Hello there!

Thanks for making **PixelBattleText** the new inhabitant of your project :) Here's how to take advantage of it to animate pixel perfect damage counters, altered states and level ups.

First, take a look at the sample scene. On it, you will find a preview of the animations included in this asset pack.

Open the sample scene, you can find it at:
Assets > **PixelBattleText** > **Example** > **DemoScene**

Hit Play and BE AMAZED!



Demo scene after you press the LEVEL UP button.

Getting Started

Now that you have experienced the beauty of the powerful and captivating animations that **PixelBattleText** can create, lets learn step by step how to put together a scene capable of displaying such marvelous contraptions.

Create a new scene and follow the instructions below:

- **1-** Create an empty object and name it *PixelTextController* (The name is not important; it is just to keep things organized)

- **2-** Select the new game object. In the inspector, press the **Add Component** button. Find the **PixelBattleTextController** component and click it.

Note: There must always be a **PixelBattleTextController** in the scene for all animated text to work.
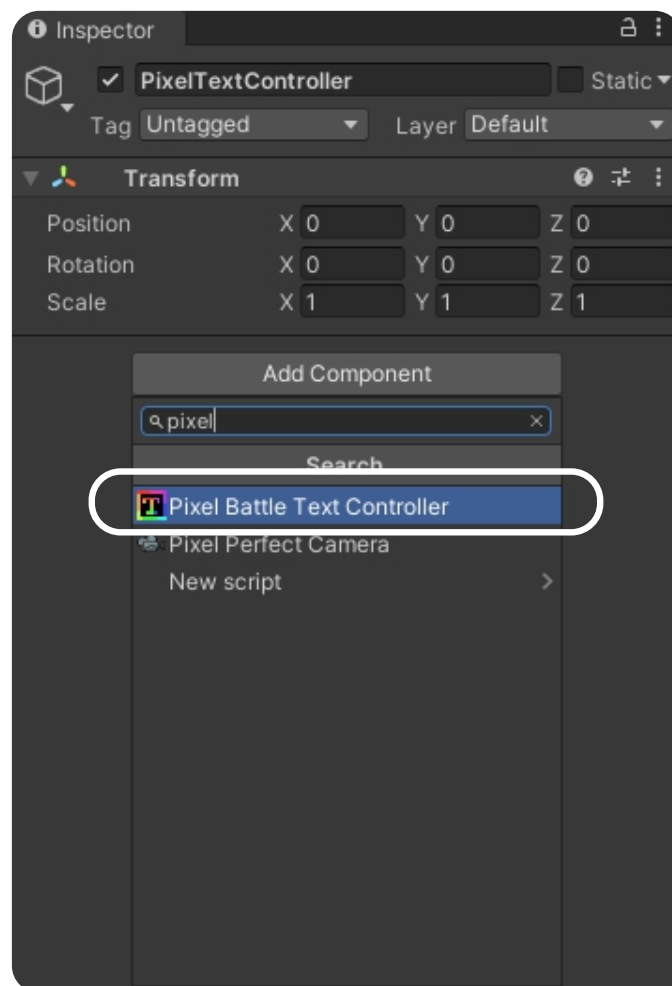
- **3-** Create a Canvas in the menu: **GameObject** > **UI** > **Canvas**, and drag it to the corresponding field in our ■**PixelBattleTextController**.
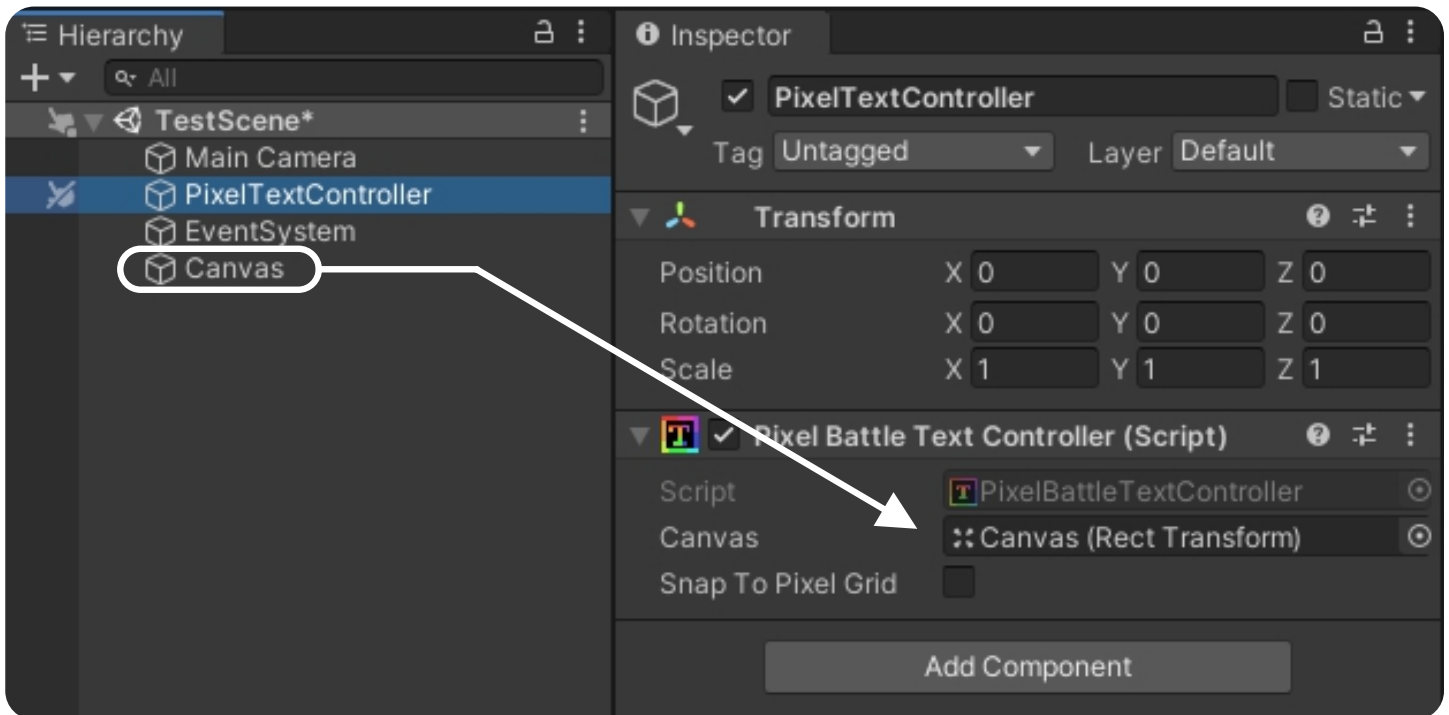

If we try to show pixel perfect text using this canvas as it is, the letters would look small. An 8 pixel tall font would take up exactly 8 vertical pixels of your screen (wich is very, very small)

- **4-** We need to change the ■**CanvasScaler** in the **Canvas** game object.
  - Change **UI Scale Mode** drop down to **Scale With Screen Size**.
  - Set **Reference Resolution** to **320**x**180**.
  - Change **Screen Match Mode** drop down to **Match Width or Height**.
  - Set the **Match** slider to **1** (full height).
  - Set **Pixels per Unit** to **32.**
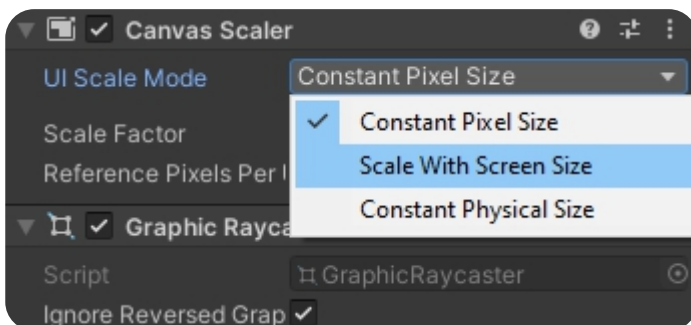
*1- Change the **UI Scale Mode** to **Scale With Screen Size.***
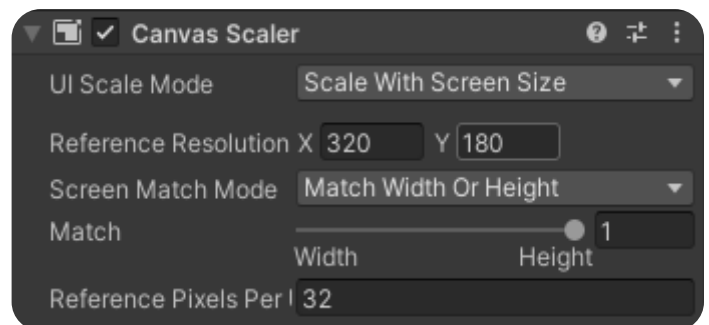
*2- Set your **CanvasScaler** component settings to match these.*

- **5-** Create a script from which you will call the controller. You can do it directly in the same *PixelTextController* game object by pressing *Add Component* and choosing the option *New script*. Name it "*TestingPixelBattleText*" and press *Create and Add.*


- **6-** Open our new script and replace it's content with the following

```csharp
using UnityEngine;
using PixelBattleText;

public class TestingPixelBattleText : MonoBehaviour {
    public TextAnimation textAnimation;

    void Update() {
        if(Input.GetKeyDown(KeyCode.Space))
            PixelBattleTextController.DisplayText(
"Hello World!", textAnimation, Vector3.one * 0.5f);
    }
}
```

- **7-** Now your script should have a public field called *"Text Animation"*. Touch the circle to the right of the field and choose a 🖼 **TextAnimation** preset. (You should already be familiar wiht the names as they are the same as in the demo scene)
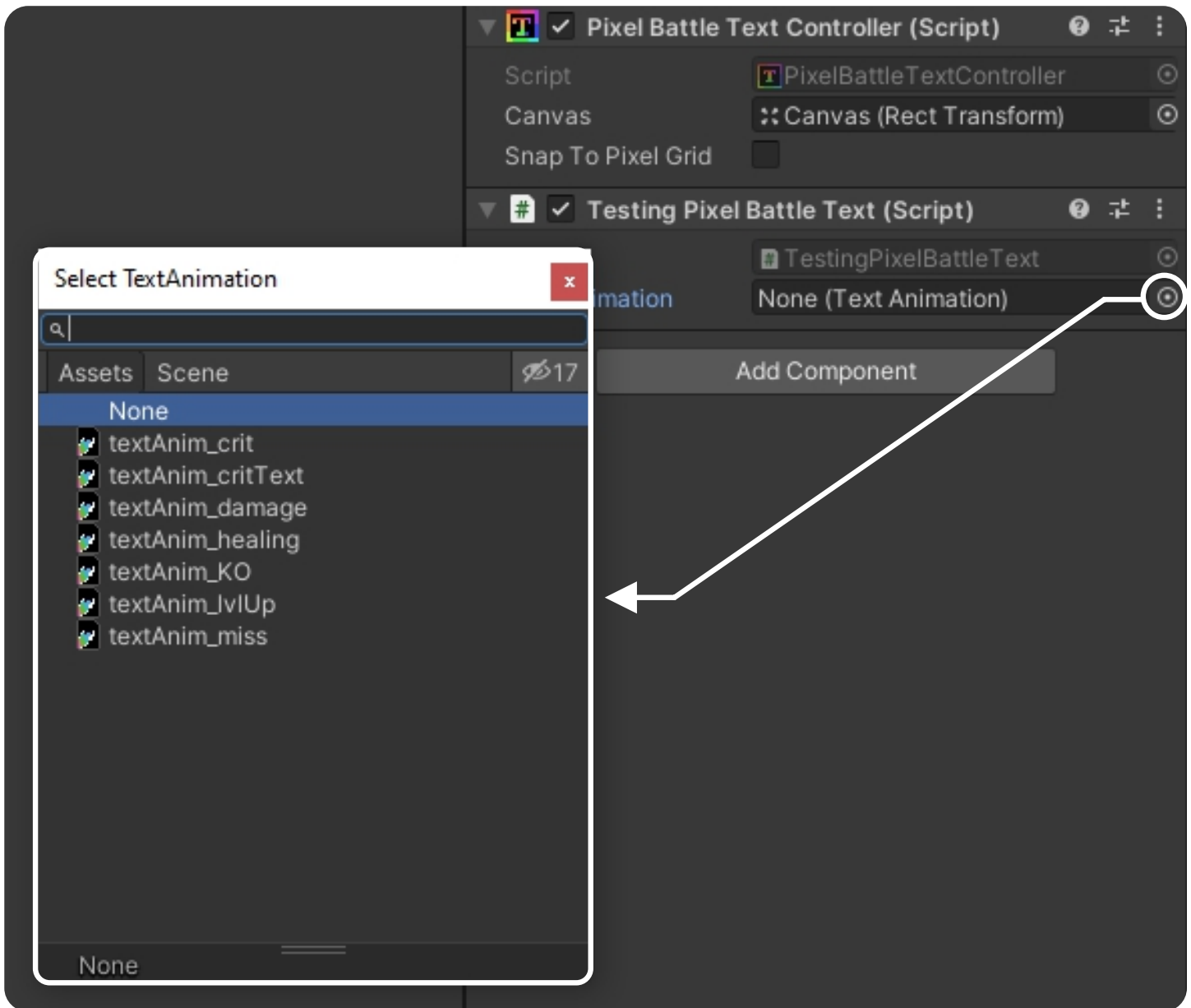
- **8-** With your 🖼 **TextAnimation** attached and ready, press **Play** in the Unity editor and hit your space bar. Pixel perfect text with the words *"Hello World!"* will appear in your **Game View**!

> **Quick Tip**: (You can change the "**Text Animation**" field while the game is running and preview the change by pressing space bar)

Understanding 🅣PixelBattleTextController

PixelBattleTextController is the component responsible for displaying the sexy text animations you saw in our demo scene. There has to be a 🅣**PixelBattleTextController** in the scene in order for the text to be displayed.

This chapter works both as a manual for understanding the right way for using **PixelBattleTextController** as well as its documentation.
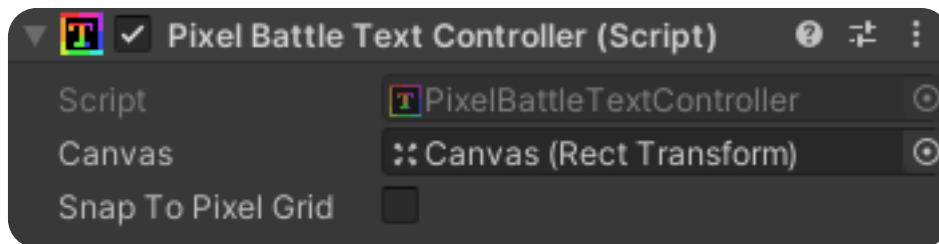
This is how the component looks

    - **Canvas:** Is the ✖**RectTransform** that the component will use both as the parent object for all spawned **animated texts** and as the base scale for the animation, as that specific ✖**RectTransform** pixel grid will be used for the positioning scale.

> **"Canvas"** field **must not be left unassigned** when trying to spawn text or it will show a **warning** and wont work (this only happens in **Editor Mode**, in **production** it will directly throw an error and **crash**) ⚠️

> Even if we directly uses the ✖**RectTransform** from the root 🎬**Canvas** in the demonstration in the **Getting Started** chapter. The ✖**RectTransform** of the **"Canvas"** field does not have to be the root of the scene's 🎬**Canvas**. It can be any of its children too! This is useful when you want your animated text to appear on front of some UI elements and behind others.
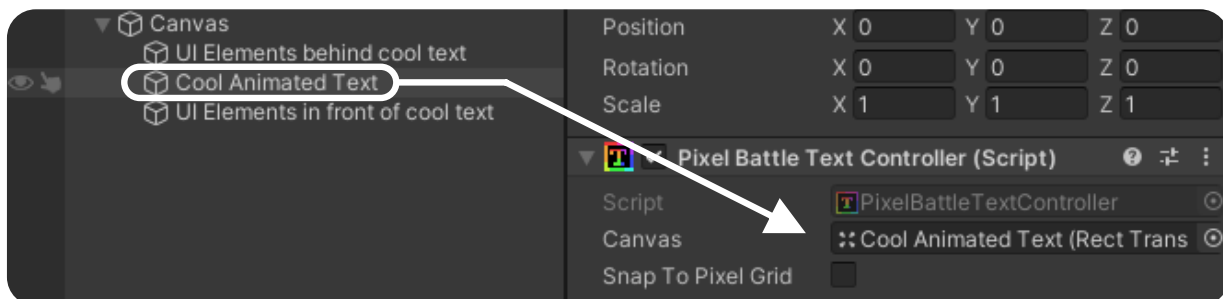
- Snap To Pixel Grid: This toggle changes 🔳**PixelBattleTextController** to "throw away" decimal values through text position calculations. This makes our text snap exactly into its parent ✖**RectCanvas** pixel grid. It is useful for chopping animations looking "too smooth" if you go for a more realistic, retro style.

Now we know all about 🔳**PixelBattleText**'s properties, lets focus on its one and only function :)

• **Display Text**: Displays and animates an efimeral text UI element at a given position in "Canvas" space. It receives the next data as parameters:
• **Text**: The string to display.
• **Style**: 🖼**TextAnimation** with parameters for animating every letter.
• **Position**: Position for where to display the text (in canvas space)

Its a **static** function so all you need to make text appear is to call the following:

```
PixelBattleTextController.DisplayText("meh", animation, position);
```

from anywhere in your scene.

The position we give to the 🔳**PixelBattleTextController** in **DisplayText** is *"normalized"* and *"relative"* to the **"Canvas"**. Wich means that the lower left corner is (0,0) and the right top corner is (1,1). If we give the function the position (0.5, 0.5) the text will be displayed in the center of the ✖**RectTransform**.
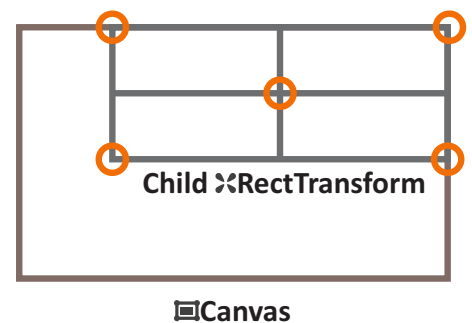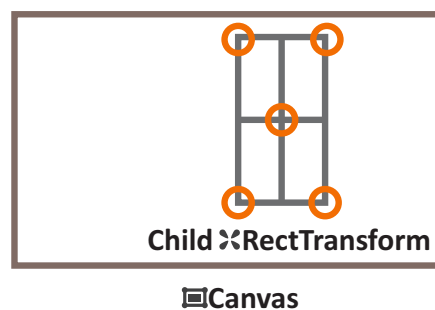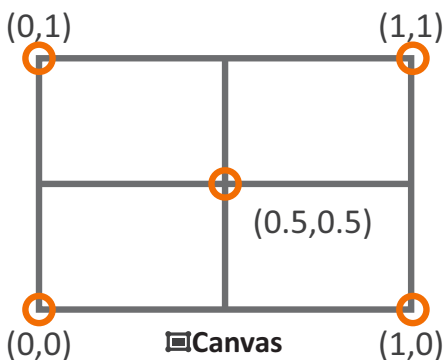
Making your own pixel perfect 🄵 Font

Having a library of effects, a powerful, efficient and simple animation controller, a correct pixel scale in our canvas and talent for UI design, is not enough to achieve perfection. You need a font file with the right properties for our text rendering to be truly stellar.

You can find plenty of "pixel" style font files in a google search. For transforming this "raw" state font into an asset we can reference in a 🄵 **TextAnimation** we have to use the **Font Asset Creator** Window. You can open it at:

> **Window** > **TextMeshPro** > **Font Asset Creator**

You should see a window like this:
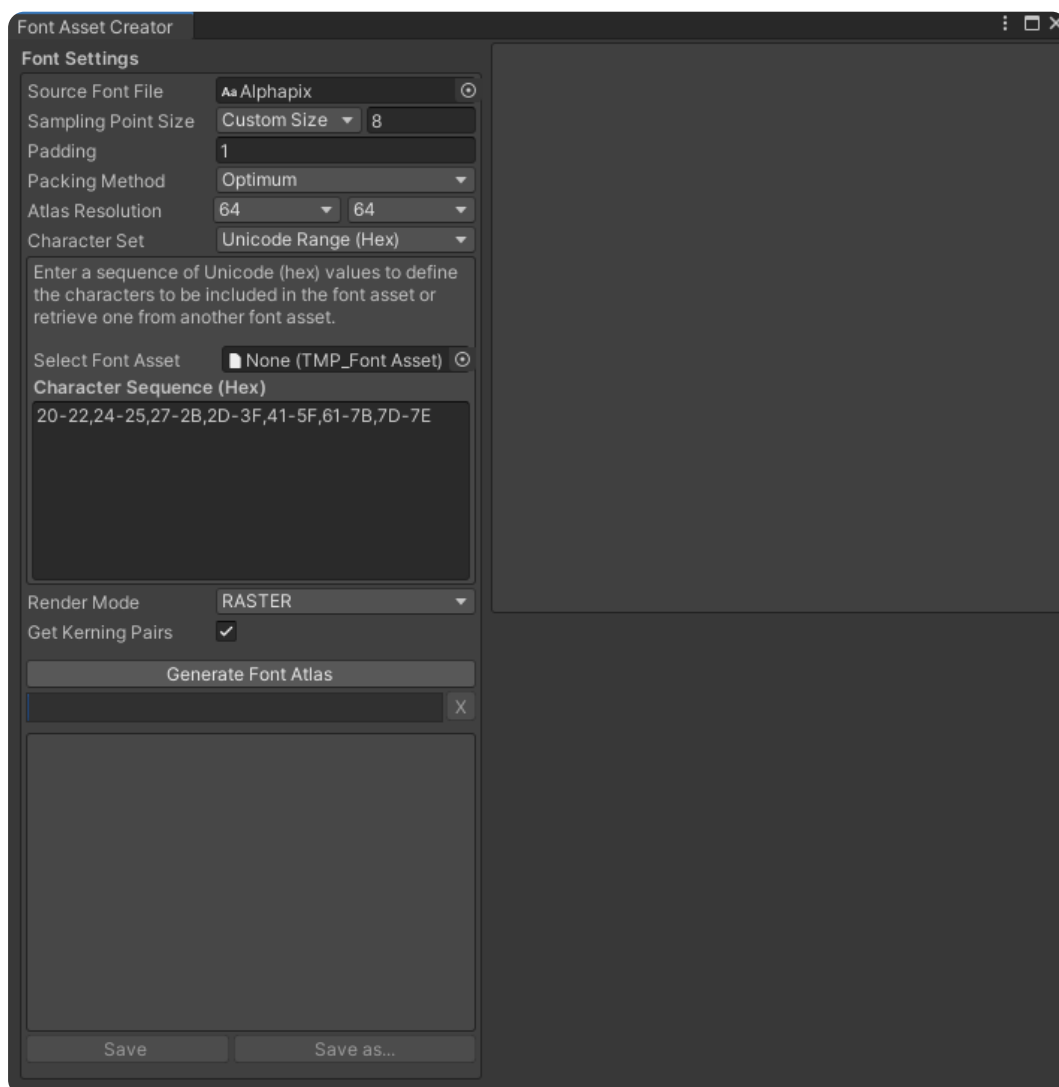
Now that we have the window open, is time to set its parameters for generating a crips, pixel perfect font file:

- - Set the **Source Font File** to "Alphapix" (or your google search result)
- - Set **Sampling Point Size** to eight(**8**) (This has to coincide with the pixel height of the font file. The space occupied by the height of a letter not only takes into account the visible part of it, but also the space below the baseline and above the top of the letter, which can be occupied by accents and tall symbols)
- - Set the **Padding** to one(**1**) (This is the spacing in pixel from one letter to another in the texture map we will generate. One pixel should be enough)
- - Set **Packing Method** to **Optimum**
- - Set **Atlas Resolution** to **64x64** (This has to suffice for all characters, for Alphapix 64x64 is enough. Keep it squared, different values in X and Y components will deform the font)
- - **Character Set** should already be set to unicode by default.
- - And MOST IMPORTANT! Set the **Render Mode** to **RASTER**

Finally hit **Generate** and a pixel font atlas should greet us in crisp black and white! Click **Save As** and save the file in a suitable place in your project. You have now a crisp font with no soft borders.

Always make sure all your characters have made it to the texture atlas you generated. If the resolution is too small for the entire font to be rendered **some characters may be missing**. No alert will be displayed to notify you, but you can watch for it in the additional info display in the **Font Asset Creator** window. In case any characer is missing, just set the **Atlas Resolution** property one step higher.
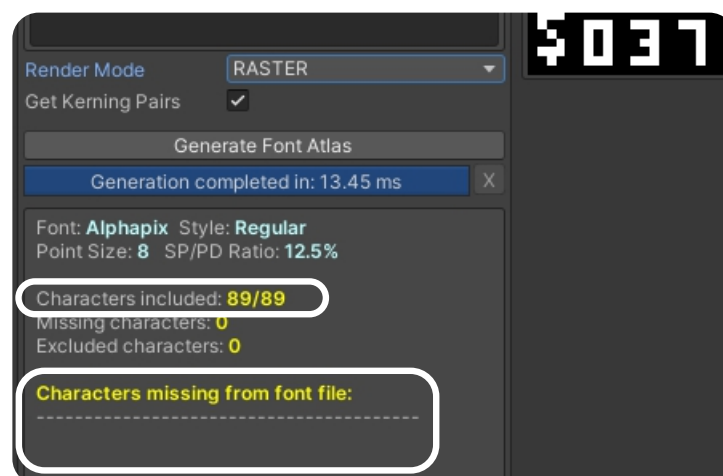
You can se the character total here, and check if where included.

Making your own 📄TextAnimation preset

In case our juicy animation presets are not enough to express your game developer creativity, you can always make your own ;)

You can create a 📄TextAnimation asset in the **Project** tab by:

Right Click > **Create** > **PixelBattleText** > **TextAnimation**

You can easily recognize **a TextAnimation** asset by its icon:

As you can see in the figure below, the 📄TextAnimation properties are divided into several groups: **Font Controls**, **Time Controls**, **Spacing Animation**, **Offset Animation** and **Color Animation**.

In this short chapter we will learn the role of each field in a 📄TextAnimation asset and how to "calculate" pixel font pivots in order for better understanding pixel perfect rendering for text.
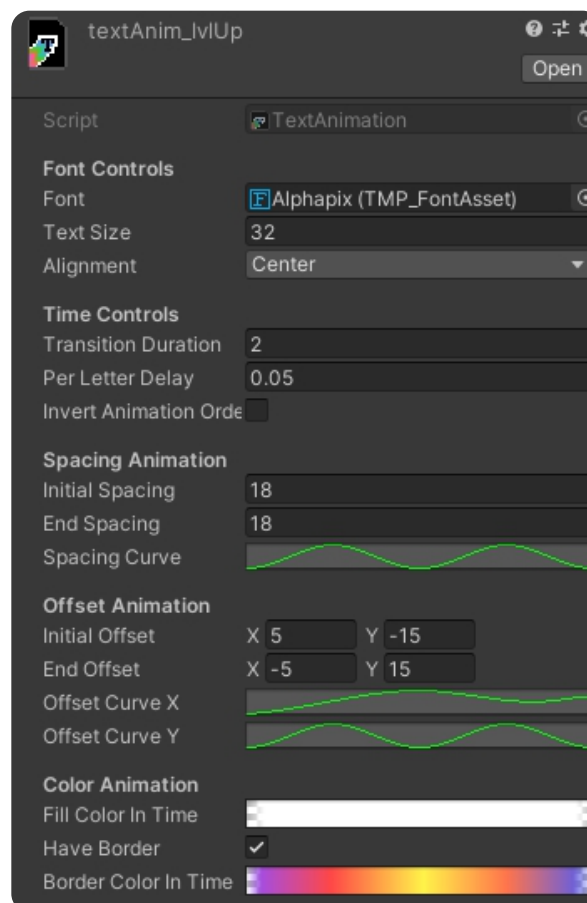
- Font Controls

- **Font**: The **TMP_FontAsset** file for displaying in this animation. this allows to have different animations, each with a different font.

- **Text Size**: The size of the font to display. It changes the **Size** property in the **TMP_Text** component used to represent the animated text

> **Size** changes the height of the font. This includes the empty space below the font's baseline or above the letter. **Size** is set in "canvas pixels", independent of the actual pixels the font would take on screen. The border will always take one (1) "canvas pixel" on each side, independent of the font size.

The pixel border is kept constant, independent of the font size

= 1px border

Size = 8
8 pixels high

Size = 16
16 pixels high
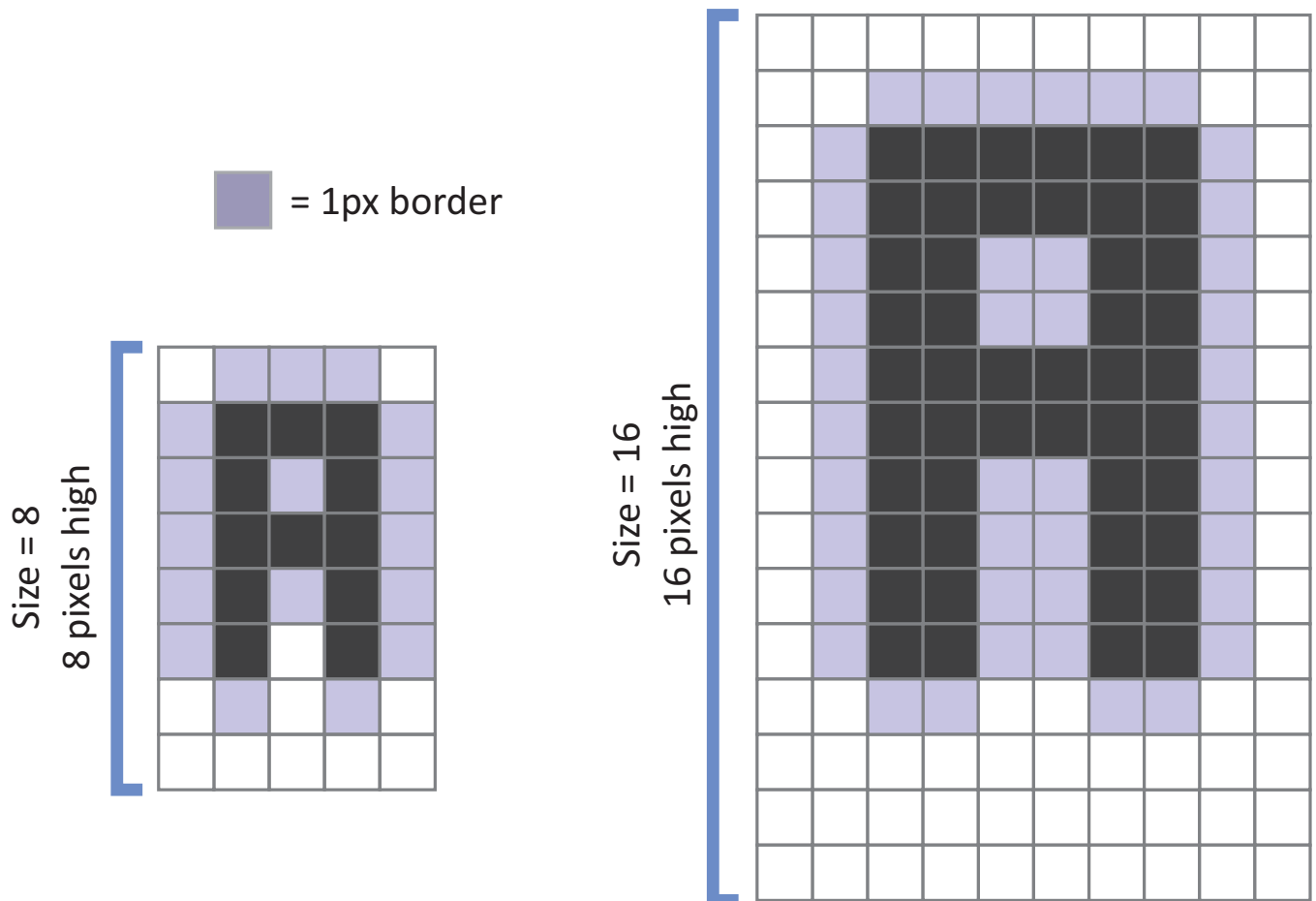
- Alignment: It controls the horizontal alignment of the font. and the position of the font pivots.

= letter pivot

Right Alignment

Center Alignment

Left Alignment
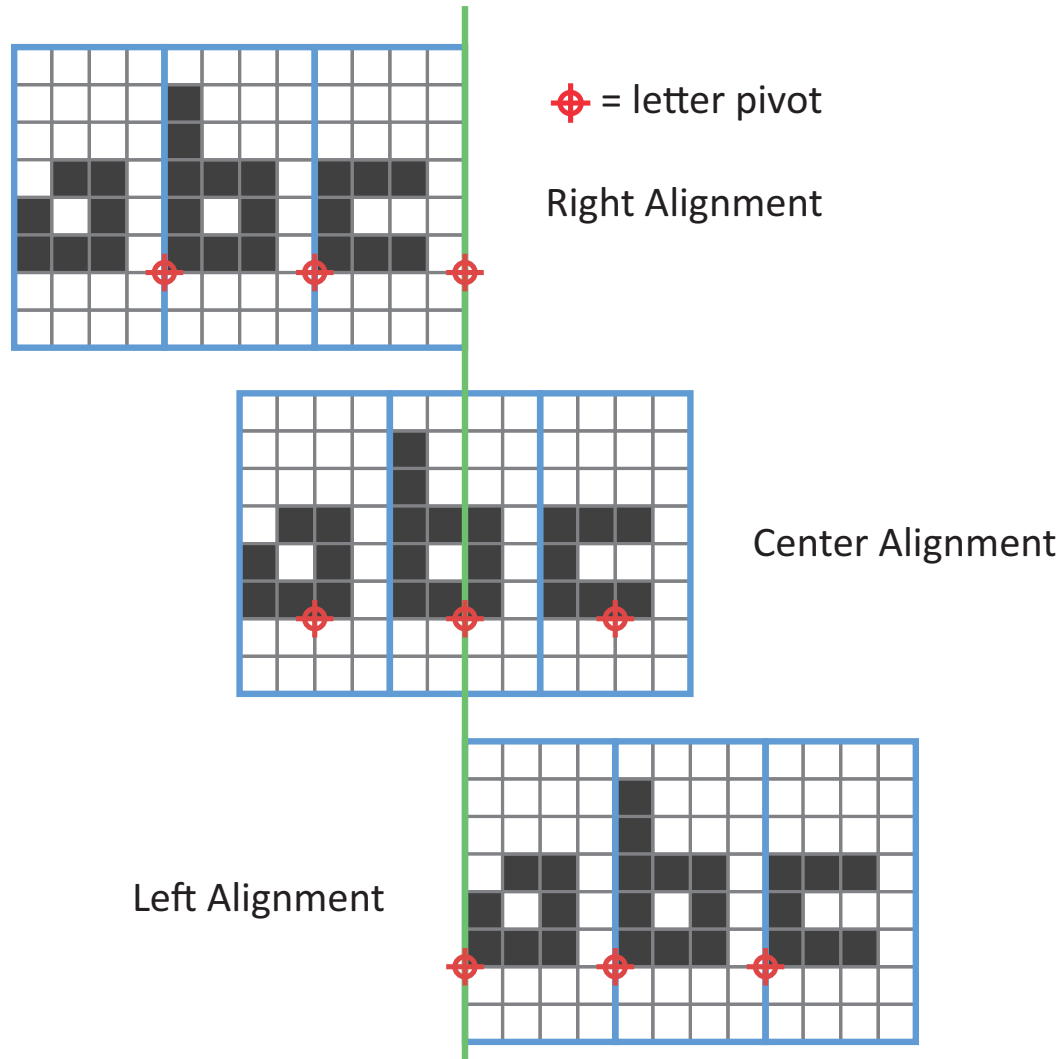
- Time Controls

- Transition Duration: The time it takes a single letter to finish the animation.

- Per Letter Delay: The time it takes for each letter to begin its animation. Each letter must wait for the delay of those in front to end plus its own, to start its animation. This is independent of the duration of the transition. Each letter can start its animation as soon as its delay ends, even if other letters ahead have not finished their animation.

- Invert Animation Order: Determine if the order in which the letters proceed to animate starts with the first or last. It is always taken by "first letter", the one that is further to the left. Regardless of the lineup. Reversing the order of the animation would therefore cause the rightmost letter to always start animating first.

- Spacing Animation

- **Initial Spacing**: Determines the initial space (in canvas pixels) between the pivots of the letters. This space is taken in the animation as the starting point for the position of the letters.

- **End Spacing**: Determines the final space (in canvas pixels) between the pivots of the letters. This space is taken in the animation as the end point for the position of the letters

- **Spacing Curve**: Determines the progress of the animation from initial spacing to end spacing following a curve from 0 to 1, where 0 is the **Initial Spacing** and 1 the **End Spacing**.
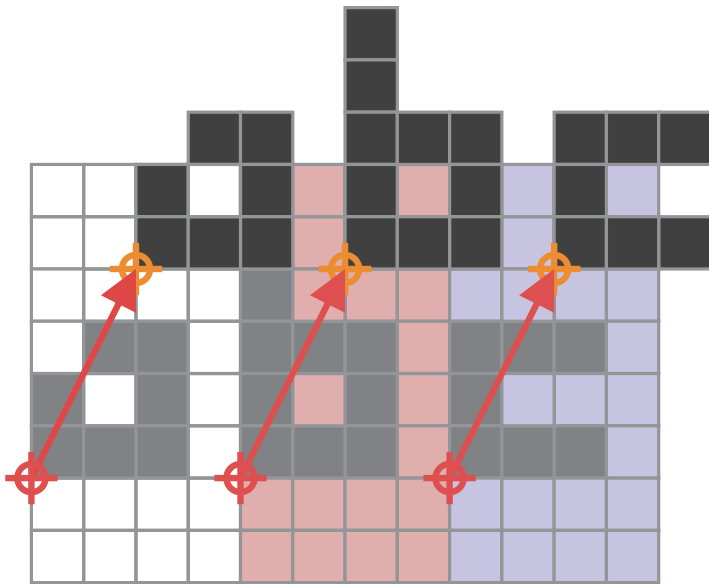

animation

4 px spacing

2 px spacing
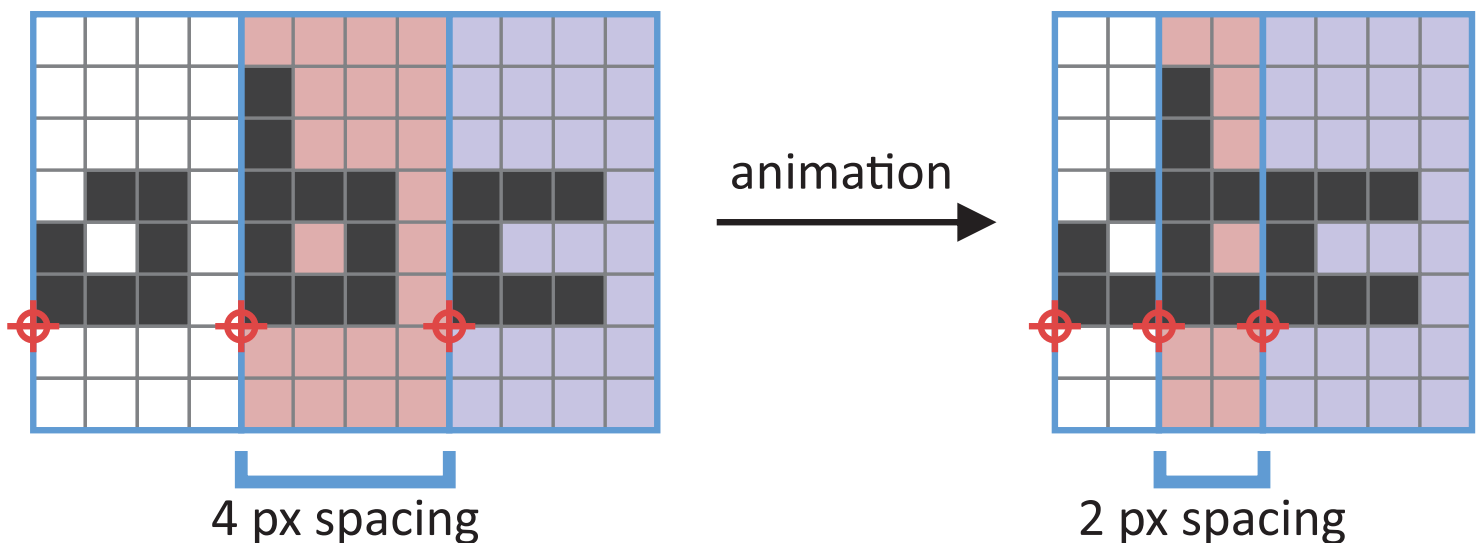
- Offset Animation

- **Initial Offset**: Determines a vector (in canvas pixels) to be added to the pivot at the begining of the animation.

- **End Offset**: Determines a vector (in canvas pixels) to be added to the pivot at the end of the animation.

- Offset Curve X: Determines the progress of the animation from **Initial Offset** to **End Offset** following a curve from 0 to 1, where 0 is the initial spacing and 1 the end. This only affects the position's X factor.

- Offset Curve Y: Determines the progress of the animation from **Initial Offset** to **End Offset** following a curve from 0 to 1, where 0 is the initial spacing and 1 the end. This only affects the position's Y factor.

Offset changing final position of letters with no delay.
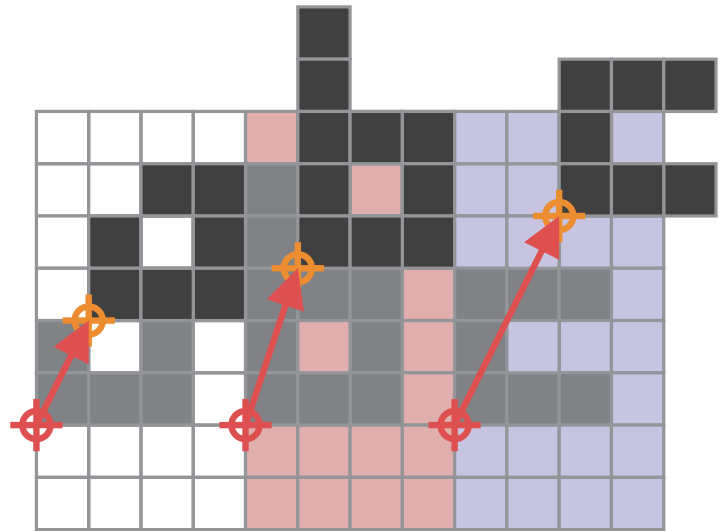
Offset changing final position of letters at different moments in time beacuse of letter individual delays.

- Color Animation

- Fill Color In Time: Determines a color gradient that represents the changing of the letter´s inner color during the animation.

- Border Color In Time: Determines a color gradient that represents the changing of the letter´s border color during the animation.

- Has Border: Determines whether the edge of the letters should be animated or not even displayed.

FAQ

Q: How do I make damage counters spawn over my mob's head?

```
A:
    float3 mobsHeadPosition;

    //transform world space position to viewport position
    float2 viewportPosition =
Camera.main.WorldToViewportPoint(mobsHeadPosition);

    //proceed normally using the new position
    PixelBattleTextController.DisplayText(damage.ToString(),
animation, viewportPosition);
```

Q1: Why does my font border looks wrong/oversized when I use small font sizes in my animation?
Q2: Why do my letters show cut out parts of other letters in the side?

A: Even when the font is small, the space that Text Mesh Pro generates around it in the atlas is somewhat constant. This makes possible for a letter to "bleed" into the UV space assigned for it by the **Font Asset Generator**.

You can fix this by simply generating the font asset again, but setting the "Padding" property of the atlas, higher than the value you used (two(2) pixels is enough in most pixel fonts)

Q: I call `DisplayText(text, animation, position)` but it doesn't spawns where it should.

A: If you are using a child ⚙**RectTransform** instead of the main **Canvas**, make sure your transform's pixel rect is ocupying all the available space of your viewport. Make sure to look at the graphs in **Undersanding PixelBattleTextController** chapter.

Aaaand... that's all folks

Now you have the necessary tools to make the battle text for your next Final Fantasy/Dragon Quest clone (or Undertale clone if you are one of those)

An Official
PSYCHIC FUR
Product