Using Candera TextEngine to draw Text

Description

The simplest way to draw text with Candera is using a TextNode2D, a 3D text widget from CGI Studio Widget Library, or the 2D TextBrush effect. Refer to:

- Candera::TextNode2D
- Candera::TextBrush

To learn about how to use Candera::TextNode2D, please refer to the following section:

- Using Candera::TextNode2D

To learn about details, how to use the Candera TextEngine interface directly, refer to the following chapters.

Using **Candera::TextNode2D**

Overview

TextNode2D represents a RenderNode specialized for fast text rendering.

Text and style have to be provided. In addition, it requires a BitmapBrush effect for rendering and a Candera::TextNodeRenderer object used for generating the images, which will be then rendered by the associated BitmapBrush effect.

There are four Candera::TextNodeRenderer implementations available, each corresponding to one CacheType option from the existing Candera::TextBrush rendering concept (Bitmap, Surface, Glyph and Glyph Cache).

Text layouting is supported by attaching a Candera::TextNode2DLayouter to the node. It is responsible for arranging and truncating the associated text of a Candera::TextNode2D. The default layouter implements both horizontal and vertical truncation.

Renderer

Candera::TextNodeRenderer object is used for generating the correct images from provided text.

The level cache strategies which apply for a Candera::TextNode2D are described below:

- Bitmap: very large memory footprint, very large update times, small render time
- Surface: large video memory footprint, large update times, small render time
- Glyph (former NoCache): small memory footprint, small update times, very large render time.
- GlyphCache: small memory footprint, small update times, medium render time
- GlyphAtlas: small (but global) video memory footprint, small update times, small render time.

> Choose the appropriate cache for a balance in performance. It is recommended to use either Bitmap or Surface rendering type when static texts are used, while for dynamic texts, GlyphCache is recommended.

Associated Effect

Please consider that only BitmapBrush type effects apply to a Candera::TextNode2D. In case other effects are used nothing will be rendered. The text appearance can be altered by the type of effect chosen, either by modifying the in-place effects (color, mask, HSL transformations, etc.) or by changing blending effects.

List of BitmapBrush effects which can be used are described below:

- Candera::BitmapBrushBlend: Outputs a bitmap image, and blend it with the store buffer.
- Candera::BitmapBrushColorBlend: Output a bitmap image, modulate the color, and blend it with the store buffer.
- Candera::BitmapBrushColorMaskBlend: Output a bitmap image, modulate the color, apply an alpha mask, and blend it with the store buffer.
- Candera::BitmapBrushHslBlend: Output a bitmap image, apply a HSL transformation, and blend it with the store buffer.
- Candera::BitmapBrushMaskBlend: Output a bitmap image, apply an alpha mask, and blend it with the store buffer.
- Candera::BlurBitmapBrushBlend: Output a blurred and alpha blended bitmap image.
- Candera::MirrorBitmapBrushBlend: Output includes the bitmap image and it's reflection.
- Candera::ShadowBitmapBrushBlend: Output a shadowed or glowing bitmap image, both shadow/glow and image alpha blended.
- Candera::ShearBitmapBrushBlend: Output a sheared image from the original one.

> Not all BitmapBrush effects have the *Color* property available. Default color for a TextNode2D is white.

Layouter

Candera::TextNode2DLayouter is responsible for arranging and truncating the text associated to a Candera::TextNode2D. Candera::DefaultTextNode2DLayouter, derived from Candera::TextNode2DLayouter, is used for text truncation. More details are provided in the following section.

Candera::TextNode2D Guidelines

The minimum necessary steps needed in order to render a TextNode2D are:

- Create a TextNode2D instance
- Set a Text: Text to be rendered.
- Set a Style: Style object to be used for rendering.
- Define a TextNodeRenderer: TextNodeRenderer used for generating the correct images from provided text.
- Attach a BitmapBrush effect: The associated BitmapBrush effect is responsible for rendering the generated images.

Define node and effect

Define a Candera::TextNode2D and the associated BitmapBrush effect:

```
    TextNode2D* m_textNode;
    BitmapBrushColorBlend::SharedPointer m_textNodeEffect;
```

Create node and effect

Create an instance of Candera::TextNode2D class by using Candera::TextNode2D::Create() method:

```
    m_textNode = TextNode2D::Create();
```

In similar way create the BitmapBrush effect and add this effect to the newly created Candera::TextNode2D:

```
    m_textNodeEffect = BitmapBrushColorBlend::Create();
    m_textNode->AddEffect(m_textNodeEffect.GetPointerToSharedInstance());
```

Set Text

Use Candera::TextNode2D::SetText() to specify the text to be rendered:

```
    m_textNode->SetText("TextNode2D");
```

Set Style

Use Candera::TextNode2D::SetText() to specify the style object:

```
    m_textNode->SetText("TextNode2D");
```

Renderer

Create and set a Candera::TextNodeRenderer to Candera::TextNode2D using Candera::TextNode2D::SetTextNodeRenderer() method:

```
    m_textNode->SetTextNodeRenderer(&GlyphCacheTextNodeRenderer::GetInstance());
```

Add to Scene

Finally, add the textNode directly to a Candera::Scene2D or a Candera::Group2D:

```
    m_textNodeGroup->AddChild(m_textNode);
```

Candera::TextNode2D Properties

Text Color

Changing the color of a Candera::TextNode2D is done through the associated BitmapBrush effect. Depending on the effect use, not all BitmapBrush effects expose a color property. In this case, the rendered text will be white.

```
    m_textNodeEffect->GetColorEffect().Color().Set(Color(0.0f, 0.0f, 0.0f, 1.0f));
```

Text Alignment

Text alignment is realized through the layouter properties of the Candera::TextNode2D, meaning that the position inside a layouter, as well as text position inside the Candera::TextNode2D will now be changed by the same Candera::TextNode2DLayouter::SetHorizontalAlignment() and Candera::TextNode2DLayouter::SetHorizontalAlignment() methods the layouter. In case it in necessary to have a different text position than the provided layouting options, it is possible to nest Candera::TextNode2D and set different alignment values for the inner TextNode.

```
    TextNode2DLayouter::SetHorizontalAlignment(*m_textNode, Candera::HLeft);
    TextNode2DLayouter::SetVerticalAlignment(*m_textNode, Candera::VTop);
```

Truncation

Candera::TextNode2D provides only one truncation type - in case truncation is needed, the text gets cut and trailing ellipsis are added to fit the space. Truncation is not supported by default. For this, the Candera::DefaultTextNode2DLayouter needs to be attached to Candera::TextNode2D (see snippet below) and the text will be automatically truncated whenever the size of the text no longer fits in the specified *Layout Size* property of Candera::TextNode2D. Bounding rectangle size is not taken in account when performing text truncations.

```
    m_textNode->SetLayouter(&TextNode2DLayouter::GetDefault());
```

Multiline Support

Multiline layouting is enabled by default for a TextNode2D. This property can be enabled/disabled using Candera::TextNode2DLayouter::SetMultiLineEnabled() method.

Wordwrap Support

Wordwrap is not enabled by default for a TextNode2D. This property can be enabled/disabled using Candera::TextNode2DLayouter::SetWordWrapEnabled() method.

Text Height

Glyph based height computation is still done, but it is not used for laying out. The rectangle in which the text is laid out, if Size is (-1;-1), will be the one measured with old TextBrushConstantHeight option. The TextBrushActualHeight is always used for retrieving the bounding rectangle of the node, but it is also possible to retrieve the layout rectangle of the node too (see information about text length calculation below).

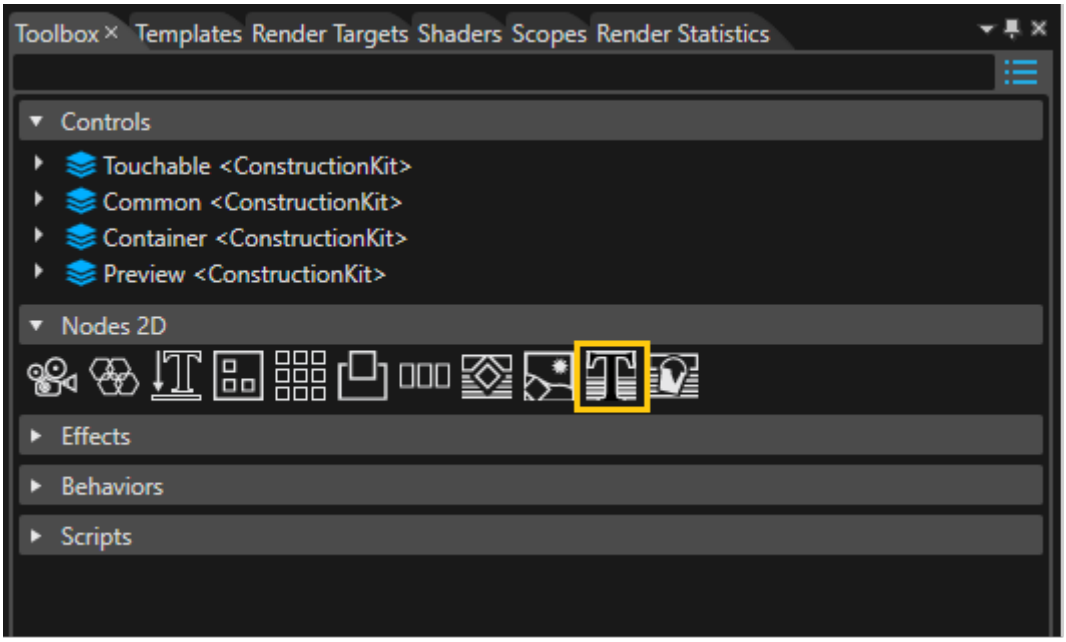
Text Length

The following methods are provided for retrieving the size of the rendered text:

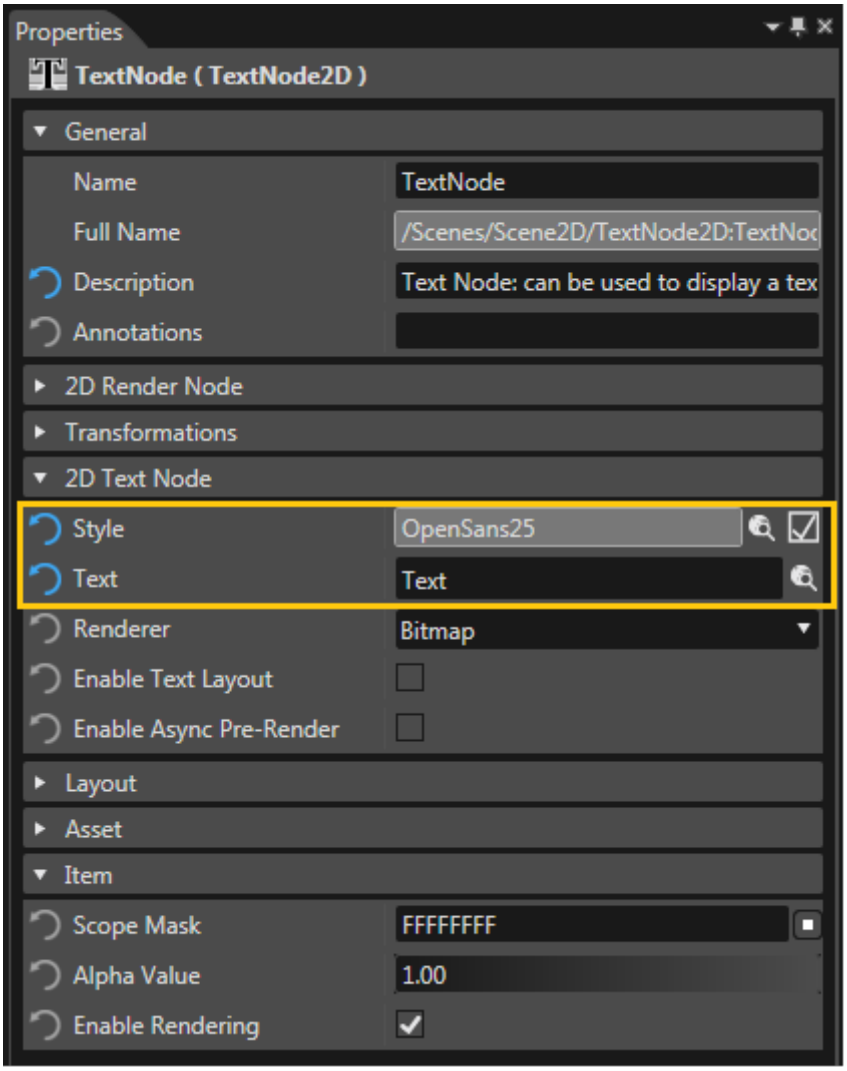
- Getting the layouting text box (measured with constant height, cursor to cursor from left to right):
  - Candera::TextNode2D::GetLayoutTextRectangle()
- Getting the bounding text box (glyph limits vertically and horizontally):
  - Candera::TextNode2D::GetBoundingTextRectangle()
  - Candera::Node2D::GetComputedBoundingRectangle()

Candera::TextNode2D in SceneComposer

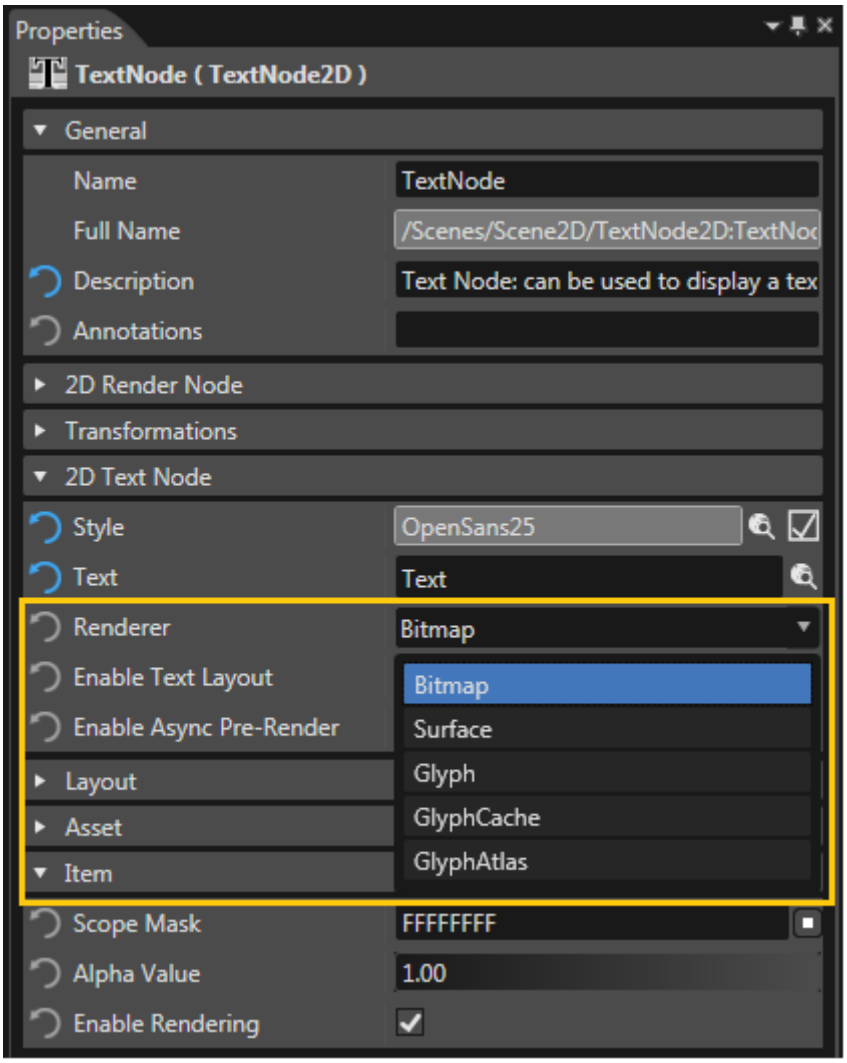
The new TextNode2D is located in the Toolbox panel. In order to add a text to the scene, drag a TextNode from the panel to a Scene2D node.



From the Properties panel of the TextNode, configure the necessary properties for rendering: *Text*, *Style*.



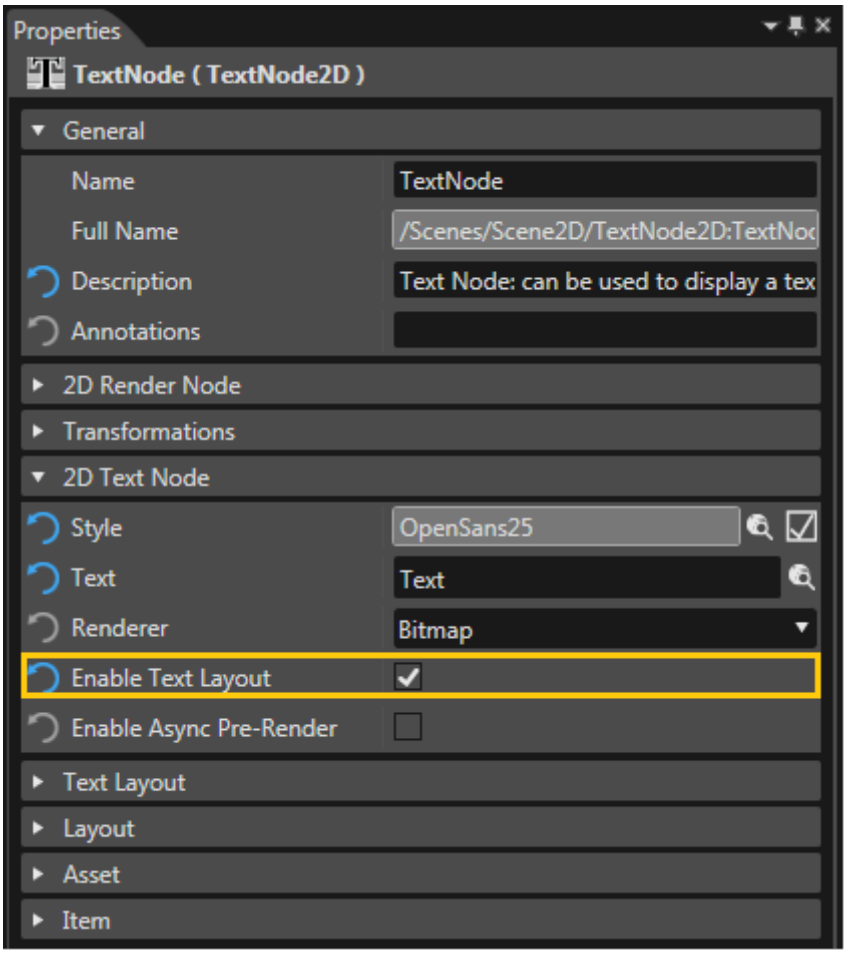
Candera::TextNodeRenderer type of Candera::TextNode2D can be changed using the *Renderer* property. Default value in SceneComposer is Bitmap.



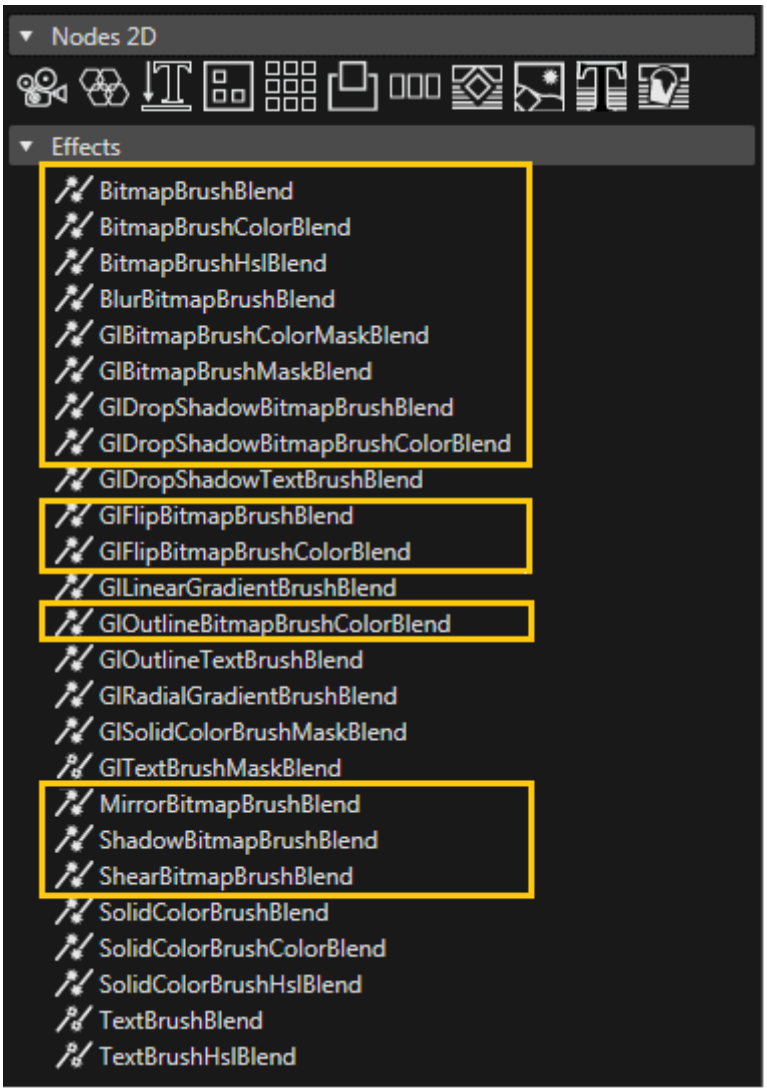
Candera::DefaultTextNode2DLayouter is attached by checking *Enable Text Layout*. Additional layouter properties are exposed in the Properties Panel:

- Multi line (enabled by default)
- Word wrap (disabled by default)

> For truncation to occur, *Enable Text Layout* property needs to be enabled.



When a TextNode2D is created, a Candera::BitmapBrushColorBlend is automatically associated with the node, which is used for changing text color only. Any BitmapBrush effect present in the list of effects from Toolbox Panel can be used for text rendering. The visual appearance of the text will be affected depending on the type of effect chosen.



TextEngine Render Interface

Overview: Render Interface

The main interface to render text is the method

- Candera::TextRendering::TextRenderer::Render(TextRenderContext &context, const LayoutingOptions &layoutingOptions, const ShapingOptions &shapingOptions, const TextProperties &textProperties) const

> Compare this interface also with the functional steps introduced in chapter Fonts and Styles.

Text Render Context

A Candera::TextRendering::TextRenderContext is required to define a target for text rendering i.e. where to render the text to.

Configure Bitmap as Target for Text Rendering

A Candera::TextRendering::BitmapTextRenderContext can be used to draw text into a bitmap. So first of all the bitmap which shall be drawn needs to be defined with

- the Candera::TextRendering::BitmapTextRenderContext::SetBitmap() method to assign a bitmap to render to,
- the Candera::TextRendering::BitmapTextRenderContext::SetPenColor() method to define the text color
- the Candera::TextRendering::BitmapTextRenderContext::SetClipRect() method to set a clipping rectangle for the text.

```
    BitmapTextRenderContext bmpRenderContext;
    static_cast<void>(bmpRenderContext.SetBitmap(m_bitmap));
    bmpRenderContext.SetPenColor(m_color);
```

Layouting Options

The Candera::TextRendering::LayoutingOptions define the layout of the rendered text with

- the Candera::TextRendering::LayoutingOptions::SetHorizontalAlignment() and Candera::TextRendering::LayoutingOptions::SetVerticalAlignment() methods to specify the text alignment,
- the Candera::TextRendering::LayoutingOptions::SetMultilineTextEnabled() method to toggle multi line and single line layout.
- the Candera::TextRendering::LayoutingOptions::SetLineSpacing() method to modify the space between lines of multi line text.
- the Candera::TextRendering::LayoutingOptions::SetWordWrapEnabled() method to enable automatic word wrap for multi line text.

See also:

Candera::TextRendering::LayoutingOptions for further options.

Shaping Options

The Candera::TextRendering::LayoutingOptions define how the text shall be transformed from logical to display presentation. The usage of one of the constructors is recommended:

- Candera::TextRendering::ShapingOptions::ShapingOptions(const StylePtr &style)
- Candera::TextRendering::ShapingOptions::ShapingOptions(const StylePtr &style, const CulturePtr &culture)

Text Properties

The Candera::TextRendering::TextProperties simply define the text itself which shall be rendered.

See also:

the constructor Candera::TextRendering::TextProperties::TextProperties(const TChar *text, TextLength length).

Font Metrics and Text Sizes

Font Size

In Candera the font size (respectively the font height) has to be provided in pixel size. Note that in several word processing programs, like *Microsoft Word*, fonts are defined in point size. The conversion between pixel size and point size is calculated as follows (see FreeType Glyph Conventions):

- pixel_size = point_size * resolution / 72

Microsoft Windows e.g. defines for historical reasons a default resolution of 96 PPI (DPI). A font in *Microsoft Word* with defined point size 18 has the same height as the font defined in Candera with pixel size 24 (24 = 18 * 96/72).

Font Metrics

For a font, static font metrics are provided describing font ascender, descender values in device pixel, the font line height and the maximum vertical advance of a font character: Candera::TextRendering::Font::GetMetrics()

```
    TextRendering::Metrics metrics;
    if (!m_font.GetMetrics(metrics)) {
        return false;
    }
```

Text Bounds

Further, the text bounds (width, height, etc.) of a given string can be retrieved by calling *GetTextRectangle* method of Candera::TextRendering::GlyphTextMeasureContext or Candera::TextRendering::CursorTextMeasureContext.

Candera::TextRendering::GlyphTextMeasureContext builds a rectangle corresponding to the smallest surface covered by all the glyph bitmaps. The position of this rectangle(represented by (left, top) coordinates) can be offset from the text position. Therefore, when rendering the text, the Layouting Options should be constructed using the inverse of this offset.

Candera::TextRendering::CursorTextMeasureContext builds a rectangle corresponding to the whole surface swept by the cursor when moving while processing the text. The cursor is considered to be a segment of length equal to the style height. This rectangle does not usually have an offset and it contains the final advancement of the cursor.

```
    GlyphTextMeasureContext glyphContext;
    static_cast<void>(textRenderer.Render(glyphContext, LayoutingOptions(), ShapingOptions(m_sty
    TextRect textBound = glyphContext.GetTextRectangle();
    mTextStartPos.SetX(-textBound.GetPosition().GetX());
    mTextStartPos.SetY(-textBound.GetPosition().GetY());
    Int16 textWidth = textBound.GetWidth();
    Int16 textHeight = textBound.GetHeight();
```

```
    static_cast<void>(textRenderer.Render(bmpRenderContext, LayoutingOptions(mTextStartPos), Sha
```

The Layouting Options describe the layout of the text. The Shaping Options define how the text is shaped (ligatures, text direction, ..) and they also contain the style.

Simple Text Rendering

Rendering Simple Text

The Candera::TextRendering::TextRenderer provides the core function to render text.

For simple text rendering the Candera::TextRendering::TextRenderer::Render() method is called. Besides the actual text (type TChar*) and the BitmapTextRenderContext as target, layouting and shaping can be specified.

```
    static_cast<void>(textRenderer.Render(bmpRenderContext, LayoutingOptions(mTextStartPos), Sha
```

Finally the image has to be updated to apply the changes.

```
    bool success = m_textureImage->Update();
```

Character-Glyph Position Mapping

Overview

Candera text engine provides an interface which allows to map positions in character string to positions in glyph string. This feature can be useful for applications and widgets in order to perform various operations depending on the character position like, for instance, highlighting text.

Example

For the convenience, we choose to improve the TextTextureWidget with this highlighting feature. The widget will highlight the text by rendering in red all the glyphs corresponding to the characters having the index in the interval *m_start* and *m_end*.

The character position information is accessible in the Blit method of the Candera::BitmapTextRenderContext class. Consequently, this class should be derived and the Blit method overwritten as follows:

```
class BitmapTextRenderContextSelection : public BitmapTextRenderContext
{
public:
    BitmapTextRenderContextSelection(Int selectionStart, Int selectionEnd, Color
 color) : m_start(selectionStart), m_end(selectionEnd), m_color(color) {}
    virtual void Blit(Int16 x, Int16 y, const GlyphBitmap& glyph);
private:
    typedef BitmapTextRenderContext Base;

    Color ComputeColorForCharPosition(TextPosition position);
    Int m_start;
    Int m_end;
    Color m_color;
};

void BitmapTextRenderContextSelection::Blit(Int16 x, Int16 y, const GlyphBitmap& glyph){

    SetPenColor(ComputeColorForCharPosition(glyph.characterPosition));
    Base::Blit(x, y, glyph);
}

Color BitmapTextRenderContextSelection::ComputeColorForCharPosition(TextPosition position){
    if(position >= m_start && position <= m_end){
        return Color(255,0,0);
    } else {
        return m_color;
```

```
    }
}
```

In TextTextureWidget::UpdateTextureImage(), use the newly created class instead of Candera::BitmapTextRenderContext:

```
    BitmapTextRenderContextSelection bmpRenderContext(m_selectionStart, m_selectionEnd, GetText(
```

Shortening Support For Bidirectional Text

Feature Overview

This feature ensures that the bidirectional text shall be shortened whenever it does not fit into its designated display area. The algorithm which does the shortening can deal with different levels of embedding of right-to-left text in left-to-right text and vice versa. Depending on the text direction of the last chunk, the shortening string is inserted after the last fitting text segment, on the left or on the right end.

You can take advantage of this feature by using one of the following, depending on the type of scene being rendered:

Scene type	Node name	Widget name
2D	Candera::TextNode2D	-
3D	-	TextTextureWidget

Truncation using TextNode2D

This feature is active only when the node parameter *Enable Text Layout* is enabled. The text will be automatically truncated whenever its size no longer fits in the specified *Size* property of the node. TextNode2D provides only one truncation method, by cutting text and adding trailing ellipsis to fit the space.

Truncation using TextTextureWidget

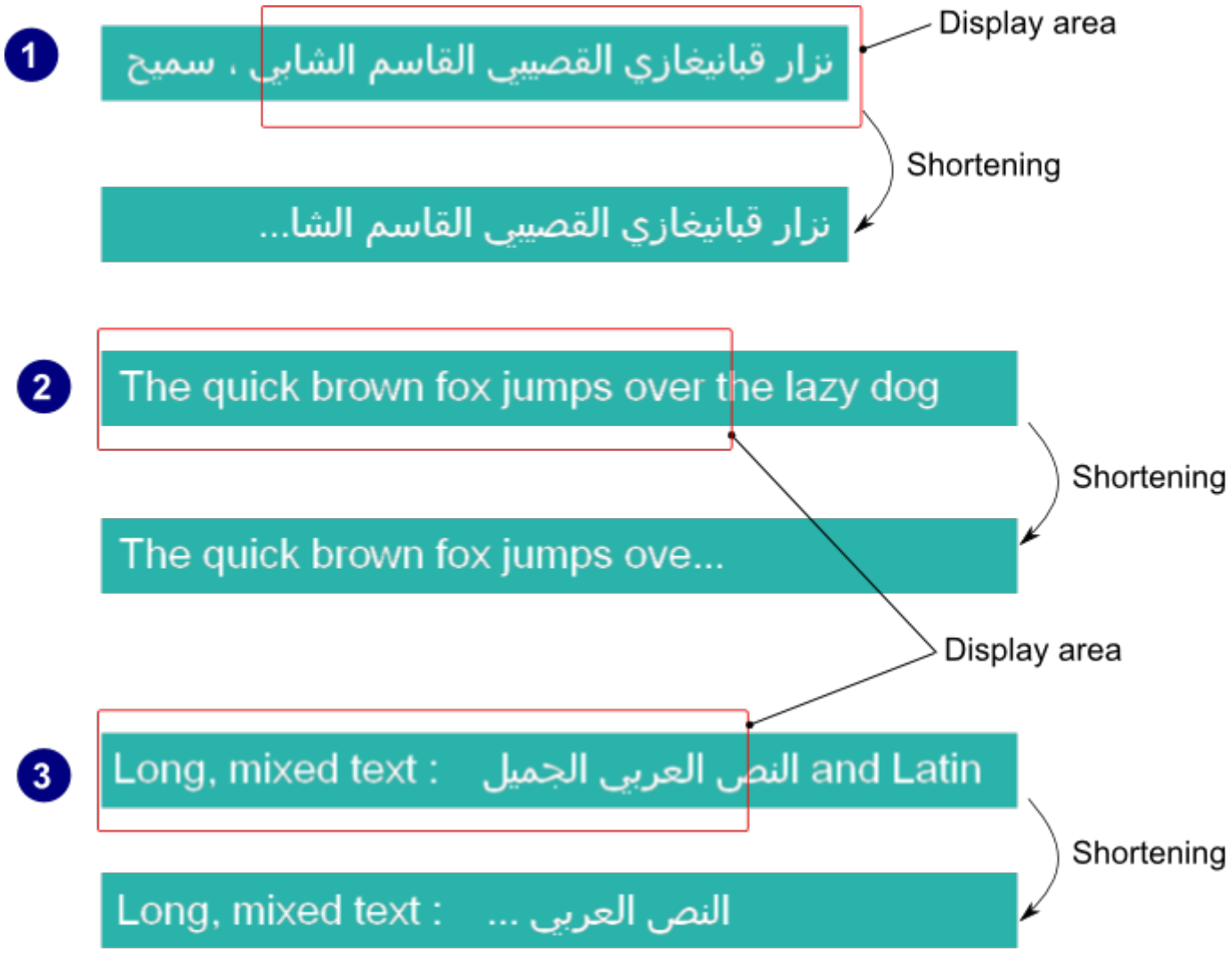
This feature is active only when the widget parameter *Truncation method* is set to the **Text** value. The character(s) used for visual feedback of text shortening is configurable by setting the *Truncation text* property of the 3D widget.

Examples

The image below exemplifies three scenarios :

1. Shortening of the right-to-left text (Arabic)
2. Shortening of the left-to-right text (Latin)

3. Shortening of right-to-left text embedded in left-to-right text (left-to-right and right-to-left)



Font Hinting

Overview

Font hinting is the use of mathematical instructions to adjust the display of an outline font so that it lines up with a rasterized grid. This is done by interpolating the pixels in order to more clearly render the font. At low screen resolutions, hinting is critical for producing clear, legible text. It can be accompanied by antialiasing and (on liquid crystal displays) subpixel rendering for further clarity.

> The hinting applies only to **outline** fonts so the bitmap fonts and the stroke fonts will not be affected by this feature.

Hints are usually created in a font editor during the typeface design process and embedded in the font. In general, there are three possible ways to hint a glyph:

- The font contains hints to guide the rasterizer, telling it which shapes of the glyphs need special consideration. The hinting logic is partly in the font and partly in the rasterizer.
- The font contains exact instructions (also called bytecode) on how to move the points of its outlines, depending on the resolution of the output device, and which intentionally distort the (outline) shape to produce a well-rasterized result. The hinting logic is in the font; ideally, all rasterizers simply process these instructions to get the same result on all platforms.
- The font gets auto-hinted (at run-time). The hinting logic is completely in the rasterizer. No hints in the font are used or needed; instead, the rasterizer scans and analyzes the glyphs to apply corrections by itself.

Enabling Hinting

The hinting can be enabled or disabled by the means of the method Candera::TextRendering::Font::SetHintingEnabled(). When hinting is disabled, the glyphs may appear blurred, as you can see below. The text was rendered with a small size font (9px) and then zoomed in:

Enabling AutoHint

The usage of the automatic hinter of the font driver can be controlled by calling the method Candera::TextRendering::Font::SetAutohintEnabled(). When the automatic hinter is disabled either the native hinter of the font is used either the hinting is disabled (zoomed in image):

Enabling Force AutoHint

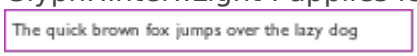
If a given font driver doesn't provide its own hinter, the auto-hinter is used by default. If a format-specific hinter is provided, it is still possible to use the auto-hinter by enabling it using the method Candera::TextRendering::Font::SetForceAutohintEnabled().

Hinting Mode

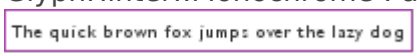
The driver autohinter can be configured by calling the Candera::TextRendering::Font::SetHintingMode() method with one of the following parameters(hinting algorithms):

- GlyphHinter::Normal : normal hinting, optimized for standard gray-level rendering

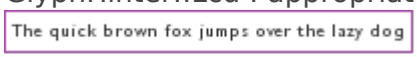
- GlyphHinter::Light : applies less corrections than normal hinting for better performance



- GlyphHinter::Monochrome : appropriate for monochrome rendering



- GlyphHinter::Lcd : appropriate for horizontally decimated LCD displays



- GlyphHinter::VerticalLcd : appropriate for vertically decimated LCD displays

> All modes except GlyphHinter::Monochrome use 256 levels of opacity.

Code Example

```
    m_font1.SetHintingMode(GlyphHinter::Monochrome);
    m_font2.SetHintingMode(GlyphHinter::Light);
```

Glyph Bitmap Format

The output type of the glyph rendering process can be configured by calling the method Candera::TextRendering::Font::SetRequestedGlyphFormat() with one of the following render modes as parameters:

- GlyphBitmap::Unknown : format is unspecified
- GlyphBitmap::Monochrome : correspond to 1-bit bitmaps (two levels of opacity)
- GlyphBitmap::Lcd : format use with horizontally decimated RGB or BGR sub-pixel LCD displays
- GlyphBitmap::VerticalLcd : format use with vertically decimated LCD displays
- GlyphBitmap::Grayscale : format is grayscale, one byte per pixel; this is the default render mode

Each of the render modes above correspond to a specific type of scanline conversion performed on the outline.

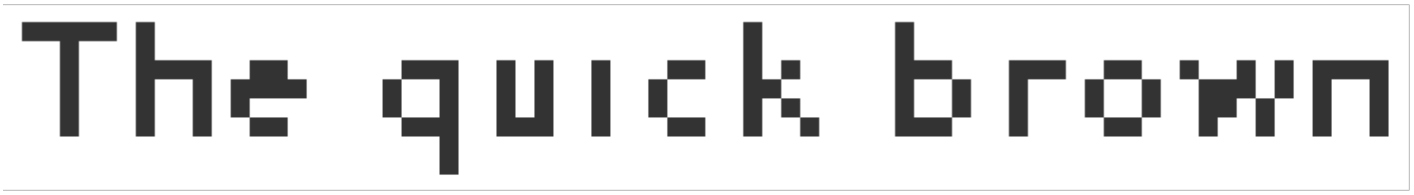
Code Example

```
    m_font1.SetRequestedGlyphFormat(GlyphBitmap::VerticalLcd);
    m_font2.SetRequestedGlyphFormat(GlyphBitmap::Grayscale);
```

Monochrome Render Mode

The monochrome render mode removes anti-aliasing because it's using only two levels of opacity and gives clearer appearance for small size fonts (no blurring). Please see below a comparison between monochrome and grayscale render modes:

Font size: 9 px, Render mode: GlyphBitmap::Monochrome, Hinting mode: GlyphHinter::Monochrome:



Font size: 9 px, Render mode: GlyphBitmap::Grayscale, Hinting mode: GlyphHinter::Normal:

An another advantage of using this render mode is the smaller memory footprint for glyphs because of 1bit format.

Revision #15
Created 2023-02-22 01:46:23 UTC by Tsuyoshi.Kato
Updated 2025-01-17 10:29:22 UTC by Elisabeth Zauner