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
- Second case: Layout process but no TextNode2D layouter
- Third case: Layout process with TextNode2D layouter

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, triggering 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.

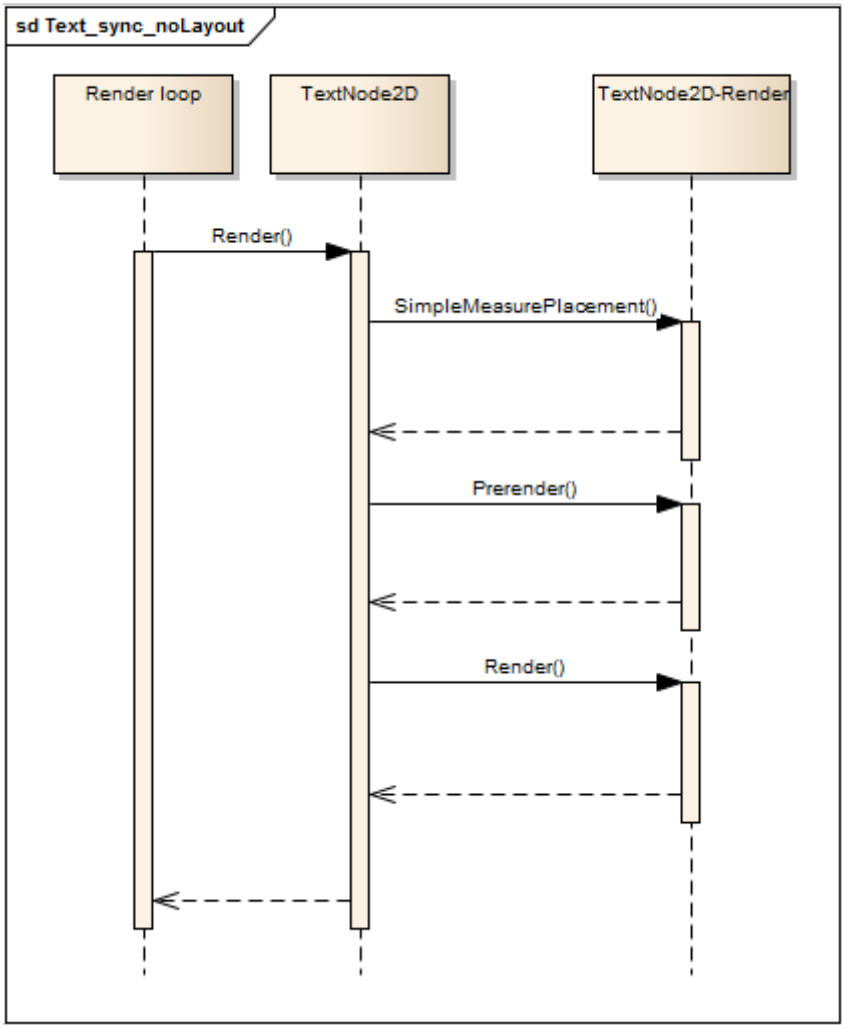
Pros:

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

Cons:

- 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 layouter attached.

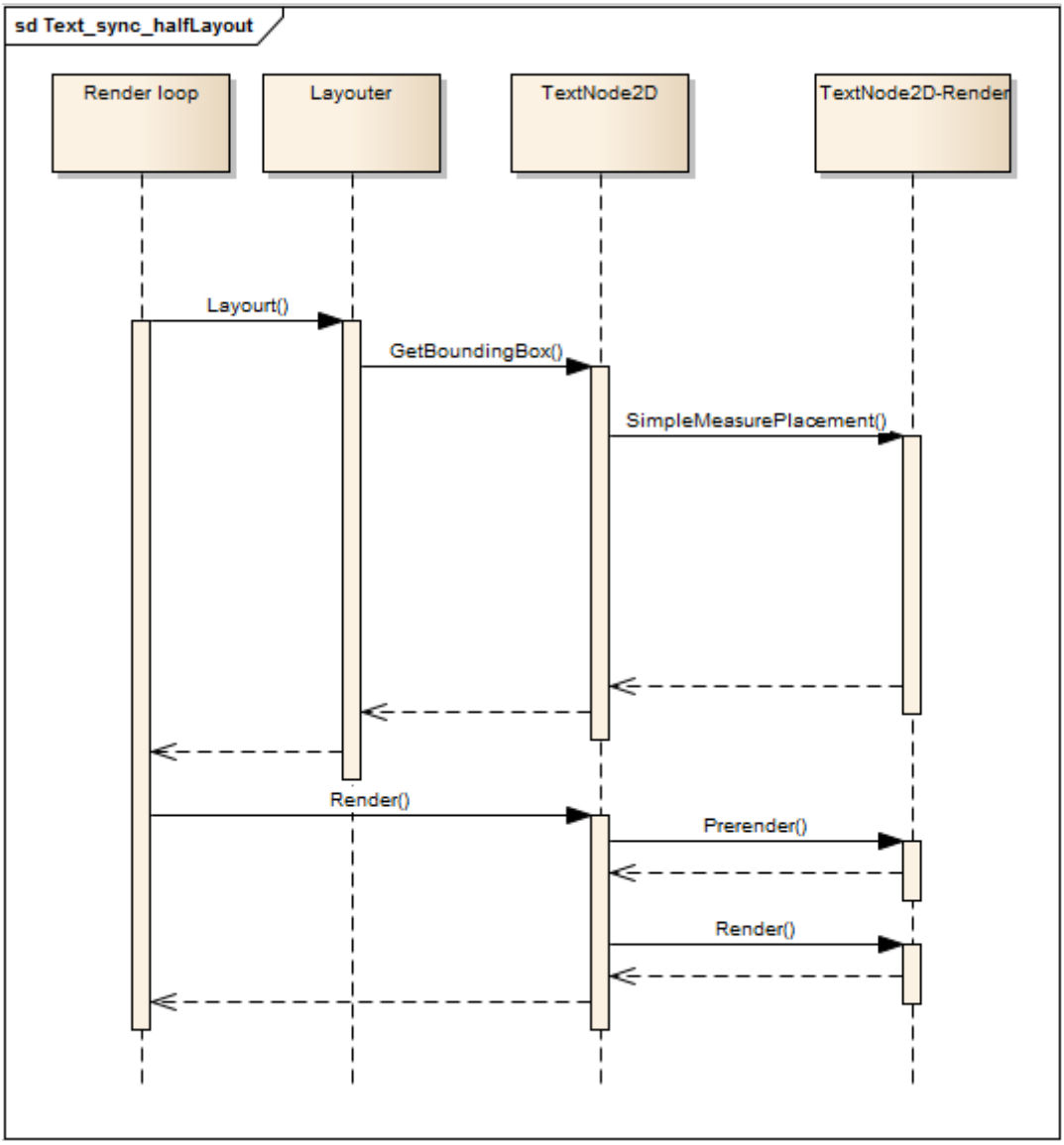
Pros:

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

Cons:

- 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.

Pros:

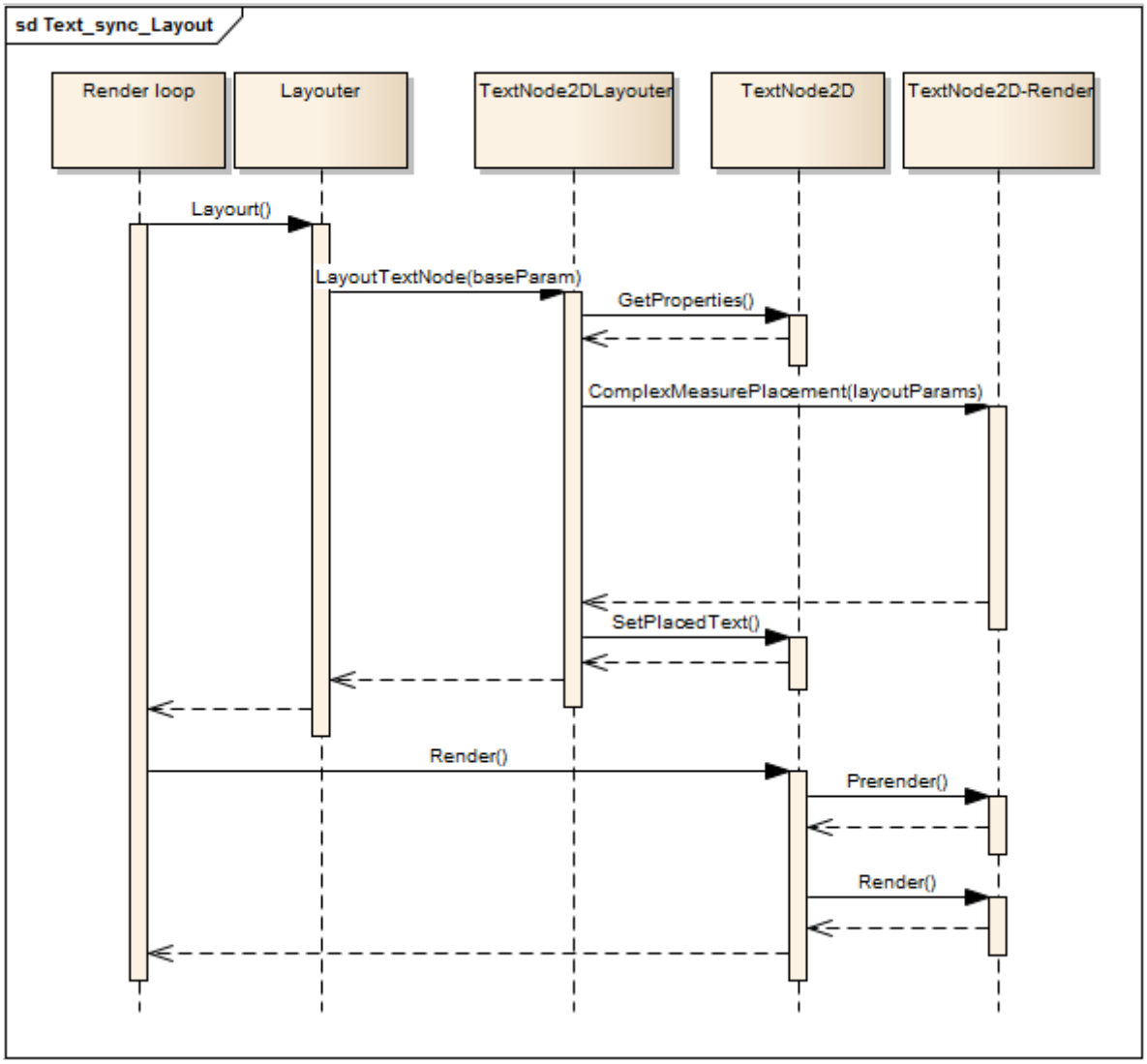
- Full feature set for text placement integrated in the scene graph arrangements.

Cons:

- 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 can 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 a 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.

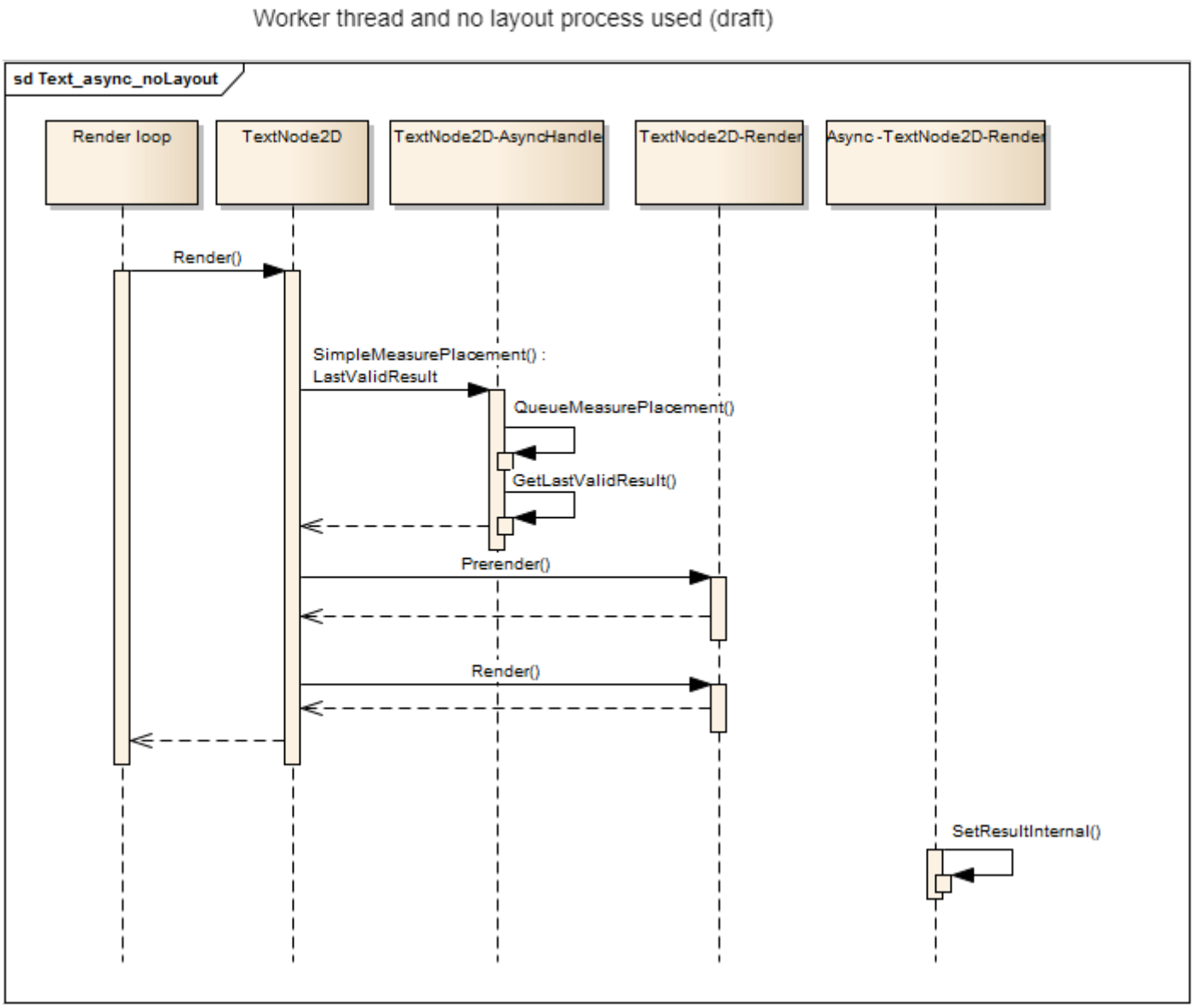
Asynchronous text rendering – Worker thread

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, the strategy without a layouter is used. All other strategies 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.

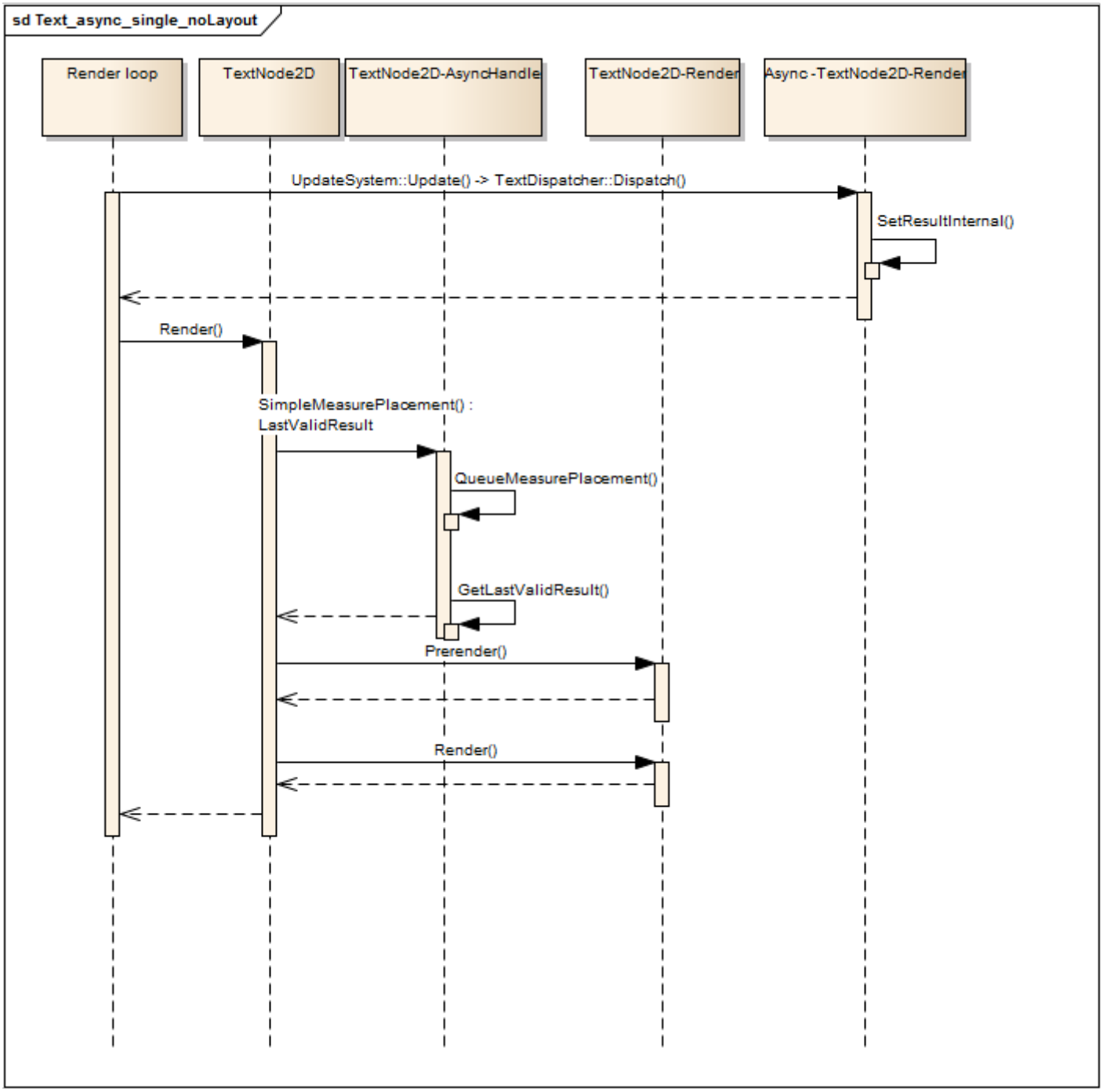


Asynchronous text rendering - Single threaded

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 has 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 causes 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 becomes valid, 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
  - Lifecycle
  - Synchronization - flow
  - Synchronization - Restrictions

- Text-Synchronization – Custom implementation
  - A new synchronization group
  - Additional trigger objects
  - Implementation of a custom synchronization

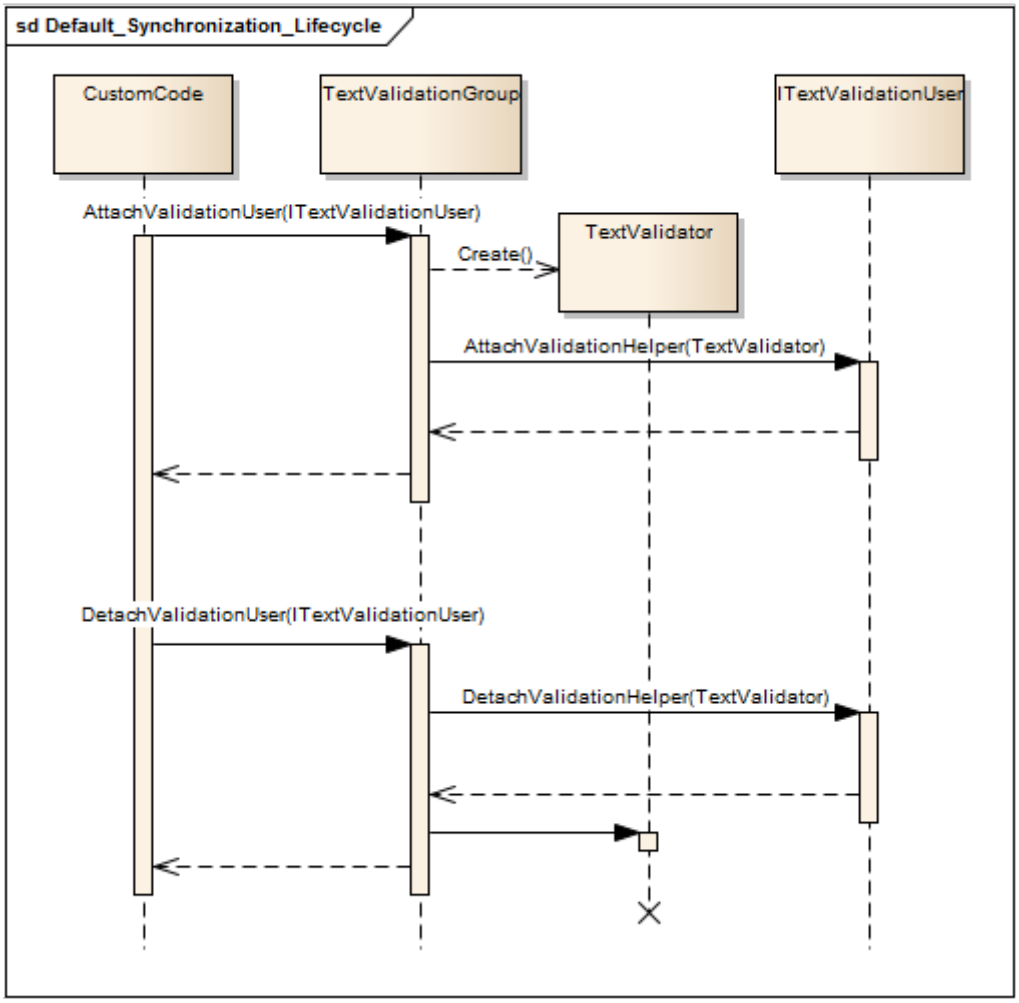
Text-Synchronization – Default synchronization

The default synchronization is kept as simple as possible. This section describes the lifecycle of the validation and how 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 similar to this process.

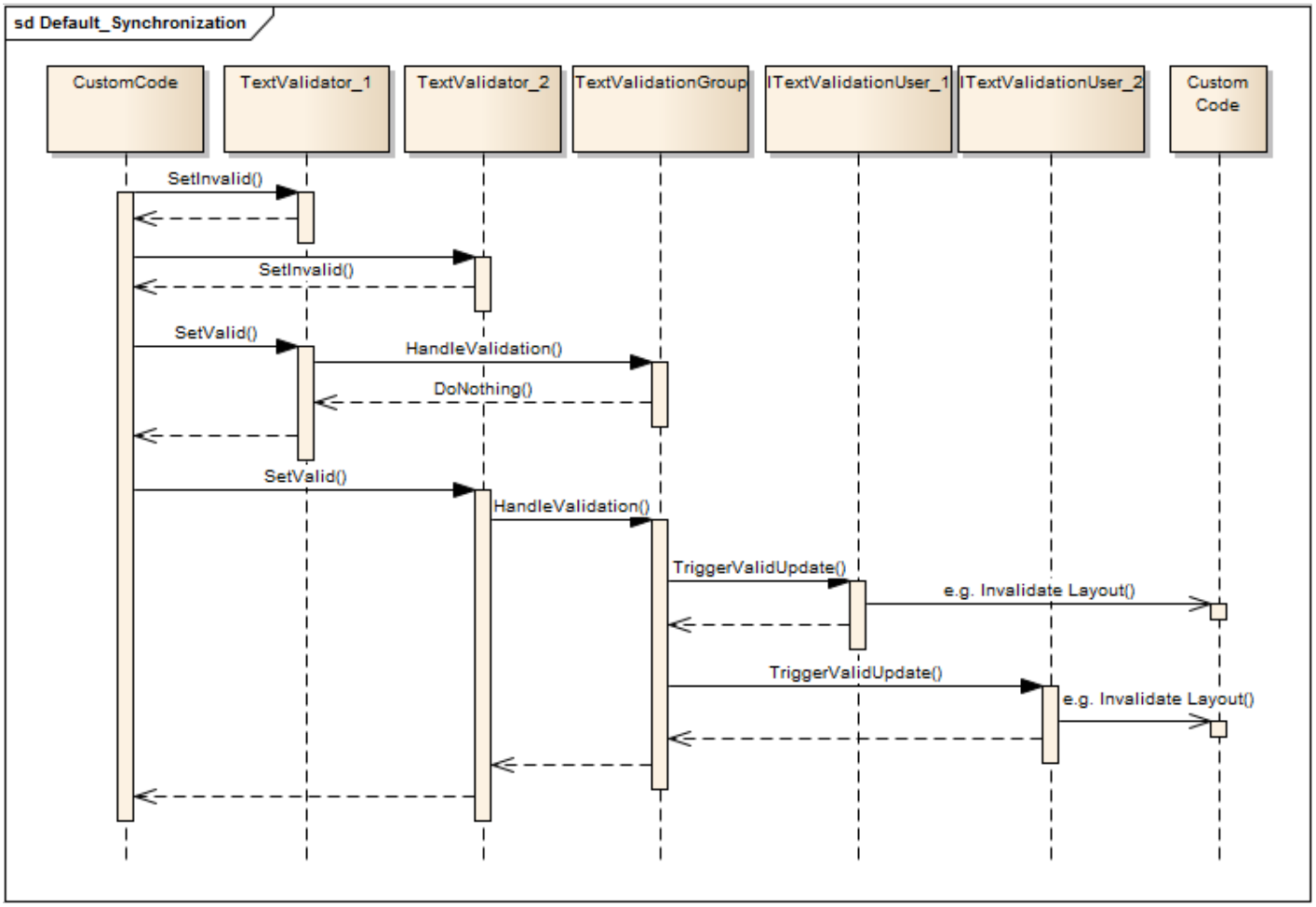
Lifecycle of the default synchronization



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



For the Candera::TextNode2D with an attached layouter, the layout of the scene in which the Candera::TextNode2D lies will be invalidated. If there is no 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 will check whether the whole group is valid or not. Only when 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 a variety of combinations that 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 that when the text has been changed, the text has to be visible in the first render cycle afterwards. So this guarantees that the text is truly displayed. Using the synchronization groups on these nodes breaks this certainty. 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, text which should have been shown will never occur. If two texts are invalidating itself at a higher rate, it can also happen that the 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 strategy produces other side effects in another.

Therefore, custom 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 strategy 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 triggers 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 its 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 a custom synchronization

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

There are two important interfaces for a custom 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. A custom implementation must be provided for it to 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()

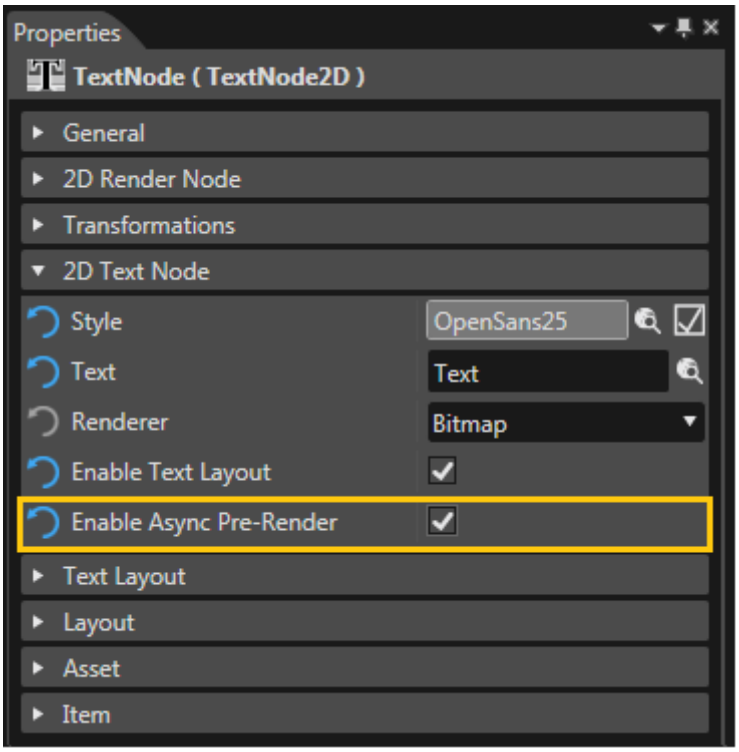
Configuration possibilities

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;
```

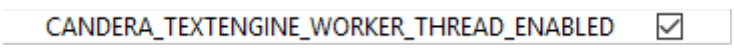
Example usage:

```
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 strict restriction is the update interval of an asynchronous text. Updating the text faster than the text engine can handle will definitely break the system's predictability. Three different approaches can be outlined for handling fast updated texts. The first approach is to visibly show every text change. Every piece of text is guaranteed to be displayed. Even if it only appears for one frame at an unpredictable time. The second approach is overriding the old queued text with the new one. The guarantee 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.

So, the only available approaches 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 has 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 is 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.

Revision #15
Created 2023-02-22 01:46:46 UTC by Tsuyoshi.Kato
Updated 2026-08-19 02:17:58 UTC by Tsuyoshi.Kato