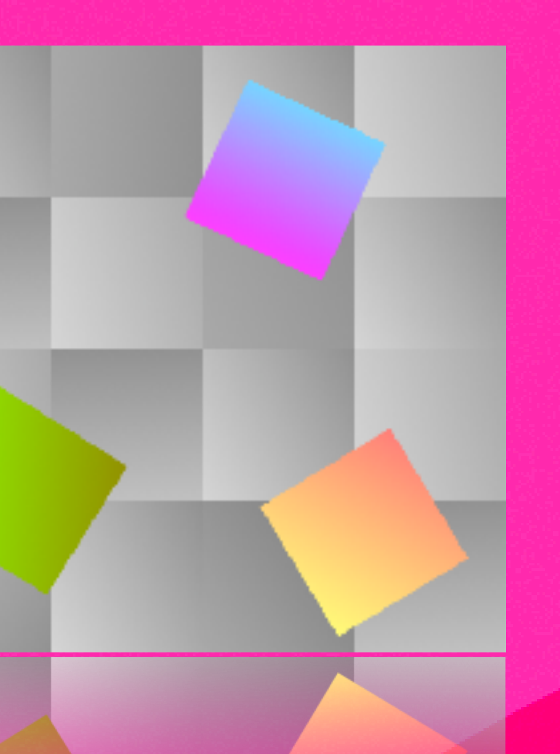


Fun with Pixels

Core Image as a core Technology

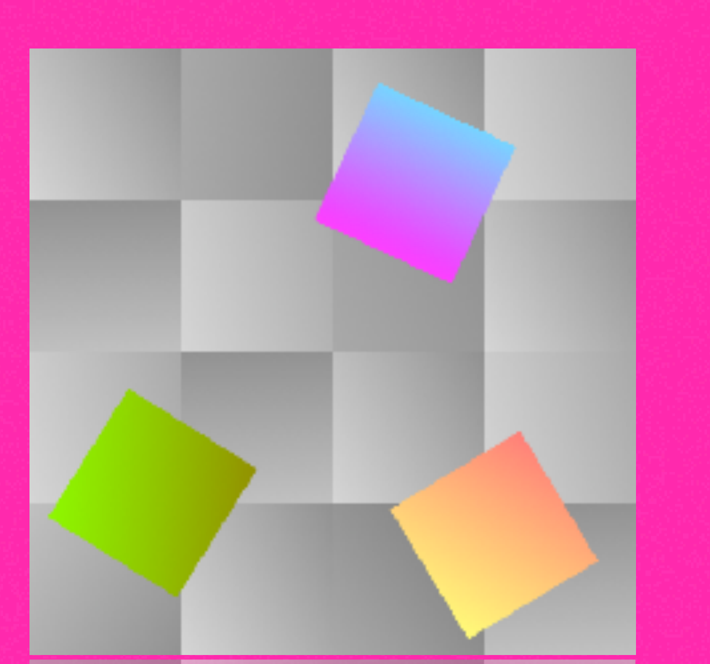
Core Image is like Core Graphics and similar one of Apple's underlying work horses. It is an image processing and analysis technology both for still and video images.

Compared to other technologies like Core Graphics, which is a rendering API, it is not based on data but on representations.

At its center are CIFilters as processing units. They can create, as generators, or manipulate existing CIImages.

In addition Core Image provides support for file formats, both as raw from different kind of cameras with CIRawFilters, and for reading and writing on file, like TIFF or heif.

Werner Lonsing on Jan/30/2025



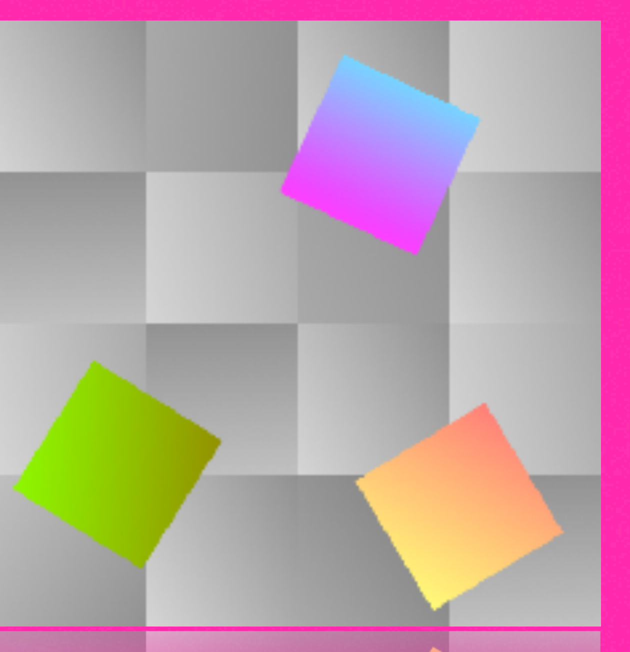
Fun with Pixels

Characteristics

A CIImage is much more a recipe than a rectangular area of pixels. Dimensions are not mandatory. Convenient initialisers from various image sources are available. The property in UIImage is usually nil, so it cannot be used.

To present a CIImage a CIContext is needed to render the image suitable for further deployment through Core Graphics towards the UI. As mentoined, CIImages can be written directly on file.

- CIImage
- CIFilter
- CIContext
- CIColor
- CIVector
- CIKernel
- CIRawFilter
- CIFeature
- and more



Fun with Pixels

CIImage

CIImage is the major data type in Core Image. Usage is straightforward with some handy initialisers. 'ciImageWithErrorText' here is a separate function.

```
func ciImageFromPath(path:String) -> CIImage
    {
        guard let img = UIImage.init(contentsOfFile: path)else
        {
            return ciImageWithErrorText(error: "No image at Path")
        }

        guard let ciImg = CIImage(image: img) else
        {
            return ciImageWithErrorText(error: "No CIImage from UIImage")
        }
        return ciImg
    }
```


Werner Lonsing on Jan/30/2025

Fun with Pixels

CIContext

CIContext provides the context for rendering the processed images into various places such as CGImage, CVPixelBuffer or MTLTextures, or as data in various file formats to select from.

A CIContext should be created only once as a singleton. It is always immutable and thread safe, CIImages from multiple threads may share it.

It is a common mistake to initialise a CIContext, use it once, forget about it and next time crate a new one.

```
class Helper {
. . .
static let context = CIContext(options:
                                        [.workingColorSpace: CGColorSpaceCreateDeviceRGB(),
                                         .outputColorSpace: CGColorSpaceCreateDeviceRGB()])
```

Using the classical class-variable pattern the context is accessible from elsewhere like:

```
  let context = Helper.context
```

Fun with Pixels

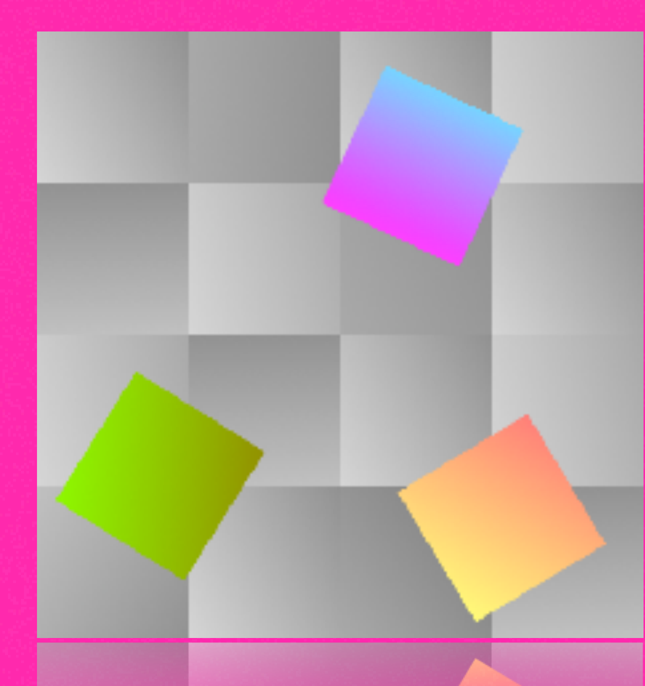
CIFilter with key and value

result image

Filters are at the center of Core Image's processing capabilities. The name is already shortcoming, because some filters generate images based on types of input parameters other than images,e.g. they do not filter the image. However, typically a filter takes one or more images as its source of input and some parameters to alter the result.
Traditionally all CIFilters object are set and retrieved through the use of key-value pairs:

```
let ctx = Helper.context
let myFilter = CIFilter(name: "CISepiaTone")!
var workerImage = CIImage(image: UIImage.init(named: "Mandrill.png")!)!
myFilter.setValue(workerImage, forKey: kCIInputImageKey)
myFilter.setValue(1.0, forKey: kCIInputIntensityKey)
workerImage = myFilter.value(forKey: kCIOutputImageKey) as! CIImage
let savePath = workerImage.saveJPEG("Mandr",  quality: 0.1, inContext: ctx)
```


Werner Lonsing on Jan/30/2025



Fun with Pixels

Built-in CIFilters

result is the same

To avoid key-value coding with its flexible types all 242 CiFilters are built-in. To access them, an additional framework 'CoreImage.CIFilterBuiltins', is deployed. Otherwise the code and its result are identical. From a swift point of view, the built-in filters should be preferred.

```
import CoreImage.CIFilterBuiltins

  let ctx = Helper.context
  let myFilter = CIFilter.sepiaTone()
  var workerImage = CIImage(image: UIImage.init(named: "Mandrill.png")!)!
  myFilter.inputImage = workerImage
  myFilter.intensity = 1.0
  workerImage = myFilter.outputImage!
  let savePath = workerImage.saveJPEG("Mandr",  quality: 0.1, inContext: ctx)
```

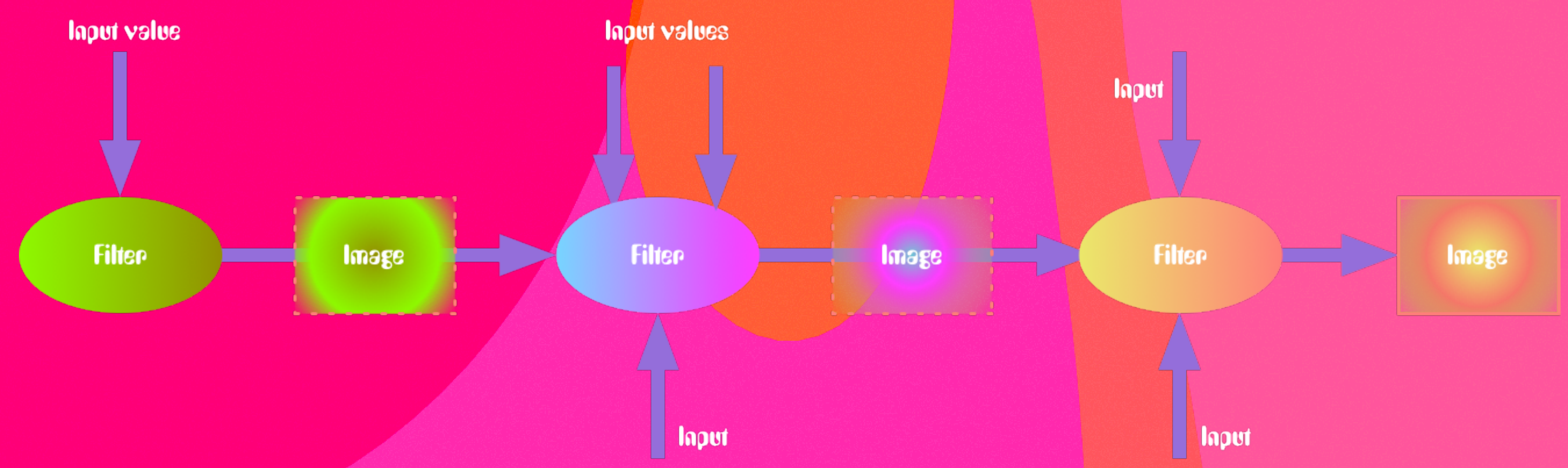
Fun with Pixels

Principal nature of a CIFilter

At first, all CIFilters must have one and only one output-value, the output image, which is always of type CIImage. Commonly the output image is based on the input image, but there are filters with an output image of only one pixel. It does not stop there. Sometimes even these color values must be translated into some kind of CIVector or else.
The input parameters are ranging from one, as for a single color image, to almost infinite for color curves.
Hence a CIFilter has at least one input value and exactly one output value, which is always of type CIImage.

A simple filter chain

The power of Core Image unfolds once filter are chained. Because the CIImage is rather a recipe to draw than an image by itself these CIImage can be directly injected into the next filter. At the end of the chain the final image will be rendered only once.

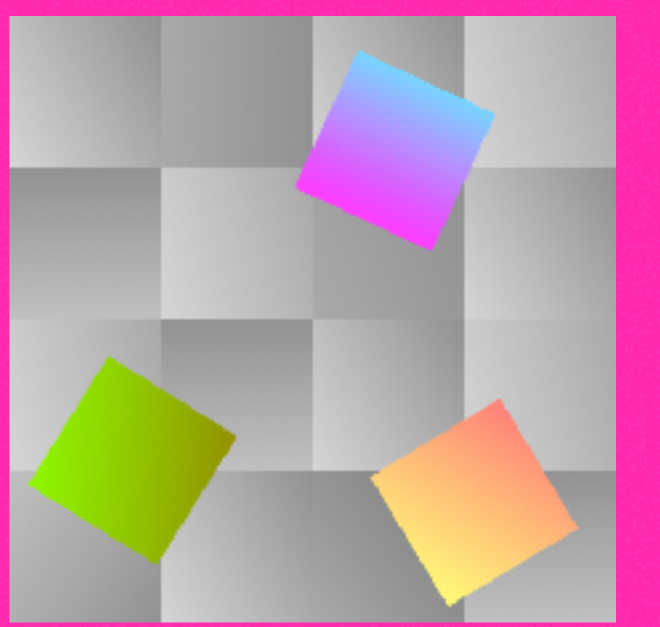


Fun with Pixels

21 categories for filters, not all are documented as public:

- CICategoryBlur
- CICategoryBuiltIn (242 filters)
- CICategoryColorAdjustment
- CICategoryColorEffect
- CICategoryCompositeOperation
- CICategoryDistortionEffect
- CICategoryGenerator
- CICategoryGeometryAdjustment
- CICategoryGradient
- CICategoryHalftoneEffect
- CICategoryHighDynamicRange
- CICategoryInterlaced
- CICategoryNonSquarePixels
- CICategoryReduction
- CICategorySharpen
- CICategoryStillImage
- CICategoryStylize
- CICategoryTileEffect
- CICategoryTransition
- **CICategoryVideo**
- CICategoryXMPSerializable

Only one is important.
The ‘video’-category indicates, that the filter is suitable for live-viewing.
All 242 filters are built-in.



Fun with Pixels

The complete list

242 Filters on all platforms: MacOS and iOS

CIAccordionFoldTransition
CIAdditionCompositing
CIAffineClamp
CIAffineTile
CIAffineTransform
CIAreaAlphaWeightedHistogram
CIAreaAverage
CIAreaBoundsRed
CIAreaHistogram
CIAreaLogarithmicHistogram
CIAreaMaximum
CIAreaMaximumAlpha
CIAreaMinimum
CIAreaMinimumAlpha
CIAreaMinMax
CIAreaMinMaxRed
CIAttributedTextImageGenerator
CIAztecCodeGenerator
CIBarcodeGenerator
CIBarsSwipeTransition
CIBicubicScaleTransform
CIBlendWithAlphaMask
CIBlendWithBlueMask
CIBlendWithMask
CIBlendWithRedMask
CIBloom
CIBlurredRectangleGenerator
CIBokehBlur
CIBoxBlur
CIBumpDistortion

CIBumpDistortionLinear
CICannyEdgeDetector
CICheckerboardGenerator
CICircleSplashDistortion
CICircularScreen
CICircularWrap
CIClamp
CICMYKHalftone
CICode128BarcodeGenerator
CIColorAbsoluteDifference
CIColorBlendMode
CIColorBurnBlendMode
CIColorClamp
CIColorControls
CIColorCrossPolynomial
CIColorCube
CIColorCubesMixedWithMask
CIColorCubeWithColorSpace
CIColorCurves
CIColorDodgeBlendMode
CIColorInvert
CIColorMap
CIColorMatrix
CIColorMonochrome
CIColorPolynomial
CIColorPosterize
CIColorThreshold
CIColorThresholdOtsu
CIColumnAverage
CIComicEffect

CIConstantColorGenerator
CIConvertLabToRGB
CIConvertRGBtoLab
CIConvolution3X3
CIConvolution5X5
CIConvolution7X7
CIConvolution9Horizontal
CIConvolution9Vertical
CIConvolutionRGB3X3
CIConvolutionRGB5X5
CIConvolutionRGB7X7
CIConvolutionRGB9Horizontal
CIConvolutionRGB9Vertical
CICopyMachineTransition
CICoreMLModelFilter
CICrop
CICrystallize
CIDarkenBlendMode
CIDepthBlurEffect
CIDepthOfField
CIDepthToDisparity
CIDifferenceBlendMode
CIDiscBlur
CIDisintegrateWithMaskTransition
CIDisparityToDepth
CIDisplacementDistortion
CIDissolveTransition
CIDistanceGradientFromRedMask
CIDither
CIDivideBlendMode

CIDocumentEnhancer
CIDotScreen
CIDroste
CIEdges
CIEdgeWork
CIEightfoldReflectedTile
CIExclusionBlendMode
CIExposureAdjust
CIFalseColor
CIFlashTransition
CIFourfoldReflectedTile
CIFourfoldRotatedTile
CIFourfoldTranslatedTile
CIGaborGradients
CIGammaAdjust
CIGaussianBlur
CIGaussianGradient
CIGlassDistortion
CIGlassLozenge
CIGlideReflectedTile
CIGloom
CIGuidedFilter
CIHardLightBlendMode
CIHatchedScreen
CIHeightFieldFromMask
CIHexagonalPixellate
CIHighlightShadowAdjust
CIHistogramDisplayFilter
CIHoleDistortion
CIHueAdjust
CIHueBlendMode

CIHueSaturationValueGradient
CIKaleidoscope
CIKeystoneCorrectionCombined
CIKeystoneCorrectionHorizontal
CIKeystoneCorrectionVertical
CIMorphologyRectangleMaximum
CIMorphologyRectangleMinimum
CICameraCalibrationLensCorrection
CIPageCurlWithShadowTransition
CIPerspectiveTransformWithExtent
CIEdgePreserveUpsampleFilter
CIRoundedRectangleStrokeGenerator
CILabDeltaE
CILanczosScaleTransform
CILenticularHaloGenerator
CILightenBlendMode
CILightTunnel
CILinearBurnBlendMode
CILinearDodgeBlendMode
CILinearGradient
CILinearLightBlendMode
CILinearToSRGBToneCurve
CILineOverlay
CILineScreen
CILuminosityBlendMode
CIMaskedVariableBlur
CIMaximumComponent
CIMaximumCompositing
CIMaximumScaleTransform
CIRoundedRectangleGenerator

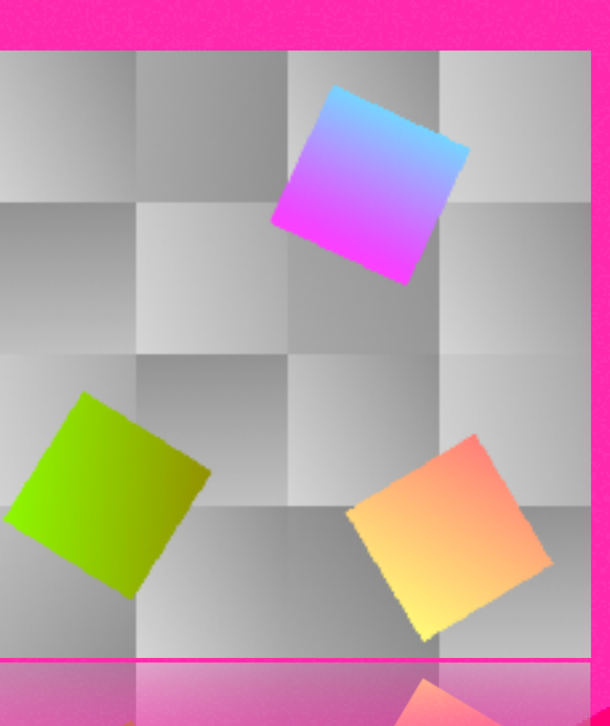
CIMaskToAlpha
CIKMeans
CIMedianFilter
C CIMeshGenerator
IMinimumComponent
CIMinimumCompositing
CIMix
CIModTransition
CIMorphologyGradient
CIMorphologyMaximum
CIMorphologyMinimum
CIMotionBlur
CIMultiplyBlendMode
CIMultiplyCompositing
CINinePartStretched
CINinePartTiled
CINoiseReduction
CIOpTile
CIOverlayBlendMode
CIPageCurlTransition
CIPaletteCentroid
CIPalettize
CIParallelogramTile
CIPDF417BarcodeGenerator
CIPersonSegmentation
CIPerspectiveCorrection
CIPerspectiveRotate
CIPerspectiveTile
CIPerspectiveTransform
CIPhotoEffectChrome

CIPhotoEffectFade
CIPhotoEffectInstant
CIPhotoEffectMono
CIPhotoEffectNoir
CIPhotoEffectProcess
CIPhotoEffectTonal
CIPhotoEffectTransfer
CIPinchDistortion
CIPinLightBlendMode
CIPixellate
CIPointillize
CIQRCodeGenerator
CIRadialGradient
CIRandomGenerator
CIRippleTransition
CIRowAverage
CISaliencyMapFilter
CISampleNearest
CISaturationBlendMode
CIScreenBlendMode
CISepiaTone
CIShadedMaterial
CISharpenLuminance
CISixfoldReflectedTile
CISixfoldRotatedTile
CISmoothLinearGradient
CISobelGradients
CISoftLightBlendMode
CISourceAtopCompositing
CISourceInCompositing

CISourceOutCompositing
CISourceOverCompositing
CISpotColor
CISpotLight
CISRGBToneCurveToLinear
CIStarShineGenerator
CIStraightenFilter
CIStretchCrop
CIStripesGenerator
CISubtractBlendMode
CISunbeamsGenerator
CISwipeTransition
CITemperatureAndTint
CITextImageGenerator
CIThermal
CIToneCurve
CIToneMapHeadroom
CITorusLensDistortion
CITriangleKaleidoscope
CITriangleTile
CITwelvefoldReflectedTile
CITwirlDistortion
CIUnsharpMask
CIVibrance
CIVignette
CIVignetteEffect
CIVividLightBlendMode
CIVortexDistortion
CIWhitePointAdjust
CIXRay
CIZoomBlur

Talk @ Cocoaheads-AC

Werner Lonsing on Jan/30/2025



Fun with Pixels

Get all filters

There is one simple function in CIFilter to get all filter names. CIFilter.filterNames(inCategories: nil).
If the category is nil or an empty array, it presents all filter names, otherwise those of the selected category.

```
func ciFilters()
{
    let names = CIFilter.filterNames(inCategories: [])
    let builtInNames = CIFilter.filterNames(inCategories: ["CICategoryBuiltIn"])
    if(names == builtInNames)
    {
        print("all filters are built in!")
    }
    . . .
```

Fun with Pixels

Get the attributes of a filter

The filters have to be instantiated in order to access the attributes.
Printing (filtersInCategory) yields about the same result, but almost not readable

```
for aFilterName in filtersInCategory
{
    print("\n\(aFilterName)\n")
    let filter = CIFilter.init(name: aFilterName)
    print(filter?.attributes ?? "")
}
```

Fun with Pixels

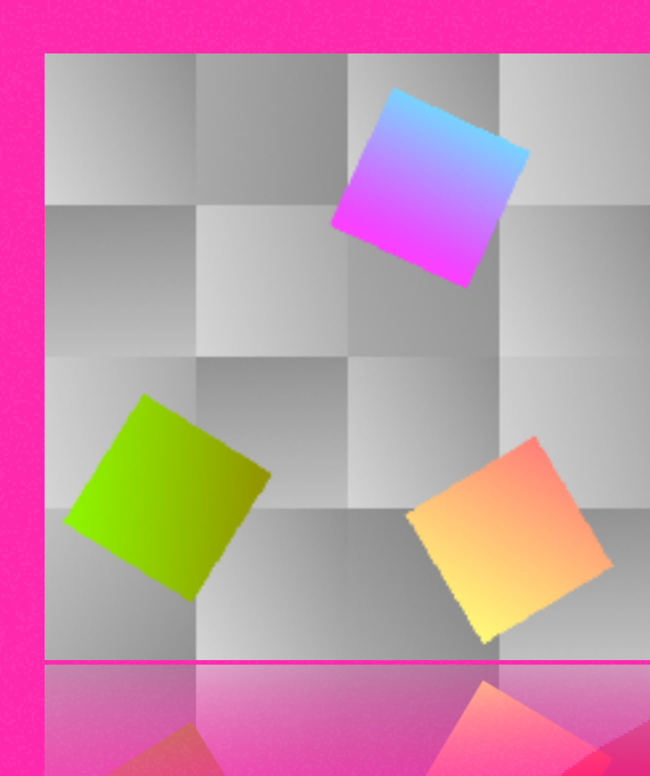
Some words about filters

At first an example of a simple chained filters, checkerboard and twirl.

Generators: Graphic generators like simple color, stripes or checkerboard, textual as text or some kind of barcodes.

Reduction filters like kMean or histogram evaluate and analyse an image. Their output image contains these informations, hence these images are not modified images. They cannot be part of a chain with reasonable results

General purpose filters like CIColorMatrix allow some customization based on simple math to create individual filters. Probably even some of the built-in filters were made this way.

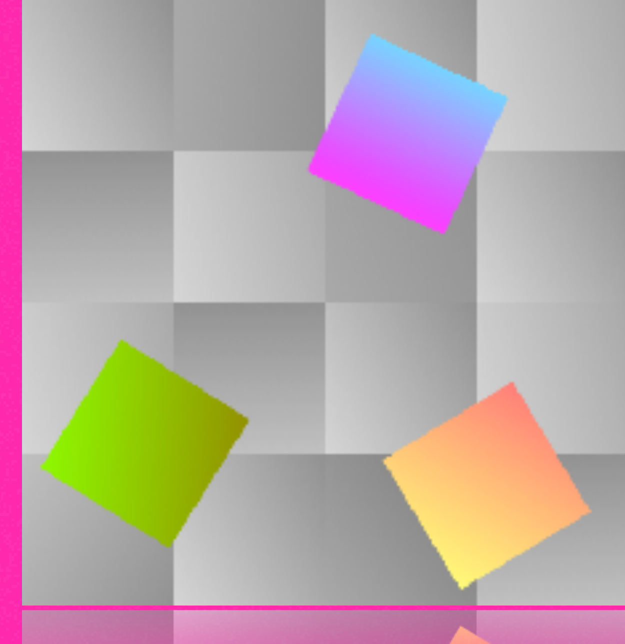


Fun with Pixels

The others

Core Image has a long history, starting with Mac OSX 10.4. Under the hood the kernels changed from the the OpenGL shading language to Metal, the transition to iOS took years and was very iterative. At some time filters were available in the simulator, as part of OSX, but not on the device.

As consequence some people were tempted to write their own repository of kernels, like Brad Larson with his GPUImage-framework, now as version 3. Like with all third party frameworks he is busy keeping up with the ever changing technology, re-writing all filters from Objective-C and shading language first to Swift and then to Metal.
Another approach are the attempts to create a unified interface for all filters. There is even an app, that creates some code of it. But there some 20 filters are missing, and the question remains, who needs it. How to feed all animals in a zoo? In essence, it is just a production of boilerplate-code. Selecting and applying some filters is a task on its own and not necessarily subject to coding.



Fun with Pixels

Original image

From Bonn
to Cologne

Fun with Pixels

5 pre-configured black-and-white filters:

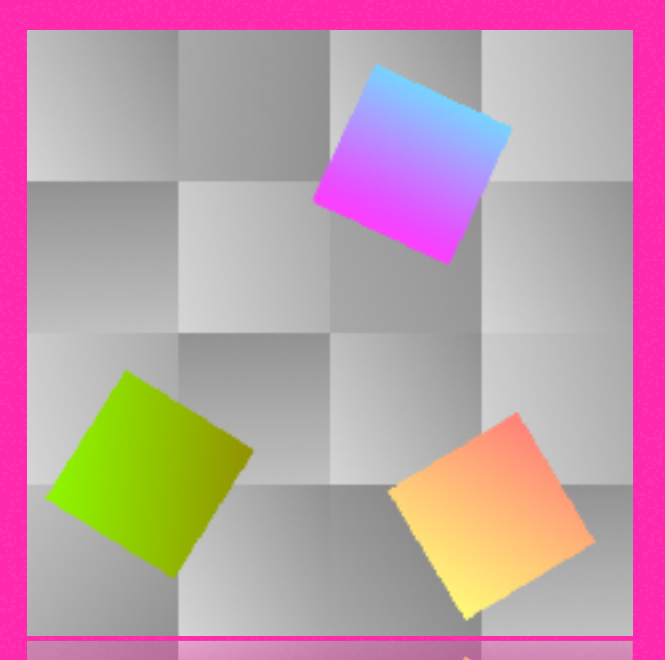
CIMaximumComponent

CIPhotoEffectTonal

CIPhotoEffectMono

CIMinimumComponent

CIPhotoEffectNoir



Fun with Pixels

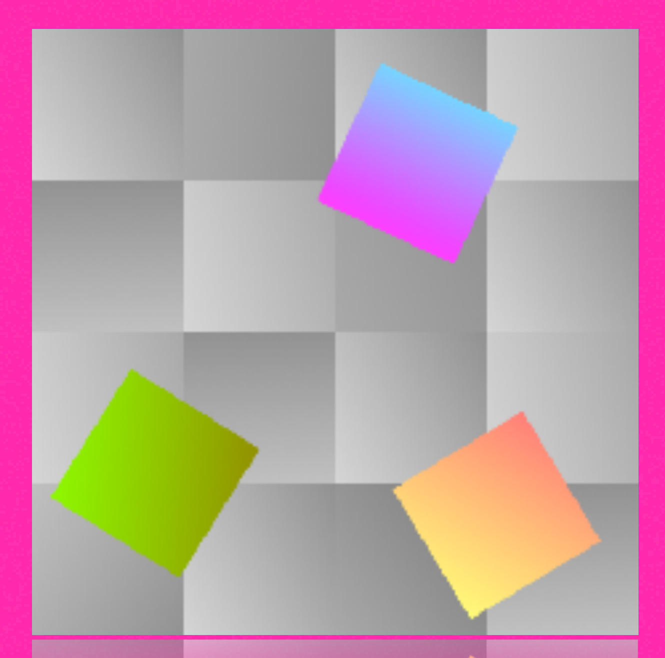
CIPhotoEffectNoir

Imitates black-and-white photography film with exaggerated contrast.

Fun with Pixels

CIMinimumComponent

Returns a grayscale image from min(r,g,b).

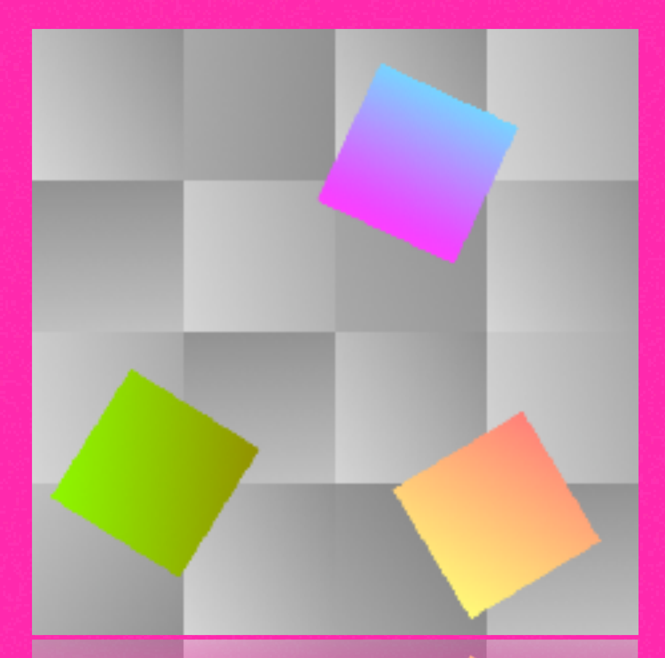


Fun with Pixels

CIPhotoEffectMono

Applies a preconfigured set of effects that imitate black-and-white photography film with low contrast.

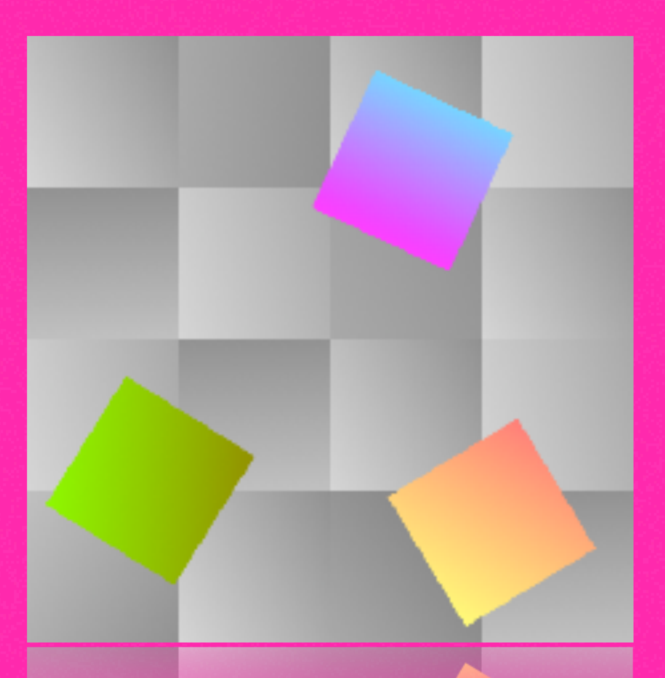
Werner Lonsing on Jan/30/2025



Fun with Pixels

CIPhotoEffectTonal

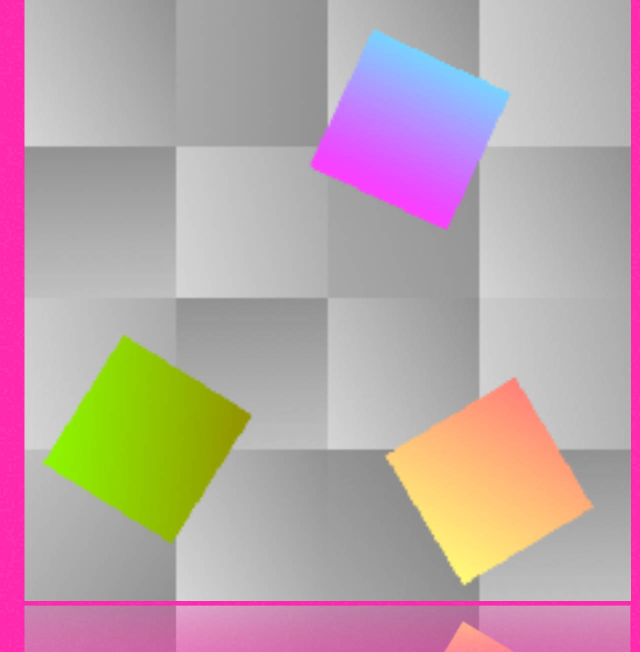
Applies a preconfigured set of effects that imitate black-and-white photography film without significantly altering contrast.



Fun with Pixels

CIMaximumComponent

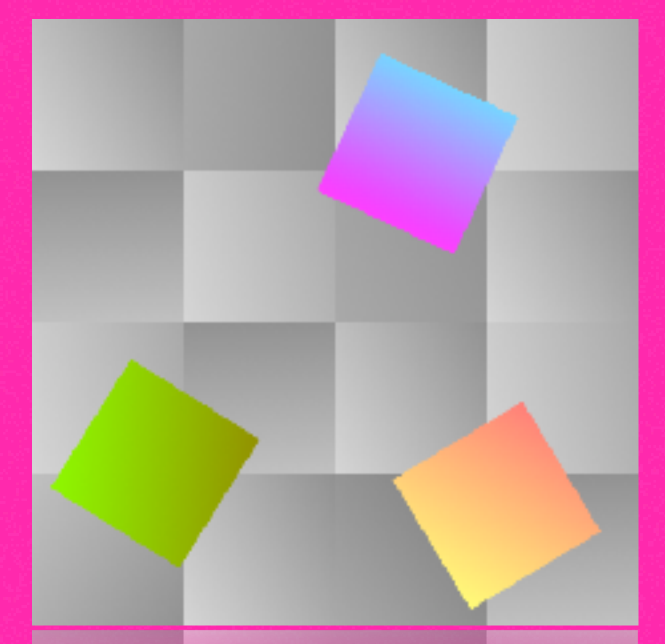
Returns a grayscale image from max(r,g,b).



Fun with Pixels

Modified image:

From Bonn to Cologne



Fun with Pixels

original

Source images are the original, a b/w image thereof and the grid.
With MaskToAlpha and some layerings the final image was composed, as seen before.

final

Fun with Pixels

Better than perfect.
Autoadjustment filters applied to the grid.

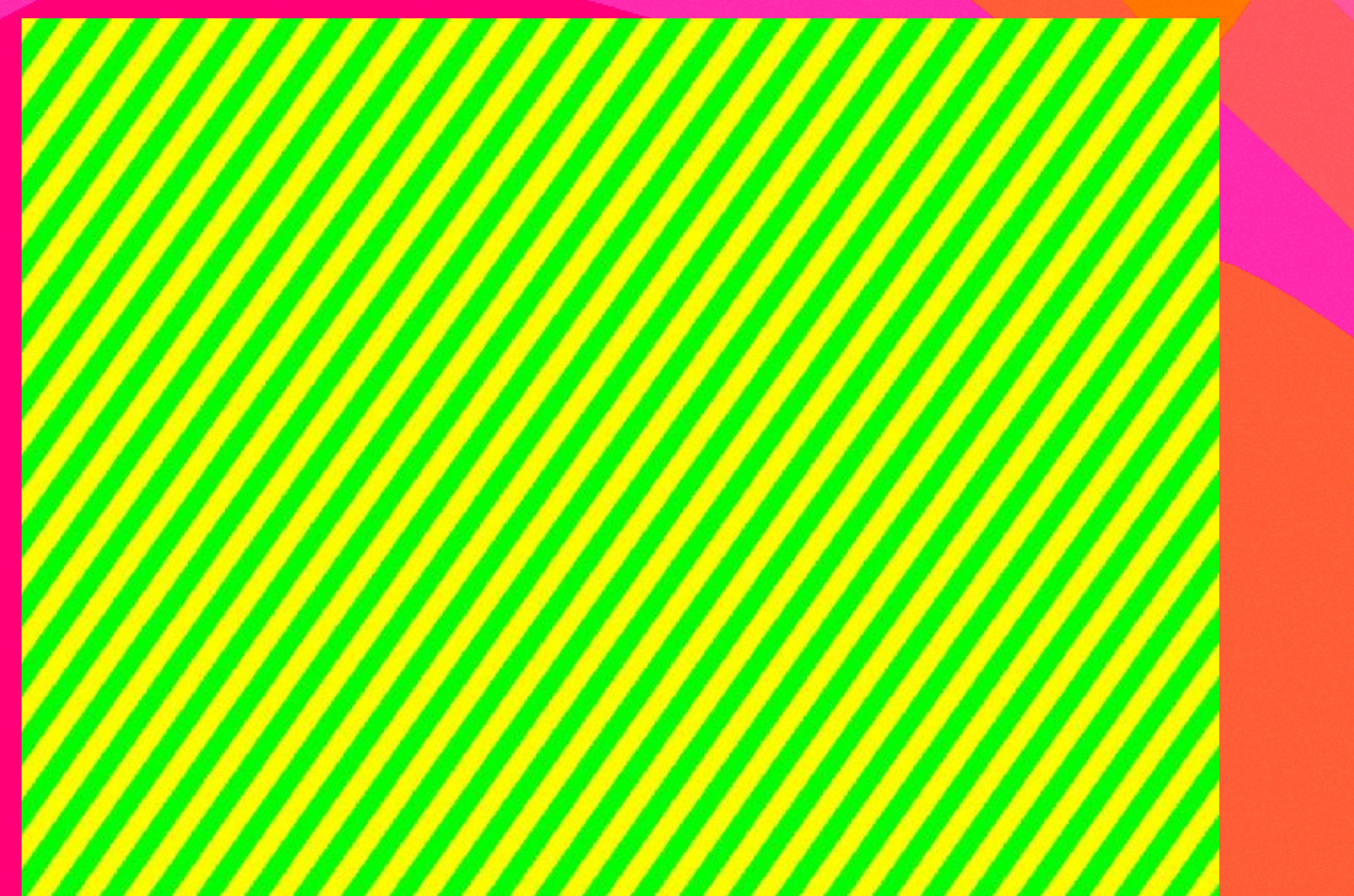
Fun with Pixels

Original image

Fun with Pixels

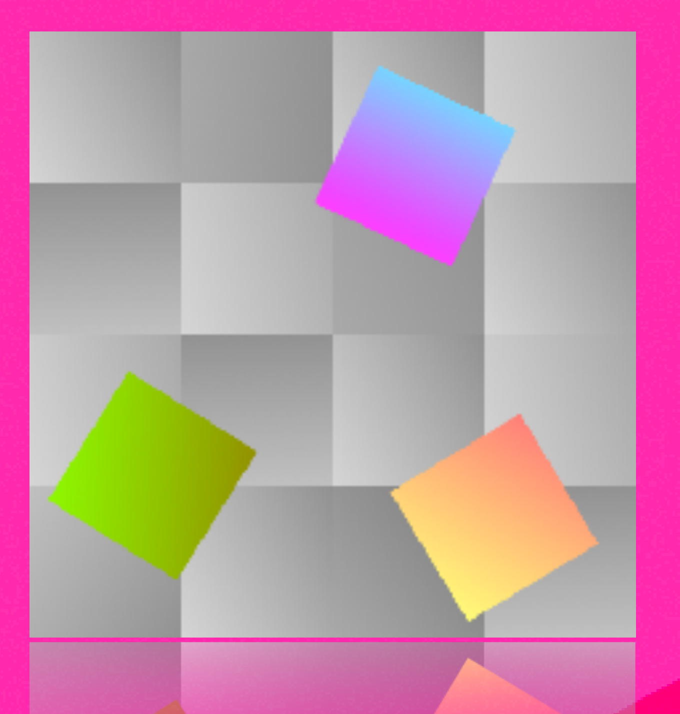
Generate two pattern images

(Own function)



```
createCheckerBoard(color0: CIColor.yellow, color1: CIColor.green, width: 64.0, angle: 145.0, stretching: 640.0)

createCheckerBoard(color0: CIColor.blue, color1: CIColor.red, width: 0.52 * 64.0, angle: 20.0, stretching: 0.8)
```



Fun with Pixels

Select from 5 different Choice

Based on the b/w filters

Pattern applied to both channels

With maskToAlpha and some composition kind of two different channels are established as foreground and background.

Fun with Pixels

Composite binary image

Filters used:
CICheckerboardGenerator
[CGAffineTransform]

CIMaximumComponent
CIVignette
CIColorThreshold

[pattern cropped]
CIMaskToAlpha
CISourceInCompositing
CISourceOutCompositing
CISourceOverCompositing

Fun with Pixels

In VisionOS, simulator for eye-tracking

visionOS 1.1

Complete Photo

Both images are sliced into tiles. Once selected, each tile can change from the original position into the colored one, and then back again just by looking at it and snipping with the fingers.

Fun with Pixels

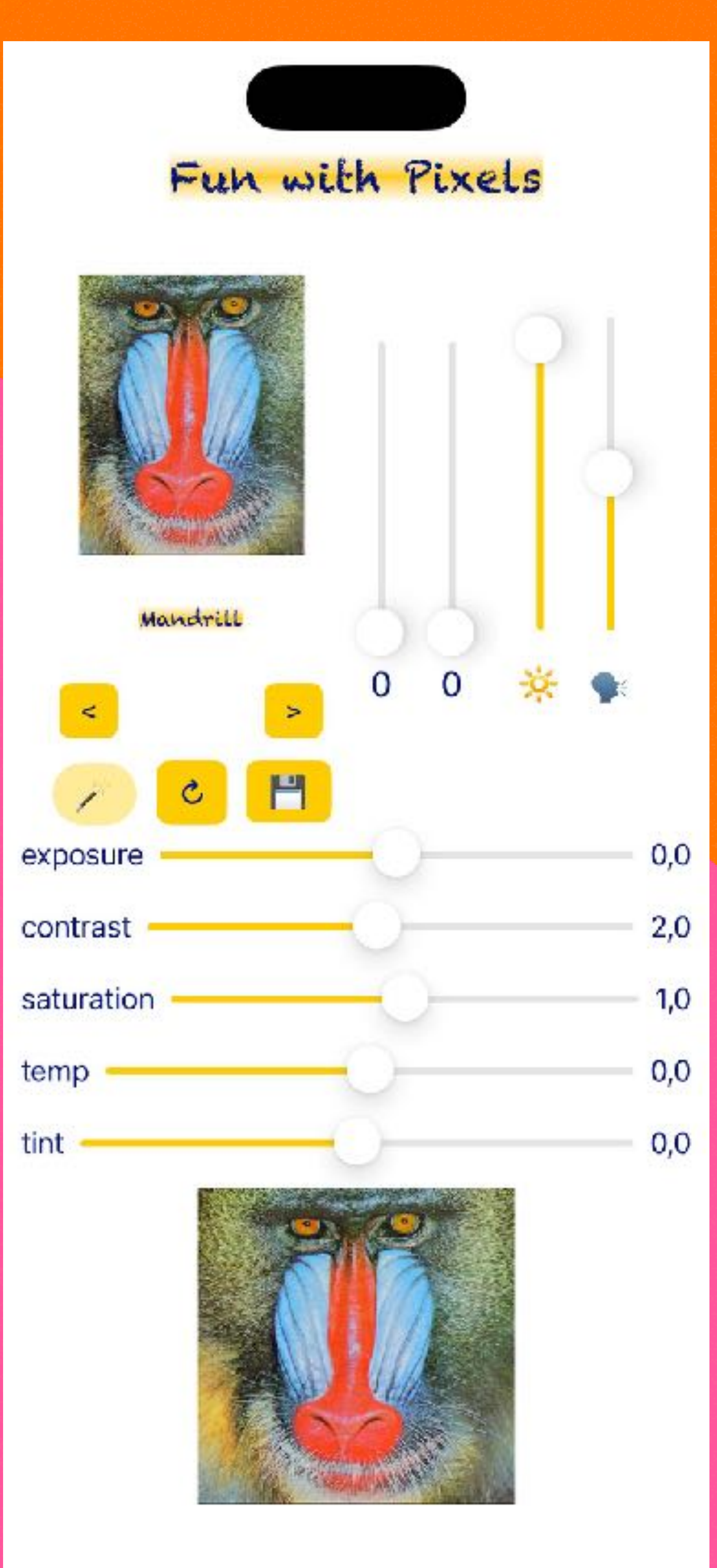
The demo-app to code with

Download it from here:
https://github.com/dialThat/FunWithPixels

Start by messing with the colors in ContentView:

```
let tintColor = Color(uiColor: )
let foreGroundColor = Color(uiColor: )
```

Then collect some images and copy them into the bundle, not the Asset-folder.
If running on a device, do not forget to inject your credentials. The app should run on a working, but otherwise empty code, like e.g. the original image can be saved unaltered.



Werner Lonsing on Jan/30/2025

Fun with Pixels

How to find a filter

A. Search for it like 'CIFilter sepia'.
B. Find the documentation on Apple's webpage.
C. Find the example code oh that page and use it. The names are meaningful and all parameters or attributes are set.

```
func sepiaTone(inputImage: CIImage ) -> CIImage {
    let sepiaToneFilter = CIFilter.sepiaTone()
    sepiaToneFilter.inputImage = inputImage
    sepiaToneFilter.intensity = 1
    return sepiaToneFilter.outputImage!
}
```

Directly from the website

https://developer.apple.com/documentation/coreimage/cifilter/3228402-sepiatone

Fun with Pixels

How to set the values

Commonly most parameters or attribute are sparsely documented. The major problem is, that the attributes are of all different types, established historically through instances of ‘NSNumber’. Even more confusing are the different ranges of the parameters, inside of which they are suitable while outside they render the image as void. Luckily the key-value encoding interface provide some help. Just call:

```
print (sepiaToneFilter.attributes)
```

Fun with Pixels

Read through all the attributes

```
["CIAttributeFilterAvailable_Mac": 10.4, "CIAttributeFilterDisplayName": Sepia Tone,
"CIAttributeFilterCategories": <__NSArrayI 0x60000212fc00>(
            CICategoryColorEffect,
. . .
          )
            , "inputIntensity": {
                CIAttributeClass = NSNumber;
                CIAttributeDefault = 1;
                CIAttributeDescription = "The intensity of the sepia effect. A value of 1.0
creates a monochrome sepia image. A value of 0.0 has no effect on the image.";
                CIAttributeDisplayName = Intensity;
                CIAttributeIdentity = 0;
                CIAttributeMin = 0;
                CIAttributeSliderMax = 1;
                CIAttributeSliderMin = 0;
                CIAttributeType = CIAttributeTypeScalar;
            }, "inputImage": {
                CIAttributeClass = CIImage;
                CIAttributeDescription = "The image to use as an input for the effect.";
                CIAttributeDisplayName = Image;
                CIAttributeType = CIAttributeTypeImage;
            }, "CIAttributeFilterAvailable_iOS": 5, "CIAttributeReferenceDocumentation": http://
developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CoreImageFilterReference/
index.html#//apple_ref/doc/filter/ci/CISepiaTone, "CIAttributeFilterName": CISepiaTone]
```

Fun with Pixels

Look for the name and its min/max values

Besides the common 'inputImage' the name of the relevant parameter is 'inputIntensity'. Applied to it are some values. The related maximum value is labelled 'CIAttributeSliderMax' with a value of '1', while the corresponding 'CIAttributeSliderMin' is '0'. Note the identity value 'CIAttributeIdentity', which is also '0'. Setting the identity value processes the image without changes when the filter is called.

```
inputIntensity”: {                                              <-
                CIAttributeClass = NSNumber;
                CIAttributeDefault = 1;
                CIAttributeDescription = "The . . . image.”;
                CIAttributeDisplayName = Intensity;
                CIAttributeIdentity = 0;
                CIAttributeMin = 0;
           ->   CIAttributeSliderMax = 1;
           ->   CIAttributeSliderMin = 0;
                CIAttributeType = CIAttributeTypeScalar;
```

Either set these values directly in the slider provided by SwiftUI, or use a normalised slider from -1…1 and calculate the values in the code.

Fun with Pixels

First round of sliders

Only three filters are needed: CIColorControls with brightness ranging from -1…1, and saturation from 0…2. Contrast seems to be missing, instead there are blue and red coefficients. Anyway, contrast works in the range -1 … 1.
The second filter CITemperatureAndTint needs extra care, while the third CIHighlightShadowAdjust suggest the range 0…1 for both the highlight and the shadow slider. The highlight-slider seems to be reversed.

CIColorControls	
exposure	0,0
contrast	2,0
saturation	1,0

CITemperatureAndTint	
temp	0,0
tint	0,0

CIHighlightShadowAdjust

Fun with Pixels

Mysterious sliders

The temperature and tint filter presents kind of a riddle with a CIVectors-values both as identity and max as [6500 0], but no Min.

“CITemperatureAndTint has three input parameters: Image, Neutral and TargetNeutral. Neutral and TargetNeutral are of CIVector type, and in both of them the first dimension refers to Temperature and the second to Tint. What the CITemperatureAndTint filter basically does is computing a matrix that adapts RGB values from the source white point defined by Neutral (srcTemperature, srcTint) to the target white point defined by TargetNeutral (dstTemperature, dstTint), and then applying this matrix on the input image (using the CIColorMatrix filter). If Neutral and TargetNeutral are of the same values, then the image will not change after applying this filter. The two sliders give the Temperature and Tint changes (i.e. differences between source and target Temperature and Tint values already) added to the source image.”

As first step, the sliders are normalised.

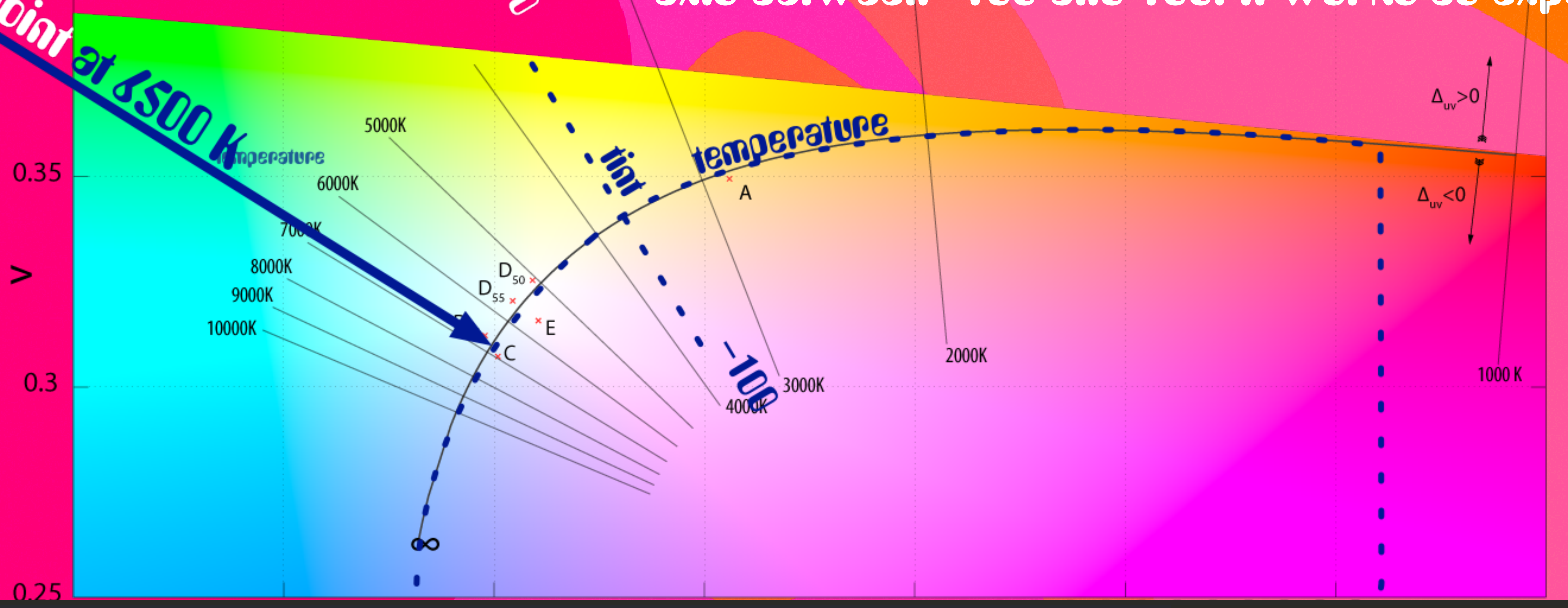
https://stackoverflow.com/questions/8629411/input-parameters-of-citemperatureandtint-cifilter

Werner Lonsing on Jan/30/2025

Fun with Pixels

CIELab colors and Kelvin

According to this diagram the min-value is set to '2000'. The max value, although potentially infinite, is set to 10500. The tint is perpendicular bouncing along the axis between -100 and 100. It works as expected.



Planckian locus

Source: Wikipedia

```
let vec = CIVector(x: 6500 + CGFloat(temperature)*4500,
                   y: CGFloat(tint) * 100).          // 2000 – 10500 Kelvin
```

Instantiating the sliders

All sliders are globally instantiated. Although not thread safe they are only called through SwiftUI, or, when an image is saved to disk. The code below is from each the code examples directly copied out of the documentation.

```
let colorControls = CIFilter.colorControls()
let highlightShadowAdjustFilter = CIFilter.highlightShadowAdjust()
let tempatureAndTintFilter = CIFilter.temperatureAndTint()
```

With these three lines and the following function all seven sliders should already work on the image.

Fun with Pixels

The main slider function

```
private func ciImageFromFilterWith(_ image: CIImage, exposure: Float, contrast: Float, highlights: Float, shadows: Float, saturation: Float, temperature: Float, tint: Float) -> CIImage
{
        var workerImage = image

        colorControls.brightness = exposure
        colorControls.contrast = contrast
        colorControls.saturation = saturation
        colorControls.inputImage = workerImage
        workerImage = colorControls.outputImage!

        highlightShadowAdjustFilter.highlightAmount = highlights
        highlightShadowAdjustFilter.shadowAmount = shadows
        highlightShadowAdjustFilter.inputImage = workerImage
        workerImage = highlightShadowAdjustFilter.outputImage!

        tempatureAndTintFilter.neutral = CIVector(x: 6500, y: 0) // 2000 - 10500 Kelvin
        let vec = CIVector(x: 6500 + CGFloat(temperature)*4500, y: CGFloat(tint) * 100)
        tempatureAndTintFilter.targetNeutral = vec
        tempatureAndTintFilter.inputImage = workerImage
        workerImage = tempatureAndTintFilter.outputImage!

        return workerImage
    }
```


Werner Lonsing on Jan/30/2025

Fun with Pixels

Autoadjustment Filters

Although a function of CIImage, the auto adjustment filters are a convenient set of pre-installed and pre-configured filters to easily enhance an image. The function 'autoAdjustmentFilters()' returns all possible automatically selected and configured filters for adjusting the image. These filters are simply subject to a loop, wherein all the filters are chained.

```
private func autoAdjust(image: CIImage) -> CIImage
 {
     var workerImage = image
     if(autoAdjustFilters.isEmpty)
     {
         autoAdjustFilters = image.autoAdjustmentFilters()
     }

     for filter in autoAdjustFilters
     {
         filter.setValue(workerImage, forKey: kCIInputImageKey)
         workerImage = filter.outputImage!
     }
     return workerImage
 }
```

Fun with Pixels

Implementation details

In order to avoid the costly collection of the 'autoAdjustmentFilters' an intermediate image is introduced to store the intermediate values of these filters. Scaling and later the mean-filters are here implemented as well. The function 'newIntermediate' is called, whenever the selected image changes or the adjust button is selected. Otherwise these filters are not triggered.

```
In ContentView

   func newIntermediate()
    {
        helper.intermediateImage(forIndex: selIndex, adjust: adjust, channels:
meanChannels, passes: meanPasses)
    }

And declaration in Helper:
    var autoAdjustFilters:[CIFilter] = []
    var intermediate = CIImage()
    let scaleFiter = CIFilter.lanczosScaleTransform()
```

Fun with Pixels

The intermediate function

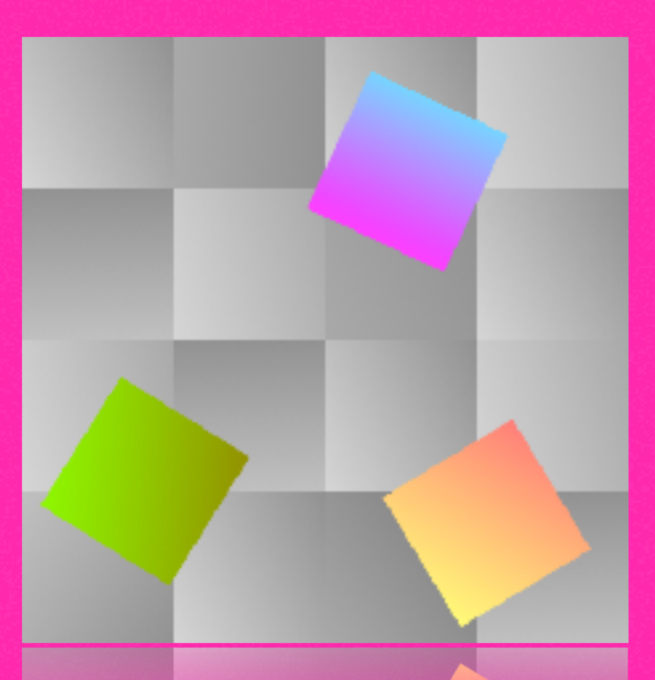
Scaling is already implemented.

```
func intermediateImage(forIndex: Int, adjust: Bool,
                       channels: Float, passes: Float) {
    autoAdjust = adjust

    selectedIndex = forIndex
    let originalImage = CIImage(image: allImages[selectedIndex])
    let max = originalImage!.extent.width > originalImage!.extent.height ?
              originalImage!.extent.width : originalImage!.extent.height

    scaleFiter.scale = 1000/Float(max)
    scaleFiter.inputImage = originalImage
    var inputImage = scaleFiter.outputImage!

    if(true == autoAdjust)
      {
         inputImage = autoAdjust(image: inputImage)
      }
      self.intermediate = inputImage
}
```



Fun with Pixels

Original image for CIKMeans

Already a very reduced set of colors.

Fun with Pixels

Color reduction via CIKMeans

By applying the kMean and the corresponding palletise filter the colors are reduced to 4 and two. Result is an almost binary image.

Palette of only 4 colors

Minimum of two colors

Fun with Pixels

Implementing the mean filter.

Compared to simple linear chaining this procedure is somewhat more complex. At first the original image is the input for the filter CIKMeans. Based on the count of channels for the number of colors and the passes to calculate them the output image of the filter is a palette, meaning a height of one pixel and a width corresponding to the count of channels. This palette of colored pixels will be one parameter of the CIPalettize-filter. The other is again the original as inputImage, the output image is the final image.

Most pieces and the interface in SwiftUI are already written, the documentation on Apple's website is sufficient. It should be no problem to fill in the code and correct the last line of the previous function.

```
private func filterMean(image: CIImage, channels: Float, passes: Float) -> CIImage
{ return image}

self.intermediate = filterMean(image: inputImage, channels: channels, passes: passes)
```

Fun with Pixels

Different channel and passes

Channels: 2

2 passes

2/2-2/8 are the same result

8 passes

4

6

8

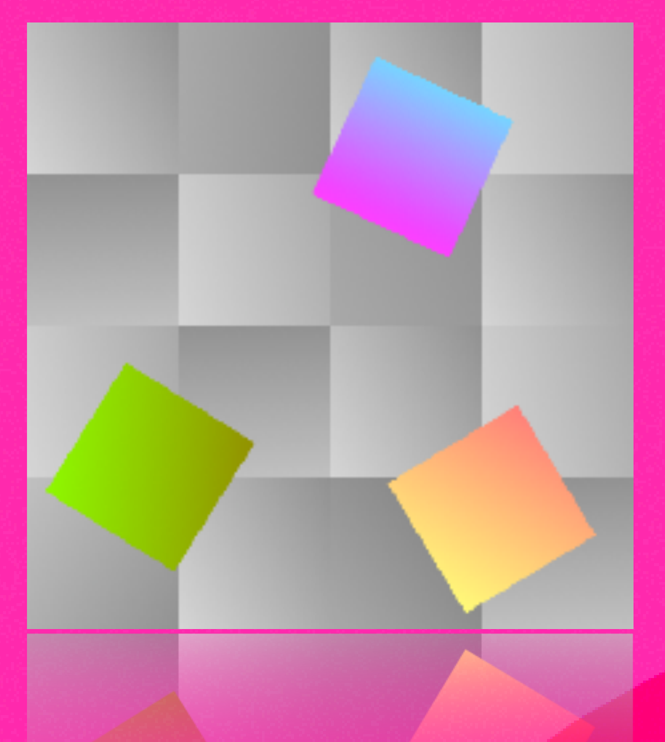
...and some tweaking

Fun with Pixels

The Palette

Just to mention it, the palette of the image distributed as a CIImage

8 channels and 8 passes.



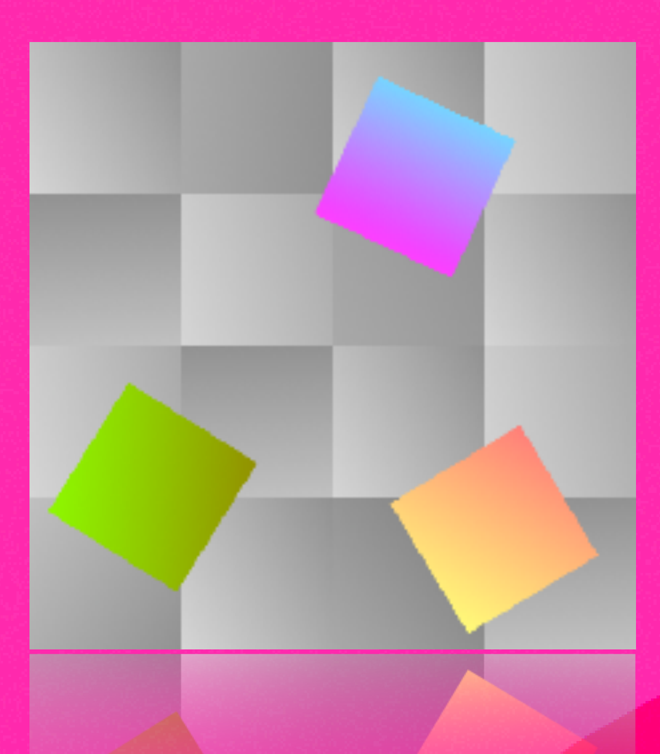
Fun with Pixels

Original and mean images

A collection of four Images side by side with their sources.

Average color filter

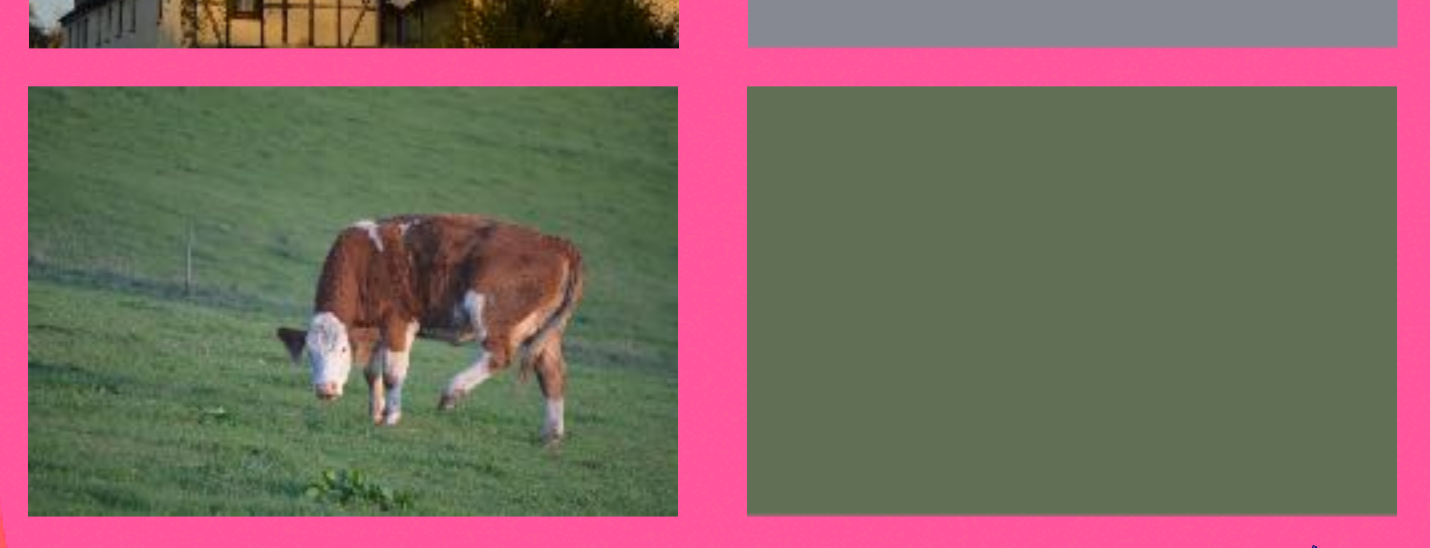
The average filter is the ultimate reduction filter. The output image Is only one pixel with one color. To view that pixel, the image has to be enlarged.



Fun with Pixels

A set of average colors

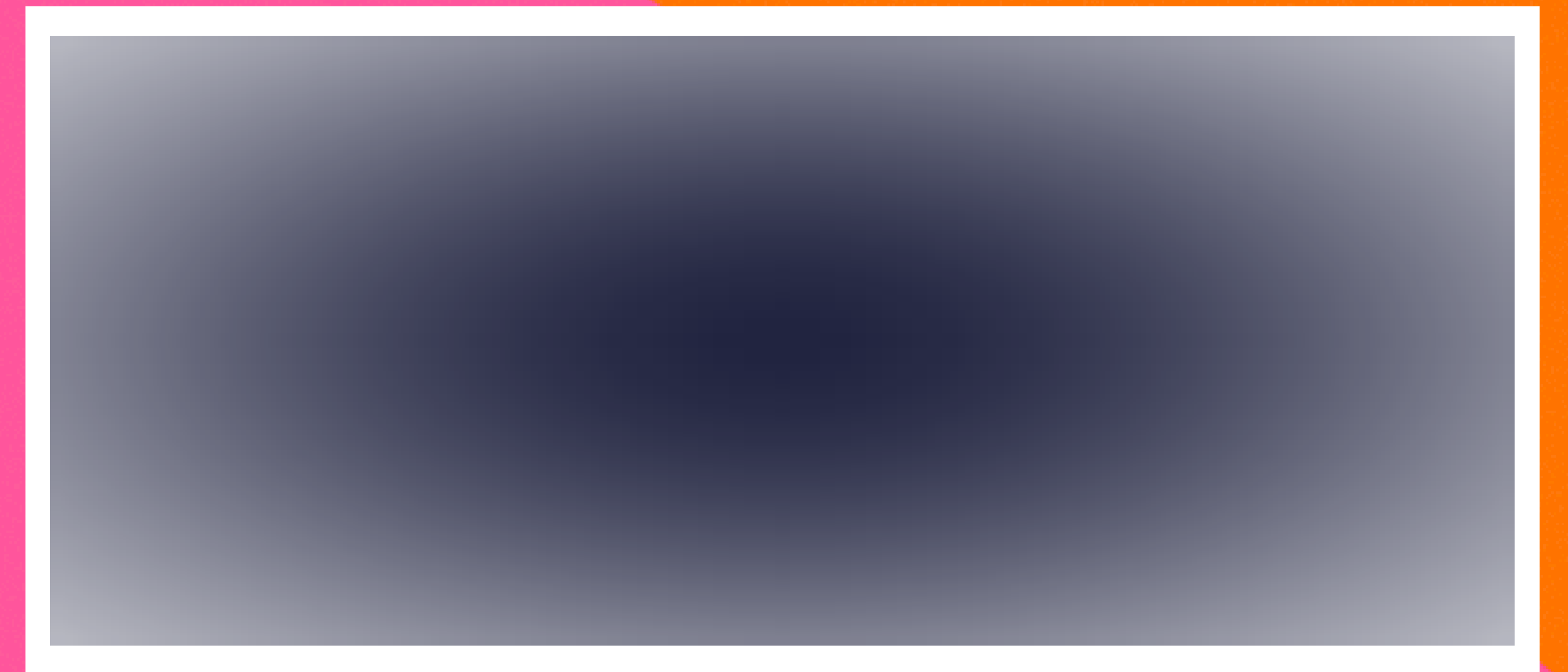
Naturally, the color tends to some shades of gray.

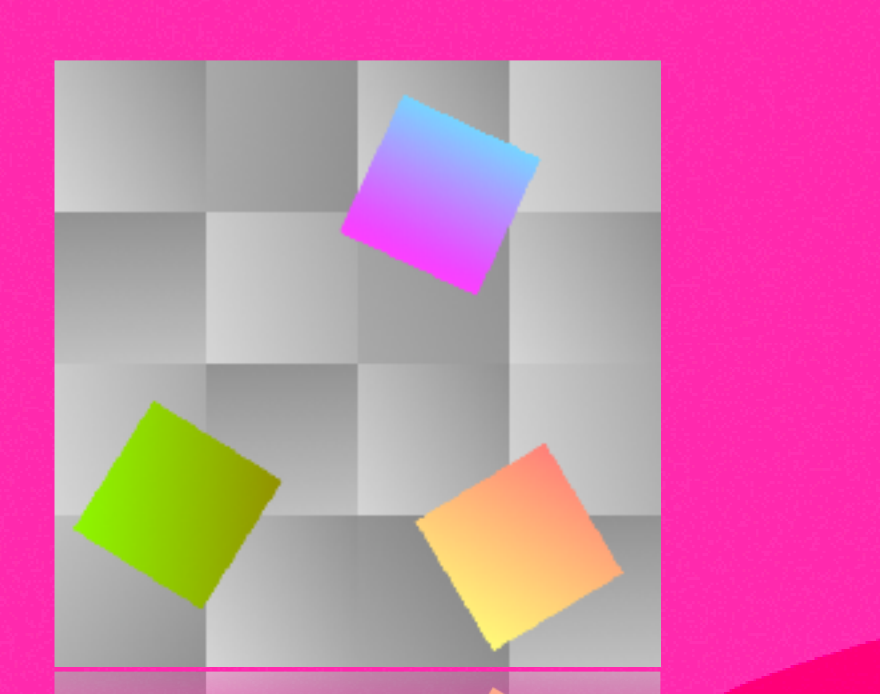


Fun with Pixels

The icing: implement the average color

There is some space left between the two arrows under the source image suitable for the final task of displaying the average color of the source. Just obtain a filter of type CIAreaAverage and set the image and its extent accordingly. Then run the filter. The main problem is that the single result pixel has to be scaled up. The common filter CILanczosScaleTransform is too sophisticated. The screenshot indicates some shading, which upon examination is verified. What is needed is a much more simple filter like the CIAffineTransform to get a flat filled rectangle. Maybe instantiating the filter requires key-value-coding.

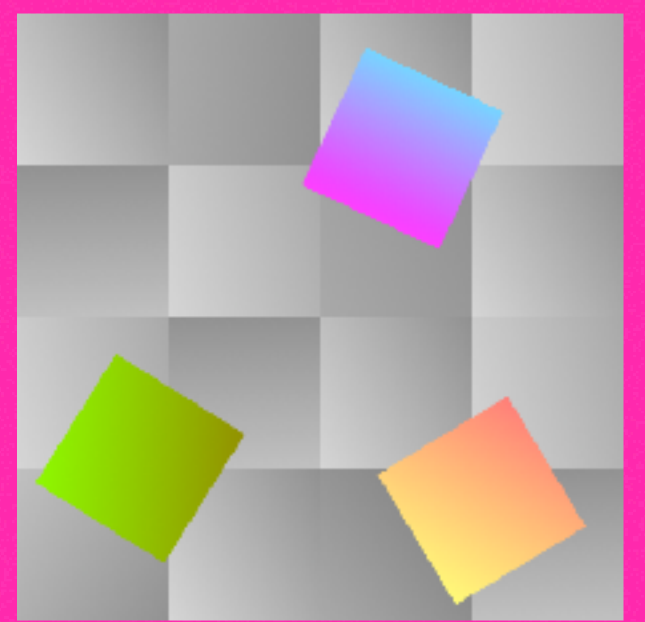




Fun with Pixels

One more thing: SwiftImage

There is a framework around with a total different approach:
https://github.com/koher/swift-image
Not atomising the image into pixels and kernels but keeping them around in a Swift-array renders an holistic view on the image. As a standard array pixels all values can be sorted, filtered and shuffled. With some constraints the framework then renders the array back into an image.
The main difference is, that Core Image with its atomic pixels in the filters' kernels provides no means to address all pixels all at once.
Especially shuffle is a kind of a useful procedure here.



Fun with Pixels

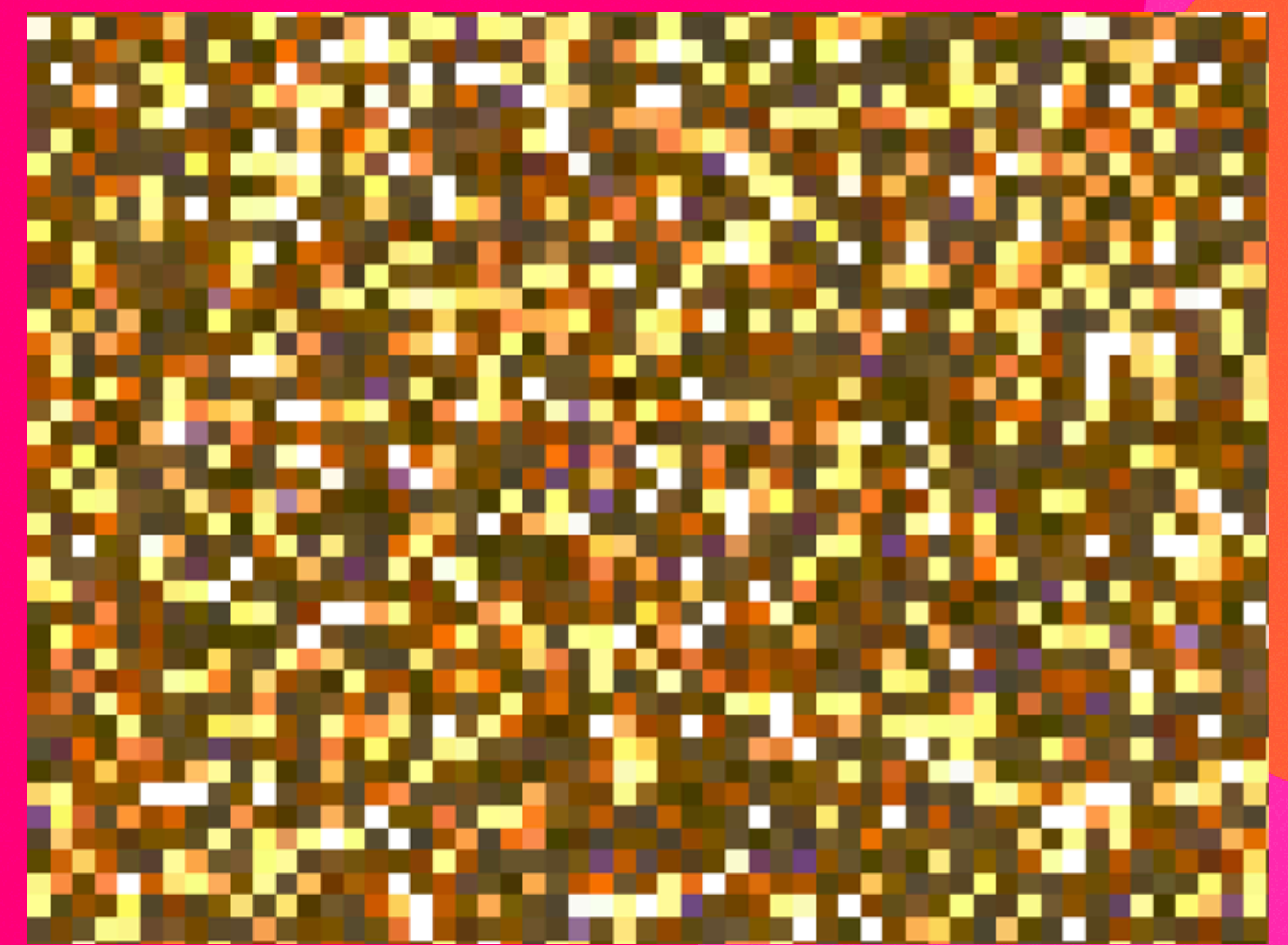
Shuffled pixels of the image

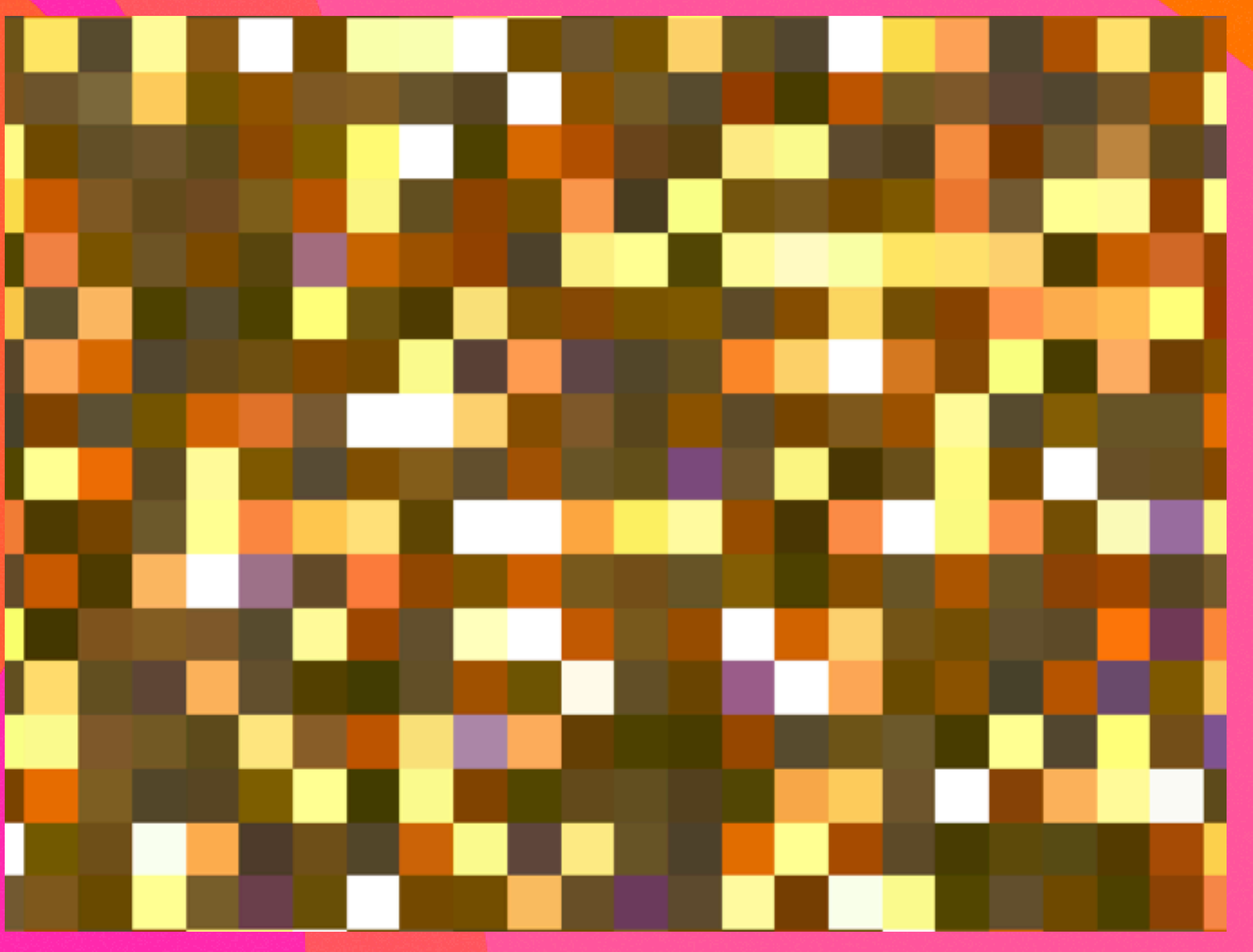
Once a pixel lost his place in the grid, the information of its position is gone. Mixed up between all the other pixels it retains the only value it is reduced to, its precious color. Although the image looks like a single color, all pixels are still there.

Fun with Pixels

Visibility of pixels

What at first glance appears to be meaningless random noise, is at second glance a remembrance of the source image. Finally the blue, white, yellow and dark pixels are awaiting reconstruction in everybody's mind.





Fun with Pixels

Average color for comparison

Only at first glance the shuffled and the averaged images are looking similar.

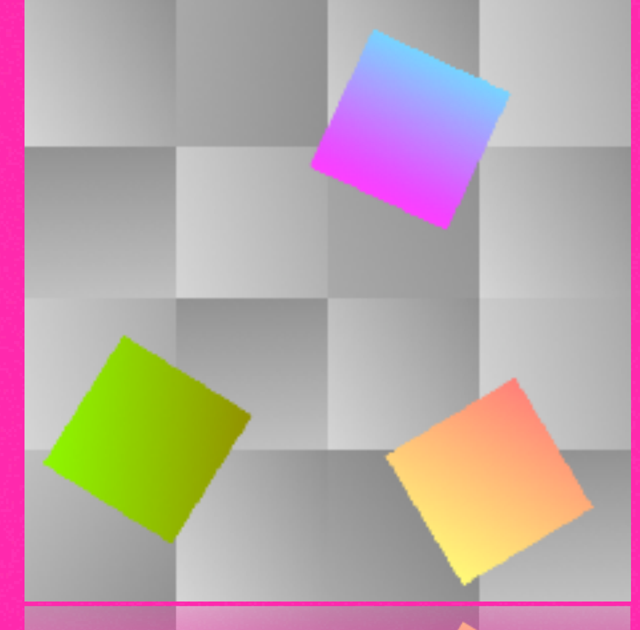
The image here is reduced to one single color and lost almost all of its informations.

Fun with Pixels

Sorted Pixel

Only at first glance the shuffled and the averaged images are looking similar.

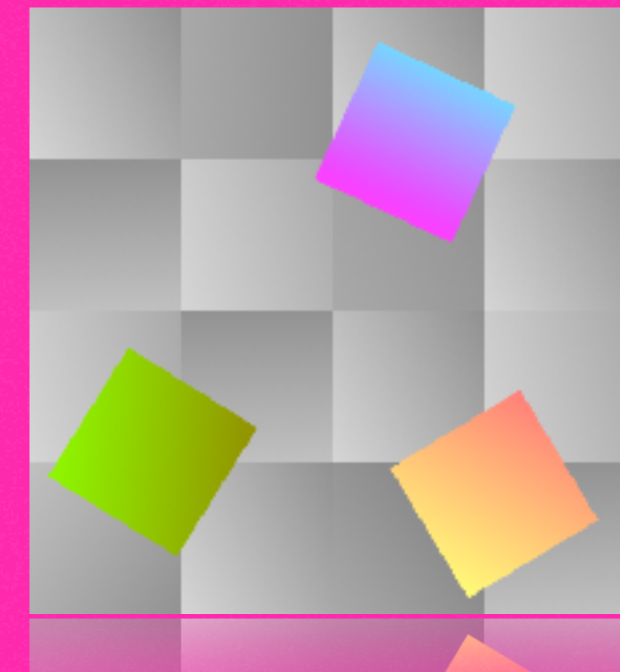
All pixels are sorted according to their color values.



Fun with Pixels

Different grades of noise applied to a binary image.

Fun with Pixels



Scaled up to get a better view on the effect.

Fun with Pixels

Thank you.