

# Improving app\_server for WebKit

Julian Harnath  
<julian.harnath@rwth-aachen.de>

Introduction

Transparency Layers

Transforms and Clipping

Little Things Add Up

Outlook

## Introduction

Transparency Layers

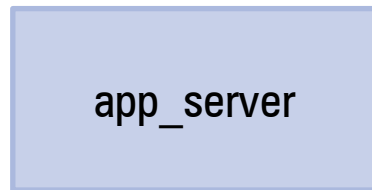
Transforms and Clipping

Little Things Add Up

Outlook

# Applications on Haiku

Server

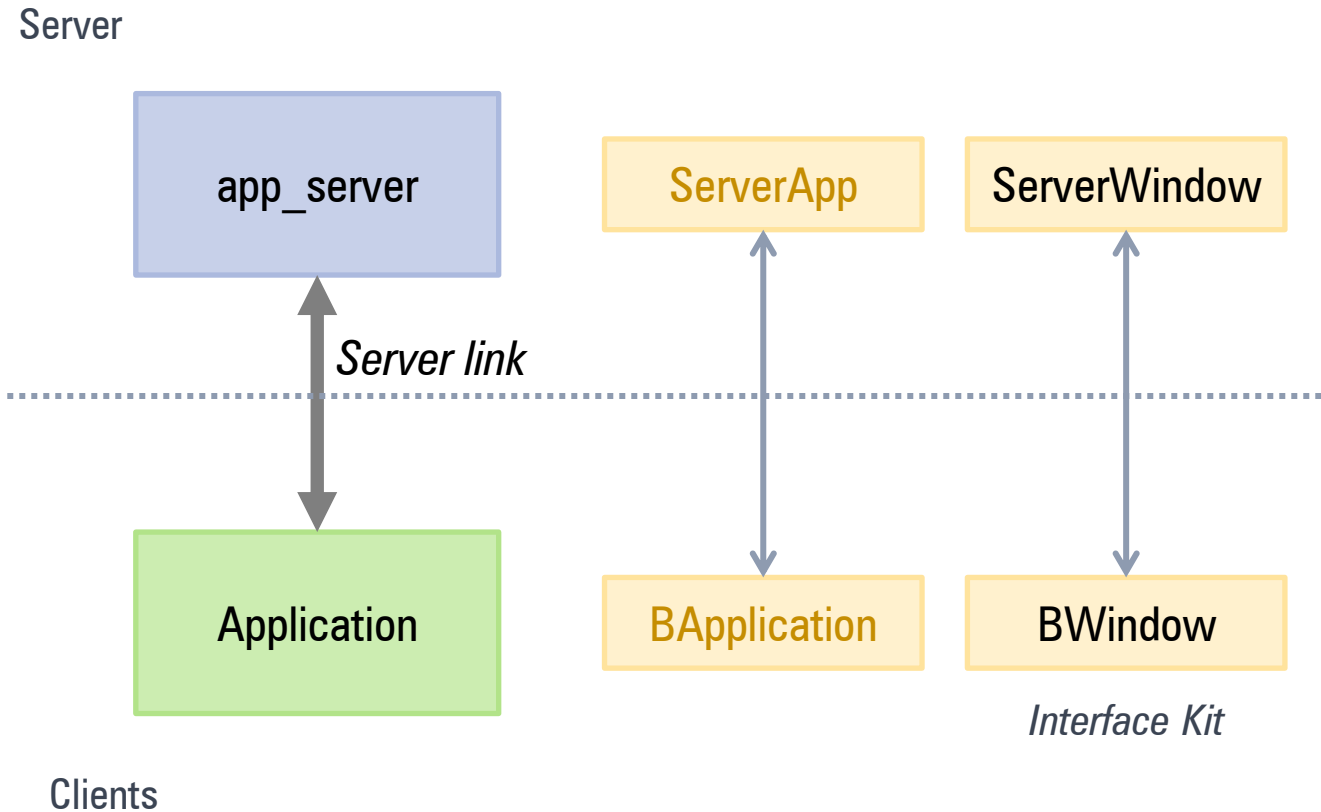


*Server link*



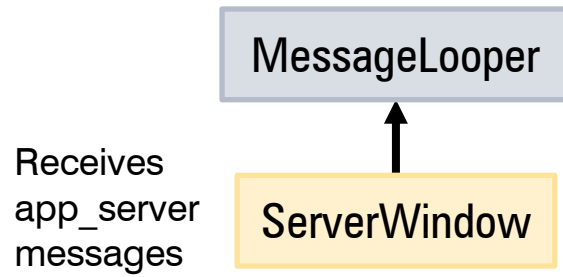
Clients

# Applications on Haiku



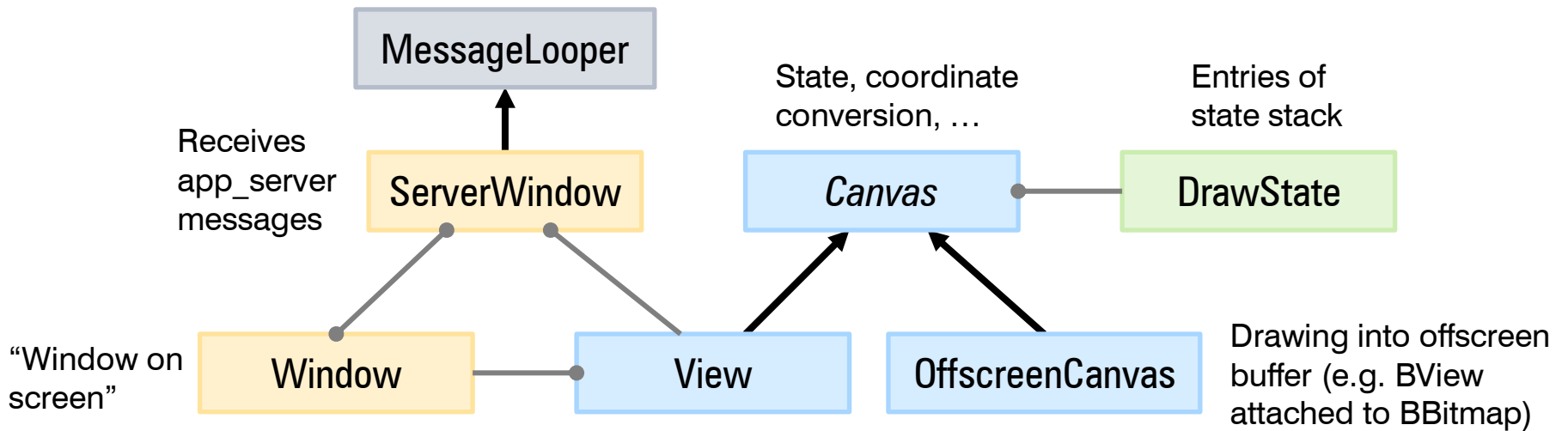
# Getting to Painter (1)

A very incomplete overview...



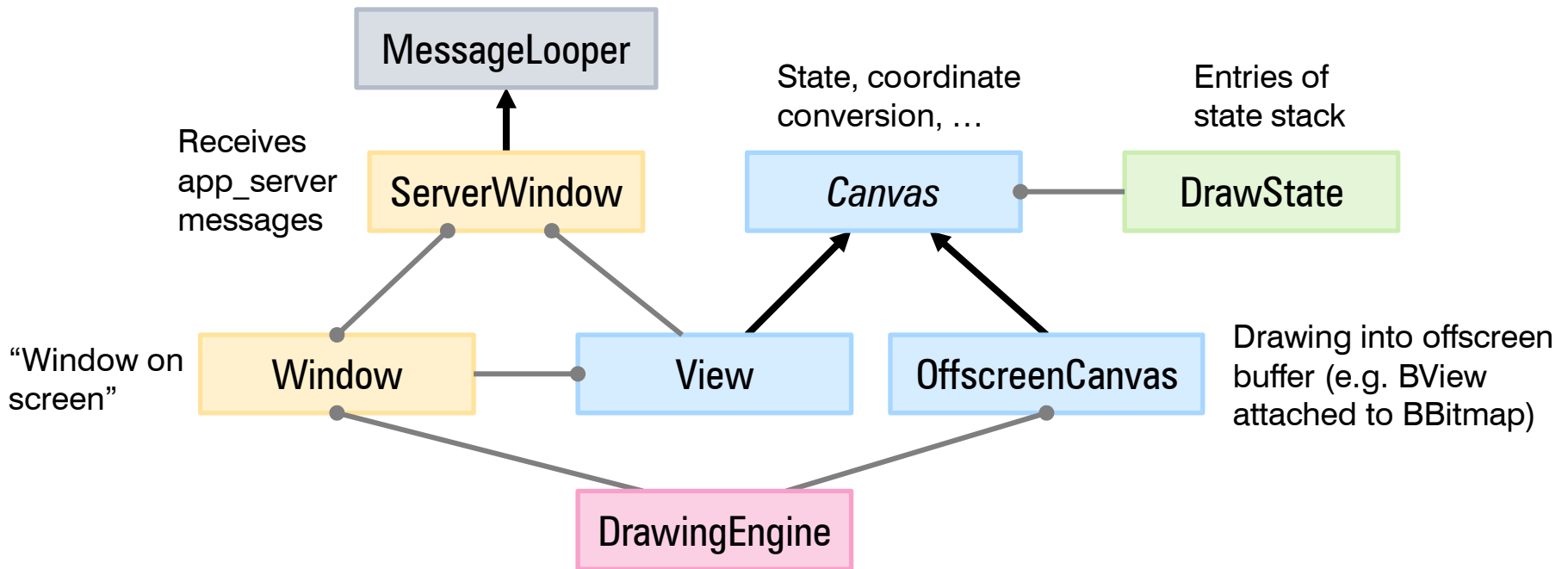
# Getting to Painter (1)

A very incomplete overview...



# Getting to Painter (1)

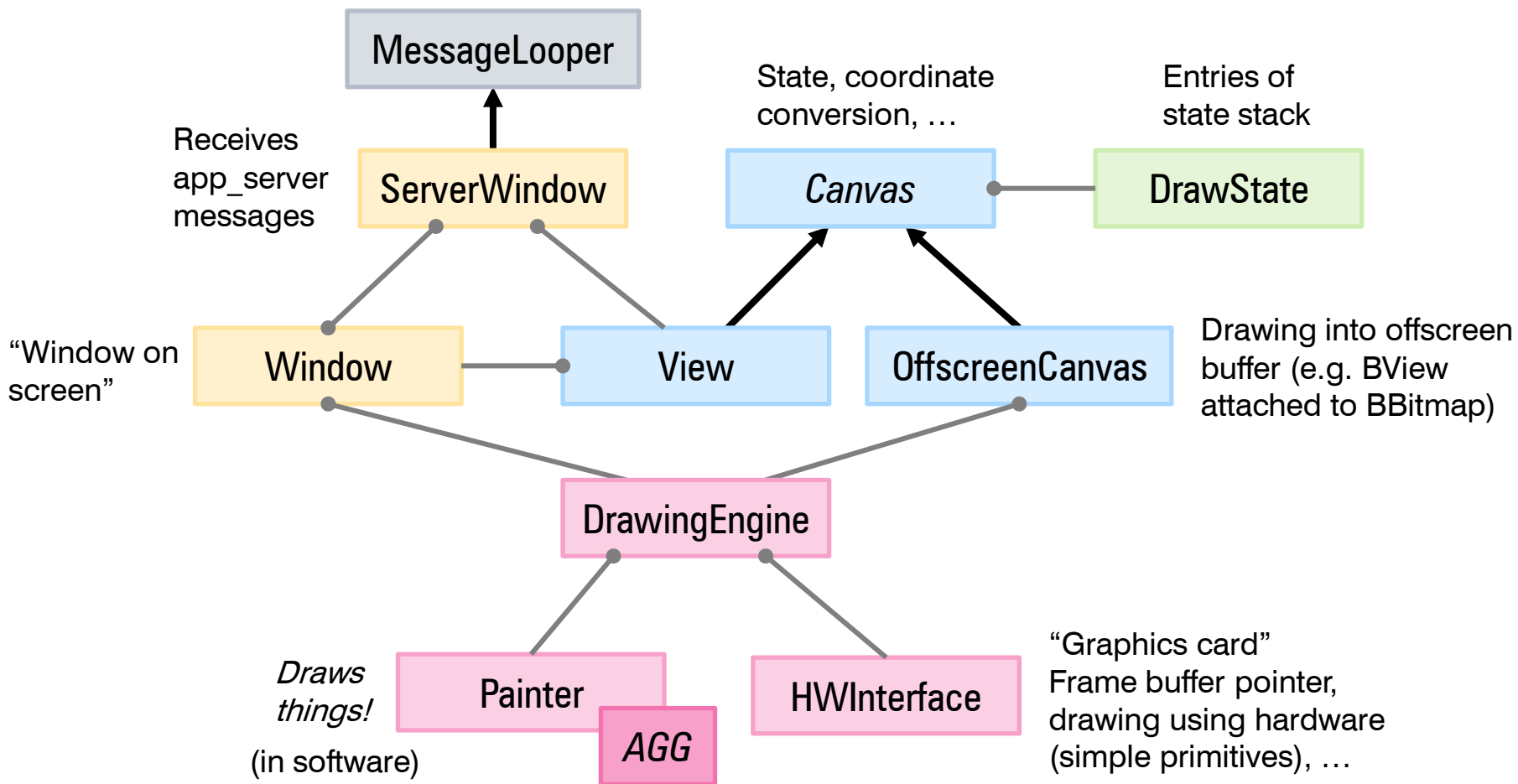
A very incomplete overview...





# Getting to Painter (1)

A very incomplete overview...



# Getting to Painter (2)

Drawing a rectangle...

BView::FillRect()

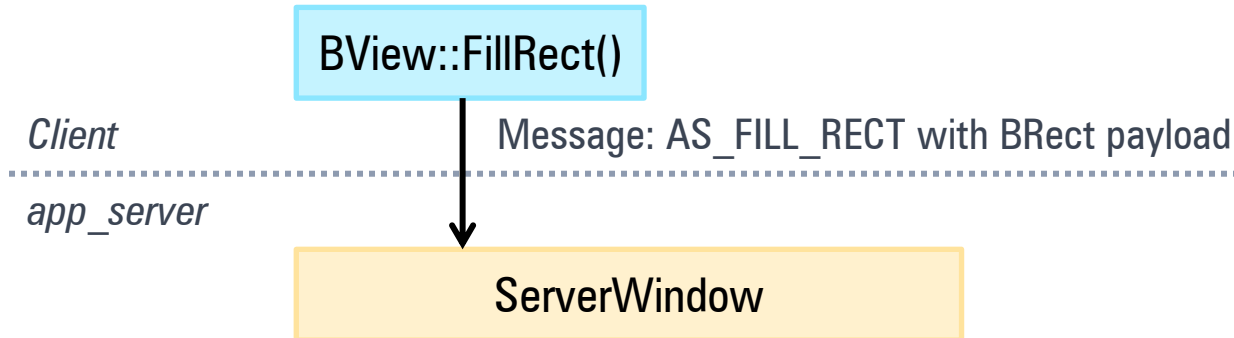
*Client*

*app\_server*

---

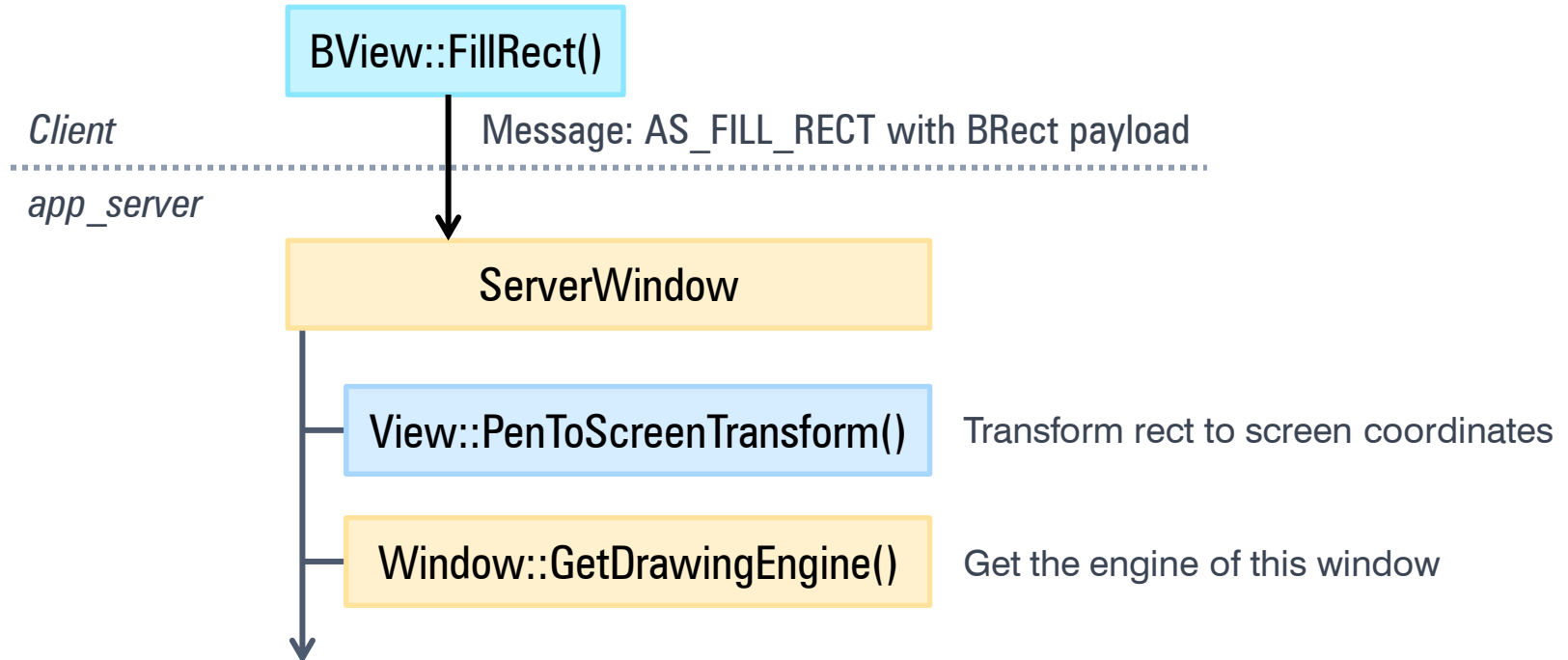
# Getting to Painter (2)

Drawing a rectangle...



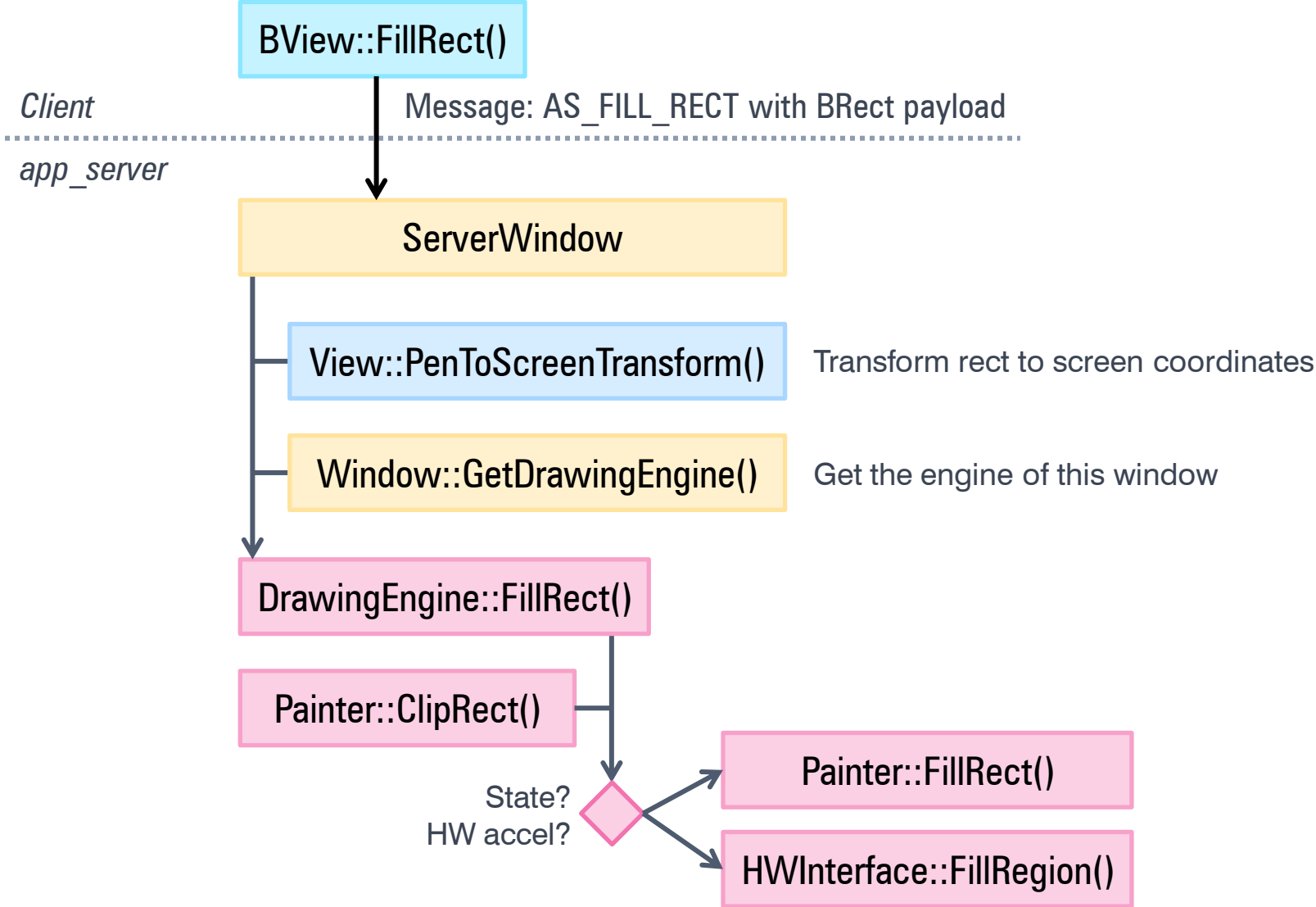
# Getting to Painter (2)

Drawing a rectangle...



# Getting to Painter (2)

Drawing a rectangle...



Introduction

**Transparency Layers**

Transforms and Clipping

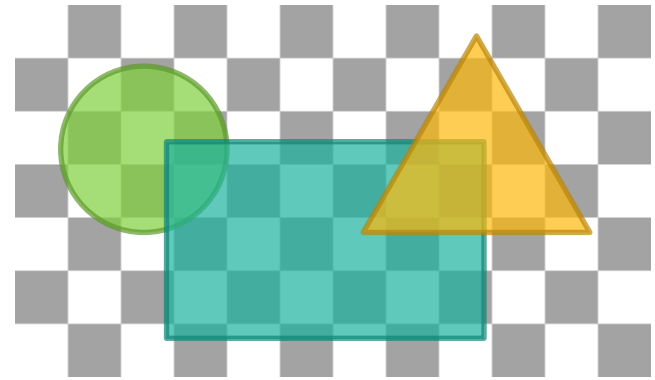
Little Things Add Up

Outlook

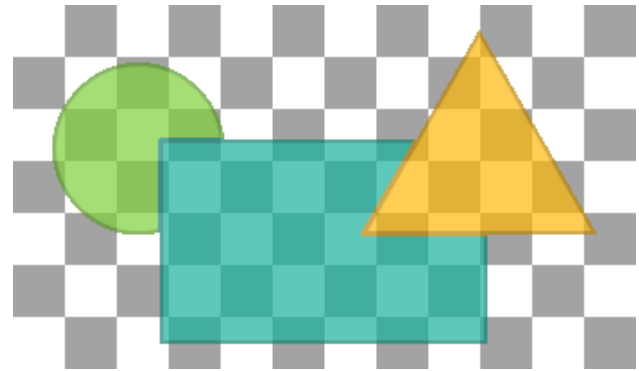
# Transparency Layers



$\alpha = 0.7$



Global alpha



Transparency layer

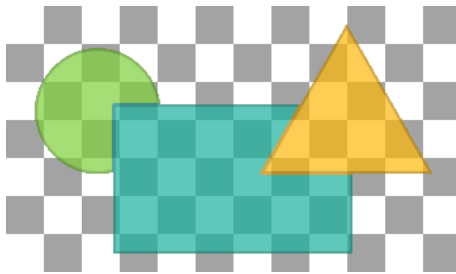
# The Workaround



(1) Create a new empty BBitmap and attach BView



(2) Draw into bitmap



(3) Draw bitmap onto background with added transparency (via ClipToPicture), throw away BBitmap



# The Problem



(1) Create a new empty BBitmap and attach BView

**On some websites, WebKit likes to use many layers, especially when doing many renders during scrolling.**

**Bad:** we don't know the size of the drawing yet, so we have to create the BBitmap at view size.

In WebPositive, this is almost the whole browser window size!

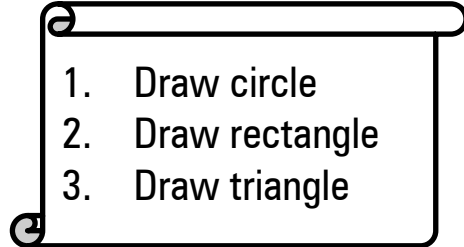
**Worse:** attaching a BView to a BBitmap spawns an offscreen window thread inside app\_server.

# A Better Solution

Let app\_server know what we're doing!

# A Better Solution

Let `app_server` know what we're doing!

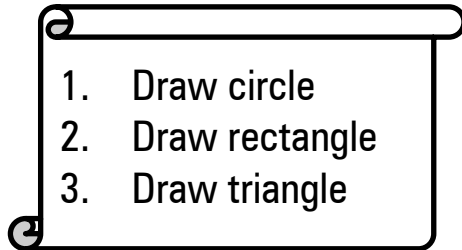


(1) Client: start layer; then draw things.

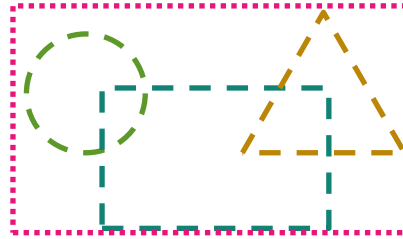
`app_server` does not draw, and instead just writes down the list of operations.

# A Better Solution

Let app\_server know what we're doing!

- 
1. Draw circle
  2. Draw rectangle
  3. Draw triangle

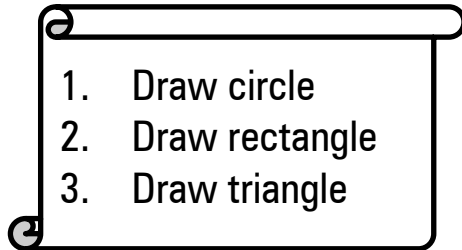
(1) Client: start layer; then draw things.  
app\_server does not draw, and instead just writes down the list of operations.



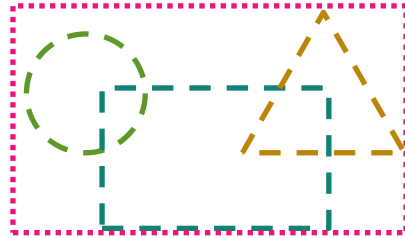
(2) Client: end layer.  
app\_server looks at the operations written down and figures out the (approx.) bounding box of this drawing, without actually drawing it

# A Better Solution

Let `app_server` know what we're doing!



(1) Client: start layer; then draw things.  
`app_server` does not draw, and instead just writes down the list of operations.



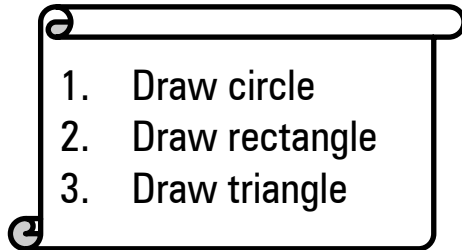
(2) Client: end layer.  
`app_server` looks at the operations written down and figures out the (approx.) bounding box of this drawing, without actually drawing it



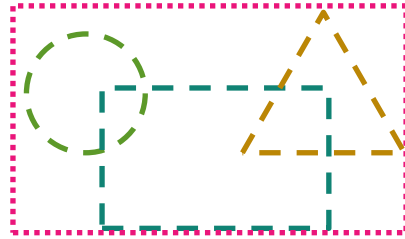
(3) Create `UtilityBitmap` of bounding box size and clear it. `UtilityBitmap` is `app_server`-internal and spawns no new thread!

# A Better Solution

Let `app_server` know what we're doing!



(1) Client: start layer; then draw things.  
`app_server` does not draw, and instead just writes down the list of operations.



(2) Client: end layer.  
`app_server` looks at the operations written down and figures out the (approx.) bounding box of this drawing, without actually drawing it



(3) Create `UtilityBitmap` of bounding box size and clear it. `UtilityBitmap` is `app_server`-internal and spawns no new thread!



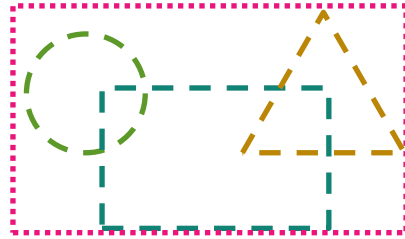
(4) Draw into `UtilityBitmap` from written down operations

# A Better Solution

Let app\_server know what we're doing!

1. Draw circle
2. Draw rectangle
3. Draw triangle

(1) Client: start layer; then draw things.  
app\_server does not draw, and instead just writes down the list of operations.



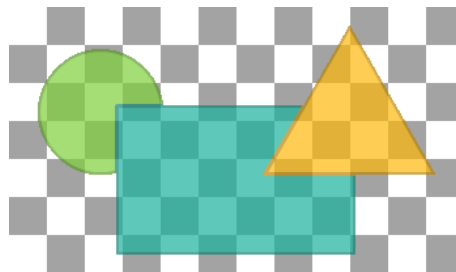
(2) Client: end layer.  
app\_server looks at the operations written down and figures out the (approx.) bounding box of this drawing, without actually drawing it



(3) Create UtilityBitmap of bounding box size and clear it. UtilityBitmap is app\_server-internal and spawns no new thread!



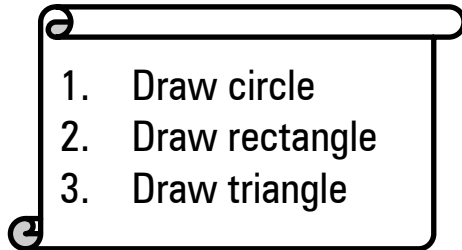
(4) Draw into UtilityBitmap from written down operations



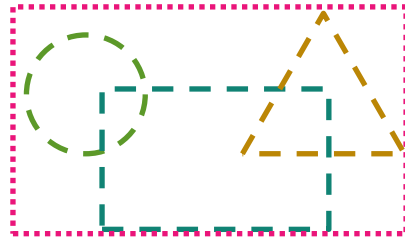
(5) Draw bitmap with transparency (via AlphaMask) and discard it

# A Better Solution

Let app\_server know what we're doing!

- 
1. Draw circle
  2. Draw rectangle
  3. Draw triangle

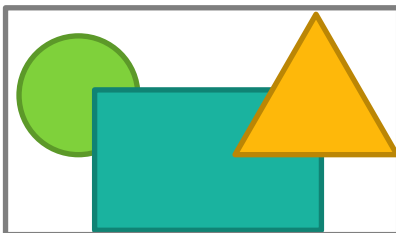
(1) Client: start layer; then draw things.  
app\_server does not draw, and instead just writes down the list of operations.



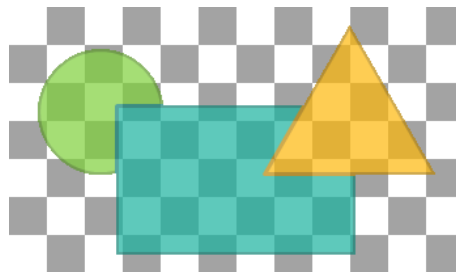
(2) Client: end layer.  
app\_server looks at the operations written down and figures out the (approx.) bounding box of this drawing, without actually drawing it



(3) Create UtilityBitmap of bounding box size and clear it. UtilityBitmap is app\_server-internal and spawns no new thread!



(4) Draw into UtilityBitmap from written down operations

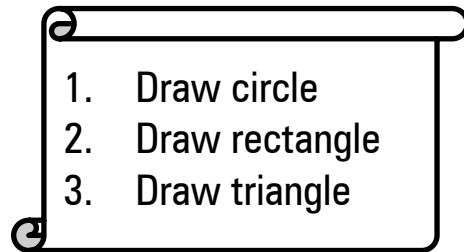


(5) Draw bitmap with transparency (via AlphaMask) and discard it

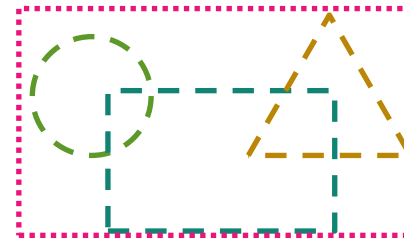
Problem solved!



# BPicture Saves the Day



ServerPicture

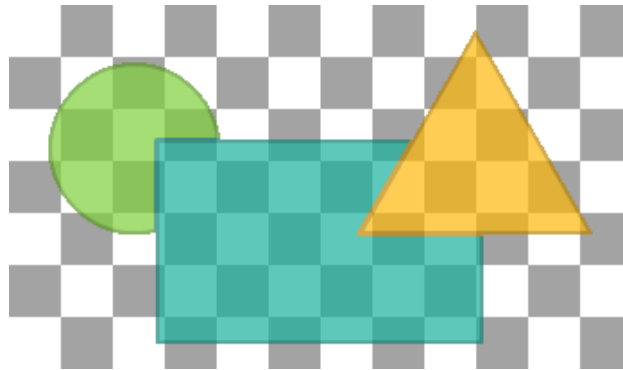


PictureBoundingBoxPlayer

Details to observe: drawing offset, transforms, clipping, draw state, drawing mode, ...

# Layer API

```
void BView::BeginLayer(uint8 opacity);  
void BView::EndLayer();
```



Introduction

Transparency Layers

**Transforms and Clipping**

Little Things Add Up

Outlook

# Transforms: R5

```
void BView::SetOrigin(float x, float y);  
void BView::SetScale(float ratio);
```

Translation, Scaling

# BAffineTransform (Haiku)

```
void          BView::SetTransform(BAffineTransform transform);  
BAffineTransform BView::Transform() const;
```

## BAffineTransform

Translation, Scaling/Mirroring, Rotation, Shearing

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & sh_x & t_x \\ sh_y & s_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

```
BAffineTransform:: ...  
    AffineTranslation()  
    AffineRotation()  
    AffineScaling()  
    AffineShearing()
```

# BAffineTransform (Haiku)

```
void          BView::SetTransform(BAffineTransform transform);  
BAffineTransform BView::Transform() const;
```

## BAffineTransform

Translation, Scaling/Mirroring, Rotation, Shearing

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & sh_x & t_x \\ sh_y & s_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

BAffineTransform:: ...  
AffineTranslation()  
AffineRotation()  
AffineScaling()  
AffineShearing()

Composition

$$\vec{p}' = B(A\vec{p}) = (BA)\vec{p}$$

BAffineTransform:: ...  
Multiply(), PreMultiply()  
TranslateBy(), ScaleBy(), RotateBy(), ShearBy()  
...

# Clipping

## R5 API: clipping region

```
void BView::ConstrainClippingRegion(BRegion* region);  
    fast
```

## R5 API: clipping mask

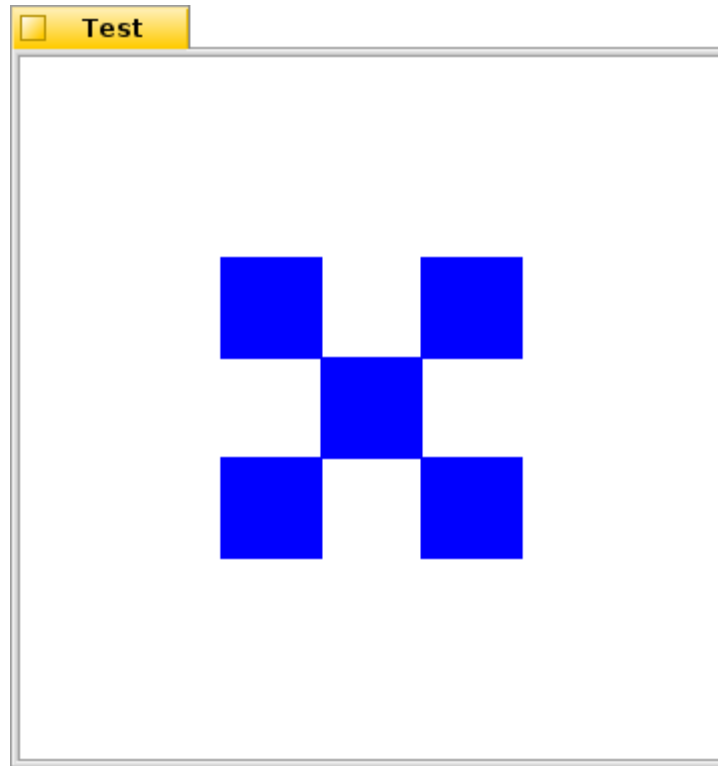
```
void BView::ClipToPicture(BPicture* picture, [...]);  
void BView::ClipToInversePicture(BPicture* picture, [...]);
```

R5: 1 bit alpha – only fully opaque or fully transparent

Haiku: 8 bit alpha – allows full pixel alpha masking

AlphaMask

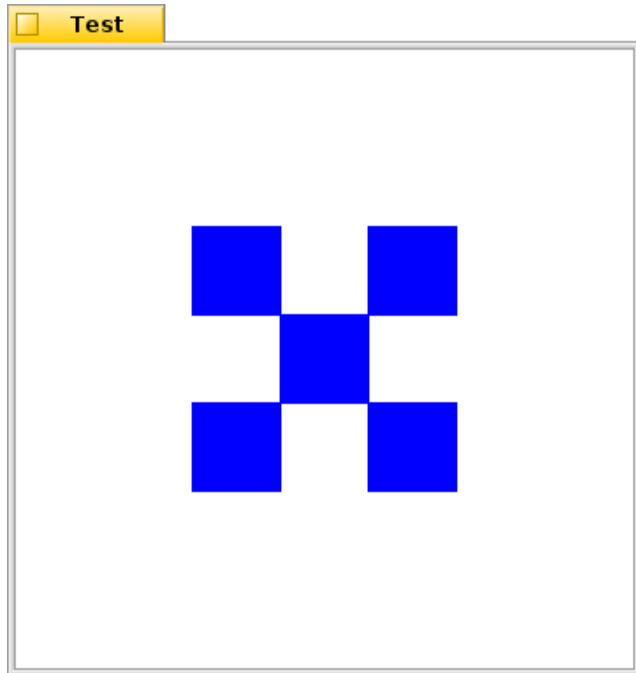
# Clipping Examples



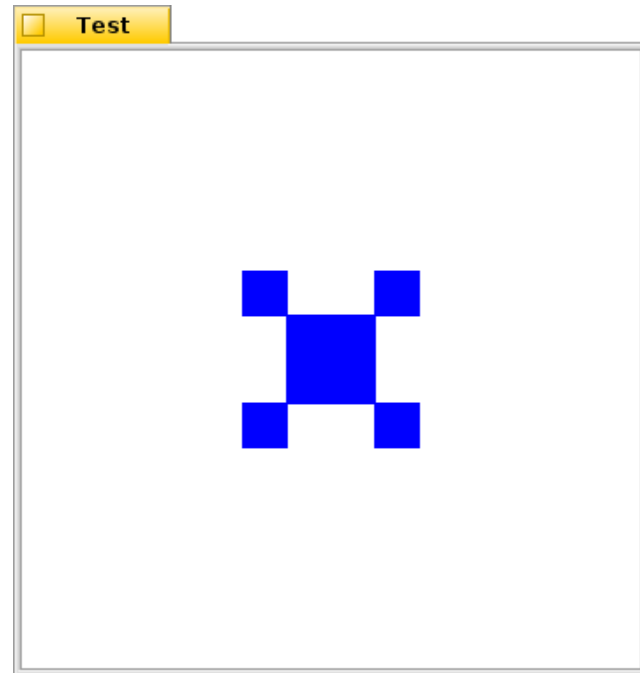
No clipping



# Clipping Region

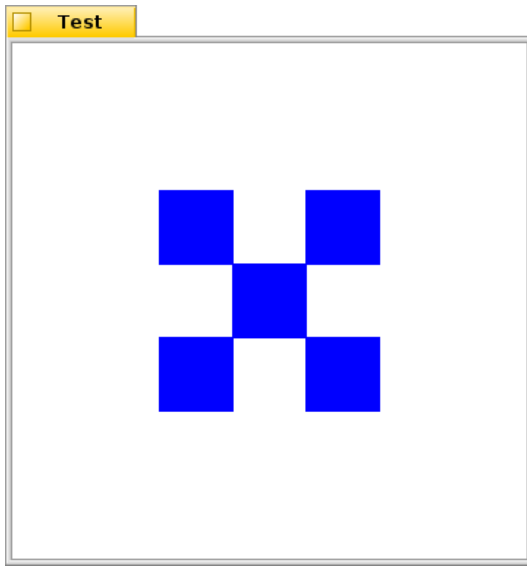


Original

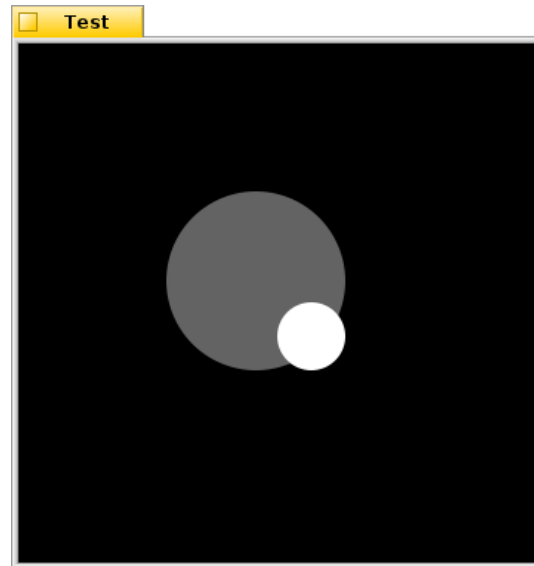


`ConstrainClippingRegion()`

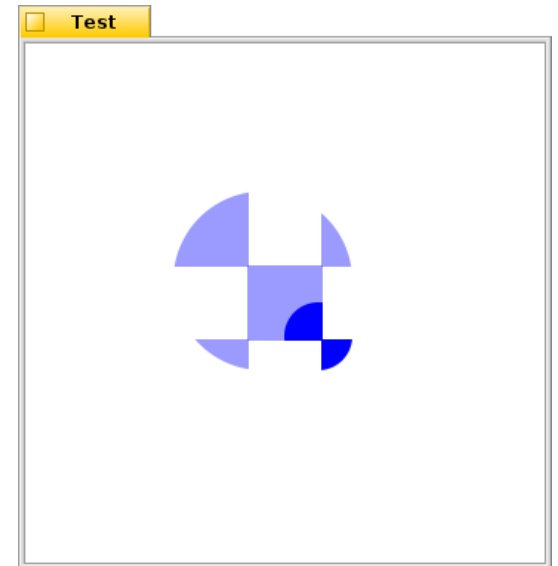
# Clipping Mask



Original



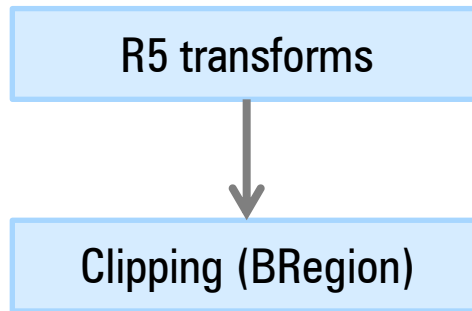
BPicture (Alpha Mask)



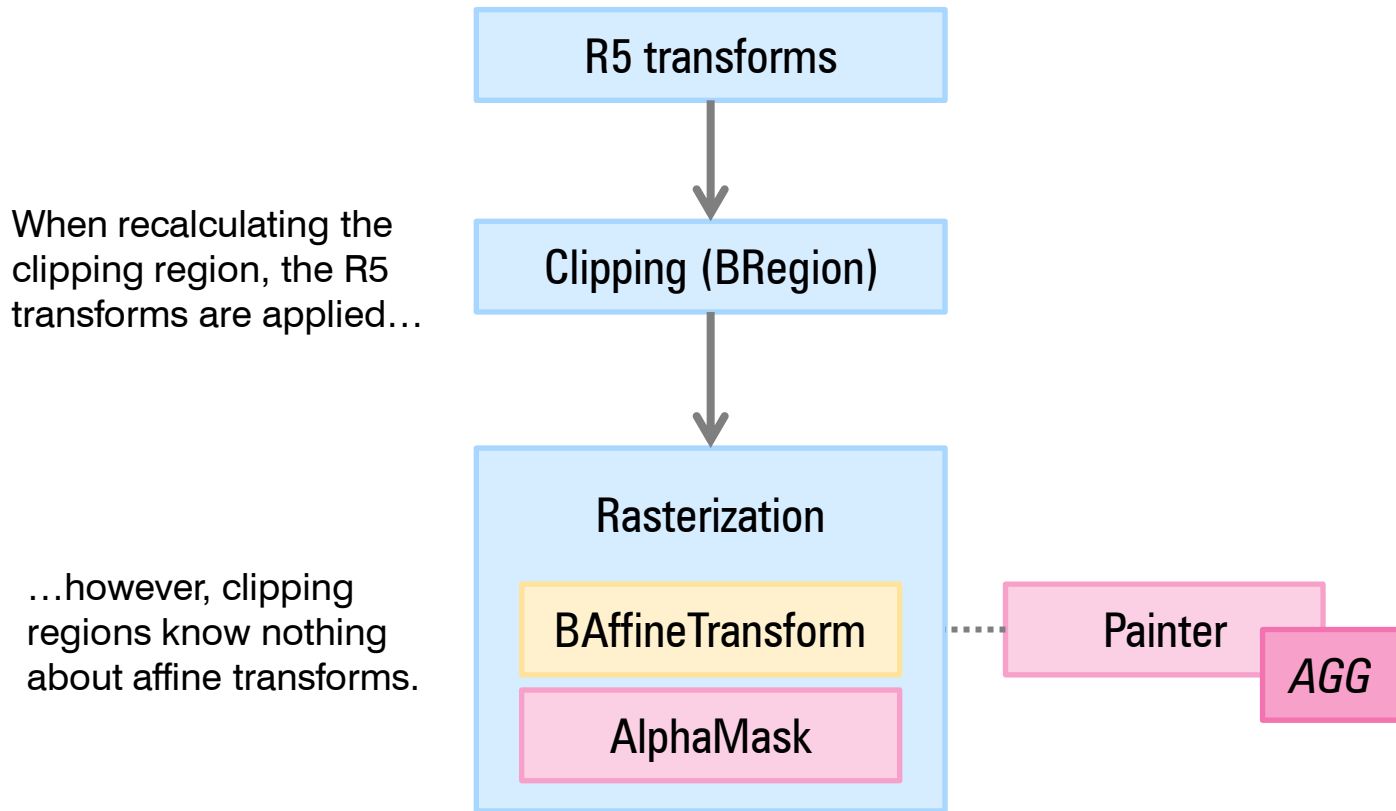
ClipToPicture()

# Transforms and Clipping

When recalculating the clipping region, the R5 transforms are applied...

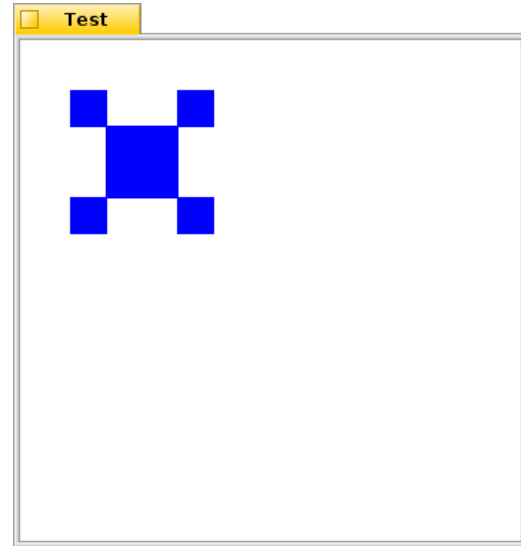
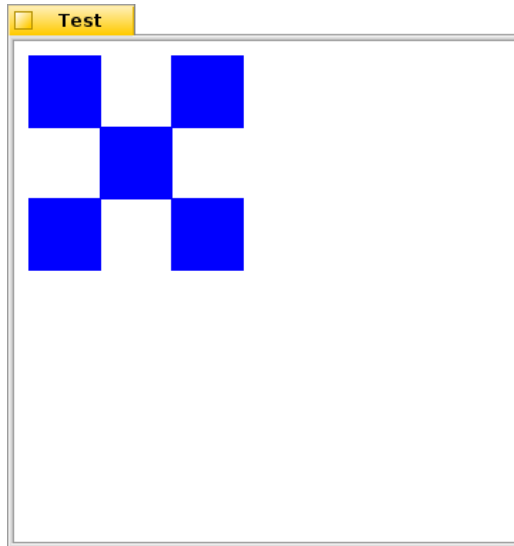


# Transforms and Clipping

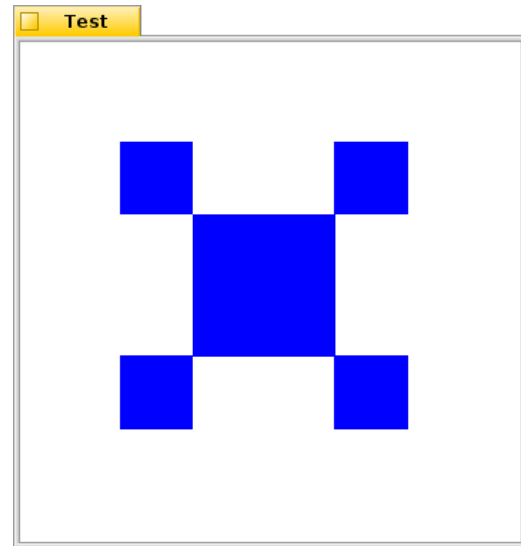
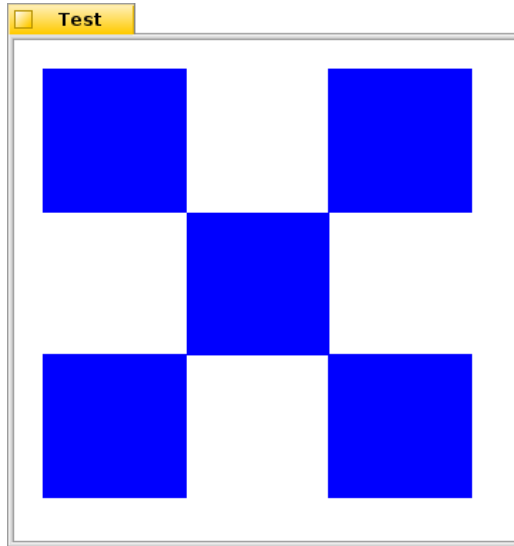


# R5 transforms...

ConstrainClippingRegion()

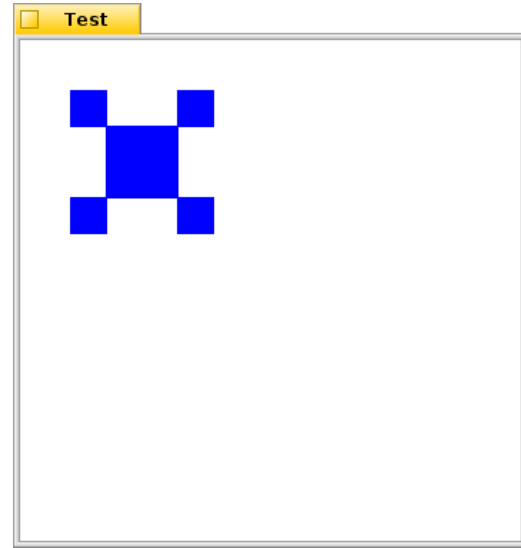
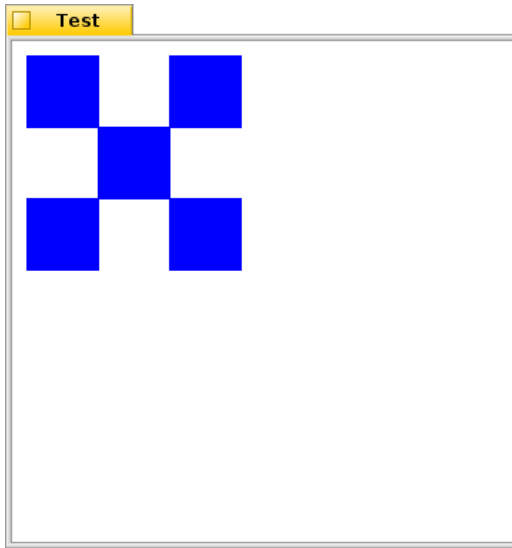


SetScale()

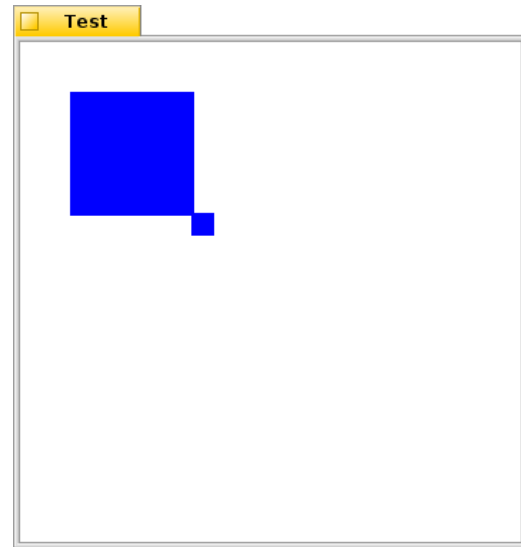
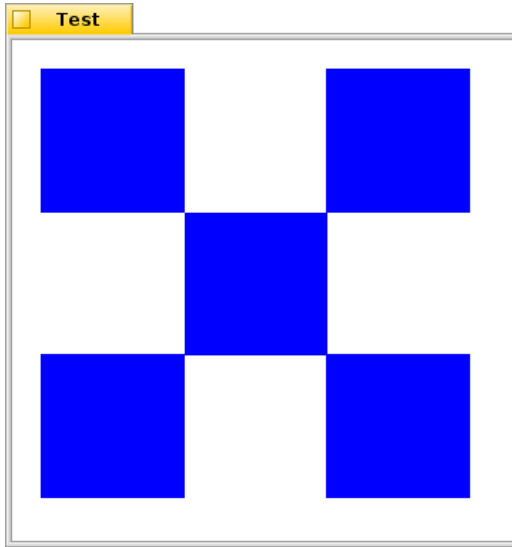


# BAffineTransform...

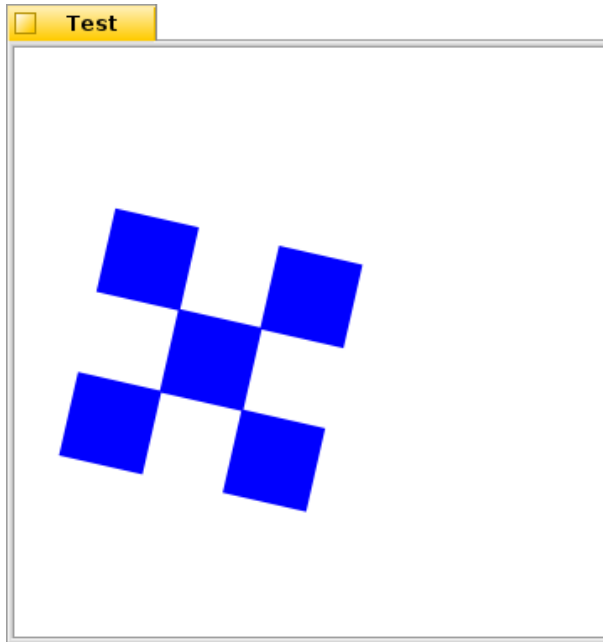
ConstrainClippingRegion()



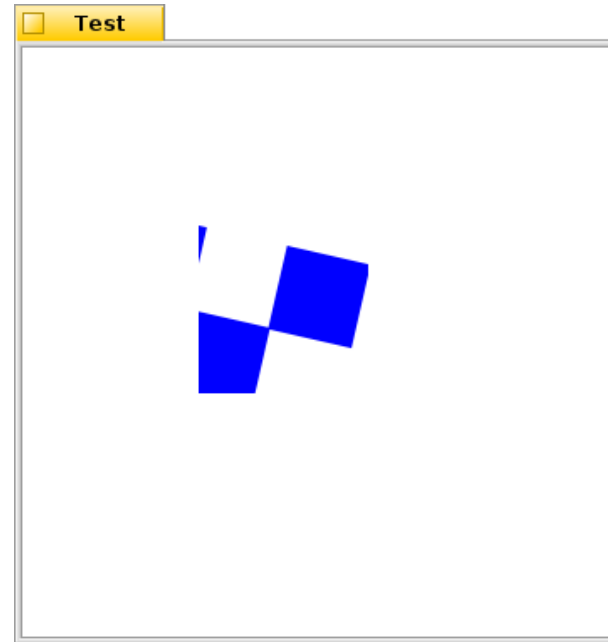
SetTransform()



# BAffineTransform...



SetTransform()



SetTransform() +  
ConstrainClippingRegion()

# Clipping Wish List

Fast for BRegion clipping

Allow complex clipping shapes

*Aware of affine transformations*

Always intersecting, no state push required



# RFC: New Clipping API

```
void BView::ClipToRect(BRect rect);
```

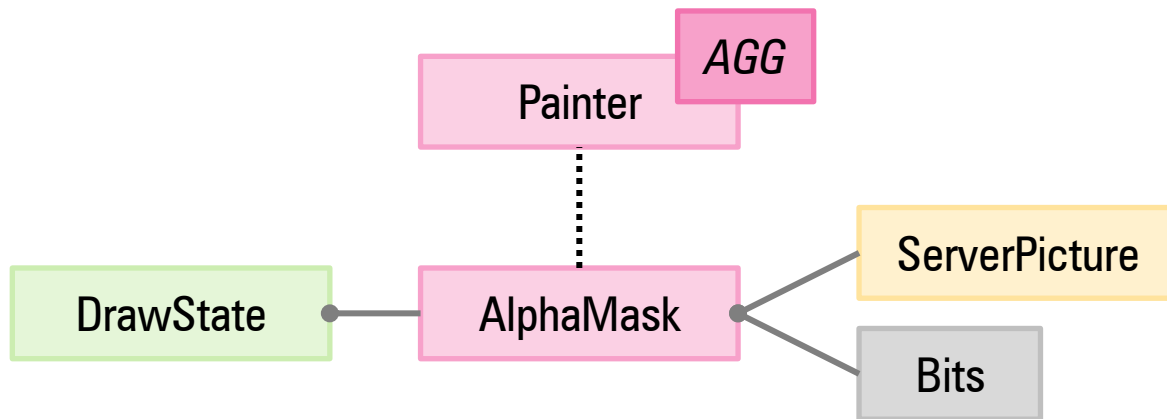
```
void BView::ClipToInverseRect(BRect rect);
```

```
void BView::ClipToShape(BShape* shape);
```

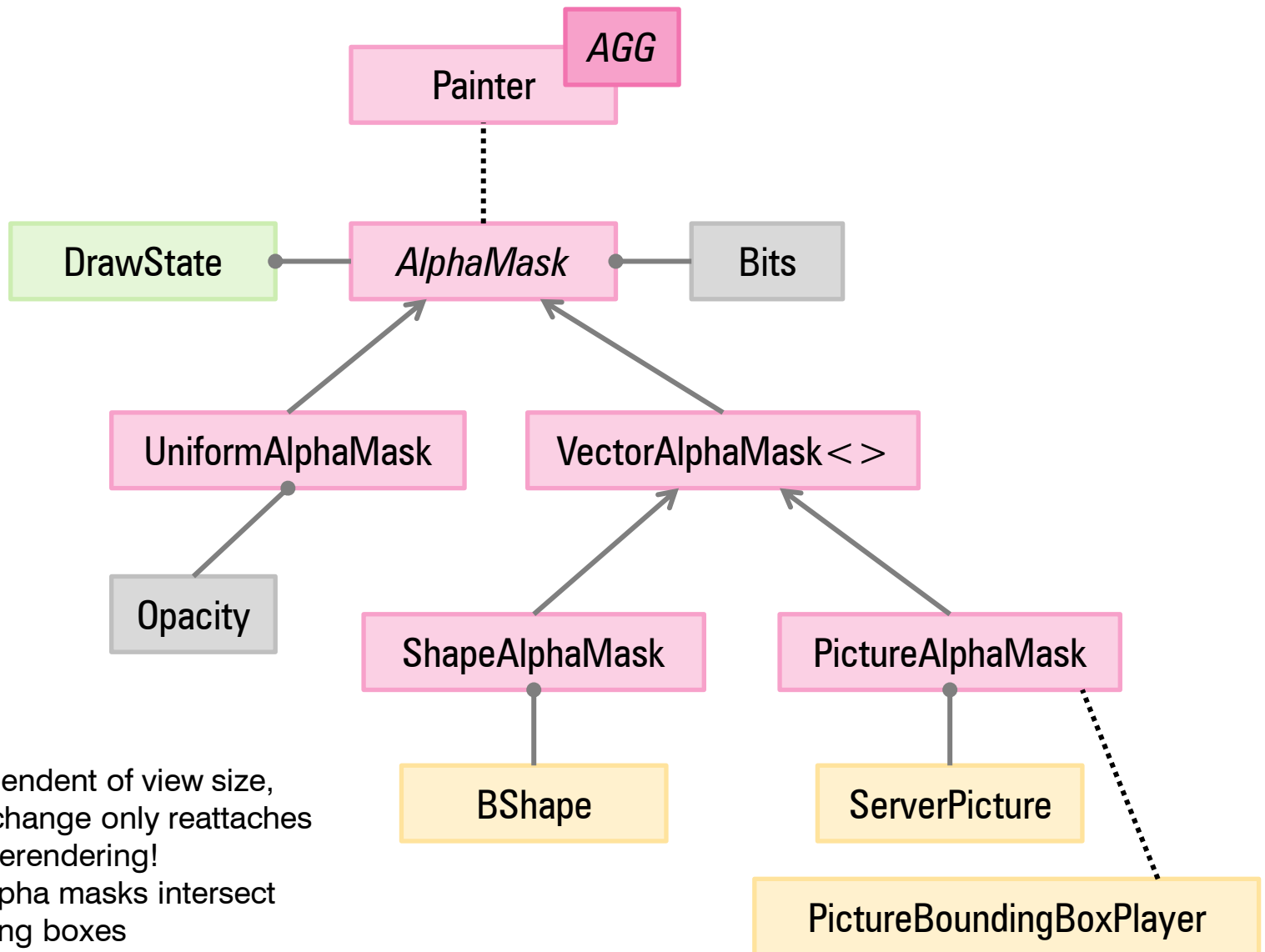
```
void BView::ClipToInverseShape(BShape* shape);
```

- Works with affine transforms
- Automatically selects between fast region clipping and alpha masks (prefers region when possible)
- Directly clip to shape without needing BPicture
- “Inverse” variants to clip out
- Always intersecting, no state push required
- All variants can be freely mixed

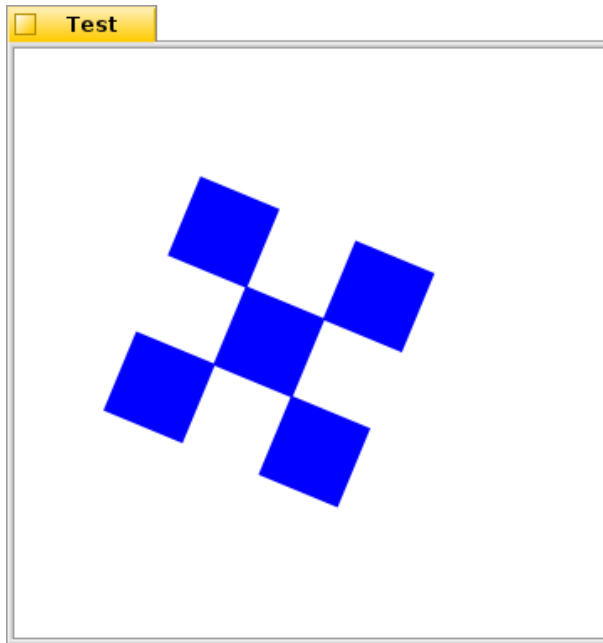
# Alpha Masks



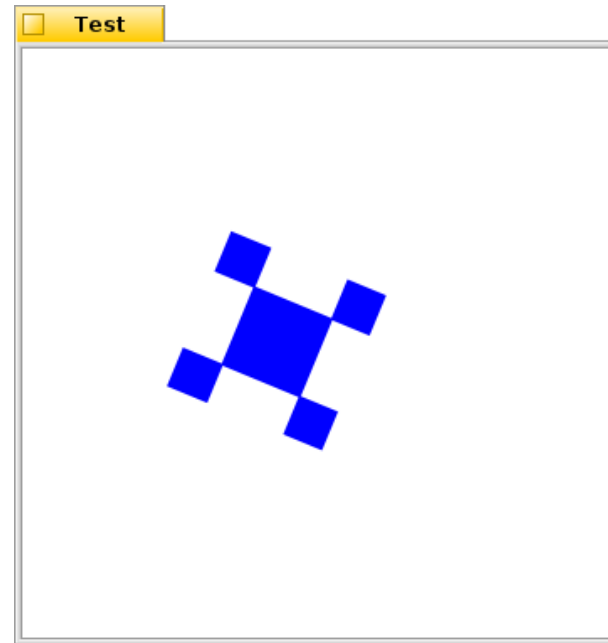
# Alpha Masks



# Clipping and BAffineTransform



SetTransform()



SetTransform() +  
ClipToRect()

Introduction

Transparency Layers

Transforms and Clipping

**Little Things Add Up**

Outlook

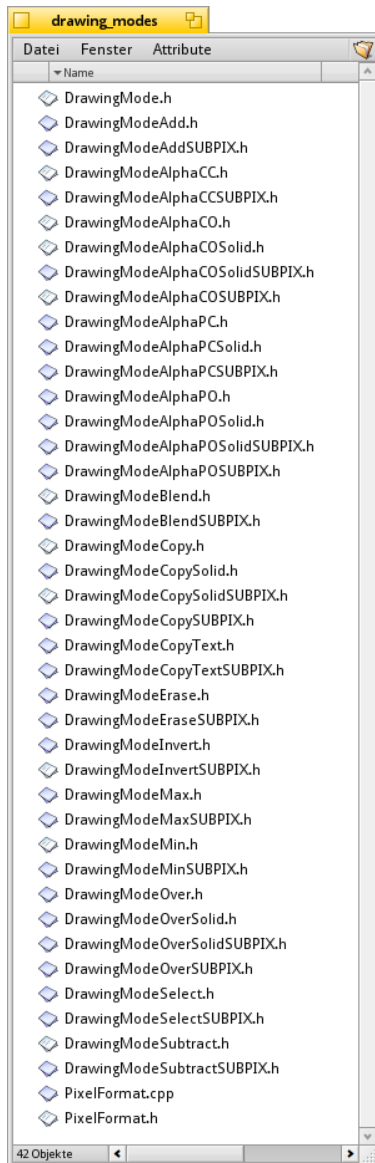
# Little Things (1): Bitmap Painter

Optimized variants in Painter



- Factored out class `BitmapPainter` and classes for optimized variants
- New optimized paths for:
  - unscaled `B_OP_COPY` with alpha mask
  - bilinear with pixel alpha overlay

# Little Things (2): Drawing Modes

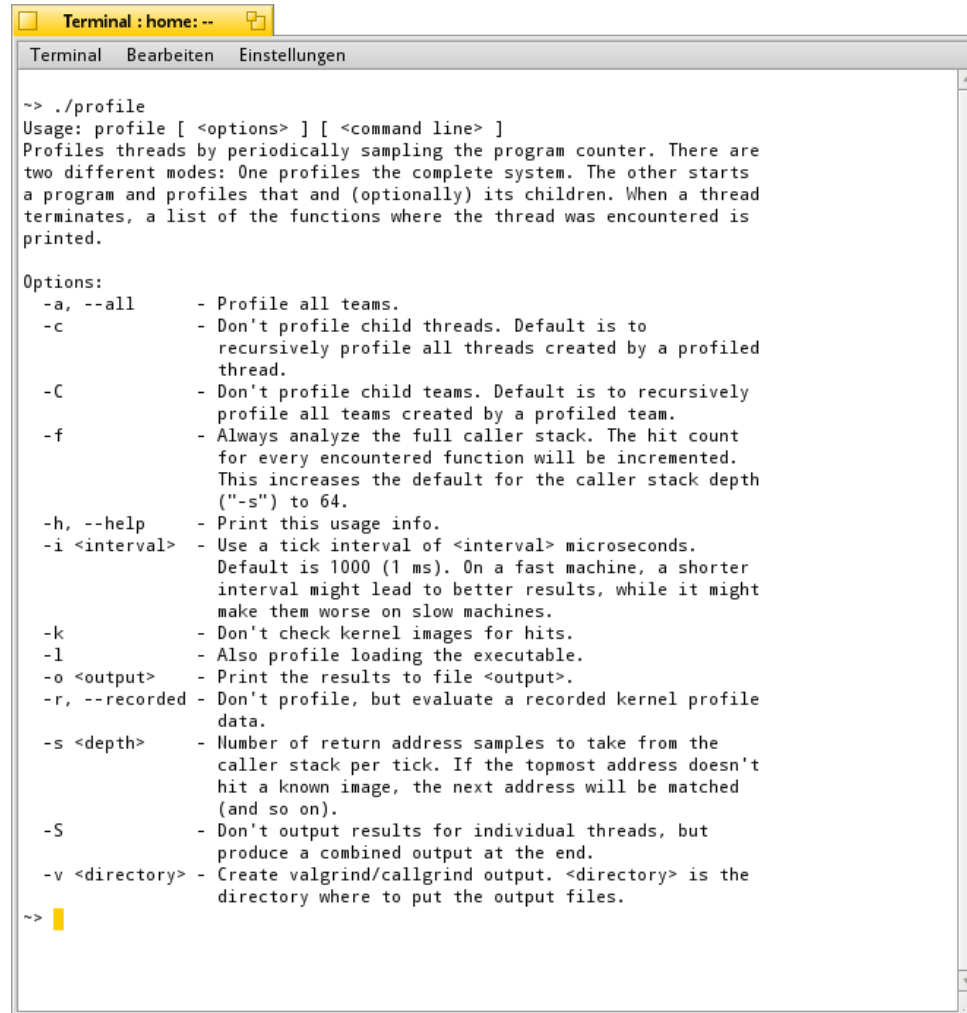


PixelFormat selects function pointers for blending  
pixels/lines/spans based on drawing mode, fill pattern, ...

Added new functions for pixel alpha composite with solid fill  
color (no pattern)...

... considerable speedup

# Little Things Can Be Hard To Find



```
Terminal : home: --
Terminal  Bearbeiten  Einstellungen

~> ./profile
Usage: profile [ <options> ] [ <command line> ]
Profiles threads by periodically sampling the program counter. There are
two different modes: One profiles the complete system. The other starts
a program and profiles that and (optionally) its children. When a thread
terminates, a list of the functions where the thread was encountered is
printed.

Options:
-a, --all      - Profile all teams.
-c            - Don't profile child threads. Default is to
               recursively profile all threads created by a profiled
               thread.
-C            - Don't profile child teams. Default is to recursively
               profile all teams created by a profiled team.
-f            - Always analyze the full caller stack. The hit count
               for every encountered function will be incremented.
               This increases the default for the caller stack depth
               ("-s") to 64.
-h, --help    - Print this usage info.
-i <interval> - Use a tick interval of <interval> microseconds.
               Default is 1000 (1 ms). On a fast machine, a shorter
               interval might lead to better results, while it might
               make them worse on slow machines.
-k            - Don't check kernel images for hits.
-l            - Also profile loading the executable.
-o <output>   - Print the results to file <output>.
-r, --recorded - Don't profile, but evaluate a recorded kernel profile
               data.
-s <depth>   - Number of return address samples to take from the
               caller stack per tick. If the topmost address doesn't
               hit a known image, the next address will be matched
               (and so on).
-S            - Don't output results for individual threads, but
               produce a combined output at the end.
-v <directory> - Create valgrind/callgrind output. <directory> is the
               directory where to put the output files.

~> █
```

← awesome

Performance can be non-intuitive – always measure



*Demo time!*

Introduction

Transparency Layers

Transforms and Clipping

Little Things Add Up

**Outlook**

# Some Ideas

- Pre-multiplied alpha
- SIMD
- Cache for alpha masks
- Cache for scaled bitmaps
- Refactoring
  - Extract more things from Painter
  - Unify clipping, transforms
- Unit tests!

**Thank you!**

Questions?