# User Interface - Feature #4029

## implement more widget rendering knowledge within the client drivers

04/01/2019 10:24 PM - Eric Faulhaber

| | | | |
|---|---|---|---|
| **Status:** | New | **Start date:** | |
| **Priority:** | Normal | **Due date:** | |
| **Assignee:** | Hynek Cihlar | **% Done:** | 0% |
| **Category:** | | **Estimated time:** | 0.00 hour |
| **Target version:** | | | |
| **billable:** | No | **version:** | |
| **vendor_id:** | GCD | | |

**Description**

**Related issues:**

| | |
|---|---|
| Related to User Interface - Bug #2971: rework tooltip support to use OverlayW... | **New** |

### History

#### #1 - 04/01/2019 10:32 PM - Eric Faulhaber

I'm not sure if there is already an issue for this potential performance improvement, but I know we've discussed it many times.

The idea is to cut down on communication of graphics primitive calls between the common UI code and the individual client driver by putting more smarts into the client drivers themselves. For instance, instead of being told to draw each primitive needed to render a button in its "pressed" state as the result of a mouse down event going from the client to the server and the server deciding which primitives need to be drawn, let the client driver take on more of the responsibility for responding to events and rendering widgets itself. This will cut down on network traffic and should eliminate cases where a widget feels "laggy" in response to user-driven events.

The downside of course is that each client driver becomes more complicated, so we'll have to decide which cases make the most sense to refactor this way.

I'll leave it to the UI experts to re-phrase this idea more accurately :-)

#### #2 - 04/08/2019 01:53 AM - Hynek Cihlar

We already push some of the "widget knowledge" to the drivers, like input-sensitive widget areas to the drivers (especially the Web GUI driver). We also employ batching of draw operations to bring the number of round trips to the JS driver. I think we still could find some room for improvement with current design before moving to smarter drivers.

#### #3 - 10/11/2020 10:25 PM - Greg Shah

*- Assignee set to Hynek Cihlar*

I think it may be time to consider a more serious implementation of this idea.

One thing I don't want to lose is the benefit of our theme-based drawing. I see two obvious ways we can improve things:

1. We can implement higher level drawing primitives at the driver level. We did this recently with drawEdge() where we pass more data into a single operation and avoid a much larger set of operations being queued. This is simpler and less risky to implement. I also suspect it has less overall impact than the idea below. Still, it probably makes sense to do this where possible.

2. The bigger impact is to implement more of the dynamic drawing in response to use interaction.  The classic example here is to implement all of the button drawing completely on the browser-side.  The Java code would not be involved in the drawing of a pressed or released button.  Of course, we would still need to send the mouse events to the Java side, but would handle the drawing immediately.  My idea about how to do this: have each widget (the button in this case) "record" the drawing sequences needed to respond to certain events.  Then these sequences would be sent down and registered such that the client could replay them itself.  This way, the original drawing occurs via our normal theme code (including any customer-specific theme subclasses), but the javascript side can still implement the feature.

I know this doesn't help us on the initial screen load, but it will likely make the UI much more responsive for usage.

#### #4 - 10/12/2020 02:44 AM - Hynek Cihlar

Has anybody explored the idea of moving the canvas rendering in a separate web worker? This could free the main thread from waiting on the rendering to finish. I know we batch the drawing operations in single messages, but this would at least split the message processing and drawing.

#### #5 - 10/12/2020 02:57 AM - Sergey Ivanovskiy

Hynek Cihlar wrote:

> Has anybody explored the idea of moving the canvas rendering in a separate web worker? This could free the main thread from waiting on the rendering to finish. I know we batch the drawing operations in single messages, but this would at least split the message processing and drawing.

I thought about moving web socket into web worker. It seems that web workers can improve performance under stress loading of application in the case of several tabs connect to the one server.
https://developer.mozilla.org/en-US/docs/Web/API/ImageData doesn't state that ImageData is available for web workers in Safari and IE.

#### #6 - 10/12/2020 01:36 PM - Hynek Cihlar

*- vendor_id changed from GCD to TIMCO*

Greg Shah wrote:

> I think it may be time to consider a more serious implementation of this idea.
>
> One thing I don't want to lose is the benefit of our theme-based drawing.  I see two obvious ways we can improve things:
>
> 1. We can implement higher level drawing primitives at the driver level.  We did this recently with drawEdge() where we pass more data into a single operation and avoid a much larger set of operations being queued.  This is simpler and less risky to implement.  I also suspect it has less overall impact than the idea below.  Still, it probably makes sense to do this where possible.

The drawing primitives are already batched in single socket message. I don't think there will be much of a performance gain if we denormalize the primitive operations into higher level primitives. This would certainly lead to size decrease of the paint messages, but the processing and UI response won't be affected.

> 2. The bigger impact is to implement more of the dynamic drawing in response to use interaction.  The classic example here is to implement all of the button drawing completely on the browser-side.  The Java code would not be involved in the drawing of a pressed or released button.  Of course, we would still need to send the mouse events to the Java side, but would handle the drawing immediately.  My idea about how to do this: have each widget (the button in this case) "record" the drawing sequences needed to respond to certain events.  Then these sequences would

> be sent down and registered such that the client could replay them itself. This way, the original drawing occurs via our normal theme code (including any customer-specific theme subclasses), but the javascript side can still implement the feature.

This could probably work for the simple widgets, like button, with very limited visual states and limited user input. But the new semantic layer describing the widget states and behavior would explode for more complex widgets.

I see one more way. Use GWT to move the theme drawing and widget input handling to the web. The obvious benefit is that with careful design we could minimize code duplication and reuse the existing implementations. This would certainly require substantial amount of effort, especially in refactoring the existing code to remove dependencies not convertible to GWT, to provide web extension points and to some extent implement the notion of widget tree in JS. But perhaps this could be done iteratively, and convert single widget type at a time. Ideally we would use the same code to draw widgets and handle input in Java and JS with the help of an abstraction layer.

#### #7 - 10/12/2020 02:40 PM - Hynek Cihlar

*- vendor_id changed from TIMCO to GCD*

#### #8 - 10/13/2020 05:53 AM - Greg Shah

> The drawing primitives are already batched in single socket message. I don't think there will be much of a performance gain if we denormalize the primitive operations into higher level primitives. This would certainly lead to size decrease of the paint messages, but the processing and UI response won't be affected.

The drawEdge() did make a difference. I suspect we have some common sequences which we implement over and over which could also benefit. The advantage of this approach is that it is quick to implement and is low risk.

> This could probably work for the simple widgets, like button, with very limited visual states and limited user input. But the new semantic layer describing the widget states and behavior would explode for more complex widgets.

I don't think this is just limited to button. I think all of the following can be "prerecorded" and cached on the client side:

- radio set
- selection list
- combo-box
- button
- slider
- toggle box
- window dragging (we already do something here, but it can't keep up with a fast drag so it still needs some work to be more completely processed on the client)
- scrollbar (not the content pane but the scrollbar itself)

All of these have something in common: they have some reasonably simple user interaction, often very mouse focused and they are usually pretty static in content once they are finished initializing.

I should also point out that every reduction in JS/Java round-trips and data load means that the rest of the system operates faster. So this is not just helping these cases, it takes a load off the system. I expect this to have an outsized effect on low-bandwidth clients. We have a customer right now that has their testing exclusively done over a VPN because their servers are all at an offsite service provider which can only be reached via VPN. This kind of situation may be common and it could really hold back the web client performance.

> I see one more way. Use GWT to move the theme drawing and widget input handling to the web. The obvious benefit is that with careful design we could minimize code duplication and reuse the existing implementations.

I worry that GWT will not give us the control needed to implement the way we need to implement. It will also bring a large number of 3rd party dependencies and complexity that will make the client heavier, more fragile and harder to debug/support. I don't know that we can live with the limitations in this layer of the system. For the admin console there was no real compatibility requirements and there was no hard performance target. I don't think we can commit to GWT for the core 4GL-compatible UI.

I am willing to consider pushing more of the implementation down to the JS side. But I think we must do this in pure javascript, written and tuned exactly to our needs.

#### #9 - 10/13/2020 07:07 AM - Ovidiu Maxiniuc

Maybe you already think about this and ruled it out because of code duplication, but what if we push the drawing primitives further on, directly to JS?

For example, for GuiButton / ButtonGuiImpl there is a single API:

```
public void drawButton(AbstractWidget<GuiOutputManager> widget,
                       @SuppressWarnings("rawtypes") GuiDriver gd,
                       GuiColorResolver gc,
                       GuiFontResolver gf,
                       int x,
                       int y,
                       int width,
                       int height,
                       boolean flat,
                       boolean highlighted,
                       boolean isDefault,
                       boolean mouseOver,
                       boolean pressed,
                       boolean keyPressed,
                       boolean noFocus,
                       boolean hasImages,
                       boolean hasPrepackagedImages,
                       ImageGuiImpl imgDown,
                       ImageGuiImpl imgUp,
                       ImageGuiImpl imgDisabled,
                       MnemonicInfo mnemonic,
                       String textLabel,
                       boolean useResolverColors);
```

Of course, there are a lot of parameters here, but they rarely change from the initial "submit". We could create a map of the parameters on JS side (the key being the widget Id) and only send the changes in a delta frame. The initial data might be large but I think it will not be bigger than the sum of all messages for drawing primitives for the first time the button is shown. Next, for example if the button is hovered (mouseOver parameter changes), only this attribute is sent and the request to redraw the button. The rest of data is already on JS. This way two short messages will replace at least 10 drawing primitives (for Windows10 theme, where the button is really plain).

Another advantage is that we can draw graphics effect which were difficult or costly to "serialize" as graphic primitives (ex: drawing a gradient) and the drawing can use the most suitable drawing primitive to JS instead of using the common-denominator with other GUI drivers.

Downsides and possible no-go:

- additional work for implementing the boilerplate; changes do not fit in the paradigm we use now;
- all themes need practically implemented in JS;
- duplicated maintenance: we will probably keep the swing client so changes in theme or implementing new themes need to be written in both places;
- extra memory for the JS client to store all properties and maybe on FWD client (to be able to compute the delta frames). The CPU might also need some cycles but I expect it won't be visible;
- I do not have a solution for passing the first parameter, if really needed. But if we can send all those parameters the direct access to widget might not be needed.

#### #10 - 10/13/2020 08:40 AM - Greg Shah

> Maybe you already think about this and ruled it out because of code duplication, but what if we push the drawing primitives further on, directly to JS?

It hasn't been ruled out, but the problems of code duplication do give us some reason to hesitate. We have customers with their own widget subclasses and who plan to implement their own themes. Implementing a hybrid drawing approach will definitely cause much more work (initially and over time) for those customers. It will likely also be more fragile.

My first idea above (implement new higer level primitives) is meant as a "middle ground" between our current approach and your idea.

> Another advantage is that we can draw graphics effect which were difficult or costly to "serialize" as graphic primitives (ex: drawing a gradient) and the drawing can use the most suitable drawing primitive to JS instead of using the common-denominator with other GUI drivers.

This suggests that we do need some new primitives and we should define these in a standard/generic manner. For example, why not define a list of supported gradients as constants and then offer those as options in our drawing primitives?

#### #11 - 10/14/2020 06:51 AM - Hynek Cihlar

I gave all the known options more thought and I still think that some kind of transcompiling the existing widget implementation sources in the web (Typescript) environment would be the best solution.

If we want to provide good UI experience by lowering the UI responsiveness, than introducing less abstract drawing primitives (like drawEdge) won't be enough.

The idea of recording the draw primitives for individual widget states won't scale for more complex widgets. I can't imagine how this could work for widgets like TREELIST for example. The more complex widgets are just those that would benefit from an increased responsiveness the most.

I agree the GWT framework has its issues, and its very narrow web UI development workflows could be limiting for us. Yes, for our case, this would be a bit of an overkill. We would only require a transcompiler which would generate Typescript sources with a certain structure and an abstraction layer these sources would depend on. The abstraction layer would provide a graphic context primitives and compatible user input handling. No other runtime dependencies should be needed. There seem to be viable tools that could handle the transcompilation process, like jsweet.org, or we could even write this inhouse.

#### #12 - 10/14/2020 08:26 AM - Greg Shah

The transpiling is a reasonable way to handle the code duplication issue, IF we don't have to edit the result.  Any hand-coded features would need to be helpers or part of the API which would be used by the transpiled code.

- Only certain code would be moved to the client side.  I don't think it makes sense to move the entire widget processing, just the interactive parts (drawing and editing/input handling).  Even so, that is a huge amount.
- The theming code would also need to be included.
- Significant limits would exist on the techniques used in the transpiled code.  We probably would have a significant amount of rework to clean, make safe and separate the necessary parts.  Over time, we would have to be very careful in the changes applied in those places.
- We would have to expose all this to external widgets as well.

At this time, I think we are not ready to go this route.  I do see it as a possible future, especially if we need to greatly improve the scalability of the solution (to large numbers of simultaneous clients) and/or overcome bandwidth limitations of the web client.

#### #13 - 10/14/2020 08:28 AM - Greg Shah

Are there some "quick wins" we can achieve immediately using either the higher level primitives or the recording approach?

Please also note that we already have an infrastructure for recording/caching/playback of primitive sequences today.  So expanding that usage is not a massive effort.

#### #14 - 10/14/2020 09:30 AM - Hynek Cihlar

Greg Shah wrote:

> Are there some "quick wins" we can achieve immediately using either the higher level primitives or the recording approach?

One UX area which would deserve higher level primitives IMHO is text input. Currently every key stroke triggers network IO from JS client to Java client (and eventually to server). Even when JS client runs on the same machine, the lag is noticeable. With real life deployments where JS and Java clients will be separated with full network stacks, the lag will become pronounced.

I was thinking about introducing primitives to handle user input in text areas for FILL-IN, TEXTAREA and any other widget taking input from text areas.

> Please also note that we already have an infrastructure for recording/caching/playback of primitive sequences today.  So expanding that usage is not a massive effort.

The recording itself is only part of the problem. This solution would require new meta model for declaratively describing widget behavior, in a sense a UI "template". The template would instruct the interpreter what recorder sequence to run during any particular state and user input. Obviously for button widget this would be very simple, but not so much for more complex widgets.

Using this technique to speed up even only certain parts of the UI wouldn't help the UX. Mentally the user would have to cope with two-speed UI system. Some parts of the UI would react instantly for his input, some other parts with a lag.

#### #15 - 10/14/2020 09:48 AM - Ovidiu Maxiniuc

We live in the era of intensive streaming. Not only Hollywood productions but games also. This year https://www.nvidia.com/en-eu/geforce-now/ was officially launched. If they can capture mouse/key events, sent to server, process (meaning very intensive computational) and send back to client a 1080@60 fps "movie" without perceptive lag, we should be able to do it as well. We can even steal the idea: the client is just a video player with sensitive frame for mouse/keyboard. The server updates an in-memory screen which is encoded as movie with variable frame-rate to be streamed to client. The compression could be handled by GPU, a resource we do not use at this moment.

#### #16 - 10/14/2020 09:59 AM - Ovidiu Maxiniuc

BTW, yesterday, because of an incremental compile issue, I had a debug session with mouse events. I had a hard time tracking the events. After they are delivered to FWD client, they switch threads several times, using shared queues with other internal events. This is far from perfect and I guess the routing can be optimized.

#### #17 - 10/14/2020 10:05 AM - Hynek Cihlar

Ovidiu Maxiniuc wrote:

> We live in the era of intensive streaming. Not only Hollywood productions but games also. This year https://www.nvidia.com/en-eu/geforce-now/ was officially launched.

If you look at their official requirements, "You'll need to use a hardwired Ethernet connection or 5GHz wireless router."

Certainly the network speeds and latencies improve on a yearly basis, but the reality is that we still have to factor in the network IO. Especially if we want to also provide client for mobile devices.

It would be interesting to see how much time the users of 4GL applications spend on inputting text. I imagine the two major functional areas will be input text and run reports. Assuming this is right, making text input any less fluent/convenient would make many of our end users not happy.

#### #18 - 10/14/2020 10:06 AM - Hynek Cihlar

Ovidiu Maxiniuc wrote:

> BTW, yesterday, because of an incremental compile issue, I had a debug session with mouse events. I had a hard time tracking the events. After they are delivered to FWD client, they switch threads several times, using shared queues with other internal events. This is far from perfect and I guess the routing can be optimized.

I agree, mouse handling in FWD is far from ideal, to be polite :-).

#### #19 - 10/14/2020 10:08 AM - Hynek Cihlar

Hynek Cihlar wrote:

> Ovidiu Maxiniuc wrote:
>
> > BTW, yesterday, because of an incremental compile issue, I had a debug session with mouse events. I had a hard time tracking the events. After they are delivered to FWD client, they switch threads several times, using shared queues with other internal events. This is far from perfect and I guess the routing can be optimized.
>
> I agree, mouse handling in FWD is far from ideal, to be polite :-).

But apart from bad code handling I don't think this poses any immediate performance hits.

#### #20 - 10/14/2020 11:18 AM - Greg Shah

I really like the idea of pushing the text editing down to the JS side. All of our text editing is very slow, I agree this needs work. I'd like to get this done with less work than the full "our widgets are now in JS" approach. I'm hoping that the format string processing + data type + alignment may be enough data to specify the editing for fill-in widgets. The editor has no customer-specific formatting, just a few options that can be set for the entire text area. Both of these would require some real work to push down, but I prefer if we can do it without the full transpiling approach.

> This solution would require new meta model for declaratively describing widget behavior, in a sense a UI "template". The template would instruct the interpreter what recorder sequence to run during any particular state and user input. Obviously for button widget this would be very simple, but not so much for more complex widgets.

I don't think we need this to be generic. I'm OK with encoding the user interaction model directly in JS, on a per-widget basis. The recording is used for the visualization part, where we have all the theming and widget hierarchy still on the Java side. Perhaps the transpiling can be used for the interactions part, if the code can be separated cleanly. But I'm less confident in achieving that quickly.

If button, radio set, slider and toggle box can be easily implemented with recording and client-side interactions, then I'd like to go ahead with these cases.

I think that the selection list and combo-box are also possible, but they are harder. For things like the combo-box drop down or the selection list items, we can draw the entire set of items and then scroll on the JS side. A tricky part there is the selection interaction.

**#21 - 04/20/2021 07:54 AM - Greg Shah**

*- Related to Bug #2971: rework tooltip support to use OverlayWindow added*