hi everyone, and welcome to GUADEC

just so you know, there won't be memes this year.

this talk is less of a presentation, and more of an intervention.

for the past few years, I've been doing what's commonly known as "the Clutter talk at GUADEC". this talk was a generally recap of what landed in Clutter in the year since the previous GUADEC: features, new API, deprecations; and a talk about future things meant to happen to Clutter: features, new API, deprecations. a "state of the union", if you will.

well, during the past year nothing much happened to Clutter. bug fixes. some new (experimental) API. a bunch of further deprecations. compared to the activity between 2011 and 2012, when I wrote the Apocalypse documents, and then implemented them, the year between 2012 and 2013 has been eerily quiet.

the main reason for this is that I think Clutter, as it is, it's pretty much how it should look like.

there are a ton of things that need to improve: layout management, painting, hit detection. basically all of them require an API break. so, during this year, I made the master branch the playground for the 2.0 API, while the 1.16 branch is stable and gets love in terms of performance optimization and bug fixing. all seems fine, and the roadmap is still pretty open.

except that I'm pretty convinced that there shouldn't be a "Clutter 2.0".

now, before you go and take pitchforks and torches, let me clarify that statement.

I don't think there should be a stand-alone Clutter 2.0.

there are various reasons why I changed my mind on Clutter 2.0, and to be fair I still reserve the right to think about it and reverse my position, while simultaneously denying ever having any doubts.

one of the reasons is that Clutter is being used by the GNOME Shell. it's great, and I think it made possible basically everything that the Shell is these days - while, at the same time, enabling Clutter to become more efficient and better. if we look at the adoption rate outside of the Shell, though, Clutter has been an abysmal failure. nobody wants to use it because it's a separate toolkit - and a pretty limited one at that. yes, it was by design; yes, you can integrate it with GTK using clutter-gtk, but the results are pretty bad. the fault lies in GTK as much as it lies in Clutter, but at the end of the day application developers don't care about that: they care about making their application work.

if you look in the moduleset, we have just a handful of users that rely on Clutter (and clutter-gtk, and clutter-gst); of those, two use Clutter just because it's really convenient if you want to push a video on the screen; I think that's more of an abject failure of GStreamer than a benefit of Clutter, to be honest.

on top of that, by virtue of being a library used by our compositor with barely no application involved in the loop, it makes it easy for the Shell developers to ask for special API that should have no place in an application development toolkit.

let's go outside GNOME, though. after all, Clutter was developed in the beginning to be used to write applications for media centers. applications with a very different UI than desktop ones. the main issue is that you need to start writing a toolkit, and that's really *hard*. people get it wrong constantly. I mean, we've had three toolkits based on Clutter - Tidy, Nbtk, and Mx - and it's still pretty much ad hoc code that is never going to be useful as a general purpose toolkit. nevertheless, if you think you can do a better job at writing a toolkit than people that have been doing that for the past 10+ years, feel free to try.

finally, exposing a separate toolkit, with its own rules about event handling, drawing, and layout management is not what I'd call an holistic approach to creating a stable platform for third party developers. when should you use Clutter? when should you use GTK? can you use them both? what will happen to GTK, now that Clutter is here? I've had to field these questions since we presented Clutter to the community at large. I still do.

so, to sum up:

- Clutter is being used to write compositors, not applications
- and even there it has to be gutted and twisted until it's something very different than an application toolkit
- applications use it as a convenience library to paper over lack of features in GTK and GStreamer
- and if they don't, they need to use or write a toolkit an effort that will inevitably go wrong
- in the meantime, we're sending mixed messages to developers interested in our platform

this has to change. we need to change. Clutter needs to change.

I think we should start implementing the scene graph inside GTK.

I think we should start implementing the scene graph as an extension to the current GTK API, as a separate shared library, so that we can progressively migrate applications to use it, until we can migrate GTK itself to be based on it.

let's call it Clutter. let's call it GTK Scene Kit. let's call it Bob.

the more knowledgeable of you will say: "Emmanuele, you're so stupid. GTK already has a scene graph. it has relations between parent and child widgets, and you can traverse the whole tree. you can affect the nodes and leaves in the graph and it will cause actions like size negotiation and drawing".

well done, you; you were paying attention to how GTK was implemented in 1997. and if that design were enough to compete with Motif, then we should all just go home early, and pat ourselves in the back for a job well done.

the truth is, though, that the GTK scene graph is terrible. it's all implicit: the base class only expresses the parent-child relationship, but it does not track the children at all. we have an abstract class that gives you the API to do that, and then classes upon classes to handle actually storing the children and do the layout management. having all the cleverness at the leaf nodes of the class hierarchy means having to carefully chose what to subclass when implementing a new widget. the farther away you go from the root of the hierarchy, the more behaviour you have to deal with, either by relying on it and accepting that it may change, or by working around it; the closer you go to the root of the hierarchy, the more you have to implement yourself. that not only means pain for application developers: it also means pain for the toolkit developers.

this is a ridiculous waste of time. it's ridiculous because it can be completely avoidable through proper use of delegation and API design. it's also ridiculous because we already know that it can be avoided in a backward compatible way. how do we know? it's on the GNOME wiki, under the title of "Apocalypse 1".

let's start designing this scene graph, then.

first of all, let's lay down a few constraints:

- GDK as the windowing system backend
- no input or event handling
- 2D offscreen surfaces blended in 3D space
- limit as much as humanly possible the chances of API breakage

GDK has a lot of issues, most notably the ton of cruft in the API to paper over the fact that Xlib is insane. it satisfies the main requirement for any graphical toolkit, though, in the sense that it works in being able to shove a buffer on screen. that's all we need, so that's all we're going to take.

no input or event handling means that we don't have to care about how users interact with the UI; that is something that has to be worked on top of the scene graph. it also means that all we need is hit detection, to see which node of the graph is at a given coordinate pair. input is hard, so we should all going to shopping instead - but since it's pretty orthogonal, and should reside at a

higher level than the elements on the scene graph, we should safely punt it.

drawing is Cairo's remit, and we can use that plus GL compositing when it involves 3D transformations, to get high quality, and high performance results. after all, this stuff works in delivering your web browser's contents every single day, there is no reason why it should not work for drawing the rest of your application.

finally, breaking API should be a last resort. just because we don't want to take 9 years to go from 3.x to 4.x it does not mean we should break API so soon. plus, there are so many things that we ought to break API for: adding, and using internally, a scene graph should not be one of them.

let's start with some basics.

the ClutterActor graph API is pretty much okay. it maps to a tree and allows to traverse the scene pretty easily and efficiently in both directions. random access is not constant time, but we can take a cue from Gecko and Webkit there, and build a local array of actors for fast indexed access; the local array gets invalidated when you operate on the tree. in general, most of our operations are going to be traversal and enumeration, so we should only optimize this case after profiling.

we should not have a top level class, but top level actors are fine; since we're using GDK, we should just be able to say that a specific actor should be backed by a GdkWindow, and that would make it a local "top level". the reason why Clutter has a top-level are largely historical and wholly not interesting: all we need is a windowing system surface that we can attach to a GL context and that we can use to draw on.

size negotiation is a sore point. doing relayouts for complex scenes, even when hitting the caches, erodes the frame budget we have at our disposal. a lot of this stuff is due to signal emission and property notification, so we should keep those at a minimum, especially in critical paths. from an API perspective, the scene graph node should only have a fixed size, and delegate the preferred size computation to a layout manager object. this way, if you have size constraints pre-computed and assigned by yourself when you’re creating applications, you don't pay any additional penalty. every actor should have the ability to use a layout manager delegate instance, to avoid encoding the size negotiation code inside the actor class itself. I am partial to using objects representing layout meta-information that get connected and composed on top of the actors, more than an actual algorithmic approach to layout. while the latter is easier to write inside the toolkit, the former is easier to use from a UI design tool, and easier to manipulate programmatically without dealing with container or layout properties, and without moving layout properties like expansion control and alignment into the base scene graph class. yes, it adds lines of code, but really: if your goal is to make the shortest application possible you should probably be doing Perl golfing instead of writing applications.

both redraw and resize requests should propagate upwards from the leaves of the graph to the root, and contain the node that made the request; this way we can create nodes that intercept requests from their children and grandchildren and filter them out if they don't cause changes in the size or paint state. relayout and redraw should also be deferred to local roots, instead of performing a full traversal of the scene. scene traversal can be arbitrarily complicated, and it can end up notifying on properties and emitting signals.

an important part is transformations: we not only should allow affine transformations, but also 3D ones. as I said before, if a scene graph is all made of affine transformations we could simply traverse it, and compose it as we do right now; as soon as something requires a 3D transformation we would then need to break out the sub-graph and compose it with GL.

all of this, though, it's pretty much doable without breaking API in GTK at all. actually, all of this would be new API. there is a very good chance that we could transition existing widgets to use this scheme - and, if we can't, there's a good chance that we could simply make widgets draw on a surface that is wholly backed by a scene graph element. this way we would at least be at a point where we can do what we do these days with Clutter-GTK and with GTK, except that it would not require an additional library, and it would be integrated with the GTK frame clock.

"that's great Emmanuele" - I'm sure you're thinking; "but how much work is this going to be, and what have you done to get there?"

well, glad you asked.

what I've done is added GL support to GDK, which is not much, but at least it's a start. we don't need fancy stuff, and we don't really need to wrap GL at all; all we need is being able to create a GL context on any platform supported by GDK. this is actually a feature that should land regardless of being able to have a scene graph.

after that, I started moving the basic parts of Clutter inside the GTK repository - the basic scene graph API - and hook it up to the GDK frame clock. it doesn't do much, right now, but it's a good proof of concept.

during, and after GUADEC, I'm going to actually have a small scene rendered on a GdkWindow.

the next step is being able to shove a GtkWidget inside that scene graph without making the whole thing blow up. this would bring us to basic functional parity with the current Clutter, Clutter-GTK, and GTK triad.

from there on, I guess the sky's the limit - that is, being able to take widgets apart and instead of just drawing them on a Cairo context compose them through nodes in the scene graph, thus allowing transforming parts of them. this last step should be the only one that breaks the API of a GtkWidget.

in short:

- add GL capabilities to GDK
- have a scene graph API sitting between GDK and GTK
- render each (non-3D transformed) sub-graph into a Cairo context
- blend each sub-graph using GL
- ensure that widgets can be embedded into the graph
- decompose the widget drawing code into nodes of the scene graph

thank you for listening to the ramblings of a mad man.

if you have any questions, and if we have time, feel free to ask. if we don't have time, corner me during GUADEC. in any case, I'll try to answer your questions at the best of my abilities.