The Guru's UI Development Diagram (Level 3)

Hey!

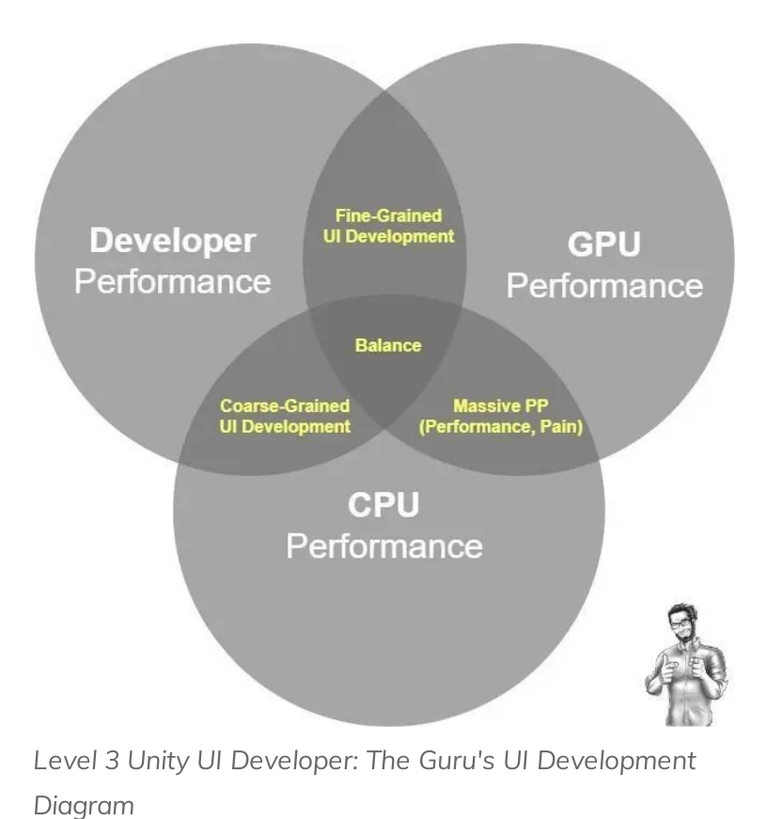
Congratulations on taking action! I am truly happy to see you in our little exclusive corner.

This is a continuation of our little introductory post. Here, you will level up your skills as a Unity UI developer.

You and I are busy people, that's why I want to get straight to the heart of the matter and say...

Let's optimize the hell of the UI!

Now, while I was writing this, I realized I forgot some parts of the connecting article. So here is a quick recap of what we are checking out today: the sweet spots of *The Guru's UI Development Diagram*.



Level 3 Unity UI Developer: The Guru's UI Development Diagram

Let's do it like this. You and I will tackle each UI development method in its specific chapter.

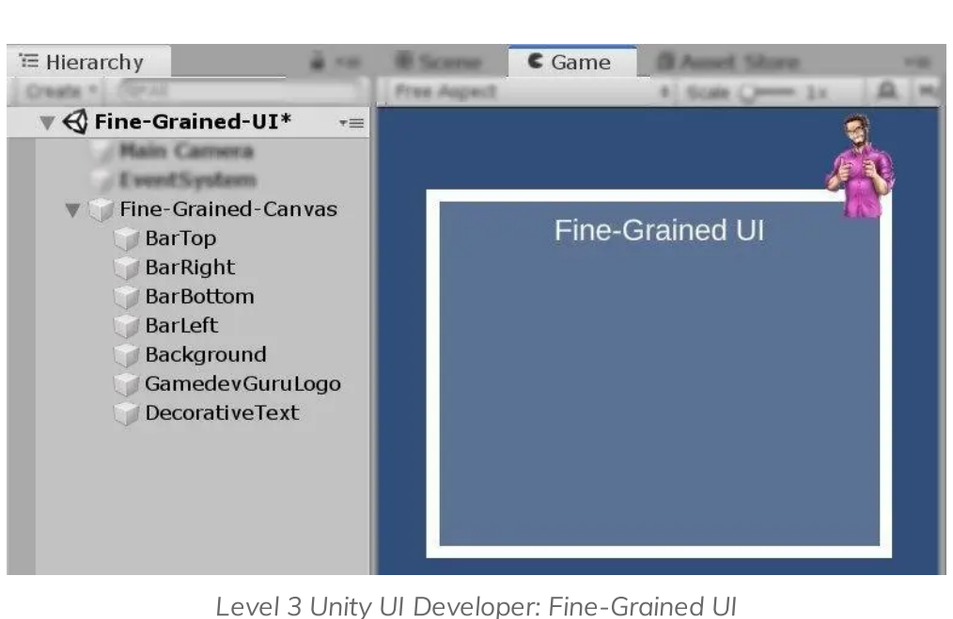
In each section, you will see the Unity setup I prepared for you. I will briefly describe the performance optimization technique and then you and I can discuss the performance analysis of the final result.

I know you might want to follow along, so I included a simple Unity 2019.2.0 project in a GitHub repository. Most Unity 2018+ versions are likely to work.

Download the linked Unity project now. There, you will find a scene for each UI development strategy, which is pretty handy.

Download the project in ZIP format directly from GitHub
Or alternatively, visit the GitHub Repository

I'd like us to start with *Fine-Grained UI development*. Let's go ahead.



Level 3 Unity UI Developer: Fine-Grained UI

Level 3: Guru's Fine-Grained UI Development

The *Fine-Grained UI Development Technique* is an interesting species. It lies in the intersection between *Developer performance and GPU performance*, so the goal is well defined:

In Fine-Grained UI development, we seek to prioritize GPU performance

This approach might be worth for you in cases where your game is, surprise, GPU-bound.

If you decide to follow this approach, you will have a **medium to high number of highly polished elements**.

Think of this as having many small-sized game objects, each of which has very specific, optimized graphics. Each element is usually small in screen-space size to reduce blending overhead. Also, you put special effort in avoiding transparent areas in your sprites as much as possible. Overdraw is therefore greatly reduced.

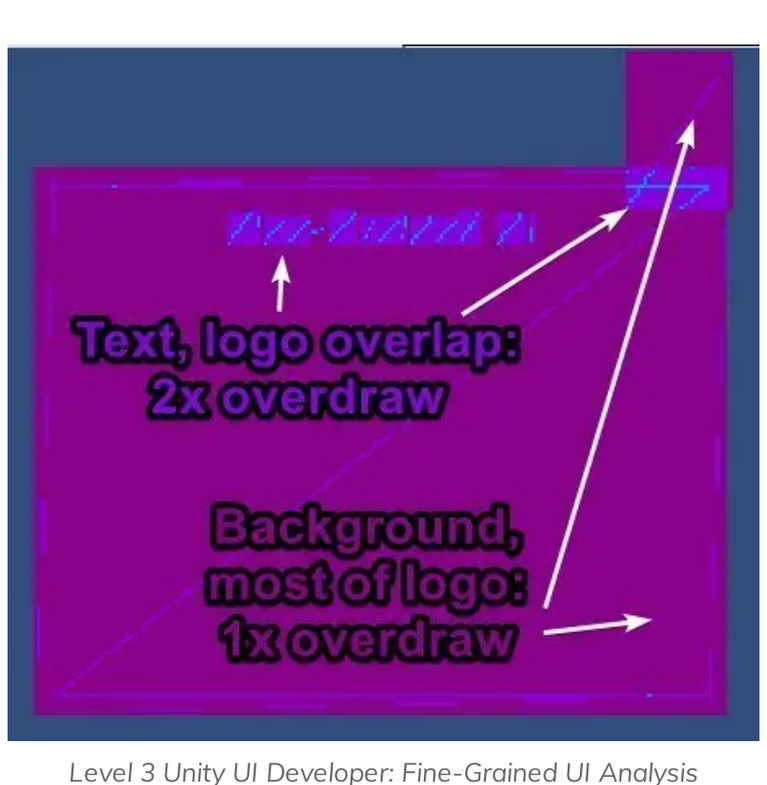
This is a great way of pleasing your players' GPUs.

Let me be clear to you: there are downsides to this approach. The main drawback is the extra cognitive load placed on the developer to keep track of more UI elements. Maintaining larger scenes takes more time and is riskier. Also, keep in mind that the CPU will have a harder time dealing with vertex generation and batching.

See for yourself! In this scene we have seven different elements inside the Canvas.

Now, I promised you we would analyze each solution. It is the moment for doing just so.

Here I present to you a overdraw analysis of the very same UI. No secrets here: this profile was captured with RenderDoc. This is a tool every Unity expert must have in their arsenal.



Level 3 Unity UI Developer: Fine-Grained UI Analysis

I want to have a closer look with you at it. In the analysis we can see the overdraw levels of our *fine-grained UI Design*. Each overdraw level is represented by a different color; in this case two.

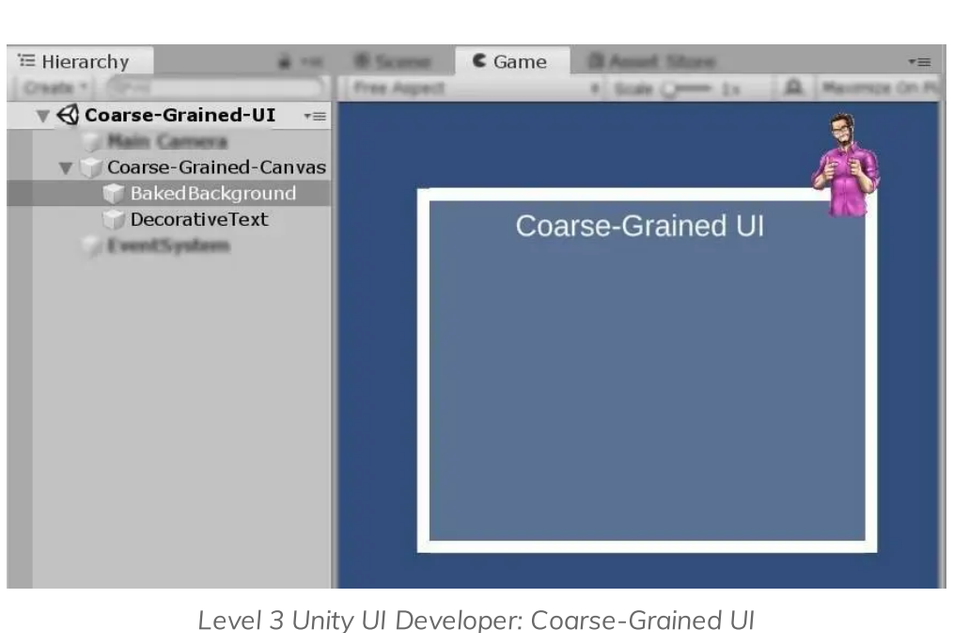
The ~~pink~~* *ciclamino* color presents a single (1x) overdraw level, which is mainly attributed to the background and the most part of the logo.

The light purple color, however, represents a second level of overdraw where pixels were overwritten twice. This is the case for the logo and the bottom part of the logo, which are blending with the background.

As you can see, there is a single level of overdraw for the majority of our elements and only a minor portion shows a second overdraw level. This is minimal and totally acceptable, which is one of the benefits of *Fine-Grained UI Development*.

** p.s. I will confess to you that I had to ask my girlfriend for opinion on how those colors were actually named*

How does the *Coarse-Grained UI Development* differ from this approach? Let's find out.



Level 3 Unity UI Developer: Coarse-Grained UI

Level 3: Guru's Coarse-Grained UI Development

Here we part ways with our previous approach and go the other way around.

In this case, we are intersecting the CPU and Developer performance sections of *The Guru's UI Development Diagram*.

For us, that means...

With Coarse-Grained UI Development, we aim improve the CPU performance

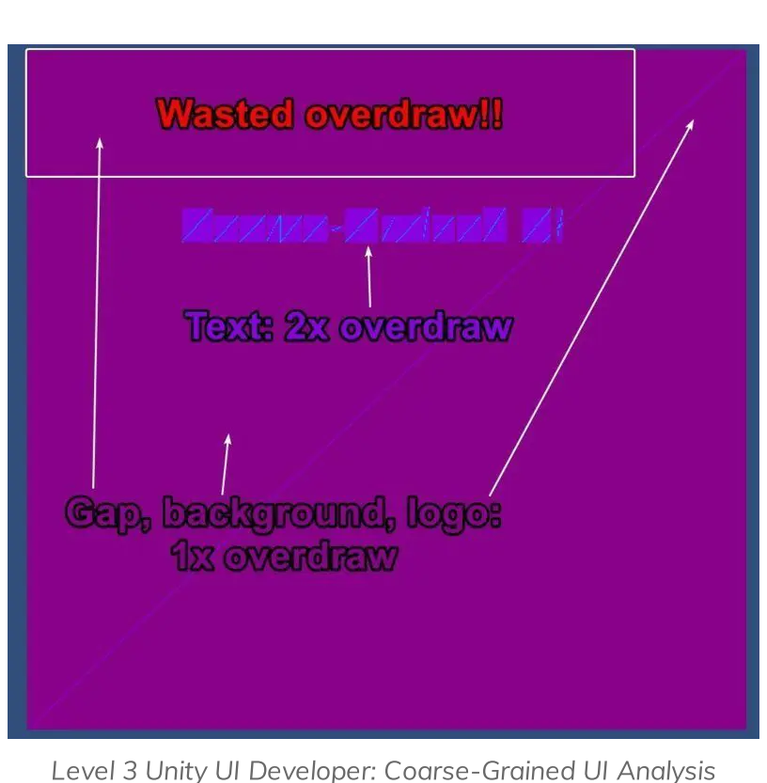
No need to say it, but we, Unity professionals, consider this technique in situations where the CPU is the bottleneck of our game.

You and I can follow this strategy mostly by reducing the number of UI objects in our *Canvas*. Doing this will help us reducing the work the CPU has to perform in order to generate vertices, batches and submit draw calls.

See, what I did here is to combine as many details as I could into a single sprite. This includes all the frame borders, background and the logo. That's an achievable feat even for me, a programmer! Just open *Affinity Photo*, press a few keys and.. *BAM*!

Then, we just need to submit that big combined guy to the GPU, which will take care of it.

Talking about the GPU... How enthusiastic will it be about it? Well, let's ask our analyzer.



Level 3 Unity UI Developer: Coarse-Grained UI Analysis

Oh, fuck, that's no good business. Gimme my money back!

There is a huge rectangle to the left of the logo where we are wasting precious memory bandwidth! This happens as we add tons of useless overdraw. Why useless? Because those pixels we are rendering to will stay blue anyway! In other words, we are drawing fully transparent pixels and that is costing us GPU time.

This extra overdraw happens because Images in Unity UI are drawn as full rectangles. Great, huh? 😑

Again, here's my advice: avoid transparent areas as much as possible in your sprites to contain the overdraw risk.

But this is all very good for the CPU, which only sees two blocks to render (the baked background and the text). So no complains here from its side.

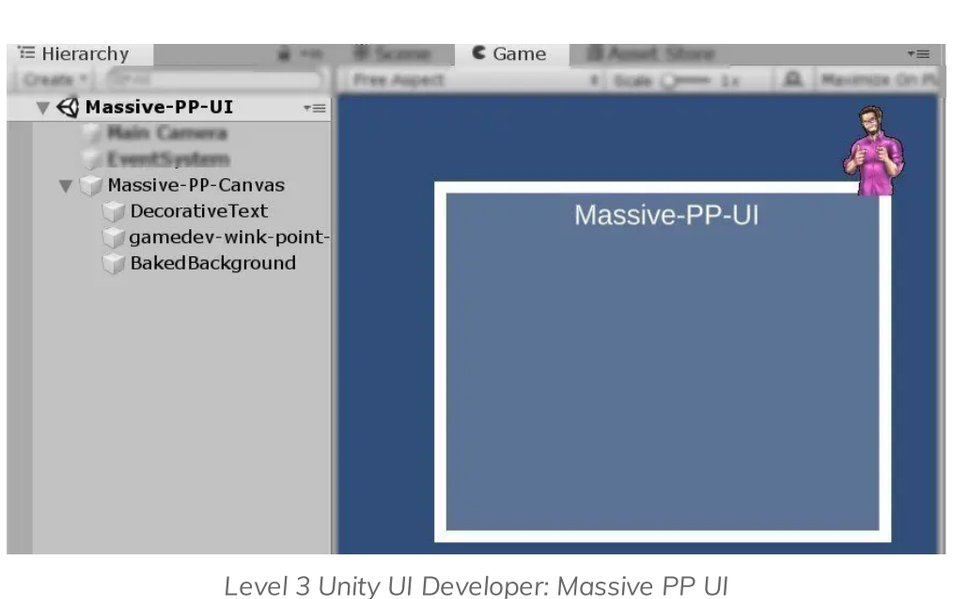
Is this balanced for your application? That you can only answer.

Hey Rubén, I know you are saving your best trick for the last possible moment, cut the chase and give me all you've got!

Right. At some point, you and I might be so desperate to gain massive performance that we decide it is time to go ALL-IN.

Here we cut all the BS excuses and get our hands REALLY dirty. You and I will optimize the crap out of the UI and pull off all-nighters if necessary.

Great, we agree! Let me present you the **Massive Performance-Pain UI Development**.



Level 3 Unity UI Developer: Massive PP UI

Level 3 Unity UI Developer: Guru's Massive PP UI Development

Let me warn you: this is not for everybody. This is HARD.

But we, as Unity Professionals, must have all possible tools at our disposal for all kind of emergencies.

Honestly, the only reason I show you this is because I want to make sure you will be safe, no matter what the challenge is.

The idea here is simple...

In Massive Performance-Pain UI Development we trade our Time and Tears for Performance

This technique has significant changes over the previous ones. Those will take longer to assimilate, but this will be REWARDING as a glass of ice-cold soda after a hard day working in a farm under the sun.

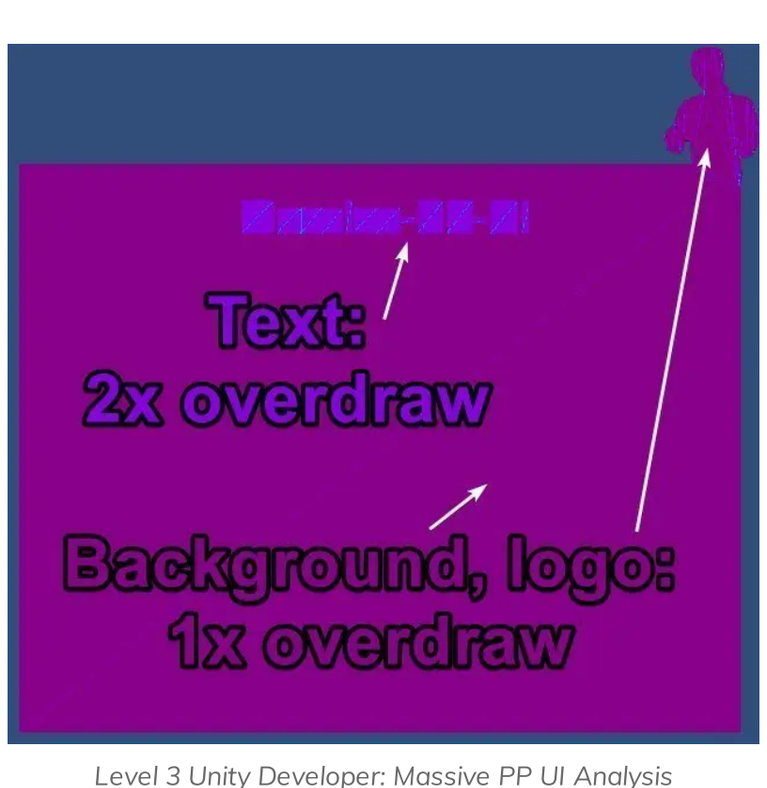
Open the scene now and see for yourself. Some changes are subtle and easy to miss, so be on your guard. Just in case, I have your back covered.

Here is a list for you containing the major changes I performed:

- **Text**: I trashed *TextMeshProUGUI* and used instead *TextMeshPro*. This variation is lighter on the CPU, but its layout is harder to tweak and maintain.
- **Image → Sprite**: the logo image was swapped with a sprite. The logo is now less attached to the Unity UI system, it became more of a *normal* 3d object in the scene. Congrats!
  But why would we do this?
  My friend, here is the key: sprites are not necessarily rendered as rectangles. That means, you can create a custom, tight mesh out of those. And that is cool.
- **Custom Sprite Opaque Shader**: oh dear, this one is fun.
  Since we have a tight mesh on the sprite, we do not need blending anymore. This is why I got the default sprite transparent shader canned.
  I made an opaque variant that fills its content in a way that nothing below it will be drawn.
  In other words, the logo will be drawn before the background and will write a value in the *stencil buffer* to say "*hey dude, I drew here, please do not redraw this pixel*".
  Less overdraw for us. Additionally, blending was disabled; less work for the final stages of the render pipeline.
- **Custom UI Stencil-Tested Shader**: the default UI shader will not listen to our sprite's words; it will decide to ignore its request to avoid drawing below the sprite. Bad boy.
  For this very reason I adapted the UI shader, so we make it GPU friendly.
  I did so by adding a stencil test to avoid the overdraw coming from the opaque sprites drawn on top. If a sprite wrote its mark in the stencil buffer, then the GPU will skip rendering this fragment.

If you think I just talked gibberish, that's not ok. Drop me an e-mail so we can sort things out. I can consider writing a more in-depth explanation on all of these steps.

The P's are there for a reason. And the analysis below proves it.



Level 3 Unity Developer: Massive PP UI Analysis

Massive Performance? Yes, amazing!

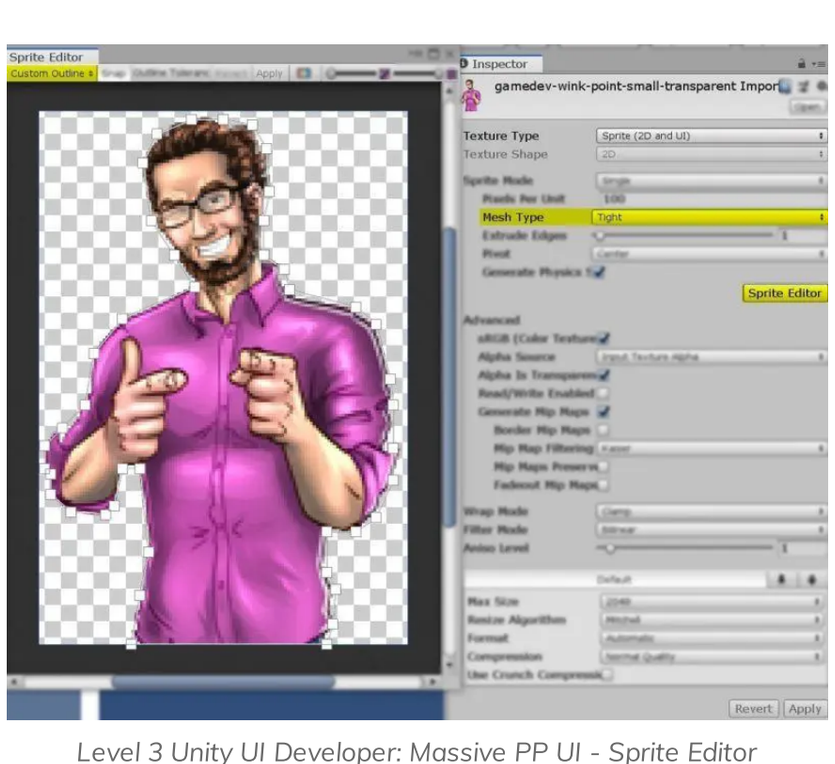
Three single objects, almost no second level of overdraw!

Theoretically, you can easily get it in even one. What about this? Bake the text and the logo into the background, make it a tight sprite and reap the rewards.

That is Massive Performance.

Massive Pain? Your call.

But before I go..



Level 3 Unity UI Developer: Massive PP UI - Sprite Editor

Level 3 UI Developer: Sprite Mesh Generation (Extra!)

I couldn't leave you guys without briefly discussing something you might have noticed.

The logo is a sprite with a custom outline attached to it! And that is cool because it is not rendered as a quad but as a mesh, reducing tons of overdraw.

How the heck did you do that, Rubén?

First, it has to be a *Sprite*. Forget *Images*. It is a bit of manual work, but it poses no great difficulty, really. Just follow the steps below and the previous screenshot. Make sure to zoom in and yeah, give it a shot.

The first thing you have to do is to navigate to your sprite in the project view. Select it so you can see the import settings. There, you want to make sure that its mesh type is set to tight instead of full rectangle. That will allow us to open the sprite editor and then switch to the custom outline mode.

In this magic window you can then start creating the outline of your sprite. You will trade overdraw for vertices, quite a nice deal for mobile. I did this process for the logo and honestly found it more entertaining than watching the last season of *Game of Thrones*.

Level 3 UI Developer: Conclusion

You read till the end. Or, maybe you skipped till the end. In any case, congratulations, you did it.

Now I can say this: my first goal was to awake your interest in improving your UI development workflow. My second objective was to give you the tools you need to quickly evaluate the implementation of user interfaces in Unity.

For this to happen, **The Guru's UI Development Diagram** was the best concept I could come up with for you.

This is what I will ask of you:

- Tolerate no BS.
- Use your Level-3 understanding in Unity UI to...
- Speak your mind in front of your colleagues.
- And **become the final boss no one can defeat**.

Now it is your turn to put them into practice. Send me an e-mail about how this all went and go have some fun!

Ruben