Tutorial for Text Rendering

Introduction

Text Rendering involves both the handling of Unicode standards regarding character encoding as well as the generation and rendering of the glyphs. The sub-chapters here explain the basic principles.

Character Encodings

character encoding system consists of a code that pairs each character from a given repertoire with something else such as a bit pattern, sequence of natural numbers, octets, or electrical pulses (Source: Wikipedia).

Unicode is the most important standard for character encodings and defines (among other things) the mapping of a character to a Unicode code point (bit pattern):

- Fixed length encodings of code points (ASCII, UCS2, UCS4 etc).
- Variable length encoding of code points (UTF-8, UTF-16, UTF-32)
- Chinese GB18030 is the only relevant other encoding standard

All texts which are rendered within the TextEngine have to be of type TChar*. Text in TChar* have to be encoded in UTF-8. UTF-8 uses a varying number of TChars to encode one glyph. E.g. a UTF-8 encoded string with special characters: "&auml&Auml&ouml&Ouml&uuml&Uuml&szlig&#8364" consists of 8 glyphs but of 17 bytes because several of these special characters are encoded in more than one byte.

The methods FeatStd::String::GetCharCount() and FeatStd::String::GetCodePointCount() can be used to get the numbers as explained in the example above. An instance of FeatStd::String class can be constructed with TChar* by the constructor FeatStd::String::String(const TChar *).

> Char data type must not be used for I18N text and always maps to ASCII.

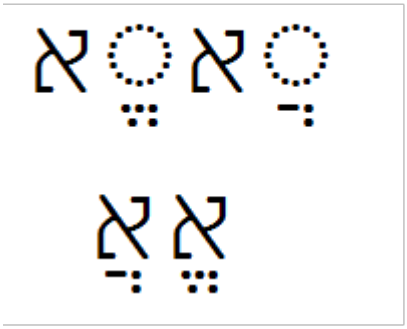
Complex Script Languages

A complex script is a writing system which requires complex transformation of glyphs like substitution and positioning. Examples of such writing systems are arabic, hebrew and thai. Complex transformations include

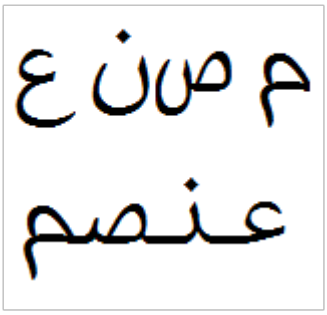
- bi-directional ordering of characters,

- contextual shaping which means that displayed glyphs depend on the surrounding code points in the text - displayed glyphs can have different shapes depending on their position in a word or can be combined to ligatures,
- combining characters where several characters are combined into one shape

Example of hebrew script:



Example of arabic script:



Above the characters as they are written and below as they are displayed.

Text Rendering - Functional Steps

Text Rendering involves the following functional steps:

1. Glyph Rendering
2. Shaping
3. Layouting

1. Glyph Rendering

Takes a single code point and font metrics to render the according glyph. The Freetype font engine for example takes a code point and outputs a 8 bit alpha.

2. Shaping

Shaping is the transformation from logical to display presentation of a text. The Shaping process is responsible for considering complex scripts. The Shaping process is applied on paragraph level.

GPOS/GSUB Tables

OpenType fonts may include GSUB (glyph substitution) and GPOS (glyph positioning) tables.

The GSUB table contains (among other things)

- Ligature substitution (replace multiple code points with single code point)
- Contextual substitution (alternative presentation of the glyph depending on his context) If no GSUB table is available, the standard substitutions defined by Unicode apply (eg. mandatory set of Arabic ligatures).

The GPOS table contains (among other things)

- glyph placement
- adjustment of marks
- kerning
- attachment points for combining characters

Shaping Libraries

- FEAT ComplexScript library does shaping for Arabic language
- FreeBidi - functional equivalent to FEAT ComplexScript
- Harfbuzz - Arabic + many other scripts

A Candera feature define is available in CMake to select between the 3rd party software components *ComplexScriptLib* and *HarfBuzz* for text shaping.

3. Layouting

Layout text that has been shaped. Typical operations:

- Line breaking
- Word breaking
- Text alignment (horizontal and vertical)

Fonts and Styles

Description

This section contains information about how to use fonts and styles.

Accessing Fonts

SceneComposer: Import Fonts

For using fonts, first of all the fonts which shall be used need to be imported into the SceneComposer solution.

> The font will only be available in the finally generated asset, if:
> - it is either used in any scene (as part of a style which is used in a widget style property for example)
> - or the "Always include in asset" flag is enabled in the font resource.

See also:

- How to Import Resources in SceneComposer User Manual

Initializing FontEngine

Next, the application needs to create a Candera::TextRendering::FontEngine instance and initialize it for using the Candera::AssetFontStore. AssetFontStore will access the font part of the loaded asset and provide it to the TextEngine.

```
    static AssetFontStore s_fontStore;

    result = result &
&
Candera::TextRendering::FontEngine::Instance().Init(&
s_fontStore);
```

Using a Font

The easiest way to use a font is to setup the font as part of a Candera::TextRendering::Style and use the style as a widget property within SceneComposer. This allows to specify the FontFamily and the FontSize via SceneComposer GUI. FontSize has to be specified in pixel size (see Font Metrics and Text Sizes). In case the application needs to switch the font dynamically, at runtime, setup the font and create a style:

```
    Font font;

    font.Setup("openSans", 20);

    m_style = SharedStyle::Create();

    m_style->
SetDefaultFont(font);
```

Access to AssetFontStore

Include FontFaceName and FontFaceSize Property in the header file of the widget.

```
// Add Property
            CdaProperty(FontFaceName, const Candera::Char*, GetFontName, SetFontName)
                CdaDescription("Font Face Name for the text (used only if \"Use Text
 Font Property\" is disabled).")
                CdaCategory("Accessing Fonts from FontEngine")
            CdaPropertyEnd()

            CdaProperty(FontFaceSize, Candera::UInt16, GetFontSize, SetFontSize)
                CdaDescription("Font Face Size for the text (used only if \"Use Text
 Font Property\" is disabled).")
                CdaCategory("Accessing Fonts from FontEngine")
            CdaPropertyEnd()
// end add Property
```

> The FontFaceName property must be set with a valid Candera path. Keep in mind that, unlike paths generated by the Font dialog where the; the "/" character, the Candera paths use "#" as delimiter.

Also, do not forget to initialize member variables in the constructor, otherw; e SceneComposer will not be able to work with the widget.

```
SimpleTextWidget::SimpleTextWidget() :
    m_text(0),
    m_style(SharedStyle::Create()),
    m_color(),
    m_shader(0),
    m_fontName(0),
    m_fontSize(12),
    m_bb(0),
    m_bitmap(0),
    m_bitmapPixels(0),
    m_defautScaleVector(1.0,1.0,1.0),
    m_useTextFont(true),
    m_;
UpdateRequired(true),
    mTextStartPos(0, 0)
{
}
```

Setup the font as described below.

```
        static Font f;
        if (f.Setup(m_fontName, static_cast<Candera::Int16>(m_fontSize))) {
            m_font = f;
        }
```

And update the style used for rendering:

```
    m_style->SetDefaultFont(m_font);
```

AssetFontStore Load Strategies

AssetFontStore supports two font load strategies:

- PreloadStrategy: entire fonts are loaded into memory once and font data; accessed from there by Text Engine.
- OnRequestLoadStrategy: small font portions are streamed directly from the asset repository when Freetype (or indirectly Harfbuzz) needs them.

Use SetDefaultFontLoadStrategy to select one of these load strategies.

Use SetDefaultFontLoadStrategyExceptionL; t to exclude a l; t of fonts from the selected font load strategy.

Example:

```
    // Load fonts with OnRequestLoadStrategy
    selector.SetDefaultFontLoadStrategy(
```

```
Candera::AssetFontStore::LoadStrategySelector::OnRequestLoadStrategy);
    // "Bitstream Vera Sans" font shall be loaded with PreloadStrategy:
    const Char* g_fontName[2] = { "ConstructionKit##ConstructionKit#Resources#Fonts#Bitstream Ve
    selector.SetDefaultFontLoadStrategyExceptionL;
t(g_fontName);
    fontStore.SetLoadStrategySelector(&selector);
```

> Courier applications can use Courier::ViewHandler::SetFontStoreProviderCallback to provide specific settings for the Candera::AssetFontStore in use.

Text Style: A Composition of Fonts

Text Style: Introduction

A Style can be seen as a composition of fonts.

Basically for a range of glyphs a proper font can be specified (also called *style entry*). Beside the style entries a style also cons; ts of a default font and a base style.

- The style entries are parsed from the first to the last style entry until a code point range that contains the input code point; found.
- All glyphs which do not match to one of the specified *style entry* range are rendered with the default font.
- The same parsing order; applied to code points which match to a range of a style entry, but the font does not contain a glyph for th; code point
- Each style can use another style as base style, which allows an arbitrary complex composition of fonts.

The **TextRenderingSolution** provided in the content folder of *cgi_studio_player* gives an example how text features can be applied in SceneComposer.

See also:

- Text Style Palette in SceneComposer User Manual
- Candera::TextRendering::Style in Candera API

Text Style: Example

Example

Use **TextRenderingSolution** provided in the content folder of *cgi_studio_player* to see how text features can be applied in SceneComposer.

Select *Scene2D_StyleExample* of the solution, which gives just a brief overview of text features and how they look like. It cons; ts of

- a Candera::TextNode2D.
- Fonts for the text node are defined in a style *Mystyle*.

Lets consider the configured style *MyStyle* set in the TextNode2D's style property:

- The **Default Font** property is set with predefined *DefaultFont* (Vera size 20)
- No **Base Style**, therefore for this field *Empty*, is defined

Expanding the Children panel shows the defined style entries. The example style consists of three style entries.

- **Entry 0** defines the font *BigLetter* (Comic size 60) for all glyphs in the code point range from 65 to 90. These upper and lower bounds represent the glyph range 'A' (65) - 'Z' (90) in decimal representation. In the example therefore all upper case letters are rendered with font *BigLetter*.
- **Entry 1** defines font *BigLetter* for numbers '0'(48) - '9'(57).
- **Entry 2** defines font *Korean* (UnBom size 40) for Korean glyphs. Code points for asian glyphs are in range 44032 - 55215 (in decimal representation). So using styles, glyphs from different cultures can be rendered without changing the font/style settings.

In case of this example all lower case letters do not have a proper style entry and the default font is taken to render.

> If the code ranges of different style entries overlap, the style entry with the higher number has priority (e.g. font in entry 2 has higher priority than font in entry 1).

Bitmap Font Engine

Overview

A bitmap font is essentially an image file which consists of several characters images and a header that depicts the size and location of each character inside the image. Each bitmap font might contain several fonts - mainly one font face with multiple sizes. One advantage of utilizing bitmap fonts is that the rendering to the screen requires very little resources and since each character is represented as a sub-texture, they can be reused without requiring additional memory on the GPU.

CMake configuration

In order to be able to use the bitmap font engine the Candera CMake project should be configured as follows:

- CANDERA_FONTENGINE should be set to "*Bitmap Font*"
- CANDERA_BITMAPFONT_FACE_SIZE_COUNT_MAX represents the number of font sizes supported. Default value is set to 6.
- CANDERA_BITMAPFONT_GLYPH_CACHE_SIZE represents the size of the buffer to cache glyph bitmaps. By default the value is set to 24*24*100.

> CANDERA_TEXTSHAPER should be set to "*NoShaping*" or "*ComplexScriptLib*" because Harbuzz Text Shaper cannot be used with BitmapFont.

Font Setup

This font engine uses a proprietary format to describe glyph bitmaps and glyph meta info. In order to use a bitmap font, provide an implementation for Candera::TextRendering::FontStore.

Bitmap fonts require memory mapped assets. So, when the faces are loaded in be sure the storageType is set correctly:

```
    TextRendering::FontResourceDescriptor descriptor.storageType = TextRendering::FontResourceDe
```

Generating Bitmap Fonts

The following steps need to be considered:

1. Install Bitmap Font Generator from http://www.angelcode.com/products/bmfont

2. Open the application. From the **Options** menu:

   - Choose **Font Settings**. Set as needed the settings for "*Font*", "*Add font file*", "*Size*" etc.

   - Configure the **Export Options**. Set "*Font descriptor*" to "*XML*" and Textures to "*png - Portable Network Graphics*"

3. From the right panel of the main window select the needed subsets of glyphs.

4. Export the font by choosing "*Save bitmap font as ...*" from the **Options** menu.

Please see Import Fonts for details about the importing of bitmap fonts in SceneComposer.

> A bitmap font should be used at its native pixel size. However, they are scalable because of the use of bitmaps, but only scaling down is recommended. In case of up scaling, the visual quality will be reduced. Including separate font sizes for bitmap fonts is possible in order to achieve the best looking results.

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).

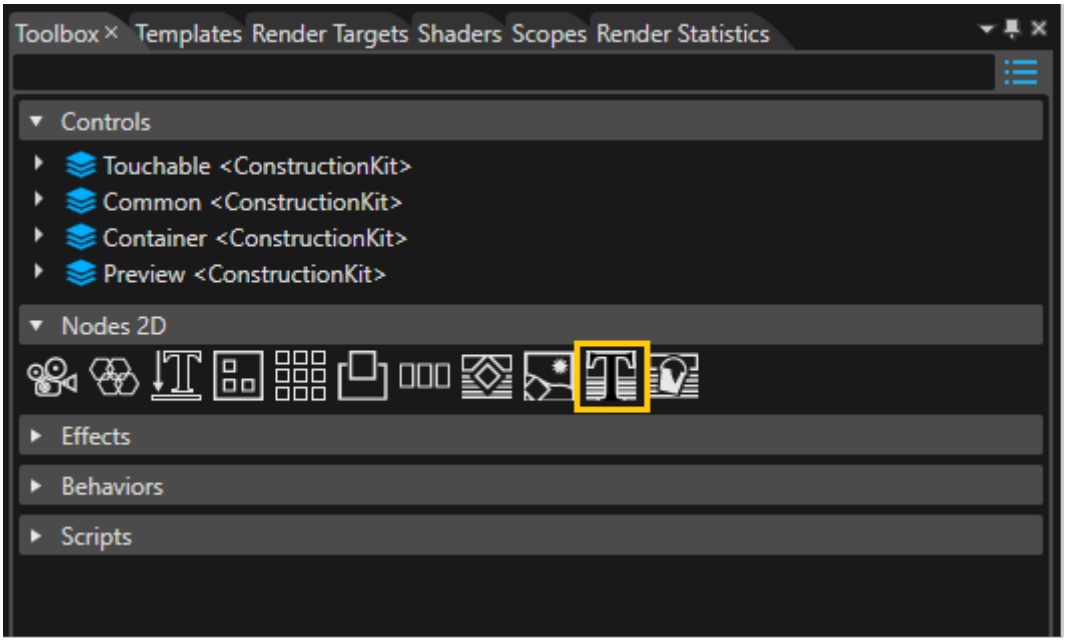
Text Length

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

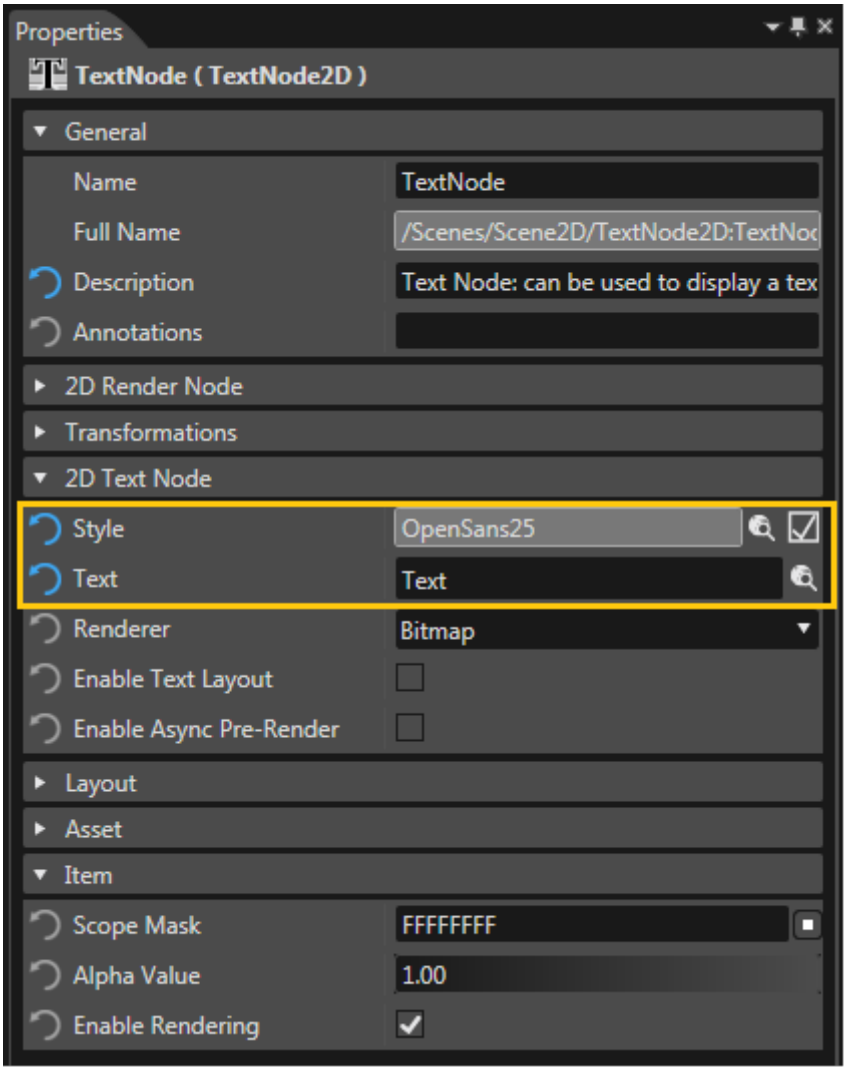
- Getting the layouting text box (measured with constant height, cursor to cursor from left to right):
  - Candera::TextNode2D::GetLayoutTextRectangle()
- Getting the bounding text box (glyph limits vertically and horizontally):
  - Candera::TextNode2D::GetBoundingTextRectangle()
  - Candera::Node2D::GetComputedBoundingRectangle()

Candera::TextNode2D in SceneComposer

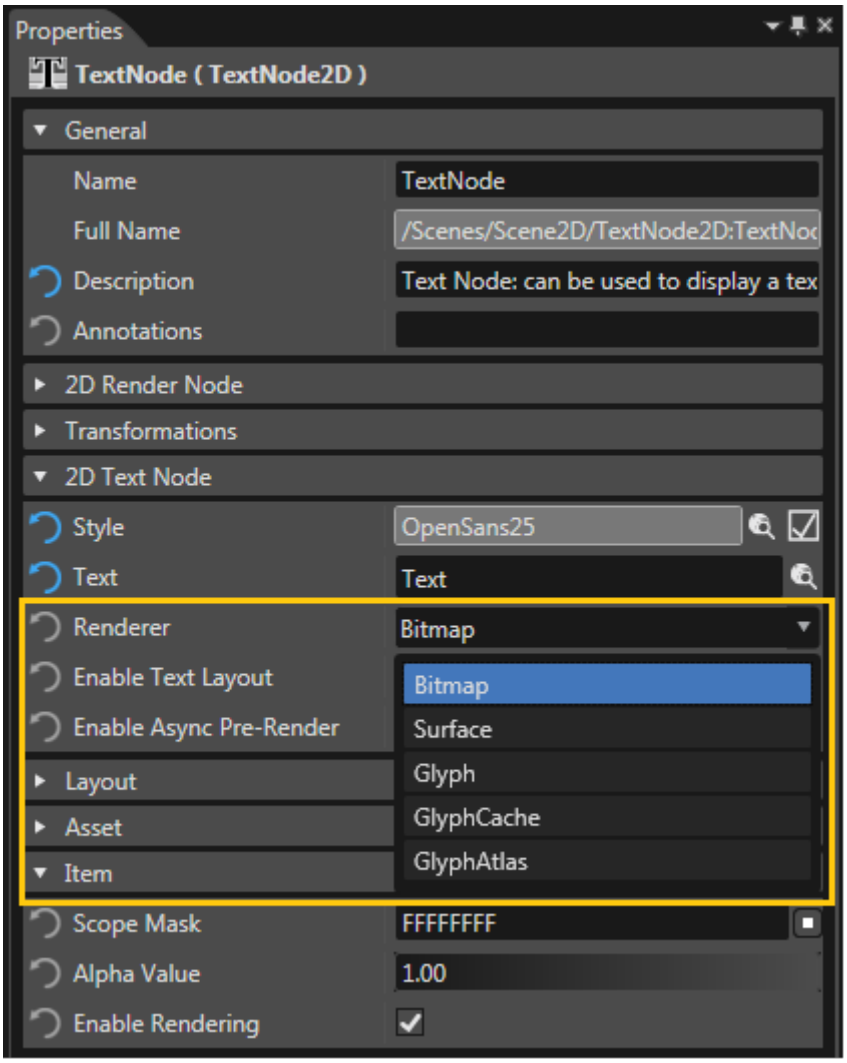
The new TextNode2D is located in the Toolbox panel. In order to add a text to the scene, drag a TextNode from the panel to a Scene2D node.



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



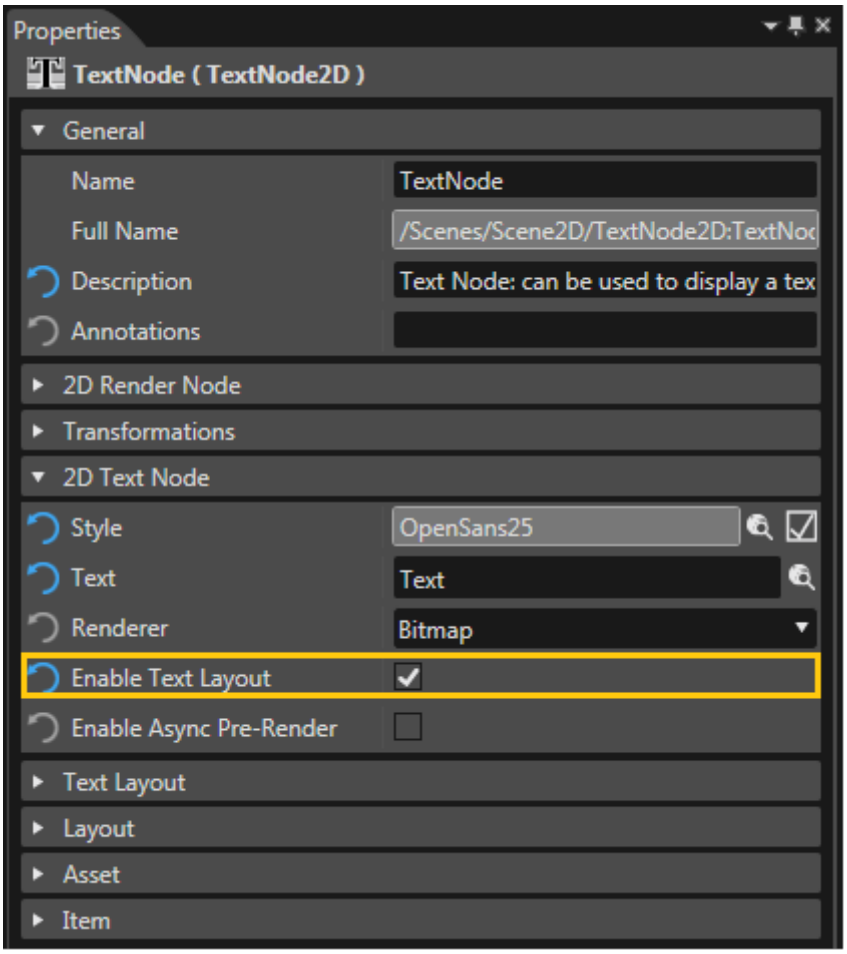
Candera::TextNodeRenderer type of Candera::TextNode2D can be changed using the *Renderer* property. Default value in SceneComposer is Bitmap.



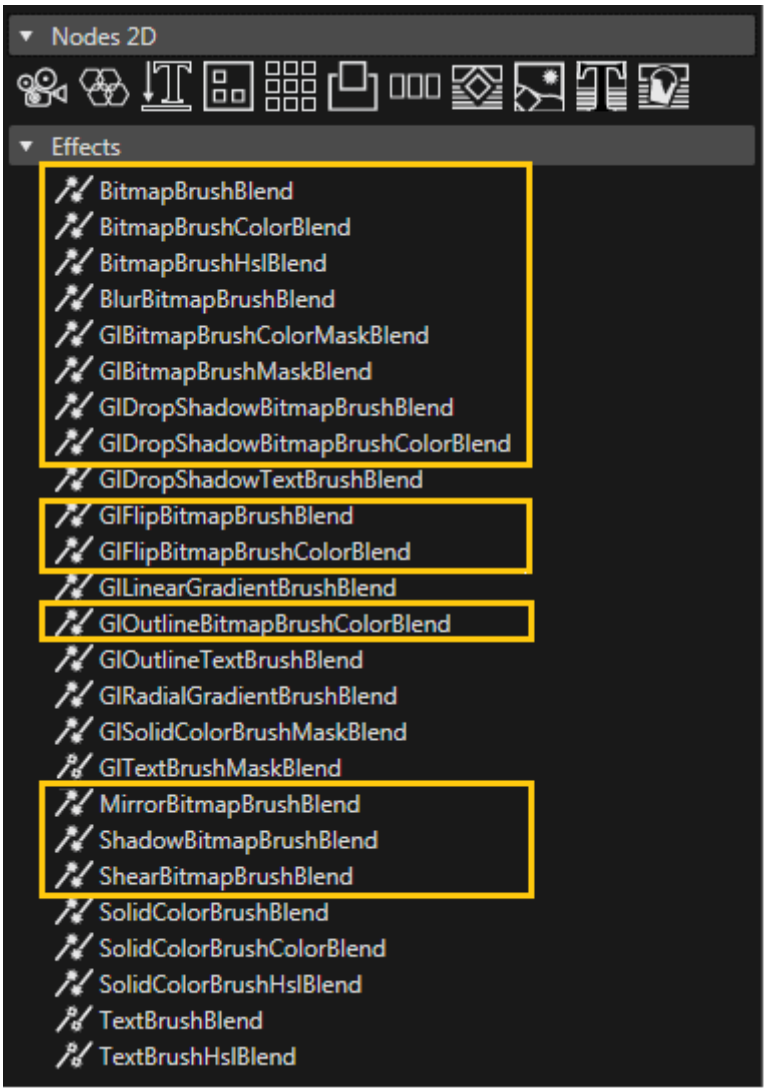
Candera::DefaultTextNode2DLayouter is attached by checking *Enable Text Layout*. Additional layouter properties are exposed in the Properties Panel:

- 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

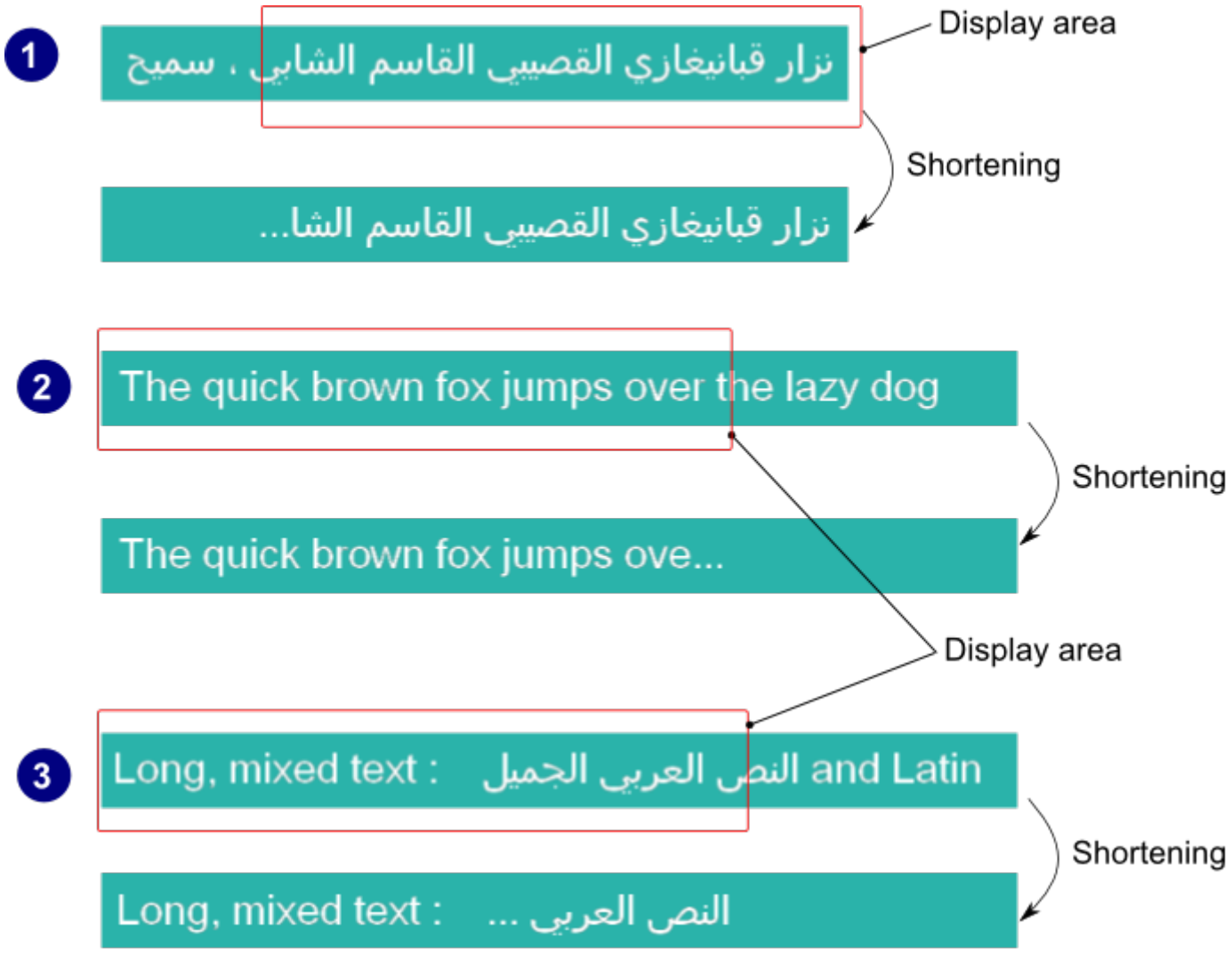
This feature is active only when the widget parameter *Truncation method* is set to the **Text** value. The character(s) used for visual feedback of text shortening is configurable by setting the *Truncation text* property of the 3D widget.

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

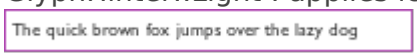
If a given font driver doesn't provide its own hinter, the auto-hinter is used by default. If a format-specific hinter is provided, it is still possible to use the auto-hinter by enabling it using the method Candera::TextRendering::Font::SetForceAutohintEnabled().

Hinting Mode

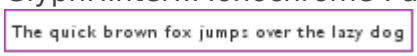
The driver autohinter can be configured by calling the Candera::TextRendering::Font::SetHintingMode() method with one of the following parameters(hinting algorithms):

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

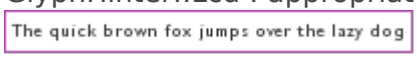
- GlyphHinter::Light : applies less corrections than normal hinting for better performance



- 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.

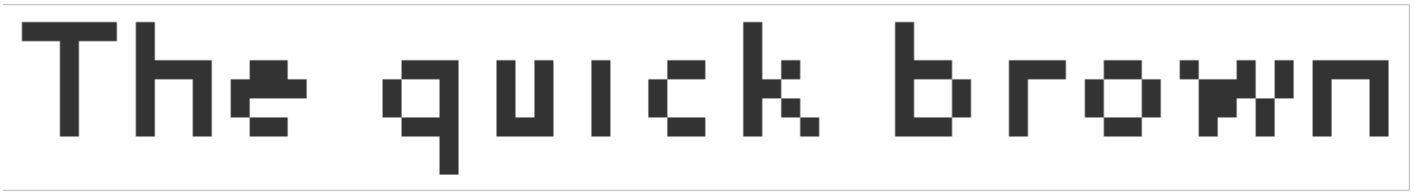
Code Example

```
    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.

Using asynchronous text rendering

Description

The Candera engine provides asynchronous text rendering to improve the handling of time-consuming text processing. The focus is not to improve performance but to improve resource handling.

Asynchronous text rendering requires knowledge of how text nodes work.

General information of how to use Candera::TextNode2D can be found in the following section:

- Using Candera::TextNode2D

Asynchronous text rendering requires further knowledge of which steps are necessary to display a text. Please refer to the following section:

- Text rendering - Useful knowledge

Based on the knowledge about the TextNode2D and CanvasText, the following chapters describe how asynchronous text rendering works. Further, these chapters explain how to customize the asynchronous text rendering and which restrictions should be considered when using this feature.

The documentation focuses mainly on Candera::TextNode2D. The concepts described inside these chapters are applicable to Candera::CanvasText, too.

Text rendering - Useful knowledge

TextNode2D – text render process

A text uses three steps in its render process:

1. Measurement and placement
2. Prerendering – Prepares the render device for the text (if necessary)
3. Render – Actual visible output

All of these steps have to be fulfilled to come to a visible result. Using a TextNode2D has three different ways to update its text and therefore three different ways to approach the final result. The main difference lies in providing information and handling the three process steps:

First case: No layout usage

Candera::TextNode2D has no real layout information. There is also no layout in which the Candera::TextNode2D is integrated. Therefore, trigger the render process for text render process is enough. The text is not bound to an area and has no relationship other than the visible layer to any other node in the scene graph.

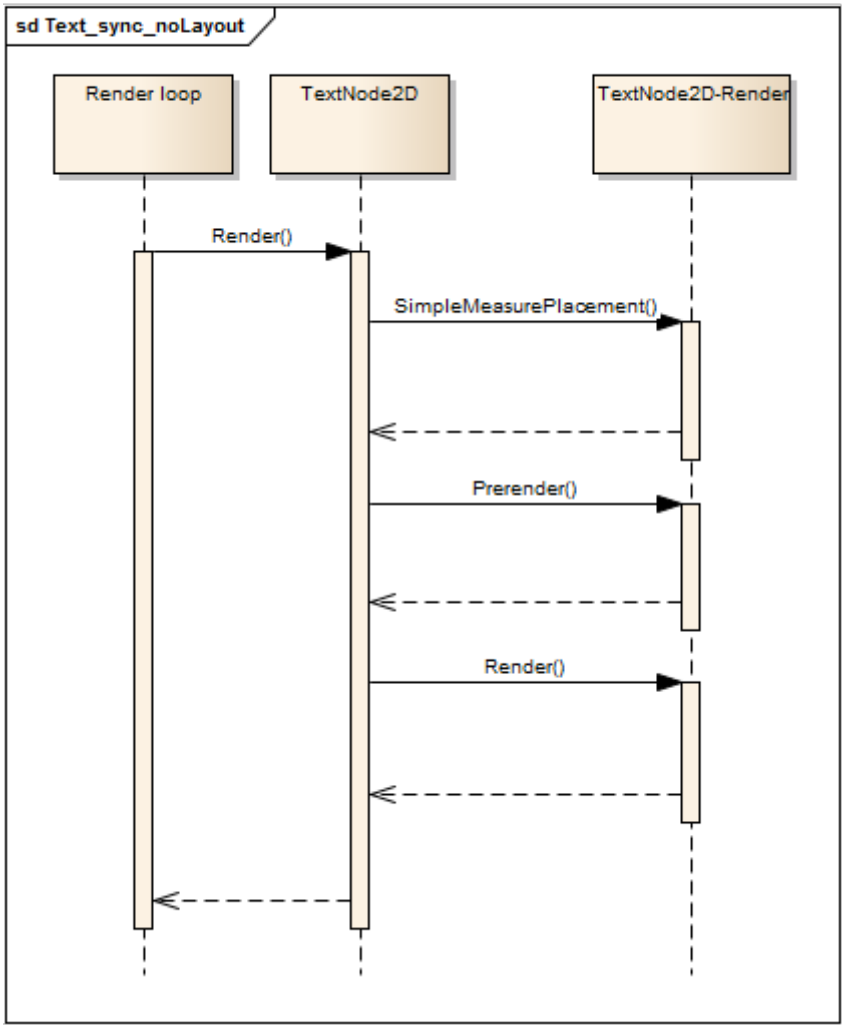
Pro:

- Lightweight version of text and Candera.
- Simple text can be shown without much effort.

Contra:

- Most of text placement features are disabled.
- No arrangement exists.

Flow chart of TextNode2D without any layouter (draft)



Second case: Layout process but no TextNode2D layouter

Candera::TextNode2D has no real layout information. However, it is arranged within a layout (e.g. GridLayout). This implies that Candera::TextNode2D has to provide actual layout results to arrange the text within the layout area. Therefore, the layout process has to start the text render process beforehand. The text render process has to complete the first step (measurement and placement) when the layout process evaluates the bounding box of the text. The layout process has no influence on the size of the bounding box. It only receives the information of the bounding box and can place all the other nodes around the text. The actual handling of this information is defined by the used layouter.

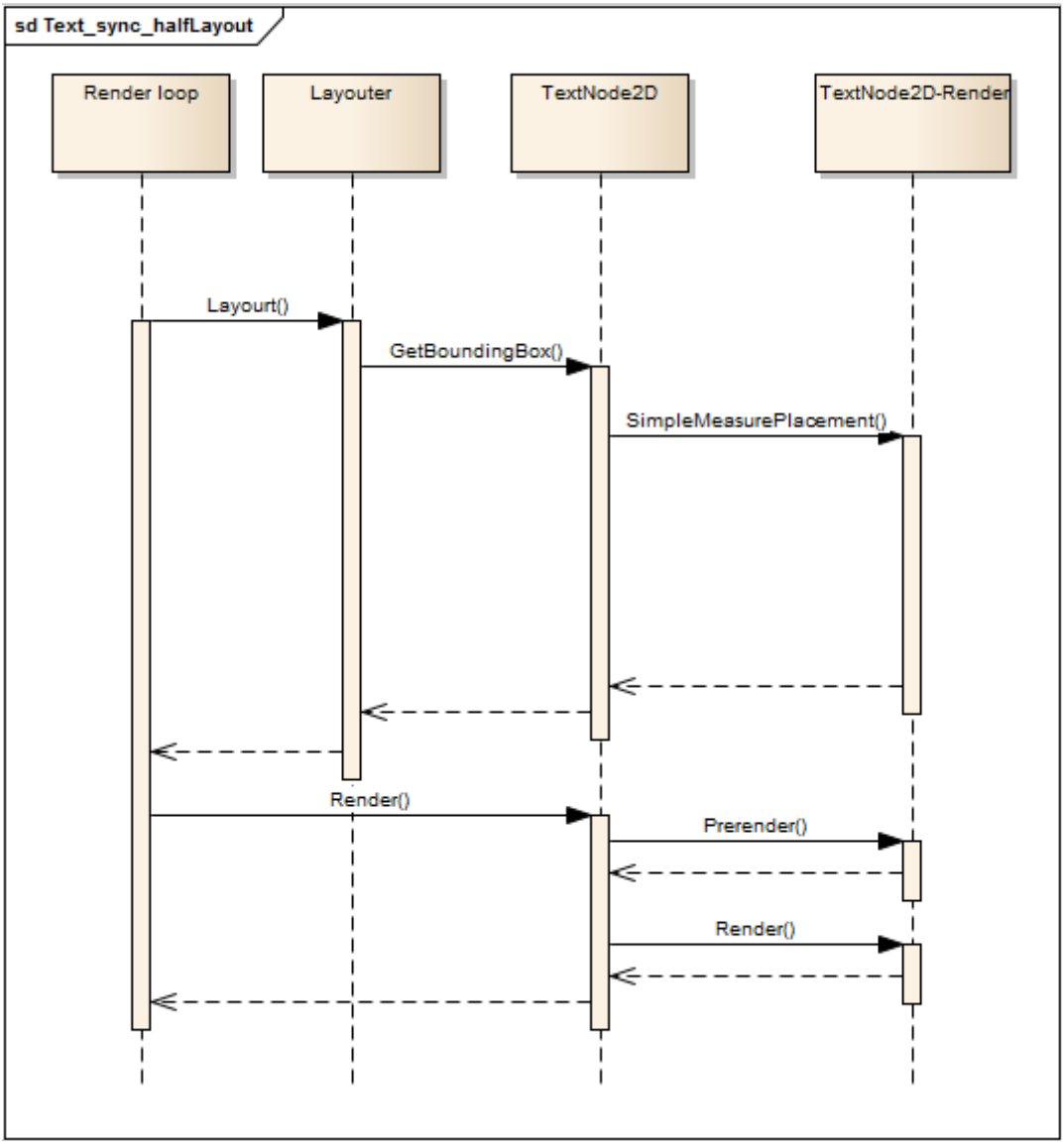
Pro:

- Lightweight version of text with advantages of an arrangement of scene graph nodes.
- Switching between first and second case is normally without major issues.

Contra:

- The text placement cannot be influenced by the layout process.
- TextNode2D itself cannot distinguish between first case and second case.

Flow chart of TextNode2D with layout process but without a TextNode2D layouter (draft)



Third case: Layout process with TextNode2D layouter

Candera::TextNode2D provides layout information. Additionally, the general layout process can influence the way text is displayed, too. The layout process can propose a maximum area in which the text has to fit. This has a direct impact on the placement. The text itself provides the layout process a bounding box which were actually used by it.

Pro:

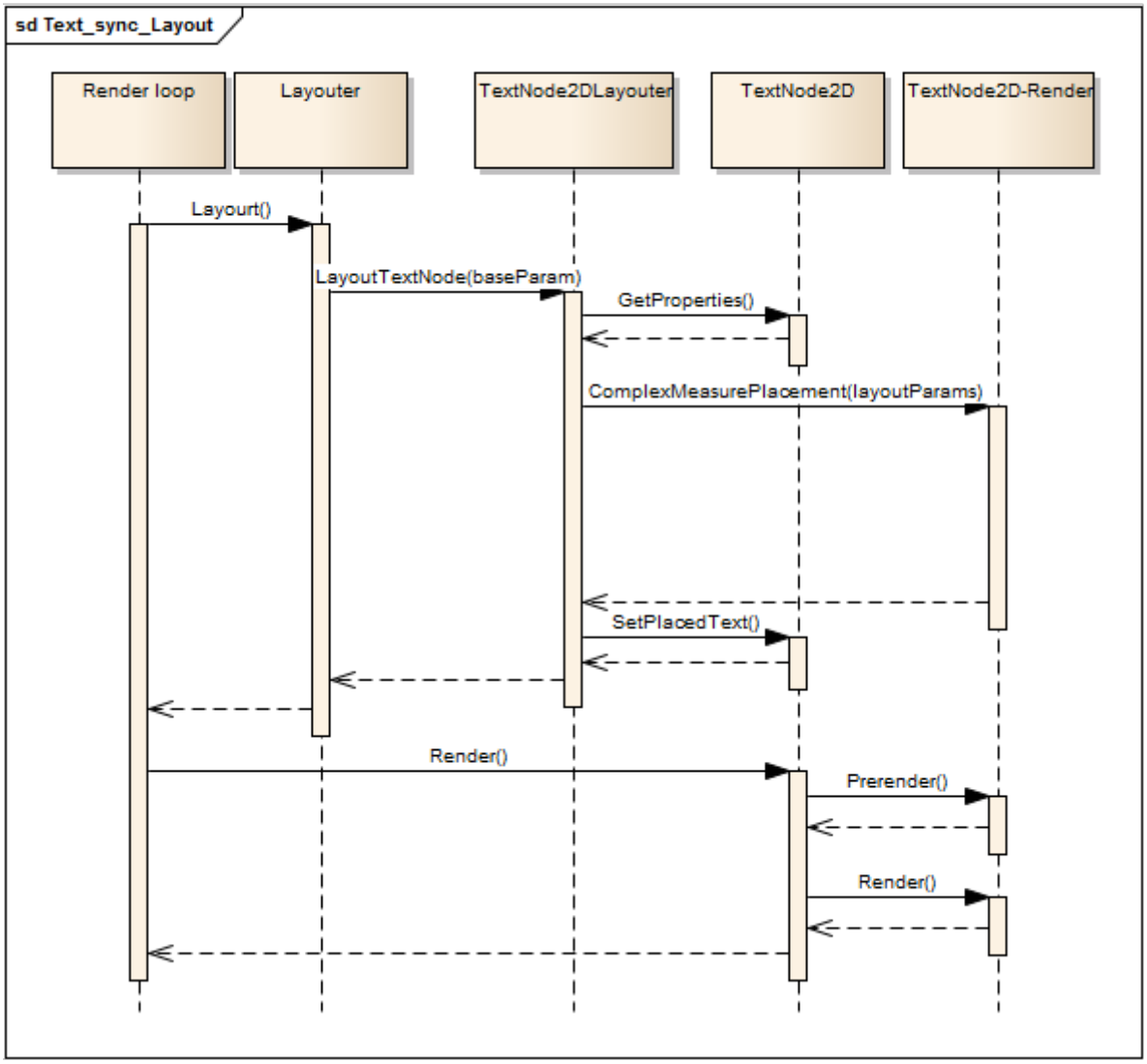
- Full feature set for text placement integrated in the scene graph arrangements.

Contra:

- Heavyweight version of text; more calculations have to be done.
- An attached layouter expects using the layouter.

> When a layouter is attached, it has to be used as it replaces the default behavior with a complex one. However, the information has to be provided by triggering the layout process otherwise it misses most of the information. This implies that other usages are wrong and lead to undefined or unwanted behavior.

Flow chart of TextNode2D with layout process and TextNode2D layouter (draft)



> Transformations like scale and rotation are applied after the text has been rendered. This means that you cannot use scaling to produce narrow or wide text, because the visual layout would be altered (right-aligned text will not be aligned any more after scaling). In this case, import a scaled font instead.

CanvasText – text render process

Candera::CanvasText uses the third approach of Candera::TextNode2D:

- Third case: Layout process with TextNode2D layouter

All other approaches have been removed from Candera::CanvasText to reduce the complexity of processing. Another difference to Candera::TextNode2D is the actual Render() part. Whilst the Candera::TextNode2D calls the Render() on each node, Candera::CanvasText renders it in a batched form by collecting texts with equal render settings and drawing them with a single draw call. Candera::CanvasText uses GlyphAtlas.

Please note also the following restriction when using asynchronous text processing:

- Cache type restrictions

Asynchronous text rendering

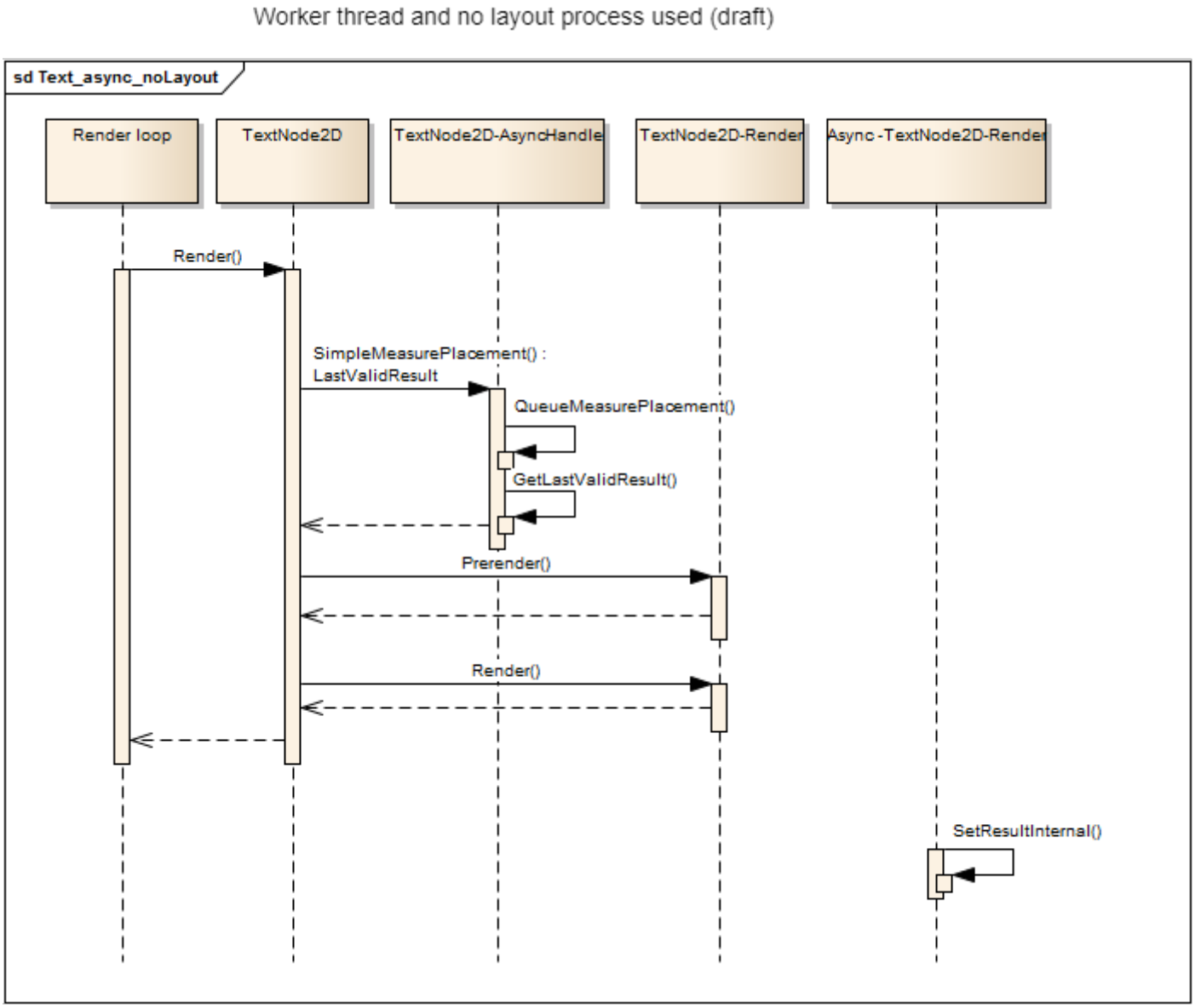
Asynchronous text rendering is a feature introduced to move text render steps to an own thread or render slot. To be precise only the first of the three render steps can be handled asynchronously. All the other steps are possibly bound to GPU or has to extend the complexity of how to receive the desired output for both framework and user application.

Asynchronous text rendering – Worker thread

The default way of asynchronous text rendering is the usage of a worker thread. The render loop delegates the first step, Measurement and Placement, to a worker thread. Every delegate is placed in a queue and ready for processing. Using a worker thread has its main advantage in not blocking the render loop. The render loop itself will not block until the text has been measured and placed. The disadvantage of this approach is that there is no control over the execution moment and result arrival time. It can happen after a single render cycle but also after an undefined amount of cycles.

The next figure shows in a draft how the measure and placement process has been replaced when asynchronous text rendering is used. For simplicity only the way without a layouter is used. All other ways do replace the measurement and placement process with the same approach in their flow chart.

The function which actually triggers the asynchronous process has to be triggered twice. One time to queue the process and a second time to retrieve the result. In this flow chart sample the Render-method has to be called twice. The second call has to be done when the asynchronous process has actually finished.

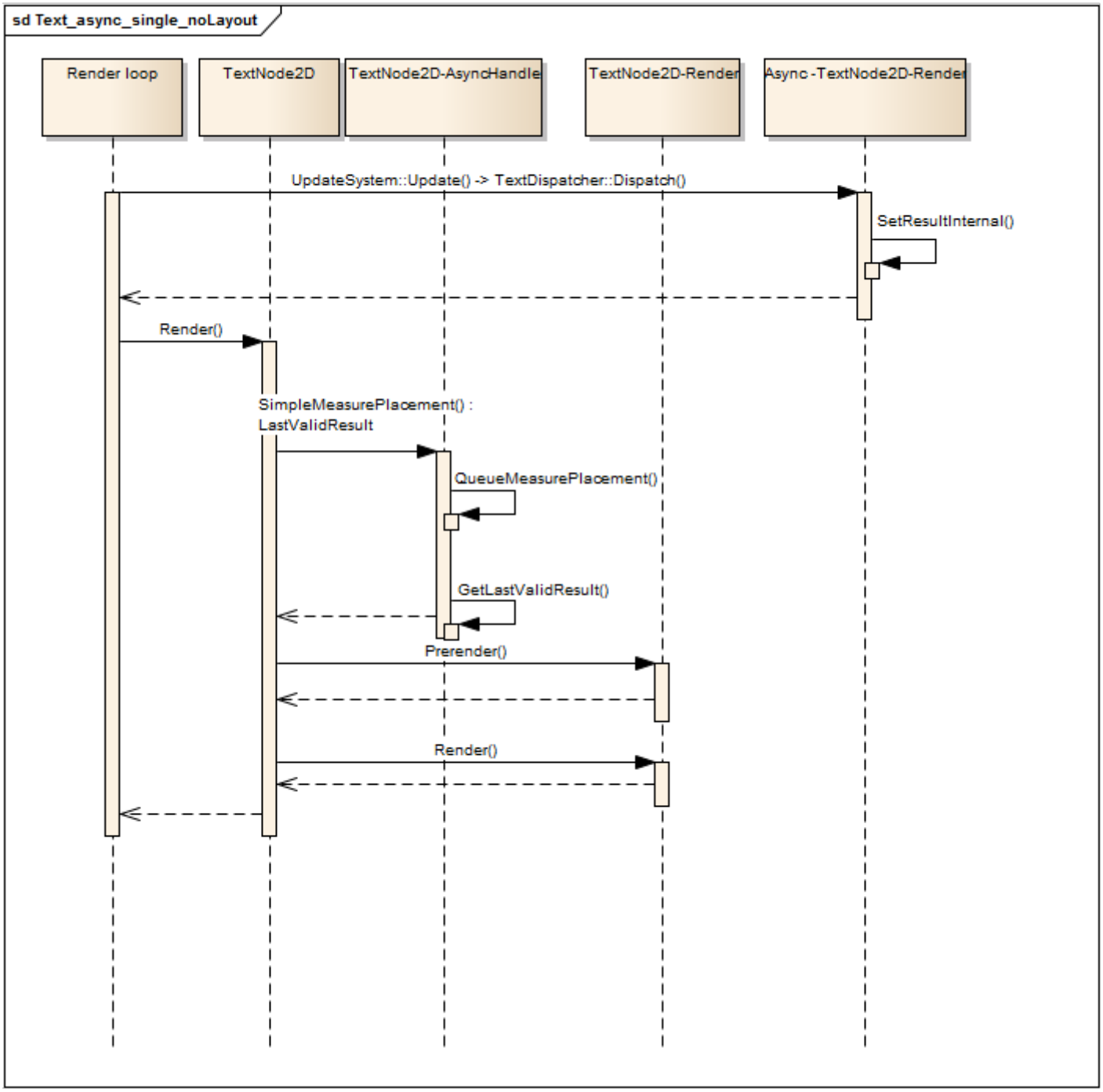


Asynchronous text rendering - Single threaded

Asynchronous does not mean the work load has to be done by another thread. It can also be done by the same thread at another point in time. The main disadvantage of the single threaded version is that the render loop thread is still handling the text. However, this time it does not have to handle every text in a single render cycle. The amount of texts which will be rendered in a cycle can be determined.

This implies that two factors are given: Text/cycle and text to handle. Based on this information the amount of cycles needed to calculate all texts can be estimated. The duration of each cycle cannot be determined, though.

Single threaded and no layout process used (draft)



Synchronization methods

Text-Synchronization

With the introduction of asynchronous text rendering a way to synchronize all text nodes have to be provided. The basic idea is to visually update all asynchronous texts at the same render cycle. It should prevent text popping up randomly. Another aspect is to decrease layout processes. When a layouter is attached to a Candera::TextNode2D, the layout process will evaluate the size of a

Candera::TextNode2D. This also means the layout process starts the asynchronous text rendering if necessary. When the layout process starts the process, it should not wait for the result as this results in the same blocking issue as before.

The layouter uses the old information and text which is still valid as long as the asynchronous text rendering is not done yet. Rerunning the layout process will check whether the new text is valid yet or not. If not, it will continue using the old information. If the new text is valid now, it will update the text. The same procedure happens without a layouter attached. However, the part of starting and collecting the results is done by the render call itself. In short an asynchronous text rendering has to be started and the result has to be received in a minimum of two calls. It depends on the usage of layout procedures who the caller really is.

This section describes the possibilities of synchronization:

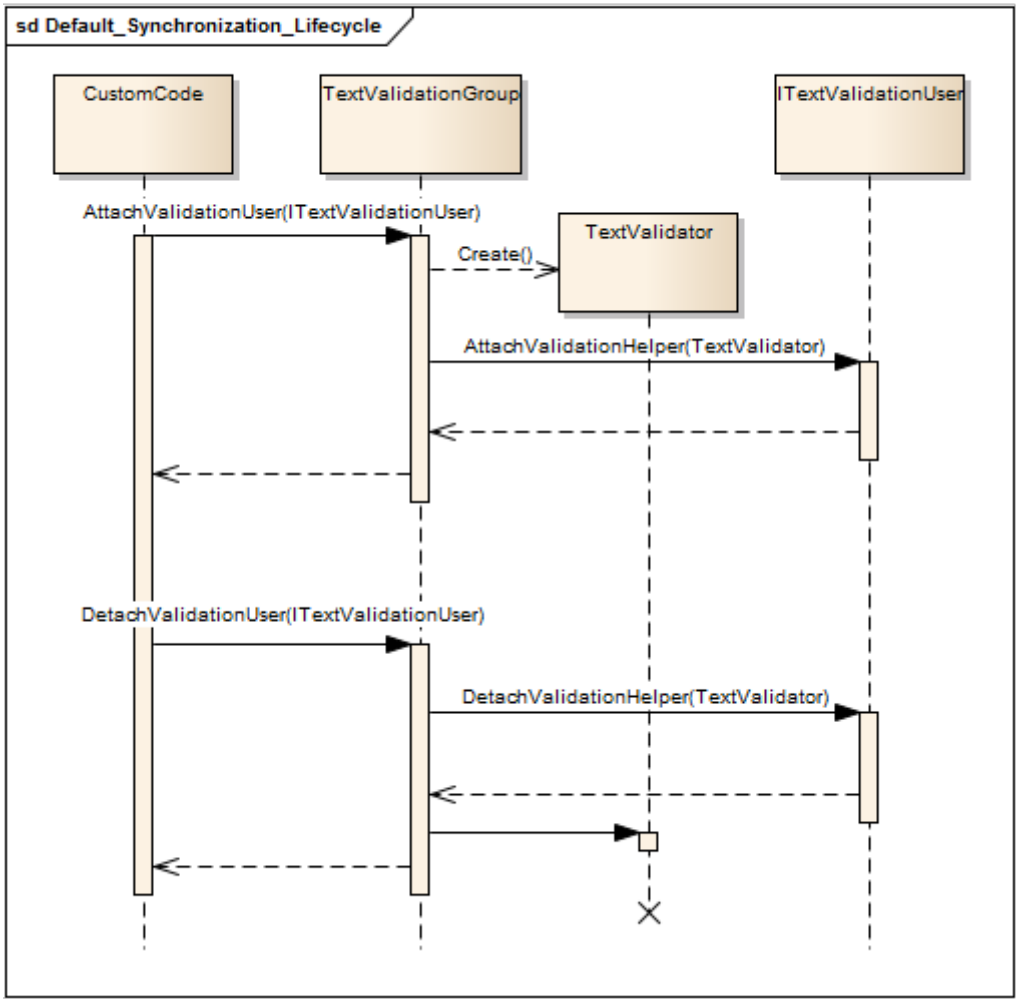
Text-Synchronization – Default synchronization

The default synchronization is kept as simple as possible. This section describes the lifecycle of the validation and how does the synchronization works including its flaws.

Lifecycle

Every object which has to be synchronized implements the interface Candera::TextRendering::ITextValidationUser, e.g. TextNode2D. In the next step this object has to be attached to a Candera::TextRendering::TextValidationGroup. The default group is a singleton instance. The Candera::TextRendering::TextValidationGroup creates a specific validator and attaches it to the object. Detaching the validator is comparable to this process.

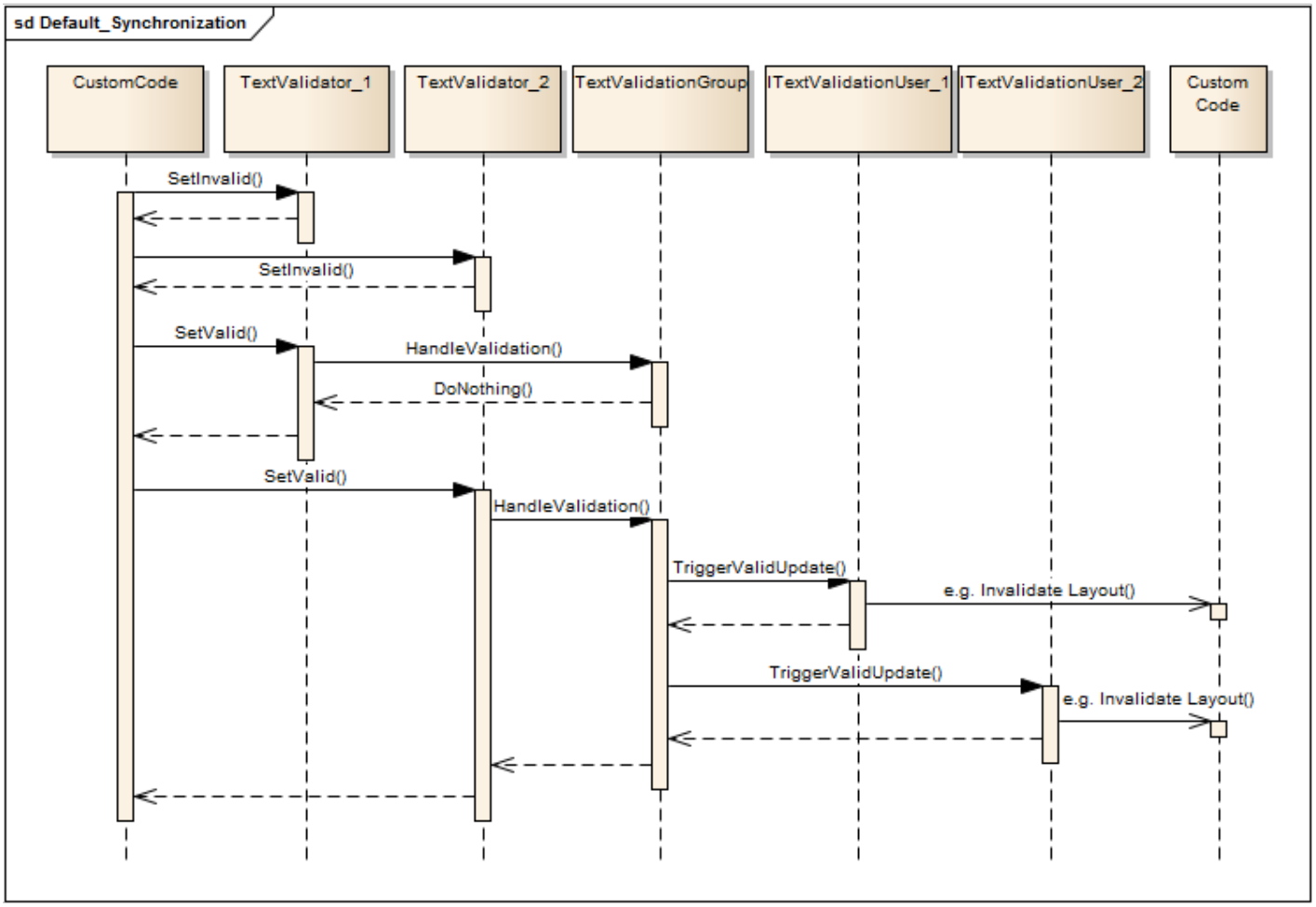
Lifecycle of the default synchronization



Synchronization - flow

The object can be invalidated at any time by calling the invalidation method of the Candera::TextRendering::TextValidator. Every time an invalidated object changes it state to validate, the Candera::TextRendering::TextValidator notifies the group of its validation change. If all objects of the group are valid now, all update methods of these objects will be called.

Draft of how a synchronization group is working



In case of the Candera::TextNode2D with an attached layouter, the layout of the scene in which the Candera::TextNode2D lies will be invalidated. If there is layout process used at all (disabled Candera layout functionality) nothing will be done in the update method. However, the render method itself can handle this case when it is newly called. The issue here is that it is not explicitly specified who will trigger the render. Both render and layout mechanisms of will check whether the whole group is valid or not. Only in case that the whole group is valid, the text will be updated. If the layout is triggered by another object despite the fact that the group is not valid, the text will not be updated.

> Candera::TextNode2D attaches itself to the Candera::TextRendering::TextValidationGroup singleton instance per default. To exclude it from the synchronization group it has to be explicitly detached first.

Synchronization - Restrictions

The default synchronization is as lightweight as possible and still provides the feature that all texts in the group have to be valid until the text can be visibly updated. Due to the variety of combinations a text can be used, it is not a universal solution for synchronization. Therefore in some cases it is not as practical as it should be.

A Candera::TextNode2D can be synchronous or asynchronous or switch between both cases at runtime. Therefore it is not advisable per default to attach/detach the TextNode2D to the group when the asynchronous flag changes. Pending requests are one of the major issues here.

A synchronous Candera::TextNode2D has to be synchronous. This implies when the text has been changed the text has to be visible in the first render cycle afterwards. So it becomes deterministic that the text is truly displayed. Using the synchronization groups on these nodes is opposing this determinism. The node is attached to the group but it is always in the valid state. The node will be ignored completely by the default synchronization.

Another critical restriction is the update rate of a text. This is an issue that happens even without a synchronization but will be intensified by it. In short, every time a text changes, it invalidates the group. When the invalidation rate is higher than the overall text render process of the whole group needs to render, text which should have been shown will never occur. If two texts are invalidating itself in a higher rate, it can also happen that whole group is always in an invalid state. The group itself has a mechanism to count already finished texts and is therefore able to update the text based on the fact that the group is valid or has pending valid texts which are not displayed yet. Based on this information it can recognize a valid group even if one of the texts already changed back to invalid due to setting a new text. However this is not a guarantee that the texts will be updated correctly.

Because of the simplicity of the synchronization it cannot link text updates to each other. For example changing all texts at once to another language is working. There is only one issue with already pending text changes. It is not possible to generally specify which text changes are related to each other and have to be changed at the same time.

Not every use case of a customer can be satisfied by a single way of synchronization. One way produce other side effects than another.

Therefore, own synchronizations can be implemented or used.

Text-Synchronization – Custom implementation

The customization of the synchronization has several levels of complexity and several different goals.

> Changing the way of synchronization during runtime should only be done, when the affected text object (e.g. Candera::TextNode2D) has no pending text changes – as this

> can have unexpected side effects.

A new synchronization group

The simplest customization is an additional synchronization group. As default, all TextNode2Ds are attached to the singleton instance of the Candera::TextRendering::TextValidationGroup. It is possible to create an additional instance of this class in custom code. Now two steps have to be done to correctly bind a Candera::TextNode2D to the new group.

First, detach the Candera::TextNode2D from its old group by using the Detach-method of the old Candera::TextRendering::TextValidationGroup.

Second, attach the Candera::TextNode2D to the new group by using the Attach-method of the new Candera::TextRendering::TextValidationGroup.

Additional trigger objects

A common issue is that setting the underlying scene layout of a Candera::TextNode2D to invalid is not enough to trigger the actual render cycle. For example a render wake-up mechanism needs a trigger to wake up and start the render cycle, too.

The Candera::TextNode2D itself does not know which additional trigger are needed because this depends on the given application.

An extension of this would be to update nodes within the scene graph as soon as the group is valid. A simple use case is to set a new scene visible only when all texts in the new scene are valid or a control pops up only when its text in it is valid.

To accomplish this the specified object which is responsible for these calls has to implement the interface Candera::TextRendering::ITextValidationUser. The object can be attached to the Candera::TextRendering::TextValidationGroup equal to the described life cycle of the default synchronization. The Candera::TextRendering::ITextValidationUser::TriggerValidUpdate()-method in it will be called as soon as the whole group is valid and the custom code can be executed.

Additionally, the object receives an own Candera::TextRendering::TextValidator which can influence the time when a group is valid or when the Candera::TextRendering::ITextValidationUser::TriggerValidUpdate() on all objects in the whole group has to be additionally called.

Implementation of an own synchronization

The most customized solution is to write an own synchronization mechanism.

There are two important interfaces for an own implementation:

- Candera::TextRendering::ITextValidationUser implemented by Candera::TextNode2D,
- FeatStd::ValidationHelperBase to be implemented

Candera::TextRendering::ITextValidationUser is the interface which provides the Candera::TextNode2D with any validation method. It correctly attaches and detaches such a validation method. And Candera::TextRendering::ITextValidationUser::TriggerValidUpdate() invalidates the scene layout of the Candera::TextNode2D. The interface FeatStd::ValidationHelperBase provides functionality to validate and invalidate the object. It is a custom implementation how it will work. There is only one restriction here. A synchronously used TextNode2D will always ignore the valid state of the custom implementation. It will call the validate and invalidate methods, though. A Candera::TextNode2D calls the functions as following:

- Candera::TextNode2D starts measure and place: SetInvalid()
- Candera::TextNode2D finishes measure and place: SetValid()
- Candera::TextNode2D checks if it can handle the result: IsValid() (only asynchronous nodes)
- Candera::TextNode2D has handled the result (ready for display): ConfirmValidHandled()

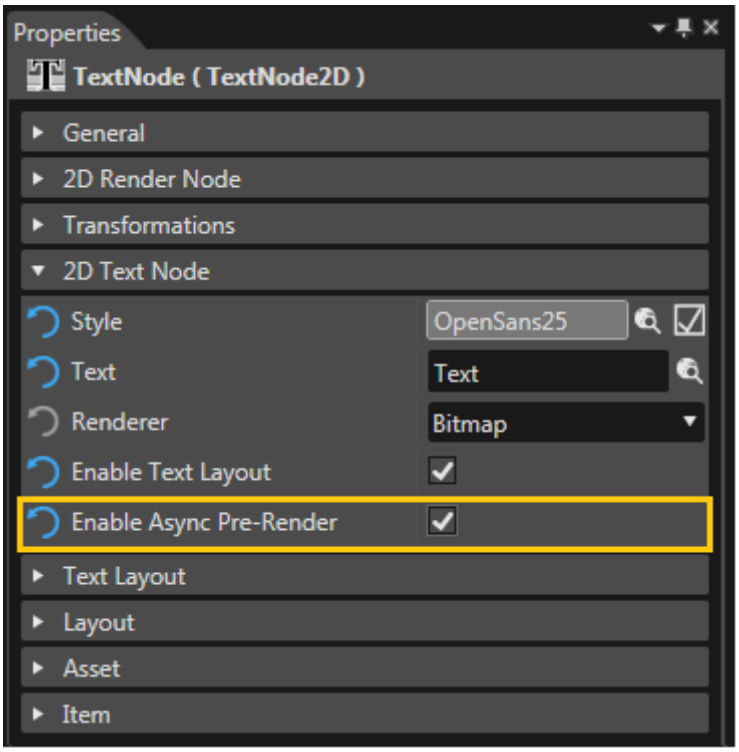
Configuration possibilities

General setup and configuration possibilities

This section describes how to configure the asynchronous text rendering.

Asynchronous TextNode2D and CanvasText

Enabling the asynchronous text rendering is something that has to be done on each Candera::TextNode2D or Candera::CanvasText. This can be done within the SceneComposer:



Or in code itself:

```
void SetAsyncPreRenderEnabled(bool enabled);
bool IsAsyncPreRenderEnabled() const;
```

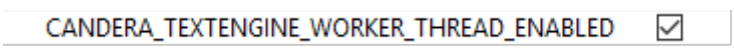
Example usage:

```
TextNode2D node;
node.SetAsyncPreRenderEnabled(true);   //Enables async rendering
node.IsAsyncPreRenderEnabled();        //returns true
node.SetAsyncPreRenderEnabled(false);  //Disables async rendering
node.IsAsyncPreRenderEnabled();        //returns false
```

The flag in SceneComposer and code can be equally configured in Candera::CanvasText.

Asynchronous dispatching methods

CMake provides a flag to choose between the worker thread solution and the single threaded version. When the flag is enabled a worker thread is used. The flag is enabled by default. If threading is not available or thread safety is disabled, the code will fall back to the single threaded version ignoring this flag. The flag is system wide. It counts for all asynchronous TextNode2D. This can also be defined programmatically, whereas there are no thread safety checks.



Dispatcher settings

The TextRenderDispatcher contains a settings object which can be retrieved by calling:

```
class Candera::TextRendering::TextRenderDispatcher::TextRenderSettings{…};

Candera::TextRendering:: TextRenderDispatcher::GetTextRenderSettings();
```

To programmatically change the asynchronous dispatching method use the function of settings:

```
#include <Candera/TextEngine/Async/ AsyncTextRenderDispatcher.h>
#include <Candera/TextEngine/Async/ ThreadingTextRenderDispatcher.h>

void SetDefaultTextRenderer(Candera::TextRendering::TextRenderer * val);
```

If the single threaded version is used, the amount of processed text per DispatchNext can be set:

```
void SetAsyncLoopCount(UInt8 const val);
```

Restrictions

This section describes issues which should be kept in mind when using the asynchronous feature.

Performance improvements

The main idea behind asynchronous text rendering is not to improve the performance of text rendering but to improve the performance of the render loop itself. Even if a second cycle through the render loop is necessary to display the results. The second cycle is mandatory as the first cycle only adds the text render call into a queue. The first cycle uses the old result to handle the text.

Using a worker thread and having a minimum of dual core processor is mandatory to gain an actual performance improvement as the worker thread can be handled by the second core.

High frequency changes

The most mandatory restriction is the update interval of an asynchronous text. Changing the text faster than the text engine can handle will definitely break the determinism of a system. Three different approaches can be outlined for handling fast updated texts. The first approach is to visibly show every text change. It is deterministic to see every text. Even if they are only visible for a single frame and it is not deterministic when the text is shown. The second approach is overriding the old queued text with the new one. The determinism of seeing every text has been lost. However, rendering a text which is already outdated before the render process started is a performance drop which will not occur. The third solution is to render these texts synchronously.

Currently available are the second and the third approach. This decision is based on the possible use cases.

The first use case is an alternating text, e.g. frame rate, clock time. Basically such a text type has three things in common. First, it changes its content all the time in a specified time interval. The logic behind the text is already broken when the update is visible the first time after a single frame

and the second time after six frames. Second, the texts are usually short texts. A high frequency alternation of a long text is most probably an issue in the usability. Third, if the frequency is high enough to outrun itself and the text is alternating all the time, the queue will stack up indefinitely. There is no time to show all queued text until new texts arrive. The visual text itself becomes more and more outdated. The length of the text would increase the change of queuing text even more.

Therefore, it is recommended for cyclic alternation to render them synchronously. Short texts will not affect the render loop itself that much. And long texts should be reconsidered and checked against its reasonableness.

The second use case is to have occasionally a fast switch between texts, e.g. a text will be shown, but a user input already triggered the next text update. In this case the text can have any length. The main difference is that the text update does not queue indefinitely and there is most probably not a hard time interval deadline. In this case asynchronous text rendering can be used but rendering both texts can also impact the user experience. In most cases these will be noticeable in long texts. So there will be a visible delay until the outdated text is displayed and another visible delay until the new text is displayed.

Therefore, the asynchronous approach overrides the old text with the new one as soon as possible. If the old text is already rendering, the old text will be displayed. If the old text have not even started rendering, the old text will be replaced. If the old text and the new text both are already rendered and the render loop had no time to show the old text, the new text will be shown. In this case it would be most likely that the old text has only a visibility time of a single frame.

Destruction of a TextNode2D

Destroying a TextNode2D while the attached text is being rendered can lead to undefined behavior.

Layout process calls

Using an asynchronous approach means that there a functionality which starts the asynchronous process and another one which handles the result of the process. Based on the synchronization the check for a valid result can be solved by polling until a result is available.

If the synchronization is done properly for a given use case, no polling will be needed.

However, the asynchronous process requires layout information to start. And the layout process requires the result to place the text node and depending nodes. Therefore, it is required to run the layout process twice now. Two calls are the minimum requirement. If the text has not been finished until the second call, more calls are necessary as this is equal to polling for results.

Cache type restrictions

Some cache types are GPU context bound. Therefore, the upload of rasterized glyphs has to happen in the correct context. Using a worker thread cannot guarantee the upload at the correct location. This might require an additional rasterization and upload per glyph within the render

thread. Expected cache types to have these issues are SurfaceCache and GlyphAtlas.

Candera::CanvasText uses primarily GlyphAtlas. A shared context and the non-threading method is preferred in this case.

Text rendering: Caching/rendering systems

Description

The rasterization of a glyph is a very costly operation. Paired with shaping and positioning algorithms, it becomes even more expensive. One strategy to counter this is a caching and rendering mechanism. A cache is a storage type which maps a key to a specific representation of a value. The transformation of the rasterization into the value of a cache entry might differ from cache type to cache type. Based on the difference an optimal render strategy has to be internally selected. This is the reason why the text renderer and the caching system are strongly related.

Candera supports different caching and rendering systems with different backgrounds and goals.

No Caching

The easiest form is to cache nothing. This strategy has very little performance but requires the least memory. Its strategy is to retrieve a rasterized glyph, blit it to the display and free all the memory content. This is a strategy which is only advised to use on very low end devices. Otherwise it is not recommended.

Bitmap Cache

Bitmap caching is a strategy to store all the glyphs of a single text within a bitmap. When the text or its properties changes, the bitmap has to be generated. This bitmap will be uploaded and stays uploaded. Every time the visual area of the text becomes invalidated but the text has not been changed, it will rerender the bitmap but not generate it newly. The bitmap is generated with a software blit and therefore generates CPU load.

Glyph Atlas

The glyph atlas is an uploaded texture which contains all cached glyphs. If a glyph is not available there, the texture will be updated by adding the rasterized glyph. A text change requires only an update of a vertex buffer which references the glyphs within the atlas. Therefore, glyphs do not have to be uploaded anymore as soon as they are in the cache. The glyph atlas has a learning curve to fill up the glyph atlas. This solution is a GPU load solution.

Glyph Cache

The glyph cache stores the rasterized glyphs. Which then will be uploaded and rendered in every render call. This is typically a performance bottleneck but useful for very specific use cases. It is not recommended to use glyph cache in common use cases.

Surface Cache

The surface cache uses the same glyph storage mechanism as glyph cache but renders it similar to the bitmap cache. However, surface cache does this on GPU side whilst bitmap cache does it on CPU side. A change within a text leads to a draw call for every single glyph. If the new text requires a larger area to display the text, a new FBO will be created. FBO creation is an expensive operation. This cache is made for specific hardware devics. It has more drawbacks than advantages in common use cases.

Cache Usages

This section describes where the cache types are available and what are recommended usages.

TextNode2D:

- Bitmap
- Glyph atlas
- Surface cache (Freetype only)
- Glyph/No cache (Freetype only)
- Glyph cache (Freetype only)

TextBrush:

- Bitmap
- Glyph atlas
- No cache (Freetype only)
- Glyph cache (Freetype only)
- Surface cache (Freetype only)

CanvasText:

- Glyph atlas

The recommended usages are bitmap cache as CPU solution and glyph atlas as GPU solution. The usage depends on the given project. All other cache types are written for specific use cases (e.g. specific devices).