An Intro To Graphics In JUCE

Nicole Alassandro

Components

Examples / 1 - Components

Component

- A drawable widget on screen that can accept user input, hold properties, etc.
- Has size, position, visibility, enablement information
- Can accept mouse or keyboard events
- Can have child components to create complex user interface hierarchies

Included Types

- JUCE Drawables
- Widgets
  - Sliders, Buttons, Labels, etc.
- Data Views
  - TreeView, ListView, etc.
- Containers
  - Viewport, etc.
- Windows

Example 1 / 1

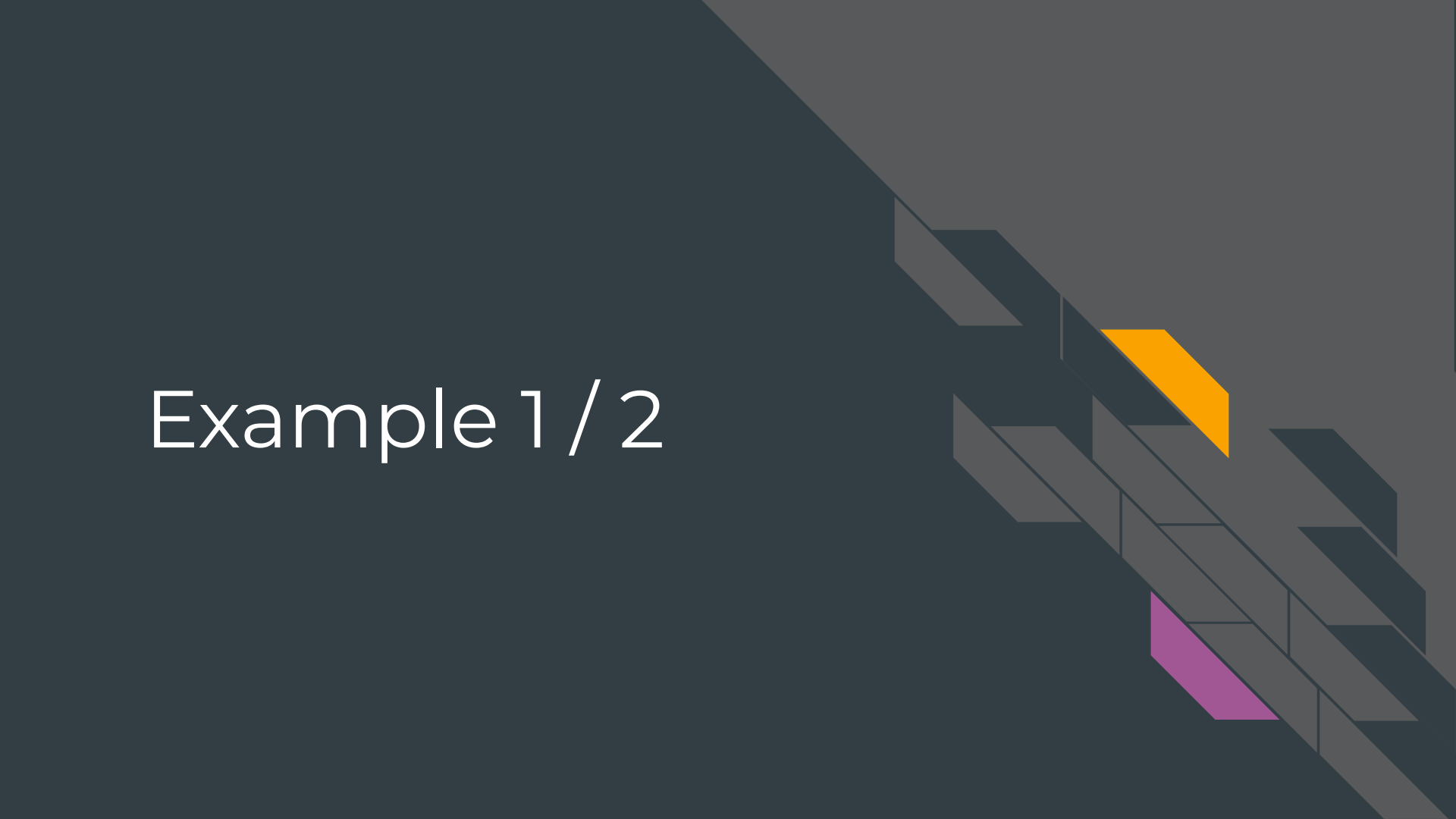
Component Painting

- Each Component is allotted space to draw to the screen according to its bounds
- A Component and the Components in its parent hierarchy must be visible on screen to be drawn
- Children draw over their parents, but Components can draw over their own children as well

Repainting Components

- Component::repaint() triggers a message to be sent to the OS indicating that a region of the screen is “dirty”
- The entire Component can be repainted, or just sections of it at a time
- If too many repaint calls are made the OS may de-prioritize them and throw some away
    - FPS drop, if repainting regularly

Example 1 / 2



Component Hierarchies

- Components may have many children, but only a single parent at a time
- Components normally cannot draw outside of their parents bounds
  - Components normally cannot draw outside their own bounds as well
- If a Component is deleted, it automatically removes itself from its parent (if it has one)

Example 1 / 3

Graphics

Examples / 2 - Graphics

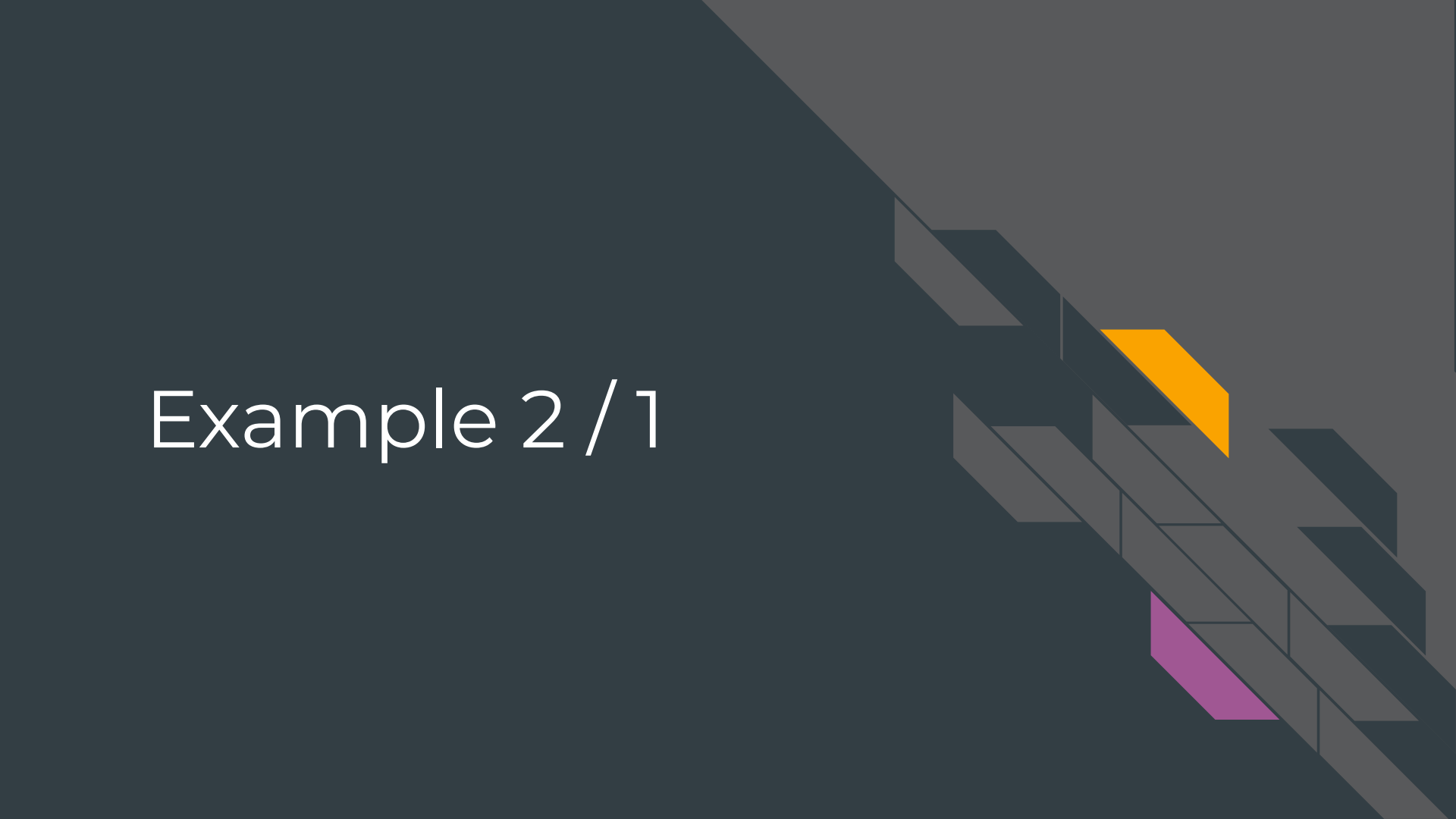
Traversing The Hierarchy

- The highest level Component is one placed directly on the desktop
- The paint loop:
  - Component::paint() is called
  - The child hierarchy is traversed and painted
  - After a Component paints itself and its children, Component::paintOverChildren() is called

The Graphics Context

- Manages drawing graphics into an allotted buffer of pixel data
  - This data is usually allocated and managed by JUCE when drawing Components
- Mostly deals in vector graphics, *usually* no raster graphics
- Provides the interface over a juce::LowLevelGraphicsContext

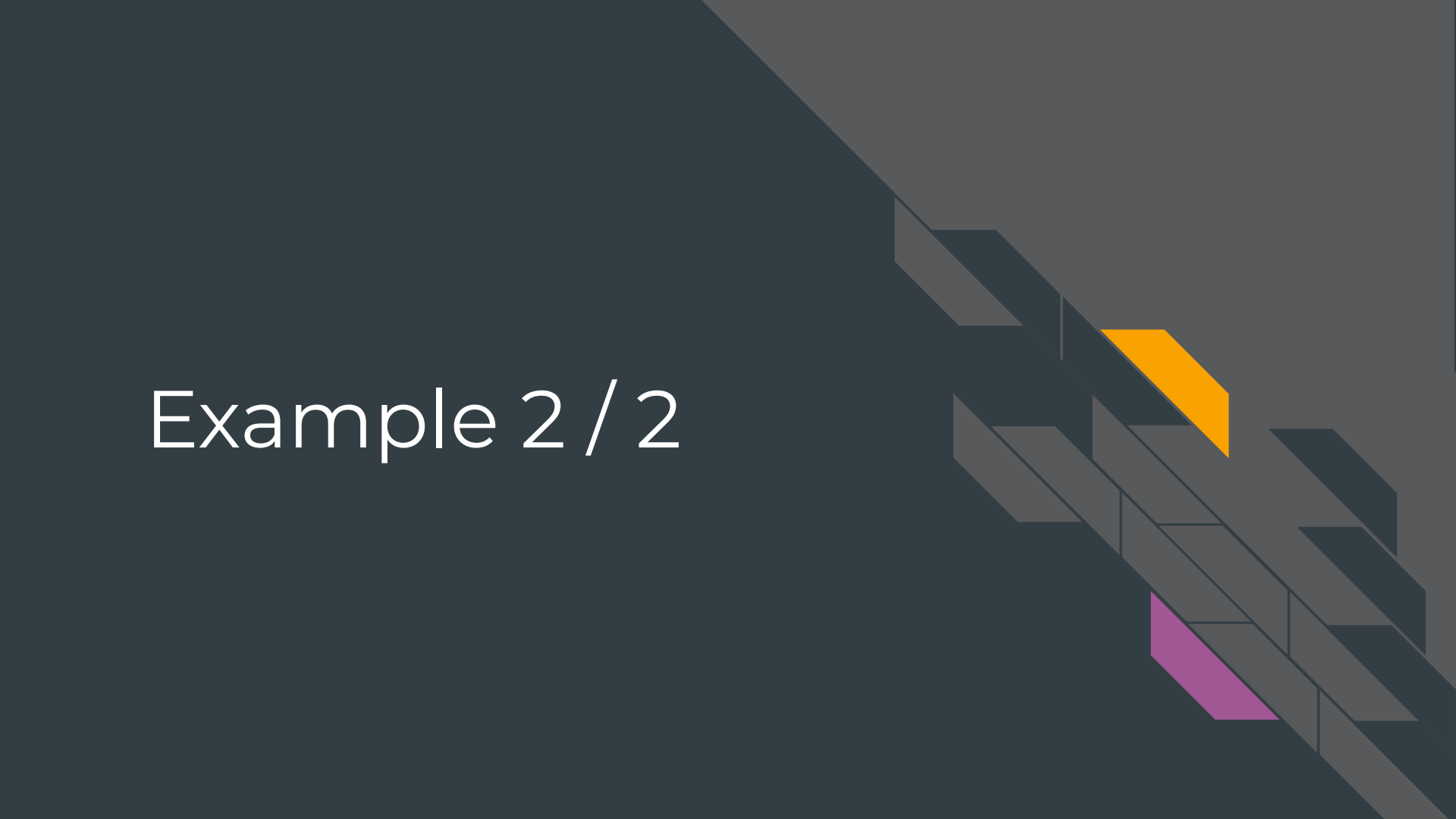
Example 2 / 1



Drawing Shapes

- The Graphics class provides many helpful methods for drawing simple objects
- Several methods provide a “fill” shape and an “outline” shape
- Graphics class does not manage connecting multiple shapes, lines, points, etc. together

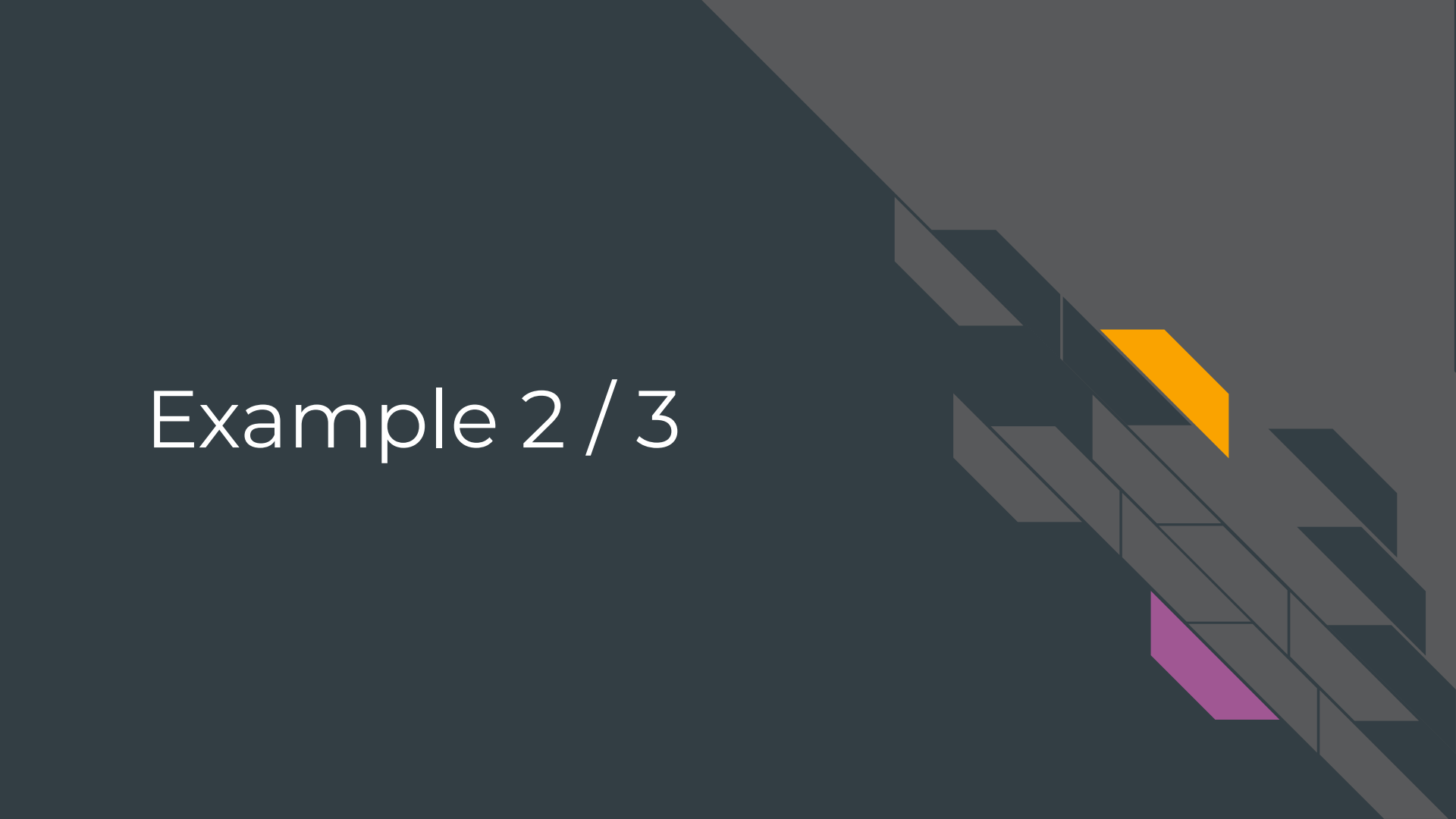
Example 2 / 2



Complex Paths

- JUCE's Path class provides a way to manage vector shapes as a collection of points
  - Provides lines, bezier curves, and even some predefined shapes
- Can be allocated and stored outside of the Graphics context
- Provides a PathStrokeType class for defining outline drawing properties

Example 2 / 3



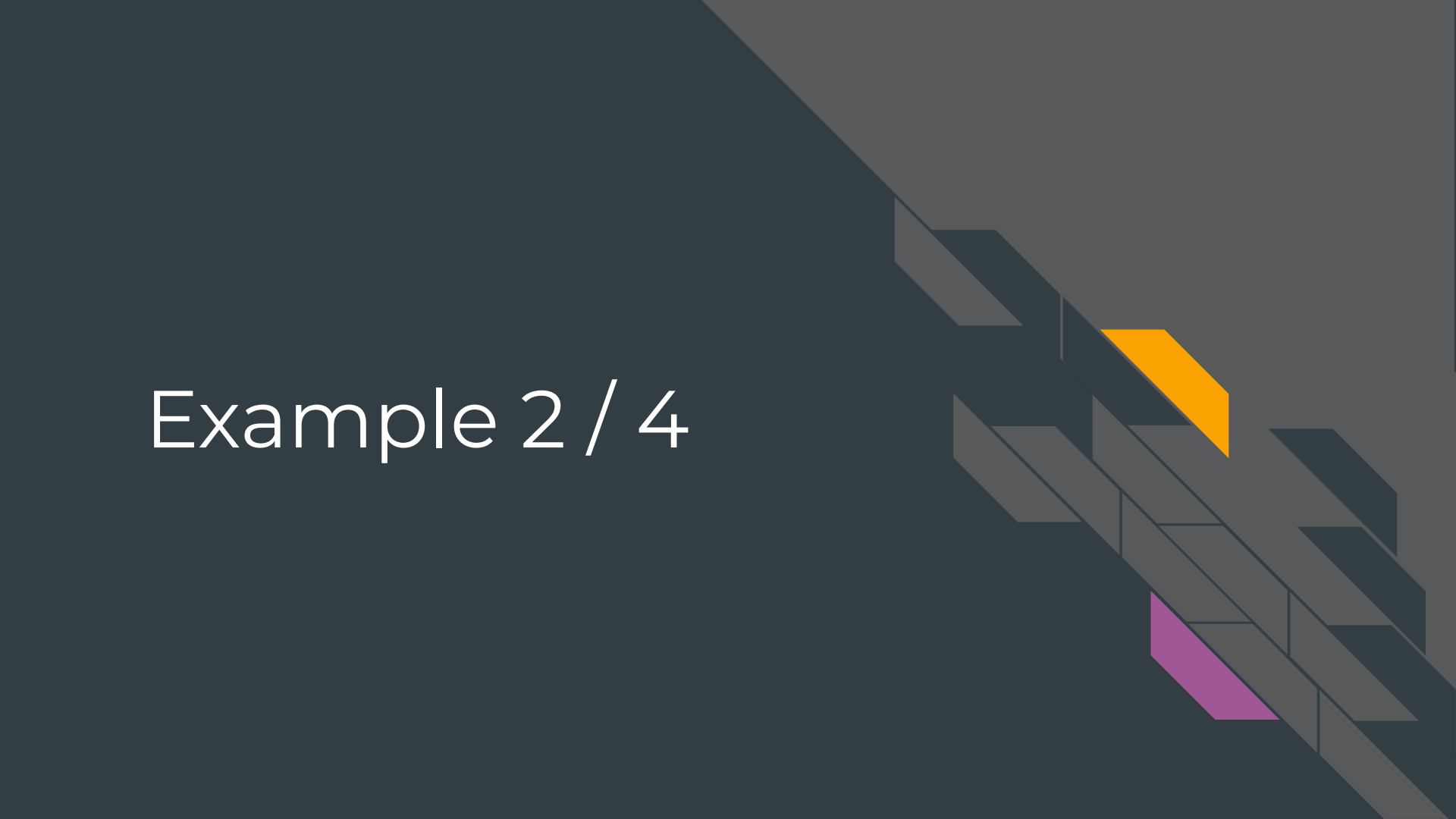
Fill Types

- Colour
- ColourGradient
  - Maps a list of colours across two points
  - Can be linear or radial
- Image (tiled)
- Graphics::setFillType(), Graphics::setColour(), Graphics::setGradientFill(), Graphics::setTiledImageFill()

Opacity

- Graphics::setOpacity()
  - Only applies to the current fill
- Graphics::beginTransparencyLayer(), Graphics::endTransparencyLayer()
  - Allows an entire section of Graphics calls to have an opacity applied

Example 2 / 4



Text

- The Graphics class can draw text when given String, bounds, and Justification arguments
- Characters are converted to Paths, laid out, and then rasterized
- The area you’re drawing into must be long enough to fit the text, or tall enough to fit the text in a multi-line format

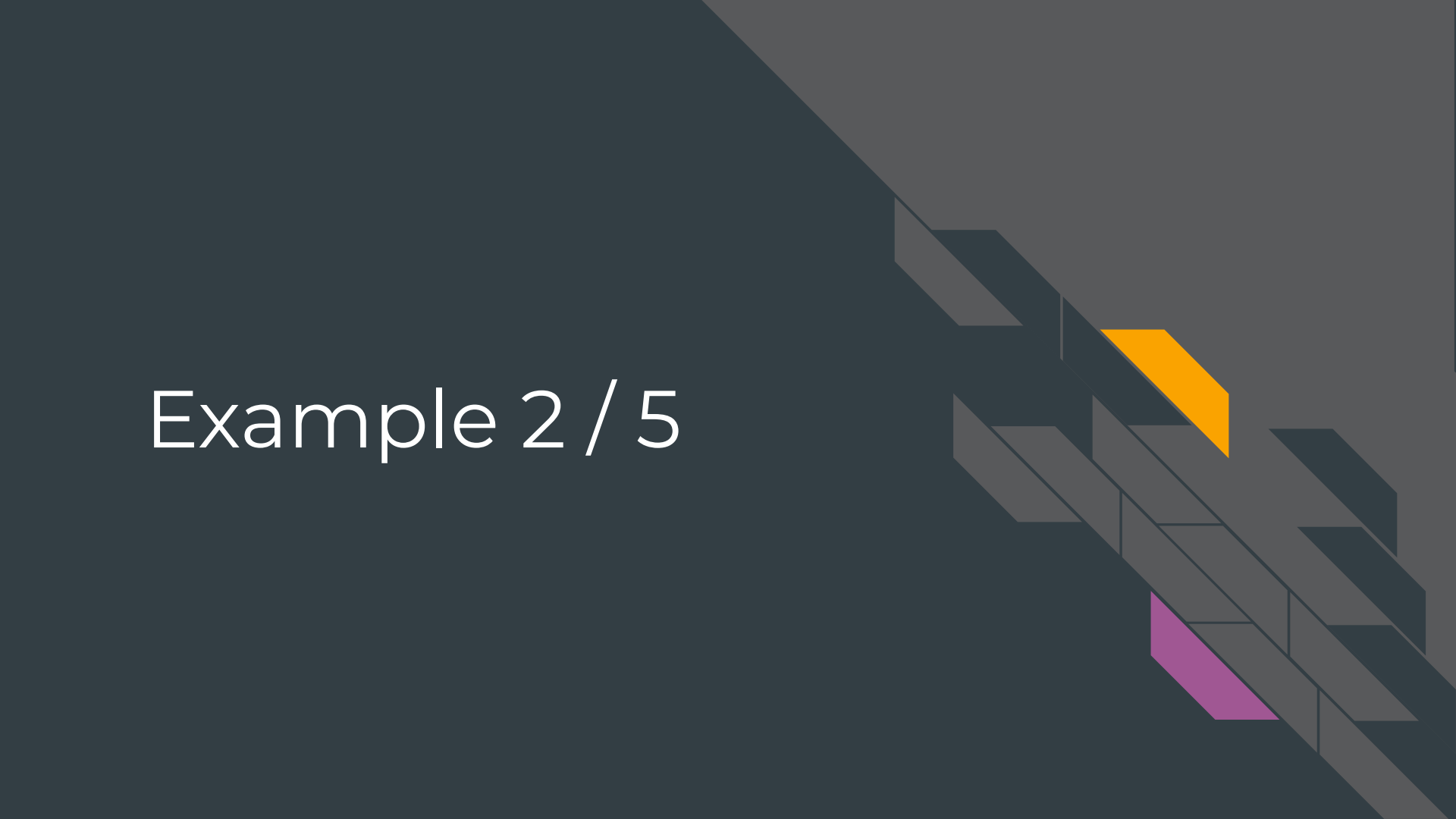
JUCE’s Font Class

- Provides access to the available fonts on the user’s system
  - juce::Typeface
  - Custom fonts with juce::Typeface::createSystemTypefaceFor()
- Manages size, kerning, typeface, typeface-style, and other style flags such as bold, italicised, underlined, etc.

Laying Out Characters

- juce::AttributedString, juce::TextLayout
- Font::getStringWidth()
- juce::GlyphArrangement

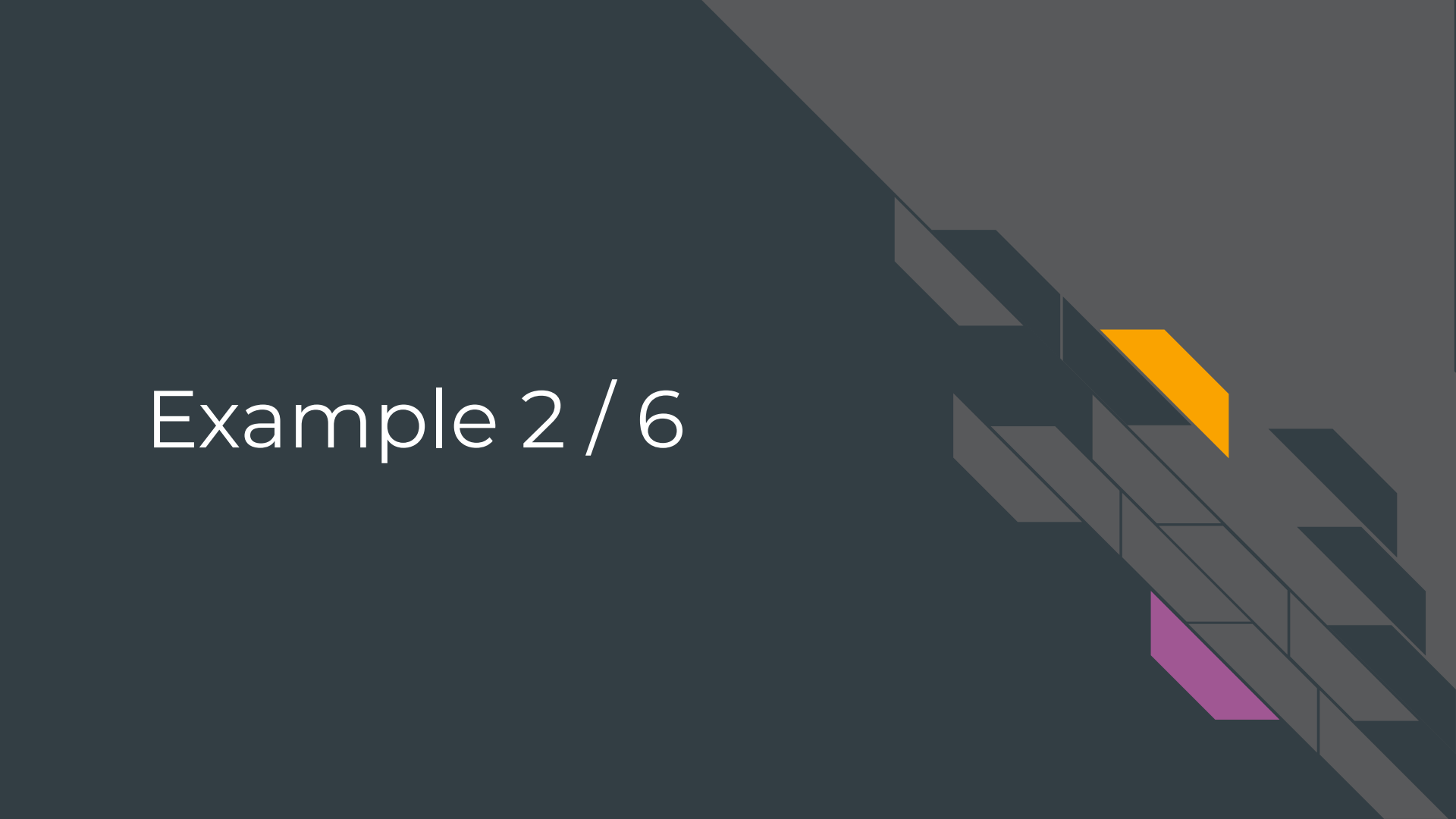
Example 2 / 5



Affine Transforms

- A 3x3 matrix that can be applied to the Graphics context or to individual Graphics calls
- Paths, Images, geometric types, etc. can make use of the AffineTransform class
- Transforms can be stacked

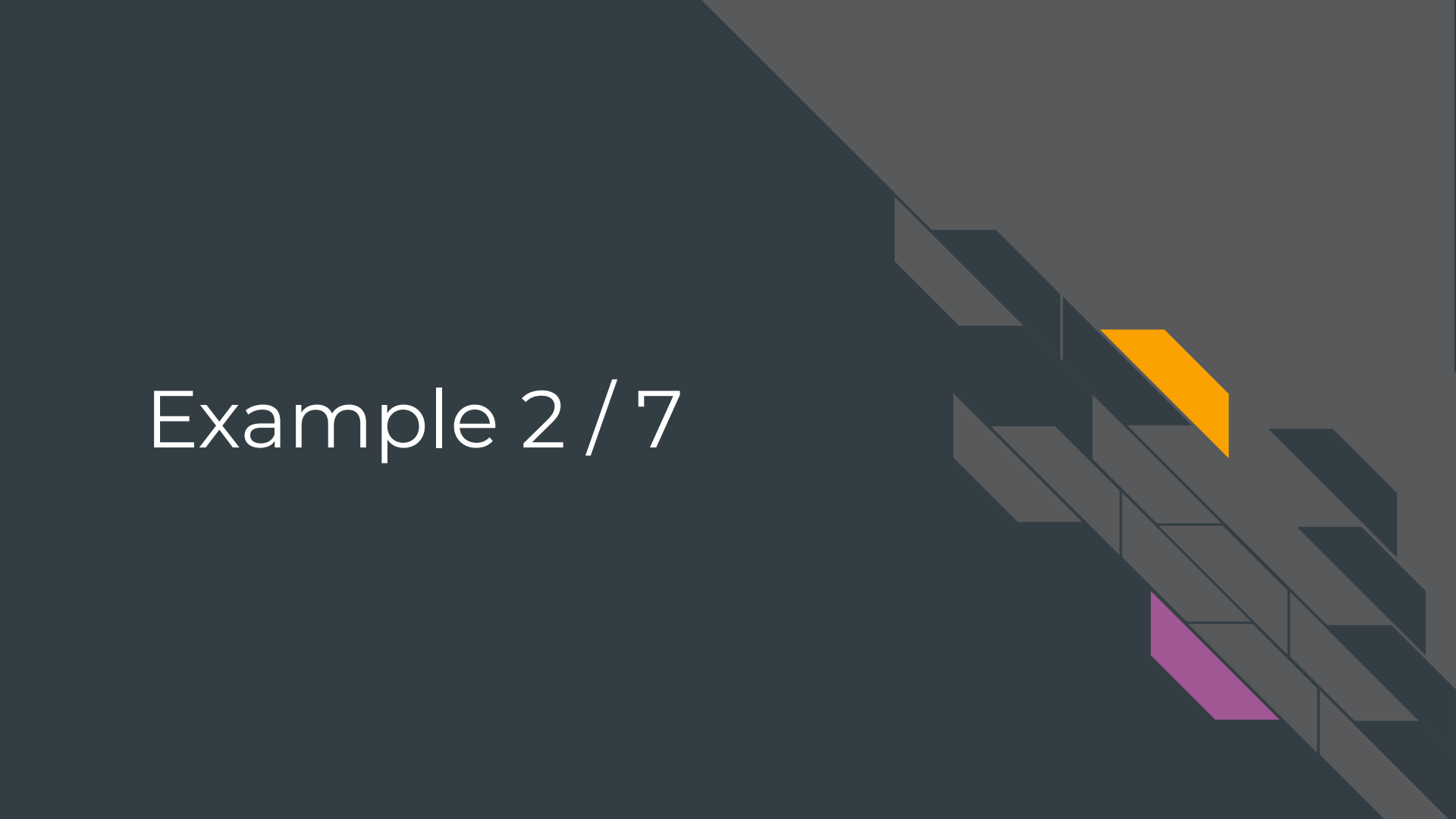
Example 2 / 6



Clip Regions

- By default, the Graphics context can only paint inside of the area it was given when created
- Can only reduce clip region, not increase
- Graphics::reduceClipRegion()
  - Rectangle, RectangleList
  - Path
  - Image

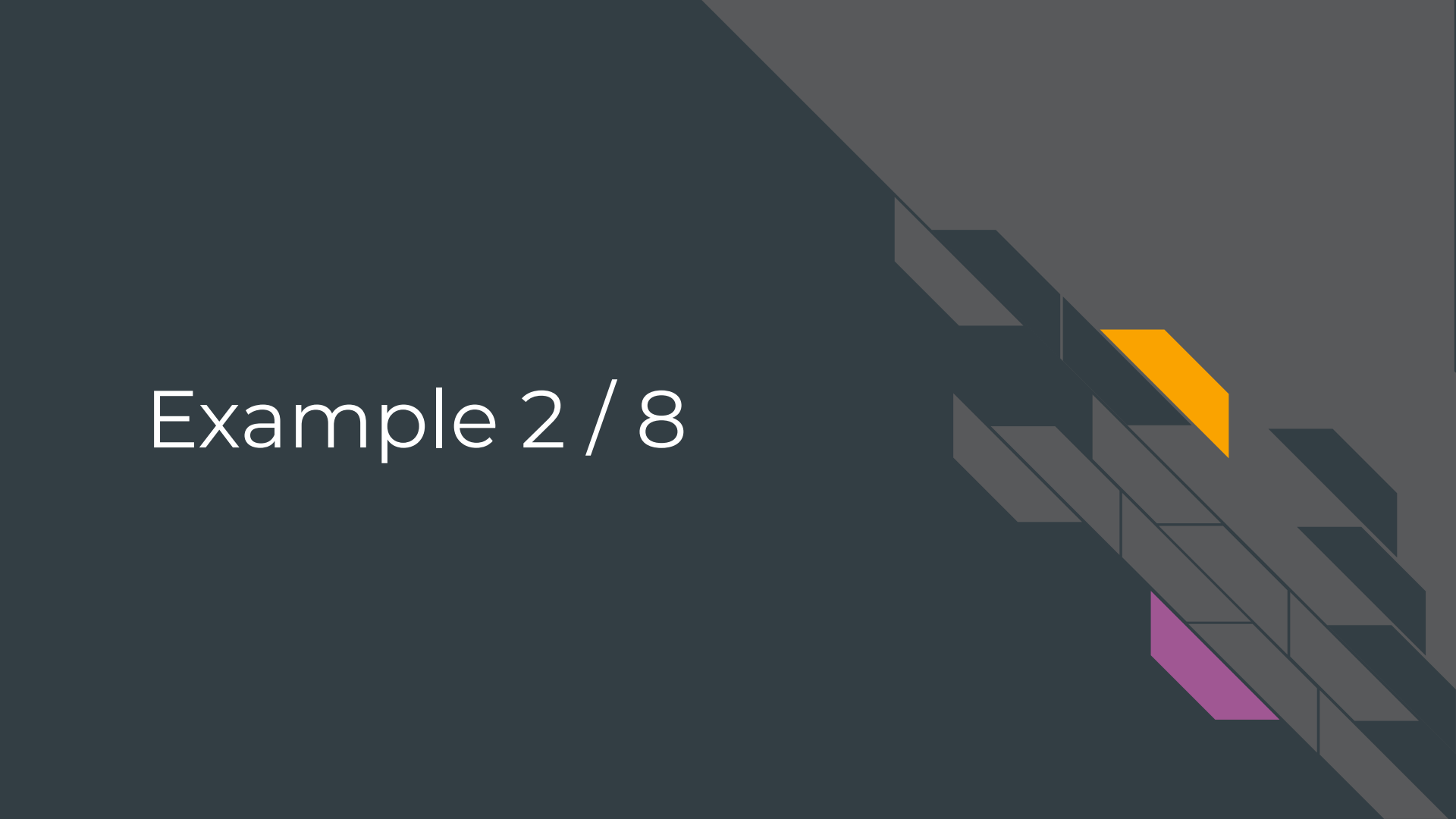
Example 2 / 7



State Stack

- The Graphics class maintains a stack of state structures
- The stack can be pushed/popped to manage complex state like clip regions, transformations, etc.
- Graphics::ScopedSaveState

Example 2 / 8



Images

Examples / 3 - Images

Image Basics

- JUCE Image objects simply hold a pointer to some heap-allocated pixel data
  - Image objects themselves are cheap, can be copied around easily
  - Image::createCopy() creates a new image with new data
- When no Image objects exist that point to a block of pixel data that data gets deleted

Image Basics

- Images can point to “nothing”, e.g. Image::isValid()
- Images provide interfaces to working with raw pixel data, unlike the Graphics class
  - Image::BitmapData
- Has a properties map just like Components do
  - Is shared across all Images that refer to the same underlying data

Image Formats

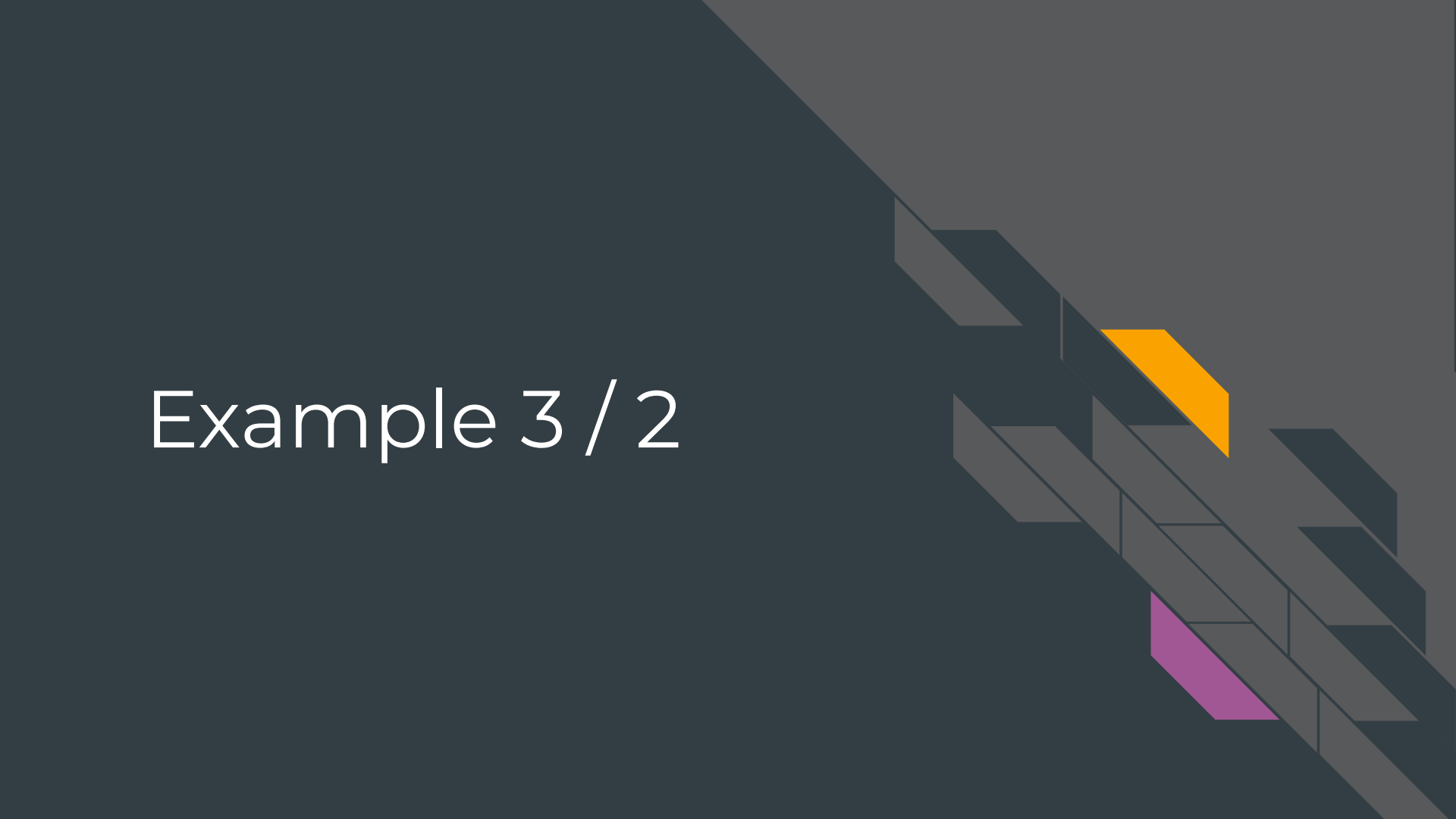
- Single channel (8-bit)
  - Useful for masks
  - No color
- RGB (24-bit)
  - Cannot use transparency
- ARGB (32-bit)
  - Supports transparency
  - macOS will only return this type

Example 3 / 1

Image Buffers

- A Graphics context can be instantied give an Image object
- Images can be used as a back buffers, allowing complex visuals to be updated only when needed
- Useful in multi-threaded situations

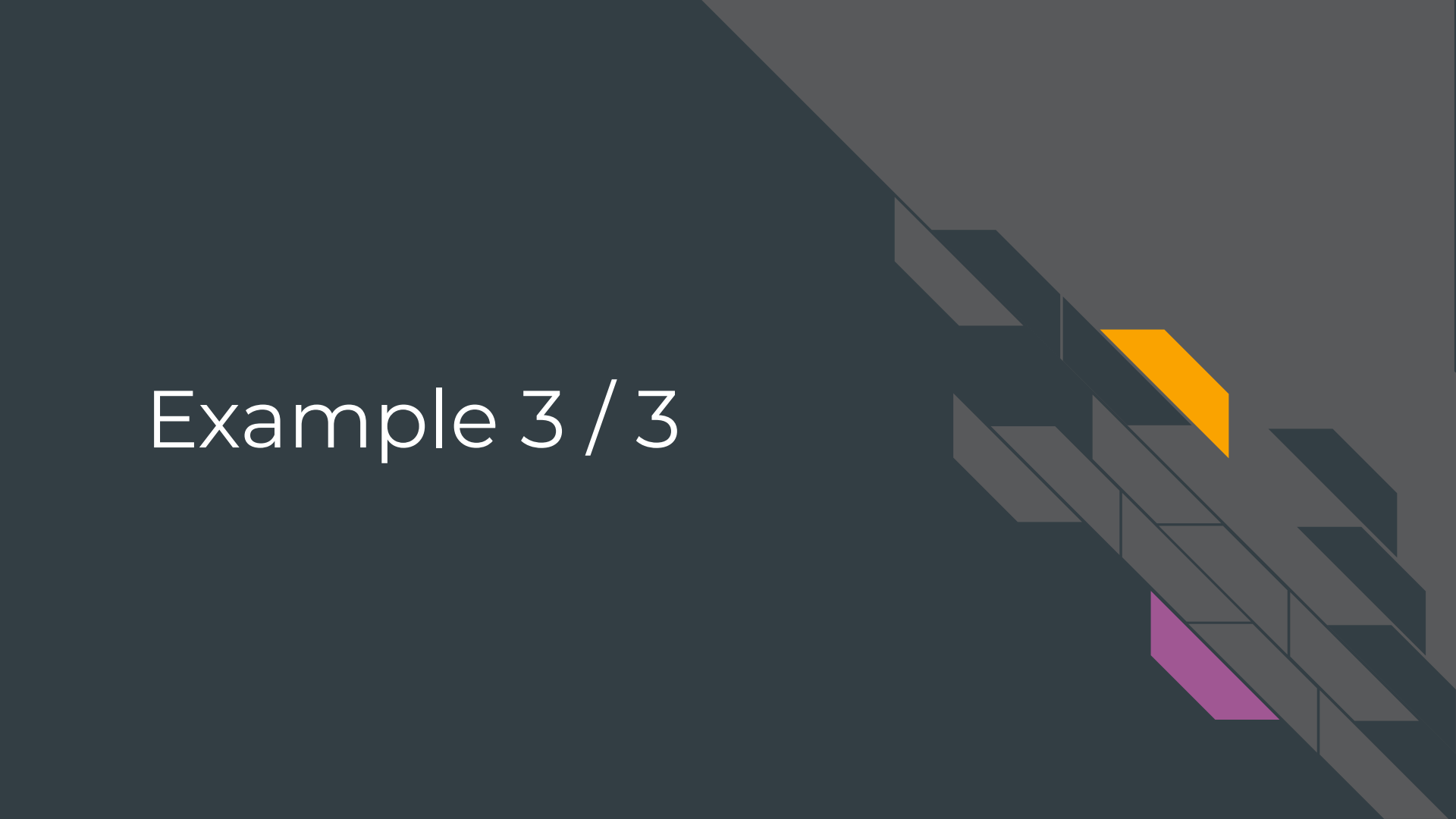
Example 3 / 2



Caching

- Components provide mechanisms for caching their contents
  - juce::Component::setBufferedToImage()
  - Caches will only be updated if the Component directly has repaint called
- juce::CachedComponentImage for custom caches
- Only useful when painting overhead costs more than image data saving & lookup

Example 3 / 3



LookAndFeel

Examples / 4 - LookAndFeel

LookAndFeel API

- JUCE provides UI customisation through its LookAndFeel class, which is comprised almost entirely of virtual methods
  - Derives different LookAndFeelMethods structs to compose its API
- ColourId enums provide unique identifiers to associate colours for a given Component or LookAndFeel

Example 4 / 1

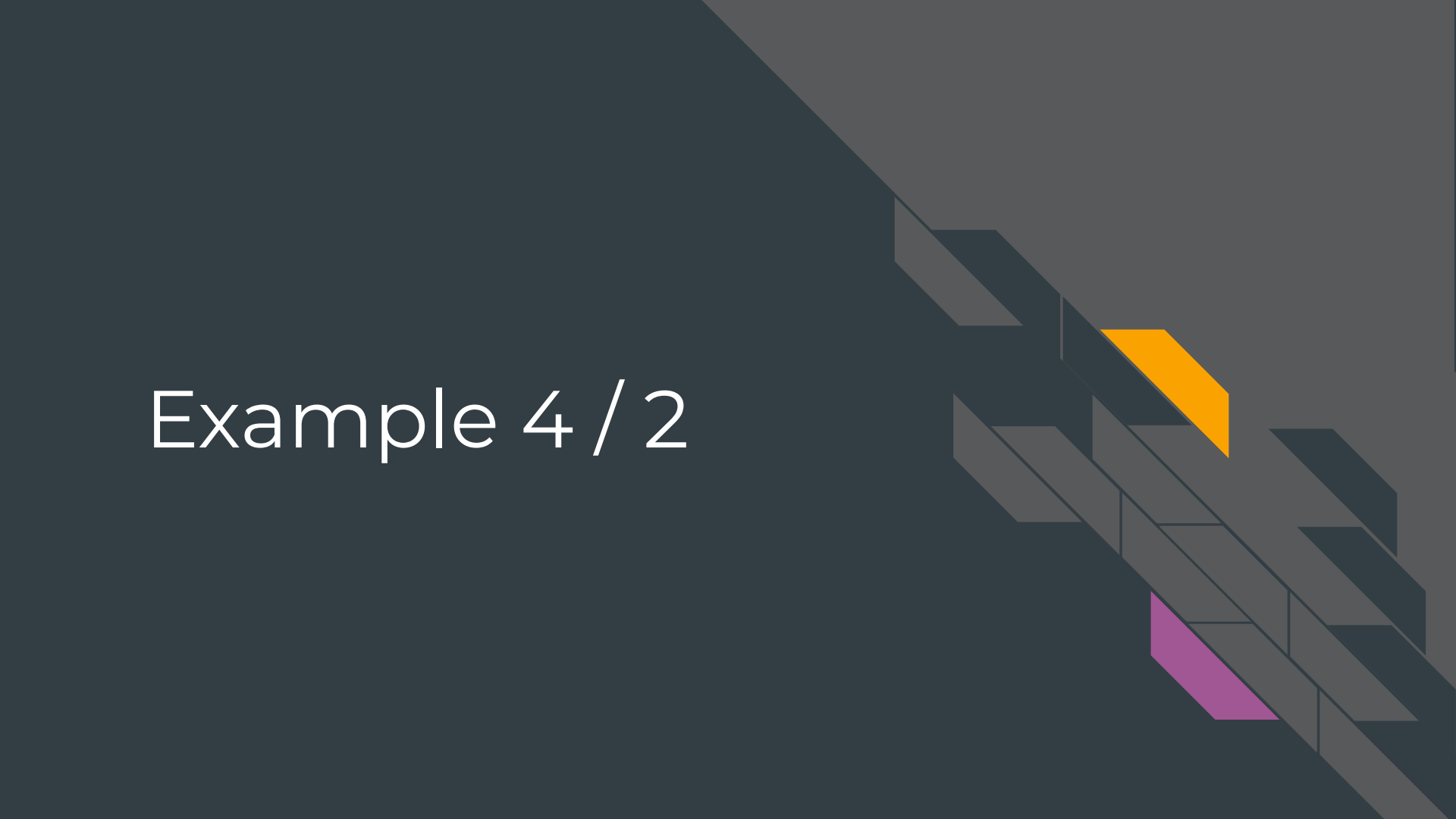
Custom LookAndFeels

- To create a custom LookAndFeel, you must derive one of the LookAndFeel subclasses
  - LookAndFeel_V1 ... LookAndFeel_V4
  - LookAndFeel can be used but you must override all the pure virtual methods yourself
- Can easily provide custom fonts and even behaviour using your own LookAndFeel

Custom Component L&F

- Subclasses of provided widget types (Slider, Label, etc.) should use Component::getProperties()
  - Provides access of custom data members without having to type-cast
- Entirely custom Components require a custom LookAndFeel subclass
  - Custom LookAndFeelMethods struct
  - Will require dynamic casting

Example 4 / 2



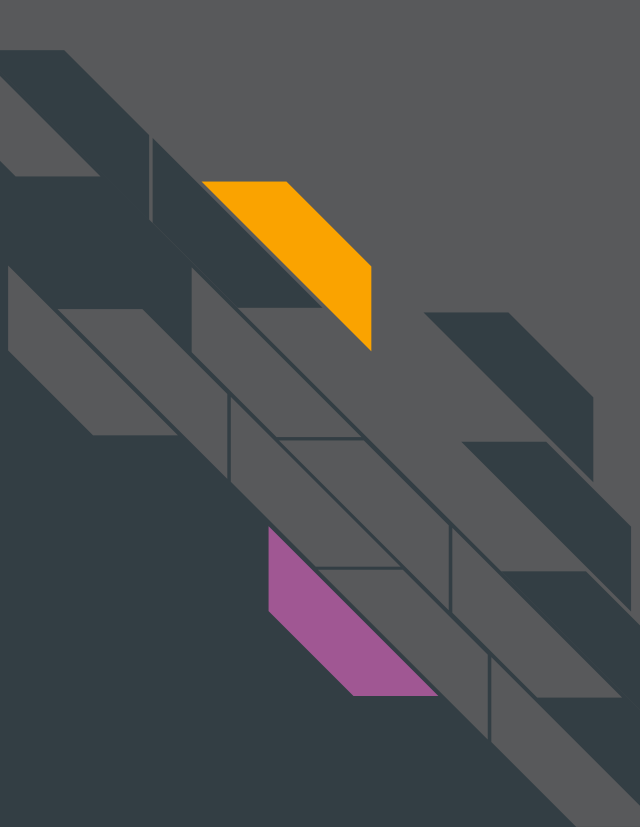
Layout

Examples / 5 - Layout

The “Simple” Way

- Components can be laid out by hand
  - Absolute positioning
  - Relative positioning
- Component::resized()
- Usually hard to maintain and read
- Makes resizable applications more difficult
- Strongly-tied to child Components
  - child1.setBounds(...); child2.setBounds(...)

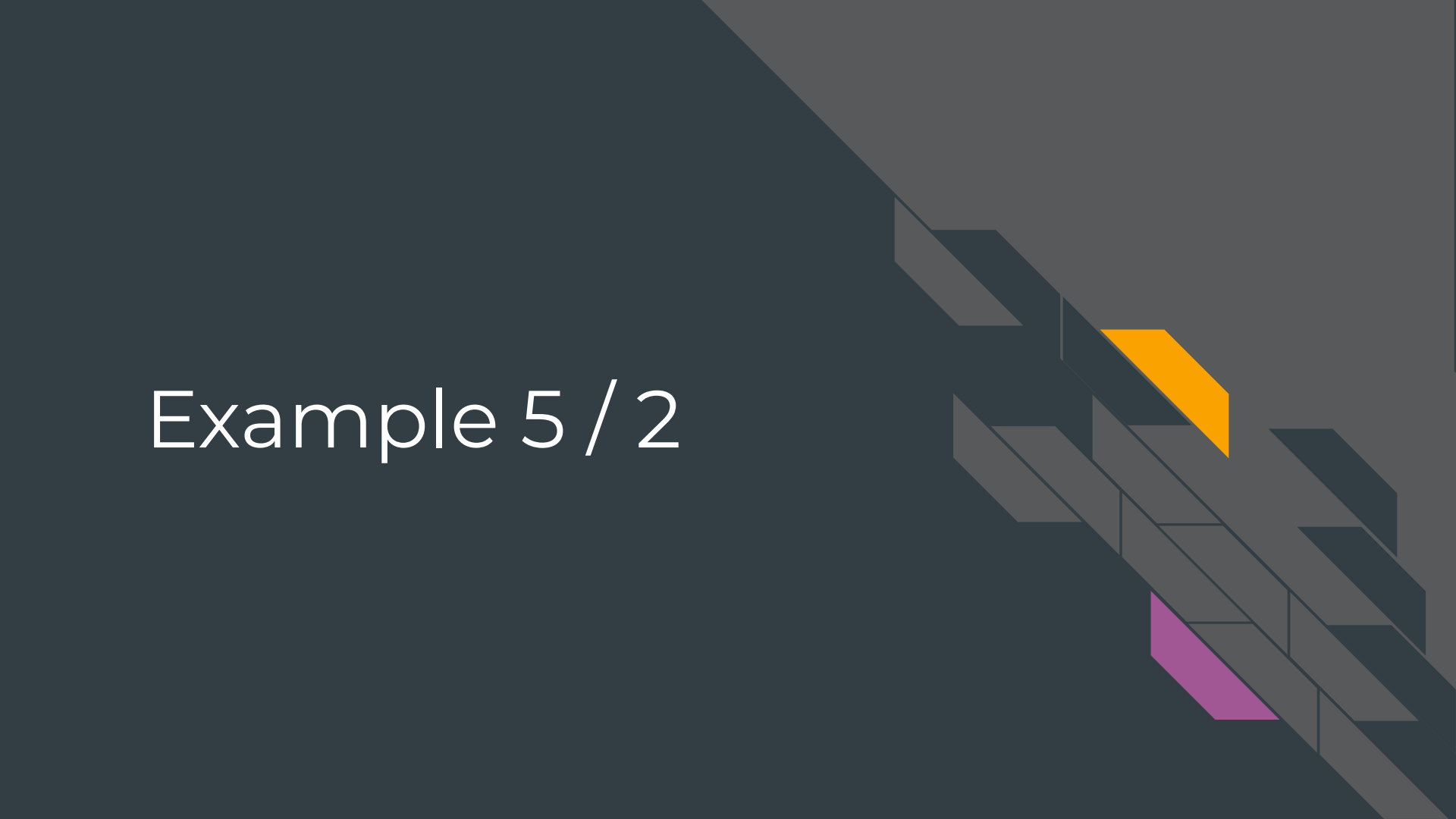
Example 5 / 1



Rectangle Slicing

- Easy to read
- Slightly easier to create layouts
- Still requires hand calculating positions absolutely or relatively
- Most useful with low number of components
- Not very useful when components may be “floating” around in the layout
    - Skeuomorphic UIs

Example 5 / 2



FlexBox

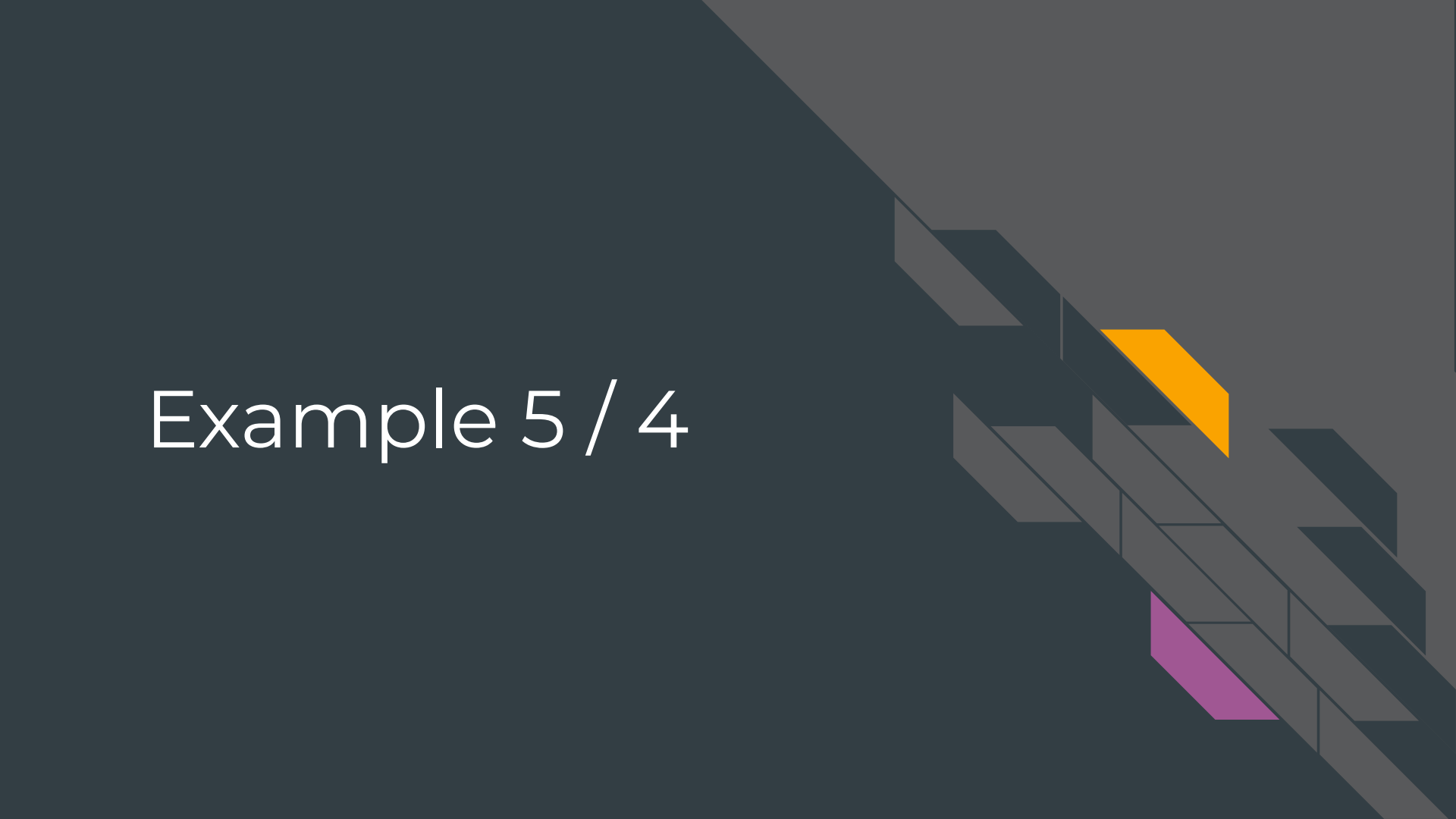
- Originally came from web development (CSS3)
- Allows dynamically sized components to be laid out across a given axis
- Much easier to create reactive layouts for resizable applications
- Requires little boilerplate
- Cannot directly handle 2D grid layouts

Example 5 / 3

Grid

- Implementation of CSS3 Grid
- Handles aligning Components according to a 2D grid
- Not necessarily required for a grid layout
  - FlexBox managing FlexBoxes
  - Rectangle-sliced grids
- Requires more boilerplate than juce::FlexBox

Example 5 / 4



Performance Tips

Graphics-Dependent Data

- If there are complex structures that are only used for drawing, you can perform the calculation in your Component's paint method
- Repaint calls can be thrown away by the OS, so calculating the data before calling repaint may cause work that the user isn't seeing anyway

Always Use Components

- Any “piece” of a user interface that accepts user interaction should be its own dedicated Component
- Use Components for pieces that may move around frequently over top of other Components
  - JUCE will handle repainting the old and new positions of the element, otherwise there will be fragmenting

Draw As Little As Needed

- Ensure proper bounds and clip regions
- Cache complex graphics that require many overlapped layers or calculations
- Effects like drop-shadows, glows, etc. should be buffered as they are expensive
- Use opaque Components wherever applicable
- Avoid frequent/unnecessary repaint calls, especially over large areas

Questions?