LayoutNG

Emil A Eklund <eae@chromium.org>
Ian Kilpatrick <ikilpatrick@chromium.org>
Last updated: February 27, 2017

Objective

A new layout system for Blink designed with fragmentation, extensibility and interruptibility in mind.

Status

Under active development. The code lives in Source/core/layout/ng in the blink section of the main chromium repository. It can be enabled by passing the "`--enable-layout-ng`" command line flag.

At the moment (February 2019) full blockflow support (milestone D as outlined below) is undergoing final testing and is on for 50% of canary users. Launch target is M75.

Background

The current layout system used in Blink has served us well, however it was designed for a web that's very different from the web as a platform we have today. It's starting to show strain and is holding us back from being able to add new layout primitives and functionality.

Overview

This document outlines a new layout system designed from the ground up to support the requirements of the web as a platform going forward.

It was specifically designed with the following requirements in mind:

- Immutable layout tree structure.
- Low-cost of adding new layout primitives.
- Support for CSS custom layout.
- Native fragmentation support.
- The ability to interrupt and resume layout.

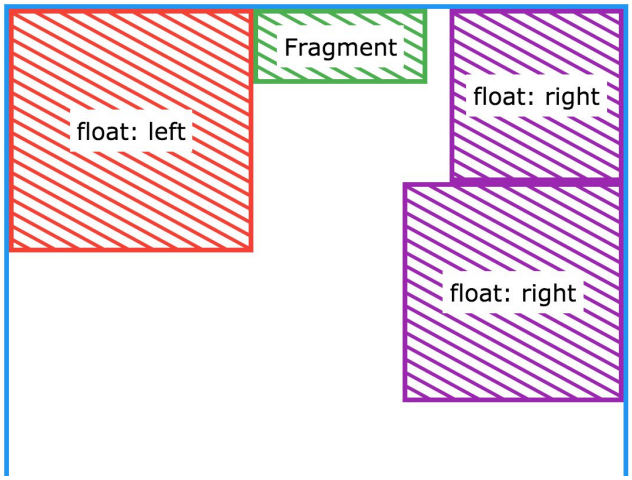
Another key design goal was to support an incremental transition to the new system, essentially allowing the new system to co-exist with the existing one thereby enabling a gradual transition.

Heavily influenced by the current (Spring 2016) CSS Custom Layout [1] proposal which was developed in conjunction with this design.

Detailed Design

Constraint Space

A NGConstraintSpace represents the available space to perform the current layout in.



The figure above shows a NGConstraintSpace with a fixed width, infinite height, with exclusion areas for floats and a positioned NGFragment.

Sizing

The NGConstraintSpace has inlineSize and blockSize attributes, which if defined specify the dimensions in which the layout can perform its layout. If either is infinite (represented by -1), the layout can assume that it has an infinite space to perform its layout in that direction.

In the above example the constraint space has a fixed inlineSize, however an infinite blockSize (representing that it may be in a scrollable Element).

Some layout modes "stretch" their children to a fixed size (e.g. flex, grid). This will be represented with a flag on the NGConstraintSpace for each direction (fixedInlineSize, fixedBlockSize). If true the current layout must produce a fragment which satisfies those constraints.

The NGConstraintSpace will have attributes (inlineSizeForPercentageResolution, blockSizeForPercentageResolution) for resolving percentages against. See "Additional Constraint Spaces" below for why this is desirable.

Exclusions

The NGConstraintSpace has a list of exclusions (NGExclusion) which specify which areas the layout should not position *some* children within. These exclusions are used to represent floats, shape-inside/outside, and NGFragments which have been placed in the space.

Each NGExclusion will have a type of "Float" or "Normal". This distinction is needed as some children (block level boxes inside a different formatting context) ignore float exclusions.

You will be able to query the position of all of the current exclusions for handling things like nested floats. Nested floats need the ability to align their top with the top of the bottom-most float. A nested float will:

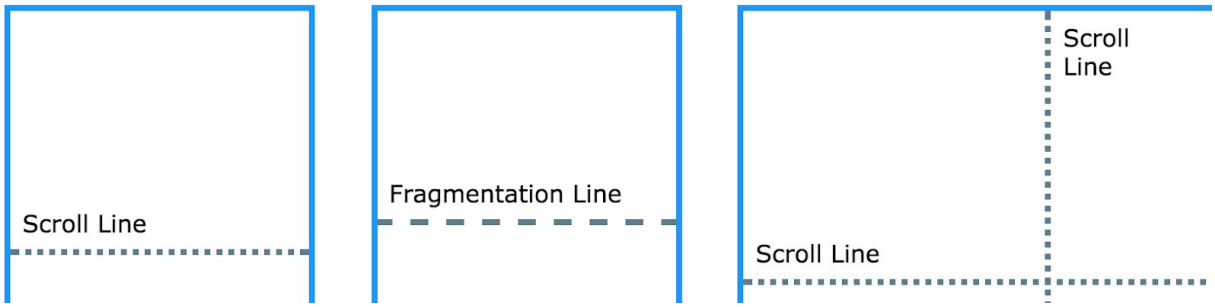
- Query the constraint space for the "Float" NGExclusion with the lowest top.
- Add a "Float" NGExclusion which aligns its top with that exclusion if possible (in accordance with standard float positioning rules).

This allows arbitrary siblings in the layout tree to communicate the position of their respective floats.

Most NGExclusions will just be of type NGExclusionRect, i.e. rectangular in shape. To support CSS Shapes[2] there will also be NGExclusionShape which an arbitrary shape.

Scrolling & Fragmentation

The NGConstraintSpace will have a series of lines representing the scroll trigger offset (in both the inline and block direction) and well as the fragmentation trigger offset and type (only in the block direction).



The figure above shows three NGConstraintSpaces; (1) a single block scroll line, (2) a block fragmentation line, (3) two scroll lines in both the inline and block direction.

The NGFragment contains two rects. One is it's "border-box" rect (inlineSize, blockSize), the other is it's overflow rect (inlineOverflowSize, blockOverflowSize). An NGFragment's rect *should* not be larger than the NGConstraintSpace allows, but the overflow rect may be any size. Some layout algorithms take the overflow rect into account when positioning (block-flow), while others do not (flex).

If the current layout will produce a NGFragment which will produce an overflow rect which will be larger than the scroll line, it can finish its current layout early with a "Trigger Scroll" flag.

If a parent layout receives a NGFragment with this flag, it *must* perform layout on the child again with a new constraint space which allows space for the scroll bar. This new constraint space will not have the "scroll trigger offset" on it. (It must perform layout again as the child may have run an incomplete layout).

With this system it is possible to do three layout passes for a block (one for triggering first scroll line, two for triggering the next scroll line, and finally one to produce the correct NGFragment). However we expect this will have different performance characteristics to the current layout system for two reasons.

1) It is possible for a layout to early opt-out, meaning that we only need to calculate the information to determine if we need to scroll in a direction on the first pass.
2) The immutability of the tree means that we can potentially aggressively re-use results from a previous layout pass. E.g. it should be possible to re-use NGFragments from the incomplete layout passes if the child NGConstraintSpaces match.

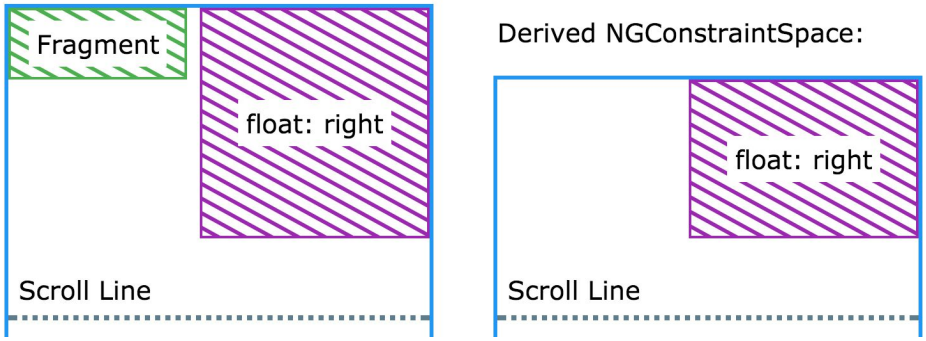
For fragmentation (see "Fragmentation & Break Tokens" below) the layout must produce a fragment which fits inside the area defined by the fragmentation line. If possible the current layout should fragment, instead of producing an atomic fragment. (It may be impossible for the current layout to fragment, e.g. an image element).

The fragmentation line will also have a type. This will can be "page", "column" and "region". This information can be used for avoiding fragmenting certain types of CSS Boxes (with the "avoid-break" property for example).

This won't be in the initial system but can be added later once we want to support CSS Fragmentation[3].

Additional Constraint Spaces

A new NGConstraintSpace can be created off an existing NGConstraintSpace to carry over exclusions, scroll/fragmentation lines, etc.



The figure above shows a NGConstraintSpace (left) and a NGConstraintSpace representing the next block level layout opportunity (right).

This is used for giving simplified NGConstraintSpaces to children. Derived NGConstraintSpaces (aka. Layout opportunities) can be generated from an existing NGConstraintSpace for the next block level layout opportunity or inline level opportunity.

When creating a derived NGConstraintSpace for the next inline level layout opportunity, inlineSize and blockSize of the NGConstraintSpace may be different from its parent. This is why having additional inlineSizeForPercentageResolution/blockSizeForPercentageResolution members are important, it allows smaller NGConstraintSpaces while allowing percentage resolution to work correctly.

The layout opportunities will be exposed through a generator style interface. There will be three arguments for producing the generator:

1) "clear" - This is similar to the "clear" CSS property except it applies to all types of exclusions. You will be able to clear "float-left", "float-right", "fragment". As an example to get the next block level layout opportunity which clears right floats, you'd pass ["float-right", "fragment"].
2) "avoid" - This argument is used for producing derived NGConstraintSpaces which don't contain a particular exclusion type. As an example inline layout opportunities should avoid "all".
3) "step-size" - This argument is used for non-rectangular exclusions, e.g. CSS Shapes. This is the amount to "step-size" by for the next NGConstraintSpace.

Exposing derived NGConstraintSpaces through a generator should allow efficient querying instead of calculating all possible derived NGConstraintSpaces.

Logical & Physical Units

To aid in readability and to ensure proper conversion between coordinate systems a set of dedicated units will be used to represent offsets, locations, sizes, and rects for each coordinate system where they are applicable. Conversion between coordinate systems must be done explicitly and no implicit conversion will be allowed.

Logical Units

NGLogicalOffset and NGLogicalSize represents an offset or a size. Each has a pair of LayoutUnits [11] that

represents the offset or size in the inline and block directions.

NGLogicalRect represents a rectangle and contains an offset and a size, represented by the two units described above.

These are all in a logical coordinate space in the sense that they do not take writing mode or directionality into account.

Physical Units

NGPhysicalOffset, NGPhysicalLocation, and NGPhysicalSize represents an offset, location or a size. The location unit is different from the offset one in that it represents the resolved location from the the top of the tree while the offset unit represents the delta from the parent. Each has a pair of LayoutUnits that represents the offset or size in the horizontal and vertical directions.

NGPhysicalRect represents a rectangle and contains an offset and a size, represented by the two units described above.

These are all in a physical coordinate space in the sense that they represent coordinates in a top-to-bottom, left-to-right coordinate system. Converting from a logical unit to a physical one thus requires the coordinates to be resolved. The *Writing Modes and Coordinate Systems* section goes into more details.

Precision & Snapping

Regardless of the coordinate system used all values are in CSS pixels and represented as LayoutUnits. A LayoutUnit is a fixed-point unit with a precision of 1/64.

For painting locations and sizes need to be snapped to pixel grid to ensure crisp rendering. This process is called pixel snapping and involves translating from CSS pixels to device pixels. Unlike rounding the location and size or computing the enclosing rectangle pixel snapping produces a rectangle aligned to pixel boundaries as close to the original location and size as possible.

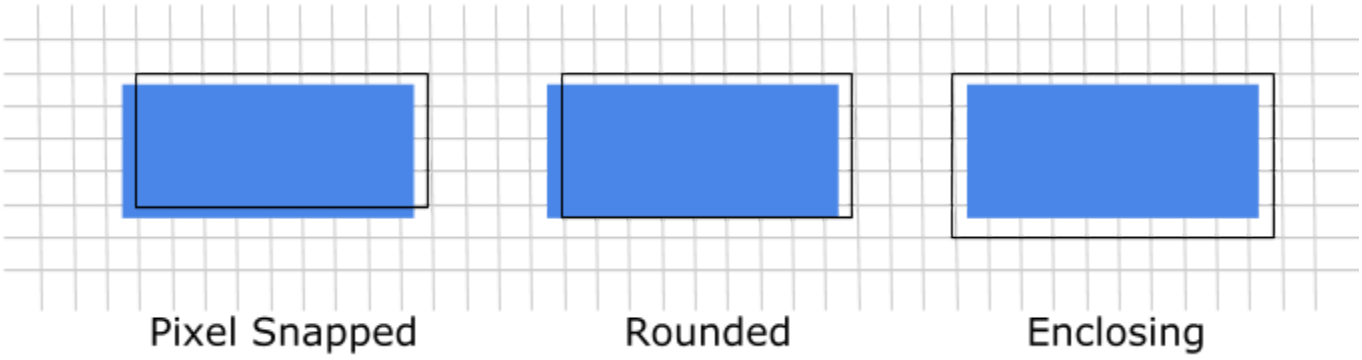


Figure comparing the pixel snapping, rounding, and enclosing translation strategies.

Pixel snapping ensures that each edge and the size is off by at most one pixel. More importantly it ensures that rounding errors do not accumulate as the location is taken into account.

Imagine two columns; the first one has a single element that is `105px` tall; the second one has ten elements, each 10.5px tall. If the first element starts at a `top: 0px` it would be pixel snapped to a height of `11px`. The next element would has top: `10.5px` and is snapped to a height of `10px`. The process then repeats. This way the total error for the entire column is at most `1px`.

If either of the other rounding strategies was used instead each element would have the same size (`10px` or

`11px`) but the errors would accumulate and the second column would be `100px` or `110px`.

Pixel Snapping Logic

When snapping the left and top edge is simply rounded to the nearest device pixel. The right and bottom edges are computed by subtracting the rounded left/top edge from the precise location and size. This ensures that the edges all line up with device pixels and that the total size of an object, including borders, is at most one pixel off.

The values are computed as follows:

```
left: round(left)
top: round(top)
right: round(left + width)
bottom: round(top + height)
width: round(left + width) - round(left)
height: round(top + height) - round(top)
```

Snapping in LayoutNG

In the legacy layout system this translation logic is not well contained and is sprinkled throughout the layout and paint code. In LayoutNG there will only be one way to do pixel snapping and again a unique unit will be used to represent a pixel snapped rect to ensure that the right type of unit is used at all times.

The process of pixel snapping will take a NGPhysicalRect and return a NGPixelSnappedPhysicalRect. Unlike all other coordinate units used in LayoutNG the NGPixelSnappedPhysicalRect does not use LayoutUnits. Instead it has four integer fields representing the top, left, width and height in physical device pixels.

```
NGPixelSnappedPhysicalRect NGPhysicalRect::snapToDevicePixels()
```

Writing Modes & Coordinate Systems

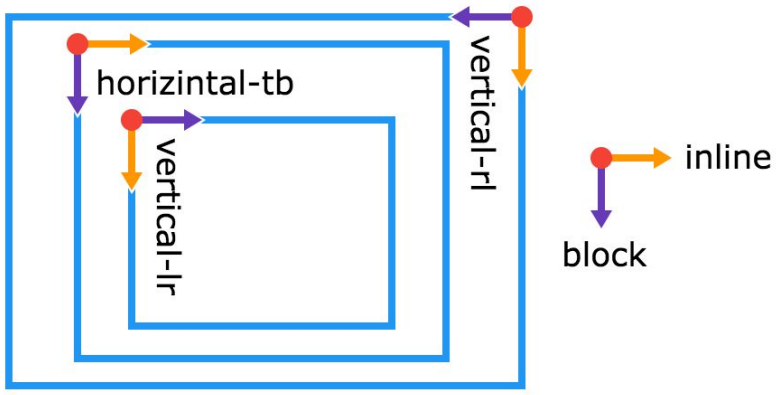


Figure shows the relationship between inline and block direction. Where inline is horizontal and block is vertical in a top-to-bottom writing mode.

LayoutNG will mainly use a logical coordinate system, as described above, for the purposes of placing content. Fragments includes an origin and a direction which represents the desired anchoring and directionality of the fragments children.

For text nodes the origin and direction applies to the text content of the node itself and represent the text

directionality and the writing mode.

The reason behind this is to simplify the logic and reasoning. By supporting directionality and writing modes through a transformation stage all combinations can be supported with little overhead and without the error-prone special-case logic employed today.

During paint or when querying the layout tree (for clientRects, etc) a set of convenience methods are provided that gives the physical location or size of a NGFragment relative to its parent.

```
NGPhysicalLocation NGFragment::getPhysicalLocation(const NGFragment* parent) const;
NGPhysicalSize NGFragment::getPhysicalSize(const NGFragment* parent) const;
```

Layout Tree

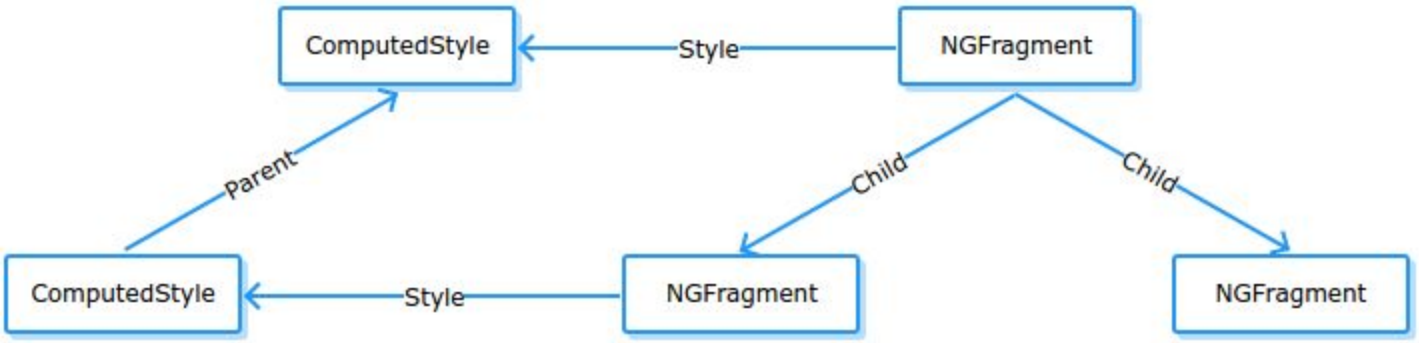


Figure shows the style tree (left) and layout tree (right) and their relationship.

The layout tree is made up entirely of NGFragment and NGText nodes and is immutable. Each node includes a back pointer to the computed style, a list of children and the size and location of the node itself. The location is relative to the immediate parent.

Given the immutability of the tree, use of relative coordinates, and the lack of a parent pointer entire subtrees may be reused across layouts. This concept is discussed in detail in the subsequent layouts section.

Tree Construction

Tree construction is done using builders, separating the construction logic from the resulting representation. The tree is constructed from the style tree and during the tree building stage a NGBuilderCoordinator object, containing a stack, is created.

The coordinator primes the stack with the appropriate type of NGBuilder, passing it the Node associated with the style tree root and an initial NGConstraintSpace. The coordinator then processes the stack: the most recent builder is inspected. NGBuilders corresponding to its Node's descendants are placed onto the stack in depth-first order.

When a Node has no children, its NGBuilder creates zero or one NGUnplacedFragment, which is passed to its parent. As needed the builder is called again until all content associated with the Node has been consumed, each time creating at most one NGUnplacedFragment. The builder is responsible for maintaining the consumption state. In addition to an NGUnplacedFragment the builder may also return a NGBreakToken representing the relationship between NGUnplacedFragments associated with the same Node.

Its NGBuilder is popped from the stack. Then the depth-first walk continues by processing the completed node's siblings and their descendants. When all the children of a Node have created their

NGUnplacedFragments and passed them up to the parent, the parent has everything it needs to lay itself out and does so.

Figure shows the differences between NGUnplacedFragment and NGFragment, namely the addition of a location field in the latter.

During this process the child NGUnplacedFragments are assigned a relative x and y coordinates and then transformed into NGFragments before being added to children collection of the NGUnplacedFragment for the node. NGFragments are completely immutable. The NGUnplacedFragment for the node is then passed up to its parent.

Its NGBuilder is also popped from the stack. The depth-first walk ends when there are no more NGBuilders on the stack, per usual.

Memory Management

The NGLayoutTree, composed of NGFragment and NGText nodes, will be ref-counted as the number of expect objects i very high (5-10x the number of LayoutObjects), the life-time long and the life-cycle simple. This decision was reached in conjunction with the oilpan (blink garbage collector) and blink system architecture teams.

NGBuilders will be stack allocated during layout and thus their life-time is limited to a single layout.

NGInput objects will be garbage collected using oilpan. This will simplify memory management and reduce the risk of memory leaks or stale pointers. Given that the way these objects are created will change throughout the phases of the project this is a significant benefit.

Fragmentation & Break Tokens

Each CSS Box (an Element, TextNode, ::before/::after pseudo element) can produce multiple fragments. The most common case where this occurs is in the inline direction when a TextNode being splits up into multiple lines. Multi-col layout and pagination also create fragments in the block direction.

The NGBreakToken is used to "resume" layout of a CSS Box, it is passed to the constructor of the NGBuilder.

Each NGFragment may have a NGBreakToken. The NGBreakToken is used for producing the next NGFragment for that CSS Box. If a NGFragment doesn't have a NGBreakToken that CSS Box doesn't have any more fragments to produce.

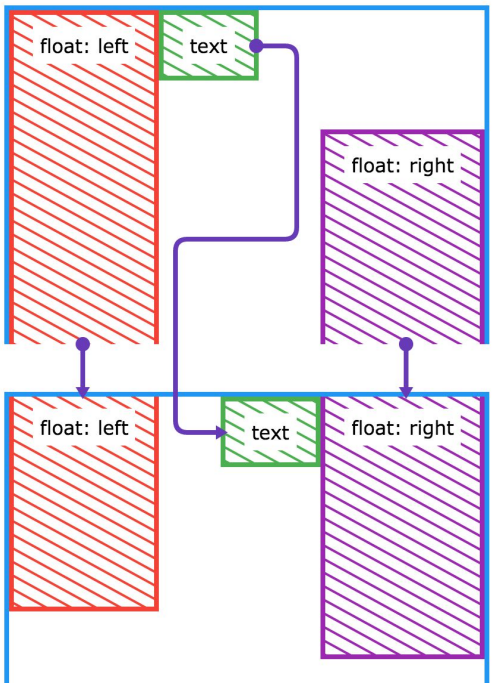
A NGBreakToken represents the "point" inside the CSS Box where it last fragmented. The NGBreakToken may contain:

- The text offset from where a TextNode fragmented.

- The child offset from where an Element fragmented.
- A list of child NGBreakTokens to "resume" child layouts.
- The type of break which occurred.

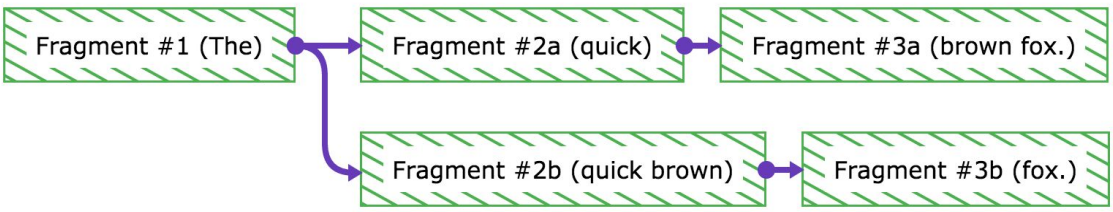
Figure showing the class diagram for a NGBreakToken.

In the pagination example below the NGBreakToken for the first "page" would contain three child NGBreakTokens. The second "page" would have no NGBreakToken as it didn't fragment.



The figure above shows a pagination scenario, where the parent NGBreakToken will contain three child NGBreakTokens (for the left float, right float, and the text fragment).

A layout may only return a set of NGFragments which have a consistent NGBreakToken chain. In the example below returning the fragments [1,2b,3a] would be invalid. An invalid break token chain would break selection and hit testing and is therefore explicitly disallowed.



The figure above shows possible fragments for the text "The quick brown fox." Valid NGFragments to return are [1], [1-2a], [1-2b], [1-2a-3a], [1,2b,3b]. Other combinations are invalid.

Minimum & Maximum Content Calculation

Builders may provide a method for computing the preferred content size. If no such method is provided the same layout algorithms as per normal layout is invoked.

To compute the maximum content size in this mode a the layout algorithm is called with a NGConstraintSpace with infinite inline/block size. For minimum content a NGConstraintSpace with zero inline/block size is used.

The member inlineSizeForPercentageResolution is used for correctly resolving percentages in this mode when the inline/block size is infinite or zero.

Inline builders, especially when text is involved, are likely to provide a custom preferred content size implementation as that allows the minimum and maximum content size to be computed in a single pass.

Containing Blocks & Absolute Positioning

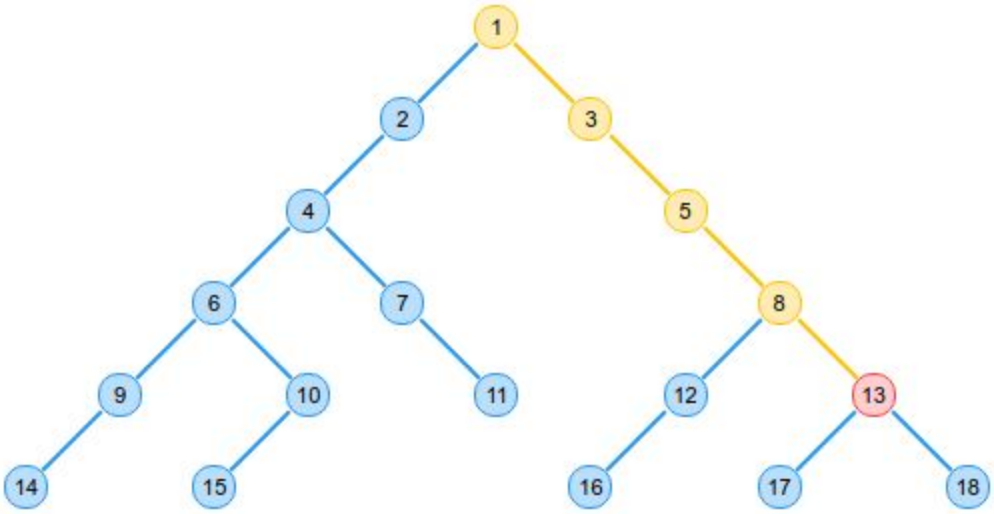
A NGFragment may return a list of child CSS Boxes which should be positioned by a parent NGBuilder. For example all of the children which need to be absolutely positioned.

A nice property of this is that if the containing block or the actual abs. pos. child changes, the NGFragment can still be reused as it only contains information about the CSS Box.

The list of child CSS Boxes will be represented as a location in the DOM tree E.g. three members (depth, offset, pseudoType) which uniquely describes the location of the CSS Box in the tree.

Subsequent Layouts

For subsequent layouts as much of the tree as possible is retained however as it is immutable at a minimum the NGFragments associated with the Nodes that changed will need to be reconstructed together with the spine all the way up to the root.



The figure above shows a tree of NGFragments where node 13 needs to be recreated.

In the illustration above the node with the number 13 changed. Subsequently the path up to the root (node 1) is identified and NGBuilders are created and pushed onto the NGBuilderCoordinator stack. This in turn creates a new set of NGUnplacedFragments for each node, starting with 13, and passes them up to the immediate parent for placement. Note however that the NGFragments for node 17 and 18 are reused. As are all fragments on the left side of the tree.

See the Fragment caching in LayoutNG [12] design doc for details on how this works in practice.

Hosting NGLayout in a Legacy Tree

The process described above outlines the eventual end goal of entirely replacing the current (henceforth referred to as legacy) layout system. During a transition period, however, the two system will coexist. As such the legacy tree needs to be able to host a NGLayout subtree (hosting) and the other way around (reverse hosting).

Initially NGLayout subtrees will be created from the legacy layout tree. In this model the layout phase is modified to add a pass that identifies suitable subtrees for LayoutNG. Once such a subtree has been identified it's marked as an NGRoot and an NGBlockBuilder is created. This in turn constructs NGFragment objects for the root and each descendant in the subtree.

Layout would then be called on the NGRoot, invoking the NGLayout logic and producing a NGFragment tree.

NGFragment objects are constructed from Node and ComputedStyle information, and do not directly reference the layout tree.

Note: Ultimately, we hope to be able to identify NG subtree points during style computation and forego generation of legacy layout trees altogether for supported content. This requires us to divorce style resolution from layout tree construction (crbug.com/595137).

Integration with the Legacy Tree

The original plan was to initially push the resulting layout geometry information from the NGFragments into the corresponding legacy LayoutObjects and then in a second stage create an interface bridging the legacy

and LayoutNG trees. The final step would then be to either remove the old legacy layout system entirely (once all layout types have been converted) or to create NGFragments for the remaining legacy layout types.

Stage 1: Populate LayoutObject

Once layout has been completed for a LayoutNG subtree and a NGFragment tree has been created the geometry information from each NGFragment will be pushed over to the LayoutObject. This essentially replaces the layout implementation for the given subtree but does not change the API as the LayoutObject is used both as the input and as the output.

The primary benefit of this is that it allows us to focus on the layout implementation without having to worry about the API exposed to systems such as paint, selection, editing, and accessibility in the early stages of the project. It also simplifies both hosting and reverse hosting.

Stage 2: Common Interface

Geometry information and tree walking for the painters will be extracted into an interface that can be used to walk both layout and LayoutNG trees. This will aid in the transition from legacy layout to layoutNG as it abstracts away the transitions between the two trees and allows the same tree walk logic to be employed for both.

Stage 3: Fragment Tree [optional]

Once the majority of layout implementations have been converted to LayoutNG and one can reasonably expect that the majority of web sites will use LayoutNG for most, if not all, of the content the next step is essentially the reverse of Stage 1: Creating NGFragments from legacy LayoutObjects.

This involves extracting the geometry information from the LayoutObject and constructing corresponding NGFragments. While LayoutObjects and NGFragments can be mapped fairly well for block layout that is not the case for inline layout. Thus this will likely not happen until after the conversion of all of inline layout to LayoutNG has been completed.

Problems with LayoutObject population

During the implementation of inline layout in LayoutNG we realized that populating the legacy layout tree with the resulting layout geometry was going to be significantly harder than originally anticipated.

Due to the differences in how inline elements and text are represented in LayoutNG the mapping between LayoutObject and NGFragment is precarious at best. Exposing the resulting layout geometry though LayoutObjects would thus either require creating an entirely new set of LayoutObjects and carefully injecting those into the legacy tree or keeping a large subset of the old line generation code.

Both options would be quite fragile and have both runtime and memory penalties.

Producing Fragments

For inline layout in particular it turned out to be a lot easier to get the paint code to understand fragments than to inject layout results into LayoutObjects. This entails modifying paint, selection, and editing to operate on fragment trees rather than layout trees. For inline layout this isn’t too big of a change.

Given that we had to modify the interface for inline layout an investigation into doing the same for block layout and it was determined that the expected amount of work needed to produce fragments for the legacy tree was only slightly higher than the work needed to populate LayoutObjects with LayoutNG geometry

(stage 1 above).

It was therefore decided to skip stage 1 and 2 and to proceed directly to stage 3, as outlined above.

Please see the LayoutNG Fragments for Legacy Layout [10] design document for further details.

Reverse Hosting

Nodes which are candidates for reverse hosting (legacy layout inside LayoutNG) are handled by a NGLegacyBuilder which calls the relevant layout() method on the legacy LayoutObject and then extracts sizing information so that an appropriately sized NGUnplacedFragment can be constructed in the new tree.

This allows nodes to be reverse hosted but still be represented in the new tree so that siblings are placed correctly.

The reverse hosting technique works for inline nodes too - in this case, the inline node's nearest block ancestor's layoutInlineChildren() must be invoked (currently only implemented for direct parent). There are some caveats here:

- Every inline participant that flows into the same block must be reverse hosted.
- Each line box needs to create an NGUnplacedFragment in the new tree.
- NGUnplacedFragments in the new tree need to be attached at the correct point.

These caveats are likely acceptable during the transition period as the plan is to convert most, if not all of, flow layout to the new system in phase 1 thus avoiding reverse-hosting at inline boundaries. This will allow reverse hosting with "simple" NGConstraintSpaces, avoiding the complexity associated with "complex" NGConstrainSpaces at boundaries between the two systems.

Note: Children of Fragmented elements can't be candidates for reverse hosting, as this would require the child's layout() to be aware of the fragmentation up the tree.

An implication of this is that (e.g.) multicols with legacy content have to be reverse hosted at the multicol level, rather than at the legacy content level. Hence, we will eventually need to convert all legacy content to the new world.

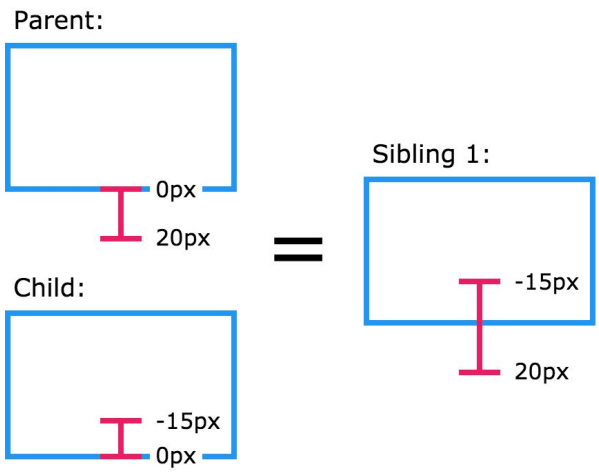
Walking up the Tree

ClientRect calculations and IntersectionObserver need to perform a tree-walk up to the root in order to apply transform/layout information. Transform information is not on the layout tree. Adding layout-parent pointers in the Layout Tree would mean that we lose the immutability property; so instead the parent-layout pointers will be stored on the DOM tree. These operations will begin at the DOM node, ask its NGUnplacedFragment for layout information, then walk up the DOM tree to its parent-layout and so forth.

Margin Collapsing

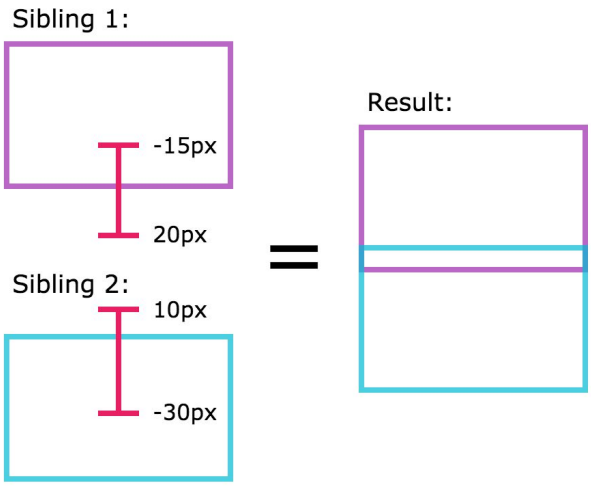
Each NGUnplacedFragment will have two tuples (or "struts") which will be used for margin collapsing. This tuple will contain the maximum and minimum margin for that fragment subtree, as an example:

```
<div style="margin-bottom: 20px">
    <div style="margin-bottom: -15px"></div>
</div>
```

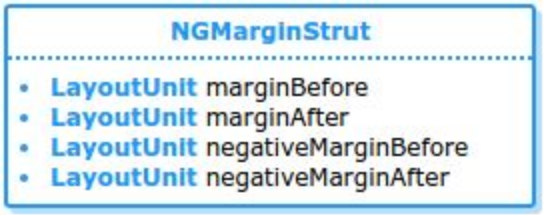


In the above example the resulting fragment has the margin-block-end tuple of `[-15, 20]`. This tuple can be used to collapse margins together of adjacent siblings.

```
<div style="margin-bottom: 20px">
    <div style="margin-bottom: -15px"></div>
</div>
<div style="margin-top: 10px">
    <div style="margin-top: -30px"></div>
</div>
```



Above the margins are collapsed using the two tuples, resulting in an overlap of `10px (=max(20px, 10px) - max(abs(-15px), abs(-30px)))`.



These tuples are represented using the NGMarginStrut class (pictured above) on the NGFragment for the block-start and block-end margins in the parent's coordinate space.

Given that margin collapsing only happens in the block direction there is no need to track margins in the inline direction. For the inline direction margin information from the ComputedStyle object will suffice.

Inline Layout

For inline layout a new pass is introduced before layout tree construction where, for each inline layout root, an NGInlineTextList is constructed. The list contains all inline and out-of-flow items up until the next block level element.

In this list each inline is represented by a node that has a pointer to the style information as well as the text content.

Once this list has been constructed BiDi resolution and text shaping is performed on the entire list. This allows for shaping across element boundaries [4]. The shaping information is stored on the NGInlineTextList object itself.

During layout tree construction for inlines the NGInlineBuilder operates on the NGInlineTextList structure and performs line breaking based on the NGConstraintSpace, producing NGFragment and NGText objects.

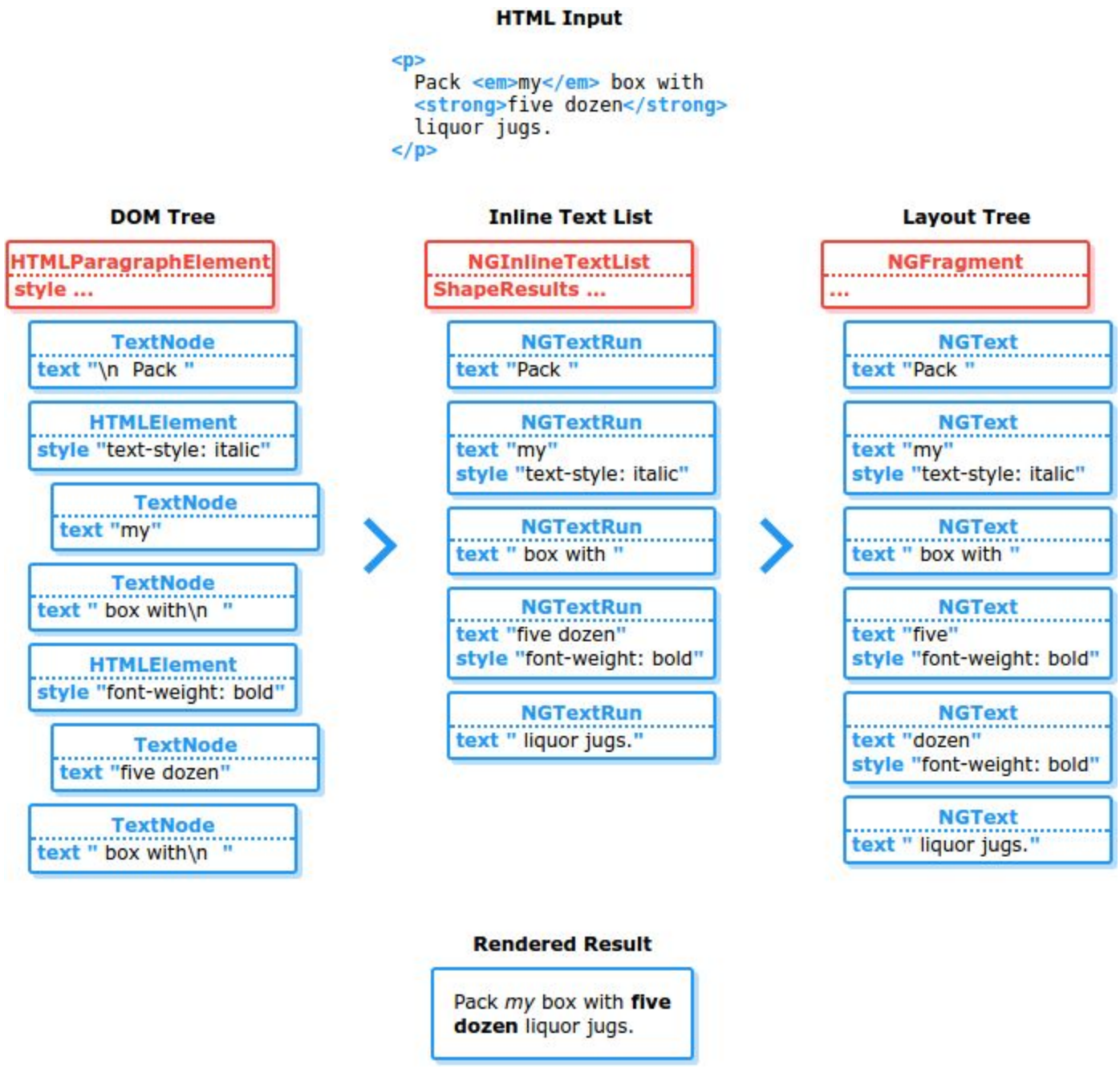


Figure showing the DOM Tree, Inline Text List, and Layout Tree for a given HTML fragment as well as the rendered output. Note that the element hierarchy is flattened and white-space collapsed when going from the DOM to the Inline Text Tree. Also note that the "five dozen" string was split into two NGText fragments, each with an NGBreakToken, as a result of line breaking during Layout Tree construction.

NGText and NGPlacedText are special types of NGUnplacedFragment and NGFragment respectively that are guaranteed to be leaf nodes, as such they has no children collection. Instead they have a reference to a text node in the inline text tree and and offset/size pair. A NGPlacedText object still has style information associated with it, allowing the same mechanism to be used to handle styling, line breaks, and directionality changes.

The inline text list replaces the concept of TextRuns in Blinks legacy layout tree.

White-space collapsing

The current (legacy) blink layout implementation performs white-space collapsing at measure, line break, and line-box construction time. This results in added complexity and extra work as each call site needs to be aware of the white-space collapsing rules and perform the work ad-hoc.

In LayoutNG white-space collapsing will occur during the construction of the inline text tree. In addition to simplifying preferred width calculations, line breaking logic and NGText construction this would also reduce the number of times white-space collapsing would need to occur. In the current system white-space collapsing is performed at least twice for each text node per layout pass. In layout NG it would happen once and be reused across layouts.

First-Letter and First-Line

The First-letter and first-line pseudo classes are supported by creating a separate NGText object for the first letter or line respectively and giving it the relevant style.

Line Breaking

Line breaking will be implemented by subtracting from the constraint space while iterating over the inline text tree. The largest (widest) NGUnplacedText object that can fit within the constraint space is constructed and then the space for that is subtracted from the remaining available width. If the remaining available width is insufficient to place the smallest following atomic text or block the constraint space is modified to push down to the next line.

As NGText is a type of NGFragment (as outlined above) this essentially means that the block fragmentation logic is reused for line breaking.

See the Shaper Driven Line Breaking [4] design document for details.

Baseline

LayoutNG will be the basis to implement multiple types of baselines as the CSS dominant-baseline property defines.
See the Baseline in LayoutNG design document for details.

Performance

During the implementation a performance problem was identified for cases where new inline content is repeatedly appended to a large container. See the LayoutNG Inline Performance design document for details.

Selection

In this model, given that the LayoutNG tree is immutable selection can no longer be represented in the layout tree. Instead a separate selection controller will be introduced that maintains weak pointers into the layout tree.

For content separated into multiple NGFragments the break tokens are used to support selection across the

boundaries.

Note: For inlines this information could theoretically be maintained in the inline text tree but that would complicate things further as it would require separate logic for block and inline content. Furthermore it would require positions to be recomputed to account for white-space collapsing.

Scroll Information

Similarly to how selection state cannot be maintained in the immutable LayoutNG tree neither can the paint layer tree, which in turn contains scroll offsets. Instead the paint layer tree will be attached to the DOM tree.

Project Information

Implementation Strategy

1. Simple block layout without exclusions.
2. Support for floats & block of blocks.
3. Simple inline layout (partial block flow).
4. Full block flow layout.
5. Reverse hosting, giving children simple NGConstraintSpaces without exclusions.
6. Incremental transition of layout types to LayoutNG.
7. Remove legacy layout tree and hosting capabilities.

Milestones

A. Simple block layout with float and block of blocks support (step 1-2).
B. Simple inline layout with partial block flow support (1-3).
C. Full inline and block flow support (1-4).
D. Full hosting and reverse hosting support (1-5). At this stage full CSS custom layout support will be supported and the majority of content will be supported on LayoutNG.
E. Editing transition to LayoutNG.
F. Flexbox, Table, and Grid transition to LayoutNG.
G. MultiCol/Fragmentation transition to LayoutNG.

Internationalization and Localization

Wide support for scripts and typing systems will be maintained and are unaffected by this change.

The way bidirectional text and writing modes support will be implemented is detailed in the detailed design section of this design. Maintaining or improving the support for international scripts and writing modes are important design considerations for LayoutNG and as such should be considered part of the core design.

Specifically we expect LayoutNG to reduce the complexity associated with supporting vertical writing modes and to add support for shaping across element boundaries [5].

Accessibility

There is already a fairly well defined API for populating the Accessibility tree. This will be adopted to work

both with the legacy layout tree and the layout NG tree similarly to how the paint tree will be populated.

Security Considerations

The mutability of the legacy layout tree and the lack of a proper API between it and other subsystems, such as editing and paint, has been a constant source of security bugs over the years. By moving to an entirely immutable tree and by enforcing all interactions with the layout tree to go through an explicit API both of these classes of bugs are eliminated.

Furthermore, as outlined under the Memory Management heading in the Detailed Design section, by garbage collection and weak references use-after-free bugs are avoided.

Performance

As outlined in the Testing Plan section below, existing performance tests will be used to progressively verify the performance of the LayoutNG system as we progressively move over. This is for milestones (A-C) in which we slowly support more sub-trees that can be fully supported by LayoutNG.

Before shipping milestone D (full block layout support with reverse hosting; in which the old block flow layout code will be removed), we'll perform a UMA trial to verify that there aren't any performance regressions.

Once we have migrated all of the legacy code over to LayoutNG there are a lot of performance optimizations that we can explore at that stage. This includes but is not limited to:

- The ability to cache the last N passes of a layout subtree. This will be useful for multi-pass layouts including table, flexbox, scrollers which need to do multiple passes.
- Memory usage tuning. Evicting old LayoutNG trees from memory will be possible with the cache mentioned above. If blink receives a memory pressure warning a memory coordinator will be able to purge the layout tree cache separably.
- Pipelining layout passes speculatively. This is useful for layout animations where we could compute the next N frames of the animation ahead of time.
- Speculative layout while main thread is idle. It will be possible to perform a partial layout pass will the main thread is idle ahead of the frame budget boundary.
- Ability to pass the entire layout tree to a paint thread. This may be useful to perform the paint walk in parallel to another layout or background script.

Testing Plan

The existing layout and performance tests will be used to verify the correctness and performance respectively of LayoutNG. In addition each NGBuilder and all major logic will be unit tested.

Definitions

BiDi	Bi-directional text. Text containing both left-to-right and right-to-left segments.
Block size	Size in the block direction; height in a top-to-bottom writing mode.
ClientRect	Size and position of an Element relative to the viewport.

ComputedStyle	List of applied and resolved CSS values as they apply to an element.
CSS	Cascading Style Sheets. Language for describing presentation of a document.
DOM	Document Object Model. Tree like structure representing an HTML document.
Element	Representation of an HTML element in the DOM.
First-letter	CSS selector allowing the first letter in a paragraph to be styled separately from the rest of the line.
First-line	CSS selector allowing the first line in a paragraph to be styled separately from the rest of the paragraph.
Fragment	The portion of a box that belongs to exactly one fragmentainer. A box in continuous flow always consists of only one fragment. A box in a fragmented flow consists of one or more fragments. Each fragment has its own share of the box's border, padding, and margin, and therefore has its own padding area, border area, and margin area. [6]
Fragmentainer	A box—such as a page box, column box, or region—that contains a portion (or all) of a fragmented flow. Fragmentainers can be pre-defined, or generated as needed. When breakable content would overflow a fragmentainer in the block dimension, it breaks into the next container in its fragmentation context instead. [7]
Inline size	Size in the inline direction; width in a top-to-left writing mode.
IntersectionObserver	An API that can be used to understand movement of DOM elements relative to another element or the browser top level viewport. [9]
LayoutObject	Node in the legacy blink layout tree.
LayoutUnit	Fixed-point unit representing CSS pixels. Has a precision of 1/64.
NGBreakToken	Object indicating that a Node was split into multiple fragments. Details how and where the break occurred.
NGBuilder	Stack allocated object responsible for building a NGFragments for a specific type of layout.
NGConstrantSpaces	A 2D representation of the space that a NGBuilder can perform a layout within. The resulting NGFragment should adhere to all the constraints imposed by the NGConstraintSpace.
NGDirection	Text direction for inline content, one of: horizontal-tb, horizontal-bt, Vertical-lr, vertical-rl.
NGExclusion	An exclusion area inside the NGConstraintSpace. This is used as "keep-out" information for certain types of NGFragments. Floats, CSS Shapes, CSS Exclusions are all represented as NGExclusions.
NGExclusionRect	A rectangular NGExclusion.

NGExclusionShape	A NGExclusion of any shape. Will be used for implementing CSS Shapes.
NGFragment	Representation of a CSS Fragment in LayoutNG. A portion of a box in the layout tree.
NGInlineTextList	List of NGTextRun nodes associated with a block element.
NGLogicalLocation	A direction-neutral location with two fields: inlineOffset and blockOffset.
NGLogicalSize	A direction-neutral size with two fields: inlineSize and blockSize.
NGOrigin	Start point for inline content, one of: TopLeft, TopRight, BottomLeft, BottomRight.
NGText	A type of NGFragment with text content. Always a leaf node.
NGTextRun	A node in the interim Inline Text Tree representing a styled TextNode.
NGUnplacedFragment	An unplaced NGFragment. I.e. a fragment without a location.
NGUnplacedText	An unplaced NGText fragment. I.e. a text fragment without a location.
Node	A node in the DOM.
Oilpan	Garbage collector, see Source/platform/heap/BlinkGCAPIReference.md [8].
PhysicalLocation	A location in screen coordinates as an offset between the top-left corner of a container and the top-left corner of an object.
TextNode	A node in the DOM representing text content.
White-space collapsing	The process of trimming leading and trailing white-space and collapsing sequences of multiple white-space characters into a single one. Space, Tab, and New Line are considered white-space characters.

Document History

Initial revision	Emil A Eklund & Ian Kilpatrick	June 28, 2016
Clarified percentage resolution for the block direction based on review feedback from layout team.	Ian Kilpatrick	June 30, 2016
Expanded on inline layout and tree construction based on review feedback from layout team.	Emil A Eklund	July 12, 2016
Further clarified inline layout section and added illustration. Started a *definitions* section.	Emil A Eklund	July 13, 2016
Clarified derived constraint spaces and scrolling & fragmentation behaviour.	Ian Kilpatrick	July 15, 2016
Completed *Accessibility*, *Security*, *Performance*, *Work Estimates*,	Emil A Eklund & Ian Kilpatrick	July 19, 2016

and *Definitions* sections.		
Updated and expanded *Integration with the Legacy Tree* section to reflect a change in plan. Also updated *Memory Management* section given recent changes.	Emil A Eklund	February 8, 2017
Added *Logical & Physical Units* section and updated *Writing Modes & Coordinate Systems section*.	Emil A Eklund	February 27, 2017

References

1. https://drafts.css-houdini.org/css-layout-api/
2. https://drafts.csswg.org/css-shapes/
3. https://drafts.csswg.org/css-break-3/
4. https://docs.google.com/document/d/1eMTBKTnWEMDu00uS2p8Xj-l9Pk7Kf0q5y3FbcCrWYjU
5. https://crbug.com/6122
6. https://www.w3.org/TR/css-break-3/#fragment
7. https://www.w3.org/TR/css-break-3/#fragmentation-container
8. https://chromium.googlesource.com/chromium/src/+/master/third_party/WebKit/Source/platform/heap/BlinkGCAPIReference.md
9. https://github.com/WICG/IntersectionObserver/blob/gh-pages/explainer.md
10. https://docs.google.com/document/d/11BLMp0ZQsQcAvRCCm3STLCRzBBlsSvGmj1NakXX0FMk/
11. https://trac.webkit.org/wiki/LayoutUnit
12. https://docs.google.com/document/d/1RjH_Ofa8O_ucGvaDCEgsBVECPqUTiQKR3zNyVTr-L_I