Article

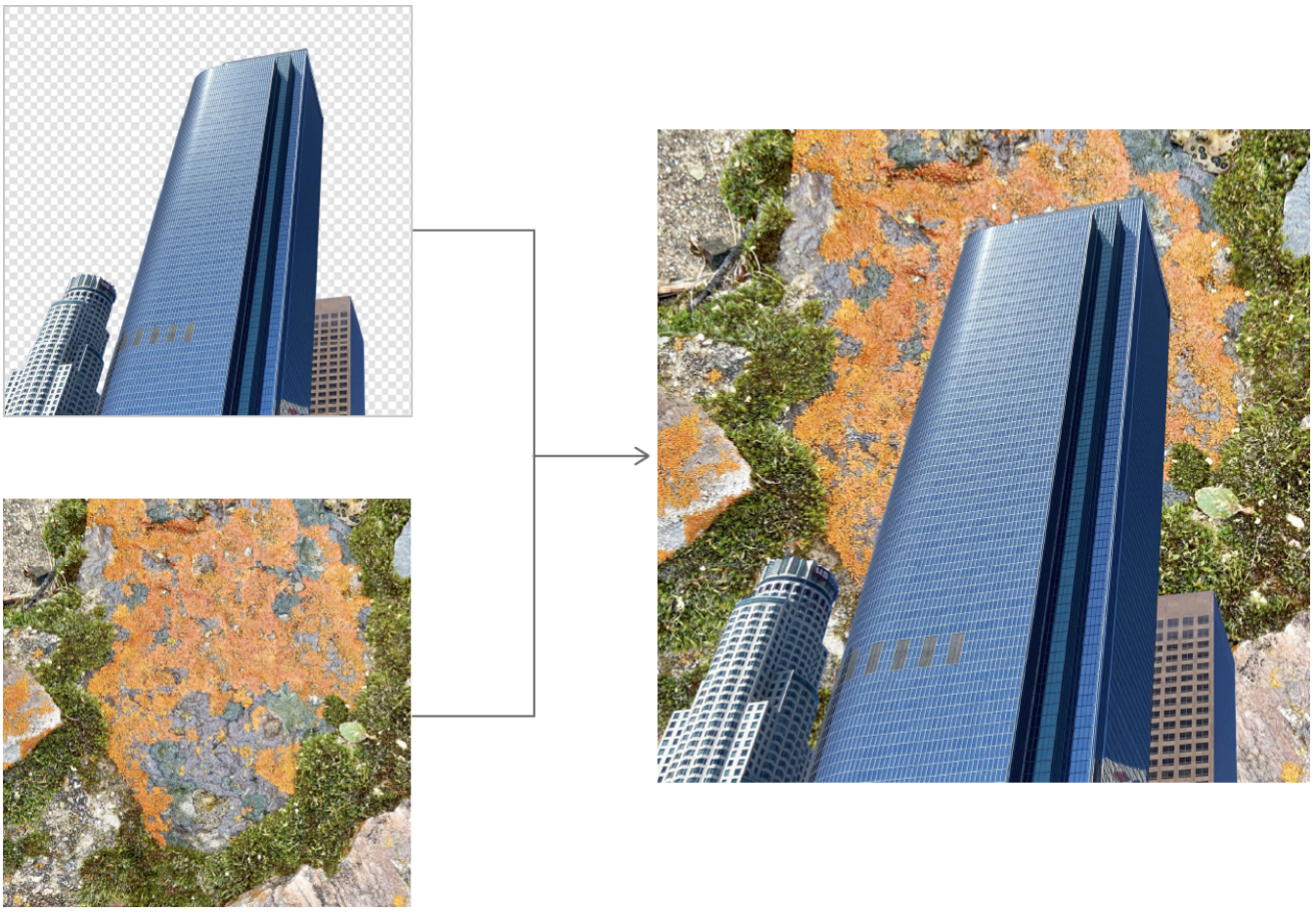
Compositing images with alpha blending

Combine two images by using alpha blending to create a single output.

Overview

vImage provides a suite of functions for compositing two source images into one output. *Alpha blending* uses the alpha value of each pixel in the top layer to determine the visibility of the bottom layer.

The figure below shows the alpha composite of an image of a building with a transparent background over an image of a texture. The transparent part of the building image has an alpha of zero, indicating the background layer is fully visible.



Perform an alpha composite

Use the vImagePremultipliedAlphaBlend_ARGB8888(_:_:_:_:) function to blend two images when the top image contains alpha information (for example, when compositing text over a photograph). The images that you pass to vImagePremultipliedAlphaBlend_ARGB8888(_:_:_:_:) need to have the premultiplied alpha information in the first channel.

The following function performs an alpha composite of two CGImage instances and returns the result as a CGImage instance:

```
func alphaComposite(topImage: CGImage, bottomImage: CGImage) -> CGImage? {
    // Create source and destination vImage buffers.
    guard
        let topImageBuffer = try? vImage_Buffer(cgImage: topImage),
        let bottomImageBuffer = try? vImage_Buffer(cgImage: bottomImage),
        var destinationImageBuffer = try? vImage_Buffer(size: topImageBuffer.size,
                                                        bitsPerPixel: 8 * 4)
    else {
        return nil
    }

    defer {
```

```
        topImageBuffer.free()
        bottomImageBuffer.free()
        destinationImageBuffer.free()
    }

    withUnsafePointer(to: topImageBuffer) { topPtr in
        withUnsafePointer(to: bottomImageBuffer) { bottomPtr in

            // Ensure the bottom image and top image are ARGB.
            convertToARGB(bottomPtr, alphaInfo: bottomImage.alphaInfo)
            convertToARGB(topPtr, alphaInfo: topImage.alphaInfo)

            // Ensure the top layer is premultiplied.
            premultiply(topPtr, alphaInfo: topImage.alphaInfo)

            // Perform the composite operation.
            vImagePremultipliedAlphaBlend_ARGB8888(topPtr,
                                                   bottomPtr,
                                                   &destinationImageBuffer,
                                                   vImage_Flags(kvImageNoFlags))
        }
    }

    if let destinationFormat = vImage_CGImageFormat(
        bitsPerComponent: 8,
        bitsPerPixel: 8 * 4,
        colorSpace: CGColorSpaceCreateDeviceRGB(),
        bitmapInfo: CGBitmapInfo(rawValue: CGImageAlphaInfo.first.rawValue)) {
        return try? destinationImageBuffer.createCGImage(format: destinationFormat)
    }
    return nil
}
```

Because the blending function `vImagePremultipliedAlphaBlend_ARGB8888(_:_:_:_:)` requires that the alpha channel source buffers are first, the code calls `convertToARGB(_:alphaInfo:)` to ensure the correct ordering. The `convertToARGB(_:alphaInfo:)` function checks the channel ordering from the source image's alpha information and permutes the channels if the ordering isn't alpha-first.

```
func convertToARGB(_ buffer: UnsafePointer<vImage_Buffer>,
                   alphaInfo: CGImageAlphaInfo) {
```

```
    let alphaLasts = [ CGImageAlphaInfo.last,
                       CGImageAlphaInfo.premultipliedLast,
                       CGImageAlphaInfo.noneSkipLast ]

    if alphaLasts.contains(alphaInfo) {
        vImagePermuteChannels_ARGB8888(buffer,
                                       buffer,
                                       [3, 0, 1, 2],
                                       vImage_Flags(kvImageNoFlags))
    }
}
```

The `premultiply(_:alphaInfo:)` function uses a `CGImage` instance's alpha information to determine whether an image contains premultiplied alpha. This function ensures that the top layer that the function passes to the composite operation contains premultiplied pixels.

```
func premultiply(_ buffer: UnsafePointer<vImage_Buffer>,
                 alphaInfo: CGImageAlphaInfo) {

    let premultiplieds = [ CGImageAlphaInfo.premultipliedFirst,
                           CGImageAlphaInfo.premultipliedLast ]

    if !premultiplieds.contains(alphaInfo) {
        vImagePremultiplyData_ARGB8888(buffer,
                                       buffer,
                                       vImage_Flags(kvImageNoFlags))
    }
}
```

Perform an alpha composite with a single alpha value

vImage provides functions to perform an alpha composite using a single alpha value. The functions apply the constant alpha value you supply combined with the top image's existing alpha to the top layer's color channels and alpha channels. For each pixel, the constant alpha functions perform the following operation:

```
destColor = (srcTopColor * constAlpha * 255  + (255*255 - srcTopAlpha * constAlpha)
destAlpha =  (srcTopAlpha * constAlpha * 255 + (255*255 - srcTopAlpha * constAlpha)
```

To perform an alpha composite with a constant alpha, replace the call to `vImagePremultiplied AlphaBlend_ARGB8888(_:_:_:_:)` in the code listing in Compositing images with alpha

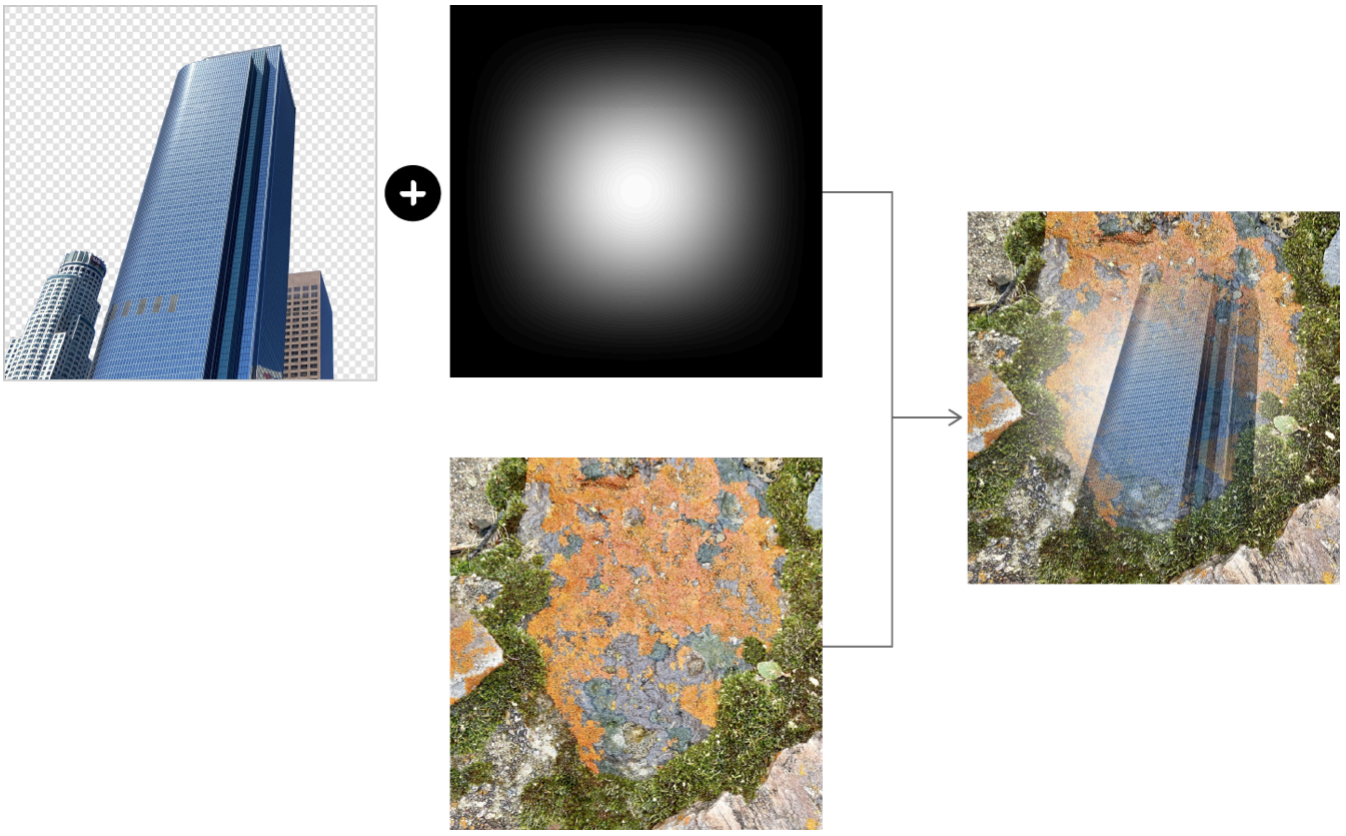
blending with the following code:

```
// Perform the composite operation.
vImagePremultipliedConstAlphaBlend_ARGB8888(topPtr,
                                            Pixel_8(191),
                                            bottomPtr,
                                            &destinationImageBuffer,
                                            vImage_Flags(kvImageNoFlags))
```

The following image shows the result of compositing with `vImagePremultipliedConstAlphaBlend_ARGB8888(_:_:_:_:_:)` using a constant alpha with a value of 191. The background of the building remains transparent, but the parts of the image that were originally opaque now show some of the lower layer.

Perform an alpha composite with an overwritten alpha channel

Use the `vImageOverwriteChannels_ARGB8888(_:_:_:_:_:)` function to overwrite an image's alpha channel. The following shows an example of setting a radial gradient as the alpha channel of an image to produce a vignette style blend:



The `vImageOverwriteChannels_ARGB8888(_:_:_:_:_:)` function requires buffers with nonpremultiplied alpha.

The `radialComposite(topImage:bottomImage:)` function performs an alpha composite of two `CGImage` instances with a procedural radial gradient.

```
func radialComposite(topImage: CGImage,
                     bottomImage: CGImage) -> CGImage? {
    // Create source and destination vImage buffers.
    guard
        let topImageBuffer = try? vImage_Buffer(cgImage: topImage),
        let bottomImageBuffer = try? vImage_Buffer(cgImage: bottomImage),
        var gradientImageBuffer = try? vImage_Buffer(size: topImageBuffer.size,
                                                     bitsPerPixel: 8),
        var destinationImageBuffer = try? vImage_Buffer(size: topImageBuffer.size,
                                                        bitsPerPixel: 8 * 4)
    else {
        return nil
    }

    defer {
        topImageBuffer.free()
        bottomImageBuffer.free()
        gradientImageBuffer.free()
        destinationImageBuffer.free()
```

```
    }

    withUnsafePointer(to: topImageBuffer) { topPtr in
        withUnsafePointer(to: bottomImageBuffer) { bottomPtr in

            // Ensure the top image and the bottom image are ARGB.
            convertToARGB(topPtr, alphaInfo: topImage.alphaInfo)
            convertToARGB(bottomPtr, alphaInfo: bottomImage.alphaInfo)

            // Populate `gradientImageBuffer` with a radial gradient.
            makeRadialGradient(&gradientImageBuffer)

            // Unpremultiply before overwriting, if required.
            unpremultiply(topPtr, alphaInfo: topImage.alphaInfo)

            // Overwrite the top layers's alpha channel with the radial gradient.
            vImageOverwriteChannels_ARGB8888(&gradientImageBuffer,
                                             topPtr, topPtr,
                                             0x8,
                                             vImage_Flags(kvImageNoFlags))

            // Premultiply the top layer.
            vImagePremultiplyData_ARGB8888(topPtr, topPtr,
                                           vImage_Flags(kvImageNoFlags))

            // Perform the alpha blend of the top layer over the bottom layer.
            vImagePremultipliedAlphaBlend_ARGB8888(topPtr,
                                                   bottomPtr,
                                                   &destinationImageBuffer,
                                                   vImage_Flags(kvImageNoFlags))
        }
    }

    if let destinationFormat = vImage_CGImageFormat(
        bitsPerComponent: 8,
        bitsPerPixel: 8 * 4,
        colorSpace: CGColorSpaceCreateDeviceRGB(),
        bitmapInfo: CGBitmapInfo(rawValue: CGImageAlphaInfo.first.rawValue)) {
        return try? destinationImageBuffer.createCGImage(format: destinationFormat)
    }
    return nil
}
```

The code calls `unpremultiply(_:alphaInfo:)` to unpremultiply an image if its alpha information indicates it contains premultiplied pixels.

```
func unpremultiply(_ buffer: UnsafePointer<vImage_Buffer>,
                   alphaInfo: CGImageAlphaInfo) {

    let premultiplieds = [ CGImageAlphaInfo.premultipliedFirst,
                           CGImageAlphaInfo.premultipliedLast ]

    if premultiplieds.contains(alphaInfo) {
        vImageUnpremultiplyData_ARGB8888(buffer,
                                         buffer,
                                         vImage_Flags(kvImageNoFlags))
    }
}
```

The `radialComposite(topImage:bottomImage:)` function calls `makeRadialGradient(_:)` to populate the gradient buffer with a radial gradient. The code generates the radial gradient by multiplying a `height * 1` matrix by a `1 * width` matrix. Both factors contain values that follow a bell-shaped curve.

```
func makeRadialGradient(_ destination: inout vImage_Buffer) {
    let width = Int(destination.size.width)
    let height = Int(destination.size.height)

    var gradientValues = [Float](unsafeUninitializedCapacity: width * height) {
        buffer, initializedCount in

        let verticalWindow = vDSP.window(ofType: Float.self,
                                         usingSequence: .hanningDenormalized,
                                         count: height,
                                         isHalfWindow: false)

        let horizontalWindow = vDSP.window(ofType: Float.self,
                                           usingSequence: .hanningDenormalized,
                                           count: width,
                                           isHalfWindow: false)

        vDSP_mmul(verticalWindow, 1,
                  horizontalWindow, 1,
                  buffer.baseAddress!, 1,
                  vDSP_Length(height),
```

```
                    vDSP_Length(width), 1)

        initializedCount = width * height
    }.map {
        return Pixel_8($0 * Float(Pixel_8.max))
    }

    gradientValues.withUnsafeMutableBufferPointer { gradientPtr in
        let gradientBuffer = vImage_Buffer(data: gradientPtr.baseAddress,
                                           height: vImagePixelCount(height),
                                           width: vImagePixelCount(width),
                                           rowBytes: width)

        try? gradientBuffer.copy(destinationBuffer: &destination,
                                 pixelSize: 1)
    }
}
```

See Also

Image Processing Essentials

- Converting bitmap data between Core Graphics images and vImage buffers
  Pass image data between Core Graphics and vImage to create and manipulate images.
- Creating and Populating Buffers from Core Graphics Images
  Initialize vImage buffers from Core Graphics images.
- Creating a Core Graphics Image from a vImage Buffer
  Create displayable representations of vImage buffers.
- Building a Basic Image-Processing Workflow
  Resize an image with vImage.
- Applying geometric transforms to images
  Reflect, shear, rotate, and scale image buffers using vImage.
- Compositing images with vImage blend modes
  Combine two images by using blend modes to create a single output.
- Applying vImage operations to regions of interest

Limit the effect of vImage operations to rectangular regions of interest.

Optimizing image-processing performance

Improve your app's performance by converting image buffer formats from interleaved to planar.

vImage

Manipulate large images using the CPU's vector processor.