Lay out OOFs in one sweep across all nested multicol containers.

Morten Stenshorne <mstensho@chromium.org>, November 2020

This document attempts to describe a way of laying out all out-of-flow positioned descendants inside nested multicol containers as part of finishing layout of the outermost multicol container.

Since OOFs may break and flow into columns that belong to subsequent column rows, which may even be in subsequent multicol fragments, we're going to need to know the size of all the columns before we can begin layout. This means that we need to lay out all multicol containers before we can lay out the OOFs.

When laying out the OOFs, we'll therefore need to mutate the column fragment of inner multicol containers that get OOF fragments.

During regular layout, we bubble OOFs that are contained within a multicol container (i.e. when the OOF is a descendant of a relatively positioned box, which is a descendant of the multicol container) up to their nearest multicol container. We will need to keep track of all multicol containers that are nested and contain such OOFs.

Example 1: When we're finishing #outer, we need to lay out #oof, by walking the list of OOFs inside #inner.

```
<div id="outer" style="columns:2;">
  <div id="inner" style="columns:2;">
    <div style="position:relative; height:400px;">
      <div id="oof" style="position:absolute; height:400px;"></div>
    </div>
  </div>
</div>
```

Example 2: When we're finishing #outer, we need to lay out #oof, by walking the list of OOFs inside #inner. Nothing to do in #middle, though.

```
<div id="outer" style="columns:2;">
  <div id="middle" style="columns:2;">
    <div id="inner" style="columns:2;">
      <div style="position:relative; height:800px;">
        <div id="oof" style="position:absolute; height:800px;"></div>
      </div>
    </div>
  </div>
</div>
```

We'll create a list of multicol containers that have at least one fragment containing such pending OOFs. When finishing layout of the outer fragmentation context, we'll go through each of them, and walk through all its fragments, and lay out OOFs where they belong.

Example 3: #outer creates 2 columns. #inner creates two multicol fragments (one for each outer column) with 2 columns each - 4 inner columns in total. We need to make sure that #oof1 ends up in the first column, and that #oof2 starts in the third column (even if its containing block starts in the second one), and breaks into the fourth column.

```
<div id="outer" style="columns:2; column-fill:auto; height:100px;">
  <div style="height:20px;"></div>
  <div id="inner" style="columns:2; column-fill:auto;">
    <div style="position:relative; height:150px;">
      <div id="oof1" style="position:absolute; height:50px;"></div>
    </div>
    <div style="position:relative;">
      <div id="oof2" style="position:absolute; top:50px; height:150px;"></div>
    </div>
  </div>
</div>
```

Example 4: The OOF doesn't start in the same column nor multicol fragment as its containing block.

```
<div id="outer" style="columns:2; column-fill:auto; height:100px;">
  <div id="inner" style="columns:2; column-fill:auto;">
    <div style="position:relative; height:400px;">
      <div id="oof” style="position:absolute; top:250px; height:100px;"></div>
    </div>
  </div>
</div>
```

The nesting hierarchy isn't important when processing nested multicol containers. All we care about is the columns created by a multicol container, while whether the multicol container itself fragments into outer fragmentainers doesn't matter.

Example 5: #oof1 is fragmented by #inner, and #oof2 is fragmented by #middle. This can be done independently.

```
<div id="outer" style="columns:2; column-fill:auto; height:100px;">
  <div id="middle" style="columns:2; column-fill:auto;">
    <div style="position:relative; height:400px;">
      <div id="inner" style="columns:2; column-fill:auto;">
        <div style="position:relative; height:400px;">
          <div id="oof1" style="position:absolute; bottom:50px;
height:100px;"></div>
        </div>
      </div>
      <div id="oof2" style="position:absolute; top:150px; height:200px;"></div>
    </div>
  </div>
</div>
```

Doubly nested fragmentation contexts isn't a very common use case, so laying out this correctly isn't a top priority - but we DO need to lay out OOFs in there somehow. The approach in this document should get it right, with minimal additional efforts. We need to handle multiple inner multicol container "siblings" anyway, e.g. when printing.

Example 6: The #outer multicol container has two inner multicol containers, one (#inner1) wrapped inside a child DIV, and the other one (#inner2) as a direct child. #inner1 will start in the first outermost column and create two fragments (column rows). The height of the first one (in the first outermost column) is 90px, and the height of the second one (in the second outermost column) is the remainder - 25px. #oof1 will start in the second inner column, and end in the fourth. #inner2 will start in the second outermost column and create three fragments (column rows). The height of the first one (in the second outermost column) is 75px (100px minus the space taken up there by the #inner1). The height of the second and third fragments will be 100px. #oof2 will start near the bottom of the second inner column (in the second outermost column), and end in the fourth inner column (in the third outermost column).

```
<div id="outer" style="columns:3; column-fill:auto; height:100px;">
  <div style="padding-top:10px;">
    <div id="inner1" style="columns:2; column-fill:auto; height:115px;">
      <div style="position:relative; height:230px;">
        <div id="oof1" style="position:absolute; top:120px; height:100px;"></div>
      </div>
    </div>
  </div>
  <div id="inner2" style="columns:2; column-fill:auto; height:175px;">
    <div style="position:relative; height:350px;">
      <div id="oof2" style="position:absolute; top:140px; height:150px;"></div>
    </div>
  </div>
</div>
```

In our current implementation, if an OOF is taller than the in-flow contents of the multicol container, we create proxy fragmentatiners / columns.

Example 7: #relpos needs two columns, and that's all the in-flow content we have. #oof starts in the second column, and needs a third one. The third column will be added when handling the OOFs when finishing layout of #mc. This already works fine.

```
<div id="mc" style="columns:4; column-fill:auto; height:100px;">
  <div id=”relpos” style="position:relative; height:150px;">
    <div id="oof" style="position:absolute; top:120px; height:100px;"></div>
  </div>
</div>
```

Ideally, we should create such proxy fragmentainers also when nested.

Example 8: #inner has 150px of in-flow content. This is short enough so that it will create only one multicol fragment (in the first outer column). However, #oof starts in the second inner column and needs a third one.

```
<div id="outer" style="columns:2; column-fill:auto; height:100px;">
```

```
  <div id="inner" style="columns:2; column-fill:auto;">
    <div id=”relpos” style="position:relative; height:150px;">
      <div id="oof" style="position:absolute; top:120px; height:100px;"></div>
    </div>
  </div>
</div>
```

Creating proxy fragments in an inner multicol container would require us to regenerate the inner multicol fragment, which in turn would require us to mutate its parent, to replace the old multicol child fragment with the new one. This should be possible, and would be necessary in order to be able to paint rules between the additional fragments (at least with the current solution). However, it would complicate things by quite a bit, so the proposed solution here will just add the extra fragments generated for an OOF to the last column in the last multicol fragment, without generating additional columns. We can always add this later, if we feel it’s necessary. We’d probably need to keep track of the last fragment generated for each affected multicol container, along with the parent fragment (to be able to replace the NGLink for the regenerated multicol child fragment).

Proposed design details

We already bubble OOFs contained within a multicol container to the nearest containing multicol container. We’ll keep this, but if the multicol container is nested inside another fragmentation context, we won’t lay them out as part of finishing the nearest multicol container. Instead we store the OOFs for later (in the NGPhysicalBoxFragment(s) generated by the multicol container, and add the multicol container to a list of inner multicol containers that we’ll need to revisit when finishing the outermost fragmentation context.

When we’re about to finish layout of the outermost fragmentation context, we’ll go through this list, and process all OOFs for each individual multicol container, for each column, now that we know the block-size and amount of columns. Each column fragment that gets at least one OOF needs to be regenerated. If any OOF requires more columns than we have, we’ll just add the additional fragments to the last column in the last multicol fragment, with an additional inline-offset, to fake the column stride for columns that don’t exist. As mentioned earlier, creating actual proxy columns is an option, but it complicates the solution. We would need to regenerate the multicol fragment itself, and we’d probably also need to store the multicol containers in a tree structure when discovering them during layout, and process the innermost ones first. Since it’s not certain that anyone will actually care, I suggest that we don’t do it initially, at least.

Part 1: Discovering inner multicol containers with OOFs

Store the multicol containers that we discover in a hash set.

Code snippets that probably doesn’t compile / pseudo code follows.

```
typedef HashSet<NGBlockNode> NGInnerMulticolsWithPendingOOFs;

class NGPhysicalBoxFragment ... {
  // ...
 public:
  // ...
  const NGInnerMulticolsWithPendingOOFs& MulticolsWithPendingOOFs() const;
  // ...
 private:
  // TODO: Store things in RareData?
};

// Building the data structure.

class NGBoxFragmentBuilder {
  // ...
 public:
  NGInnerMulticolsWithPendingOOFs& MulticolsWithPendingOOFs() {
    return multicols_with_pending_oofs_;
  }
  // ...
 private:
  NGInnerMulticolsWithPendingOOFs multicols_with_pending_oofs_;
  // ...
};

NGColumnLayoutAlgorithm::Layout() {
  // ... (after having added all the child fragments)
  if (ConstraintSpace().HasBlockFragmentation())
    if (has OOFs bubbled past containing block inside the multicol) {
      container_builder_.MulticolsWithPendingOOFs().insert(Node());
      // TODO: Store the OOFs somewhere in the resulting fragment.
    }
  }
  // ...
}

void NGBoxFragmentBuilder::AddChild(const NGPhysicalContainerFragment& child, ...) {
  // ...
  if (child.IsCSSBox()) {
    // ...
    const auto& box_child = To<NGPhysicalBoxFragment>(child);
    if (has_block_fragmentation_) {
      for (auto it : box_child.MulticolsWithPendingOOFs()) {
        // Note that the same multicol container may be added to multiple
        // fragments.
        multicols_with_pending_oofs_.insert(*it);
      }
    }
    // ...
  }
  // ...
}
```

Part 2: Laying out the OOFs

Go through all the multicol containers discovered. For each multicol fragment, add the pending OOF nodes to a list. Go through each column in every multicol fragment and start laying out the

OOFs in the right column. Resume any OOF that breaks in the next column(s). If there are still OOFs to lay out or resume when we're past the last column, add them to the last column, with an additional inline offset to mimic the column stride.

For each column:
We could create a fragment builder and populate it with the old column fragment, and add any OOFs that we lay out. Then generate a new fragment, and replace its child entry in the multicol fragment (the parent). Then move to the next column.
Or, rather than creating a fragment builder, do something similar to NGPhysicalBoxFragment::CloneWithPostLayoutFragments(). Creating a builder is probably more trouble.

Maybe something like this - rough code that must be taken with a pinch of salt (such as the naming, maybe), and lots of TODOs:

```
NGColumnLayoutAlgorithm::Layout() {
  // ... (after having added all the child fragments)
  if (!ConstraintSpace().HasBlockFragmentation()) {
    // We're at the outermost fragmentation context.
    for (NGBlockNode multicol : container_builder_.MulticolsWithPendingOOFs())
      LayoutOOFsInMulticol(multicol);
    container_builder_.MulticolsWithPendingOOFs().clear();
  }
  // ...
}

void LayoutOOFsInMulticol(NGBlockNode multicol_node) {
  struct NodeToLayout {
    NGPhysicalOutOfFlowPositionedNode node;
    scoped_refptr<const NGBlockBreakToken> break_token;
  };

  Vector<NodeToLayout> pending_nodes_to_layout;
  base::Optional<CurrentColumn> current_column;

  LayoutUnit consumed_block_size;
  for (auto multicol_fragment :
         multicol_node.GetLayoutBox()->PhysicalFragments()) {
    // TODO: Add all pending OOFs in multicol_fragment to
    // pending_nodes_to_layout.

    for (auto child : multicol_fragment.Children()) {
      if (!child->IsColumnBox())  // Ignore spanners
        continue;

      // We found a new column to put stuff into. Finish the previous one, if
      // any. We'll do this lazily when finding a next column, so that, if there
      // are no columns left, but we still have OOFs, we'll just continue
      // putting them in the last column.
      if (current_column)
        RegenerateColumnFragmentIfNeeded(&(*current_column));

      NGConstraintSpace column_constraint_space = ...;

      // When we're done with this column, we need to regenerate its fragment
```

```
    // and replace its child entry in the parent (the multicol fragment).
    current_column.emplace(&child);

    const auto& column = To<NGPhysicalBoxFragment>(*child);
    Vector<NodeToLayout> current_nodes_to_layout;

    // TODO: Check consumed_block_size (and more?), and move the OOFs that
    // belong in this column over from pending_nodes_to_layout to
    // current_nodes_to_layout. Using a vector for this isn't very elegant /
    // performant, but maybe not a problem.

    // We'll build a list of nodes that need to be resumed in the next
    // multicol container because they broke.
    Vector<NodeToLayout> next_nodes_to_layout;

    for (auto node_to_layout : current_nodes_to_layout) {
      // TODO: Set up a constraint space with a fragmentainer block-size equal
      // to the block-size of |column|.
      NGConstraintSpace space;

      scoped_refptr<const NGPhysicalBoxFragment> oof_fragment = Layout(
          space, node_to_layout.node, node_to_layout.break_token);

      if (oof_fragment.BreakToken()) {
        // The OOF broke. To be continued in the next column (which may be a
        // sibling column of the current one, or a column in the next row,
        // either after a spanner in this multicol fragment, or a row in a
        // later multicol fragment).
        next_node_to_layout.emplace_back(
            node_to_layout.node
            To<NGPhysicalBoxFragment>(oof_fragment.BreakToken()));
      }

      current_column->new_children.push_back(std::move(oof_fragment));
    }
    current_nodes_to_layout = next_nodes_to_layout;

    if (const auto* break_token =
        To<NGBlockBreakToken>(column.BreakToken().get()))
      consumed_block_size = break_token->ConsumedBlockSize();
  }
}

if (!current_nodes_to_layout.IsEmpty() ||
    !pending_nodes_to_layout.IsEmpty()) {
  // We still have more to lay out, but we don't have any more columns
  // created. Keep on laying out what's left into the last column in the last
  // multicol container fragment, but for each OOF fragment, increase the
  // inline offset by the column stride (column inline size + column-gap).
  DCHECK(current_column);
  LayoutUnit column_stride = last_column.InlineSize() + column_gap;
  LayoutUnit additional_inline_offset = column_stride;
  while (!current_nodes_to_layout.IsEmpty() ||
         !pending_nodes_to_layout.IsEmpty()) {
    // TODO: Move any nodes left in pending_nodes_to_layout that should start
    // layout now, and add them to current_nodes_to_layout.
    Vector<NodeToLayout> next_nodes_to_layout;
    for (auto node_to_layout : current_nodes_to_layout) {
      // TODO: Do something similar as in the previous loop (share the code,
      // perhaps?). Add the new OOF fragments to
      // current_column->new_fragments.
    }
```

```
      current_nodes_to_layout = next_nodes_to_layout;
      // TODO: Increase consumed_block_size.
      additional_inline_offset += column_stride;
    }
  }

  if (current_column)
    RegenerateColumnFragmentIfNeeded(&(*current_column));
}

struct CurrentColumn {
  NGLink& child_entry;
  Vector<scoped_refptr<const NGPhysicalBoxFragment>> new_children;

  explicit CurrentColumn(NGLink* link) : child_entry(*link) {}
  void AddChild(scoped_refptr<const NGPhysicalBoxFragment> oof_fragment) {
    new_children.push_back(std::move(oof_fragment()));
  }
};

void RegenerateColumnFragmentIfNeeded(CurrentColumn* current_column) {
  if (!current_column->new_children.size())
    return;

  // TODO: find the old result in LayoutBox, based on
  // current_column->child_entry.fragment.
  scoped_refptr<const NGLayoutResult> old_result = ...;

  scoped_refptr<const NGLayoutResult> new_result =
      NGLayoutResult::CloneWithAdditionalChildren(old_result,
                                                  current_column->new_children);
  const auto new_column =
      To<const NGPhysicalBoxFragment>(&new_result->PhysicalFragment());
  // TODO: NGLink::fragment refcounting needs to be done manually.
  current_column->child_entry->fragment = new_column;

  // TODO: Also replace the layout result entry in LayoutBox; see
  // NGBlockNode::AddColumnResult() for inspiration.
}

scoped_refptr<const NGLayoutResult>
NGLayoutResult::CloneWithAdditionalChildren() {
  // See NGLayoutResult::CloneWithPostLayoutFragments() for inspiration. If it
  // seems better to use a fragment builder (in CurrentColumn) that we initially
  // populate with the old fragment instead, we can do that, but I think
  // skipping the builder would be easier.
}
```