UI Invalidation

Save CPU usage by marking widgets to selectively recalculate and repaint when changes happen.

Invalidation is a system that reduces the CPU usage of UI by limiting how often it repaints widgets. When widgets within the Invalidation system change their layout within the UI, they are marked as **invalidated**. After that, only invalidated widgets and their children are repainted.

There are two ways to use Invalidation in your projects:

- Implement Invalidation on a per-widget basis using **Invalidation Box** and **Retainer Panel** widgets.
- Implement Invalidation across your entire UI using **Global Invalidation**.

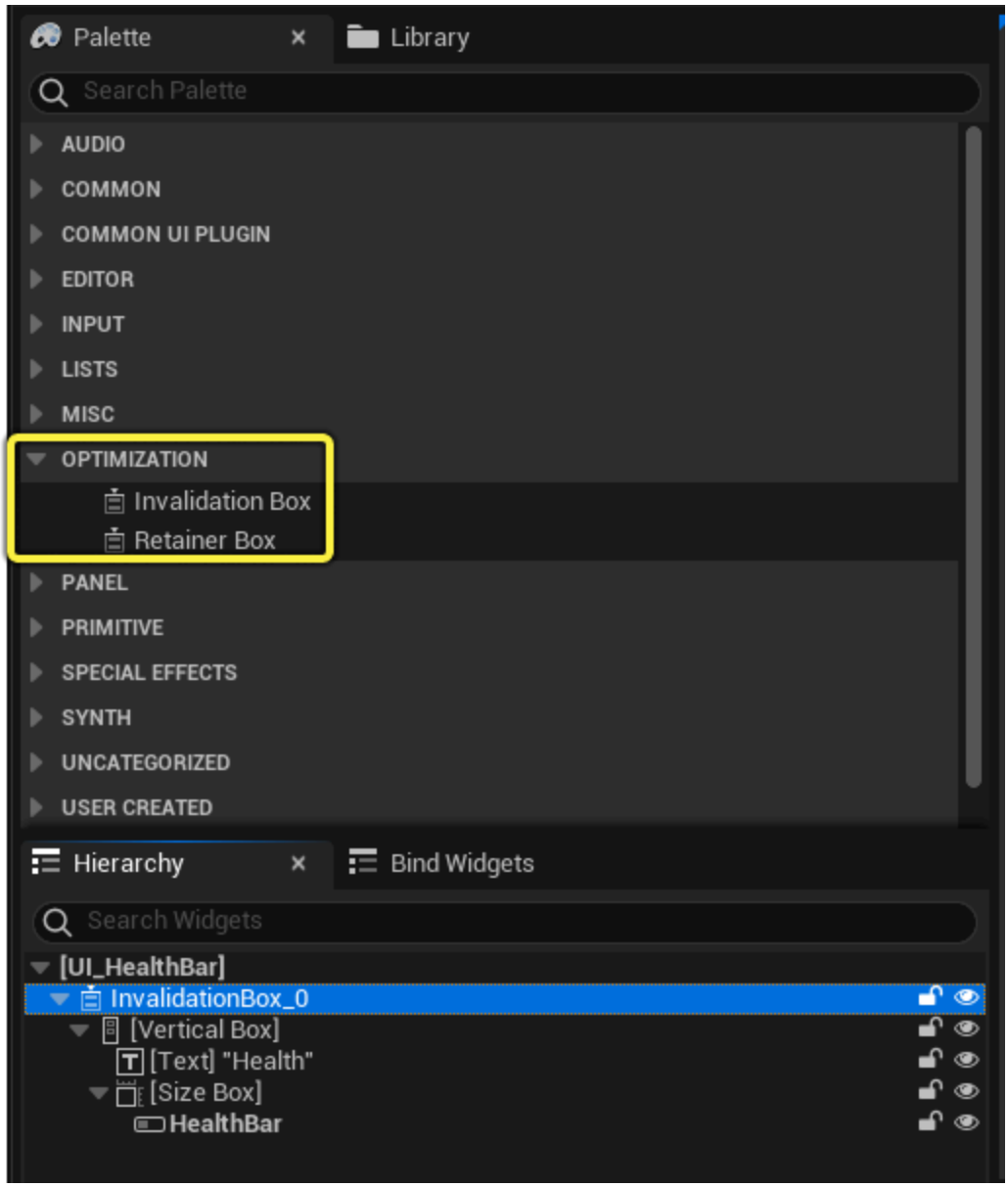
The sections below describe how to implement Invalidation in your project and how it works so that you can make decisions about where to apply it.

How to Implement Invalidation

Invalidation Boxes

Invalidation Box widgets cache their child widgets' geometry, then monitor those widgets for changes. As long as the widgets in the Invalidation Box do not change, Slate falls back on the cached geometry instead of repainting it, which dramatically reduces those widgets' CPU usage.

You can find Invalidation Box widgets in the UMG editor's **Palette**, in the **Optimization** section.



An Invalidation Box handles any widget that it is wrapped around, as well as all of its children in the Hierarchy.

For more information about Invalidation Boxes and their settings, refer to the Invalidation Box widget reference.

Global Invalidation

Global Invalidation enables the Invalidation features in `SWindow`, effectively wrapping your entire UI in an Invalidation Box. Any Invalidation Boxes contained within that window will be deactivated, and only `SWindow` will cache information.

You can enable Global Invalidation by setting `Slate.EnableGlobalInvalidation` to true.

Retainer Panels

Retainer Panels flatten all child widgets into a single texture before rendering them on the user's screen. In addition, you can configure how retainers render with the following options:

- Use phase-based or framerate-limited rendering to configure each retainer to draw on separate frames, making it so that Slate does not draw the entire UI all at the same time.
- Configure different Retainer Panels to have different frame rates. For example, some parts of the UI could run at 30 FPS, while others could run at 60 FPS.

All of these features serve to reduce or manage the number of draw calls your UI makes in a single frame.

Retainer Panels have a high overhead when they repaint, and they use more memory than the individual widgets would with an Invalidation Box. This is because each Retainer Panel has its own render target in addition to using the individual widgets' Invalidation data. When looking for opportunities to reduce your UI's CPU usage, you should try using Invalidation Boxes first. If you still need to reduce draw calls, then implementing Retainers can condense them further. This can be helpful in environments where the performance budget is especially tight, like low-end mobile devices.

How Invalidation Works

When widgets paint on the screen, several computations happen in the following order:

1. **Hierarchy:** Slate builds the tree of widgets within the hierarchy, including root widgets and all their children.
2. **Layout:** Slate calculates widget size and on-screen location based on their render transforms.

3. **Paint:** Slate calculates the geometry of individual widgets.

Each step in this process requires going through all the steps that follow it. For example, a Layout calculation requires running the Paint step, and rebuilding the hierarchy requires running both the Paint and Layout steps.

The Invalidation system caches each of the above types of data in memory, either in the widgets' parent Invalidation Box, or, if you are using Global Invalidation, in the SWindow they're drawn in. As long as the cached information does not change, Slate falls back on it instead of re-doing the calculations. Whenever a widget does change, it is added to a dirty list. On the next frame that renders, all dirty widgets will be re-calculated according to the type of data that changed.

There are several types of Invalidation that correspond to what parts of the recalculation and painting process they skip.

Invalidation Type	CPU Cost	Description
Volatile/Visibility	Very Low	The **Is Volatile** or **Visibility** flags change on the widget and its children.
Paint	Low	The geometry of the widget is recalculated. This occurs if you change non-layout parameters, like color or materials.
Layout	Moderate	As Paint Invalidation, plus the widget's size and location on the screen are changed. This occurs if you change the render transform of the widget.
Child	High	As Layout Invalidation, plus Slate rebuilds the list of child widgets. This entails a full recalculation of the widget and its children, and it occurs if you add or remove children from the widget.

As an example of how this impacts your project, if you only change the color of a widget, it will trigger **Paint Invalidation** on that widget. Slate will skip rebuilding its list of children and its layout, and it will only need to recalculate its geometry. The cached Layout data remains valid, so it does not need to be updated.

If you move or resize a widget, either in code or with a Sequencer animation, it will trigger **Layout Invalidation**. All of the widget's cached information will need to be re-computed each time this occurs, including both the Layout data (location and size) and the Paint data (widget geometry). This is still fairly fast compared with Child Invalidation.

If you add or remove a child from the widget's children, it will trigger **Child Invalidation**. After rebuilding its list of children, the Layout and Paint calculations will need to be updated on the widget and all of its children. In most use cases, this is a single-frame cost that is worthwhile, but if your UI does it each frame, it can be a substantial performance bottleneck.

Volatile/Visibility Invalidation applies only when the Is Volatile or Visibility flags change. Either of these circumstances requires re-painting the widget on the next frame, but does not necessarily mean the other data is invalid. Refer to the Volatile Widgets section below for more information.

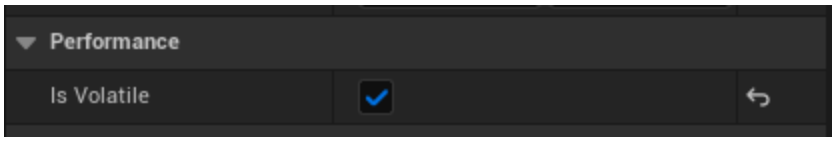
Invalidation is especially well-suited for widgets that do not change often, as Slate can fall back on their cached data for long periods of time, alleviating a large amount of the CPU load for your UI. This is especially important when creating large or complex UIs, such as those in an MMORPG or a live-service game with deep menus.

Volatile Widgets

Sometimes, you may need to implement a widget that needs to update very frequently, possibly on a frame-by-frame basis. In this case, the widget will Invalidate every tick that it changes, which potentially uses the same CPU load as if the widget didn't use Invalidation at all, but consumes the memory needed to cache the widget's geometry anyway.

To address this, you can mark widgets that update often as Volatile widgets. When a widget is marked as Volatile, the Invalidation system does not cache its Paint data or its childrens' Paint data. Its geometry will be recalculated and repainted each frame, but Slate will still skip the layout calculations unless they are needed. This can be helpful for situations where you want to apply Invalidation broadly across your UI, but want to tailor a handful of widgets which wouldn't benefit from caching due to how frequently they update.

To mark a widget as Volatile, toggle the **Performance** > **Is Volatile** setting of the widget in the widget's Details panel.



Alternatively, in C++ you can mark a widget as Volatile by setting the the `Volatile` parameter to `true`.