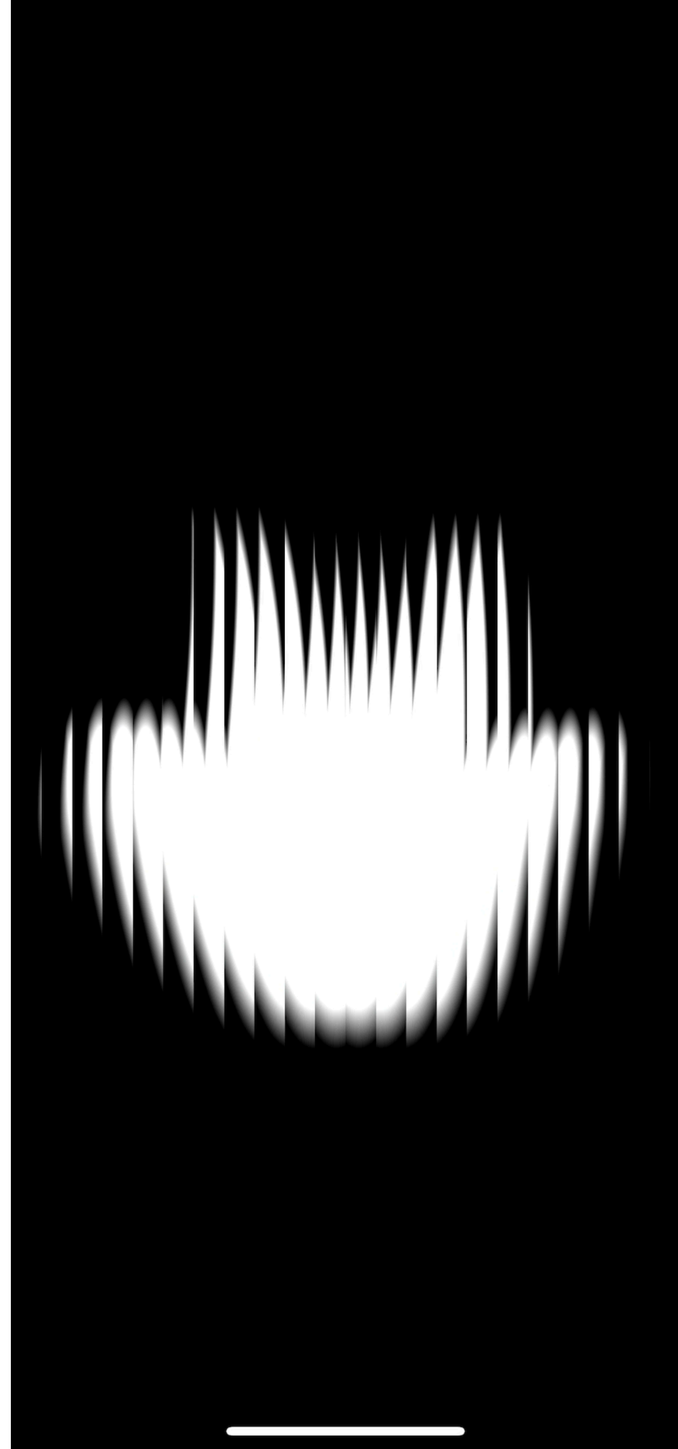
```
kernel void computeShape(...) {
    ...
    float dist = 0;

    if (uniforms.circleMul > 0) {
        dist += circleSDF(vec, uniforms.rad, uniforms.time) * uniforms.circleMul;
    }
    if (uniforms.fireMul > 0) {
        dist += fireSDF(vec, uniforms.rad, uniforms.time) * uniforms.fireMul;
    }

    float outputColor = step(dist, 0);
    ...
    output.write(outputColor, gid);
}
```


12:48
Музыка
Главная
Коллекция
Книги и шоу
Обзор
Слушать VK Mix
Музыкальные рекомендации для вас
Любовь
Проведите, чтобы включить ваши треки
Мои треки
1240 всего
Благоволител...
Аудиокнига
Топ 2023 года
Плейлист
Два по цене о...
Подкаст
Сниппеты

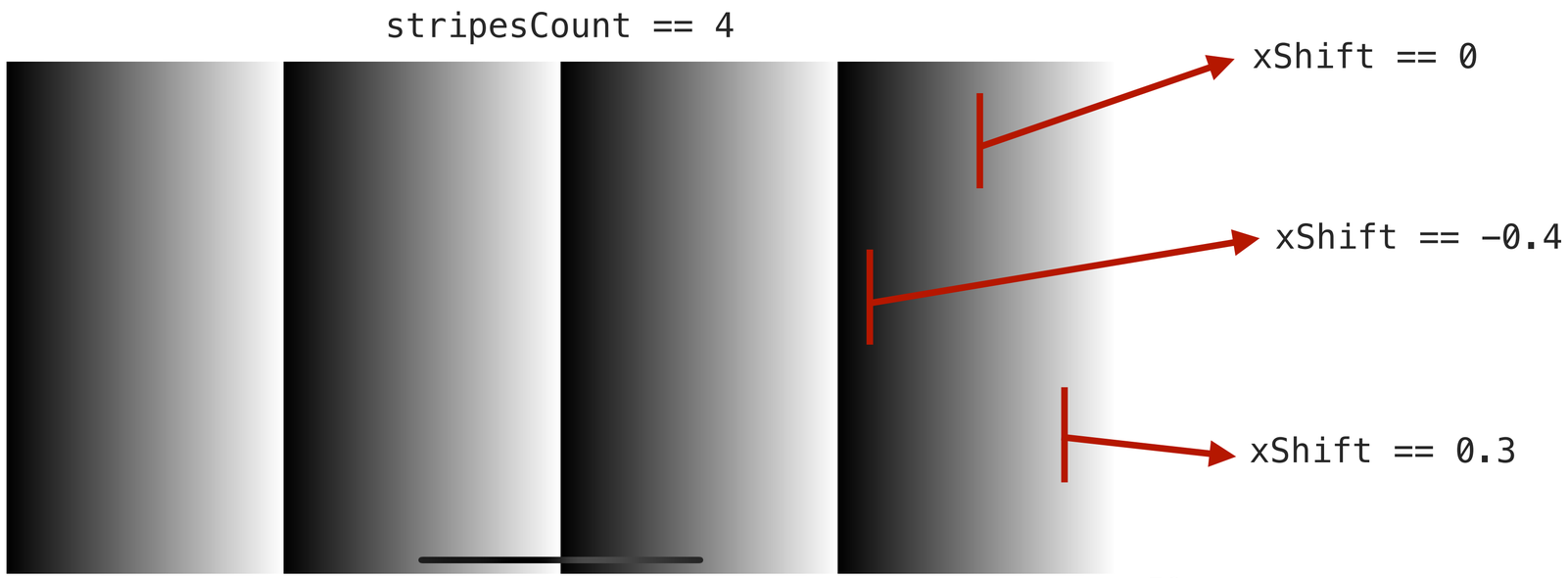

12:48
Музыка
Главная
Коллекция
Книги и шоу
Обзор
Слушать VK Mix
Музыкальные рекомендации для вас
Любовь
Проведите, чтобы включить ваши треки
Мои треки
1240 всего
Благоволител...
Аудиокнига
Топ 2023 года
Плейлист
Два по цене о...
Подкаст
Сниппеты

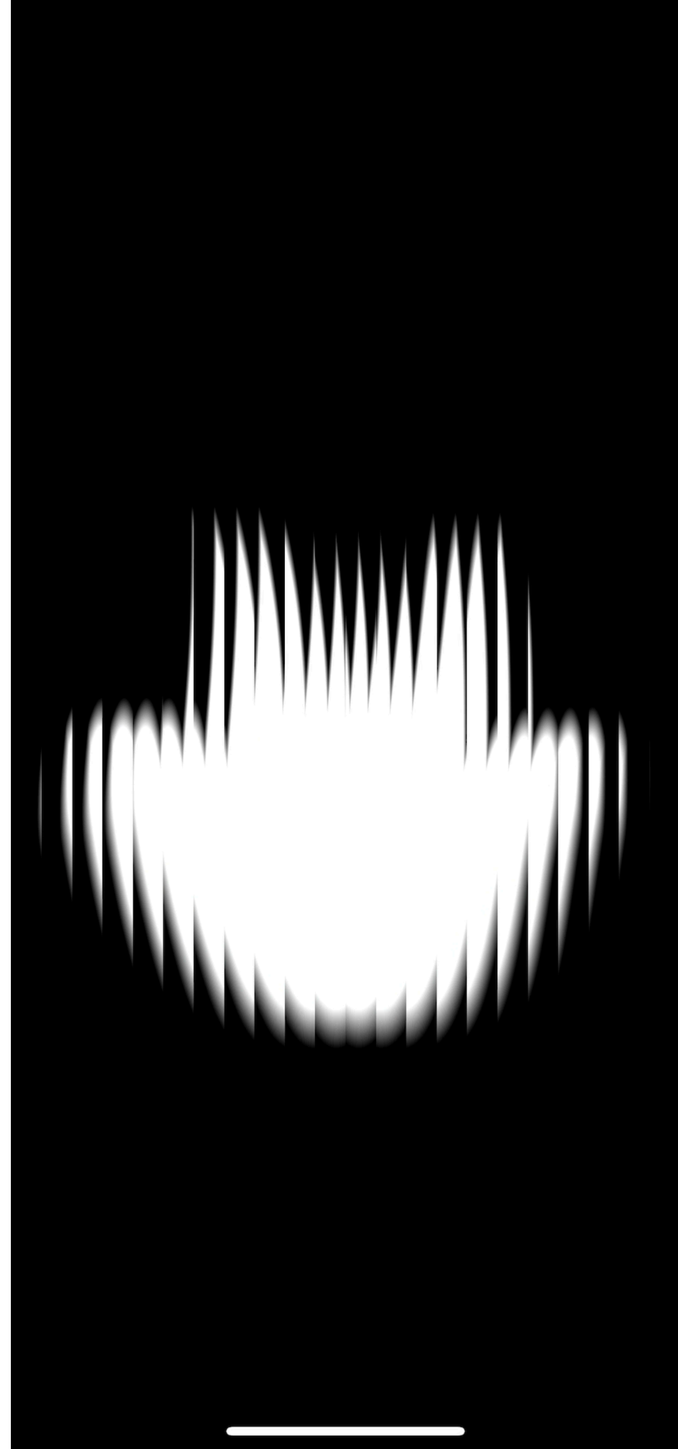
```
#define GLASS_REFRACTION 0.125

float2 glass(float2 in) {
    float stripesCount = 22;
    float xShift = fract(in.x * stripesCount) - 0.5;
    in.x += xShift * GLASS_REFRACTION;
    return in;
}


kernel void computeShape(...) {
    ...
    uv = glass(uv);
    ...
}
```



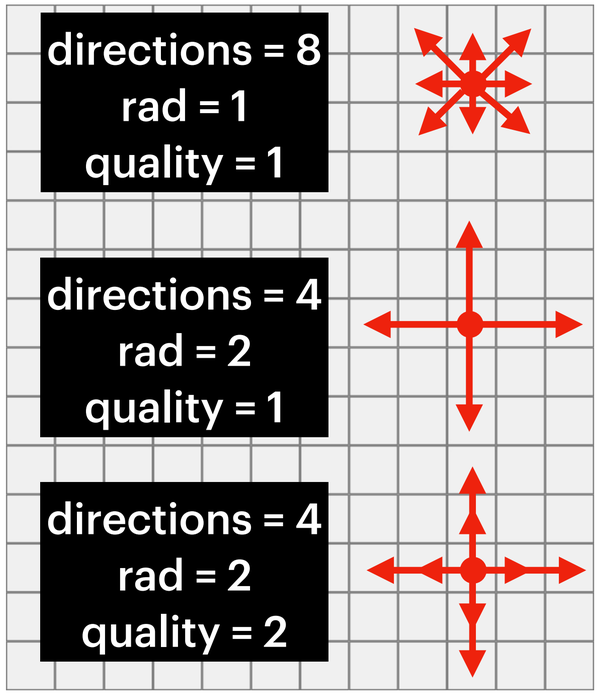


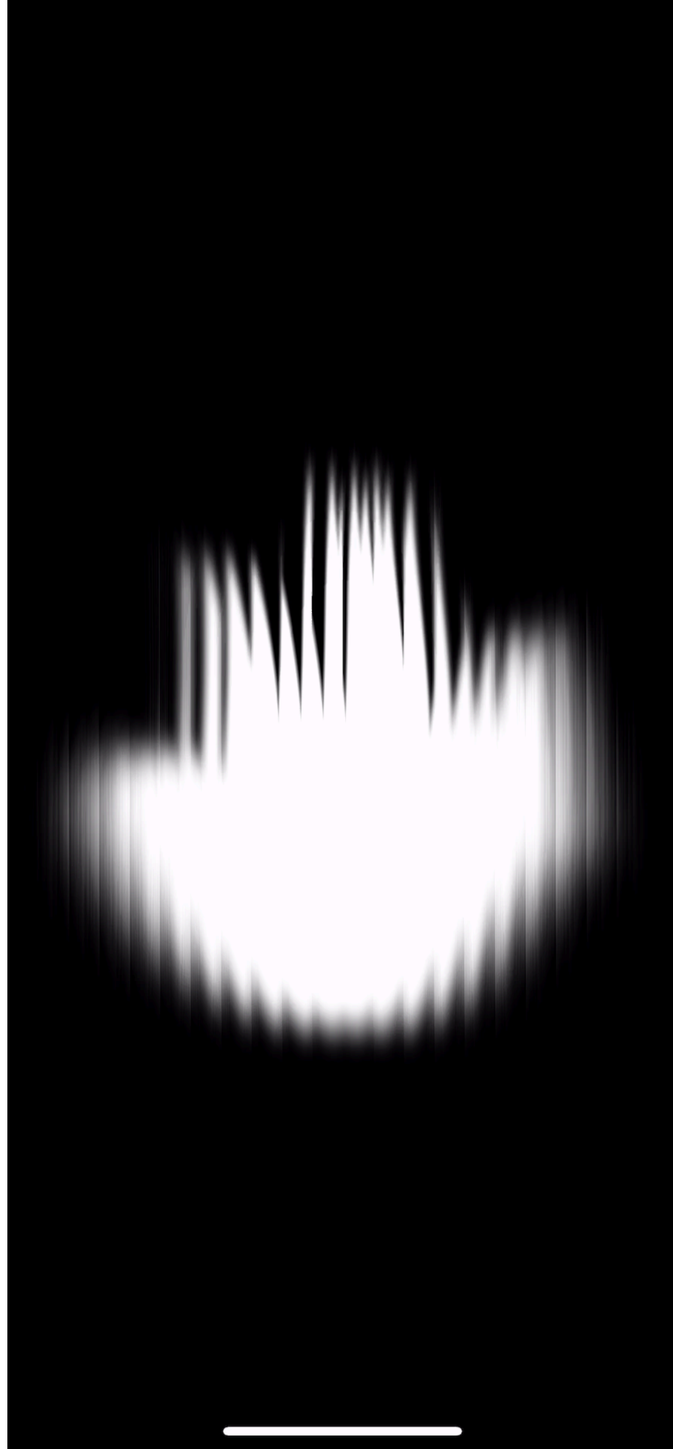
```
#define GLASS_REFRACTION 0.125

float2 glass(float2 in) {
    float stripesCount = 22;
    float xShift = fract(in.x * stripesCount) - 0.5;
    in.x += xShift * GLASS_REFRACTION;
    return in;
}


kernel void computeShape(...) {
    ...
    uv = glass(uv);
    ...
}
```



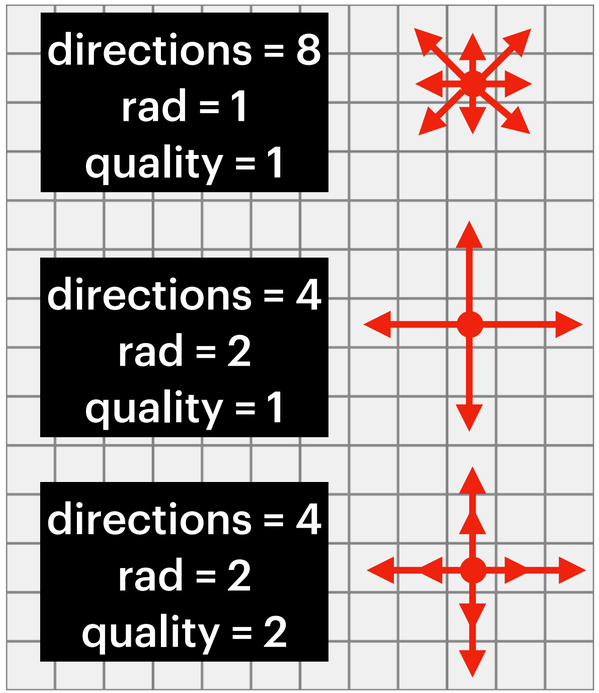


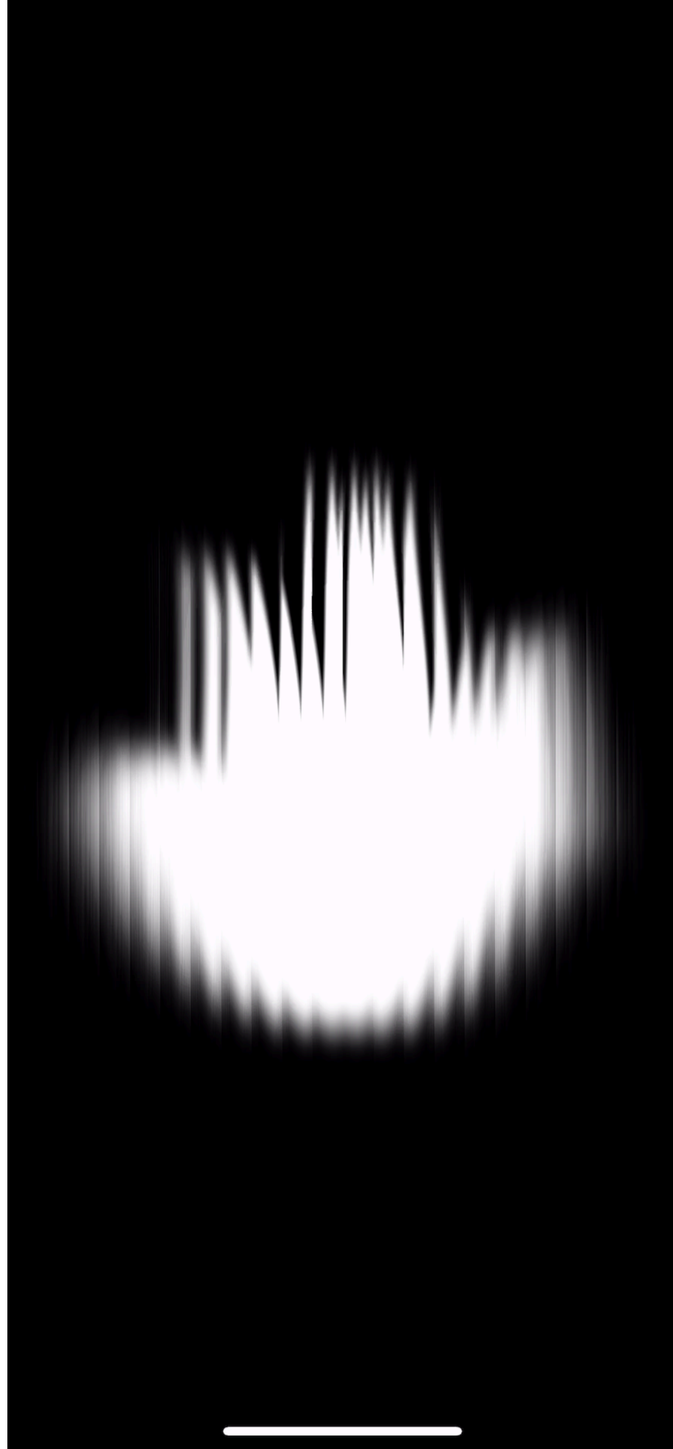
```
kernel void blurCompute(texture2d<half, access::write> output [[texture(0)]],
                        texture2d<half, access::sample> input [[texture(1)]],
                        uint2 gid [[thread_position_in_grid]]) {

    int width = output.get_width();
    int height = output.get_height();
    float2 uv = float2(gid) / float2(width, height);
    constexpr sampler s(coord::normalized, address::clamp_to_edge,
filter::nearest);

    float dist = max(0.0, length(uv - 0.5));

    half directions = 8;
    half quality = 4;
    half rad = max(0.0h, half(dist) / 10);

    half3 col = input.sample(s, uv).rgb;

    if (rad > 0) {
        half twopi = M_PI_F * 2;
        half step = twopi / directions;
        for (half a = 0; a < twopi; a += step) {
            for (half i = 1./quality; i <= 1.; i += 1./quality) {
                float2 coord = uv + float2(cos(a), sin(a)) * rad * i;
                col += input.sample(s, coord).rgb;
            }
        }
        col /= quality * directions + 1;
    }

    output.write(half4(col, 1), gid);
}
```



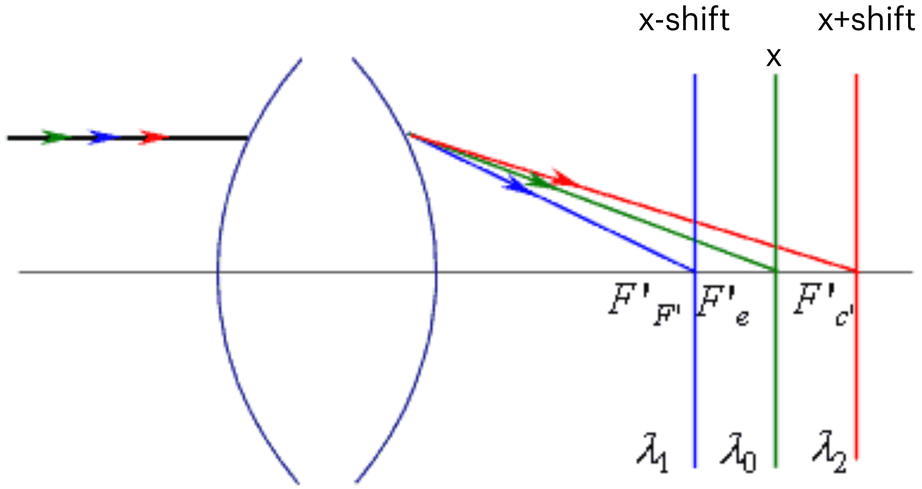


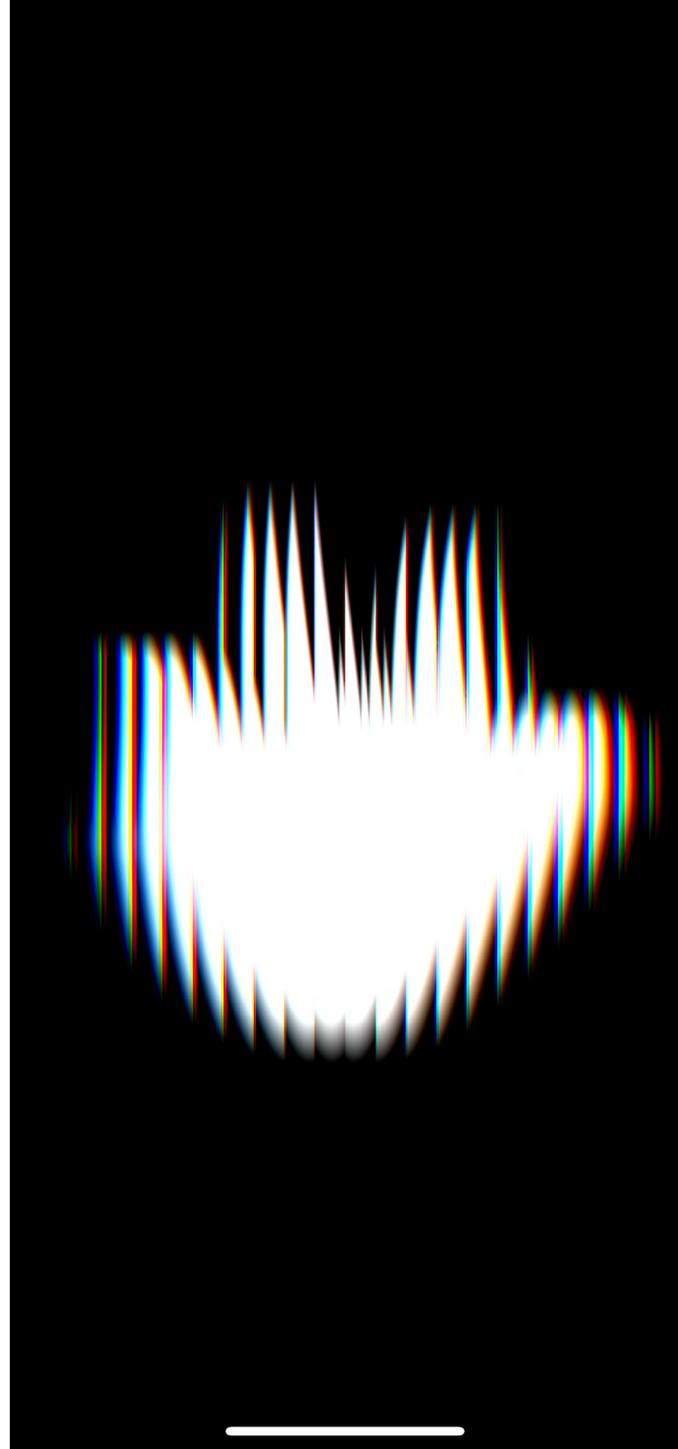
```
kernel void blurCompute(texture2d<half, access::write> output [[texture(0)]],
                        texture2d<half, access::sample> input [[texture(1)]],
                        uint2 gid [[thread_position_in_grid]]) {

    int width = output.get_width();
    int height = output.get_height();
    float2 uv = float2(gid) / float2(width, height);
    constexpr sampler s(coord::normalized, address::clamp_to_edge,
filter::nearest);

    float dist = max(0.0, length(uv - 0.5));

    half directions = 8;
    half quality = 4;
    half rad = max(0.0h, half(dist) / 10);

    half3 col = input.sample(s, uv).rgb;

    if (rad > 0) {
        half twopi = M_PI_F * 2;
        half step = twopi / directions;
        for (half a = 0; a < twopi; a += step) {
            for (half i = 1./quality; i <= 1.; i += 1./quality) {
                float2 coord = uv + float2(cos(a), sin(a)) * rad * i;
                col += input.sample(s, coord).rgb;
            }
        }
        col /= quality * directions + 1;
    }

    output.write(half4(col, 1), gid);
}
```



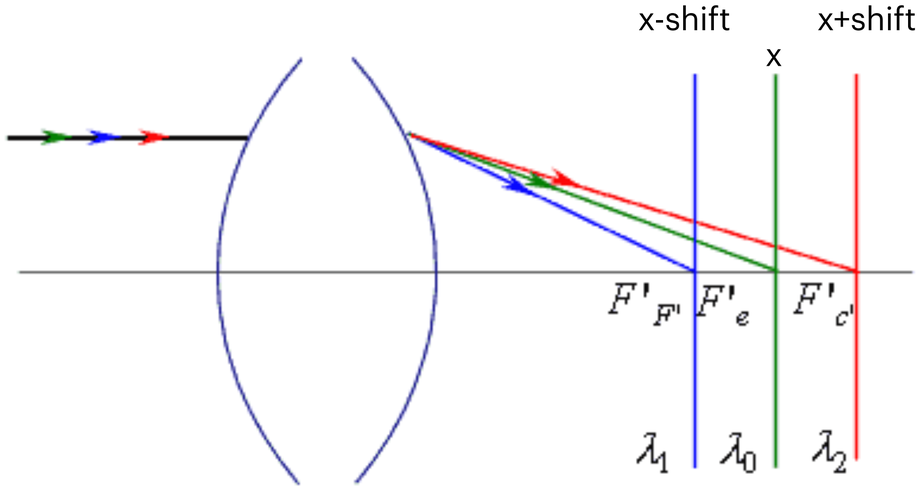


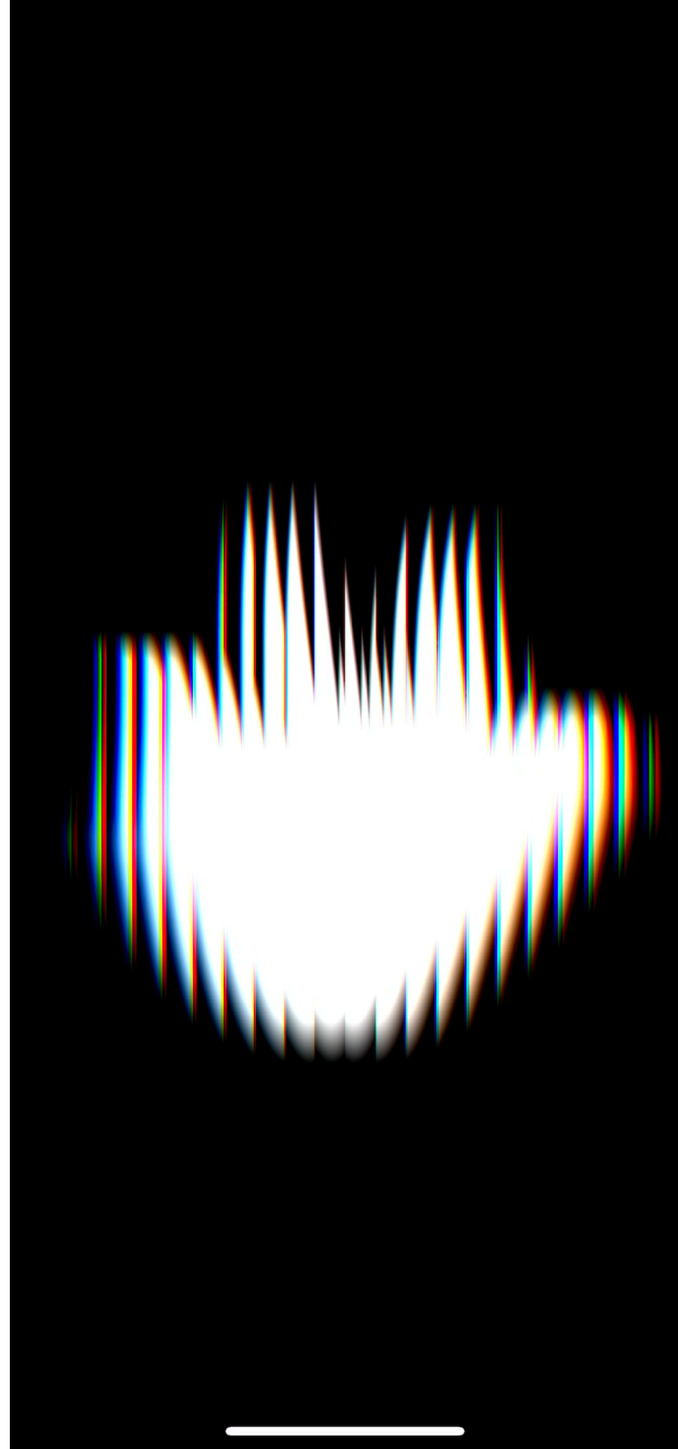
```
kernel void aberrationCompute(texture2d<half, access::write> output [[texture(0)]],
                              texture2d<half, access::sample> input [[texture(1)]],
                              uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 uv = float2(gid) / float2(width, height);

    constexpr sampler s(coord::normalized, address::clamp_to_edge, filter::nearest);

    half abShift = abs(uv.x - 0.5) * 0.02;
    float2 uvr = uv;
    uvr.x -= abShift;
    float2 uvb = uv;
    uvb.x += abShift;

    half3 col = half3(input.sample(s, uvr).r,
                      input.sample(s, uv).g,
                      input.sample(s, uvb).b);
    output.write(half4(col, 1), gid);
}
```



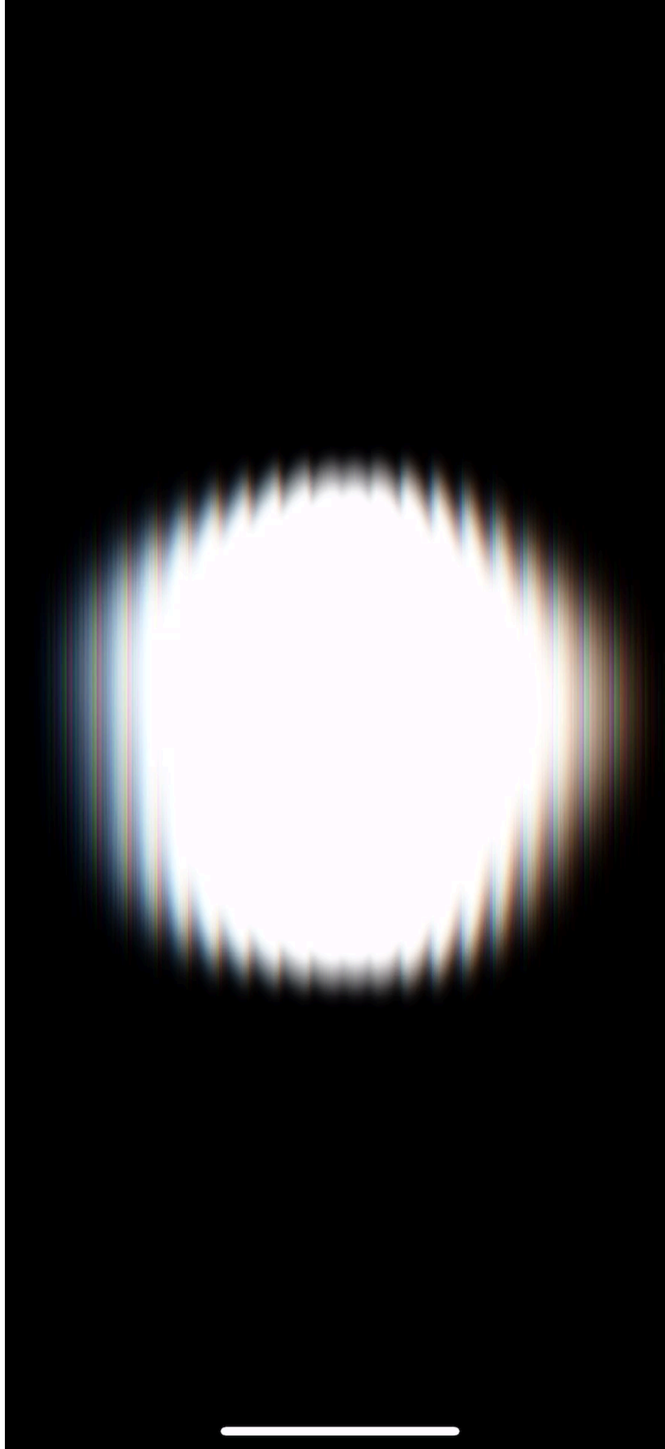


```
kernel void aberrationCompute(texture2d<half, access::write> output [[texture(0)]],
                              texture2d<half, access::sample> input [[texture(1)]],
                              uint2 gid [[thread_position_in_grid]]) {
    int width = output.get_width();
    int height = output.get_height();
    float2 uv = float2(gid) / float2(width, height);

    constexpr sampler s(coord::normalized, address::clamp_to_edge, filter::nearest);

    half abShift = abs(uv.x - 0.5) * 0.02;
    float2 uvr = uv;
    uvr.x -= abShift;
    float2 uvb = uv;
    uvb.x += abShift;

    half3 col = half3(input.sample(s, uvr).r,
                      input.sample(s, uv).g,
                      input.sample(s, uvb).b);
    output.write(half4(col, 1), gid);
}
```

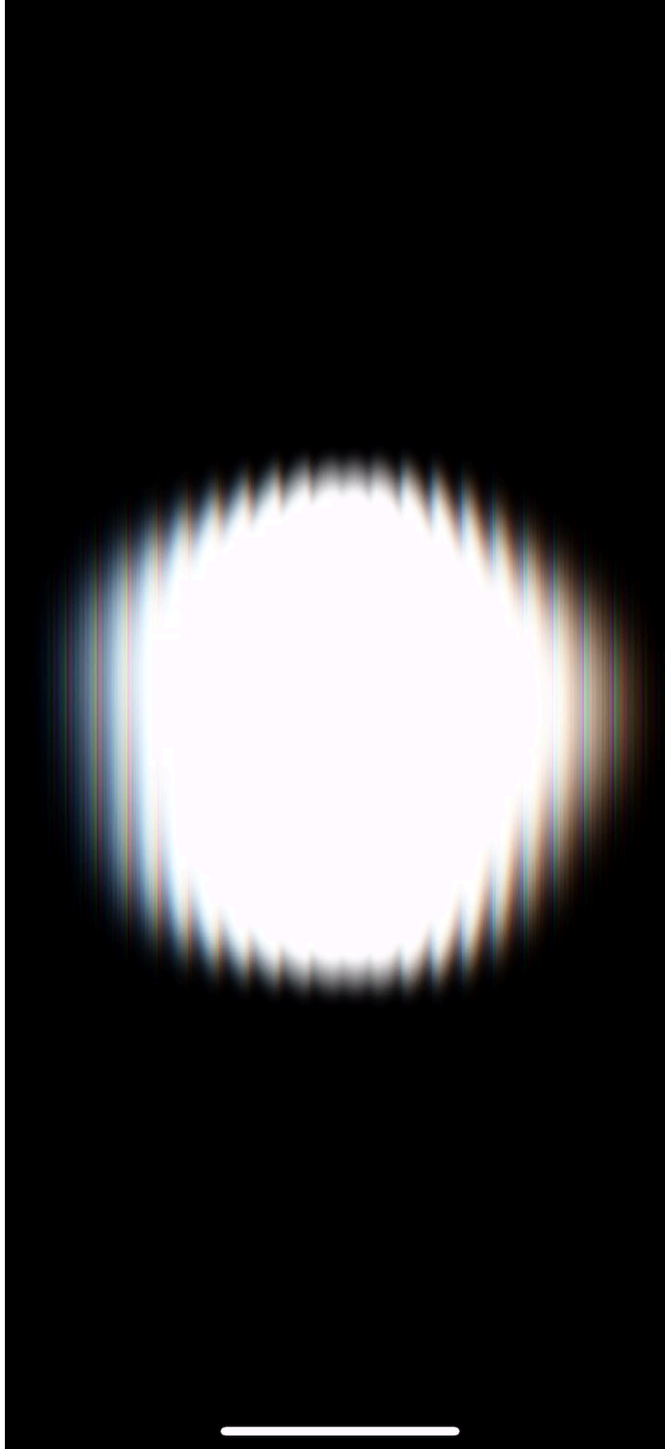




```
func draw(in view: MTKView) {
    ...
    let shapeCommand = buf.makeComputeCommandEncoder()!
    shapeCommand.setComputePipelineState(shapePipelineState)
    shapeCommand.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
    shapeCommand.setTexture(drawable.texture, index: 0)
    shapeCommand.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
    shapeCommand.endEncoding()

    let blurCommand = buf.makeComputeCommandEncoder()!
    blurCommand.setComputePipelineState(blurPipelineState)
    blurCommand.setTexture(backTexture, index: 0)
    blurCommand.setTexture(drawable.texture, index: 1)
    blurCommand.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
    blurCommand.endEncoding()

    let aberrationCommand = buf.makeComputeCommandEncoder()!
    aberrationCommand.setComputePipelineState(aberrationPipelineState)
    aberrationCommand.setTexture(drawable.texture, index: 0)
    aberrationCommand.setTexture(backTexture, index: 1)
    aberrationCommand.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
    aberrationCommand.endEncoding()

    buf.present(drawable)
    buf.commit()
}
```



```
func draw(in view: MTKView) {
    ...
    let shapeCommand = buf.makeComputeCommandEncoder()!
    shapeCommand.setComputePipelineState(shapePipelineState)
    shapeCommand.setBytes(&uniforms, length: MemoryLayout.size(ofValue: uniforms), index: 0)
    shapeCommand.setTexture(drawable.texture, index: 0)
    shapeCommand.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
    shapeCommand.endEncoding()

    let blurCommand = buf.makeComputeCommandEncoder()!
    blurCommand.setComputePipelineState(blurPipelineState)
    blurCommand.setTexture(backTexture, index: 0)
    blurCommand.setTexture(drawable.texture, index: 1)
    blurCommand.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
    blurCommand.endEncoding()

    let aberrationCommand = buf.makeComputeCommandEncoder()!
    aberrationCommand.setComputePipelineState(aberrationPipelineState)
    aberrationCommand.setTexture(drawable.texture, index: 0)
    aberrationCommand.setTexture(backTexture, index: 1)
    aberrationCommand.dispatchThreads(size, threadsPerThreadgroup: threadsPerGroup)
    aberrationCommand.endEncoding()

    buf.present(drawable)
    buf.commit()
}
```



thebookofshaders.com

отличное начало для тех кто
хочет познакомиться с
основными концепциями

iquilezles.org/articles/distfunctions2d

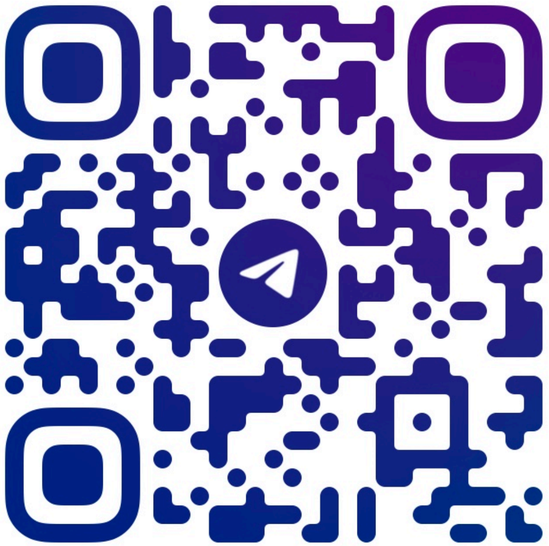
примеры 2d sdf функций для
базовых фигур

shadertoy.com

огромная и активно пополняемая
база пользовательских шейдеров

kodeco.com/books/metal-by-tutorials/v4.0

классическая книга от команды
Рея Вендердлиха о том как с нуля
написать 3D движок на Metal



tg: @nextster

vk: @bes

github.com/nextster/MobiusDemo
демо проект