ADC Audio Developer Conference

Optimising Graphics Performance

ADC Workshop

November 2018

ADC Audio Developer Conference

Julian Storer

Head of Software Architecture at ROLI

Author of JUCE

Tom Poole

Senior JUCE Software Engineer

Workshop resources

I have emailed all of you a link to a Google Drive directory containing:

- These slides
- The projects we'll be using in this workshop

In the `repo` directory you will find a git repository containing compilable projects demonstrating some of the points we'll be talking about

If you don't know how to use git you'll also find a set of numbered directories containing the relevant files

The slide titles will tell you which git tag or directory to look in

Workshop resources

There will be an Xcode project, a VS20(17/15/13) project and a Linux Makefile in the ProjectName/Builds/[Platform] directories

If you're building on Linux, the source files are in ProjectName/Source, otherwise they will show up in your IDE

The projects are all JUCE-based, but a lot of the advice in this workshop is more general

You don't *need* to compile the projects as we go, but you are strongly encouraged to play around and use a profiler

Try compiling a project now [tag/folder 1] - if it doesn't work let us know

Talk to us

The slides and materials will continue to be available to you after this workshop, so use this opportunity to find out more about the things that matter to you. Feel free to interrupt.

We'll have at least a couple of breaks so people can stretch their legs, grab a drink, etc. This is also another opportunity for a conversation about your projects.

Profiling

You need to **measure** performance

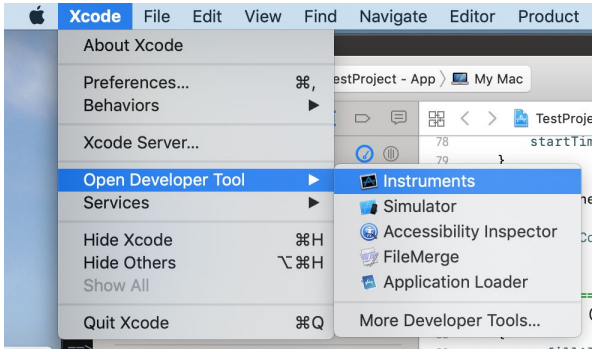
- Modern hardware is complicated
- Compiler optimisations are complicated

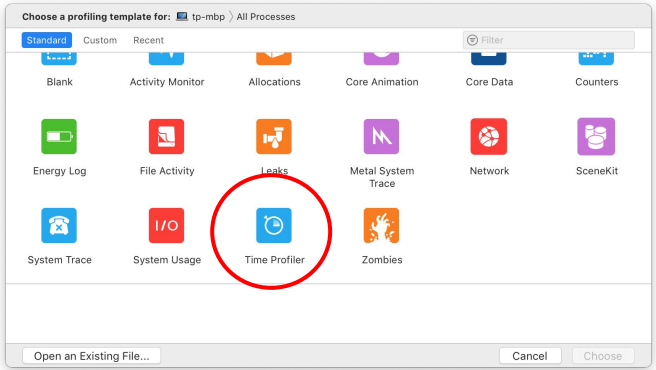
Profiling code compiled with debugging enabled is unlikely to tell you anything enlightening about your optimised code

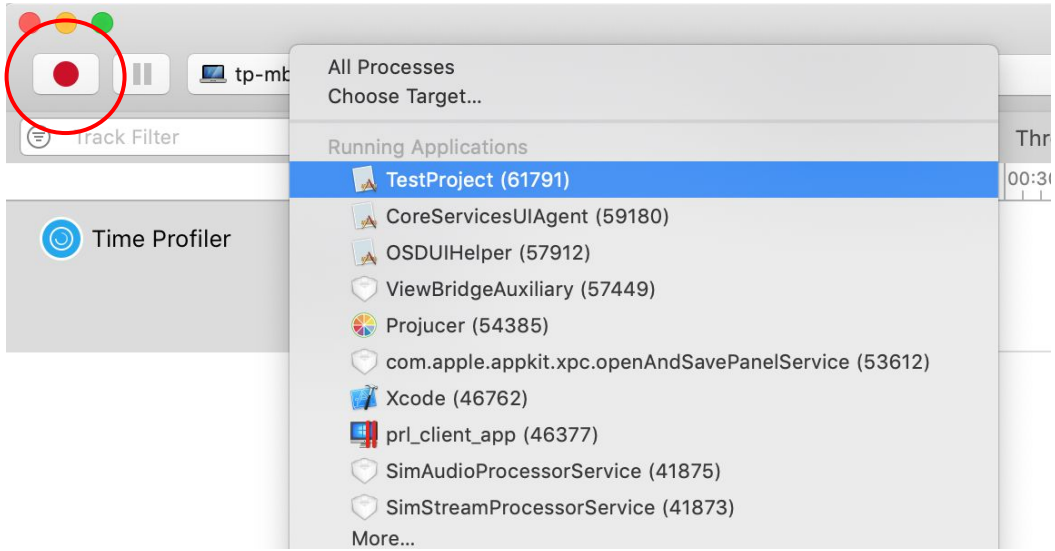
Guessing which parts of your code are the slow bits is only fruitful in fairly obvious or special cases

Be aware of fixed overheads - some things cannot be sped up significantly

Profiling - Xcode call graph

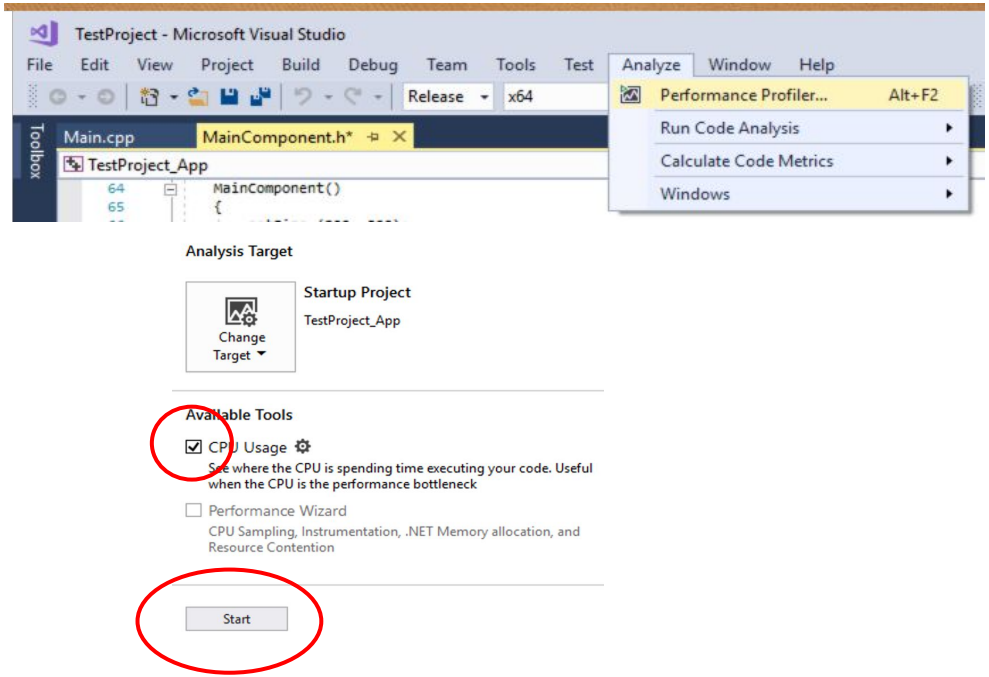


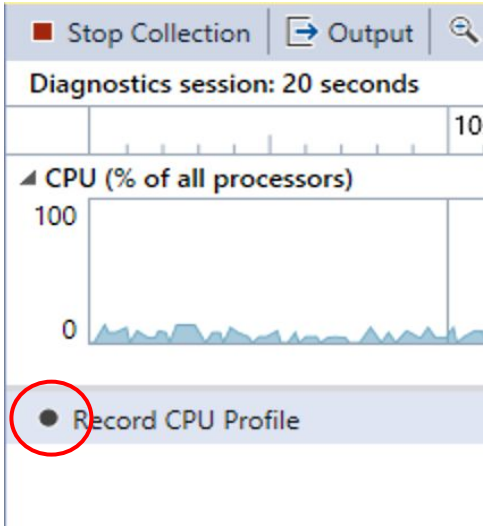




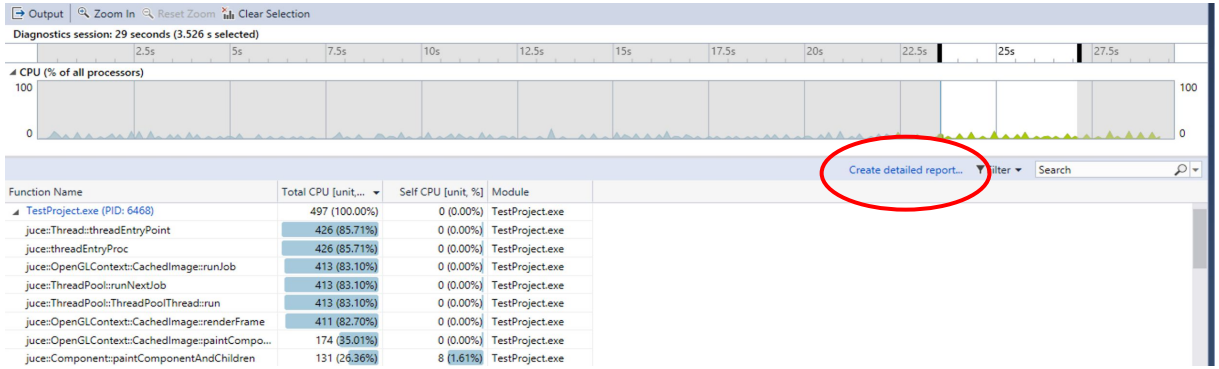
Profiling - Visual Studio call graph

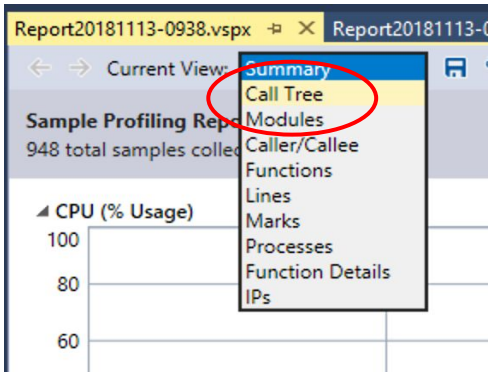
Get symbol names in your call tree first: Project Settings -> Linker -> Generate Debug Info





Profiling - Visual Studio call graph





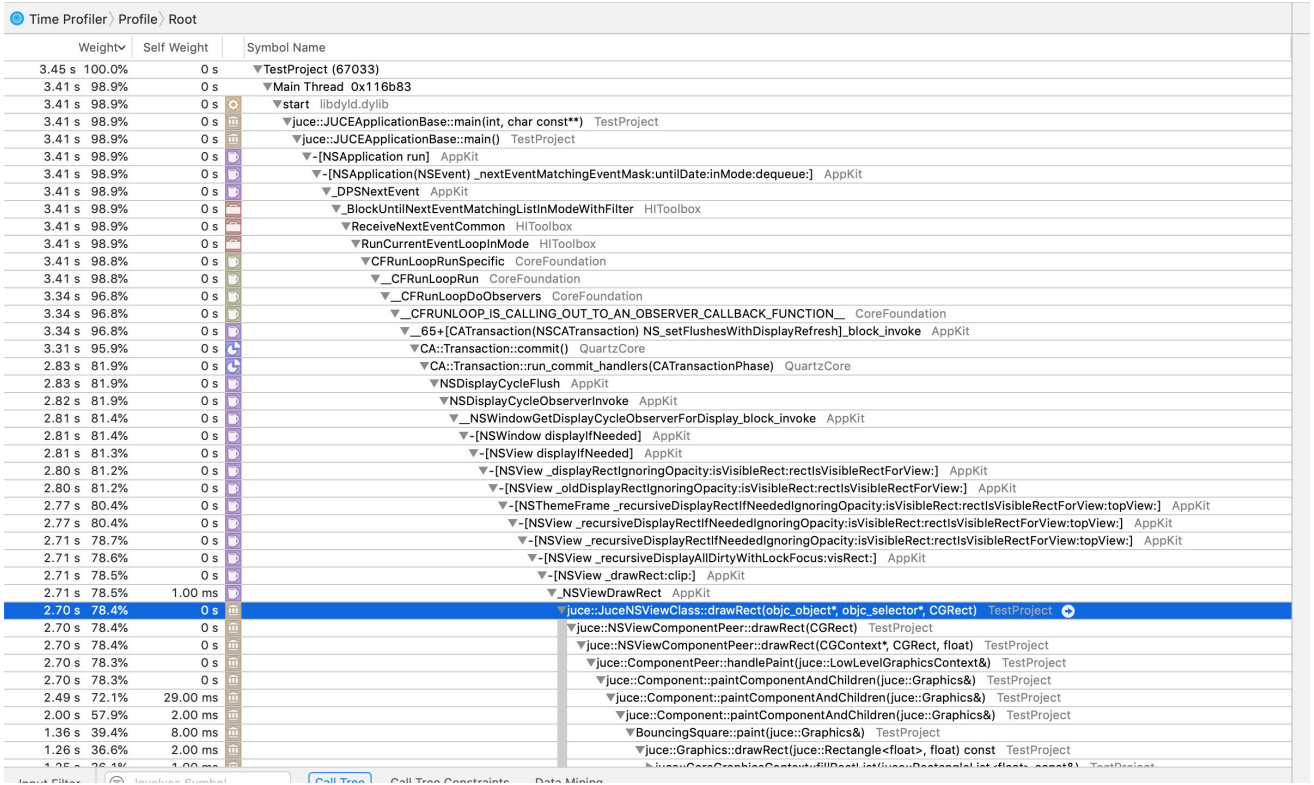
Profiling - gprof call graph

- Build with the “-pg” compiler flag
- Run the executable
- A “gmon.out” file is produced
- `gprof your_exe gmon.out > analysis.txt`

Lots of other ways to query the data:

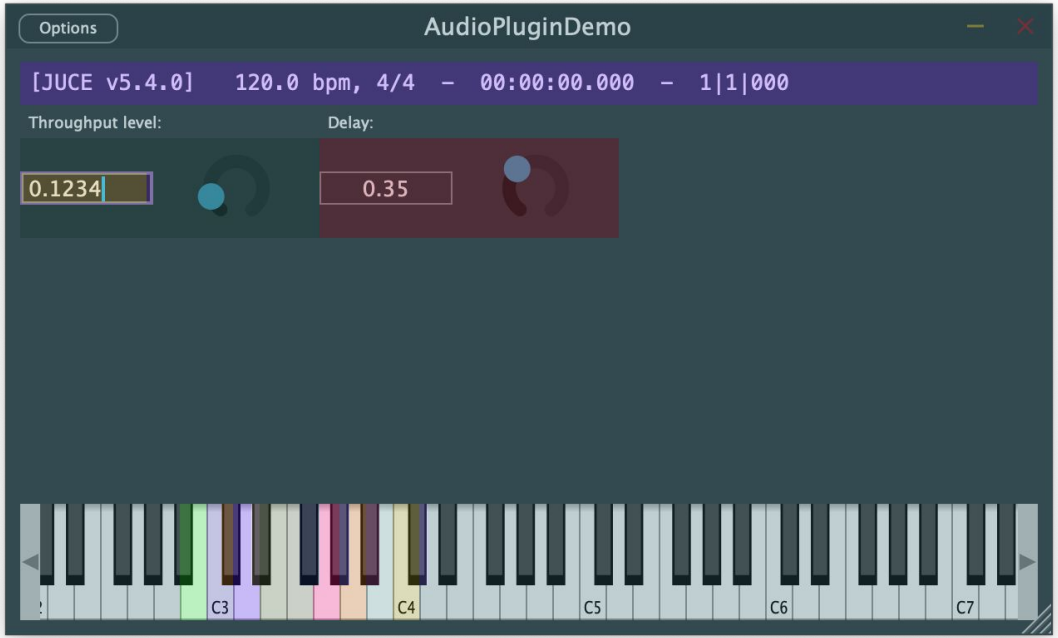
https://www.thegeekstuff.com/2012/08/gprof-tutorial/

Call graph


Time Profiler 〉 Profile 〉 Root
Weight
Self Weight
Symbol Name
3.45 s 100.0% 0 s TestProject (67033)
3.41 s 98.9% 0 s Main Thread 0x116b83
3.41 s 98.9% 0 s start libdyld.dylib
3.41 s 98.9% 0 s juce::JUCEApplicationBase::main(int, char const**) TestProject
3.41 s 98.9% 0 s juce::JUCEApplicationBase::main() TestProject
3.41 s 98.9% 0 s -[NSApplication run] AppKit
3.41 s 98.9% 0 s -[NSApplication(NSEvent) _nextEventMatchingEventMask:untilDate:inMode:dequeue:] AppKit
3.41 s 98.9% 0 s _DPSNextEvent AppKit
3.41 s 98.9% 0 s _BlockUntilNextEventMatchingListInModeWithFilter HIToolbox
3.41 s 98.9% 0 s ReceiveNextEventCommon HIToolbox
3.41 s 98.9% 0 s RunCurrentEventLoopInMode HIToolbox
3.41 s 98.8% 0 s CFRunLoopRunSpecific CoreFoundation
3.41 s 98.8% 0 s __CFRunLoopRun CoreFoundation
3.34 s 96.8% 0 s __CFRunLoopDoObservers CoreFoundation
3.34 s 96.8% 0 s __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ CoreFoundation
3.34 s 96.8% 0 s __65+[CATransaction(NSCATransaction) NS_setFlushesWithDisplayRefresh]_block_invoke AppKit
3.31 s 95.9% 0 s CA::Transaction::commit() QuartzCore
2.83 s 81.9% 0 s CA::Transaction::run_commit_handlers(CATransactionPhase) QuartzCore
2.83 s 81.9% 0 s NSDisplayCycleFlush AppKit
2.82 s 81.9% 0 s NSDisplayCycleObserverInvoke AppKit
2.81 s 81.4% 0 s __NSWindowGetDisplayCycleObserverForDisplay_block_invoke AppKit
2.81 s 81.4% 0 s -[NSWindow displayIfNeeded] AppKit
2.81 s 81.3% 0 s -[NSView displayIfNeeded] AppKit
2.80 s 81.2% 0 s -[NSView _displayRectIgnoringOpacity:isVisibleRect:rectIsVisibleRectForView:] AppKit
2.80 s 81.2% 0 s -[NSView _oldDisplayRectIgnoringOpacity:isVisibleRect:rectIsVisibleRectForView:] AppKit
2.77 s 80.4% 0 s -[NSThemeFrame _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect:rectIsVisibleRectForView:topView:] AppKit
2.77 s 80.4% 0 s -[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect:rectIsVisibleRectForView:topView:] AppKit
2.71 s 78.7% 0 s -[NSView _recursiveDisplayRectIfNeededIgnoringOpacity:isVisibleRect:rectIsVisibleRectForView:topView:] AppKit
2.71 s 78.6% 0 s -[NSView _recursiveDisplayAllDirtyWithLockFocus:visRect:] AppKit
2.71 s 78.5% 0 s -[NSView _drawRect:clip:] AppKit
2.71 s 78.5% 1.00 ms _NSViewDrawRect AppKit
2.70 s 78.4% 0 s juce::JuceNSViewClass::drawRect(objc_object*, objc_selector*, CGRect) TestProject
2.70 s 78.4% 0 s juce::NSViewComponentPeer::drawRect(CGRect) TestProject
2.70 s 78.4% 0 s juce::NSViewComponentPeer::drawRect(CGContext*, CGRect, float) TestProject
2.70 s 78.3% 0 s juce::ComponentPeer::handlePaint(juce::LowLevelGraphicsContext&) TestProject
2.70 s 78.3% 0 s juce::Component::paintComponentAndChildren(juce::Graphics&) TestProject
2.49 s 72.1% 29.00 ms juce::Component::paintComponentAndChildren(juce::Graphics&) TestProject
2.00 s 57.9% 2.00 ms juce::Component::paintComponentAndChildren(juce::Graphics&) TestProject
1.36 s 39.4% 8.00 ms BouncingSquare::paint(juce::Graphics&) TestProject
1.26 s 36.6% 2.00 ms juce::Graphics::drawRect(juce::Rectangle<float>, float) const TestProject
Call Tree
Call Tree Constraints
Data Mining

Profiling JUCE graphics

#define JUCE_ENABLE_REPAINT_DEBUGGING 1



General advice

Make things opaque

- Avoid repainting the components behind other components

Use floating point sizes

- Pixels are much less relevant than they used to be

Small changes in drawing can have dramatic effects

Temporarily disable link time optimisation (which is set by default for Release builds) so you can rebuild faster

Calling repaint()

When you mark an area on the screen as needing to be redrawn (calling Component::repaint() or Component::repaint (dirtyArea)):

- A message is sent to the operating system specifying the area
- A short while later the operating system calls the appropriate “drawRect” routine in the corresponding native view
- The native view ultimately calls paint() on your component
- An asynchronous process!
- The operating system may coalesce multiple repaint messages into a single, larger, paint() region

Calling repaint()

Avoid hammering your app's message queue with repaint requests

Use JUCE's Timer class to limit the number of repaint requests

- Rather than attempting to repaint() on every change, call repaint on a timer

JUCE's AsyncUpdater is another option

- Automatically squashes multiple rapid update requests into a single message

BouncingSquares - initial state [1]

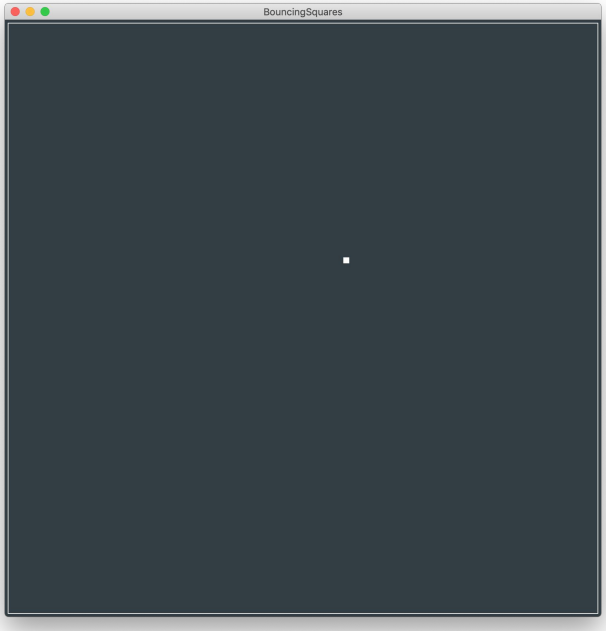
The first commit in the git repository, or directory 1

Compile the project in the Release config

You should see a single bouncing square…

… using about 29% of a CPU

(Your percentages may differ wildly!)



BouncingSquares - initial code [1]

Audio Developer Conference

A single BouncingSquare component taking up most of our MainComponent

```
//==============================================================================
// A component that displays a bouncing square
class BouncingSquare    : public Component,
                          private Timer
{
public:
    //==========================================================================
    BouncingSquare (Point<float> initialVelocity)
        : velocity (initialVelocity)
    {
        setSize (100, 100);

        // We will update the square's position at 60 Hz
        startTimerHz (60);
    }

    //==========================================================================
    void paint (Graphics& g) override
    {
        g.setColour (Colours::white);
        g.drawRect (getLocalBounds(), 1);

        g.fillRect (square);
    }
```

```
private:
    //==========================================================================
    void timerCallback() override
    {
        // Update the square's position
        square += velocity;

        // If the square has gone beyond an edge bounce it

        if (square.getX() < 0.0f)
        {
            square.setX (-square.getX());
            velocity.setX (-velocity.getX());
        }
        else if (square.getRight() > getWidth())
        {
            auto overshoot = square.getRight() - getWidth();
            square.setX (getWidth() - (square.getWidth() + overshoot));
            velocity.setX (-velocity.getX());
        }

        if (square.getY() < 0.0f)
        {
            square.setY (-square.getY());
            velocity.setY (-velocity.getY());
        }
        else if (square.getBottom() > getHeight())
        {
            auto overshoot = square.getBottom() - getHeight();
            square.setY (getHeight() - (square.getHeight() + overshoot));
            velocity.setY (-velocity.getY());
        }

        // Tell the OS to redraw the component
        repaint();
    }

    //==========================================================================
    Rectangle<float> square { 8.0f, 8.0f };
    Point<float> velocity;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (BouncingSquare)
};
```

BouncingSquares - repaint dbg [2]

Turn on JUCE_ENABLE_REPAINT_DEBUGGING in

BouncingSquares/JuceLibraryCode/AppConfig.h

```
//==============================================================================
// [BEGIN_USER_CODE_SECTION]

// (You can add your own code in this section, and the Projucer will not overwrite it)

#define JUCE_ENABLE_REPAINT_DEBUGGING 1

// [END_USER_CODE_SECTION]
```

BouncingSquares - background [2]

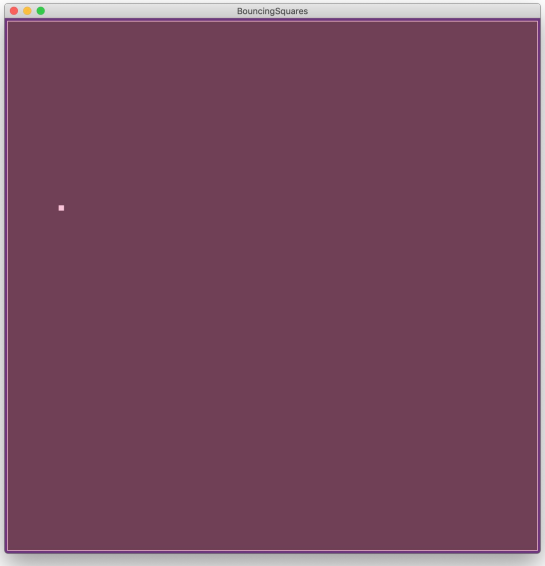
The whole component is being repainted at on each redraw, at whatever refresh rate your monitor supports (which is probably less than 60 Hz)

But only a small amount is actually changing

How much is this slowing us down?

Get the profiler out

(Turn off repaint debugging first!)

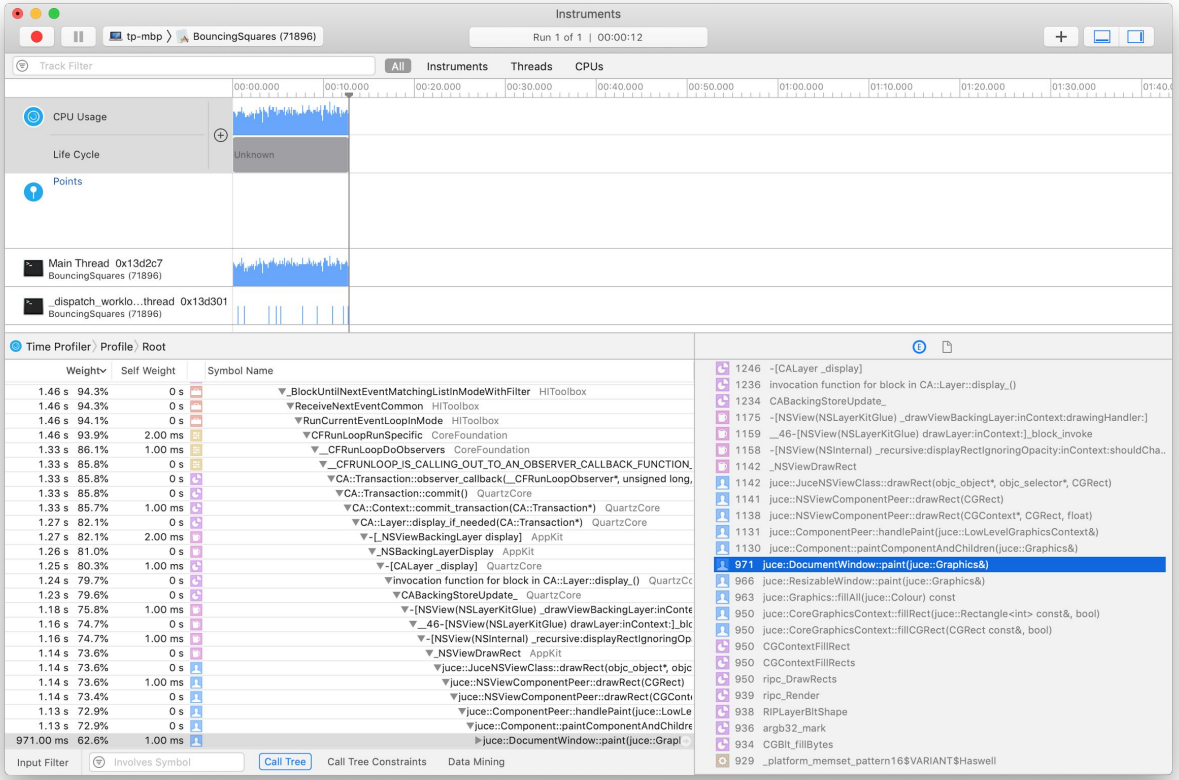


BouncingSquares - profile

DocumentWindow::paint()

(It’s the background)

About 60% of time there



BouncingSquares - redraw changes [3]

```
void timerCallback() override
{
    Rectangle<float> dirtyArea (square);

    // Update the square's position
    square += velocity;
```

--- bouncing logic is here...

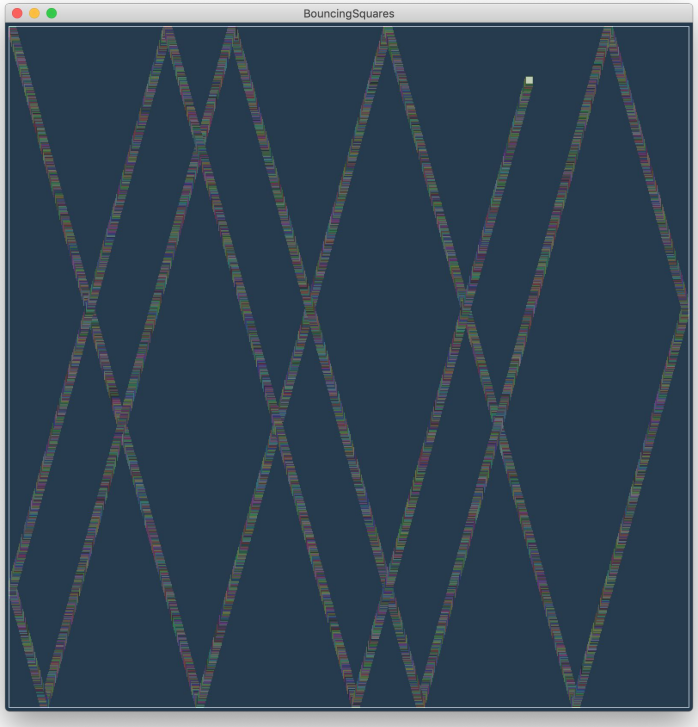
```
    dirtyArea = dirtyArea.getUnion (square);

    // Only repaint the bits that have changed
    repaint (dirtyArea.getSmallestIntegerContainer());
}
```

BouncingSquares - repaint dbg [3]

Compile the project and check that we're only repaining what we need to

Disable repaint debugging and see how the CPU use has changed

Down to 5% (from 29%)

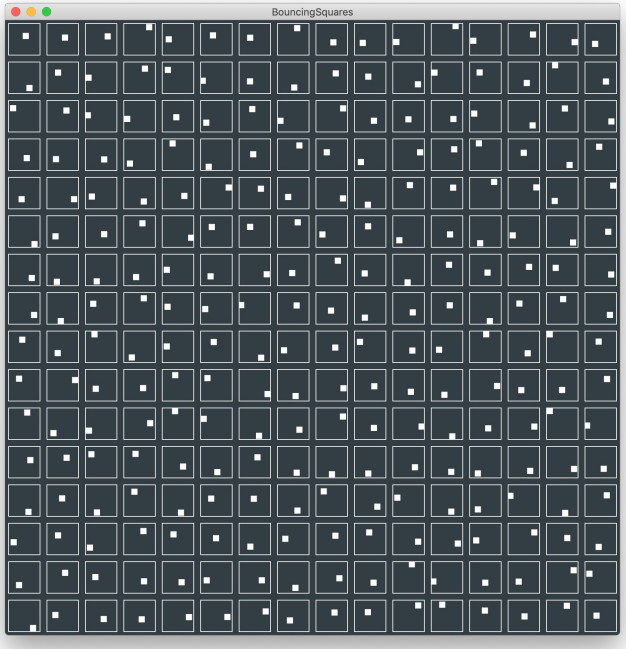


BouncingSquares - multiple squares [4]

Edit MainComponent.h so that we're now displaying a grid of squares

```
int numRows = 16, numCols = 16, margin = 4;
```

CPU usage now at 97%



BouncingSquares - repaint dbg [4]

This bit is Apple only

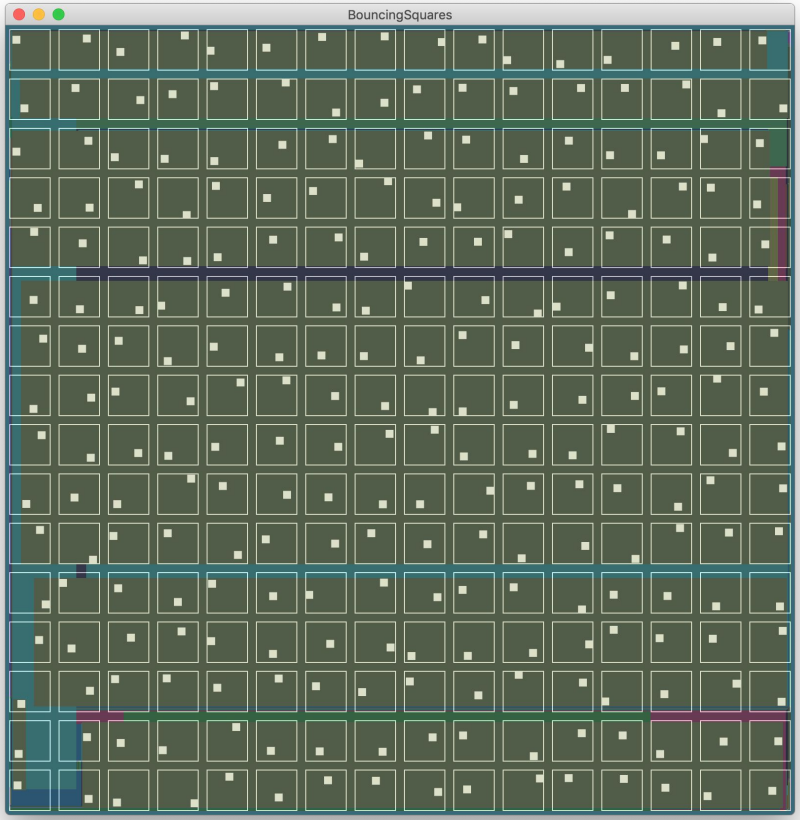
CoreGraphics is consolidating all of our individual dirty rects into much larger areas

Can counter this by setting

JUCE_COREGRAPHICS_RENDER_WITH_MULTIPLE_PAINT_CALLS

in AppConfig.h

Does not work in Xcode 10...



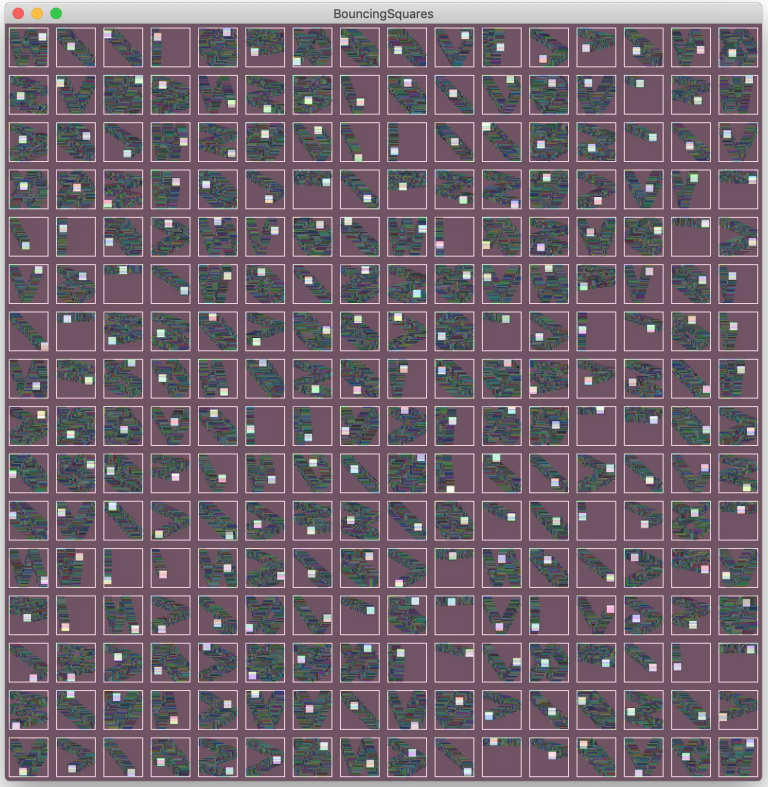
BouncingSquares - Multiple paints [5]

```
//==============================================================================
// [BEGIN_USER_CODE_SECTION]

// (You can add your own code in this section, and the Projucer will not overwrite it)

#define JUCE_ENABLE_REPAINT_DEBUGGING 1
#define JUCE_COREGRAPHICS_RENDER_WITH_MULTIPLE_PAINT_CALLS 1

// [END_USER_CODE_SECTION]
```

Apple and other OSs are now in the same state

We've made art! Very slow art.

100% CPU, huge lag.



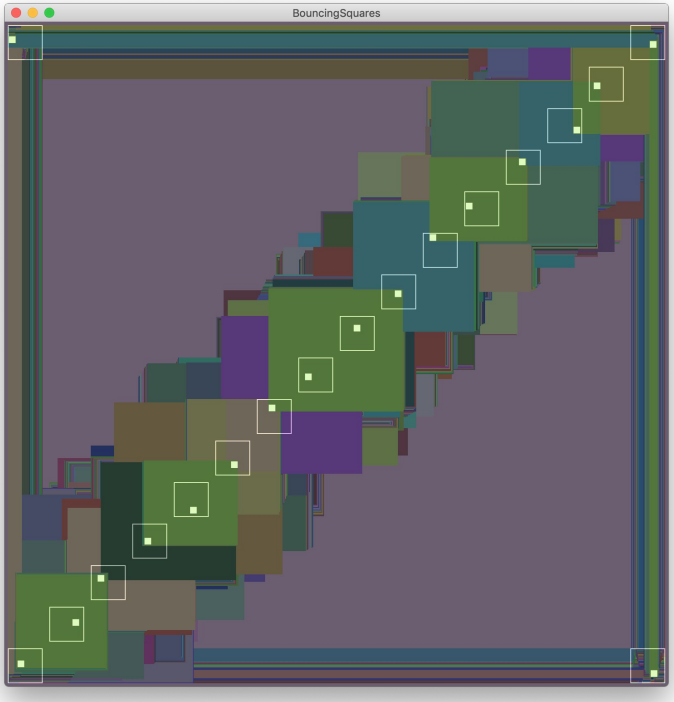
BouncingSquares - multiple calls

JUCE_COREGRAPHICS_RENDER_WITH_MULTIPLE_PAINT_CALLS

Can be very useful

- Well separated animations
- An expensive background

Not working in the latest CoreGraphics SDK!



BouncingSquares - repaint parent [6]

Repainting less has made performance much worse

- Repaint areas very fragmented
- Much more strain on the message thread

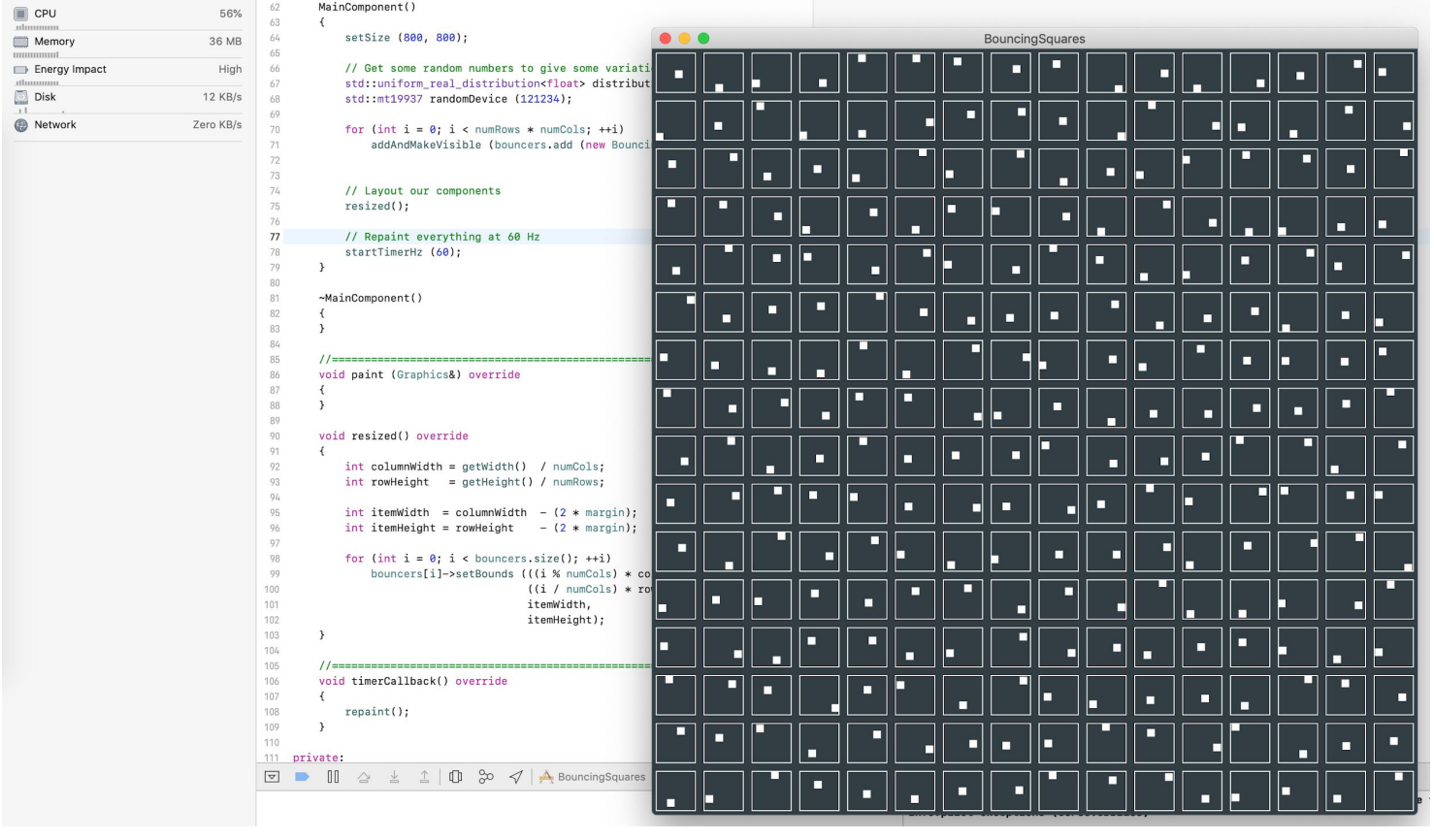
Instead we can repaint the whole thing on a 60 Hz timer

- Remove repaint() calls in the child components
- Make the MainComponent inherit from Timer
- At 60 Hz call repaint()

BouncingSquares - parent results [6]

56% CPU

(from 97%)

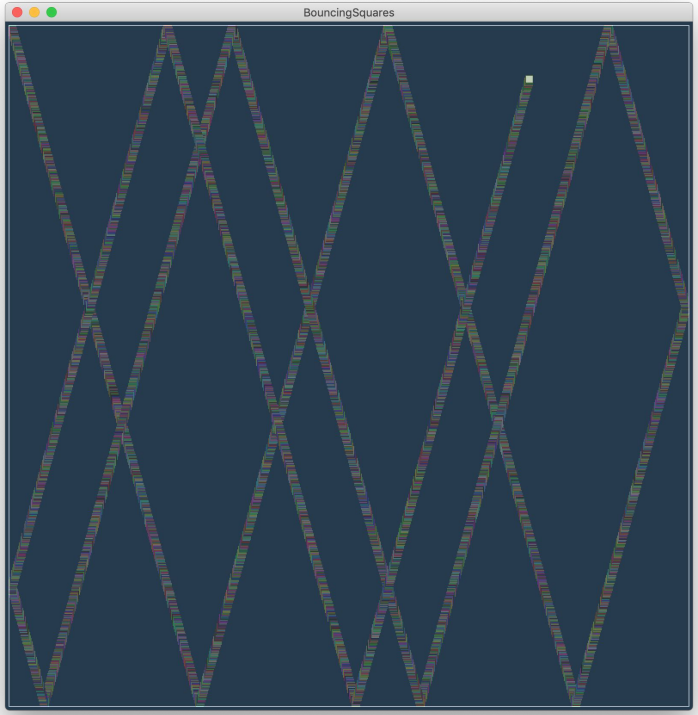


Components vs direct drawing

By moving a “Square” component, rather than drawing a square directly, you can minimise the amount of redrawing required automatically

- JUCE will handle repainting the dirty region

There’s some overhead, unless...



Unclipped painting

Component::setPaintingIsUnclipped (true)

This removes the overhead of clipping the graphics context that gets passed to the component's paint() callback.

Clipping can be surprisingly expensive!

Make sure you know that the component will not draw outside of its boundary

- Avoid things like Graphics::fillAll()

How does JUCE actually draw things?

Audio Developer Conference

```
void Graphics::drawLine (Line<float> line, const float lineThickness) const
{
    Path p;
    p.addLineSegment (line, lineThickness);
    fillPath (p);
}


void Graphics::fillPath (const Path& path) const
{
    if (! (context.isClipEmpty() || path.isEmpty()))
        context.fillPath (path, AffineTransform());
}
```

How does JUCE actually draw things?

```
void fillPath (const Path& path, const AffineTransform& t)
{
    if (clip != nullptr)
    {
        auto trans = transform.getTransformWith (t);
        auto clipRect = clip->getClipBounds();

        if (path.getBoundsTransformed (trans).getSmallestIntegerContainer().intersects (clipRect))
            fillShape (*new EdgeTableRegionType (clipRect, path, trans), false);
    }
}
```

The CoreGraphics renderer has a higher level interface - you can fill a path directly

How does JUCE actually draw things?

Audio Developer Conference

Everything boils down to edge tables and images

- Edge tables are lists of horizontal slices
- The simpler the edge table, the faster the rendering

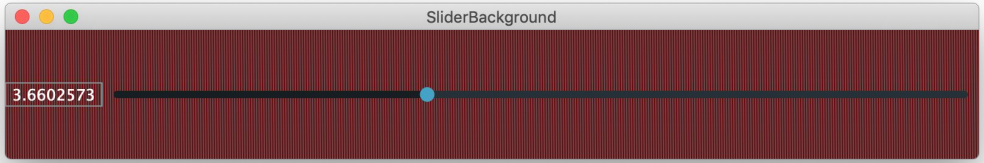
Clip regions make a real difference

Calling into the graphics stack has some overhead

- Minimising the number of draw*() calls can be beneficial

SliderBackground - lines [7]

```
class BackgroundComponent    : public Component
{
public:
    BackgroundComponent()
    {
        setOpaque (true);

        setSize (800, 100);
    }

    void paint (Graphics& g) override
    {
        g.setColour (Colours::maroon);
        g.fillAll();

        g.setColour (Colours::darkgrey);

        for (float i = 0.0f; i < (float) getWidth(); i += 2.0f)
            g.drawLine (i, 0.0f, i + 1.0f, (float) getHeight(), 1.0f);
    }

private:
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (BackgroundComponent)
};
```

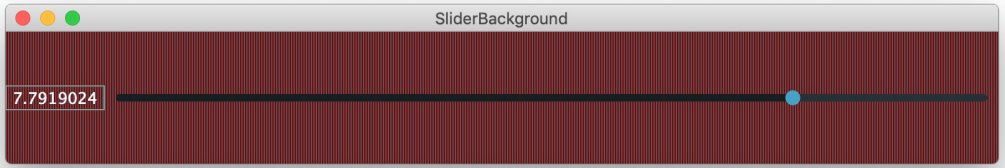


50% CPU usage when moving the slider rapidly

Don’t use drawLine for horizontal or vertical lines!

SliderBackground - rects [8]

```
for (float i = 0.0f; i < (float) getWidth(); i += 2.0f)
    g.drawLine (i, 0.0f, i + 1.0f, (float) getHeight(), 1.0f);
```

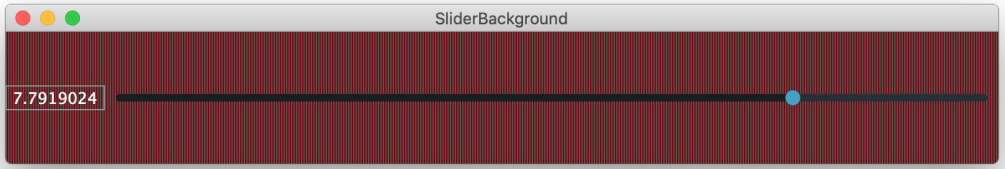



```
for (float i = 0.0f; i < (float) getWidth(); i += 2.0f)
    g.fillRect (Rectangle<float> (i, 0.0f, 1.0f, (float) getHeight()));
```

35% CPU usage when moving the slider rapidly (down from 50%)

SliderBackground - RectangleList [9]

```
for (float i = 0.0f; i < (float) getWidth(); i += 2.0f)
    g.fillRect (Rectangle<float> (i, 0.0f, 1.0f, (float) getHeight()));
```




```
RectangleList<float> rl;

for (float i = 0.0f; i < (float) getWidth(); i += 2.0f)
    rl.addWithoutMerging ({ i, 0.0f, 1.0f, (float) getHeight() });

g.fillRectList (rl);
```

24% CPU usage when moving the slider rapidly (down from 35% <- 50%)

Images

JUCE’s Image class is a reference countered wrapper around image data

- Avoid accidentally reloading images
- But don’t be afraid to make copies!

If you’re using the same image in multiple, separated, places, have a look at ImageCache

- Global scope
- Images kept in memory for a few seconds after being released
    - Helps prevent unintentional reloading

Using images

The time taken to draw images is both system and rendering engine dependent

Some broadly applicable advice:

- CoreGraphics is usually faster than the software renderer (but the performance varies wildly)
- The software renderer is slow
- OpenGL is really fast

(More on OpenGL later)

Component setBufferedToImage

Your component could be expensive to draw

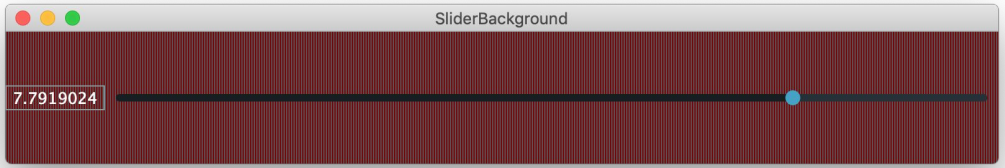
- Lots of calculations required in paint()
- Large size

If the paint method takes longer than painting an image of the same size call Component::setBufferedToImage (true)

- Draws the content of the component into an image
- If the content is unchanged, but needs to redraw (if it’s in the background and something moves in front of it, for example), then the buffered image will be used

SliderBackground - buffered image [10]

```
BackgroundComponent()
{
    setOpaque (true);
    setBufferedToImage (true);

    setSize (800, 100);
}
```



18% CPU usage when moving the slider rapidly

(down from 24% <- 35% <- 50%)

Saving and restoring graphics contexts

Graphics::saveState()

Graphics::restoreState()

Graphics::ScopedSaveState

If you've modified the clip region, added a transform, set some unusual colours, or otherwise changed the graphics state, you can pop back to a previous state

Affine Transforms

$$\begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

A compact way to represent

- Scaling, rotations, shearing, reflections (a, b, c, d)
- Translations (tx, ty)

of any 2D point (x, y)

Affine Transforms in JUCE

You don’t need to know much geometry

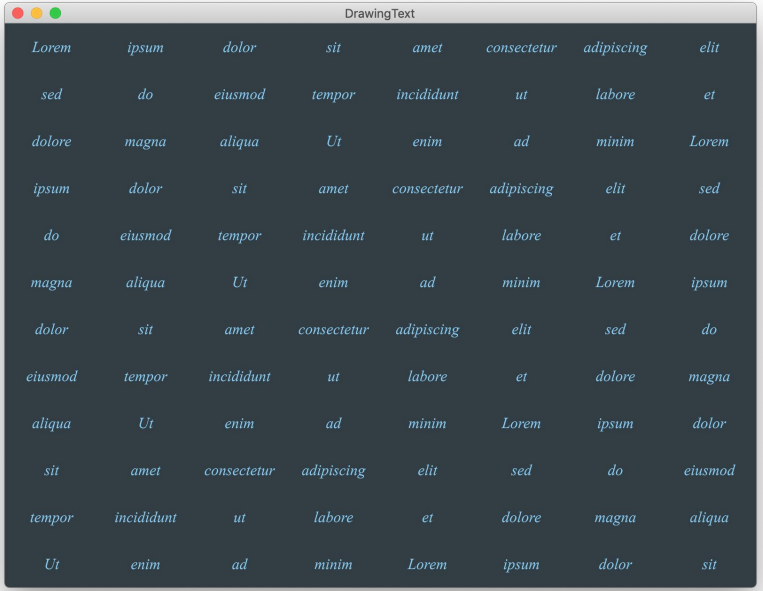
```
AffineTransform transform = AffineTransform::scale (2, 2).translated (0, 15);
```

Everything in JUCE has an implicit transform

Applying additional AffineTransforms does not increase the computational overhead

However, drawing between a pixel barrier or moving away from horizontal or vertical lines may be more expensive

DrawingText [11]



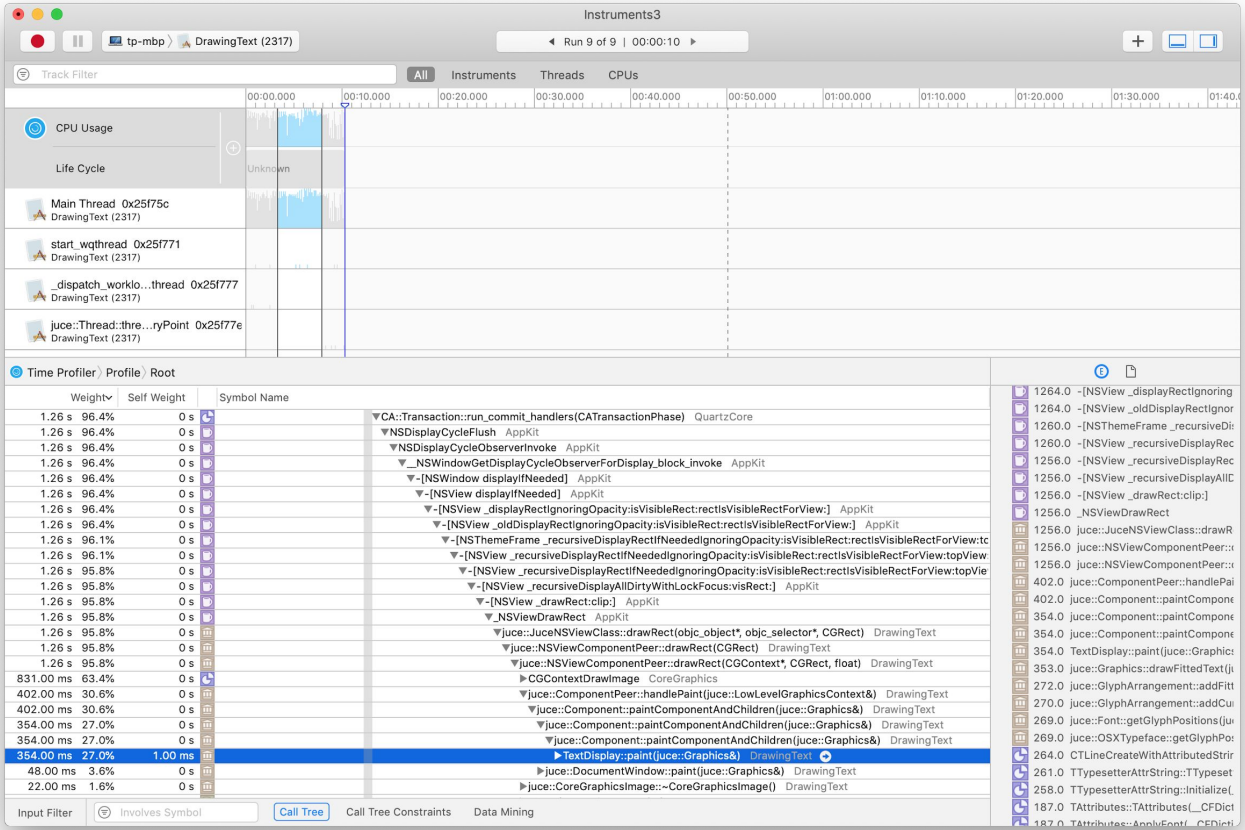

```
void paint (Graphics& g) override
{
    g.setColour (Colours::lightskyblue);
    g.setFont (textFont);
    auto wordIndex = 0;

    for (int i = 0; i < numRows; ++i)
    {
        for (int j = 0; j < numCols; ++j)
        {
            Rectangle<int> bounds (j * columnWidth, i * rowHeight, columnWidth, rowHeight);
            g.drawFittedText (words[wordIndex++ % words.size()], bounds, textJustification, 1);
        }
    }
}
```

DrawingText - GlyphArrangement [12]

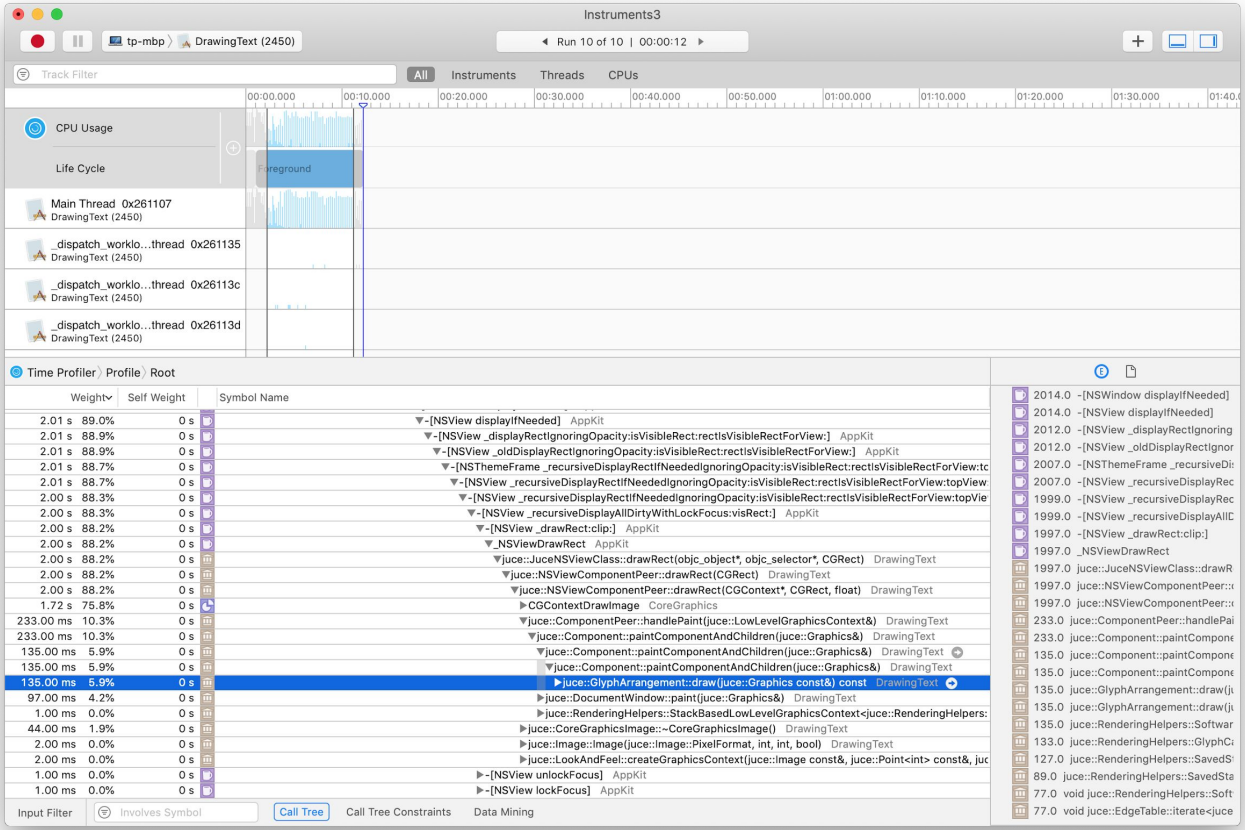
Audio Developer Conference

```
void paint (Graphics& g) override
{
    g.setColour (Colours::lightskyblue);
    glyphs.draw (g);
}

void resized() override
{
    columnWidth = getWidth()  / numCols;
    rowHeight   = getHeight() / numRows;

    glyphs.clear();

    auto wordIndex = 0;

    for (int i = 0; i < numRows; ++i)
    {
        for (int j = 0; j < numCols; ++j)
        {
            Rectangle<int> bounds (j * columnWidth, i * rowHeight, columnWidth, rowHeight);
            glyphs.addFittedText (textFont, words[wordIndex++ % words.size()],
                                  bounds.getX(), bounds.getY(),
                                  bounds.getWidth(), bounds.getHeight(),
                                  textJustification, 1);
        }
    }
}
```

DrawingText [11]



DrawingText - GlyphArrangement [12]



AttributedString / TextLayout

By moving the layout of the glyphs out of the paint() routine we've dropped from drawing text in 27% of frames to 5.9% of frames in the profiler

JUCE has more sophisticated classes to help you do text layout outside of rendering

- AttributedString: A tagged string where you can change colour, font, style, etc. midway through
- TextLayout: From an AttributedString you can create a TextLayout, which is a lightweight wrapper around lines of GlyphArrangements

Note that in the DrawingText project we could not have used a background image instead - the text size stays constant when the component is resized

Threads

Drawing to the screen happens on the message thread

If you spend too much time drawing, then your entire app will become unresponsive!

Clogging up the message thread means other events like keyboard and mouse interactions, window manager notifications, timers, etc, will all become delayed

If possible, do the heavy lifting on a different thread

Threads

Calling drawing functions on threads other than your app's main thread must be done very carefully

- Recent macOS versions may flag this as a warning even if you hold a MessageManagerLock!

Solutions:

- Take a MessageManagerLock (easy to deadlock!)
- Lock with a Mutex/CriticalSection
- Use a lock-free FIFO to pass data

OpenGL

OpenGL is a specification for communicating graphics information between a CPU and a GPU

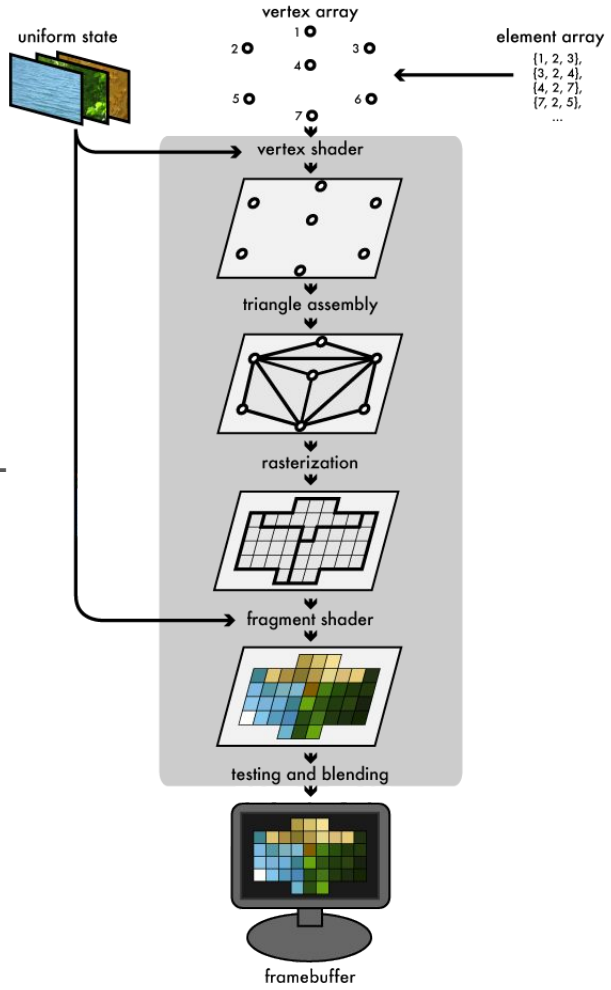
- Using OpenGL relies upon OpenGL implementations, frequently provided by proprietary graphics card drivers
- OpenGL has changed a lot over the years and versioning is not obvious
- Support in devices is very variable, but usually fast if it works
- Very verbose

OpenGL

Image borrowed from

http://duriansoftware.com

A good introduction to OpenGL



OpenGL for JUCE components

If you don’t want to use bespoke OpenGL code in your JUCE app, there are convenience methods to get JUCE to use OpenGL as the current graphics context

- Create an OpenGLContext
- Attach your top level component to the OpenGLContext

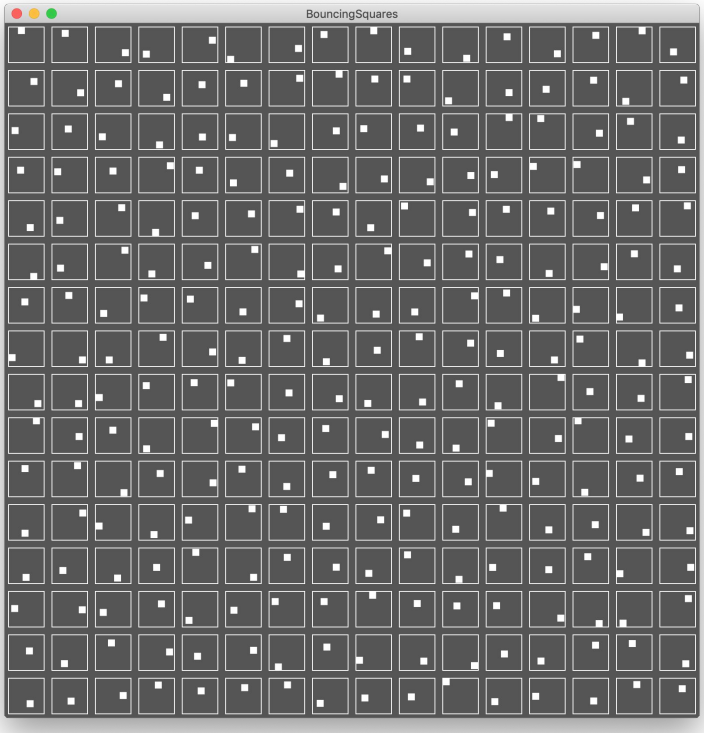
When you draw to the graphics context you take the same path as the software renderer, but you send edge table rectangles (as two triangles) to the graphics card instead

1.01 s	68.5%	3.00 ms	▼BouncingSquare::paint(juce::Graphics&) BouncingSquares
1.00 s	67.9%	0 s	▼juce::Graphics::drawRect(juce::Rectangle<float>, float) const BouncingSquares
991.00 ms	67.3%	4.00 ms	▼juce::RenderingHelpers::SavedStateBase<juce::OpenGLRendering::SavedState>::fillRectList(juce::RectangleList<float> co
761.00 ms	51.7%	2.00 ms	▼juce::RenderingHelpers::SavedStateBase<juce::OpenGLRendering::SavedState>::fillShape(juce::ReferenceCountedObjec
737.00 ms	50.1%	1.00 ms	▼juce::RenderingHelpers::ClipRegions<juce::OpenGLRendering::SavedState>::EdgeTableRegion::fillAllWithColour(juce::C
730.00 ms	49.6%	133.00 ms	▼void juce::EdgeTable::iterate<juce::OpenGLRendering::StateHelpers::EdgeTableRenderer<juce::OpenGLRendering::St
597.00 ms	40.5%	131.00 ms	▼juce::OpenGLRendering::StateHelpers::ShaderQuadQueue::add(int, int, int, int, juce::PixelARGB) BouncingSquares
424.00 ms	28.8%	3.00 ms	▼glBufferSubData_Exec GLEngine
413.00 ms	28.0%	6.00 ms	▶glrWriteBufferData AMDRadeonX4000GLDriver

Revisiting BouncingSquares [13]

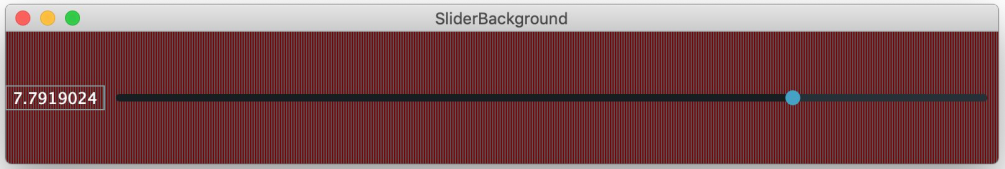
```
OpenGLContext glContext;


MainComponent()
{
    glContext.attachTo (*this);
```

25% CPU (down from 56%)



Revisiting SliderBackground [14]

```
OpenGLContext glContext;


MainComponent()
{
    glContext.attachTo (*this);
```



8% CPU (down from 18%)

OpenGL for JUCE components

Potentially an easy win!

- Reducing everything to scanlines is more work than necessary (OpenGL in JUCE will get even better when we fix this)
- Very useful for Android phones (high number of pixels, no CoreGraphics)
- Extremely fast for images

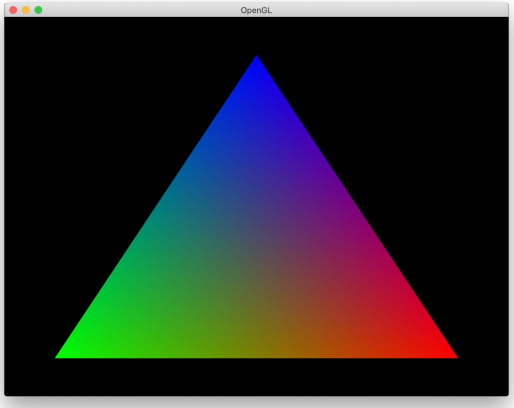
Drawbacks:

- Sibling OpenGLContexts cannot overlap
- Using more than one OpenGLContext can slow things down (will be fixed...)
- Reliant upon third-party graphics drivers
- Threading

Raw OpenGL [15]

Quite a lot of boilerplate

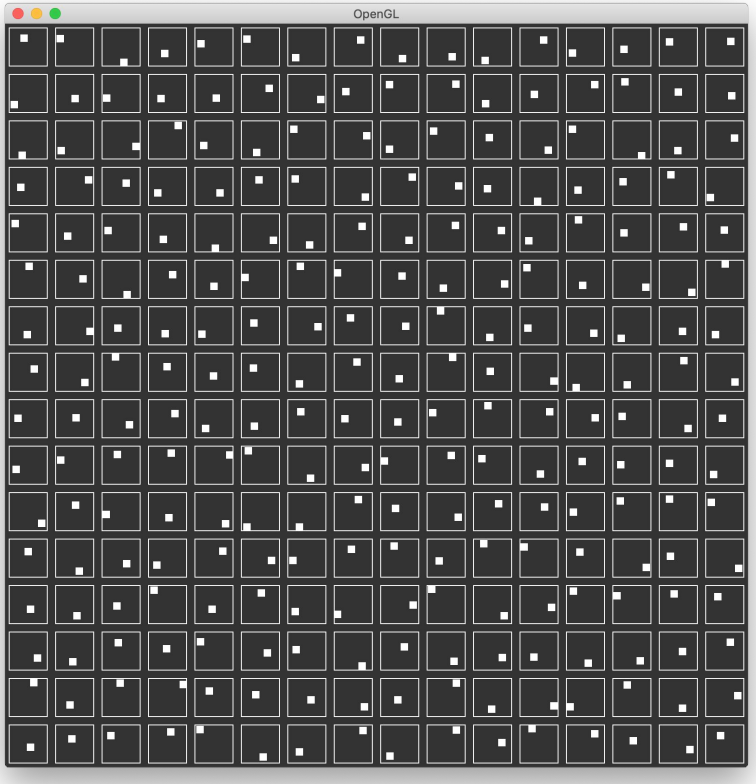
- Define some buffers of data
- Compile and link shaders into a program
- Bind attributes to shader program variables
- Set attributes



Raw OpenGL BouncingSquares [16]

Pull in the BouncingSquares code, but this time each component returns a list of vertices to pass the the graphics card

Down to 12% CPU (down from 25% using OpenGL indirectly) on the OpenGL thread (not the message thread)



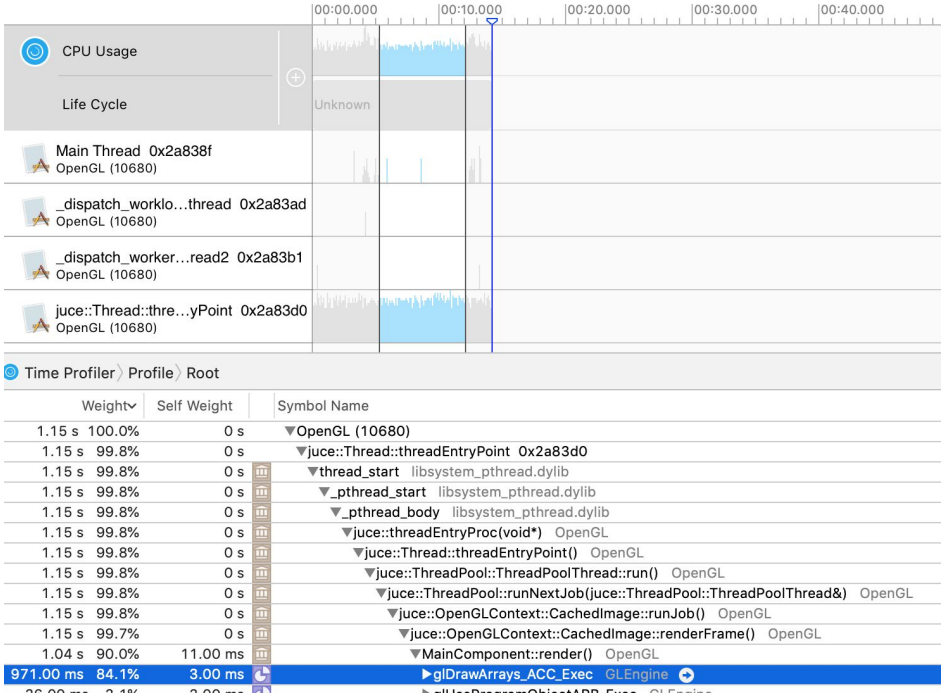
Can we go faster? [17]

Reduce the number of calls to glDrawArrays calls by combining all the vertices into one big list

- Down to 8% CPU

Use glDrawElements to reduce the number of duplicated vertices in the list

- Probably another 2 or 3%



DemoRunner custom shaders [18]

Could also do everything in a shader...

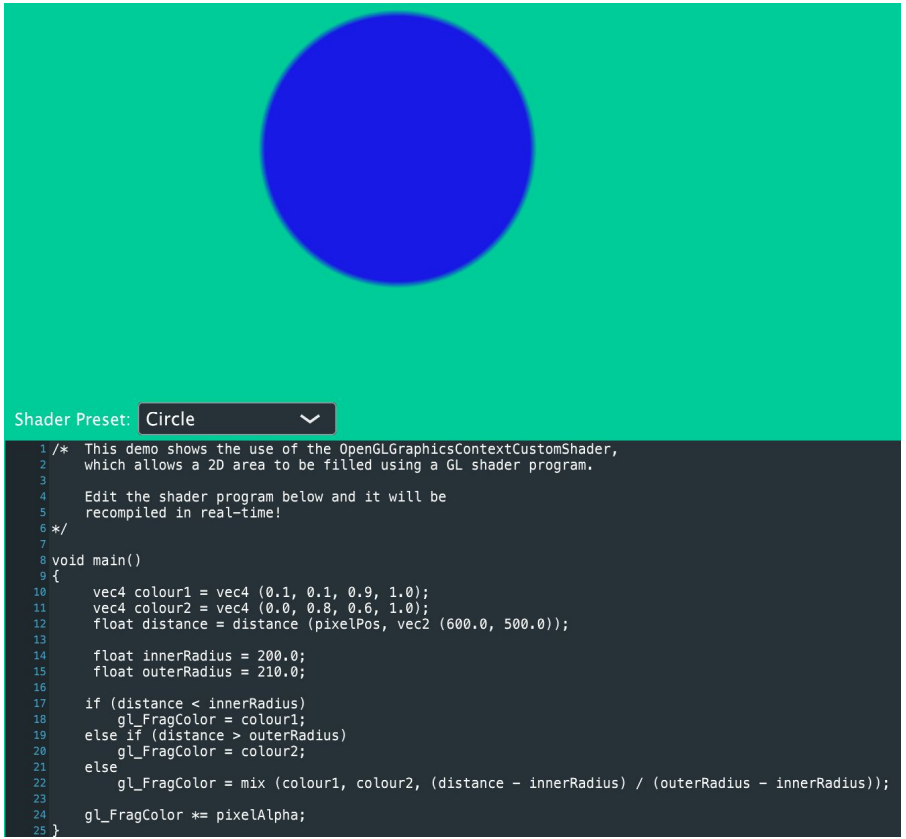
Much nicer JUCE wrappers are available for custom shaders

JUCE/examples/DemoRunner/Builds/*

Compile

Navigate to GUI -> OpenGLDemo2D

Have a play with some GL



OpenGL successors

Apple has announced they will be moving away from OpenGL

Two newer, lower level alternatives

Metal (2):

- MacOS and iOS

Vulkan:

- Linux, Windows, Android, (macOS and iOS via MoltenVK)

Both have lower overhead driver access, configurable parallel command queues, precompiled binary shaders.

Summary

- Profile in Release to find slow points
- Look at what's being repainted
- Make things opaque
- Use floating point coordinates
- Don't thrash the message queue
- Only redraw the minimum required
- … unless it's a very complex region or requires many individual draw calls or requires a lot of clipping
- Turn clipping off if you can
- Draw simpler shapes: Multiple rectangles > Path/object
- Reduce number of draw calls: RectangleList > Multiple rectangles
- Buffer complex drawings to images
- Use prebuilt text layouts
- Use OpenGL