Typesetting in Aegisub

Introduction

This guide was written mostly in 2011-2013, with only a few updates after that, and therefore is obsolete in many ways. Some changes in the future are possible but unlikely.

Things were very different in 2011 when the first chapters were written.
There were (almost) no automation scripts, only a handful of people knew how to use mocha, nobody was using xy-vsfilter yet, and everything lagged like shit.
Much of this guide still reflects those times, so you shouldn't necessarily take every advice you find here. For example typing any tags instead of using automation would be pretty stupid today and would cost you a lot of time. There are scripts for almost everything, written both by myself and others.

It should also be noted that nobody gave a shit about libass when this guide was written, so everything is written assuming that you're using vsfilter and nobody will watch it with libass.
Recently, though, libass has been evolving probably faster than vsfilter and thus become much more relevant, so typesetters may want to consider making sure their signs work the same on both renderers.
For most things, this is not a problem, but there are some issues that each renderer handles differently. I don't really know what they are, as I'm mostly out of the game, but I know some of them concern rendering fonts.

If you want to typeset seriously and want to have up-to-date information,
I highly suggest joining **#irrational-typesetting-wizardry** on rizon.
There you can talk to a lot of people who can help you with all kinds of problems, as well as directly with authors of commonly used scripts, Aegisub/vsfilter/libass developers, etc.

You can find a lot of stuff on **this github link**.
It includes my scripts, lyger's, torque's, line0's, Youka's, etc.
You can find all these people in that irc channel I mentioned, as well as Plorkyeran and jfs who (used to?) work on Aegisub, and rcombs who works on libass. At least that's the situation at the time of writing this, February 2015.
(2018 Note: Aegisub hasn't made any progress since 2016)

What has changed since the guide was written?

Things lag much less, so warnings about lag probably aren't relevant anymore. Today you can have many transforms in one line, track several signs at once, or make huge gradients without any lag. It's usually only if you combine too many of these that things get a bit slow. If there are two methods to do a sign, it's not always easy to tell which one will be less laggy, so sometimes you just have to try.

Mocha tracking is a must, and if you want to typeset decently, you definitely have to learn that.

Vector drawings have become very common for masking and other purposes, and there are tools to create them easily. Don't use stupid shit like FansubBlock.ttf.

Some of my own examples in this guide, while they may have looked great in 2012, don't look so good now. A lot of them were made without any scripts and with incomplete understanding of all available tools, since even when I was writing the guide, I was still learning myself.

Aegisub evolved a lot between 2011 and 2016. Some things in this guide regarding the program may be obsolete.

It's not uncommon to do one sign in 5 layers or more. If it's relatively easy to do and makes the sign look better, go for it. It's unlikely to lag even when mocha-tracked.

While I haven't gone in that direction myself (I only use Aegisub, ASSDraw, and Mocha), it's pretty common to use other software, like Illustrator. Check line0's and torque's stuff for tools and guidelines for that.

Sorting by time is not needed anymore. Only old vsfilter had that bug, and both xy-vsfilter and libass have no problem with it.

Some general notes about typesetting

There is no single, universal goal of typesetting. Different typesetters have different methods and goals. So don't really let people tell you how you "should" do it.
Typesetting is largely about the ratio of time and effort you put into it and the quality of what comes out. What you want this ratio to be is up to you. You may want the best results, or you may want it done fast, or you may want something in

between.
Hdr was a typesetter who usually went for the best possible quality, at the cost of endless time spent on it and often breaking people's computers with massive lag.
Examples of those who go for speed over quality are of course many.
My goal has always been to find a reasonable ratio. There were certainly times when I went for higher quality than today, but I've (almost) always refrained from spending too much time making only little improvements, especially for things that people normally watching the video (without pausing) would hardly even notice. Of course, you will always find people who will criticize you for that, but fuck them. They usually don't know shit about typesetting anyway.

You should never listen to typesetting advice/criticism from people who don't typeset, and you should never listen to ANY advice/criticism about anything from Dark_Sage and Kristen.

There's no rule saying that you have to typeset every single sign you can find in the video. It's really up to you and your group to decide which signs you should or want to do.

I would also like to mention a few things about matching the original text with your TS. While you should definitely learn to do that and keep it as a general guideline, I don't think it's necessary to follow it as a rule set in stone. Sometimes a sign typeset differently from the original may look better than one that imitates the original as much as possible. This can have various reasons.
For example, there may not be enough space to do the same in the same way the original looks, and trying the same look in smaller size could look really bad (especially with borders and shadows), so making a smaller typeset in a completely different font, possibly without or with fewer borders/shadows, may be a decent solution. What really matters is whether it looks good and the viewer doesn't stop and think "WTF is that?"
If you pay attention to cases where Japanese studios add an English sign alongside the Japanese one themselves, you will find that the English one often looks very different from the Japanese. If they don't deem it necessary to match the two perfectly, you shouldn't really have to either. But it has to blend in somehow.
I have recently gone more in the direction of not matching things that much. I prefer my typesetting work to be creative rather than just tedious slavery following exact rules.
If you make a sign that looks good and someone who doesn't typeset complains that you didn't match the original and you should have, just tell them to fuck off, and don't waste your time on arguing with them.

This may be even more relevant with episode titles. Episode titles are usually not a part of the "picture". They're added text, with little to no graphical connection to the video.
So for starters, I see little reason why you should place your typeset right under or above the original. You can place it at the opposite side of the screen, or whatever happens to be convenient or look good.
As for matching the look, it will obviously look weird in most cases if you just pick a completely different font, but that also depends on the font. English never looks like Japanese, and English text that matches the Japanese font often looks a lot more "boring". So using a font that doesn't match 100% but looks nicer can be a good solution for titles.
I also never understood why people try to acquire the actual Japanese font and use the Latin part of it for TS. This to me is mechanical, robotic stupidity. Firstly, just because it's in the same ttf file doesn't mean the English actually looks like the Japanese, and secondly, most of those I've seen could easily be replaced by hundreds of similar fonts, saving you from using a 10 MB font and/or from looking for the exact Japanese font. It's of course up to you, though, how much time you want to spend on pointless shit like that.
Then again, not everyone is creative, and some people just like to imitate and match things blindly because they don't really have a mind of their own.

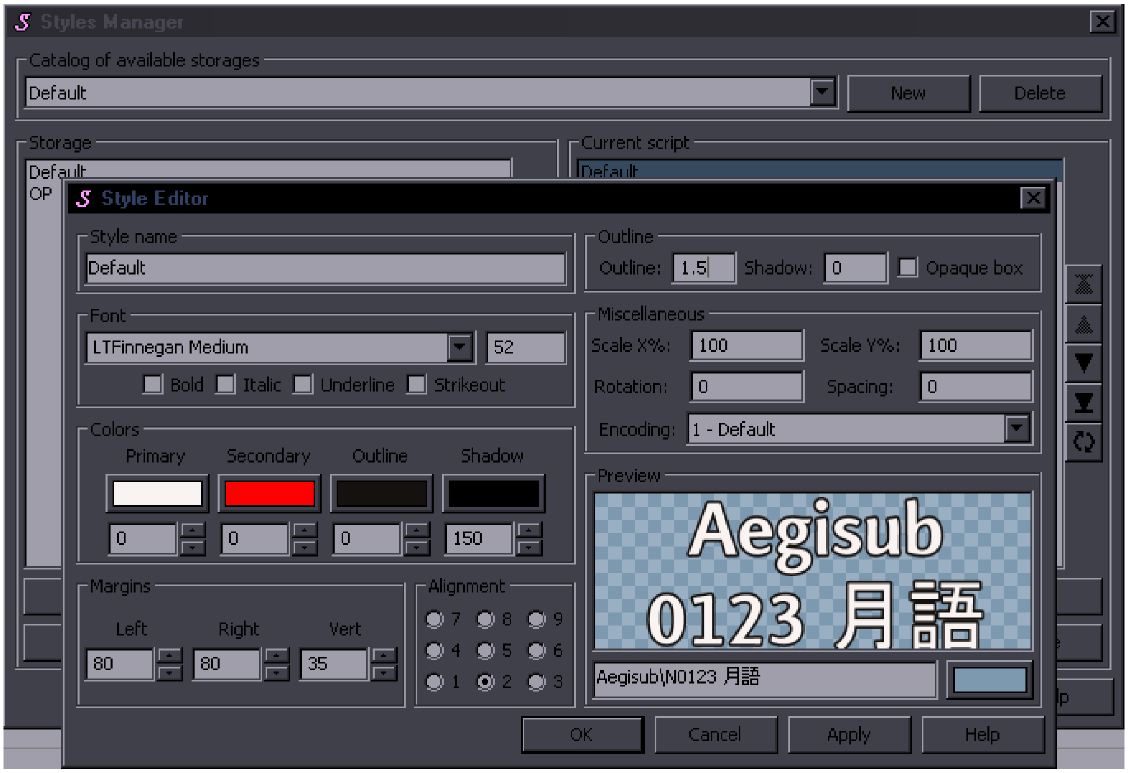
Consistency. Yeah, there's that. While it's certainly nice if you can make repeating signs consistent, the fact is that the more I typeset, the more I notice that Japanese studios are inconsistent as fuck, and making a consistent typeset may often be not only difficult but virtually impossible. So now that I don't really care so much, I just do whatever works instead of wasting time trying to do the impossible. If the studio is entitled to inconsistency, then so am I.

Instead of rewriting things, I have recently added some notes in this colour reflecting on what has changed since that particular part of the guide in question was written.

« Back to Typesetting Main

Typesetting: Creating Styles

The first thing you'll want to do before you start typesetting is to create some styles. You do that in the Styles Manager / Styles Editor, which looks like this:



Catalog of available storages: Default

Storage: Default, OP

Current script: Default

Style name: Default

Font: LTFinnegan Medium, 52

□ Bold □ Italic □ Underline □ Strikeout

Colors: Primary 0, Secondary 0, Outline 0, Shadow 150

Margins: Left 80, Right 80, Vert 35

Alignment: ○ 7 ○ 8 ○ 9 ○ 4 ○ 5 ○ 6 ○ 1 ● 2 ○ 3

Outline: 1.5 Shadow: 0 □ Opaque box

Miscellaneous: Scale X%: 100 Scale Y%: 100 Rotation: 0 Spacing: 0 Encoding: 1 - Default

Preview: Aegisub\N0123 月語

First you need a Default style, which will be used for all the regular lines. It already exists when you open Aegisub, but looks terrible, so you'll modify it. First you need to **choose a font.** This is an important choice, because you'll be using the font for all episodes of the show and people will have to be able to read it. So you need one that is easy to read, doesn't have any distracting, unnecessary decorative elements, and isn't ugly. It should also have real italics and should contain all necessary punctuation characters [.,!?-—'"].

Next you need a reasonable **font size.** That is related to script resolution, which will usually be 1280x720. Even if you were in a group that does 720p and ~~480p~~1080p, I'd still recommend making scripts in 720 (or 1080), since it will work for 480p just the same and lets you have more detail in almost everything. To determine what the right size is, you can use a long line from an existing script, and just see how it looks with the size you want. It shouldn't be too big so that you don't get 3-liners, and it shouldn't be too small 'cause then you can't read it. (duh)

Depending on what kind of font you choose, you may want to check the box for **Bold**, since some fonts look bad in regular mode but good in bold. You will not use Italic for the default font, because that's dumb, and you will most certainly not use Underline and Strikeout, because that's even a lot dumber and only jdp does that. (that actually happened once)

Then you have the colours. **Primary colour** is obviously the main one. I strongly suggest you don't use anything other than white for the default font. You may think something else looks cool, but it won't seem so cool when you have to read it for 20 minutes.

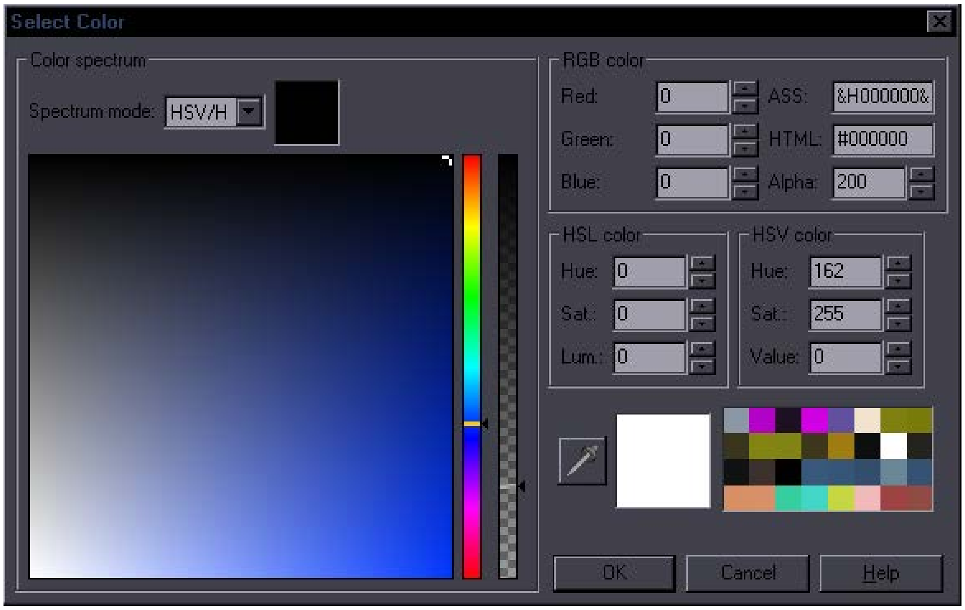
Secondary colour is mostly just for karaoke, therefore quite irrelevant.

Outline colour is for the border. Again, for the default font, use black or something dark.

Shadow colour should clearly be black, because things really don't cast green shadows.

Under the colours, you see boxes with zeroes where you can set transparency for each of the colours. 0 is fully visible, 255 is fully invisible. Clearly 0 for Primary (and secondary), possibly a low non-zero value for Outline, if you know what you're doing (but recommended 0), and something in the range of 120-200 for the shadow (still talking about default font), because shadow with no transparency will make you look like some retard who learned typesetting in Hadena. (Check some Hadena releases from around 2011. It's hilarious.)

In Aegisub 3.0 and later it looks like this when you click on the colour:



Spectrum mode: HSV/H

Red: 0
Green: 0
Blue: 0
ASS: &H000000&
HTML: #000000
Alpha: 200

HSL color: Hue: 0, Sat.: 0, Lum.: 0

HSV color: Hue: 162, Sat.: 255, Value: 0

Transparency is the black/white vertical slider and/or the "Alpha" box with the "200" on the right.

Margins are important as well. I don't suppose I have to explain why it's bad if the margins are either too small or too large. If you actually need that explained, give up on being a typesetter and save us the trouble of trying to teach you. Reasonable left/right margin for 720p is around 80-110; reasonable vertical margin (that means both top and bottom) is around 30.

Alignment is obviously 2 for the default subs. The other ones can be useful for OP/ED. I generally use 5 for signs.

Outline / border size. For the default font, don't try to invent anything much. It has to be readable before anything else, so no ultra thin lines, and no crazy borders thicker than the main font. 1.5-2.5 will probably work fine, but it depends on the font etc.

Shadow distance. For default font, I don't like using shadow at all, but if so, make it a low value (like 1 or even less) and a lot of transparency. Unlike all the previous values, border and shadow don't have to be whole numbers, so you can use something like 0.6.

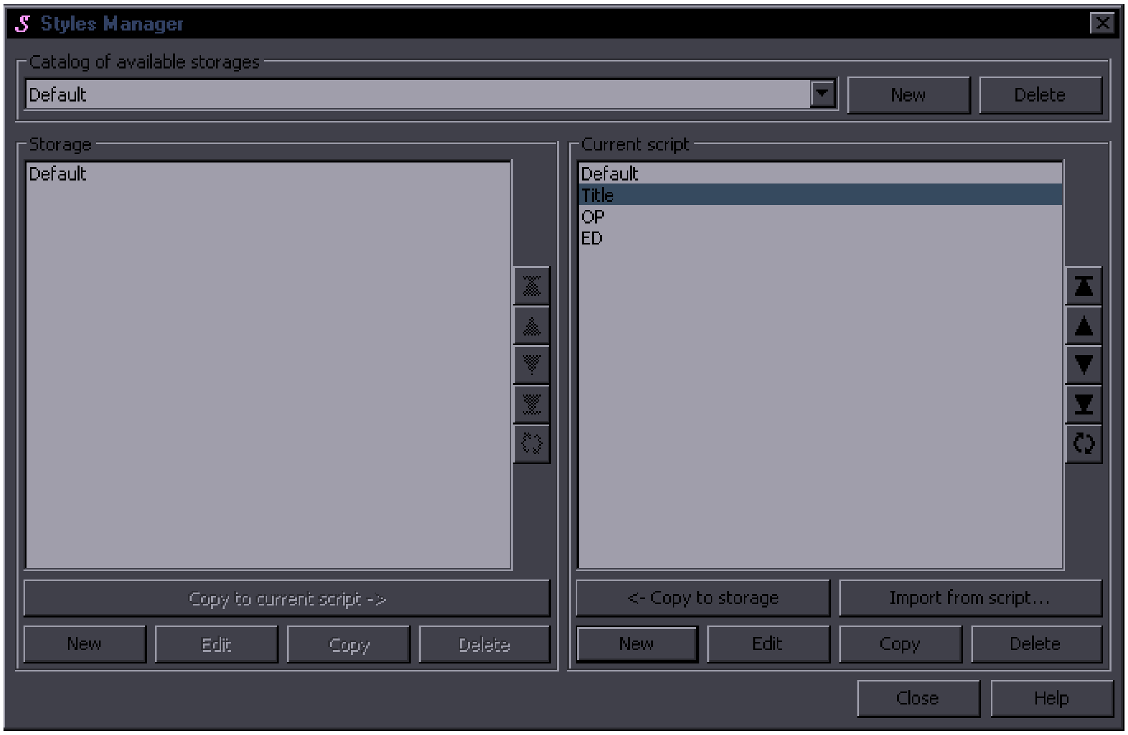
Scaling. If your font looks good but is too narrow or wide, you can adjust that here. And again, for the default font, don't do anything silly.

Rotation. After ~~2~~ 7 years of fansubbing I can safely say that you will never need this.

Spacing. Increases spaces between letters, obviously. For default style, some low values like 0.5 may be helpful for some fonts.

Encoding... is probably useful if you're making chinese subtitles. Which you're probably not. So whatever.

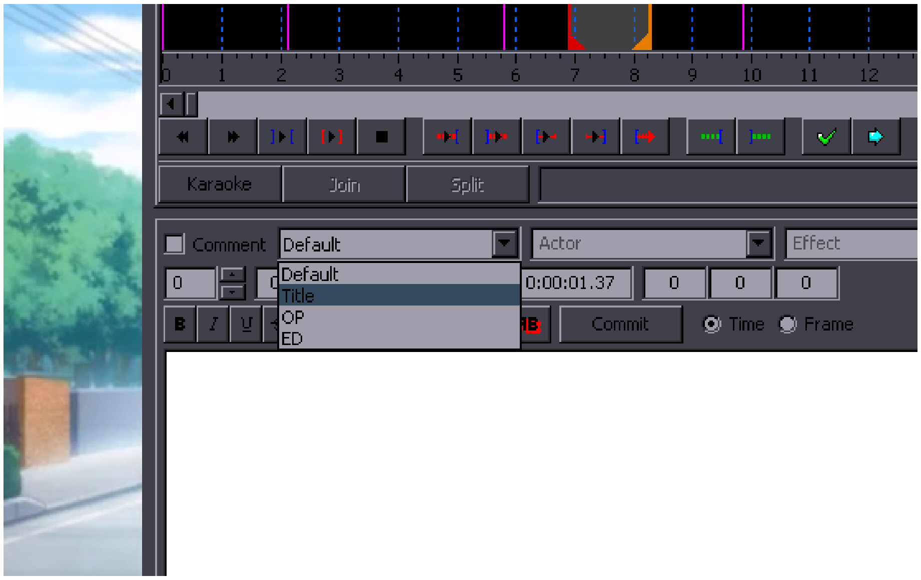
That takes care of the default font.



You will certainly want to have more styles than just the default. You'll need a style for the OP and ED and probably for episode titles. You'll create them by simply clicking on **New** and then it's back to what we've just gone through... starting with a new name. For OP/ED you can divert from the default values much more - all the colours, border, shadow, size, scaling, spacing... are pretty much without limitation. For the episode title, you'll ~~obviously~~ probably be picking something that resembles the original title on the screen.

After that, you can create as many styles as you want for whatever you need to typeset. Some people create one style for typesetting and then override everything with tags for each sign. I think that's pretty silly, gives you extra work, and swarms the script with unnecessary tags. I prefer to create a separate style at least for each font I'm gonna use, since it's a lot more convenient than changing the font with tags. And obviously if there are certain kinds of signs that are used repeatedly throughout the show, it's good to have a style for each of those.

How many styles it's useful to have depends on the amount and type of signs in whatever you're typesetting. Sometimes you'll want 2 styles for the same font. For example one black and one white, or one with border and shadow and one without. If the episode has 20 signs with that font, 10 of them white and 10 black, I'll create 2 styles rather than having only a white one and using the \c&H000000& tag 10 times. To do that, you create one style, then click on **Copy**, change the name and the one thing you want different, like colour. This can be used even for alignments. When you have the same kind of sign 10 times on the left and 10 times on the right, you can have 2 styles rather than typing \an? 10 times. After all switching styles is easy:

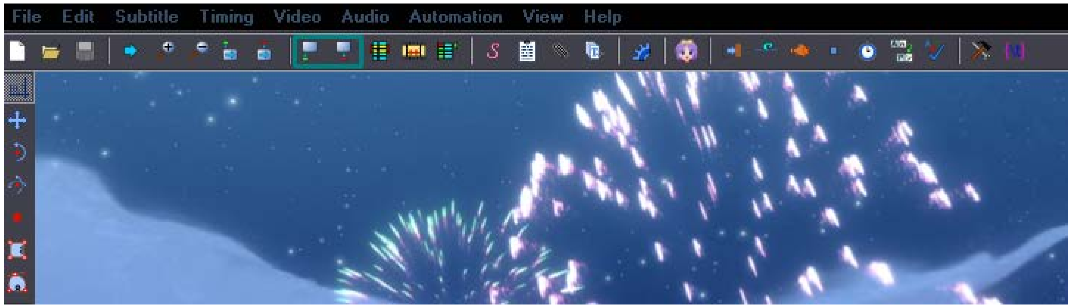


One last note. Pay attention to what fonts you're choosing. Remember each font has to be muxed into the mkv, so don't pick 10 MB fonts. Any font, even a fancy one, should be under 1 MB. The only exception is if you need to use kanji. From the list of fonts in Aegisub, avoid those that start with @, avoid those that have Adobe or MS in the name, as those are likely to be huge.

« Back to Typesetting Main

Typesetting Basics

In order to typeset a sign, you need to time it first. As this needs to be precise, you don't do it on the audio track like with regular lines.
Signs need to be frame timed, because if your sign is one frame off, you belong in Hadena.
So you rough time the sign first, whether on the audio or by inputting the timecode you got from the translator/typist/editor.
(Or, if you don't live in the past, you use **this script**.)
Then you go frame by frame with arrow keys till you find the first frame where the sign appears [the relevant line must be selected in the script].
Then you click the first of the blue icons here: (No. Use fucking hotkeys.)

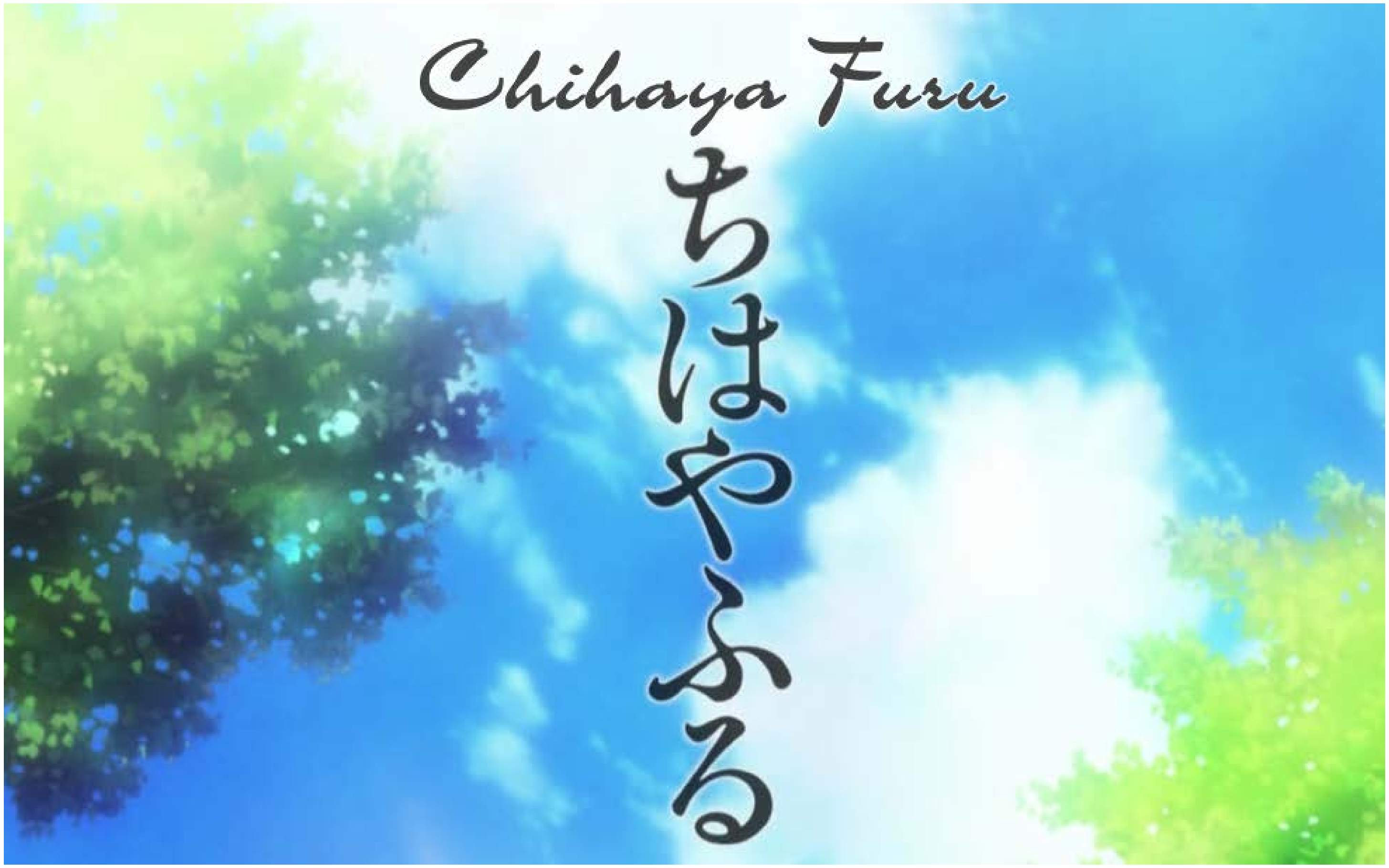


That sets the start time. Use arrow keys to check if you did it right. Then navigate to the last frame the sign is visible on and click the second one.
Again try if it's right. The first one sets the time at the start of the visible frame, the second one at the end, so if you use both at the *same* frame,
the sign will be visible on that frame, in other words the duration of the sign will **not** be zero.
This can be used for typesetting signs frame by frame by hand.

Many signs start/end at a keyframe, so you can use the audio track for those; for others, you'll need this method.

When you've timed your signs, you can begin typesetting. You already know how to create styles, so you'll make one.
If you need to override anything, you'll use a few tags like \fs \bord \shad etc. You should know all the basic tags and what they do **from here.**
Note 2018: I've never really used \fs once I learned to TS properly. **Recalculator** + scaling = size changed in 2 seconds.
Border is probably what you'll be changing the most often, maybe the shadow and font size, so you should remember \bord \shad \fs (nope) at least.
You should remember the hotkeys to the **Cycles** script, that is.
You may also switch between regular and bold quite a bit, but bold/italics have buttons above the typing area so no need to type those.

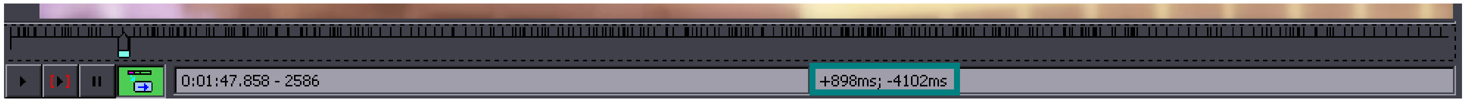
Let's begin. Here's a simple typeset...



This is something that will appear in every episode, so you want to set as much as possible in the style.
The only tags I add here is \fad and \blur. The font, colours and border are set in the style.

Don't make the mistake of using the default (or any extreme) values!

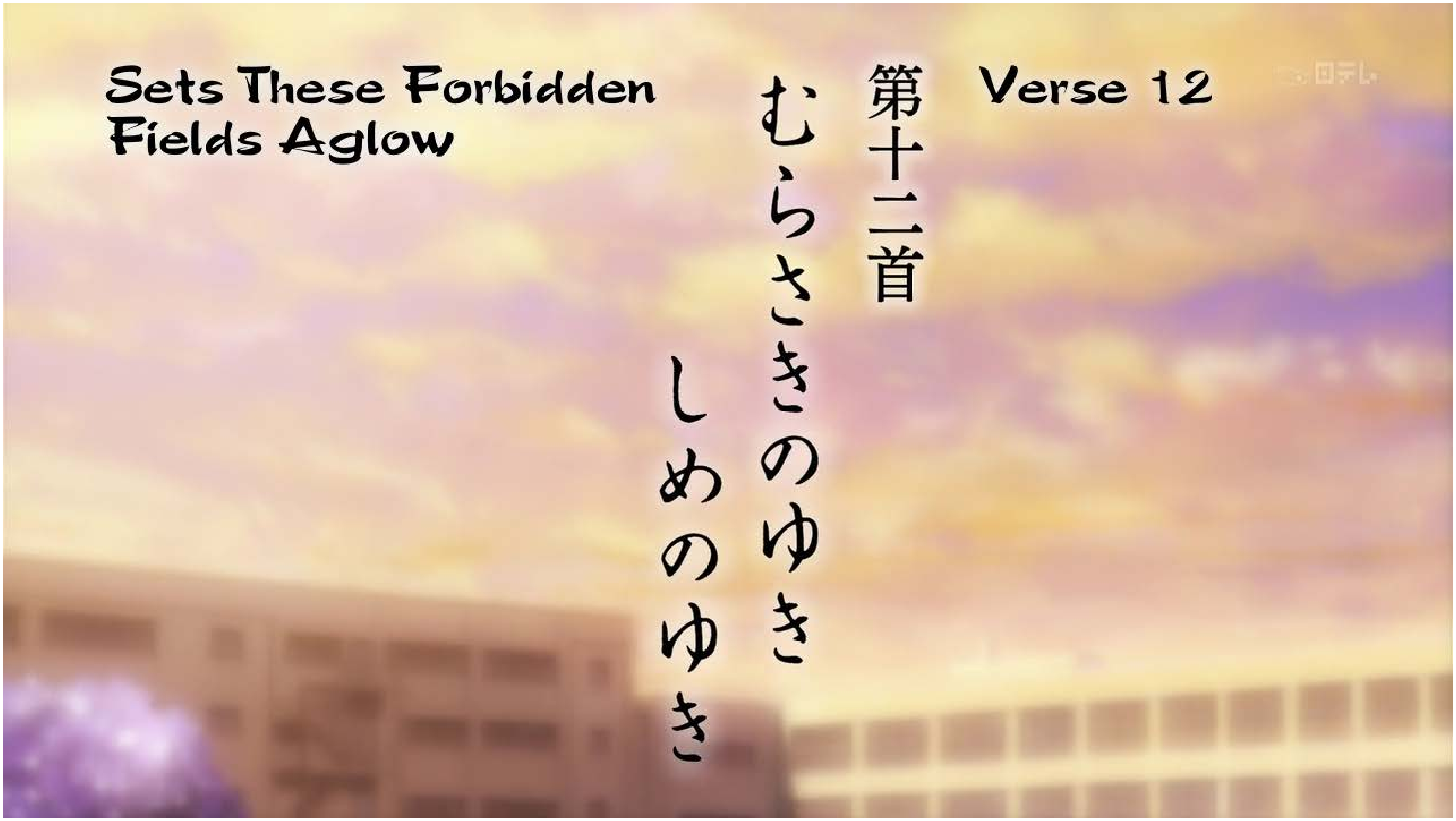
For example the default has a shadow, but you definitely don't want a shadow here, so make sure you set it to 0.
Also don't just assume it's black and white. If you do that, you're going Hadena-style. Get the actual colour from the Japanese sign with the eyedropper tool.
Aside from the colours and the border, what will make a difference between a good and bad typeset here is the font choice.
So don't just take some basic serif or sans serif font, or something like ComicSans, but find something that will actually match and look good.
Speaking of which, you need to know your fonts, and you need to have them in the first place.
Also, that sign has no layers and looks like shit anyway.



0:01:47.858 - 2586

+898ms; -4102ms

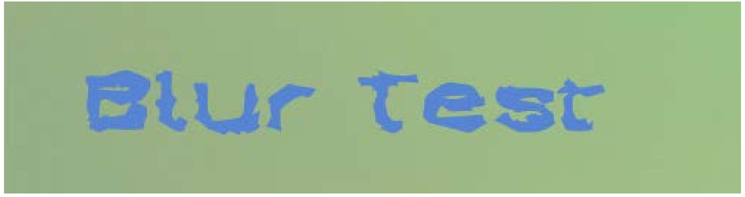
Now about that fade...
Use arrow keys to go frame by frame and find the place where the fade of the jp sign ends. ^ Check the numbers here.
They refer to the currently visible frame, in relation to the start/end time of the line you have selected.
So this frame is 898ms after the start of your sign and 4102ms before the end of it. If this is where the fade in ends, you need about 900ms fade in.
No need to be too precise, one frame is about 40ms, so 10ms more or less won't make a difference. You will then have \fad(900,0).
If there's lead out too, you do the same but use the second number.
2015 Note: **Apply fade** makes it *much* easier.



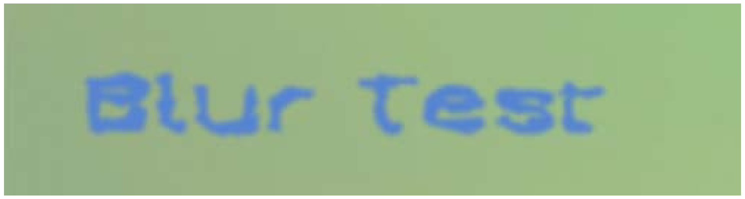
Another example of a title. This one uses \blur10 (for the border). Be aware that using this much blur may cause lag, so only use it on simple static signs.
2015 Note: That was 2011. \blur10 shouldn't be a problem anymore.
2018 Note: That was 2015. \blur10 definitely isn't a problem.
Notice the positioning, too. The top of the typesets is aligned with the top of the Japanese sign.
Don't just throw the signs somewhere randomly, try to align them with *something*. Also for episode titles and such, try to keep the same positioning in following episodes.

A few notes on using **blur**.
It works differently depending on the borders and shadows you're using.

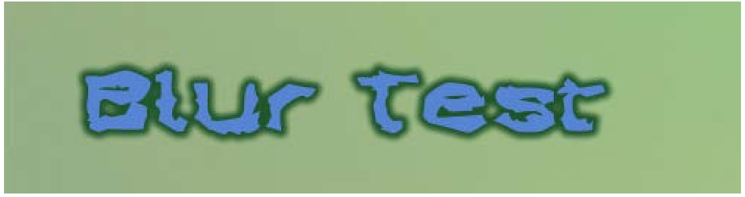
This is text with no border, no shadow, no blur... \bord0\shad0\blur0:



No border, no shadow, with blur... \bord0\shad0\blur2:

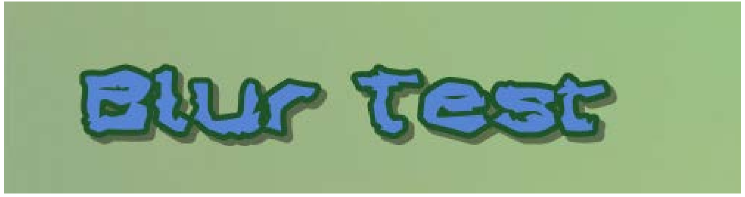


Border, no shadow, blur... \bord4\shad0\blur2:

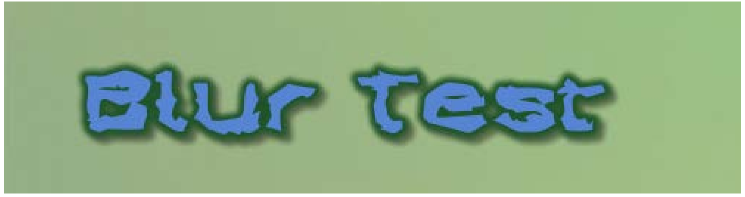


You see it will only blur the outline, not the body.

Border, shadow, no blur... \bord4\shad4\blur0:

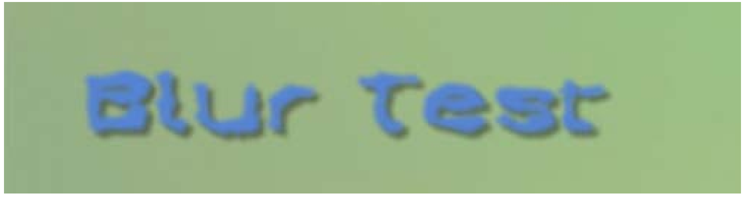


Border, shadow, blur... \bord4\shad4\blur2:



This blurs both the border and shadow, but not the body.

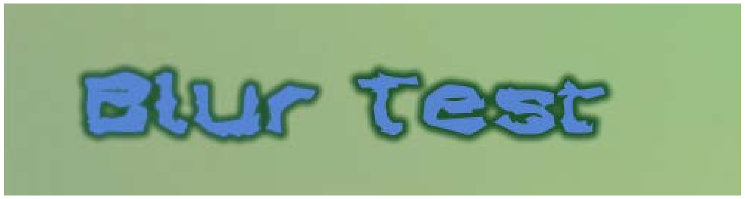
No border, shadow, blur... \bord0\shad4\blur2:



This, however, will blur both the shadow AND the body.
So you see it's the border that determines whether the body will be blurred or not.

You can't separate the border from shadow for blurring.
If you use both, both will be blurred.
To bypass that, you'd have to use 2 layers. More on that later.

Same if you want to blur the outline AND the body, you need layers.
It would look like this:



"Test" is regular mode, "Blur" is 2 layers with the body blurred.
Again, more on that in the "Layers" section.

Note: This works the same with 'blur edges' - \be.
Experiment to find out the difference between one and the other (shows more with higher values).

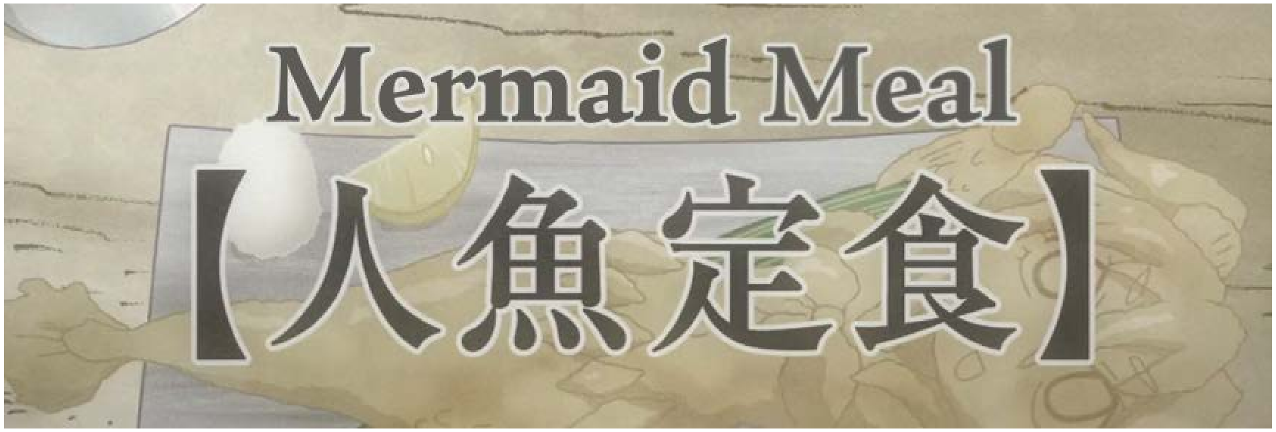
Two things related to blur:

Blur is the most essential tag for typesetting.
Signs without blur look like shit, so never forget to use it.
What I do before I start is to use a script to add blur to all signs. Check the **scripts section**.
Note 2018: Actually, that doesn't really matter. If you're doing things right, adding a default blur should require pressing 1 (one) key.
That way you start with blur already present on all signs. 0.5-0.6 will work most of the time. You'll change it to higher when needed.

NOTE 1: Do not use \blur.5 instead of \blur0.5
NOTE 2: \blur0.3 does nothing visible. \blur0.4 blurs VERY LITTLE and is only applicable for really sharp video.
Don't swarm the script with \blur0.3 when you can see it's not doing anything. ALWAYS check signs at 100% zoom. Use your damn eyes.

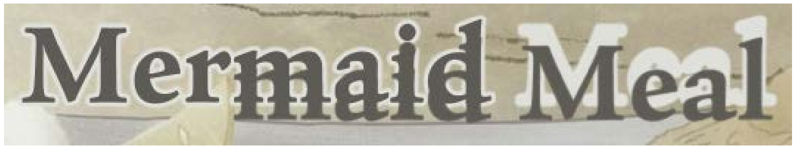
Here you can see 3 modes of using blur:

1. The "Mer" part. Completely wrong, because it's only 1 layer and the body is not blurred.
2. The "maid" part. Also wrong. It's two layers, but the primary colour of the bottom layer is the same as the top layer.
3. The "Meal" part. This is correct. You can see it looks like the Japanese sign.



If you can't see the difference, you're blind (or your monitor is terrible) and probably shouldn't typeset.

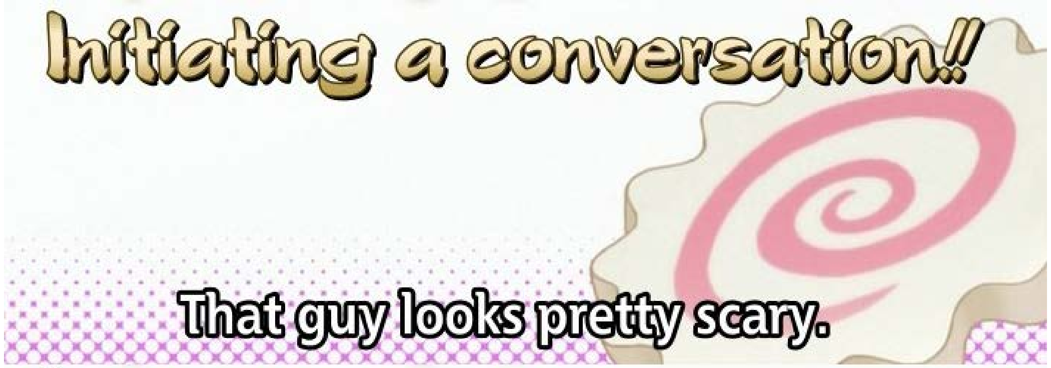
When I separate the layers, it looks like this:



This is one of the most important things to learn about typesetting, so make sure you get this right.
The middle part doesn't work because the blurred edges of the top layer create partial transparency,
so you can partly see the sharp edges of the font body of the bottom layer.
Same thing happens when you use \1a&HFF& on the bottom layer, so DON'T do that either.
The "Layers" section of this guide explains it in detail.

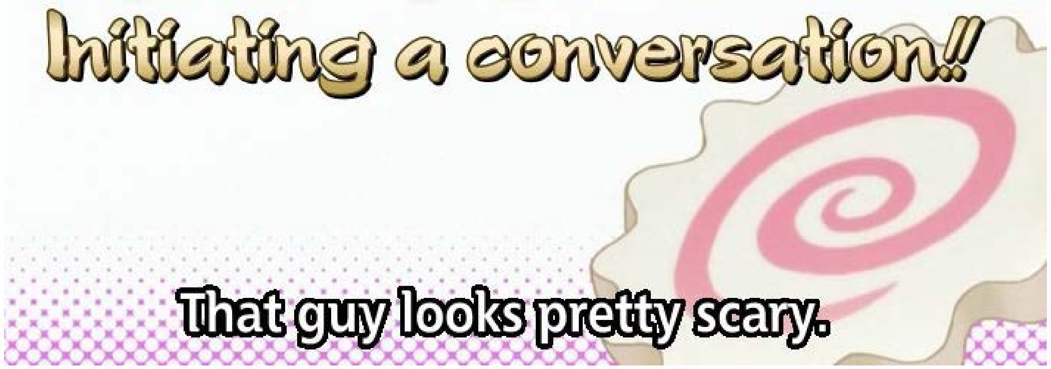
Sort the script by time! (At least if you live in the early 2010's)
[menu -> Subtitle -> Sort All Lines -> Start Time]
If you don't do this, vsfilter will screw up blur pretty much whenever there are two or more lines visible on the screen at the same time.

Top is sorted by time:



Initiating a conversation!!

That guy looks pretty scary.

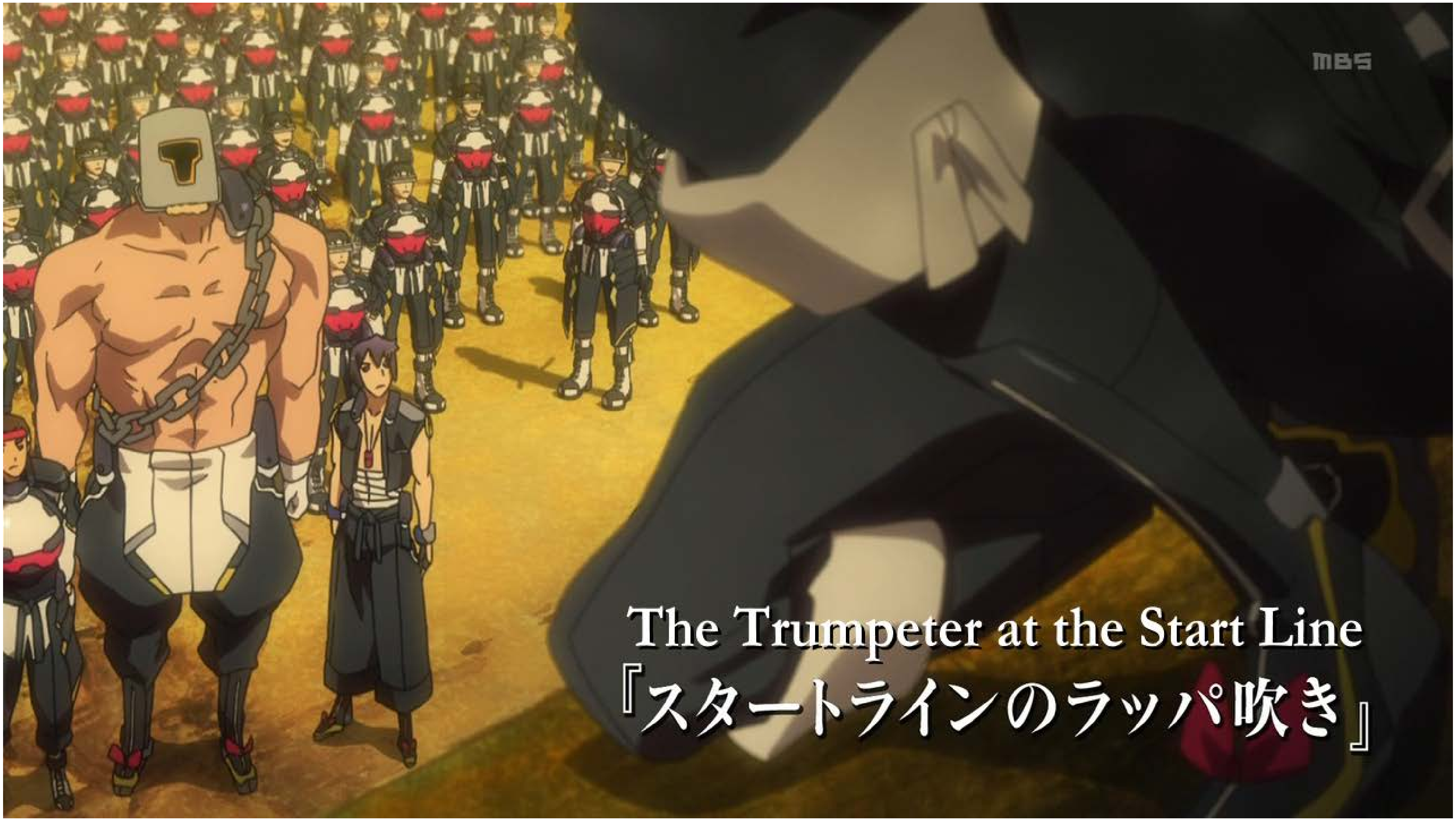


Initiating a conversation!!

That guy looks pretty scary.

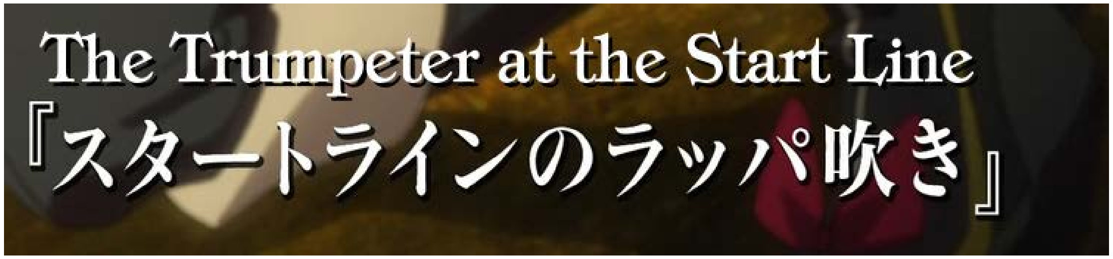
Bottom is not sorted by time. You can see the border on the default style is screwed up.
This and worse things happen when you don't sort the sript by time.
Of course you don't need that when working, and it's more convenient to have different sorting while working,
but always sort the final script you're putting in a release.
2015 Note: This is not an issue anymore.

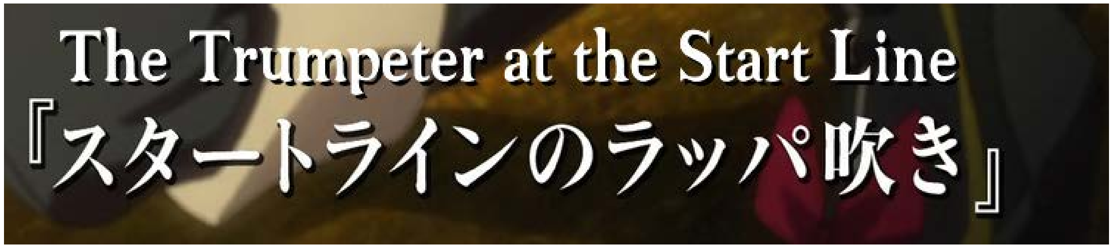
Back to the basics...



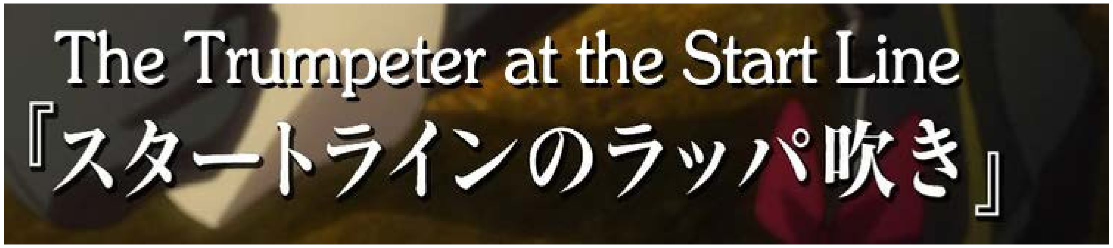
Here's another title. Very simple but beginners will often fail. This one has a shadow but does **not** have a border.
Beginners will often use border because there's something black around there and who would bother to look carefully? Border is first so... bam! Well, nope. If your style has border, use \bord0 to kill it.
Align your typeset properly. It would be dumb if it was clearly closer to one side. Don't put it under the sign here because it might overlap with main dialogue.
Other things to pay attention to: the shadow in this case is not transparent at all; get the shadow distance close enough to the original;
try to match the thickness of the letters; and for god's sake don't use a sans serif font like Arial for this.

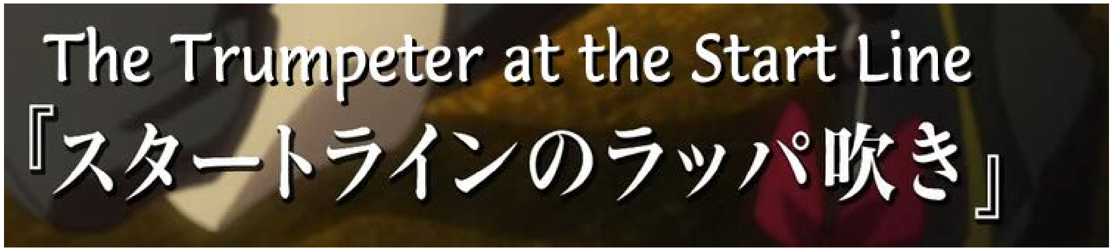
Alternatives that work:





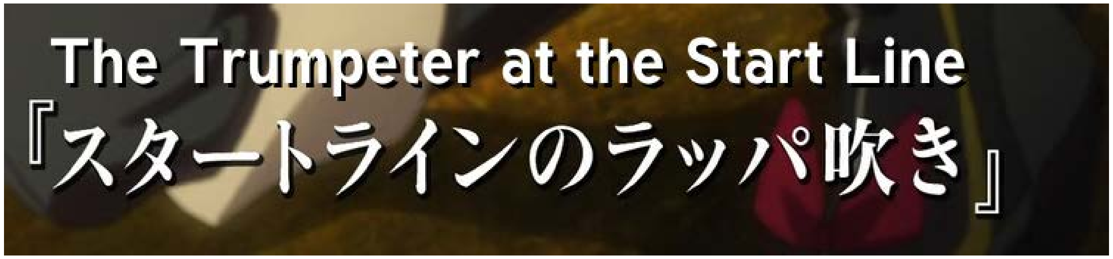
These two are fine. The thickness matches, they have some pointy ends like the original, horizontal lines a bit thinner than the vertical ones... everything all right.



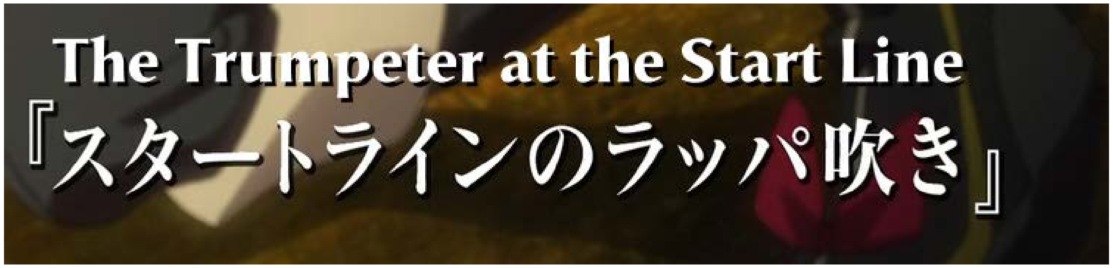


These two are not too bad, but not as good as the previous examples. They're a bit too roundish, lacking any pointy/thin parts.

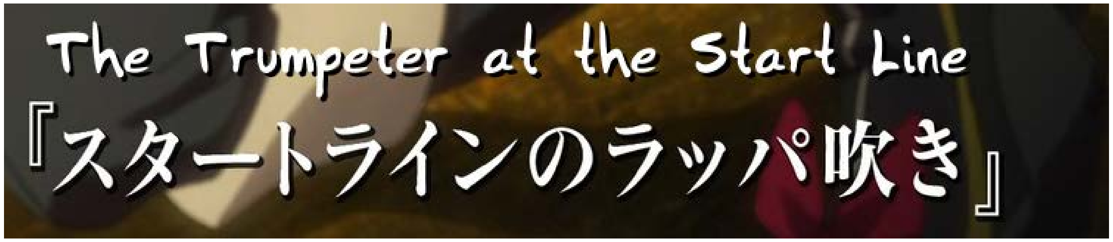
Alternatives that **don't** work:



Sans Serif doesn't fit here. Square ends don't match at all. Looks dull and inelegant.

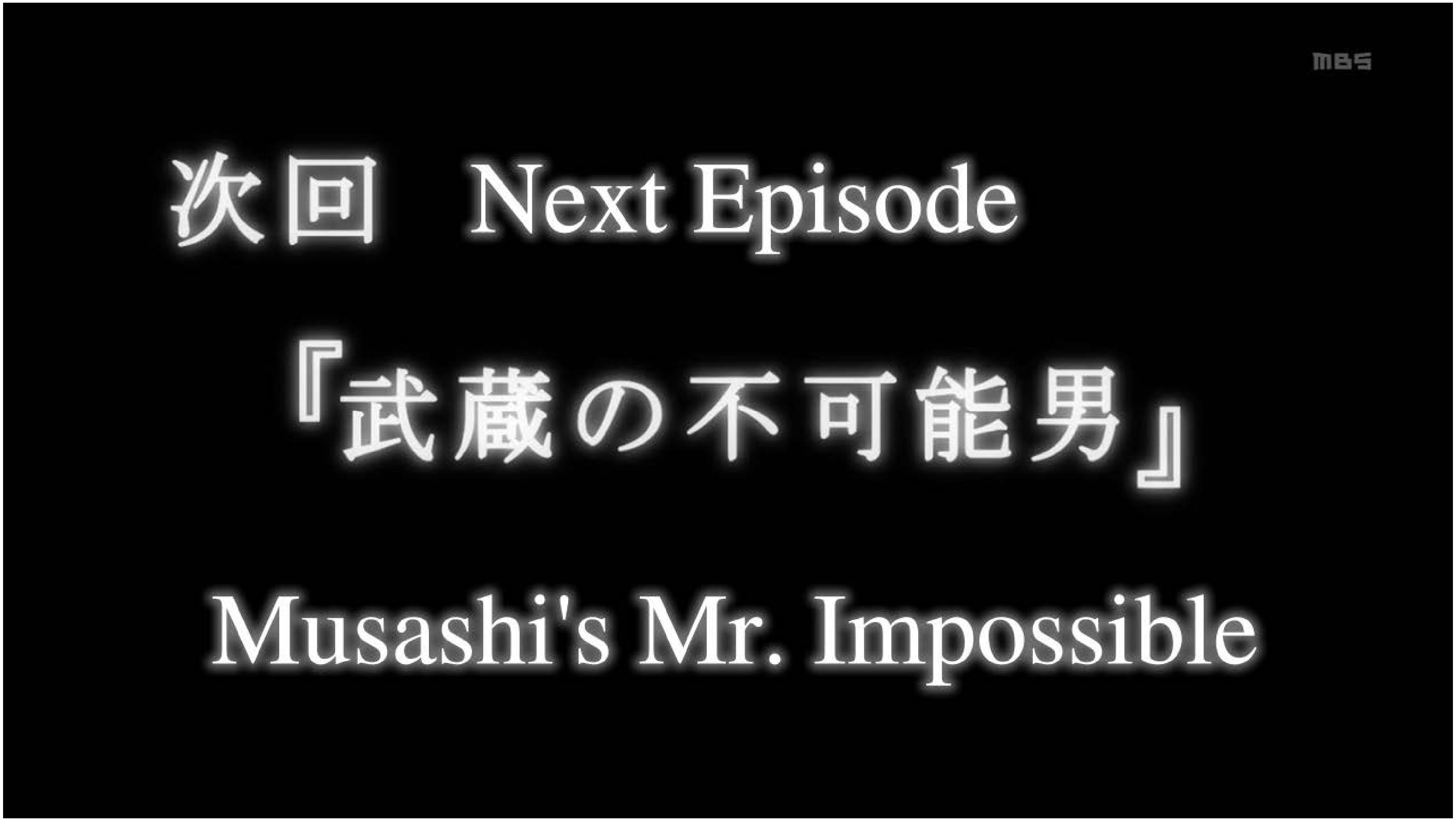


This is too thick/wide.



While handwriting is often useful for typesetting anime, because of the calligraphic nature of kanji, here the kanji is actually pretty simple and orderly. The handwriting looks too disorganized.

(If you use a Japanese font to add those Japanese "quotation marks", you're a fucking moron.)

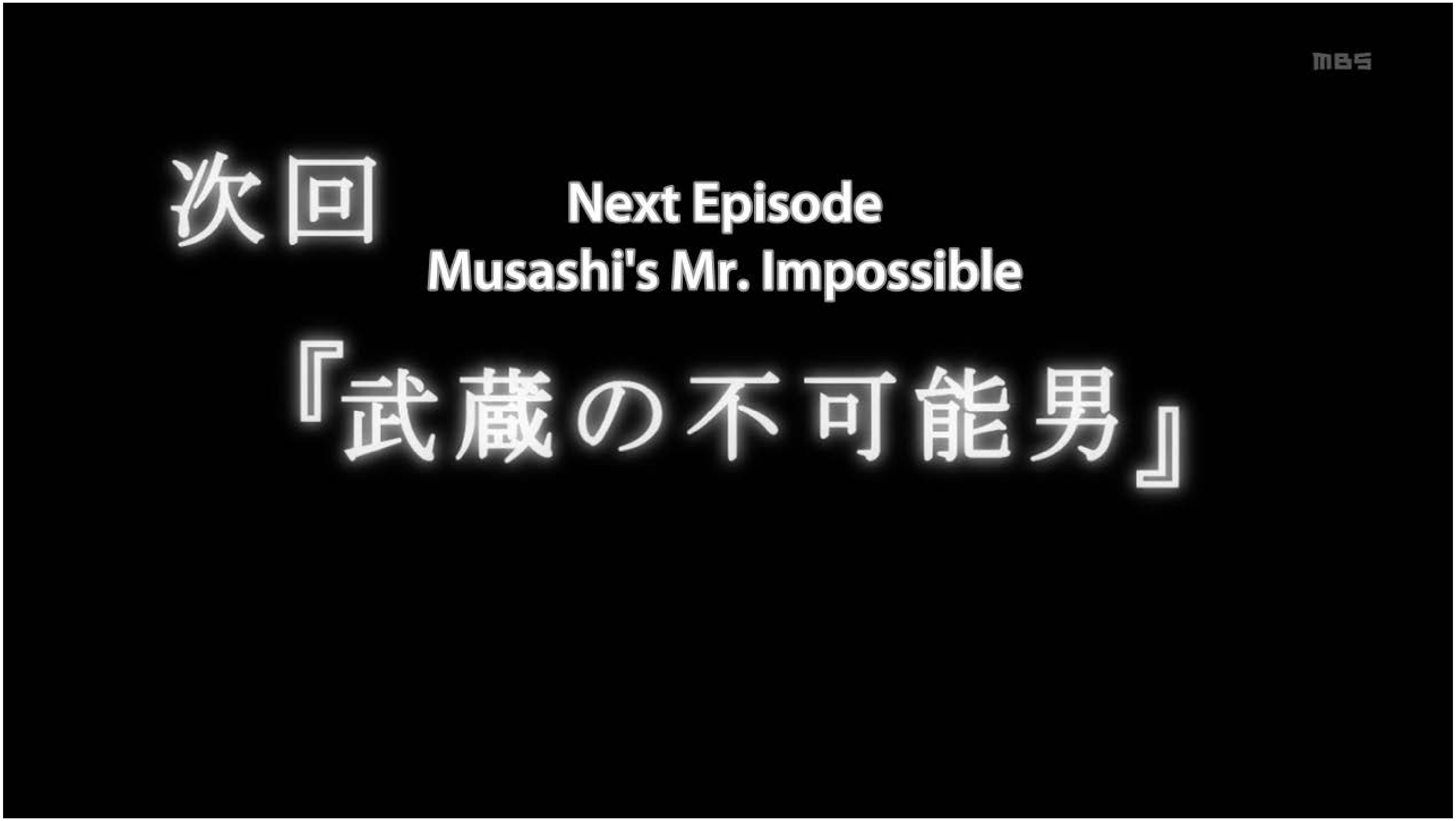


次回　Next Episode

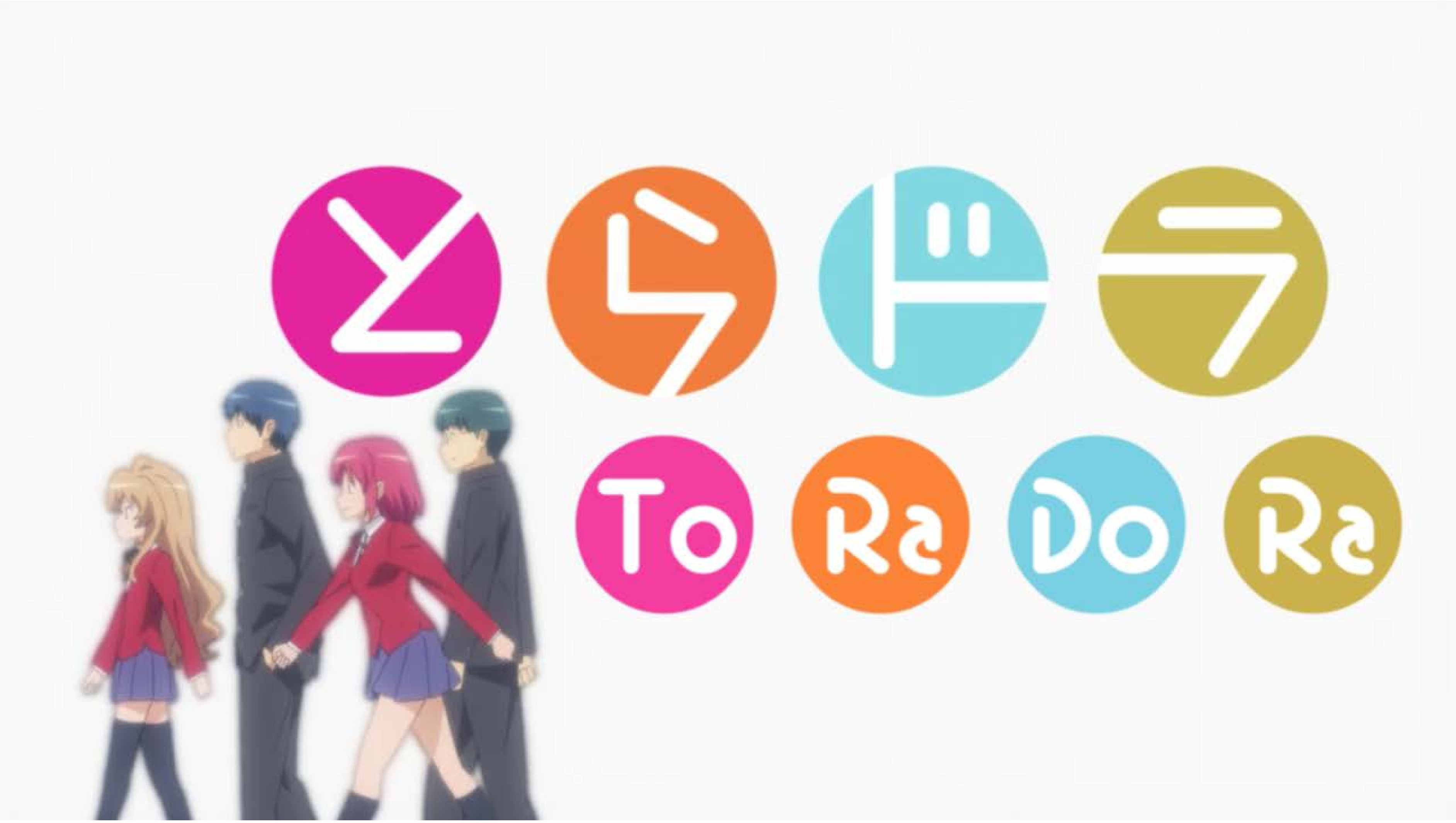
『武蔵の不可能男』

Musashi's Mr. Impossible

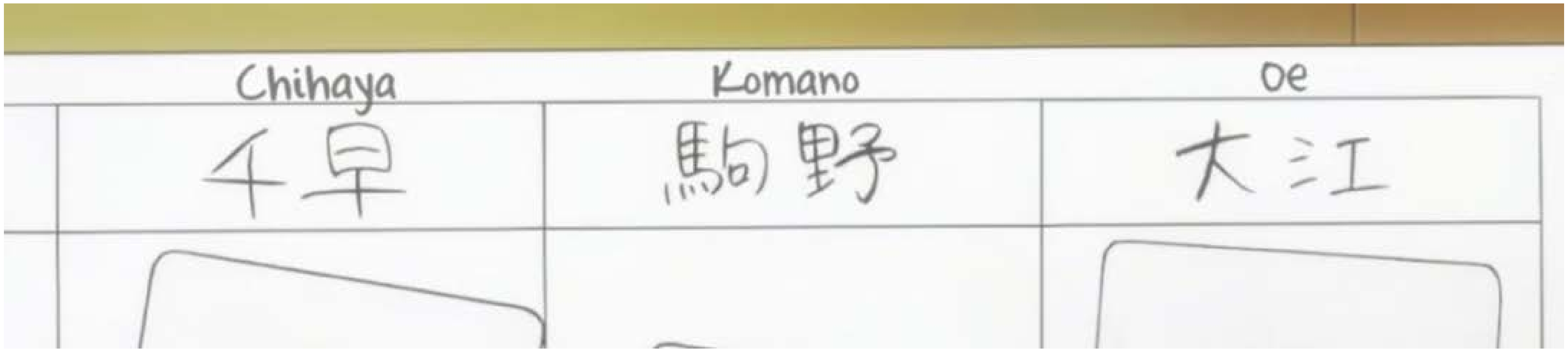
Next episode title. Pretty simple - get the sizes right, choose a reasonable alignment, get the border colour right, and use blur.
2015 Note: The inner part is actually lacking blur. (I sucked in 2012.) See section on Layers.
You can see it's pretty easy to match the original, so I don't want to see things like this:



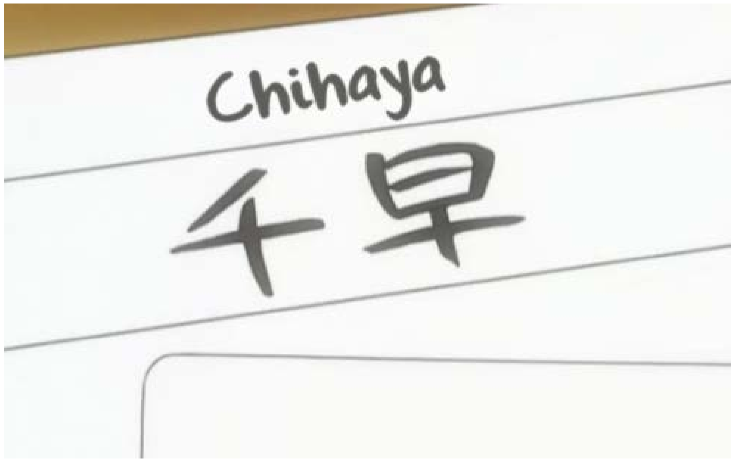
Using thin outline without blur = nope. Using thick sans serif font = nope. Vertically it's not aligned with anything. That's a fail on a sign that takes a minute to do right.



Here's something more interesting. In case it wasn't clear, the smaller circles with To Ra Do Ra are typeset.
So what you need is letters **and** circles. The easiest way to make circles is to use a font with symbols, like wingdings. Find out which letter is a circle and use that.
2015 Note: Please no. Use vector drawings. **Masquerade** makes circles and other shapes really easy to use.
Then find a font that has round edges and isn't too thick. Mine was actually too thin but I solved it by adding some outline in the same colour as the primary - white.
It was also narrow so I used something like \fscx120. All of this can be set in the style, so no tags needed.
To get the letters exactly in the middle of the circles, I used \an5 - align to centre. That way you can use the same \pos coordinates for both the circle and letters and you know it's right in the centre.
Now you just need to find the right place to put the circles. Make sure the vertical coordinate is the same for all of them, and that the spaces between them are always the same.
The only thing left is to get the right colour for each circle. Tools for colours are above the typing area. Use the eyedropper tool to get the exact ones you need.
Speaking of which - always match the colours exactly, not just approximately.



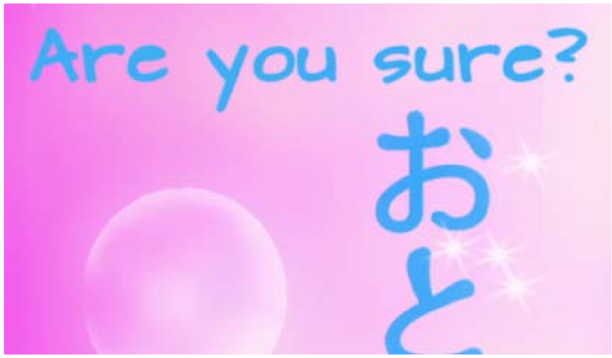
Simple typesets for some names. Handwriting font, match the colour, no border, no shadow, use blur. Easy.



This close up is different than what they had in the first screenshot.
It's thicker and darker so you can use bold, or outline in the same colour... however...

If you're gonna use some outline to make the font look thicker,
make sure you can actually afford it without making the font look unreadable.

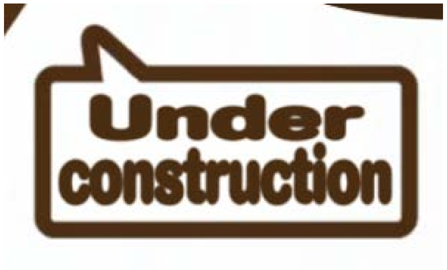
This is already pushing it, though still not too bad:



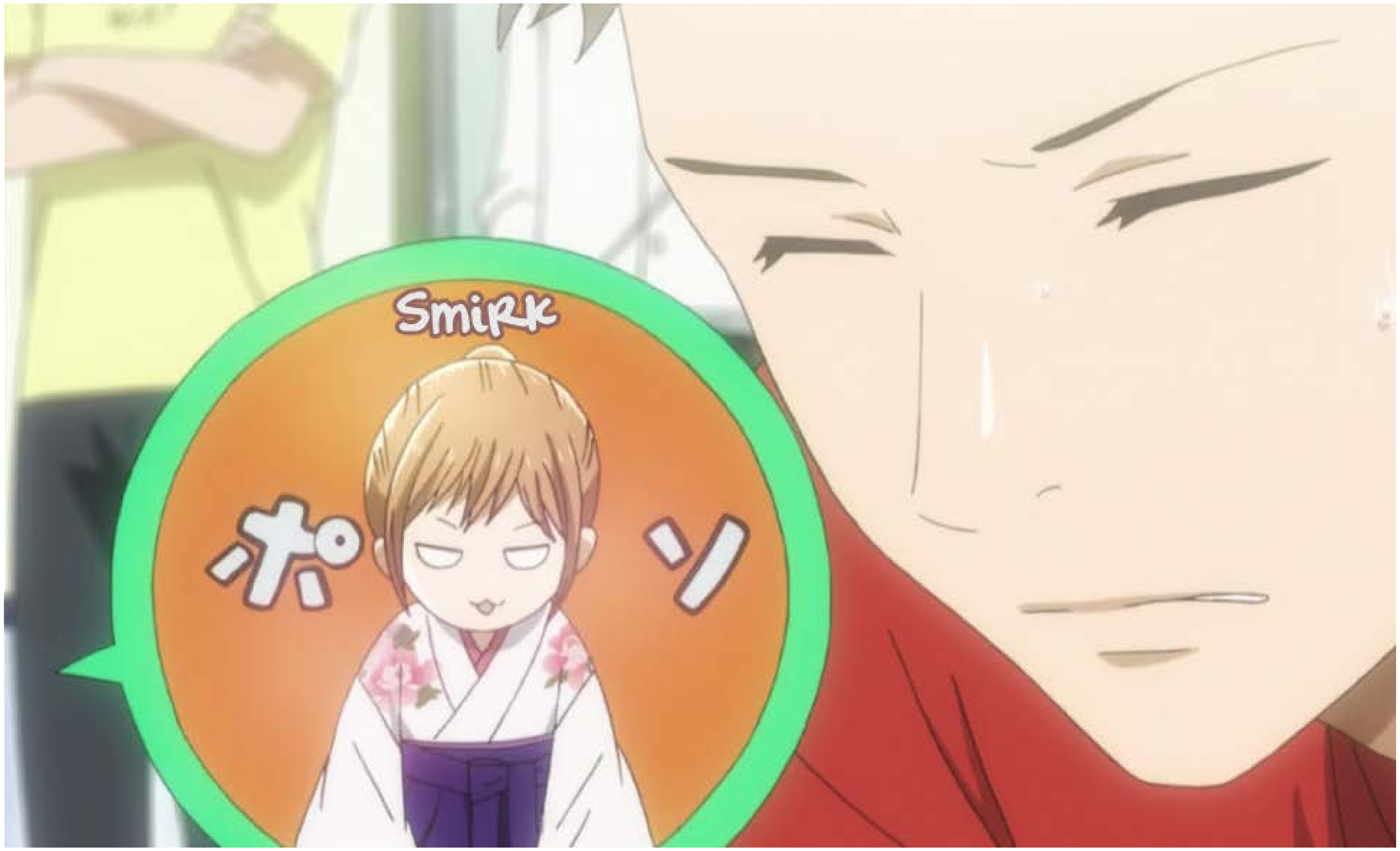
Without the border for comparison:

Are you sure?
お

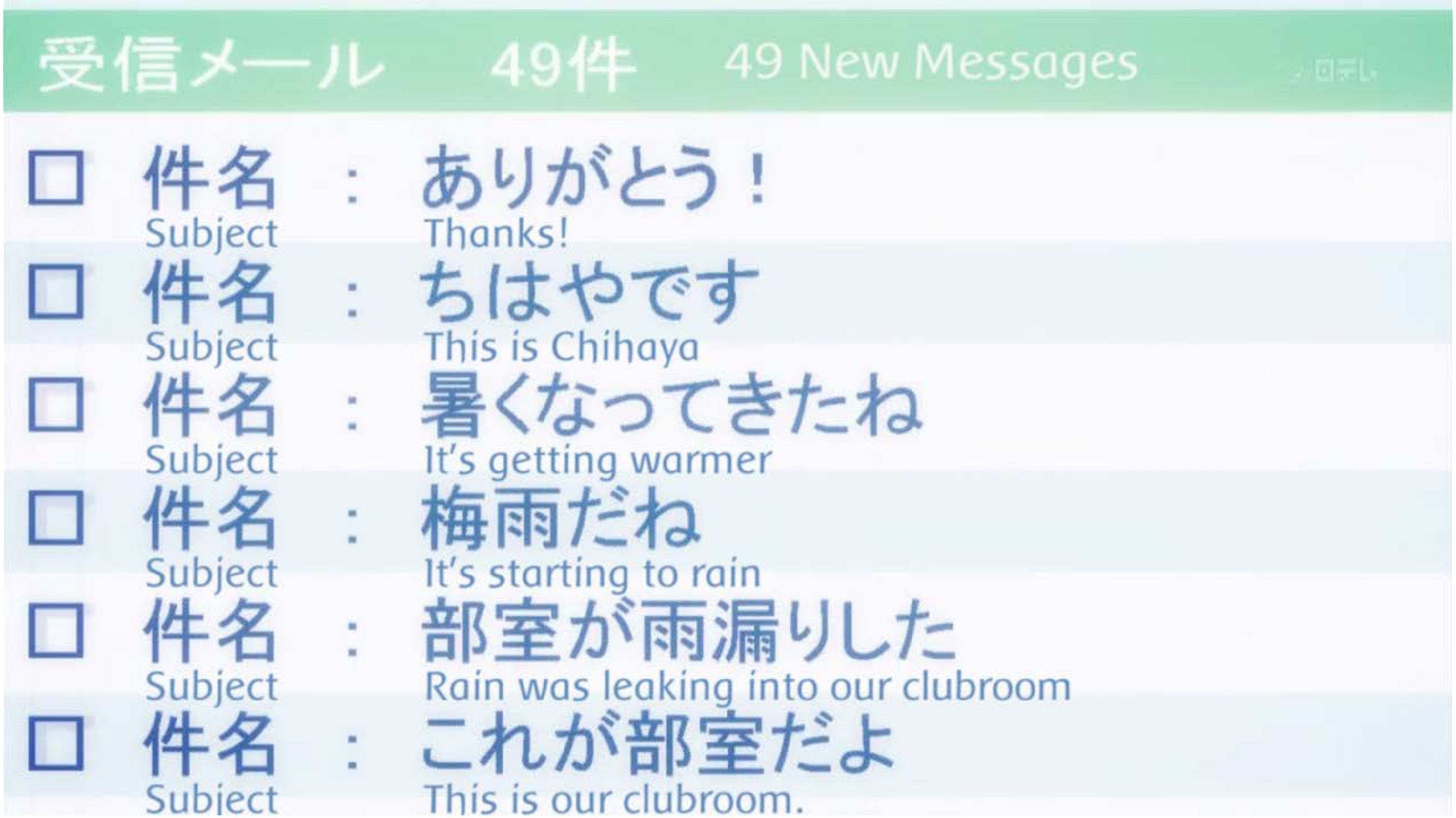
This is pretty bad:



Letters like 'e' or 's' become hard to read, especially if you stretch the font in one direction.
Please avoid stretching fonts more than about 10% in one direction unless you have an extremely good reason.
In this case the letters even merge with one another, so try to find a better font instead.



White font, thick dark red border, no reason to fail on this. Clearly here you need some handwriting/cartoonish font, and not some Arial/Times New Roman thing.
[Actually this fails with blur, but hey, it was a long time ago.]



受信メール　49件　49 New Messages

□ 件名 ： ありがとう！
Subject　Thanks!

□ 件名 ： ちはやです
Subject　This is Chihaya

□ 件名 ： 暑くなってきたね
Subject　It’s getting warmer

□ 件名 ： 梅雨だね
Subject　It’s starting to rain

□ 件名 ： 部室が雨漏りした
Subject　Rain was leaking into our clubroom

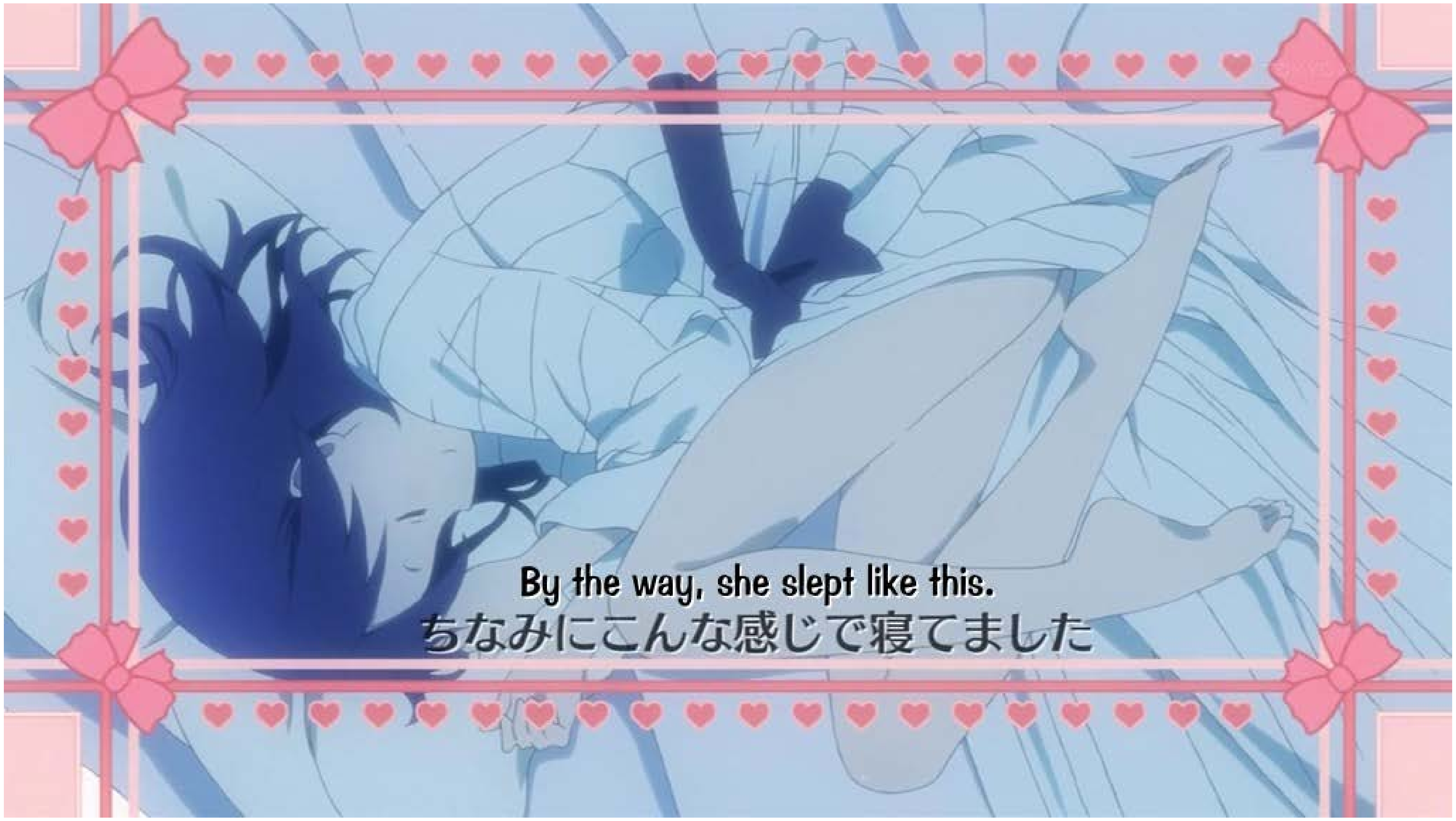
□ 件名 ： これが部室だよ
Subject　This is our clubroom.

Sometimes you have a bit more to typeset than one line.
Here you need a simple sans serif font. I used this one not because it was the best but because I was already using it in the episode and it was good enough.
2015 Note: It should be at least bold/thicker, and the colour is wrong. It should also have a "glow", but back then that would have lagged.
Aside from the "49 New Messages" in white, this is all done in one line.

```
Dialogue: 0,0:08:08.63,0:08:08.67,mail,Caption,0000,0000,0000,,{\blur0.8\c&HBD8B5F&\pos(126,186)}Subject            Thanks!\N\N\N\NSubject            This is Chihaya\N\N\N\NSubject            It's getting warmer\N\N\N\NSubject            It's starting to rain\N\N\N\NSubject            Rain was leaking into our clubroom \N\N\N\NSubject            This is our clubroom. \N\N\N\NSubject            I got in trouble with Dr. Harada \N\N\N\NSubject            How do I cut down on faults? \N\N\N\NSubject            This is Chihaya \N\N\N\NSubject            Karuta players are... \N\N\N\NSubject            About hakama \N\N\N\NSubject            Guess what happened today \N\N\N\NSubject            Notice for the \N\N\N\NTokyo regional tournament
```

You can see there are 4 line breaks between the text lines (\N\N\N\N) so that I don't have to make 6+ separate script lines to typeset.
Choose font size that will make the lines fit in between the Japanese lines. When you have the font size, make spaces between the Subject and the rest of each line.
You could typeset each line separately, but... the whole thing was scrolling up in a non-linear fashion. That also means you can't use \move.
So I did this frame by frame, always changing just the \pos tag (you may notice the whole line has more text than you see on the screen - this text scrolls up in the following frames).
It was about 20 frames so I had 20 lines in the script. If you typeset each line of text separately, you'd have more than 10 times as many lines in the script.
A 20-frame sign is usually pretty pointless to typeset, but the way I did this wasn't really difficult and didn't take much time so I did it anyway.

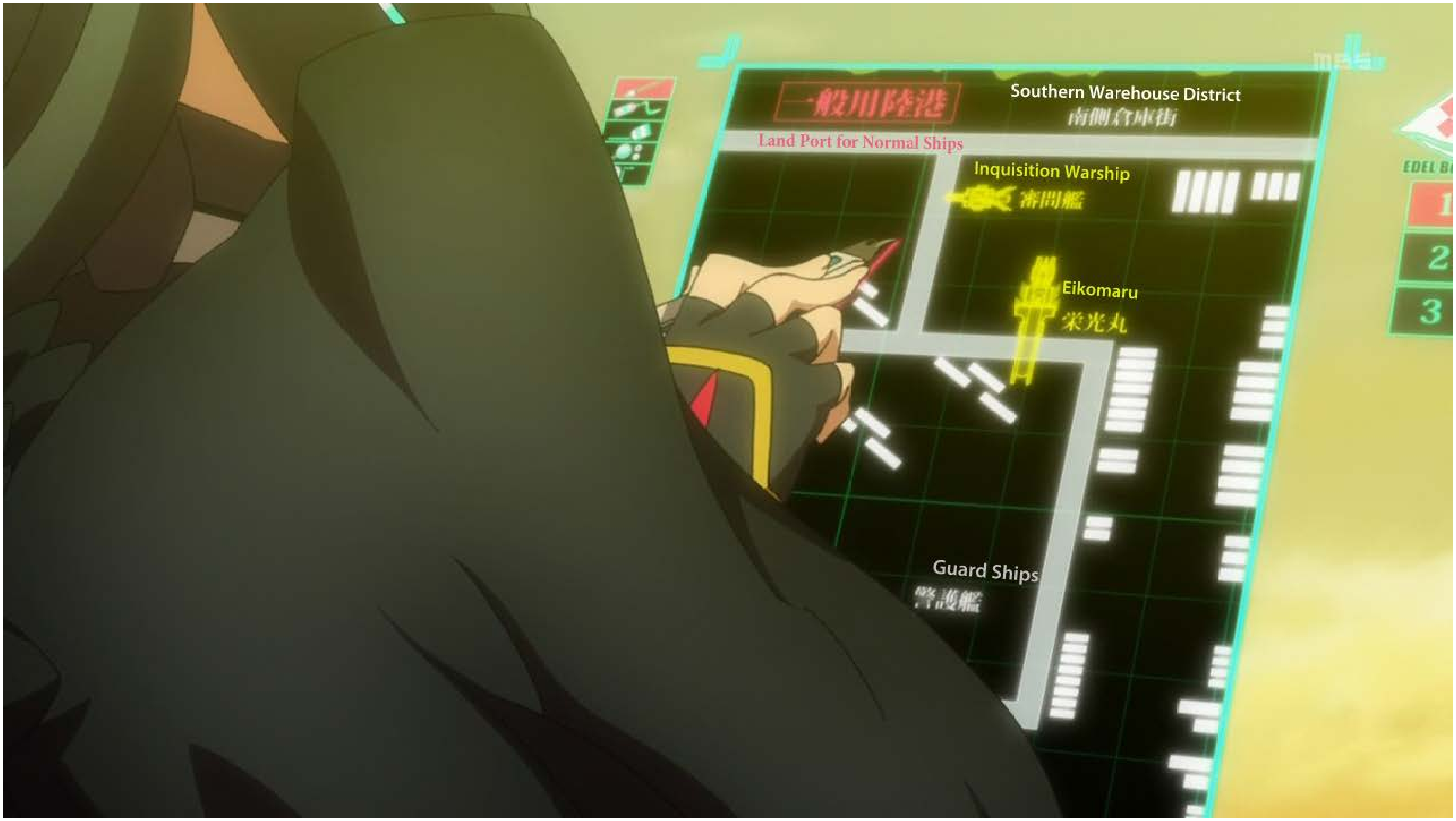
[Note: The colour should be darker, and the font should be thicker.]



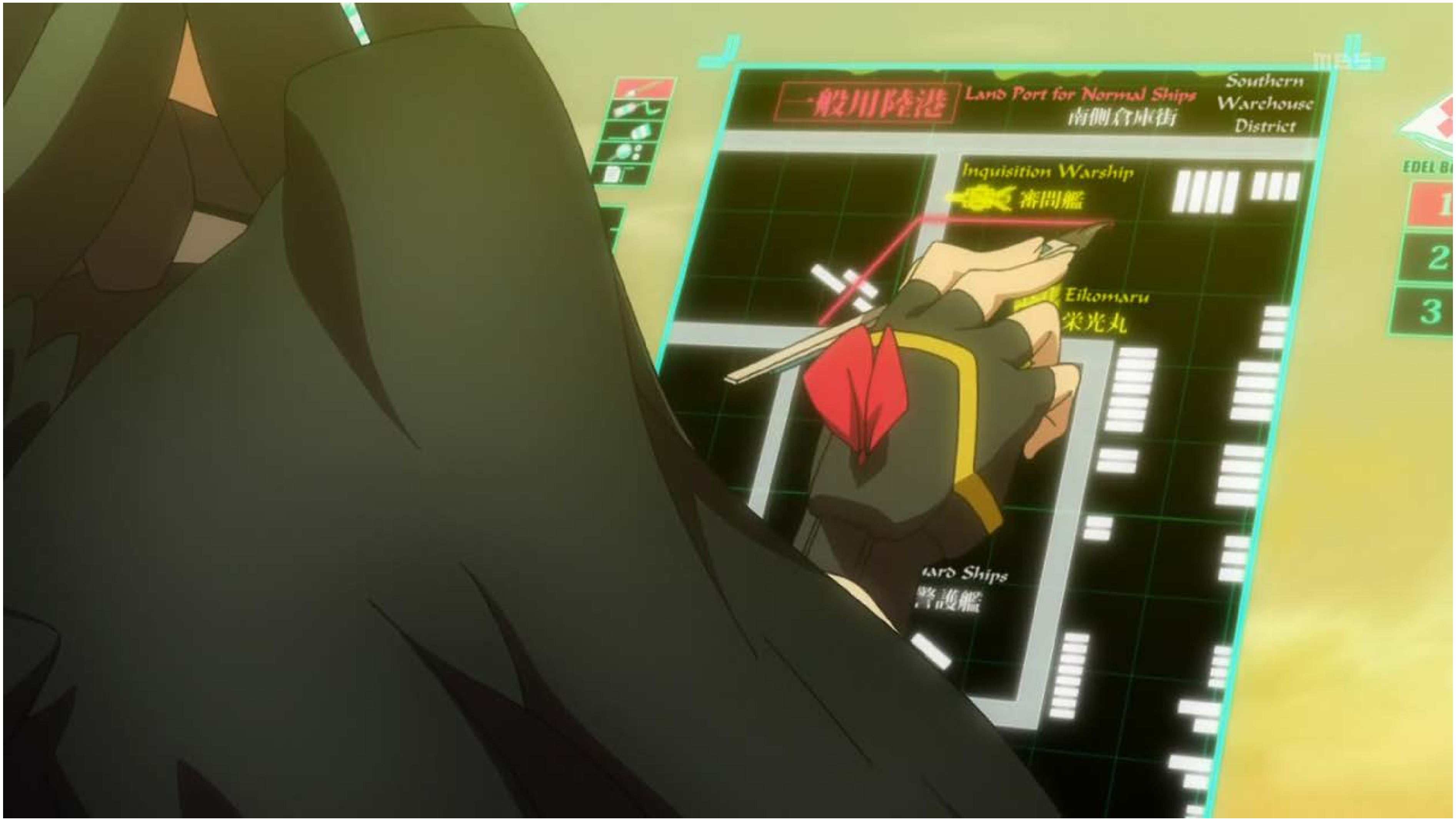
By the way, she slept like this.

ちなみにこんな感じで寝てました

This was not bad a few years ago but is pretty bad now. If you can't match the colours precisely, you suck.
The Japanese is not black and white, so use the eyedropper tool to get it right.
You can easily make this so natural that it won't even look like it was typeset. You could also use \fscx110 or so for a better match. And it's missing blur.

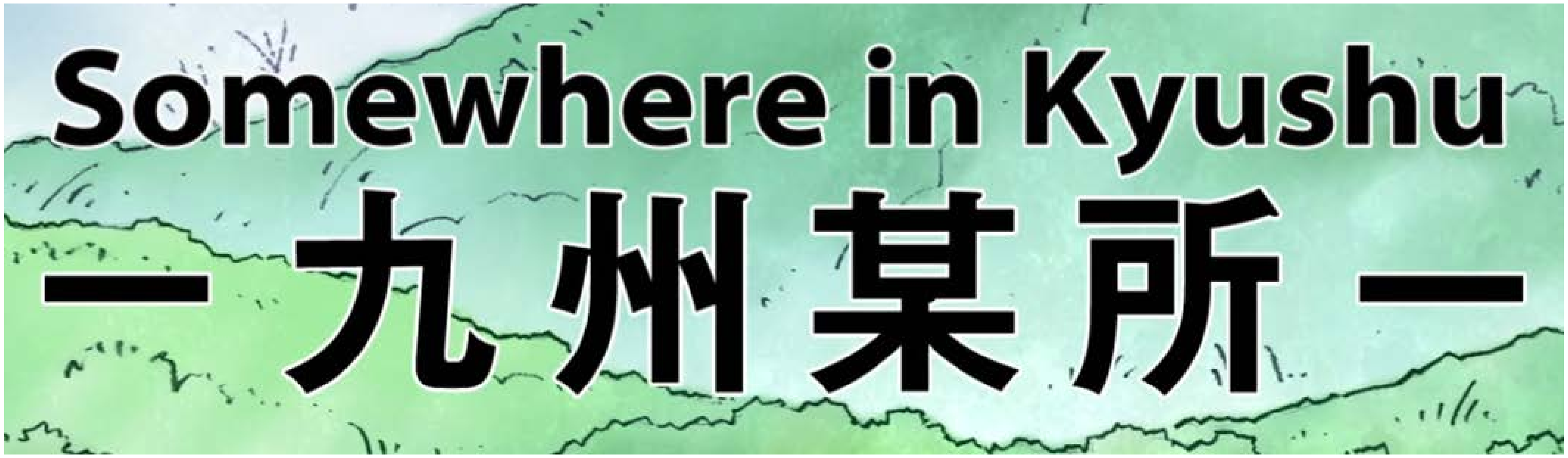


So I gave somebody the task of typesetting this... and this was his first attempt.
Positioning is ok. 1 point there. Colours are fine as well. Another point. Alignment of the text is... well, pretty default. More on that in the next chapter.
I don't know why the red sign is serif and the rest is sans serif when the JP signs are all the same font. Also the red looks like crap on the light grey background.
All that would be passable for a beginner if it wasn't for one obvious problem - no blur. Just adding blur would make it look **much** better even with the other problems.



Here for reference is my own typesetting. You can see the blur makes it blend in beautifully, though the slant helps a lot as well, and the font is much better than the Arial-ish thing above.
2015 Note: It needs "glow".
As a sidenote, see the hand moving "over" the Guard Ships sign? That can be done with the \clip tag. More on that later.

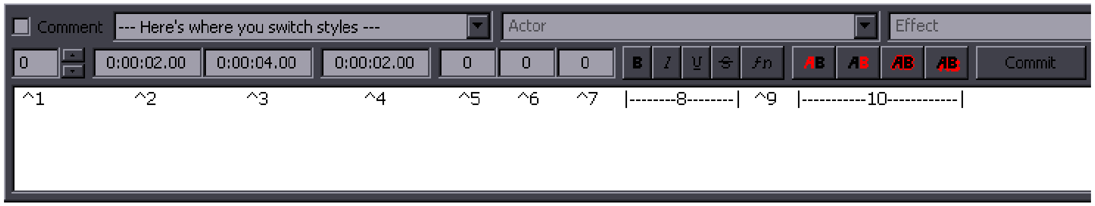
A simple typeset:



1. Matching font with roughly matching thickness of letters.
[as the English is usually longer, you may need a lot more letters than the jp, so you can't always match the thickness.]
2. Matching colours.
3. Matching border size.
4. Two layers for blur.

That should cover the basics. Just a few more notes. Sometimes instead of blur you can use \be - blur edges. With value 1 they're pretty much the same but with higher values you'll see the difference.
Other tags you can use to override the style are \fscx, \fscy, \fsp... again, you should know all these from the link mentioned at the top.
I didn't explain \pos because it's so basic that if you can't figure it out on your own, you're hopeless.
\an can be useful for signs with a line break - \N. Type something short, then \N, then something long, like "This is \N a meaningless test sentence."
Use \pos to place it somewhere on the screen. Then add \an9 or \an1 to see how the text changes alignment while using \pos.

One last note about changing margins. Let's use this screenshot:



Numbers 5, 6 and 7 are left/right/vertical margin.
Change those numbers to change the margin. The values don't add up, they override the defaults.
It's only meaningful when you're NOT using the \pos tag, mostly for default dialogue.
You can use this if you need to move the subs to avoid overlapping with something else.
For example changing right margin to 500 will move them to the left, changing vertical to 100 will move them up etc.

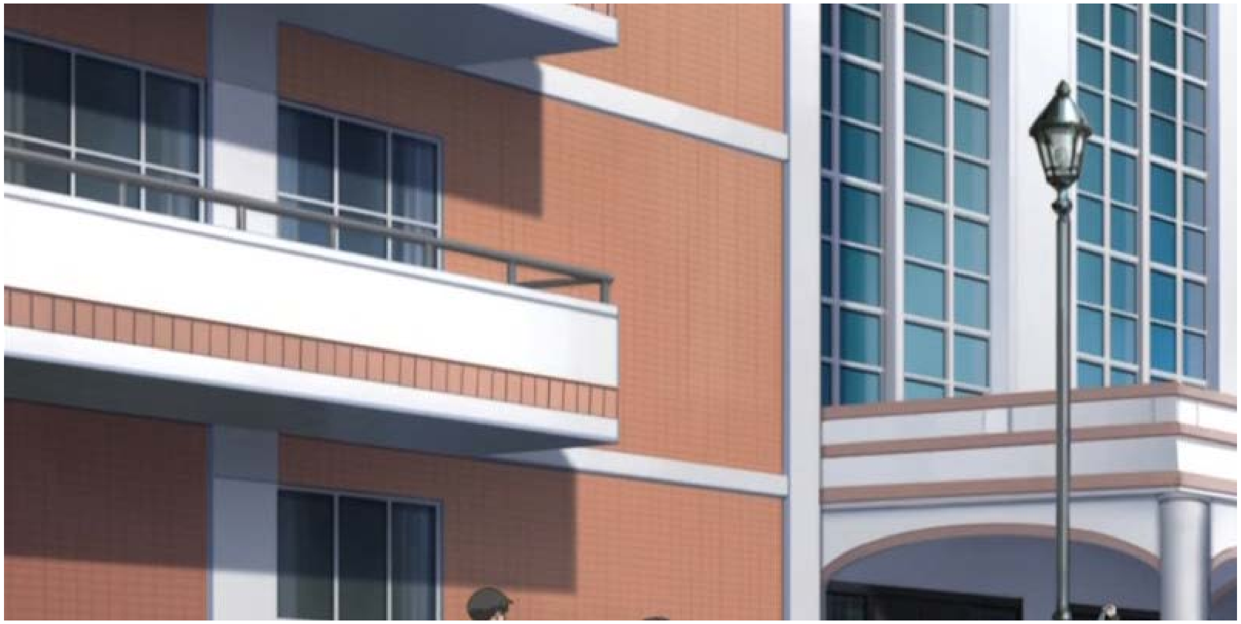
« Back to Typesetting Main

Typesetting: Aligning Signs [this silly text is here only to align the title with the pictures below...]

Aligning signs is actually pretty easy, but for some reason many typesetters fail in this area all the time.
I'm not sure if they don't know how to do this or they're just too lazy to do it right, but it's pretty lame because just aligning the sign correctly will make it look a lot better.

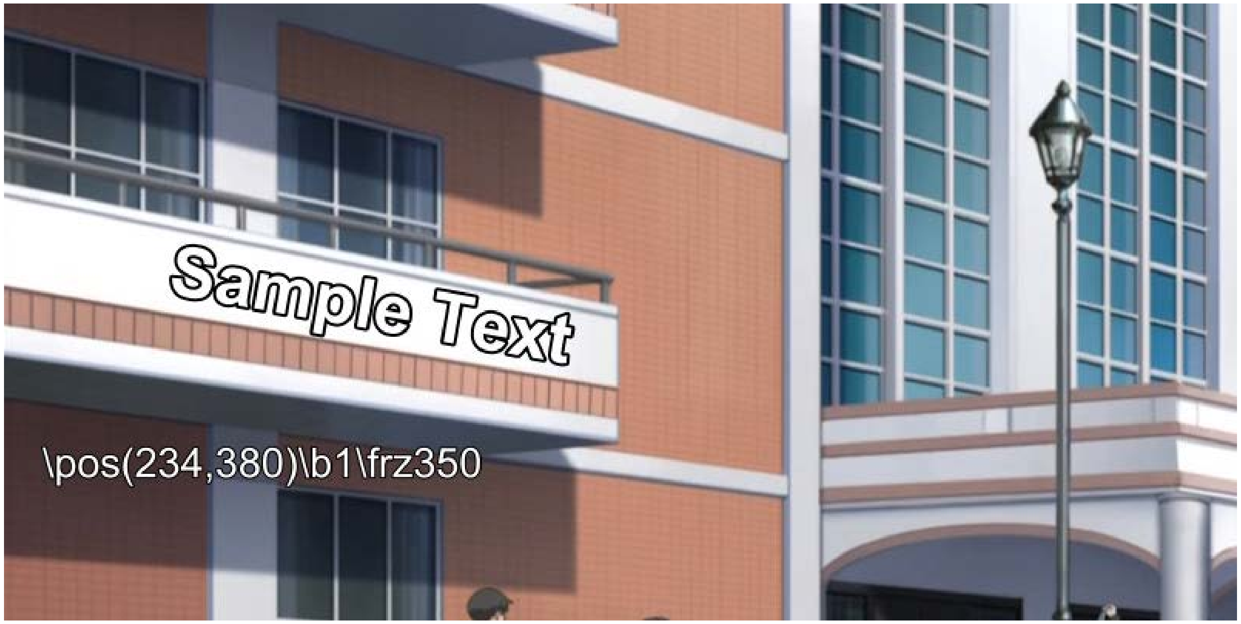
The tags you use for this are the three rotations - \frz, \frx, \fry, shearing - \fax, \fay,
and to get the \frx and \fry right, the origin point of the rotations - \org.

Let's try a basic example. Let's say you want to put a sign on this balcony:



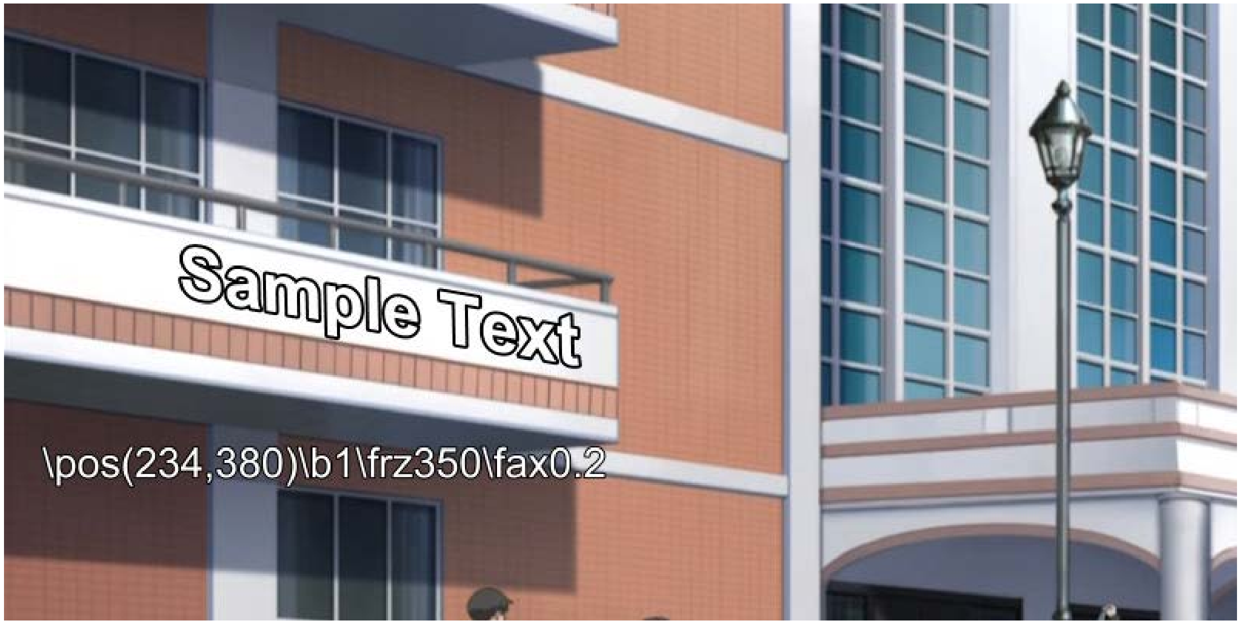
This is pretty typical for anime. You will need to make lots of signs of this type, whether it's the notorious nurse's room at school, a sign above or on the door of an office, or various signs on buildings etc.
These signs usually have a horizontal slant, but vertically they're pretty much straight.

This is important to acknowledge if you don't want the sign to look like shit...



```
\pos(234,380)\b1\frz350
```

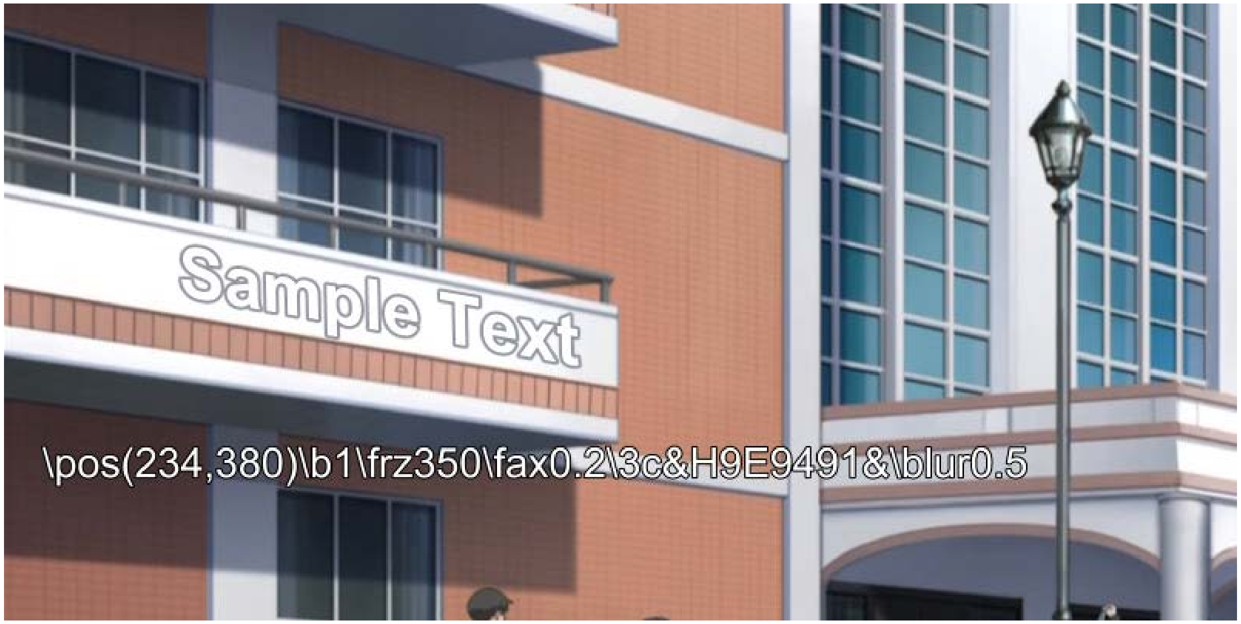
This could be the first thing to do - use \frz to rotate the sign.
Unfortunately that's where some typesetters end. If you can call them typesetters.
It's the same people who will also pick an incredibly shitty font,
often some M$ or Adobe nonsense that looks like crap but is 10 MB large.
Anyway, we'll get to fonts later...
So if you actually want to do some real typesetting, here's the next step...



```
\pos(234,380)\b1\frz350\fax0.2
```

The tag you use here is \fax. When used with a negative value, like \fax-0.1, the effect is like using italics.
With positive values it leans in the other direction.
You have to use low values, usually in the range of 0.05-0.5.
It seems that many typesetters don't even know this tag, or are too lazy to use it
because unlike the rotations, this one doesn't have a tool in aegisub and has to be typed out. (lol typing tags)
It is, however, more useful than \frx\fry, because anime doesn't have much of 3D effects.
Most of the time it's really just horizontal slant while vertically it stays the same.

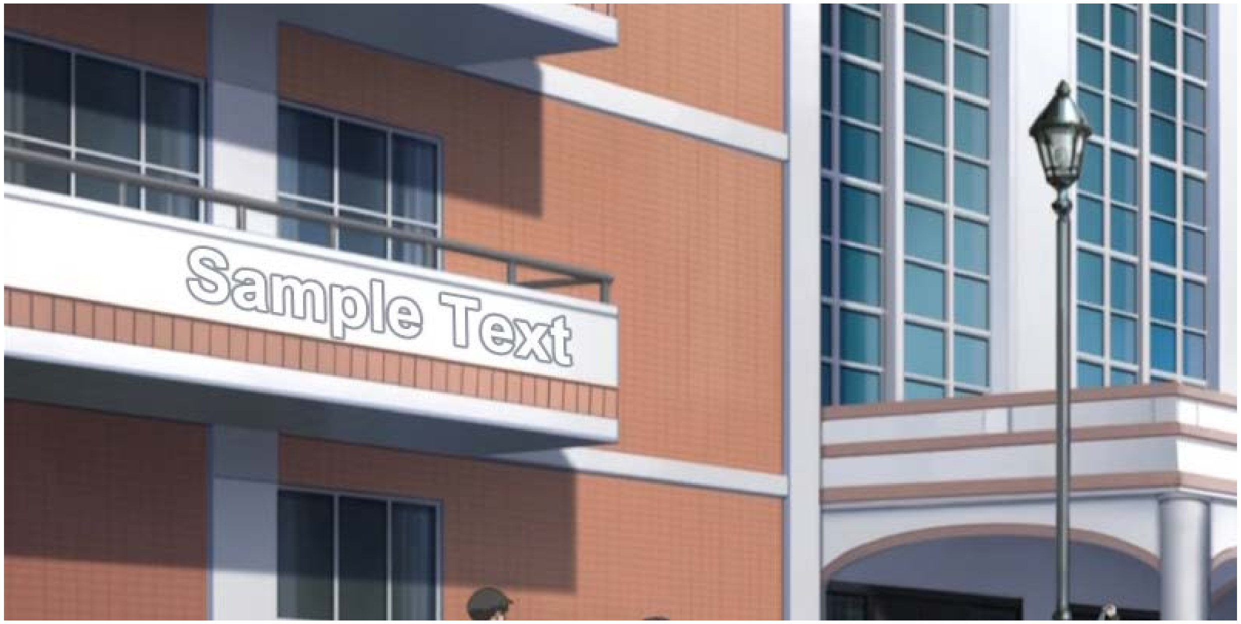
So now you have a sign that's fairly well aligned. Now you just need it to blend in a bit better...



```
\pos(234,380)\b1\frz350\fax0.2\3c&H9E9491&\blur0.5
```

...which you achieve by changing the colour to match whatever is relevant in the picture.
Here I don't have an original sign to imitate, so I just go with the outline of the balcony.
You will also notice that i used blur.
On signs, using at least 0.5 blur is a must if you don't want them to look obviously added and out of place.

So now you have a pretty decent sign. You might as well go with it.
If you look more carefully though, you'll see that it's aligned to the bottom of the balcony but not the top. Why is that?
Well, because here you actually have a sort of 3D effect, where the left end of the balcony is taller than the right.
So what now? Since we've gone this far, we're not gonna redo it with rotations, so we'll use a simple trick...

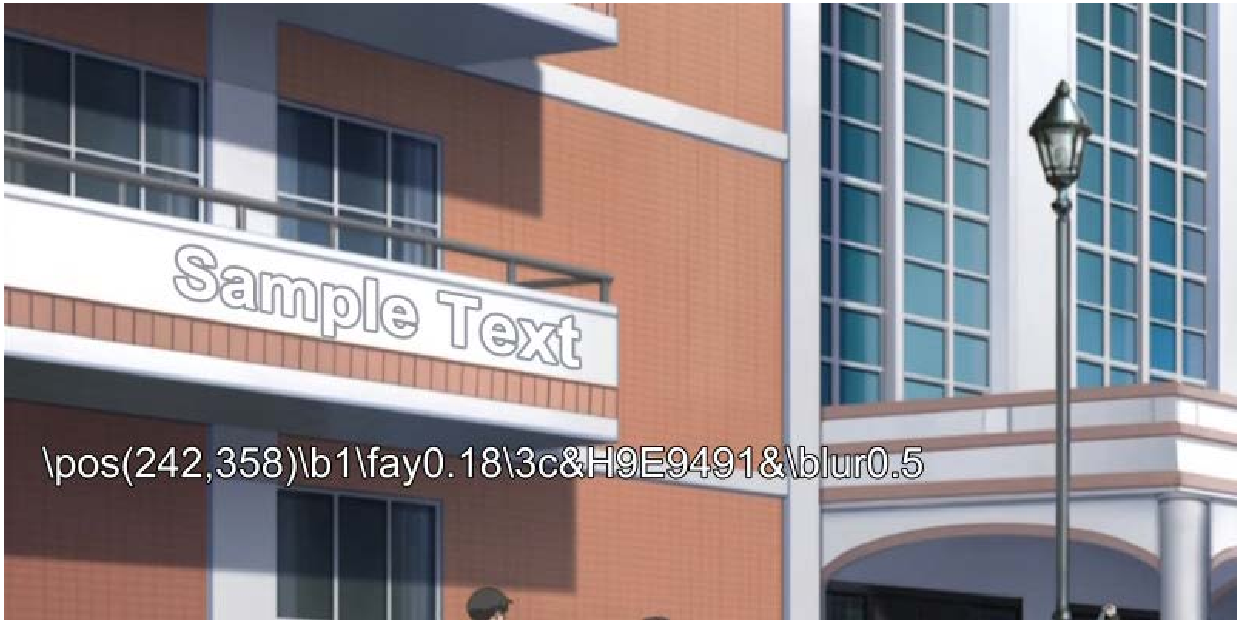


Looks better, doesn't it? (By the way if you *don't* see the difference here, you should probably not be a typesetter.)

So what sorcery is this? Very simple. The original font size was 50.
If you want the end of the sign to look smaller than the beginning, instead of all kinds of rotating and skewing you can do this...
Sa{\fs49}mp{\fs48}le {\fs47}Te{\fs46}xt (Did you know HYDRA can do gradient by 2 characters? No typing required.)
As you can see, after each 2 letters I decreased the font size by 1.

If you're learning to typeset, this should be good enough.
If you're pro, you'll notice the end of the sign is still a bit too tall, the 'p' is too close to the bottom etc.
And obviously you'll want something better than Arial. Then again if you were a pro, you wouldn't be reading this guide.

So we have what we wanted, but let's look at other ways of doing this.

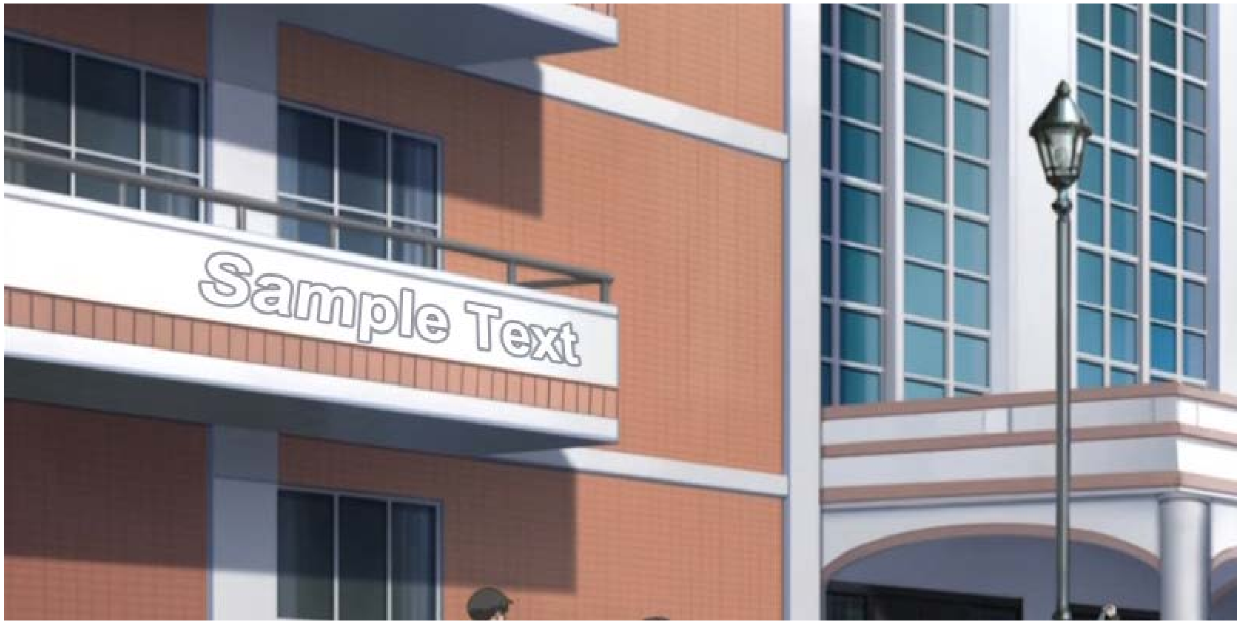


```
\pos(242,358)\b1\fay0.18\3c&H9E9491&\blur0.5
```

This is without using \frz and \fax, but using \fay instead.
Often this is more convenient because you don't have to use any rotation. It'll work fine for signs on/above doors much of the time.
The problem with \fay is that you can't use additional tags in the text like I did for the font size before, because of a vsfilter bug.

The last way to do this that I'm gonna mention, possibly the most pro if you can do it right [but epic fail if you can't],
is using just the rotations and moving the origin point.
This means we're gonna use \frx and \fry, instead of \fax and \fay and do it right.
Why would I do that when I've described how \fax\fay is a lot more convenient?
Because the rotations actually **can** give you a 3D effect when you need it,
and we've noticed that the balcony here is "closer" on the left.
We've managed to bypass it pretty well with the font size, but that may not always work. Like when the sign is only 2-3 BIG letters.

First let's look at what will happen if you use rotations without moving the origin point.



You'll see many "typesetters" create these abominations. It's possibly even worse than our first example in black colour way above.
What you have here is a sign where maybe 2 out of 10 letters are aligned somewhat correctly.
The rest is FUBAR. This is like "I has Aegisub so I can into typesetting yeah?"
Nope. You can't.

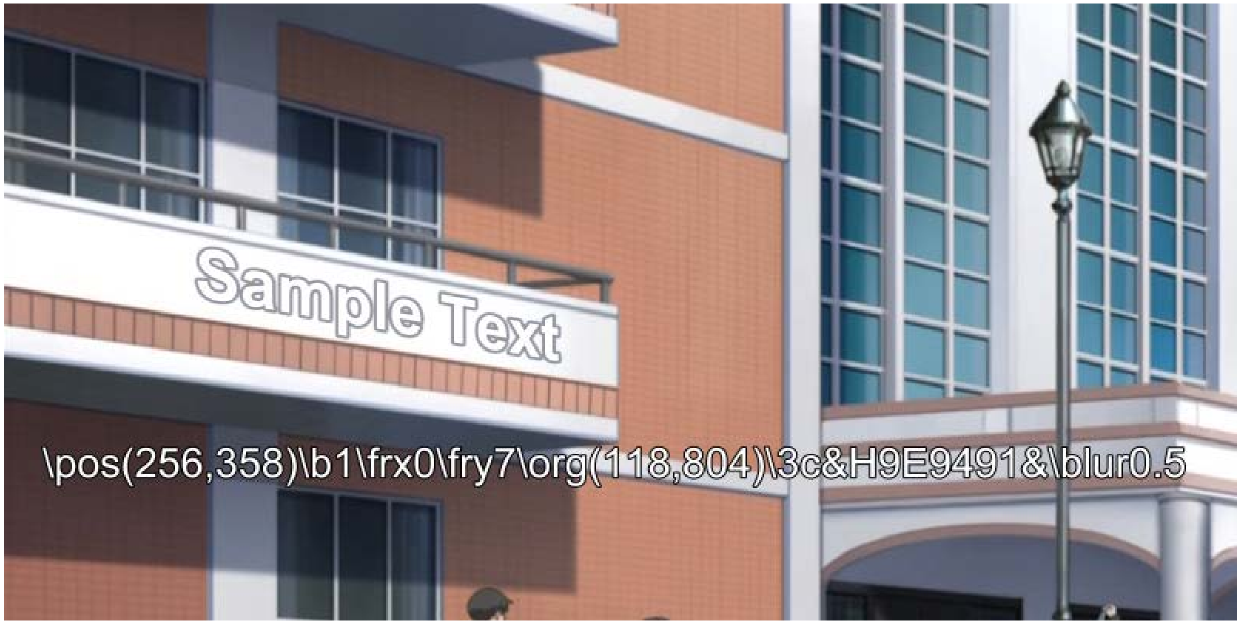
So what now?

If you try playing with those rotations, you may find out that no matter how much you rotate it around, it just doesn't fit.
It will always be aligned at one end and not the other. Why?
Because if your default alignment is \an2 or \an8, it will always align to a vertical line that goes through the centre of the sign.
That means both sides of the text will lean towards the centre, like you see above.
Now if you use \an1, 3, 4, 6, 7 or 9, you may get slightly better results, but still pretty derp.

So what you need is to move the origin point. The tag is \org(x,y), but you can use the rotation tool.
You'll use the tool for \frx\fry and rotate slightly to the side - like \fry7.
Usually you want to use only a little of these rotations and get the rest done with \org.
If you use too much, you'll get too much difference between one side and the other.

When you adjust \fry, you grab the triangle in the centre of the grid and move it.
In the tags, you'll see \org appear and you'll notice the sign changes alignment as you move it.
Then you just have to go and find where to drag it to make it align right. It may often be off the screen.
Once you go off the screen and let go, you won't be able to grab the triangle again but you can adjust the numbers in the tag.
But since that's inconvenient, try to learn to get it right the first time by dragging.

Now, you'll notice that while this will let you get the alignment right, it may drag the whole sign away from its position.
This is not a problem, you'll get it back, just get it to align right first.
When it seems fairly good, you switch to the Drag tool. You will now see 2 points you can drag - the regular square and the triangle.
You won't see the triangle if it's off the screen, but the red line will tell you roughly where it is.
So now you drag the the sign back to position with the square. If it misaligns the sign, you need to drag the triangle a bit again.
It may take a while to balance them out, especially if the triangle is off screen & you have to change the numbers in the tag or try again.
With a bit of practice though, you'll learn to do it more easily. More importantly, you'll get some good results...



```
\pos(256,358)\b1\frx0\fry7\org(118,804)\3c&H9E9491&\blur0.5
```

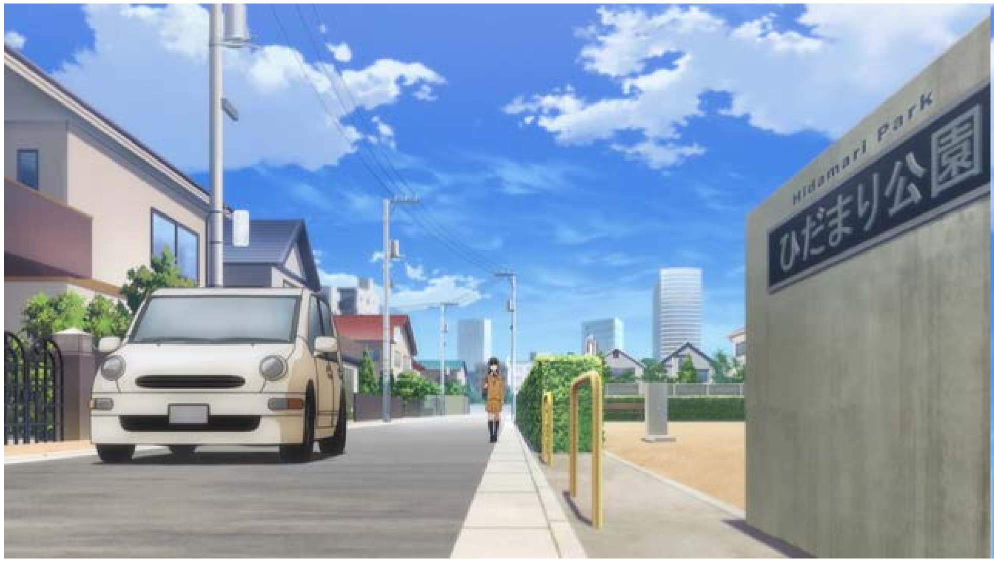
You can see that the sign looks pretty good and the only tags creating the alignment are \fry and \org.
(\frx0 can be deleted, it's just that the tool adds both tags even if they're 0)
The script resolution is 1280x720 so you see the origin point is off the screen in this case.
This method is especially useful when you need to do something more complicated
like you'll see in the last example at the bottom of the page.
Of course you can still fine tune this using \fax and/or \frz.

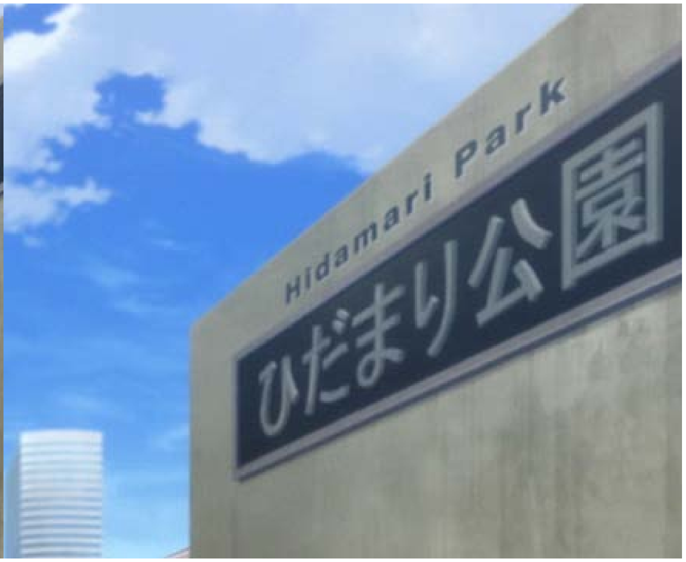
You may use various approaches and combine the tags as you wish.

For static signs \fry\org may be the best, but if you need the sign to move, you can't use \org,
so go with \frz\fax and possibly scale down the font size for some perspective if needed.

2018 addition:

So I had this sign in this week's Hinamatsuri:



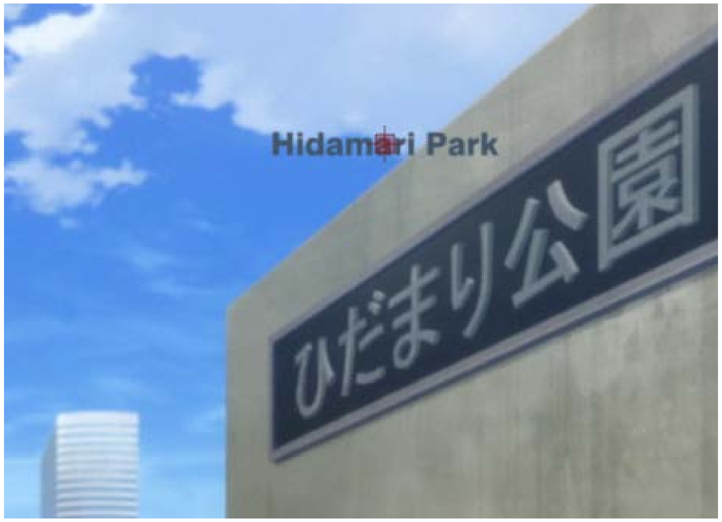


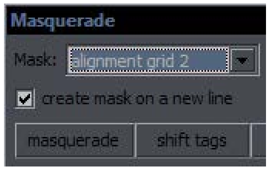
You'll come across this kind of sign fairly often, and it can be a pain in the ass. This was what I did. (above)
I went with \org and \fry, but I'm not sure if that was the best idea.
There's more about how to do signs with \org in **this blog post**, but here I'll explain the other option: \frz and \fax.

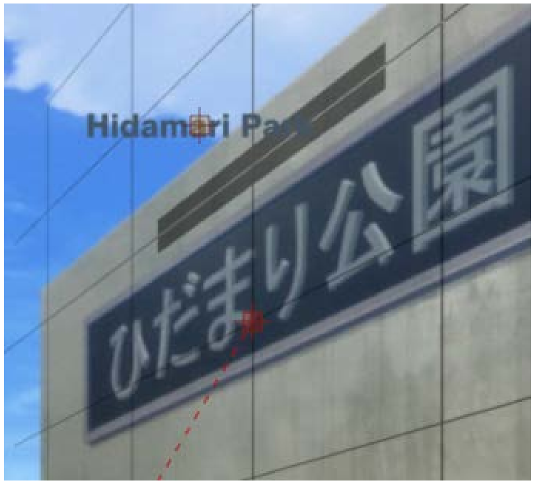
Many people would think this is not the sign to use it for, but the available tools have significantly improved, and it can actually be done quite easily and without much hassle... as long as you know how.

I'm gonna use the alignment grid here, but it's not really needed (and is the most tedious part of this).
I'll include it because this is a guide, but skipping it would save you some time.

Below is what I start with. I use **Masquerade** to get an alignment grid on a separate line.



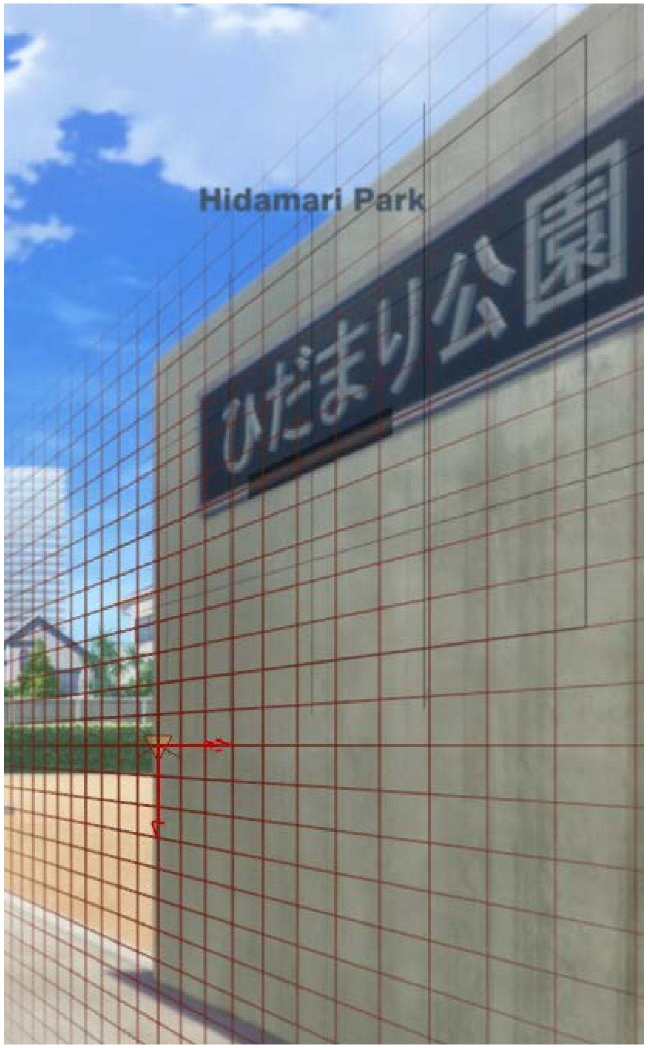

Masquerade
Mask: alignment grid 2
☑ create mask on a new line

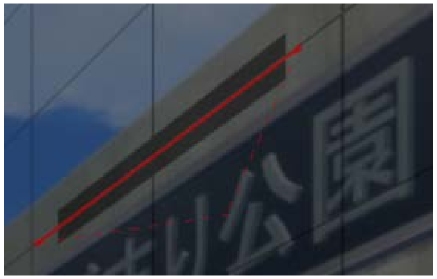


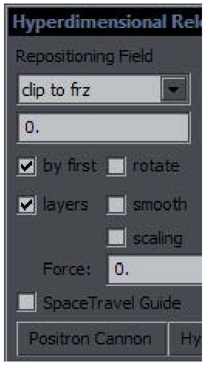
On the right, you can see it being aligned with the wall with the inbuilt rotations tool.
That's the part where you have to tinker with it a bit, but you can see both top and bottom of the wall, so it's not that bad.

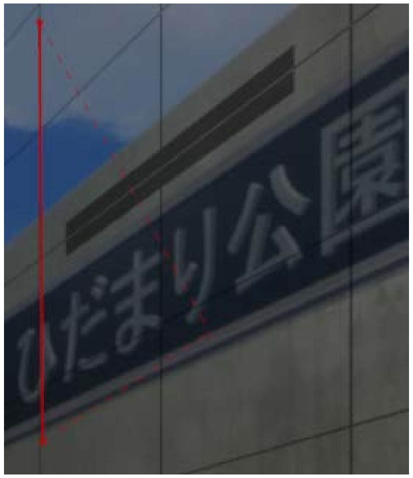
Once it's aligned, I move the dark rectangle to where I roughly want the sign. (left image)

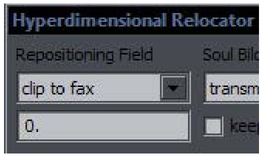
The only use for the grid here is that it can guide me for drawing the clips, but it could be done without it.



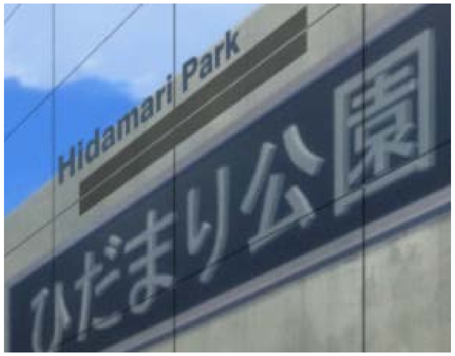



Hyperdimensional Rel
Repositioning Field
clip to frz
0.
☑ by first ☐ rotate
☑ layers ☐ smooth
☐ scaling
Force: 0.
☐ SpaceTravel Guide




Hyperdimensional Relocator
Repositioning Field
Soul Bil
clip to fax
transm
0.
☐ kee

So now I need to rotate the sign and set \fax. This only takes a few seconds. Draw two points with a clip for horizontal alignment, and use clip2frz from **Hyperdimensional Relocator**. This sets \frz. Draw another two points for vertical alignment, and use clip2fax to set \fax. (Also in **NecrosCopy**)



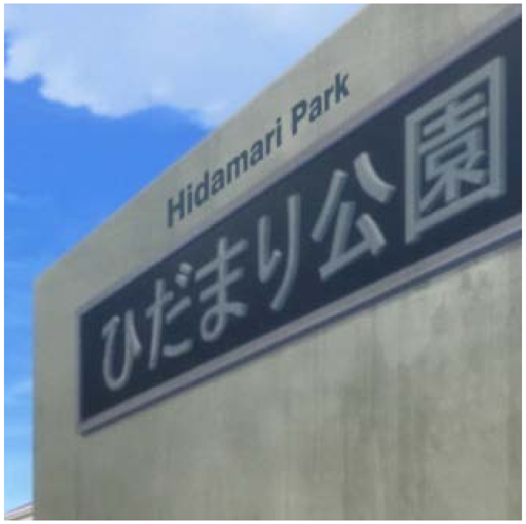
The result is on the left.

The one issue you need to realise is that if you draw the horizontal clip along the Japanese sign, the rotation will be slightly different.

To be precise, for the top of the JP sign it's \frz34.8,
and at the top of the wall it's \frz39.4 - a noticeable difference.
So you need to be somewhere in the middle, thus drawing at the grid mask.

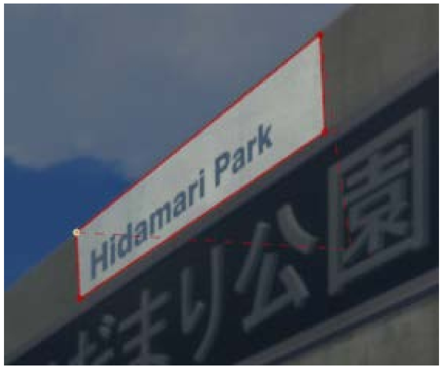
But you could just draw the line in the middle between the sign and wall top and get it accurate enough without the grid, so that's why I said the grid wasn't necessary. Also, I had totally forgotten that clip2frz can take average of two lines, so you can draw one at the top of the wall and the other at the JP sign, and the script will use the "midlle line", i.e. the average.
Just remember to draw points 3 and 4 in the **same** direction as 1 and 2. (Like a "Z")

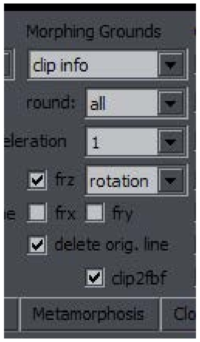
The vertical clip can be drawn at the edge of the wall or edge of the screen.



So on the right, you can see the result put in place, with the grid now removed.

The alignment is now good, but the remaining problem is that the font size is constant, while it should be larger on the right.





Morphing Grounds

clip info

round: all

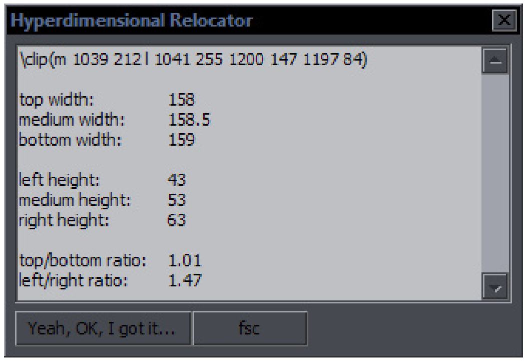
eleration 1

☑ frz rotation

□ frx □ fry

☑ delete orig. line

☑ clip2fbf



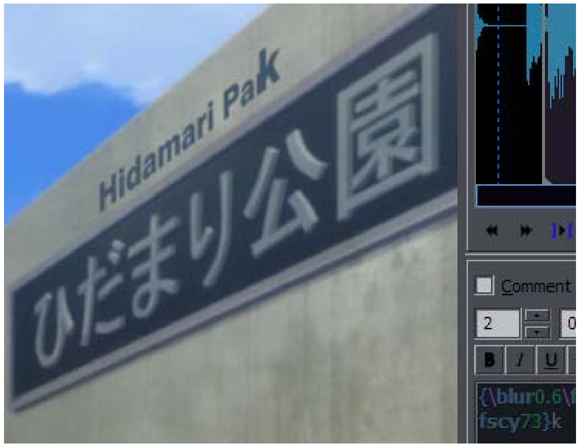
Hyperdimensional Relocator

```
\clip(m 1039 212 l 1041 255 1200 147 1197 84)

top width:        158
medium width:     158.5
bottom width:     159

left height:      43
medium height:    53
right height:     63

top/bottom ratio: 1.01
left/right ratio: 1.47
```

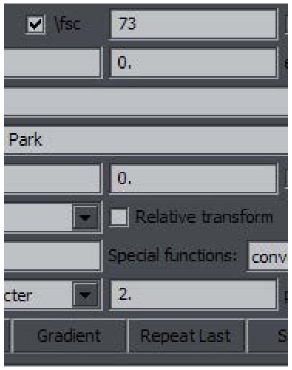
So here we use the clip tool one more time. Draw a rectangle roughly around the sign, with the left and right sides more or less vertical and the top/bottom lines matching some edges. The idea here is to get the difference in size between the left and right sides. The width of the clip should roughly match the width of the final sign. (Thus larger than the text here.)

Once you have the clip, use Relocator again, this time the "clip info" function. What you get is a bunch of measurements from the clip.
The one that interests you here is the last one, the ratio between the left and right heights. It's roughly 1.5 here, so the last letter should be about 1.5 times larger than the first.

This info would be enough to do the rest, but for convenience, the "fsc" button will help. It takes the starting values of \fscx\fscy, multiplies them by 1.47, and sets \fscx\fscy tags before the last letter in the line, thus preparing it for gradienting.



{\blur0.6\(
fscy73}k



☑ \fsc 73

0.

Park

0.

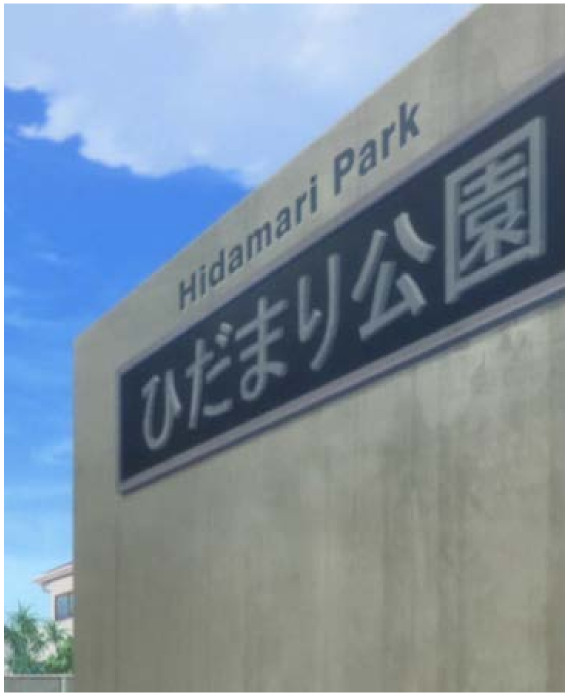
□ Relative transform

Special functions: conv

cter

2.

As you can see, there's "fscy73" visible before the last character. (I had scaling at 50 at the start.)
You can either use lyger's GBC or, as the second image shows, use **HYDRA**, set "fsc" to 73, and click on Gradient. (Make sure "by character" is selected.)



And that's it. Now you can just make some final cosmetic changes: tweak colour, alpha, blur, shadow, or whatever makes the sign blend in better.

If you feel like the letters are too tall or too wide, you can change \fscx or \fscy. The inbuilt tool won't work because you have a gradient, so you use **Recalculator**.
If you only change x or y, it will throw off \fax, but you fix that in 5 seconds with clip2fax again.
(BTW, if you make 4 points for the clip, first 2 points left side, second 2 points right side, each side determines \fax for that end, and a \fax gradient is created.)

So if we skip the grid with the rotations, this is all pretty simple and no \org or messing with the rotations tool is needed.

To recapitulate:

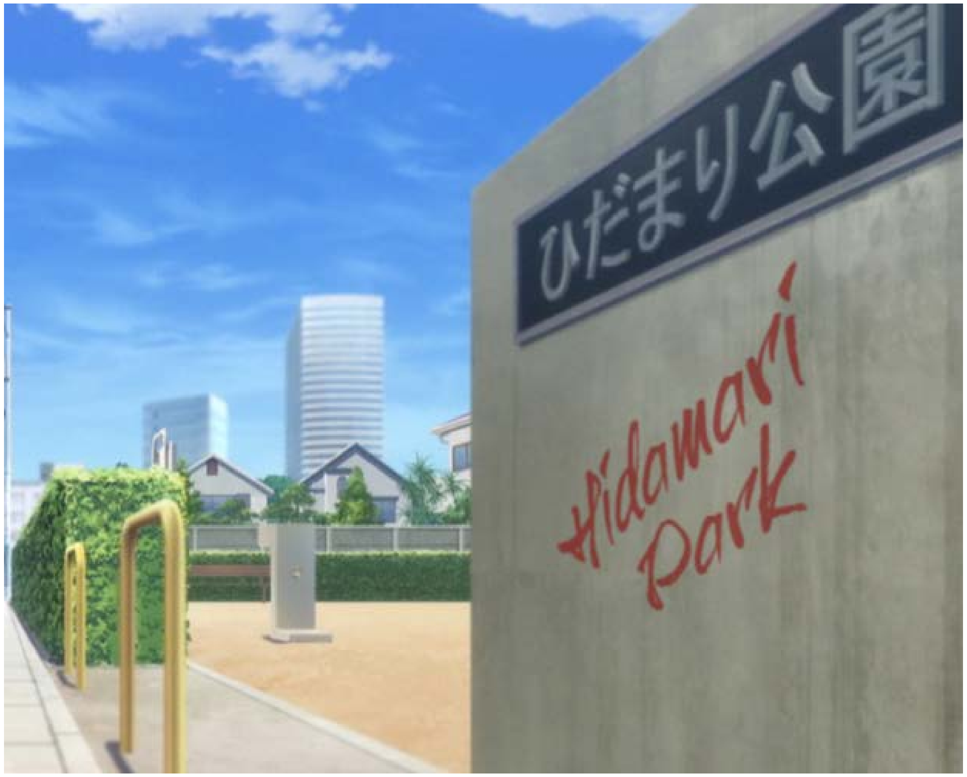
1. two clip points -> clip2frz -> you have rotation
2. two clip points -> clip2fax -> you have the whole alignment done
3. rectangular clip -> clip info -> you have scaling set up
4. gradient the fscx and fscy

All 4 steps done in under a minute. (I just tried it - took about 40 seconds.) The rest is just colours and stuff.

One more thing regarding that sign. It's one of those where you think about how to do it not as much in terms of technical approach, but more in the sense of what you really want the result to look like. Should I try to fit small letters on that blue plate? Should I replicate the whole plate and make an English copy? (This is dedicated, but nobody would solve it that way in real life, so it's bound to look stupid.) The way I chose to do it, what should the colour really be? Light grey? Dark grey? Blue?

There's basically no way to make it look natural. If in real life they wanted added English, it would be on the blue plate but certainly larger than you can fit here. I suppose enlarging that plate for that is an option too. Enjoy the gradients.

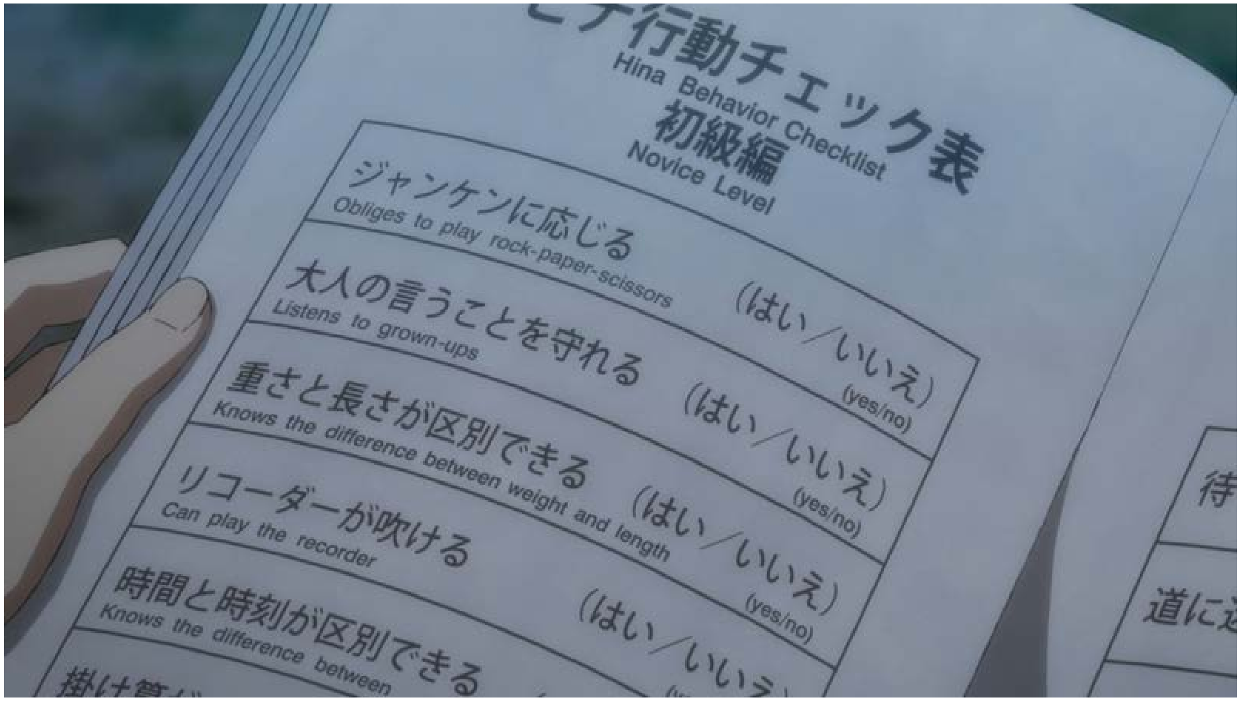
So really, sometimes I wonder if it's better to just do something like this:



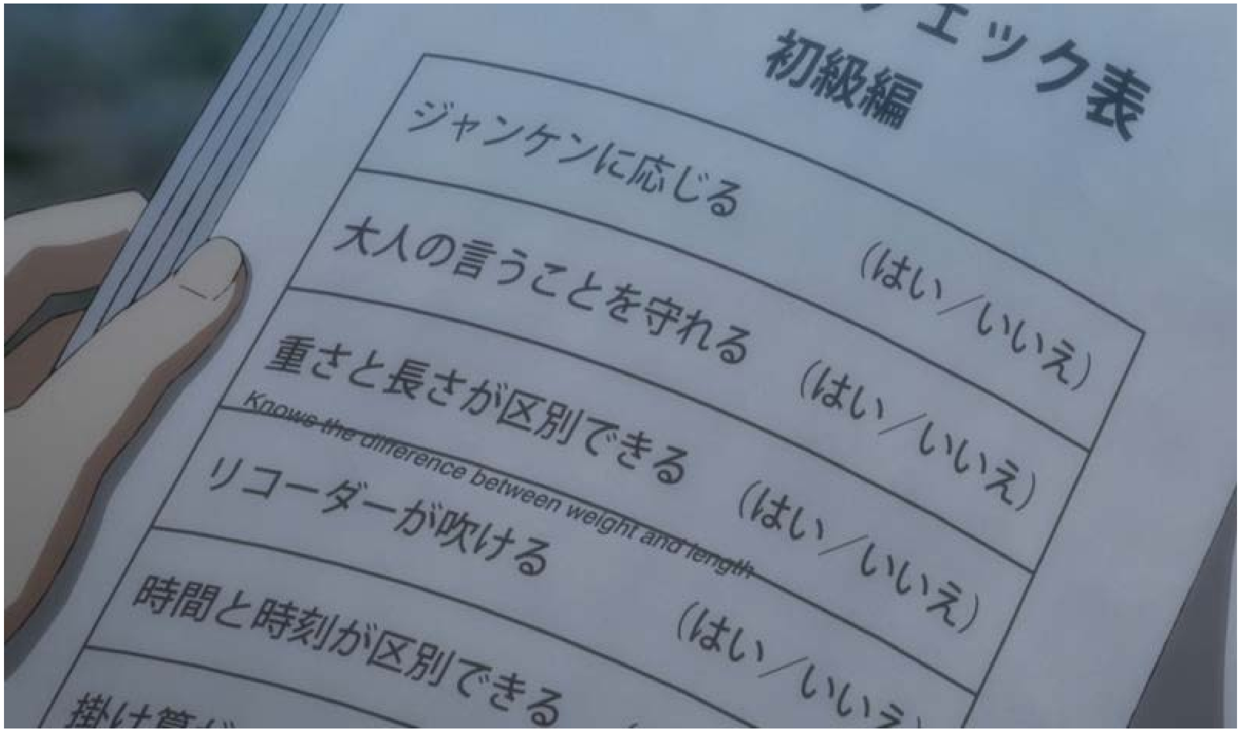
After all, the notion that somebody just sprayed it on the wall is a lot more realistic than the sign I did before.
And you can wing all the rotations and not bother with any measurements.

Anyway, to make a final point, in the vast majority of cases, between \org\frx\fry and \frz\fax\fscx\fscy, I would recommend the latter. (And you usally don't need the scaling.)

Let's move on to something a bit more advanced. Another week in Hina...

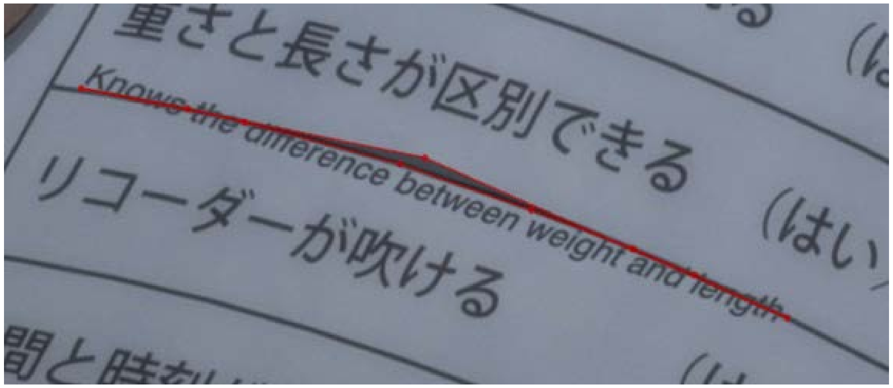


I'm sure you love when you get one of these. This episode had 40 signs (counting this as one), so I already had plenty to do, and I knew this one would take a while. And it did.
I did it all the tedious way, splitting into words, adjusting positions, rotations, \fax, colours, and making two layers for glow.
I actually did it more tediously than I had to because I guess I wasn't thinking too straight at 1 am after 3 hours already at it.
But this is what propels my script writing. This was the most annoying sign in the episode, so I figured I needed something to do it more easily.
So I thought about it, and the next day, I wrote something that was surprisingly easy and turned out to work better than I expected.

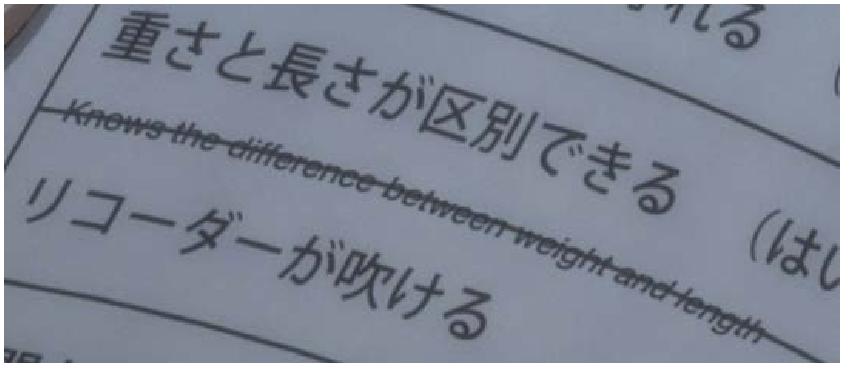


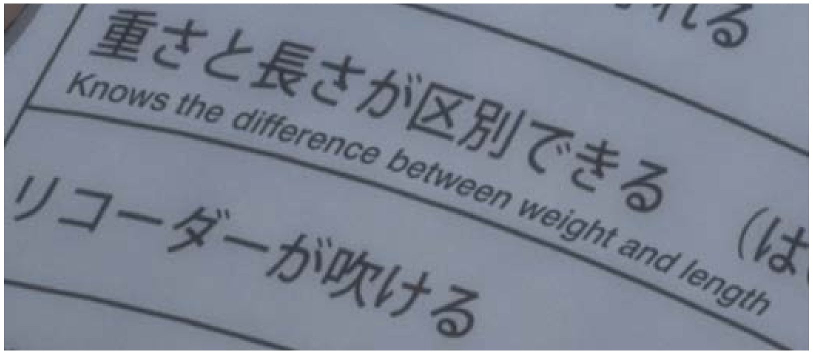
What I'll be using here is NecrosCopy's "Split by \N", which I improved for splitting along a clip.
Now, line0 has this MoveAlongPath script that does something like that. I've never seen/used it, but afaik, it splits into letters along a curved clip.
Which is great and probably quite precise, but here I have 200 characters and 2 layers. I don't really want 400 lines for this.
So my idea was to keep the splitting to words, but somehow position and rotate each easily. You already know how to do the \fax thing.

So I take the line and put it somewhere that allows me to draw the clip. I want to draw the clip along the grid line, which will guide me.
I can move the text above it later. I set the size and rotate the text to get the closest alignment I can with a straight line.

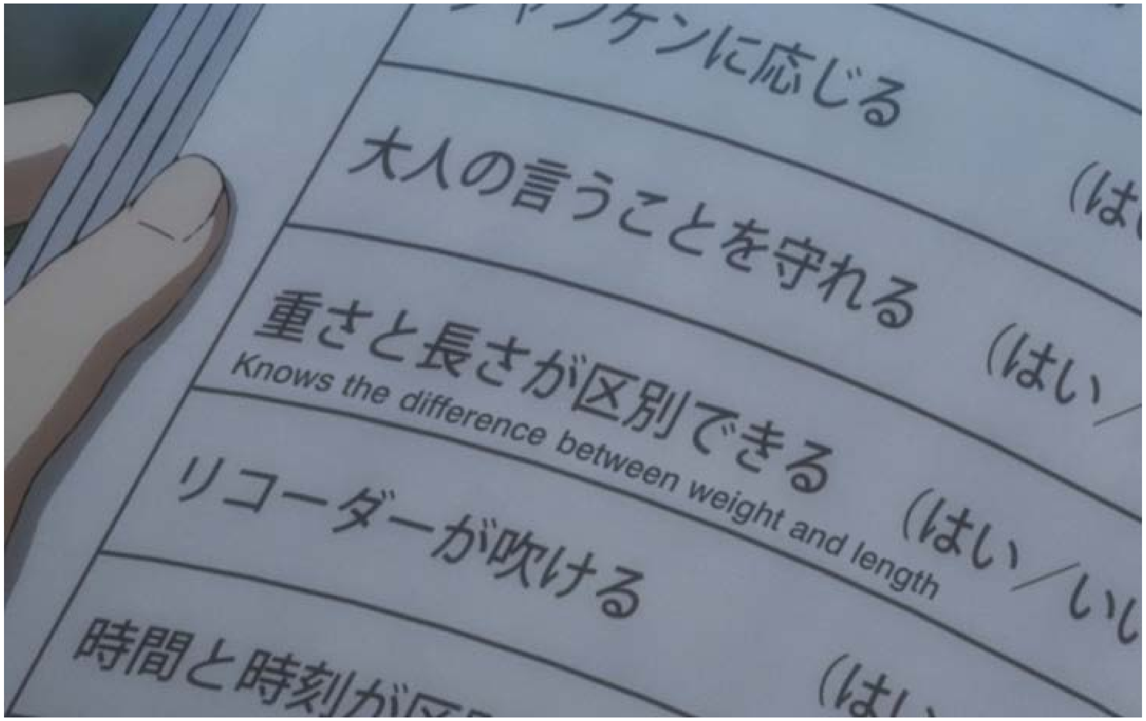


You need to see the text while you draw, so either duplicate the line, or convert the clip to iclip right after making the first point.
What you do then is make points at the edges of words, like you see above, going along that grid line.
(That one point above "be"[tween] is an extra last point, just so i could see the words better.)
The text should have centre alignment (\an5 or 2 or 8), so \an5 will be set no matter what you have there.
What the script will do is set position for each word to the centre between the 2 clip points around that word and also set the rotation based on that segment, like with clip2frz.



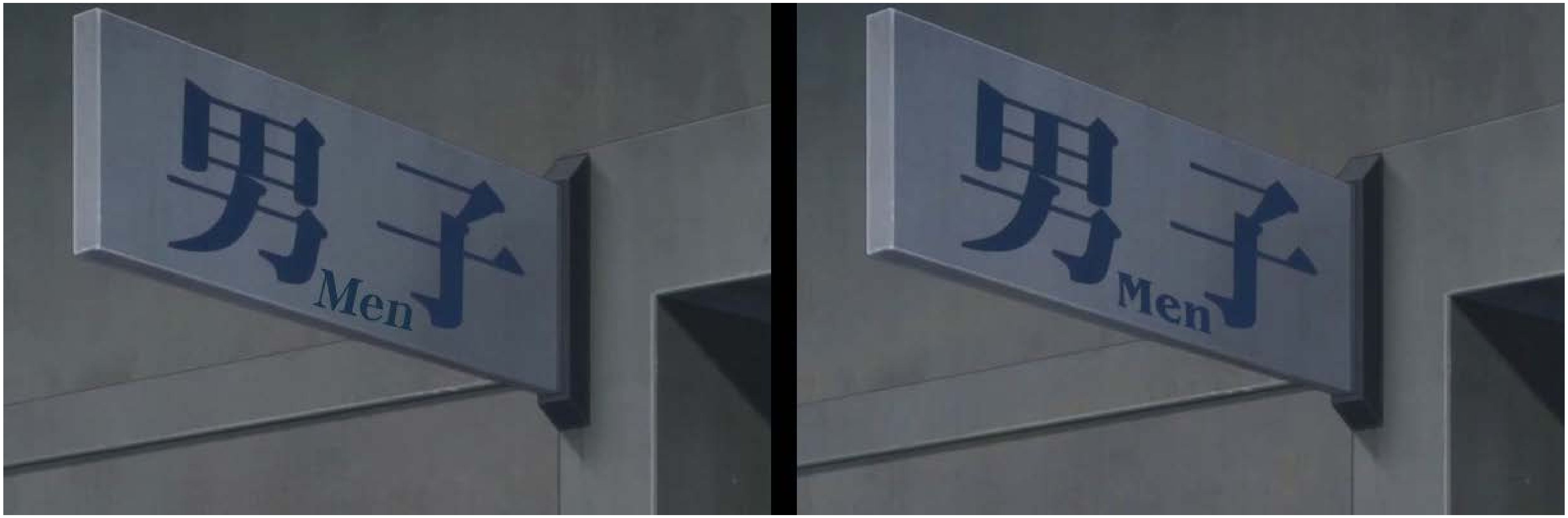


Now you simply run "Split by \N" and have it split by spaces, and the image above shows the result. Then you drag the text to the right position.

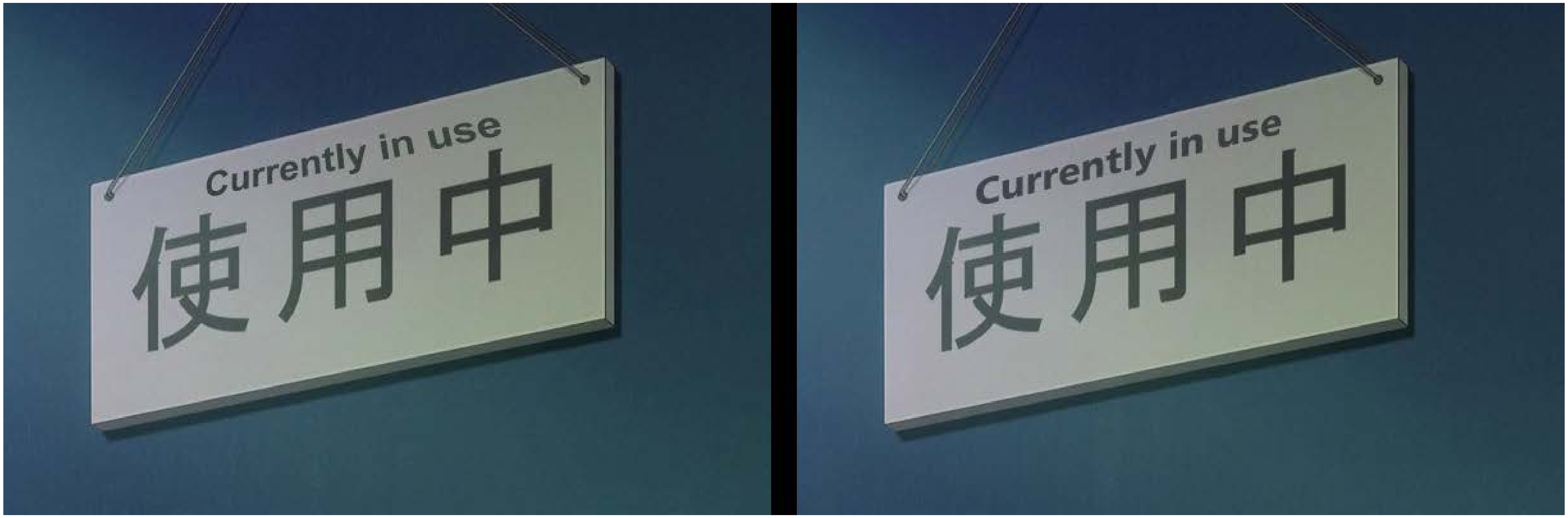


Last tweak is to draw a clip line along the left edge of the grid and use clip2fax. And that's it. I don't think the alignment needs any more work.
For long words, you might split them in two or put a frz tag in the middle and rotate each half slightly differently, but in most cases, this will do.
And it's probably about 5-10 times faster than what I did originally.
The clip must have as many points as there are words, plus one. Words joined by dashes or something will be considered one word, as we're splitting by spaces.
If you have something like I had, "rock-paper-scissors", just put spaces before or after the dashes, and you will nudge the words a bit when it's split.
The inaccuracies won't be significant.

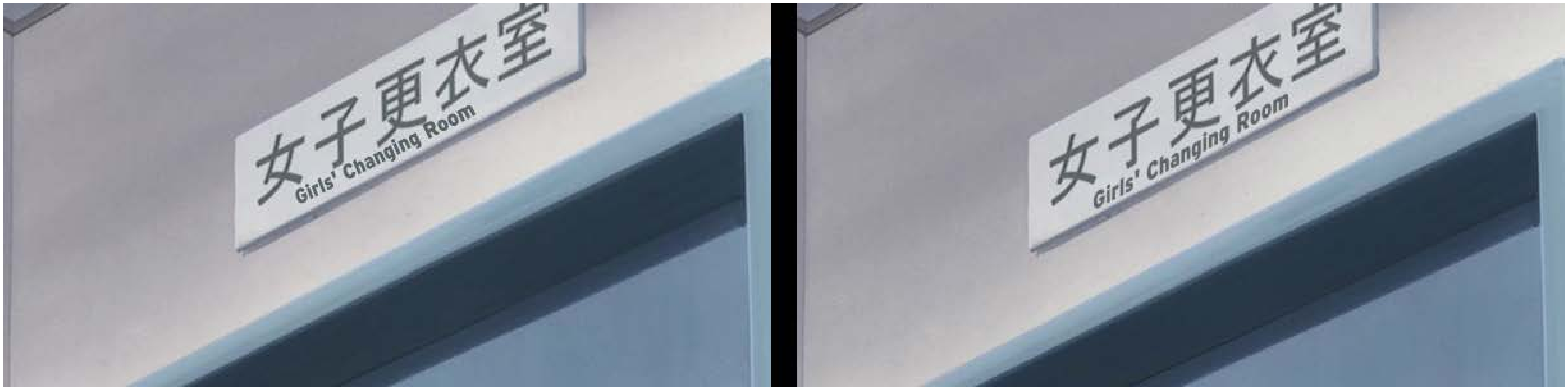
A few examples of signs I've found in some groups' releases... (These are from aroud 2011 and before. We've gone a long way.)



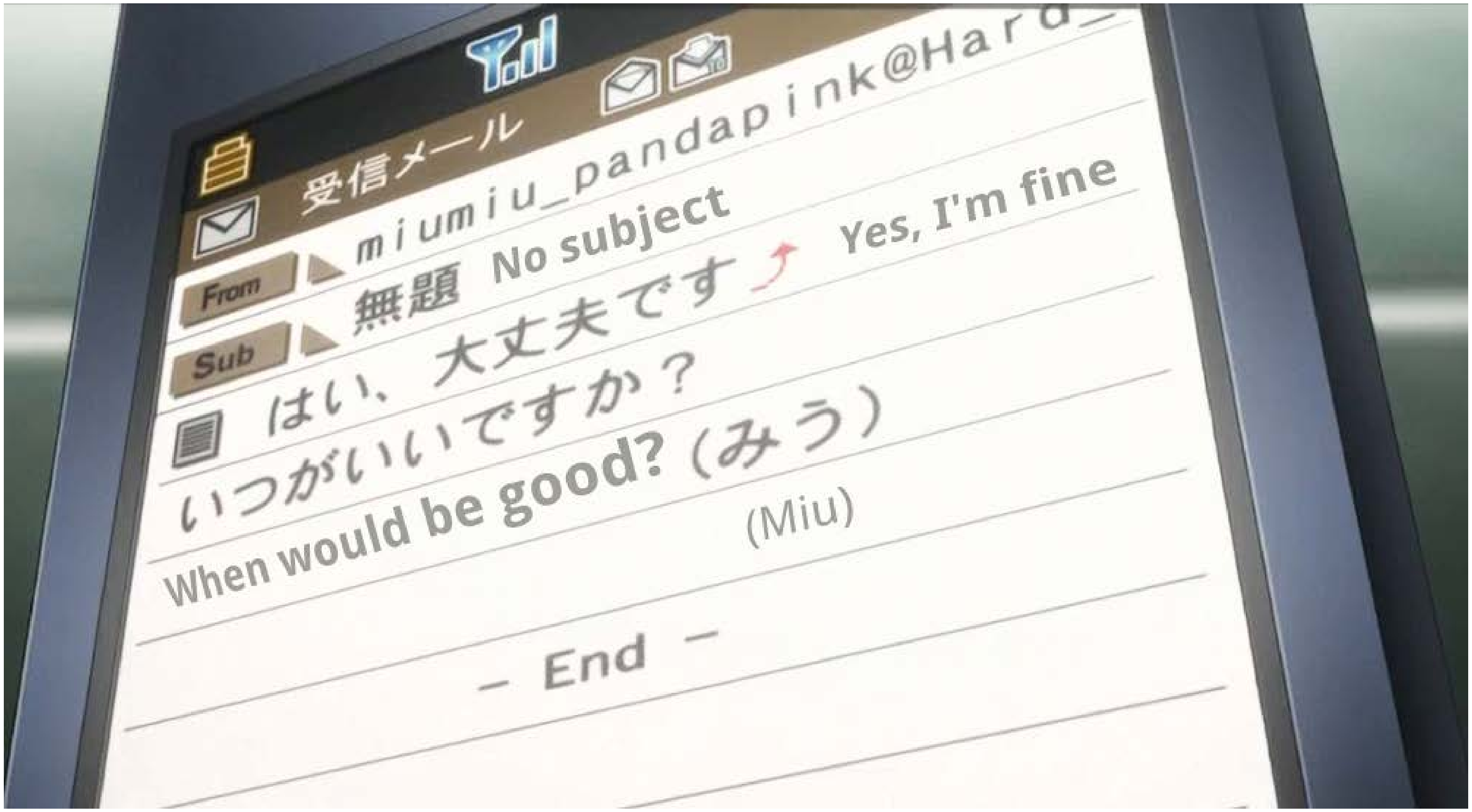
On the left is the original sign, on the right is my quick adjustment.
Clearly this looks bad and totally out of place - alignment is terrible, even the \frz isn't quite right.
You can also see the colour is off. I didn't like the choice of font either but that's the least of its problems.
Even my sign doesn't look too great, but at least it doesn't look retarded.



While in the previous example it was only \frz and good-bye, here some "pro" went for \frx and \fry, clearly without a clue how to use it.
It is sad, because it only takes a minute to do what I did on the right. Whoever did the one on the left should be fired.
I wanted to just quickly fix this to make an example so I went for \fay. If I actually wanted to release this,
I'd use \frz and \fax, and then change colour after every 2-3 letters to match the change of colour on the japanese sign.



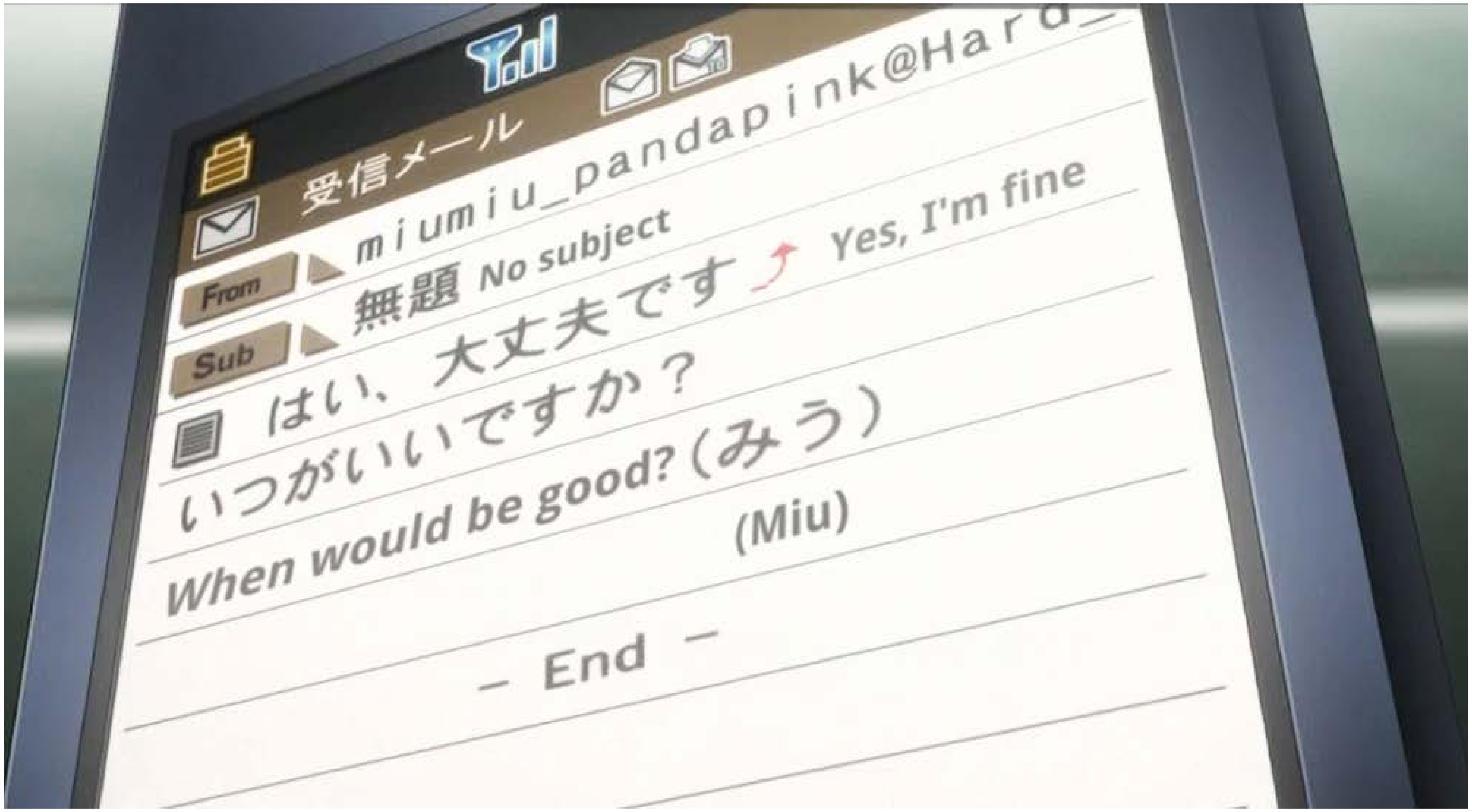
It just keeps happening, doesn't it. I'm not sure why groups that have been around for years have problems with this.



Now this... is admittedly not an easy one, but this implementation is pretty poor, especially with each line having different (mis)alignment.
The "When would be good?" line is so bad that I'm guessing the author must have been in quite a hurry to finish this.

Here's where \fax and \fay won't be enough and you'll have to use rotations, and you'll **have to** use the \org tag, if you really want it to look good.
2018: No, fuck you, younger me. No \org, just \frz and \fax. Scroll down...
One other thing you'll have to use is blur. Its lack in the one above makes it even worse. And the colours are off as well.

It will certainly take more than a minute, possibly even 10-15 minutes to get all the 4 lines right.
But hell, if you look at the one above, and the one below that I made, you'll have to admit that it's worth the time.



受信メール

From ▶ miumiu_pandapink@Hard_

Sub ▶ 無題 No subject

はい、大丈夫です Yes, I'm fine

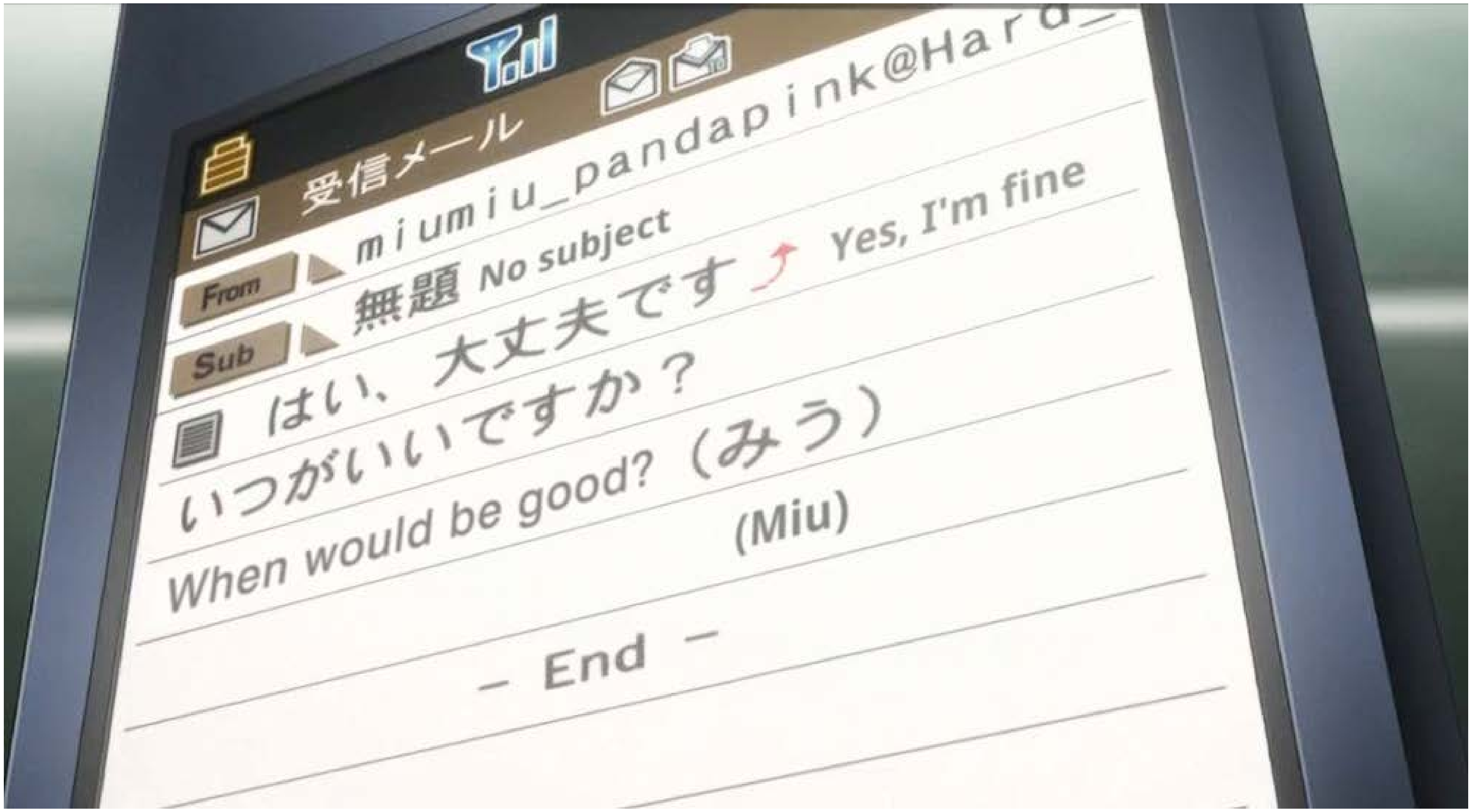
いつがいいですか？

When would be good?（みう）

(Miu)

− End −

2018: So I saved the image, loaded it in Aegi, and redid the "When would be good?" part with \frz\fax. Two minutes or less.



miumiu_pandapink@Hard_

無題 No subject

はい、大丈夫です Yes, I'm fine

いつがいいですか？

When would be good?（みう）

(Miu)

– End –

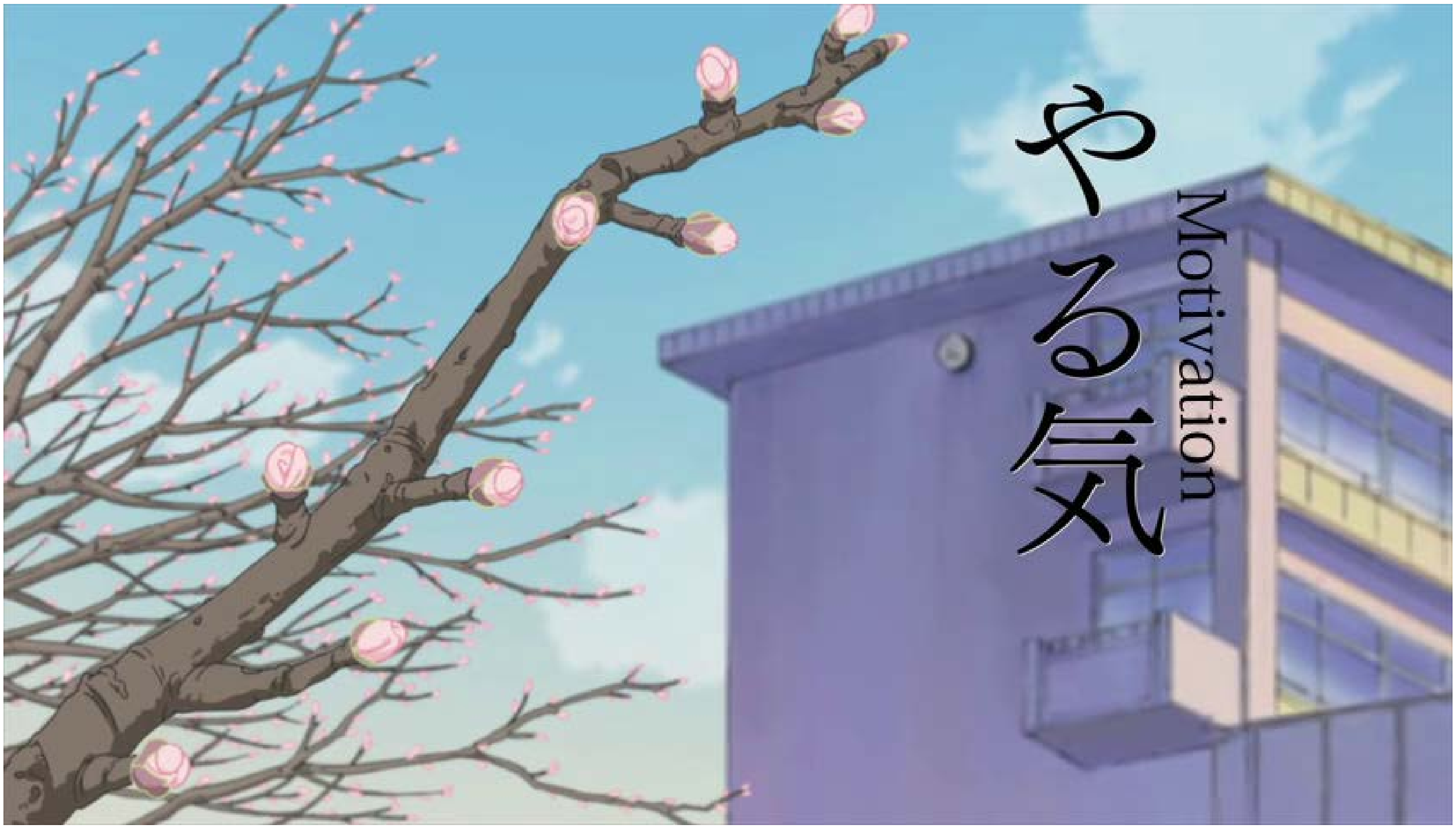
« Back to Typesetting Main

Typesetting: Positioning Signs

There are several things that determine how good a typeset is and how well it blends in.
Obviously, the colours and sizes of font, border, shadow. Then of course the choice of font.
The next thing would be the positioning of the sign.

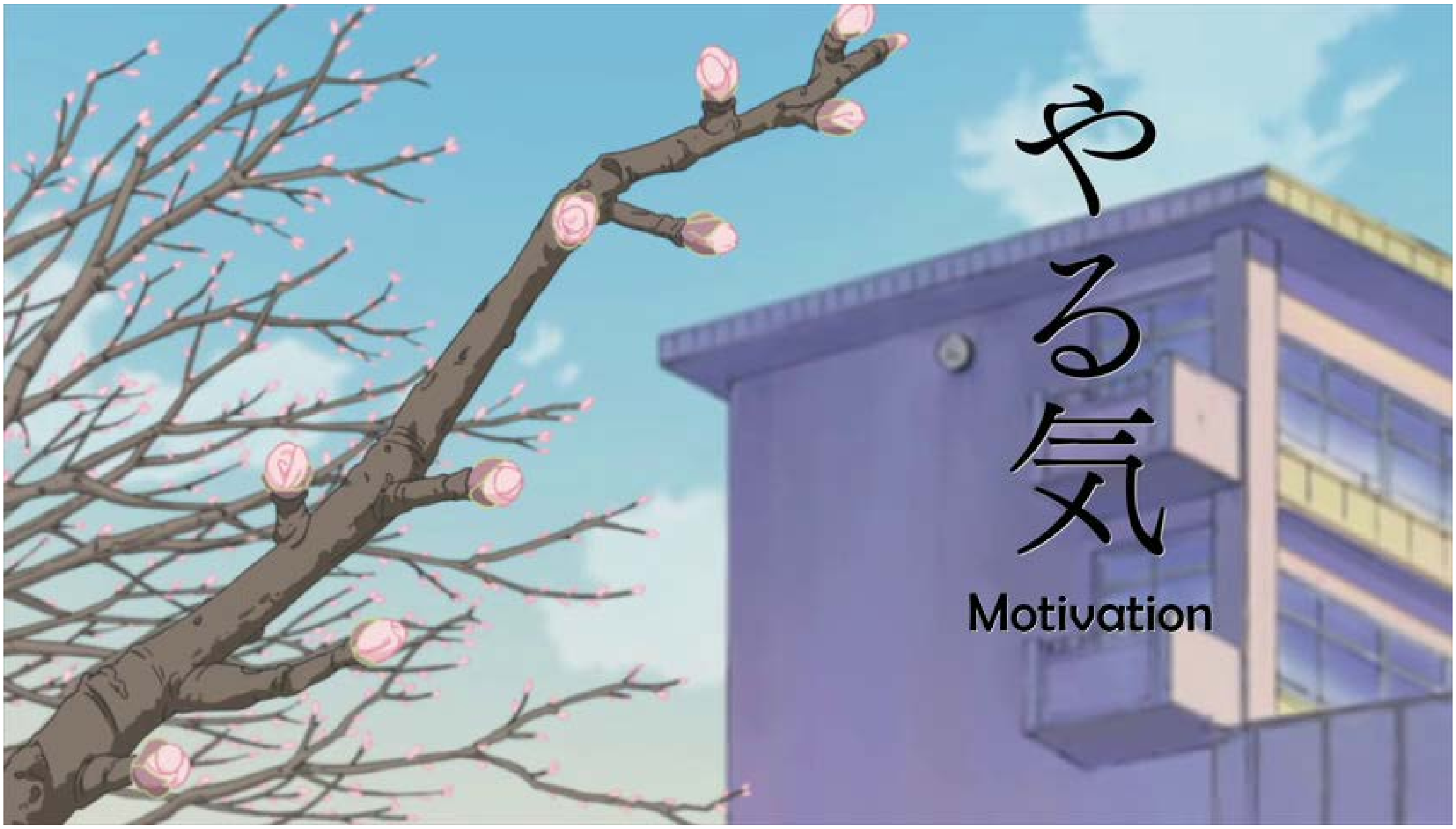
Beginners will have this idea that the typeset for a sign should be as close as possible to the original sign.
Please get this idea out of your head, because half of the time that won't work very well.
It is especially unnecessary for typesetting titles.
If it's a sign that belongs to a specific place on the screen, then sure, you wanna get your TS close to that.
But an episode title has little relation to what's on screen in terms of placement,
so no need to go crazy trying to match the position and orientation of the sign while sacrificing readability.

I used an episode of Nichijou for a test. Here are the results from some guys.

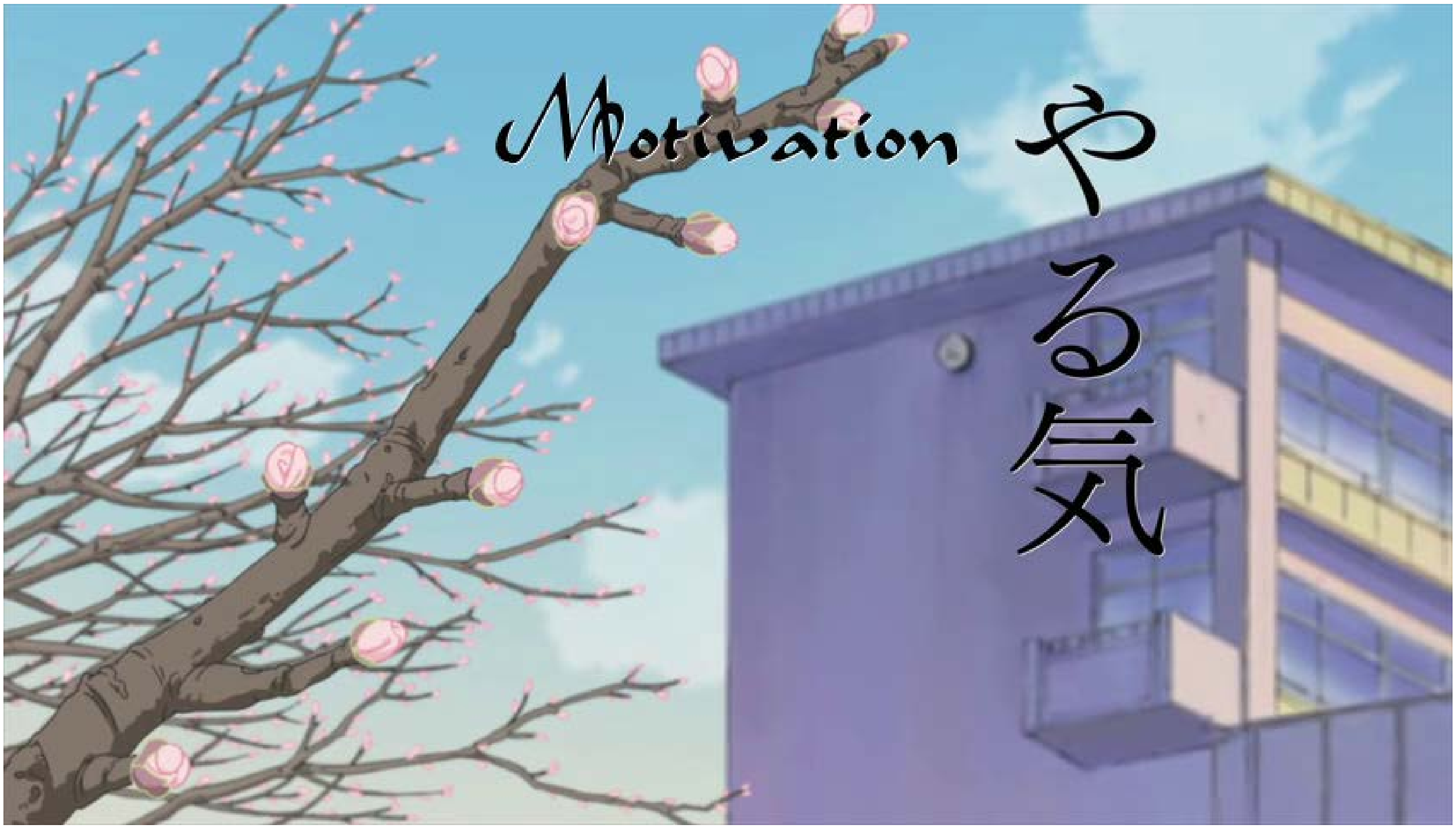


Kanji is often written top-to-bottom, rather than left-to-right.
It works great with kanji. It does not with English.
So don't try rotating the text like this just to match the orientation of the japanese sign.
It decreases readability and doesn't really look that great.
You might think of using M\No\Nt\Ni\Nv\Na\Nt\Ni\No\Nn. In other words top-to-bottom, without the rotation. Well... try it.
For one, you will usually have large spaces between the letters. But even if you pick a font that doesn't do that,
or do each letter separately, you'll run into another problem. Kanji has pretty consistent width. English letters don't.
The difference in width between M and i can be anywhere from 500-1000%, depending on the font. So this will usually not work either.

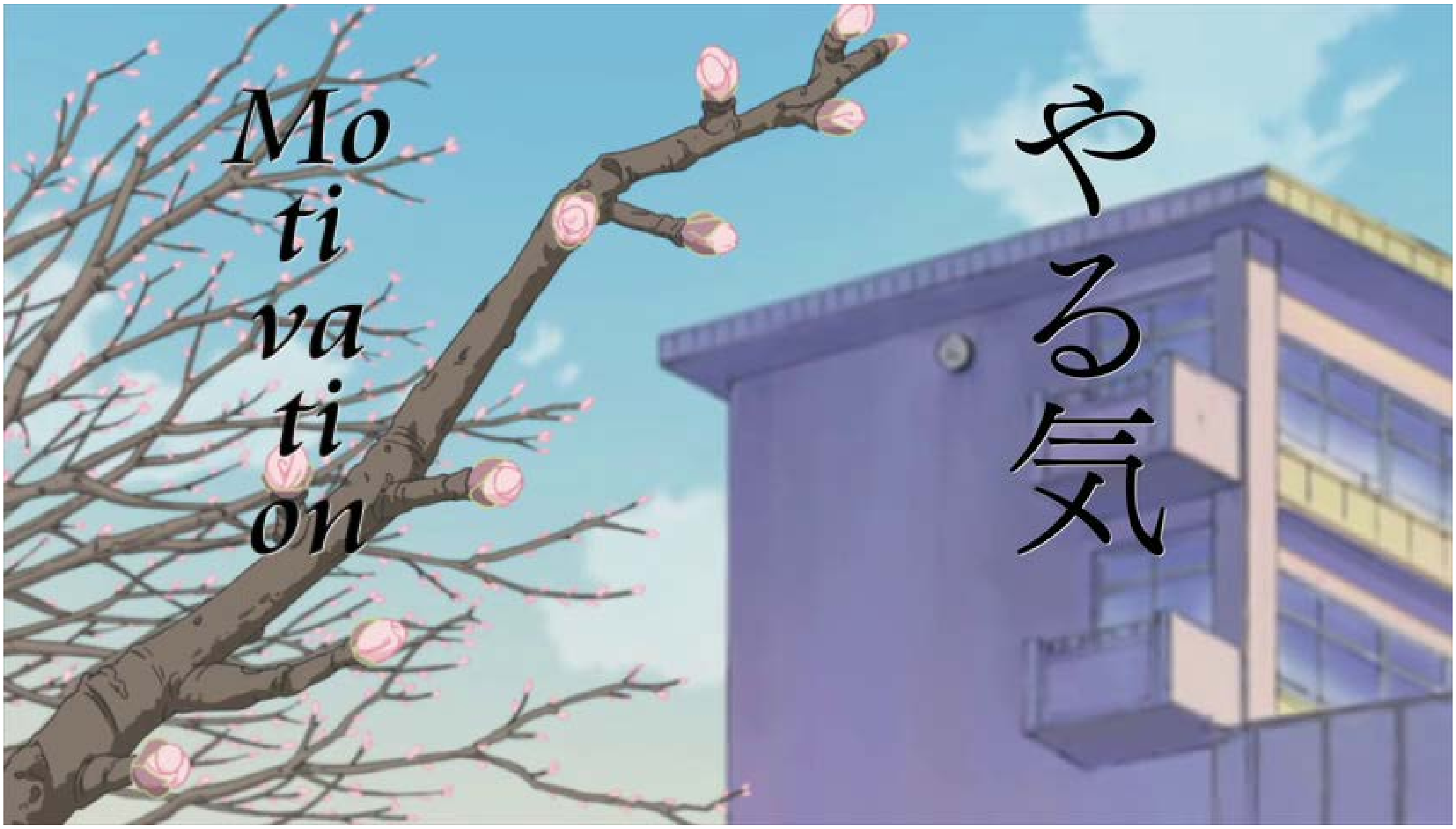
Also here ^ the white 'shadow' is missing.



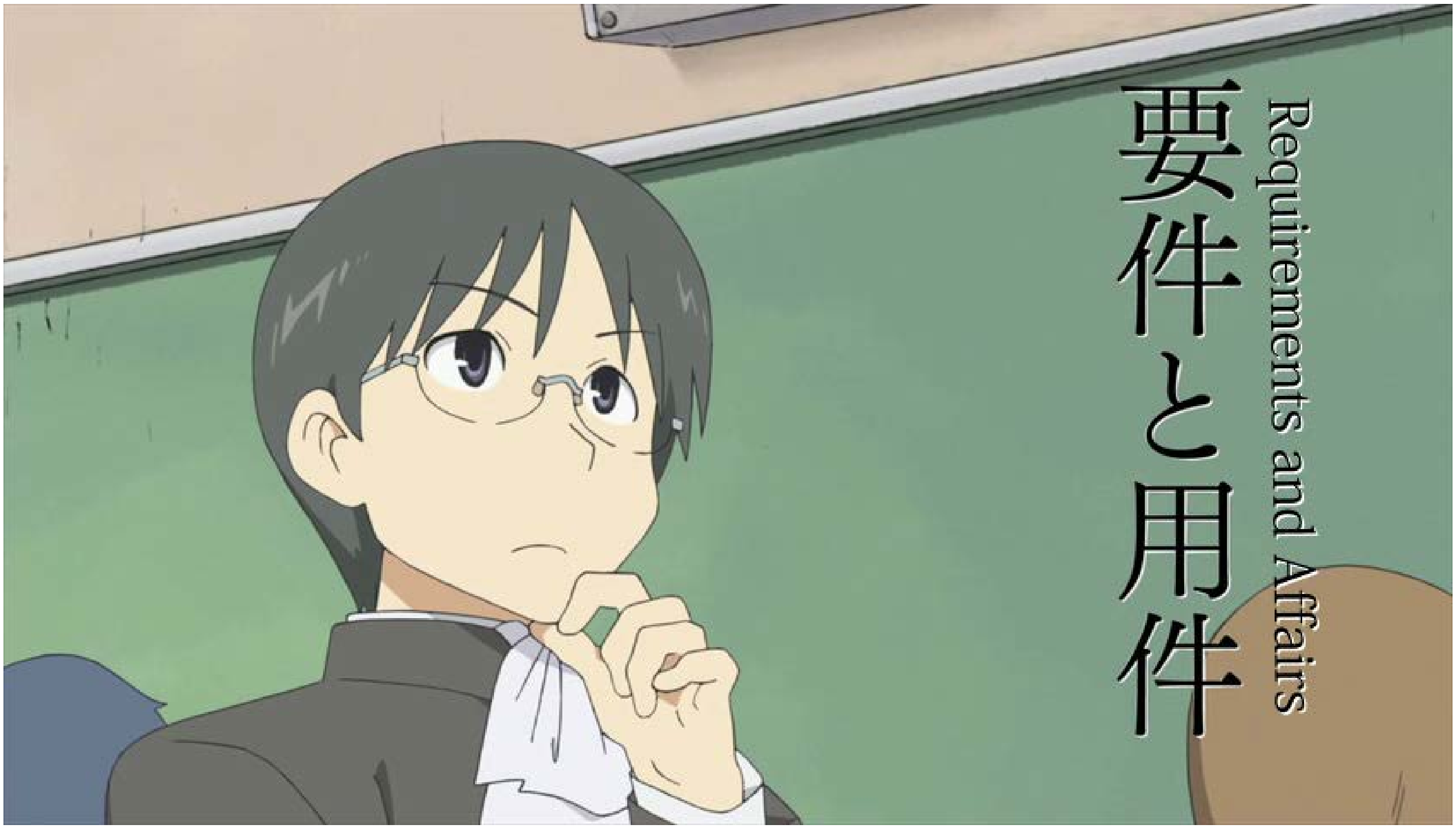
This is pretty good placement. It's much better to sacrifice font size and orientation if the result is well readable and looks good. The shadow is a bit too thin and jagged but this is overall fine, except that the font should be a serif one.



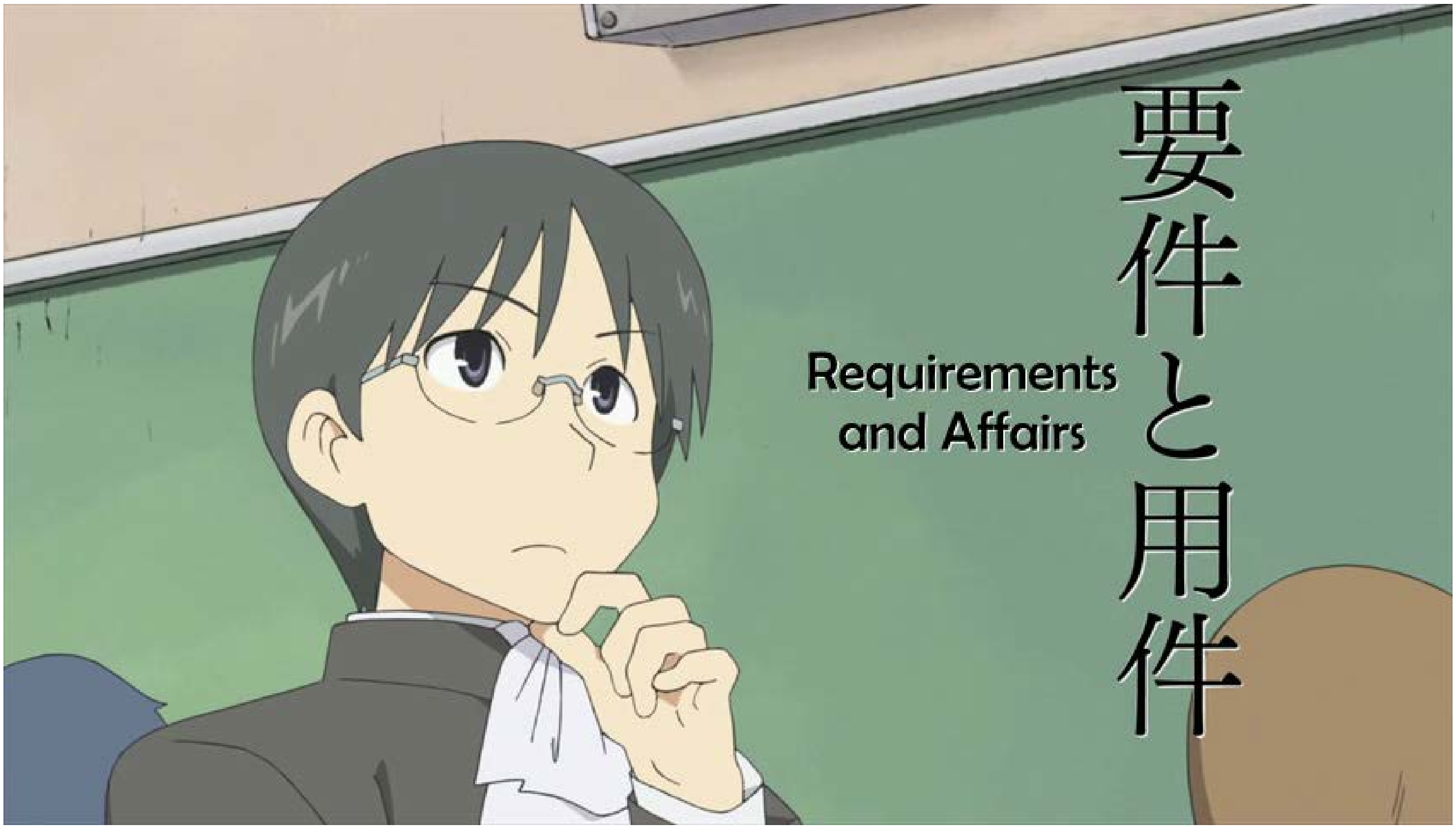
You can do something like this. Pick a font that will look good, you can make it large enough so that it's both readable and close to the size of the kanji, and keep at least some kind of alignment when posotioning it, in this case the top aligned with top of the the kanji.



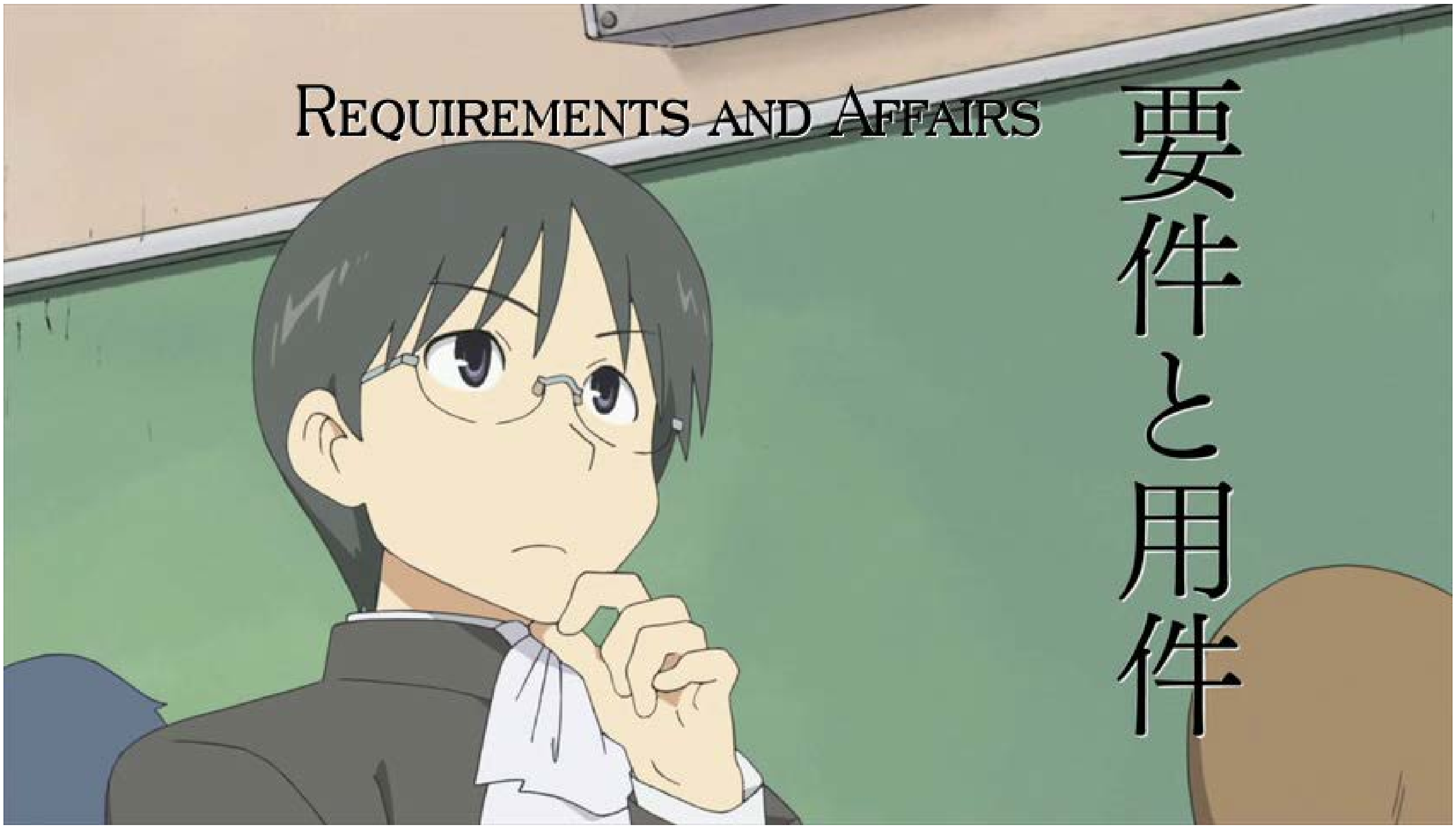
You can even put it on the other side of the screen for overall symmetry.
If you want it top-to-bottom, it may be better this way than rotating, but I'd say it's almost always better to keep it left-to-right.
Trying to split the text like this will rarely work well for a number of reasons, like the width inconsistency that you see here.



Another title. Here I think if the font was larger, to match the span on the kanji precisely, thicker, and placed a bit farther,
it might actually not *look* too bad (putting aside the readability issue)...

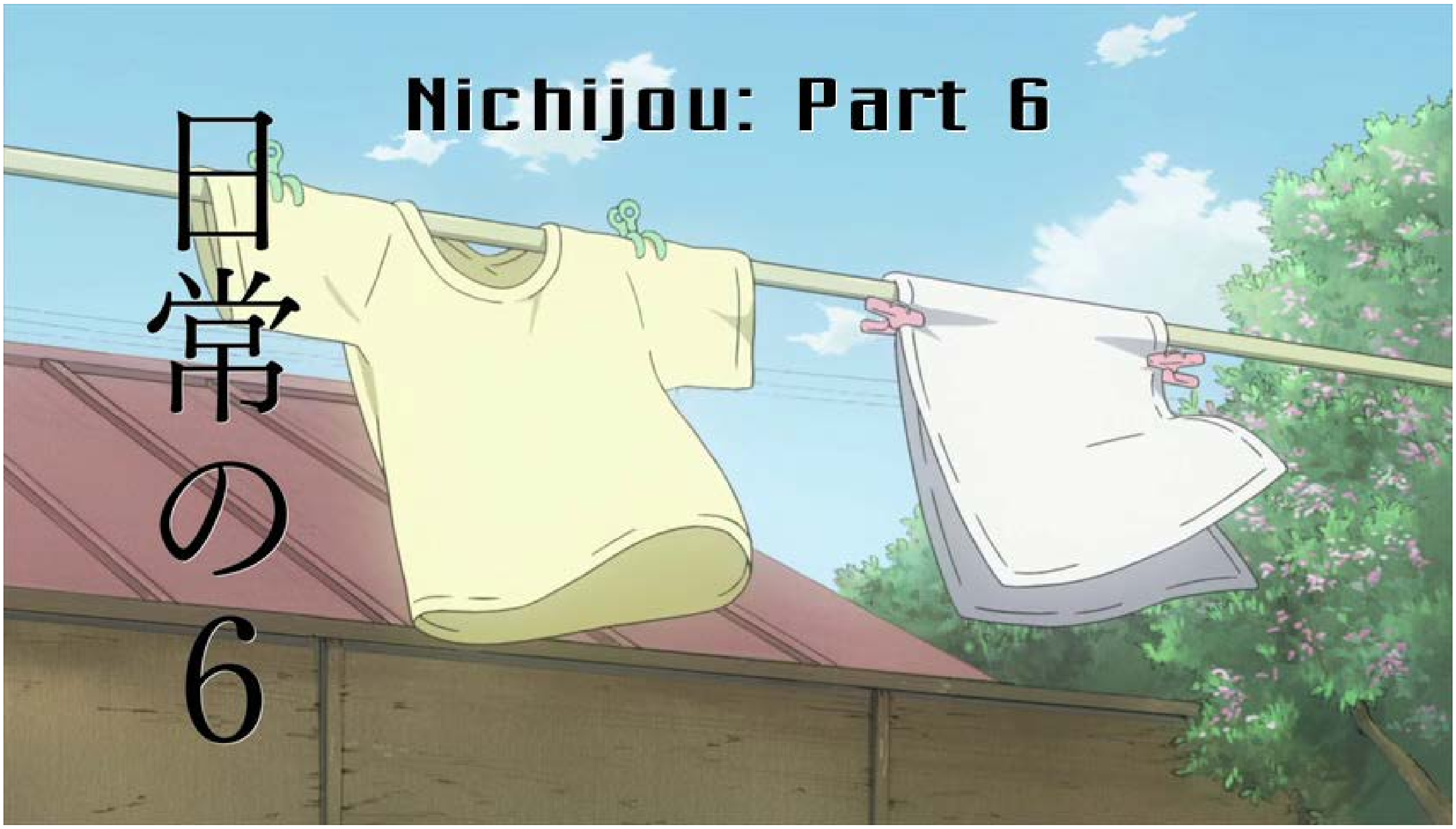


...but clearly this works much better. It's aligned to the centre of the kanji, looks fine except for the font.

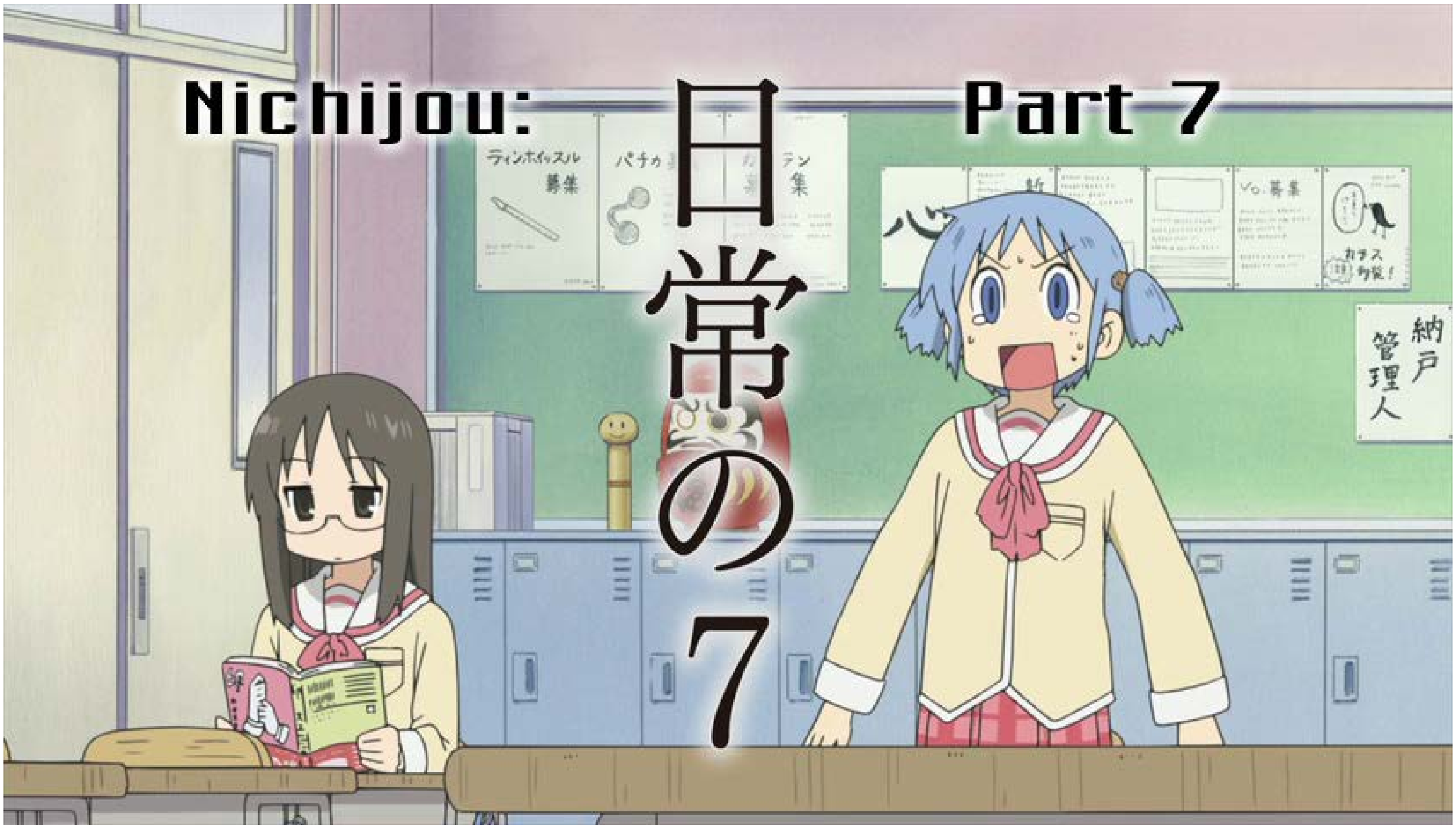


Here aligned to top, with a different font.

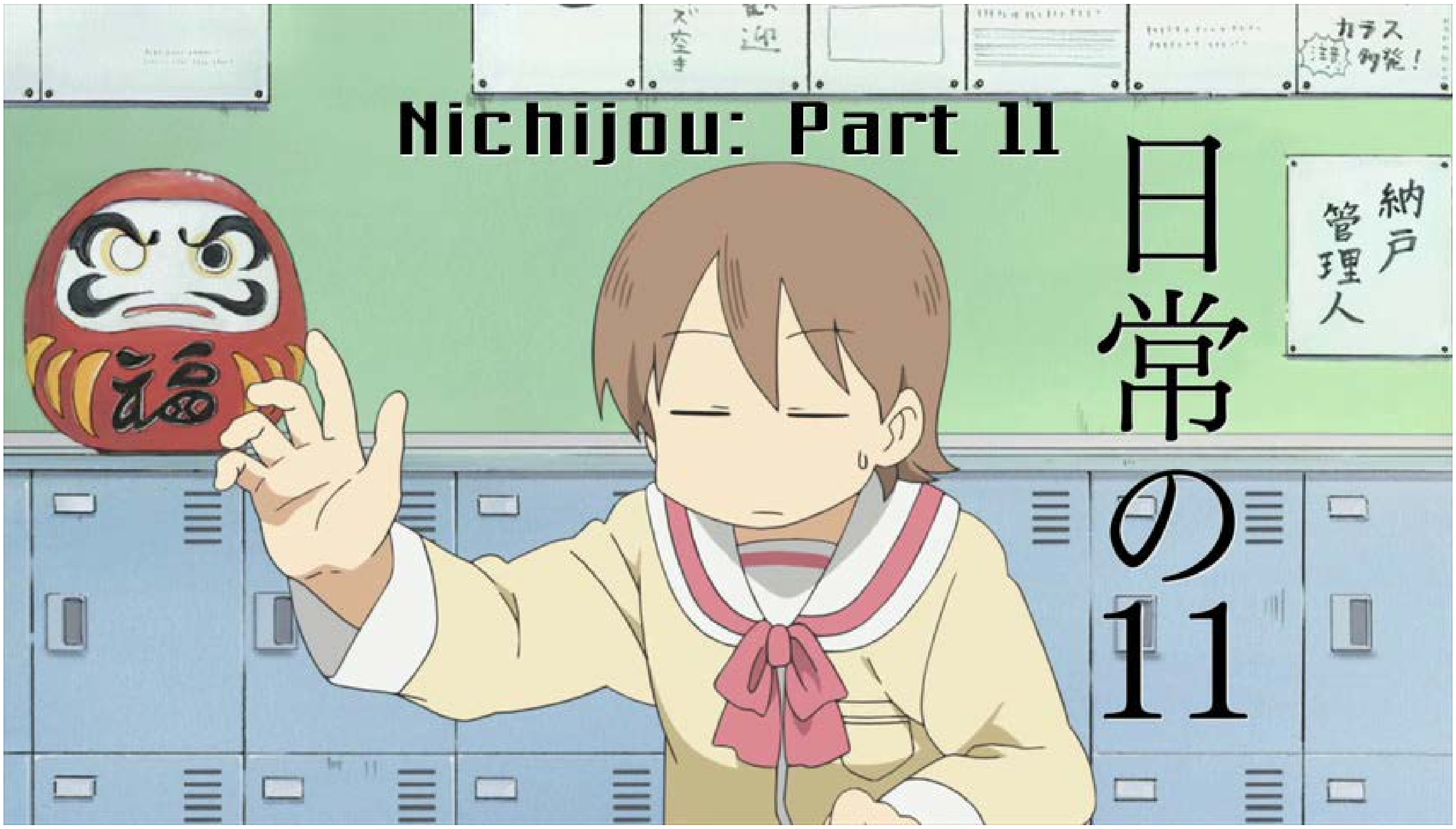
There's never one "correct" place for typesets.
Rather than following some rigid logic for where a sign should be, just make it "look good."
It doesn't even necessarily have to be aligned with the original sign in any way.



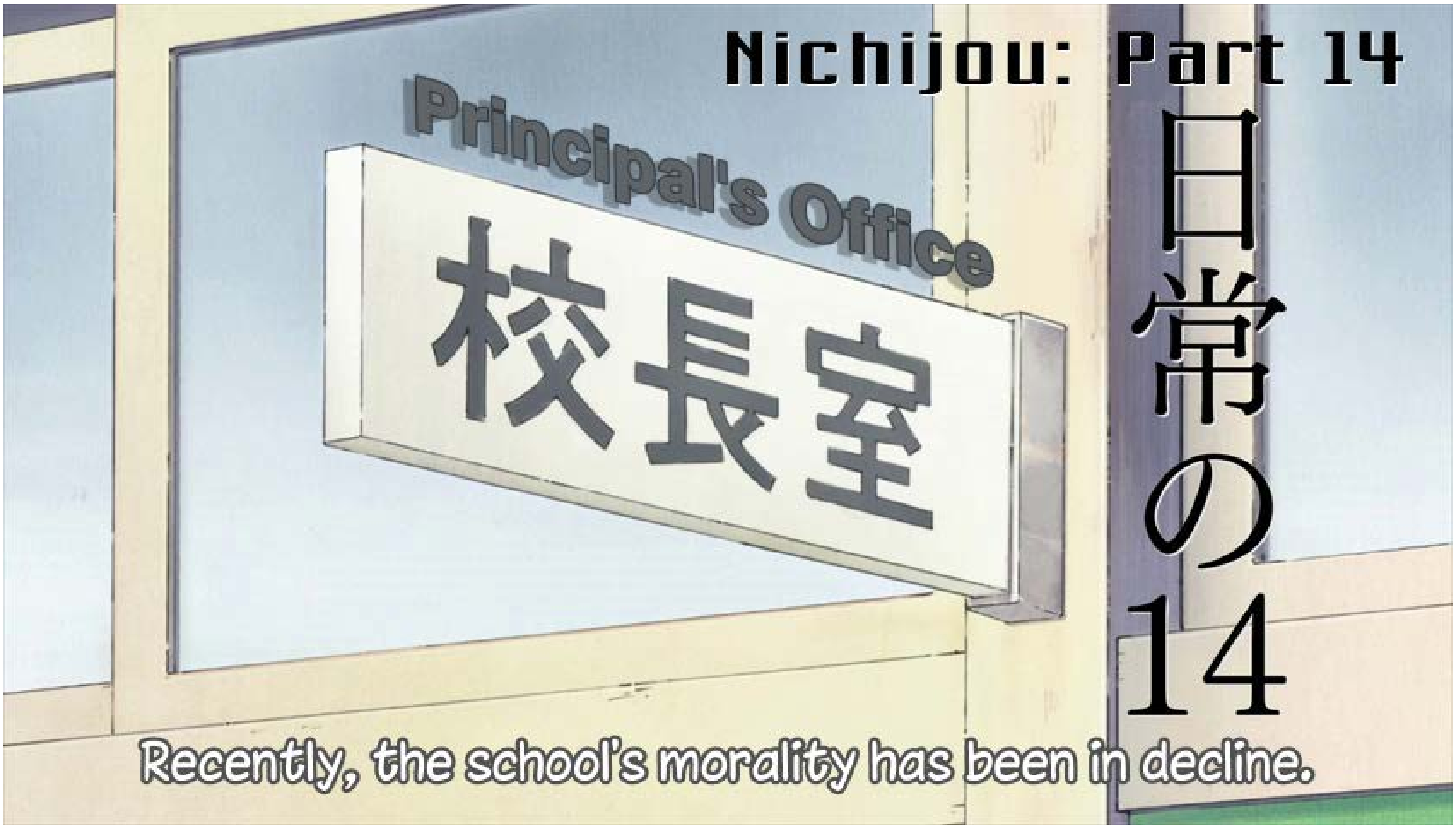
Here it pretty much ignores the JP sign.
Instead it's in the middle of the "background" area - the sky - where it doesn't interfere with the foreground.
It's kind of where you'd put it in the first place, if the JP one wasn't there at all...
pretty much the most natural place for a title on this screen, assuming the title is horizontal.



If the JP is in the middle and you can't exactly fit the typeset in the middle as well, you may split it like this (if the words allow it).



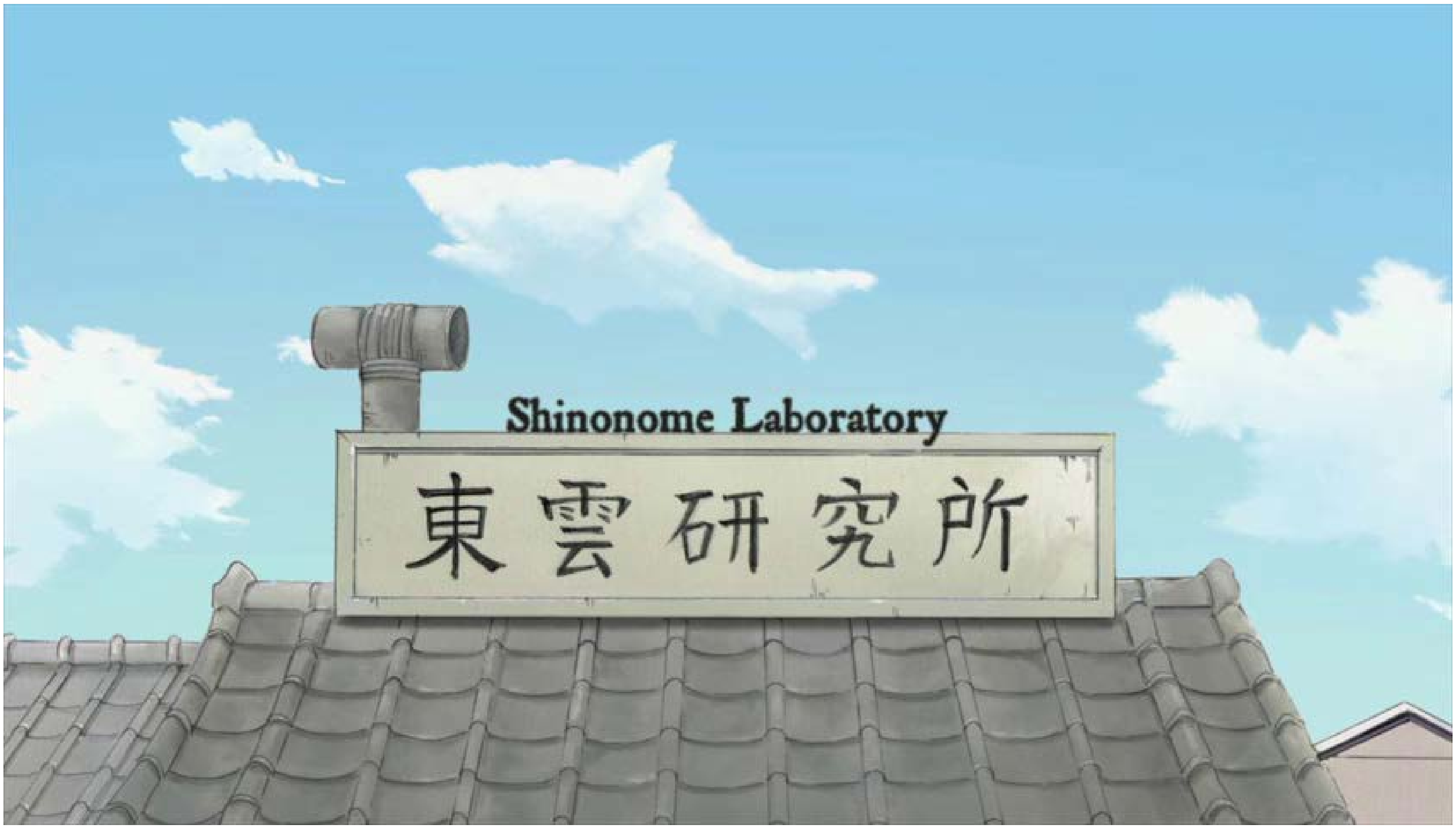
This should be pretty obvious. You don't want the sign over Yukko's head, and you don't want it over the papers above either.
The neutral green area is the most suitable place. If you split the title in 2 lines, you could put it in the large green area on the left.



Recently, the school's morality has been in decline.

Here you have 2 typesets and main dialogue, so first of all you want to avoid any of them overlapping.
The title is where it is pretty much out of necessity. There's hardly any other place suitable for it.
The office sign... many would try to fit it **on** the white board, along with the japanese.

That can certainly be done, but you'll have to have **really** small font size and it'll still look cramped.
So I put it above, and matched the width of the sign and thickness of the letters roughly.
It looked too artificial without that shadow, so that was added to give it some sense of space, even though as a "shadow" it's illogical.

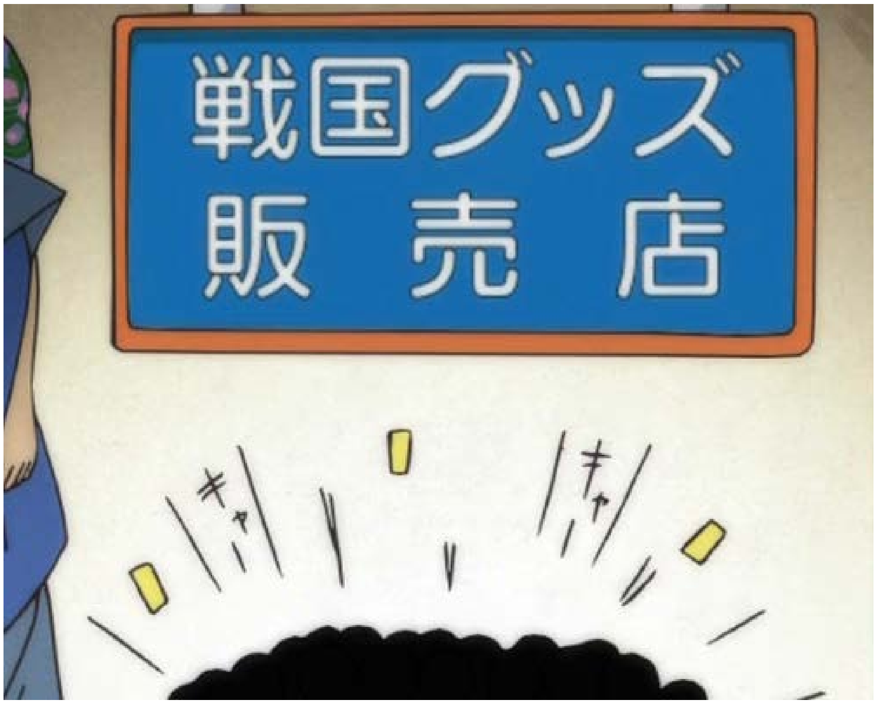


Here again trying to squeeze the Shino Labs **on** the signboard would be frustratingly difficult & just wouldn't look good no matter what.

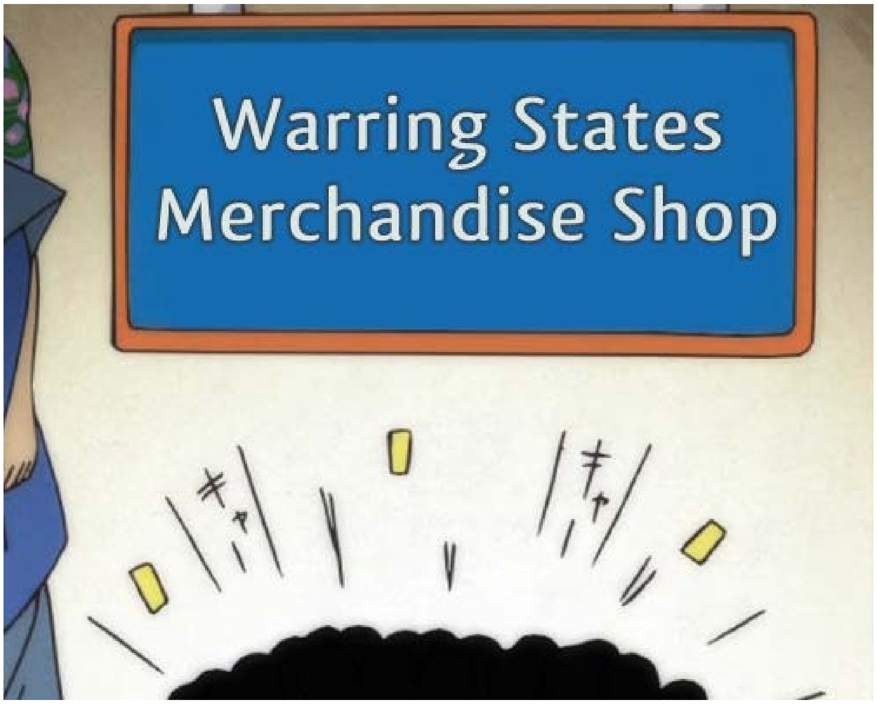
On an unrelated note, did you know sharks can fly?

Anyway...

One other option is to simply replace the JP sign with the English one.
That will, however, only work well if the background is one solid colour.
Here's one where it will be simple enough:

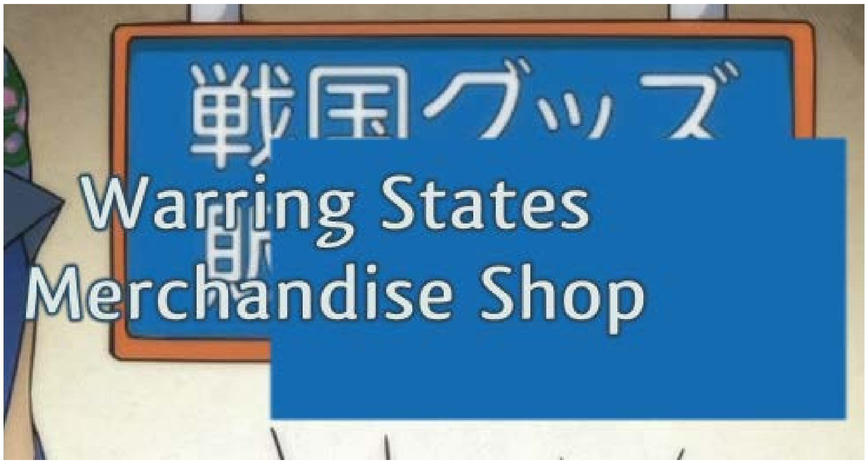


You'll be creating 2 layers. One will just draw a blue rectangle over the kanji.
The other will put the English text over it.
Normally I make the actual sign first. Then I duplicate it and delete content.
Then paste this instead: {\p1}m 0 0 l 100 0 100 100 0 100{\p0}
2018 Note: No. Make the sign. Use **Masquerade**. One click.
That's a basic square in drawing mode. Nuke border/shadow if present.
Expand it using the scaling tool (\fscx\fscy) to the size needed to cover the kanji.
Add \blur1 to make the edges softer to prevent them from being noticeable.
Match the blue colour and adjust layers so that text is on top of the rectangle.
The result will look like this:



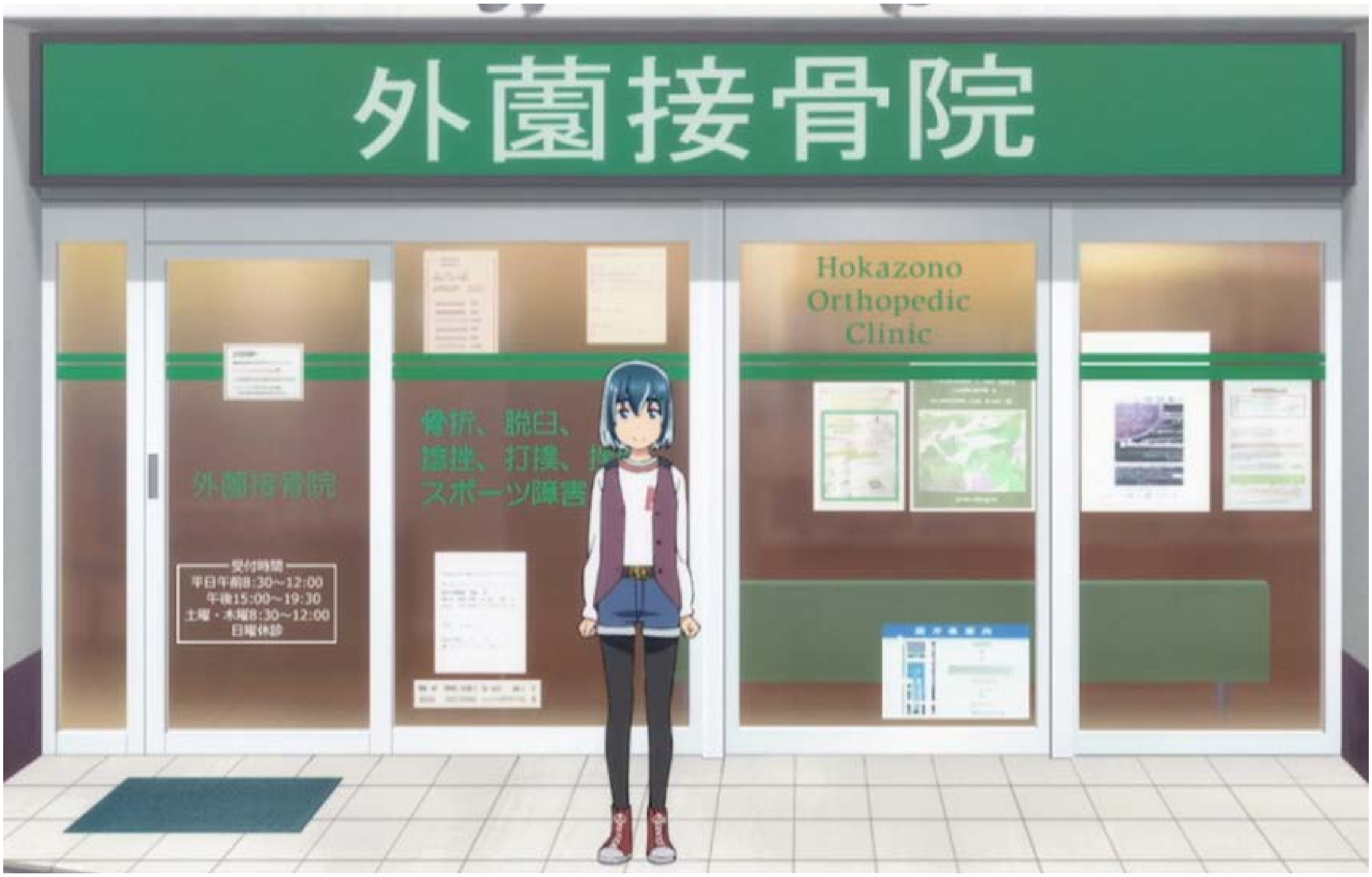
(i.e. like shit because text has no layers for blur)

And just in case this was difficult to comprehend, here it's disassembled:

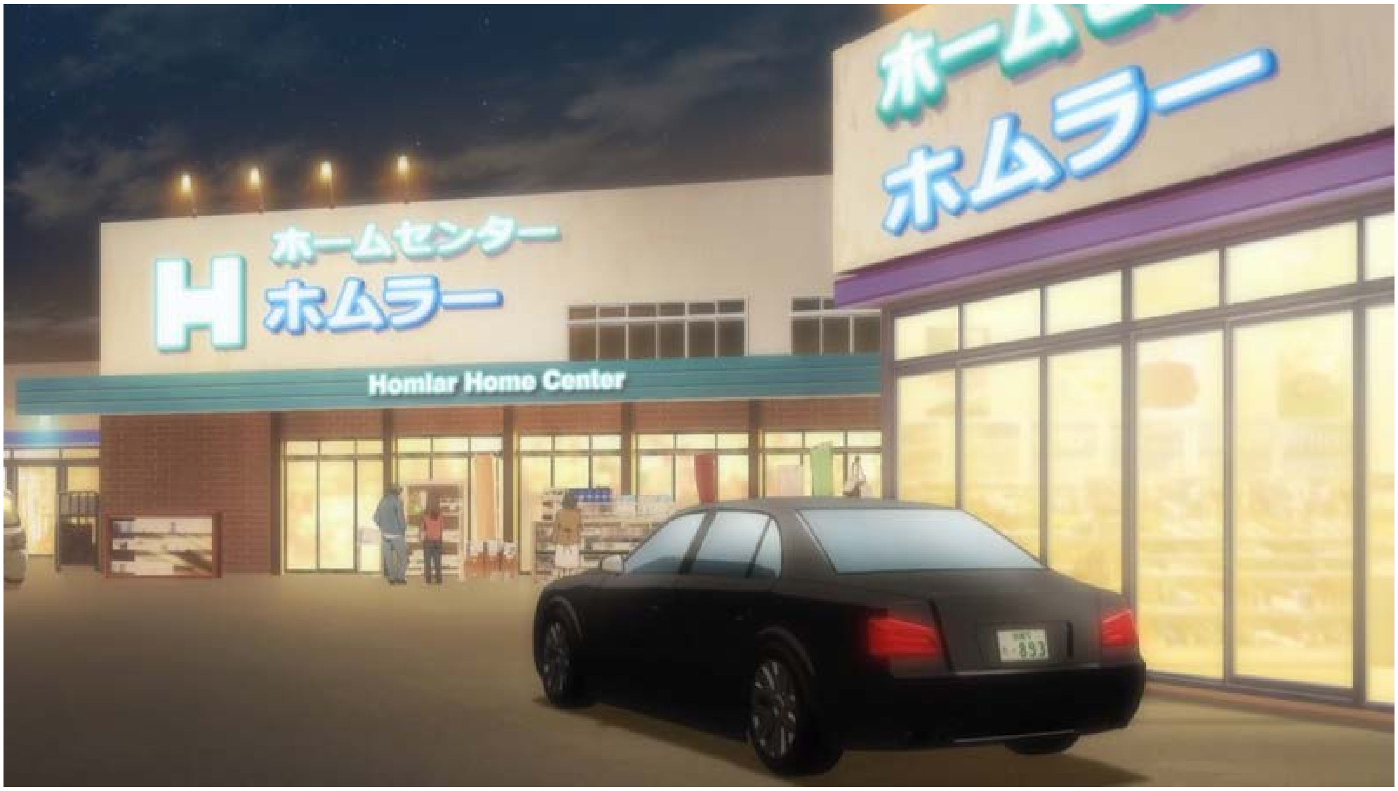


You could create more complex shapes than a rectangle.
More on drawing and other stuff in **this section**.

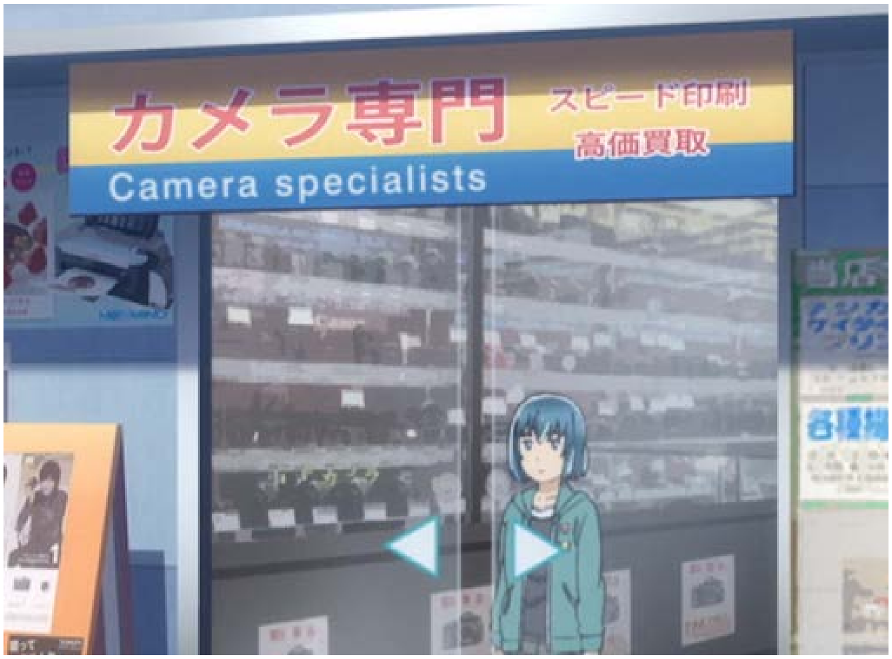
Here I want to show how the sign doesn't always have to be in the most obvious place, right next to or under the Japanese.



The sign is for the white letters at the top, obviously, but what to do with that? Doesn't fit under, and while there's space on either side, putting the text on one side will make it asymmetric and awkward, and putting half on each side is gonna look weird too, especially since I have three words. So instead, I look around and see what else can be done. The glass has some posters and inscriptions on it already, and there's enough space. They already have green letters on some of the glass, so it won't look too strange if I add some more.
When you glance at the picture, nothing looks out of place, but the sign is there, easy enough to read.

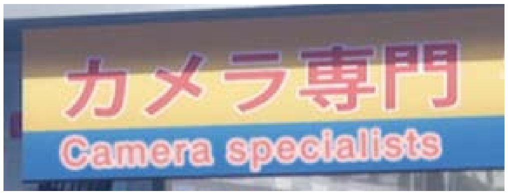


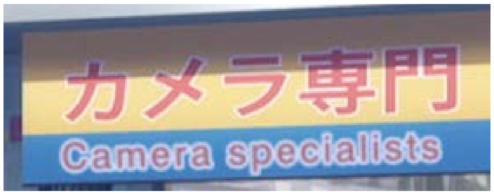
Here you have enough space and even a few choices for where the sign could obviously go.
But I tried to typeset it at the wall above, and it just didn't look too good. It looked like nobody would really put the sign there.
So I went for "a different spot from where the Japanese is" but one that might actually work in real life.
Enough space there too, easy to do, and I think it looks pretty natural.



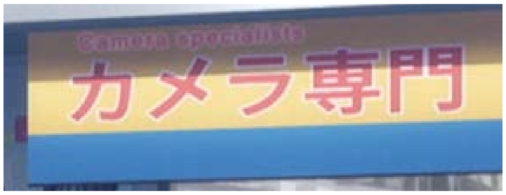
So the sign is red with white border, right? So you should probably do that?
Well, maybe, but it's also on yellow background, and trying the same on blue will not look the same.
You could use the space above the Japanese, but the space below is larger,
so it would feel strange that somebody used the small space and ignored the larger one.
It's nothing strange that different parts of a sign are styled differently,
so I just wanted to do whatever would look OK on that easily available blue stripe.

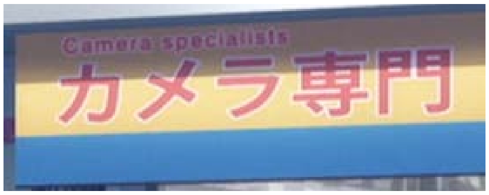
Just for the sake of comparison, I tried the more obvious choices.





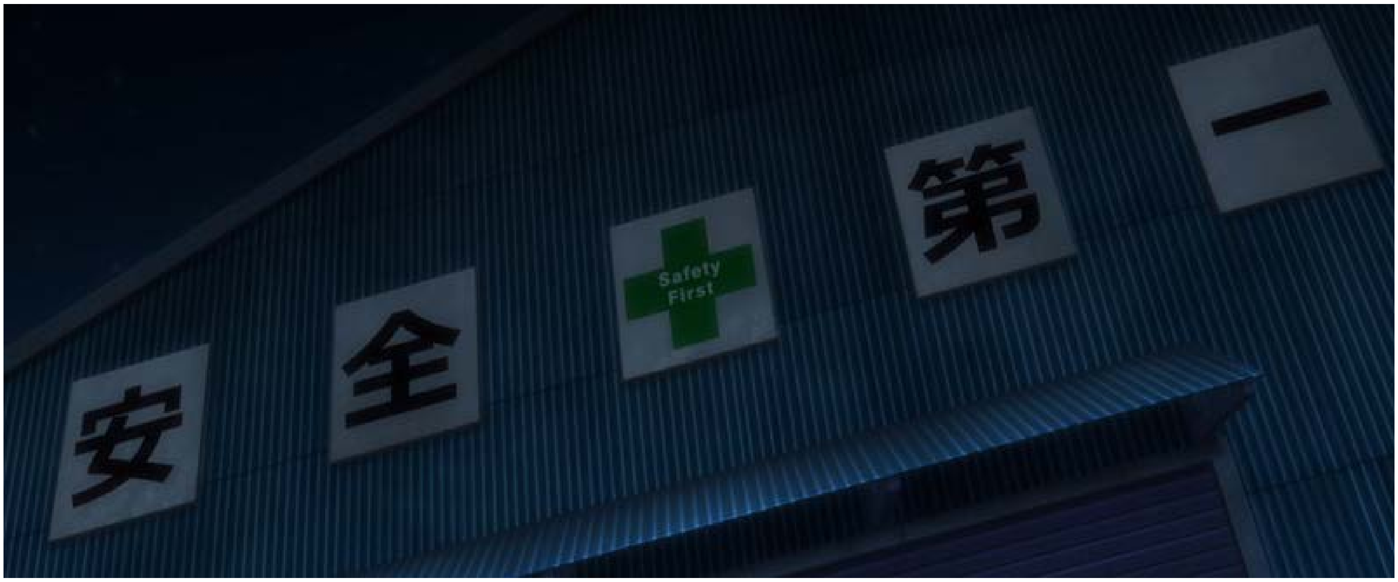
Matching colours, but staying in the blue area... and tweaking the colours a bit by eye.
Doing the same stuff on different background usually doesn't work too well.





Now at the top, matching the colours first... and tweaking them by eye again.
The first one is fucking horrible, even though it "matches" the Japanese the most of all these.
(Thought it doesn't even look like it matches, but that's the colours from colour picker.)
The second one blends in better, but it has several problems.
1. It's barely readable, especially compared to what I did.
2. It spans over slightly more than half of the Japanese, which just looks dumb.
3. It uses a relatively small area in the corner while there's plenty of space elsewhere.

I'll leave it to you to decide which one of the five looks the least terrible.



This is another example where I quickly decided that trying to put the TS close to the Japanese is never gonna work.
Once you make that call, then pretty much anything goes. Instead of "matching stuff", it's more about:
1. making it visible and readable enough
2. doing something that could plausibly happen in real life
(3. choosing something that won't take you an hour to do)

I suppose you could do something on the striped area around, but that would go against point 2,
unless you do something really magnificent, which will probably go against point 3.
Some typesetters would try to replicate the grey blocks and maybe place one above the green cross,
but again, that wouldn't happen irl, and the blocks are like 120 shades of grey and not even in a gradient manner.
I thought about putting the words above and below the "hyphen" on the right, but there's that weird asymmetry again.
So I chose pretty much the most symmetric solution possible that also happened to be really easy.
As you saw on the examples with the camera above, the eyedropper doesn't always work well,
so just getting the grey from the block probably won't be that great here either.
You have to tweak it by eye, but it should be clear that the grey basically needs a tint of green, because it's on green background.

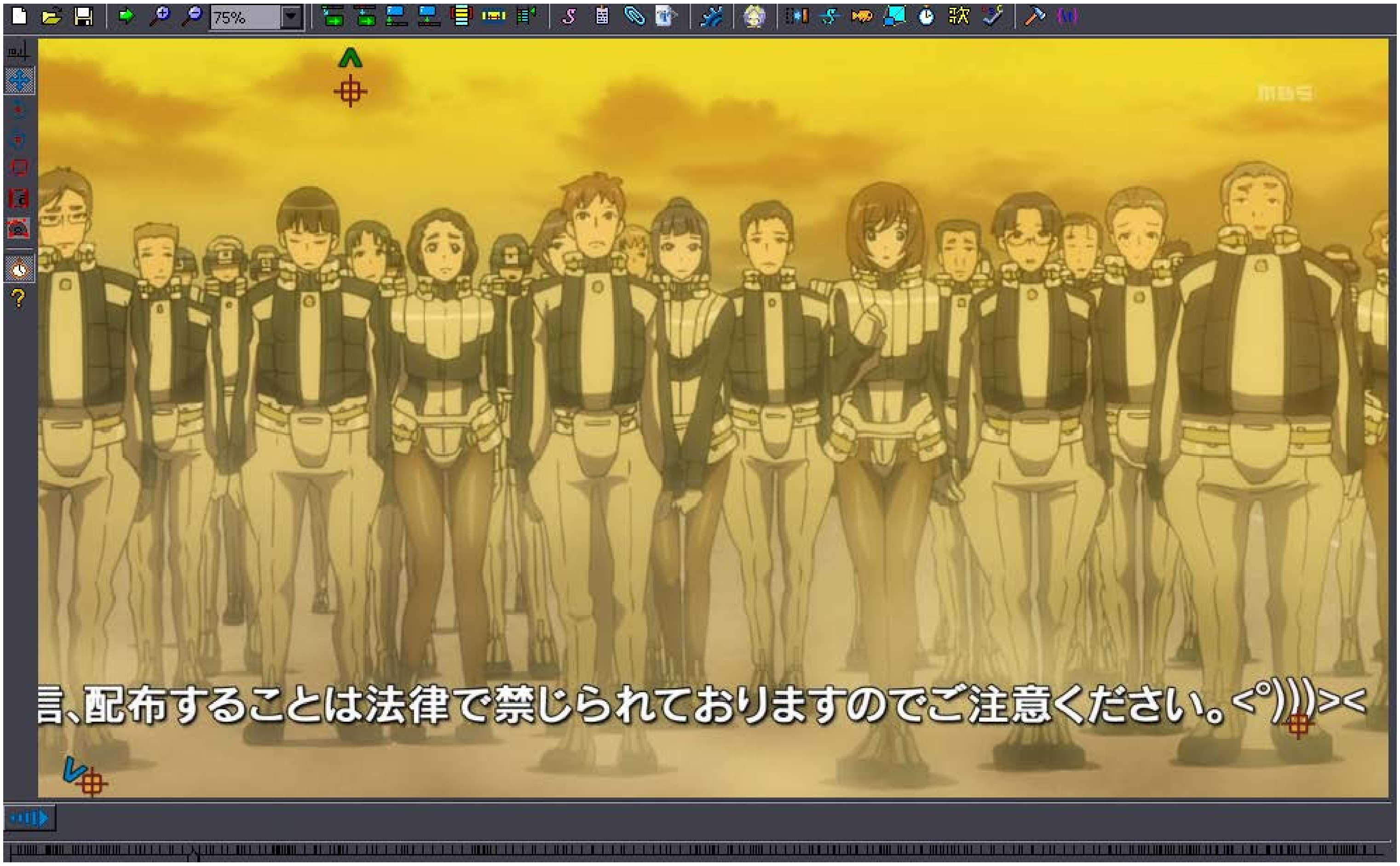
From the typesetter's point of view, this is a good sign because:
1. it looks OK
2. nobody really gives a shit about where the sign is as long as they can read it
3. you've only wasted 2 minutes tops of your life on this (well, aside from the 5 minutes spent thinking about what to do)

« Back to Typesetting Main

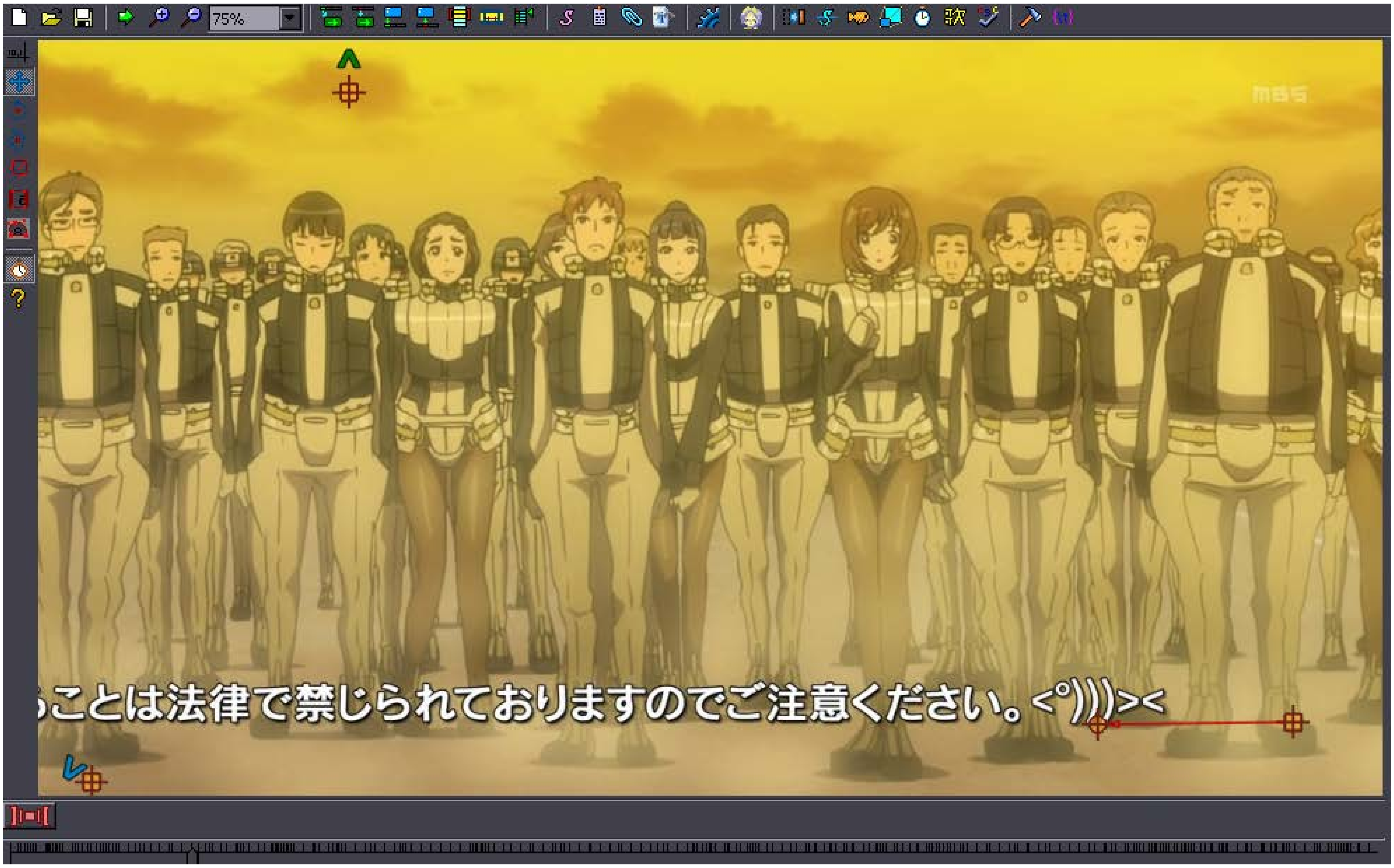
Typesetting: Moving & Animated Signs

There are several kinds of moving signs.
Ones that are moving in a constant linear fashion, ones that are accelerating/decelerating, and ones that do various other things - rotate, shake, etc.

Linear movement is simple. You use \move(x1,y1, x2, y2).
Let's say you want that fish on the bottom right follow the kanji moving to the left.


75%
言、配布することは法律で禁じられておりますのでご注意ください。<°)))><

First you use the Drag tool and position the fish in the first frame. Then you click the blue arrow that the other blue arrow is pointing at.
That switches from \pos to \move. Then you click on the right green arrow the other green arrow is pointing at. That gets you to the last frame of this fish.
That is assuming you've timed your fish correctly. If you haven't, then you're dumb, because what are you gonna do with a timeless fish? Anyway...
When on the last frame, you grab the circle that appeared over the square (on the fish's belly) and drag the fish to where it's supposed to have swum.

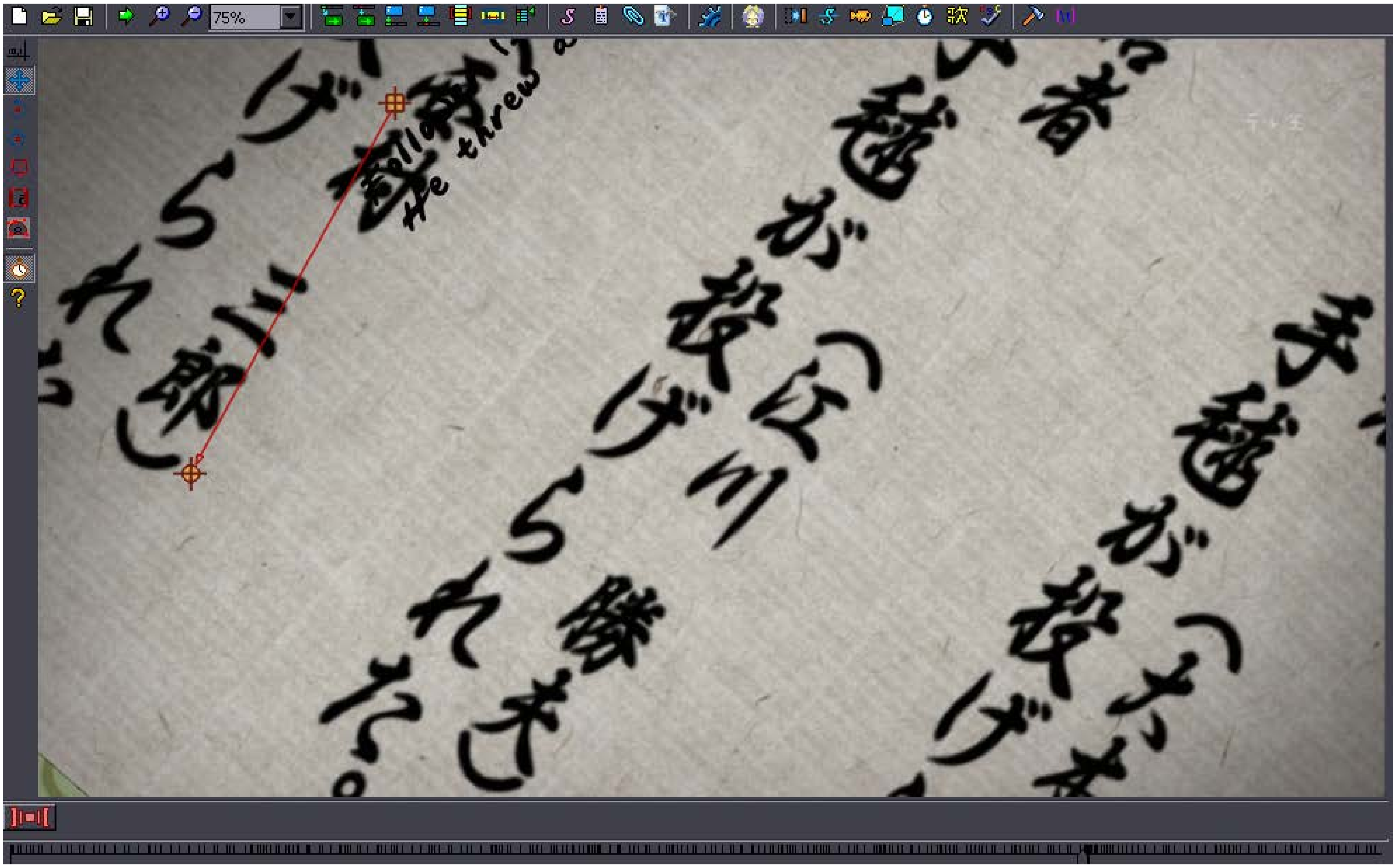


Like there^. You'll get a tag like this: {\move(1195,650,1009,652,0,799)}<°)))><
Now when you see the vertical coordinates are 650 and 652, you did it wrong.
The fish is supposed to be swimming just horizontally, not up and down, so the coordinates have to be the same. So you'll correct whichever one is wrong.

The last 2 numbers are the start frame and end frame timecodes. They are useful if the movement occurs only over a part of the sign's duration.
For example if the fish changes its mind in the middle and stops swimming, the timecodes will be "0,400".
If you're using the whole duration, then I suggest you remove those last 2 numbers once you've done the positioning,
since they tend to make the fish 'slow down' at the last frame. It may throw the positioning off a bit, so you'll correct it by typing,
because if you use the tool again, it will add those numbers again, and you'd be chasing your tail like... a fish? (or whatever animal does that, I dunno)
[As far as I can tell those last 2 numbers don't seem to cause any such issues in Aegisub 3.0. I got used to leaving them in and everything works fine.]
2015 Note: Actually things will be wrong if you nuke them. (It's a bit complicated, but the sign on the first frame isn't really at "0".)
If the movement doesn't cover the whole duration of the sign, then whatever frame you click the square on sets the start time,
and whatever frame you click the circle on sets the end time. Again you can adjust that by typing if needed.
2015 Note: In later versions, clicking isn't enough. You have to actually move it.

Speaking of Aegisub 3.0, here's a trick for making the movement precise [doesn't work in 2.1.9]:
First place the sign next to something that makes it easy to define the position precisely. For example make the fish's nose touch a corner of a letter.
Or you could overlay the fish's eye with the circle/period at the end of the moving text. Then go to the last frame and do the same there.
The point is to choose a reference point where you can easily tell it's positioned precisely the same on the last frame as on the first.
Then go back to first frame, switch to standard mode [above the drag tool on the left], and double click somewhere on the screen.
This will place the starting point of the fish there, while keeping the movement the same [keeping the distance and direction].
Then just click a few times until you get it positioned exactly where you need it.

Make sure to check the video once you're finished, so that you don't end up like dickpants with the sign starting god knows where
and moving in the opposite direction than it was supposed to...



If you haven't done anything incredibly dumb like this, you'll still check if the fish is swimming at the right speed.
If it seems like it's getting closer to the kanji, then you change the ending X coordinate to higher value [ie. more to the right]. If fast then other way round.

2018 Note: There's a function called "clip2move" in **Relocator**. You can use a clip to mark the start and end point for a move. So with that fish thing above, you'd mark the circle at the end of the kanji on first frame and last frame with 2 points of the clip, and the move coordinates will be derived from the clip and current \pos.

OK, so... that was the easy part.
The trouble comes when the movement is not constant / linear. If that happens, you have several options.

1. Quit fansubbing.
2. \an8
3. Ignore the inconsistencies and use linear movement even if it doesn't match.
4. Use \move but split it into several phases to decrease the inconsistencies to minimum.
5. Do it right, ie. frame by frame, either using a tracking software (yes) or by hand (no).

ad 1. Good Bye

ad 2. Nope. See point 1.

ad 3. This may be ok when the actual movement is not too far from a linear one and there won't be significant inconsistencies.
For example if the movement is generally linear but a bit twitchy. Otherwise this option would be pretty dumb.

ad 4. Nope. Don't do this. Don't even read this. Skip to 5. ~~This is kind of the middle way. The more phases you split it into, the better it'll look, so it's just up to you and how much time you wanna spend on it.~~
~~This will mostly be useful for movement that is pretty much linear in direction, but accelerates/decelerates.~~
~~What you do is duplicate the line a few times, and time all the lines to make a sequence. So if it's 5 seconds, you can split into 5 lines,~~
~~and time them (in seconds) 00-01, 01-02, etc. In reality you will need like 2-3 segments per second to make it look somewhat decent.~~
~~Then you just use linear movement for each line, the next line always starting where the previous ended (or a little bit farther).~~

~~You'll get something like this:~~

~~Dialogue: 0,0:17:15.94,0:17:16.44,Default,,0000,0000,0000,,{\move(238,315,373,315)}text~~
~~Dialogue: 0,0:17:16.44,0:17:17.05,Default,,0000,0000,0000,,{\move(377,315,465,315)}text~~
~~Dialogue: 0,0:17:17.05,0:17:17.61,Default,,0000,0000,0000,,{\move(467,315,508,315)}text~~
~~Dialogue: 0,0:17:17.61,0:17:18.16,Default,,0000,0000,0000,,{\move(511,315,542,315)}text~~
~~Dialogue: 0,0:17:18.16,0:17:18.68,Default,,0000,0000,0000,,{\move(544,315,552,315)}text~~

[Note: It's 2013. We don't do this shit anymore. Use mocha.]

ad 5a - using tracking software. We use this thing called Mocha, and we now have a guide for using it **here.**
At this point this is pretty much a must.

ad 5b - by hand. This should rarely be needed, but there are times when mocha fails.
Instead of splitting into just several segments and using \move, you'll split it into as many segments as there are frames for the sign,

and you'll be changing the \pos coordinates for each frame. As the japanese save on animation, often 2-3 consecutive frames are the same, so sometimes you can have 2-3 frames per line instead of 1. If it's 2-3, then you'll need regular frame timing, if it's each frame, you can time the 1st line to 1 frame, right click on the line and select "Duplicate and shift by 1 frame" [or Ctrl+D].
This way you'll be getting consecutive frames, each timed to 1 frame, which is exactly what you need.
[If each 2-3 frames are the same, you can still do ctrl+D and then "Join (keep first)" the lines that are the same.]
After that you go through all the frames and adjust position with the Drag tool.

The interesting thing is that while this method produces the best results, it's not even difficult. Pretty much all you need to be able to do is time the lines and set \pos, which is really the very basics. The problem is that it's quite time consuming and somewhat tedious. And may kill your wrists.

One more thing to cover is signs that rotate, expand etc. You could still do this frame by frame, but usually there's a better option.
You can use the \t tag, which allows you to apply gradual change for specific tags. How it works is described **here.**
If you need the sign to spin 360 degrees, you'll use \t(\frz360). If you need it to spin twice, it's \t(\frz720).
To spin and stop after 1 second it's \t(0,1000,\frz360). To do the same but start spinning slowly and accelerate it's \t(0,1000,3,\frz360).
So you have \t(start time, end time, acceleration,\tag1\tag2\tag3...) as explained in the link.
If you need it to rotate around a different point, you'll use \org for that (described in **Aligning Signs**).
You can combine this with \move, so it can be moving and spinning.

Other tags you can use with \t are \fscx \fscy \fsp \fs \blur and a few others, including colours. It doesn't work with everything, like \pos \org etc.
Experiment with this to find out what works well and what doesn't. You can use several tags at once.
You can achieve all kinds of things with the \t tag. The main downside is that if you use too much of it, playback will lag. And it can lag A LOT. Nah.

A few more related things wil be explained further.

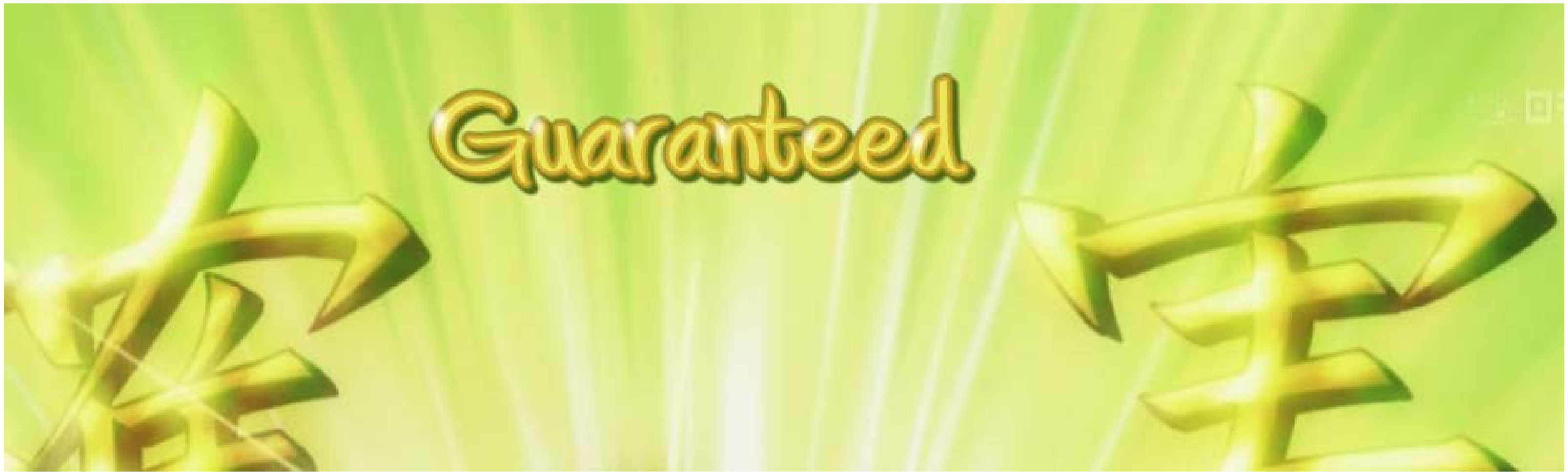
« Back to Typesetting Main

Typesetting: Using Layers

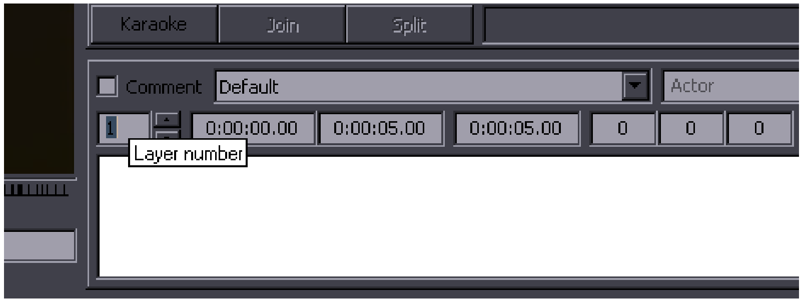
Probably one of the most useful tools, once you make it far enough that you're actually trying to make signs look nice, is layers.
You can use them for various effects. The two most important ones are:

1. You can have multiple borders/shadows
2. You can have blur between primary and outline colour, which makes signs look a LOT better

If you're inventive, you can always add some extra effects to that.



^ This sign here has 2 borders - orange and dark grey. Clearly you can't make it in one line.
So what you do is you make one sign, with orange border and no shadow, then duplicate it (right click on the line in script),
and change one of them to larger border in dark grey and add shadow.
For example you'll have \bord2 for the orange one and \bord4\shad2 for the dark grey. The important part is that the orange one has to be on top.
For that purpose there's this thing called layers.
You can change the layer number above the typing area, next to the start time:



So the orange one will be layer 1 and the other layer 0, or whatever other numbers but the orange one has to be higher.
You will also use this when regular dialogue subs overlap with signs! The dialogue has to be on top, obviously.

Some notes about this sign: The JP doesn't have an orange border. The primary colour ranges from yellow to orange. Choosing only one would make it look too plain and you can't use a texture instead.
(Well, you could probably imitate it somehow if you were inventive but this is about basics.)

I wanted to have both yellow and orange in there and this was the easiest way to do that. Also the font was pretty thin and this made it thicker.

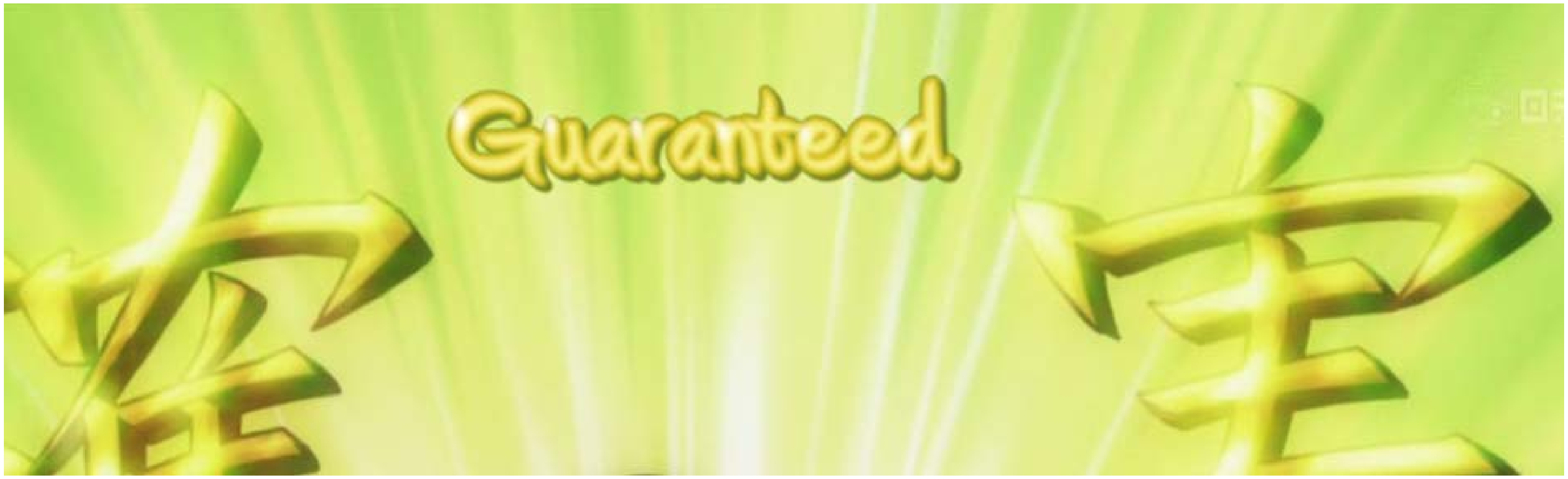
If you're looking carefully and actually paying attention, you have noticed the white "dots" on the sign.
The JP sign has light reflecting off of it (coming from top left), creating blurry white areas.
Since I like doing things that nobody else does, I wanted to try to imitate this somehow.
It's a kind of silly idea that nobody would bother even thinking of, much less trying to do it, but for me, work without creativity would be too boring.
Obviously this is nothing like real light effects, but I think considering the tools available it's not all that bad.
So how did I do this? Create a third line, layer 3, white colour, no border/shadow. Type some commas with a few spaces between them. Use a lot of blur.
Then just figure out the sizes and spaces and rotation etc. to make the dots fit into places you want them at (keeping the light from top left pattern). That's it.

This sign still has some flaws and could be better, but let's just say that if you have a script with 30 signs of varying difficulty and you need to sleep soon, you don't always do your best.

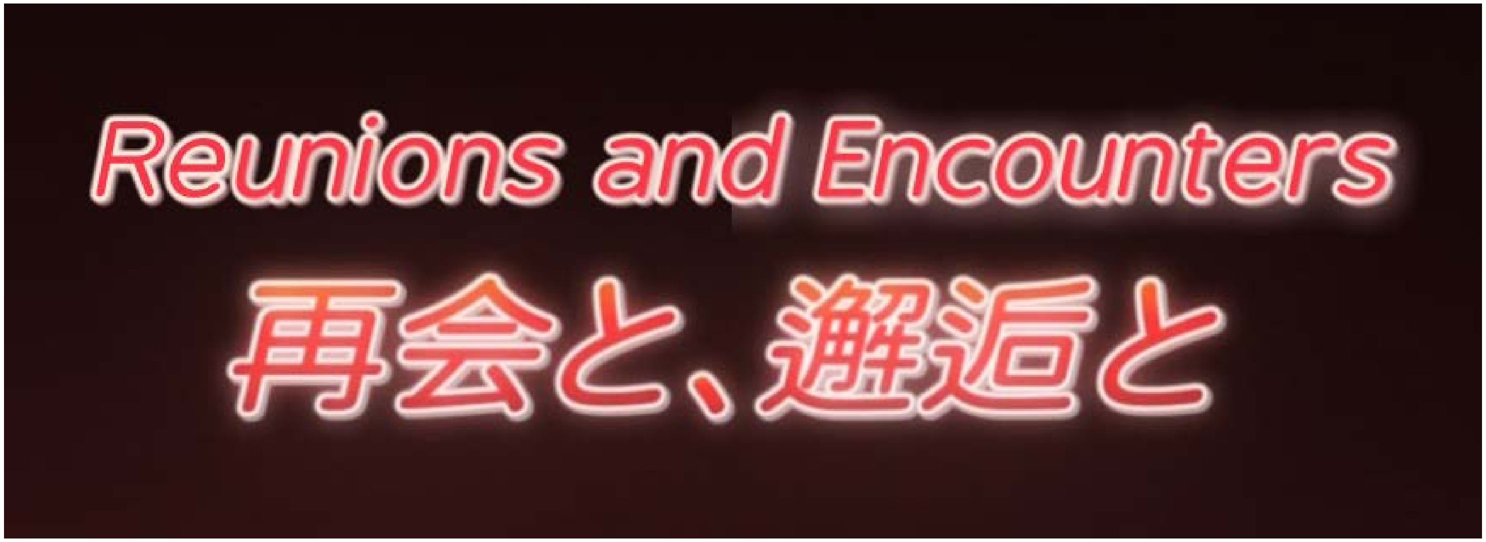
I use double layer signs quite often mostly for one specific reason. If you check the sign above, you'll see that despite the blur on the outside, the border between yellow and orange is sharp.
That's because when you use blur, it only blurs whatever's on the outside, ie. if you have an Outline, it won't blur the inner letters.
That is sometimes a problem because the Japanese signs are usually blurred on the inside as well, and the sharp edge between the inside and the border just looks bad.

The solution to that is to create another layer without the border and blur it as well.
In other words you duplicate the sign, make one layer 1 and add \bord0.

I recently redownloaded the video and fixed the colours and blur [keeping the font though]:

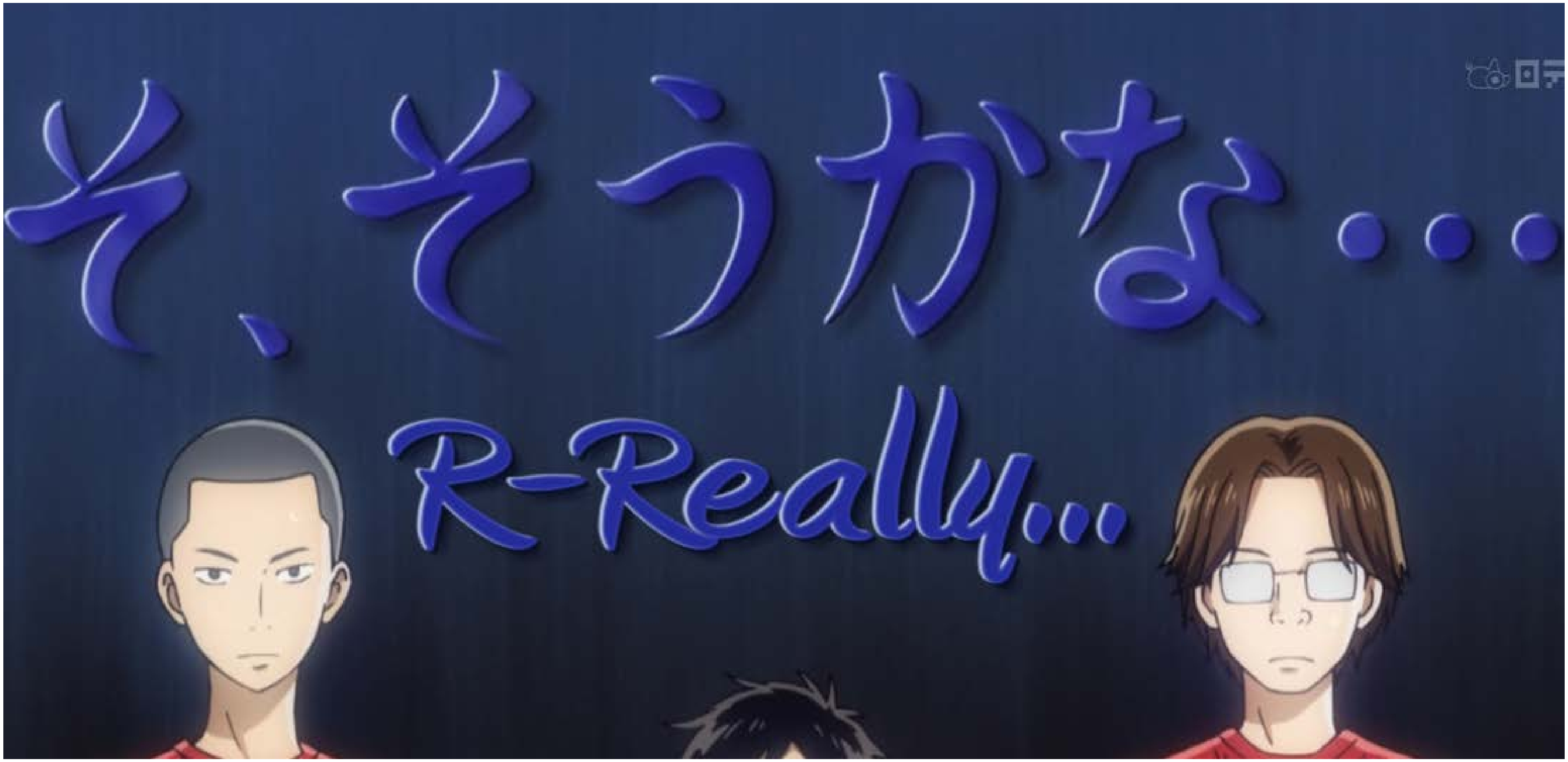


Here's another example:



The first 2 letters "Re" are only one layer, **without** the inner blur. The rest is 2 layers as described above. The red in the "Re" is sharp and compared to the kanji looks bad.
For illustration I also added one more layer to the right half of the sign, adding the soft, blurry, reddish background.
It's the lowest layer and uses reddish border with lots of blur. I didn't use this in the release, this is just to show that it can be done.
It's actually \blur12 and the sign is fading in and would have 3 layers, so it might lag.
Note 2018: These days you can throw around \blur20 without worrying about that.

This sign could look much better - I think I did this on a workraw, meaning small, blocky video with poor colours, so I couldn't see details very well.
Anyway, 12 episodes got released at this point and no one's complained so far, so I guess it's ok.
2015 Note: I think the inner blur is still wrong because \1a for the white layer is probably red. (Explained further below.)



This was quite a challenge (in 2011), for several reasons.
Blue letters, white edge on the left, black on the right, and blurry shadow behind it... what to do?
On top ot if, the letters were appearing one by one... but before we get to that, we have to deal with the basic design first.
Obvious things like font, size, position and main colour should be... obvious. If we want the white/black edges and the shadow, we'll need at least 3 layers.
You can probably figure out how to do the black edge on the right and the blurry shadow. The black edge is regular shadow, and the background is another layer with more blur.
But what about the white? Well, aside from needing another layer for that, there are some extra tags that will help us.
Besides \bord there's also \xbord and \ybord, and besides \shad there's \xshad and \yshad.
This allows you to extend the border differently vertically than horizontally, and make the shadow be cast in any direction.
Here I needed shadow (white) so i used \xshad-3\yshad-1, which positions the shadow 1 pixel above and 3 pixels to the left (yes, negative values to go up and left).
The whole thing looked like this:

```
layer 2: {\fad(0,600)\fs150\fax0.1\bord2\xshad-3\yshad-1\4a&H00&\blur0.8\pos(620,280)\3c&H9E362E&\4c&HE7C4B4&\c&H9A2B24&}R-Really...
layer 1: {\fad(0,600)\fs150\fax0.1\bord2\shad3\4a&H00&\blur1\pos(620,280)\c&H9B352E&\3c&H9E362E&\4c&H250F09&}R-Really...
layer 0: {\fad(0,600)\fs150\fax0.1\bord0\shad0\blur4\pos(628,288)\c&H230E0C&}R-Really...
```

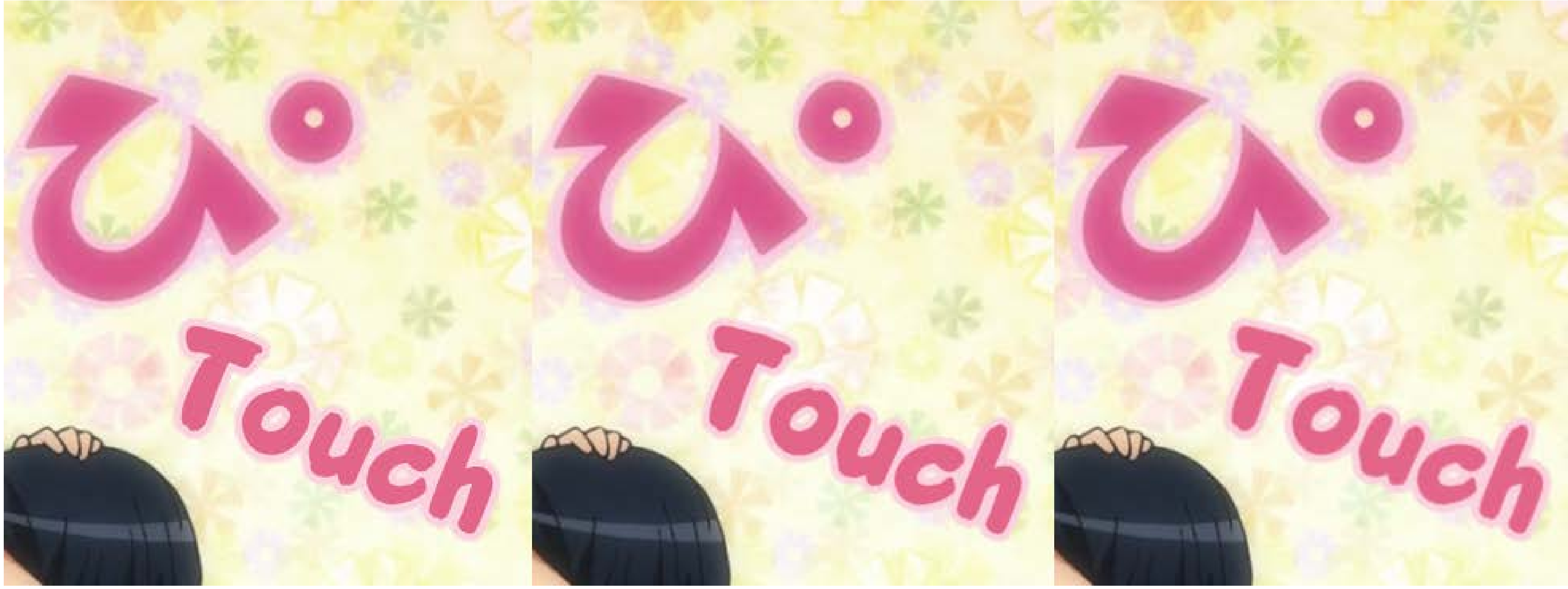
[\4a&H00& changes the shadow to opaque, because the default was partly transparent - it's a variation of the \alpha tag.]

Well, this was only the last part. As I said, it was appearing letter by letter. First I used \t and \clip to make it appear continuously.
Good thing was I still had only 3 lines in the script but it turned out it was lagging like hell. [Note: having 3 layers with \t on all of them ~~is~~ was quite likely to cause lag.]
So instead I had to go for alpha timing, making it appear letter by letter (almost, because the JP has fewer letters).
This was still pretty intense but worked out without lag. It was a bit more work, and I ended up with 18 lines for this sign, but it looked pretty cool.

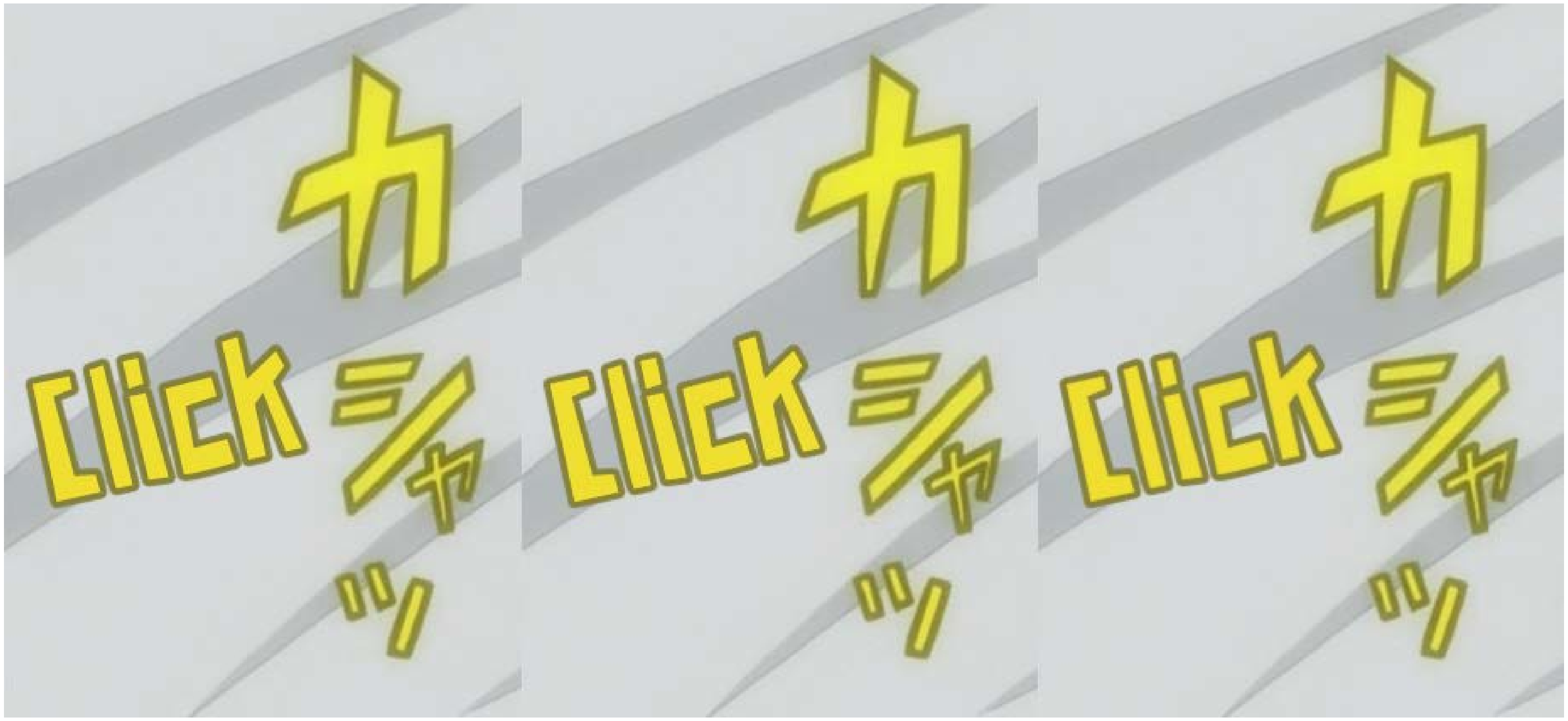
Signs with border and blur - how to do them right

After typesetting a dozen shows I realized that making 2 layers for all signs with a border is a pretty essential thing if you want the signs to look good.
It's pretty much one of the most important things that will distinguish a good typeset from a mediocre one, without much effort.
It's fairly simple and easy to do but it took me some time to figure out how to do it right.

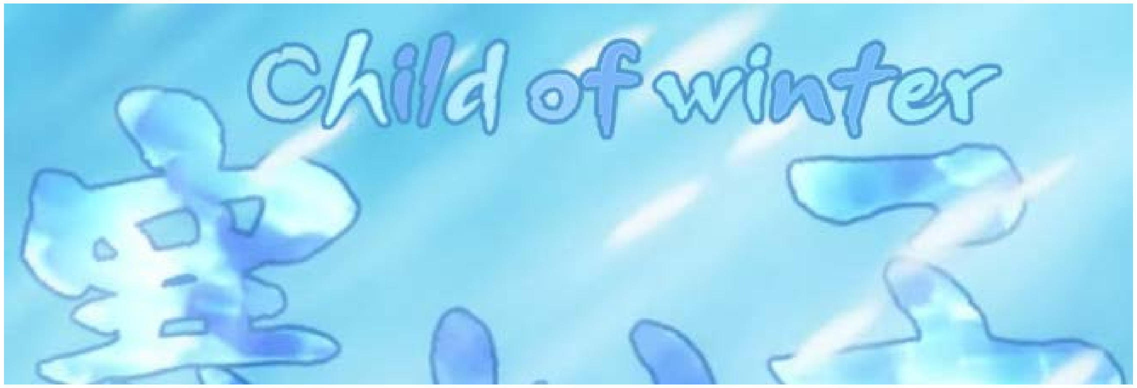
The first step was to just make 2 layers and nuke border on the top one [of course blur would be there from the beginning, so by duplicating the line it's on both].
Then I realized that there's a problem with that. Let's say you have black primary colour and white outline.
You make 2 layers, nuke the border on the top one, and have let's say \blur0.6. The result, however, will not be real blur0.6 on the black colour.
The blurred outline has by definition some transparency, which means that the sharp black border from the bottom layer is still partly noticeable.
How much will depend on the colours. Sometimes it looks pretty good; other times it's almost as sharp as it was in one line.



Here we have two examples (above & below). Left image is one line, middle is two lines as described above, right is the correct way [we'll get to that in a minute].
You can see the left ones are too sharp and the right ones look good.
The middle one for Touch is not bad, but the letters got a bit thicker. In the Click example, the middle looks just as bad as the left one.
So you see this is not working right.

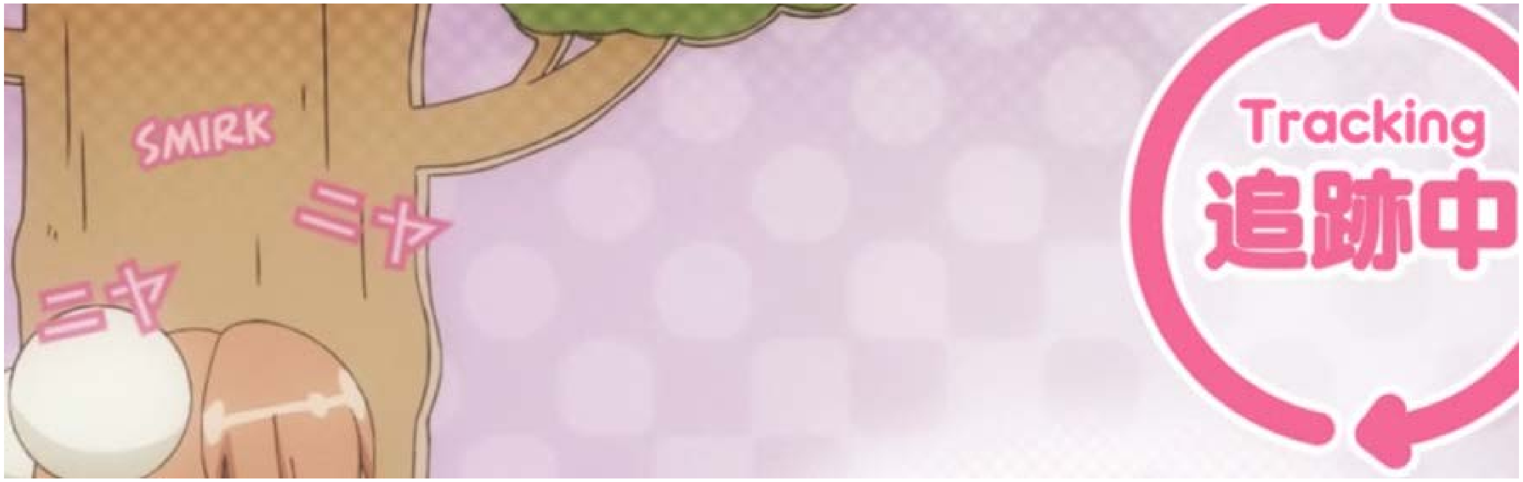


One idea that I had, and apparently others too, was to make the primary colour for the bottom layer transparent.
While that sounds reasonable, it doesn't really work either, because the inner edge of the outline is still sharp.
So again it will depend on colours used how that's gonna look. Here's an example:



You can see on the lighter colours it looks ok [because there's a light colour *behind* them],
while on the darker ones there's a thin white line between the primary and outline.
The blur on the top layer makes partial transparency just before the outline, and the bottom layer is fully transparent for the primary colour,
so you can see a bit of whatever the background colour is. If the bg happens to be similar to your primary colour, it may look ok.
But it's not something you can rely on.

So the final trick that makes it look good is to make the primary colour of the bottom layer **the same** as the outline.
You can see the result on the right part of the Touch and Click pictures, or on these two signs here:



While it may seem like a bother to make 2 layers for almost every sign, it's not really that much effort. (Yeah, one click.)
Especially since most of you reading this probably don't have more than 20 signs per episode most of the time.
I've worked on Maria Holic Alive and Acchi Kocchi [in case you can't tell], which means about 80 signs per episode and I make 2 layers for anything that has border.

Here's the routine to do it [except it's 2013 now and we use scripts for pretty much everything we do]:
1. type the needed blur and border to your line
2. use eyedropper to select primary and outline colour
...that's what you'd do anyway, now the layers...
3. duplicate the line [I use just ctrl+C/ctrl+V since it's probably the fastest]
4. change the top line to layer 1 and rewrite border to 0; hit Enter to get to the second line
5. double click on the tag for outline colour, ctrl+C, double click on tag for primary colour, ctrl+V

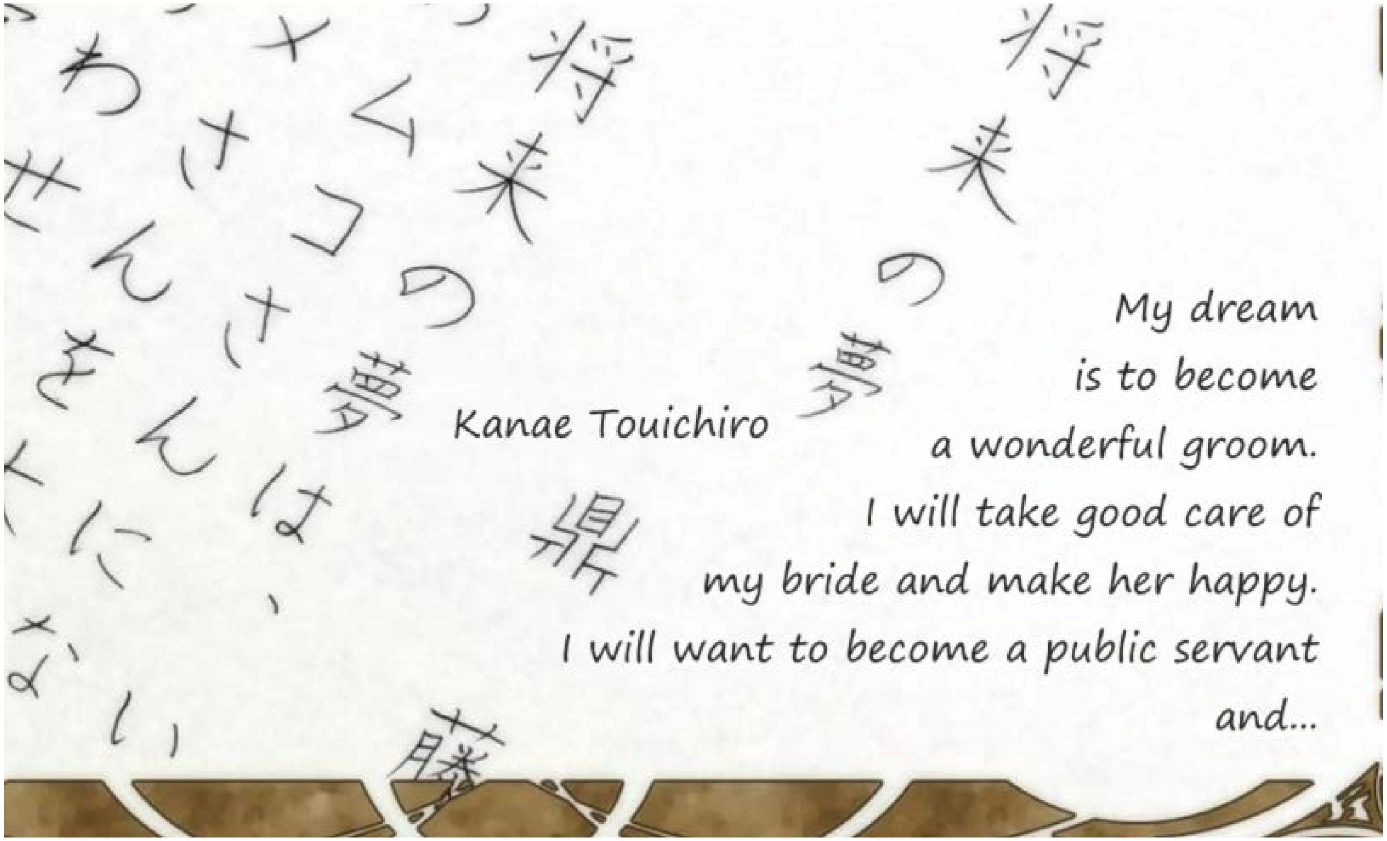
In other words, once you have 2 layers, you cange one to layer 1 and set \bord0, and in the other one you copy the colour from \3c to \c. It's pretty fast.

But in case you're really lazy, **here's lyger's script**.
I also made **my own version** with some different options, but it's less tested.

Each of them works differently and one may work better in specific situations than the other, but in most cases either one should be fine.

2015 Note: I don't know if anybody still uses lyger's script for this (I think it's unlikely), but 'Blur and Glow' is well tested and many times upgraded, so it should probably be your primary tool for layers.

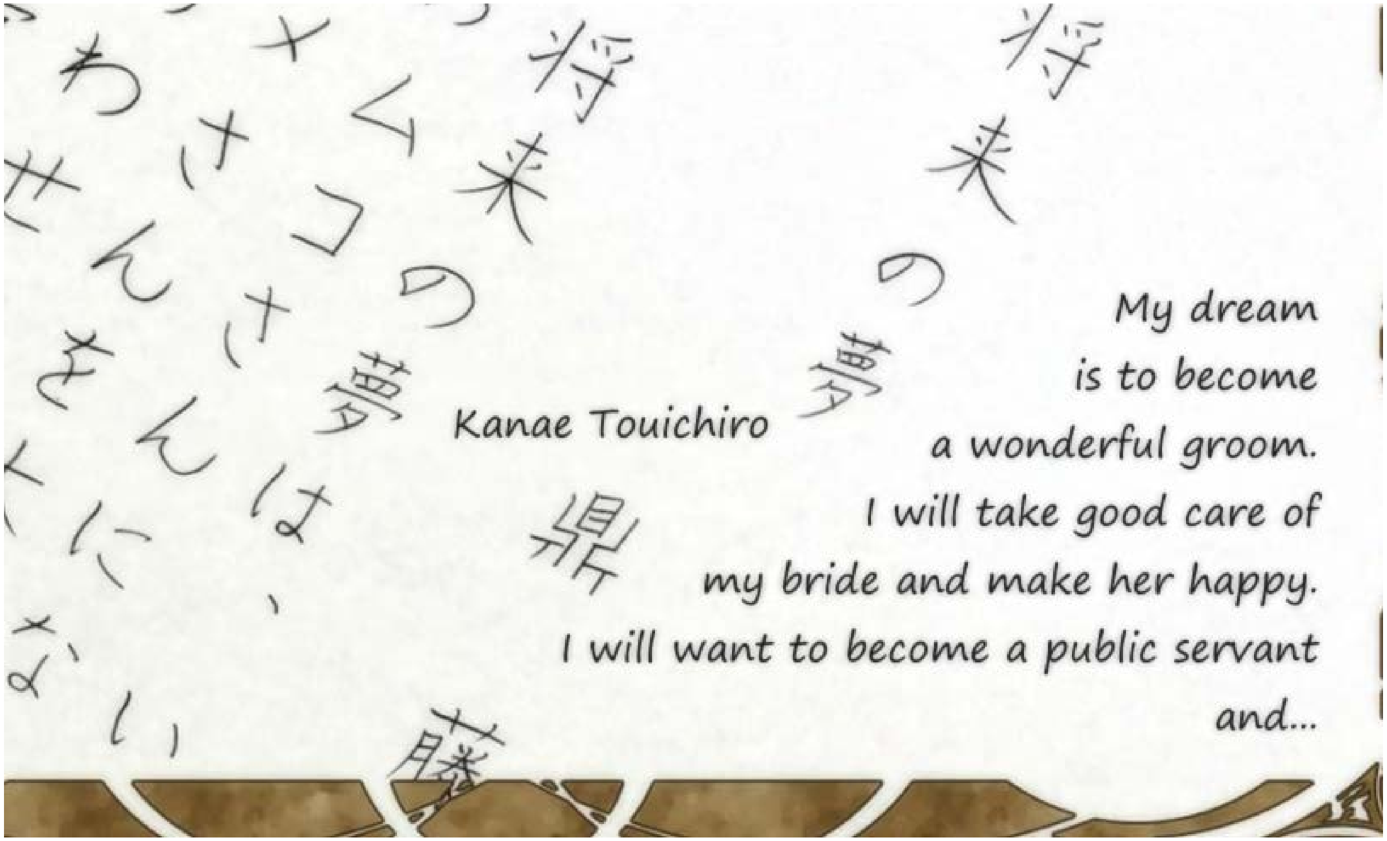
Another thing I do with layers is actually for signs that don't have a border [now included in my script as 'Glow'].
Sometimes you have signs that have this vague, hazy, blurry surrounding, like in this picture:



Kanae Touichiro

My dream
is to become
a wonderful groom.
I will take good care of
my bride and make her happy.
I will want to become a public servant
and...

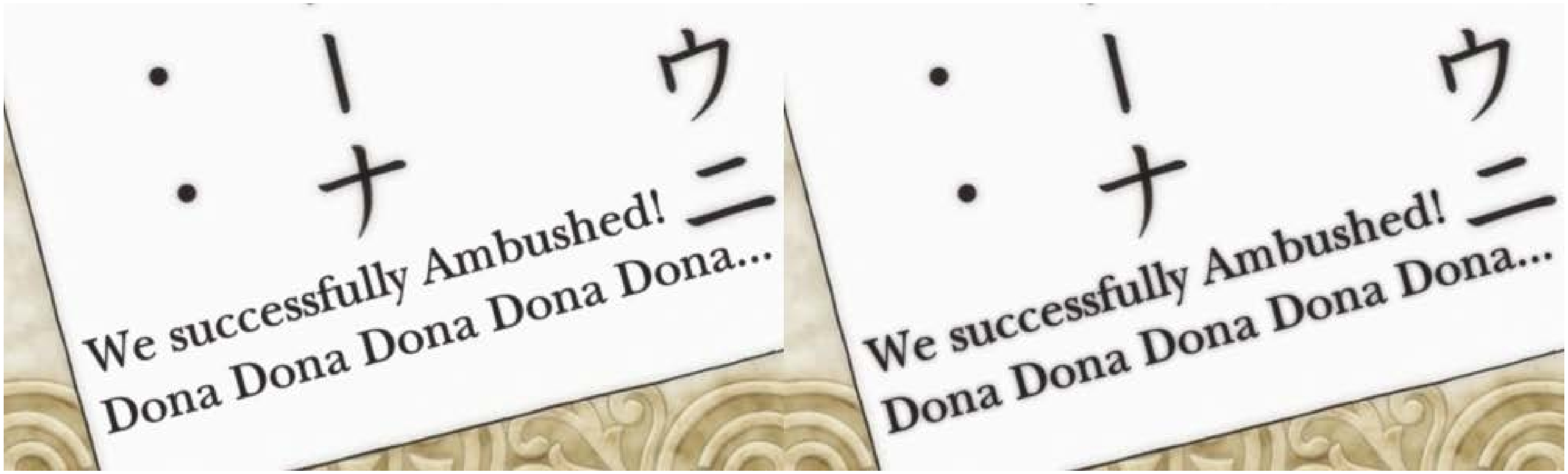
This is how you'd normally typeset it, and I'm sure you can see the difference. This is a regular blur of about 0.6.
If you use more blur, the letters will become unreadable and won't look like the original at all.
If you use a border and blur it a lot, it will be all kinds of messed up and look even worse.
So we need two layers again. However, I use both layers without border.
For the bottom layer, we need something in the range of \blur1 - \blur3. With a border you'd have problems...
If you use \blur3 with \bord1 or less, you'll get a sharp edge between primary and outline colour. Smoothness gone.
If you use \blur3 with \bord2 or more, it becomes unreadable because it makes the letters too thick.
The solution is to not use border, and use \blur3 for the bottom layer and \blur0.6-0.8 or so for the top.
This basically gives you a double blur and looks like this:



Kanae Touichiro

My dream
is to become
a wonderful groom.
I will take good care of
my bride and make her happy.
I will want to become a public servant
and...

At first glance it may not look like much of a difference, but if you switch between the views [commenting the bottom layer], you can see it clearly.
Again, how much better this looks will depend on the colours used, the background etc.
Here's another example:



We successfully Ambushed!
Dona Dona Dona Dona Dona...

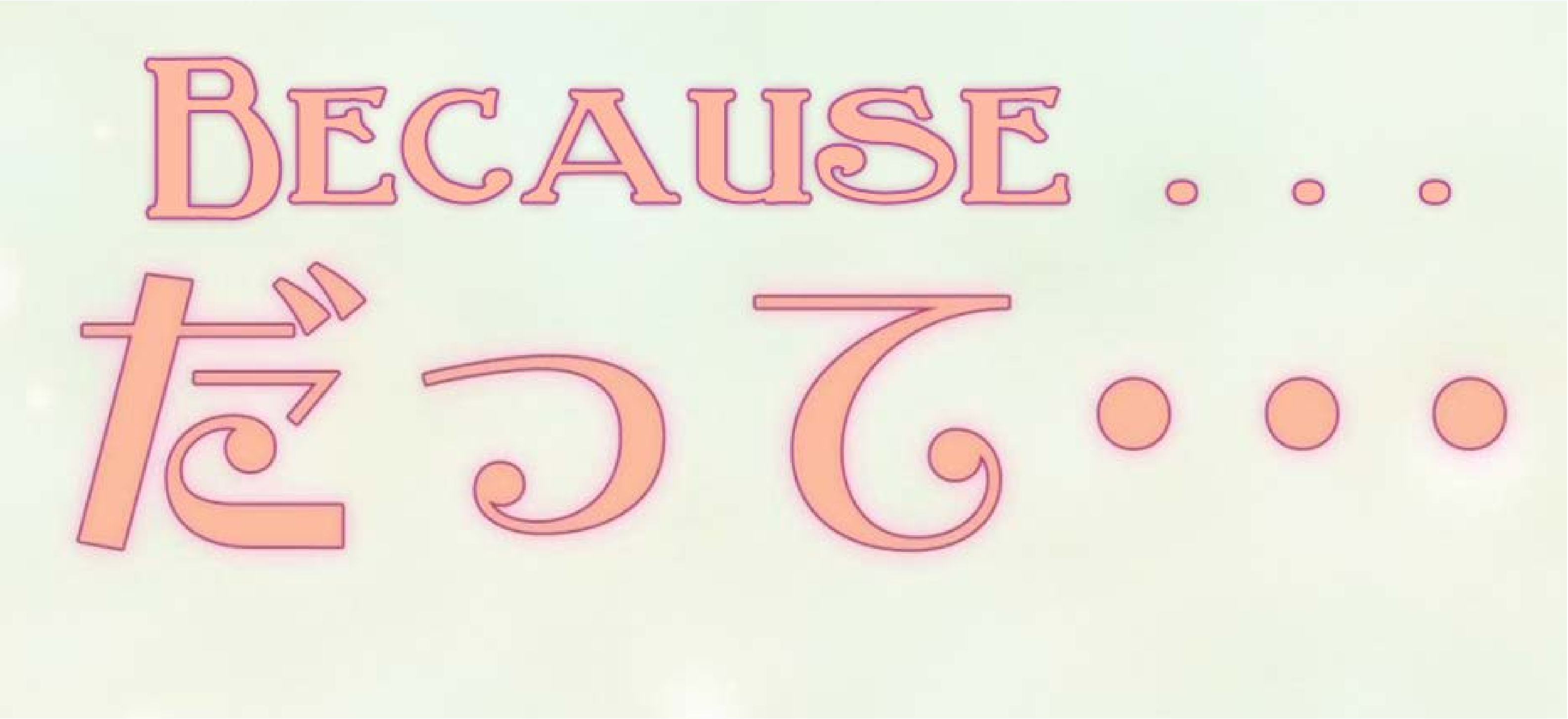
We successfully Ambushed!
Dona Dona Dona Dona Dona...

This may be slightly overdone, but at least you can see it more clearly. You can't do this in one layer, and you can't do it with border either.
The one thing I haven't mentioned yet is that you need to adjust the bottom layer not only with the value of blur, but also colour.
With the same colour as the top layer it will usually look too thick. So you just make the colour brighter/darker till you get the effect you need.

You could also do this using the alpha tag. With the eyedropper tool changing colour may be faster but in some cases alpha might look better.
For the colour you usually only need to change brightness, but sometimes the outline has a different hue as well.

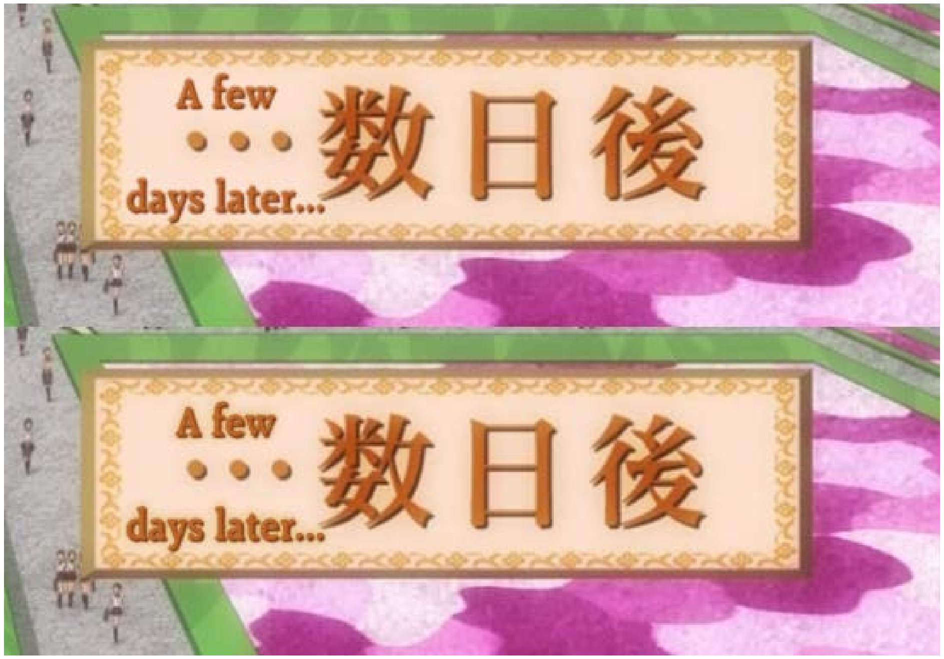
Note 2018: Since you're putting 2 layers on top of each other, the resulting colour for the body of the letters may get a bit too dark.
You can fix that by adding some alpha to the top layer as well. Just add slowly until it "looks" good.

Here's a more complex example with 3 layers. 2 for blurring the primary and outline, and 1 to add that soft haze around.

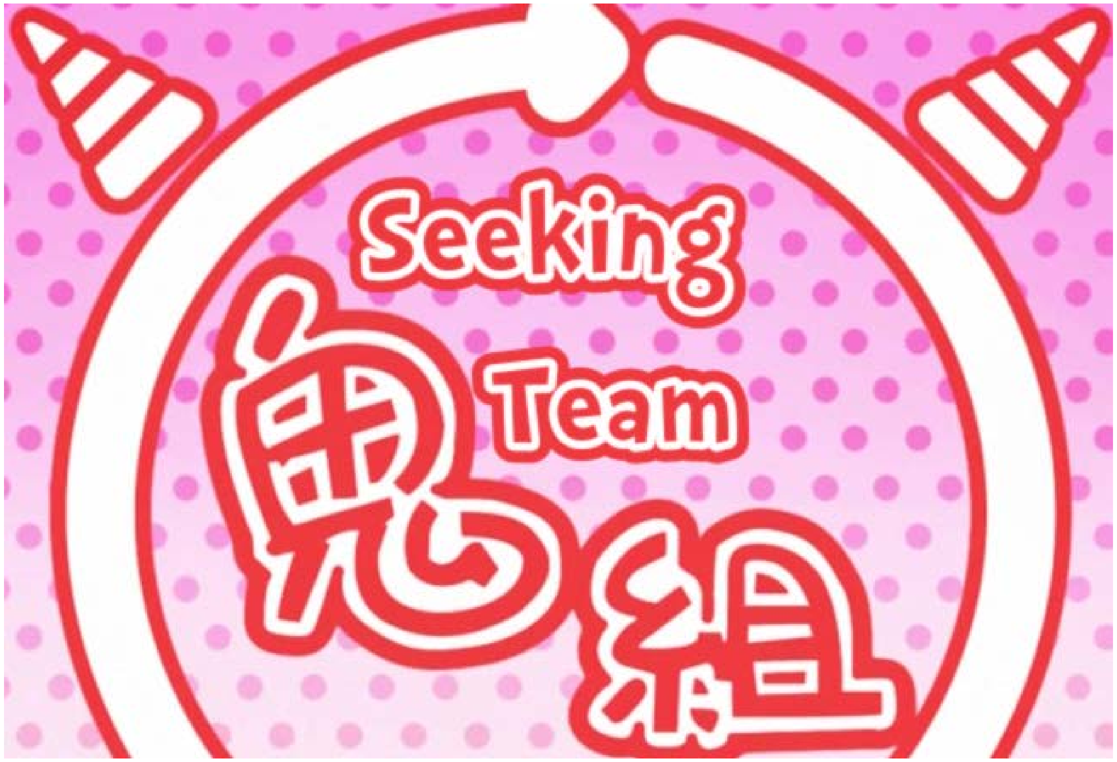


When you just look at this, you'll barely notice the 3rd layer. But when it's not there, you'll clearly notice that it's missing and the typeset will stand out.
Good typesetting is more about the viewer not noticing somethnig rather than noticing.

Another example:

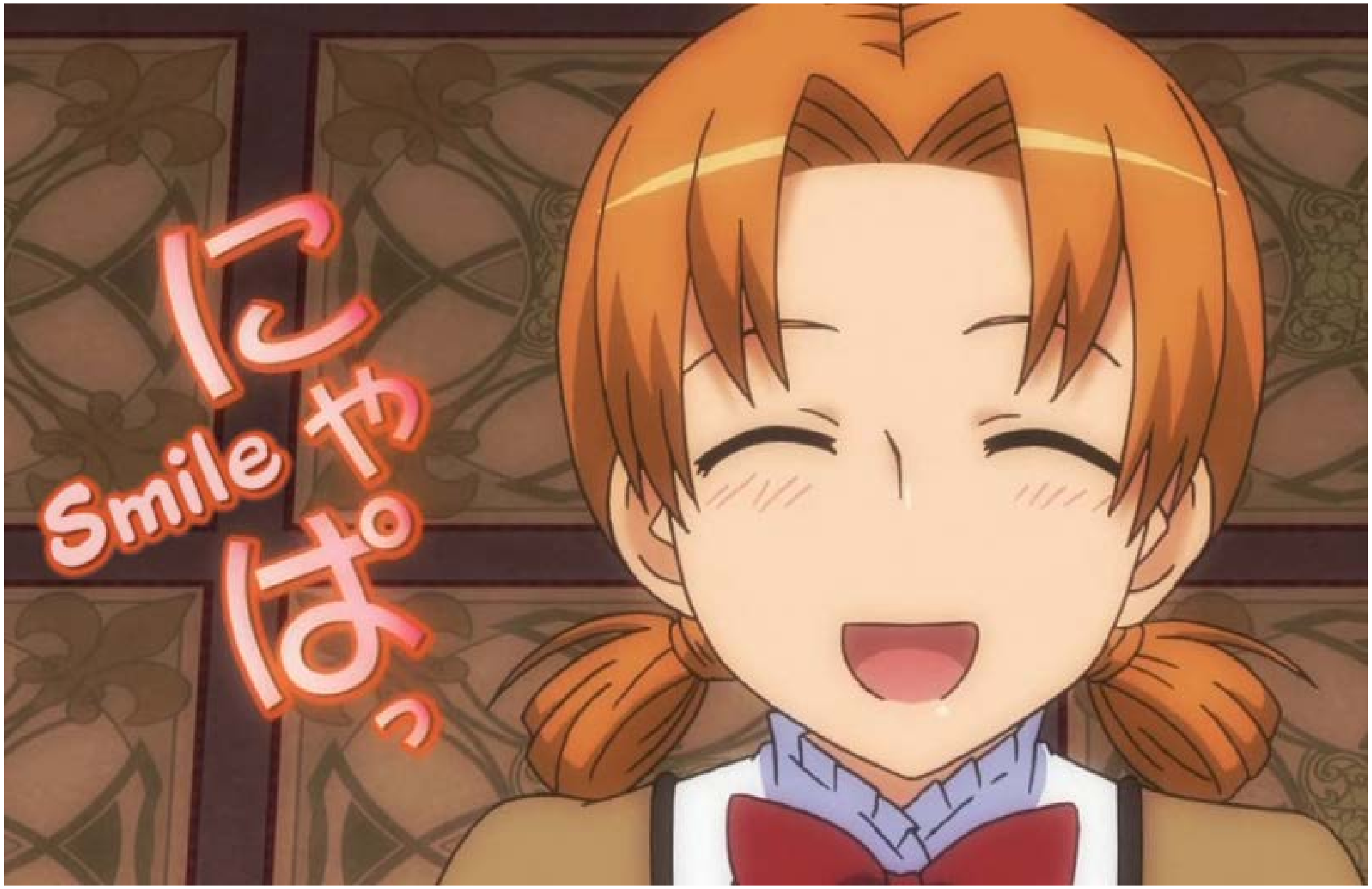


And some more examples of using layers...

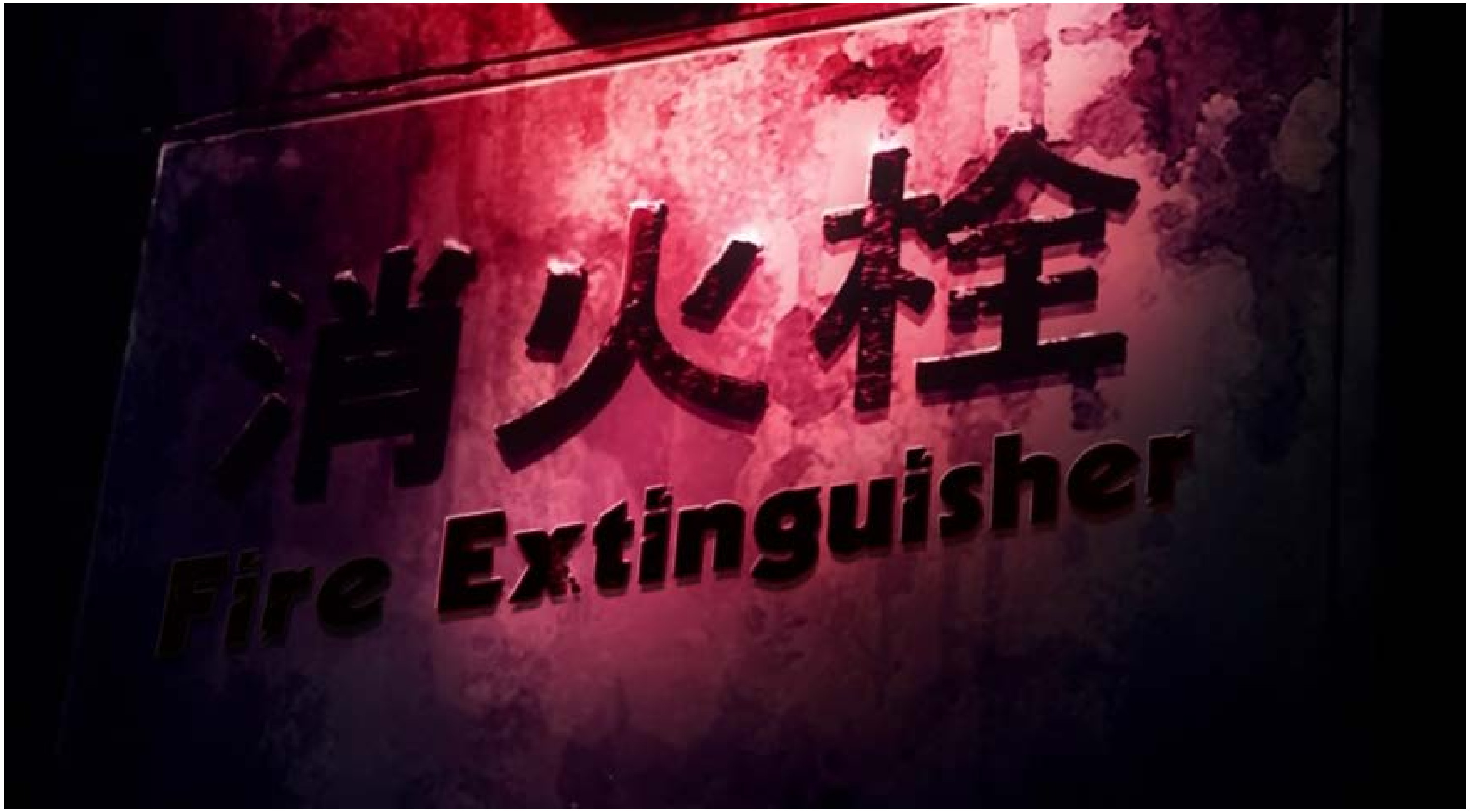


Typical case of 2 borders. You can make this blend in really well... if you have 3 layers, all with blur. (One click with Blur and Glow. OK, maybe two.)

3 layers...



Top layer - shadow, bottom layer - outline.



This one... well... let me just show you. Layers top to bottom:

```
{\blur0.8\yshad-0.5\fax-0.32\frz8.624\move(597,70,597,788)\4a&HCC&\4c&HBEB4FF&\c&H000005&}Fire {\4a&H00&\yshad-0.5\4c&H1700A4&\c&H010007&\fax-0.25}Ext{\fax-0.22}in{\4a&H80&\fax-0.2}gui{\fax-0.19}sh{\4a&HAA&}er
```

```
{\blur1\shad6\xshad0\fax-0.32\frz8.624\move(597,70,597,788)\4a&H60&\c&H000000&\4c&H000000&}Fire {\c&H06001D&\4c&H000016&\fax-0.25}Ext{\fax-0.22}in{\fax-0.2}gui{\fax-0.19}sher
```

```
{\blur1.5\yshad-0.5\fax-0.32\frz8.624\move(597,70,597,788)\3c&H3511B1&\alpha&HFF&\c&H2E0E96&\4c&H240778&}Fire {\alpha&H00&\fax-0.25}Ext{\fax-0.22}in{\fax-0.2}gu{\alpha&HFF&}i{\fax-0.19}sher
```

```
{\blur1\yshad-0.5\fax-0.32\frz8.624\move(597,70,597,788)\4a&HCC&\c&H020108&\4c&HBEB4FF&}Fire {\4a&H00&\yshad-2.2\xshad-0.8\4c&HF6F5FF&\fax-0.25}Ext{\fax-0.22}in{\4a&H80&\fax-0.2}gui{\yshad-1.8\fax-0.19}sh{\4a&HAA&}er
```

Kinda hard to explain this, but aside from multiple layers with multiple colours [red, almost black, almost white] you need different effects in different parts.
You need more light on the "Extin", "Fire" needs to be much darker, different shadow colours, different shades of red around the letters,
different \fax every few letters, varying transparency etc. And the whole thing is moving.
Yep, took about half an hour. If an episode only has like 5-10 signs, I play with them a bit more.

One more thing layers are used for - to mask the orignial sign and typeset over it. More on that in the next chapter.

« Back to Typesetting Main

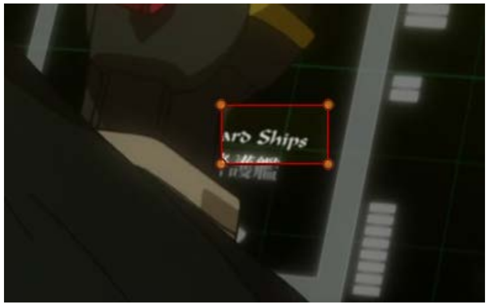
Advanced Typesetting

So what remains is clips, animation, drawing mode, and using all the things together.

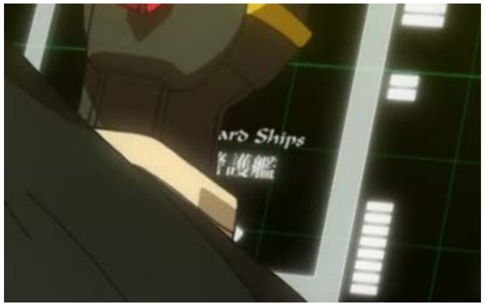
\clip

The clip tools are under the rotation and scaling tools in Aegisub. In principle they are simple.
You have a sign, and you want only part of it to be visible, so you use the basic clip tool and draw a rectangle over the area you want visible.
You'll get somethnig like \clip(54,25,380,110) in the tags. That's the coordinates of the visible area.
Rather than this part being visible the idea is the other part not being visible, so it doesn't matter if you expand it to empty areas.

It can look something like this:



< with / without the tool selected >



If it's a pixel off, you can either drag the orange dots or type in the tag. Typing without the tool selected may be better as the red lines won't be in the way.

Obviously, a rectangle won't always do, so you have the other tool that lets you draw a more complex shape.

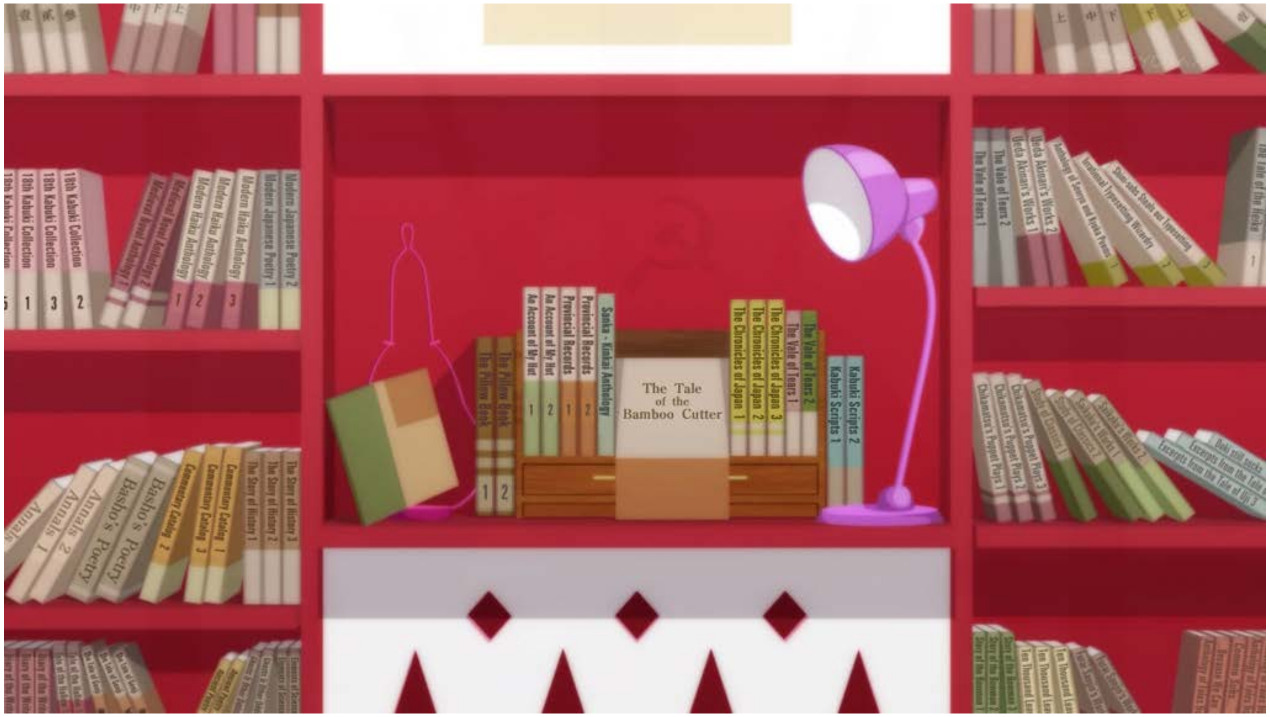
There's also \iclip, which does the opposite - selects the area that will **not** be visible. No special tool for that, so just add the **i** & adjust coordinates by typing.
\clip has more compatibility, so I use that one wherever possible. (Scripts handle easy conversion.)

That's the basics of using clips, so now for the **drawing mode**. You should already know how it works from **ASS Tags**.

Drawing is useful when you need to cover some area with solid colour and put a sign over that. I have briefly described that in the **Positioning Signs** section.
I rarely use any complicated shapes with this. Usually only square or circle. Shape can be adjusted with the scaling tool.
If I need a specific shape, I draw it with the clip tool and use a script to convert it to drawing.

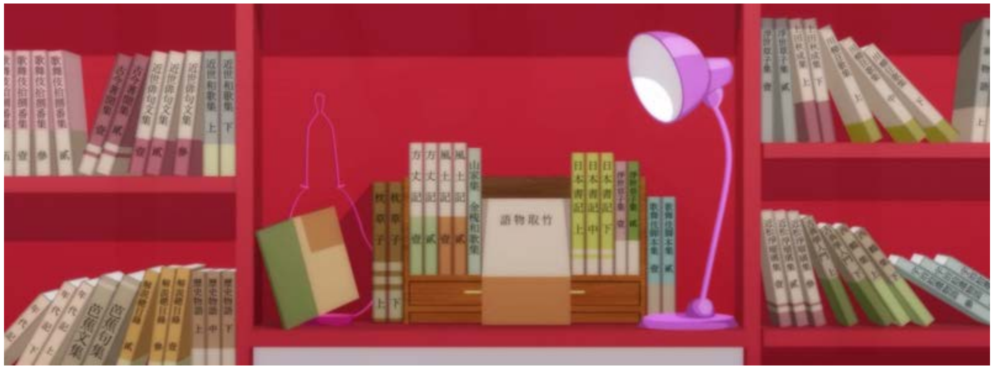
Normally I use the drawing mode only for **masking.** Using the basic rectangle with \fscx\fscy\fax\frz i can usually get what i need.

Here's what you can do with it, if you feel like spending 5 hours on 1 frame:

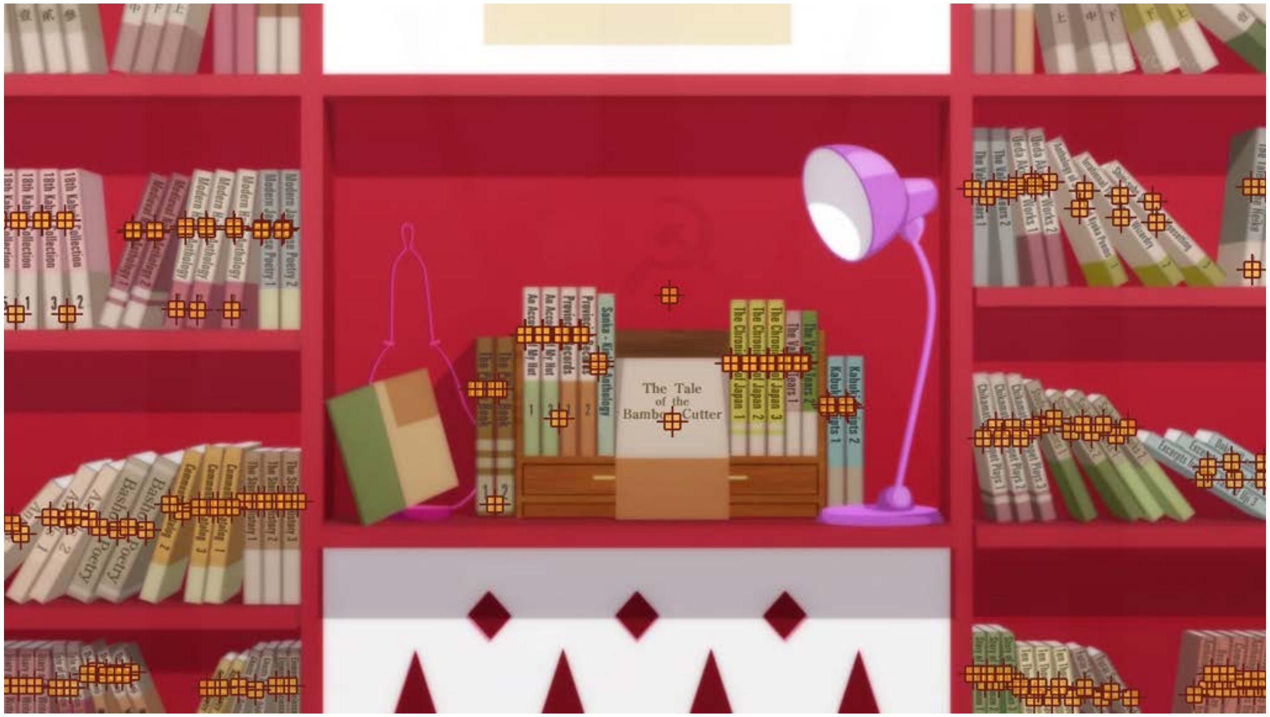


Some colours of the masks are off because back then I didn't know about the issues with colourspaces and used ffmpeg in Aegisub 2.1.8 or 2.1.9.
If you're using those versions, use avisynth to load the video. If you're using Aegisub 3.0, use the BT.601 colourspace [which I think is on by default].
2015 Note: And if you're reading this, do NOT use BT.601. (Turn it off in Options-Advanced-Video.) Yeah, shit keeps changing.

In case it wasn't clear to someone, all the books had Japanese titles, of course...



So it really was 5 hours of pretty tedious work. I don't recommend that you ever try that. Unless you like... don't need hands for the rest of your life.
Each book has one typeset for the title with matching colour and size, and a mask in the colour of the book to hide the JP title.
Sometimes there are additional ones for the numbers.
Here's what it looks like in working mode. I think it's almost 200 lines.



OK, but back to a bit more sane things...

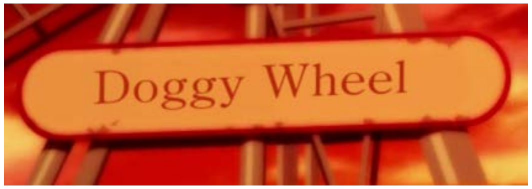
If you need a rectangle with round edges, or even a circle, you can still do it with the basic square.
For rounded edges, use border with a value as high as you need to get the right shape. \bord20 or \bord30 may be useful values.
Of course you need the exactly same colour for \c and \3c.

To get a circle, you do the same but scale the original square down to 1 pixel.

You can actually use this with a regular font. For example if you use a period with \bord50, you get a pretty good circle.
You can use the letter O for an ellipse or whatever else gives you a shape you need. Just match the primary and outline colour and it works.

Speaking of that, sometimes you need a **mask with a slight gradient.**
Well, you might need a strong gradient but then you'd have to use an actual gradient and have hundreds of lines...
But for a mild one, you can actually use some symbols from a font with large border.

For example in one Nise ep I had this sign:



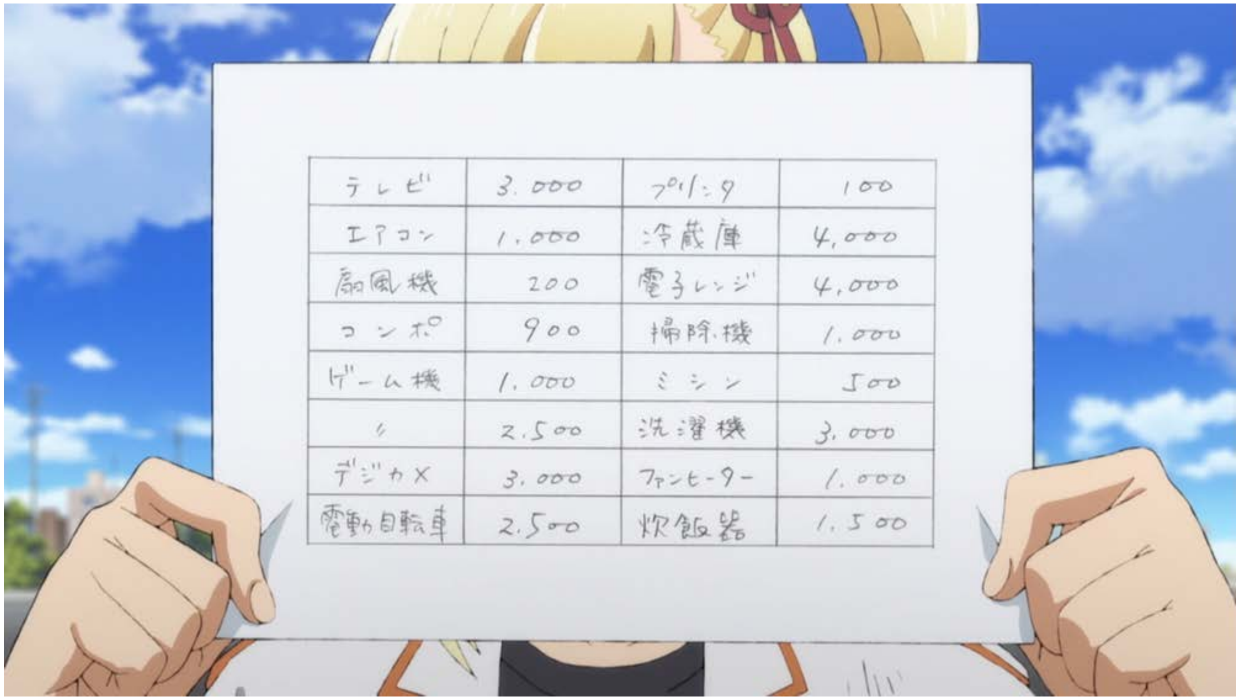
The background on the left is darker than on the right. Or maybe you could say the left is more orange, right is more yellow.
So a mask in one colour didn't work, because it was always too visible on one end. And you can only have one colour for the drawing mode.
So what i did was use 333333333333 with \bord10 as the mask and changed colour every few letters. I mean numbers.
You can use OOOO or 8888 if you need a roundish mask, or use IIIII or ||||| if the range of colours is larger.
Of course for each section the outline colour must match the primary. But to make it work you need one more thing.

That brings us to the last part about masks... **blurring the mask**.

While I start with \blur0.5 on all signs, I put \blur1 on the masks, and more if needed/possible.
The reason is that more blur helps it blend better. So if there's enough space around, you can blur it a lot, like \blur5 or more,
and then even if the sign has a slight gradient, you may get away with just one colour, because with blur5 you always have 5 pixels of fade.
[Actually this doesn't really relate to pixels, but you get the idea.]

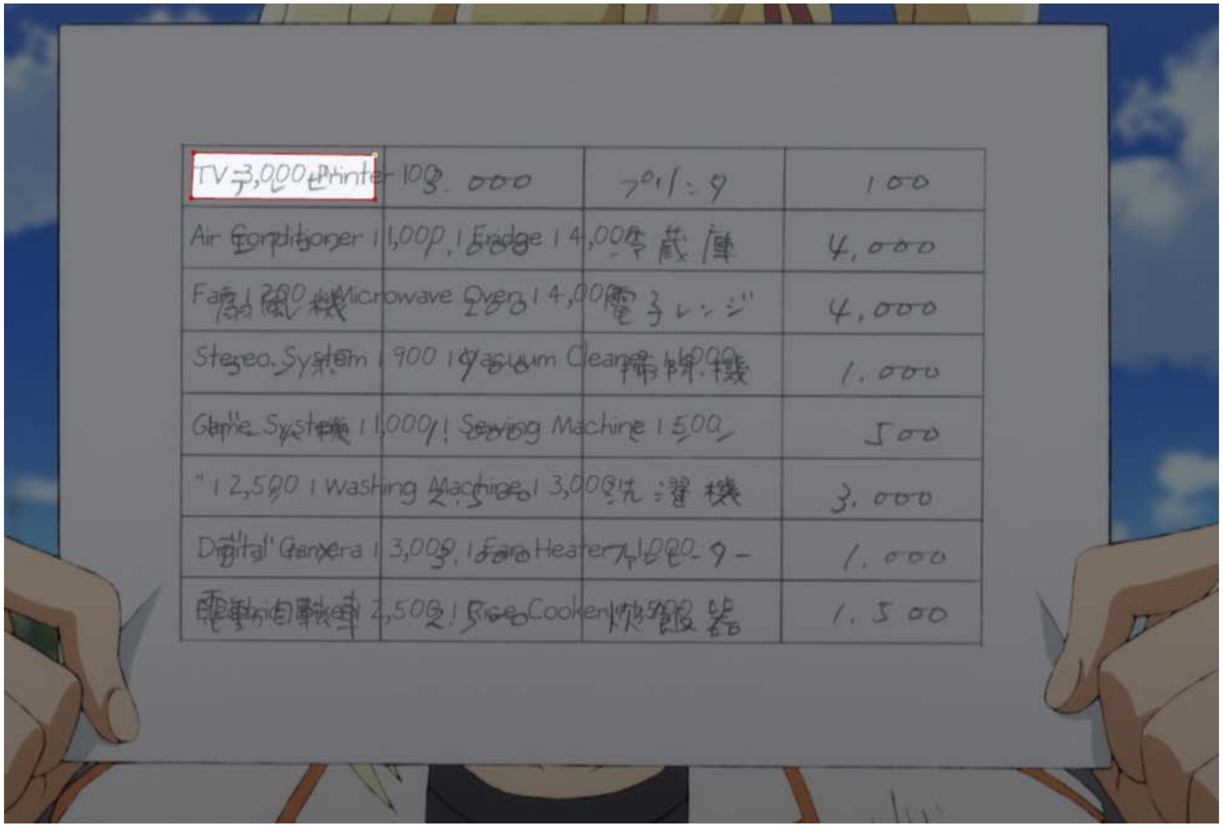
So back to this sign. We have those 333s changing colour. For that to work you need that blur so that the different shades can blend into each other smoothly.
I used \blur3 here. Couldn't afford more, because it would either start showing the kanji under it, or grow outside of the orange area.
If you look hard, you may still find some barely visible discrepancies, but basically here you have a mask with a gradient.
For reference the colours are &H3D8FE9 on the left and &H4094EF on the right.

2018: This week on Hinamatsuri...

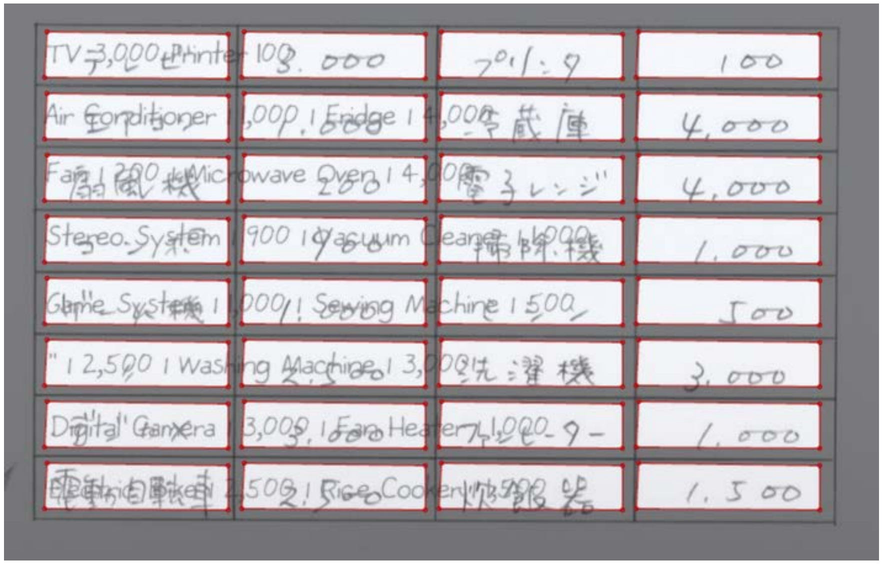


テレビ	3.000	プリンタ	100
エアコン	1,000	冷蔵庫	4,000
扇風機	200	電子レンジ	4,000
コンポ	900	掃除機	1.000
ゲーム機	1.000	ミシン	500
〃	2.500	洗濯機	3.000
デジカメ	3.000	ファンヒーター	1.000
電動自転車	2.500	炊飯器	1.500

When I saw this, I thought, Oh, cool, I'm gonna try some fun stuff here.
So clearly this has to be masked. I suppose you might want to mask the whole thing and redo the grid somehow,
which probably wouldn't be too hard either, but I had something else in mind. I started by drawing a clip in the first field.

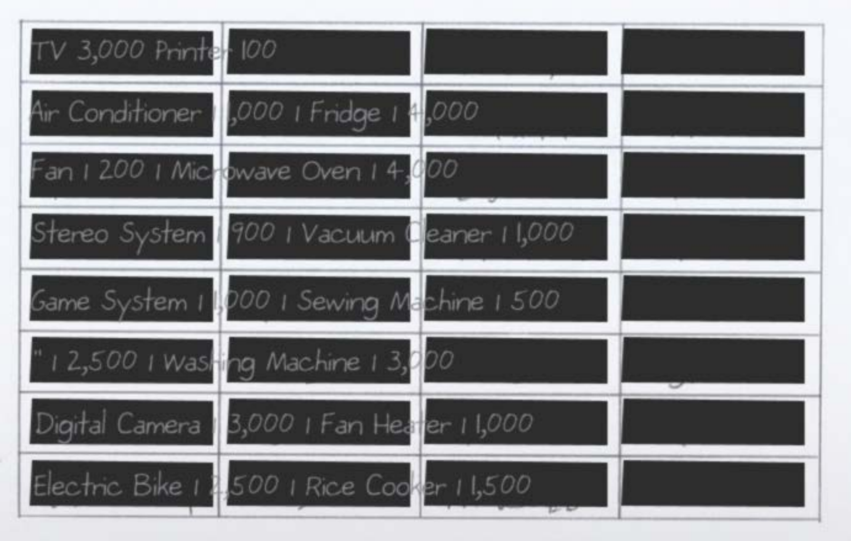


Of course we don't wanna do that shit 32 times, which is why I write a ton of lua functions.
This was probably the first good opportunity to use Clone Clip from *Significance*.
So I tell it to make 4 horizontal and 8 vertical clones, and how wide and tall one field is, and there we go.
This is all in one line.
(BTW, you can actually draw this in one line with the clip tool, meaning you can make separate shapes.
All you have to do is, every time you set the coordinates for a new one, add "m" before the coordinates that appear in the Edit Box.
Like this: *m 326 166 l 324 203 473 204 473 168 m 486 166* ... and then you keep drawing.)



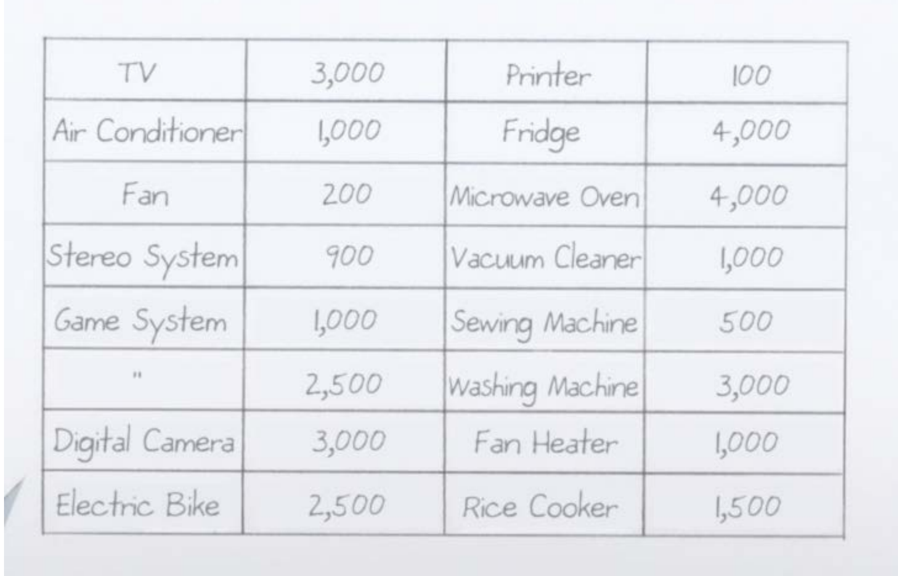
Now we easily convert the clip to drawing. (I was just clipping an empty line.)

I suppose I could have added one pixel to the horizontal distance, but I just used \fscx101.



TV 3,000 Printer 100
Air Conditioner I 1,000 I Fridge I 4,000
Fan I 200 I Microwave Oven I 4,000
Stereo System I 900 I Vacuum Cleaner I 1,000
Game System I 1,000 I Sewing Machine I 500
" I 2,500 I Washing Machine I 3,000
Digital Camera I 3,000 I Fan Heater I 1,000
Electric Bike I 2,500 I Rice Cooker I 1,500

You may not see it, but I actually had to gradient the mask once I set the white colour.
The bottom is a bit darker. But it's only slightly, so I made like 100-pixel strips, and there were only 4 or 5.
Then it was just adjusting spaces in the text. (It could be split, but I didn't wanna have that many lines.)
I was quit pleased with how easy this was.



TV	3,000	Printer	100
Air Conditioner	1,000	Fridge	4,000
Fan	200	Microwave Oven	4,000
Stereo System	900	Vacuum Cleaner	1,000
Game System	1,000	Sewing Machine	500
"	2,500	Washing Machine	3,000
Digital Camera	3,000	Fan Heater	1,000
Electric Bike	2,500	Rice Cooker	1,500

Now for the most fun part...

\t

This is the tag with which you can do almost anything..... and make everyone's player lag.
As a demonstration you can try this:

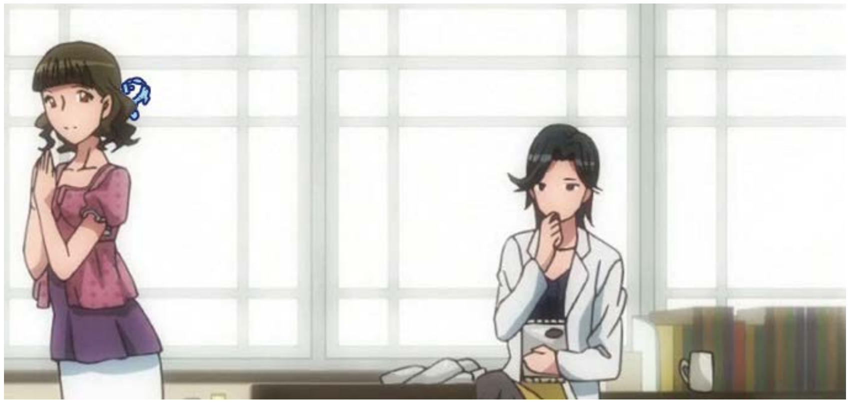
```
Dialogue:
0,0:00:00.00,0:00:06.00,Default,,0000,0000,0000,,{\an5\q2\fs40\b1\bord1\blur0.1\shad0.1\1a&HFA&\4a&HF0&\t(0,3000,3,\fs75\bord4\xbord10\shad22\blur1\1c&H00FFA9&\3c&H9B2664&\4c&H0C1A4C&\4a&H90&\1a&H00&\fscy150\fsp15\frz15\fax-0.4)}Unlimited Eyecancer Works
```

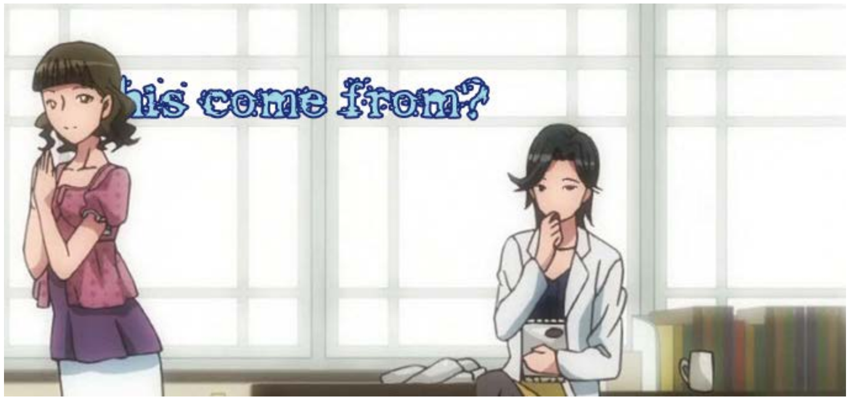
This will change font size, border size, shadow distance, blur, all colours, transparency, font scaling, font spacing and rotation. Oh and CPU usage.
For even more CPU usage, add movement, \clip, the other rotations and \blur15.
2015 Note: It would be hard to make one line lag nowadays, no matter how many transforms. But if it's many lines at the same time...

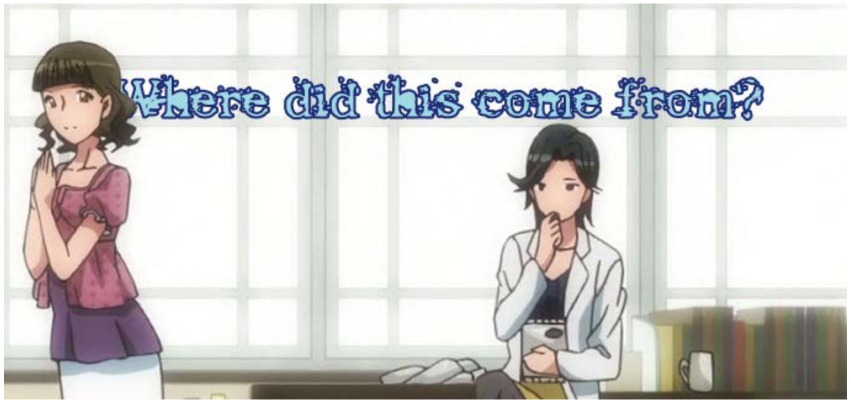
So this gives you the idea of how you can change the basic properties. Use your imagination to figure out how far the options go.

Let's try something more practical though. Like text appearing bit by bit.
Type some text and place it somewhere. Now use the clip tool to make only the first letter visible.
Let's say you get \clip(50,150,100,250). Now add \t, use the same clip in the \t tag, but change the second X coordinate to somewhere after the last letter.
If the text ends at 400, you'll use \clip(50,150,100,250)\t(\clip(50,150,**400**,250)). This will be showing static text gradually from the first letter.
See how it works out, and adjust coordinates as needed if something's off.
If you need the text to appear in the first 500ms and stay on screen, use \t(0,500,\clip.....)

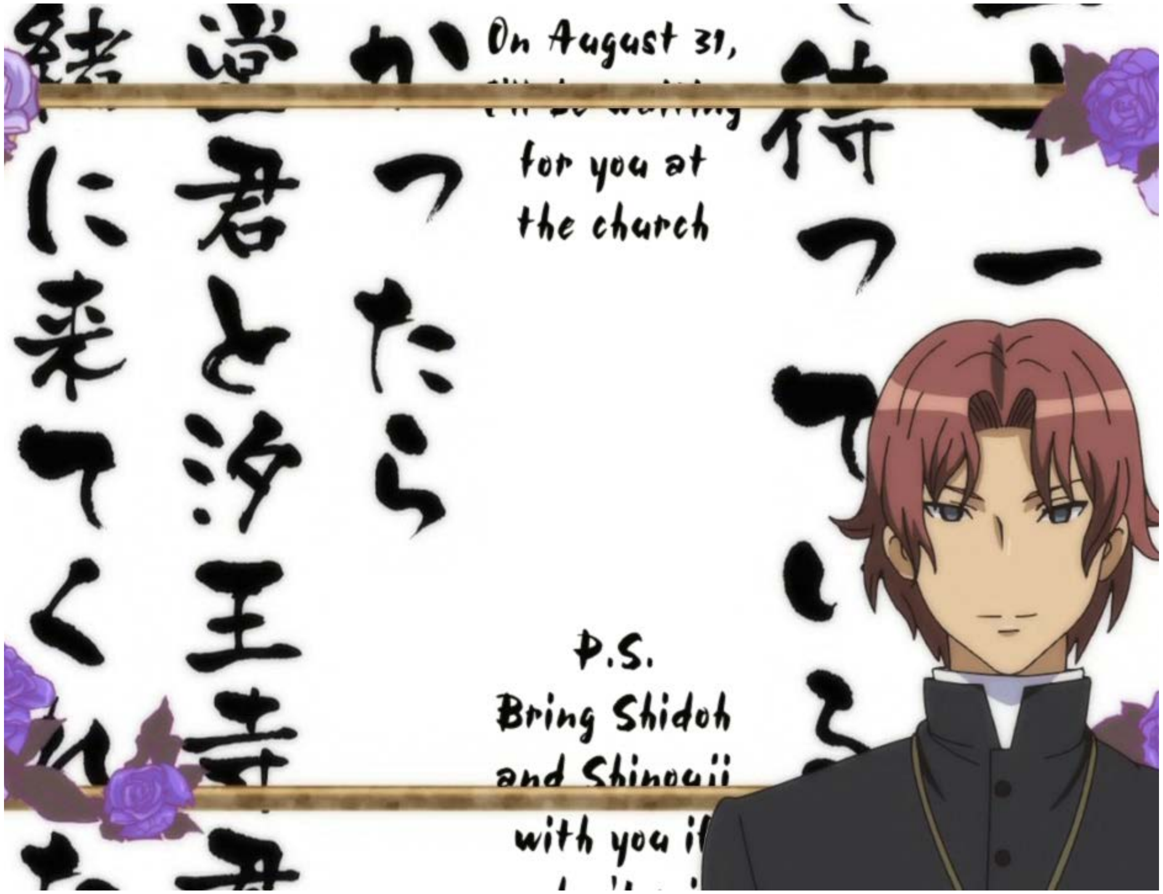
Something similar would be text that moves into a visible area, though here you don't actually need \t.
Let's say a person is standing in the picture, text is generated behind his\her back, no pun intended,
and the text comes out on the right. So what you do is use \move to make the text move from left to right,
and you'll use \clip to make sure the text is not visible behind that person. You'll get a sequence like this:





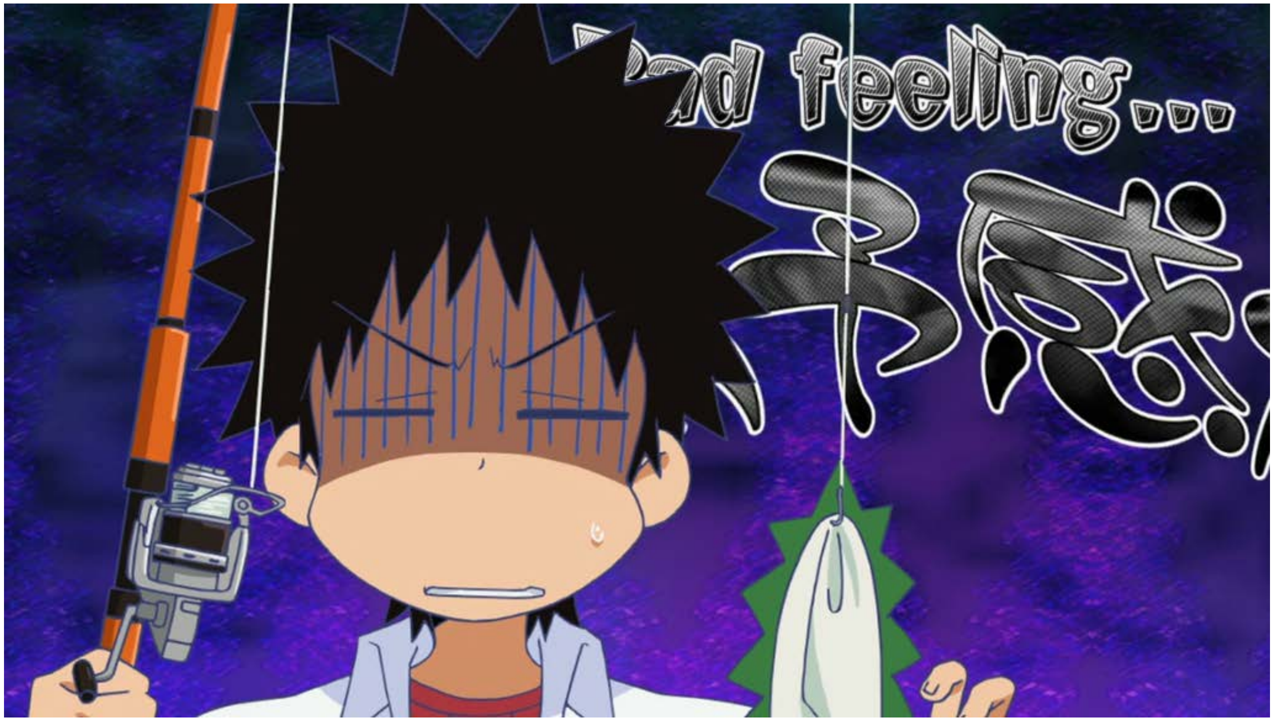


You'll have to use the vectorial clip for this, to follow the hairline.
You could also expand the clip to the other side of the person and just make the text scroll behind her...
...as long as she isn't moving. If she is and you still wanna do this, change \move to \pos & go frame by frame.
If after a few hours of that you feel like screaming "Zetsubou shita!", no one will blame you.

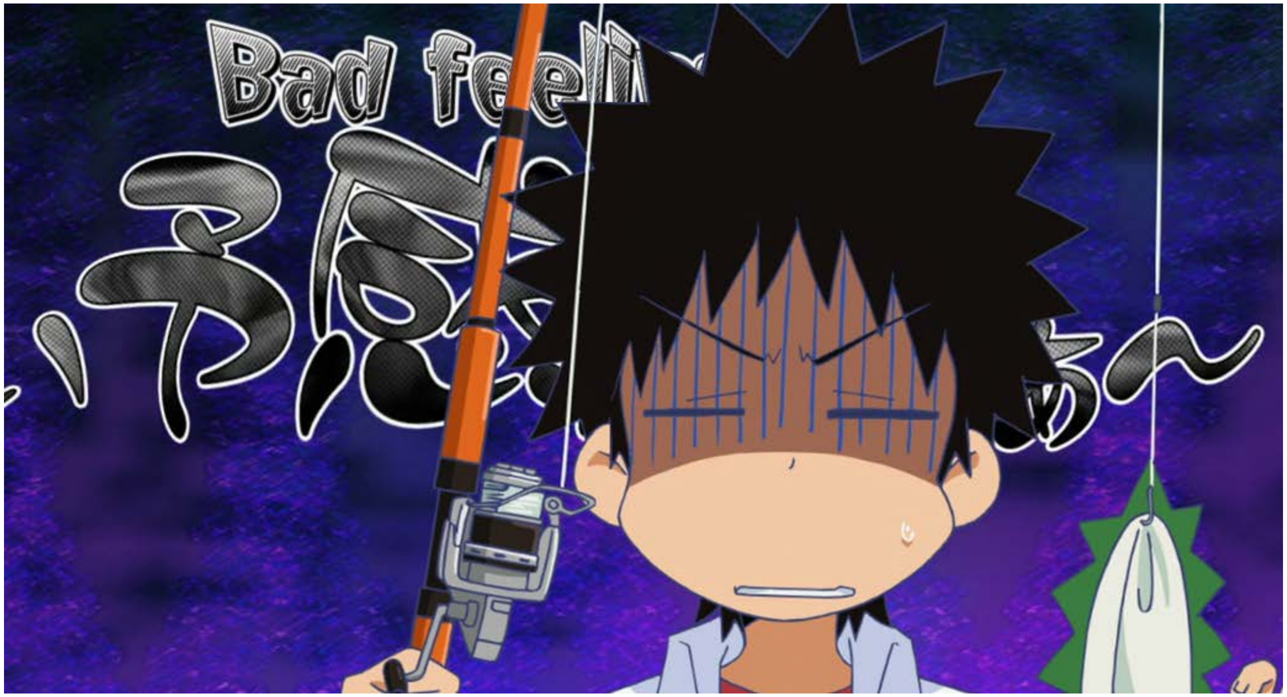


Here the text is scrolling bottom to top, hiding behind the bars and the guy.
Use a simple \move, tune it so that it moves precisely along with the kanji first. Then use a clip to outline the bar.
I set it up so that each of the 2 portions of the text only goes under one bar so that I wouldn't have to clip both for one line.
You could, however, lead the clip on the side of the screen to the other bar and cover both of them with one clip.
Instead of clipping everything except the bars, you'll just clip the bars and then rewrite the \clip to \iclip, which inverts it.
Of course you also have to include the guy's shoulder in the clip.

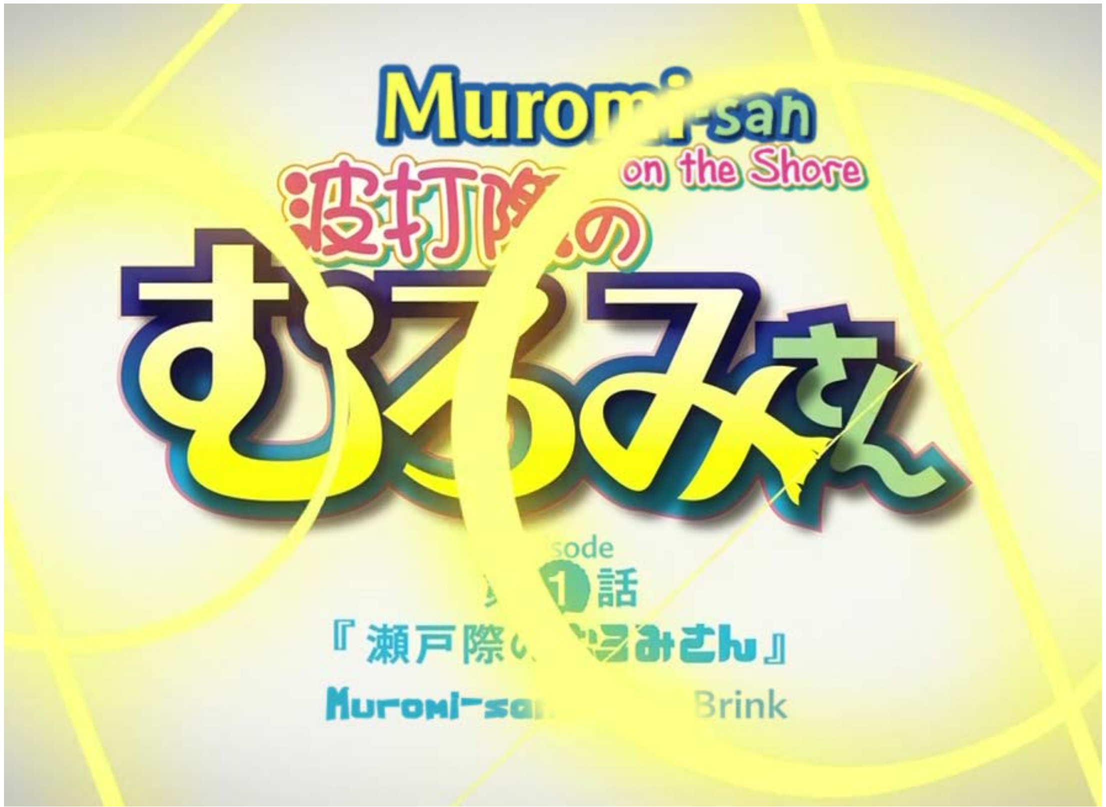
Here's more fun stuff:



Find an awesome font. Set the colours, border etc. Clip around the guy's head and his tools. Apply \move.



This is even more tricky:



Create all signs with colours, borders, shadows, layers etc. Use ASSDraw to create parts of the yellow circle to mask parts of your signs.
Change colour of the letters that are close to the yellow circle. Blur appropriate things for the glow effect.
Now mocha track over more than 100 frames, as the whole thing is zooming out with random flashes of light.
(Then change the episode title and colours every episode.)

Another example where you can combine \move and \t is when text is growing larger.
You know the trailer kind of stuff where a line appears and seems to get closer, then a few images and another line.
You'll use \t(\fs) to make the sign 'grow,' and \move to make up for any inconsistencies that may arise.
If the alignment is \an2, then the sign will only expand upward. \an5 will make it expand to all sides,
but if you're placing this above a JP sign that's already expanding, you'll need to \move up a bit,
otherwise the expansion of both signs may bring them too close together or even overlap.

This kind of signs may also use other effects, like the line appears and then slowly gets blurry,
or the letters move apart from each other - \fsp.

For illustration, try these things out to get the idea of what you can do (use 720p video AND script resolution):

```
Dialogue: 0,0:00:20.00,0:00:25.00,Default,,0000,0000,0000,,{\pos(300,300)\bord0\frx0\fry90\t(0,2000,2,\fry0)\org(20,300)}What is this I don't even...

Dialogue: 0,0:00:25.00,0:00:30.00,Default,,0000,0000,0000,,{\an5\fad(1500,0)\move(155,87,1040,670,0,1500)\t(0,1500,\frz-1080)}I still don't...

Dialogue: 0,0:00:30.00,0:00:35.00,Default,,0000,0000,0000,,{\move(400,200,800,200,0,3500)\t(0,4000,\fry720)}What Now?

Dialogue: 0,0:00:35.00,0:00:40.00,Default,,0000,0000,0000,,{\move(1000,450,300,200)\t(\fs120\bord8)\b1\clip(600,10,800,710)\frx14\fry24\frz10}That's enough!

Dialogue: 0,0:00:40.00,0:00:45.00,Default,,0000,0000,0000,,{\b1\org(640,50)\fax1\frz-60\t(\frz60\fax-1)\move(640,630,680,260)\clip(240,85,860,680)}Or is it?!
```

That's about all I can think of right now. The rest is up to your imagination.
Remember though that overusing this will cause lag, so nuke any tags that you don't really need,
and don't use too high values of the really laggy things like blur, or all rotations at the same time.

If there's anything specific that I haven't mentioned anywhere, let me know.

2018 Note: Creating transforms is done using **HYDRA**. Just check the tags you want, and click on Transform.
With scripts like Hyperdimensional Relocator, you can also easily create \move from \pos, align a bunch of lines, move vectorial clips, and all kinds of other things.

« Back to Typesetting Main

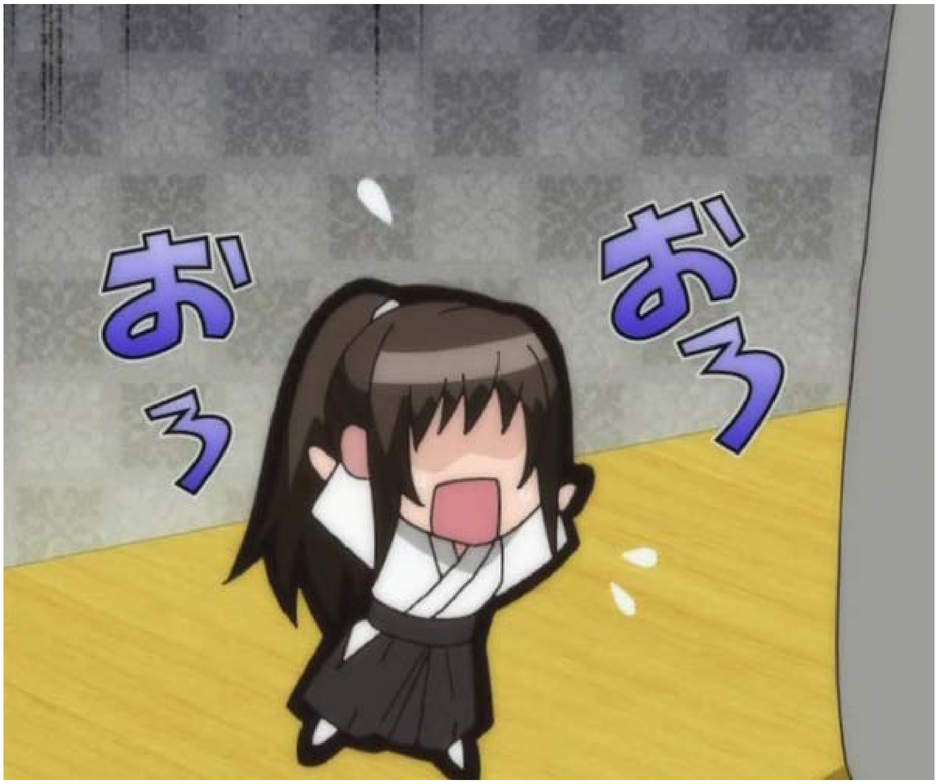
Typesetting: Creating Gradients

Creating gradients is simple in principle, but since the **Gradient Factory** script is a bit lame, you can run into some problems.

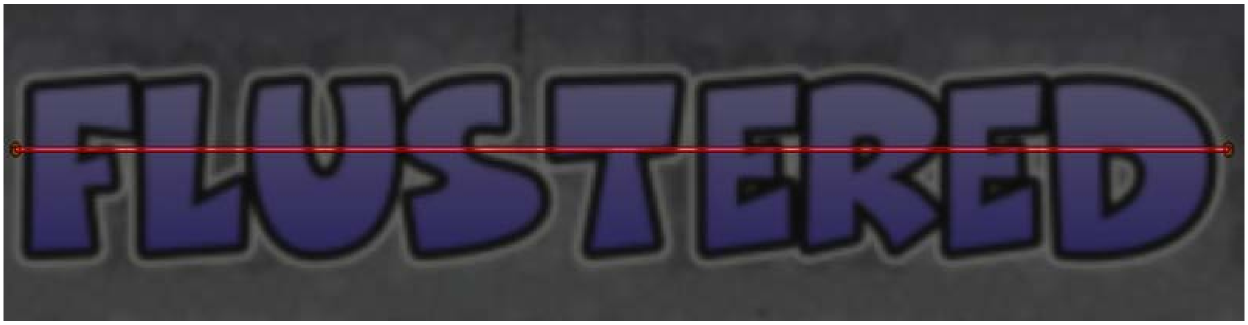
2015 Note: Nobody uses GradFactory anymore (I hope). Read this part only for understanding how gradients work, and ignore that shitty script. Further down the usage of lyger's 'Gradient everything' script is explained.

2018 Note: **HYDRA** can create all types of gradients - vertical, horizontal, by character, by X characters, by line, with colours going there and back, etc.

First of all, what *is* a gradient?

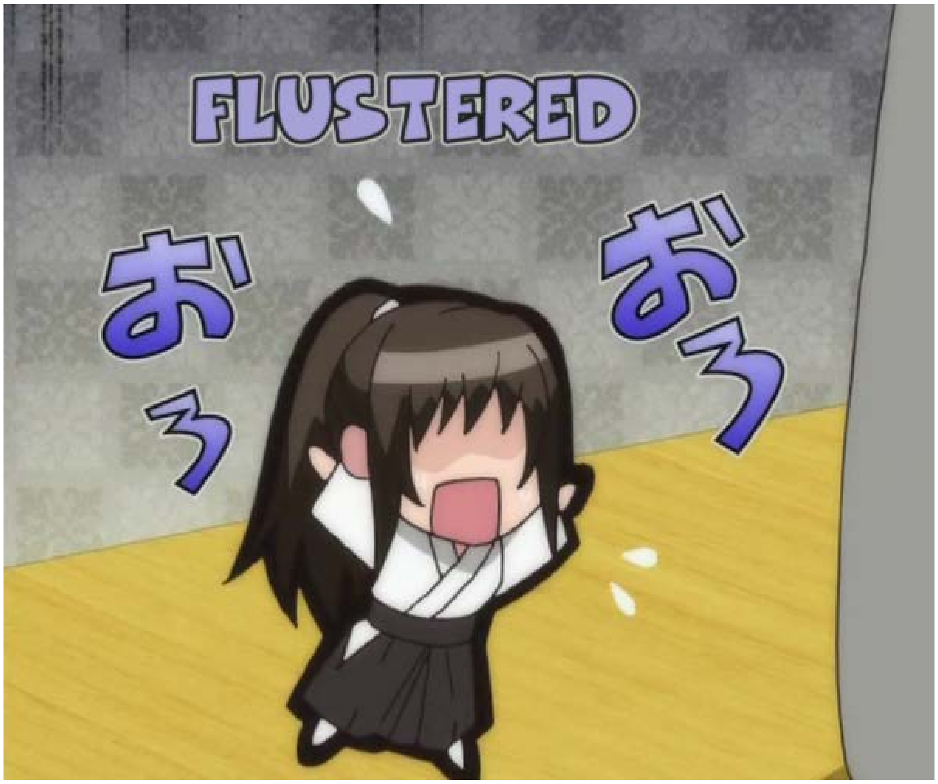


Here you can see the colour goes from light blue at the top to dark blue at the bottom.
The way to replicate this in typesetting is to chop the sign into a large number of clipped lines, each with a different shade of the colour, like this:

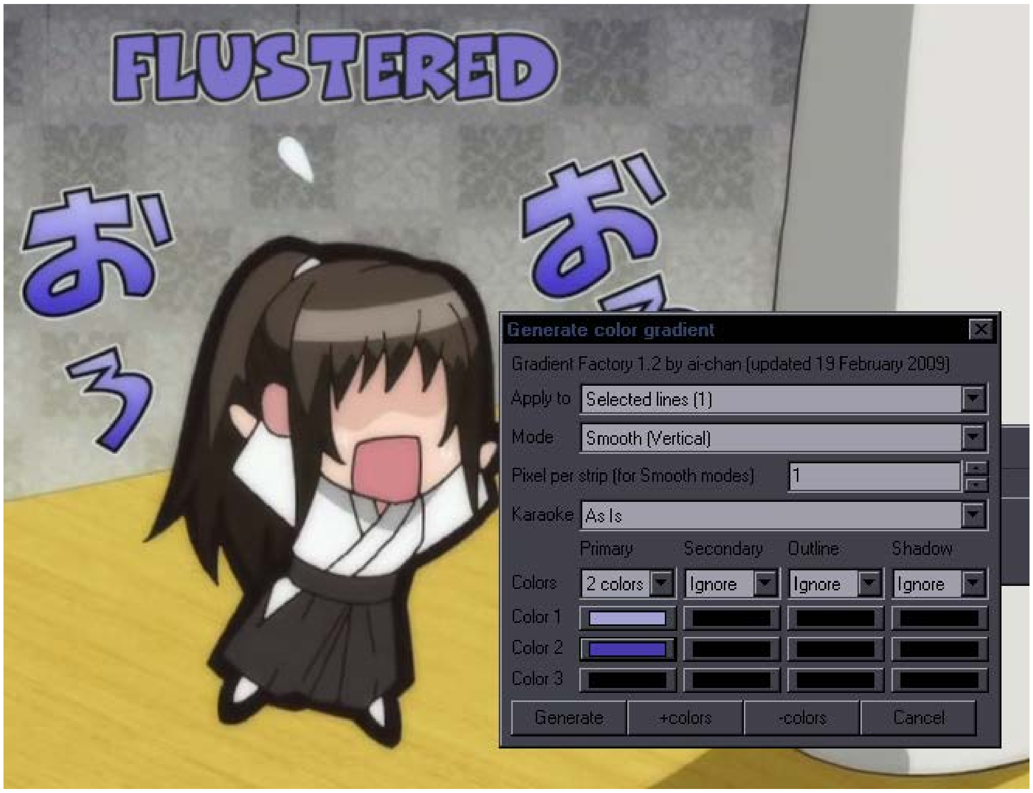


Here I have one line selected, containing this: {\bord0\blur0.6\clip(120,173,411,174)\pos(266,209)\1c&HCE8189&}FLUSTERED
You can see that vertically the clip goes from 173 to 174, i.e. the strip is only one pixel wide.
You can do 2- or 3-pixel-wide ones, but of course the more pixels per line, the worse it looks.
On the other hand, the less pixels per line, the more lines, and the more lag.

What Gradient Factory does is create all these lines and give each one a different colour.
With 1 pixel per line this can look great, but may lag, especially with fades and stuff.
In this example you would choose the top colour, the bottom colour, and the width of the clips.
The script does the rest... or that's the idea anyway. It has its own problems though.
Now how to start:



Create your basic typeset and set the sizes and colours in the style rather than with tags.
Grad Factory nukes all tags from the line except \pos, and creates the clips and colours,
which means all your scaling, borders, and other things will be nuked.
You can re-add them later, but for several reasons it's best to have as much set in the style as you can.
I commonly create a special style just for the gradient I'm making.
One reason for that is that the area of the gradient is calculated from the font size in the style.
If the style has \fs20 and your typeset has a tag with \fs50, the gradient will create lines for \fs20 and you're fucked.
So I recommend setting the style as close to the required result as you can.
In the image above, I picked a font and set the colours, borders, layers and blur.
The blue colour can be anything, as it will be overwritten by the gradient anyway.



Generate color gradient

Gradient Factory 1.2 by ai-chan (updated 19 February 2009)

Apply to: Selected lines (1)

Mode: Smooth (Vertical)

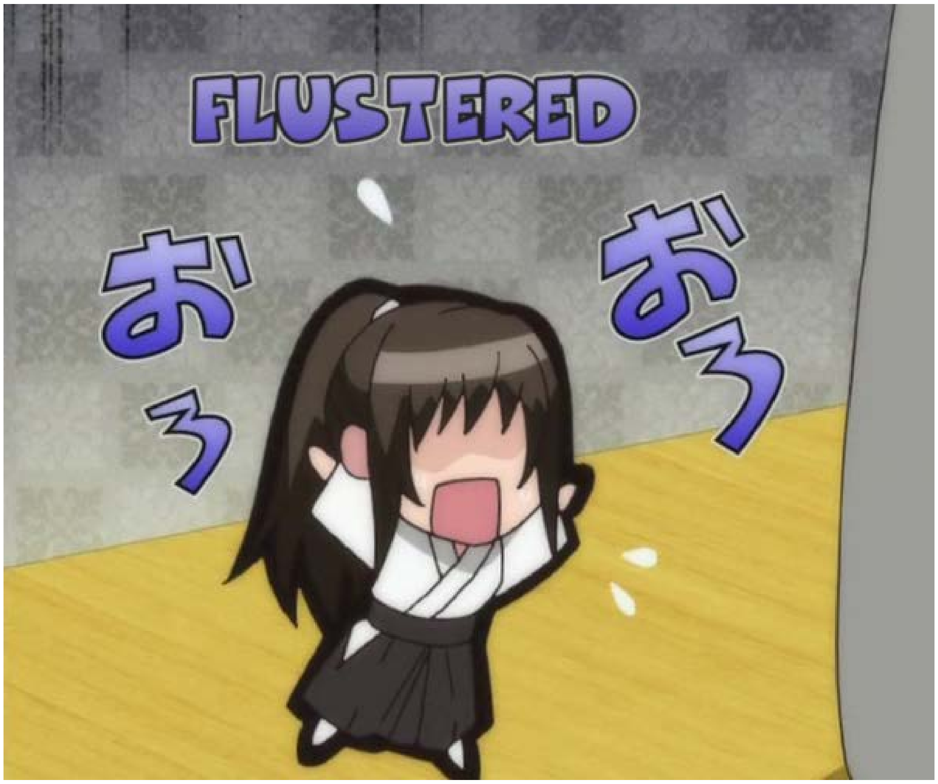
Pixel per strip (for Smooth modes): 1

Karaoke: As Is

	Primary	Secondary	Outline	Shadow
Colors	2 colors	Ignore	Ignore	Ignore
Color 1	🟪	⬛	⬛	⬛
Color 2	🟦	⬛	⬛	⬛
Color 3	⬛	⬛	⬛	⬛

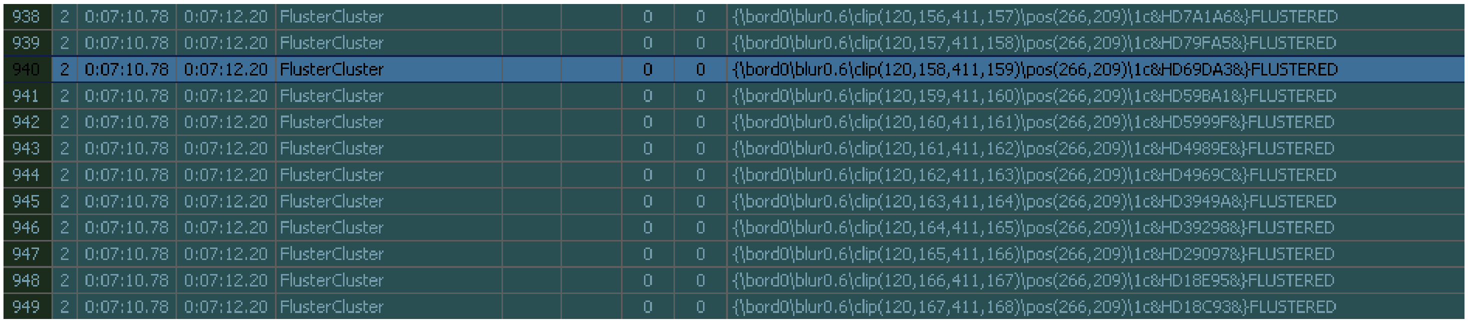
When you have that, run the GradFactory script on the blue layer.
Settings are as you see: apply to this one line, vertical gradient, 1 pixel strip, and you only need 2 colours here.
If you need more than 3 colours, you use that +colors button. The rest should be self-explanatory.
Aside from the font size from the style and nuking the tags, there's another "problem."

The gradient isn't calculated for the visible part of the font, but for the whole "box."
That means the gradient usually starts a few pixels above the letters and ends a few pixels below,
so the start/end colours you selected will be outside the visible range.
To compensate for that, you could select a bit lighter colour for the top and a bit darker for the bottom.
It may take a few tries and personally I find it the most annoying thing about this script.
However, this can be bypassed, and the colours can be tuned with the **fbf transform** script.



Like I said, the GradFactory will nuke other tags than \pos, \clip and colours, so you'll have to re-add things like blur,
but when you do, the final typeset will look something like this^.

You may also need fades or whatever, but you get the idea.
This is what the gradient will look like in your .ass file:



938	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,156,411,157)\pos(266,209)\1c&HD7A1A6&}FLUSTERED
939	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,157,411,158)\pos(266,209)\1c&HD79FA5&}FLUSTERED
940	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,158,411,159)\pos(266,209)\1c&HD69DA3&}FLUSTERED
941	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,159,411,160)\pos(266,209)\1c&HD59BA1&}FLUSTERED
942	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,160,411,161)\pos(266,209)\1c&HD5999F&}FLUSTERED
943	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,161,411,162)\pos(266,209)\1c&HD4989E&}FLUSTERED
944	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,162,411,163)\pos(266,209)\1c&HD4969C&}FLUSTERED
945	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,163,411,164)\pos(266,209)\1c&HD3949A&}FLUSTERED
946	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,164,411,165)\pos(266,209)\1c&HD39298&}FLUSTERED
947	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,165,411,166)\pos(266,209)\1c&HD29097&}FLUSTERED
948	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,166,411,167)\pos(266,209)\1c&HD18E95&}FLUSTERED
949	2	0:07:10.78	0:07:12.20	FlusterCluster			0	0	{\bord0\blur0.6\clip(120,167,411,168)\pos(266,209)\1c&HD18C93&}FLUSTERED

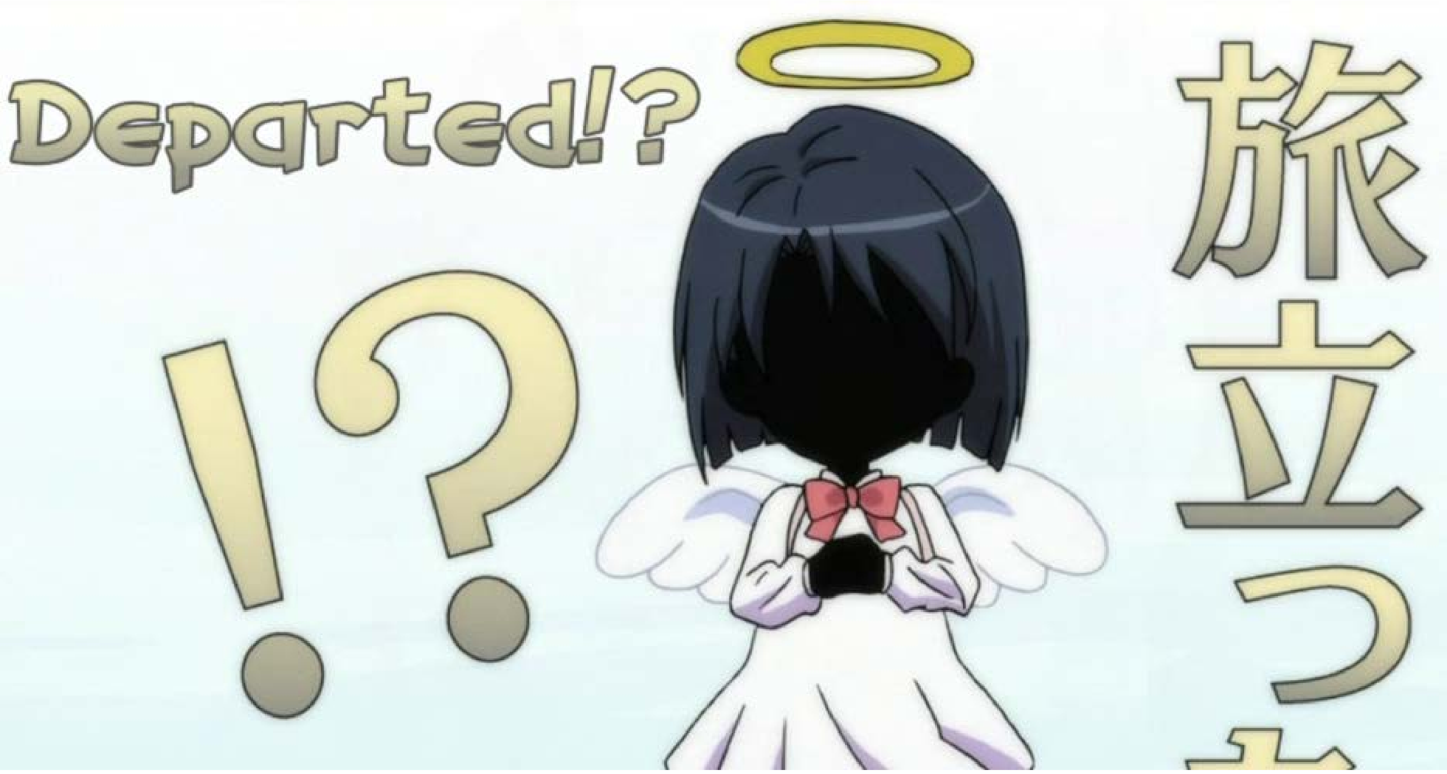
Each line is a clip with a different colour.

As I mentioned before, you can use the transform script to tune the colours.
Find the top and bottom VISIBLE lines.
You'll find that several of the first and last lines actually aren't visible on screen,
because, like I said, the gradient is created for the whole "box" of the font.
You can delete the extra lines.

Then you can set exact colours on the first and last line and run the transform script between them.
The transform script even allows you to have different colour gradients for different letters etc.
Refer to the info inside the script for how it works.

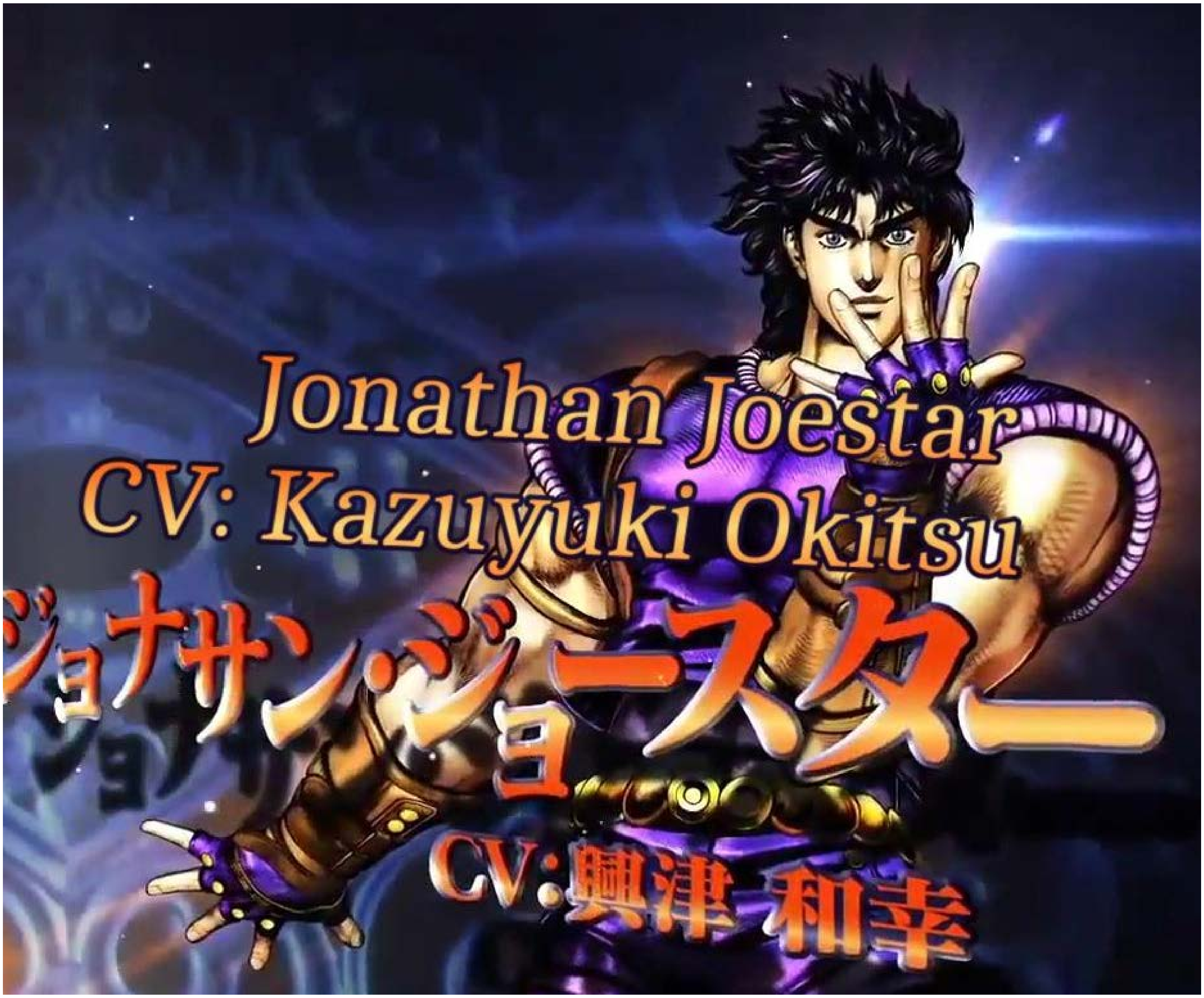
Another problem is gradients for typesets with \frz.
The gradient will only be vertical or horizontal, but you may want to rotate the actual text.
Since the Factory uses only the style for calculations, the clips wouldn't extend across the whole rotated text.
The way to bypass it is to change font size in the style to larger, run the gradient, change size back, and rotate the text.

Since larget typesets can mean that you'll have over 100 lines for the gradient, there's a danger of lag.
If the typeset is static and has no fade, you can get away with 1-pixel strips. It looks awesome.

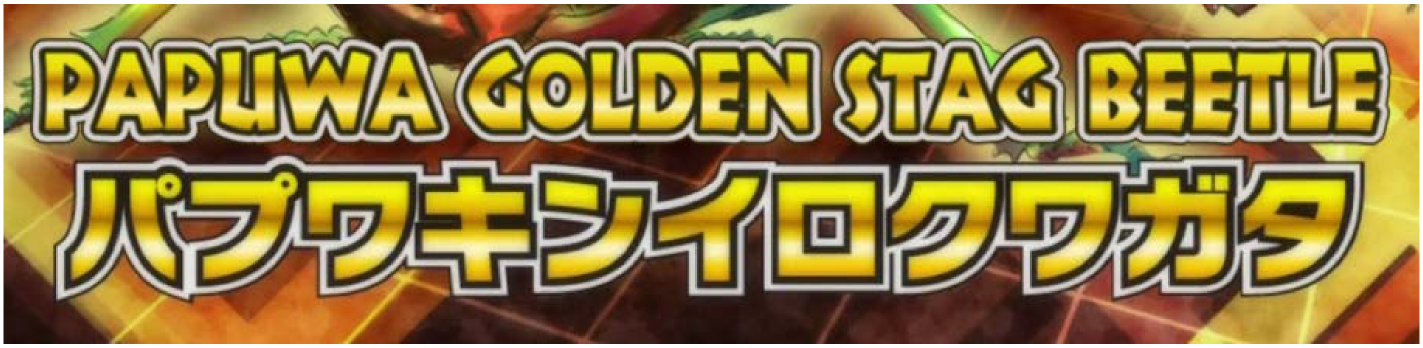


Fades are a killer so I'd suggest at least 2-pixel strips. [Also the big "!?" was part of the video and is not a rotated typeset gradient.]

Here's one of those \frz ones:



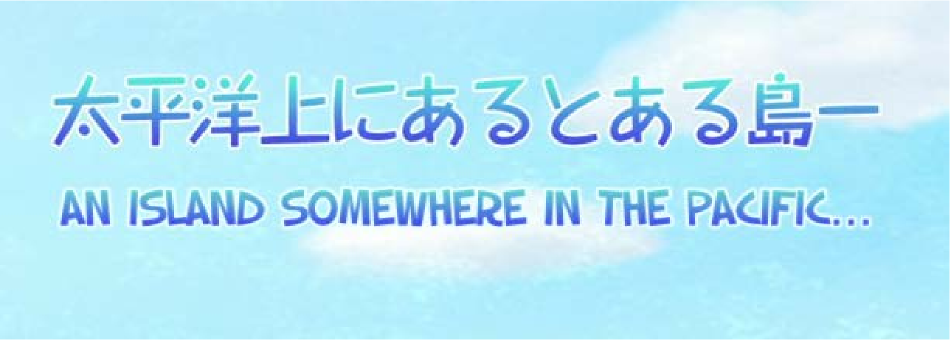
You can also do one with multiple colours:



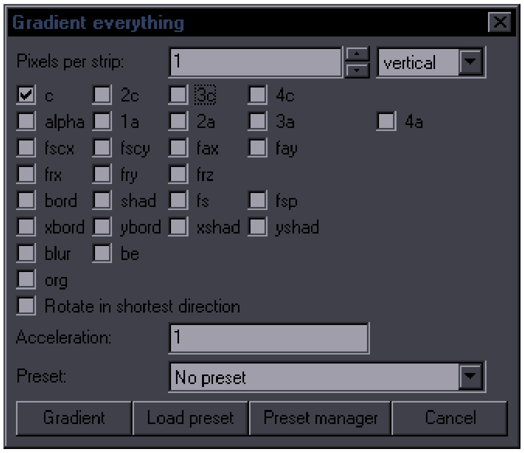
Yeah, this is a bit laggy but... looks pretty good.
2015 Note: Probably wouldn't be laggy today.

Now that I wrote all this... lyger has made creating gradients much easier, so just **go here** to get the gradient scripts.

Let's just quickly explain how to use "Gradient everything" on this example:



1. Create a regular 2-layer sign with blur and set the primary colour to the green at the top.
2. Duplicate the top, green layer.
3. Change the second of the two lines to the blue colour at the bottom.
4. Draw a clip around the text on either the green or the blue layer.
5. Select both lines and run the script with these settings:



Gradient everything

Pixels per strip: 1 vertical

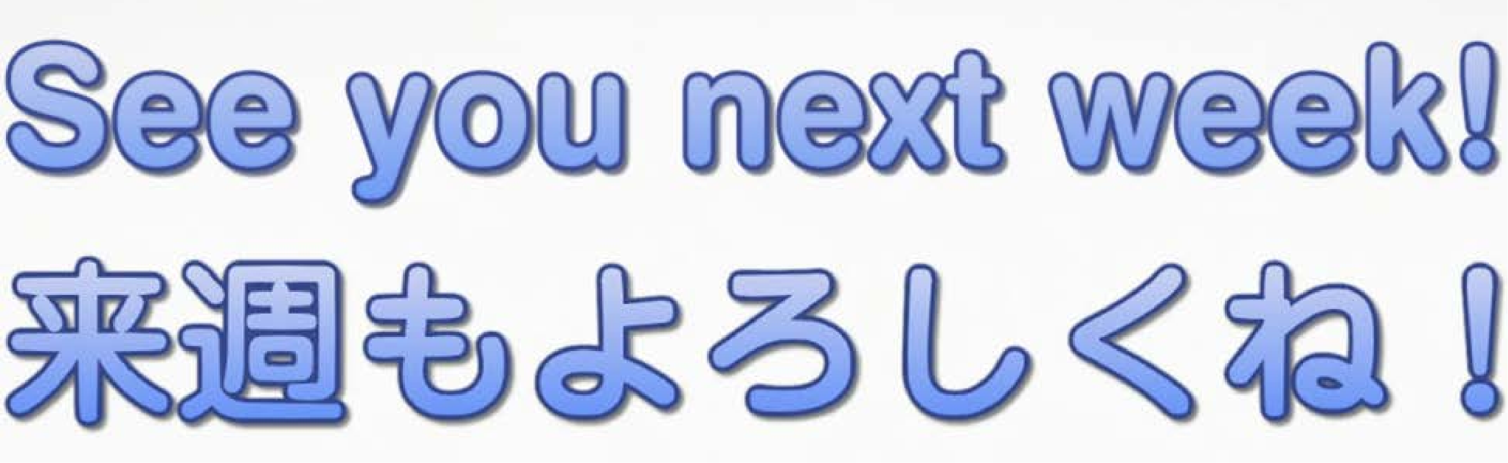
☑ c □ 2c □ 3c □ 4c
□ alpha □ 1a □ 2a □ 3a □ 4a
□ fscx □ fscy □ fax □ fay
□ frx □ fry □ frz
□ bord □ shad □ fs □ fsp
□ xbord □ ybord □ xshad □ yshad
□ blur □ be
□ org
□ Rotate in shortest direction

Acceleration: 1

Preset: No preset

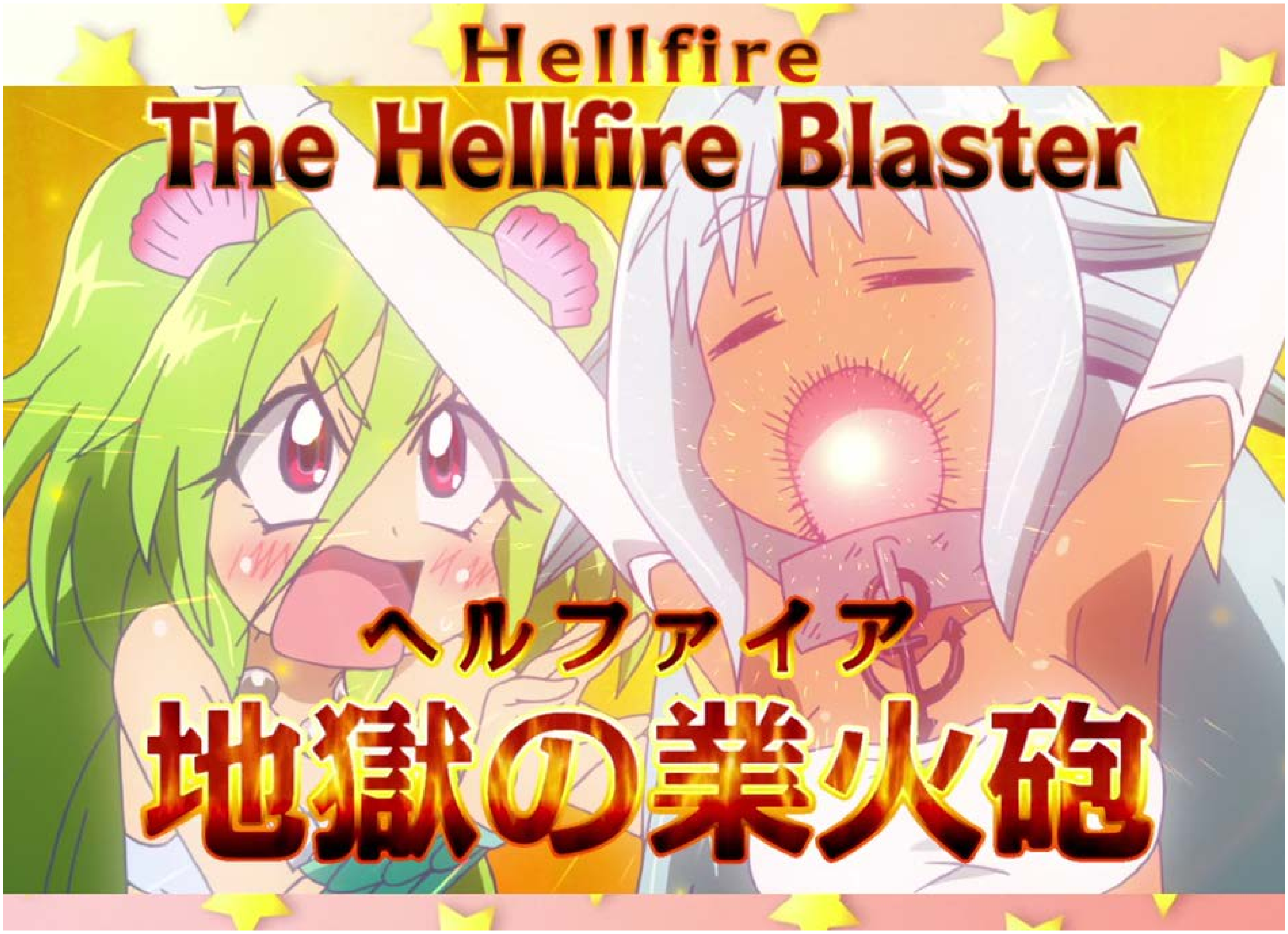
As you can see, with this script you can use gradient with a lot of things.

Simple gradients are really easy to make:



波打際の
むろみさん
Muromi-san
on the Shore

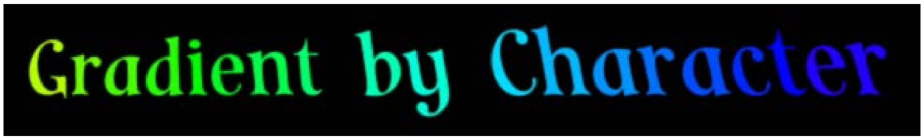
An example with gradients on multiple layers.



Aside from the obvious - colours - there's more going on here.
There's an extra, white layer, which transforms from alpha 00 to FF both up and down.
Since the white is only at the top half, the yellow border needs to be larger there than the red border at the bottom.
You can see the border difference on the JP sign too. So there's also a gradient for border on my sign.
By the way, you don't have to do all this with the gradient script. Do the basics and use "frame-by-frame transform" to tune the rest.

Gradient by Character

A different version of a gradient is by character, which means you get new tags for each letter. It's a horizontal gradient with only 1-letter precision rather than by pixels, but you can keep everything in one line.
Here's an example:



\fax and \fs is gradiented with lyger's 'Gradient by character'; colour is gradiented with my **Colorize** script.
lyger's can do all applicable tags. Colorize only does colours, but it has some extra options, like the HSB gradient used here, instead of RGB.
Of course this could then be combined with a vertical gradient.

HYDRA can gradient pretty much anything in various ways, so check the **gradient section** in the manual.

« Back to Typesetting Main

Tracking Signs with Mocha

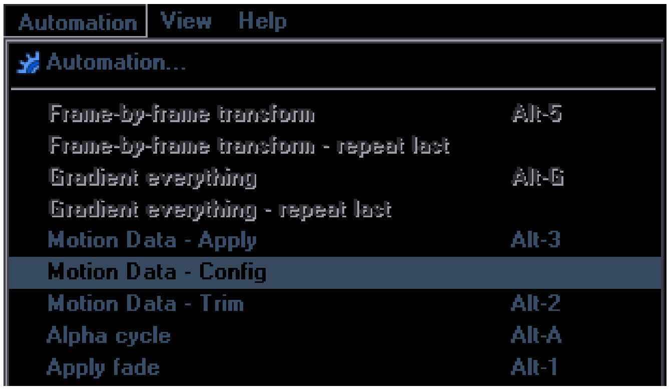
2015 Note: Even though I've updated this a number of times, stuff keeps evolving and changing.
Check this site for torque's updated guide to his Motion script.

What you need to install:

1. Mocha for After Effects. Any version should be fine. You don't need After Effects.
2. Quicktime or Quicktime Lite (Otherwise Mocha won't load mp4)
3. **Aegisub-Motion.lua**. Place it in \Aegisub\automation\autoload\.
4. **x264 8bit-depth** (Mocha can't read 10-bit)

Now you need a folder where the clips for mocha will go, and a folder where your x264.exe will be. To make it simple, use the same folder for both. Let's say we'll use D:\mocha\.

Open Aegisub and go to the Automation menu.


Automation
View
Help
Automation...
Frame-by-frame transform Alt-5
Frame-by-frame transform - repeat last
Gradient everything Alt-6
Gradient everything - repeat last
Motion Data - Apply Alt-3
Motion Data - Config
Motion Data - Trim Alt-2
Alpha cycle Alt-A
Apply fade Alt-1

First you need to set up the config.

If you chose "D:\mocha\" as your folder, you'll put that in the top part (don't forget the trailing slash),
and the same with x264.exe in the middle box, like you see here.
I recommend to uncheck the scaling and border/shadow for default settings.

Let's mention what all the options do now, but you can get back to this later.

Sort method: Default - If you apply data to 2 lines, you'll get line 1 tracked first and line 2 tracked after it.
Sort method: Time - Resulting lines will be sorted by time.

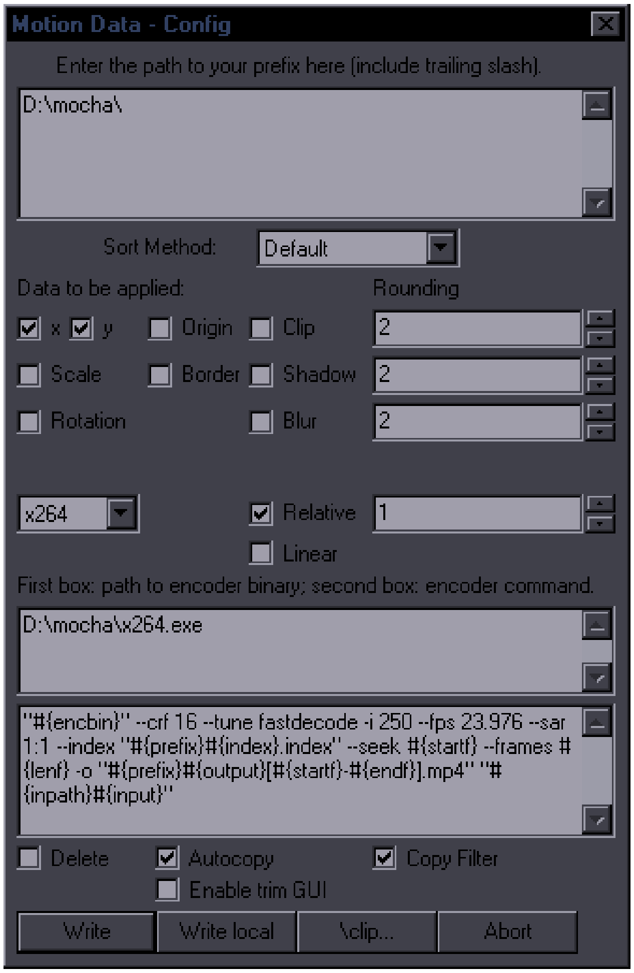
x, y - track \pos horizontally/vertically.
Origin - track \org.
Clip - track \clip.
Scale - track scaling/zooming, ie. \fscx and \fscy.
Border/Shadow - apply scaling to border / shadow.
Rotation - track \frz.
Blur - scale \blur.
Rounding - how many decimals you'll have for each tracked value.

x264 - I never thought of changing this, so... ???

Relative - This is a very useful thing. If it's hard to position the sign on the first frame, you can do so on the last frame and set this to -1. Or you can style the sign on frame 6 and set this to 6, etc.

Linear - this will use \move and \t instead of frame-by-frame tracking.

Delete - Delete the source lines instead of commenting them out.
Autocopy - Automatically copy from clipboard into the tracking data box.
Copy Filter - Only autocopy if the clipboard appears to contain tracking data.
Enable trim GUI - show GUI for trimming.



Enter the path to your prefix here (include trailing slash).

```
D:\mocha\
```

Sort Method: Default

Data to be applied: ☑ x ☑ y □ Origin □ Clip □ Scale □ Border □ Shadow □ Rotation □ Blur

Rounding: 2, 2, 2

x264 ☑ Relative 1 □ Linear

First box: path to encoder binary; second box: encoder command.

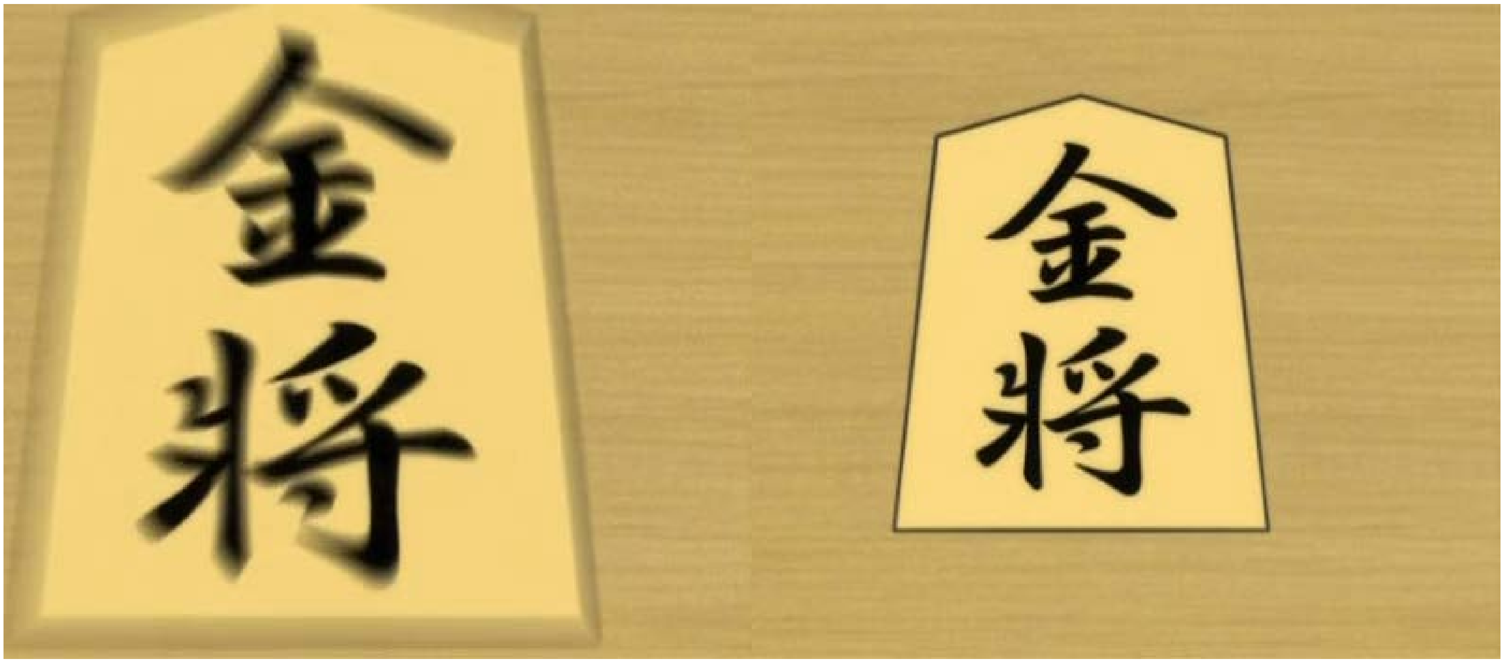
```
D:\mocha\x264.exe
```

```
"#{encbin}" --crf 16 --tune fastdecode -i 250 --fps 23.976 --sar
1:1 --index "#{prefix}#{index}.index" --seek #{startf} --frames #
{lenf} -o "#{prefix}#{output}[#{startf}-#{endf}].mp4" "#
{inpath}#{input}"
```

□ Delete ☑ Autocopy ☑ Copy Filter □ Enable trim GUI

\clip... - This allows you to use different tracking data for clip than for the sign itself. (Remember this. You'll need it one day.)

Let's look at a sign that needs tracking.



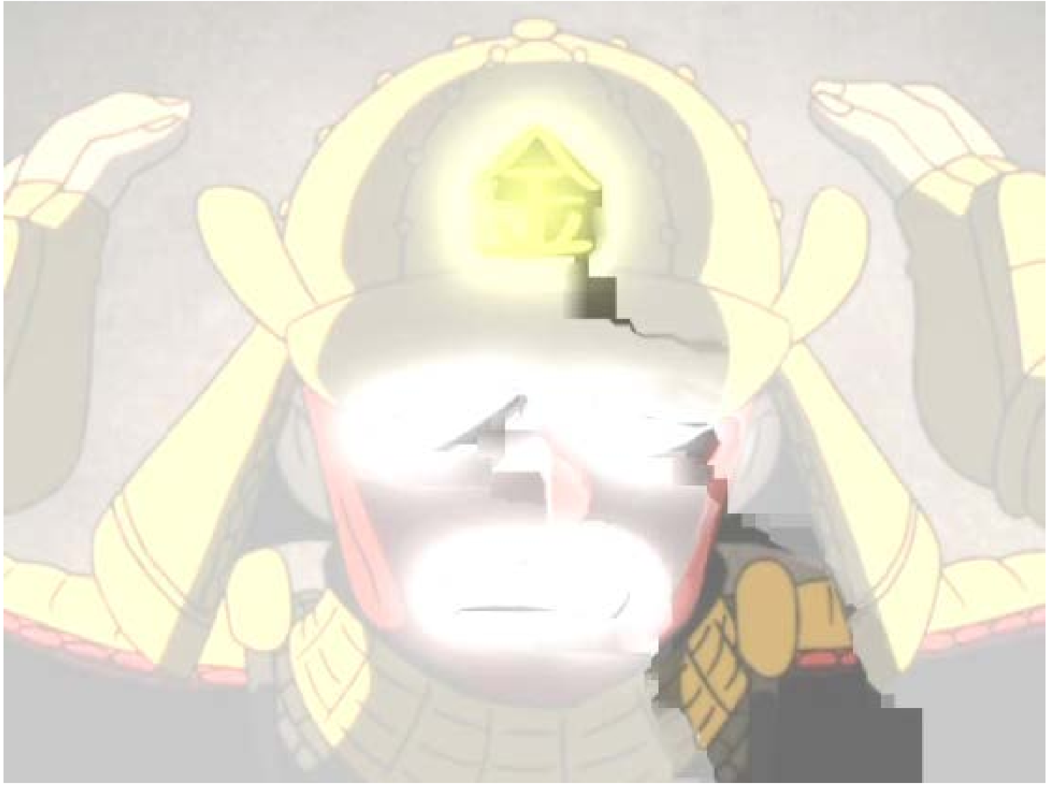
This is what it looks like on the first frame...and this is the last frame.

This is the kind of sign that's easier to track from last frame, and it's better to create it on the last frame too.
So we'll start with this:

Time to encode a clip for mocha. Click on "Motion Data - Trim" in the automation menu, and it'll just encode without needing any further action.

If it didn't work and it's your first try, restart Aegisub and see if it works after that.

If it worked, then you can skip this next section. In case it doesn't, I'll explain **an alternative method for encoding the clips.** What sometimes also happens is that the script encodes a clip that looks like this:



You can try encoding from a previous keyframe, but sometimes even that doesn't help, so I use this other method. (Though in some cases the sign is still trackable even on that shitty encode.)

Create mocha.bat with this text in it:

```
x264 --profile baseline --level 1.0 --crf 18 --fps 24000/1001 --seek %2 --frames %3 -o %1_%2.mp4 %1
```

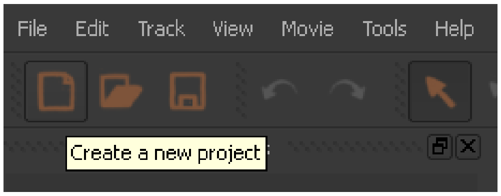
This is with 23.976 fps, so obviously change if needed.
Put the mocha.bat where your premux is, along with x264.exe.
Open cmd.exe and navigate to the folder with these files (or open it from there in the first place).
Then type this command:

```
mocha seki02_premux.mkv 4193 40
```

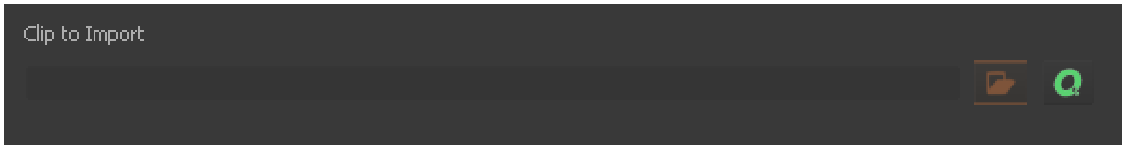
"mocha" is mocha.bat, seki02_premux.mkv is the video you're encoding from, 4193 is the frame where it starts, and it will encode 40 frames.
It's actually 33 frames, so you can be precise if you want, but I think you need to encode one more than it shows in Aegisub.
I usually just encode more and then move the red marker for the end frame in mocha.

2015 Note: Another encoding alternative is my **Encode - Hardsub** script. (But hopefully torque's script is reliable now.)

When you have your clip encoded, start up mocha.
Open a New Project:



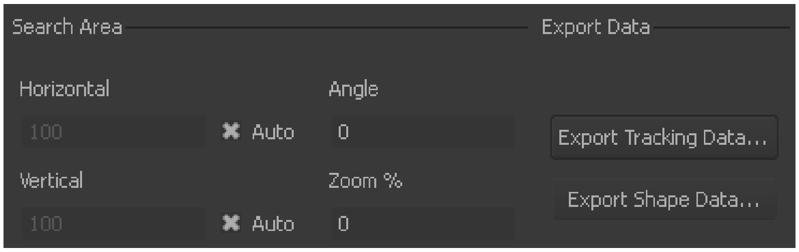
Import a clip:



After you select your clip, hit Enter 4 or 5 times, until all the menus go away. (No need to change anything there.)

NOW YOU TRACK THE SIGN. (I'll get back to the tracking itself in a bit. Let's just finish the whole process now.)

To export the data, you click on "Export Tracking Data",


Search Area

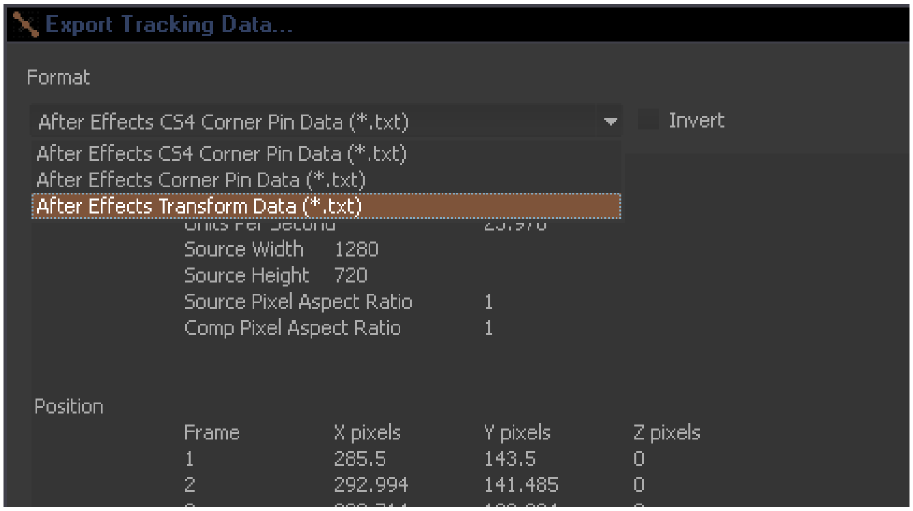
Horizontal: 100 ☒ Auto

Angle: 0

Vertical: 100 ☒ Auto

Zoom %: 0

change the format to "After Effects Transform Data",



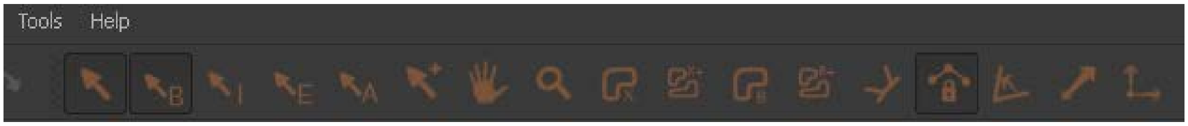
```
Export Tracking Data...

Format

After Effects CS4 Corner Pin Data (*.txt)            Invert
After Effects CS4 Corner Pin Data (*.txt)
After Effects Corner Pin Data (*.txt)
After Effects Transform Data (*.txt)
                    Source Width     1280
                    Source Height    720
                    Source Pixel Aspect Ratio        1
                    Comp Pixel Aspect Ratio          1


Position
                    Frame            X pixels         Y pixels         Z pixels
                    1                285.5            143.5            0
                    2                292.994          141.485          0
```

and click on "Copy to Clipboard".

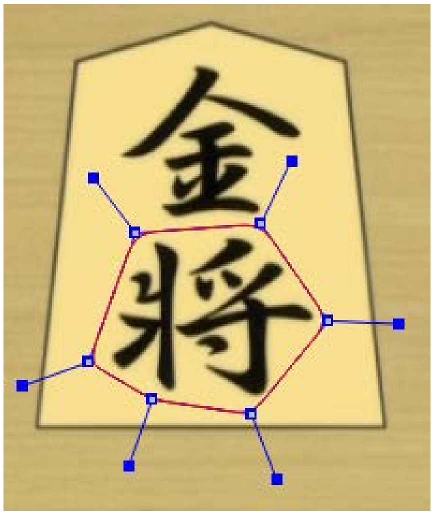
Move over to Aegisub, Automation menu, Motion Data - Apply.
If the data isn't autopasted, paste it into the top box. I explained the options earlier, so check what you need, and click "Go".
Tracking data gets applied to the selected lines, and you're done (as long as you didn't fuck anything up).
That's all regarding the Aegisub-Motion.lua. Now for Mocha itself...

Tracking a Sign in Mocha

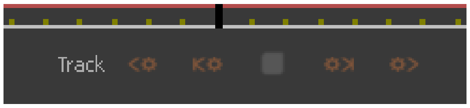
So you have your video loaded. Now you need to create a spline to track.



I'm using an old version, but the new ones shouldn't look all that different.
You will mostly be using the icon with an X (9th from left). Next to it is X+, then B, and B+.
Click on the one with X, and draw a spline like this:

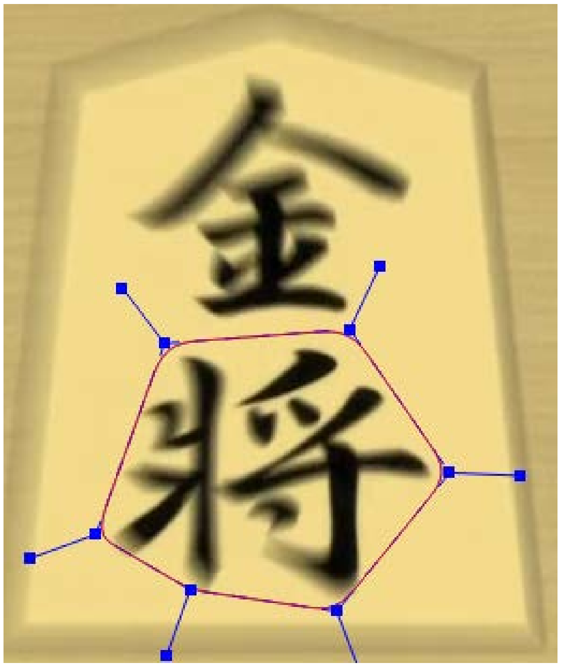


Now you can start tracking. There are 4 buttons you can use...

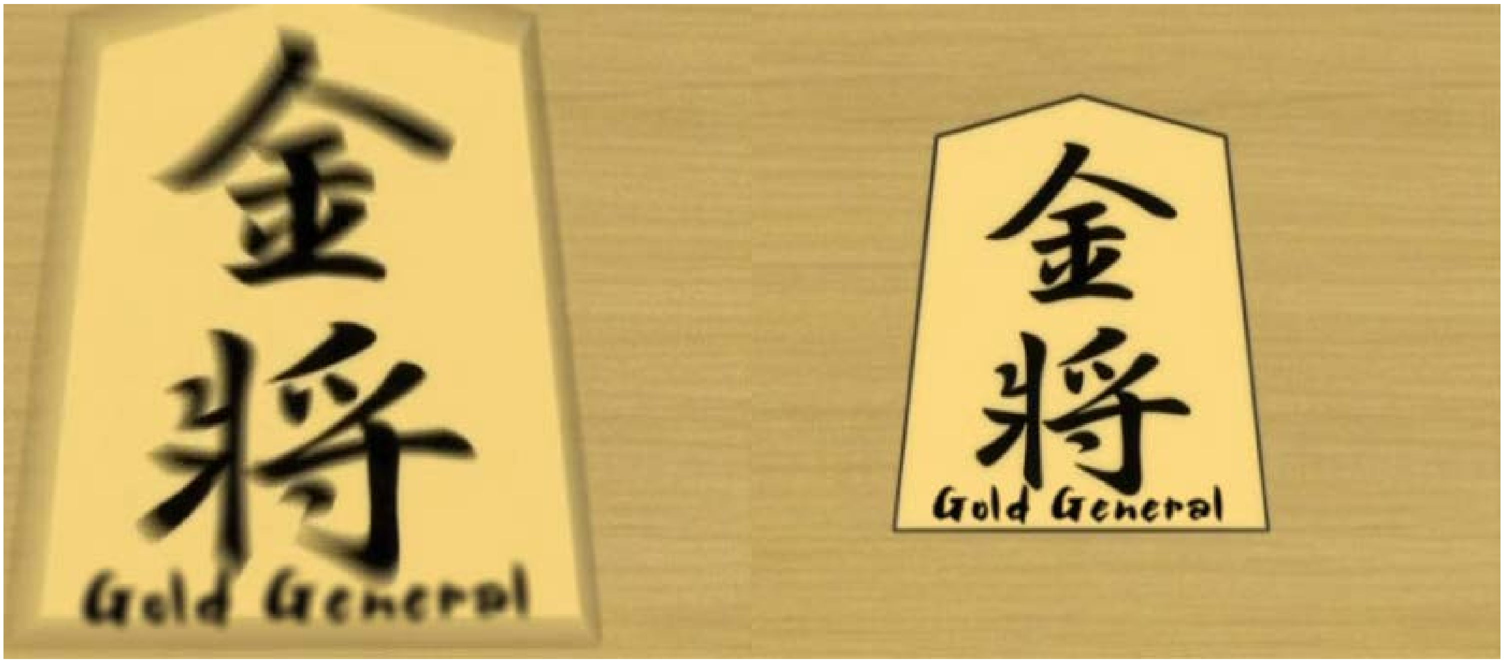


Track backwards | Track to previous frame | Track to next frame | Track forwards

It doesn't matter in which direction you track or where you start from.
We're tracking this one from the last frame, and on the first frame we should end up with something like this:



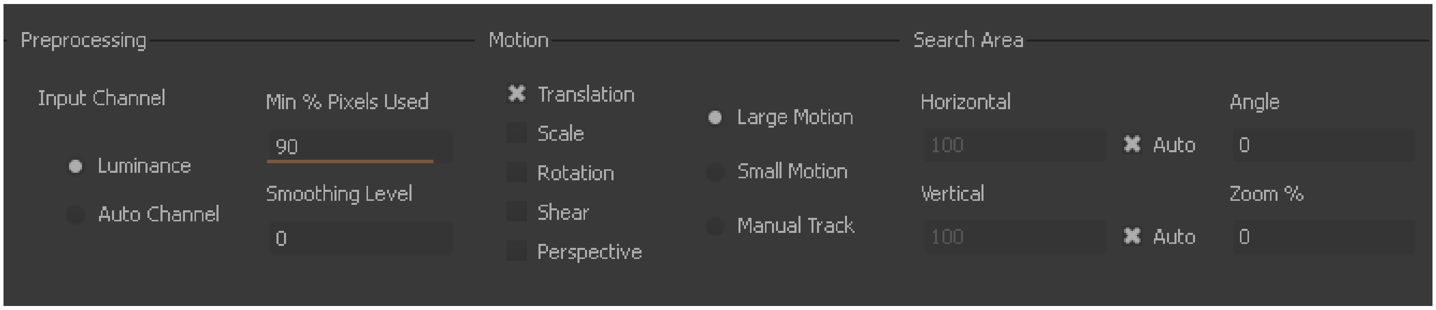
And when you apply the data, the first and last frame will look like this:



You can try scaling the blur with the script, but it may not match the size scaling, so I usually change the blur by hand. Normally it's only a few frames that need more blur. This sign had some rotations in the middle part, so we track and apply position, scaling, and rotation.

For simple signs, you often don't need anything else than to draw a spline and track.
If there's no scaling, things usually work out fine.

If the tracking isn't going well, you have to tweak the options here:



Preprocessing

Input Channel

● Luminance

○ Auto Channel

Min % Pixels Used: 90

Smoothing Level: 0

Motion

☒ Translation

□ Scale

□ Rotation

□ Shear

□ Perspective

● Large Motion

○ Small Motion

○ Manual Track

Search Area

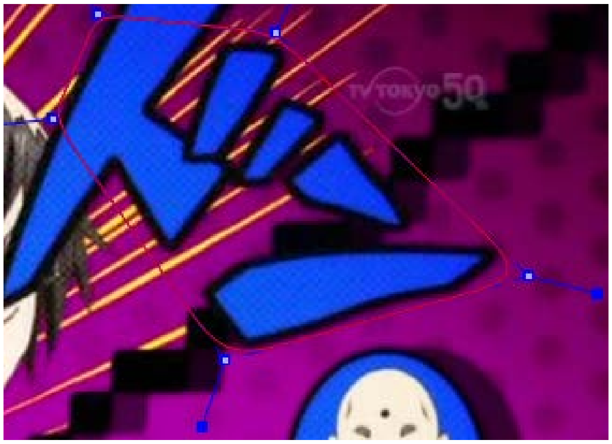
Horizontal: 100 ☒ Auto

Vertical: 100 ☒ Auto

Angle: 0

Zoom %: 0

Luminance / Autochannel: Luminance follows overall brightness/contrast. Autochannel picks red/green/blue channel, depending on which has more contrast (or so I imagine it works). Usually Luminance works, but then there are signs like this:



The blue sign is jumping around, but so are some other lines in the background, so the whole thing can be pretty messy, and since the blue and purple have about the same brightness, they're not so easy to distinguish, so Luminance would mainly see the black lines, but since there's some black in the background too, it can influence the tracking. If you choose Autochannel, mocha will (I assume) track the blue channel since it shows clear contrast, and the thing you need to track is all blue and the background isn't, so it works out well.
I have no way of knowing what mocha actually does track, but for example for this sign switching to Autochannel helped, so I imagine it works something like I explained. But really all you need to know is that if you fail to track a frame correctly and you think it might be related to colours, undo the frame, switch to Autochannel and try again. Either it'll help, or it won't.

Min % Pixels Used: To be honest, I don't remember a single time when changing this value would help me track anything.
If your spline is going off, you can try increasing or decreasing this value and see if it helps, but I wouldn't bet on it.

Smoothing Level: I have no idea what this does, and I've never heard anyone mentioning it either, so probably not important.

Translation: This equals position and is always checked.
Scale: Check this if the sign gets bigger/smaller, ie. zooms, etc.
Rotation: Check this if the sign rotates like what \frz does.

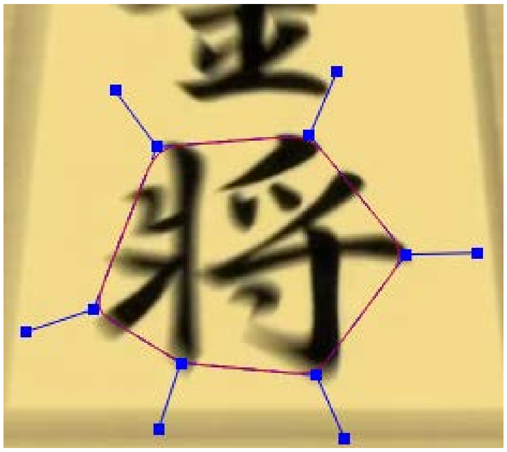
Do NOT check the options you don't need, because they'll only make tracking harder in such a case.
The other two options are not applicable for Aegisub, so don't check those.

Large / Small / Manual... I almost never change this. Just know that you can track manually, adjusting the spline on each frame by hand.

Horizontal / Vertical: If a sign makes long-distance jumps, like a fast pan, you may need to increase this value. On the other hand, if the spline jumps off somewhere far when it shouldn't, you can set smaller numbers to limit the area it searches. With simple, continuous, short-distance movement, Auto usually works.

Angle / Zoom: Similarly to previous, use larger values if there's more zooming/rotations, and smaller values if the spline is going too crazy.

That would be the basics of tracking. Now more about what if it's not working and what tricks you can use.

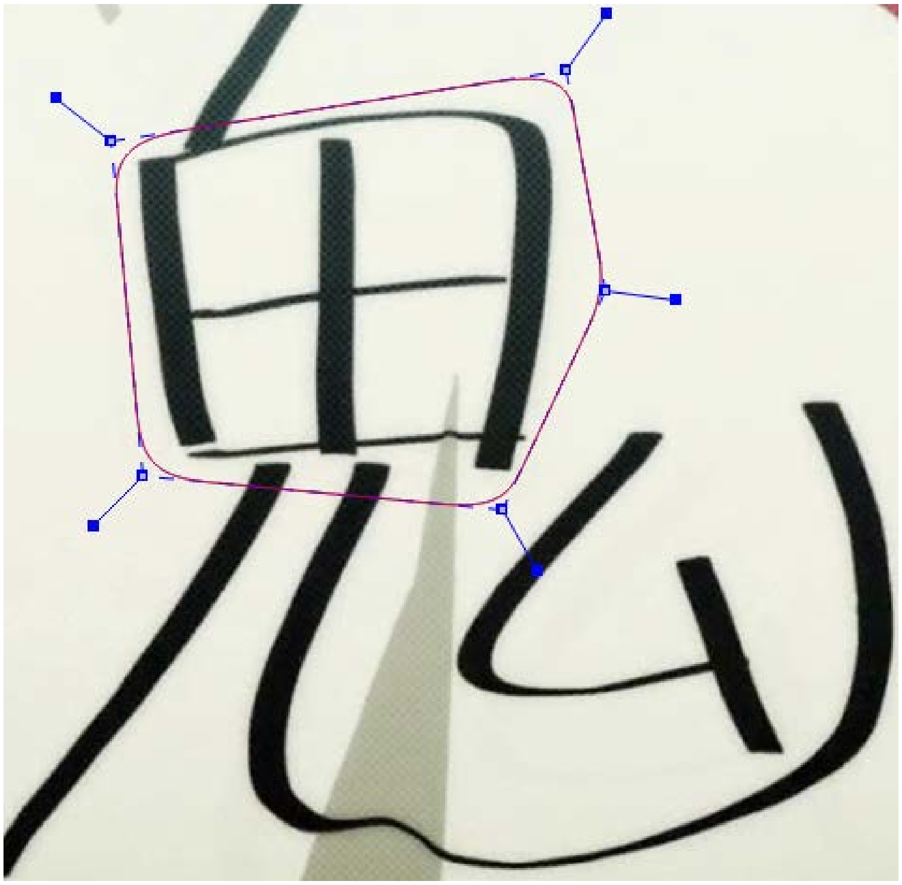


If you compare to the previous pictures, you'll see this spline is too small. That means it didn't scale enough, so you need to increase the Zoom value. It should supposedly be in %, but from experience I can tell you that that's bullshit. I have no idea what exactly the number is, but larger value allows for more scaling, so you just have to try. If the difference seems small, try 10. I usually get the results I need with values between 10 and 20. If it's zooming a lot, then maybe 30. You probably won't need to go over that.

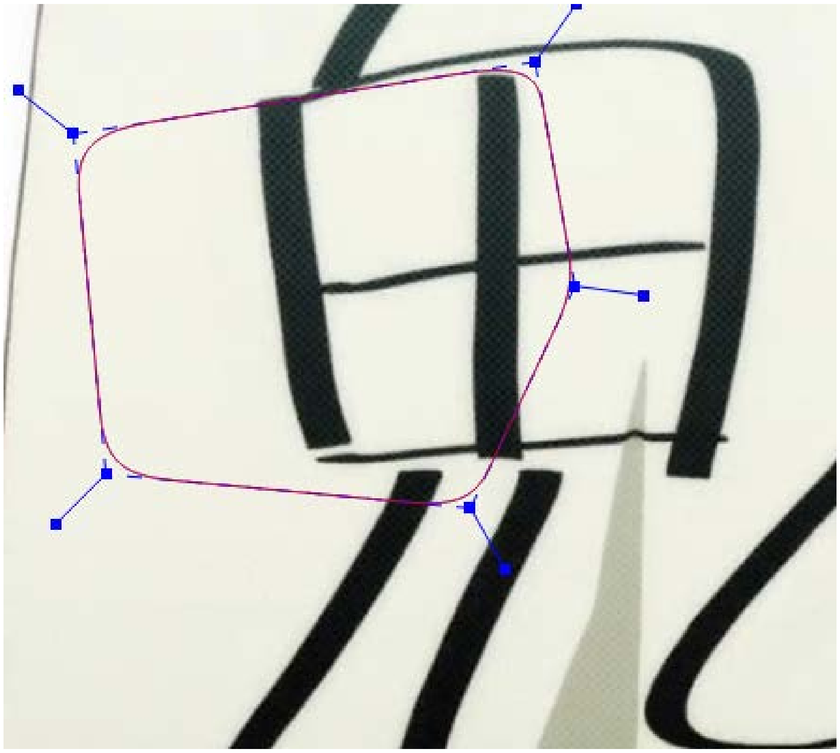
If you set some value and the spline actually gets bigger than it should, decrease the value bit by bit until it seems right. It's a bit of a trial-and-error game, but after tracking 100 signs you'll have some idea from experience what values you should use.

Tweaking the Angle for rotations follows the same principles as Zoom.

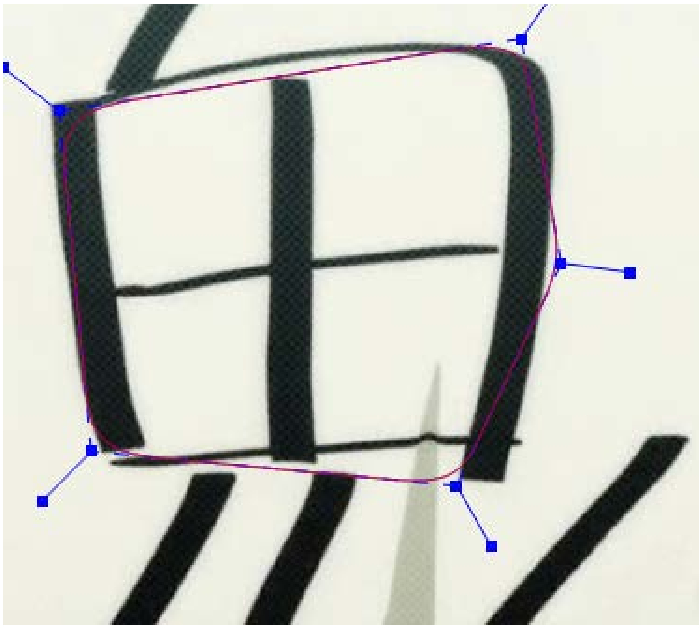
Here's an example of what usually happens with zooming signs, and how to fix it.



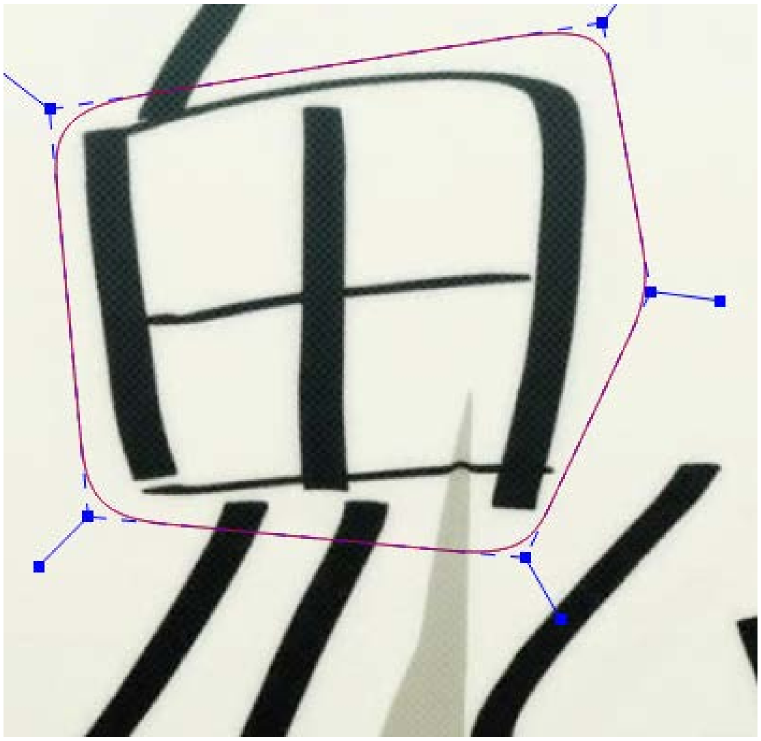
This is the starting spline.



This is what happens with default settings. (1 frame tracked.)
You can't see the whole frame, but the sign hasn't really moved much. It just got bigger.
Since the spline is jumping far off, we can limit the movement, so I set Horizontal/Vertical to 20. Ctrl+Z; retrack.



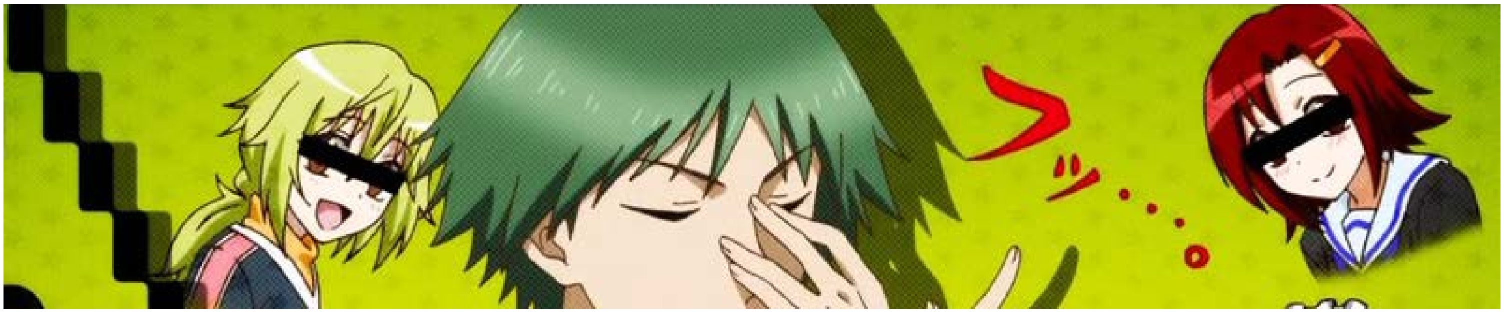
And this is what I get. You see it didn't scale enough, but limiting the movement helped to keep the spline in the right area.
So I change the zoom to 20. Ctrl+Z; retrack.

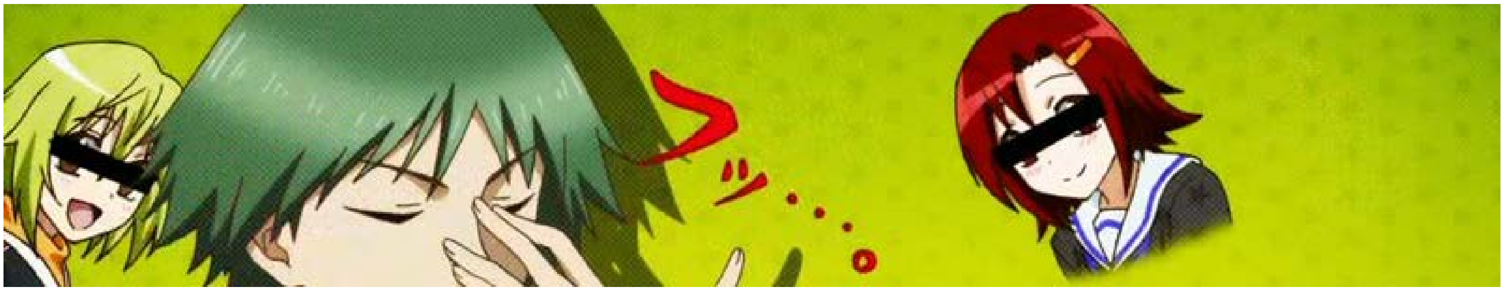


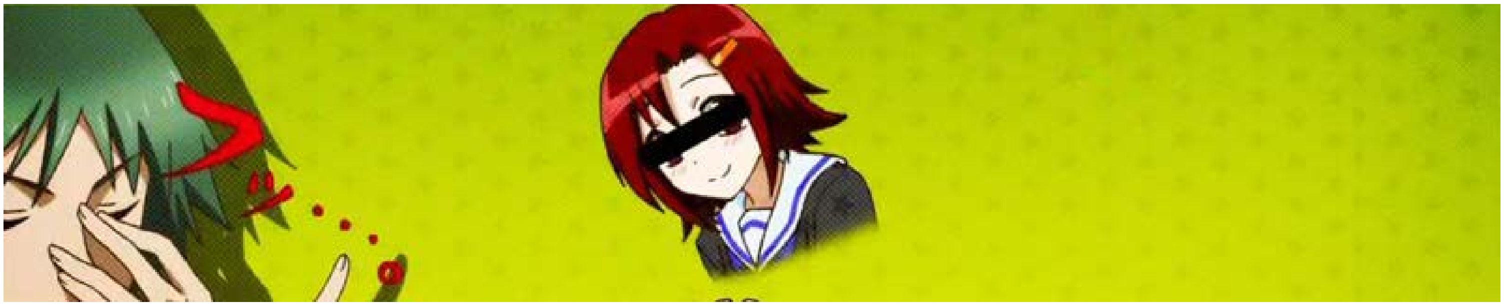
And here we go. I got exactly what I need.

If a sign changes on one end more than on the other (zoom, blur, etc.), it's probably easier to track it from the end that changes less, and leave the most difficult frames as last. Or, if needed, you can track it from the middle to the ends. This helps for example with fades.

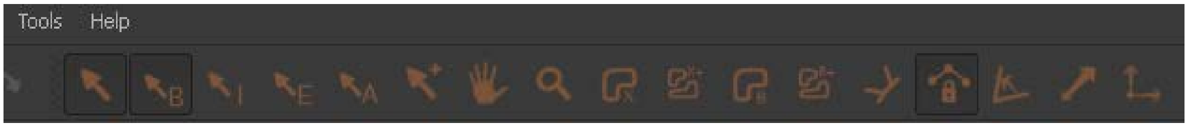
Here you need to track the red sign, moving from right to left...





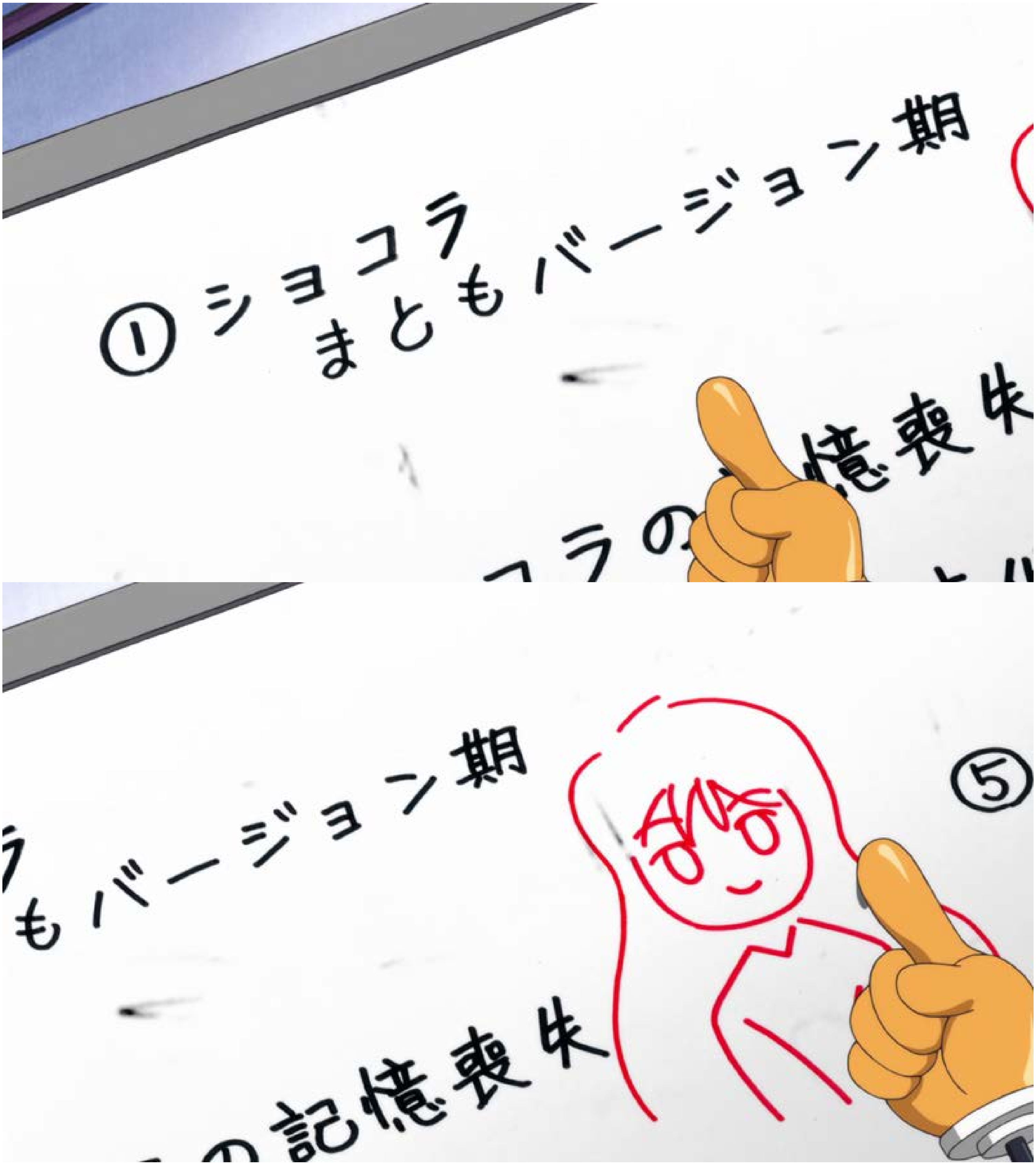


The sign jumps in fast, then moves slowly across the middle, and then jumps out fast again.
It moves horizontally a lot, and vertically a little. The first two and last two pictures are the actual first and last two frames.
The middle picture is where it slows down for about 10 frames.
Tracking from frame 1 to frame 2 would be hard because there isn't much to select on frame 1, so it's better the other way.
The main thing about this sign is that you need to set Horizontal to the max - 500, and Vertical to something like 50.
Less Horizontal won't track far enough, and more Vertical will only give mocha more useless area to search and possibly confuse it.

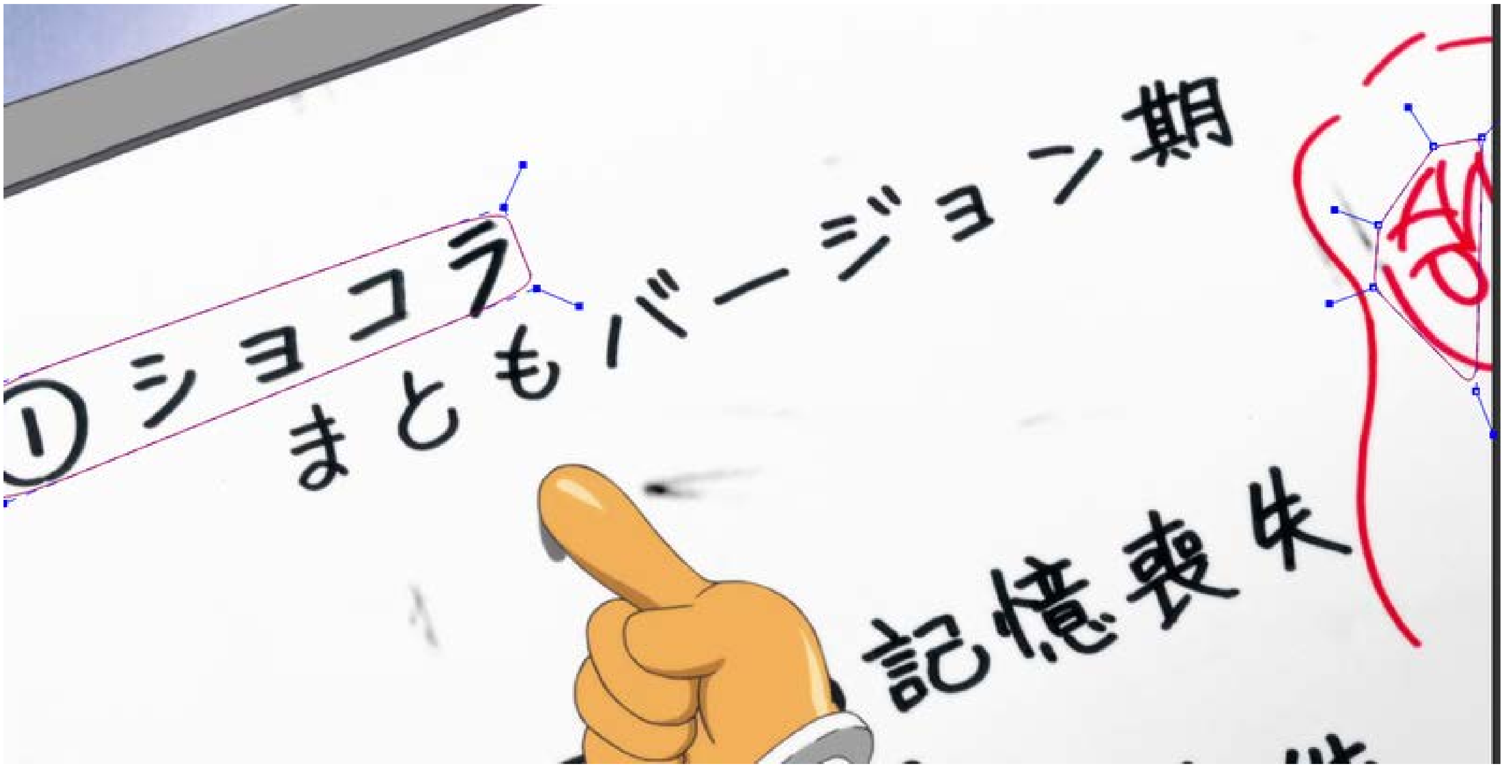


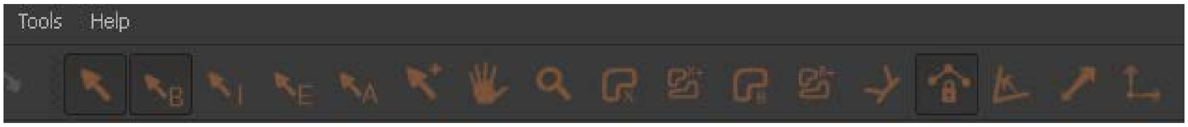
Remember those X, X+, B, and B+ icons? The B ones are for Bezier curves. It's usually faster to just make many points with the regular X one, so you can pretty much ignore the Bs. But what about the X+? That one is very useful. It allows you to add another spline to the one you're already tracking, or even replace the old one, like when it starts going off the frame.

Let's say you have a pan like this:

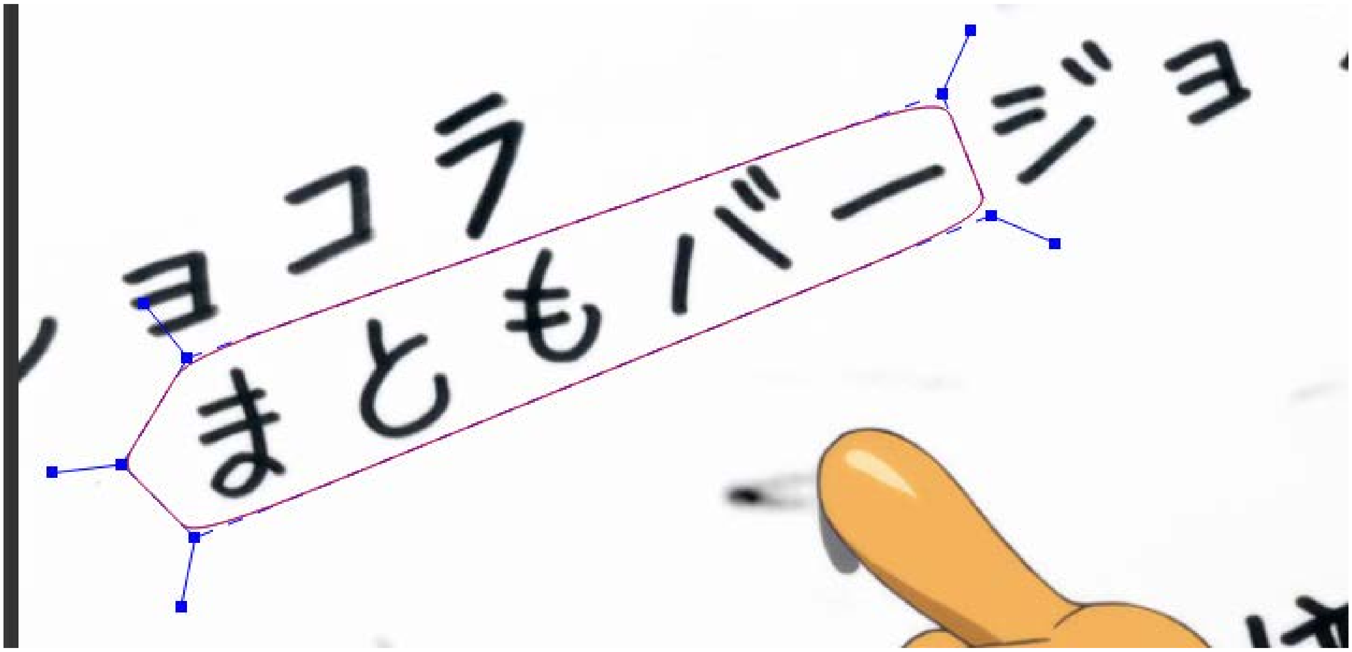


You can start with tracking the text on the left. When it starts going off the screen, you use the X+ tool, and draw another spline on the right. You can either continue tracking like that, or you can now select and delete the left spline. (You have to draw the new one first before you can delete the old one.)



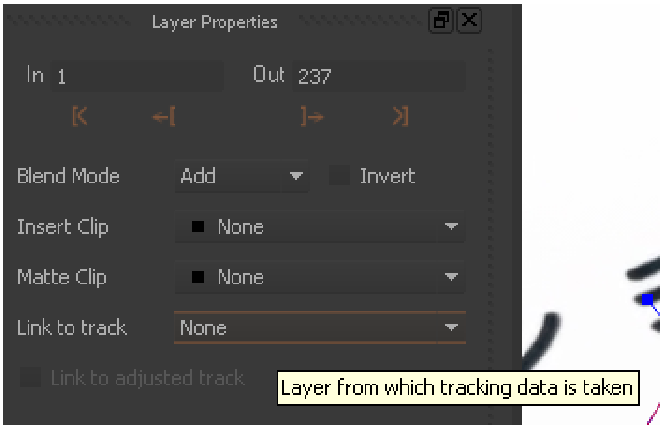


Alternatively, you can press the Q key (or use the icon on the far right) and move the spline to a new place:



And you can continue tracking.

Another trick for whole-screen pans is this:



Layer Properties

In 1 Out 237

Blend Mode Add □ Invert

Insert Clip ■ None

Matte Clip ■ None

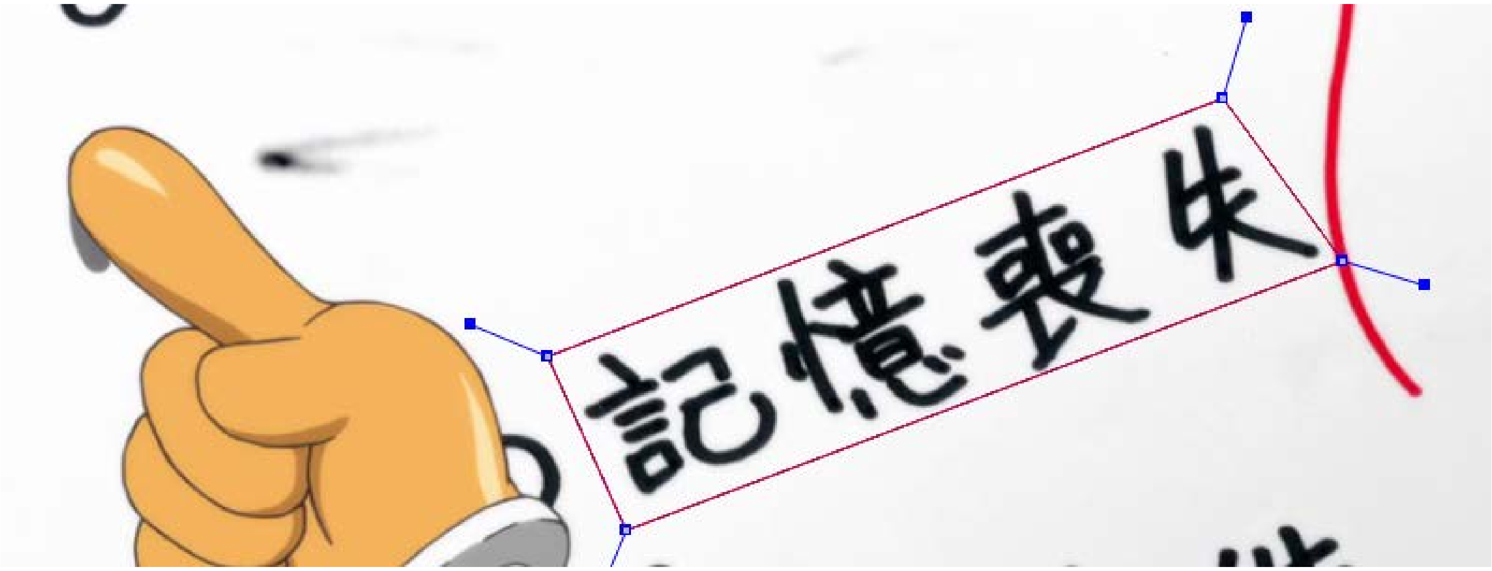
Link to track None

□ Link to adjusted track

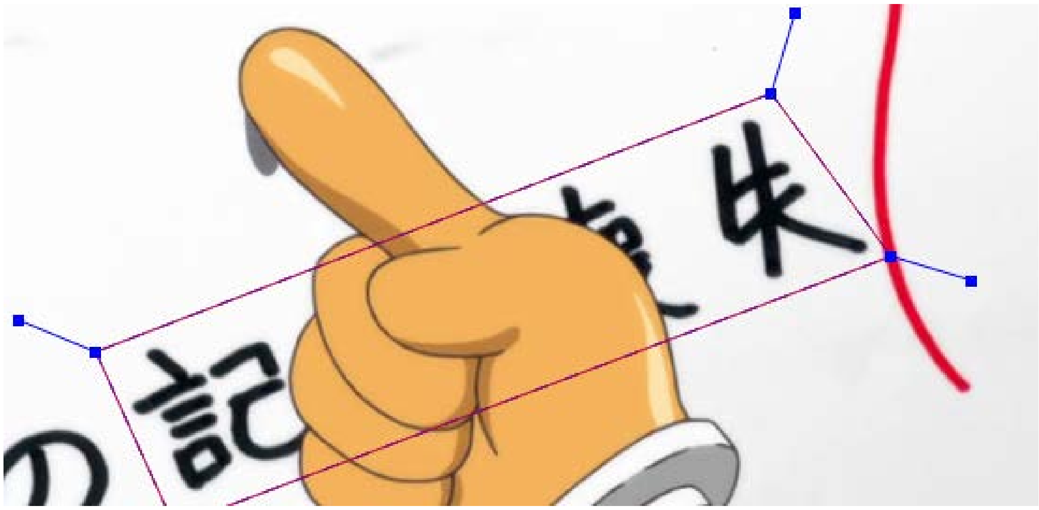
Layer from which tracking data is taken

Set "Link to track" to "None" instead of the layer you're tracking. If it's a whole-screen pan, you can draw the spline anywhere, even over the whole screen. The spline won't move when tracking, but it tracks the pixels moving within it, so when you have very long pans, it tracks them well while the spline stays in place.

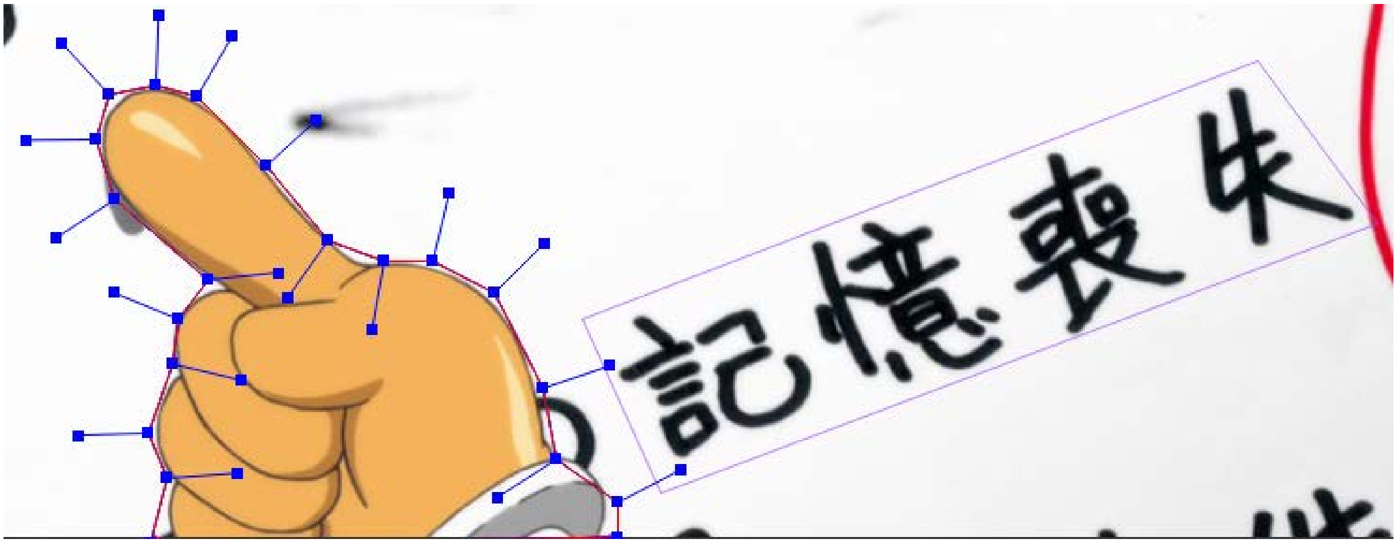
Sometimes things are in the way of the tracked sign.



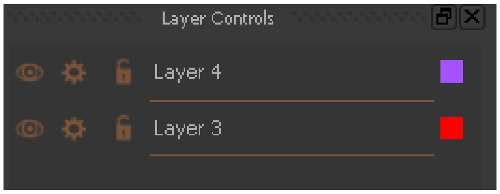
You want to track this sign...



...but it goes under this hand. In this case it actually tracked perfectly well, but that's not always the case.



Use the regular X tool to draw a new spline around the hand.

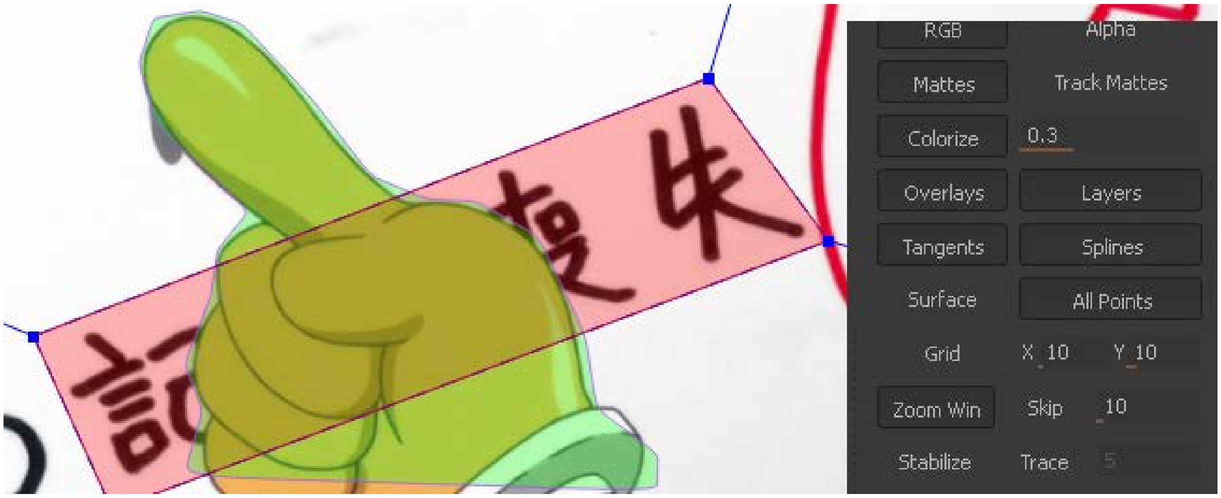


Layer Controls

Layer 4 🟪

Layer 3 🟥

New layers will be placed above the old ones. When they overlap, the bottom layer will not track areas that the top layer is "blocking". If the hand doesn't move, you can click on the middle icon next to "Layer 4" so that it doesn't track the layer. If the hand moves, you can just track with focus on layer 3. Layer 4 will track the hand, and layer 3 will track its spline except the areas covered by layer 4.



RGB | Alpha

Mattes | Track Mattes

Colorize 0.3

Overlays | Layers

Tangents | Splines

Surface | All Points

Grid X 10 Y 10

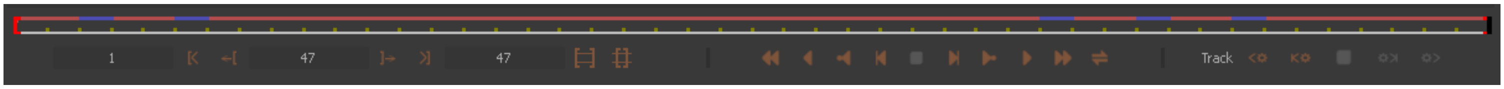
Zoom Win | Skip 10

Stabilize | Trace 5

If you enable the "Mattes" in this menu, you'll see the splines filled with colours. Layer 3 will track only the red area here.
So for example if you have a slow pan of a building with a sign on it that you're tracking, and a car goes by in front of the building, you can make a new spline for the car to make sure the car won't throw off the sign you're tracking.

While we have this menu here, you can also notice the Stabilize button. If you click on that, the focus won't be on the whole frame, but on your sign, so the sign will stay in place while the rest of the stuff moves around it. When you "play" the tracked sign with Stabilize on, you can see more easily how well the spline sticks to the sign.

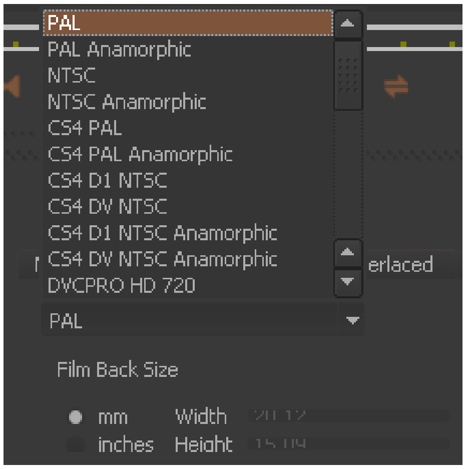
If a sign lasts 30 frames and moves only on the first 5, you don't need to track the remaining 25 frames. Just track the 5 and apply the data. For the typical Japanese animation where stuff moves only every 3rd frame, you can just track every 3rd frame.



This is tracking data that I actually used, since the sign only moved on those 5 frames. No need to flood the script with hundreds of static lines, plus the tracking can actually make the sign wobble a little, while this way you ensure that it doesn't move on the static frames.

While we have this image here... If the sign's duration is shorter than the whole clip, you can move the red markers at the ends to set the start and end for the tracked data. The exported data is only what's between the red markers.

A problem you may run into is that the video loads stretched vertically in mocha, as in, it shows wrong aspect ratio. If that happens, look to the bottom-left part of the screen, and switch from Track to Clip. Then instead of those things like Search Area, you will see other stuff, including this dropdown menu:



Changing to one of those Anamorphic will probably do the trick. If not, just switch to whatever will look right.
(Press * to centre the screen after.)

That's about all I can think of now. This should explain how it all works. Learning to track difficult signs is largely a matter of experience, so you have to practice. This guide is mainly to explain how to make a video clip, get it into mocha, track a sign, and get the data back to Aegisub, so you should now be able to do that.

For more help, you can check some mocha videos on youtube and other places.

« Back to Typesetting Main

Typesetting: Fonts

Seeing from the results produced by typesetting apprentices, choosing fonts seems to be where they tend to fail.
The main problem being that they don't really **have** any fonts aside from whatever is installed by default.
So first you need to get at least some small collection of useful fonts.

You can get those from various places. Obviously there are websites with fonts.
Here are some you can try out: **ufonts** | **fontspace** | **azfonts** | **fontsner** | **dafont** | **ffonts**
You can also extract fonts from decently typeset fansub releases, which is not necessarily as dumb as it sounds,
since you'll easily get to a lot of fonts that are actually suitable for anime typesetting.
Or you can do what I do - download large font packs from torrents, go though thousands of fonts, delete all the useless ones
(yeah, about 80%) and sort the useful ones so that you know what you have and where. Obviously this takes the most time,
many hours, but once you get through that, you'll have much easier time finding what you need when you're typesetting.
In other words, takes time at first, but saves you time later. If you need a specific kind of font and you alreeady have 300 fonts installed
that you have selected as good and suitable, it'll be easy to find one quickly. If you have to go to a website and try looking for something
that would work searching by keywords, it might take long and produce poor results.

2018 Note: Seriously, sorting your fonts in folders by type makes things much easier later. Having a good font browser helps too. (Font Navigator, FontViewOK)

You can certainly typeset with 20 fonts and get away with it. You will certainly get better results with 400 fonts to choose from.
So it depends on what you're aiming for and how dedicated you actually are to this.

Before we get to what to use, let me make a few comments on what NOT to use.

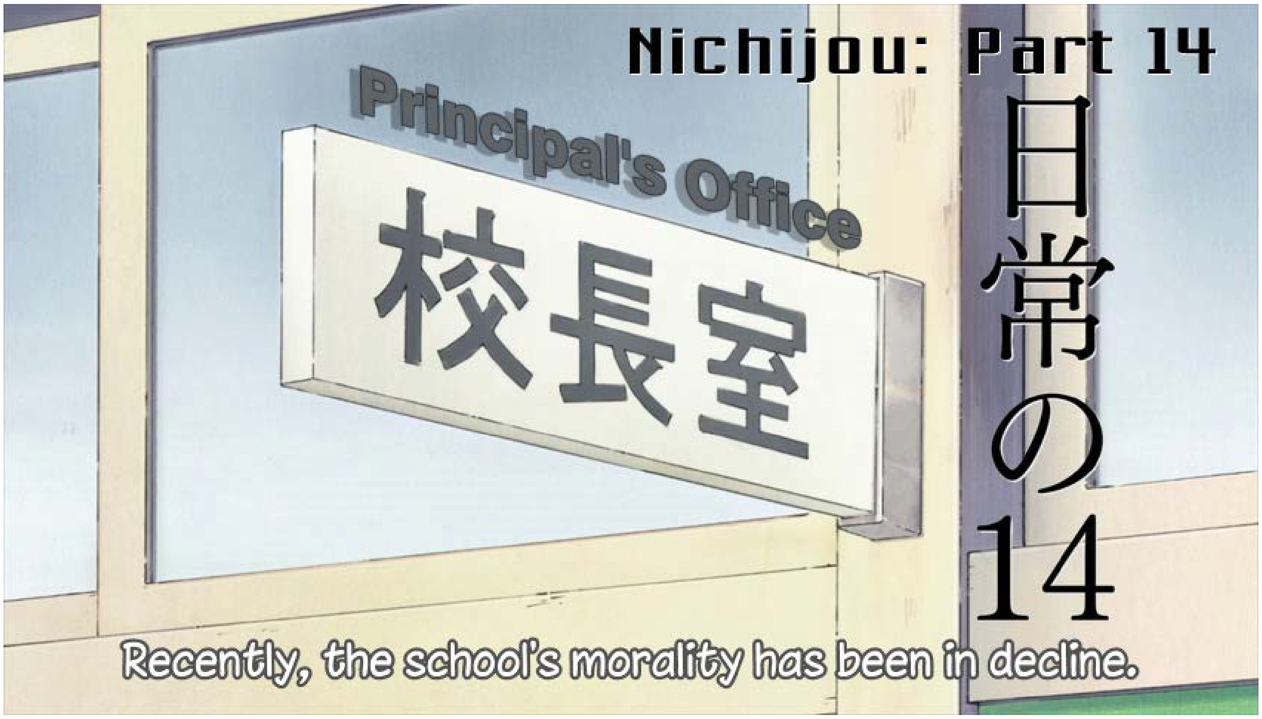
• Do not use 10 MB fonts!
It's stupid to increase a 200 MB mkv's size by 10 MB with just a font.
Mainly for the reason that just about any 10 MB font can easily be replaced with a 100 KB one. 10 MB fonts are only justified for kanji.
My general rule is to not use any fonts over 1 MB. Even fancy decorative fonts are mostly under 1 MB.
Larger fonts are simply bloated with crap you don't need. Not that 2 MB would really be a problem,
but just get used to checking the size to make sure you don't pick those 12 MB ones for no good reason.

Which leads to another point...

• Avoid system fonts, fonts starting with @, MS, Adobe etc. fonts.
Not that they're all bad, but those often tend to be the large ones, and there are always plenty of similar fonts with sane filesizes.
Aegisub has those @ fonts at the top of the list, so beginners will choose them 'cause they're first.
Just learn to skip those every time you're choosing a font. There are no "good" ones among them anyway, trust me.
They're only useful if you need kanji. Even if they happen to "fit" what you need, you can always find a similar one that isn't bloated.

• Avoid notoriously known fonts like Arial, Times New Roman, ComicSans etc.
If you use ComicSans, you will break rizon servers because they will get flooded with >*ComicSans* messages.
Times New Roman might work sometimes, but it's obviously a severly overused font, and you can always find other ones of that kind.
As for Arial, pretty much no Japanese sign is so ugly that you could imitate it with Arial.

• Basic Serif & Sans Serif fonts... [if you don't know what that means, look it up]
These are not suitable for most signs. Serif fonts may be useful for titles, as seen in the basics chapter.
Basic Sans Serif fonts may be useful for cellphone/e-mail messages, or signs that clearly have no decorative elements whatsoever,
like the Principal's Office sign here:



You will, however, need something thicker than Arial [even when bolded] for this, usually ones with Black or Blk in the name.
For most signs you'll want to avoid these basic fonts, as it will look more like you're "putting text on screen," rather than typesetting.
The typesetter's job is to make the signs look good. If we wanted to just put text on screen, we'd do it ourselves instead of hiring you.

So what WILL you actually need?

- **Some better looking [Sans]Serif fonts**
That means ones that aren't as plain and ugly as Arial and TNR.

- **LOTS of handwriting fonts**
Handwriting is all over anime, so the more of these you'll have, the better.

- **Some brush / calligraphy fonts**
Kanji works great with brushes and the Japanese know it.
If you want the typesets for such signs to look really good, you'll need some nice fonts of this kind.

- **Round / rounded fonts**
Some with the whole letters being round, some where just the ends are rounded. [Note: Arial Rounded MT Bold looks like shit]

- **Square fonts**
...when they use square-ish kanji, obviously.

- **Chalk / pencil fonts**
Where would anime be without signs on school blackboards. Better have Eraser Dust and some others ready.

- **Fonts with eroded outline**
Meaning ones where the outline isn't clean but kind of jagged and stuff.

- **Some thick / thin / wide / narrow ones**
This might as well apply to each category separately. Using \fscx150 doesn't exactly produce the same results as having an actually "wide" font.
Sometimes extremes are needed to either match the original or to fit the sign where you need it.

- **All kinds of deformed and distorted fonts**
Not needed most of the time, but if you want to do a really good job on a SHAFT show, it'll be useful to have all kinds of things around.
Blood splatter, ripple effect, various patterns and textures, lightning and wobbly shapes.
No need for beginners, but it might still be good to have a few of those around.

- **Cartoonish fonts**
These are quite common, so I'd say have as many as possible.

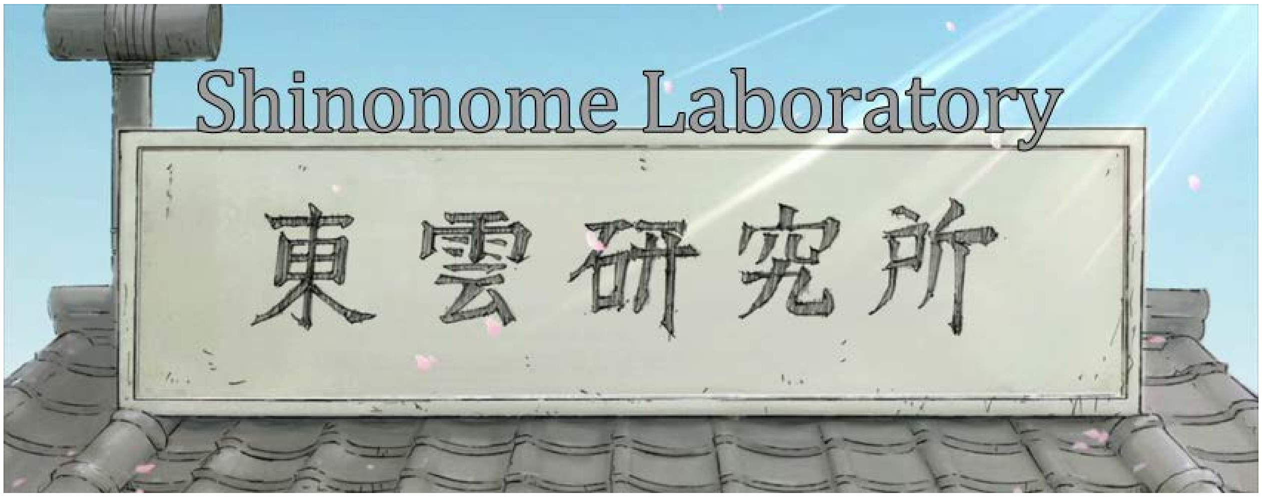
- **Decorative fonts**
Some script fonts and whatever fancy stuff you can find. It is kind of hard to predict what you will need, but good to have a few lying around.

- **Digital looking fonts, ones made out of dots, pixelated ones, typewriter fonts etc.**

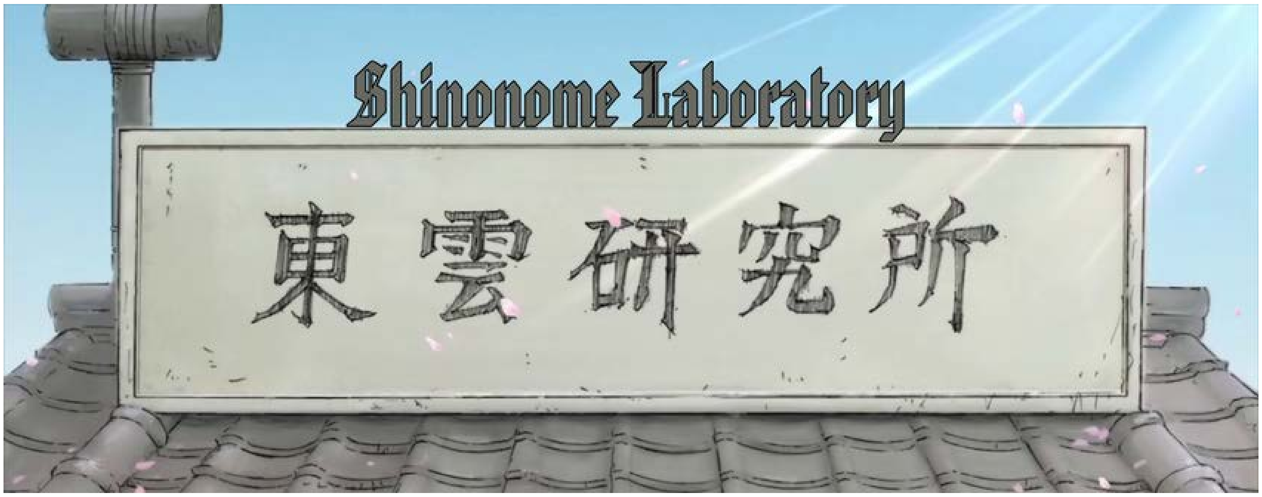
That about covers the basics. Handwriting is a must, other than that you should have at least 2-3 of each of the other categories.
I usually operate with 350-400 fonts installed, though I have at least 10000 at my disposal.
[A year and a half later, 900 installed and rising.] [I think I was around 1200 before I bought new PC.]

So that's what you need. Some image examples will follow below.

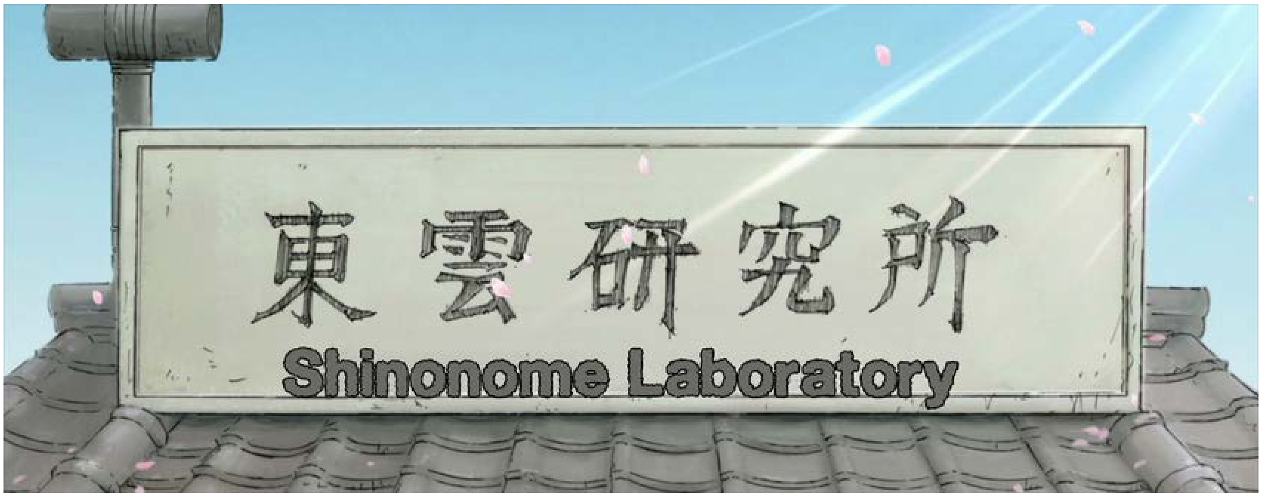
The next step is what to use when. I'd think that would be pretty easy but... I dunno, I've seen enough fails to prove me wrong.
So let's start with an example where I let some guys typeset some Nichijou and we used this Shinonome Laboratory sign to experiment with fonts.



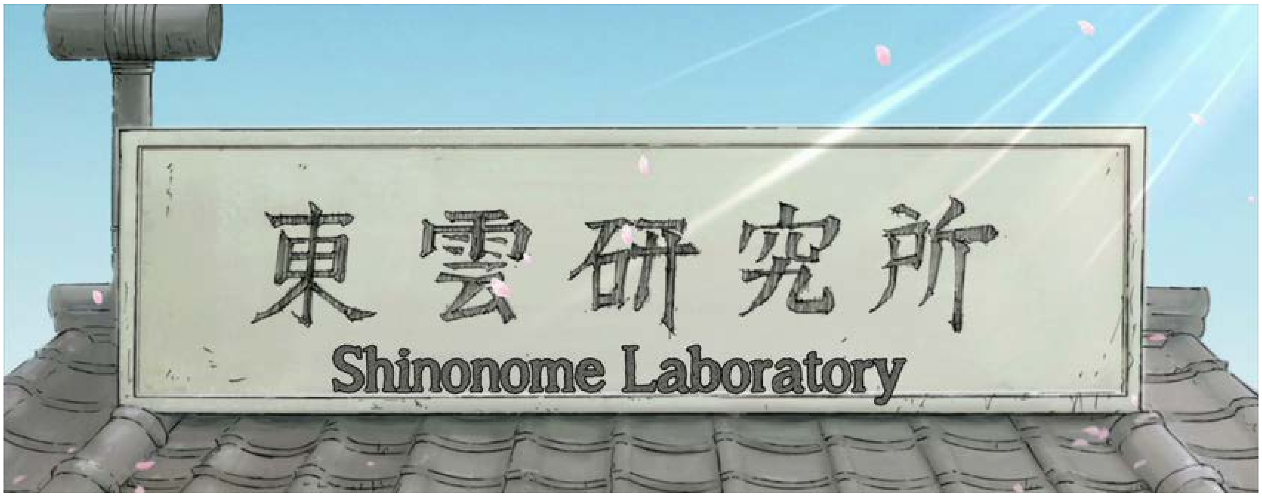
The first attempt looked like this.
What should be obvious is that the JP sign looks handwritten rather than printed, the outline is kind of free rather than precise,
and it has an overlay pattern. Not that you can easily imitate that pattern but it gives you an idea of the general look you should go for.
Which means this attempt fails quickly. Besides the font mismatch it is poorly positioned and the colour doesn't match either.



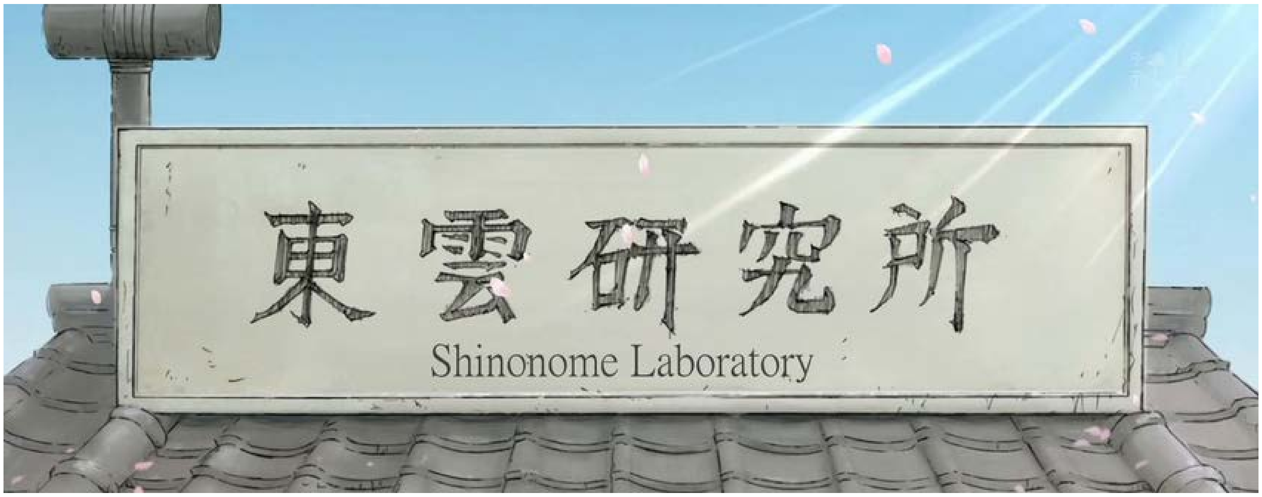
Second attempt.
Improvement in colour and position, but not so much in font choice.
It doesn't look like a plain typewriter font anymore, ok, but this is still too regular, mathematical, clean, and generally doesn't fit.



Third attempt [I think].
This has some eroded outline, so it's not as clean as the previous ones, but clearly it's almost as fat as the average American.
That's bad, by the way.

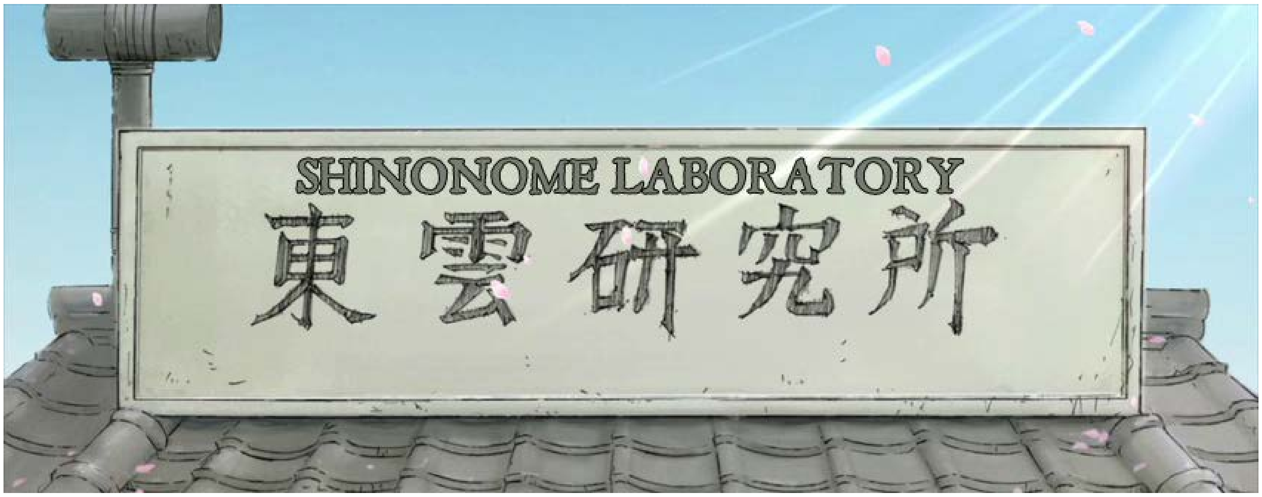


This was, I believe, the last attempt. Eroded outline, some overlay pattern, thickness matches... this passes.

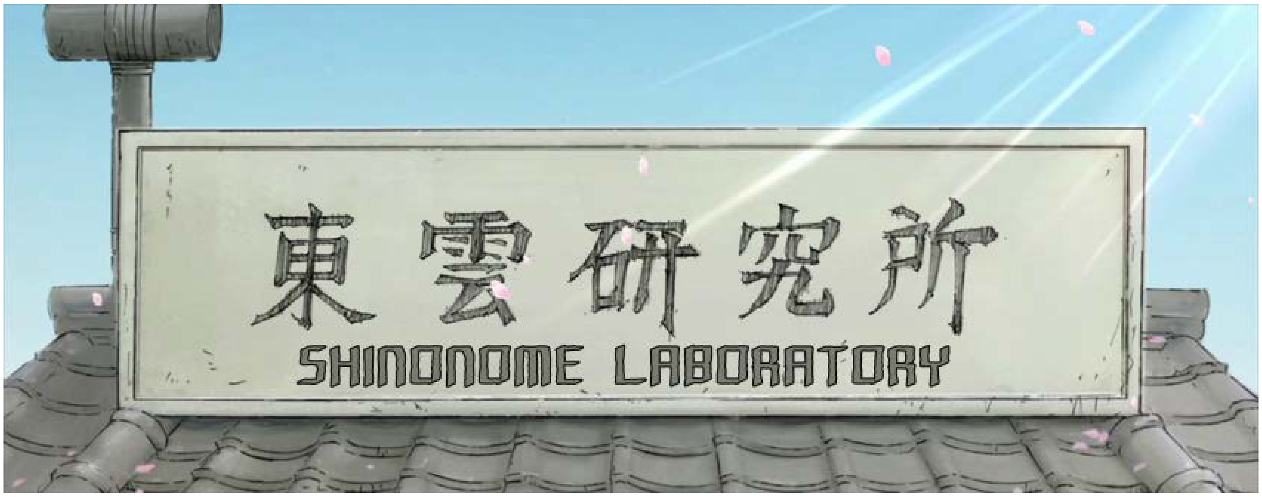


I hear this was in the actual Commie release. It... doesn't pass. Low budget typesetting edition, I guess.

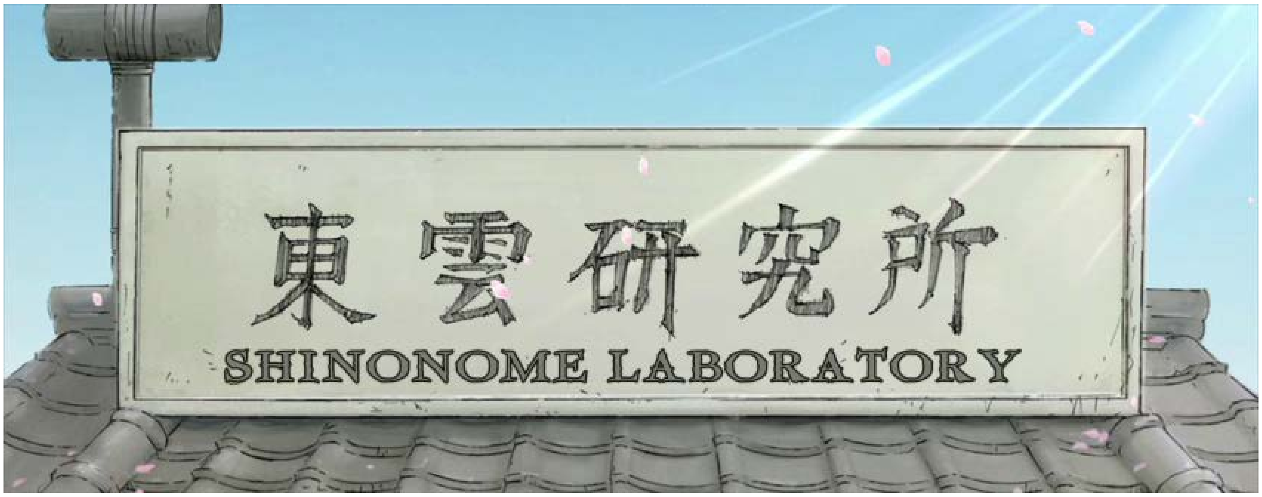
The next few are my examples of fonts that could be used...



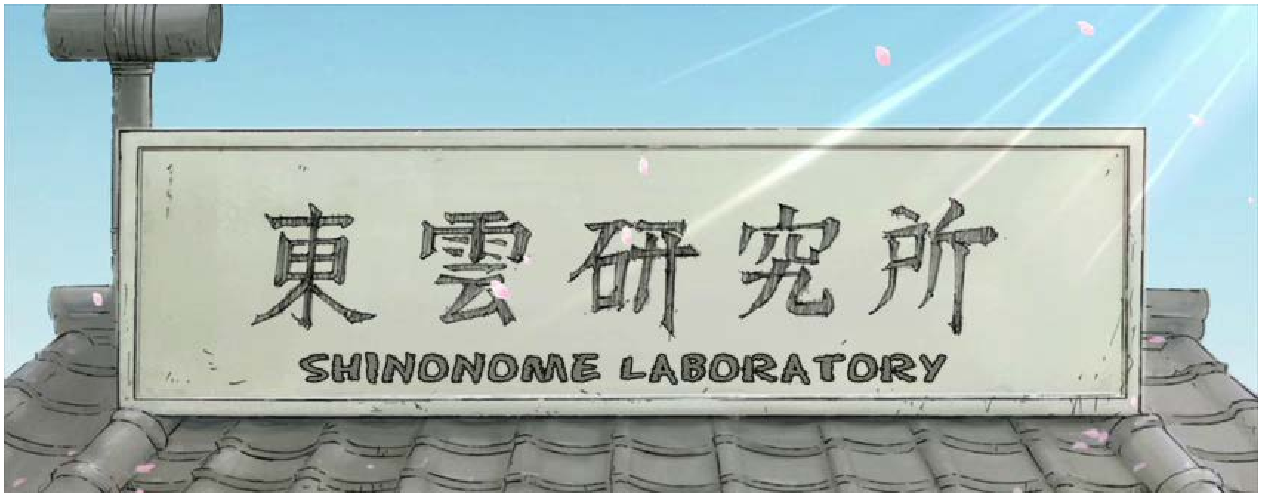
Oh yeah, I used caps because it seemed to fit better and never tried changing it for the other examples.
While you will mostly get the script in capitalized lowercase, there's really no reason why you couldn't use caps for a sign like this.
So, this works fine.



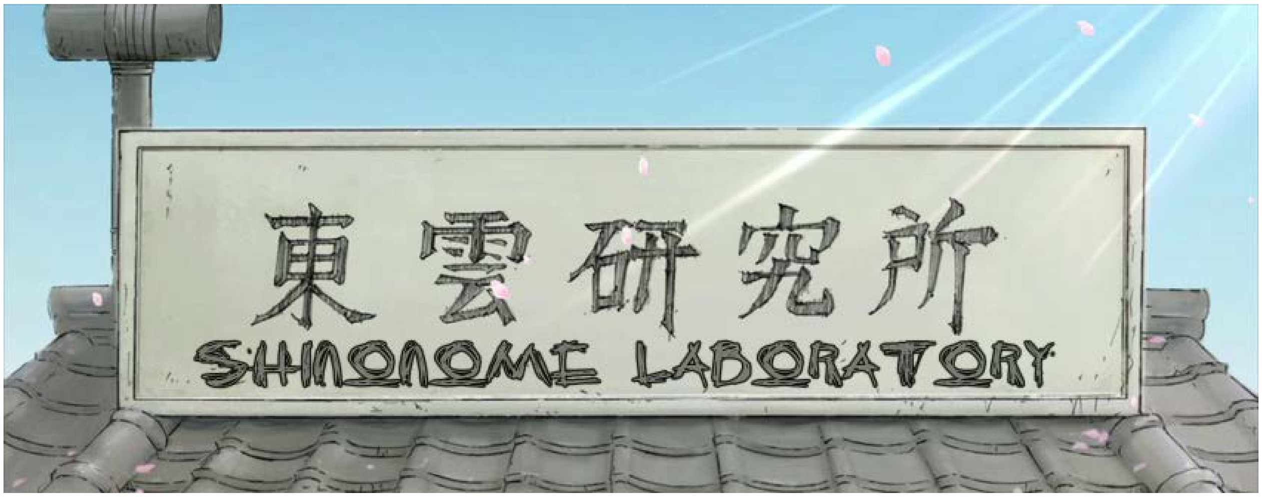
This font is squarish, but the thickness matches well enough and it's kinda irregular so... works.



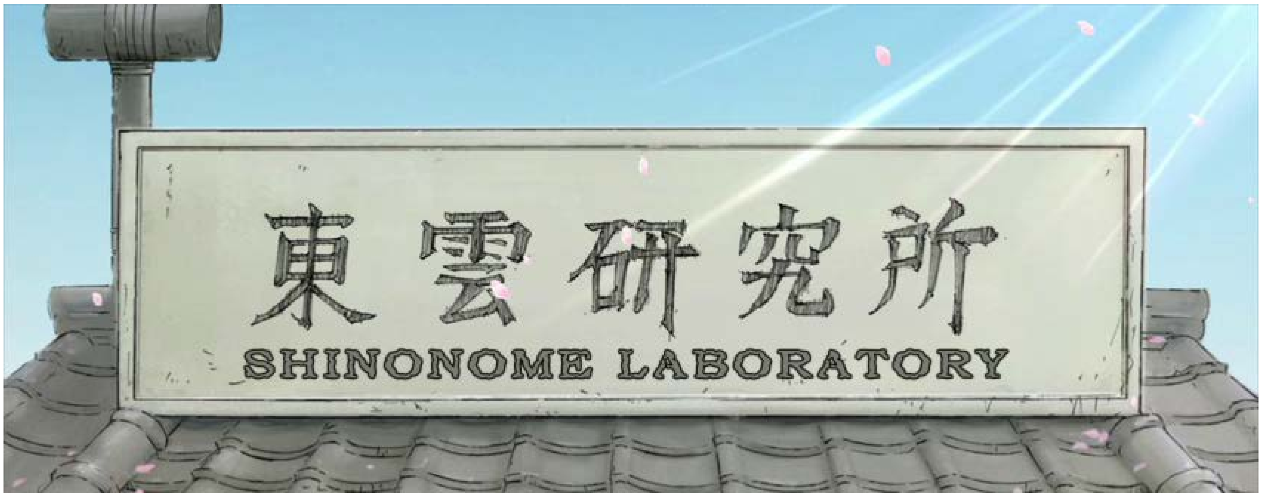
This is better. It has more space between letters which makes it look less out of place. Might actually be my favourite of these.



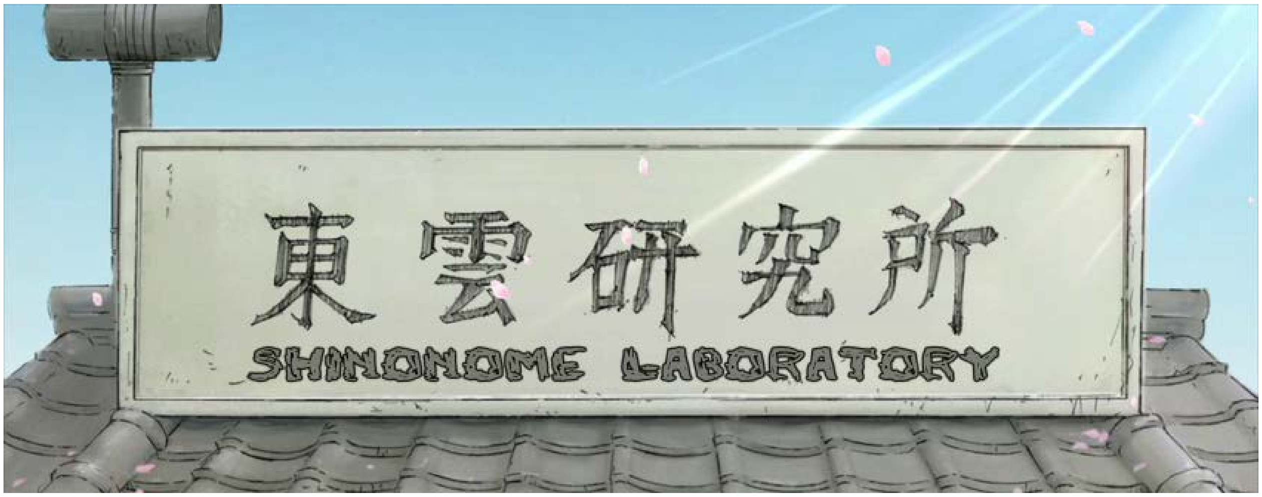
This was just a test to see how Eraser Dust would work. Nothing really great but better than those first attempts up there.
Like, if your font collection is poor, this might be ok.



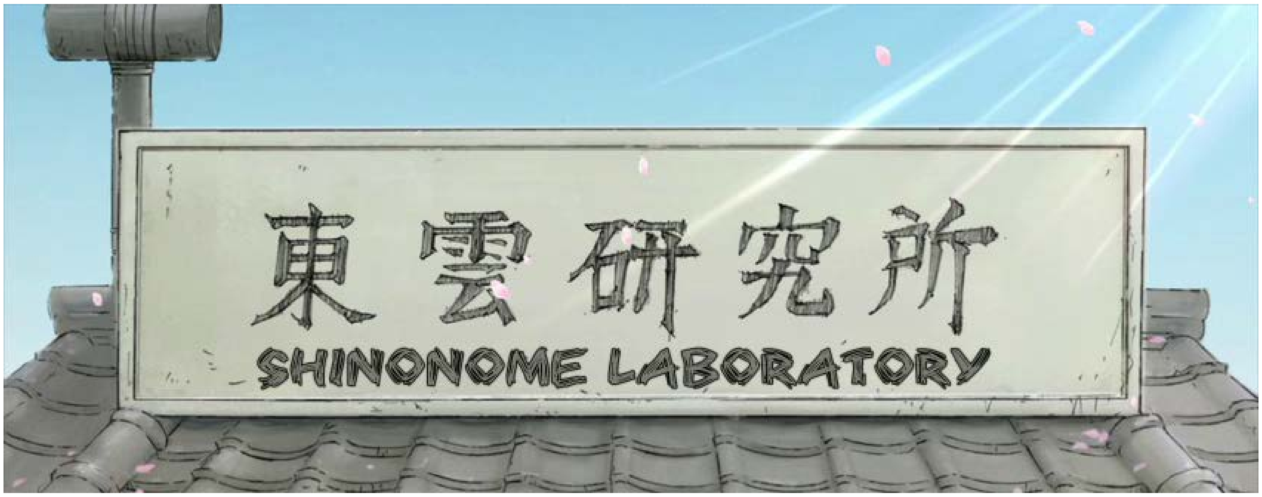
This one's pretty irregular and has lines all over the letters, but is a bit too wild. Still acceptable.
In my opinion it's better to make the sign more interesting than more boring. Then again, that should have limits as well.



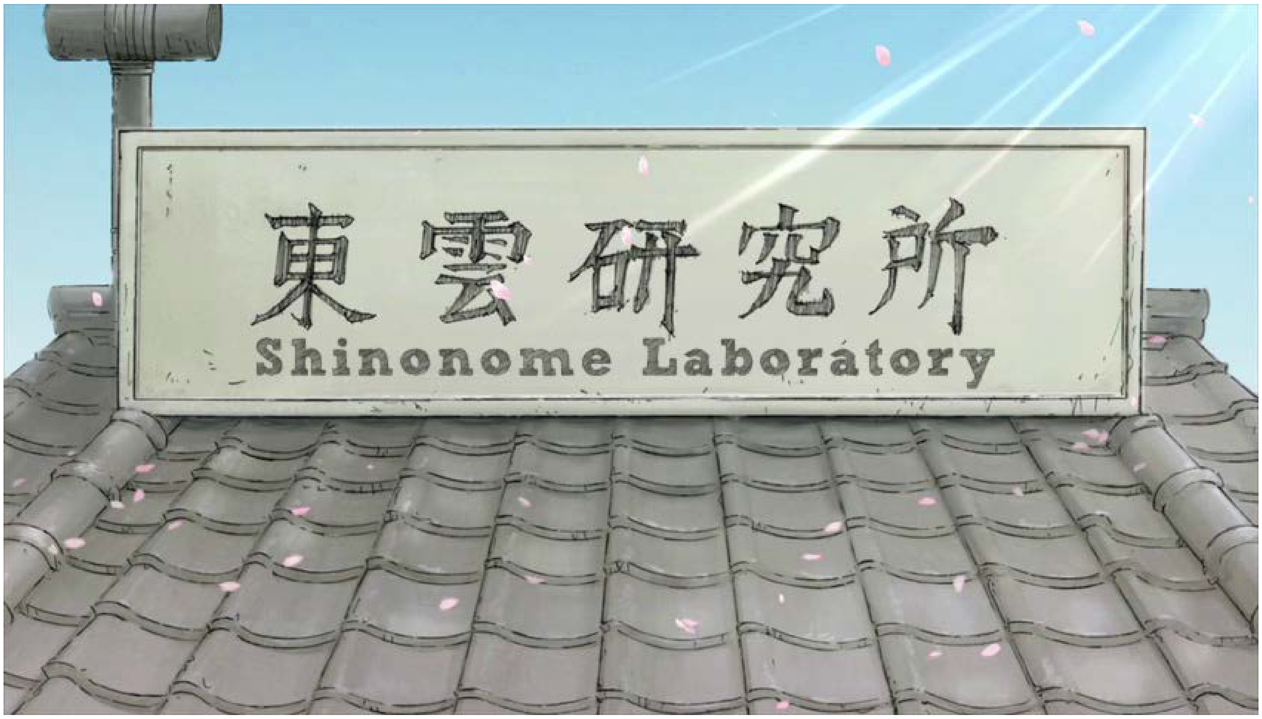
...just another font that works.



This is again going a bit beyond the original, but it's much better than plain boring fonts.

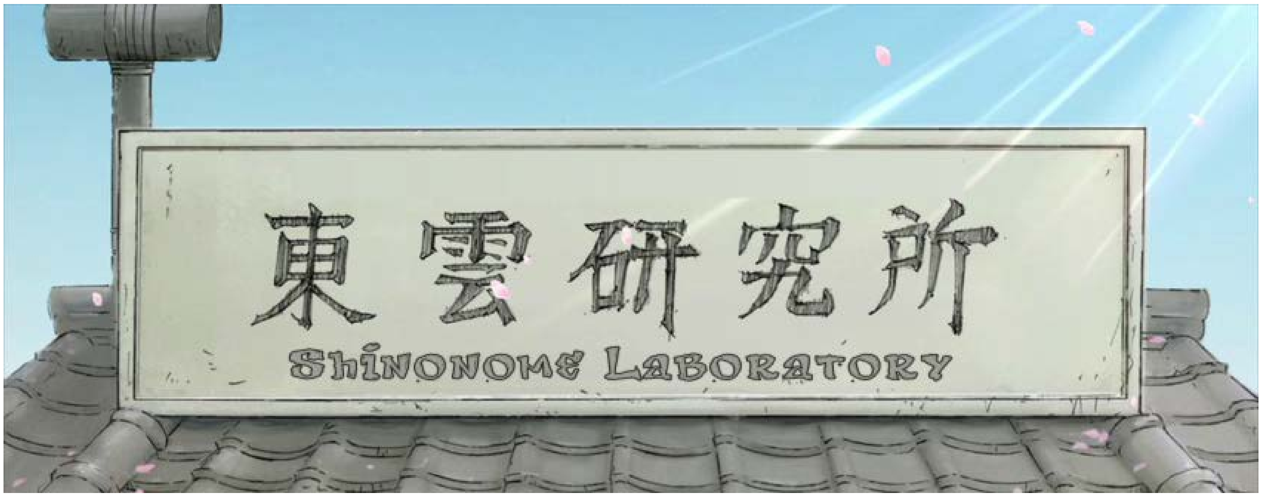


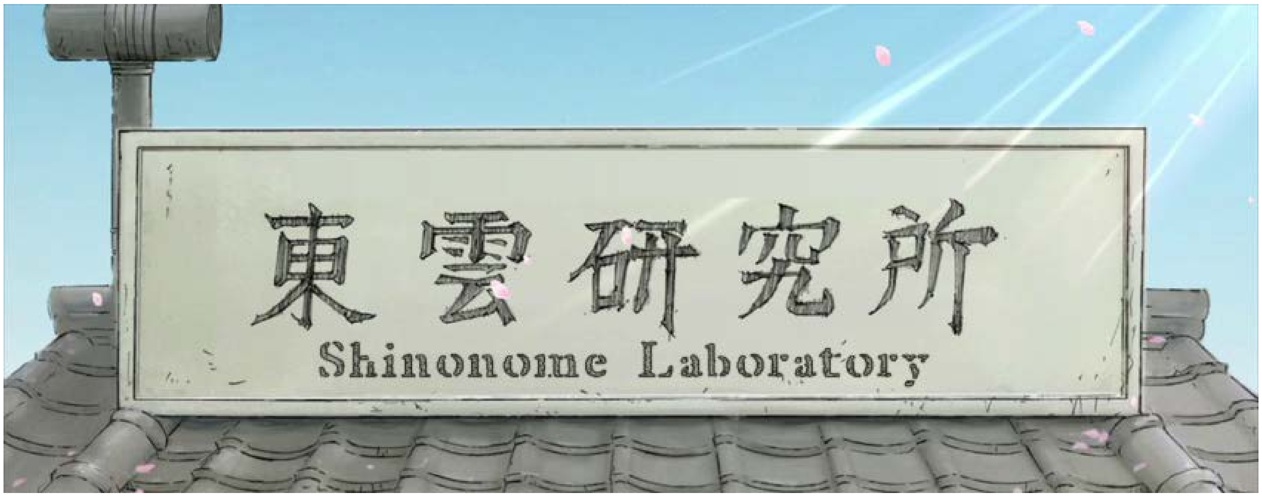
This would be another of my favourites. It has pretty much all the properties of the original and isn't too crazy either.



This was actually the first attempt of another guy in my "TS class", and it's fucking awesome.
Very unintrusive, maybe the best of all examples. If memory serves me well, it was actually Xythar.

So... I took one of the images, masked the English at the top, and tried a few fonts, so here's 2018 edition for fun:





東雲研究所
Shinonome Laboratory

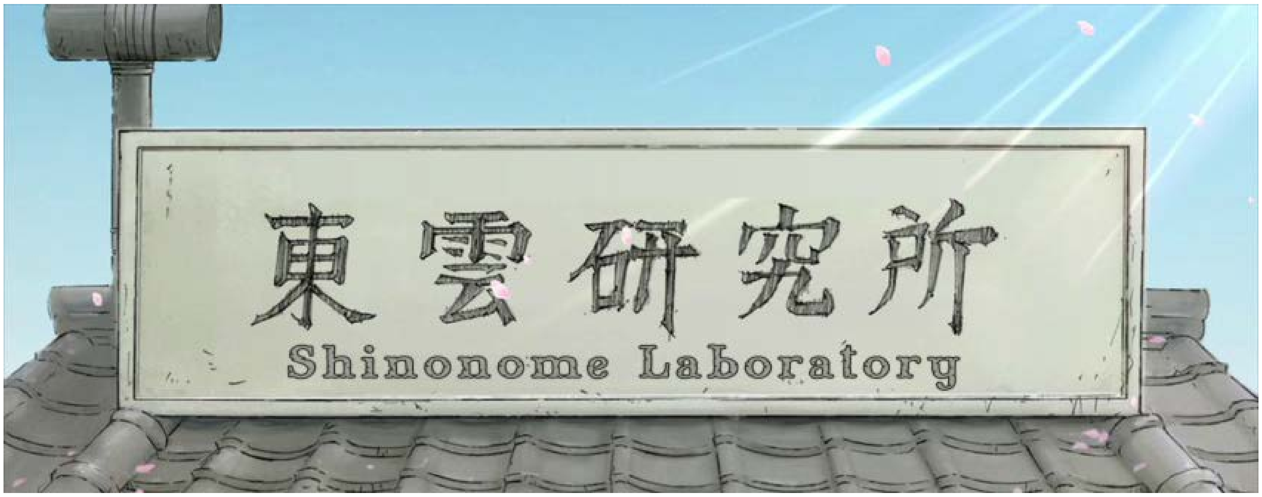
東雲研究所
Shinonome Laboratory

東雲研究所
SHINONOME LABORATORY

東雲研究所
Shinonome Laboratory

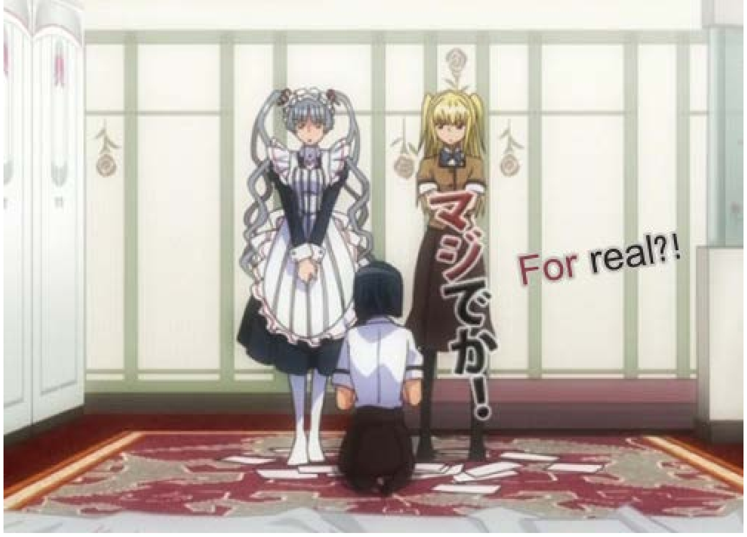
東雲研究所
Shinonome Laboratory

東雲研究所
SHINONOME LABORATORY

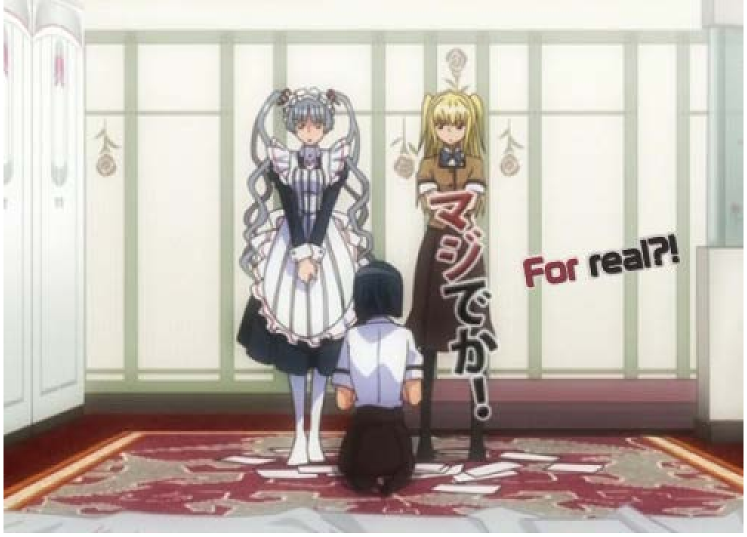


I think they somehow all look better than pretty much all the old ones...

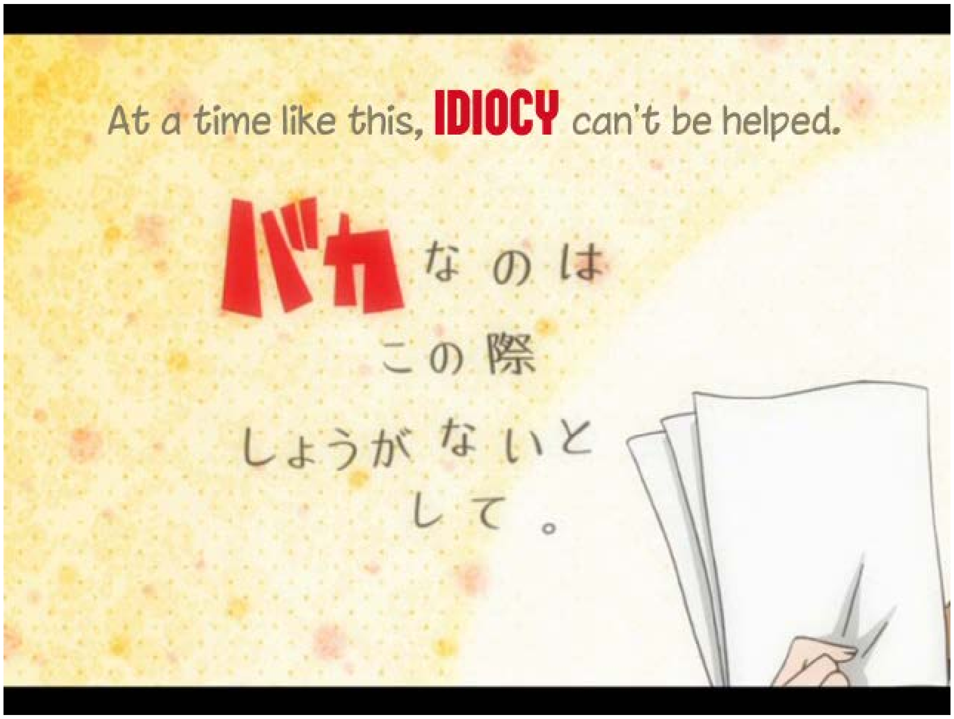
The larger your font collection is, the more you have to choose from and the higher chance of your final choice being really good.
If you only have Arial, TNR, ComicSans and other basic stuff, this sign won't look too good.



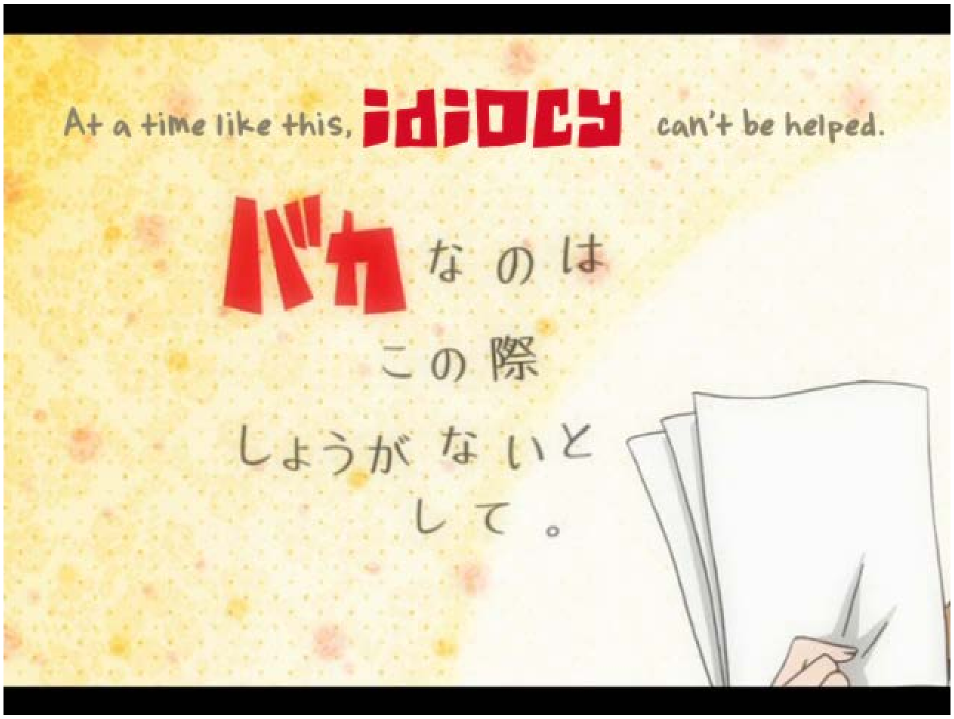
This one's from another test. It's too plain and flat.



I just changed the font and added a little bit of shadow to make it more 'plastic'. (blur fail)

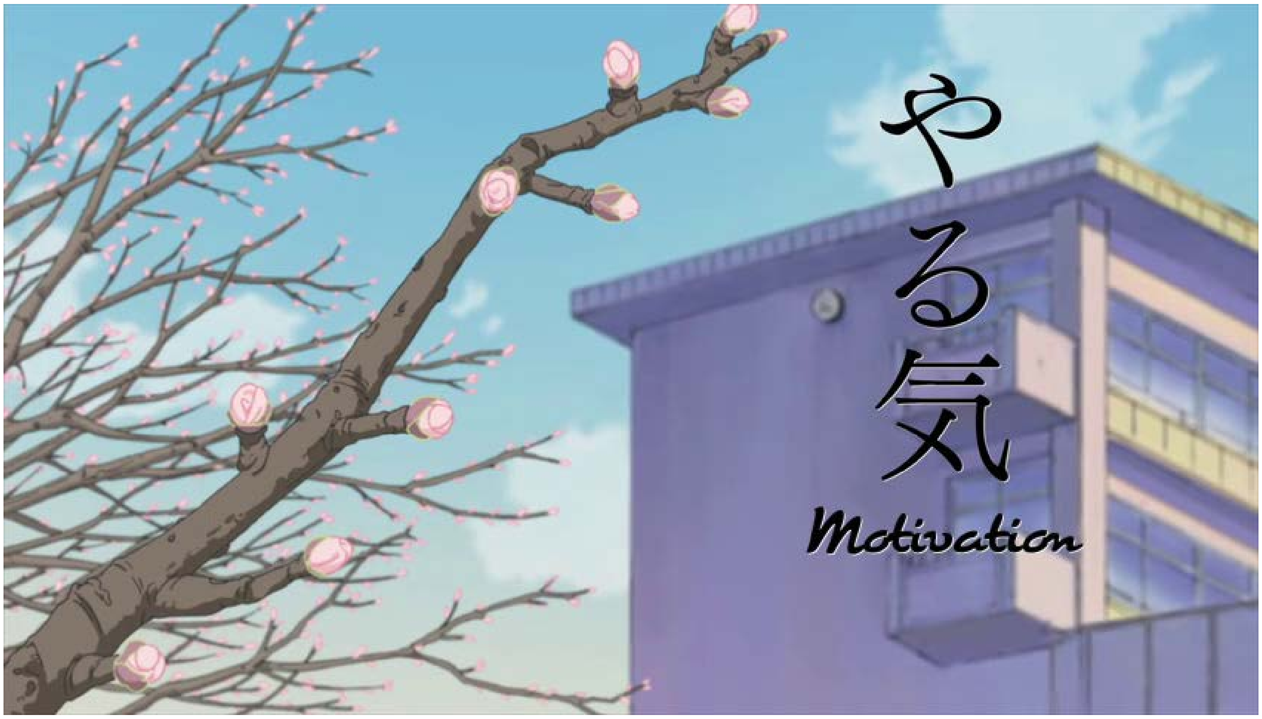


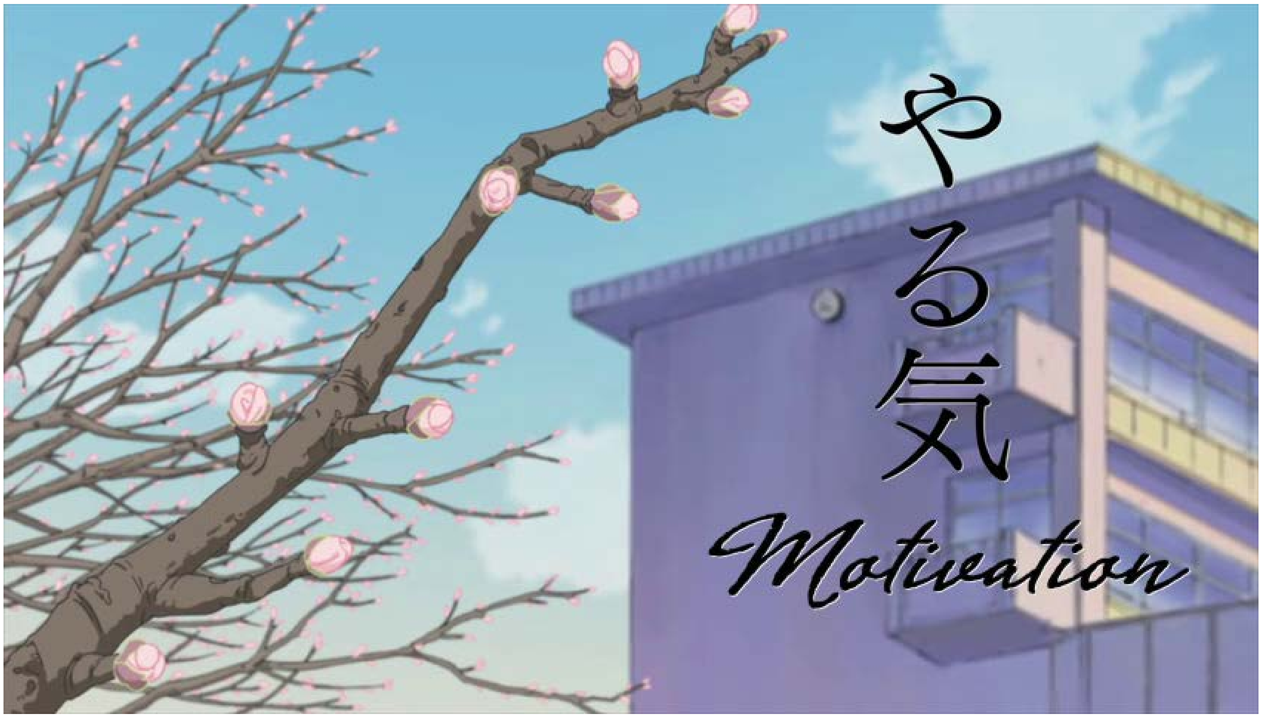
This doesn't look bad, though it lacks blur, but the red font doesn't correspond with the JP one much.
[Also personally I'm a bit tired of chinacat.ttf - the other one]



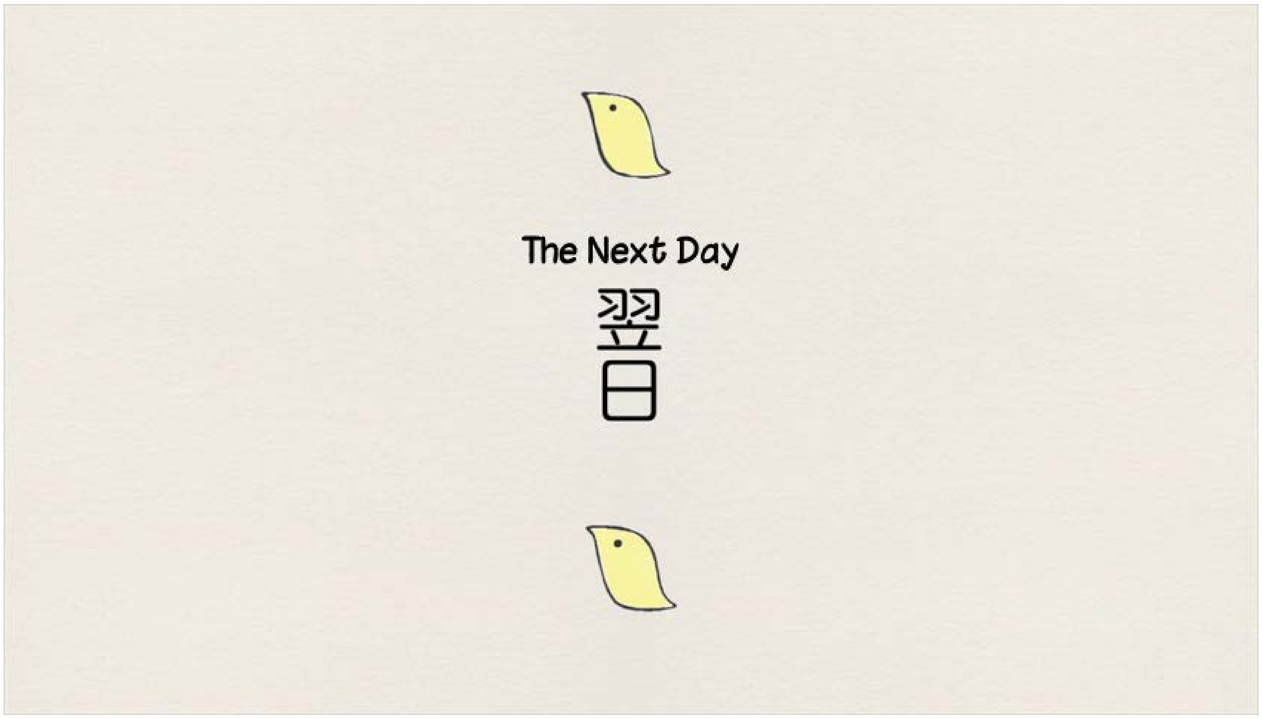
If you have enough fonts, it can make a world of difference.
This red one, btw, is what I meant by 'cartoonish fonts.' Stuff like that.
2015 Note: The red colour is clearly wrong. Also, something could be done about the uneven spaces on each side of "idiocy".

We've seen this next sign in the Positioning section. Here are two random examples of fonts that would work with this.

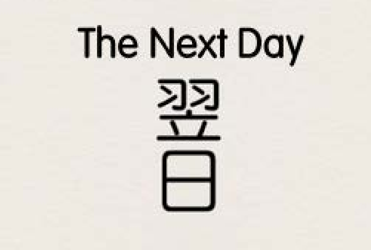




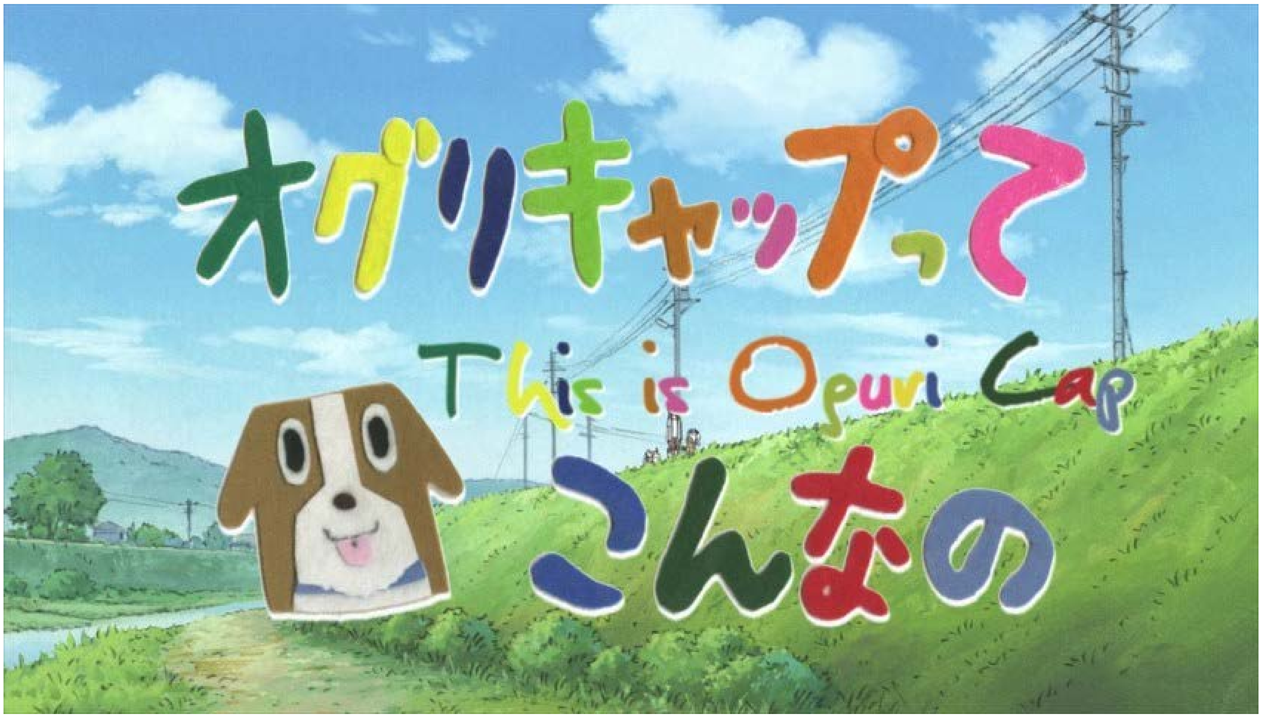
You may argue that the original sign isn't really as fancy as the typesets, and you may be right, but I'd say...
I'd rather make the typeset look better than the original then make it look uglier.



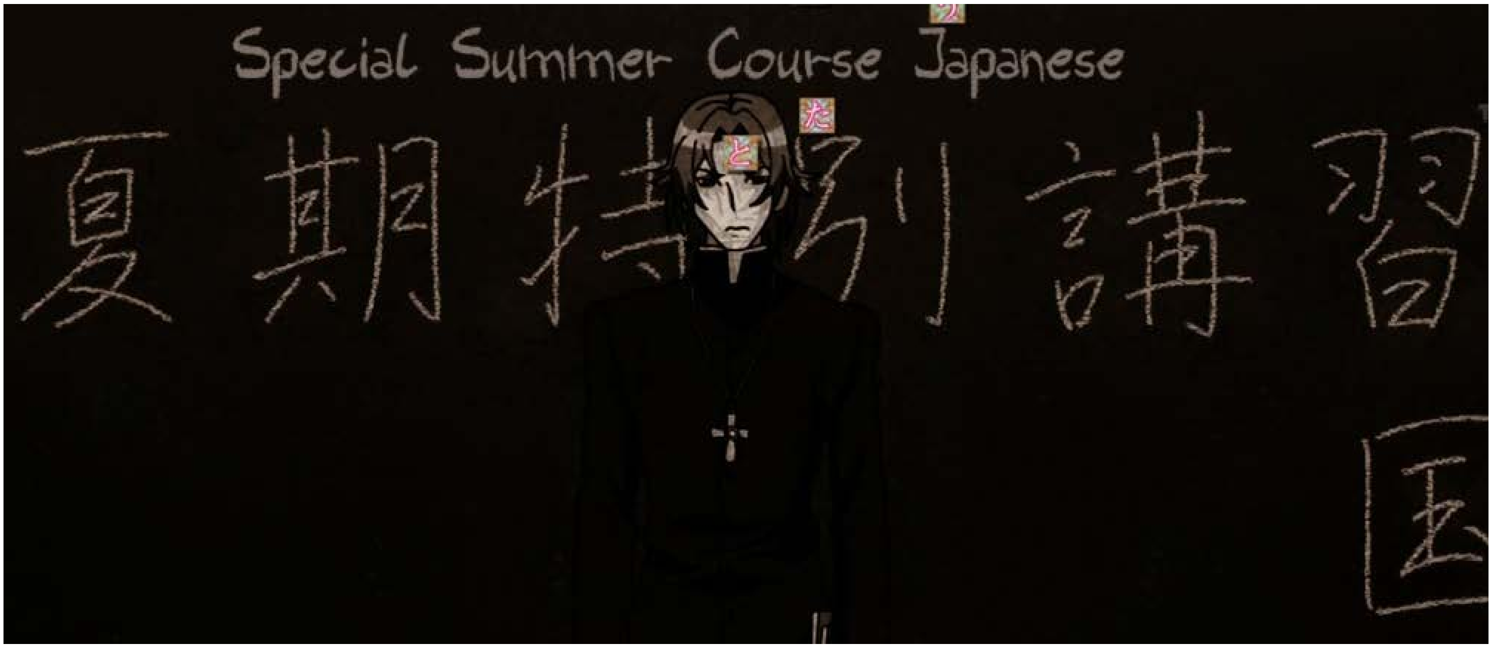
This is something you'll definitely come across. A font that looks fairly plain but is rounded at the ends.
This is chinacat again, but here ~~it's just perfect~~. It's a lot more regular than most handwriting fonts,
and here you can match the size and thickness of the letters to the original really well.



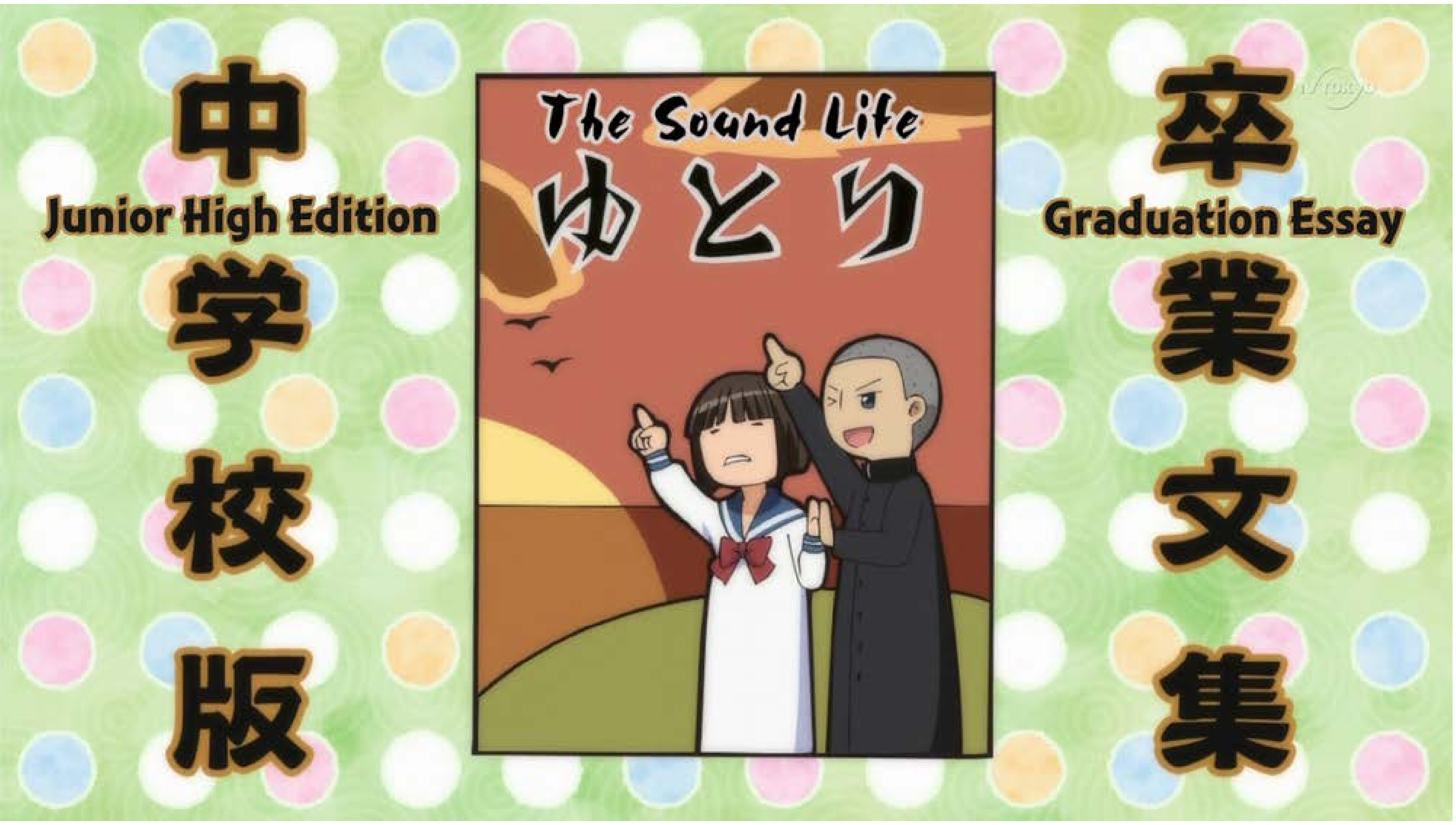
Actually no, fuck chinacat. This is how you do it.



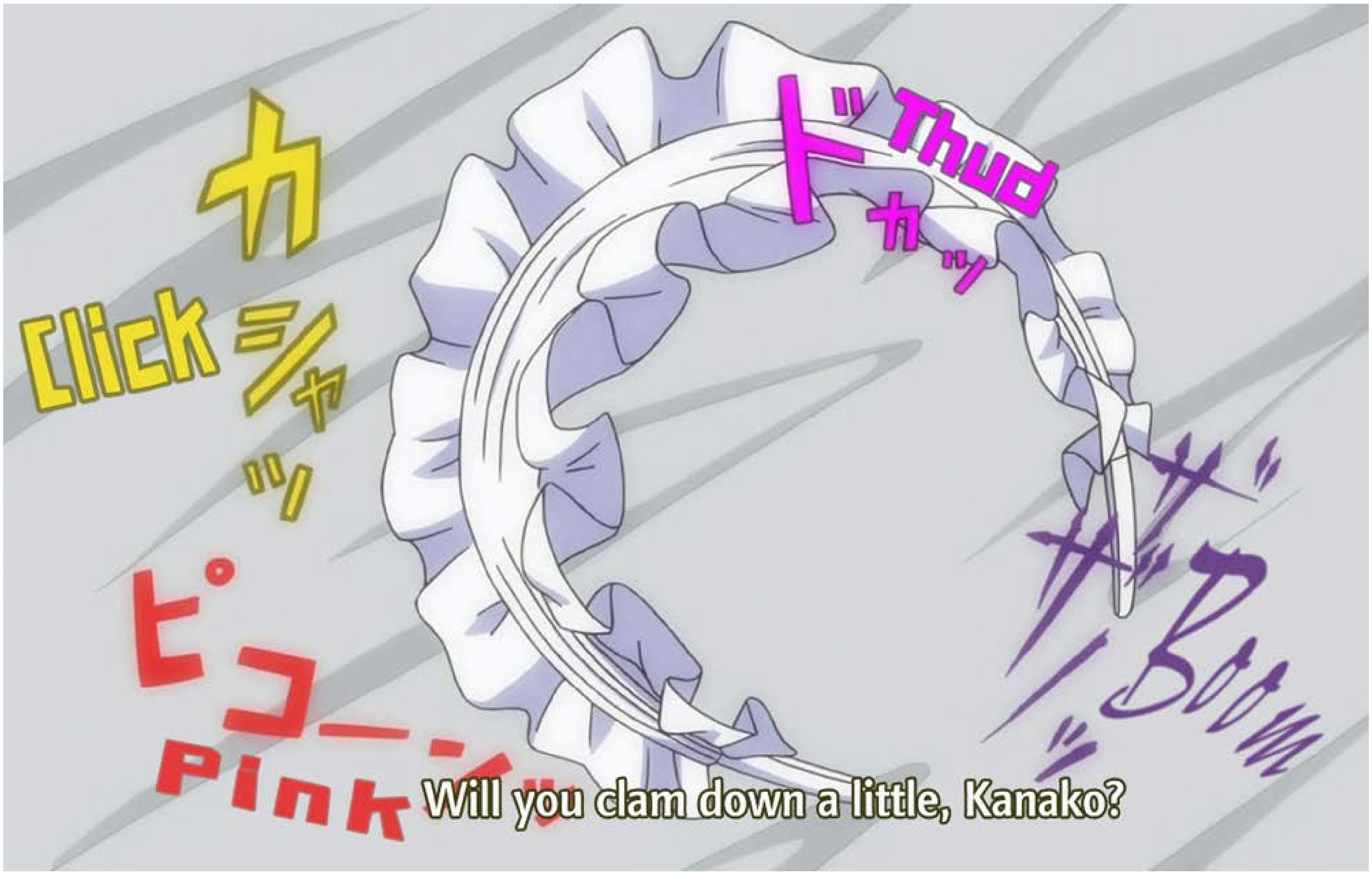
Children-style handwriting.



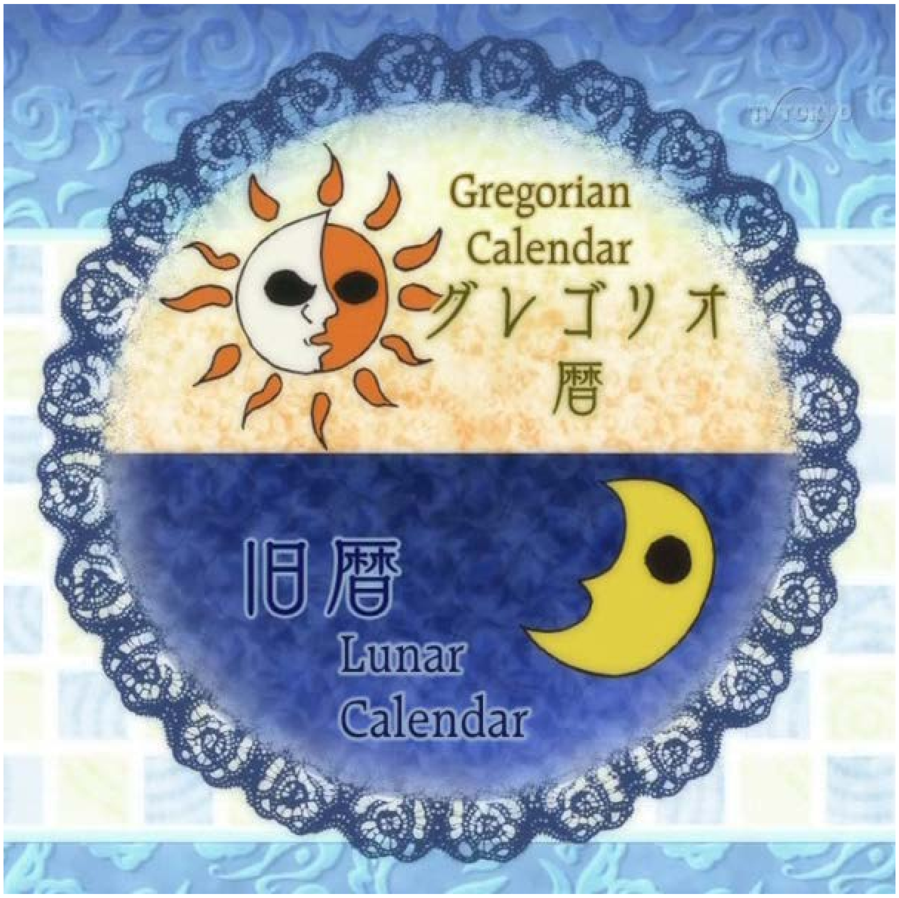
One of those chalk fonts. The pattern isn't that visible with small size, but it's good enough.



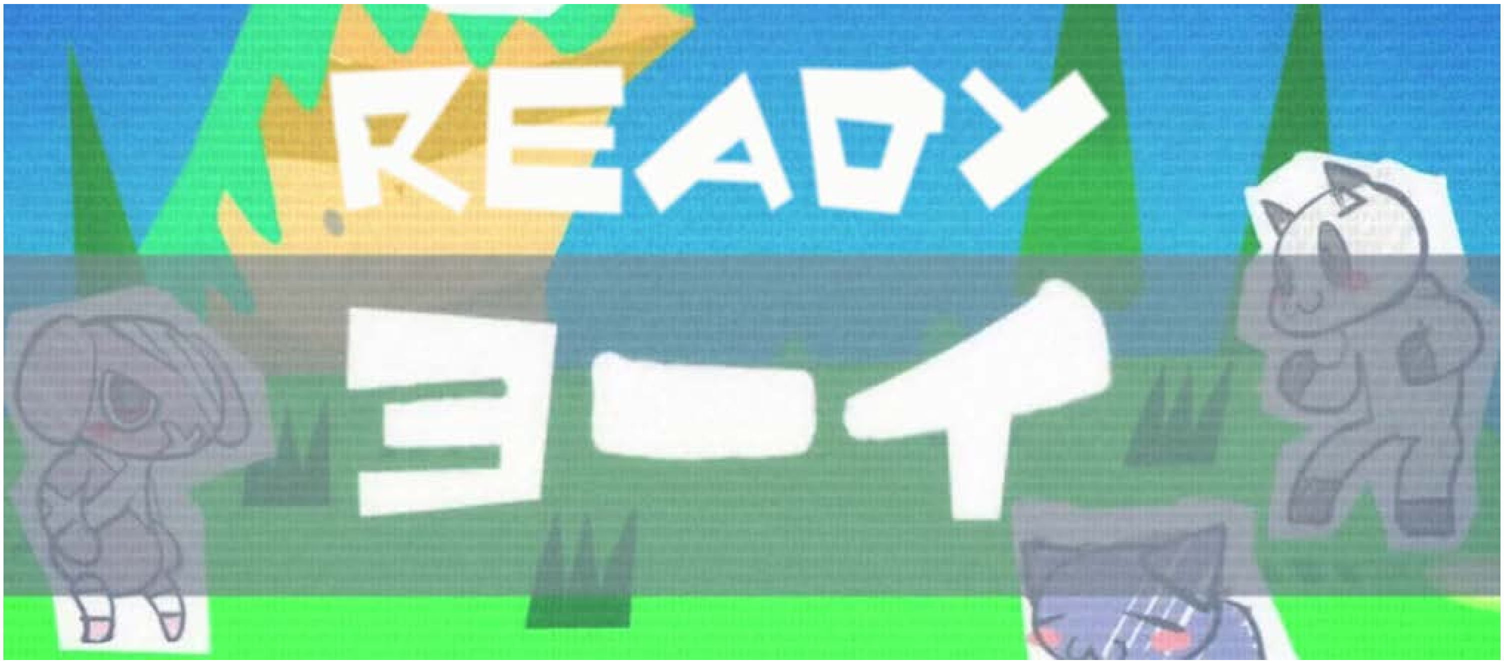
Here on the sides you have one of the not-so-plain sans serif fonts.
For the middle sign, the choice of the right font is what will make it look like it belongs there.
It has two layers. Both the black and white are slightly blurred to make it look natural.



This is quite self-explanatory, so nothing to say except... I think the editor made a mistake here!
~~Good thing it hasn't been released yet.~~ Fixed for release.

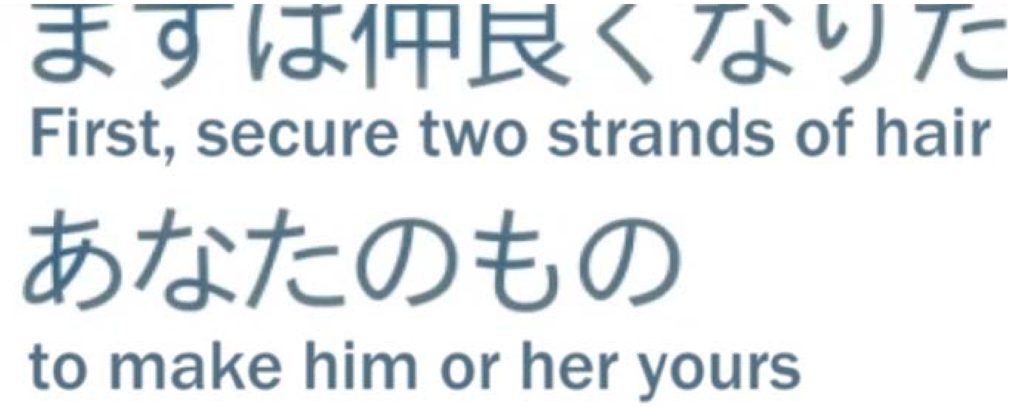


Does this work? I think it does.



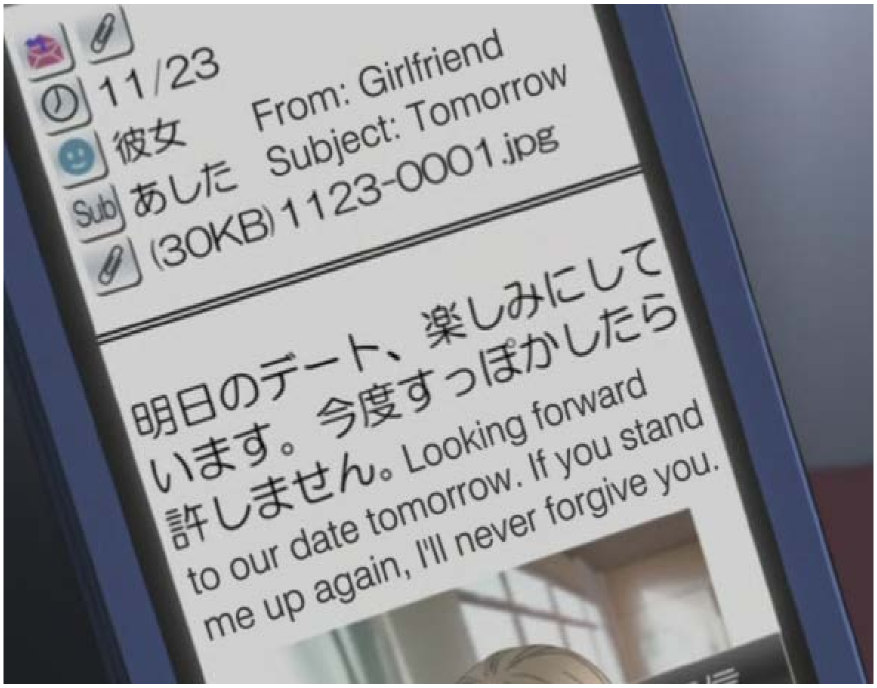
Sometimes you just find exactly the right font...

Now a few things about what plain standard Japanese fonts look like, ie where you should use plain fonts too.



まずは仲良くなりた
First, secure two strands of hair
あなたのもの
to make him or her yours

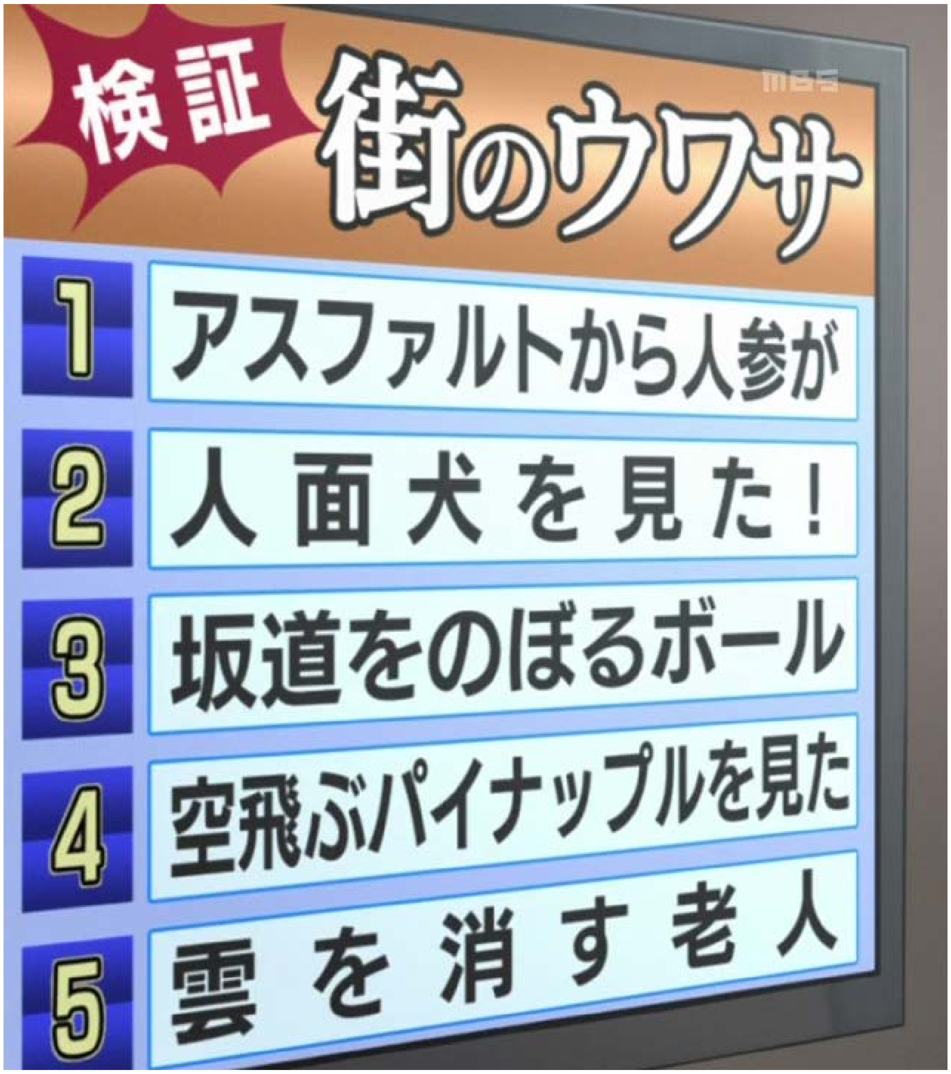
This is an ordinary sans serif font.



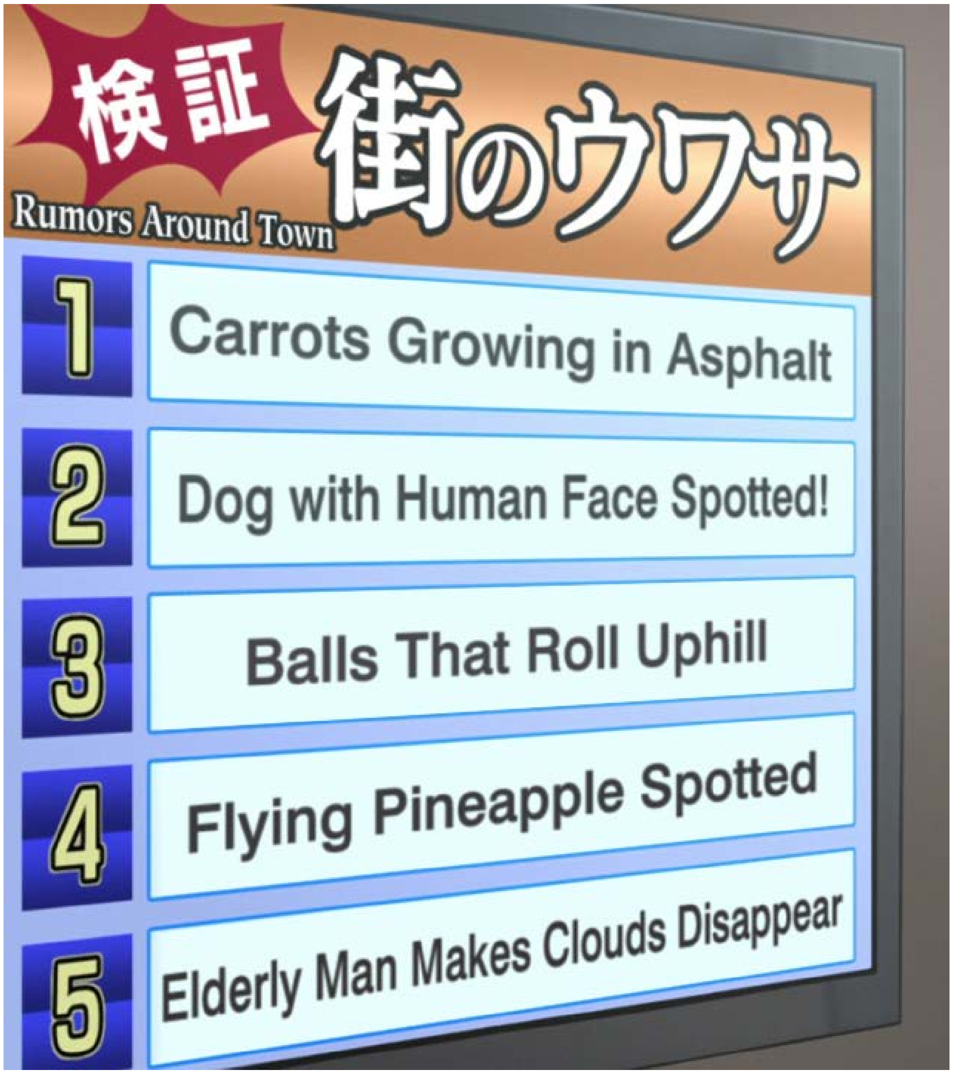
11/23
彼女 From: Girlfriend
あした Subject: Tomorrow
(30KB) 1123-0001.jpg

明日のデート、楽しみにして
います。今度すっぽかしたら
許しません。Looking forward
to our date tomorrow. If you stand
me up again, I'll never forgive you.

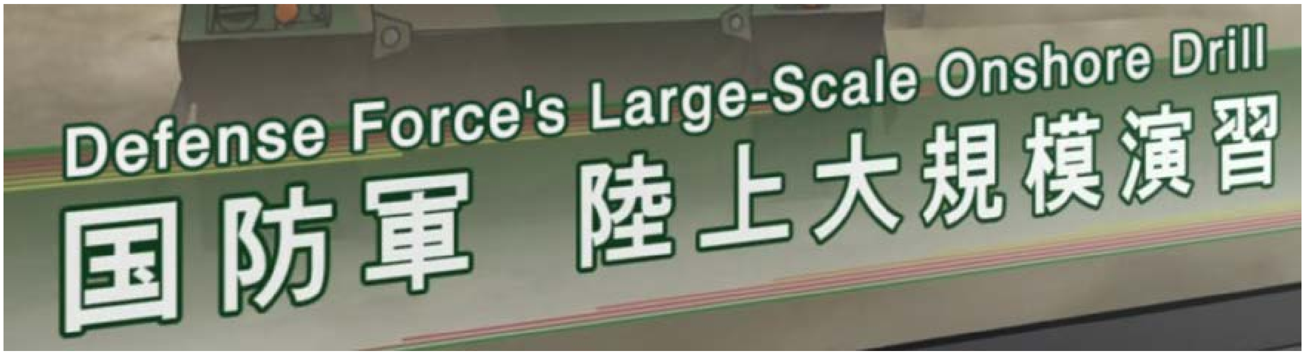
Same here. Adjust the size so that it matches the thickness, match colour, add blur, done. Well, almost, 'cause you need \frz and most likely some \fax.



Here you have bold serif at the top right and bold sans serif below.

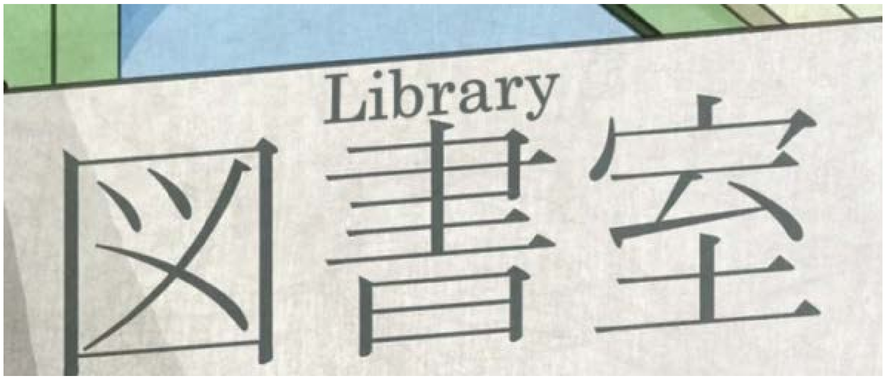

検証
街のウワサ
Rumors Around Town
1
Carrots Growing in Asphalt
2
Dog with Human Face Spotted!
3
Balls That Roll Uphill
4
Flying Pineapple Spotted
5
Elderly Man Makes Clouds Disappear

No space for the five, so you have to mask it, but you can get quite close to the original.

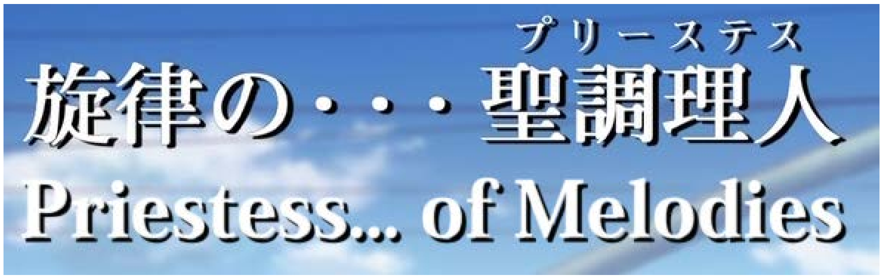


Simple sans serif bold/heavy/black. The border gets brighter to the right, so you can change the colour for each word.
Also contains: little bit of \fry and \org, and of course blur and two layers.

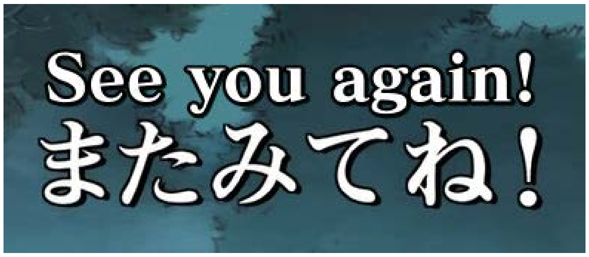
Uhhh... the right side should be much smaller, for the record.



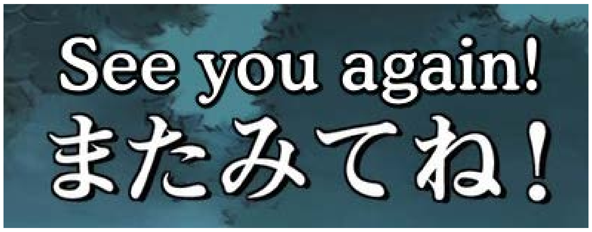
Ordinary serif font. This one's thinner than usual but since the typeset is much smaller, making it thinner wouldn't look too good here.



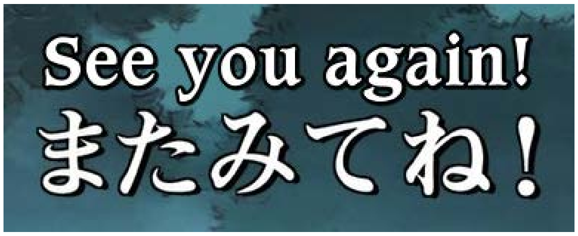
This is quite typical serif.



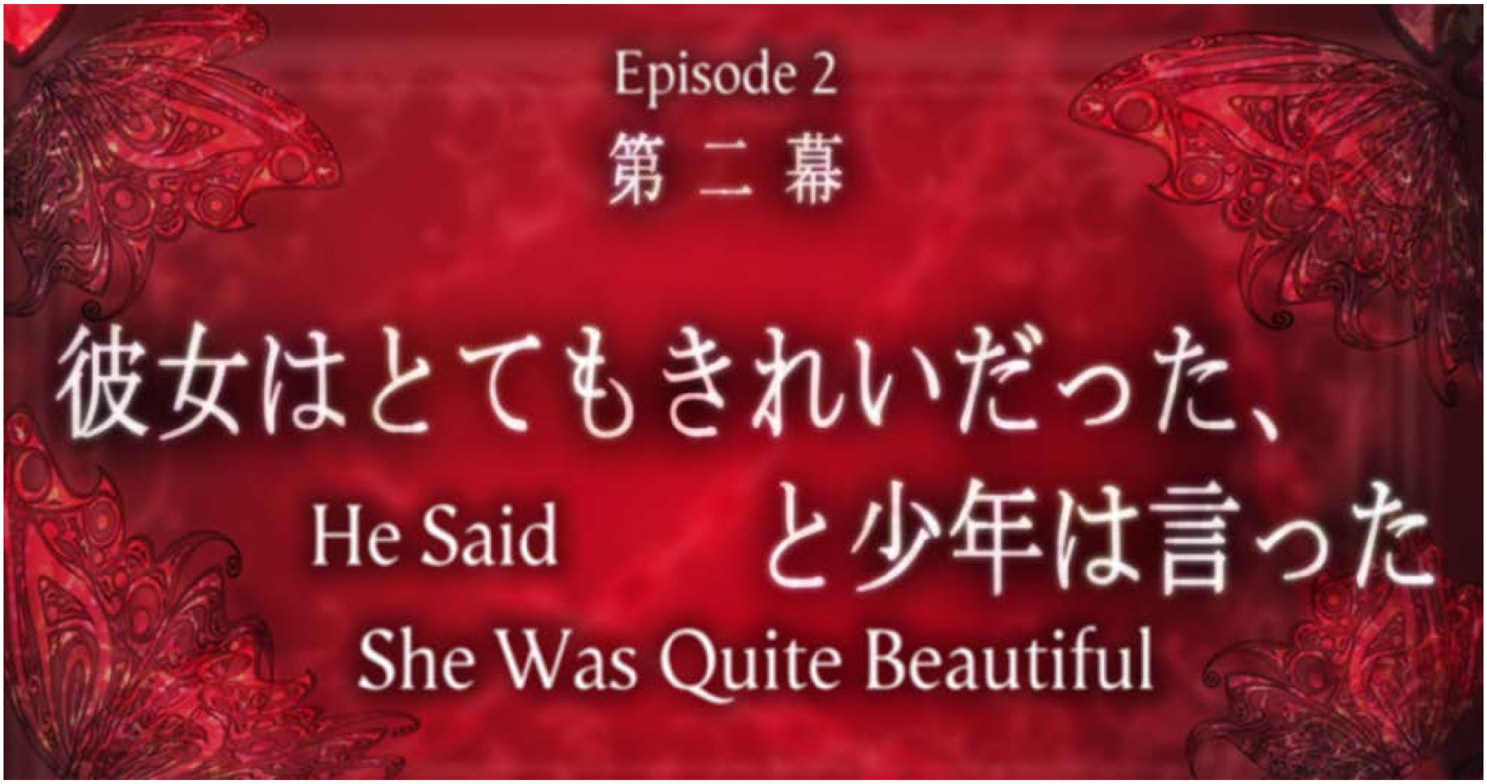
This may be a bit more rounded, but I think it's mostly because it's hiragana as opposed to kanji in the previous example.



You could use somethnig more round...



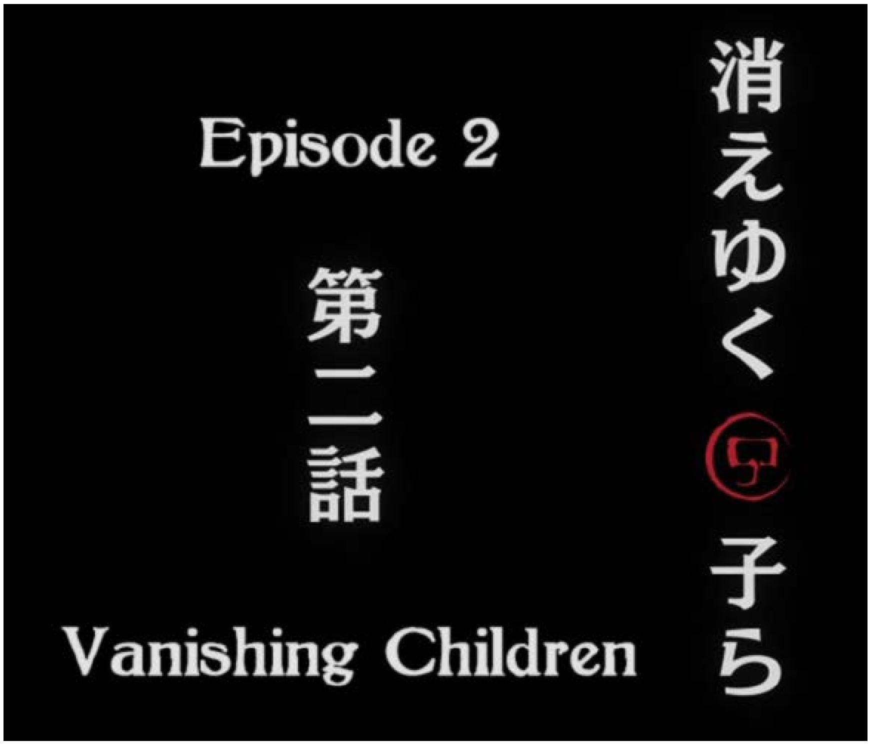
...or something a bit more fancy. But the first one works.



Another serif font, but this one's thinner and taller.

Also play with border/blur/colour on the bottom layer until you get it right.

You might wanna go for more perfection with episode titles since it's gonna be used in all episodes, so it might as well look really good.



This is kind of half-serif.

There are several things you should consider when picking a font for something like this.

Height/width and overall thickness. This should be obvious.

Then there's the ratio of horizontal vs vertical thickness. You can see the vertical lines are thicker than the horizontal ones.

That is usually the case, except for simple sans serif like the "Defence Force" sign above.

The ratio may differ though. On the red one just above, the difference between horizontal and vertical lines isn't big.

But on the "Priestess..." higher up the horizontal lines are really thin.

Here on the "Vanishing Children" there are no really thin lines. So you should try to match that.

The last thing that will determine whether the font matches or not is the ends of the letters. This of course makes the serif/sans difference.

The serif endings may vary a lot though.

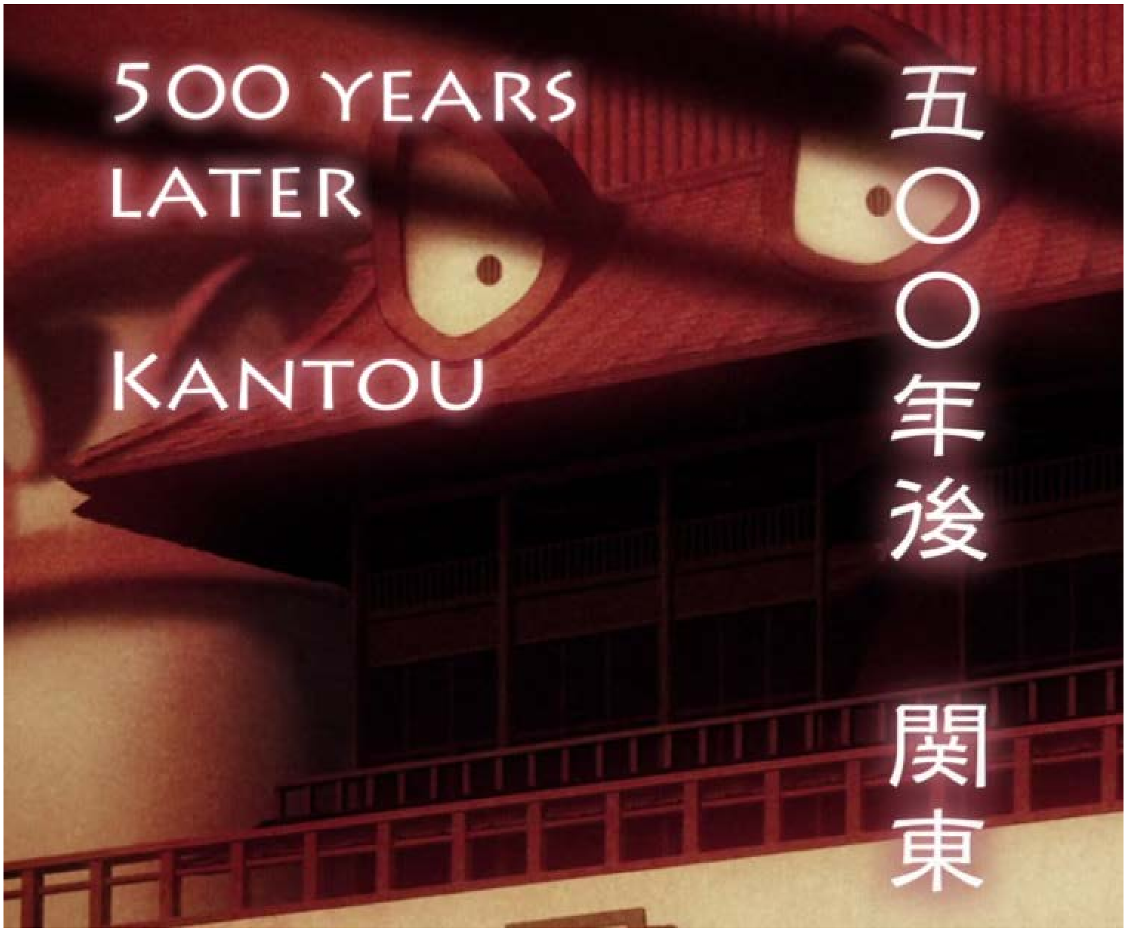
They can be sharp/pointy in various ways [Garamond, Footlight, Timpani], less so like Raleigh [Gregorian Calendar sign above] or Arno Pro, rounded [Cooper, Souvenir], short/heavy/square-ish [Acclaim, Albertus, Strayhorn, GeoSlab(numbers)] or with some decorations [Zapf Chancery].

So here for "Vanishing Children" you need serif where the endings aren't too pronounced and horizontal lines are thinner than vertical but not too thin.

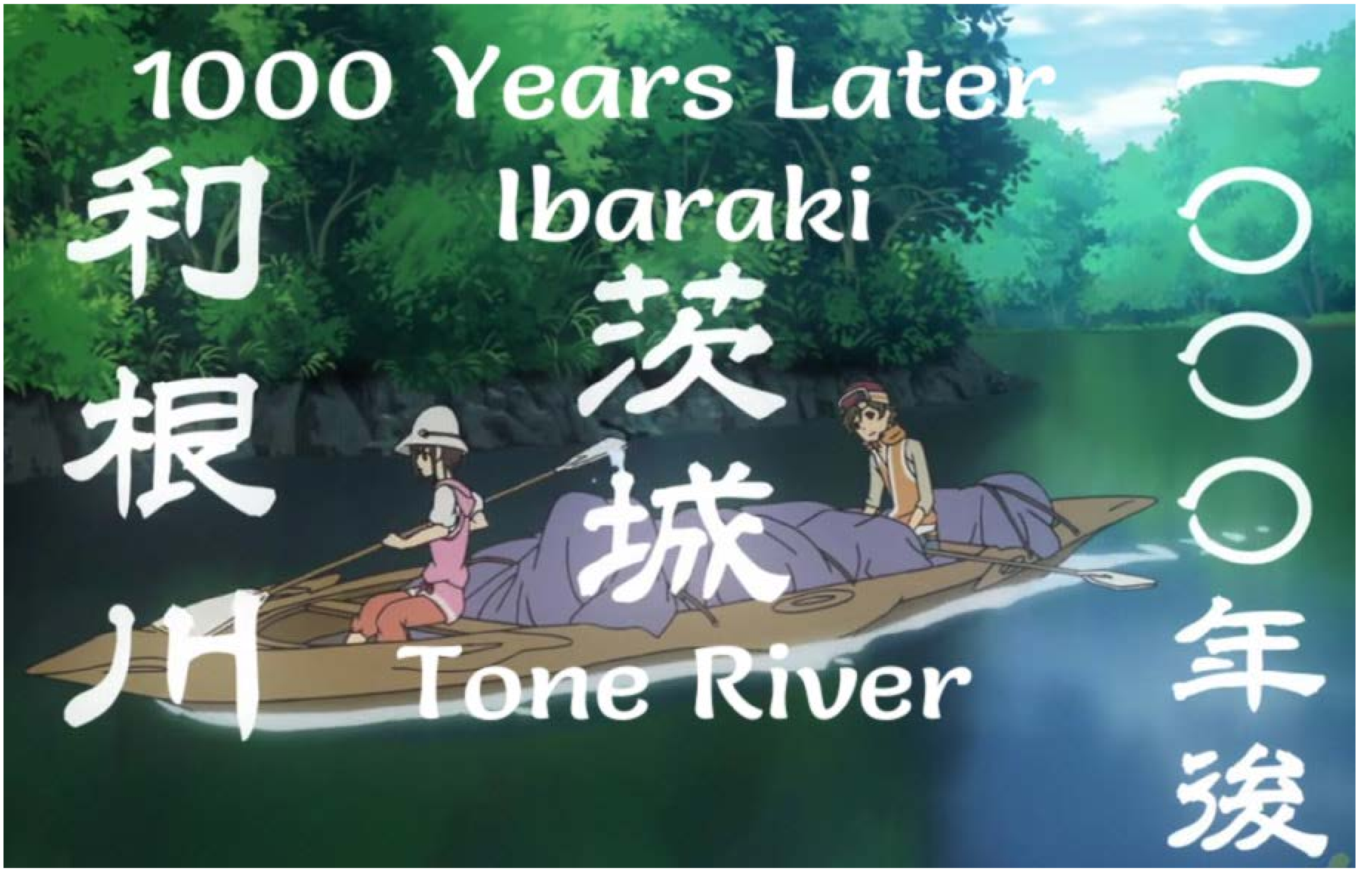
I think the choice here is pretty much perfect.

A few typical serif examples if you have no clue what to look for:
[Not Times New Roman], Garamond, Goudy Old Style, New Baskerville, Georgia, Thames, Palatino

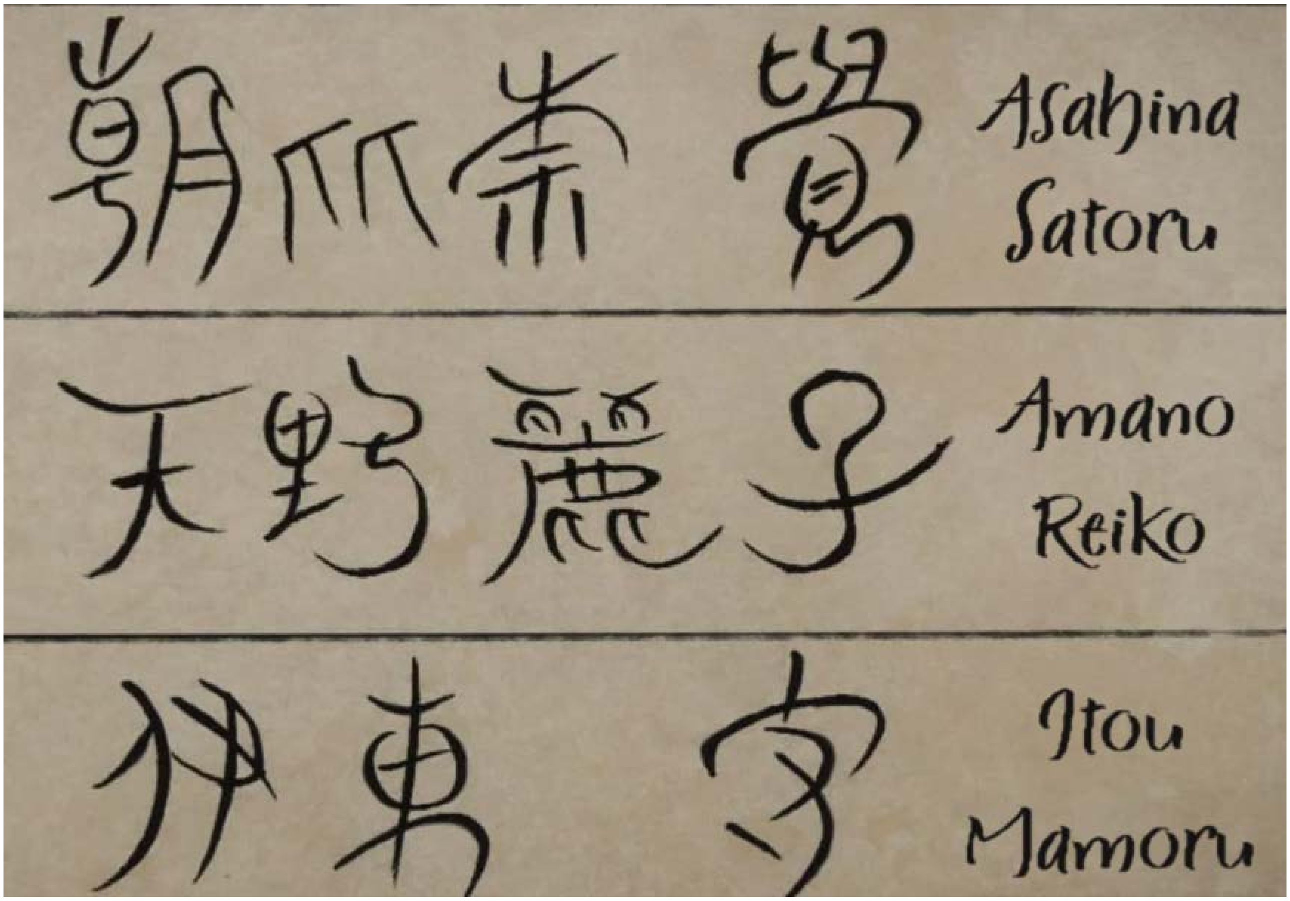
A few typical sans serif examples:
[Not Arial], Franklin Gothic, Myriad Pro, Zurich, Frutiger, Liberation Sans



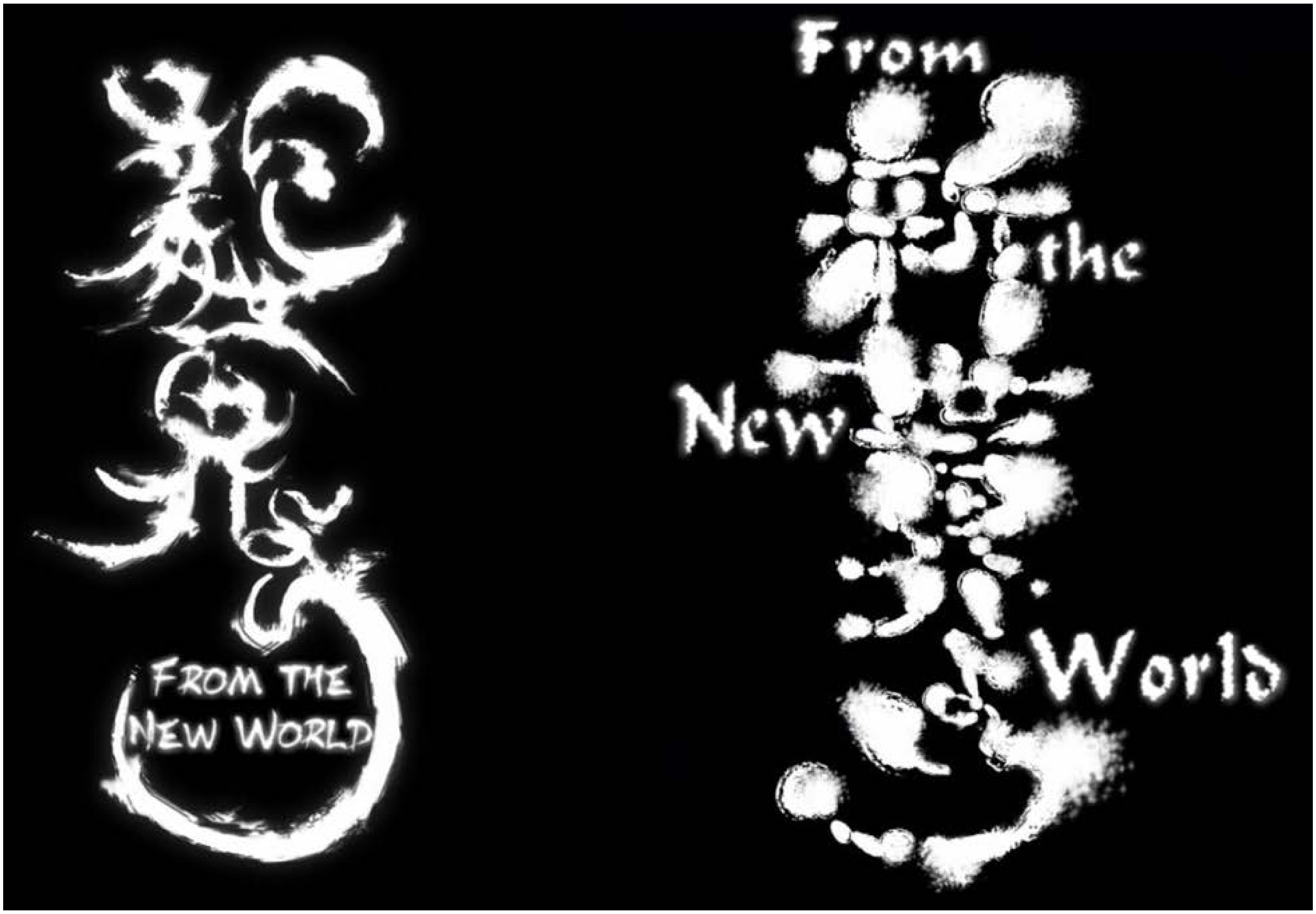
This font is very "open," with a lot of space between the lines and [almost] the same thickness of all lines.
A characteristic feature here is that the zero has the same width as height.



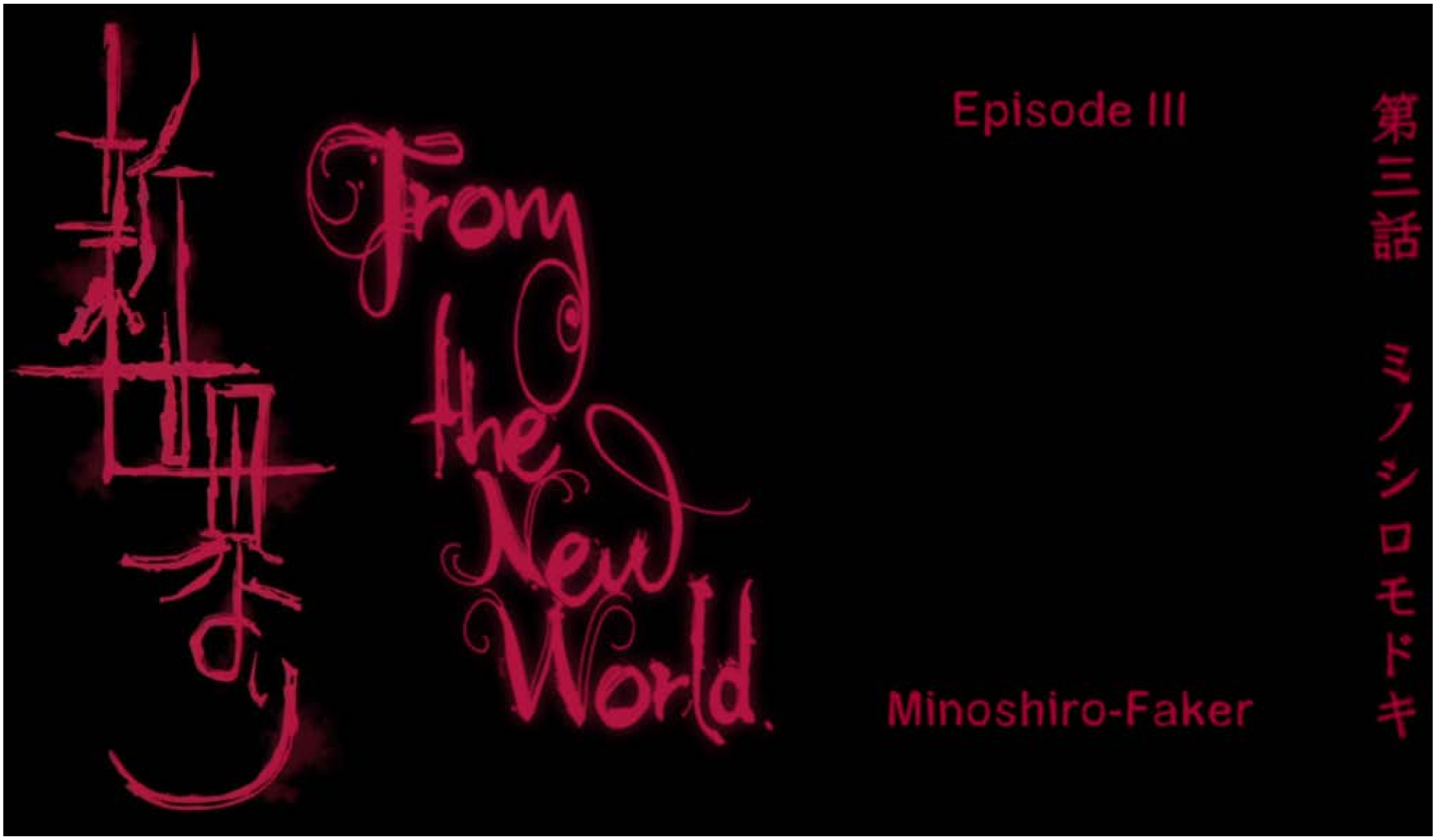
Here the Japanese is a bit closer to handwriting but handwriting or brush fonts would be too messy.
The font I used has that varying thickness and the ends of letters fit pretty well.



This shouldn't need any comment.

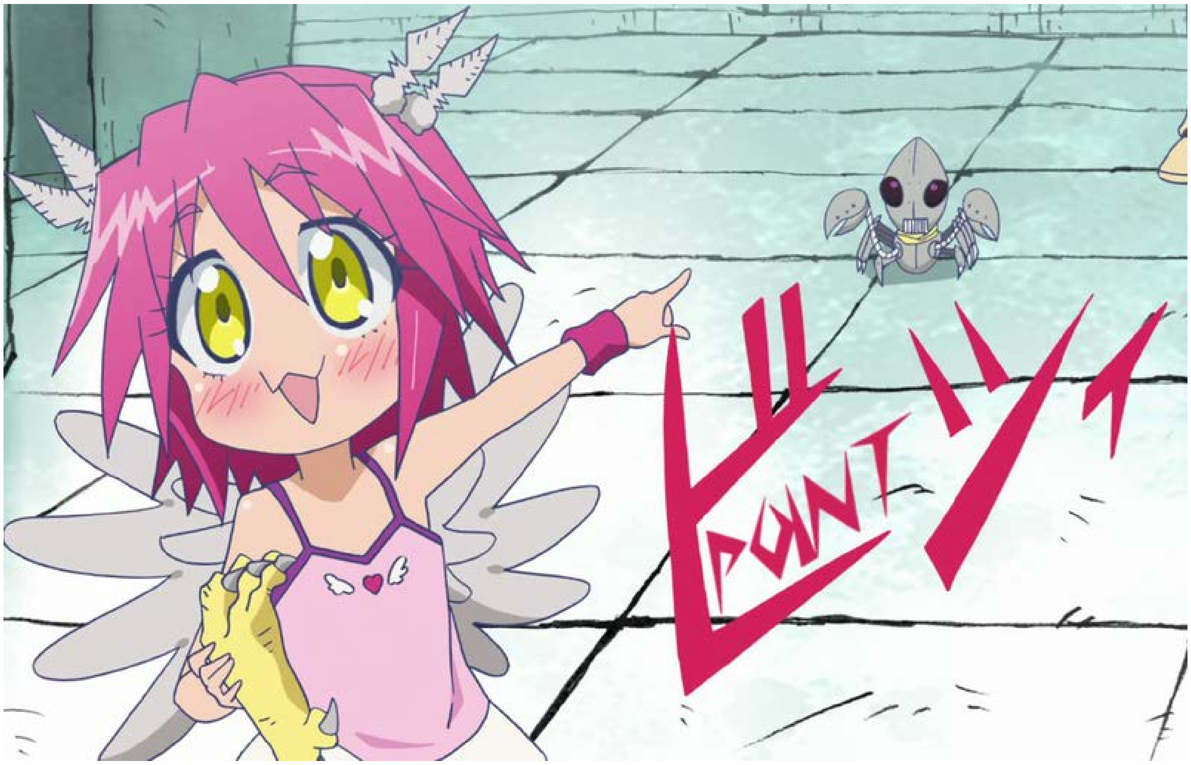


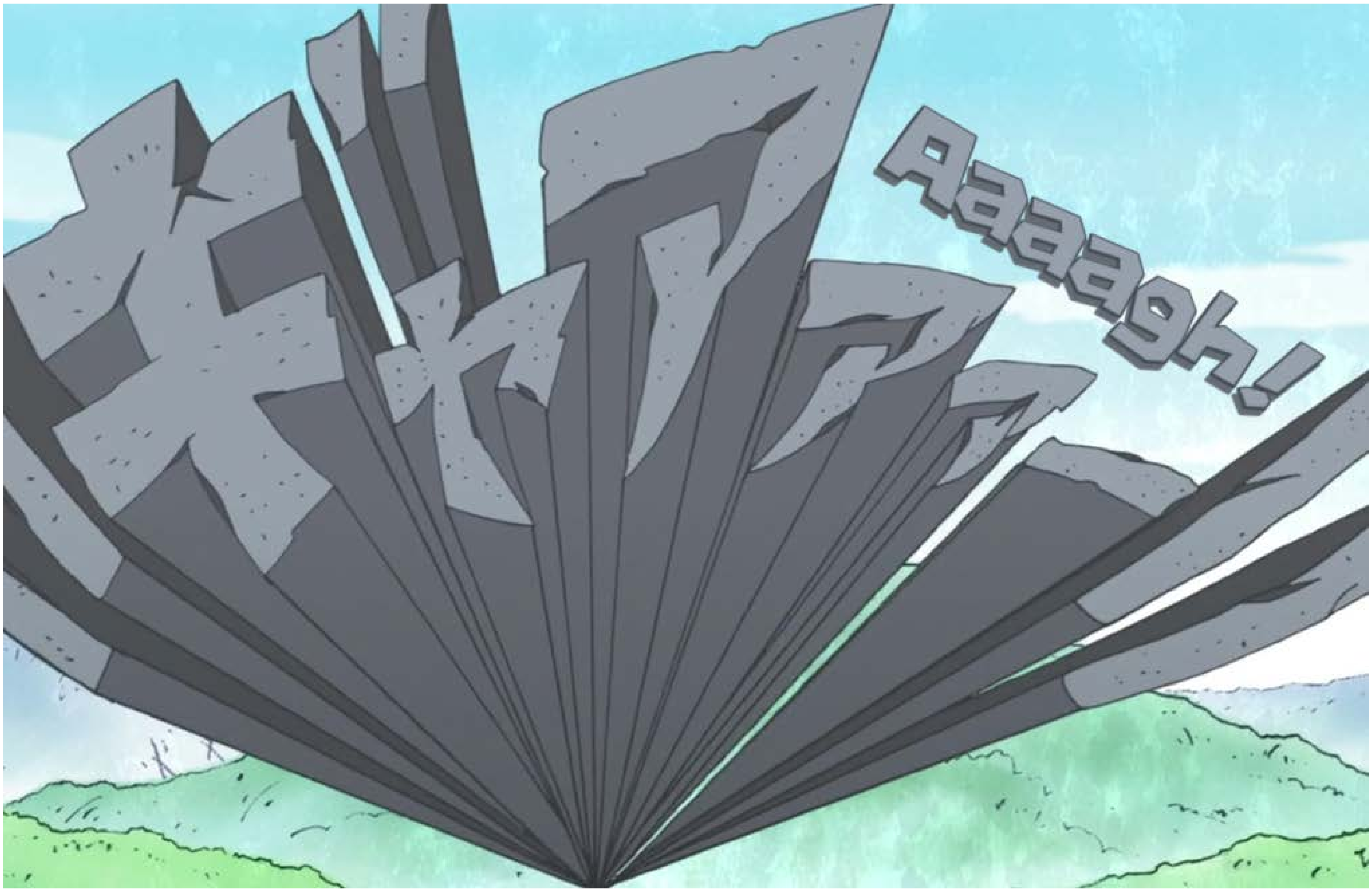
I like typesetting Shinsekai Yori because it has new challenges every episode.
These are titles from episodes 1 and 2.



And this is from episode 3. I have no idea how all these moonrunes are even readable.

A few examples from Muromi-san:

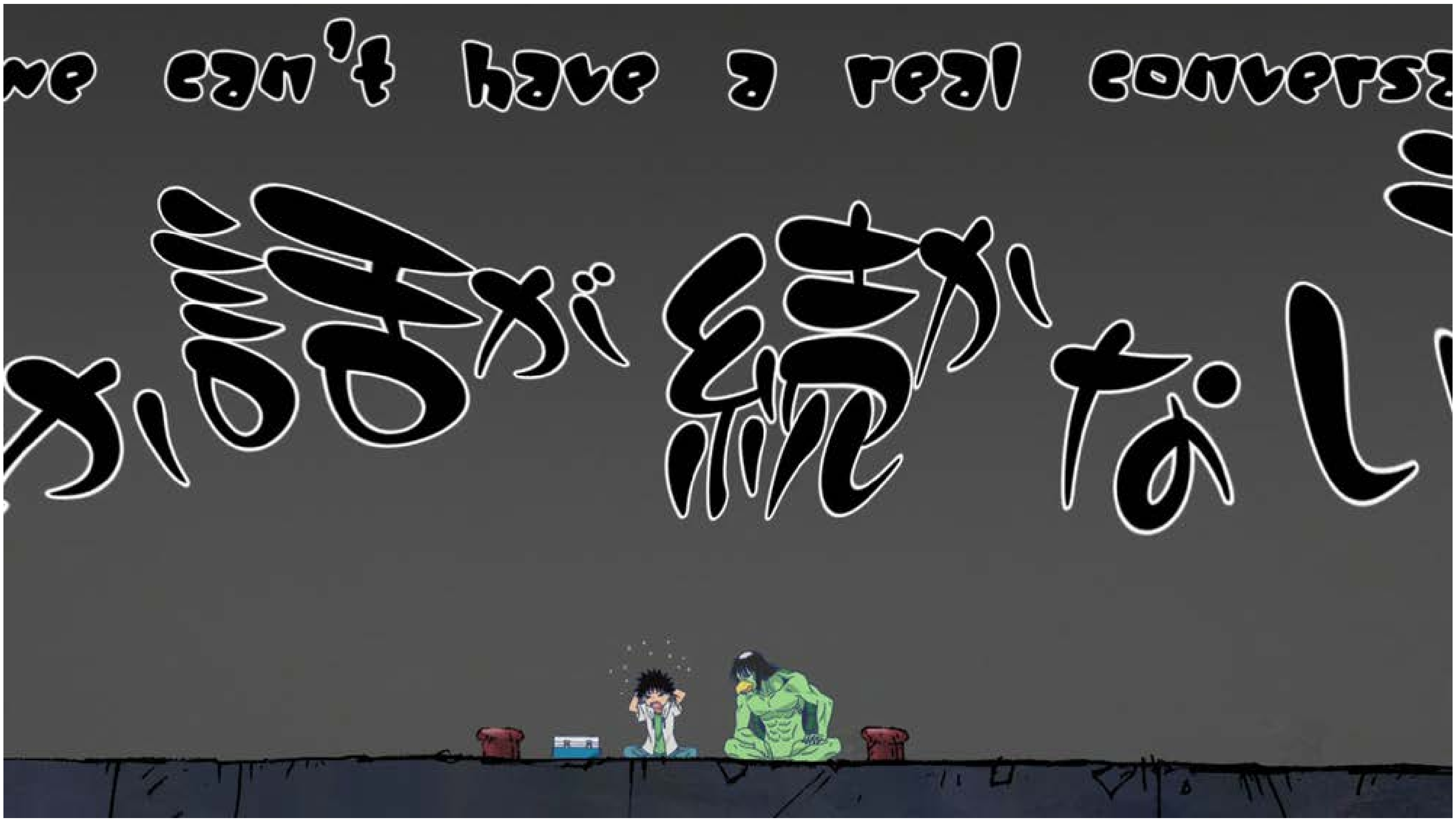



Aaaagh!

あぁーん..
Ahh!

ダメ〜んん...
Naaa...

あーっ！
Hahaha!
ハハーャハハ



Just imagine how bad they would look with Arial...

« Back to Typesetting Main

Typesetting: Blending In

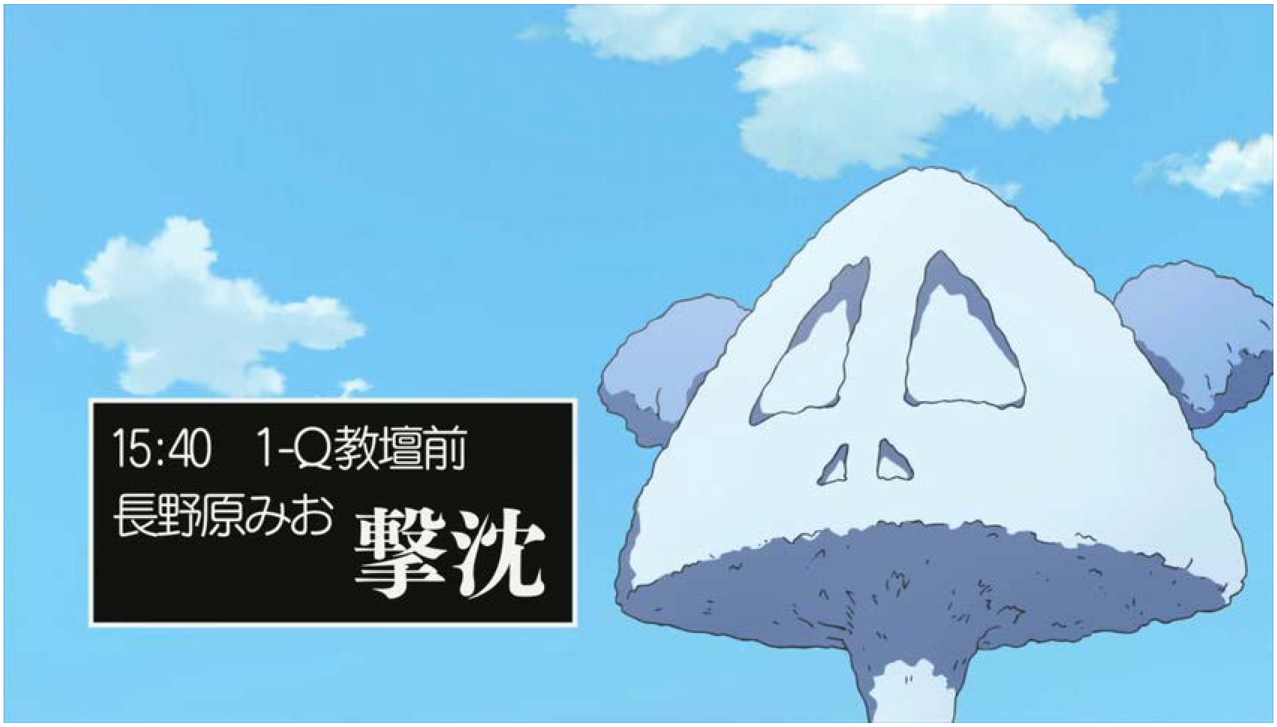
Ultimately what you want to achieve is to make your typesets blend in, to make them look like they belong there.
If possible, to make them look like they were there in the first place, so that when somebody watches the episode,
they don't think "oh, somebody put text here to tell me what this sign means."
Ideally the viewer shouldn't even notice something was added.
The typeset shouldn't stand out, being the first thing you notice because it looks out of place.

Making it blend in prefectly is not always possible, so sometimes you just need to make sure it doesn't look distractive/odd/ugly.

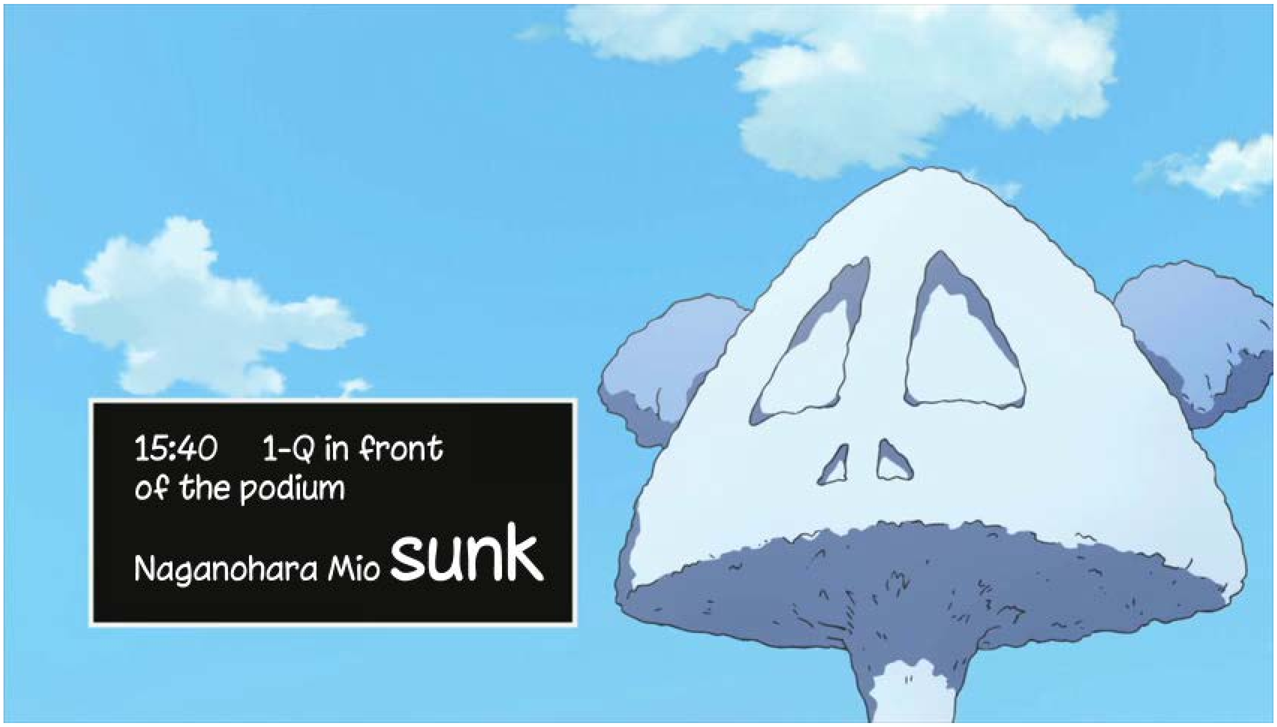
The way to achieve this is by combining all the things you've learned in the previous chapters.
Choose a good font, find a reasonable position for the sign, make sure to get all the colours, borders and shadows right,
align it correctly, use as much blur as needed to make it look natural, etc.

Sometimes there are several reasonable ways you can typeset a sign, so choosing the best one will make a difference too.
[Sometimes that requires thinking outside the box a bit.]

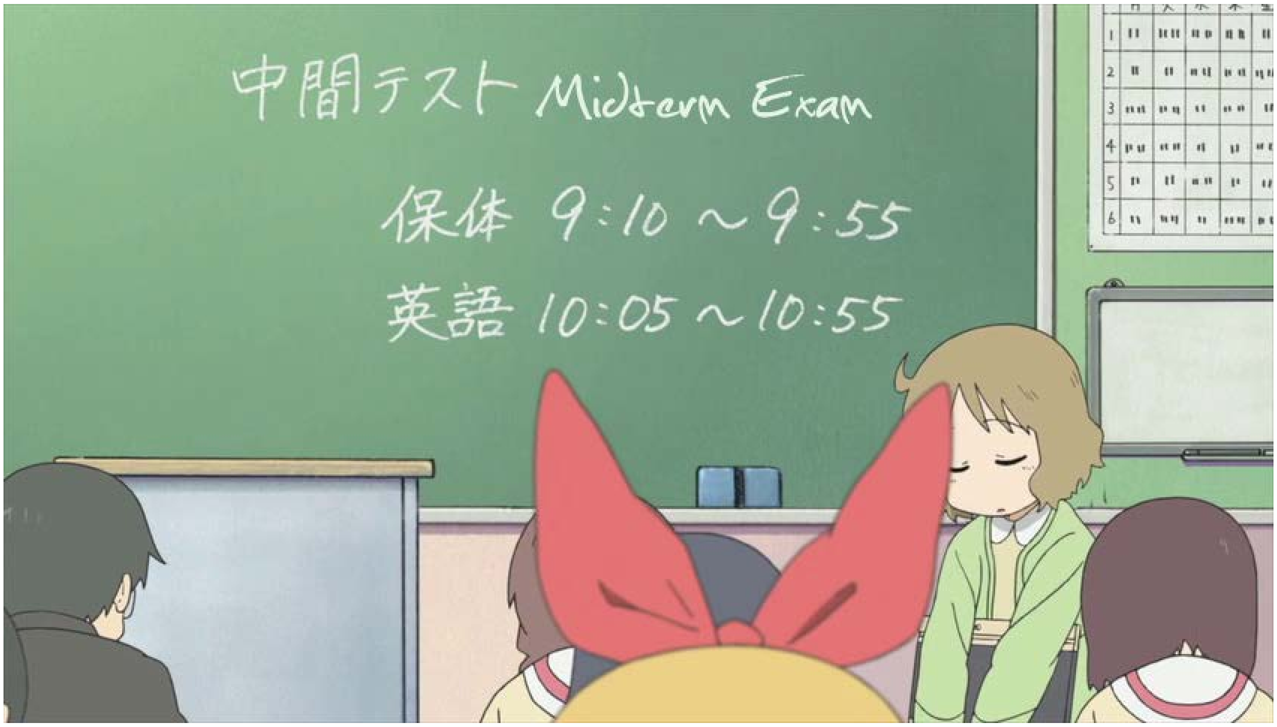
For example this one...



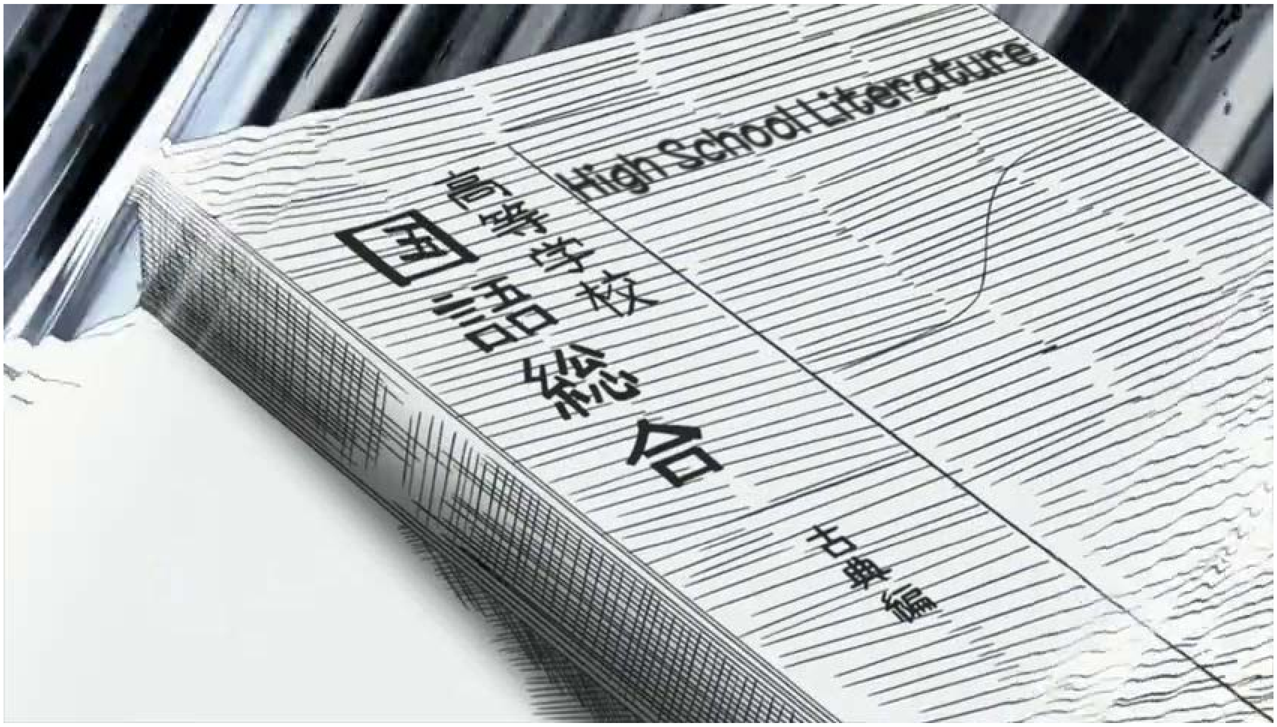
You could try to squeeze the typeset somewhere around the JP letters, or put it outside the box, but that would pose obvious problems.
The thing you can make use of here is that the box is a regular rectangle and the background is one solid colour.
Which means it will be extremely easy to just replace the sign without any interference.



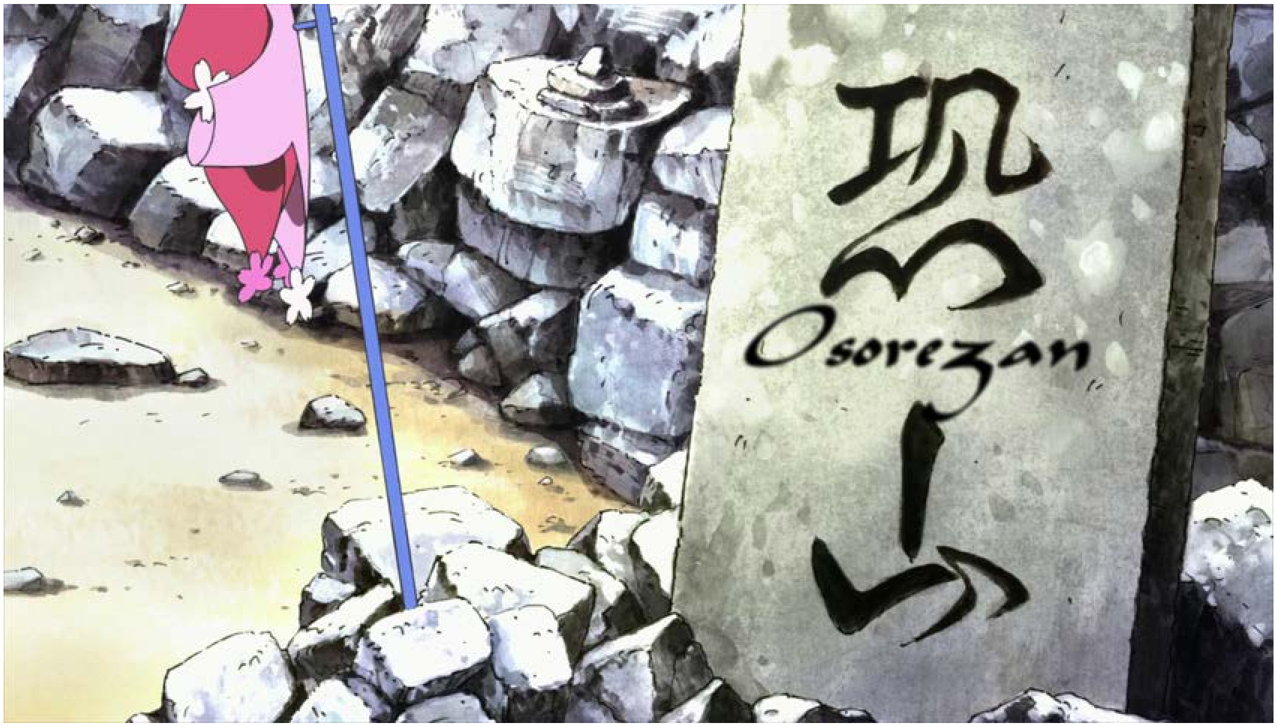
I didn't even use drawing mode here, just 'opaque box' for border.
2015 Note: But that's dumb, so don't do that. It was a long time ago.



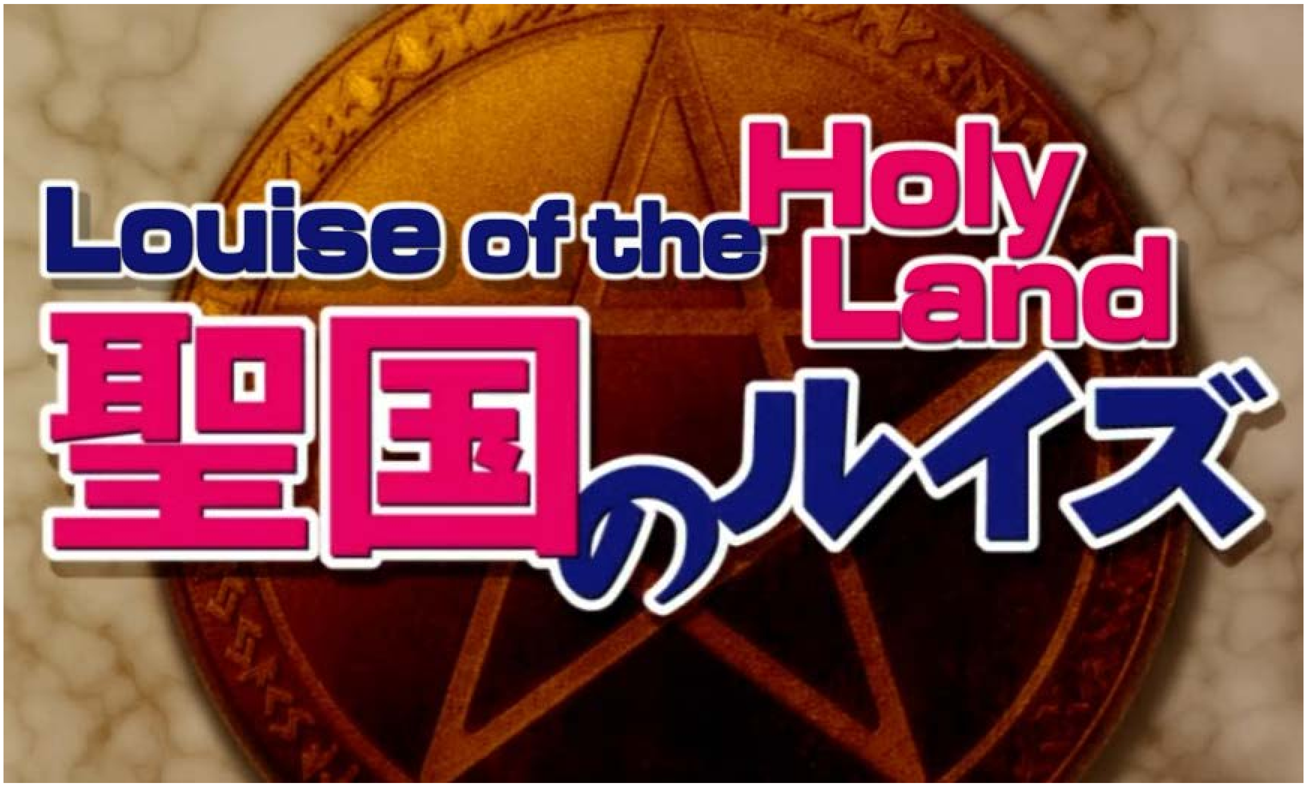
Here you need to pick the right font and match the size and thickness of the letters. [And obviously have no border/shadow.]
At first glance you'll hardly notice anything was added here.
Looking at it some more, I'd say it could use a little bit more blur, and a little bit of alpha or slightly darker colour.



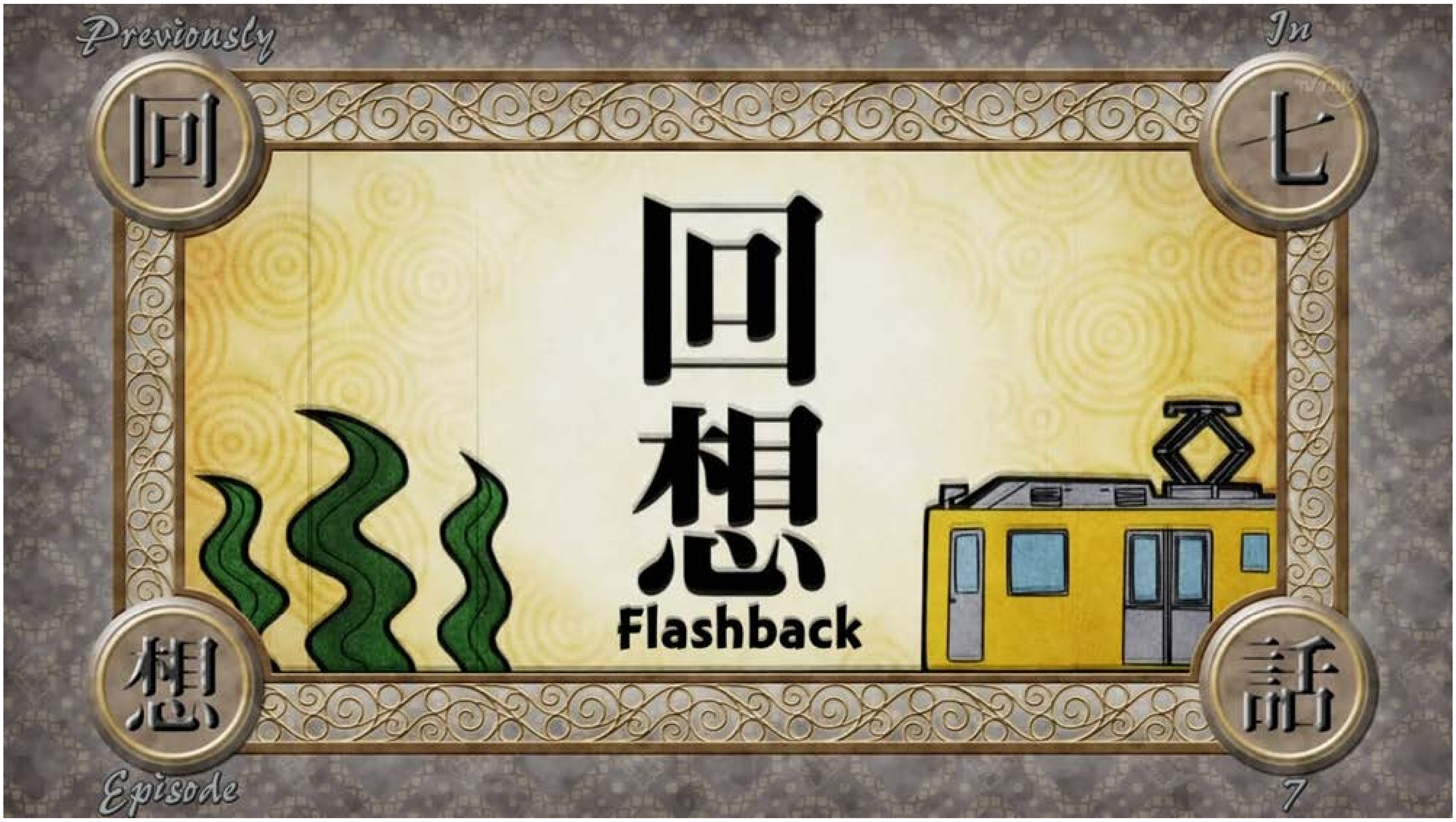
Here the challenge is aligning and positioning.
Forget trying to squeeze it next to the JP or rotate it ~90 degrees [in relation to current orientation].
Just use a place where it comfortably fits and looks like it might be the title of the paper.
Align correctly in both directions and add some blur to avoid jagged outline.
Also make sure you're not using pure black just 'cause your style has it. Almost nothing in anime is pure black. Match colours precisely.



Use good font, match colour, add blur. Now imagine Arial with some stupid white outline and no blur, like some derps would do.



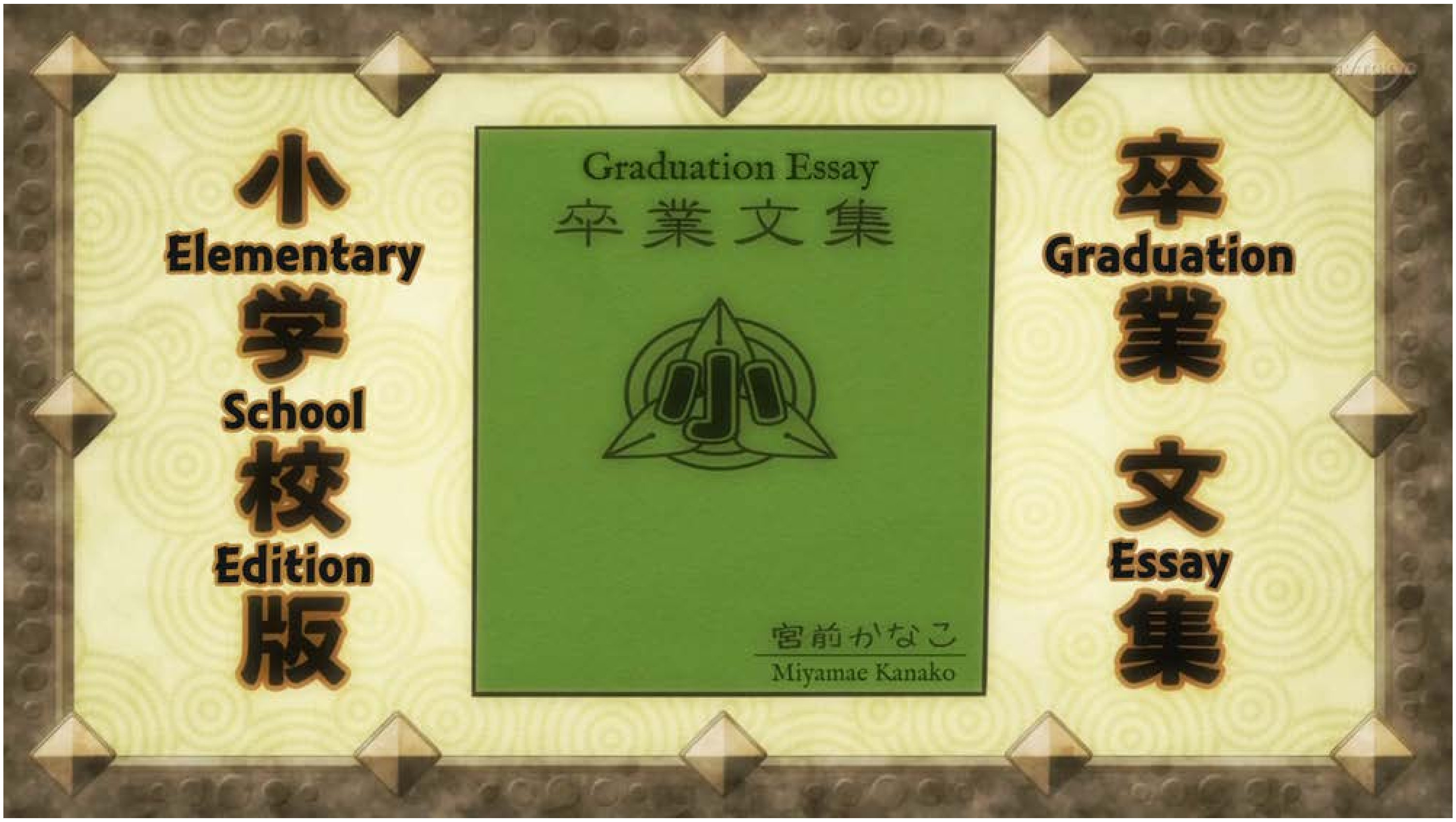
Nothing much inventive here, just follow the basic guidelines and imitate the original sign.
Use a somewhat square-ish, thick font. There are 3 lines [in script], each of them has 2 layers.
Top layer has matching colour [red-ish/blue] and a shadow [no border]. Match the shadow distance.
Match the transparency/colour of the shadow, not just black "cuz it was there".
Bottom layer has white outline and another shadow, this one with a lot more transparency.
Since the whole trick here is to really just get the basics right, make sure you notice all the borders and shadows correctly.
I almost missed the second shadow here. Sometimes you'll need more than 2 layers to get it all done right.



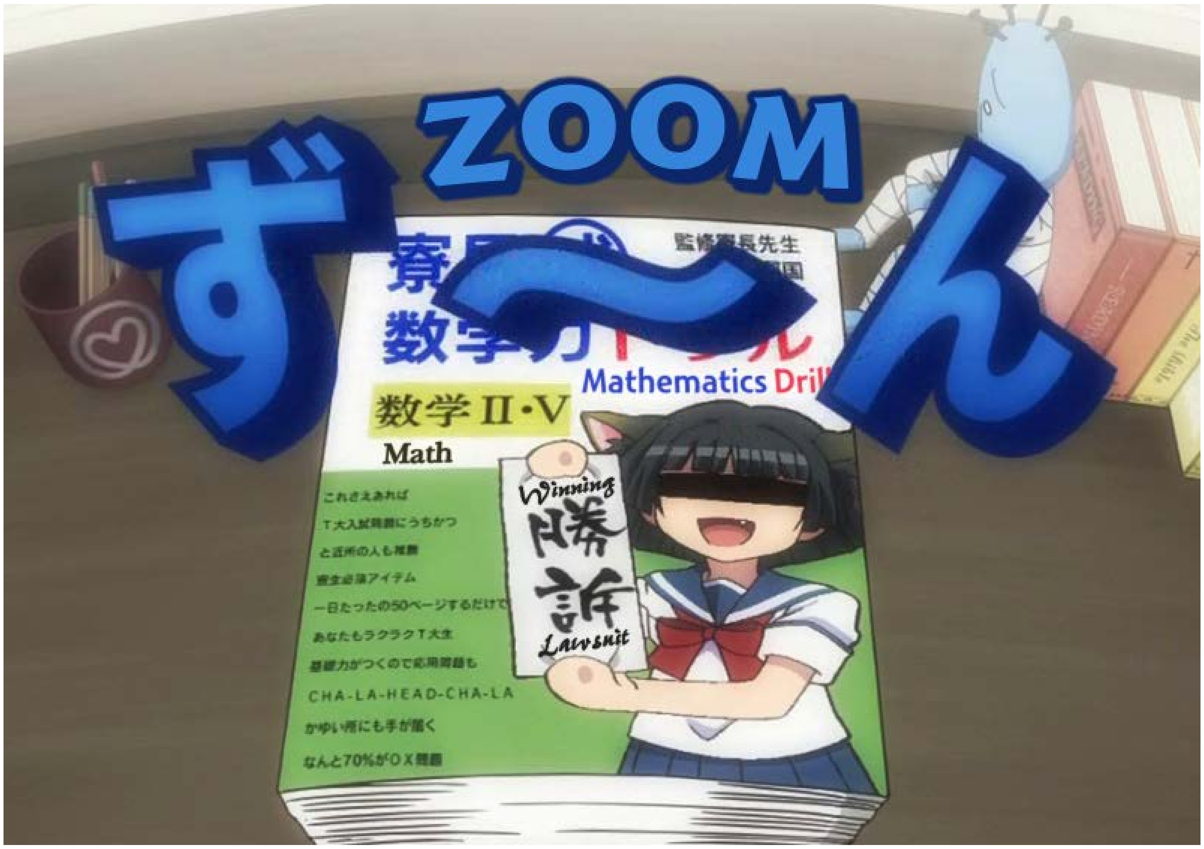
The 4 corners... would be hard or pretty much impossible to match, since there are multiple effects, embossing, texture etc.
So I decided to kind of approximate the colours and at least make it look nice. Which means mainly using a nice font.
Add shadow for at least some sense of depth, and play with the shadow/border till it looks decent enough.
Since you can't match this exactly, there would be plenty of variations of how you could do this.

IMO you're better off using some nice script font even if it doesn't quite match, than using TimesNR which might technically be closer but would look like shit.

As for the Flashback... the font is the easy part. More interesting is the shadow.
And even more interesting is that the sign was moving around earthquake style, changing the direction of the shadow on every frame.
So what I did was to time it frame by frame, use \xshad and \yshad on each frame to change the shadow direction,
and adjust \pos for each frame to folow the shaking movement of the JP sign. Good thing was it only took 20 frames.
[If you're learning, you're not required to follow the movement & shadow frenzy. Just saying it can be done and how.]



The ones on the sides obviously can't blend in perfectly. There's hardly any space for them anywhere else,
so the best you can do is to match everything as closely as possible. This isn't even as close as possible, but it's good enough.
The other two can be done much better. Positioning is pretty obvious, so good font, thickness of letters etc., correct colours and a bit of blur.
The top one has even that background blur [you add very little border, like 0.1-0.2, a lot of blur [4-6], and adjust colour or transparency till it looks good].
The bottom one doesn't have that because I was too lazy. Negtive points for me. The ones on the sides also need more blur.

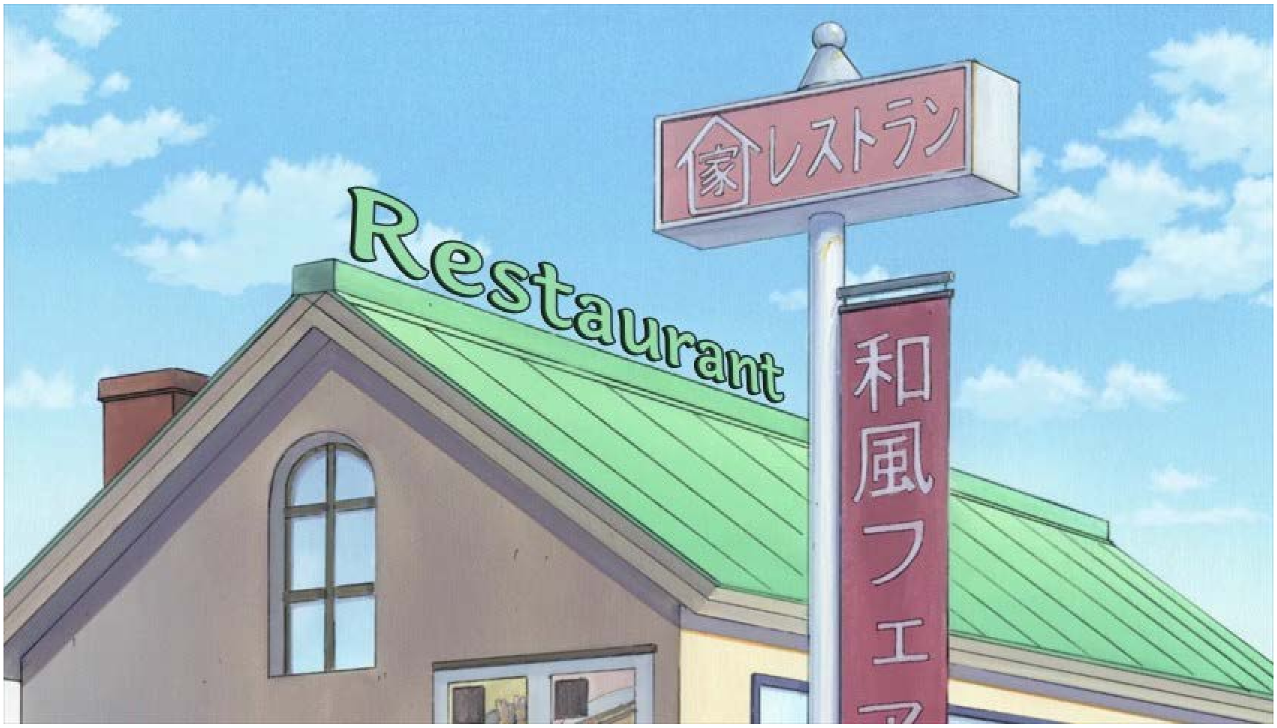


Four typesets here. The Zoom would look much better with another, top layer without border/shadow and with some blur.
However the whole sign was moving and increasing in size, which means \move and \t, so adding another layer and more blur might munch on the CPU too much, so I left it like this, especially since there are 3 other signs in the picture.
The 'Math' doesn't look too great, but font is ok and it doesn't stand out too much, so it's good enough.
You could replace the letters on the yellow background with the 'Math,' but I don't like replacing signs if I can't replace ALL of them.
Having some kanji replaced with English while there are other kanji in the picture looks lame to me [though I wouldn't say it's 'wrong'].
Third one is 'Mathematics Drill.' The big blue letter covering the Drill is moving away till it's not covering the word at all.
So I used \iclip to remove what was needed from the Drill, and changed the \iclip frame by frame. [Well, I was cheating, so each 2 frames.]
Fourth one is the Winning Lawsuit. Didn't really give me many options as to what to do with it.

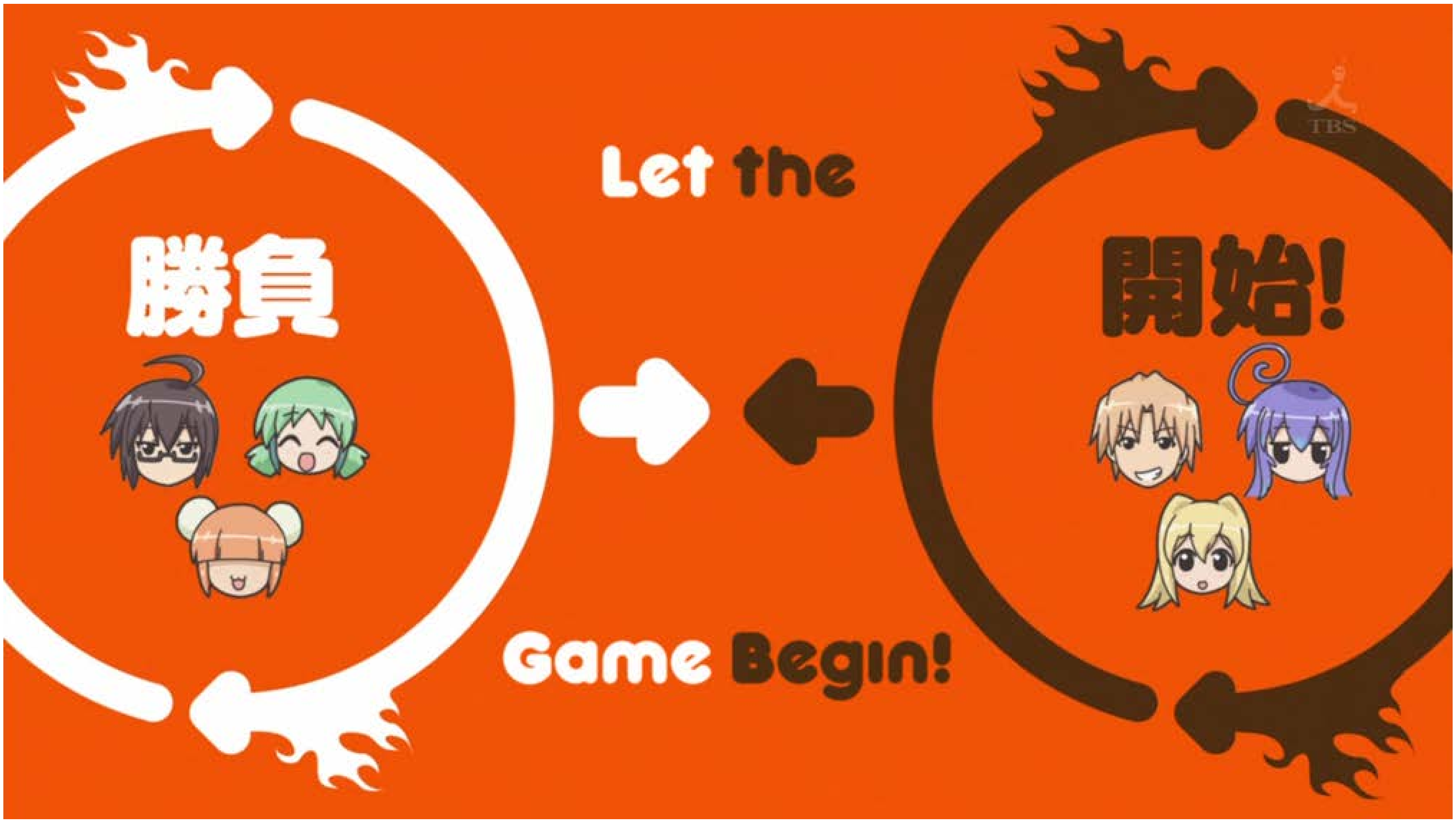
2015 Note: In retrospect, this whole sign looks pretty bad and could be much improved, and without lag as of now.

As you can see, I used 4 different fonts here, trying to match the style, thickness, size etc. [as much as possible]
It should be clear that all of the signs use blur. The Lawsuit could use a bit more, but then it was getting hard to read.

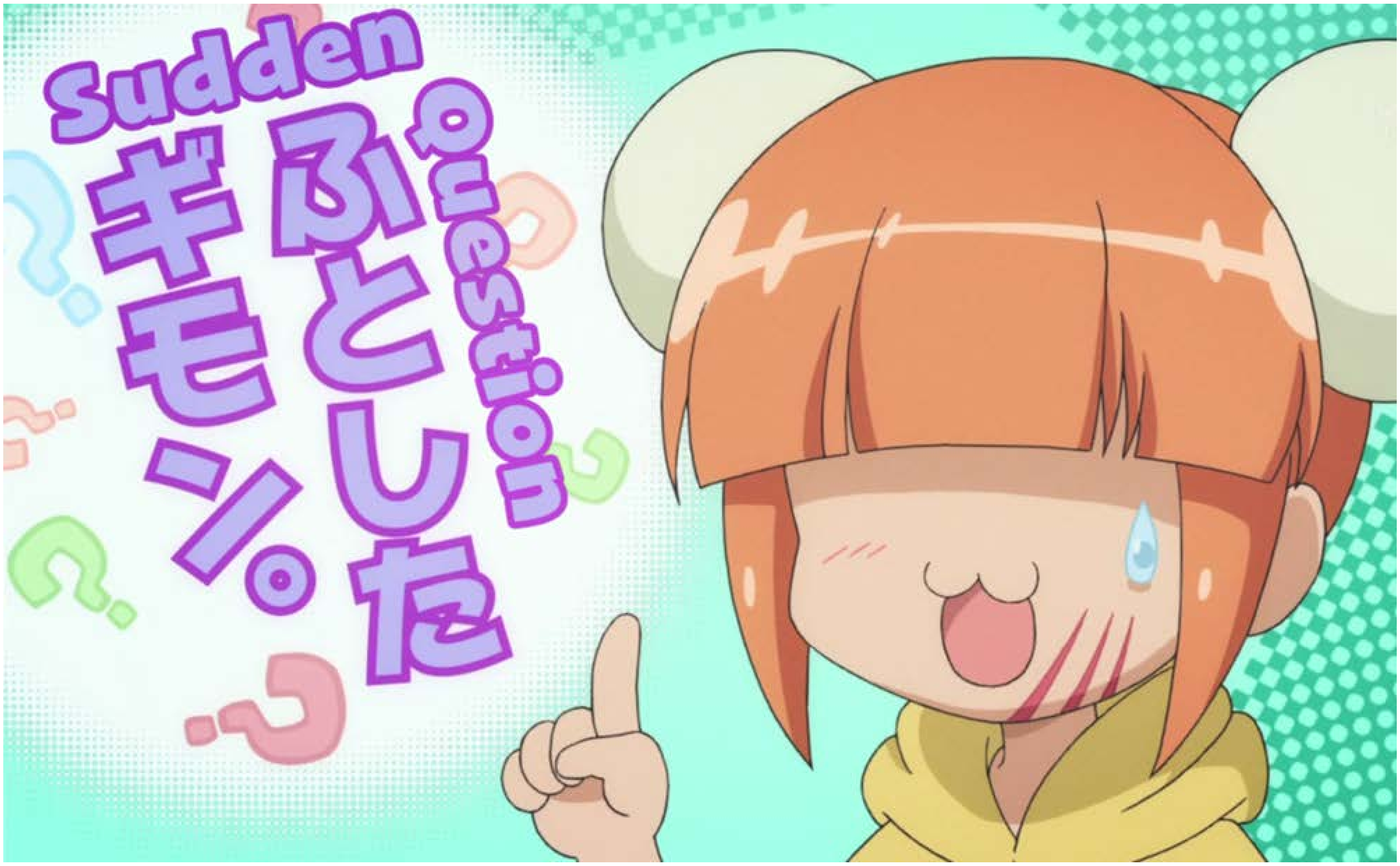
If you can't see the sign, it's blending in well.



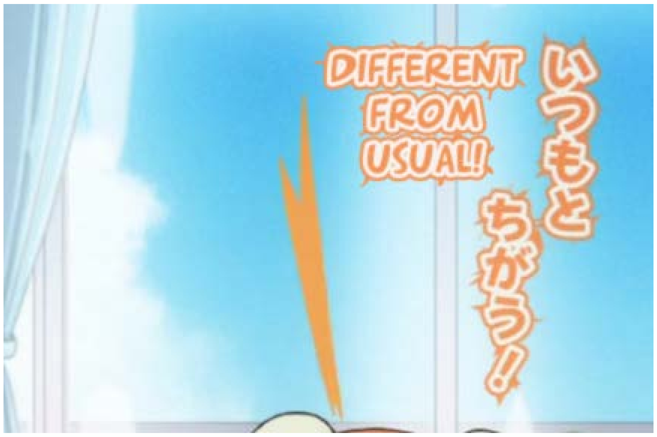
I dunno, I just thought this was a cool idea...
It's a typeset for the white-on-pink sign at the top. Clearly there wasn't enough space there, so...



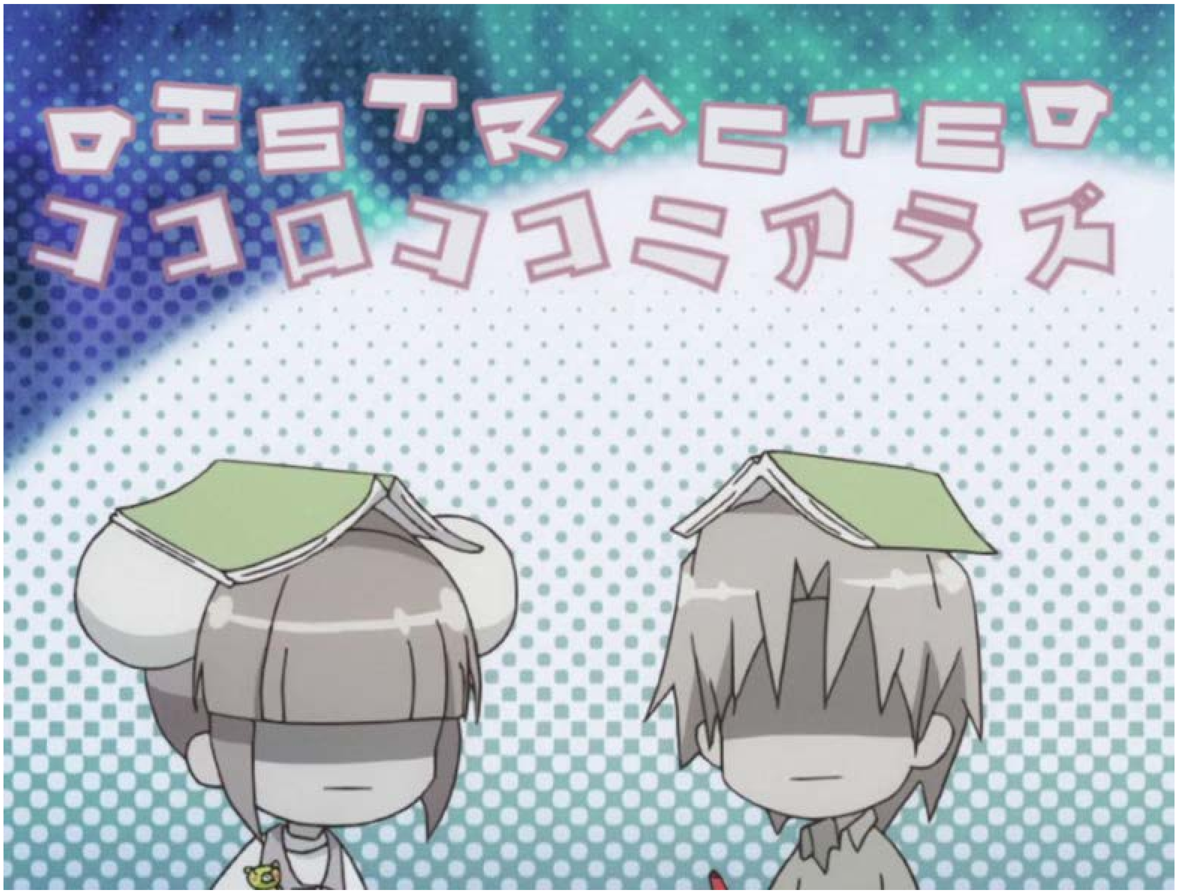
Speaking of having some imagination... an example of how you don't always have to stick the typeset right next to the jp.



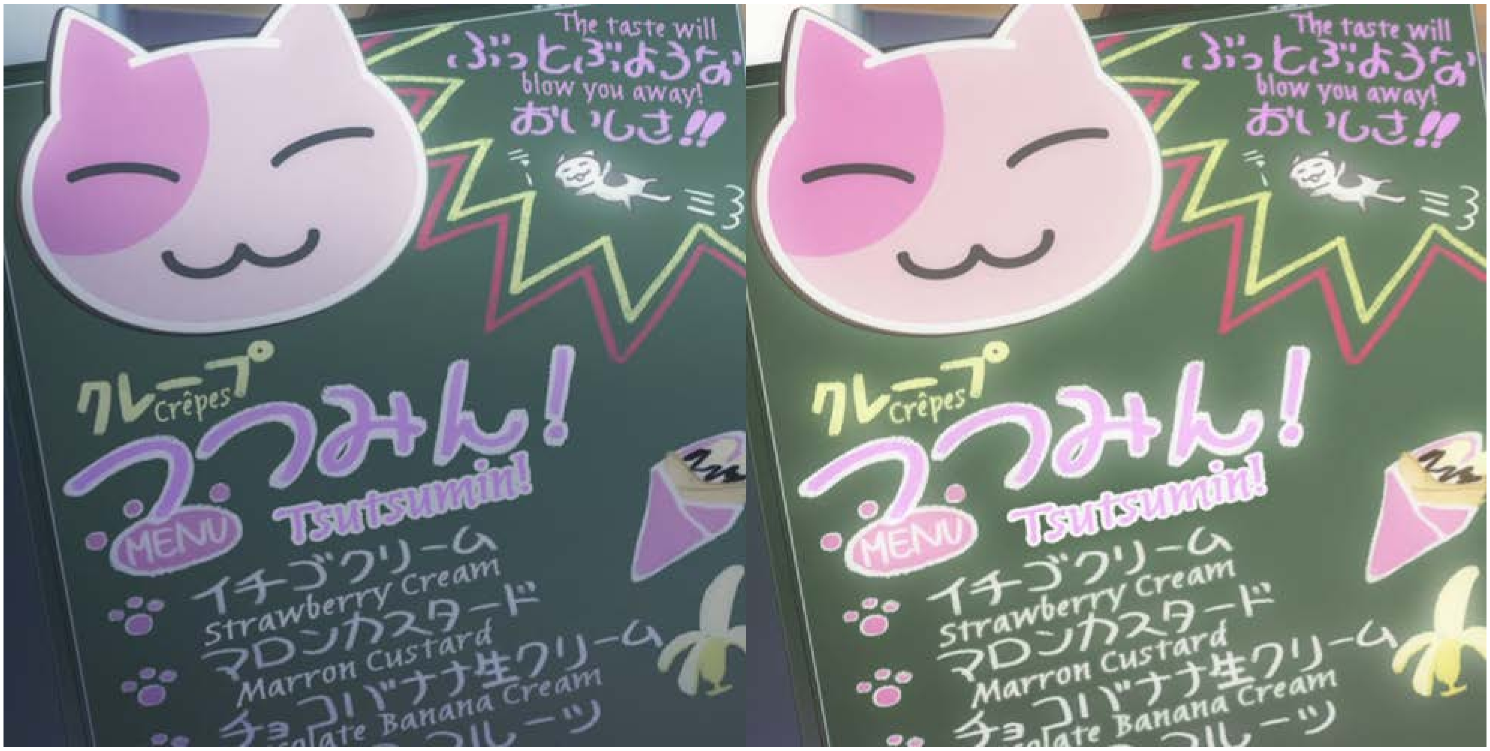
This is perfectly readable & looks good. Two layers for blur, of course, and changing colours letter by letter for the "Question."



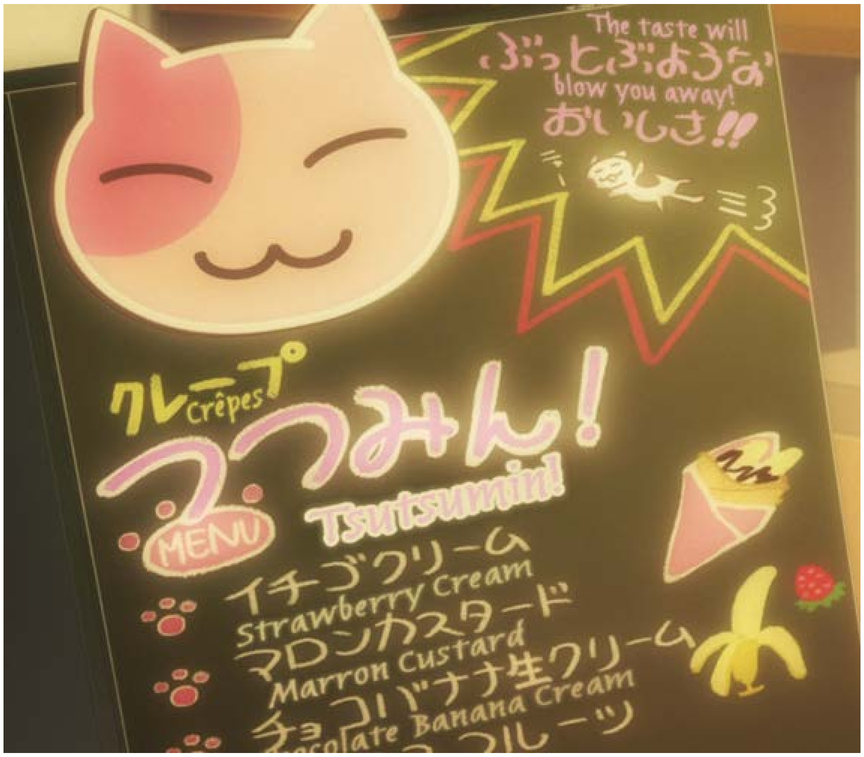
Spikes? How? Think about it... there must be some spiky symbols you could put under the letters...



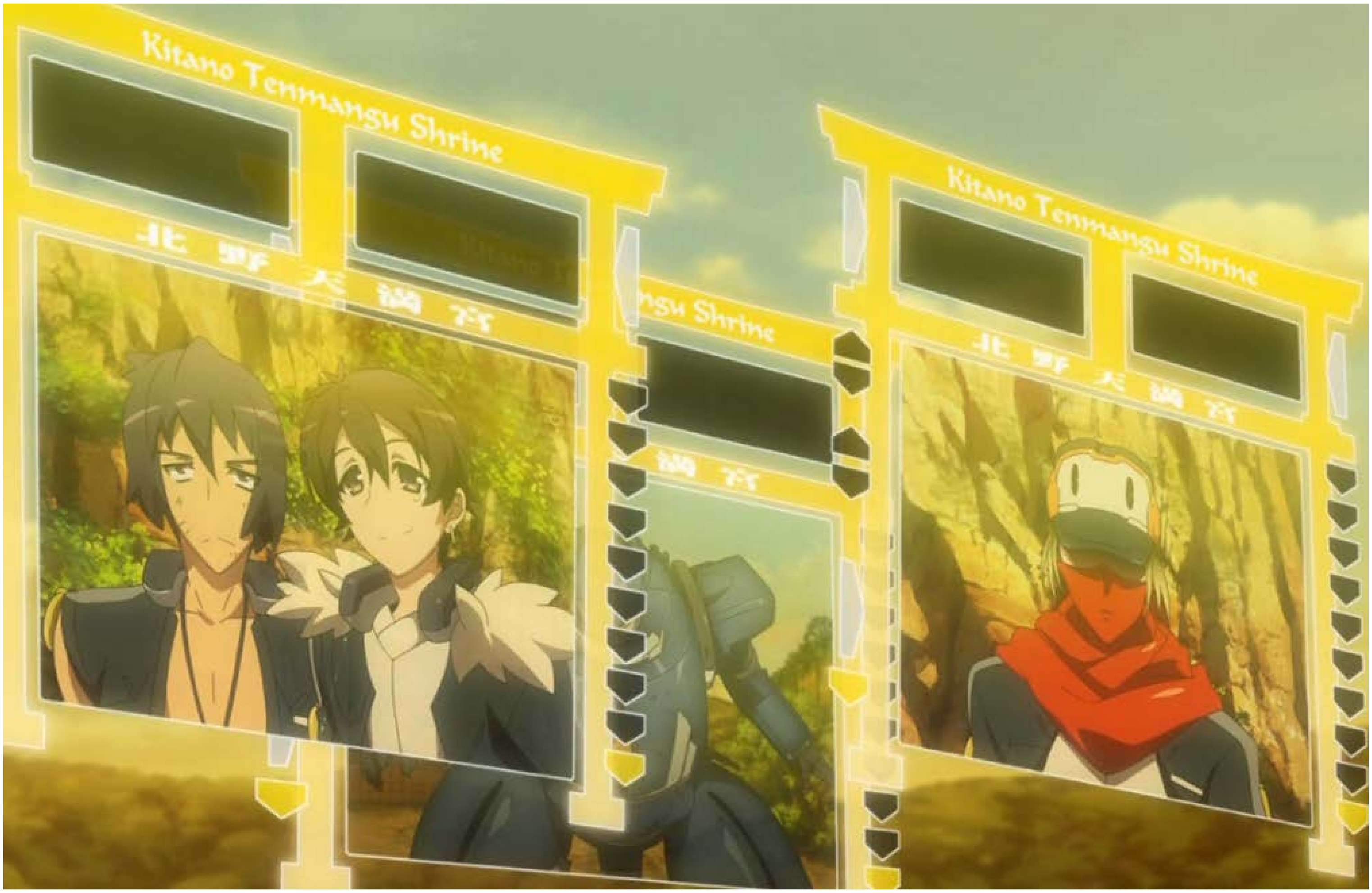
All in one line [but 2 layers for blur]. Different \frz for each letter.



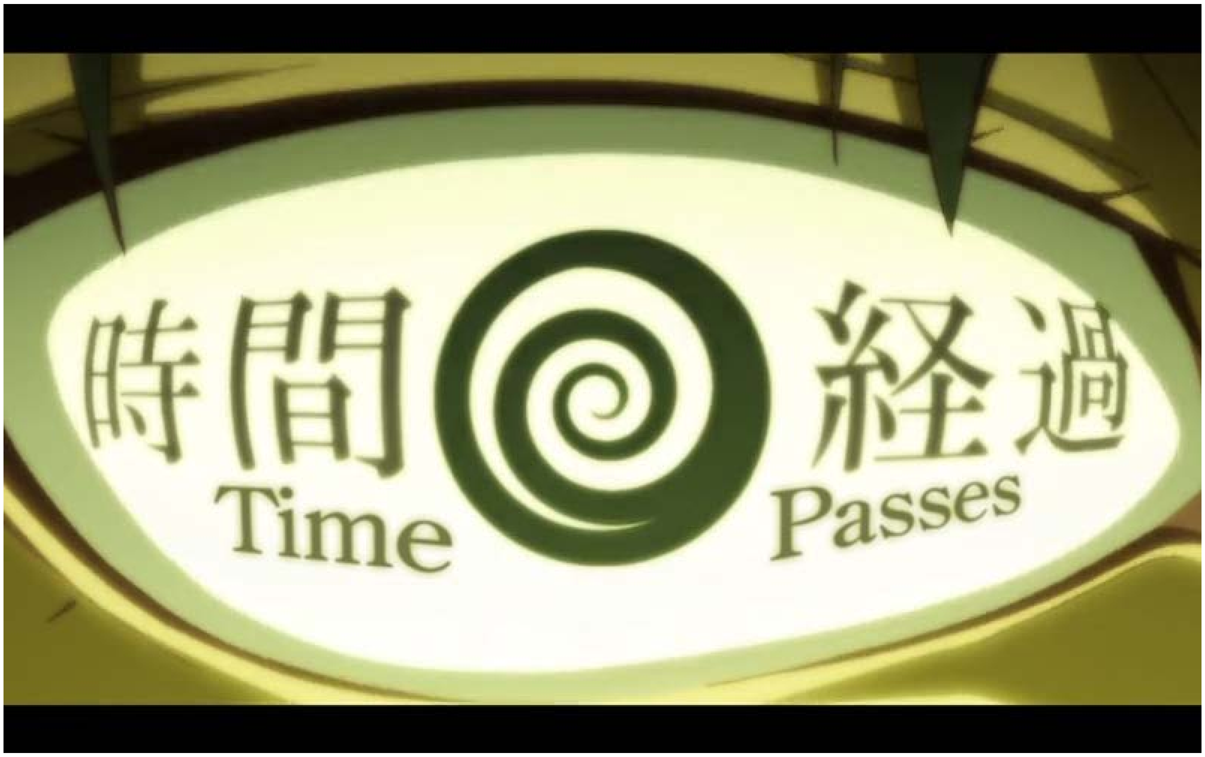
This is the kind of sign that makes most typesetters go "Oh shit [am I really gonna have to typeset this?]"
But it's really not that bad [unless it appears 20 times from different angles].
This only appeared 4-5 times.
The one on the right has extra layers for that glow [though less glow than the JP - \blur15 on several signs at once might not be so good].



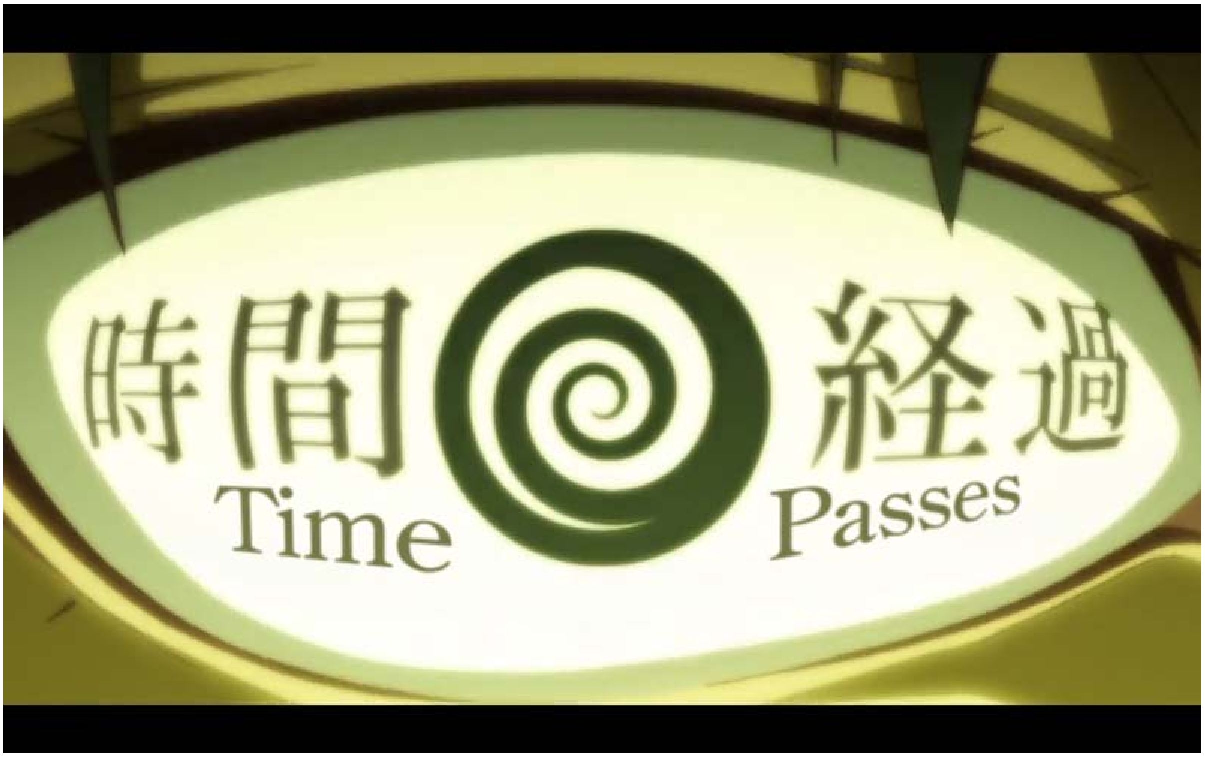
Here's one more in the evening.



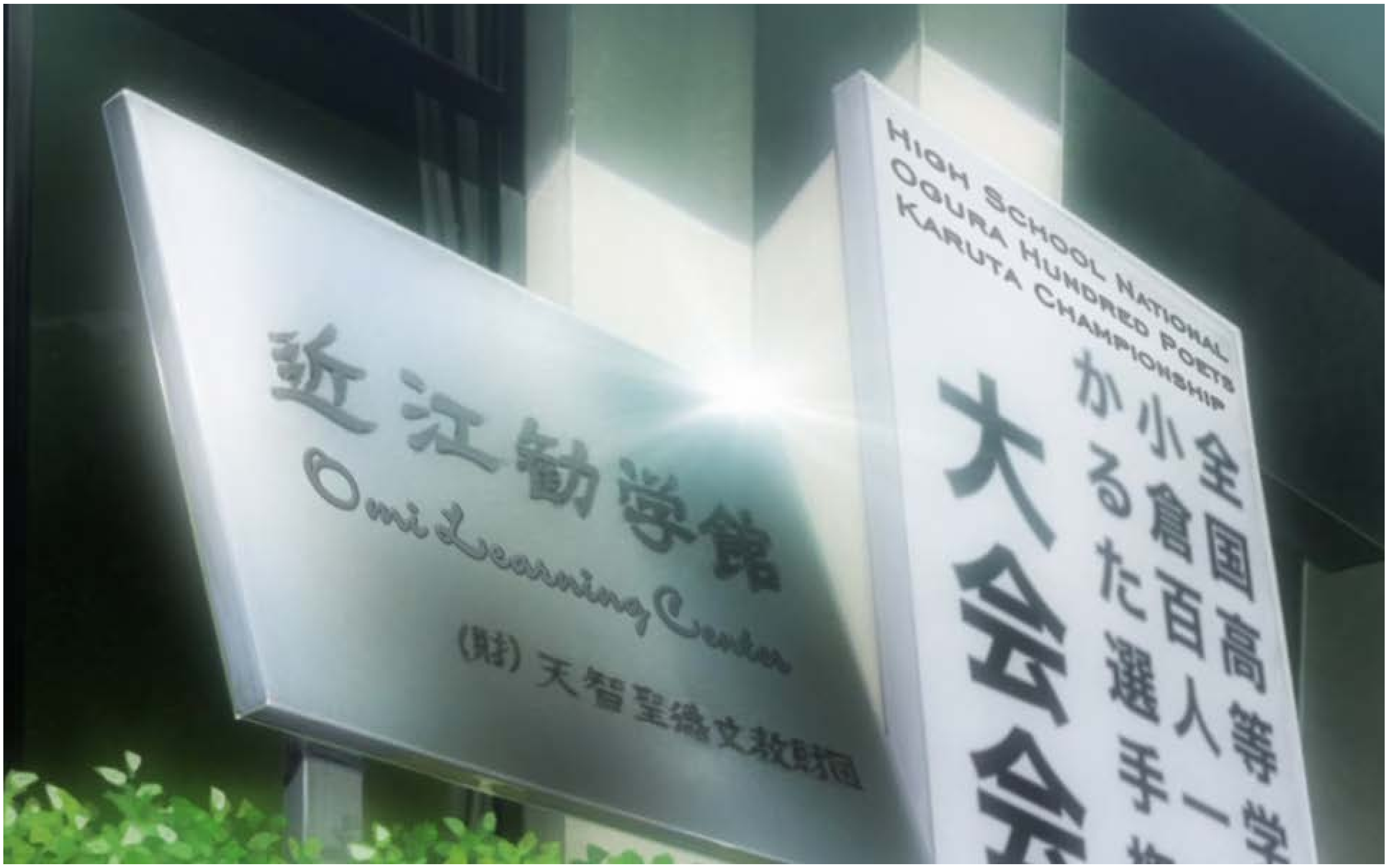
I've done so many Kitano Tenmangu Shrines that I couldn't count them. Here's 3 in one picture.
Aside from aligning & blurring, the middle one has some extra features in order to blend in.
I tried doing it including the semi-hidden part, and it happened to work really well.
The trick is pretty much this: {\alpha&HF0&}Kitano Tenma{\alpha&H00&}ngu Shrine
Btw nobody tells you it should go at the top of the shrine. You just know you're supposed to typeset the white kanji.
So most people would try to put it somewhere around the kanji. Like I said, don't do that. Find a less retarded / more elegant solution.



This may look simple enough but... if you do just one layer...



...it looks like this. See the difference? So yeah, another layer, different colour, a lot more blur.



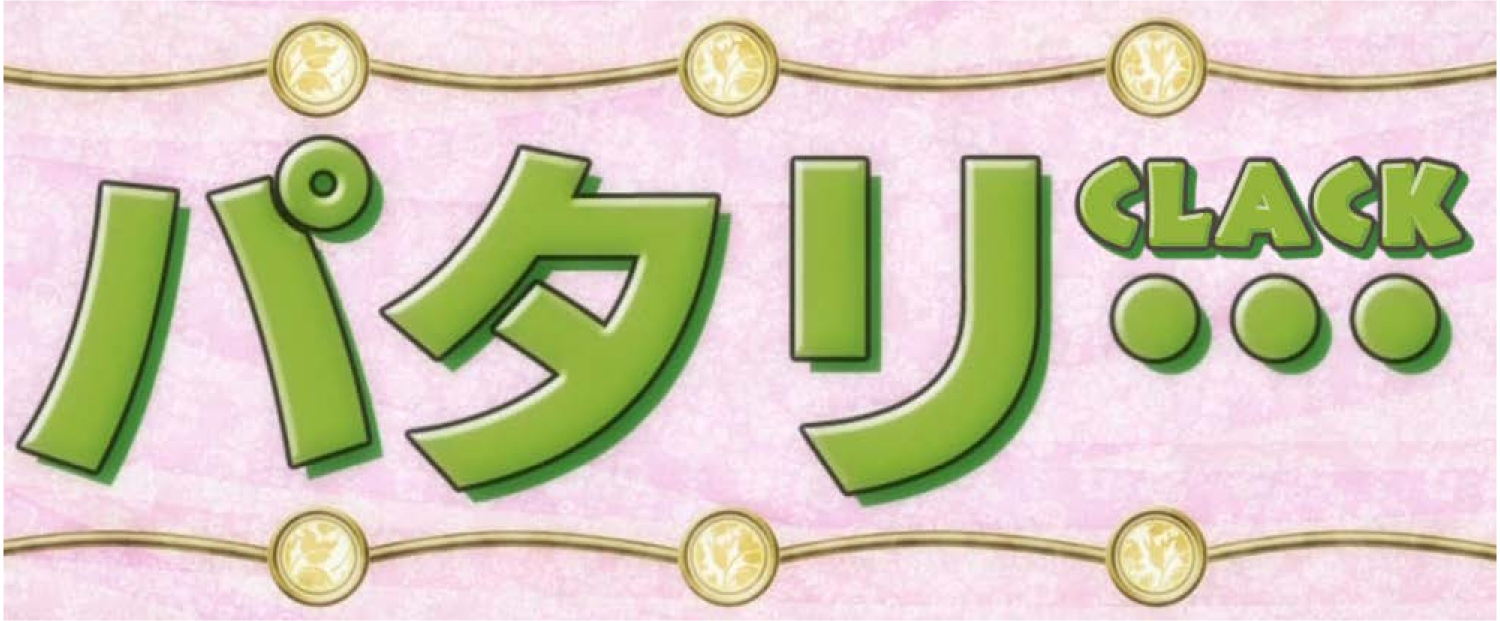
On the right... First choose a font. Then add some \fry and place \org about a mile away to get the alignment you need. [OK, maybe not that far]
That may take a while to get right. Adjust font size and position till it fits. Use \fax for fine tuning. Add blur, obviously. Then the colours...
The left side is brighter than the right, so I used 3 shades for each line. The whole thing looks like this:

```
0:02:54.02,0:02:56.07,names,Caption,0000,0000,0000,,{\an7\fs34\fax0.04\fscy110\bord0.5\blur0.8\frx6\fry18\org(839,950)\frz6.258\pos(800,-45)
\alpha&H80&\c&H464036&}High Schoo{\alpha&H75&}l {\alpha&H70&}Nat{\alpha&H60&&}ional \N{\alpha&H80}Ogura Hu{\alpha&H75&}nd{\alpha&H70&}red
{\alpha&H60&}Poets \N{\alpha&H80&}Karuta {\alpha&H75&}Ch{\alpha&H70&}ampion{\alpha&H60&}sh{\alpha&H55&}ip
```

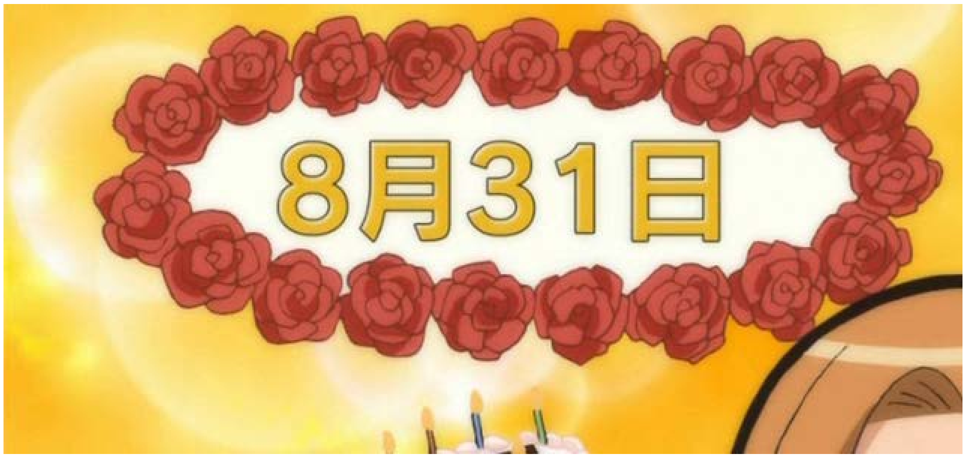
Omi Learning Center was even more interesting. Font, positioning and aligning is obvious. I don't even remember but apparently \frz\fry\fax.
The background changes colour/brightness **a lot** from left to right. That was the biggest challenge. \alpha probably helped with a bit of that.
Most of it was done by changing shades for main + outline colour for almost each letter. It also has both border and shadow.
There's \yshad on top of that, which actually created a bit of an effect that I didn't even expect, where the top edge of the letters looks brighter.
I'm not even sure why that happened but it was a nice bonus. Whole thing here:

```
0:02:54.02,0:02:56.07,Chiha-title,Caption,0000,0000,0000,,{\fs45\shad1\yshad1.5\blur1\fax1\b1\alpha&H99&\frx0\fry9\c&H928C7B&\3c&H595445&
\4c&H2E2D27&\frz338.838\pos(459,408)}O{\alpha&H90&}mi                                   {\alpha&H80&\c&H96927C&}Le{\c&H8B8E73&}ar{\c&H767A65&}ning
{\c&H666851&}C{\c&H5C5B45&
\3c&H4B4637&}e{\c&H515038&\3c&H423D2F&}n{\c&H48472F&\3c&H353024&}t{\c&H3E3D2B&\3c&H2C271C&}e{\c&H393826&\3c&H241F16&}r
```

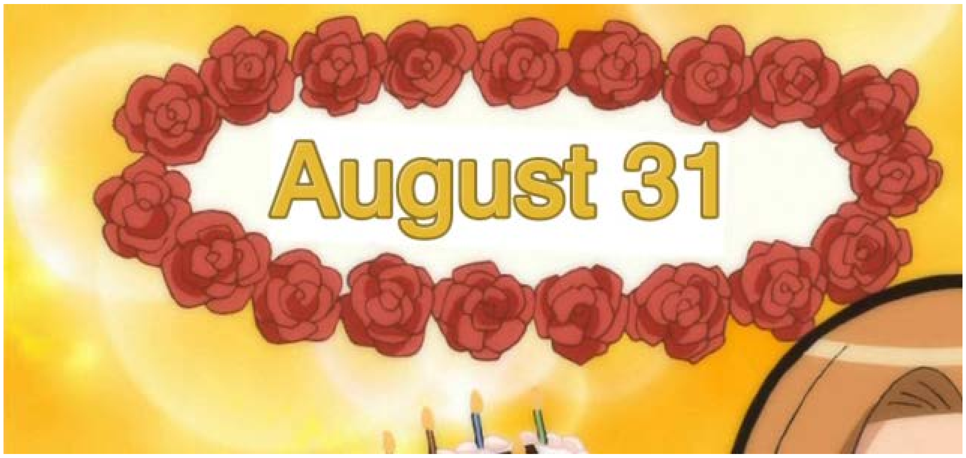
So yeah, sometimes it takes a bit longer to make it look good.



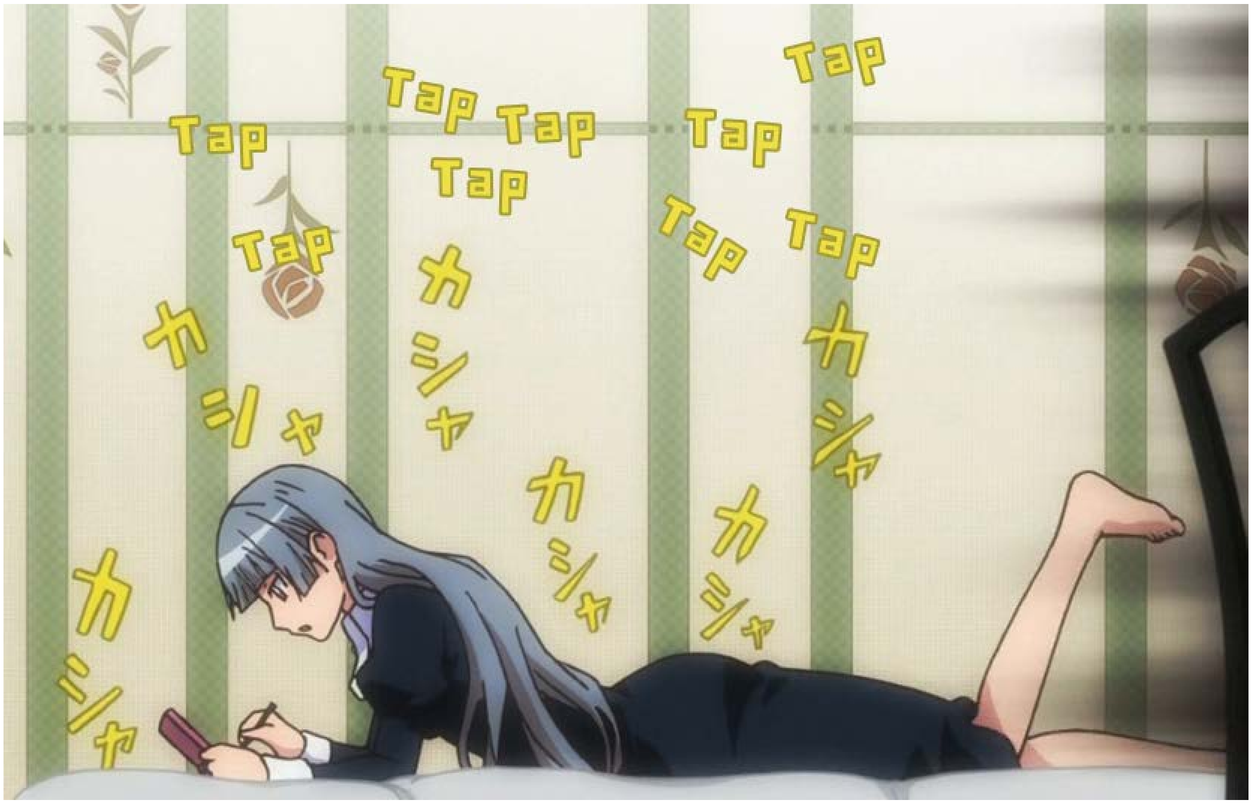
Signs like this are awesome. They're not moving, so you don't have to chase them across the screen, but you get to have fun with the details.
So how do you imitate all the borders/shadows? 5 layers.
Layer 5 - just the green letters without border. Increase letter spacing to avoid too much overlapping of the letters.
Layer 4 - \xshad-3\yshad-3 with the light colour.
Layer 3 - \shad3 with the darker green.
Layer 2 - \bord3 with the same colour as Layer 5 - to fill in the corners.
Layer 1 - \bord6.5 in dark grey and \shad5 in dark green.
...and of course blur on everything.



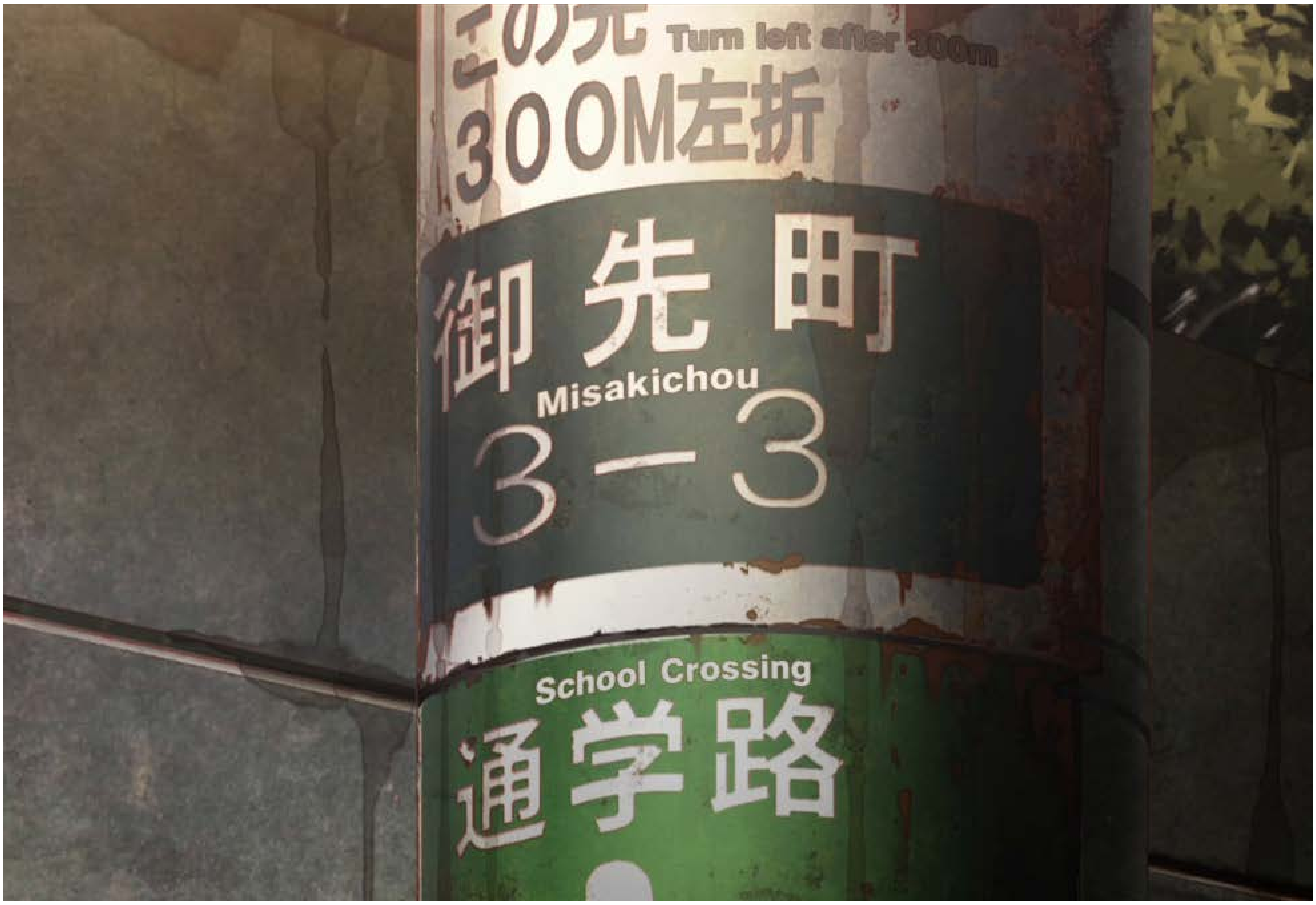
Similarly here you have a date with an outline but also a 'light reflection' on the left and at the top.
Pick a simple sans serif font that will roughly match the thickness of the letters.
Use layers with \xshad \yshad, or just an extra layer shifted a pixel up and left. Fairly simple.



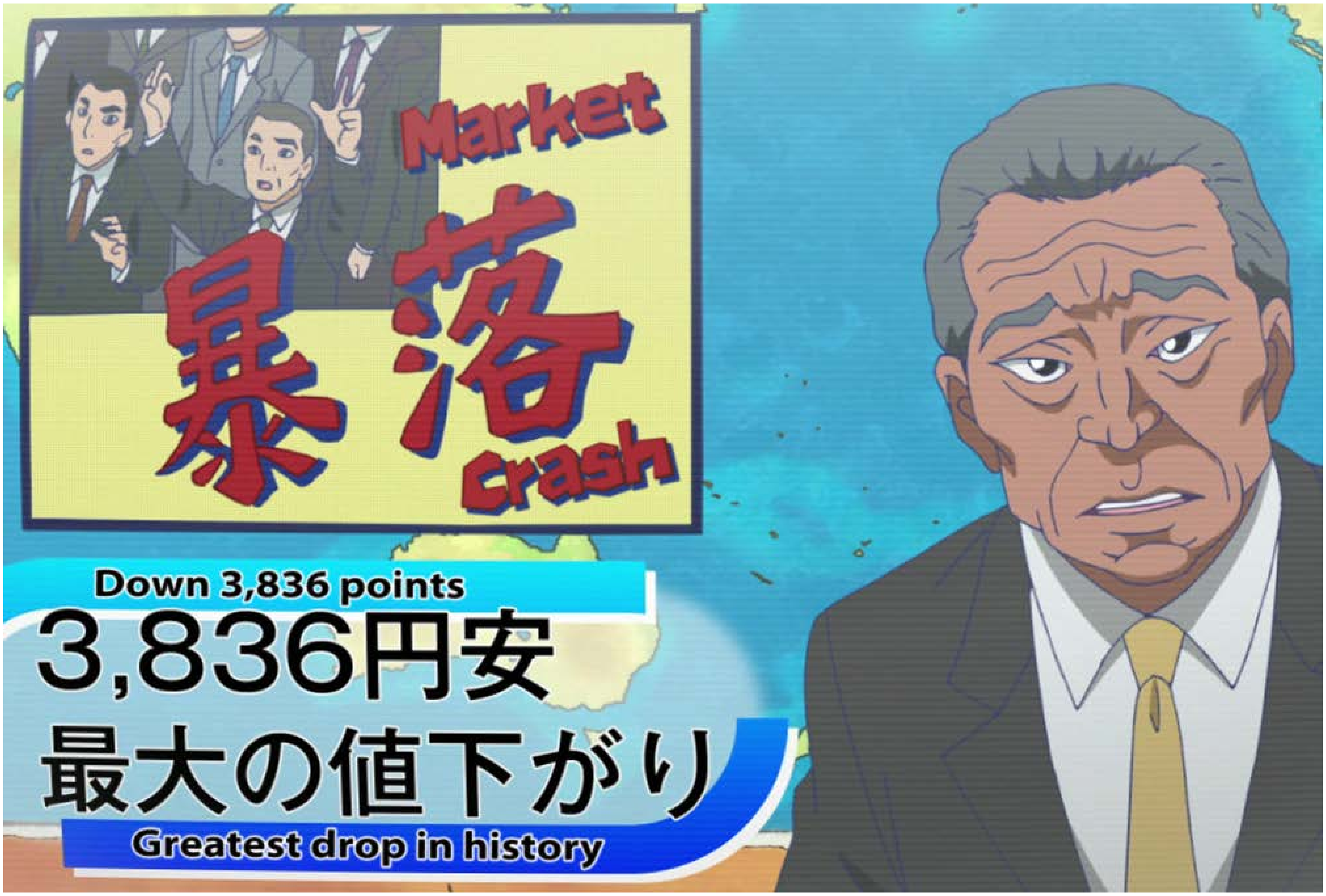
On the other hand, you don't really have to match this sign, in my opinion.
If you do something else that doesn't look out of place, it should work.
If someone says you're a retard because the sign doesn't match [I've seen that happen], he's just being dumb himself.
It seems unnecessary to me to "match" something that isn't actually on the screen [because you masked it].



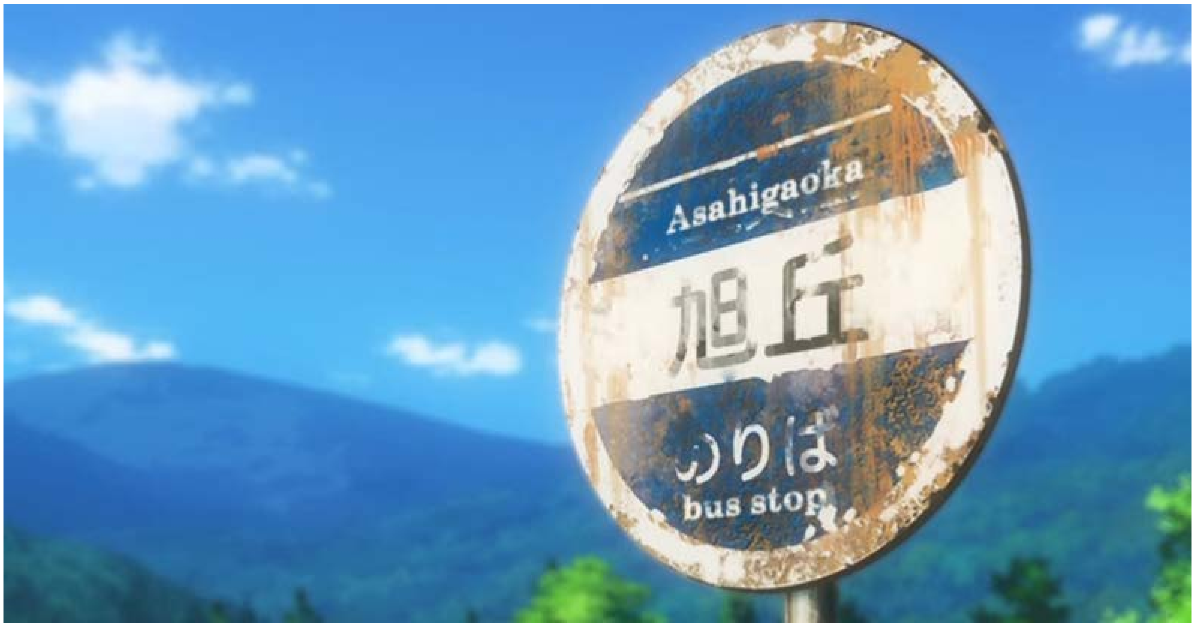
Another fun sign [Maria Holic is full of them].
The cool thing here is that it's all done in 1 line. All you need is one \N, different \frz for each word, and move \org somewhere far.
Additionally the words are moving back and forth every 2-3 frames, so I timed frame by frame and changed \pos by 1 pixel back and forth.
While 1 pixel isn't much normally, the effect when you have \org somewhere far is pretty cool with all those \frzs.

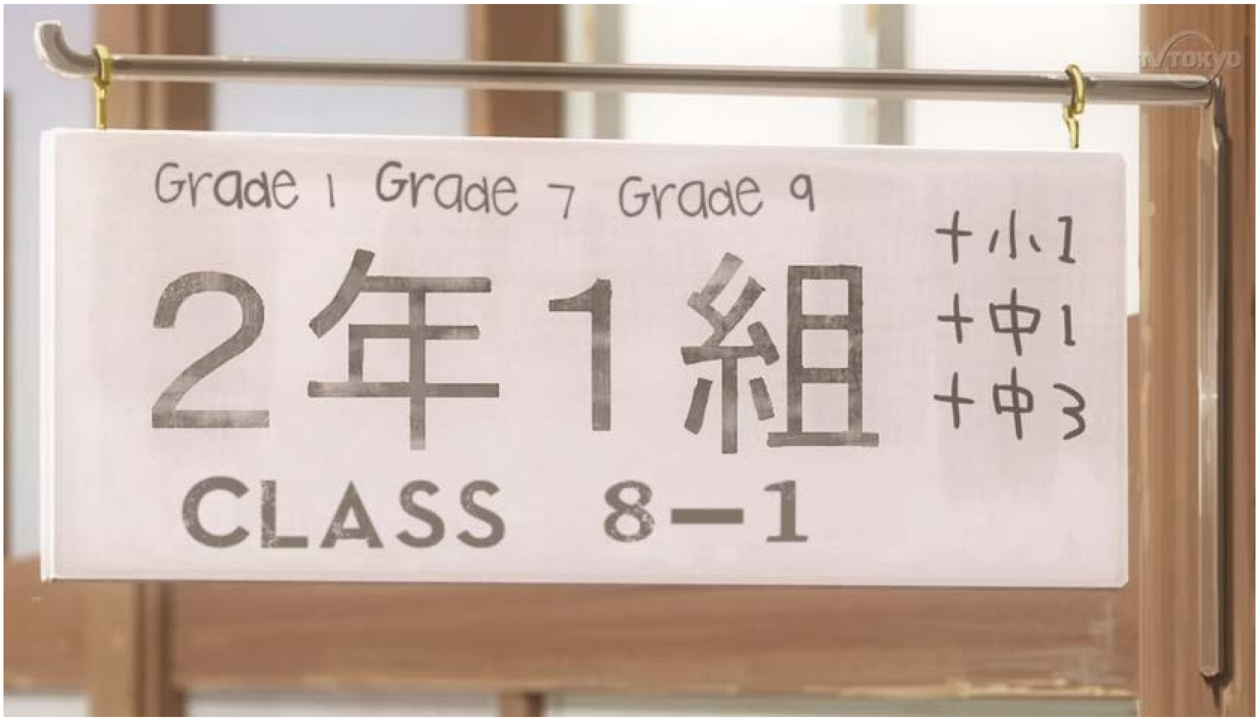


One hour of work. 39 lines. Just because I can. [Still not perfect, though.]
Original line in script was "Misakichou 3-3". I decided while I'm at it, I might as well do the rest.
All signs are in 2 layers, to get blur on both the outline and primary colour.
Split to make a line for each ~2 letters. Positioning, rotating, \fax-ing. Takes a while to get it right [and the top one is still wrong].
For the top one I also had to change colour for each letter [and outline] because the background darkens a lot from left to right.



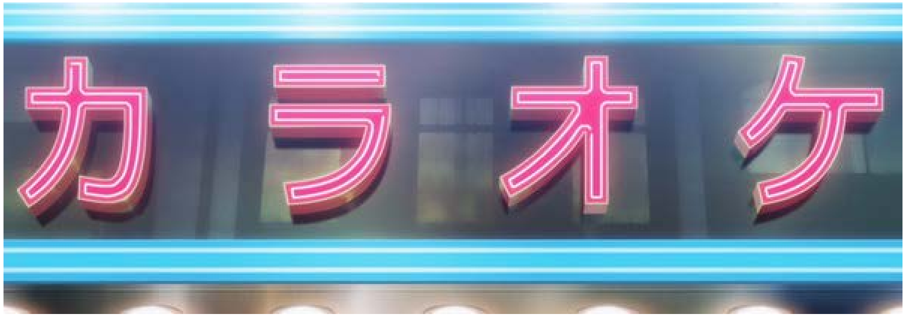
Nothing should stand out as obviously added.



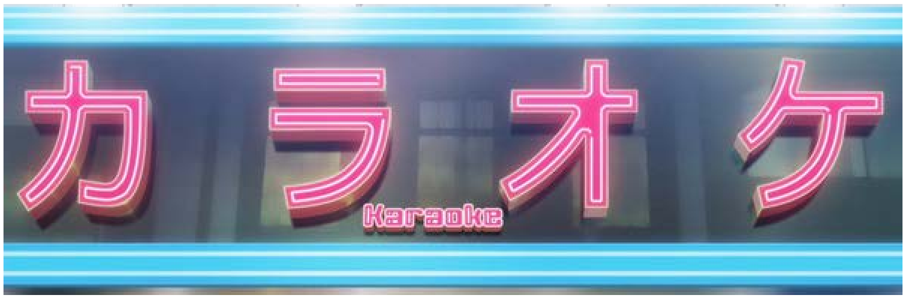


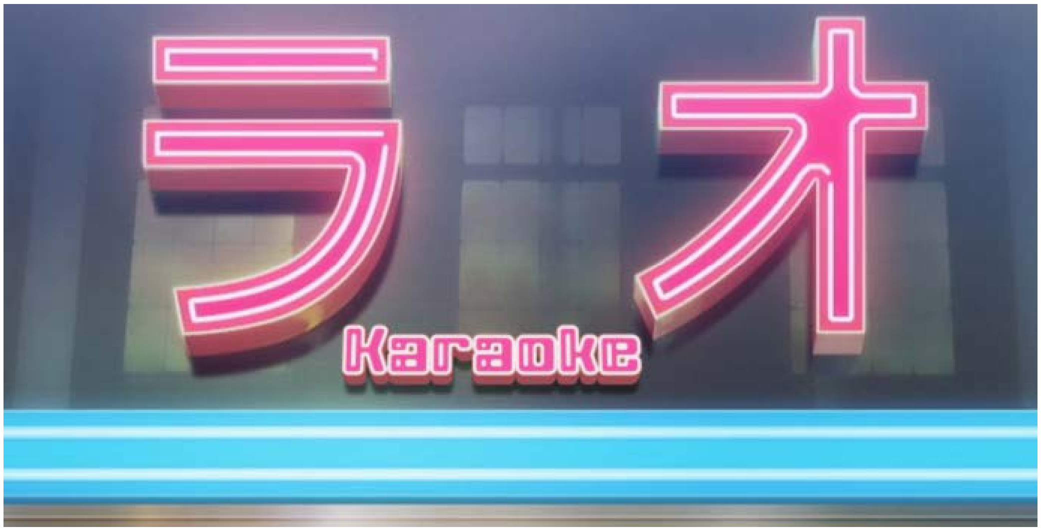
More examples of smooth blending. Good fonts, right blur, correct colours, good positioning and alignment.

Recently I had this "karaoke" sign to typeset, which seemed rather challenging.

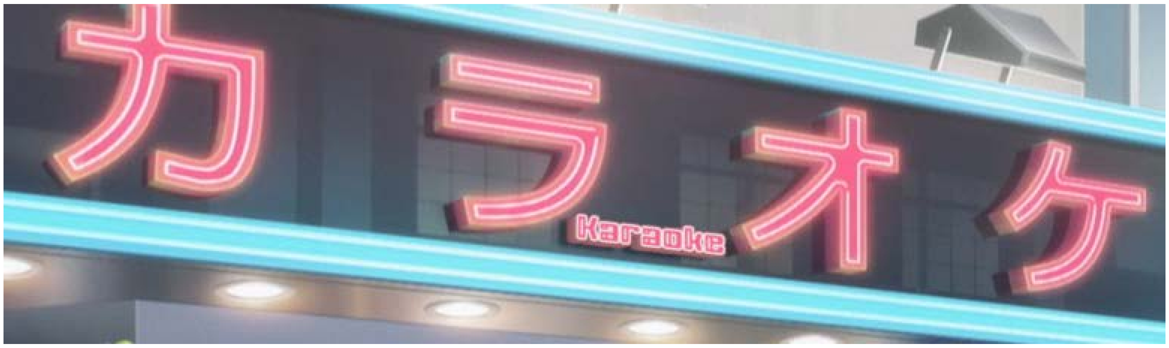


I knew I couldn't get it to be too similar, so I had to figure out something that just wouldn't stand out too much.
I ended up with this:

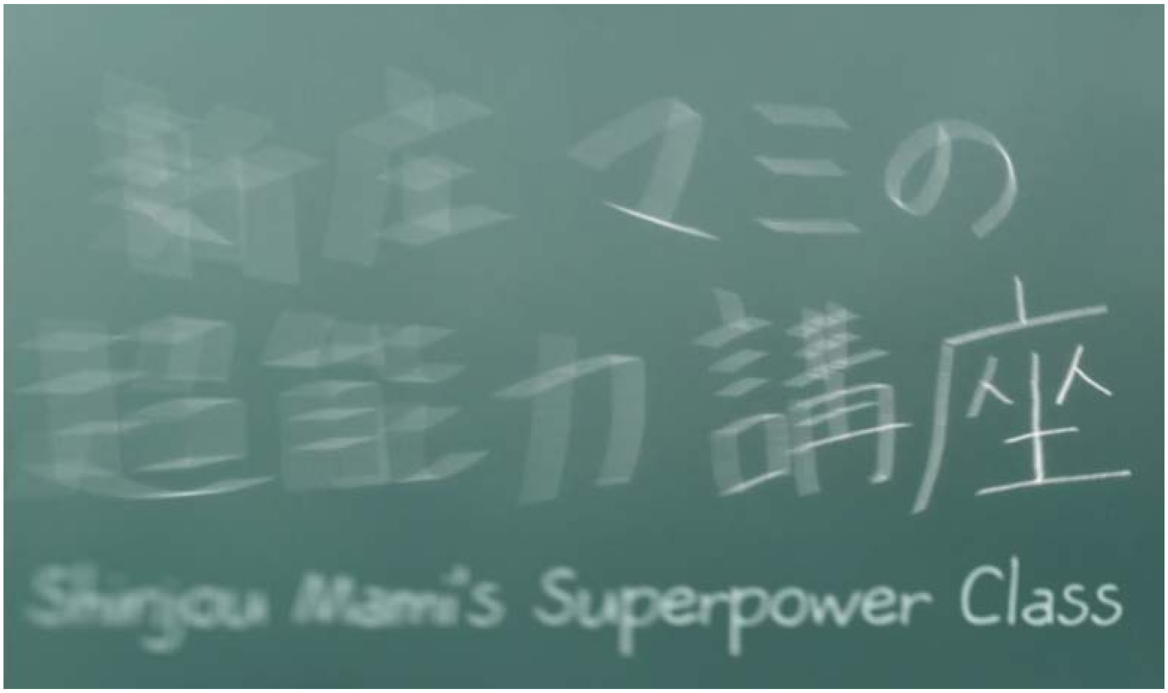




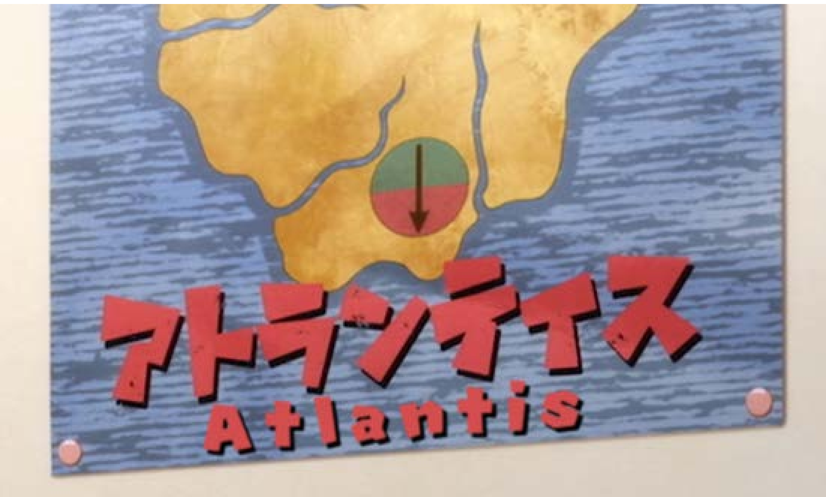
There was another one, from a different angle:



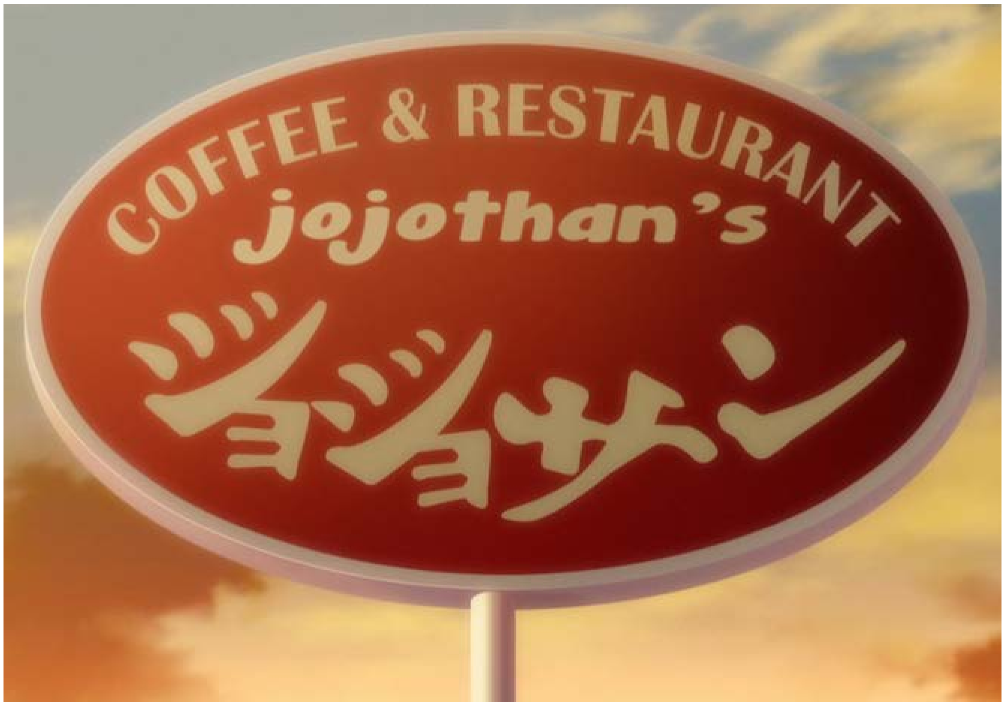
I think it looks pretty plausible, all things considered, especially given that it didn't really take that much time.



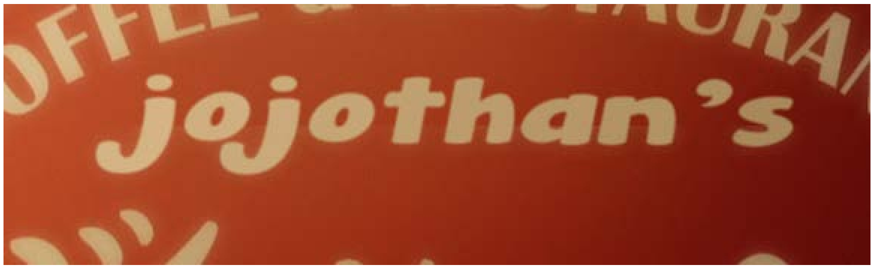
This was on the first few frames of a zoom out. What you have to do to get it like this is two things:
1. gradient the blur
2. make 2 or 3 lines with slightly different \fsp and align them on the right

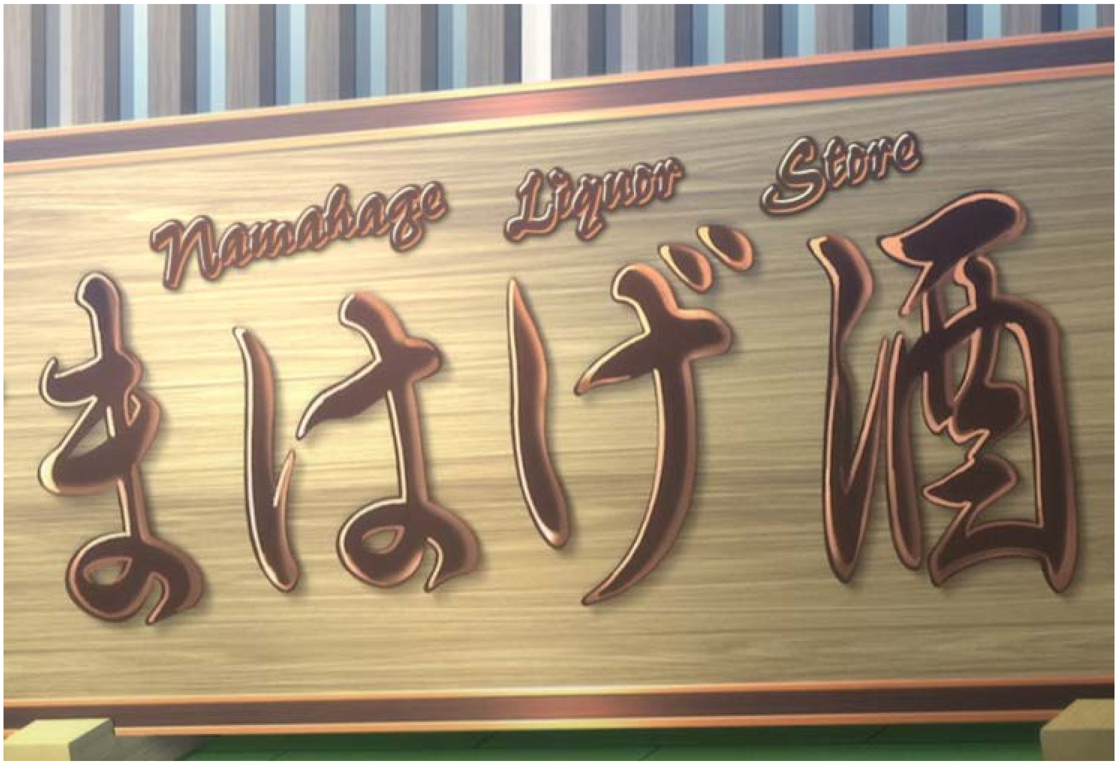


Here the right font means about 70% of the success.
What I often do for signs like this one is use quite a bit of \fsp.
When you don't have enough space to make the English as large as the Japanese, this helps.



Only the middle part is typeset. What helps in this one is the glow. This is \blur8 and \alpha&HB0&.
For comparison, here's a split image - top half without glow, bottom with glow:





I like signs like this one, where you get to do something interesting that looks nice, without it being difficult.
4 layers:
1 (top): brown centre with that (more or less) orange shadow
2. similar but shadow is light and on the other side (top left)
3. brown border
4. (bottom): separate layer for the shadow, as it was much more blur than the border

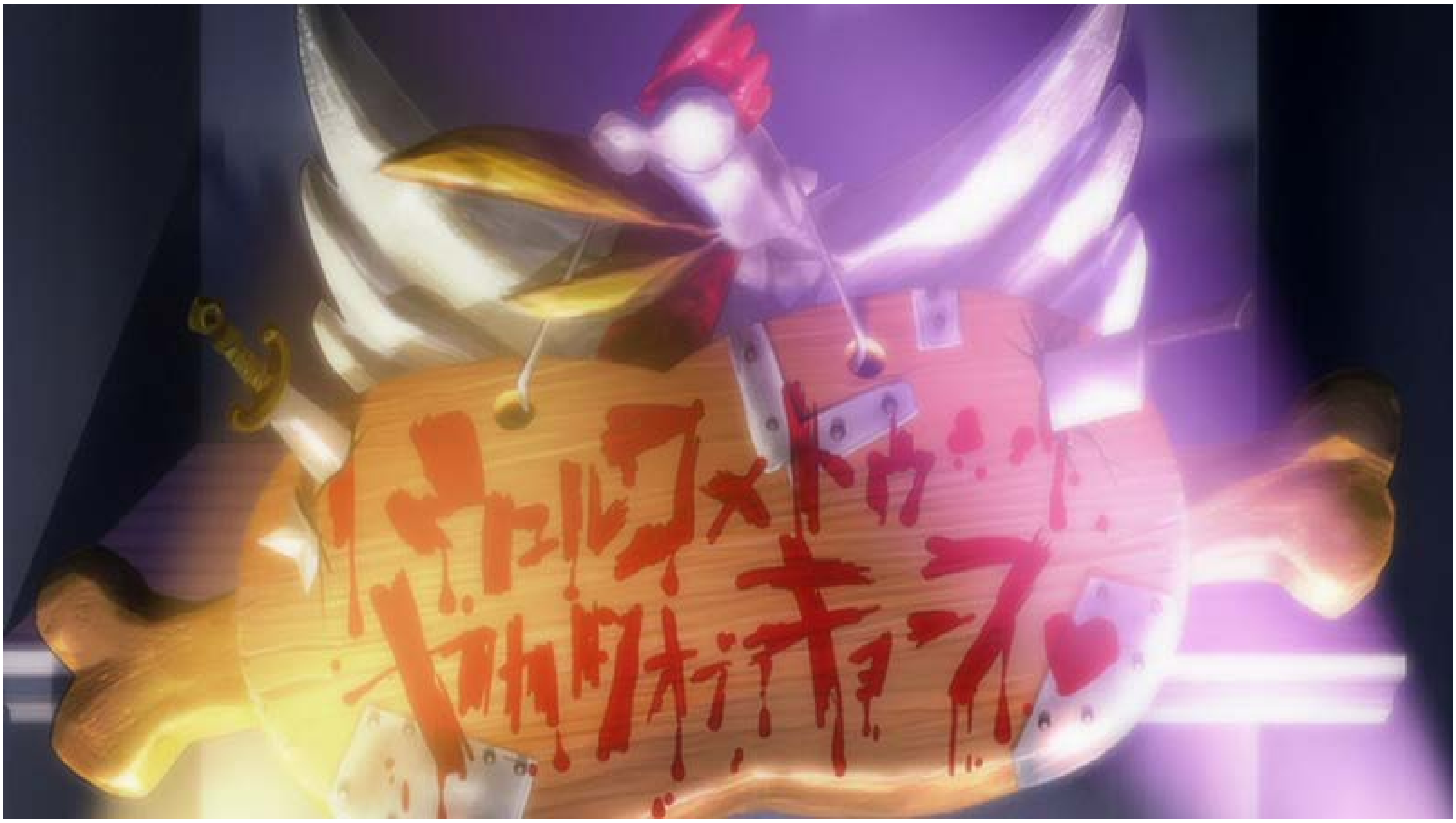
« Back to Typesetting Main

How to Use ASSDraw

ASSDraw can be pretty useful for masking, but also for actually drawing some fun things,
if you're a typesetter with imagination.

Most of the time, for masking, this will be enough: {\p1}m 0 0 l 100 0 100 100 0 100
You'll use the scaling tool to stretch it and mask anything with roughly rectangular shape.
If you add border, like \bord20, you can have round edges too.
If you scale it down to 1 pixel and add border, you'll even have a circle.
I'm sure you now understand how to make an ellipse, which you can then rotate etc.
So for most masks, you won't need ASSDraw. But what if you do...

I'll use a funny example from a recent episode. It's a bit unusual way of using ASSDraw, but it will do for this guide.



So I was supposed to typeset this sign.
I'm thinking "What the fuck am I supposed to do with this? There's no space for the typeset unless I make it so small that it will be unreadable. I can't mask it either unless I wanna pull some retarded hadena stunt that would forever ruin what little reputation Commie has left. It's even fading out just to make it more complicated. What now?"

After considering my options for a minute I decided that there's no point in trying to do the impossible.
It's either \an8 or I have to come up with some unusual idea. And I really don't like \an8. I don't think I've ever done that.
So I decided for ASSDraw.

Here's how you use it:

1. Open it. [Duh]
2. Drag the zoom slider [under 'Background' in menu] all the way to the left.
3. In Aegi [on a frame where you wanna use the mask] right click on the screen and select "Copy image to Clipboard".
4. In ASSDraw go to Canvas -> Paste [or Ctrl+V]. You can set the opacity to 100 - better in most cases.
5. At coordinates 0,0 there's a red dot [actually a cross if you zoom in]. Right-click and drag it around. Put it where you want the drag point of your drawing to be [the anchor in aegisub]
6. Last 2 icons in the menu are 'Pan drawing' and 'Pan background'. First one should always be on. The second one you'll turn on after step 5 [ie now].
7. Draw stuff.
8. Copypaste the final coordinates to aegisub.

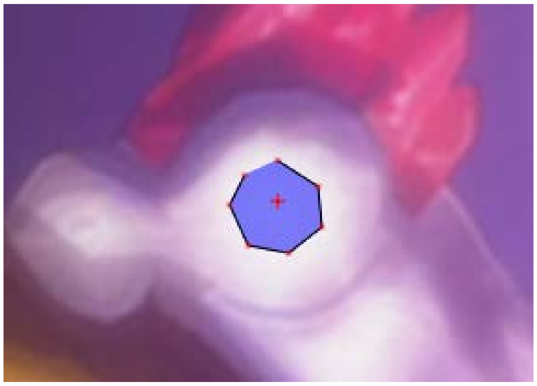
It's easy once you go through the process for the first time. The pain in the ASS for beginners is to figure out how to get the result in correct zoom and how to not have the anchor point 2000 pixels away from the actual drawing. That's what points 2, 5 and 6 are about.

Now for the actual **drawing**. You could probably figure it out by yourself.

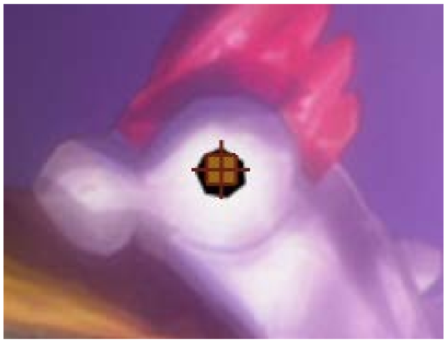
That red dot/cross is the starting point. You can either work from there, or move the point, or create a new one.
At the bottom of ASSDraw you can see "m 0 0"
That's the starting point. If you start from there, it will also be the anchor point in Aegi.
If you want the anchor point in the centre of your drawing, move the starting point or create a new one.
Select Drag from the tools to move points. Select Move to create a new one. If you make a new one, delete the "m 0 0" from the line below.

You have a starting point, so now you want lines. Obviously you'll choose Line from the tools and start drawing. Kinda like clipping in Aegi. Since you have 'Pan background' enabled, you can now zoom in [with the slider or mouse wheel] and move the screen with right click or arrows as much as you want. Just place dots with the Line tool till you have what you wanted to draw. Alternately there's the Bezier tool, just like with clipping in Aegi. Simple enough.

A little demonstration:

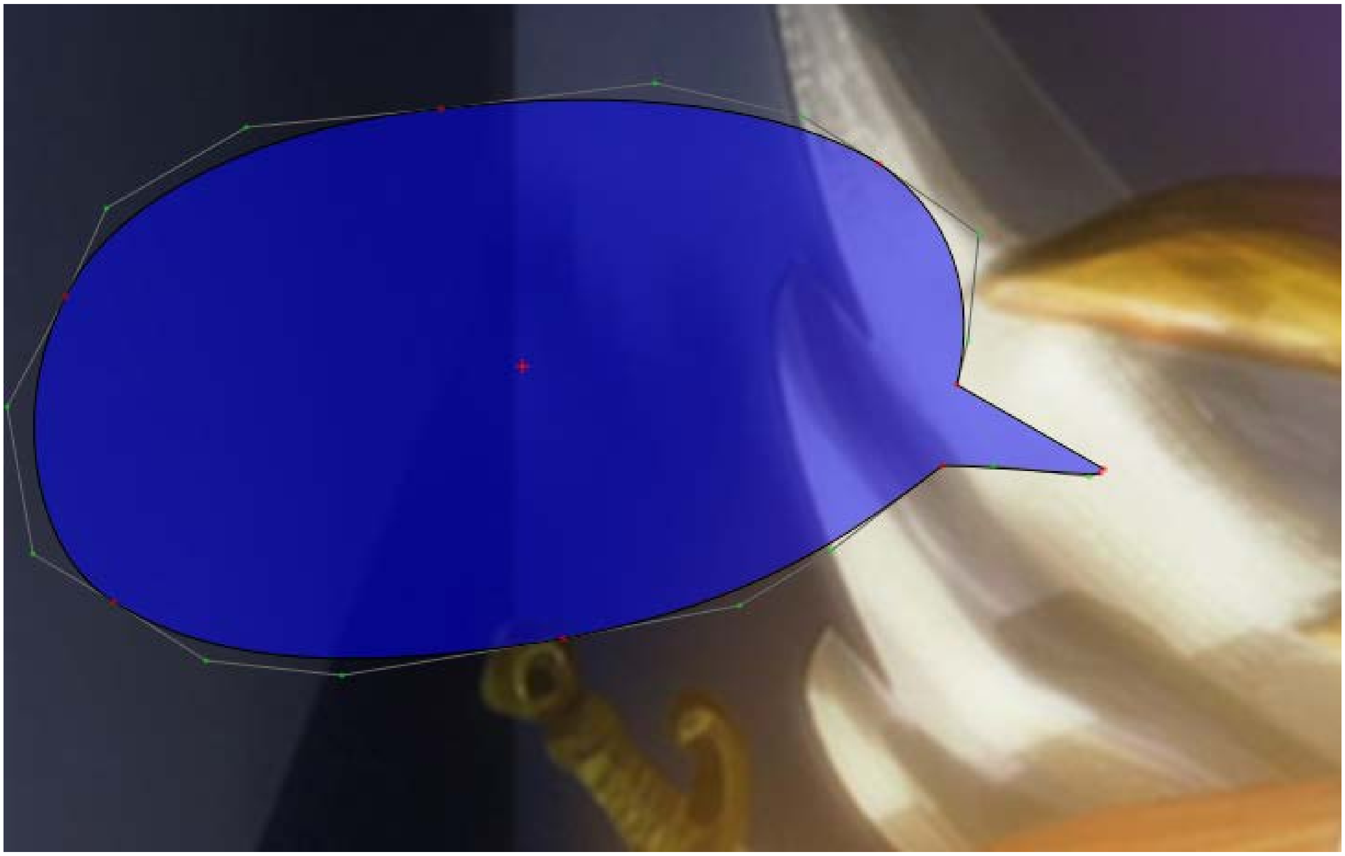


This is ASSDraw. The red cross is the centre. The line is "m 0 -11 l 11 -4 l 12 7 l 3 14 l -8 12 l -13 1 l -9 -7"
Copypaste it to Aegi with these tags: {p1\an7}m 0 -11 l 11 -4 l 12 7 l 3 14 l -8 12 l -13 1 l -9 -7
Then position it where you want it to be.

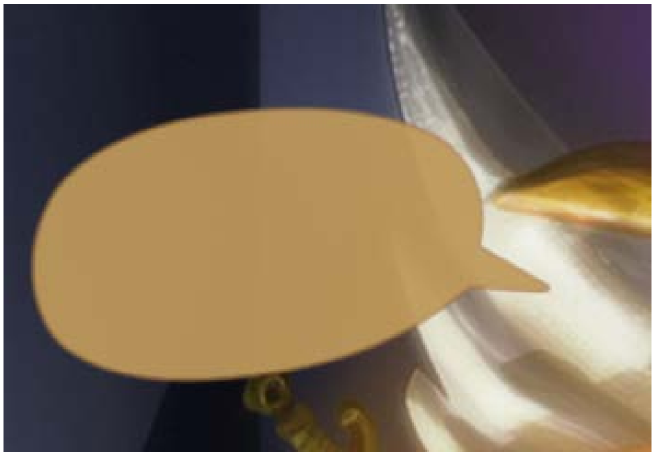


You'll end up with this.
The point of aligning the red cross and using \an7 is that you have the anchor point where you want it to be.
If you don't do that, things will still work, but your anchor point for this little drawing can end up on any place on the screen or even outside of it, which is a bit inconvenient.

So you draw what you actually want...

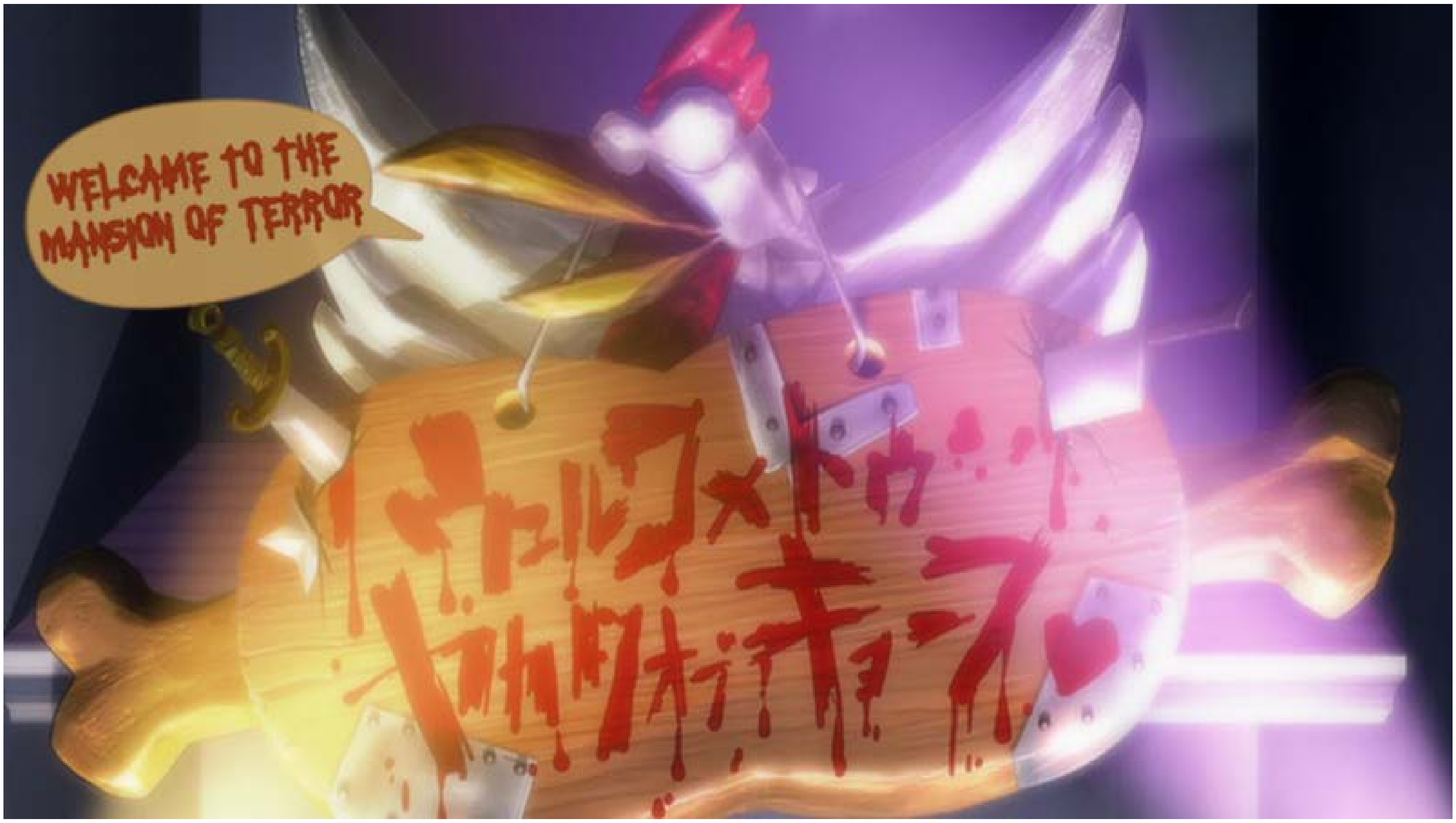


2 straight lines and the rest is Bezier.
You can paste it in Aegi and see how it looks, then fix it some more and just paste the new line over again.



This is the result, in 2 layers, one with border, set colours and some transparency.

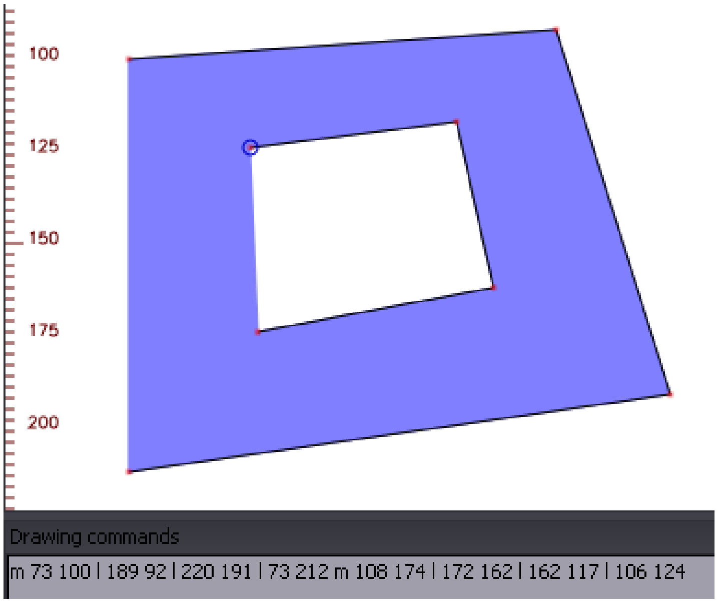
Now you just add the styled text, and you're done.



So that's how you typeset a sign like that.
It may still be a bit ridiculous, but if you can typeset this and make it look better than this, please show.
Also, the "welcame" is intentional, so you can stop laughing.

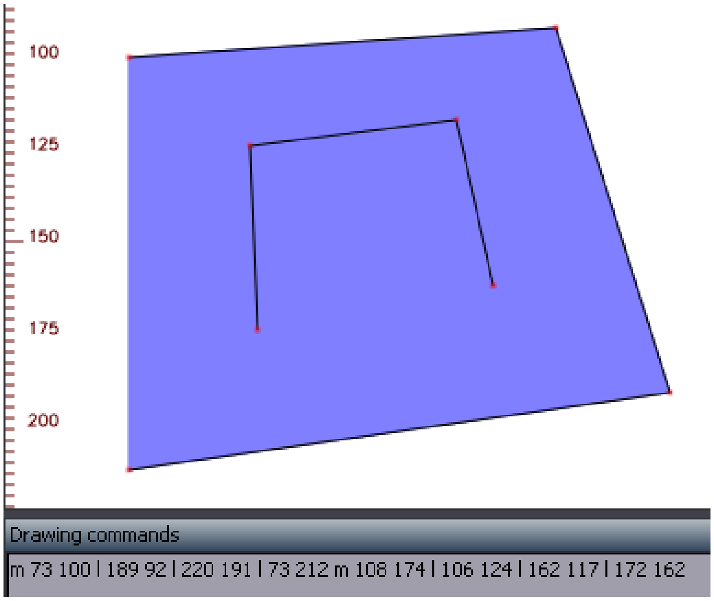
How to make hollow shapes?

One thing that may be hard to figure out is how to create an empty space inside a drawing.



```
m 73 100 l 189 92 l 220 191 l 73 212 m 108 174 l 172 162 l 162 117 l 106 124
```

Each of the two shapes starts with m.
The trick is that they have to be drawn in opposite directions.
If you draw them in the same direction, you'll get this:

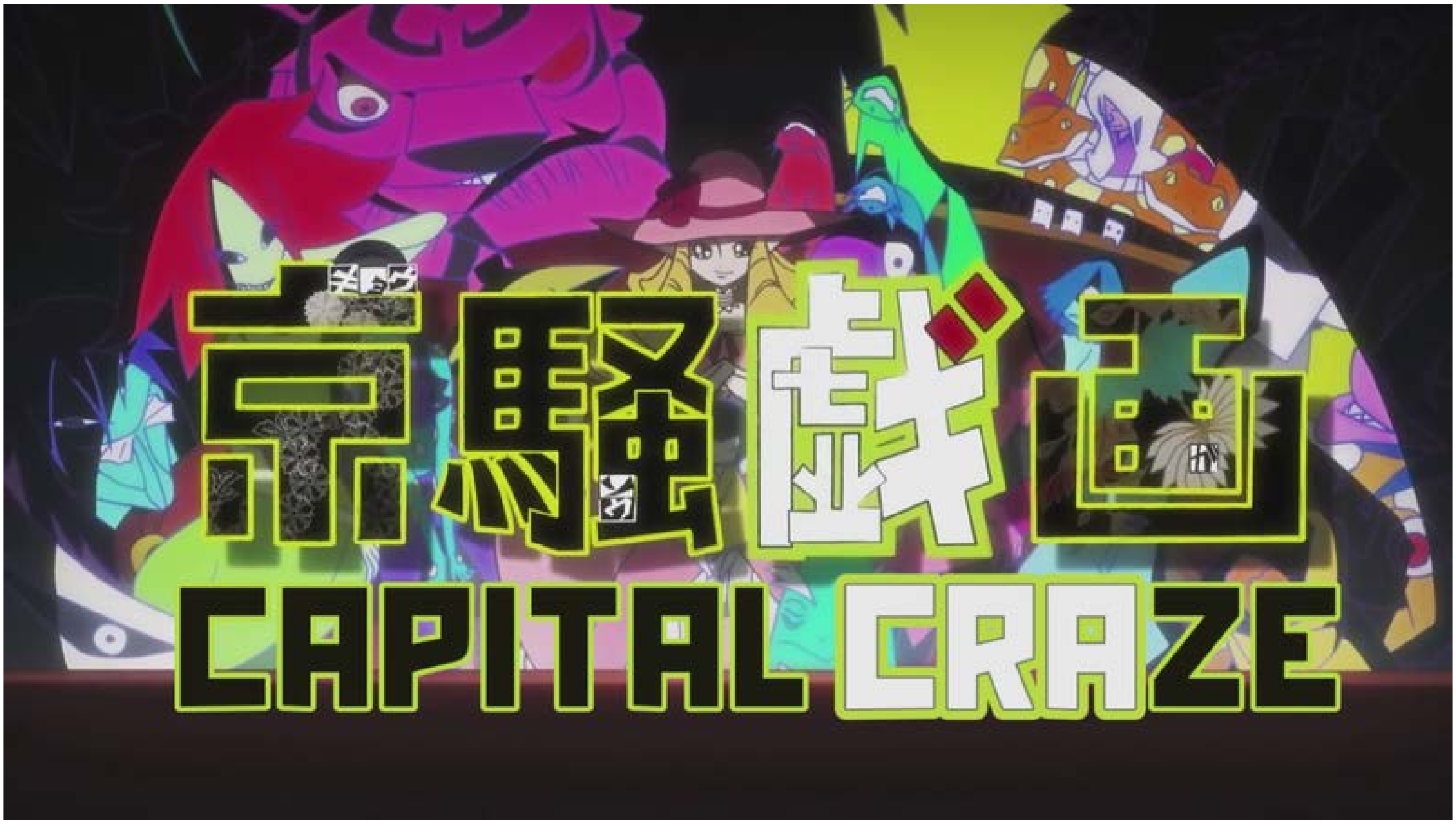


Drawing commands

```
m 73 100 l 189 92 l 220 191 l 73 212 m 108 174 l 106 124 l 162 117 l 172 162
```

For illustration of what can be done with this, check **this thing** made by KKRais.

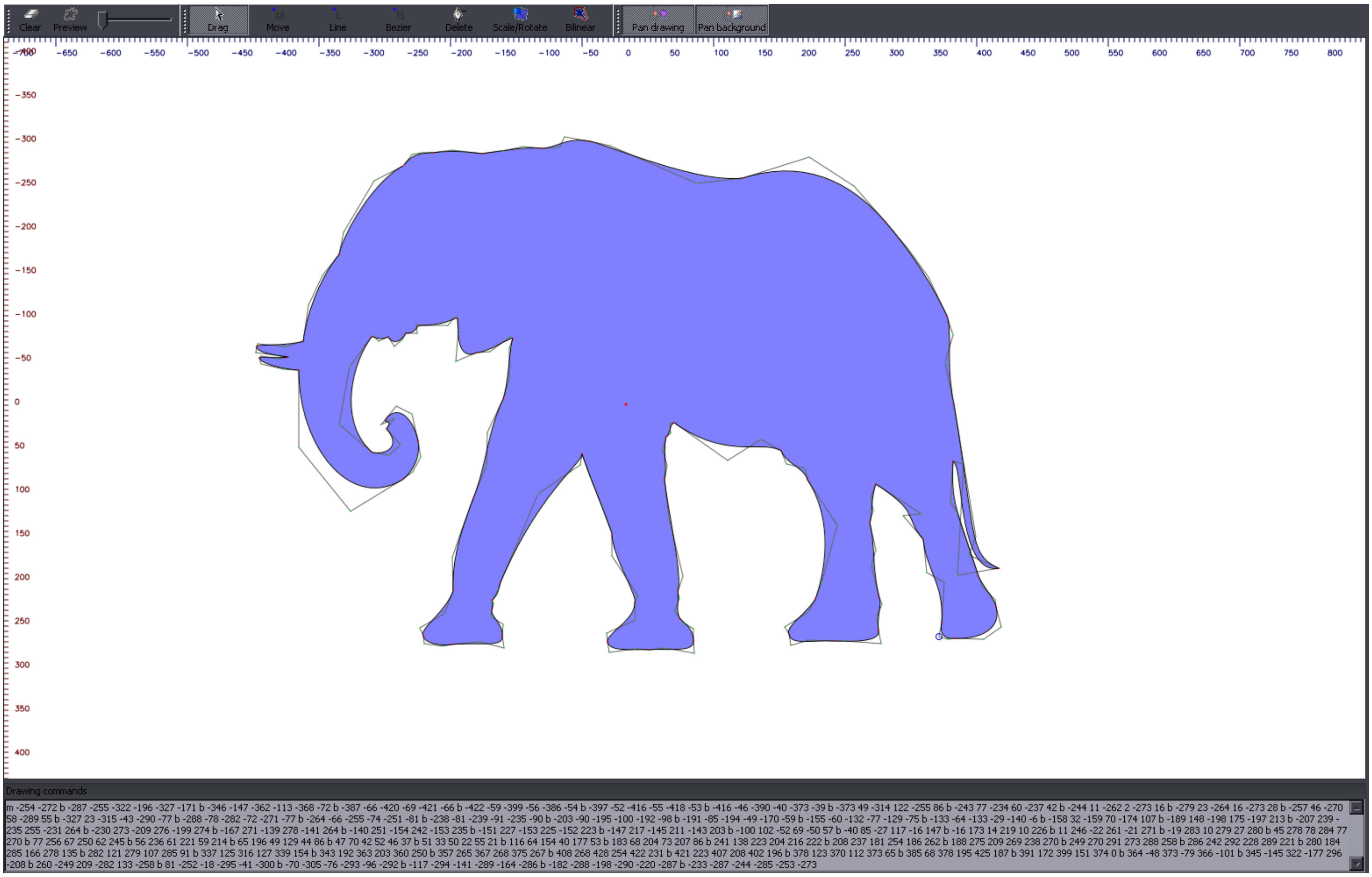
And another example of what I did recently:



I couldn't find a font that I'd be satisfied with and wanted to try some drawing anyway, so I drew the whole title.

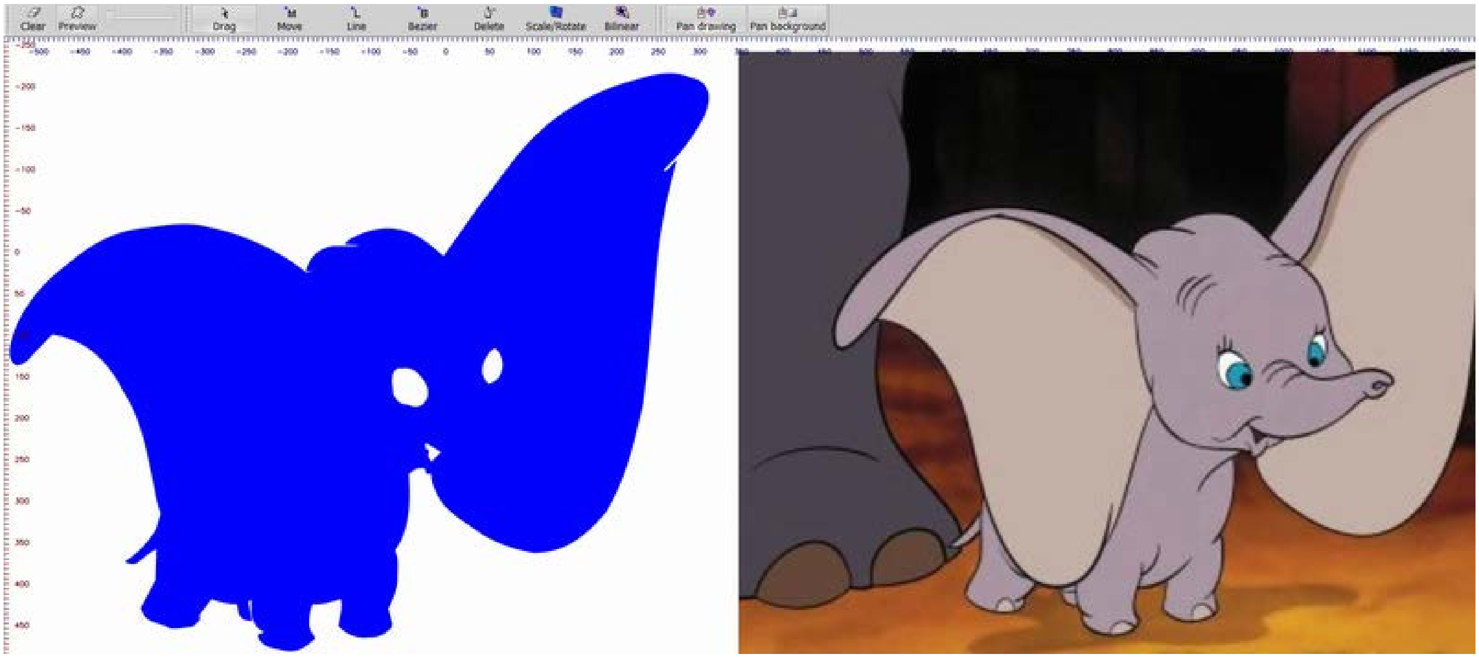
Sometimes people give me ideas...

< Haidaraaaaa > there needs to be an optional assdraw test.
< reanimated > if optional, you can just make one for yourself
< reanimated > like "draw an elephant with ASSDraw"
< reanimated > there you go
< reanimated > actually lemme try that



```
m -254 -272 b -287 -255 -322 -196 -327 -171 b -346 -147 -362 -113 -368 -72 b -387 -66 -420 -69 -421 -66 b -422 -59 -399 -56 -386 -54 b -397 -52 -416 -55 -418 -53 b -416 -46 -390 -40 -373 -39 b -373 49 -314 122 -255 86 b -243 77 -234 60 -237 42 b -244 11 -262 2 -273 16 b -279 23 -264 16 -273 28 b -257 46 -270
58 -289 55 b -327 23 -315 -43 -290 -77 b -288 -78 -282 -72 -271 -77 b -264 -66 -255 -74 -251 -81 b -238 -81 -239 -91 -235 -90 b -203 -90 -195 -100 -192 -98 b -191 -85 -194 -49 -170 -59 b -155 -60 -132 -77 -129 -75 b -133 -64 -133 -29 -140 -6 b -158 32 -159 70 -174 107 b -189 148 -198 175 -197 213 b -207 239 -
235 255 -231 264 b -230 273 -209 276 -199 274 b -167 271 -139 278 -141 264 b -140 251 -154 242 -153 235 b -151 227 -153 225 -152 223 b -147 217 -145 211 -143 203 b -100 102 -52 69 -50 57 b -40 85 -27 117 -16 147 b -16 173 14 219 10 226 b 11 246 -22 261 -21 271 b -19 283 10 279 27 280 b 45 278 78 284 77
270 b 77 256 67 250 62 245 b 56 236 61 221 59 214 b 65 196 49 129 44 86 b 47 70 42 52 46 37 b 51 33 50 22 55 21 b 116 64 154 40 177 53 b 183 68 204 73 207 86 b 241 138 223 204 216 222 b 208 237 181 254 186 262 b 188 275 209 269 238 270 b 249 270 291 273 288 258 b 286 242 292 228 289 221 b 280 184
285 166 278 135 b 282 121 279 107 285 91 b 337 125 316 127 339 154 b 343 192 363 203 360 250 b 357 265 367 268 375 267 b 408 268 428 254 422 231 b 421 223 407 208 402 196 b 378 123 370 112 373 65 b 385 68 378 195 425 187 b 391 172 399 151 374 0 b 364 -48 373 -79 366 -101 b 345 -145 322 -177 296
-208 b 260 -249 209 -282 133 -258 b 81 -252 -18 -295 -41 -300 b -70 -305 -76 -293 -96 -292 b -117 -294 -141 -289 -164 -286 b -182 -288 -198 -290 -220 -287 b -233 -287 -244 -285 -253 -273
```

So I did.

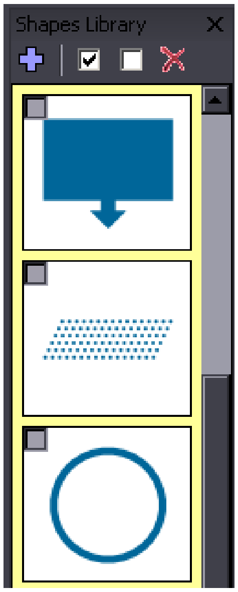


In the meantime Haidaraaaaa tried a cute elephant but got something out of a nightmare instead...

You can use ASSDraw to make additions to your text, like I did with the M in Muromi here:

ASSDraw has no saving option and some bug when using Ctrl+Z that can mess things up pretty bad, so make sure to back up your drawing coordinates every now and then.

2015 Note: ASSDraw does have a saving option.



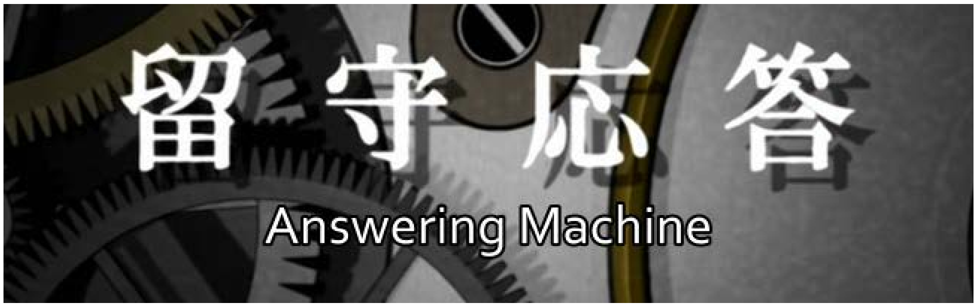
For most simple drawings, you can just use the clip tool in Aegisub and convert the clip to drawing with a script.

« Back to Typesetting Main

Typesetting - Examples

Here I'll be posting some examples of typesets that I've found in various releases.
Some of them not too bad but could easily be better, some of them pretty bad for no good reason.

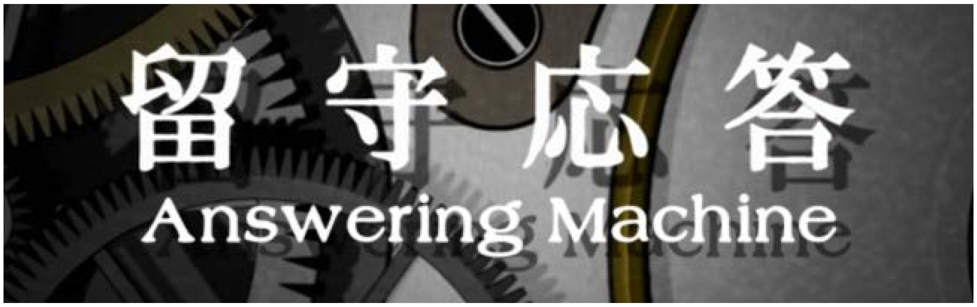
I'll be updating this page as I find new interesting things.



This is dickpants typesetting 101.
JP sign: large thick "serif" (not sure how to define that in kanji but should be close enough) font, no border, has shadow.
"Typeset:" small thin sans serif font, no shadow, fucking black border...
Question: Why?

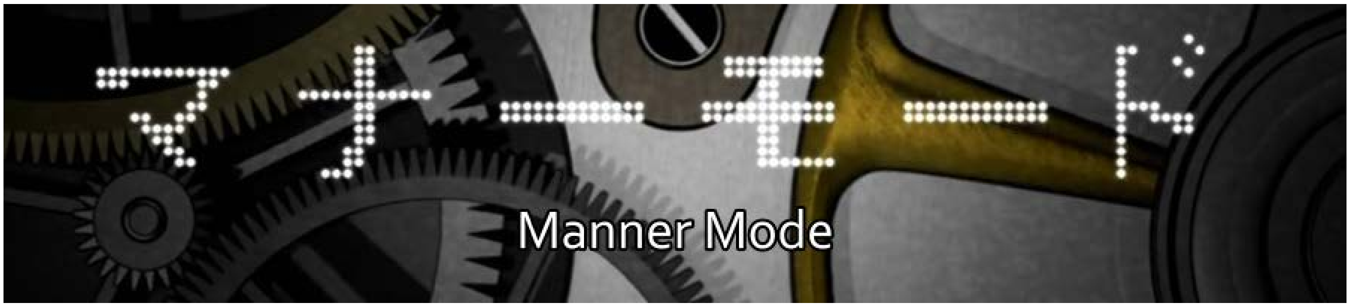
I can't believe this was actually released. No matter how little you know about typesetting,
you should be albe to handle borders and shadows. Font size is pretty easy as well.

Takes 1 minute to fix, out of which 40 seconds will be looking for a good font...

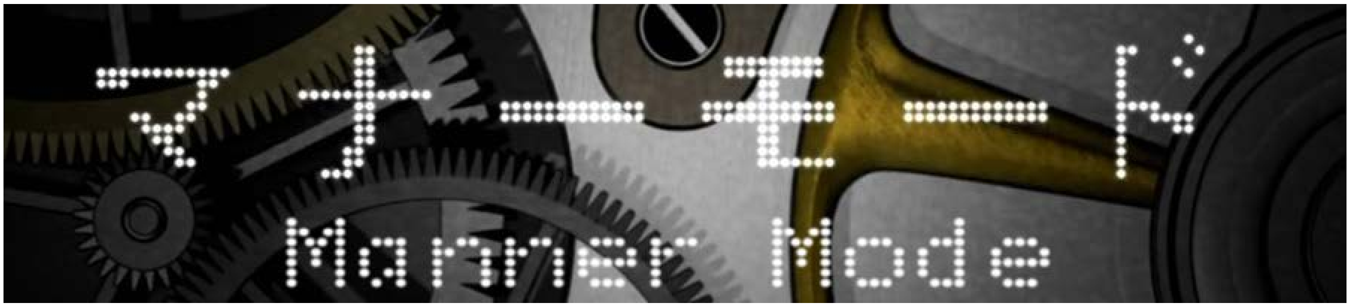


How hard can this be? ^

Same here...

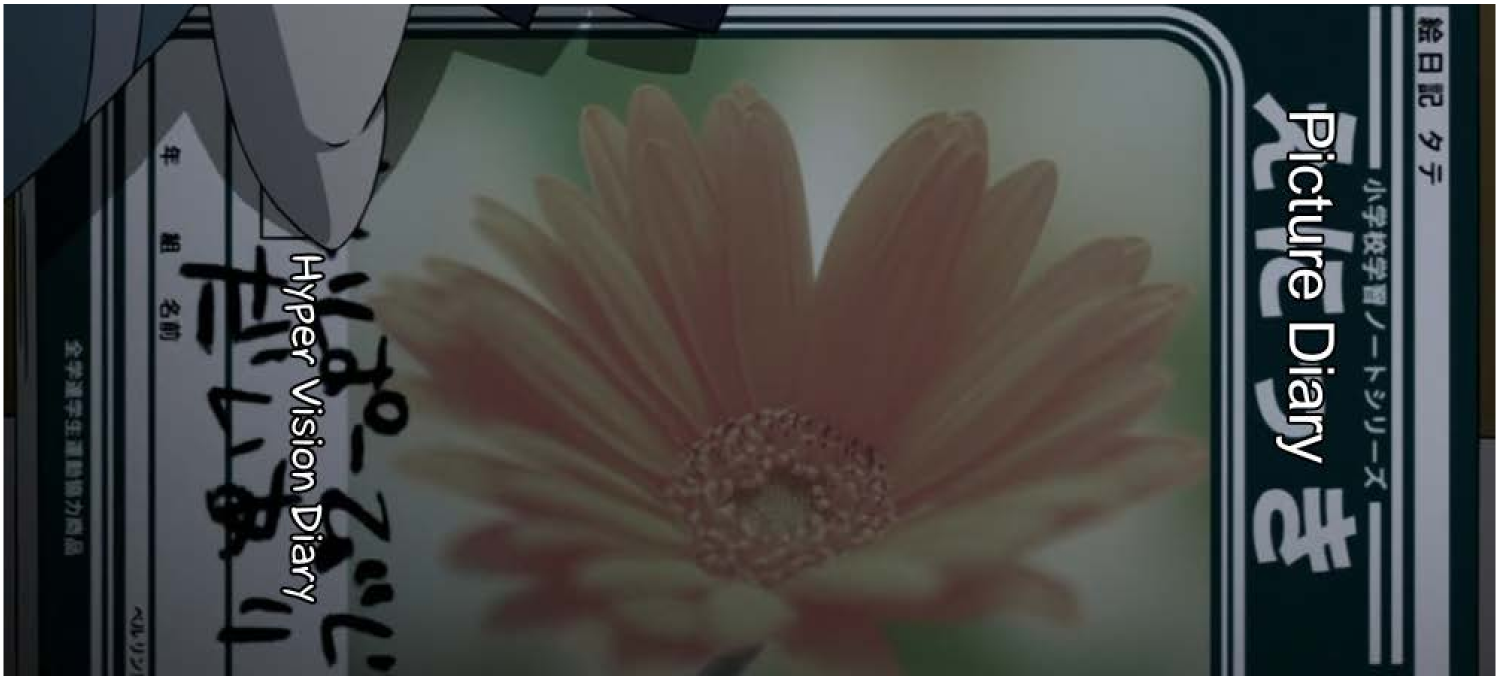


Might be good to use a font with dots, nuke that retarded border, add a little blur...



There you go. Viewers might as well think the studio used both Japanese and english title.

Even if you don't have a font with dots, it would still be common sense to at least type \bord0.

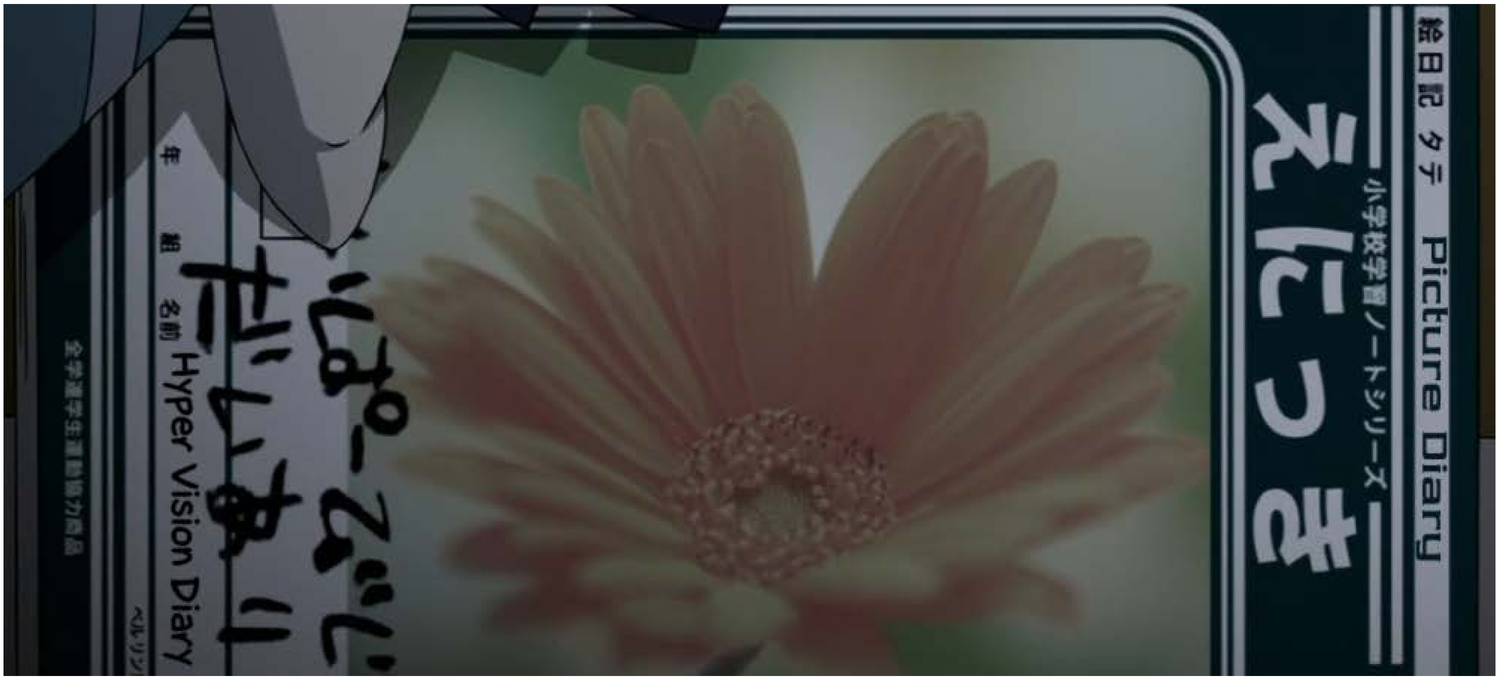


I don't even...

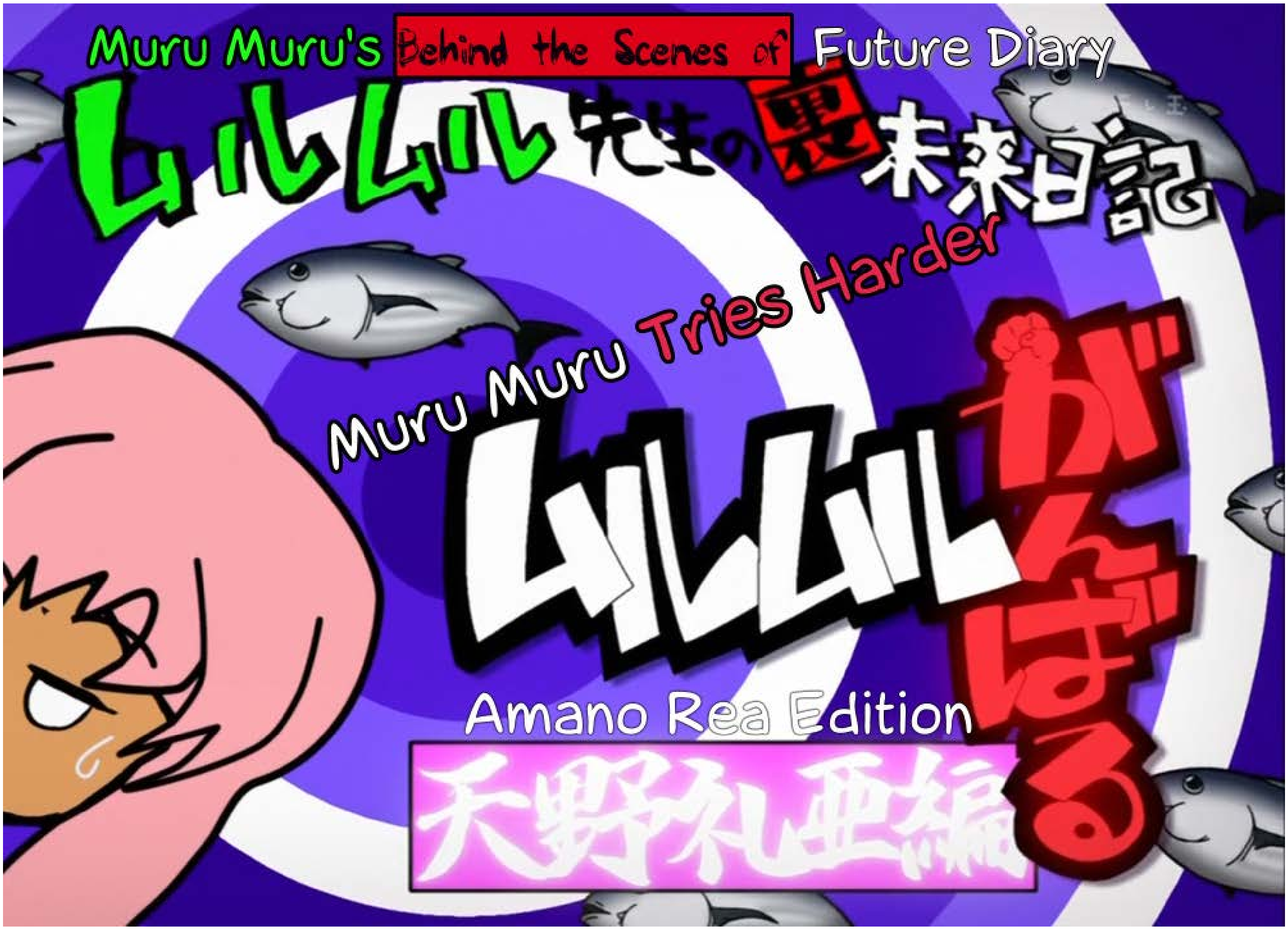
90 degrees rotation successful, yes, but that's about it. Well, I'll give credit for the font on the left. That works (but not like this).
Other than that, it's "Herp derp. Let's put some text over here... bam! Finished."

Left sign: Text is black, so keep it that way, and place it somewhere where it doesn't overlap with other things.

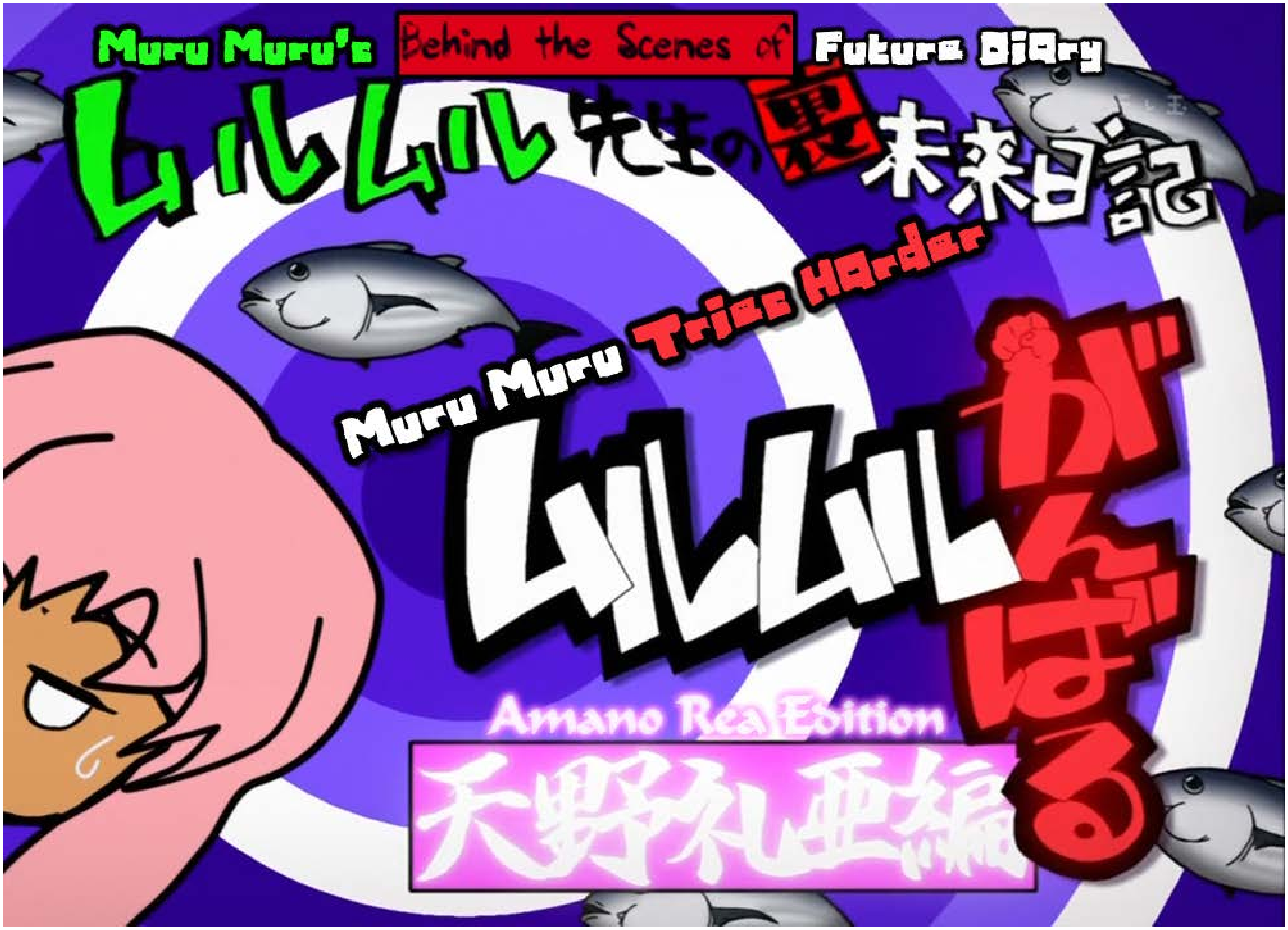
Right sign: If you have to choose between placing the TS a bit further from the sign with smaller font, and full retard mode, please don't choose full retard mode.



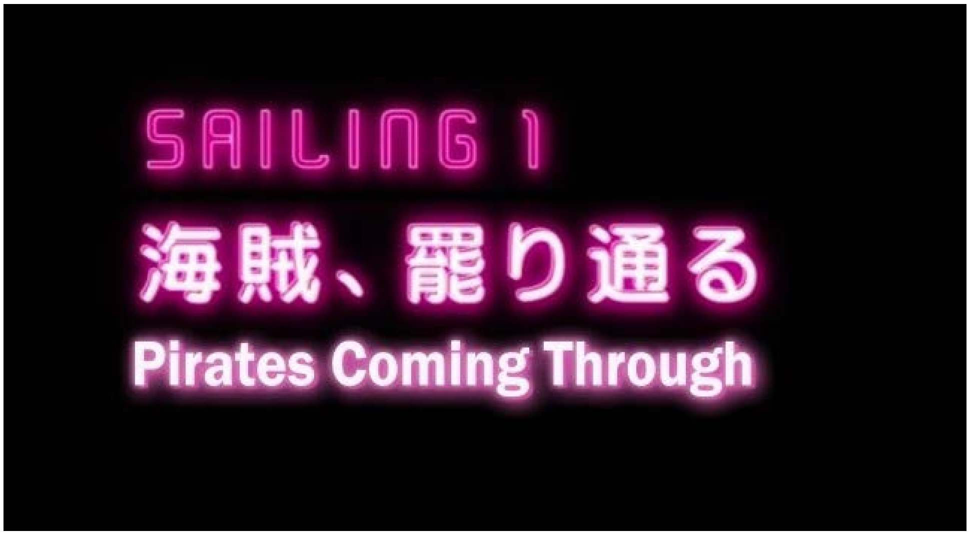
This is still all very easy.
Nuke (something that looks totally like) Arial, nuke border where it shouldn't be, don't use colours that aren't even on the screen, add a bit of blur.
Oh and don't place typesets "over" the signs, duh. This "let's put it as close to the original sign as possible" strategy is pretty dumb.



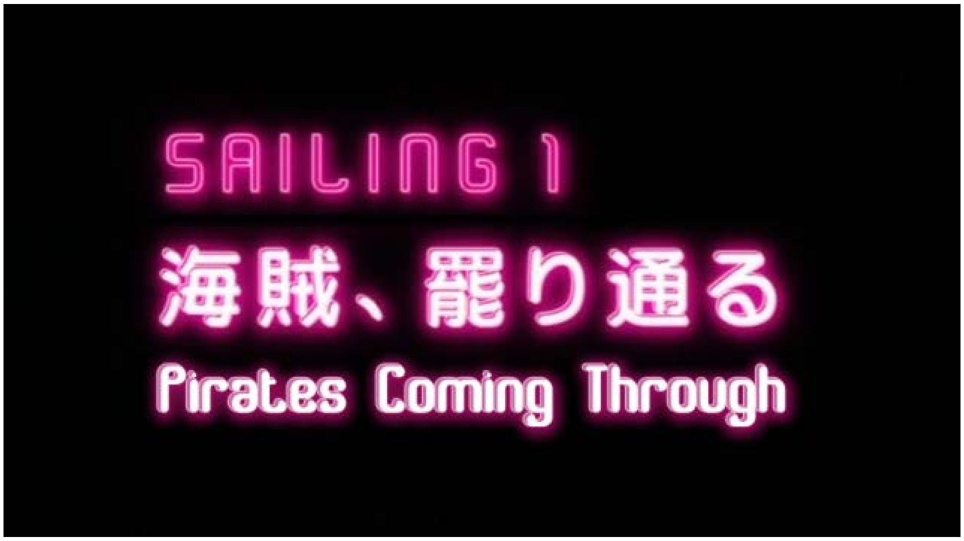
Muru Muru tries harder and so does the typesetter. Too bad he still fails.
The colours kind of match, mostly, so that's an improvement. Font is not Arial so that's another. Aside from the one in the red box though, the font still sucks.
Using a very round font for squarish signs is not the brightest choice. The next obvious thing is the border. The JP signs have a very thick one, so why use thin?
Probably the most awesome thing though... watch how the "Future Diary" casts a sharp bright shadow over dark things. Amazing, huh?
So, dear typesetting students, if you want a shadow that isn't too dark, you don't do it by choosing grey colour.
You do it by increasing transparency. (\4a&H**90**& or whatever value works) And use the damn \blur tag.



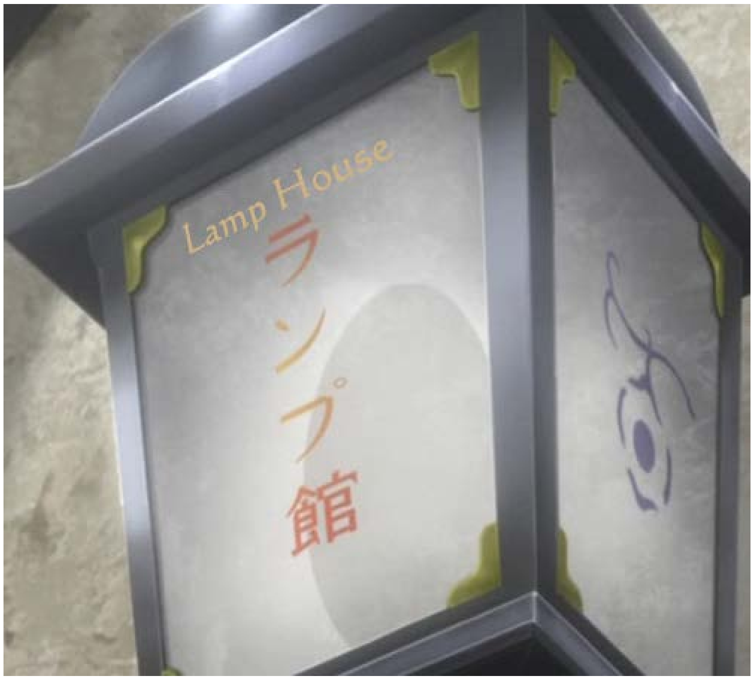
So, pick better fonts, fix the borders and shadows, and it looks a lot less fake.
This is still far from great, but it's far better than the one above and took only a few minutes to change.
The bottom sign could be much better, but since there are already 3 signs on the screen, I didn't want to add too many layers with blur, since it might lag.



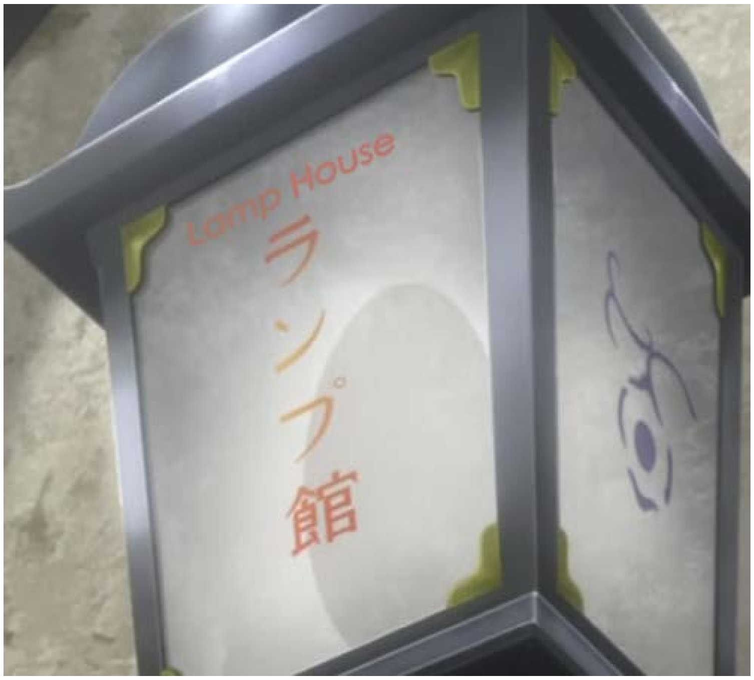
Not bad, generally fits in, but...



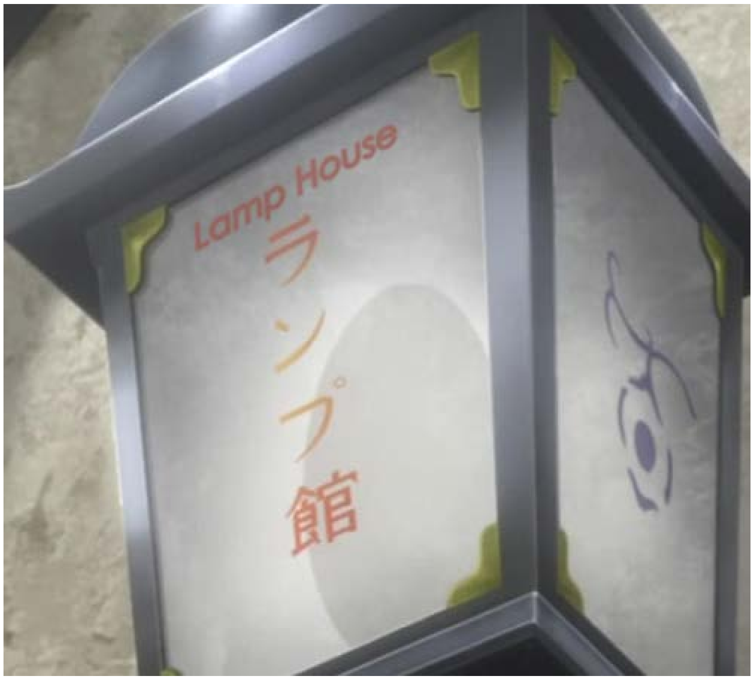
...font and one extra layer, and you're much closer.
The middle layer has dark inner colour and thin bright border and is positioned a bit up and left.
Bottom layer is just one colour and \blur5.



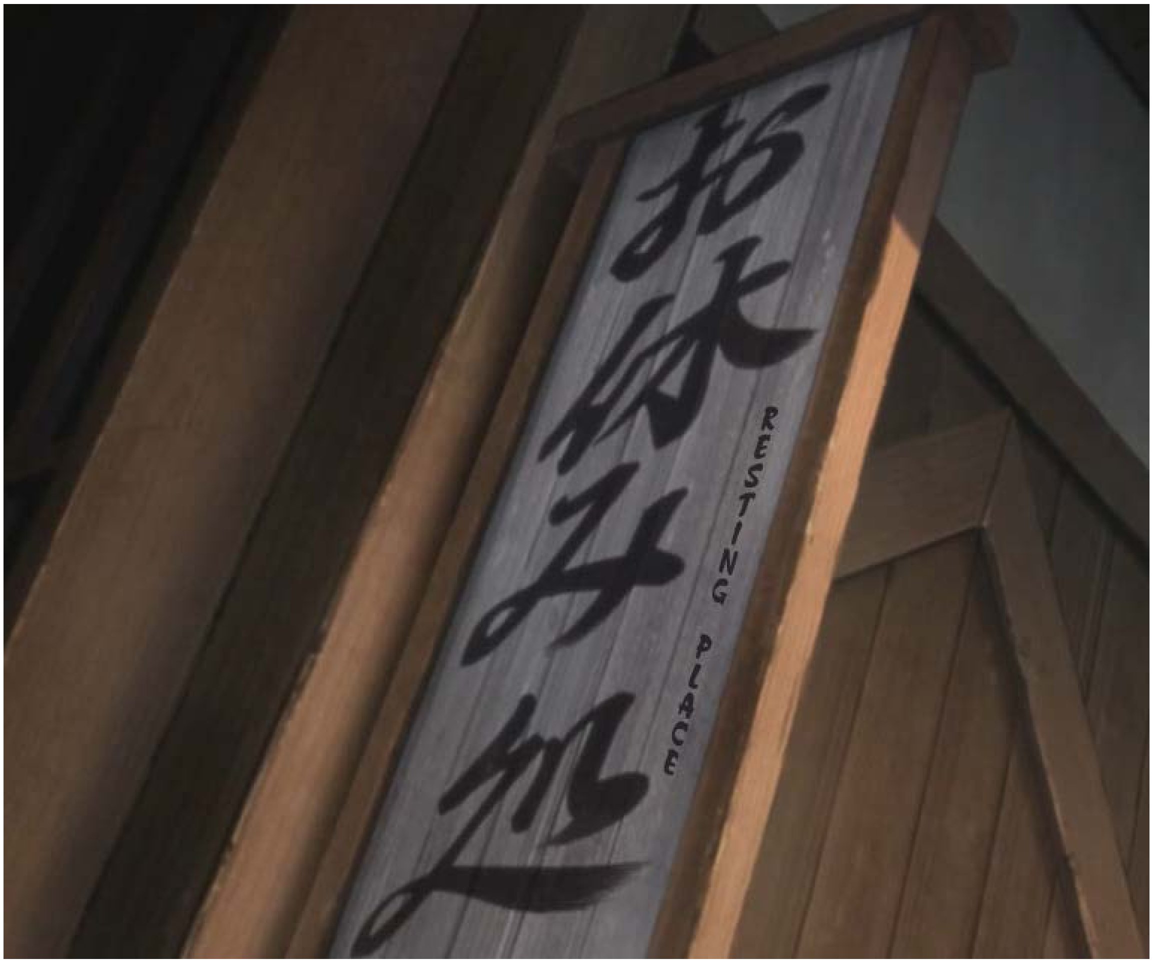
Here you obviously have a problem with alignment, and colour doesn't match.



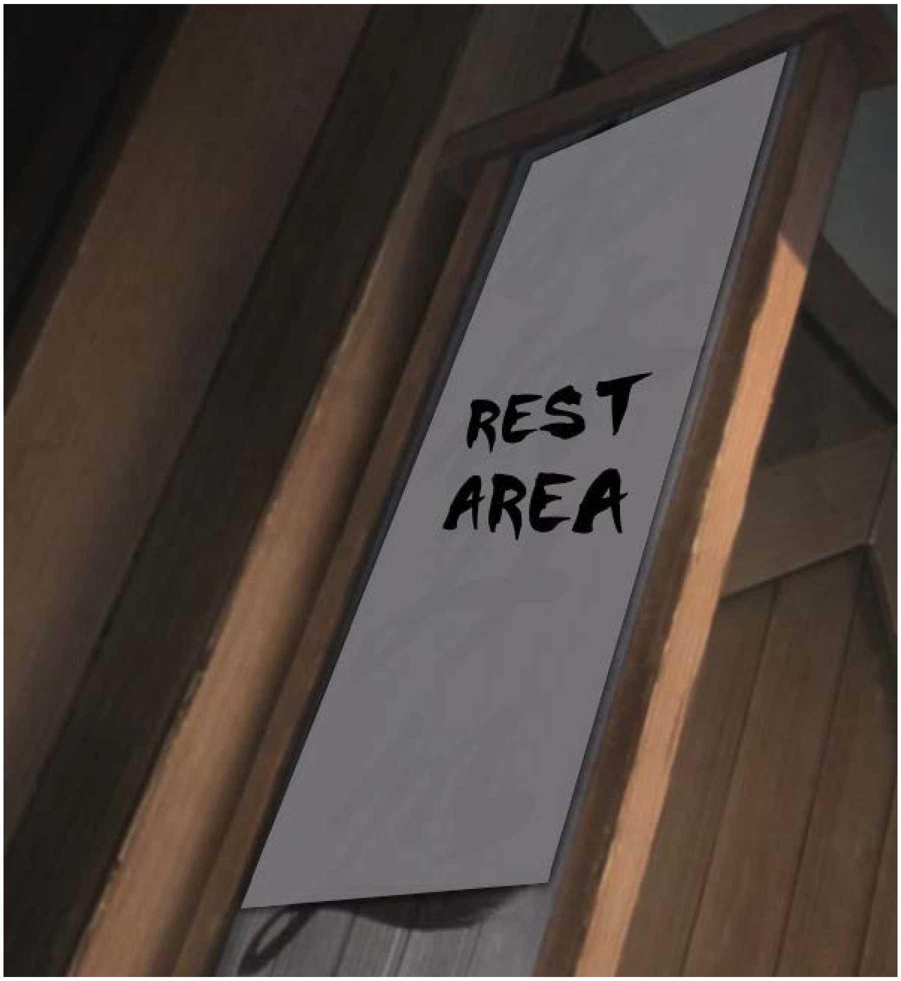
This one's much better on both counts, but still has issues.
The alignment is still a bit off, but few people would notice.
Main problem is readability, because of the colour, especially for 'Lamp'.
Let's explain one thing here... I'm sure the colour was matched with eyedropper,
but that doesn't always work. The reason is difference in background brightness.
The typeset is on darker background than most of the JP sign.
The darker the background, the darker the font has to be.
You can check with eyedropper that the JP signs follow that pattern.
Otherwise visibility goes down when the brightness is similar for both,
even if it's a different colour.



So if I fix the alignment and adjust the colours, it gets another notch better.
As it often is in such cases, the colour changes a few times throughout the line,
because so does the background.



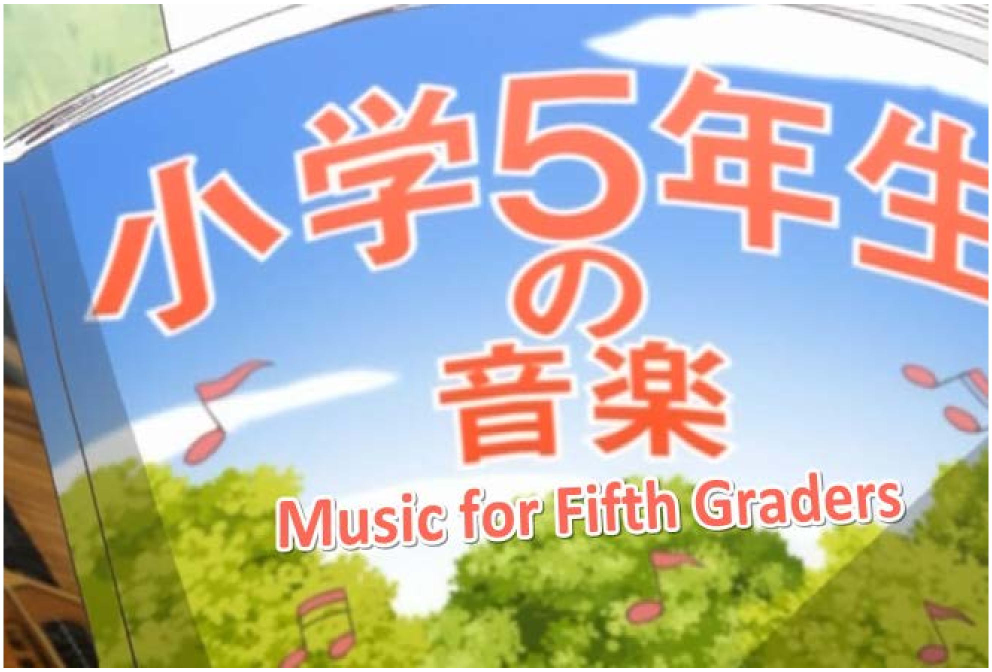
Here, have some Doki quality.
Apart from the crappy translation [no, I can't read moonrunes, but I'm told by translators this is like google translate], this is typical Doki typesetting. They kinda know how to do it, but not really. They can use rotations, but do it wrong.
See chapter on aligning to learn how to do it right.



Doki can always be beaten by experts from Hadena. This group is a true legend.
No one else can fuck up translating, editing, timing, typesetting and encoding [let's not even mention QC here] with such magnificence.
All of it apparently despite people from other groups trying to help them.
So what in the bloody hell is this?

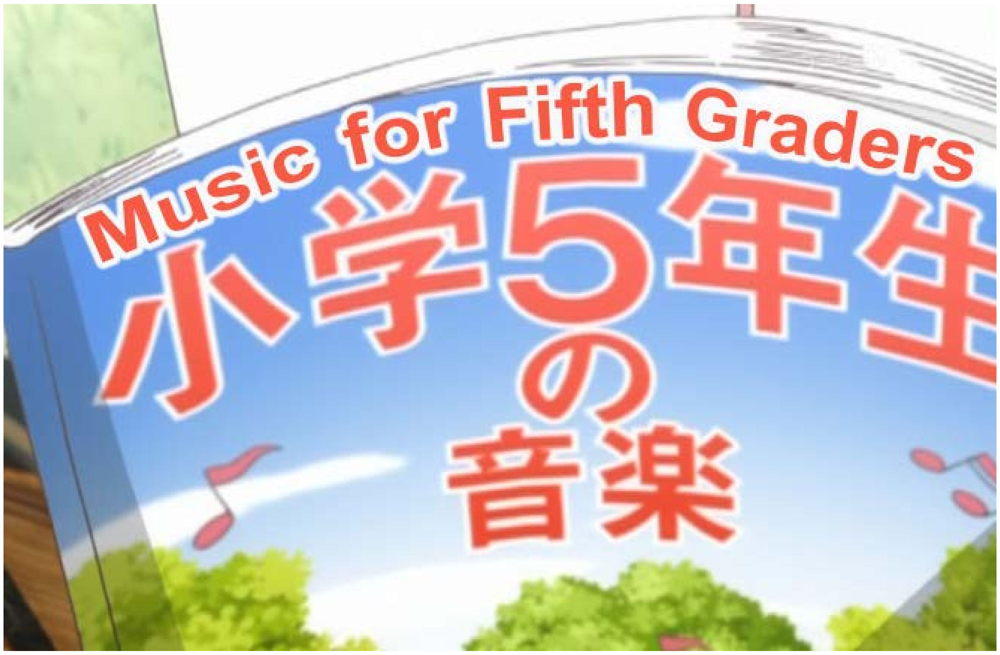
- They decided to use a mask.
- Failed to match the outline of the sign.
- Failed to cover the bottom of the sign.
- Failed to use blur on it, making it jagged as a chainsaw.
- Used a bit of transparency, so that you can see the kanji underneath, for whatever unknown reason.
- Used a FUCKING SHADOW on the mask!
- Put text on it.
- Each word going in different direction.

I. Don't. Even.

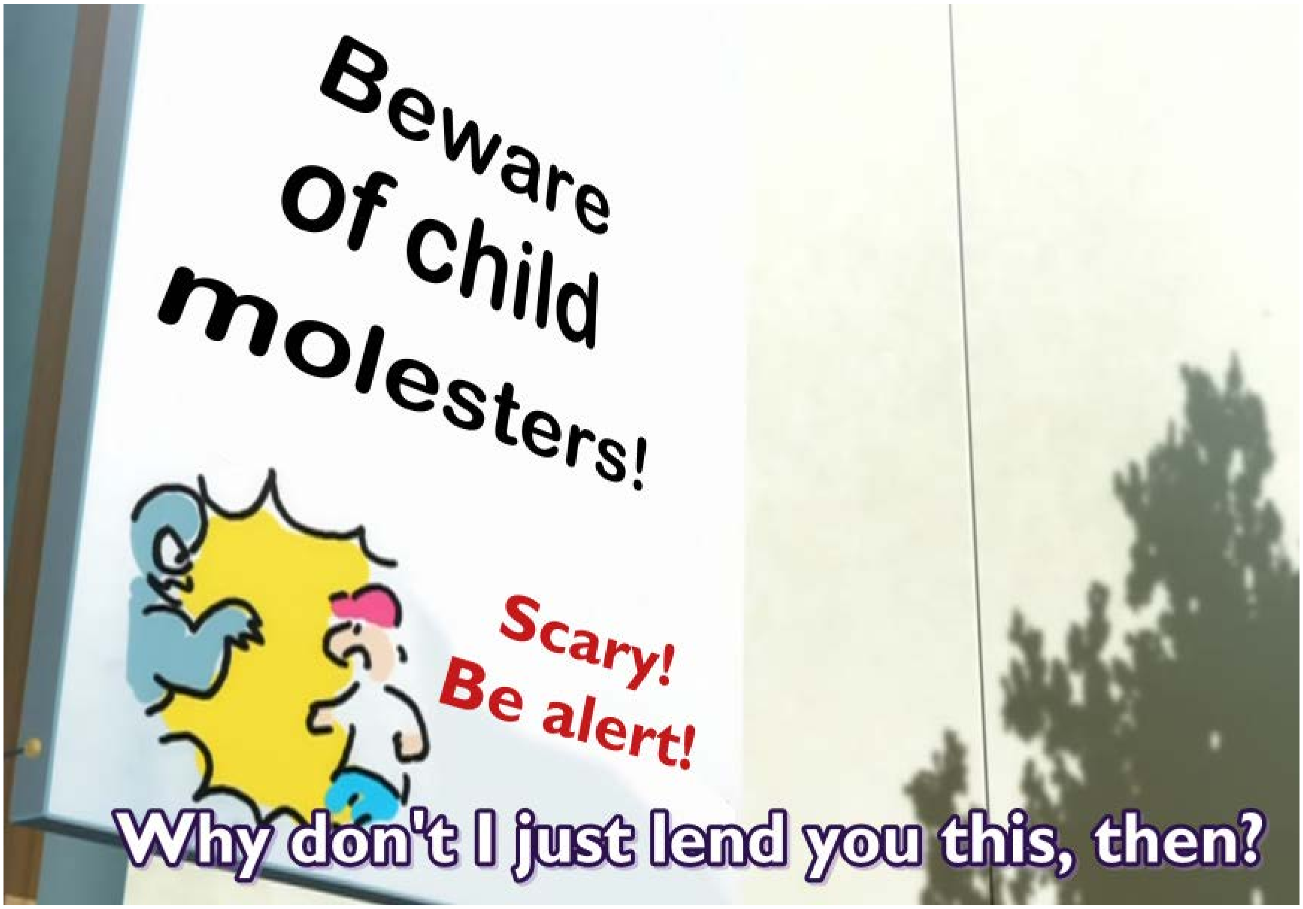


Hello. This is Hadena again. We can put text on screen, see?
What? What is blur? We don't heared about this. But sharper is better quality!
What, shadow? We can has no shadow? Oh. But. It look more proffessionull with shadow, no?

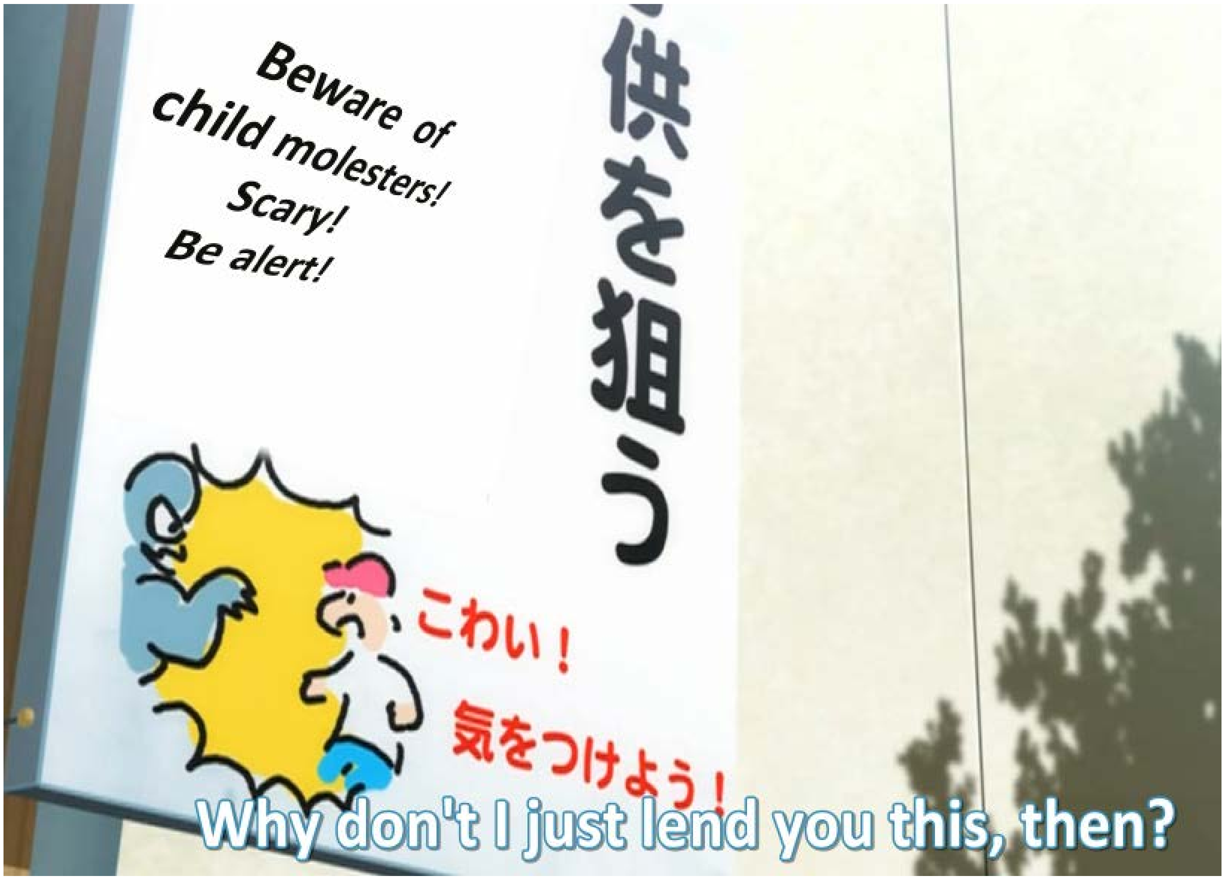
OK, anyway...



Here's some other derps. Clearly they don't have what it takes to be a legend like Hadena.
So they might as well try a bit harder and produce something decent maybe.
They could start by matching the colour right.
I know most people wouldn't bother with 2 layers but... it looks so much better if you can blur the red as well.
Oh well...


Beware of child molesters!
Scary!
Be alert!
Why don't I just lend you this, then?

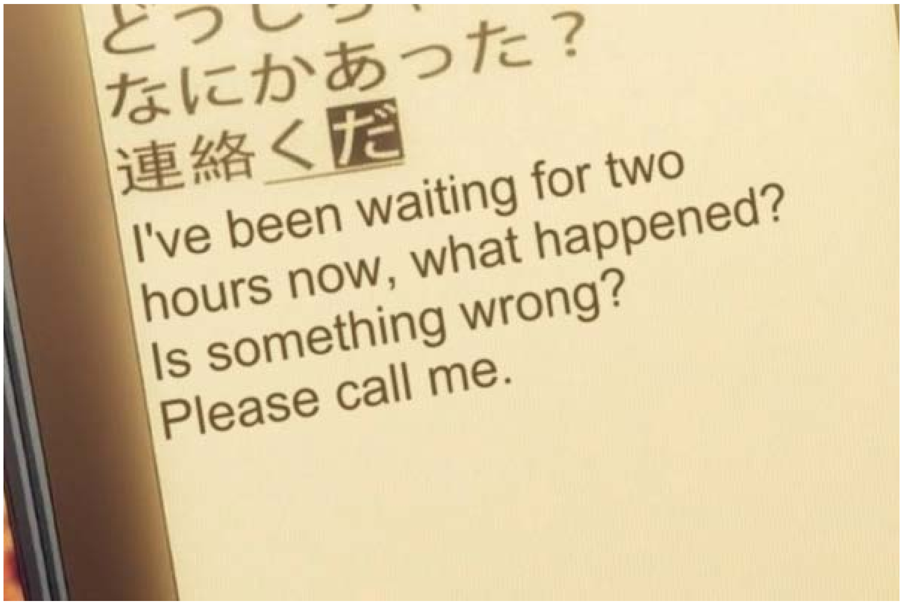
Scary indeed! Beware of brazillian typesetters! [I dunno, just guessing...]
Same guys as previous sign. Maybe they should go for a legend after all. This is pretty... scary.
Seeing that, somehow I'm not worried about molesters at all. There's something more sinister lurking around.
Also the main font with purple shadow...


供を狙う
Beware of child molesters!
Scary!
Be alert!
こわい！
気をつけよう！
Why don't I just lend you this, then?

But guess what? Beware some more, because Hadena is back, motherfuckers!
Typesets attacking you from unexpected angles, right about to drop a few child molesters on your head as they fly over you.
2015 Note: I have to wonder about the processes occurring in the head of the person(s) who thought that this was a good angle.

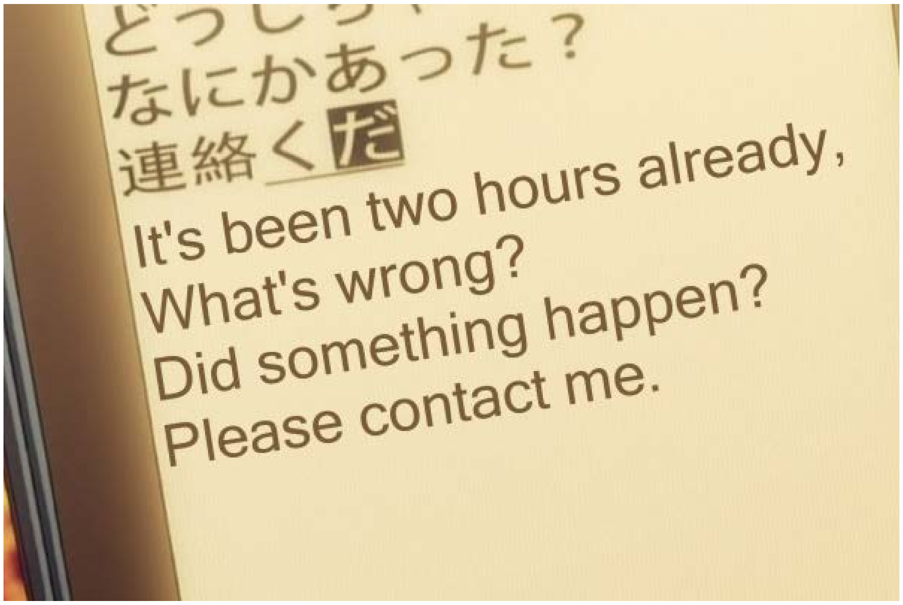
Honestly, the best group for this sign was HorribleSubs with \a6.
I thought the idea of fansubs was to *improve* the quality of crunchyroll, not make it worse. But what do I know.

Here's a bunch of groups doing the same sign:



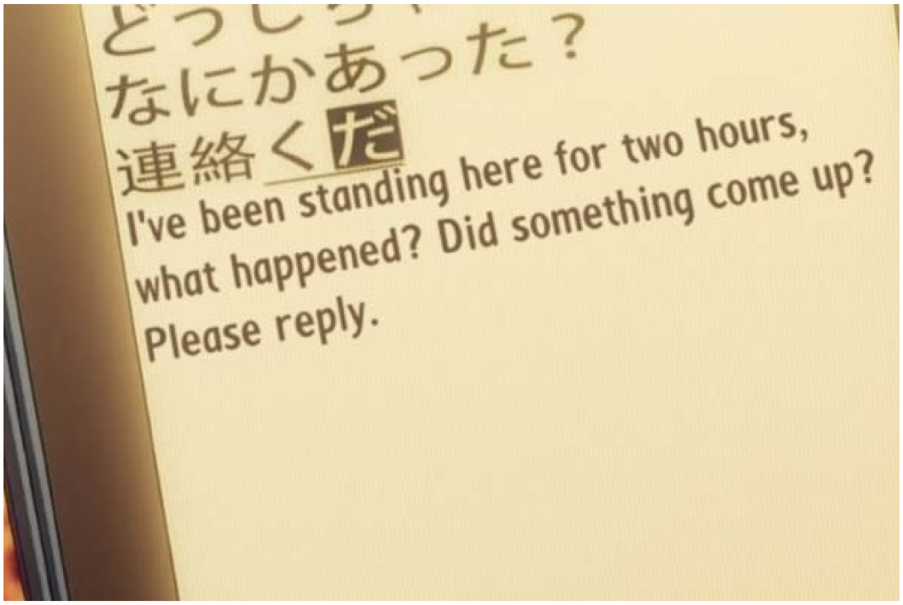
なにかあった？
連絡くだ
I've been waiting for two
hours now, what happened?
Is something wrong?
Please call me.

This looks ok, though the colour (and blur/glow) is a bit off.



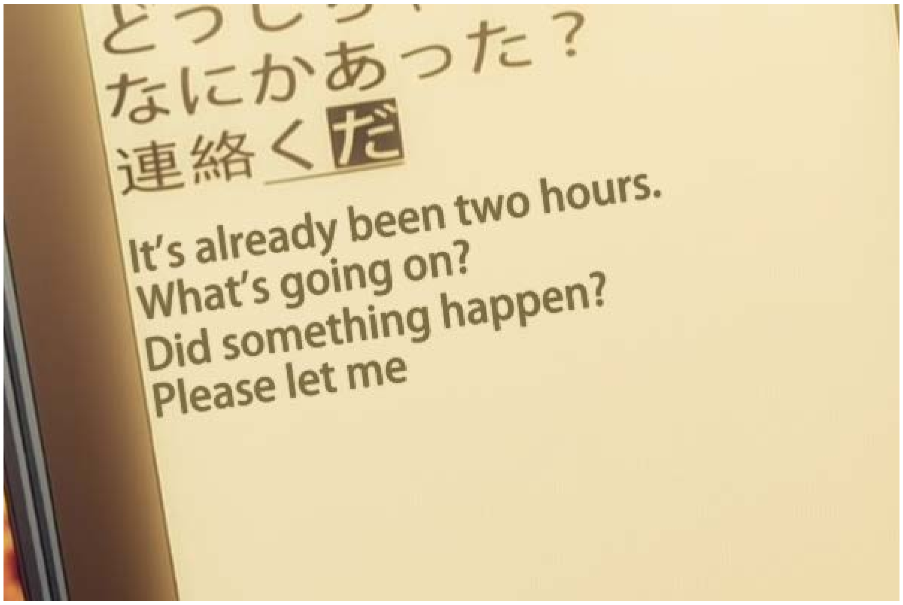
なにかあった？
連絡くだ
It's been two hours already,
What's wrong?
Did something happen?
Please contact me.

This colour is also off, and it's missing blur on top of that.



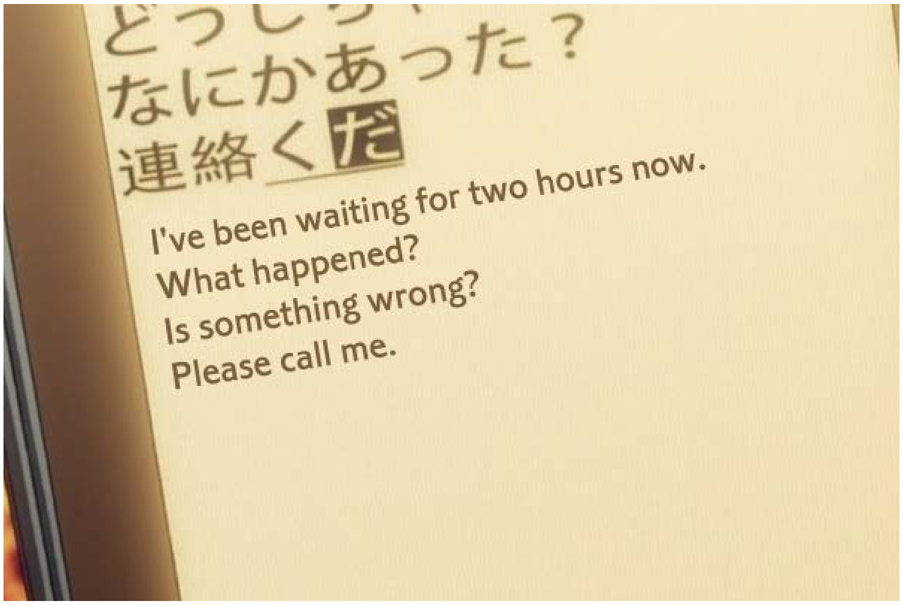
なにかあった？
連絡くだ
I've been standing here for two hours,
what happened? Did something come up?
Please reply.

Colour is off. I think this was herkz, who likes to laugh at others for exactly this, so feel free to laugh at him. Also it's only like 1 pixel away from the JP, which is pretty dumb, since there's plenty of space below.



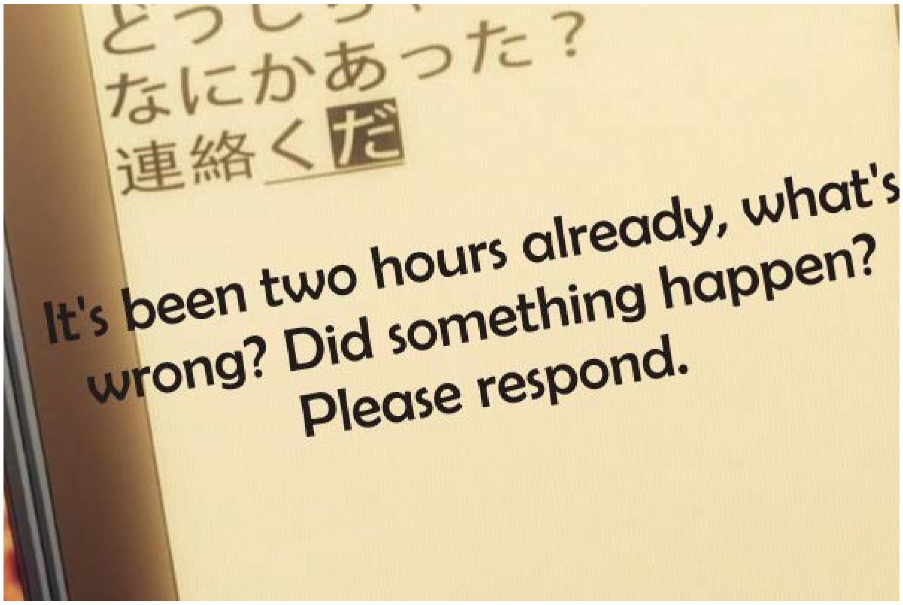
なにかあった？
連絡くだ
It's already been two hours.
What's going on?
Did something happen?
Please let me

Finally somebody got closer to the colour, best so far.


どうし
なにかあった？
連絡くだ

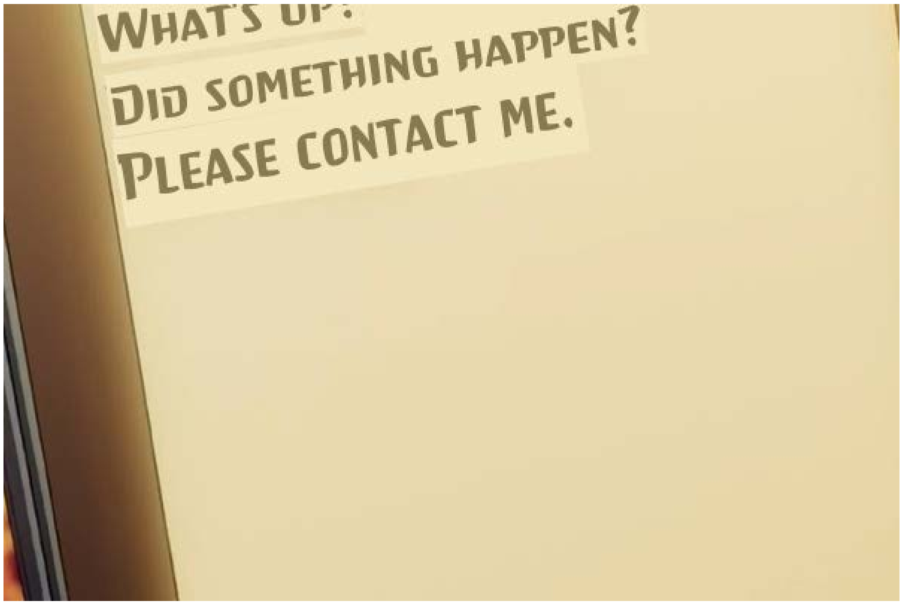
I've been waiting for two hours now.
What happened?
Is something wrong?
Please call me.

Here the colour is off again. I don't know what everyone's problem here is. Also seems to lack blur.


どうし
なにかあった？
連絡くだ

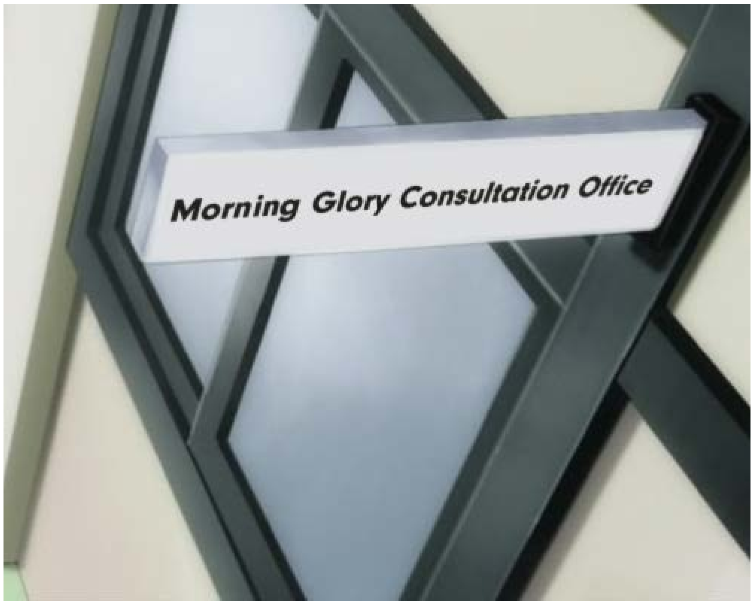
It's been two hours already, what's wrong? Did something happen? Please respond.

Yeah, I know what your question is. The answer is: asuka subs.
What can I say?

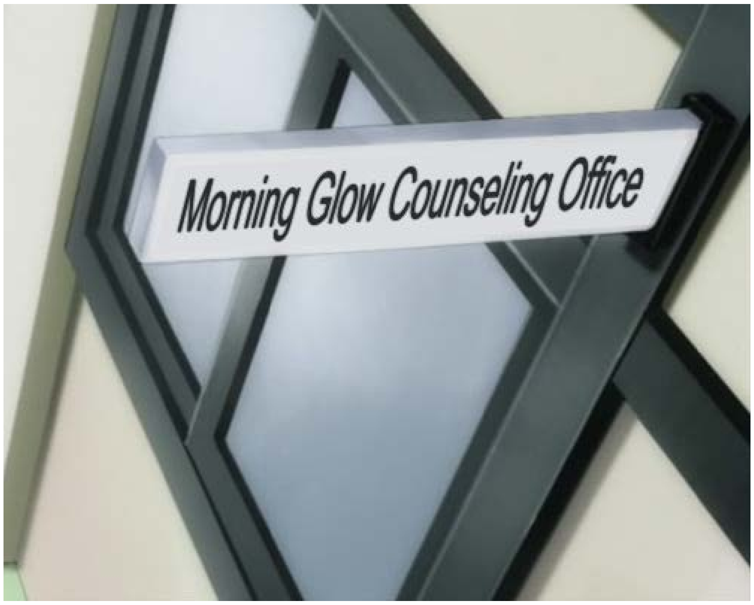


WHAT'S UP?
DID SOMETHING HAPPEN?
PLEASE CONTACT ME.

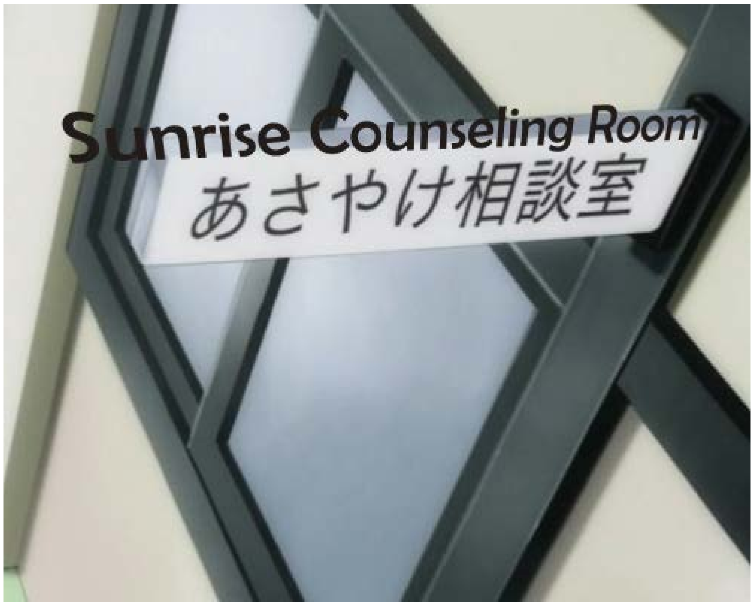
Nope.
It's true that replacing the JP seems like a good idea here since nothing's in the way.
But first, this is not the kind of font you'd have on a cell phone.
And more importantly, if you can't match the colour of the background, just find another job.



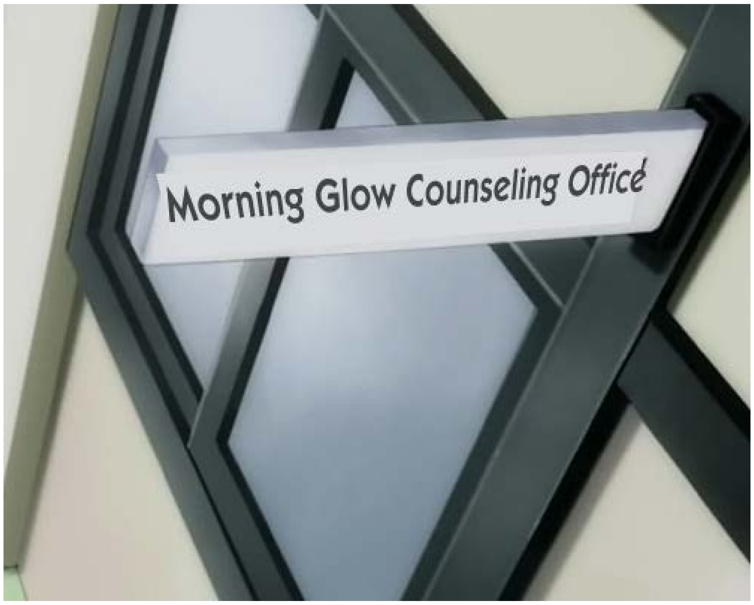
Yep.



Good enough, but could be better.
The end should be leaning a lot more to the right, you can clearly see the edges of the mask, etc.
You know, if you just put \blur2 on the mask, it'll do the trick in many, if not most, cases.

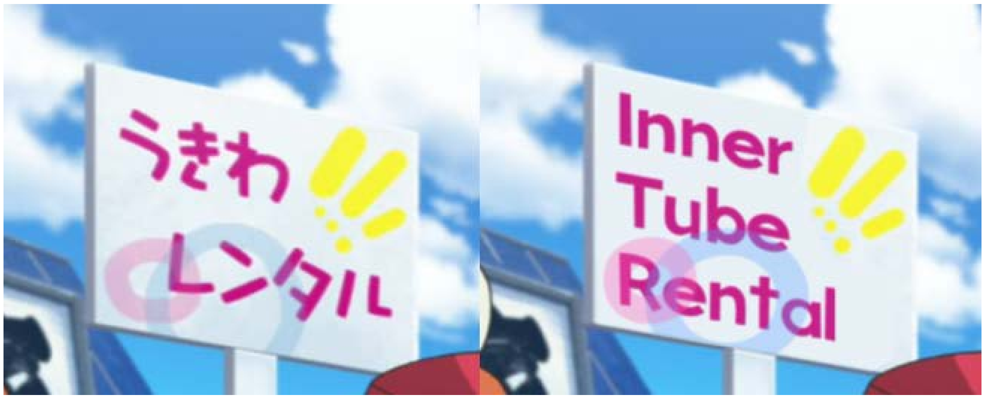


Uh-huh. Ok. You were saying you can typeset? I see. Where did you get that idea?

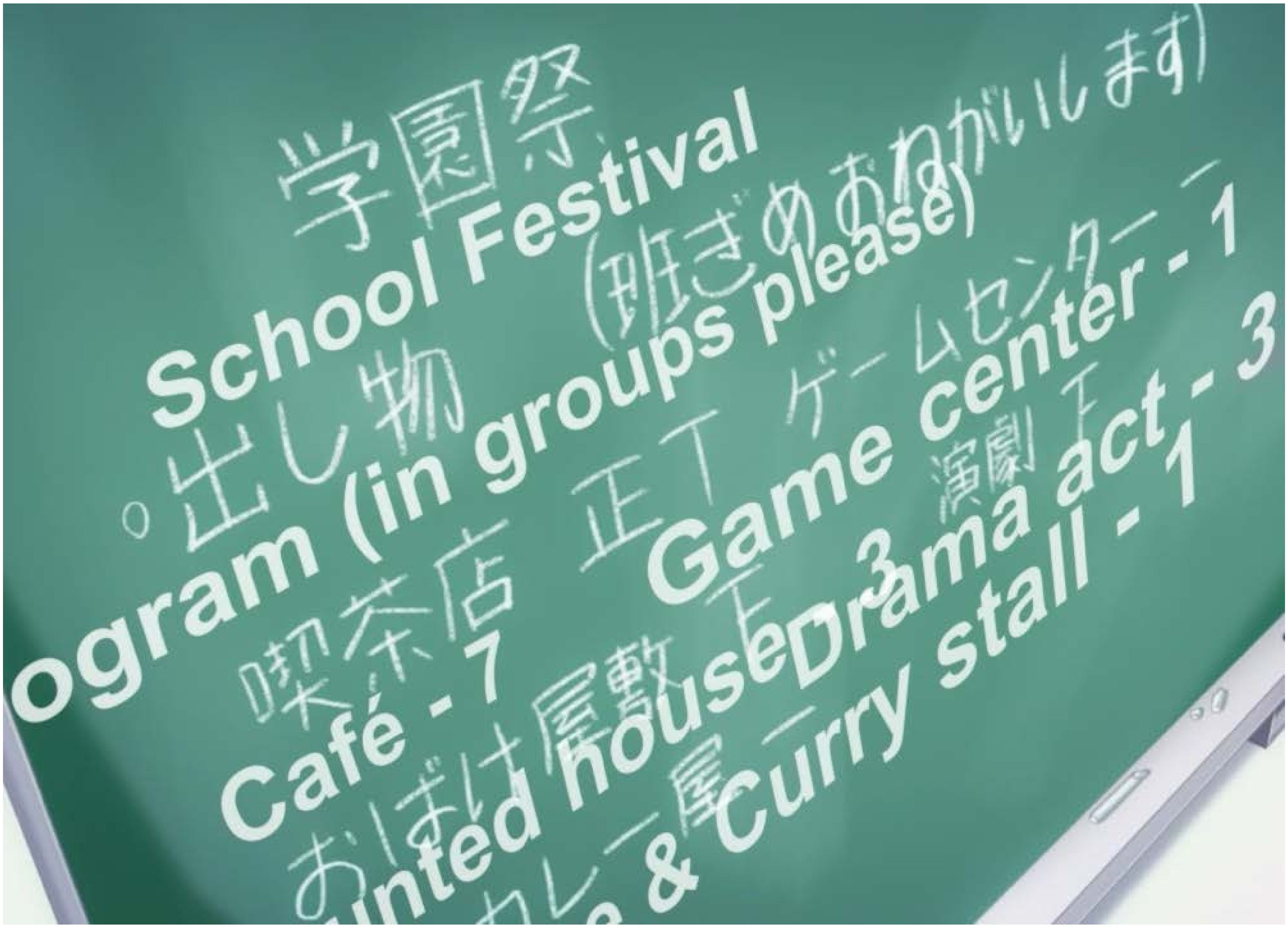


I don't even...

Here's something fun I did recently:



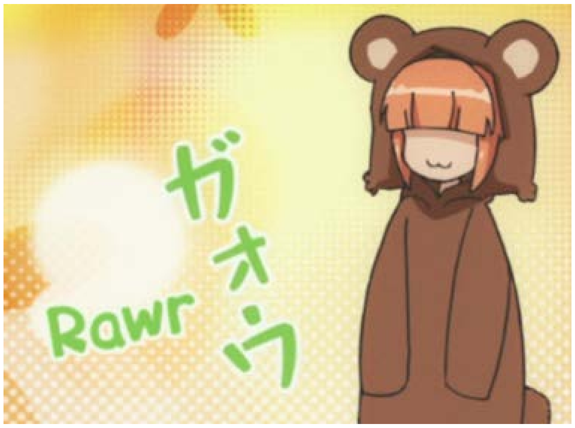
What do you need to do this?
First, a fitting font. Check.
Now, there's not enough space on the sign to put the English next to the jp, so... masking.
Make a regular square mask, scale it to the size of the sign.
Use rotation and \fax or \fay to make it fit.
Use the clipping tool to cut the mask so that the yellow thing is not masked.
The main problem is that you can't mask the letters without masking the circles too.
You could just ignore the circles and pretend like they weren't there but... we're no amateurs here, right?
So we make circles. Make letter o or O, scale it up, blur, set alpha to ~H80 [so that you can make the circles overlap], set colour.
Use clip to limit the blue circle to the sign only.
Now the letters... colour, blur, rotations and \fax to align the sign properly.
Set the layers correctly... mask is 0, circles are 1 and 2, letters are 3.
Done.

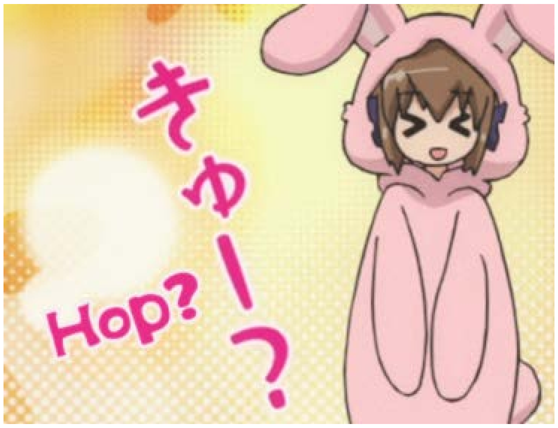


Let's take this opportunity to mention something else here. This is not really a case of bad typesetting per se.
What happened here is these guys muxed in more than 36 MB of fonts. That's not only pretty retarded, but also doesn't exactly work, as you can see.
It's a limit in haali splitter or something. If you go over 36 MB with attachments, it only takes whatever 36 MB it grabs first and the rest is ignored.
So if you stuff your release with a few 10 MB fonts, you may end up with a bunch of fonts missing for playback and some terrible typesetting.
I was using 31 fonts for Acchi Kocchi, and they were 1.3 MB all together, so I don't really get what's with this bloatcrap in many groups' releases.
Also realise that somebody probably spent like half an hour typesetting this, and this is what the viewers saw in the end.
While we're at this, also don't use mkvmerge over 5.1.1 [I think] because that causes some more problems with otf fonts.
In fact, stick to mkvkerge 4.1.1 [or if higher, check the "disable header removal compression bla bla" option].

So that was plenty of examples of the wrong stuff. Now I should probably add some more of the good ones.
I just typeset this Acchi Kocchi extra episode, so here goes...

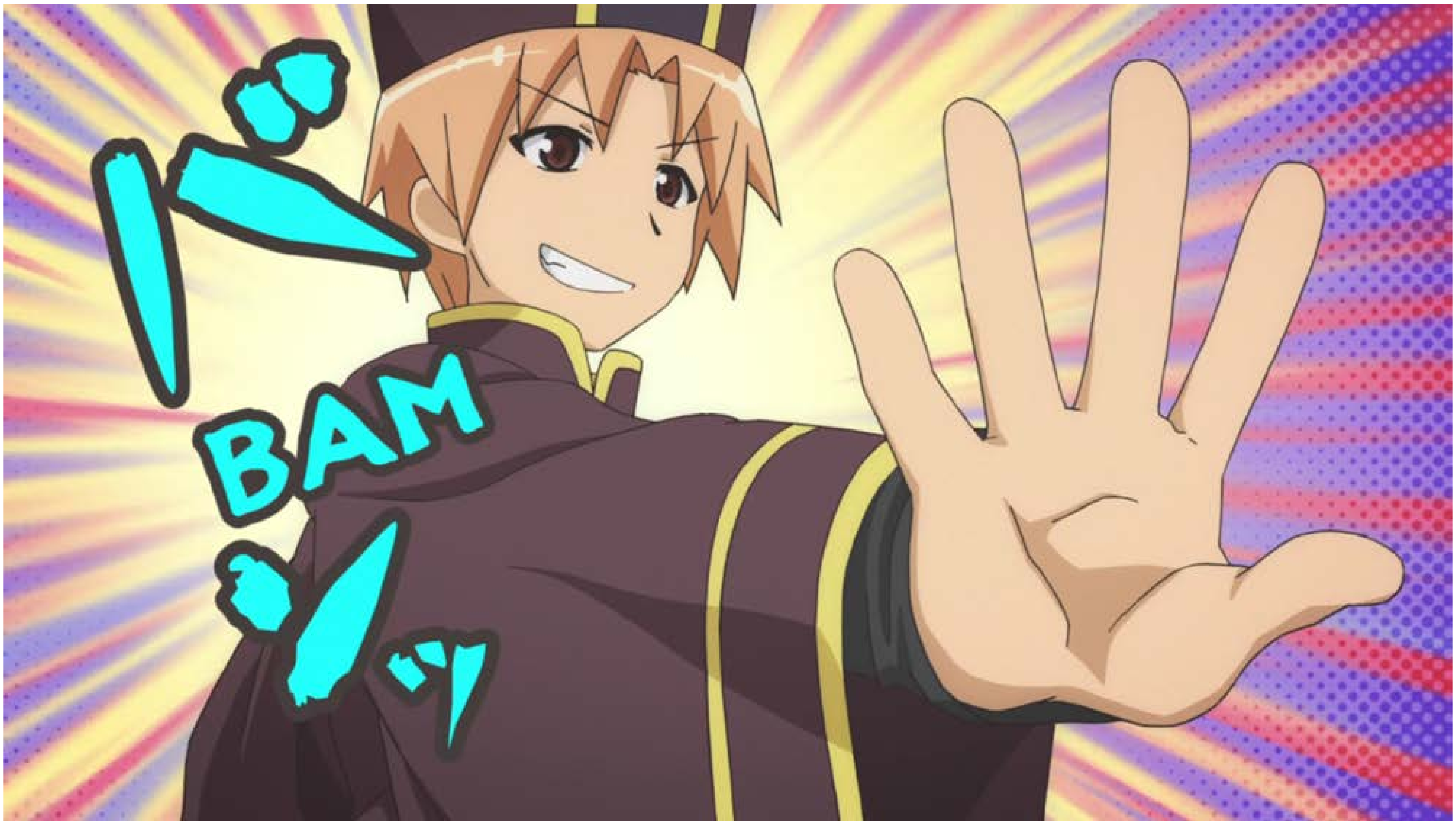
[In the light of the previous screenshot: I used 30 fonts for this episode, total size: 1.8 MB.
Hell, even all the fonts I used for all the 7 Maria episodes I typeset were 3 MB in total (about 55 fonts).]

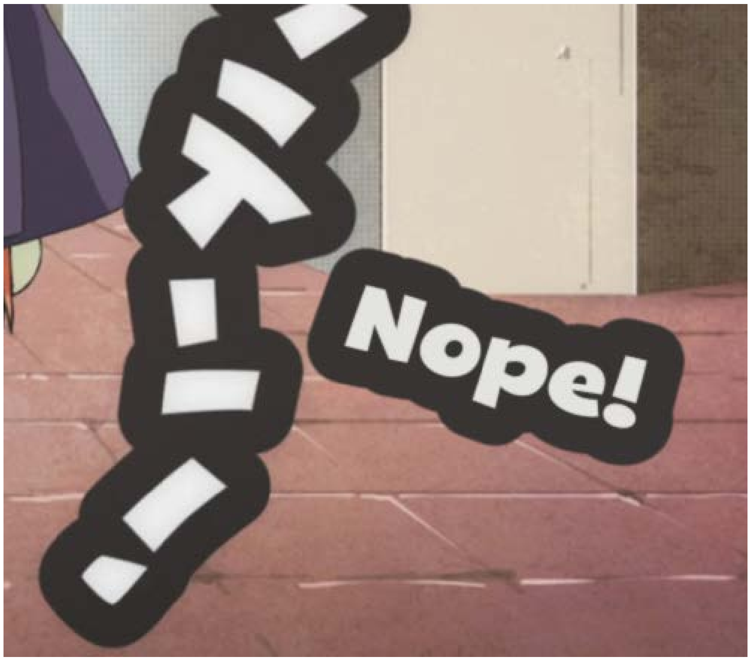


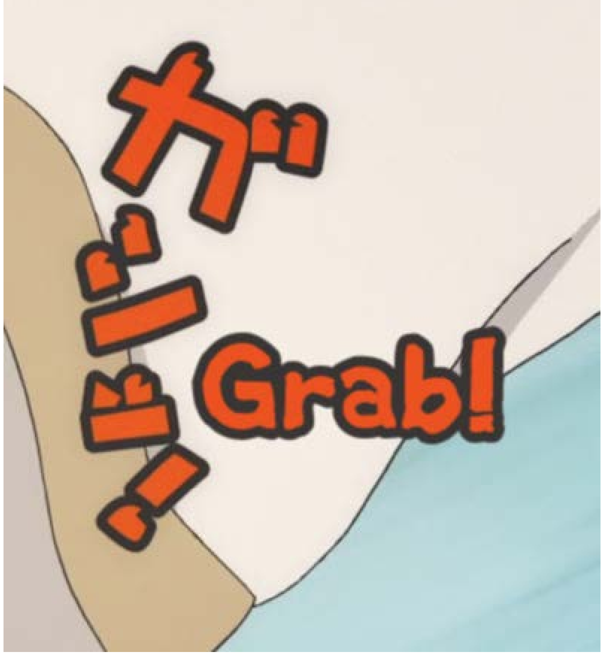


This would be the very basics:
1. appropriate font
2. correct colours
3. reasonable font size + border size
4. right amount of blur in 2 layers

It shouldn't be apparent that it's not a part of the video. This is really easy, so please don't fuck up signs like these.

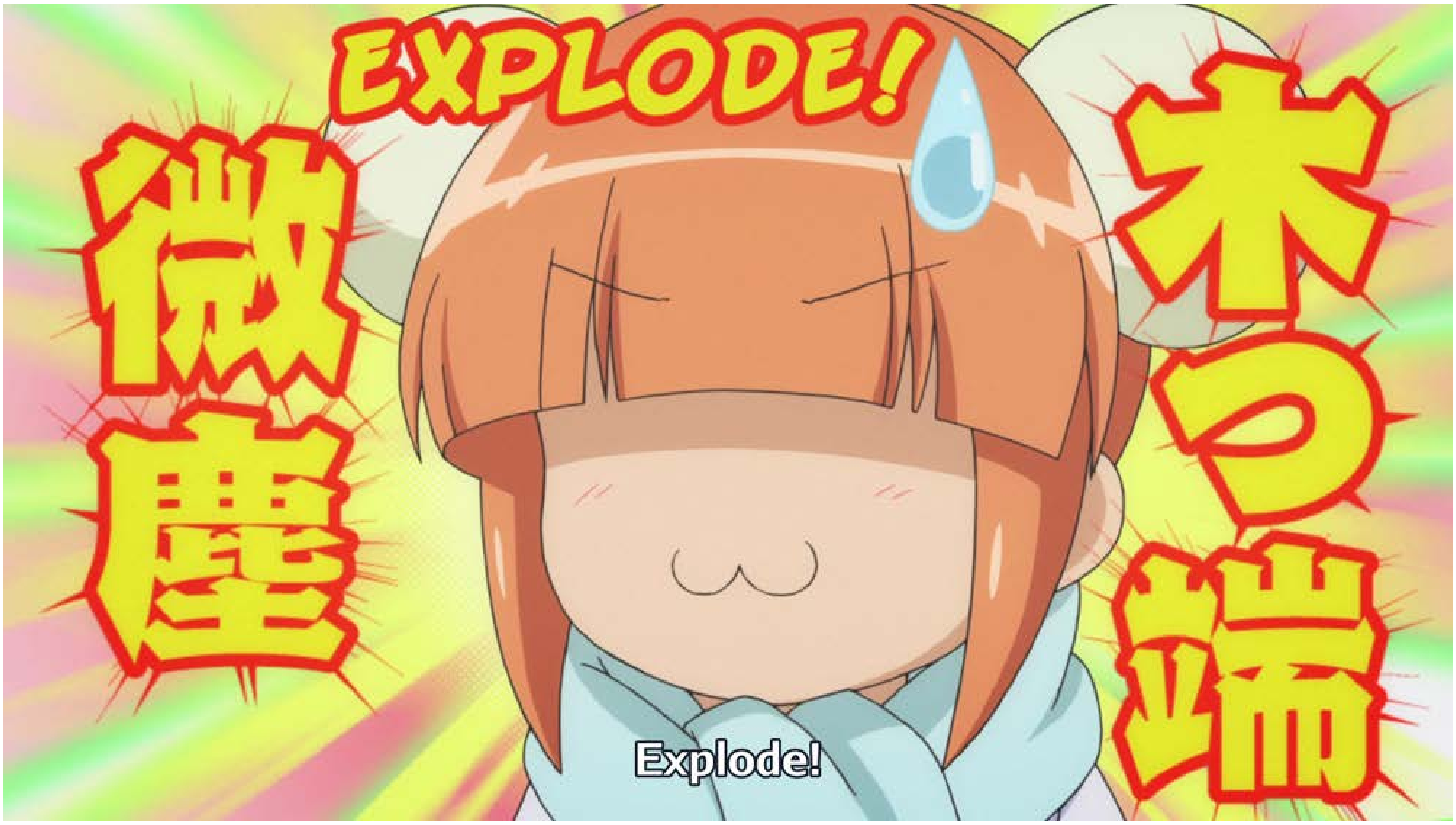




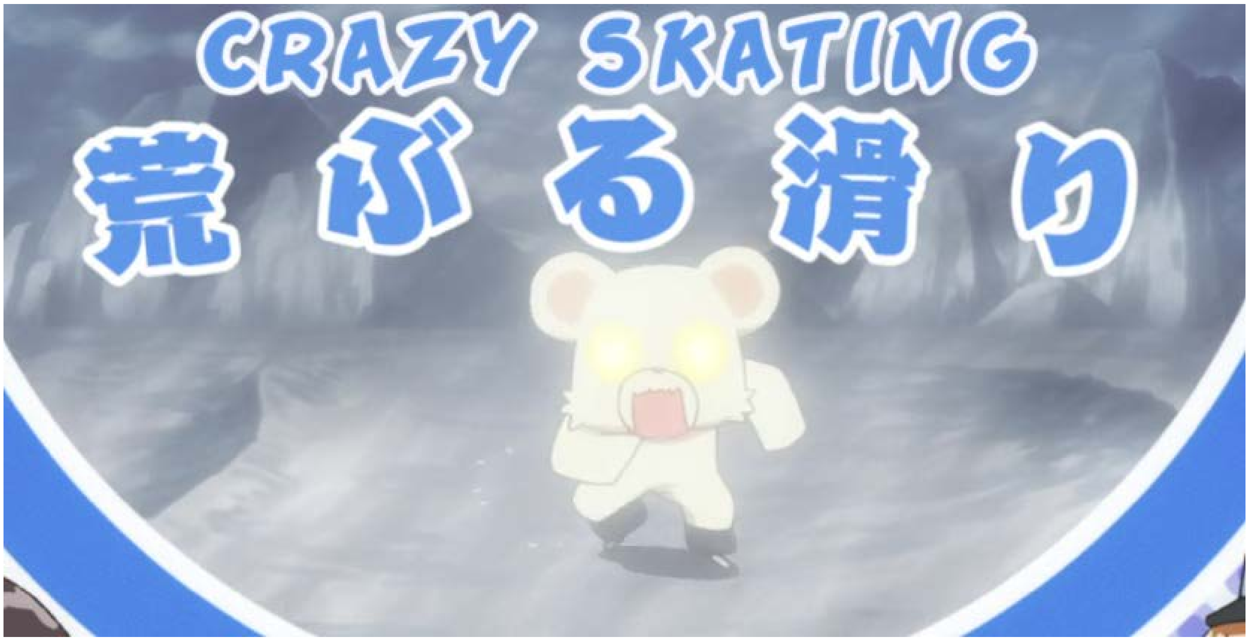


Slipped
ガーャン

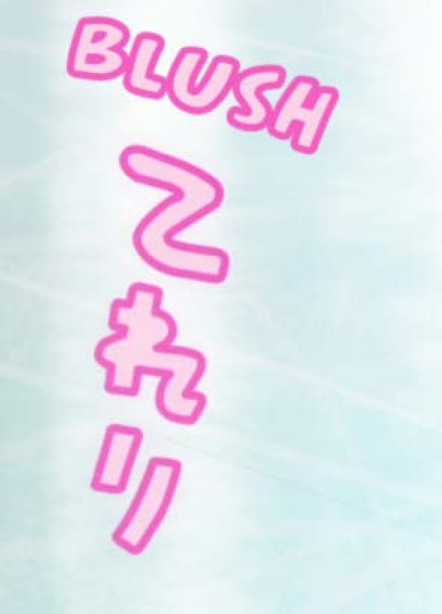
Sob
しく
Sob
しく

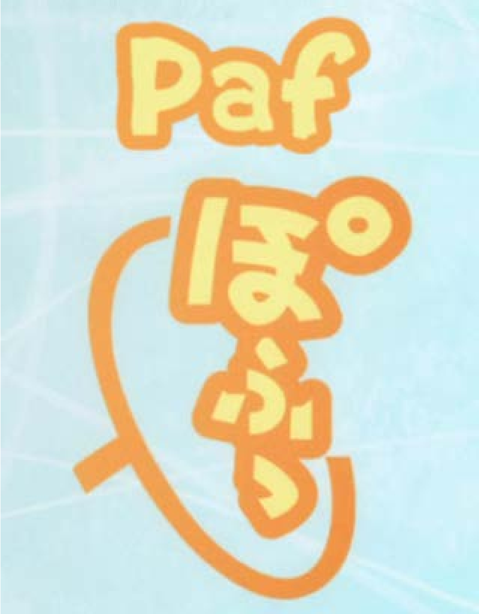


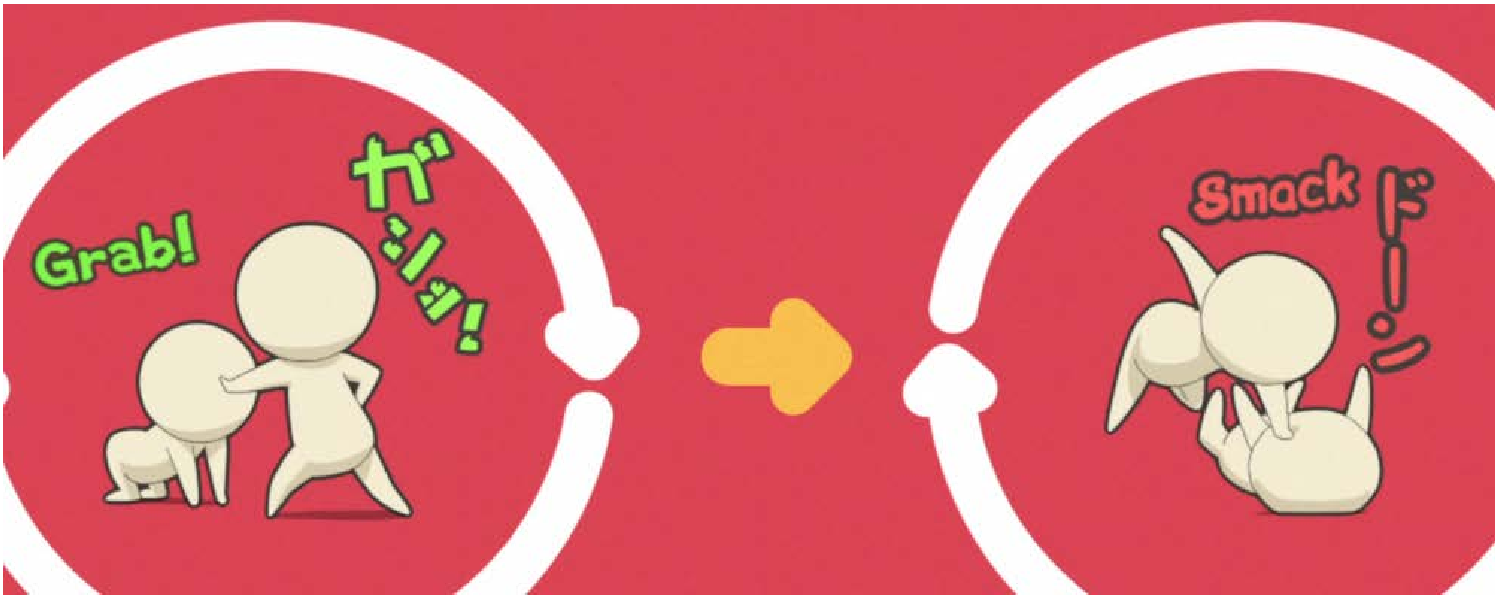
I would add the spikes if it wasn't moving but... it is, so meh.



Some \fsp might help a bit here.

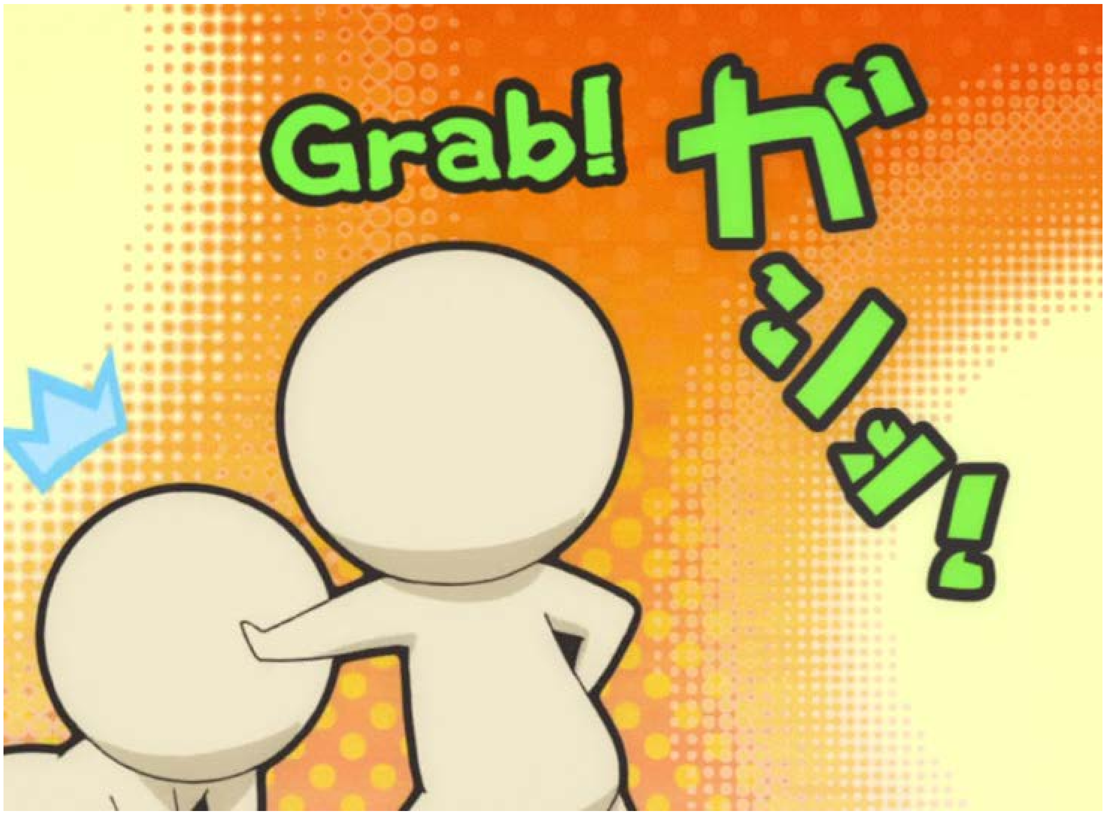


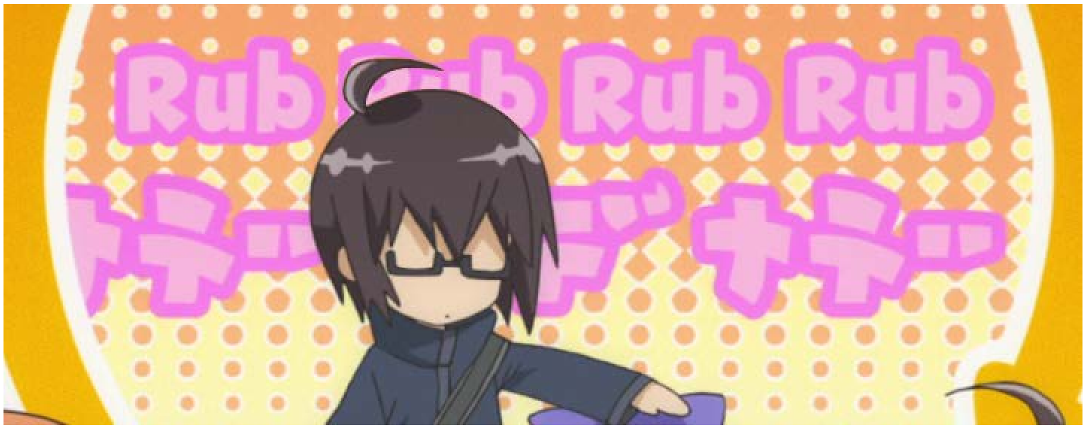



Grab!
ガシッ!
Smack
ドーン

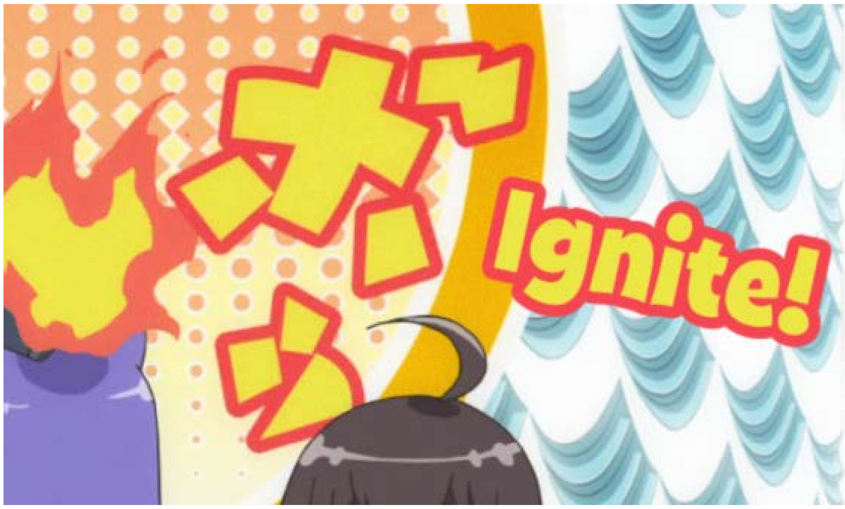
I'm dreaming
夢か

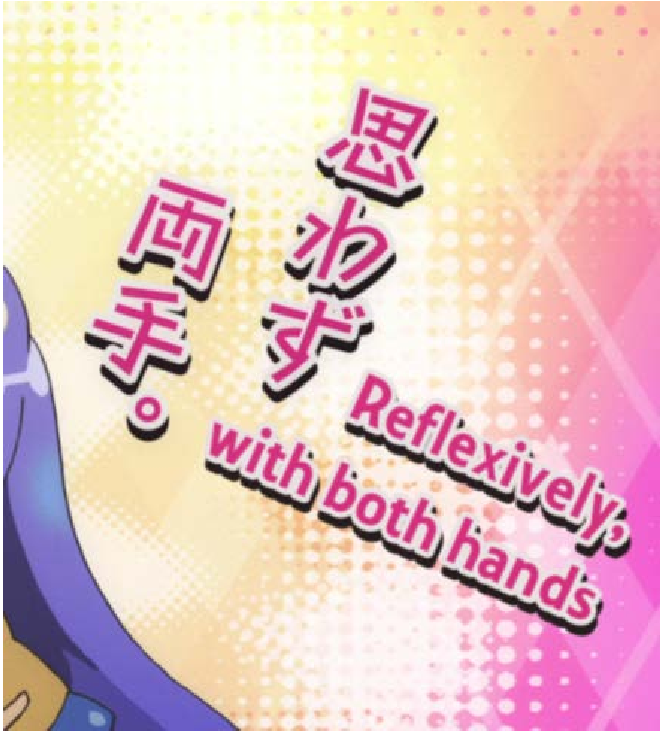
どうしよう…
What should I do...



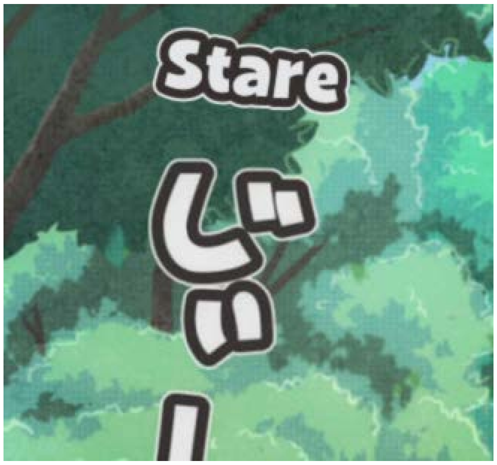


Some clipping here. If you're gonna clip your sign, you'd better do it right, and not like some hadena or doki derps.





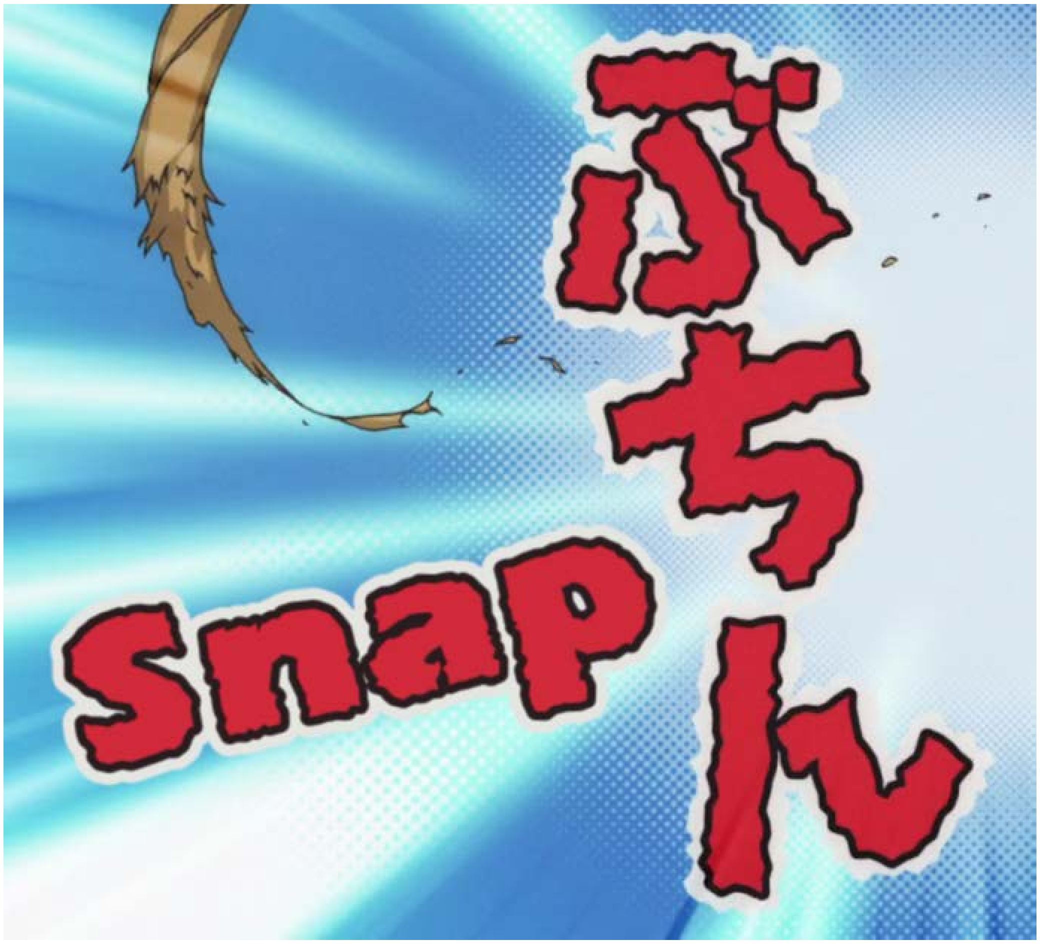
Shadows shouldn't be difficult either. Use \xshad & \yshad if the shadow needs a different direction.
This one has only \yshad, for example.



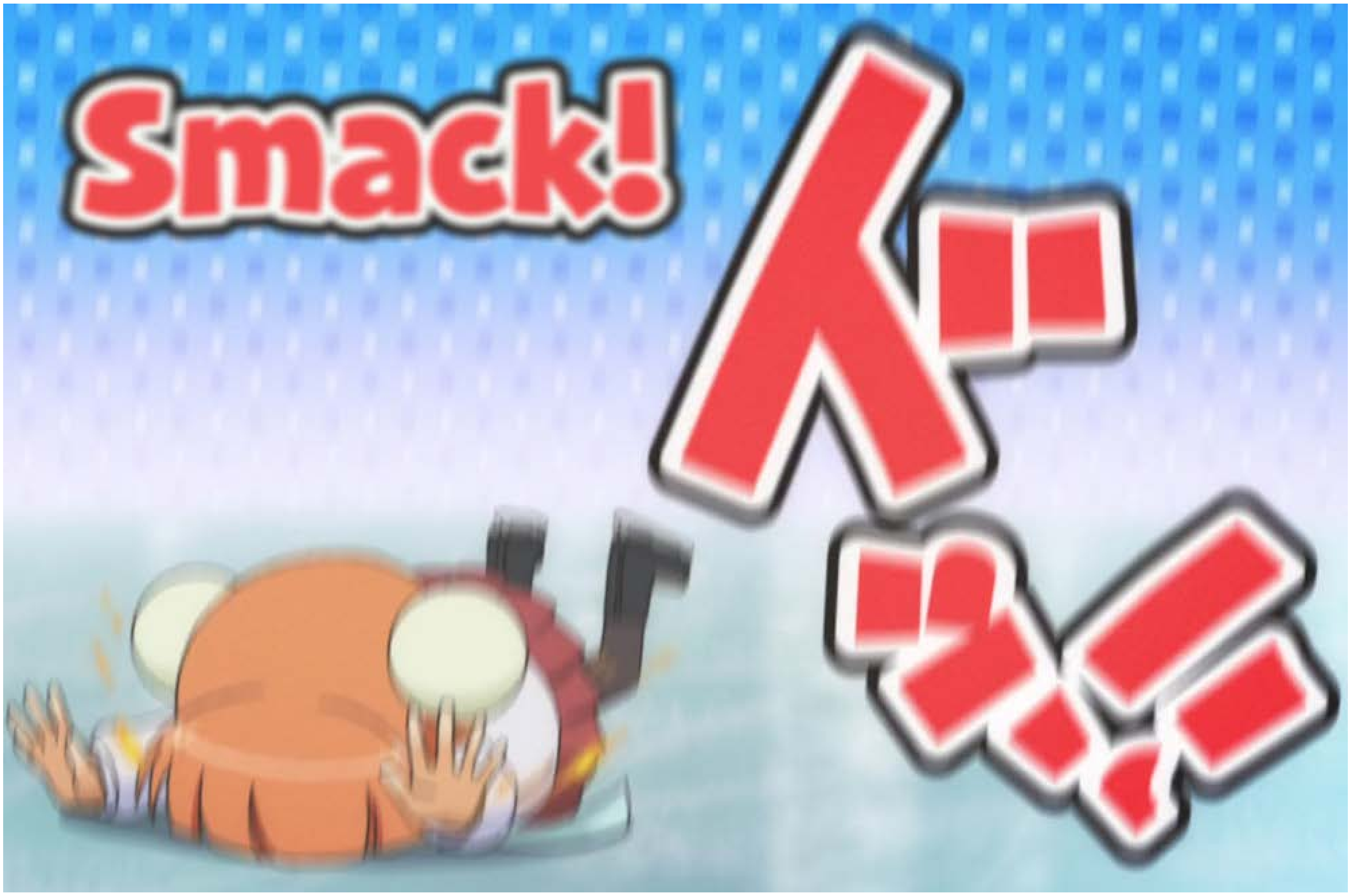
Two borders are also easy. You just need 3 layers instead of 2.

LOTS
どっきり

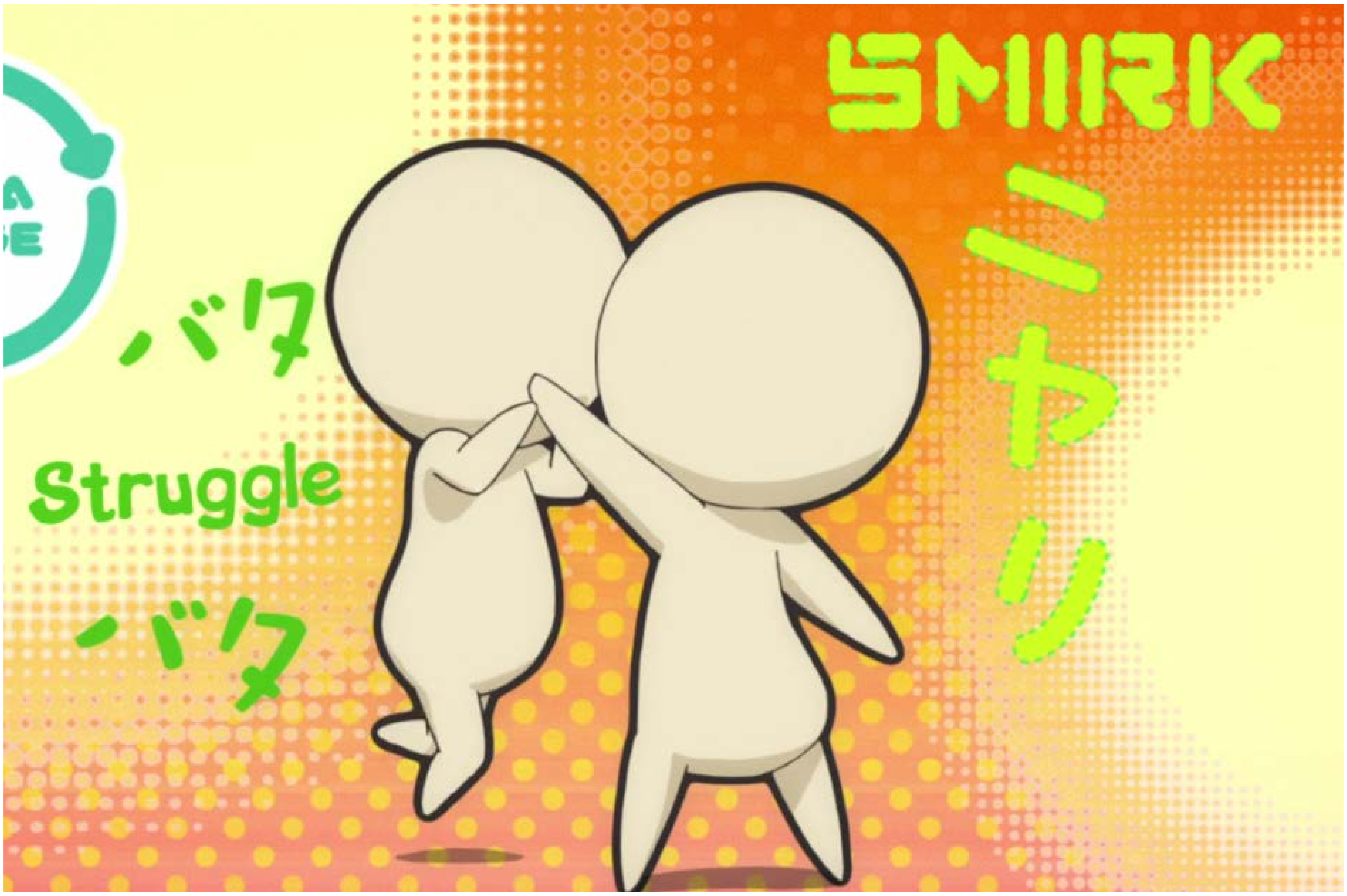
カッ
Excited



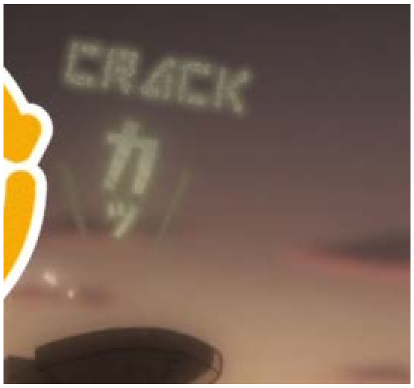
Choosing the right font can help a lot.



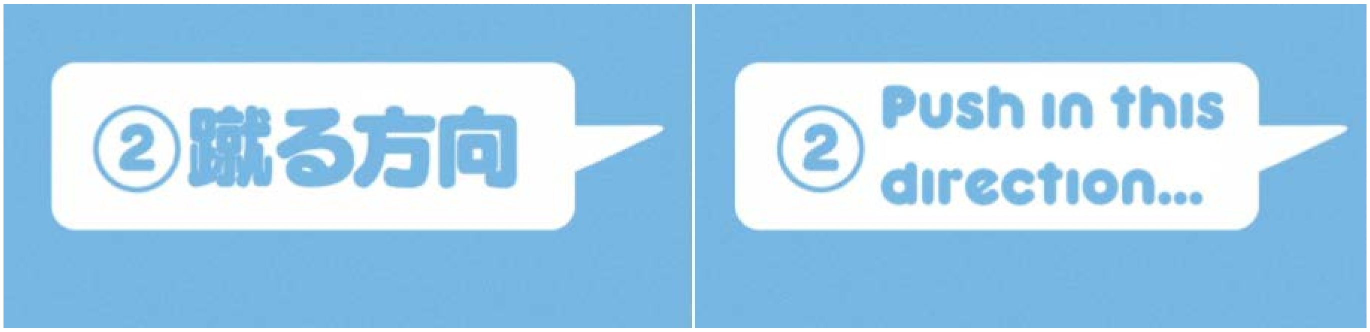
Using blur isn't really that difficult either. I don't know why more than half of the groups out there keep failing at this.



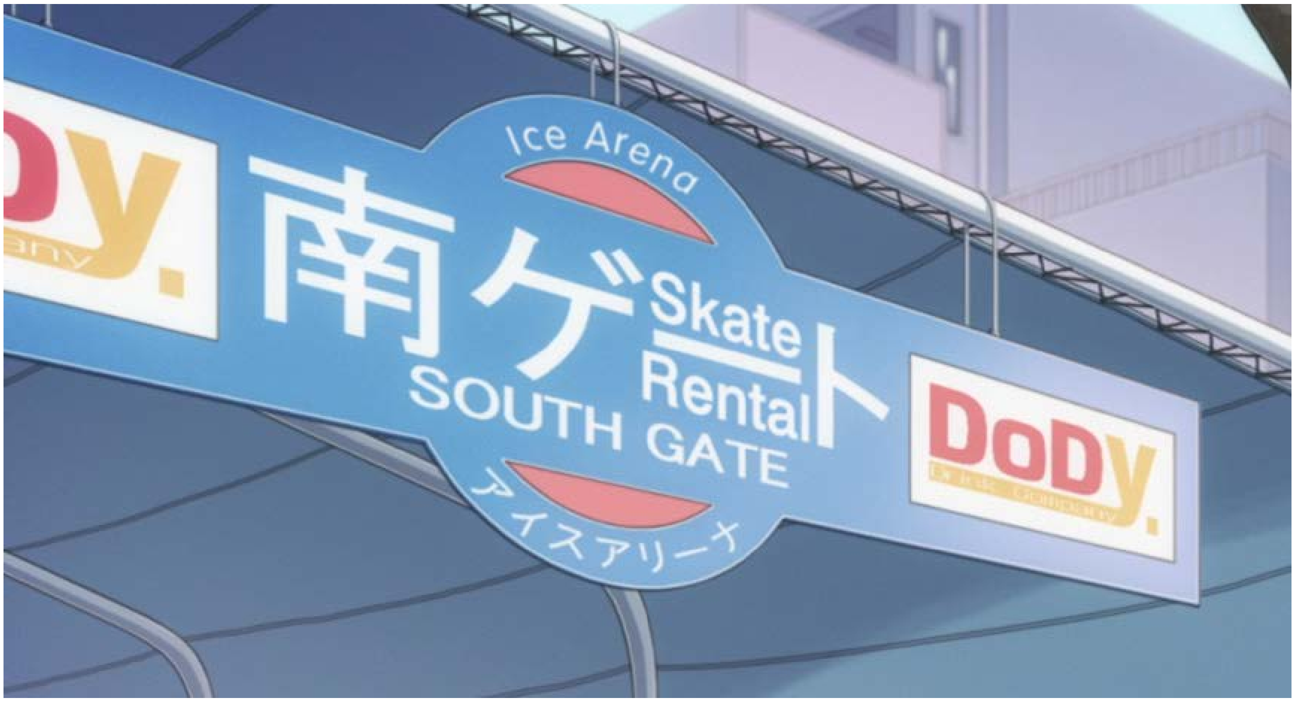
I really wanted to imitate the dotted outline here without too much effort [like ASSDrawing all of it].
So I used a font I had 2 versions of - a clean one and an eroded one [you can also see that one below].
I made the top layer with the clean one and the bottom layer with the eroded one.
I didn't actually use border so i just played with the font size, scaling and spacing till it looked like this.
It's not perfect, but at least it's going in the right direction. Considering that it wasn't much effort, I'd say it's decent.



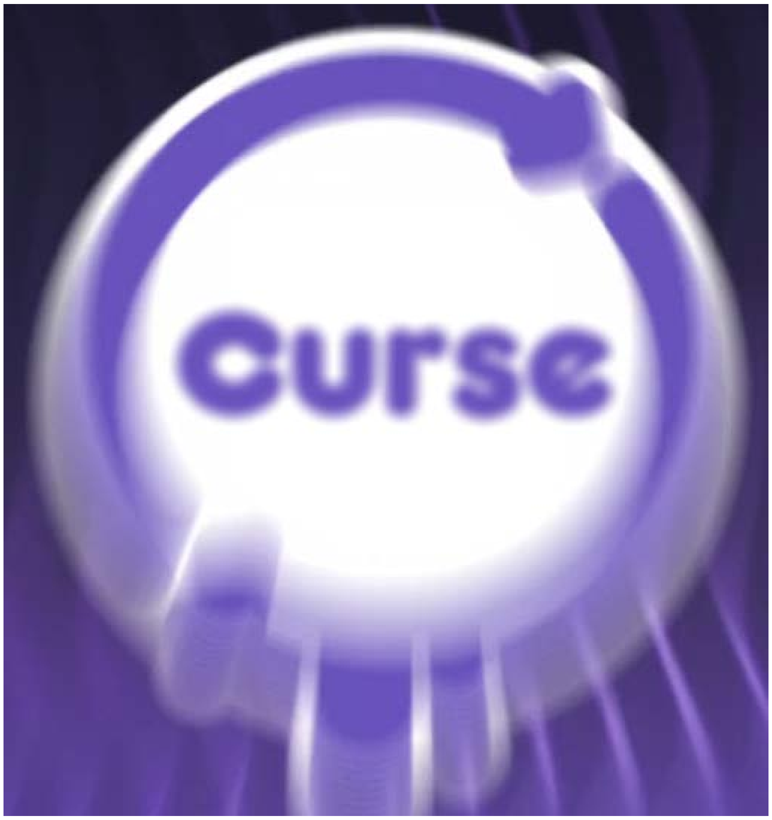
This is kinda hard to reproduce because it has a lot of colours and a lot of blur and some transparency...
but it didn't end up looking too bad, all things considered.

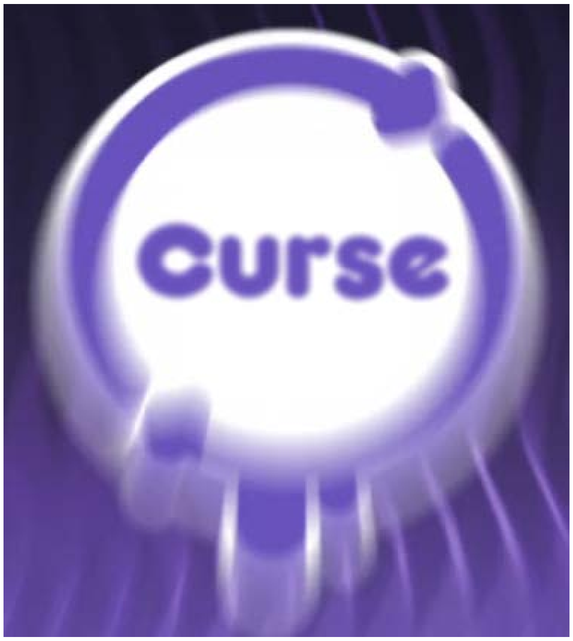


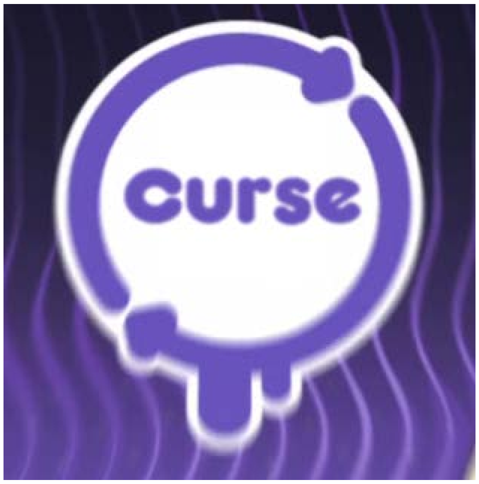
Sometimes masking is obviously the better option, because you don't wanna use \fs15 and make it look bad and unreadable at the same time.

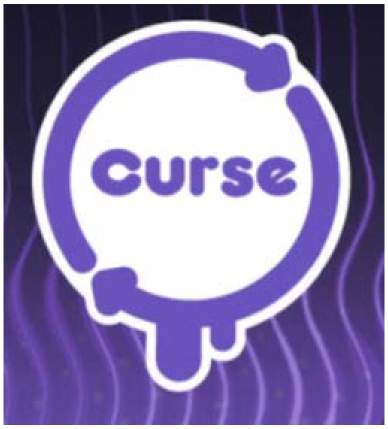


Here masking didn't work because there are like 100 shades of blue [and it moves], but the 'Skate Rental' looks pretty good like this.
The 'Ice Arena' not so much, but I didn't wanna make 8 lines that are moving, so this was about the best I could do in one line.
Not sure how much lag would doing each letter separately cause, but I didn't feel like trying.
It would also be possible to use \org to align the letters better and mocha it but... sounds like a lot of pain, doesn't it.
Besides, with the perspective here I'm not even sure what the letters should really look like [though doing each one separately would certainly improve it].
I think the main thing is to not make it stand out so that by just glancing at the picture you don't notice there's something too obviously weird.

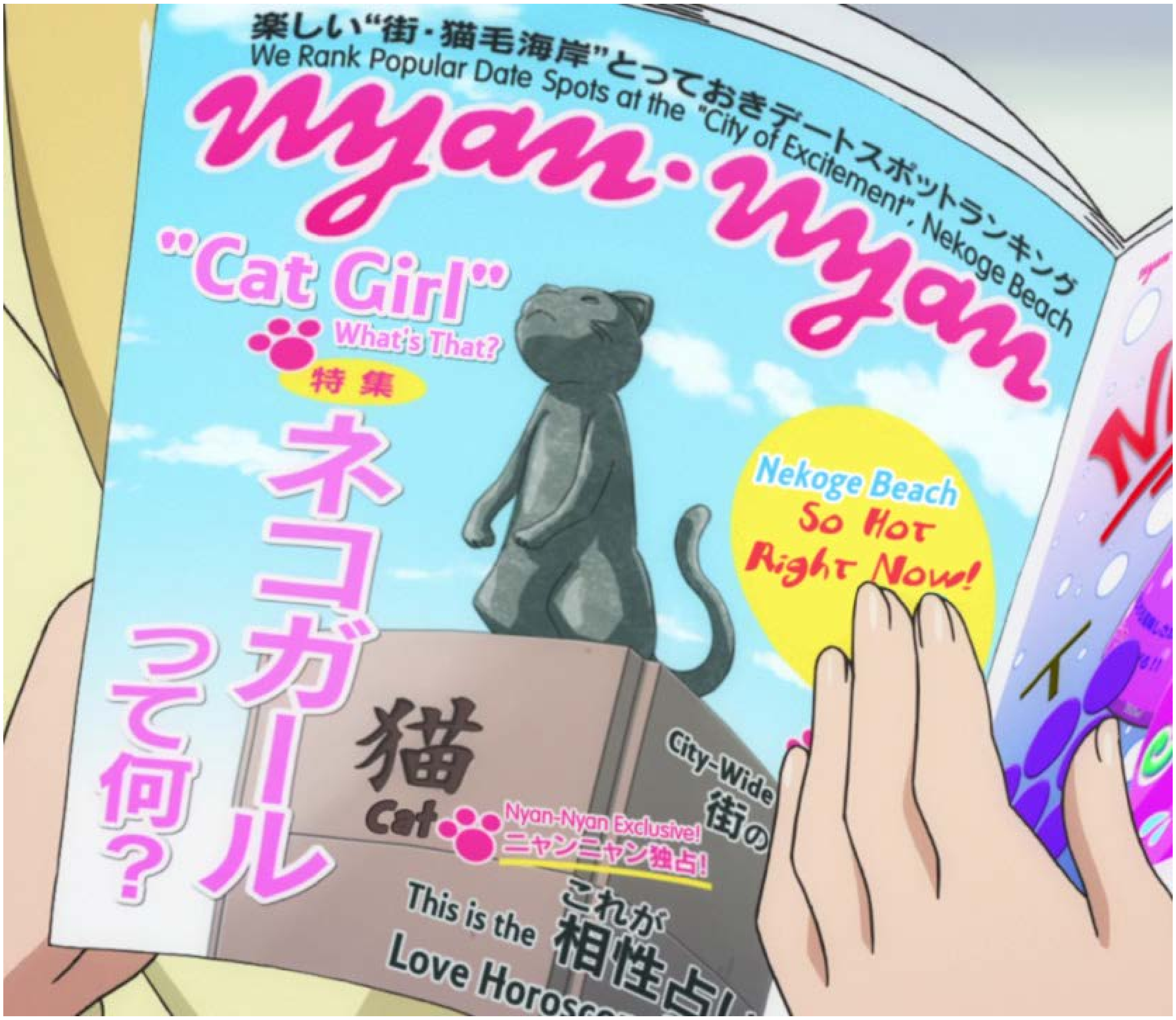




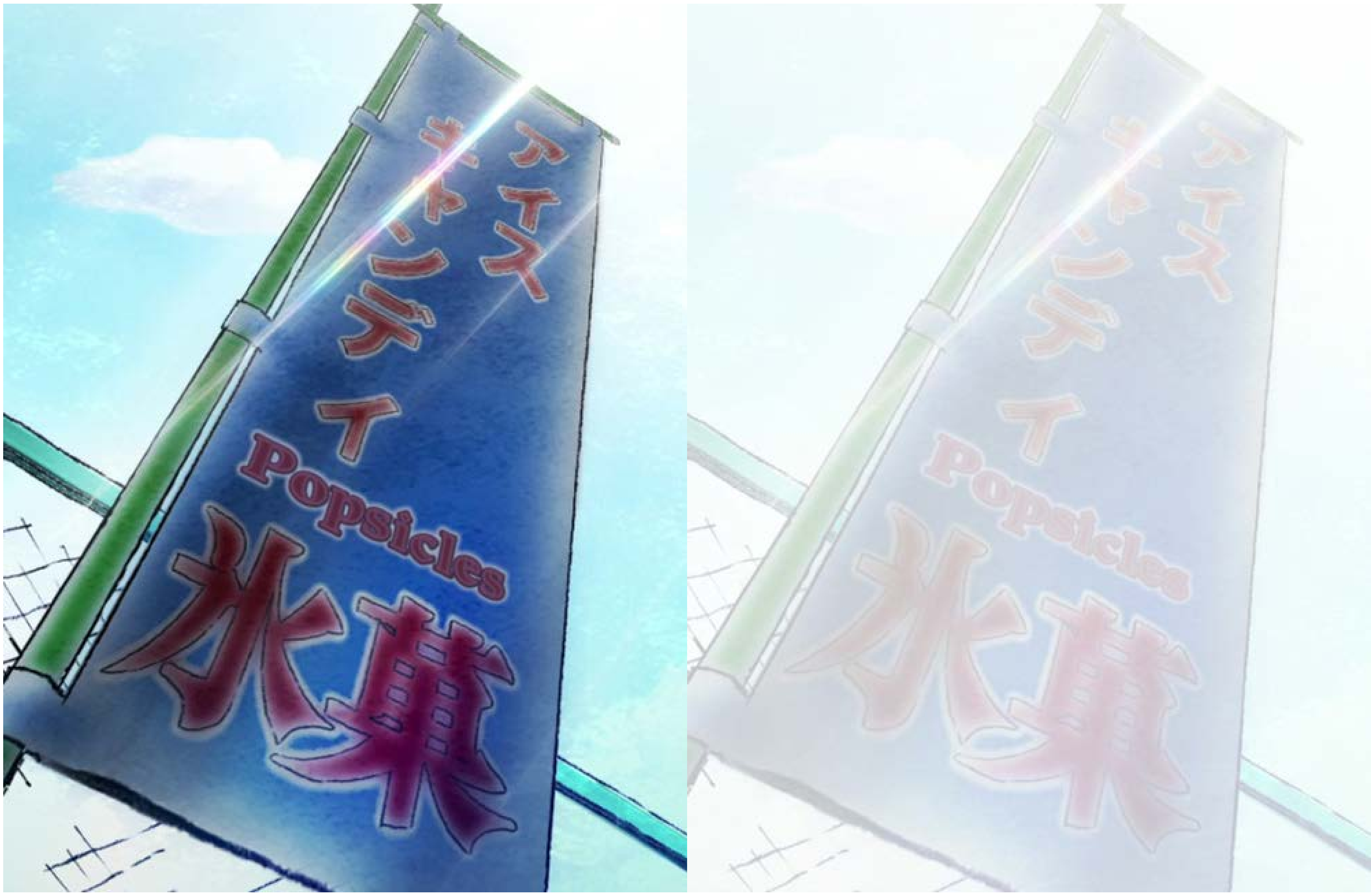




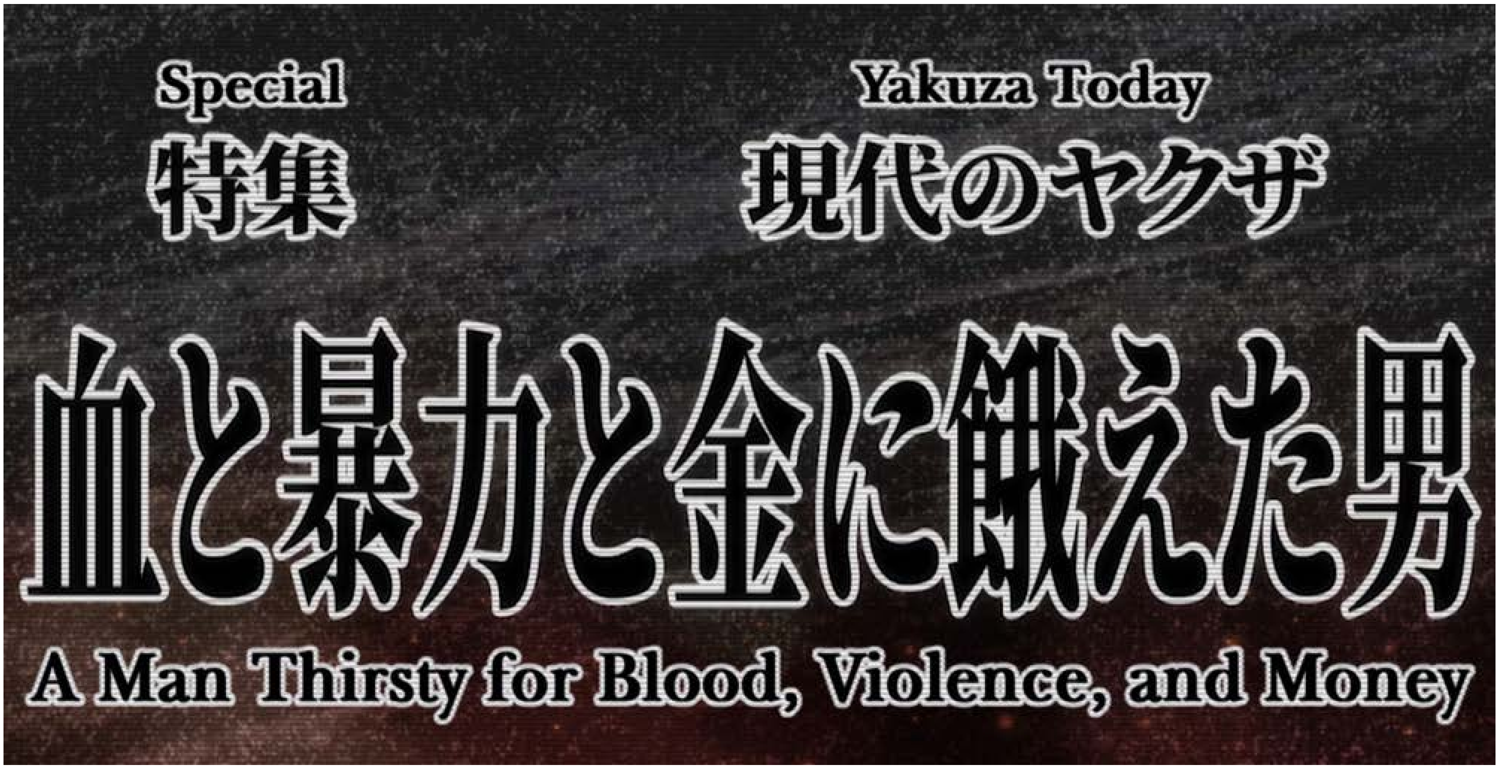
I haven't really seen much of this outside of Acchi Kocchi, but this is the case where the sign zooms out with the first few frames zooming really quickly.
Usually about 4-5 frames go from almost full screen to almost the final size, and then it goes slowly from there.
Trying to mocha the first 4 frames can be near impossible and certainly frustrating, so I usually mocha from wherever it seems reasonable.
After applying the mocha data I do the first 4 frames by hand - or if I feel like mocha could handle it, I track it all and fix the first few by hand.
Mocha may handle the size and position fine, but at least the blur needs to be adjusted manually on the first frames.
Then, after all the work you've done... nobody will notice those few frames. Welcome to the world of typesetting.
Then again if they notice nothing, it's a lot better than if they notice how much you suck.



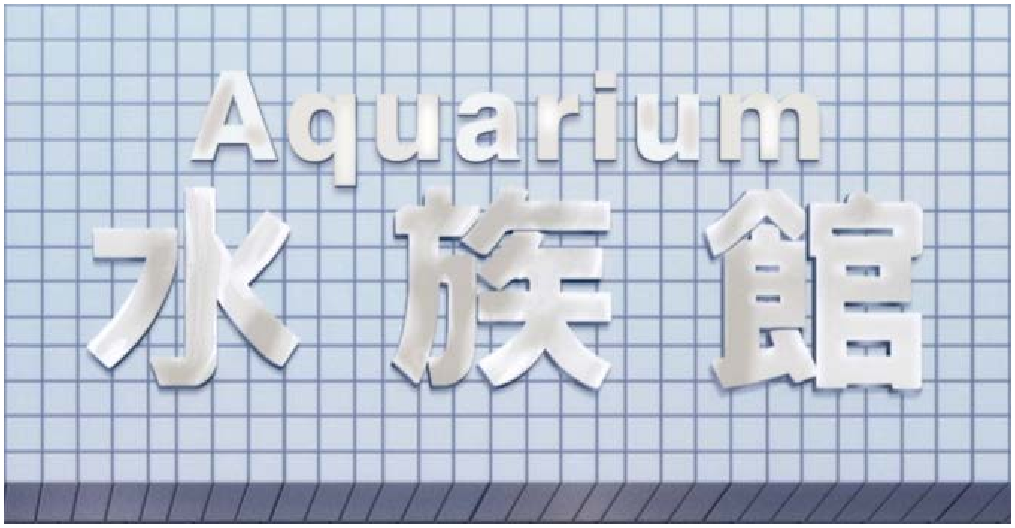
This is like 22 lines in the script [6 of them for the topmost typeset + layers for most others]. Thankfully this was at least static.
But since it is static, you can play with the details without worrying about lag.
It's a lot of stuff in one frame, but this is actually low-stress typesetting when you know you don't have to make everything move.
You can pretty easily make this look good just with the basics - fonts, sizes, colours... some rotations and that fucking ***blur!***
Your main goal here is to make it all blend in. Like if you show it randomly to somebody who doesn't know what's going on,
they shouldn't immediately go "LOL, who put these horrible captions over this?"



Layers, borders, blur, frz, fax, semitransparency on some layers... and a pretty good fade to white.



This is surprisingly easy with Clone Clip from Significance. You make one 2-pixel stripe with a clip and clone it however many times is needed to cover the height of the letters. Only 3 lines/layers are needed for this.



You won't find a font that will give you the grey/white combination like this.
So I set a slightly different colour for each letter and then used a number of circle masks with different colours resized in various ways on top of the letters.

« Back to Typesetting Main

Typesetting Faster

A few tips for how to work faster when you have a lot of signs [SHAFT shows, Acchi Kocchi etc.].

There are usually 2 ways you'll get your signs for typesetting: either in an already timed script [which may be timed to your raw by the timer or timed to CR raw by CR] or on the pad.

If it's CR, you start at the fine timing part. If it's on a pad, you're starting from scratch.
Signs on the pad usually look something like this:

{3:33} When he gets here...
{3:37} I'll bite his head off.
{3;45} Commence the Attack!
{3:45} Second Victim

If there are 80+ signs, it's really good to have a working system. From experience I can tell you that after 8 hours it gets pretty tedious and tiring. The most I did in one go was 12 hours and the last 3-4 were pretty... meh.
So here's what I do.

1. copypaste signs into Aegi

2. round 1 - rough timing

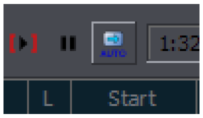
This has fortunately become easier with Aegi 3.0. Select the first line. Timecode says 3:33. Click in the Start Time field and type 3:33. Hit Enter.
In 2.1.9 this didn't do shit without also setting End Time. In 3.0 it not only does set the time [for both start/end] but also jumps to next line.
Look at next timecode [right under the Start Time field where you're typing], type it, hit Enter.
This will time all signs to one frame based on the timecodes from the pad. Good enough for rough timing and it's **fast**.
Makes no difference that it's only timed to 1 frame because you need to fine time it anyway.

Update: now you can forget about that and just use **Time signs**, which gets this round down to 2 seconds.

3. round 2 - fine timing

This is unfortunatelly a lot slower since you need to find the exact start/end frames. Basically you click on each line, left arrow back to find the start, set start, right arrow to the end, set end.
Probably the only thing that helps you here is that many signs start/end on keyframes. If you can see on the audio track that it's likely to be the case, or you know from previous episodes which signs tend to be like that, you can just click on the yellow icon [between the ones for start/end frame and the one for shifting] which will set start to previous keyframe and end to next keyframe. [The "Time Signs" script will also snap to nearby keyframes.]
Other than that this is pretty slow, but again, at least much better in 3.0 because of much faster seeking.
While encoding keyframes is usually something timers do, it's useful to have keyframes for timing signs as well, especially when you have a lot of signs, since many if not most signs start/end at keyframes.

It can be useful to disable autoseek:



so that the video doesn't keep jumping to the first frame of the sign.
I actually toggle this on/off pretty often, depending on what I'm doing.

It is also useful to hotkey 3 things: go to start time, go to end time, and that 'snap start/end to keyframes' thing.
If you set hotkeys under Subtitle Grid and Video, you can use simple keys without Ctrl/Alt/Shift, so for example numpad keys are useful for this.

4. add blur

Use a **script** to add blur to all lines. You're gonna need it on all lines anyway and doing it one by one steals your time. This takes 2 seconds and makes sure you don't forget to add it to any line.
At this point you may wanna get rid of those comments with timecodes and stuff - {TS 5:35}.
For that, and other things, you can use **this script**.
2015 Note: **Time signs** can now add blur and nuke the comments automatically.

5. round 3 - set colours

Next round of going through the signs you use the eyedropper and set correct primary/outline[/shadow] colours for each line.
While you could do other tasks along with that, I recommend doing this alone as it involves just mouse clicking and no typing so it can go pretty fast like that.
2015 Note: I don't really do that anymore, but it's up to you.

6. round 4 - typesetting

At this point it's probably no use splitting the tasks any more.
For the easy signs, you pick a style [that includes a font], set font size [or just using the scaling tool as it's faster than typing numbers], set border size, set position, use **Blur and Glow** in most cases, and you're pretty much done. These days you can use scripts for just about any task, so do it.
You may want to do all mocha tracking in one batch or leave other specific kinds of signs last or do them first. I actually prefer to go sign by sign with each of them being different, as I get bored with too much repetition.
The one reason to do mocha tracking last is that as it adds hundreds of lines, it becomes hard to find anything.

7. round 5 - checking

When you're done with everything [and have sorted the script by time], you seriously need to make one more round, click on each sign, go one frame back to make sure it doesn't appear too late, right arrow to the end to make sure it doesn't end late or early and that nothing breaks in between [mocha, layers, movement start/end times, \t tags, nobody walks in front of the sign etc.]. **This script** can help you do it faster.

Use automation scripts.

There are now scripts for almost anything you can think of, so you pretty much don't need to type any tags.
Get familiar with all the scripts that are available.

Know your keyboard shortcuts, like ctrl+D for frame-by-framing by hand. Know that in Aegi 3.0 you can drag multiple signs on the screen at once, therefore you can frame-by-frame 5 signs on the screen all at once quite easily.
The other tools can be used for multiple signs at once too - rotation, scaling.

Bind your automation scripts to hotkeys.

This will improve your speed **a lot**. Go to Preferences-Interface-Hotkeys. I prefer to set the shortcuts under "Default," but be aware that these will work even when typing in the Subtitle Edit Box, so don't make shortcuts that you might actually need for typing, like Shift+letter. If you want such shortcuts, instead of Default put them under Subtitle Grid, since that's where you'll usually be. Add a hotkey you want, and under "Command" type "auto" and aegi will autocomplete and give you a list of the available scripts, so just choose the right one from there. With Ctrl+letter, Alt+letter, or any combinations of Ctrl/Alt/Shift+letter you have tons of options, so you can usually assign scripts to logical letters, like Alt+B for Bold, Alt+I for Italics, Alt+8 for adding \an8, etc. For many scripts I just set a single letter under Subtitle Grid and Video.

Modify your scripts.

If you see you're doing a specific task often, and there's theoretically a way to make it faster or do it in fewer steps, modify an automation script or **write your own**. If you're using my scripts and think something could be improved about them, let me know, and I'll upgrade the script if it's reasonably doable.

If you have a sign that repeats itself every episode [titles and such] that needs mocha tracking, don't track it every time but copy from previous ep and shift.
Even titles that don't move but just have a lot of tags are faster to copy from last ep and shift than redoing them every time [actually I hope nobody's doing that].
In fact, what I usually do is open last ep's TS file, save under new episode name, and delete all the non-repeating stuff, thus leaving titles, eyecatches, and whatever else repeats itself every episode. Then I shift the signs to where they should be this episode and see if they need any modifications. Eyecatches tend to change position on the screen, ep titles may change colours, fades, etc.

If you're using a lot of styles, it might be good to name them so that you can easily tell them apart, and not just sign1, sign 2 etc.
You can use the actor field for some notes regarding what you need to do later so you don't forget. For example I sometimes type "mocha" in there to make sure I don't forget that I need to track that sign later.

I hope you know Aegi has 3 modes of displaying tags [the last icon in the top toolbar cycles through them] and that you're using the one that shows whole tags in the script.

It also improves your speed if you don't check IRC every 2 minutes, by the way.

« Back to Typesetting Main

Scripts for Aegisub

A bunch of scripts that don't come with aegisub by default:

» **All of my own scripts can be found here. (GitHub repository here.)**

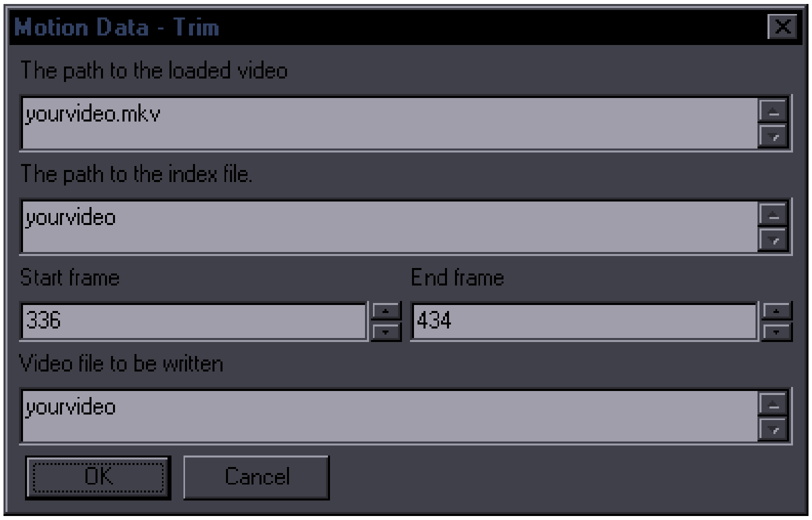
» **Here you can find links to scripts from lyger, torque, line0, and others**

Aegisub-Motion | Latest version here

Script for motion tracking with Mocha.

This is roughly what things should look like.
Set the folders in the config (below) as implied.

The 'Trim' window doesn't exist anymore. ->

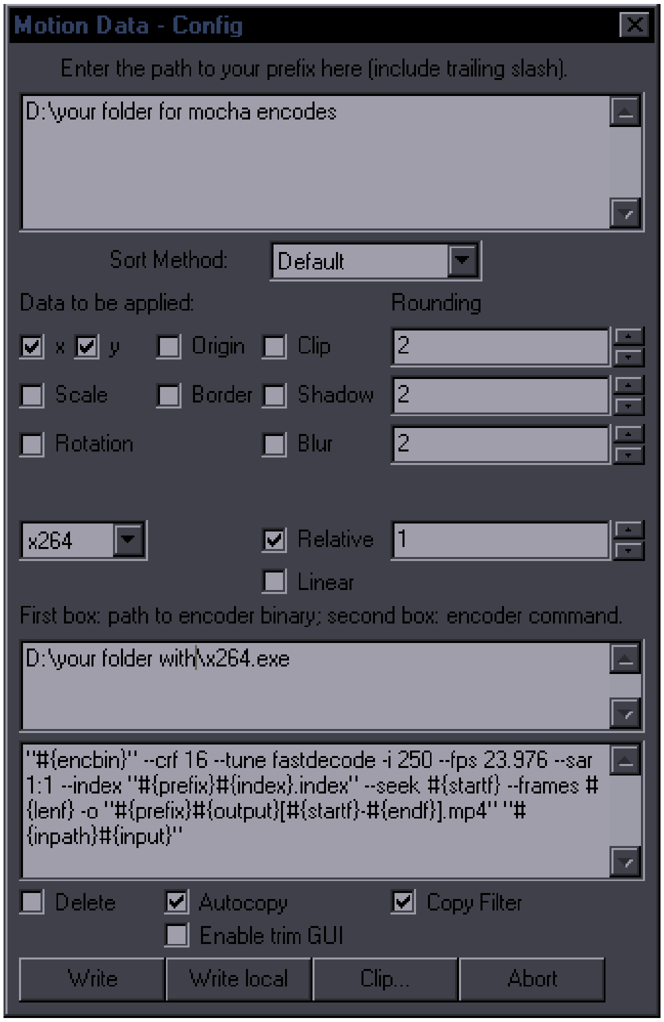


The path to the loaded video: yourvideo.mkv

The path to the index file.: yourvideo

Start frame: 336 | End frame: 434

Video file to be written: yourvideo



Enter the path to your prefix here (include trailing slash).: D:\your folder for mocha encodes

Sort Method: Default

Data to be applied: ☑ x ☑ y □ Origin □ Clip — Rounding: 2
□ Scale □ Border □ Shadow — 2
□ Rotation □ Blur — 2

x264 — ☑ Relative — 1
□ Linear

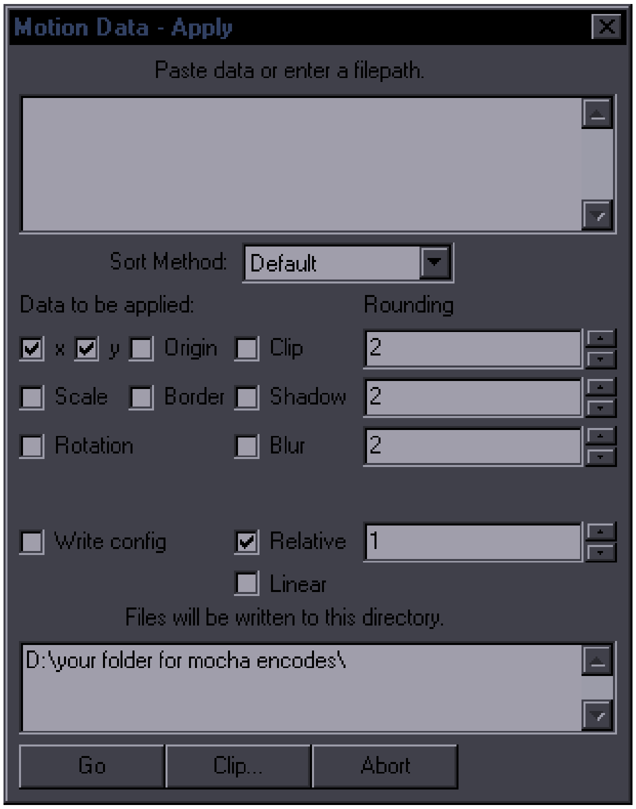
First box: path to encoder binary; second box: encoder command.

D:\your folder with\x264.exe

```
"#{encbin}" --crf 16 --tune fastdecode -i 250 --fps 23.976 --sar 1:1 --index "#{prefix}#{index}.index" --seek #{startf} --frames #{lenf} -o "#{prefix}#{output}[#{startf}-#{endf}].mp4" "#{inpath}#{input}"
```

□ Delete ☑ Autocopy ☑ Copy Filter
□ Enable trim GUI

If you copy data from Mocha, they will be pasted automatically in the top part of the 'Apply' dialogue. ->



Paste data or enter a filepath.

Sort Method: Default

Data to be applied: ☑ x ☑ y □ Origin □ Clip — Rounding: 2
□ Scale □ Border □ Shadow — 2
□ Rotation □ Blur — 2

□ Write config ☑ Relative — 1
□ Linear

Files will be written to this directory.: D:\your folder for mocha encodes\

Duplicate and shift by 1 Frame backwards

If you don't know what this is, then you don't need it.

Bezier

Aligns text along a bezier curve.
Won't do shit if you don't read the instructions in the script, so please...

BT.601 -> BT.709 Colour Converter

BT.709 -> BT.601 Colour Converter

Daiz's scripts to fix the mess he created in the first place.
(This is not needed in newer builds that do it with resample dialog.)

How to use scripts

Just place the lua in Aegisub's automation/autoload folder. It will load when you start Aegisub unless it's somehow corrupt.
If you modify the script with Aegisub open, go to "Automation -> Automation" in the menu and click "Rescan Autoload Dir."
That will reload all scripts in the autoload directory.

If you download a script and it's not loading in Aegisub, it might be because you're using Chrome and it's adding some html crap to it, so make sure to avoid that.

Bind your scripts to hotkeys.

Go to Preferences-Interface-Hotkeys, add a new line under Default. Set a hotkey and under "Command" type "auto".
Aegisub will show you a list of loaded automation scripts. Select the one you want.

Write your own scripts!

If you still need more functions, learn how to write lua scripts for aegisub **here**.
If you're familiar with any programming language, it will probably be easy.
If not, you can still learn to do so with this guide. [Even I did, so it can't be that hard.]

« Back to Typesetting Main

Creating Lua Automation Scripts for Aegisub

written by lyger

Introduction

(last revision: 2013-06-05)

This tutorial is meant to serve as a basic guide to creating Lua automations for use in Aegisub. If you�ve worked on an advanced substation alpha script, especially if you�re a typesetter, you�ve probably encountered tasks that are tedious, repetitive, and/or require more calculations than you�re willing to do. In most cases, these tasks can be automated using macros written in Lua code, and often quite easily.

This tutorial is based on my own knowledge and experience. There are many features of the Lua language and the Aegisub API that I've never used and won't cover. This tutorial should provide a solid starting point, but advanced users will have to do their own research.

The next section will cover basic programming concepts for people who have never programmed before. If you already know a programming language (HTML doesn�t count), you can skip to "Lua for Experienced Programmers". If you already know Lua, or if you'd rather start writing macros right away and learn Lua as you go, then you can skip to "The Aegisub Environment".

I recommend frequently referencing the documentation provided in the Aegisub user manual.

Programming for Beginners

This section will briefly introduce basic programming concepts using Lua. On the advice of some people who helped proofread this tutorial, I've vastly condensed this section, as it is not the main focus. You can find many good resources for learning Lua on the lua-users.org page.

Variables, data types, arithmetic, and logic

A variable is the name of the place where you store data. They behave similarly to the variables in algebra, but can store any kind of data, not just numbers.

```
x=5
y=12
z=x+y
```

Another simple data type in addition to numbers is the **string**, which represents text and is defined with quotes. In Lua, you can join one string to another using the `..` operator.

```
first_name="John"
last_name="Smith"
full_name=first_name.." "..last_name
message="Hello world!"
```

Note that the backslash (`\`) is an "escape character" (more info here). If you want to type a normal backslash, type "`\\`".

Another data type is the **Boolean**, which can only have two values: `true` or `false`. You can use Booleans to evaluate `if` conditions or loops.

```
is_happy=true
if is_happy then
    print("I�m happy too!")
end
```

In most cases, you will be using Boolean *expressions* that can be `true` or `false` depending on the situation. The following are examples of boolean expressions.

```
x > y
number <= 0
count == 5
command ~= "quit"
x < 0 or x > 10
x > 0 and y > x
```

The greater than and less than sign mean what you expect. "`<=`" and "`>=`" mean "less than or equal to" and "greater than or equal to", respectively. "`~=`" means "not equal to". Conditions can be combined using `and` and `or`.

Note the double equals sign. In many programming languages, including Lua, the single equals sign does not mean "is equal to". Instead, the single equals sign represents assignment. That is, it is a command that means "store this value in this variable". The expression `count = 5` means "store the value 5 in count", which is not what we want here. To check for equality, use `count == 5`, which means "count is equal to 5".

If you are working with number variables, you can perform arithmetic on them normally. In Lua, multiplication is `*`, division is `/`, and exponentiation is `^`. The modulo operation is `%`.

```
x=21
y=3*x+46
z=((y-x)/(y+x))^2
x=15+y
```

Keeping in mind that the equals sign is the *assignment* operator, note that at runtime, the expressions on the right are evaluated and their numeric results are stored in the variables. Changing the value of `x` on line 4 does not affect `y` or `z`.

Lua has a special value called `nil` that can be stored in any variable. It means "nothing". This is not the same as the number 0, nor is it the same as a string with no text in it. If a variable is `nil`, it means "no valid data is stored in this variable". Uninitiated variables and failed function calls will generally result in `nil`.

You might think having `nil` values is an error that should be avoided, but `nil` values can be very useful. For example, you can see whether `tonumber(foo)` returns `nil` (i.e. "failed to covert to number") to determine whether `foo` is a valid number or not.

The last thing I have to say about variables concerns naming them. A variable name can contain letters, numbers, and the underscore, but no spaces. The first character in a variable name must be a letter or the underscore. Also, you cannot use any Lua keywords as variable names. For example, you cannot name your variables `end`, `if`, `for`, or `in`, because all of these serve other purposes in Lua.

Control structures and loops

Now we can start doing some more complicated tasks than arithmetic. I already briefly touched on the `if` statement. An `if` statement checks whether a condition is true, and if it is, it performs all of the code until `end`. Otherwise, it skips the code and continues after the `end`.

```
x=5
y=10
z=0
if x<y then
    z=x*y
    print("Foobar")
end
print("The value of z is "..z)
```

Here, because `x` is less than `y`, the code inside the `if` statement executes. `z` is set to 50, and "Foobar" is printed out to the console. Afterwards, "The value of z is 50" is printed to the console as well.

You can extend an `if` statement using `elseif` and `else`.

```
if command=="quit" then
    print("Goodbye!")
    return
elseif command=="say hi" then
    print("Hello, "..name.."!")
else
    print("I didn't understand your command.")
end
```

Next up are loops. Loops will execute the code inside them over and over until some condition is met. Two basic loops are `while... do` and `repeat... until`. These should be self-explanatory, so I'll just show some examples and move on.

```
x=100
y=0
while x>0 do
    x=x-1
    y=y+x
end
print("The value of y is "..y)
```

```
repeat
    print("Say quit to exit this program")
    user_input=io.read()
until user_input=="quit"
print("Goodbye!")
```

One very important kind of loop, and one that many beginners find hard to understand, is the `for` loop. The `for` loop will cycle a variable or variables through a set of values, and execute the code in the loop once each time. The simplest use of a `for` loop is as a counter, to cycle a variable from a starting value to an ending value. For example, to count from 1 to 10, we simply do:

```
for x=1,10,1 do
    print(x)
end
```

This `for` statement says "define a variable named `x`, which will start at 1, end at 10, and count by 1." The third number is optional; if you put `for x=1,10 do` Lua will count by 1 by default. If you want to count by 2, then use `for x=0,10,2 do`, and if you want to count down, use a negative number. The commands inside the `for` statement (in this case, a `print`) will be executed for each value of `x`.

As a simple example, let's use `for` to calculate the factorial of 10.

```
result=1
for num=1,10 do
    result=result*num
end
print("10! = "..result)
```

The variable `num` stores values from 1 to 10, and the code inside the loop is executed once for each of those numbers, thus multiplying `result` by all the numbers from 1 through 10.

Arrays and other data structures

So far, we've covered three simple data types: numbers, strings, and Booleans. Now we can move on to compound data types, the most basic of which is the **array**, which exists in all programming languages. An array is an ordered list of values. Because it's ordered, each value has an associated index: 1 for the first value in the list, 2 for the second value, and so on. You use this value to access specific elements in the array.

```
my_array={8, 6, 7, 5, 3, 0, 9}
print(my_array[3])
```

In the above code, `my_array[3]` refers to the third value in the array. 7 is at the third position in the list, so the output of this code is 7.

We can use a `for` loop to perform operations on every item in an array. By placing a `#` in front of the name of the array, we can get the size of the array (the number of items in the list). We can use this as the upper bound for our `for` loop.

```
my_array={8, 6, 7, 5, 3, 0, 9}
for i=1,#my_array do
    print("The number at index "..i.." is "..my_array[i])
end
```

Since arrays are used so often in Lua (they're actually a part of a bigger data type, known as **tables**), a special function is provided that lets you cycle through an array more easily. This is the `ipairs` function. The `i` presumably stands for indexed, meaning this function gives the elements of an array in order based on their index. The "pairs" is because it returns index-value pairs. You use `ipairs` like this:

```
my_array={"Never", "gonna", "give", "you", "up"}
for index, value in ipairs(my_array) do
    print("The string at index "..index.." is "..value)
end
```

Any data type can be stored in an array: numbers, strings, and even other arrays. If you have an array of arrays, then you can, for example, access the third element in the second array using `array[2][3]`. Unlike many programming languages, Lua allows you to have different kinds of values stored in the same array. You can thus have a list of mixed numbers, strings, booleans, and tables.

The other main compound data type in Lua is the hash table (again, this is actually a part of the table data type; more on this later). Instead of storing an ordered list of values, a hash table can be thought of as a dictionary or a lookup table. A hash table stores pairs of "keys" and associated "values". You use the key to look up the corresponding value in the table. The key is usually a string.

The example below shows how to define a hash table to store data on a person.

```
 person={
    fname="John",
    lname="Smith",
    age=17,
    favfood="pizza"
}
```

To access the person's first name, simply use `person["fname"]`. Lua also allows you to access the data in a way that looks more "object-oriented" (if you don't know what this is, don't worry about it). The code `person.fname` means the exact same thing as `person["fname"]`.

Note that you cannot compare arrays and hash tables using == and other similar operators. Well, you can, but it probably won't do what you want it to.

An important caveat! When you want to make a copy of a simple data type, you can simply create a new variable and set it equal to the old one. This doesn't work for compound data types like tables, because the variable is a reference to the table data, and not the data itself.

Aegisub comes with a function to copy tables. See the "Utils" section for more details.

Functions

Functions are similar to the functions you learn about in pre-algebra. You give the function some parameters, it does some work on the parameters, and it usually gives you a result. If you have a function like f(x)=4x+2 and you ask for the value of f(6), then 6 is your "parameter" or "argument" and 26 is your result or, in programming terms, your "return value". You can have a function with multiple parameters, such as g(x,y)=x+y-4.

The below code defines a function that acts like g(x,y) above.

```
function add_and_minus_four(num1, num2)
    result=num1+num2-4
    return result
end
```

This function takes two parameters, does some math using them, and `return`s the result to us. You can use this function elsewhere in the program like this:

```
x=12
y=16
z=add_and_minus_four(x,y)
print(z)
```

The resulting value of the function, 24, will be stored in `z`. For reference, the above code can be written in one line as:

```
print(add_and_minus_four(12,15))
```

A function does not need to have a `return` value, because unlike math functions, a function in a program can do tasks other than calculating a result. Functions also need not have any parameters, but when you use the function, you *always* need the parentheses, even if there is nothing in them.

Also, unlike math functions, a Lua function can return multiple values, separated by commas. To return multiple values, just write `return value1, value2, ...` at the end of your function. Then, when you use the function, store the multiple returns in variables separated by commas: `x1, x2, ... = ...`.

It's important to note that a function immediately ends once it gets to a `return` statement. Any code after the `return` statement will not execute. Thus, if you have a `return` inside an `if` statement, there is no need to put an `else`.

```
function greater_of_two(num1, num2)
    if num1 > num2 then
        return num1
    end
    return num2
end
```

Lua for Experienced Programmers

This section will give a quick overview of what you need to know about Lua if you already have programming experience. If you're a beginner programmer who has just finished reading the previous section, I still recommend reading this section in full, especially the "Useful libraries" section. You might not recognize all of the terms, but there's a lot of essential information. I'll also repeat a lot of what I covered in the previous section, but with higher-level explanation.

Writing Lua code

First things first: what program should you use to write Lua code? Personally, I use Notepad++, which comes with syntax highlighting for Lua and many other languages. If you download Lua for Windows, it comes with an editor specifically designed for Lua. Otherwise, I'm sure you can just google "Lua editors" and find something to your liking.

Keep in mind that macros you write will only run in the Aegisub environment and can only be tested within Aegisub. However, you can test out code snippets outside of Aegisub so long as they don't use to the Aegisub API.

Basic syntax and features

Lua is a lightweight, weakly typed, interpreted (sort of) programming language designed as a scripting language (yes, this was paraphrased from Wikipedia). You can define variables on the fly without having to specify what data type they store, and the same variable can be used to store any kind of data.

There are only three simple data types (that I can think of): number, string, and Boolean. All numbers are double-precision floating point numbers.

Lua arrays start counting at 1, not 0, so be careful. The size operator for arrays and strings is the prefix `#`.

The end of a control structure or function's scope is marked by the `end` keyword. There is no `begin` keyword; the function definition or the head of the control structure serves to begin the scope. An exception is `repeat... until`, where the two keywords define the scope and no end is needed.

Lua supports the `local` keyword for variables and functions, if you need to control the scope. Generally you don't need to worry about this if there's no chance of confusion, but it's still good practice. Variables not declared `local` are global.

Uninitiated variables or array entries are set to `nil`. This is more or less equivalent to the `null` value in Java. Failed function calls often return `nil`. This can be used, for example, to detect if a string does *not* contain a certain substring: `string.match(str, substr) == nil` (incidentally, = is assignment and == is equality).

Lua supports multiple return values and arbitrary numbers of function parameters. If you pass a function more parameters than it uses, the extra parameters are ignored. If you pass the function too few parameters, the remaining parameters are `nil`. A function can receive an arbitrary number of parameters using `...` (look it up).

Lua does not have a full regular expression implementation. Lua uses a bare-bones kind of regex known as "patterns". This tutorial and Google should teach you all you need to know. Patterns will come up again later when I introduce the string library. For strings, the escape character is the backslash (`\`). Remember to escape slashes when writing .ass override tags.

In Lua, functions are "first-class citizens". That means you can toss around a function as if it were a variable. Pass a function as a parameter, store a function in an array�go wild.

Boolean operators are `and` and `or`. "Not equals to" is `~=`. String concatenation is `..`. You don't need semicolons at the ends of lines, and superfluous whitespace is ignored.

Comments are defined as follows:

```
--This is a single line comment
--[[
This is a
multi-line
comment
]]
```

The table data structure

The **table** is the only native compound data structure in Lua. It can be used like an array, a hash table, or both. Multiple data types can be stored in the same table. Fundamentally, a table is a collection of key-value pairs. If the keys are all numbers, then the table behaves like an array. If the keys are all strings, then the table behaves like a hash table. If you have both at the same time, then welcome to the joys of Lua tables!

The iterator function `pairs` returns all the key-value pairs in a table, in both the array part and the hash table part. The `pairs` function does not guarantee any order to the values it returns, not even for the array part of the table. You use the function like this:

```
for key, value in pairs(my_table) do
    ...
end
```

You might be familiar with "for each" loops in other programming languages; this is basically that. If you want to iterate through only the array part of the table, in order, then replace `pairs` with `ipairs`.

Object-oriented Lua

As you may have realized, tables are pretty similar to objects. Indeed, Lua allows you to extend tables to use them like objects using **metatables**. I've never used them personally, but if you are interested, you can find more information online.

But insofar as a table can be a hash table, the keys are a lot like an object's fields. Defining a table can look a lot like defining an object; scroll up to see an example. Lua even allows you to write `my_table["key1"]` as `my_table.key1`.

Useful libraries

In addition to the `pairs` and `ipairs` functions, Lua also provides the useful `tonumber` and `tostring` functions.

The `table` library is worth looking into, but the only functions from it that I regularly use are `table.insert` and `table.remove`.

The string library

I could just link to the string library tutorial and the patterns tutorial on the Lua users wiki and call it a day, and honestly, you should find most of what you need on there. But I'll go through it in a bit more detail anyway.

First, it's worth noting that you can call string library functions on strings in an object-oriented-like way. For example, if you have a string named `str` and a pattern named `pat` that you want to match, you can use either `string.match(str, pat)` *or* `str:match(pat)`. This is a bit shorter and somewhat more intuitive, especially if you're used to object-oriented.

The functions I use most frequently when writing Aegisub automations are `string.match`, `string.format`, `string.gmatch`, and `string.gsub`. I used to use `string.find`, but I found that in most cases, `string.match` is a better option.

These functions are, in general, pretty well explained on the Lua users wiki. Note that `string.match` returns the section of the string that matches the pattern, or the captures if the pattern contained any (these are returned as multiple return values). If you've used format strings in C before, `string.format` is basically the same thing. You can often get the same functionality just by concatenating strings (since numbers are automatically converted to string when concatenated), but it's often neater and more convenient to use `string.format`.

The real workhorse, though, is `string.gsub`. This is the bread and butter of most automation scripts that I've written, because most Aegisub automation involves modifying the text of your subtitle script. There's no better or more versatile way to modify text in Lua than `string.gsub`. Its many capabilities can be overwhelming for some, so I've written an example script that should walk you through what you can do with it.

You can download the example script here.

The math library

Here's the math library tutorial. There's not much I can add to this. The math functions you'll use depend heavily on the sort of automation you're writing, so it's best to look them up as you need them. However, I will mention a couple things.

First off, be aware that *all trig functions in the math library use radians!* I cannot stress this enough. As you're probably aware, advanced substation alpha angles are always in degrees, so if you want to do any math involving angles, it is *imperative* that you use `math.rad` and `math.deg` to convert from degrees to radians and vice versa. Many a time I have been stymied by misbehaving code, only to realize I'd forgotten to convert. Also note that the math library includes the constant `math.pi`.

Another thing to note is Lua's pseudorandom number generator. The seed is always the same, so if you run your automation multiple times, `math.random` will produce the exact same sequence of pseudorandom numbers. If you want to get a different pseudorandom number sequence, use `math.randomseed` to seed the random number generator with a different number. A good solution is to use your constantly-changing system time as a seed: `math.randomseed(os.time())`. This will produce a new sequence of numbers each time you run the automation... so long as you wait a second. Sadly, Lua doesn't do milliseconds.

The Aegisub Environment

Finally, it's time to see how we can put Lua to work in Aegisub's automation environment. All this information and more is on the official user manual on the Aegisub website, but I'll be presenting it here step-by-step, with explanations and examples. Nonetheless, I strongly encourage you to read the manual thoroughly on your own time.

Writing a macro function

The most basic skeleton of an automation script should look something like this:

```
--[[
README:

Put some explanation about your macro at the top! People should know what it does and how to use it.
]]

--Define some properties about your script
script_name="Name of script"
script_description="What it does"
script_author="You"
script_version="1.0" --To make sure you and your users know which version is newest

--This is the main processing function that modifies the subtitles
function macro_function(subtitle, selected, active)
    --Code your function here
    aegisub.set_undo_point(script_name) --Automatic in 3.0 and above, but do it anyway
    return selected --This will preserve your selection (explanation below)
end
```

```
--This optional function lets you prevent the user from running the macro on bad input
function macro_validation(subtitle, selected, active)
    --Check if the user has selected valid lines
    --If so, return true. Otherwise, return false
    return true
end

--This is what puts your automation in Aegisub's automation list
aegisub.register_macro(script_name,script_description,macro_function,macro_validation)
```

To view this skeleton script with syntax highlighting, you can download it here and open it up in your Lua editor of choice.

When you run the automation from Aegisub, it will call `macro_function` (or whatever you choose to name your function) and execute the contents of the function. The function is given three parameters: the subtitle object, a list of selected lines, and the line number of the currently active line. Most scripts only use the first two. If that's the case, feel free to not include the `active` parameter. Also, to save typing, I usually abbreviate the parameter names to `sub, sel, act`. You can name the parameters whatever you want. Much like in math, f(x)=2x+3 is exactly the same as f(y)=2y+3.

For convenience, I will use my preferred variable names `sub, sel, act` to stand in for the three parameters in code examples.

The subtitles object is the only object that contains data about your .ass script. It's a table, but it's a pretty special one. The array part of the table stores all the header, style, and dialogue lines in the script. To get line 5, you simply type `sub[5]`.

Now this is important: if you want to modify lines, you *never modify the subtitles object directly*. Not only is it a pain to type, I don't think it actually works (like I said, the subtitles object is a special table). You have to first read out the line using something like `line=sub[line_number]`. Then you modify the line, and put it back in the subtitles object with `sub[line_number]=line`. Incidentally, these lines that you read out are line data tables, which I'll cover later.

I'm going to be honest: to this day, the subtitles object is something of a black box to me. It's pretty complicated, but all you really need to know is how to retrieve dialogue lines. Aegisub comes with some very useful functions that do the rest of the work for you (I'll cover them in the section on karaskel).

Next up is the list of selected lines, which I usually call `sel`. To reiterate: only the subtitles object contains data about the script. You won't find the selected lines in `sel`. Instead, you'll find a list of the *line numbers* of the selected lines. In other words, `sel` is a list of indices for the subtitles object. To get the first selected line, you have to write `sub[sel[1]]`. This can be a bit confusing, especially if you use `ipairs` on `sel`, as I will show you how to do later. You'll end up with two indices: one that refers to a position in `sel` and one that refers to a position in `sub`. Don't get confused.

The final parameter is the line number of the currently active line. I've never used it, but you can access the relevant line using `sub[act]`.

Use these parameters to do all sorts of fancy stuff to the subtitles in the body of the function. At the end, you should set an undo point. Aegisub 3.0 now automatically does this even if you forget, but it's good practice to do so anyway. The return value of the function is optional, but if you return an array of line numbers, then the selection will be set to those lines once the macro finishes running.

With this in mind, here's a skeleton for a processing function that will allow you to modify every line in the user's selection.

```
function do_for_selected(sub, sel, act)
    --Keep in mind that si is the index in sel, while li is the line number in sub
    for si,li in ipairs(sel) do
        --Read in the line
        line = sub[li]

        --Do stuff to line here

        --Put the line back in the subtitles
        sub[li] = line
    end
    aegisub.set_undo_point(script_name)
    return sel
end
```

Download this skeleton here.

The dialogue line table

Read this. Just do it. Be aware that without using karaskel, you only have access to the "basic fields".

The dialogue line data table stores all the information you will need about a line in your script, all in a nice, convenient table. You can access the actor and effect fields, you can get the start and end time in milliseconds, you can tell whether the line has been commented out (and you can change a line to a comment or vice versa), and most importantly, you can access and modify the text of a line.

There are a plethora of useful macross that you can write using nothing but the skeletons I provided above, the string library, the math library, and `line.text`. If you just want to take care of some simple text modifications such as adding italics, or even some basic math like shifting position, then you have all you need to know. You can stop reading after the following two example scripts.

For those who want to get the most out of the Aegisub automation API, continue on to the advanced section.

Guided Example: Add Italics to Selected Lines

[Download example script]

This is one of the simplest examples I can provide. I've only added one line of code (line 25) to the skeleton scripts I've provided above.

In this script, I use the string concatenation operator to add an italics tag to the start of each selected line's text. Note that there are two backslashes. As mentioned in the variables section, the backslash is a special character. To type an actual backslash, we need to "escape" it.

When using the skeleton scripts, don't forget to fully delete pieces of code that you are not using, and make sure you reflect name changes across the entire script. Here I've named the processing function "italicize", so I also have to use "italicize" when I register the macro. Furthermore, I did not need a validation function since this script can be run on any selection of lines, so I deleted the validation function and removed it from the macro registration at the bottom.

Finally, remember to set your script properties at the top, and write a readme.

Guided Example: Modify Font Size for Mocha

[Download example script]

Nowadays xy-vsfilter, which supports non-whole-number scaling, is becoming standard, so this trick is no longer quite as useful. Nonetheless, it makes for a good example automation script that involves both the string library and the math library.

This script decreases the font size (\fs) by a factor of ten and increases the scaling (\fscx and \fscy) by a factor of ten. At the end, the size of the text still looks the same, but the scaling is ten times more precise. For font sizes that are not evenly divisible by ten, the script will round down to the nearest whole number. Then it divides the original font size by the new font size to determine how much it needs to increase \fscx and \fscy to balance it out.

All this math takes place in lines 40 to 48 of the example script, so you can see it for yourself. `math.floor` is the function that rounds down to the nearest whole number (the opposite is `math.ceil`).

The script makes use of `string.gsub` with an anonymous function. If you're having trouble with understanding this use of `string.gsub`, you can check out the example script I wrote, or the string library tutorial on the Lua users wiki. The pattern `"\\fs(%d+)"` looks for the string "\fs" followed by a number, and the parentheses around `%d+` will "capture" the number and send it to the anonymous function (another reminder that you have to escape backslashes).

The script also shows how to use `string.format` to insert values into a format string. The formatted string is then returned by the function, and substituted into the original string.

Unlike the previous script, this one has a validation function that makes sure the selected lines all contain "\fs". Note that I do not read out a line table, but directly access the subtitles object. Since I'm not going to be modifying the lines, I'm not going to bother reading the line into a full line data table. Validation functions run every time you open the automations menu, so they should be as short and as fast as possible.

The Aegisub Environment - Advanced

Before you go off writing your own functions to do useful subroutines, make sure that you're not reinventing the wheel. Aegisub comes with two libraries that will vastly extend the capabilities of your automation scripts and make common tasks much easier. Later in this section, I'll also introduce the API for creating simple GUIs that allow the user to configure the behavior of the automation script.

karaskel.lua

The full documentation for this library can be found here. To use karaskel functions in your script, put this at the top:

```
include("karaskel.lua")
```

Remember all the extra fields that you might have seen when reading about the dialogue line data tables? Well, karaskel will give you access to all of that. You'll also be able to access style data about a line, so you can detect its style defaults, which is huge if you're writing an advanced script.

Since my scripts are typesetting-focused, I only use two functions from karaskel. If you want to write an automation that actually deals with karaoke, then you'll probably find the numerous other features of karaskel quite useful.

The first essential function is `karaskel.collect_head`, which collects the meta and style data that other karaskel functions use. You'll need a line at the top of your processing function that looks something like this:

```
local meta, styles = karaskel.collect_head(sub,false)
```

You probably want `false`, because `true` will generate a lot of mysterious extra styles in your script. That's not where the magic happens, though. After you've read in your line to process it, you can do this:

```
karaskel.preproc_line(sub,meta,styles,line)
```

This gives you access to all the extra dialogue line fields that you saw earlier. In particular, it gives you access to `line.styleref`, which is *exciting*. Seriously. Be excited.

Because now you can do things like `line.styleref.color1` to figure out what the default main color of your line is. You can check the default font size using `line.styleref.fontsize`. Need to know the border weight? `line.styleref.outline` is your friend.

Be warned that the color codes extracted from the style are not ready to use in `\c` tags just yet. Color codes in style definitions contain both color and alpha data, and look a bit different from in-line override codes. You'll need a function from utils.lua to extract properly formatted color and alpha codes for use in override tags.

As a side note: a function that isn't part of karaskel but can be very useful is the `aegisub.text_extents` function, found in the miscellaneous APIs. You'll need to use `karaskel.collect_head` before you can use this function, which is why I mention it here. This function takes a style table and a bit of text and calculates the pixel height and width of the text. If you pass it `line.styleref`, then it will give you the size of the text in the line's default style. But you can do more.

If the user overrides the default font properties in-line, you can use the strings library to detect the changes. Now make a copy of the line's style table, modify it until it matches the user's override tags, and pass it to `aegisub.text_extents`. You can determine the size of any typeset that the user makes, even if he changes the font or font size in the line. *That* opens up a lot of possibilities.

utils.lua

The full documentation for this library can be found here. To use utils functions in your script, put this at the top:

```
include("utils.lua")
```

This library defines the essential `table.copy` function. If you want to create new lines in your subtitle script, you're going to need this function. As mentioned in an earlier section, copying a table is more involved than copying a number or a string. To create a new line, you're going to have to make a proper copy of your original line data table, and you'll need this function to do that.

In addition, utils contains lots of useful functions for working with .ass color and alpha codes. There are functions to extract colors and alphas from styles, to convert to and from RGB values and HSV values, to transition smoothly between colors and alphas, and so on. If you're stuck working with colors or alpha, odds are one of the functions here will help you out.

Creating GUIs

For those who don't already know, GUI stands for Graphical User Interface. Anything that has a pretty window with buttons you can click is a GUI, and we can create simple GUIs to allow users to set options for your automation before it is run.

The GUI documentation is here, and if you're anything like me, you're going to want an example so you can see how this thing works, because it's a little hard to grasp just reading the documentation.

Let's work backwards. The function that actually displays your dialog box is `aegisub.dialog.display`. This function takes two parameters, both of which are tables. The second table is easy enough. It's just a list of the buttons you want at the bottom of the table. Something like `{"Run", "Options", "Cancel"}` would work. The function's first return value is the button that was pressed.

The first parameter is the real meat of the GUI. This is the dialog definition table, which will describe how your GUI is laid out, what options the user has, and what their default values are. You can see all the options available to you here.

You position components using a grid of rows and columns. Imagine each component is a cell in a spreadsheet. The top left corner is (0,0). To the right of it is (1,0), and below is (0,1). If you want a component to occupy more than one cell, then set its coordinates to the top left cell, and use "width" and "height" to tell it how many columns and rows it takes up. You can make your GUI as many columns wide and as many rows tall as you want, and the window will stretch to fit�but keep it within reason.

The type of component is defined by the "class" property. You can have labels, which simply provide instruction text, you can have checkboxes, dropdowns, color selectors, text fields, and so forth. Components that take user input also need a "name" so that your automation can retrieve the results later on. Additional properties will depend on the class of the component. Checkboxes can be set to "true" or "false" by default, and can also have their own labels. Dropdowns will contain a list of the options to include in the dropdown menu, and so forth.

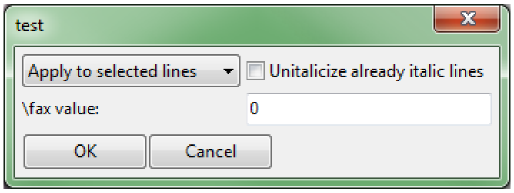
All right, enough of that. You want to see an example. So here we go.

Let's say we want to add some options to our italics script. We want a dropdown box to let the user select "Apply to selected lines" or "Apply to all lines". This will be set to "Apply to selected lines" by default. Also, perhaps the user wants to unitalicize lines that are italic already. So we should have a checkbox for "Unitalicize already italic lines", but this will be off by default. Also, maybe the normal italic isn't slanted enough, so we can provide a text box to let the user define a \fax value, to tilt the text more (humor me here). We'll need to label this text box so the user knows what it does, and let's set its default value to zero.

Here's what our dialog configuration table looks like:

```
dialog_config=
{
    {
        class="dropdown",name="lineselect",
        x=0,y=0,width=1,height=1,
        items={"Apply to selected lines","Apply to all lines"},
        value="Apply to selected lines"
    },
    {
        class="checkbox",name="unitalic",
        x=1,y=0,width=1,height=1,
        label="Unitalicize already italic lines",
        value=false
    },
    {
        class="label",
        x=0,y=1,width=1,height=1,
        label="\\fax value:"
    },
    {
        class="floatedit",name="faxvalue",
        x=1,y=1,width=1,height=1,
        value=0
    }
}
```

When we call `aegisub.dialog.display(dialog_config)`, we see this:

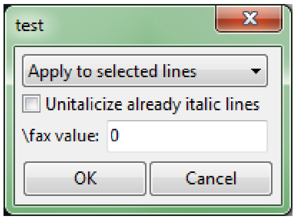


Apply to selected lines

□ Unitalicize already italic lines

\fax value: 0

Well okay, the arrangement could use a bit of work. Let's see if we can't make this look better by modifying the x, y, and width properties. Let's make the dropdown and the checkbox two columns wide, and we'll move the checkbox to (0,1), below the dropdown. The label and the float edit box will still be one column wide, and we'll move them down a row.



Apply to selected lines

□ Unitalicize already italic lines

\fax value: 0

There! Much nicer. We've created our first GUI. If you want to change the default OK and Cancel buttons, just add a second parameter containing a list of desired buttons and you'll be set.

Now that we know how to display GUIs, we need to know how to use the results of the user input. These results are stored as a hash table and are the second return value of `aegisub.dialog.display`. The keys in this hash table are the names of the components that we defined in the dialog configuration table. If, in our example, we store our results in a table named `results`, then to access the selected option in the dropdown box we use `results["lineselect"]`. To see whether the checkbox was checked, we'll see if `results["unitalic"]` is `true` or `false`. To get the value we should use in the "\fax" tag, simply take a look at the number in `results["faxvalue"]`.

So, to summarize:

```
buttons={"Italicize","Cancel"}
dialog_config= --See above

pressed, results = aegisub.dialog.display(dialog_config,buttons)

if pressed=="Cancel" then
    aegisub.cancel()
end

--Handle the results table...
```

Miscellaneous

I'll add to this section as I think of miscellaneous things worth mentioning. If you've followed the tutorial so far, you should be set for the most part.

The automation progress bar can be controlled using functions found on this page. The functions are entirely self-explanatory. It's mostly aesthetic, but can also help in debugging by giving you a rough idea of where in the processing your script encounters an error. Speaking of debugging, further down on the same page are a few debug output functions.

The miscellaneous APIs page I mentioned earlier also has some great functions for getting information about the video. Furthermore, there's the `aegisub.decode_path` function that's very useful if you want to save and load files. Aegisub defines several helpful path specifiers that let you access directories such as the application data folder or the location of the video file.

Lua patterns are powerful enough for the most part, but still limited. Aegisub's documentation includes the re module which is supposed to allow for more robust regular expressions. I tried to use it once and ended up ragequitting. Perhaps I was doing something basic wrong and the module will give you no problems, but consider yourself warned. I, for one, will stick to patterns.

My point of view when writing this tutorial was mainly that of a typesetter, but if you're a kfxer, you'll find tons more good stuff in the karaskel library that I didn't even mention here. Knowing all the capabilities of Aegisub's Lua environment more in depth will help you pull off more advanced karaoke.

Guided Example: Modify Font Size Revisited

[Download example script]

Remember the long list of exceptions in the "simple" version of this macro? It was only really useful in a specific set of cases, and relied heavily on the typesetter not doing anything that the macro did not expect.

I really hate writing macros like this, and your users will be frustrated too. This is a bit of a tangent, but truly useful automations should be robust. They should behave as the user expects them to behave in the vast majority of cases. The previous version of this script couldn't handle relatively common situations like having a \fscx or \fscy tag in the line. Well, we can fix that.

Off the top of my head, the only thing this version of the macro doesn't handle is \t transforms (and if you're going to motion-track it, you shouldn't need to use \t). The comments do most of the explaining, but I'll still walk through the script here.

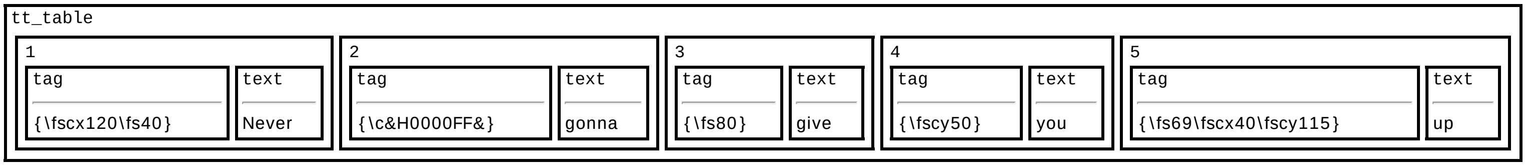
First up, we see our first use of karaskel, using the two functions explained in the karaskel section. Here karaskel is necessary to allow us to access style defaults. After that come some basic string manipulations to set up the text of our line for later. It's important that the line start with an override block, and that the first override block contain an \fs tag.

After that comes the first somewhat tricky part. I parse the line's text into a table, separating text from override blocks, structured in such a way that I can easily see what part of the line each override block affects. I can easily manipulate this table and use it to reconstruct the line at the end.

I don't have a proper name for this data structure, but let's call it a tag-text table. Here I've named the variable "tt_table". If you're having trouble telling how tt_table parses the line, I've drawn up a diagram. If our original line is:

```
{\fscx120\fs40}Never {\c&H0000FF&}gonna {\fs80}give {\fscy50}you {\fs69\fscx40\fscy115}up
```

Then once we've parsed it, our table looks something like this:



tt_table

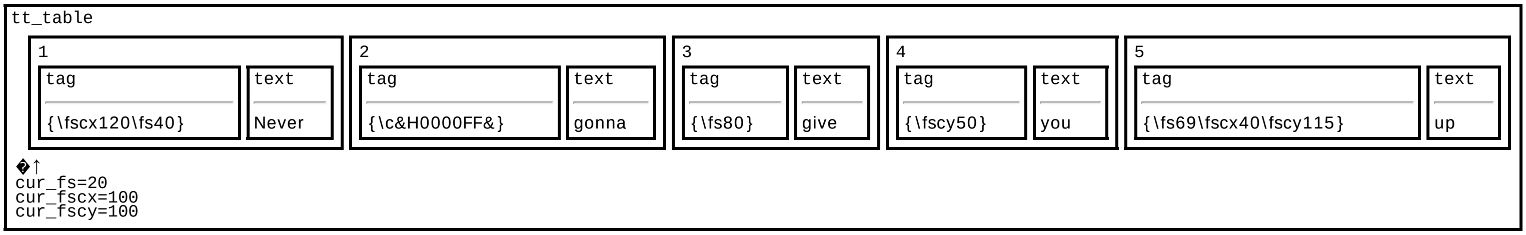
	tag	text
1	{\fscx120\fs40}	Never
2	{\c&H0000FF&}	gonna
3	{\fs80}	give
4	{\fscy50}	you
5	{\fs69\fscx40\fscy115}	up

This means `tt_table[3].text` is "give", while `tt_table[2].tag` is "{\c&H0000FF&}". Plus, since an override tag affects everything to the right of it until it gets overridden again, we know that the contents of `tt_table[2].tag` are going to affect all the text stored in `tt_table[3]` through `tt_table[5]`. In other words, we can start at the left side of the table and move to the right, and at any point in the table we'll know exactly how the text will be rendered, based on all the override tags we've seen so far.

This data structure is the key to several of my most powerful macros, including fbf-transform and gradient-everything, and is what makes them so robust. With this, someone writing a macro can tell what the typesetter is doing at any point in the line.

It is worth noting that this relies on there being an override block at the beginning of the line. It's easy to check if an override block exists at the beginning, and simply append an empty one ("{}") if it doesn't.

Now, we make use of this data structure to help us properly refactor the font size. First, we'll store the state before the start of the line using style defaults. Let's say the default font size is 20, while the default x and y scales are 100.

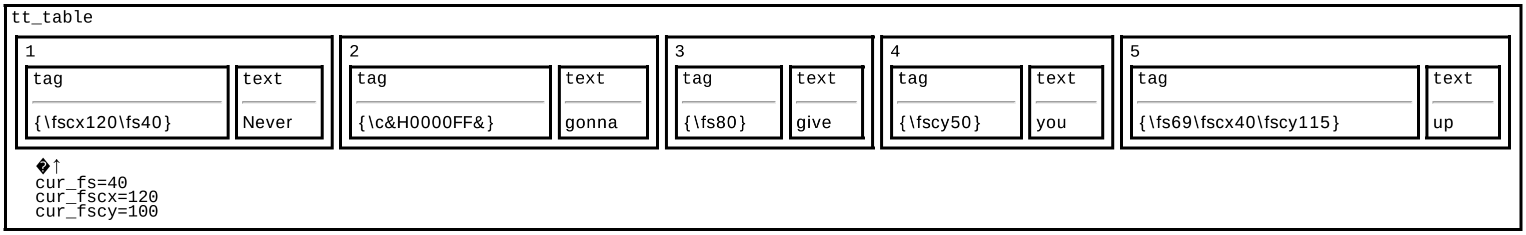


tt_table

	tag	text
1	{\fscx120\fs40}	Never
2	{\c&H0000FF&}	gonna
3	{\fs80}	give
4	{\fscy50}	you
5	{\fs69\fscx40\fscy115}	up

```
�↑
cur_fs=20
cur_fscx=100
cur_fscy=100
```

Then we enter the for loop and begin looking at the first override tag. We'll detect the \fscx and \fs values defined in `tt_table[1].tag`, and use them to update our state variables. Since there is no \fscy tag, `cur_fscy` remains unchanged.

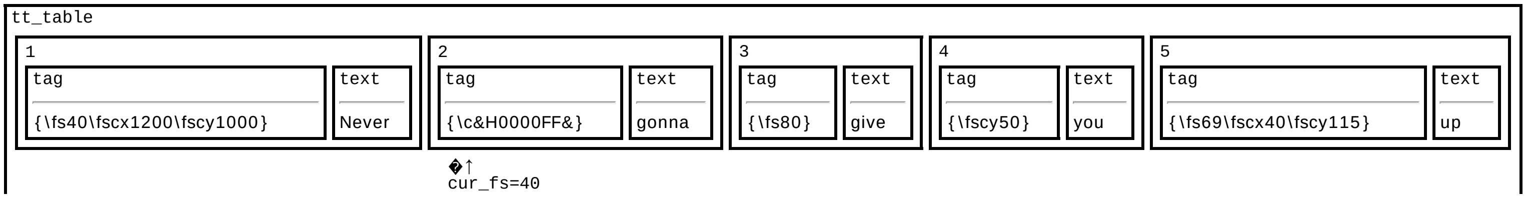


tt_table

	tag	text
1	{\fscx120\fs40}	Never
2	{\c&H0000FF&}	gonna
3	{\fs80}	give
4	{\fscy50}	you
5	{\fs69\fscx40\fscy115}	up

```
�↑
cur_fs=40
cur_fscx=120
cur_fscy=100
```

Using these values, we can calculate that the new font size should be 4. In the previous version of the macro, \fscx and \fscy were simply set to 100 times the scale factor. This time, we'll use the scale values parsed from the line, so \fscx and \fscy values will become 1200 and 1000, respectively. The macro then removes the old tags, adds the new ones, and moves on to the next element in the table.



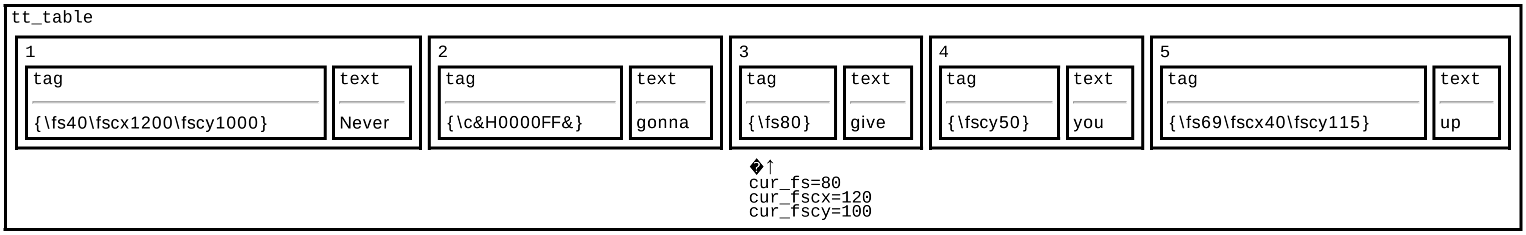
tt_table

	tag	text
1	{\fs40\fscx1200\fscy1000}	Never
2	{\c&H0000FF&}	gonna
3	{\fs80}	give
4	{\fscy50}	you
5	{\fs69\fscx40\fscy115}	up

```
�↑
cur_fs=40
```

cur_fscx=120
cur_fscy=100

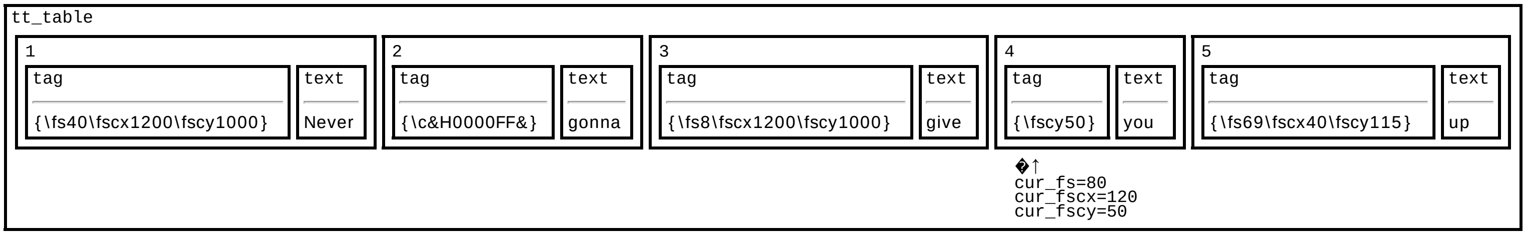
This time, there are no font size or scale tags in this override block. The text here inherets the font size and scale changes we added to the previous tag block, so there's no need to add any more tags. We move on to `tt_table[3]`.


tt_table

	tag	text
1	{\fs40\fscx1200\fscy1000}	Never
2	{\c&H0000FF&}	gonna
3	{\fs80}	give
4	{\fscy50}	you
5	{\fs69\fscx40\fscy115}	up

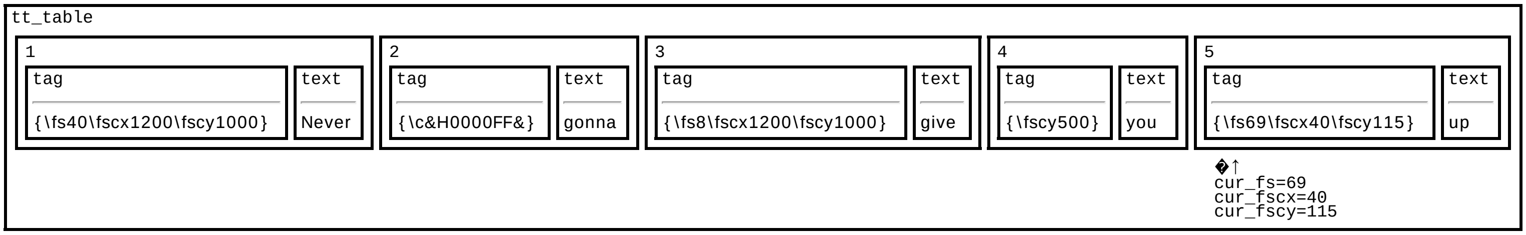
�↑
cur_fs=80
cur_fscx=120
cur_fscy=100

The macro detects the font size change, adds the relevant tags, and moves on.


tt_table

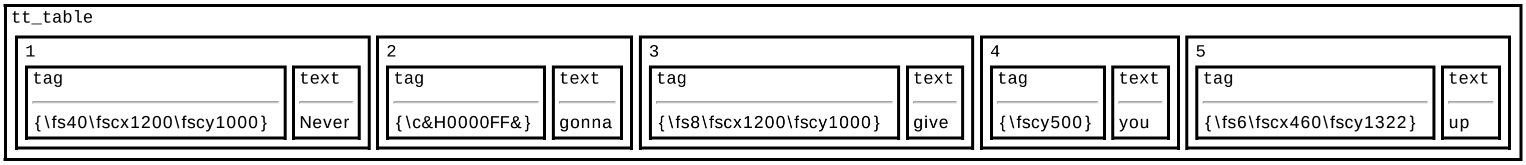
	tag	text
1	{\fs40\fscx1200\fscy1000}	Never
2	{\c&H0000FF&}	gonna
3	{\fs8\fscx1200\fscy1000}	give
4	{\fscy50}	you
5	{\fs69\fscx40\fscy115}	up

�↑
cur_fs=80
cur_fscx=120
cur_fscy=50


tt_table

	tag	text
1	{\fs40\fscx1200\fscy1000}	Never
2	{\c&H0000FF&}	gonna
3	{\fs8\fscx1200\fscy1000}	give
4	{\fscy500}	you
5	{\fs69\fscx40\fscy115}	up

�↑
cur_fs=69
cur_fscx=40
cur_fscy=115


tt_table

	tag	text
1	{\fs40\fscx1200\fscy1000}	Never
2	{\c&H0000FF&}	gonna
3	{\fs8\fscx1200\fscy1000}	give
4	{\fscy500}	you
5	{\fs6\fscx460\fscy1322}	up

And we're left with our final converted line:

```
{\fs40\fscx1200\fscy1000}Never {\c&H0000FF&}gonna {\fs8\fscx1200\fscy1000}give {\fscy500}you {\fs6\fscx460\fscy1322}up
```

Our macro handled all the crazy font size and scale variations in this line like a boss.

That being said, there's room for improvement in this example. Note that several redundant scale tags were inserted, when our script should be capable of detecting which scale tags are necessary to insert and which are not. I leave it as an exercise for the reader to come up with a way to fix this (hint: handle \fscx and \fscy the same way \fs is handled, and guarantee that \fscx and \fscy appear in the first override block).

I've shared more or less everything important about making automations that I know. Any other Lua libraries or techniques you might need will have to be researched on a case-by-case basis. Hopefully you found this tutorial useful in automating your own tasks. Happy coding!

More Lua Scripting

I keep hearing people say "I need to learn some Lua" or "I need to learn to write automation scripts", and not many seem to really have gotten into it. This should help get you started. You should read lyger's guide first, because I'm not gonna explain the same things again, but I want to provide some more practical tips. Rather than explaining lua itself, I'll explain more about scripts for Aegisub specifically.

Learning Lua is not much of an issue. You can learn all the Lua stuff you need in an hour. It's mostly just **if/then/end**, the **for** cycle, **gsub**, and a few other things.

A large part of what you need is regular expressions, or rather Lua's simplified pattern matching. That, again, is something you can learn in an hour.

What I want to talk about the most is how to work with the Subtitles object, which is not really a matter of Lua, but rather of Aegisub and the ASS format. This is explained in the Aegisub manual, but since that may be confusing for beginners, I'll provide some specific practical examples. The goal is to explain how to write a basic automation script in as simple terms as possible. Once you understand how a script that adds blur works, adding more complex functions will be easy because that's just maths.

Here's the very basics:

```
script_name="test"
script_description="testing stuff"
script_author="some guy"
script_version="1"

function test(subs, sel, act)
	-- stuff goes here
end

aegisub.register_macro(script_name, script_description, test)
```

The last line puts an entry in your automation menu. It has 3 parts. 2 of them are defined at the beginning - script_name and script_description. The name will appear in the menu and as an undo entry when you run the script. You can see description in the Automation Manager. The 3rd part means that running this script will run a function called "test".

script_author and **script_version** aren't really important, but I'm sure you get the idea.

Let's look at **function test(subs, sel, act)**. I probably wrote at least 20 scripts before I actually understood what this is. Since this function is referenced by **register_macro**, it's the main function of the script, and as such is by default given the Subtitles object to work with. The 3 parts — subtitles, selected lines, and active line — give you 3 things you can work with.

You can name them whatever you want. You just have to stick to the naming. I tend to keep everything short, though I'm sure I'm not the only one who uses subs/sel/act. It's probably best to use these even just because of the fact that others do it too, which makes it easier to make sense of each others' scripts.

subs is the whole subtitles object. You always have to use this. In simple terms, it's like a table of all lines in the ASS script, including headers, styles, etc.

sel is selected lines, and if you want your function applied to all lines, you don't have to use this. You can have **function test(subs)**.

act is the active line, and you probably won't need it very often. You can use it for functions that are supposed to run on only one line or read some info from the active line. If you select one line and use **sel**, it's pretty much the same as using **act**.

The "stuff goes here" part is where the actual function will be written.

Here's an example of a simple function that runs on the whole script:

```
function test(subs)
    for i=1,#subs do
        if subs[i].class=="dialogue" then
```

```
            line=subs[i]
            text=subs[i].text

            line.effect="test"

            line.text=text
            subs[i]=line
        end
    end
    aegisub.set_undo_point(script_name)
end
```

The green part is what you'll usually have for every script that runs on all lines.
The purple part is the actual specific function.

#subs is how many lines there are in **subs** (including headers and all). If the ASS file has 200 lines, the **for** cycle will run 200 times. You only want to apply this to dialogue lines, not to styles or headers, so you have to specify this condition:
if subs[i].class=="dialogue".

So, the iterator **i** is going from 1 to 200, so when it's let's say 25, **subs[i]** is **subs[25]**, or the 25th line in the ASS file. **line=subs[i]** means that you create element **line** and put **subs[i]** into it. Note that single **=** does not mean "equals". You could read it as "line is now subs[25]" (when **i** is 25). Then you work with **line**, and for it to be of any use, you have to put the **line** back in **subs[i]** at the end. **line** is something you created, **subs[i]** is the actual line in the subtitles, so you need the **subs[i]=line** at the end.

You see the same with **text**, even though in this case I don't need it, but usually you work with **text** the most. The purpose is to use something that's short instead of typing **subs[i].text** all the time. Also, it could also say **text=line.text** since **line** is already defined at that point. You can name those things anything you want, for example just **l** and **t**, which may be good for a short script, but again, **line** and **text** are commonly used by most of us, so it keeps things clear.

aegisub.set_undo_point(script_name) sets the undo point, and should be at the end of the main function, though I think Aegisub does it automatically anyway. You can, however, create multiple undo points, like for every function in your script, but it's usually only confusing and not very practical.

Now, the actual thing this script does is **line.effect="test"**. **line.effect** is the effect filed, and here it takes the value "test", which means the text of the effect field will be "test". So what this script does is that it puts "test" in the effect field of every dialogue line.

Now, the thing I did here with **text** would have made more sense if I'd done it with "effect" instead (because I didn't actually do anything with text), ie. **effect=line.effect**. Then the purple line could be just **effect="test"**. You have to always think about what pays off and what doesn't. For this script, the purple line would be 5 characters shorter, but you would need two extra lines, to assign value to **effect** and to give the value back to **line.effect**, so that doesn't pay off. If you use something only once, you might as well keep it as is. The more often you use something, the more sense it makes to assign it to something with a short name.

Let's look at working with selected lines now.

```
function test(subs, sel)
    for x, i in ipairs(sel) do
        line=subs[i]
        text=line.text

        if text:match("you're tie is crooked") then line.effect="the editor is an idiot" end

        line.text=text
        subs[i]=line
    end
    aegisub.set_undo_point(script_name)
    return sel
end
```

I'm too lazy to do high-effort html, but you can paste the code into Notepad++ for proper syntax highlighting. You can see the **for** loop is using **ipairs**. **sel** consists of pairs of two numbers. The first is the index of the selection, and the second is the index in **subs**. If the ASS file has 50 lines and you select the last 3, then the **x** in **ipairs** will be 1, 2, and 3, and **i** will be 48, 49, and 50. In the previous example, **x** and **i** are the same thing because it goes through all lines.

Don't forget that the function must have **(subs, sel)**. Of course you can always include the **sel** even if you're not using it, just to be sure that you never forget it. I pretty much always use **(subs, sel)** and in rare cases add **act**.

The purple line is a basic example of an action dependent on a condition. You can read it as:
If you find "you're tie is crooked" in **text**, put "the editor is an idiot" in **effect**.

return sel makes sure that you keep the selection you started with (or new selection if you changed it).

You could also use **for i=1,#sel do** instead of **ipairs**, like we did with **subs**. If your script is deleting or adding lines, you need to go backwards, because the new/deleted lines are changing the index of the selected lines. If you delete line 1, line 2 becomes line 1 and line 3 becomes line 2, so going in the normal direction you'd either be skipping lines or going through them twice.

```
for i=#sel,1,-1 do
    local line=subs[sel[i]]

    ...

    subs[sel[i]]=line
```

This is what I use. It starts at the last selected line and goes backwards to the first. **i=a,b,c** means it goes from a to b by steps of c. **i=8,2,-2** would go through lines 8, 6, 4, 2. The default for steps is 1, so unless you go backwards like here, you don't need to write it.

An important thing is that if you use this, then **line** is **subs[sel[i]]**, not **subs[i]**. here **i** is the number of the selected line, starting from 1, so if you used **subs[i]** when **i** is 1, you'd have the first line in the ASS file, not the first selected line. **sel[3]** is the number in **subs** corresponding to the 3rd selected line.

This thing kept confusing me for quite a while, so let's try a more specific example. Let's say **subs** has 50 lines (including headers and styles) and you select last 5 lines.
sel would now be **{46,47,48,49,50}**.
sel[1]==46
sel[2]==47
sel[5]==50
Using the **for** cycle will go from 1 to 5, so **i** will be 1-5, and **sel[i]** will be 46-50. **subs[i]** would be lines 1-5, which is not what you want. **subs[sel[i]]** will be lines 46-50. That's what you need.

So, that about covers the structure of the main function. With this and a bunch of if/then/end lines you can make simple scripts.

Now, let's look at some ways to manipulate **text**.

```
text=text.." end of line"
```

This attaches a string to the end of **text**.

```
text="Start of line "..text
```

This attaches a string to the beginning of **text**. This way you can add tags:

```
text="{\\blur0.6}"..text
```

This is how the old "Add edgeblur" script worked. Of course, this doesn't join it with other tags, doesn't replace existing blur, etc.

```
text="This is a test."

text=""
```

Here the first one sets "This is a test." as **text**, deleting whatever was there before.
The second one just deletes **text**, by making it an empty string.

gsub

gsub is pretty much the core of every automation script. It's what replaces one thing with another.
It works like this:

```
text2=text:gsub("string A","string B")
```

This translates to: If you find "string A" in **text**, replace it with "string B" and assign the modified **text** to **text2**.
I used **text2** for the sake of explanation, but normally you'd use **text=text:gsub**, which just keeps the result in **text**.

"I could not see him."

```
text=text:gsub("could not","couldn't")
```

» "I couldn't see him."

This way you can, for example, write a script for making contractions.

```
text=text
:gsub("([cws]h?)ould not","%1ouldn't")
:gsub("did not","didn't")
:gsub("was not","wasn't")
```

You only need the **text=text** part once. Then you can add as many **:gsub** lines as you want and create a whole list of contractions.
While you can just add them one by one, you can also use pattern matching (lua's version of regexp) to keep the code short. The first gsub line will match could, would, and should. It will also match chould and whould, but as those don't exist, that doesn't bother us. The part in parentheses is a capture. **[cws]** means "c or w or s", and **h?** means "h if it's there or nothing if it's not". In standard regexp you could replace this capture with (c|w|s|ch|wh|sh) to get the same result. Lua doesn't have this option, so sometimes you just need more lines than you'd need with full regexp.
The **%1** is the capture, so whatever it matched in the first part will be pasted in the second.

Now we can use this to replace existing blur value with our new value.

```
text=text:gsub("\\blur[%d%.]+","\\blur0.6")
```

Blur can have numbers and a decimal point, so use **[%d%.]+** to match anything that's a number or a dot as many times in a row as possible, so whatever value the blur has will be replaced with 0.6.
The same effect could be achieved in different ways:

```
text=text:gsub("(\\blur)[%d%.]+","%10.6")
text=text:gsub("\\blur[^\\}]+","\\blur0.6")
```

The first one captures the **\\blur** part, so you don't have to type it again (may be useful if it's something longer).
The second one matches anything that's not a backslash or } as many times it can, ie. until it hits something that IS a backslash or }, which is where the blur value would logically end. This can pretty efficiently capture the whole value of any tag, since any tag has to end with \ or }. Of course with tags like **\pos**, you'll want to capture the coordinates rather than include the ().

You can also use a function within **gsub**:

```
text=text:gsub("(\\blur)([%d%.]+)",function(a,b) return a .. 2*b end)
```

a and **b** are the captures. The function uses them, returning **a** (\\blur) as is, and multiplying **b** by 2, thus giving you the blur value doubled. So you can divide your pattern into a bunch of captures and do some operations with them.

Here's how you capitalize the first letter of a line:

```
text=text:gsub("^(%l)([^{]-)", function (c,d) return c:upper()..d end)
```

First capture is a lowercase letter at the beginning of a line. Second capture is from after the first letter until {, meaning before it hits a comment or tag. Returned is the first capture capitalized and second capture as is (which means it doesn't even have to be there in this case, but you could for example return **d:lower()** to be sure that the rest of the string will be lowercase).

Now you can understand how my Teleporter works:

```
text=text:gsub("\\pos%(([%d%.%-]+),([%d%.%-]+)%)",function(a,b) return "\\pos(".. a+xx.. "," ..b+yy..")" end)
```

Notice that literal (and), as in not captures, have to be escaped with %. Coordinates captures are **[%d%.%-]+**. You see that compared to what we had for blur, thse include %-, because coordinates can be negative. If you don't include that, the script will only work when coordinates are positive. So it captures X and Y coordinate, and adds to them the user input, which is xx and yy here. Yep, that simple.

One more example. This is "reverse move":

```
text=text:gsub("\\move%(([%d%.%-]+),([%d%.%-]+),([%d%.%-]+),([%d%.%-]+)","\\move(%3,%4,%1,%2")
```

That's the whole thing. Capture the 4 coordinates and return them in changed order: 3, 4, 1, 2. This is a good example of how captures can be useful. You may notice that the (is not escaped in the right half. Things in the right part of gsub don't need to be escaped with % - it's only used for captures. Only the left part uses regexp.

Escape characters

When using regexp, these characters have to be escaped with %: **. ? * - + ( ) [ ]** and **%** itself

Characters that have to be escaped with \: **" '** and **** itself

Backslashes and quotation marks always have to be escaped, even in literal strings. (An actual quotation mark ends the string.)
If you want to match an actual question mark in a sentence, you must match **%?**.

Regular Expressions

I'm not gonna explain regexp from scratch, because there's plenty about that on the Internet. What I'm gonna do is list some patterns that are useful for Aegisub automation scripts.

```
{[^\\}]-}      -- comment (stuff between { and } that doesn't have a backslash)
{\\[^}]-}      -- tags (stuff between { and } that starts with a backslash)
{[^}]-}        -- comment or tags (stuff between { and })
```

The third one shows you a typical way of matching stuff between two markers. You match the first marker, then what's-not-the-second-marker with a - or *, and then the second marker.
The difference between - and * is that {.-} matches only one comment or set of tags, while {.*}, if you have a string like "abc{def}ghi{jkl}" will match from the first { to the last }, so "{def}ghi{jkl}". You always have to think about whether you need +, -, or *. If you choose the wrong one, it may still work in simple cases, like if there's only one comment in the line, but it will break on more complex lines. I recommend creating a testing ASS file and fill it with all kinds of different lines, including with mistakes, bad tags, broken comments, etc. Have all combinations of text, tags, and comments, use some transforms, some mocha lines, anything that can be in a script. If you write a function, it needs to do what it's supposed to do no matter what line you apply it to.

```
%d+            -- sequence of numbers
[%d%.]+        -- sequence of numbers, can have decimal point (values of \bord, \blur, \fscx, and so on)
[%d%.%-]+      -- sequence of numbers, can have decimal point, can be negative (\xshad, \fsp, \frz...)
&H%x+&         -- values for colours and alpha
%([^%)]-%)     -- stuff between ( and )

%(([%d%.%-]+),([%d%.%-]+)%)
```

This will capture coordinates of \pos or \org. It could also capture fade in and out in \fad, though that doesn't need the -.
For \move, capture 4 coordinates and don't include the ending %), because \move may or may not have timecodes.

```
[%a']+          -- word (sequence of letters or apostrophes, to match words like "don't")
[^%s]+          -- word (sequence of what's not a space)
```

You may need different ways of matching a word. The first one here will not include punctuation, the second one will. Sometimes you may need one, sometimes the other. You may also wanna replace %a with %w, if you want to include "words" like AK47 or just count 20 in "I'm 20 years old." as a word.

```
\\[1234]?c&    -- colour tag (doesn't match value, just matches that the tag is there)
```

This matches \c, \1c, \2c, \3c, \4c, but not \clip (important!).
(Also note that \\fs matches \\fsp and \\fscx, so be careful about patterns that may match things you don't want.)
Since primary can be \c or \1c, in order to avoid complicated code that would deal with both,
I recommend using this at the beginning:

```
text=text:gsub("\\1c&","\\c&")
```

\c is what the inbuilt Aegisub tool creates, so keep those as standard.

Speaking off... tricks like this are often very useful. If your code needs to account for a lot of different things, see if you can reduce the number of these things with some easy trick. A common issue is for example matching the beginning of a word. A word starts either after a space, or at the beginning of a line. You need to match two patterns for that. However, you can start with adding a space at the beginning of a line, then use just one matching pattern, and then remove the space at the end of the script.

Another thing is dealing with lines with and without tags (when working with text). You can start with this:

```
tags=""
if text:match("^{\\[^}]*}") then tags=text:match("^({\\[^}]*})") end
text=text:gsub("^{\\[^}]*}","")
```

If the line has no tags, then **tags** will be an empty string. If it finds tags at the beginning, they will be saved to **tags**, thus replacing the empty string. Then the **gsub** deletes the tags.
Now you can work with the text knowing that you have no tags in the way.
When you're done with the text, you'll do this:

```
text=tags..text
```

This attaches your saved tags at the start of the line. If there were no tags, you have an empty string saved there, so basically nothing happens.

Another trick I use is when I want to add some tags, and the line may or may not already have some tags.

```
text="{\\tag}"..text                      -- add tag when no tags are present
text=text:gsub("^{\\","{\\tag\\")              -- attach tag before other tags
text=text:gsub("^({\\[^}]-)}","%1\\tag}")       -- attach tag after other tags
```

These would be the regular options. The second and third depend on what you want to do. Just a matter of preference. It works either way. But you have to first find out if the line has tags, and then use the appropriate method. So again, there's a way to avoid that.

```
if not text:match("^{\\") then text="{\\}"..text end
```

You start with adding {\} at the beginning of a line without tags. The second and third method of adding tag now work just fine, but you have an extra backslash somewhere in there. It will end up as either a doubleslash somewhere in there, or at the end before }. So at the end of the script, you do a simple cleanup.

```
:gsub("\\\\","\\")
:gsub("\\}","}")

:gsub("{}","")
```

The first two are what you really need. The third is another "cleanup" line, useful when you've removed some tags and possibly ended up with just empty {}. (Of course the **gsub** is for **text**.)

```
\\t%([^%(%)]-%)                     -- transforms
\\t%([^%(%)]-%([^%)]-%)[^%)]-%)        -- transforms with \clip in them
```

The tricky thing about transforms is that they can have () within () if there's a transform for a clip, so to efficiently get all transforms, you always need both patterns. Yeah, this is a bit messy.
Matching transforms is useful when you modify tags' values but don't want to change the tags inside transforms. You create

transforms="". Then you match those two patterns and save them for example to **tf1** and **tf2**. Then do **transforms=transforms..tf1..tf2** and you'll have transforms saved in the **transforms** string. Then you remove them from the text with gsub and work with the text... and at the end put them back. This is a bit complex and you actually need **gmatch** because there may be many trnasforms. So once you get familiar enough with the code and everything, here's what I do:

```
function trem(tags)
	trnsfrm=""
	for t in tags:gmatch("(\\t%([^%(%)]-%))") do trnsfrm=trnsfrm..t end
	for t in tags:gmatch("(\\t%([^%(%)]-%([^%)]-%)[^%)]-%))") do trnsfrm=trnsfrm..t end
	tags=tags:gsub("(\\t%([^%(%)]+%))","")
	tags=tags:gsub("(\\t%([^%(%)]-%([^%)]-%)[^%)]-%))","")
	return tags
end
```

You run this function on tags. It goes through every instance of a transform and adds it to the **trnsfrm** string. When you're done with whatever you're doing to the other tags, you put this string at the end of the tags.

Some more regexp examples:

```
\\i?clip       -- match clip or iclip
\\[xy]?bord    -- match bord or xbord or ybord

-- remove spaces at the beginning and end of a line
text=text:gsub("^%s+","")  :gsub("%s+$","")
```

GUI

Here's a very simple GUI:

```
dialog_config=
{
   {x=0,y=0,width=1,height=1,class="label",label="\\blur",},
   {x=1,y=0,width=1,height=1,class="floatedit",name="blur",value=0.6},
}
buttons={"blur","cancel"}
pressed,res=aegisub.dialog.display(dialog_config,buttons)
if pressed=="cancel" then aegisub.cancel() end
if pressed=="blur" then blur(subs, sel) end
```

dialog_config is a table with all the stuff in the GUI except the buttons. This one contains two things - a label, and an editbox for numbers. The label is "\\blur". That's what you'll see in the GUI, followed by the editbox, which will have "0.6" in it as default value.
buttons is the buttons you will click on.
aegisub.dialog.display is what displays the GUI, using the **dialog_config** and **buttons**. **pressed** determines which button was pressed. (You can name this whatever you want.)
res is the user input from editboxes, checkboxes, etc.
I started with "pressed, result", as that's what lyger's guide had, but over time changed "result" to "res" because it gets typed a lot. Again, it can be anything you want.
As you can see in the last line, if you press the "blur" button, function **blur(subs, sel)** will be executed.
To get the blur value inside that function, you'll use this:

```
blurtag="\\blur"..res.blur
```

res.blur is the value given by the user, so if you type "1.5" in the editbox, **blurtag** will now be "\\blur1.5".

Other types of input:

```
-- Checkbox
{x=0,y=1,width=1,height=1,class="checkbox",name="nobreak",label="remove linebreaks",value=false},
```

Usage: **if res.nobreak==true then** (stuff)
You don't have to type "==true" because that's implied by default, so it can be just:
if res.nobreak then followed by what should be done if the checkbox is checked.

The opposite would be either **if res.nobreak==false then** or **if not res.nobreak then**.

```
-- Dropdown menu
{x=3,y=0,width=1,height=1,class="dropdown",name="an",
      items={"an1","an2","an3","an4","an5","an6","an7","an8","an9"},value="an8"},
```

```
-- Colour
{x=4,y=0,width=1,height=1,class="color",name="c1"},
```

The colours come in this format: "#000000" in RRGGBB order. The actual tag is "&H000000&" in BBGGRR order, so you have to transform the result, for example like this:

```
colour1=res.c1:gsub("#(%x%x)(%x%x)(%x%x)","&H%3%2%1&")
```

Debugging / logging

This is very useful when you're getting errors. If you want to find where exactly your script has failed, you can use aegisub.log. There are two main ways to use it. One is to check whether the script passed a certain point, and another is to check a specific value.

The first one will usually go after **then**, like this:

```
if text:match("\\blur") then aegisub.log("check")      -- function continues after this
```

If this "check" gets logged, you know the condition has been met, ie. "\blur" was found in the text. This tells you how far the function has gone before something broke and helps you narrow down where the problem is.

The second way is for checking the value of something.

```
aegisub.log("abc: "..abc)
```

The first part is just text, so that you know what's being logged if you're using several logs. The part after .. is the value of the variable called "abc".
I often log multiple things when testing/debugging, and it gets chaotic unless each log is on a new line, so I automatically put "\n" into each log:

```
aegisub.log("\n text: "..text)
```

This would be the most common one. Usually you work with **text** and make changes to it, so this shows you which changes did or didn't happen.
So if you're getting errors, use logging to find out what exactly breaks and where.

Various stuff

```
-- duration of a line
dur=line.end_time-line.start_time
```

```
-- character count
-- this counts the string length after removing comments/tags. you can add :gsub(" ","") to not count spaces.
visible=text:gsub("{[^}]-}","")
characters=visible:len()
```

```
-- working with the line after the current one
-- if you're on the last line, there is no next line and you'd get an error, thus the condition
if i~=#subs then nextline=subs[i+1] end
```

```
-- working with previous line
-- previous line always exists, but the one before the first dialogue line would be a style or something
prevline=subs[i-1]
if prevline.class=="dialogue" then blabla end
```

```
-- counting stuff. "count=0" must be at the beginning, before the for loop.
count=0
-- then in the main function:
if text:match("stuff") then count=count+1 end
-- at the end, after the for loop, you can log the result like this:
aegisub.log("Stuff apears "..count.." times.")
```

```
-- error messages to the user
-- if a script requires \pos tag and the user runs it on a line that doesn't have one, you can do this:
if not text:match("\\pos") then aegisub.dialog.display({{class="label",
      label="No \\pos tag found.",x=0,y=0,width=1,height=2}},{"OK"}) aegisub.cancel() end
```

```
-- marking lines that have changed (running gsub doesn't tell you whether the pattern was found)
text2=text
text=text:gsub("\\1c&","\\c&")
if text2~=text then effect=effect.."colour tag modified" end
```

```
-- a simple script to convert between clip and iclip
-- you can't do one and then the other, because that would just convert the iclips you made back to clips
-- therefore, you need elseif, which only comes to play if the first condition isn't met
if text:match("\\clip") then text=text:gsub("\\clip","\\iclip")
elseif text:match("\\iclip") then text=text:gsub("\\iclip","\\clip") end
```

Functions

Click here for some small functions that I've written (or got from someone, in a few cases) and that you can use.

Here is my Italicize script explained line by line. I figured I would take the smallest script I've made and explain it, but it kinda turns out that this one is actually pretty complicated and I could barely make sense of it myself (because it checks the style, deals with inline tags, checks for some mistakes, etc.). Anyway, I explained it as best I could, so hopefully it helps. (Every comment refers to what comes after it.)

This should be more than enough to get you started. Once you learn all this, you can figure out more from looking at existing scripts.

« Back to Typesetting Main

Video Tutorials

Yep, it has come to this. Just like for timing, I made some videos for typesetting too.

Part 1 - about 10 signs in Namiuchigiwa no Muromi-san.

Download (76 MB, 16 minutes):

Some signs from one episode of Muromi. Everything from timing to final product, with typeset commentary.
No mocha tracking, nothing too difficult. Styles are already created because looking for fonts would make the video 10 times longer and more boring.

Part 2 - 50 signs in Monogatari 2nd Season.

Download (70 MB, 21 minutes):

This will probably not teach you anything; it's more of a reference point to see what can be done and maybe fun to watch. I took first 50 signs (there are usually 100-200 per episode) and timed & typeset those in 20 minutes.
I use scripts that I specifically wrote for Monogatari and the efficiency is almost ridiculous.

Part 3 - 3 signs in Kami Nomi zo Shiru Sekai: Megami Hen.

Download (45 MB, 12 minutes):

This one is pretty simple and slow, showing some rather basic stuff.

Part 4 - 6 signs in Kami Nomi zo Shiru Sekai: Megami Hen.

Download (56 MB, 13 minutes):

Some glow, some perspective, and several moving signs. Tips and different ways to do things.

Part 5 - Hyperdimensional Relocator.

Download (33 MB, 15 minutes):

How to use this script and what you can do with it.

Bonus - 60 signs in Monogatari 2nd Season.

Download (75 MB, 14 minutes):

I recorded Monogatari the next week as well, just for the heck of it. It covers pretty much the whole episode except mocha tracking/clipping for the last sign in the video and 2 more signs (Years Earlier / Present). This video is played at 2x speed, so what you see took about 28 minutes. The remaining two signs probably took at least as much and ended up looking pretty good with the effect I used. Also, this one has no commentary and is not very educational at all.

« Back to Typesetting Main

More Stuff about Typesetting

This chapter will be a collection of fairly random tips. While the rest of the guide has been more or less systematic and step-by-step, here I will address various things from different perspectives, grouped together by other criteria than before. At this point I also assume that you can already typeset, so this will be more about efficiency. But first, allow me to reflect on what has happened during my time of 'putting text on screen'.

Evolution of Sorts

It's been 7 years since I started fansubbing, and typesetting has gone through a number of phases. Back then, there were only a few people who understood the whole concept of typesetting well enough to produce something of value. Even so, matching the original sign was only marginal, nobody was using blur, nobody was using layers, there were almost no scripts (except for kara?), and for the most part, \an8 was considered typesetting.

I started as a timer and had no plans for becoming a typesetter. Had I stuck to that, I wonder where typesetting would be today. Right around that time, though, Commie's main (i.e. *only*) typesetter quit, and there was no (competent) replacement. When I saw what was being produced, I realised I could actually do better, simply because I knew a few tags and what they did. Yeah, at that time, that was enough to be better than almost everyone else. So I took over somewhere in the middle of whichever BakaTest we were doing back then. Already before the end of the season, I was doing things that made people go "Wow, how did you do that?" and "What, that was typeset?". And so it began.

I like creative work, and there was no one else to do it anyway, so I first learned everything I could about all the tags from the official documentation and then started experimenting. As the TS got progressively better, this began changing the standards rather quickly. Those who were still doing just basic work were soon being laughed at, and typesetting became quite a competition between groups.

I started writing this guide to share what I had learned, and others, being given the essential guidelines, were figuring out some more things, and the knowledge was building up. I didn't know anything about programming, so at first I was making scripts by taking existing ones and modifying them only very slightly. There was this primitive "add \be" script that I changed to "add \bord" and about 10 other versions. Pretty ridiculous in retrospect, but that was all I could do.

It was only when lyger wrote some lua scripts that things sped up again. It took a while, but he managed to teach me the basics of lua and of how the whole subtitle object in Aegisub works. Honestly, I wasn't very fast at grasping it. =/ But I really wanted to (and needed to, given that I was doing like 10 shows per season around that time), so I persevered (and bothered him a lot, to the point that he wrote that one part of this guide). Once I figured out what the fuck I was doing, the limits were only in imagination and started dropping like flies. Typesetting now and typesetting in 2011 are worlds apart.

By around 2015, I couldn't really think of anything new to write, because I had written everything I could envision. Unfortunately, with all the scripting, and after timing about 300 and typesetting about 600 episodes, my hands died, and some days I couldn't even touch anything because of the pain. It took 3 years for it to get somewhat better, and it's not over yet. But at least I can use the computer more or less normally. (Of course timing or typesetting every day does not qualify as "normally".)

So I'm typesetting a little again, writing some new functions (mainly because it's fun, rather than out of any kind of necessity for it, but a few interesting things have come out of it), and writing some new things on this website. What has happened in the last 3 years?

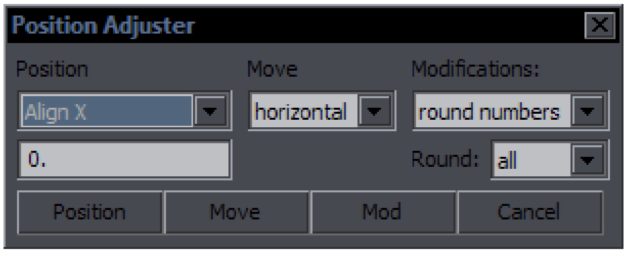
Well, mainly fansubbing in general has almost died. Only a handful of groups still edit and typeset for real. On the other hand, there are like 20 groups that download HorribleSubs, extract the subtitles, put them on a different raw without any change to the text, and release that. I guess it's easier than editing and typesetting. No issue with that per se, but when I'm looking for "some better subs than HS" to watch, I can't find shit.

I suppose the fact that almost everything is on Crunchyroll these days has convinced most people that doing anything better is not worth the effort. Crunchyroll is pretty bad in many ways, and I don't like when I have to watch HS (which is partly why I started subbing again), but that's where we mostly are now. Most people don't care about quality. I'm grateful to the few who do and still make good releases. (Heck, I'm even grateful for herkz at this point, though he almost exclusively picks terrible shows. Still, gotta give him credit for remaining probably the most productive fansubber.)

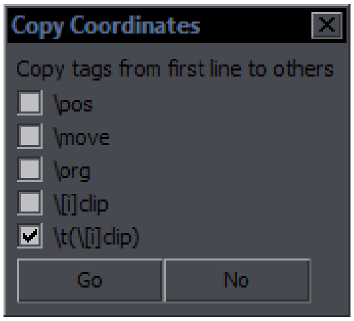
So we have come full circle, in a way. We have mostly crappy typesetting today (if any) because it's done by Crunchyroll.

Typesetting in 2013

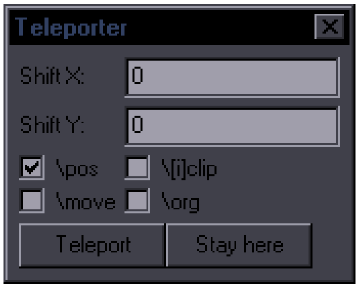
I've found some fun things on my HD, so let's have a look at that, first. June 2013 was when I started writing my own scripts. Before that, I only modified other people's scripts for completely trivial things. By the end of May, I guess, lyger managed to get me from "completely retarded" regarding lua to "ohhh! ...no, I still don't really get it." I think I was mainly just using gsub and thus figuring out the mysteries of regexp, or lua's version of it. After a month or so, I was able to do this:


Position Adjuster

Position: Align X
0.
Move: horizontal
Modifications: round numbers
Round: all


Copy Coordinates

Copy tags from first line to others
☐ \pos
☐ \move
☐ \org
☐ \[i]clip
☑ \t(\[i]clip)

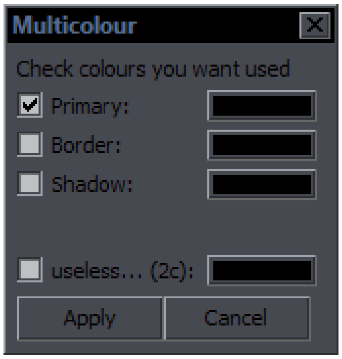

Teleporter

Shift X: 0
Shift Y: 0
☑ \pos ☐ \[i]clip
☐ \move ☐ \org

Each dropdown in Position Adjuster had... wait for it... two items! You can probably recognise that these are what 3 months later became the first version of Hyperdimensional Relocator. But this was huge progress back then.

The only advanced things that existed at this point were lyger's gradient scripts (which replaced Gradient Factory, thank Satan, because that was horrible), border-split (for layers and blur), and fbf-transform, but even those were only a few months old, i.e. early 2013. Maybe the only really cool thing before that was torque's Motion script, which, however, in 2012, almost nobody used because almost nobody knew how to use Mocha.

So mostly we just had "Add Tags", which was a small GUI where you typed something like "\bord3" and that was added to all lines, with no check whether there was already a \bord tag there or anything like that. To save myself a bit of that typing, I made variations with "\bord0", "\blur0.6", "\shad0", etc. already typed in, so I only had to change the 0.

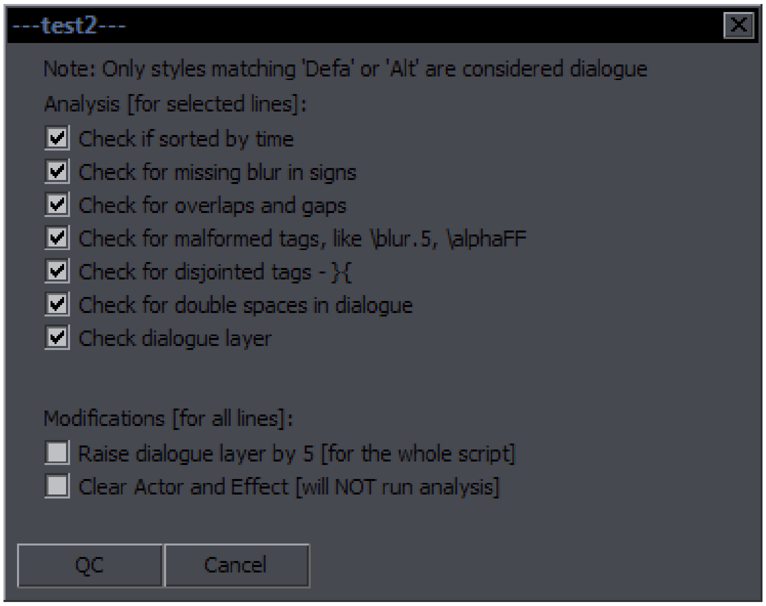

Multicolour

Check colours you want used
☑ Primary:
☐ Border:
☐ Shadow:
☐ useless... (2c):

In mid June, I made this... with a comment at the top that said "why the fuck has nobody written this yet?" which showed my frustration at the fact that I, somebody who still had no idea how this fucking lua thing worked, had to write something as basic as adding colours to selected lines, because apparently nobody else had created such modern blasphemy before. You could only use Add Tags and type (paste) the colour tag there.

A month later, this somehow evolved into the first versions of HYDRA, which had this, 20 other tags, and could do transforms. That was some cool shit. It was so cool that many people said, "OMG, this is so confusing! All these options and stuff! I don't know what to do with it... I'll just use Add Tags." And some did that for, like, years to come. Seriously.

I also found this on my HD:


---test2---

Note: Only styles matching 'Defa' or 'Alt' are considered dialogue
Analysis [for selected lines]:
☑ Check if sorted by time
☑ Check for missing blur in signs
☑ Check for overlaps and gaps
☑ Check for malformed tags, like \blur.5, \alphaFF
☑ Check for disjointed tags - }{
☑ Check for double spaces in dialogue
☑ Check dialogue layer

Modifications [for all lines]:
☐ Raise dialogue layer by 5 [for the whole script]
☐ Clear Actor and Effect [will NOT run analysis]

Judging by the awesome name "---test2---", I'm pretty sure nobody other than me ever used or had this, but as some of you can see, this is how the QC script later came to be, apparently.

To see more of this, you can check **this page**.

So yeah, things were nothing like they are now back then. Now, remember Acchi Kocchi? The show that had about 100 signs per episode? Well, I did that shit in early 2012. So imagine what that looked like. I spent 8-12 hours on each episode. Wonder how long that would take today. It was fun to work on that, but dude, 12 hours in a row...

So those of you who came to this later, if you sometimes feel like typesetting is pain, look at this and be happy.

Typesetting in 2018

I've seen a few good releases from the last 3 years, so there must be a few guys out there who can still make some use of this guide and the existing tools. Typesetting should be really easy nowadays, but maybe it's not quite that for some. Rather than there not being enough info, though, the problem seems to be that there's so much, and so many scripts and functions, that people find it hard to find what they need or what's important. Lots of time is wasted on inefficient methods. So I want to use this last chapter to address this problem and look at things from various points of view, even though I'm not sure if there will be more than like 2-3 people reading this. This section will be a bit all over the place.

I want to talk about what things can be done and how, and what things shouldn't be because they amount to completely unnecessary effort. A few weeks ago, I saw an apparently recent video of somebody typesetting. He was TYPING tags and copypasting them from one line to another by, you know, selecting the tag carefully, ctrl+C, going to a bunch of lines, and pasting it there with ctrl+V one by one. Holy shit, I thought, what the fuck is he doing? So much effort, so much time wasted, so much wrist-killing activity. (He's probably OK on the last one for now, but give it a few years...)

So, don't fucking type tags! That's like stone-age-era typesetting. But first, a few words about hotkeying.

Use Hotkeys for EVERYTHING

It's not just scripts. I'm sure you use hotkeys for timing, and the same way, just about anything in Aegisub can be hotkeyed. Create a new line, go to end of line, set times to nearest keyframes, colour pickers... hotkeying all of that makes work much faster. But don't just add hotkeys under Default.

Set Hotkeys under Video and Subtitle Grid

For typesetting, this is a must. It lets you have LOTS of hotkeys and lets you have single-key ones. The Default ones are limited because they also work in the Subtitle Edit Box, where you need keys for typing. But when the video or subtitle grid are active, you can't type anything anyway, so you can hotkey the most used scripts (and other functions) to just "A", "S", "D", etc. Using the first letter of the script's name makes it easy to remember, and even if several scripts share a letter, you can use "D", "Alt+D", "Shift+D", "Ctrl+D", "Alt+Shift+D", "Ctrl+Shift+D", "Ctrl+Alt+D", and probably even "Ctrl+Alt+Shift+D", but no need to go that far. This goes for all letters and numbers and a few other keys, so you have a few hundred options. Additionally, the numpad numbers are considered different from the regular ones.

So I hope nobody's actually clicking on the Automation menu at the top and looking for scripts there. The 10 minutes it'll take you to set up the keys (might wanna backup hotkey.json so you don't have to do it again if something bad happens) will ultimately save you many hours. You shouldn't have to look for anything in the top menu, and basically you shouldn't use your mouse for clicking anything that's smaller than an icon. Speaking of icons, I have even switching between the icons on the left of the video hotkeyed. Often I have a more complicated hotkey under Default, so that I can use it no matter what, and a simple hotkey under both video and grid, and sometimes even audio if that's useful. The video and grid ones are just clones of one another, since with typesetting you go from one to the other a lot and need the same hotkey to work in either case. So...

Don't Fucking Type Tags!

This was pretty bad even in 2013. I don't know why anyone would do that now. There are two ways you can create most tags.

One Click Solution for some of the most used ones - blur, border, shadow, alpha. Use the *Cycles* script and bind it to a single key under Video and Subtitle Grid, perhaps best to keys you can easily find with your left hand without looking, like A S Z X. It literally takes less than a second to add \bord0 to all selected lines, including incorporating it into existing tags. If you want \bord2, you just hit the key 3 times, which you can do quickly enough to still make it under one second. Why anybody would go to the edit box for a single line and type the tag including { } is beyond me. I mean, do what you want, but it's your time and your hands that are wasting away.

(The Cycles script even has instructions for how to add more macros for other tags, if you're missing something vital there.)

HYDRA is for pretty much everything else. One key to open the script, select as many tags as you need, just type the numbers for values (default ones can be saved, so no need to type those), and Apply. Transforms are done exactly the same, just clicking the Transform button instead of Apply. By clicking yet another button instead, you can actually create gradients with the same ease. With one extra click, you can create inline tags. And of course HYDRA can do a lot of more complex things with only minimal additional effort.

IMO it pays off to read the **manuals** to my scripts to find out what you CAN do. Not that you need to remember how to do everything a script does - even I don't remember it for my scripts, but it's useful to know what it CAN do. You can always look up the details when needed, instead of regularly doing something completely unnecessary and tedious.

Plus the manuals are longer than half of this whole guide and explain a lot of things, so once you've read and learned the stuff in this guide, it's probably the next logical step.

Copying Tags from Line to Line

If you're SELECTING a tag to copy, you're doing it wrong. Having to go to the EditBox, having to start and end the selection in a precise 3-pixel area, going to another line and finding another 3-pixel-wide area to paste the tag into, never mind pasting it into several lines one by one, all of that is unnecessary. Most copying can be done with one key and 1-2 clicks.

Since you usually progress from top to bottom in the grid, you mostly copy things from lines above to lines below.
This is what *NecrosCopy* was made for. Select the line from which you want to copy and the line(s) to which you want to paste, and just check what to copy from one to the other(s). Copying all start tags or text only requires one click. Copying a specific colour tag takes two clicks. Copying any other specific tag from the first line takes three. And you can select as many of the tags as you need.

But of course there's much more that can be done, including copying from bottom up. Necroscopy lets you copy just about anything you want from the first line. Aside from pasting it immediately to the other selected lines, it also keeps the tags in memory. So then you just select some lines higher up in the script, use Necroscopy again, and instead of copying use Paste Saved.

Shortly after I had written that, I added a "bottom up" option to NecrosCopy, so copying fom last line upwards is now even easier for all the copy functions.

Alternatively there's *MultiCopy*, which works in a different manner and allows for copying LOTS of things, not just tags, from one set of lines to another. Or from one to many. Again, check the manuals to find out what the options are. The possibilities are so countless it's ridiculous.

And if that wasn't enough, *Relocator* also has a simple interface for copying \pos, \move, \org, and \clip, with some additional features.

Navigating in the Subtitle File

When typesetting, you're likely to create lots of lines, and then it becomes difficult to find things. To help with that, I wrote **this script**.

The first function was to go to the next line that has a different text (ignoring tags and comments). So it would skip all mocha tracked lines for the same sign and find the next one (unless it happens to have the same text). This, again, should be bound to a simple hotkey.

I have since created several macros in that script so that each can have its own hotkey. You can read about all it can do on the page, but generally I would recommend at least 4 hotkeys.
- Jump to the next line with different text (as already described)
- Jump to the previous line with different text (the opposite)
- Jump to the next line with different style (good for finding signs among dialogue in a time-sorted script)
- The GUI, where you can select any of the other options

The bonus is that this works on multiple lines. If your sign has 3 lines with different text and is mocha-tracked and sorted by time, select the first 3 (whatever's on the first frame - good idea to hotkey subtitle/select/visible), and the script will find the first line after this that doesn't contain any of the 3 texts (or styles, or whatever you use as the marker).

It also works for Actor and Effect, so that gives you more options to find some specific lines that you may have marked in those fields. It can also find commented lines.

Split Text into Smaller Parts

Think of the signs in open books/magazines, with bow-shaped text, or any other cases where you can get some advantage of splitting the text into parts. When each part of the line is different, it may sometimes work with a gradient, but other times it's just convenient to split the line into parts. This may have been a pain in the ass years ago, but thanks to the tools we have now, it's pretty simple. Firstly, splitting the text is easy, even for multiple lines at the same time, and secondly, just about every script can work with many lines at a time, some in pretty sophisticated ways.

Splitting text can be done with the "split by \N" function in *NecrosCopy*. Obviously it splits by line breaks by default (even maintaining the position of each segment in the more basic cases), but if there is no line break (or if you check 'split GUI'), it allows you to split by spaces or by tags. By spaces usually means each word on a separate line; by tags means if you have inline tags, a new line will start at each of those. This can be applied to multiple lines, and you can also number the resulting lines, so if you have several layers with the same text and split by spaces, all lines with the first word will be number '01' (in Effect), etc.

If you want to go further than that, you can also split by letters, using the 'split into letters' function in *Significance*. This works by keeping the whole text for all lines and setting alpha FF for all letters that shouldn't be seen, which means you get to keep the sizes, rotations, and everything. (Unlike 'split by /N', which really splits the text.) Should you want to really split into letters so as to have only one letter on each line, you can use 'letterbreak' from Relocator's Morphing Grounds and then split by \N. (Letterbreak puts \N before every letter.) This way, though, you lose the alignment and have to arrange the letters from scratch. Relocator will help you realign them in various ways. (Align X/Y or FBF Retrack from Repositioning Field can be useful.)

Yet another way to split the text by letters is 'space out letters' from HR's Repositioning Field. Here you set how many pixels apart the letters should be. This basically combines 'letterbreak', 'split by \N', and some realigning. So depending on what you want to do, you can do either this or 'split into letters'. If you then select all the lines and use the \frz tool, the 'split into letters' version will rotate like whole text in one line, whereas in the 'space out letters' version, each letter will stay in place and rotate on its own axis.

The Relocator version is now also in NecrosCopy, thus accessible more quickly. A new option *split by clip*, useful for those bow-shaped book texts, is part of Split by \N in NecrosCopy. It's mentioned further down when talking about clips and described in detail in the Alignment section.

Creating Inline Tags, Even for Each Character

Creating inline tags is easy with *HYDRA*. This is done using the *Tag position* field and its presets. The simplest way is to put an asterisk in the text. (The text is shown for the first selected line.) This way all the tags you select will go to the asterisk position. It will apply to multiple lines, but only if they share the same text.

Then there are the presets. One of them is "before last character", which is useful if you want to gradient the line with lyger's gradient by character. Then there are several that calculate the positions based on proportions: middle of the line, 1/4, 5/8, etc. These are only useful when you're trying to do something very specific.

The next two are quite practical, though: "custom pattern" and "section". The former works similarly to the asterisk position mentioned above, but it's less restricted. Instead of using the asterisk in the whole text, you can use it with a pattern. So instead of "Whole *text", you use "*text", and the tags are applied before the word "text", wherever it's found, no matter what the rest of the line is. Also works for "text*" or "te*xt". You can even use a pattern like " *", and the tags will be put after every space, so this is pretty versatile.

The other option, "section" is similar except that it matches only the first instance in each line and returns tags to original position at the end of that section. This can be used when one word in the line is a different colour (or size, or whatever) from the rest. This saves you the trouble of picking tags before AND after the word.

There are presets for putting the tags before each character or each word. The point of this is that, while they are the same tags and thus are useless in this phase, this allows for various kinds of tweaking later. It can be used for setting up a gradient. It can be used for changing the colour for each word with *Colourise*. If you need a different colour for each word, set one colour for each word with HYDRA, and then use Tune Colours from Colourise to easily change them without having to click before each word for the positioning. If you add a tag before each word, you can then add transforms (for the same tag, in this case) to each block (there's an "all tag blocks" option for transforms in HYDRA). Then you can randomise either the original ones or the transformed ones, so the words will either transform from one value to various ones or vice versa. (We'll get to randomising later.)

The last preset is "text position", which uses number input, like "6", and puts the tags after the 6th character in each line, whatever the text is. So if you had several lines that should start with a red letter but the rest of the text would be blue, you first add red to all lines and then add blue to position 1. (The beginning of the line is 0.)

There's also one little hack in the Special functions (HYDRA): "convert strikeout to selected". If you really have to go to the Edit Box, instead of typing the tags, you can just apply strikeout, select this option in HYDRA's Special functions, select tags as you normally do, and the \s1 will be changed into those tags. Additionally, if you also use \s0, tags will be reset there, so selecting a word in the Edit Box and clicking on strikeout will create "{\s1}word{\s0}", and selecting border 3 and shadow 2 in HYDRA will turn it into {\bord3\shad2}word{\bord\shad}. This also allows for specifying different positions in different lines and applying the same tags to all those positions.

New additions in HYDRA 6 are presets *replace {•}* and *replace {~}*.
For those, there are also two macros in HYDRA: *Bell Shifter* and *Wave Shifter*. This has become my favourite way of creating inline tags. With pressing one key, the Shifters move "{•}" or "{~}" word by word. I rarely put inline tags in the middle of words unless it's a full line gradient, so this way I just quickly mark the right word, open HYDRA, scroll one line up in the presets, and apply tags I want. The {•} becomes {\bord3} or whatever I select.

If you need to, though, you can move them letter by letter too, if you comment the line first. The letter counting is done by lua without the re module, so ä is 2 characters, thus it takes two turns to move "through" it, but that's not an issue when you're just repeatedly pressing one key. So you can use these presets to put inline tags anywhere. You can shift the symbols to any place in any line and turn them all into tags at the same time.

Other scripts have some options for inline tags as well. Colourise can apply a set of 2-7 colours by character or by word. It can also shift these by character or word. "Set colours across whole line can set several colours in regular intervals. If you go with 7 colours, the first is at the beginning, the last before last character, and the other 5 regularly spaced in between. This, again, helps set up a gradient by character.

Masquerade's *Shift Tags* can shift inline tags left or right by a number of characters or words, so tweaking inline tags' positions isn't hard either. It can also shift start tags, so you can create a colour tag in the start tags, and then shift it by one word, instead of creating an inline tag in the first place. Inline tags get shifted as whole blocks; start tags get extracted from the block and shifted individually.

Additionally, it has an even easier way of shifting tags marked with HYDRA's *Arrow Shifter*. The Shifter, similarly to the other two mentioned above, shifts ">", not by spaces, but by tags, always ending before the tag (">\"). Shift Tags detects it and asks you where you want to shift the tag thus marked, and you only mark a place in the clean text. Going even further, if there's "{•}" in the line from the Bell Shifter, the tag goes there immediately. So if you want to move a tag somewhere else, mark the tag with Arrow Shifter, Mark the target place with Bell Shifter, and then it's just a click on Shift Tags.

Of course one of the common ways to create inline tags is making a **gradient by character**.

There are a number of scripts that can do it. The most versatile two are lyger's *Gradient by Character* and *HYDRA*.

GBC creates a gradient between any two tags of the same type. This means that you need to place at least two of them in the line first (or one if the starting value is from the style). But it also means that you can place them anywhere you want, and as many as you want, and set the line up for gradienting exactly the parts you want, exactly the way you want.

HYDRA works differently. You don't need any tags present in the first place. The starting values can be taken from style, and you set the final values in the GUI. The gradient always goes across the whole line. There are some extra options, though. You can use accel with the gradient, you can use centred gradient, and you can set the centre in a specific place. This is explained in detail in HYDRA's manual, so I won't go into that here.

Both scripts allow for mixing gradients for different tags with different values.

Colourise creates gradients for colours. An extra option here is to do a HSL gradient, rather than the normal RGB. (HYDRA can now do this too.) This means that saturation is preserved when similar on both ends, while in the RGB one, the middle tends to get greyish. There's also a double HSL gradient, which will go through the range of colours twice. (Effective with long lines in song styling, for example.) There's an option to reset after \N, which means that if there's a linebreak, the parts before and after it will be gradiented separately. And there's a Reverse Gradient option that simply reverses the order of colours in the line.

Another thing that can do a gradient is the clip2fax function mentioned below.

Apply Tags by Layers, Style, Effect, etc.

With *HYDRA*, you can restrict what lines you want to apply the tags to. This means that you can keep a large selection and apply tags only to certain layers or styles, or lines marked with something specific in Effect/Actor. Numbering lines can also help for this purpose, which you can do with *Significance*. You can even restrict by text pattern, which includes tags, so for example only apply to lines with \frz in them.

Use Clips to Do Lots of Things

Clips are useful not only for clipping but also for making some points on the screen from which other things can be calculated. If you make two points with a clip, you have two positions, X distance, Y distance, and a direction. From this, many things can be determined, so there are several functions that make use of it.

clip to frz (NecrosCopy, Relocator)
If you need to rotate a sign to align it with a straight line in the video, simply draw two points with a vectorial clip along that line, and use this function.
If there's some perspective, you can use two lines. Let's say there's a rectangular area for the sign, and the \frz at the top is different from the \frz at the bottom, and you need the \frz for the centre. You draw 2 points along the top and 2 along the

bottom, and clip2frz will calculate \frz for the middle (averaging out between the two lines). Note that the two lines have to be drawn in the same direction.
(And in the direction of the text, so right to left will give you an upside-down sign.)

clip to fax (NecrosCopy, Relocator)
This is similar, but you draw a line for the vertical alignment of the letters. Usually it's used in connection with the previous one. First you rotate the sign, then you adjust \fax with this. It supports \fscx\fscy. (The ratio of the two changes the visual effect of \fax.)
Again, this can use two lines, one on the left and one on the right end of the sign, if the \fax needs to be different on each end. So you can draw a rectangle around the sign, with the sides leaning in the intended directions of the first and last letters. In this case, the order of the drawing doesn't matter, except points 1 and 2 are left side, and 3 and 4 the right. This function will not only set the \fax on each end but also immediately create a gradient between the two.

The combination of these two is one of the things I use most frequently when typesetting. It's very easy, very quick, and very useful.

clip to reposition (in Relocator)
This takes the distance and direction of the first two clip points, and changes \pos based on that. The clip is basically used to determine values for Teleporter. The usefulness is in that you can draw the clip over something on the screen and get the precise distance you need. This is not very essential, as Aegisub can handle relocating and dragging multiple signs quite well now, but in some cases it can still come in handy, like when repositioning a bunch of signs with \move.

clip to move (in Relocator)
A rather guessable one, this will use two clip points to calculate \move coordinates for signs with \pos, so it's an easy tool for setting \move for multiple lines. By default there are no times in the \move, but if you check "times", you get times for the first and last frame. If selected lines have different timing, each will get the right timecodes. If you use Teleport fields, you can get timecodes from that.

For some reason, you can only draw two clip points with more lines selected, and at the third point, the selection drops and you're drawing in only one line. You can either use Alt and draw in all at the same time, or you can make the clip in the first line and then use Necroscopy or Cloning Laboratory to copy it to the others.

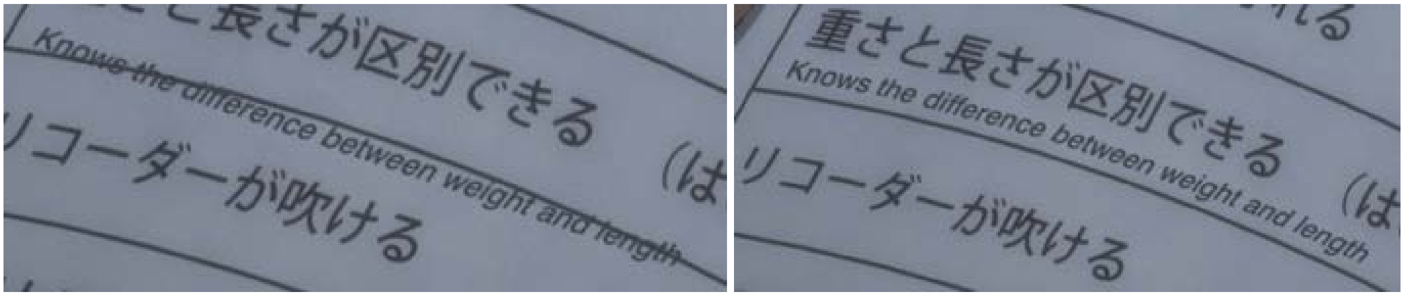
create shadow from clip (in HYDRA)
If you need a shadow in a different direction than the default, you can use a clip to set the direction, instead of tinkering with the x/y values manually. Obviously, the direction will be from the first to the second point of the clip. Shadow distance is taken from a \shad tag or from style if no tag present. (It also sets a value of 4 if the value found is 0. You can easily change the size using Multiply in Recalculator later.) The resulting shadow is created with \xshad\yshad.

size transform from clip (in HYDRA)
This is kind of a mocha-alternative for zooming, as long as it's linear. The goal is to get \t(\fscx\fscy). So you pick an object in the video that's getting larger or smaller, something that you can match the size of easily with a clip. Use 2 clip points to match the size on the first frame and 2 points on the last frame. The script calculates the ratio between the 1-2 distance and 3-4 distance, checks the starting scaling, and calculates the final size. With linear zoom, this can be a pretty quick solution without having to use mocha.
If it needs a little bit of tweaking after that, you could use "line2fbf" from Relocator and play with that.

By the way, if you do use mocha and the movement/zoom is linear, there is an option in the Motion script to apply the data as linear, creating \move and \t instead of many lines. This is actually quite nice because you don't clutter the script with hundreds of unnecessary lines.

convert clip to drawing (in HYDRA and Masquerade)
This is pretty essential, as it is the best tool to quickly create masks of any shape. Instead of using ASSDraw, you just draw a vectorial clip right in the video and convert it to a mask/drawing. Conversion is available both ways in HYDRA, so if you have a mask that needs some adjusting, convert it to clip, adjust, and convert back. (Relocator also has an "adjust drawing" function that lets you adjust the clip without removing the drawing.)

split by clip is a new option in NecrosCopy's Split by \N. When splitting by spaces (words), you can make use of a clip to easily bend the text word by word. It's described in detail in the Alignment section.



It takes only a few seconds to get from the straight line on the left to the bent one on the right, and it doesn't clutter the subs with too many lines like when you split for each letter.

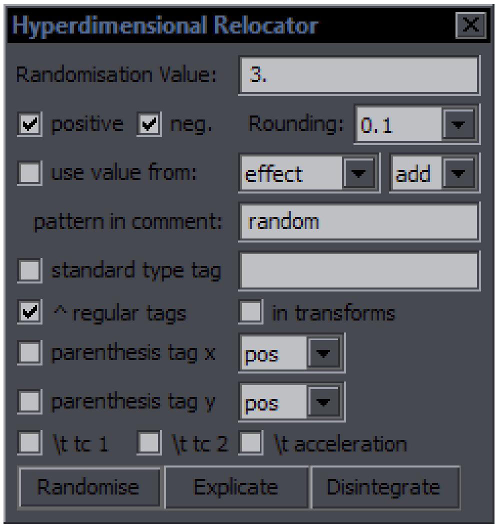
In the latest script versions, "clip to (frz/fax/reposition/move)" also works with hidden clips and iclips. This makes it possible to see the before/after states with Ctrl+Z/Ctrl+Y. (Otherwise the clip is likely to make the text invisible.)
Hiding clips, now available in Script Cleanup and hotkeyable in Relocator, is an interesting thing in itself. It allows you to see the text outside the clip but also to save the clip for later use.

Randomisation

Randomised effects can come in handy either when there's some kind of a crazy sign like letters flying all over the screen or for things like song styling. It's not needed very often, but when it is, these tools can be pretty powerful, and creating something like that without them would be tedious as hell.

Relocator - shake: This shifts position by a random value within a given limit. It's meant for fbf lines with the same pos tag. On each line, the position will be slightly different, creating a shaking effect.

Relocator - randomove: This randomly changes start or end coordinates of \move. A line split into letters can then either scatter in different directions or converge from various places to form a solid line.


Hyperdimensional Relocator

Randomisation Value: 3.

☑ positive ☑ neg. Rounding: 0.1

□ use value from: effect add

pattern in comment: random

□ standard type tag

☑ ^ regular tags □ in transforms

□ parenthesis tag x pos

□ parenthesis tag y pos

□ \t tc 1 □ \t tc 2 □ \t acceleration

Relocator - randomise...: This fairly advanced utility can randomise values for X and Y values of pos, move, org, clip, and fad separately, as well as values for any tag with a simple number value, plus alphas, and additionally also \t times. It can randomise pretty much anything except colours. (Use Colourise for that.)

This has many uses. I explained how you can apply the same tag to each letter in the line and add transforms to each of those. Now you can use this to randomise the values of that tag either outside of or in the transforms (or both if you want). So each letter can trasform from same value to different ones, or from varying values to the same one.

It will also apply different values on each line, so use it for random fbf effects. If you use it for \fs, then every \fs tag encountered will get its own random value, whether on the same line or not.

Even if you use the same tag+transform for each letter in one line, randomising the \t times will create some interesting effects. It can also be applied to \t accel, but that's a bit tricky. Maybe with an original value of 2 and randomising by 1.5, thus getting a range of 0.5-3.5, it might work well enough.

There's additional functionality, for which we need...

Significance - random numbers: You can find this as "random" under "Numbers". It will generate a random number for each line. You set min and max values in Left and Right, and each line will get a number somewhere between those two. These can go to effect, actor, a comment in the form of {random: #}, or margins (with obvious limitations).
If you replace "random:" with "random1:" in all lines, you can create another set, or basically as many as you want, and use them for different purposes.

Back to Relocator's randomise... This can make use of those values generated by Significance. Instead of generating its own values, it can take them from Effect or wherever you created them. There's only one value per line, but you can use it for multiple things, if you want some consistency. Also if you need the same values on the same frame but multiple layers/signs, Significance can use the same value for each X lines.

You can use the stored values in two ways: either add them to the existing ones, or replace those. So if Effect is "2.5" and you have \bord3, you can either add it to get \bord5.5, or just set the \bord2.5.

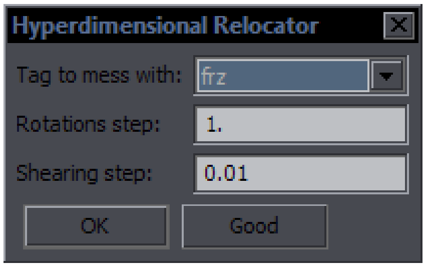
Significance - randomised transforms: This adds transforms to start tags. You can select a tag and min/max values. For colours, you set percentage of allowable change. You can also randomise duration of the line, affecting either start or end time. Combined with Relocator's randomise function, the possibilities are almost endless.

Micro-Management

There are a few things I've added recently to help tweak the smallest details. There are four Teleport-based macros in Relocator, each shifting in one direction (left, right, up, down). This is obviously best to somehow tie to arrow keys. I found that the default actions for Ctrl+arrows were pretty useless, so I kicked them out and set the Teleport macros under these (for Grid and Video). So now with Ctrl+arrows, I move signs by 1 pixel like in Photoshop, which, I must say, is pretty damn convenient because anything that requires moving a mouse by exactly 1 pixel is a fucking pain in the ass.

In continuation of going in that direction, I was recently tweaking a sign with frx and fry, and that shit is even more of a pain in the ass, because not only are you trying to change frx by 1 degree but also to NOT change fry at the same time, and since the tool handles both, this tends to be supremely annoying, especially since Aegisub has a little tendency to crash when messing with these rotations and remote org point. So I needed some kind of a plus/minus function for rotations that would change them by 1 like the Teleport macros.

I ended up with creating two macros in Relocator, called *Positive Spin* and *Negative Spin*. Positive adds 1 to a rotation, Negative subtracts 1. Or some other number you choose. To set it up, you need to open *Spin Doctor* in Morphing Grounds.


Hyperdimensional Relocator

Tag to mess with: frz

Rotations step: 1.

Shearing step: 0.01

Here you select what rotation you want to affect with the two macros, and you can change by how much it will spin in each direction. I added fax and fay since that kinda offered itself, and for those there must be a different step, so that's the 0.01.

After some testing, I thought of using HYDRA's Arrow Shifter to offer another option to mark tags that should be affected. This way you can switch between frx and fry more quickly, especially after I've tweaked Arrow Shifter to shift back as well (using the {switch} from Cycles). So whichever tag is pointed at, like >\frz, is the one the macros will affect. If the arrow points at something irrelevant, like alpha, the macros default to whatever is set in Spin Doctor, or frz if nothing.

At one point, I made a mistake while writing this and found that when pointing at bord, the macros were changing that. I thought about it for a while, about how that might work, and did kind of a compromise. I didn't add tags like bord and blur, but adding ones that work with negative values was trivial, as it basically meant just adding the tags on the list, so there's also xshad, yshad, and fsp. I decided adding bord and others wasn't necessary, as those are handled by the Cycles script pretty well anyway (and I would have to do something about not allowing negative values for some tags).

The small step is only for fax/fay; the regular one is for everything else. After some experimenting, I'd say using the Arrow Shifter is more convenient that switching with the Spin Doctor menu. You can't change the step this way, but that should hardly ever be necessary. And if you sometimes want to change value by 0.5, it may be easier to just go for Recalculator for that one half step.

Then I was trying to change \fsp for one letter in the line, and I realised that if I have fsp in start tags, it changes too. This was intended, because if you have a gradient for frz or fax, I want the spin to change all of them, but here I wanted to affect just one specific tag. So I've made the distinction that if a specific tag is pointed at with the Arrow, only that one is changed (which I think makes a lot of sense), while using the settings from Spin Doctor will change all tags of the selected type in the line. If you only have one frx and one fry in start tags, then using the Arrow to switch between them still works fine. This then gives us an extra option to affect one tag or all.

So mainly you just have to get used to all the new hotkeys, and things go pretty smoothly after that.

As I've already hinted at, the Cycles script has a *Switch* function, which puts a "{switch}" comment at the end of a line. This is primarily used by Cycles to go backward in the sequences instead of forward. The switch is an on/off function that adds or removes this comment, and since that works like a binary marker for two different states of a line that doesn't really interfere with anything else, I used it for some other things too. (I started with commenting the line as an on/off switch, but then you can't see it in the video, and while you can type things in Effect to mark the line, this requires clicking into Effect, typing, and clicking back into Grid, plus you might need Effect for other things, so I went for adding that comment in a way you can hotkey.)

So there's the backward sequence in Cycles, there's reversed direction for Arrow Shifter, and there's Masquerade's Shift Tags, which shifts tags for multiple lines by one character to the right, and when the {switch} is on, by 1 to the left. The switch aso works for Bell Shifter and Wave Shifter, switching from word mode to letter mode.

Many Ways to Do Things

The other day somebody said he doesn't know how to set fixed position (like 640,360) for a number of lines with Relocator (or potentially another script). True, Relocator doesn't specifically do that in the regular way of setting "640" and "360" for X and Y, but I could think of a few ways of doing this, and the more I thought about it, the more ways I could come up with. So I'll use this as a general example to show how many things can be done in many different ways.

So the issue here is to set "\pos(640,360)" to all selected lines. I'll list a number of options, more or less in order of usefulness.

1. Relocator - Cloning Laboratory. You'd set the position for the first line by just double-clicking on the screen, and this will copy the pos tag to the other lines. This is what I normally use to set the same position for selected lines.
2. NecrosCopy. This is pretty much the same - copy from line 1 to the others, selecting the pos tag from the list.
3. Use Script Cleaner to delete pos tags, and then just double-click on the screen to set pos for all selected lines. This works even when they have different timing. (You just have to be somewhere in the video the active line is timed to.)
4. Use HYDRA's Additional tags to simply type "\pos(640,360)". This requires typing, but it would be the most

straightforward way of "setting a fixed position for selected lines". (There won't be any problem with existing pos tags - they get overwritten by the new ones.)

Now for some more complicated and less useful solutions, just to show that there are options. The guy said he actually used regexp replacement to do it. That's certainly a way to do it too, but I'd still go for these before that.

5. Relocator - Align X and Align Y. Two ways to do this. Either with "by first", you align the other lines by the first one, so it's like Cloning Laboratory, but for X and Y separately, or you can actually set "640" for X and "360" for Y in the disPosition field, but you must uncheck "by first". This is very straightforward as well, but it's in two steps.
6. You could use MultiCopy to copy the pos tag from one line and paste it into others. You have to use the "any tag" option, type "pos" in the big textbox, copy the result, paste it with the "any tag" option, and select "loop paste" so that it goes into all the selected lines.
7. Recalculator - First Multiply pos to 0%. This gives you pos(0,0). Now Add "640" for pos X and "360" for pos Y. Obviously silly when you have the options above, but still workable.
8. You could even use Relocator's "Find Centre". This sets pos to the centre of a rectangular clip, so if you draw the clip really small (with Alt for all lines), you would get the position with enough precision, I guess.

So that's eight ways to do this, excluding the regexp replacement. HYDRA doesn't have checkboxes for all tags, but with "Additional tags", you can apply any tag you want to all selected lines if you just type it. In fact, it can even add fake tags like "\derp666", as it doesn't check the additional tags for anything except that there's a backslash.

There is also **this link**, which can be useful sometimes when you're trying to figure out how to do something specific.

Even after the 3 years that my scripts went unchanged, I see people asking "is there a script that can do ...", and most of the time, one of my scripts can. Heck, sometimes even I wonder how to do something specific easily, only to find out that I wrote a function for exactly that 4 years ago. The point is, most things you're looking for indeed do already exist.

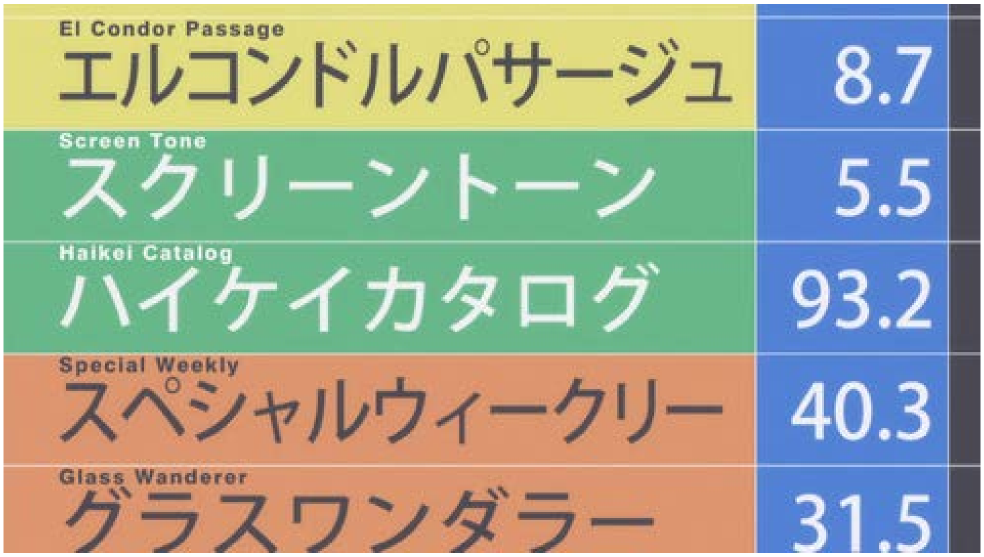
Adding More with Each Line

Sometimes you need to increase values line by line in regular intervals. There are many tools for this.

HYDRA has "Add with each line". So if you use \bord2 and add 1 with each line, you get \bord2 on the first line, \bord3 on second, \bord4 on third, etc. This works with all the regular number-value tags.

Relocator has Warped Teleport, which does a similar thing. If you use 0, 0 as regular Teleport input and 5, 0 in Warp, the first line stays where it is, second moves 5px right, third 10px right, etc. Similarly, you can use Teleport input with Cloning Laboratory to space out lines in regular intervals. If you use 0, 30 and clone position, line 2 will be 30px under line 1, line 3 60px, etc. While with Warped Teleport the starting point is each line's original position, with Cloning it's the *first* line's position for all lines.

Recently, I had this to typeset:



It was pretty clear that all lines had the same height, so I measured the distance between two horizontal lines, got that it's 146, set the first line where I wanted it, and used Cloning with 0, 146 in Teleport. All lines were immediately where they needed to be. This kind of stuff can save you a lot of tedious tinkering with things.

Another script that uses this is Recalculator. In the example above, I could place all lines at the same position as the first one and use Recalculator with this setting. I'd select all but the first line, type 146 into 'Increase values by', check pos y, and the 'add more with each line' option. That would give me the same result as you see above.

Recalculator can do this with almost any tag and has x and y coordinates separate, so there's a lot that can be done with that. It can even be applied to layers or timing.

To be continued...

« Back to Typesetting Main