



Flutter iOS PlatformView BackdropFilter (Blur)

SUMMARY

Implementing iOS PlatformView BackdropFilter. (Blur)

Authors:

Chris Yang(@cyanglaz), Emily Best(@emilyabest), Javon Thomas(@JTKryptic)

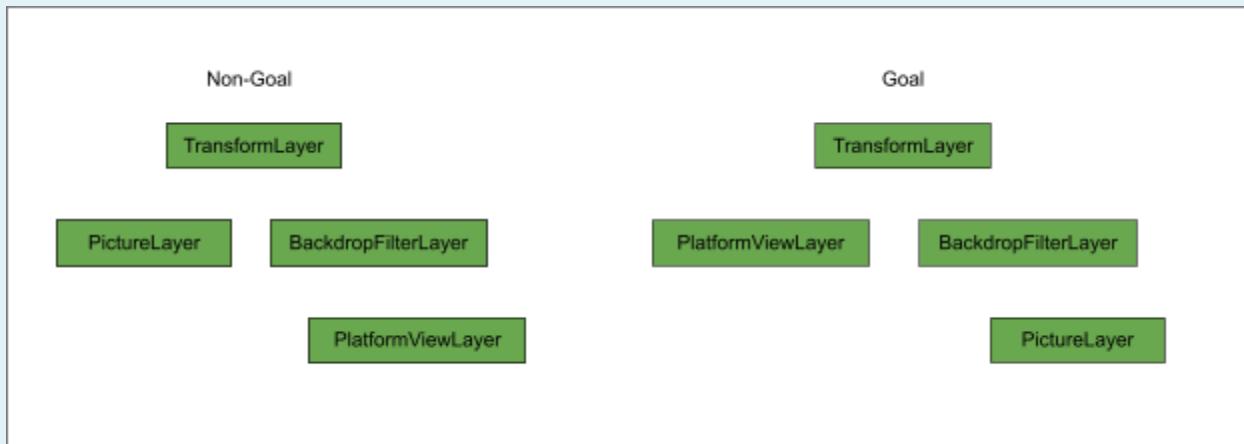
Go Link: flutter.dev/go/ios-platformview-backdrop-filter-blur

Created: 6/2022 / Last updated: 10/2022

WHAT PROBLEM IS THIS SOLVING?

Add the ability for an iOS PlatformView to be filtered when it is underneath a backdrop filtered widget. See: <https://github.com/flutter/flutter/issues/43902>

Note that when the child of BackdropFilterLayer is a PlatformView, and there are no PlatformViews underneath the BackdropFilterLayer, the PlatformView does not need to be filtered. See the layer trees below:



BACKGROUND

A [BackdropFilter widget](#) applies a filter to the existing painted content and then paints its child. The most common use case of the BackdropFilter widget is to blur the background, but any image filters can be added to the background.

PlatformViews on iOS are not rendered with Skia, instead, they are rendered with Quartz. As a result, mutations on an iOS PlatformView do not work automatically. Each mutation has to be implemented separately for iOS PlatformViews. For example, transform and clipRect on iOS PlatformView is implemented in [this PR](#). BackdropFilter works differently from other mutators because it applies to a given region rather than a view/layer. A standard approach to apply a backdrop filter is to screenshot the requested region and apply the filter to that screenshot.

In general, there are 3 options to apply mutation to a frame containing PlatformViews.

1. Use Flow to apply the mutation to all layers.
2. Use Quartz to apply the mutation to all layers.
3. Use Flow to apply the mutation to none-PlatformView layers and Quartz to apply the mutation to PlatformView layers.

See "[Decide The Top Level Approach](#)" section for more information on comparing the 3 options for backdrop filter mutation.

Glossary

- **Flow** - Flutter's compositor. Flow knows the layer structures of each frame, including PlatformView. Flow does not know the content of the PlatformView.
- **Quartz** - Apple's 2D drawing engine. Quartz has the knowledge of the PlatformView's content. Quartz does not know the layer structure of a frame that is composite by Flow.
- **FlutterView** - The UIView that Flutter's widgets are rendered on. FlutterView is the parent view of the PlatformView.
- **PlatformView** - 1. A type of flutter widget that embeds UIView to Flutter widget tree; 2. The UIView that is embedded.
- **PlatformView Layer** - The layer represents a PlatformView in the Flutter engine layer tree.
- **BackdropFilter Layer** - The layer represents a backdrop filter in the Flutter engine layer tree. The BackdropFilter layer applies its filter to the layers beneath itself in a given region. In the context of this project, when a PlatformView is beneath a BackdropFilter Layer, the PlatformView needs to be filtered.
- **PlatformView Overlay (Or Overlay)** - The UIView that Flutter's widgets are rendered on if the widgets cover PlatformView. The rect of an overlay is the intersection of all the flutter widgets and the PlatformView. If a flutter widget partially covers the PlatformView, the overlay's rect will be only part of the widget's rect. Part of the widget will be rendered on the Overlay and part of the widget will be rendered on the FlutterView. See [Unobstructed PlatformView](#) for how the overlay rect is decided and the reason behind it.

- **Mutation** - An effect that applies to the PlatformView which results in a visual change of the view. Transform, ClipRect, Opacity and Backdrop Filters are types of the mutations.
- **Mutator** - A c++ object that represents a mutation.
- **Mutator Stack** - A stack contains all the mutations to be applied to the PlatformView. Each PlatformView has its own mutator stack.
- **Layer Tree** - The layer tree contains the container layers for the mutators (transform, clip rect, backdrop filter, etc) and the leaf layers for the views that the mutators will be applied to (platform views, picture layers). When a frame is built a layer tree of all the mutators and views used is created, and in order to display the frame, two important functions are called: Preroll and Paint.
- **Preroll** - The preroll function navigates through the layer tree and determines what mutators will be applied to PlatformViews. For platform views, the function pushes the mutator to a global stack, and when it reaches the platform view layer, it creates a copy of the mutator stack for that specific view of all the mutators that should be applied to it during paint.
- **Paint** - The paint function essentially applies all the mutators to each platform view from their respective mutator stack.

OVERVIEW

The approach we decide to use is to let non-PlatformView widgets to be blurred as is and apply the blur effect to the PlatformView using a gaussian blur [CAFiler](#). This approach is similar to how we implemented other mutators. See the "[Decide The Top Level Approach](#)" section for more details on how we came to this solution.

Non-goals

We do not discuss implementing a BackdropFilter with an arbitrary ImageFilter in this integration. Supporting other ImageFilters on PlatformView is a completely separate topic and deserves its own project and design document.

DETAILED DESIGN/DISCUSSION

[Decide The Top Level Approach](#)

Option 1. Use Flow to apply the mutation to all layers.

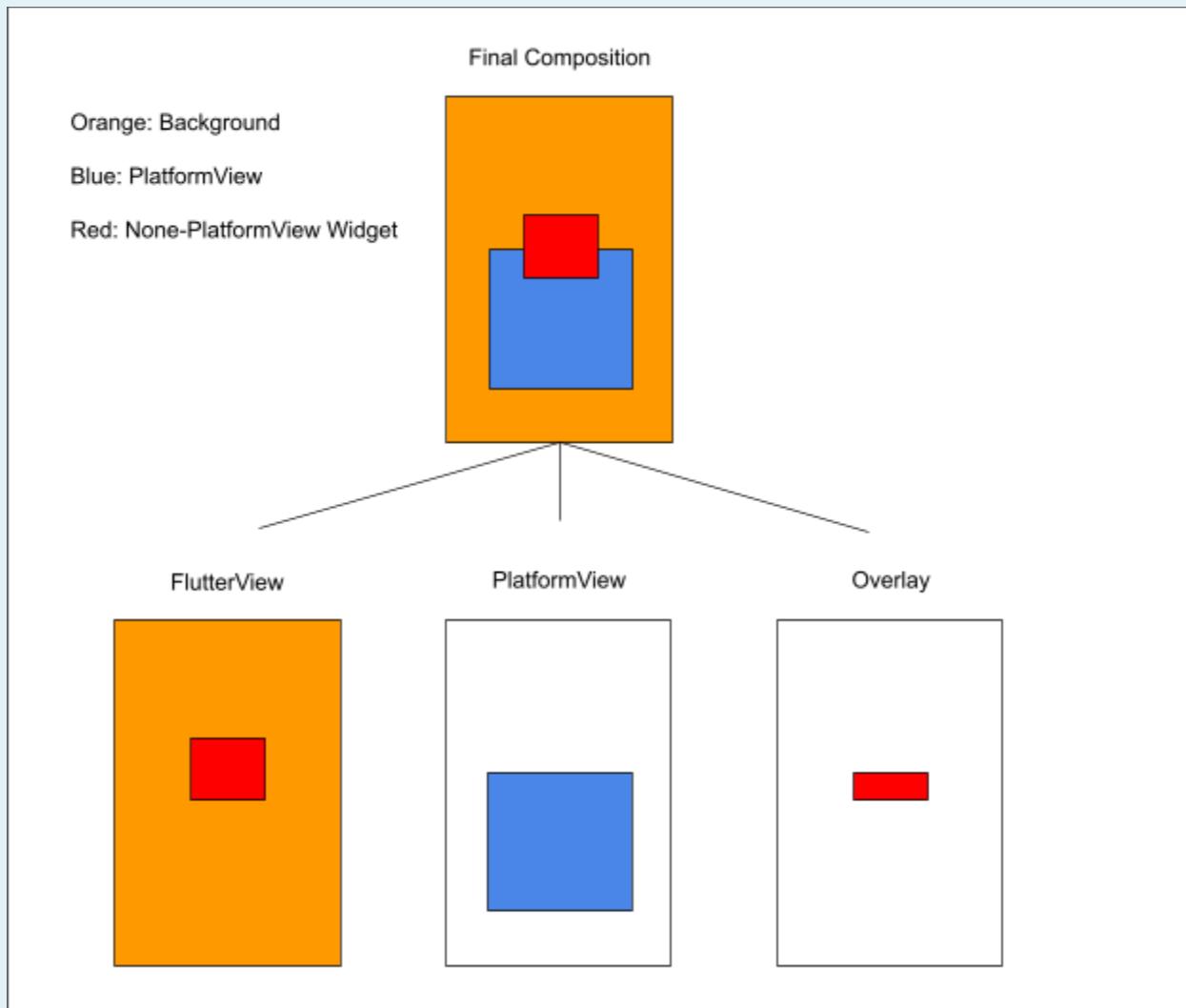
Backdrop filter mutation requires the compositor to know the content of the layer. Because Flow does not know the content of PlatformView, option 1 is impossible. One workaround would be to convert the content of the PlatformView to a texture and feed the texture to Flow. We explored the option and there are a few severe dropbacks:

- There is no arbitrary Quartz API to convert a UIView to texture directly. One has to snapshot the UIView and convert the resulting UIImage to a texture.
- Snapshotting UIView does not always work for all types of UIViews. For example, WKWebView cannot be snapshotted. See "[Take a screenshot of the PlatformView and use the CIGaussianBlur filter to blur the screenshot](#)" section for more discussion on the drawbacks of snapshotting PlatformViews.

Option 2. Use quartz to apply the mutation to all layers.

This can be potentially done by adding a UIVisualEffectView as a subView of the FlutterView and setting the right ZIndex. In a flutter app with PlatformViews, as a byproduct of "[Unobstructed PlatformView](#)", a layer's ZIndex in Flow does not necessarily match its ZIndex in Quartz.

Below is an example of a frame containing PlatformView and Overlay:



In this example, The final composition is made of 3 layers in Quartz. The FlutterView contains the background orange and the red rectangle widget, the blue

PlatformView, and the overlay contains part of the red rectangle widget that covers the PlatformView.

Let's make a scenario where the backdrop filter is to blur everything behind the red rectangle widget. If we add a backdrop between PlatformView and Overlay, the top part of the red widget will be blurred, because the top part is rendered in FlutterView, which is under the backdrop. This makes option 2 impossible under the current PlatformView/Overlay implementation.

Option 3. Use Flow to apply the mutation to non-PlatformView layers and Quartz to apply the mutation to PlatformView layers. (Selected)

Because neither option 1 or option 2 is possible, the blur on PlatformView needs to be applied with Quartz while we keep the non-PlatformView widgets to be blurred with Flow. Because the blur effect for PlatformView and non-PlatformView is applied separately without the knowledge of the pixels outside of the blurred region, the edge between PlatformView and non-PlatformView widget is ignored with a standard gaussian blur algorithm, which leads to some visual artifacts. See the [Apply Filter Effect](#) section for more details.

The solution for this approach includes 2 parts:

1. Propagate the BackdropFilter mutation to the PlatformView. This includes:
 - a. adding a new backdrop filter mutator type
 - b. push the backdrop filter mutator to the mutator stack.
2. After getting the BackdropFilter information from the mutator stack, apply the blur effect on the PlatformView. There are 3 major approaches to blur a PlatformView (iOS UIView):
 - a. Use the "gaussianBlur" filter provided by [UIVisualEffectView](#).
 - b. Take a screenshot of the PlatformView and use [CIGaussianBlur](#) filter to blur the screenshot.
 - c. Use the "gaussian" filter provided by [SwiftUI.blur](#).

[Propagate The BackdropFilter mutation](#)

Possible Approaches

- Tracking each visited PlatformView
- Rewalking the layer tree

Approach 1: Tracking each visited platform view

This approach essentially keeps each platform view leaf layer that is crossed during the tree walking. When a platform view is encountered in the layer tree, the platform view id of that view will be stored in a list. Normally, when a backdrop filter mutator is encountered in the layer tree, platform view layers that were

traversed before the backdrop filter call do not know to blur. To combat this, we would use the list of platform view ids to manually push the mutation to the mutator stack of each platform view that needs to be blurred. This would work because at any point in the layer tree walking, the list would only contain the platform view ids of previously encountered views, so we would only touch the stacks of the platform views we need to blur.

Possible Implementation

To implement this approach, a list with its getter and setter methods could be created within `embedded_views.h` to hold the visited platform views. The visited platform view ids could be pushed to the list from `platform_view_layer.cc`. To update the mutator stacks of the visited platform views, the blur mutation would be pushed from `backdrop_filter_layer.cc`, and a method to receive the mutation, cycle through the list, and push the mutation to each view could be created and implemented within `FlutterPlatformViews.mm`.

Approach 2: Rewalking the layer tree

This approach would rewalk the layer tree when a `backdrop_filter_layer`'s `preroll` method is called and push the blur mutation to each platform views mutator stack it encounters beforehand in real time. When a platform view is encountered in the layer tree, a flag will be set. As the tree walking continues, if a backdrop filter mutation is found later in the tree, it will call for a second walking of the tree up to the point of the current backdrop filter call, and each platform view encountered during this second walking will have the blur mutation pushed to their mutator stack

Possible Implementation

To implement this approach, a flag would be created within the `embedded_views.h`. When a platform view is encountered in the layer tree, the flag would be triggered through `platform_view_layer.cc`. When a backdrop filter mutation is later found in the layer tree, a separate second walking of the tree would be triggered by the backdrop filter layer, and the blur mutation would be pushed from the backdrop filter layer as well. From here, the blur could either be received by the platform view layer and pushed to the mutator stacks from `platform_view_layer.cc`, or the blur mutation could be pushed to `FlutterPlatformViews.mm`, along with the platform view id from `platform_view_layer.cc`, and the mutator stacks would be updated from there.

Approach Comparison

Readability

Approach 1 is the easier approach of the two to read and understand. The concept of storing the platform views to a list as they are encountered in the layer tree and

manually pushing the blur mutation to each view in the list is very simple and easy to understand. Approach 2 would require a little more of an understanding of the layer tree and how the tree walking is achieved by the flutter engine.

Implementation

The implementation of Approach 1 would be simpler than that of Approach 2. The process of creating the list, populating it with the platform view ids, and pushing the blur to the mutator stacks is not a very complex process and should be fairly simple to implement. While creating and triggering the flags for Approach 2 may be simple, implementing the additional walk of the layer tree will be a lot more complex of a process. In addition, the second walking of the layer tree does not walk the entire tree, but must stop at the backdrop filter mutation that triggered the second walking in the first place. Because of this, this approach would also require the implementation of a way for the layer tree to know to stop at the backdrop filter layer, and how to know which backdrop filter layer within the tree to stop at if there are multiple.

Space Efficiency

Approach 1 creates the list of platform view ids which, although minimal, still takes up space and affects the build time of the engine and apps within flutter. This can be improved slightly with lazy initialization, which would only allocate space for the list once the first platform view id is populated. Additionally, if a backdrop filter mutator is not found within the layer tree walking, a list full of platform view ids was still created but unused, taking up space unnecessarily. Approach 2 does not create any additional objects or data structures besides the flag, making this approach more space efficient.

Time Efficiency

Approach 1 adds one loop when pushing the blur mutation to the platform views logged within the list, this process being $O(n)$ as it only loops one time, but as far as overall time efficiency is concerned, this approach should not greatly affect the time composing a frame. Approach 2, when no backdrop filter mutation is found within the layer tree or no platform views were found before a backdrop filter mutation, should have no effect on the time composing a frame as it will not call for a rewalk of the tree. When a backdrop filter is found within the layer tree, and a platform view was found before its call, then this approach will call for a rewalk of the layer tree. The walking of the layer tree will be $O(n)$ as it only walks through the tree one extra time, however a rewalking of the tree will likely affect the time composing a frame more than that of Approach 1.

Chosen Approach

Approach 1 is chosen for easy implementation and readability reasons.

Step 1: Track each visited platform view

- Within [FlutterPlatformViews_Internal.h](#), a vector of int values needs to be created to keep track of the platform view id's for each visited platform view. There also needs to be a push method created to add items to the vector. This method will be declared as virtual within `embedded_views.h`, overridden within `ios_external_view_embedder.h` and `ios_external_view_embedder.mm`, and defined within `FlutterPlatformViews_Internal.h`.

[embedded_views.h](#)

```
virtual void PushVisitedPlatformView(int64_t view_id) {}
```

[ios_external_view_embedder.h](#)

```
void PushVisitedPlatformView(int64_t view_id) override;
```

[ios_external_view_embedder.mm](#)

```
void IOSEExternalViewEmbedder::PushVisitedPlatformView(int64_t view_id) {
    platform_views_controller_->PushVisitedPlatformView(view_id);
}
```

[FlutterPlatformViews_Internal.h](#)

```
void PushVisitedPlatformView(int64_t view_id) {
    visited_platform_views_.push_back(view_id);
}
```

```
private:
    std::vector<int64_t> visited_platform_views_;
```

- You will populate the vector in [platform_view_layer.cc](#) at the end of the [Preroll](#) of a visited platform view.

```
context->view_embedder->PushVisitedPlatformView(view_id_);
```

Step 2: Creating backdrop filter mutator

- To create the backdrop filter mutator, we had to update Mutator and MutatorsStack classes in [embedded_views.h](#). For the Mutator class, a backdrop filter MutatorType and its respective constructors were created. In MutatorsStack class, we defined the push function for the backdrop filter mutator. The PushBackdropFilter method is then defined within [embedded_views.cc](#)

[Embedded_views.h](#) (Mutator Class)

```
enum MutatorType { kClipRect, ..., kBackdropFilter };
```

```
Mutator(const Mutator& other) {
```

```

type_ = other.type_;
switch (other.type_) {
...
    case kBackdropFilter:
        filter_ = other.filter_;
        break;
    default:
        break;
}
}

explicit Mutator(std::shared_ptr<const DLImageFilter> filter)
: type_(kBackdropFilter), filter_(filter) {}

const DLImageFilter& GetFilter() const { return *filter_; }

bool operator==(const Mutator& other) const {
    if (type_ != other.type_) {
        return false;
    }
    switch (type_) {
...
    case kBackdropFilter:
        return *filter_ == *other.filter_;
    }
    return false;
}

private:
...
std::shared_ptr<const DLImageFilter> filter;

```

(MutatorsStack Class)

```
void PushBackdropFilter(std::shared_ptr<const DLImageFilter> filter);
```

embedded_views.cc

```

void MutatorsStack::PushBackdropFilter(
    std::shared_ptr<const DLImageFilter> filter) {
    std::shared_ptr<Mutator> element = std::make_shared<Mutator>(filter);
    vector_.push_back(element);
};

```

- Once created, a case to handle the backdrop filter mutator needed to be added to the ApplyMutators method within FlutterPlatformViews.mm and the PushPlatformViewLayer within embedder_layers.cc.

FlutterPlatformViews.mm

```

auto iter = mutators_stack.Begin();
while (iter != mutators_stack.End()) {
    switch ((*iter)->GetType()) {
...
    case kBackdropFilter:

```

```

        break;
    }
    ++iter;
}

```

embedder_layers.cc.

```

for (auto i = mutators.Bottom(); i != mutators.Top(); ++i) {
    const auto& mutator = *i;
    switch (mutator->GetType()) {
        ...
        case MutatorType::kBackdropFilter:
            break;
    }
}

```

Step 3: Pushing changes to the platform views

- Back in [embedded_views.h](#), a new function to push the filter mutation to the stack needs to be declared.

```

virtual void PushFilterToVisitedPlatformViews(
    std::shared_ptr<const DlImageFilter> filter) {}

```

- Since this change is an ios specific change, this function will be overridden in the ios_external_view_embedder.h and ios_external_view_embedder.mm files.

ios_external_view_embedder.h

```

void PushFilterToVisitedPlatformViews(
    std::shared_ptr<const DlImageFilter> filter) override;

```

ios_external_view_embedder.mm

```

void IOSEExternalViewEmbedder::PushFilterToVisitedPlatformViews(
    std::shared_ptr<const DlImageFilter> filter) {
    platform_views_controller_->PushFilterToVisitedPlatformViews(filter);
}

```

- The function is then referenced again in FlutterPlatformViews_Internal.h and fully defined in FlutterPlatformViews.mm. The function loops through the list of visited platform views and pushes the filter mutation to each of their mutator stacks.

FlutterPlatformViews_Internal.h

```

void PushFilterToVisitedPlatformViews(std::shared_ptr<const DlImageFilter> filter);

```

FlutterPlatformViews.mm

```
void FlutterPlatformViewsController::PushFilterToVisitedPlatformViews(
    std::shared_ptr<const DlImageFilter> filter) {
    for (int64_t id : visited_platform_views_) {
        EmbeddedViewParams params = current_composition_params_[id];
        params.PushImageFilter(filter);
        current_composition_params_[id] = params;
    }
}
```

- The PushImageFilter method is defined within the EmbeddedViewParams class in [embedded_views.h](#) to handle converting the filter mutation to a raw pointer that can be handled by the PushBackdropFilter method for the MutatorsStack class.

```
void PushImageFilter(std::shared_ptr<const DlImageFilter> filter) {
    mutators_stack_.PushBackdropFilter(*filter);
}
```

- Finally, the function is called within the [backdrop_filter.cc](#) file.

```
if (context->view_embedder != nullptr) {
    context->view_embedder->PushFilterToVisitedPlatformViews(filter_);
```

Apply Filter Effect

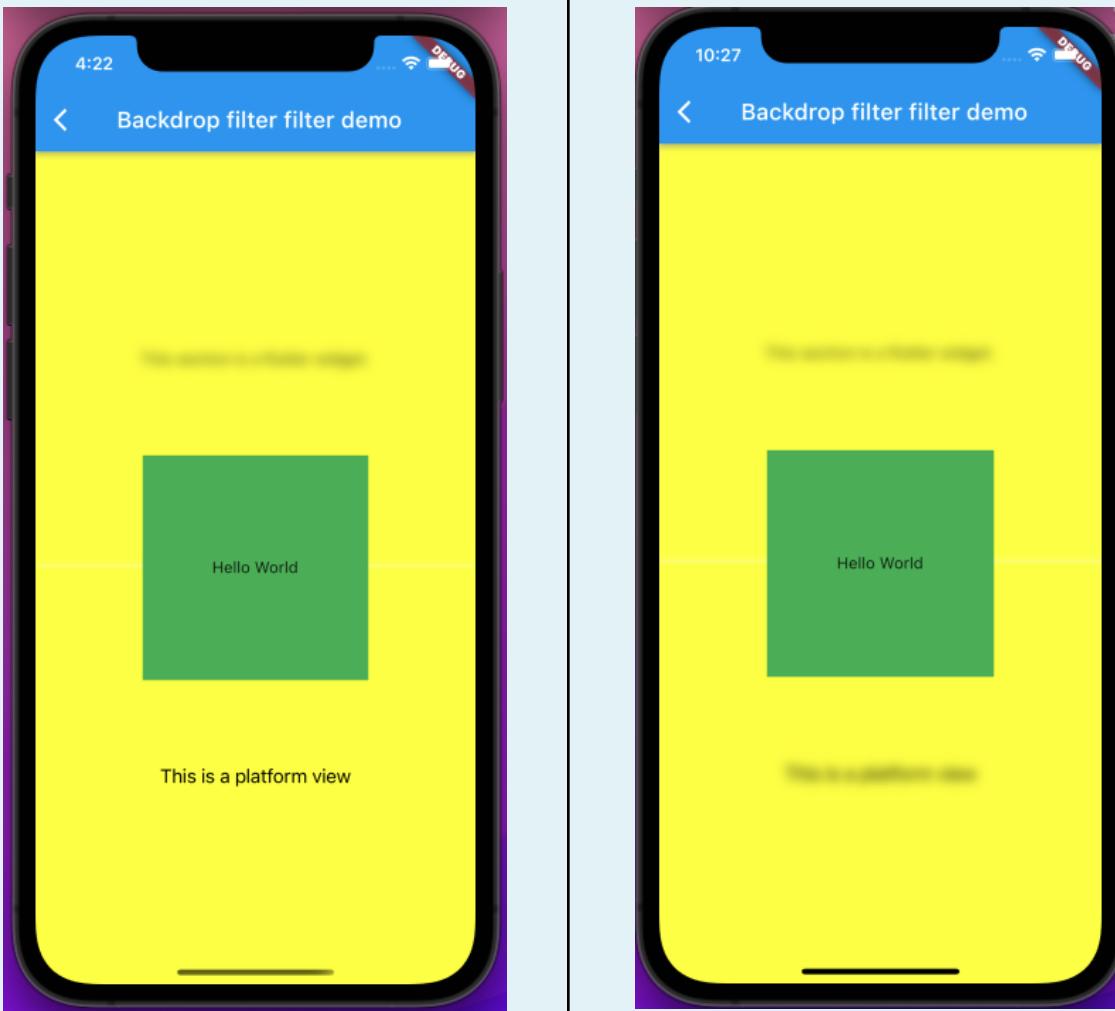
Context

The ApplyMutators method in the FlutterPlatformViewsController is called when the mutators_stack is updated. This method iterates over the mutators_stack and applies the active mutations for the current frame. When a kBackdropFilter pops from the mutators_stack, the PlatformView must blur.

FlutterViews already blur correctly when a Backdrop Filter is applied. Skia blurs these widgets with a Gaussian blur whose radius is set to the values inputted by the Flutter developer. PlatformViews must match this blur.

To test our approaches to blur the PlatformView, we created a [demo app](#) that follows the right layer tree outlined in the tree diagram at the beginning of this doc. The app applies a Backdrop Filter to a Flutter Text Widget which covers a FlutterView and a PlatformView. Since both the FlutterView and PlatformView are behind the Backdrop Filter, both views must be blurred. The table below illustrates this problem and its solution on our demo app.

Before Our Implementation	After Our Implementation
---------------------------	--------------------------



The Backdrop Filter is applied to the “Hello World” Flutter Text Widget. The FlutterView fills the top half of the screen behind the text widget and says, “This is a flutter view.” The PlatformView fills the bottom half of the screen behind the label and says, “This is a platform view.”

The next sections outline the three approaches we explored to blur a PlatformView:

1. [Initialize a UIVisualEffectView with a UIBlurEffect.](#)
 - a. [Extract the gaussianBlur filter and add it to the ChildClippingView.](#)
 - b. Control the blur strength with a [UIViewControllerAnimated](#).
2. Take a screenshot of the PlatformView and use the [CIGaussianBlur](#) filter to blur the screenshot.
3. Use the “gaussian” filter provided by [SwiftUI.blur](#).

Our final solution follows Approach 1a. The implementation extracts the gaussianBlur filter from a UIVisualEffectView that was initialized with UIBlurEffect, and adds that filter to the ChildClippingView. The next sections describe the different approaches and why we decided to implement Approach 1a.

Approach 1: Initialize a UIVisualEffectView with a UIBlurEffect

When a [UIVisualEffectView](#) is initialized with [UIBlurEffect](#), the resulting UIVisualEffectView shows a blurred view of its subviews. We need a blurred view of the PlatformView’s subviews, however solely applying the UIBlurEffect with a

UIVisualEffectView is not a valid solution because their APIs do not allow customizations. This means the default blur radius value cannot be changed and the “colorSaturate” filter also added by UIBlurEffect cannot be removed. To meet the project specifications, our solution must apply the blur radius inputted by the Flutter developer and the blur’s coloring should not change.

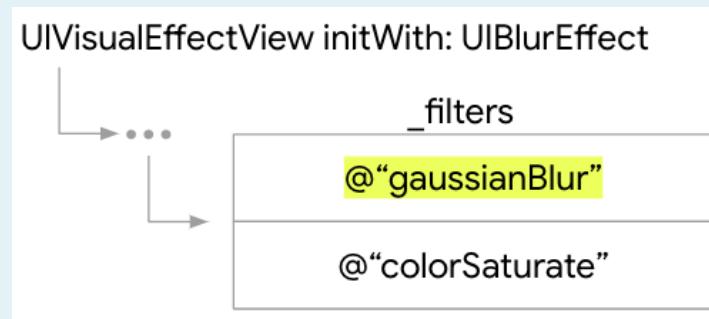
Approach 1a and Approach 1b are two approaches we explored to utilize UIBlurEffect’s Gaussian blur and customize it to meet our specifications. ~~Ultimately, we chose to implement Approach 1a because it removes the colorSaturate filter, is a simpler solution, and is more space efficient.~~

We chose 1c as the final solution because it has the configuration flexibility of approach 1a and provides support for partially blur PlatformViews.

Approach 1a. Extract the gaussianBlur filter and add it to the ChildClippingView.

PR: <https://github.com/flutter/engine/pull/34596>

The UIBlurEffect applies a CAFilter, “gaussianBlur,” to the filters of its UIVisualEffectView, as shown in the diagram below.



The main idea of this approach is to extract that gaussianBlur filter, copy the filter for each applied Backdrop Filter, update the radius values of the copies, and add them to the PlatformView’s ChildClippingView.layer.filters. The ChildClippingView is the parent view of the PlatformView. By adding the gaussianBlur filters to its list of filters, the PlatformView will display the blurs. The general steps of this approach are:

1. Initialize a UIVisualEffectView with UIBlurEffect and extract the gaussianBlur filter.
2. Update ChildClippingView.layer.filters to contain copies of the gaussianBlur filter with the blur radius values from blurRadii.

Context

In the ApplyMutators method, when a kBackdropFilter pops from the mutators_stack, its blur radius value is stored in an array, blurRadii. After all mutators have popped, ApplyMutators calls the applyBlurBackdropFilters method

on the PlatformView's ChildClippingView, and passes blurRadii as its parameter. applyBlurBackdropFilters is responsible for completing Steps 1-2 with the blur radius values from blurRadii.

Determining the blur radius

When Flutter users create Blur Backdrop Filters, they specify the blur strength by inputting sigma_x and sigma_y values. When sigma_x = sigma_y, a uniform circular blur is rendered by Skia with the [2D Gaussian blur function](#).

Quartz also uses this function to create Gaussian blurs with the gaussianBlur filter. To keep the blur circular, gaussianBlur uses a kernel that makes sigma_x = sigma_y. Thus, gaussianBlur only has one property to specify the blur radius, inputRadius, such that inputRadius = sigma_x = sigma_y.

Through testing, we verified that if inputRadius = sigma_x = sigma_y, the FlutterView and PlatformView will render the same blur. Our implementation arbitrarily chooses the sigma_x inputted by the Flutter user as the value for inputRadius.

1. Initialize a UIVisualEffectView with UIBlurEffect and extract the gaussianBlur filter.
Although the API for UIVisualEffectView and UIBlurEffect prevent editing the gaussianBlur filter, this filter can be extracted and stored as an NSObject. Using NSObject's setValueForKey method, its input radius can be adjusted.

ChildClippingView's private instance variable, "_gaussianFilter" stores the extracted gaussianBlur CAFilter. Deep copies of _gaussianFilter are added to the ChildClippingView when a new gaussianBlur is needed. By copying _gaussianFilter, our implementation avoids unnecessarily extracting multiple gaussianBlur filters from multiple UIVisualEffectViews.

When extracting the gaussianBlur filter, our implementation includes multiple checks to ensure the hierarchies of UIVisualEffectView and UIBlurEffect match the expected organization. If the APIs for these classes change any ordering or naming, the gaussianBlur will not be extracted and Flutter contributors must rewrite the checks. These checks prevent the app from crashing and instead do not apply a blur backdrop filter. If no backdrop blur filters can be applied, applyBackdropFilters returns NO to avoid future calls.

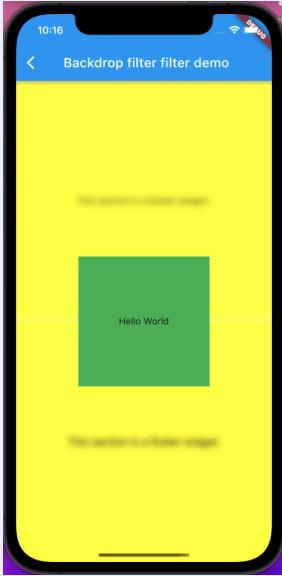
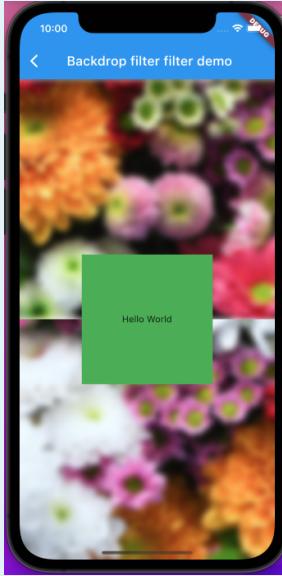
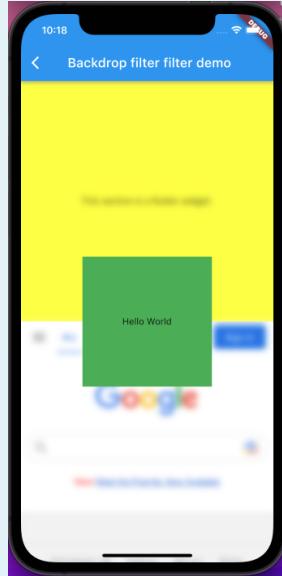
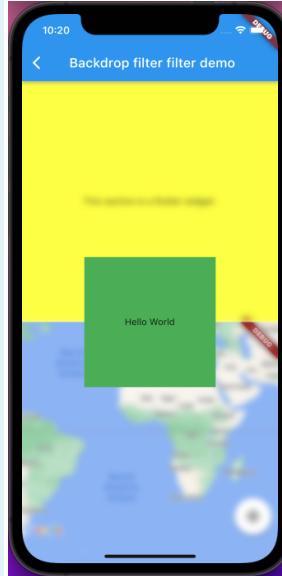
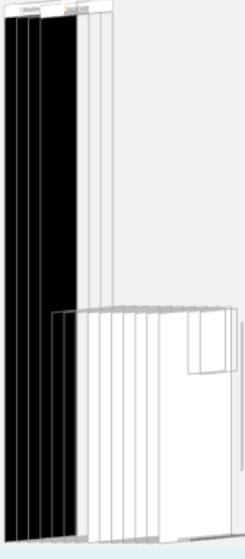
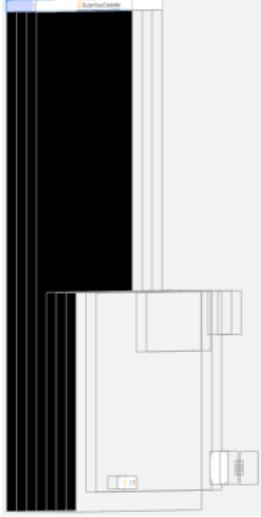
2. Update the array of active gaussianBlur filters to contain copies of the gaussianBlur filter with the Flutter developer's inputted radius values.

If the blur radius values in blurRadii match the current radius values of the ChildClippingView's gaussianBlurs, no changes are made. Otherwise, _gaussianFilter is copied for each Backdrop Filter and its inputRadius is set to the corresponding value from blurRadii. The copies are added to an NSMutableArray which is assigned to ChildClippingView.layer.filters. Since ChildClippingView.layer.filters is an

PUBLICLY SHARED

immutable array, any changes to the Backdrop Filters require reassigning ChildClippingView.layer.filters.

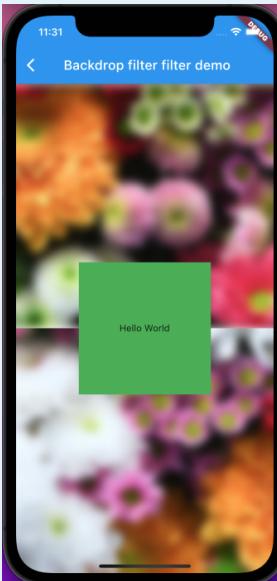
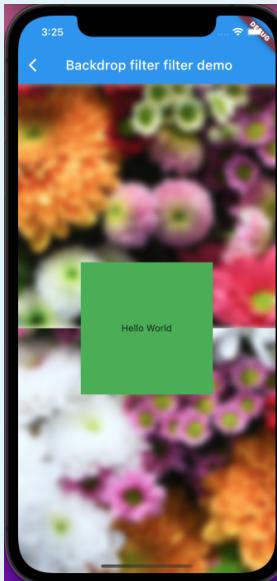
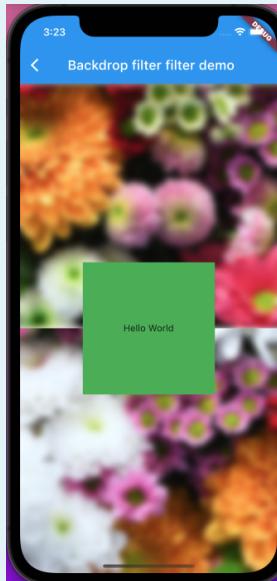
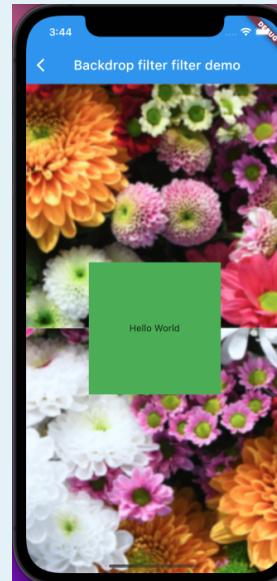
Results

One Backdrop Filter Applied to Different PlatformViews ($\sigma_x = 5$, $\sigma_y = 5$)			
UILabel with Text	UIImageView	WKWebView	Google Maps Plugin
			
			
The blur intensity for FlutterViews and PlatformViews matches. No additional views were added to the View Hierarchy.			

Multiple Backdrop Filters Applied to UIImageView

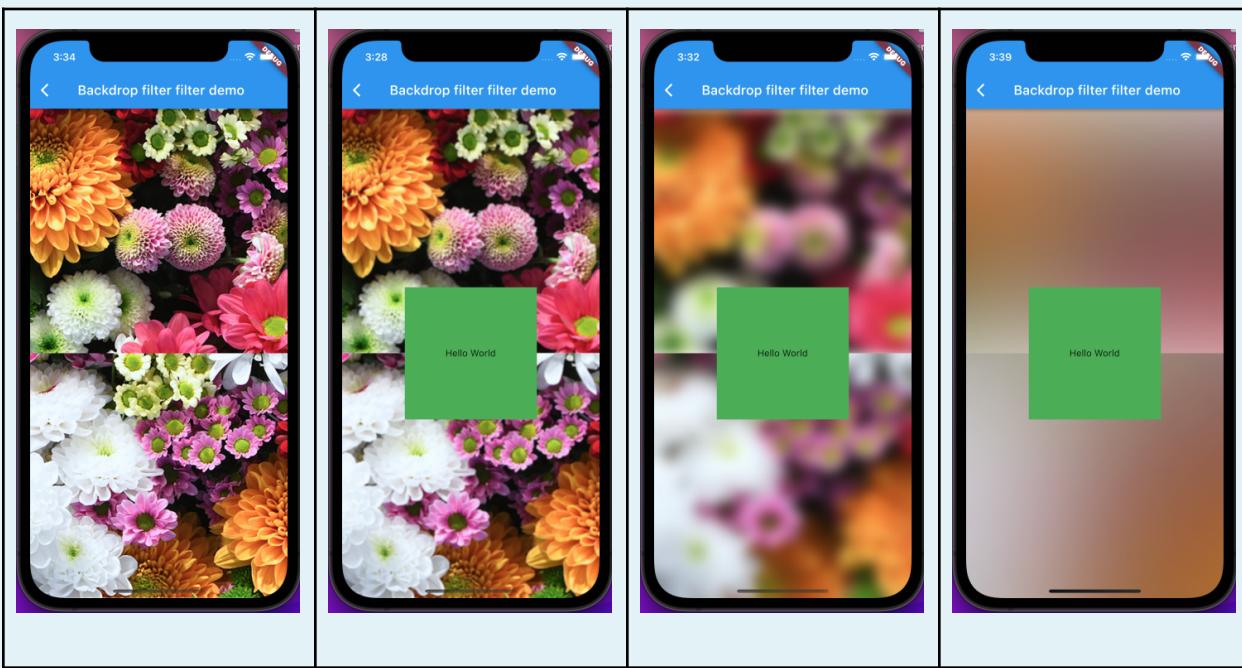
PUBLICLY SHARED

PUBLICLY SHARED

Two Backdrop Filters	Two Backdrop Filters	Two Backdrop Filters	Two Backdrop Filters
			
From outermost filter to innermost filter: All filters have sigma_x = 5 sigma_y = 5	From outermost filter to innermost filter: Filter 1: sigma_x = 5 sigma_y = 5 Filter 2: sigma_x = 1 sigma_y = 1	From outermost filter to innermost filter: Filter 1: sigma_x = 1 sigma_y = 1 Filter 2: sigma_x = 5 sigma_y = 5	From outermost filter to innermost filter: All filters have sigma_x = 1 sigma_y = 1
Layering backdrop filters with the same radius values and different radius values results in the same blur for FlutterViews and PlatformViews.			

One Backdrop Filter Applied to UIImageView with Different Radius Values			
Zero backdrop filters	sigma_x = 1 sigma_y = 1	sigma_x = 10 sigma_y = 10	sigma_x = 100 sigma_y = 100

PUBLICLY SHARED



Flutter users can input any positive number as a blur radius. The blurs match for FlutterViews and PlatformViews.

Analysis of Approach 1a

PROS:

- This approach successfully blurs PlatformViews composed of different UIViews.
- No additional UIViews are added to the View Hierarchy. The gaussianBlur filters are added to ChildClippingView.layer.filters.
- The blur strength is set to match the Flutter user's inputted values.
- The colorSaturate filter is not applied.
- This approach utilizes Apple's code for Gaussian blurs.

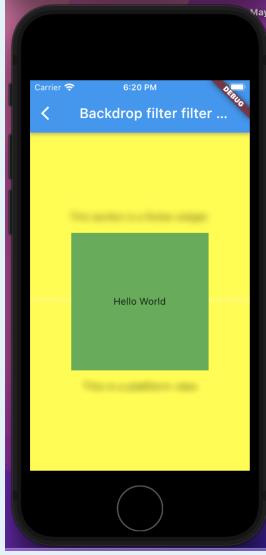
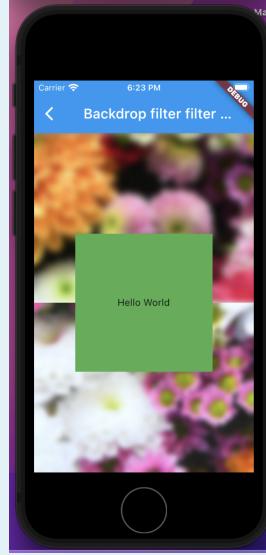
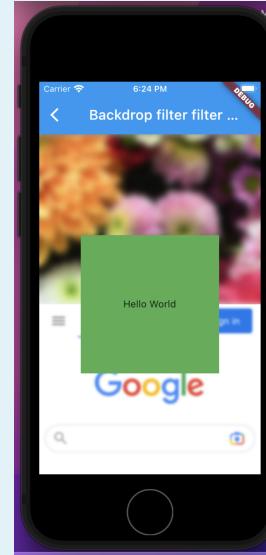
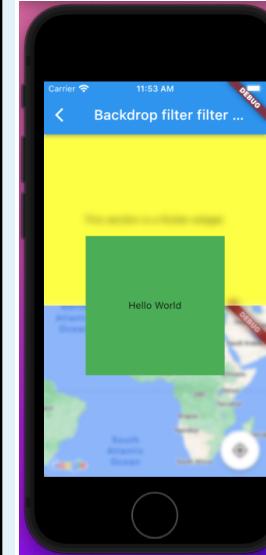
CONS:

- Can only blur the entire PlatformView; cannot blur a rect inside the PlatformView.
- This approach relies on Apple's API for UIVisualEffectView and UIBlurEffect.
 - Accessing the gaussianBlur CAFilter depends on how the API orders the subclasses of UIVisualEffectView and UIBlurEffect.
 - Updating the blur strength requires setting gaussianBlur's private property, inputRadius. We successfully deployed a demo app to the App Store, however, this practice is generally not supported by Apple.

Approach 1b. Customize the gaussianBlur filter with a UIViewPropertyAnimator

We were unsure if Apple would accept apps that directly access and update private properties, so we first prototyped Approach 1b. This approach uses a UIViewPropertyAnimator to limit the duration of applying the UIBlurEffect.

To apply the `UIViewPropertyAnimator` to the `UIVisualEffectView`, we created a class, `TimedVisualEffectView`, that derives from `UIVisualEffectView`. `TimedVisualEffectView` is initialized with `UIBlurEffect` and a `UIViewPropertyAnimator` that limits the duration of applying the `UIBlurEffect`. This duration can be related to the `sigma_x` inputted by the Flutter user such that the `TimedVisualEffectView` displays a blur that resembles the blur Skia renders for any Flutter widgets. The `TimedVisualEffectView` is added to the View Hierarchy to blur its superviews.

	<code>UILabel</code> with Text	<code>UIImageView</code>	<code>WKWebView</code>	<code>Google Maps</code> Plugin
With Blur (strength = 0.056)				

Analysis

PROS:

- This approach successfully blurs PlatformViews with different UIViews.

CONS:

- This approach adds an additional `UIVisualEffectView` to the View Hierarchy.
- We need to write a formula that relates the Flutter user's inputted `sigma_x`, `sigma_y` to the blur duration.
 - This formula depends on the default `inputRadius` value for `UIVisualEffectView`'s `gaussianBlur` CAFilter (currently 20).
- The `colorSaturate` filter is still applied and slightly alters the coloring of the PlatformView. The discoloration is especially noticeable when multiple backdrop filters are layered.
- This approach requires creating an additional class, `TimedVisualEffectView`. Creating this class does not greatly affect app performance, however it is more complicated than Approach 1a.

Approach 1c. Modify private properties in `UIVisualEffectView`

Based on approach 1a and 1b, we know that `UIVisualEffectView` cannot be used directly because 1. The `colorSaturate` filter adds unwanted visual artifacts; 2. There is no public API to update the `inputRadius` in the `gaussianBlur` filter; 3. The “`UIViewPropertyAnimator`” approach is limited because it depends on the `gaussianBlur` filter’s `inputRadius`.

This approach uses `UIVisualEffectView` to blur the PlatformView. The `UIVisualEffectView` is modified to support configurations provided by Flutter’s blur filter as much as possible:

- The frame of the `UIVisualEffectView` can be used to only blur part of the PlatformView.
- Modify private properties of the `UIVisualEffectView` so that it mimics Flutter’s blur filter. The below form shows the difference between the default `UIVisualEffectView` and the modified `UIVisualEffectView`.

<code>UIVisualEffectView</code>	Modified <code>UIVisualEffectView</code>
<code>ColorSaturateFilter</code>	No <code>ColorSaturateFilter</code>
<code>_VisualEffectSubview</code> is not totally transparent	<code>_VisualEffectSubview</code> has a clear background (total transparent)
<code>GaussianFilter</code> ’s <code>inputRadius</code> is 20	<code>GaussianFilter</code> ’s <code>inputRadius</code> is the same as <code>sigmaX</code> . The <code>sigmaX</code> is determined from the <code>ImageFilter.blur</code> object used to blur the PlatformView Widget

In summary, the modified `UIVisualEffectView` (compared to the default `UIVisualEffectView`) has these differences:

- Does not add additional colors, or color saturations to the view that it is blurring.
- The blur intensity is configurable, with no upper limit.

Analysis of Approach 1c

PROS:

- This approach successfully blurs PlatformViews composed of different `UIViews`.
- The blur strength is set to match the Flutter user’s inputted values.
- The `colorSaturate` filter is not applied.
- This approach utilizes Apple’s code for Gaussian blurs.
- Can blur a specific rect inside the PlatformView if asked to.

CONS:

- This approach adds an additional `UIVisualEffectView` to the View Hierarchy.
 - In most cases only one `UIVisualEffectView` is added to View Hierarchy,

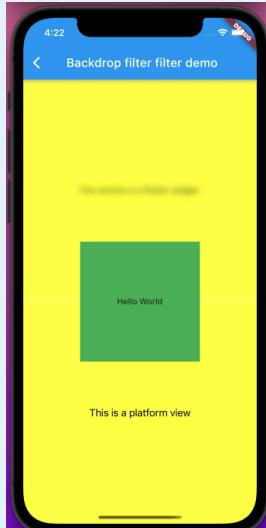
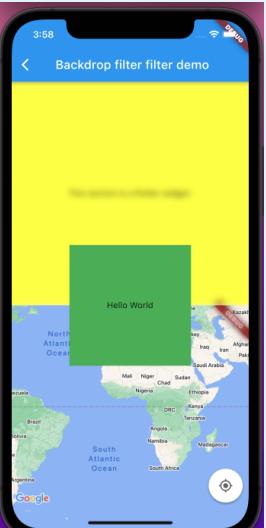
but the app can do crazy things where multiple blur effects need to be added to the PlatformView. The performance is compromised if there are many UIVisualEffectView added to the View Hierarchy; it is likely aligned with other approaches when many blur effects need to be added on top of each other.

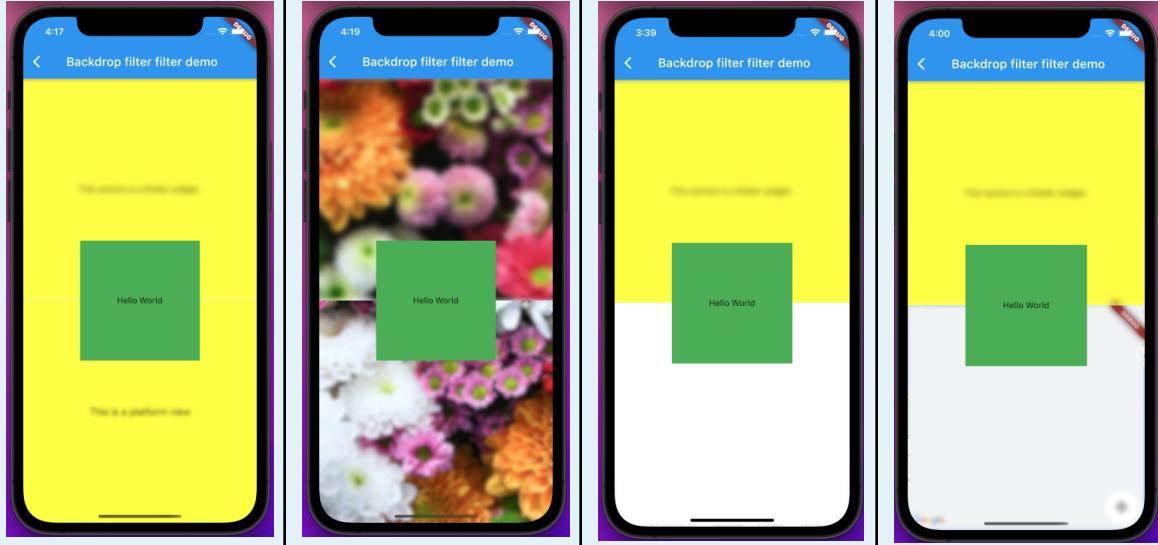
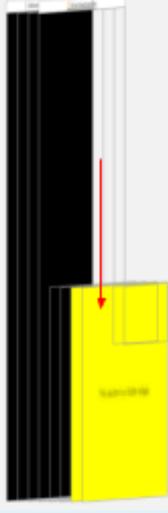
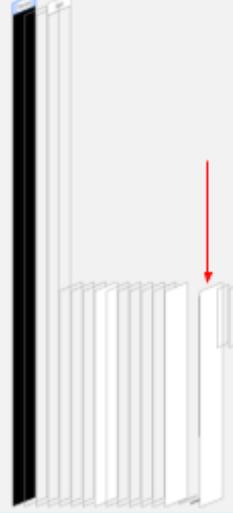
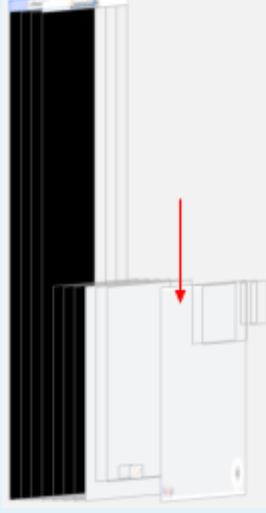
- This approach relies on Apple's API for UIVisualEffectView and UIBlurEffect. (Same as 1a)
 - Accessing the gaussianBlur CAFilter depends on how the API orders the subclasses of UIVisualEffectView and UIBlurEffect.
 - Updating the blur strength requires setting gaussianBlur's private property, inputRadius. We successfully deployed a demo app to the App Store, however, this practice is generally not supported by Apple.

Approach 2: Take a screenshot of the PlatformView and use the **CIGaussianBlur** filter to blur the screenshot

The CIGaussianBlur Filter produces a blurred copy of a CIIImage. For our prototype, we first captured a CIIImage screenshot of the PlatformView. Then we created a blurred copy of this screenshot with the CIGaussianBlur filter and added the copy as a subview to the PlatformView.

In the "View Hierarchy" row below, the red arrows point to the UIImageViews that hold the blurred screenshots. These UIImageViews and the pictures in the "With Blur" row show that not all PlatformViews successfully capture screenshots. For example, the WKWebView used by Flutter's WebView plugin uses CAEAGLLayer which requires special handling from WKWebView to support screenshots. Since WebView, GoogleMaps, and likely other plugins do not support screenshots by default, we decided to not use the CIGaussianBlur Filter in our implementation.

	UILabel with Text	UIImageView	WebView	Google Maps Plugin
Without Blur				

	PUBLICLY SHARED			
With Blur (sigma = 3)	4:17 Backdrop filter filter demo	4:19 Backdrop filter filter demo	3:39 Backdrop filter filter demo	4:00 Backdrop filter filter demo
				
View Hierarchy				

Approach 3: Use the “gaussian” filter provided by SwiftUI.blur (possible future update)

SwiftUI provides a public API ([.blur](#)) that allows us to set blur inputRadius. This approach creates an auxiliary SwiftUI view, and applies the blur on the SwiftUI view. Under the hood, SwiftUI creates a gaussian blur CAFilter. We can then access the CAFilter from the SwiftUI view and apply the filter to the PlatformView.

One drawback of this approach is that the blur CAFilter is only created if the SwiftUI View is displayed on screen, and the SwiftUI View cannot be blank or completely transparent. A workaround for this would be to add the SwiftUI View as a child view of the PlatformView and have the frame to be off-screen. (For example, set the frame to be (-100, 100, 1, 1). We have created a [sample iOS App](#) to demonstrate how to use SwiftUI’s .blur method to blur UIView.

This approach requires adding Swift/SwiftUI as a dependency of Flutter engine. Swift reached ABI stability at version 5.0, which requires a minimum iOS version of

12.2. As of July 2022, the minimum iOS version that Flutter supports is iOS 11. This means that if we add Swift as a dependency of the Flutter engine, the Swift Standard Library will be embedded in the App bundle, which noticeably increases the App size. It is not worth it to increase the App size for this feature where an Objective-C solution is available. Note that this size increase does not apply to 1) iPhone that is running iOS 12.2 and above, 2) Apps that contain any Swift code, including the Application code and plugins that are written in Swift. Also, SwiftUI requires iOS 13+, so this solution will not work for devices with a version below iOS 13.

Possible Future Update: Dividing Line

Approach 1a successfully blurs the PlatformView when a Blur Backdrop Filter is applied. Although both Flutter widgets and PlatformView widgets are successfully blurred, the widgets have a visible dividing line between them. There may also be a line between multiple PlatformViews. We explored a few ideas to remove this dividing line. The next sections describe the ideas we suggest prototyping first.

Use `CI GaussianBlur` filter to blur Picture Layer and PlatformView Layer

We prototyped capturing a screenshot of the FlutterViews and PlatformViews after everything was rendered and applying `CI GaussianBlur` to this screenshot. With the nature of different rendering engines and runtimes, we could not successfully capture this screenshot. Furthermore, similar to Approach 2, some PlatformViews could take longer to display the final view with an acceptable, visible lag to the viewer (i.e. waiting for a web browser to open, or the Maps screen to load). These screenshots would be harder to capture and maintain a consistent app user experience.

Manually blur division lines by applying Gaussian formula

The Gaussian blur is applied using a [convolution matrix](#). We had the idea of manually applying the convolution matrix on the pixels at the borders between PlatformViews and Flutter views. These pixels would be updated by the data from both widgets to carry the blur across their division.

ACCESSIBILITY

N/A

INTERNATIONALIZATION

N/A

OPEN QUESTIONS

- Will it work?

TESTING PLAN

- A set of c++ unit tests to test if BackdropFilter information is correctly propagated to PlatformViews.
- A set of XCTests can be used to test if the blur effects are applied to the PlatformViews.
- A [scenario app test](#) can be used to prevent regression.
- A benchmark with a PlatformView being blurred.

DOCUMENTATION PLAN

- On BackdropFilter widget, we need to document that:
 - On iOS, if a PlatformView is blurred, sigmaY is always equal to sigmaX regardless of the value configured by the user.
 - On iOS, if a PlatformView is blurred and its adjacent widget is also blurred, the edge between them is not blurred.
 - Add a link to this document at the end of the above explanations.

MIGRATION PLAN

N/A