

# Flying Text3D

Thank you for purchasing FlyingText3D! Now you can easily and quickly create true, dynamic 3D text using TrueType fonts. It's as simple as writing `FlyingText.GetObject("Hello!")`. Or you can create prefabs using the FlyingText3D inspector utility.

You can import the FlyingText3D\_Source package instead of using the DLL if you want to work with the source code, but the DLL is generally recommended. (Make sure you don't import both into the same project, though, since that will create conflicts.) The DLL doesn't require Pro and works on any platform; think of it as a different way to package scripts. To get started, take a look at the FlyingText3D Inspector section, then dive right in — most of it should be pretty obvious, but this documentation covers everything in detail in case something isn't clear.

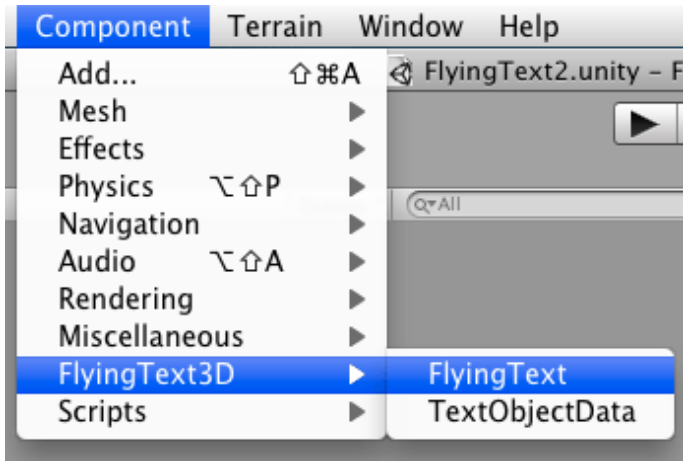
Also check out the demos to see some examples of how you might use FlyingText3D. Open the Menu scene, the GameOver scene, or the Input scene, and hit Play to see 3D text in action. The demos assume you're using the DLL, but if you're using the source, you can manually apply the FlyingText script to the FlyingText object in the scene.

Note that FlyingText3D requires Unity 3.5 or later. It should work on any platform, and can be used with any language.

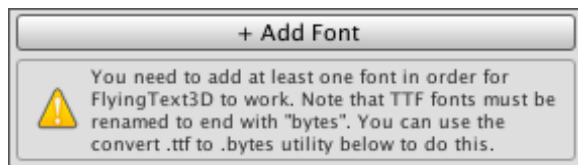
<a href="#">FlyingText3D Inspector</a> .....	2
<a href="#">Character Settings</a> .....	5
<a href="#">Text Settings</a> .....	10
<a href="#">GameObject Settings</a> .....	13
<a href="#">Tags</a> .....	14
<a href="#">FlyingText3D Coding</a> .....	17
<a href="#">Limitations</a> .....	24

---

The first thing you should do is attach the FlyingText script to an object in the scene. Frequently it's convenient to create an empty game object for this purpose. To attach the script, click on the desired game object, then select FlyingText in the Component menu.

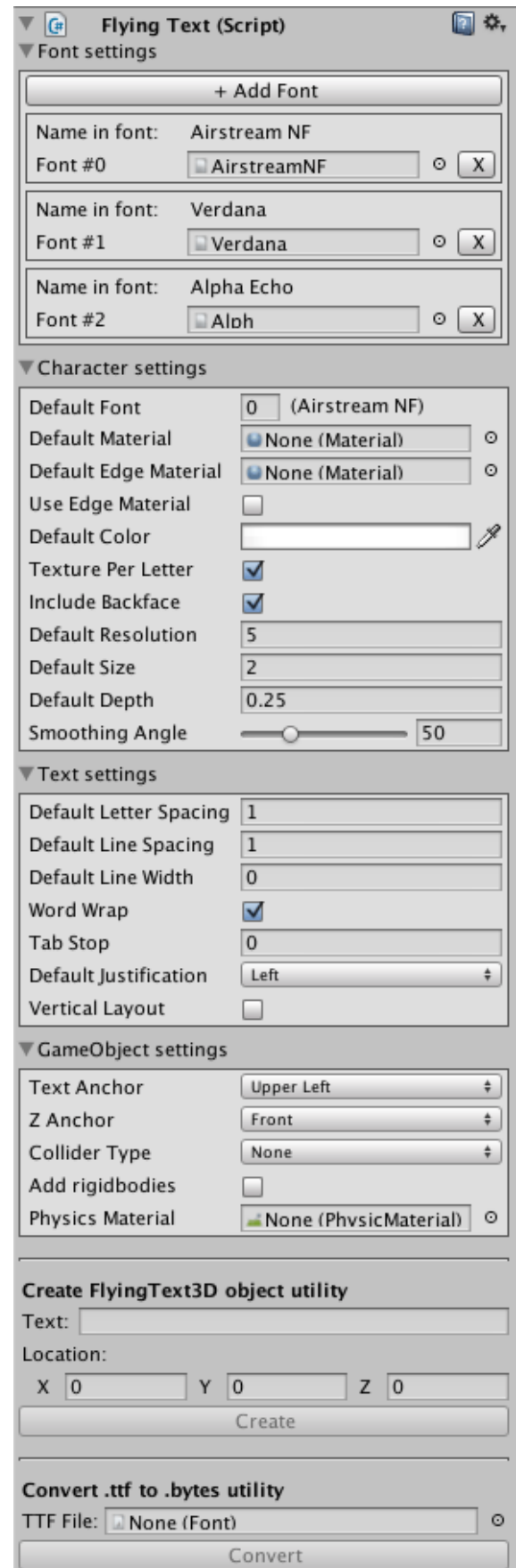


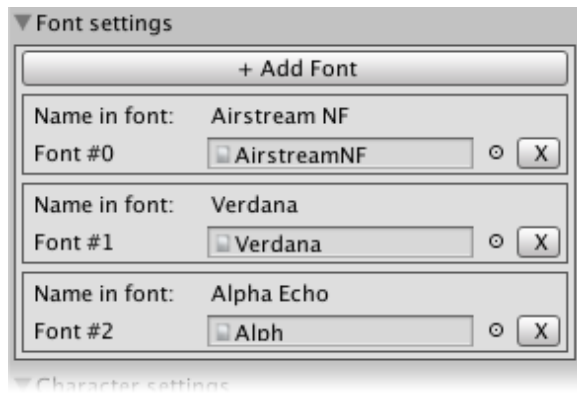
Now, we need some fonts! At first, the “Font settings” section will be empty, and it will look like this:



Click on the Add Font button, which will give you a Font slot that you can drag a TrueType font onto. An important note here is that unfortunately there doesn't seem to be a way to use a .ttf file directly; FlyingText3D needs to be able to use all of the raw data from the file, and Unity doesn't allow this when using Font assets. The good news is that it's just a matter of renaming the font to end with “.bytes” instead of “.ttf”. You can either do this outside of Unity, or else use the “Convert .ttf to .bytes” utility ([see below](#)).

If you rename the font outside Unity and are using a Mac, it's a good idea to make sure “show all filename extensions” is checked in the Finder preferences. Otherwise attempting to change “.ttf” to “.bytes” will likely result in files actually ending with “.bytes.ttf”, which won't work.

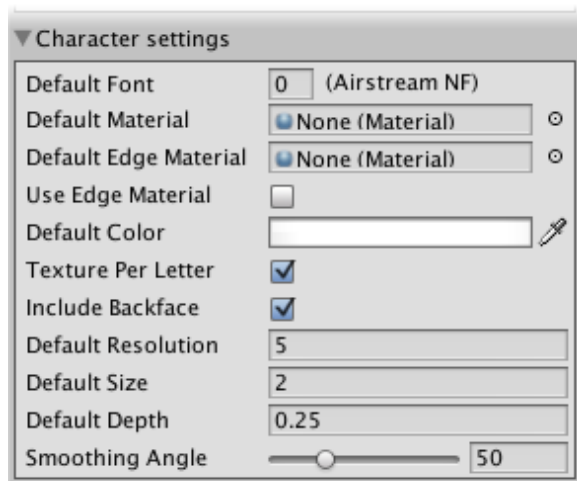




## Font settings

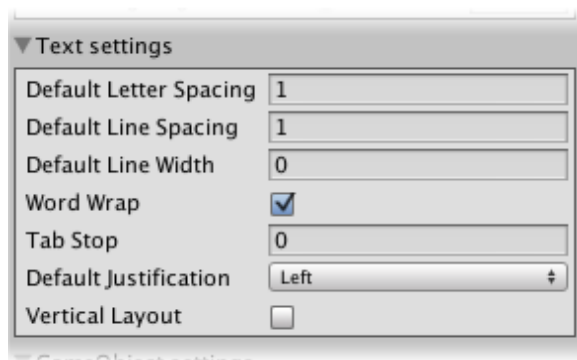
You can add multiple fonts: just click the “Add Font” button as needed, and drag additional fonts to the appropriate Font slots. The actual font name is displayed above the Font slot, since the file name may or may not be the real name. You can specify the font name when using tags (see the [Tags](#) section below).

The “X” button will remove the corresponding font from the font list.



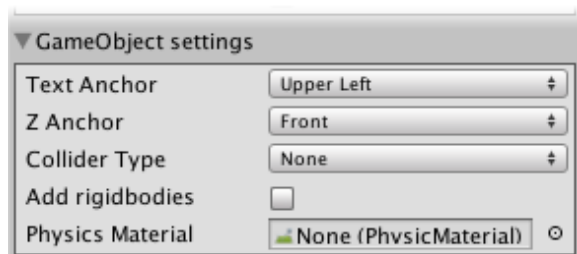
## Character settings

These are the default settings for 3D text objects that you create, which affects how the characters appear, such as what material they use, how big they are, and so on. Unless overridden with tags (see the [Tags](#) section), all text will use these defaults. They can also be set in code (see the [FlyingText3D Coding](#) section). The exact effect of each setting is covered in [Character Settings](#).



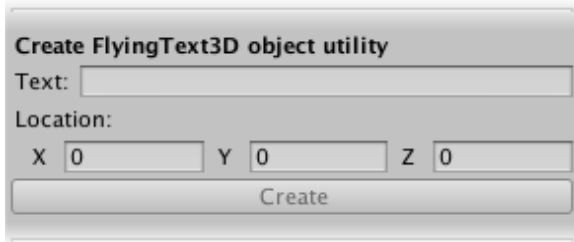
## Text settings

These are the default settings for how text is arranged, such as spacing, justification, and so on. All text will use these settings, though they can be changed in code (see the [FlyingText3D Coding](#) section). The exact effect of each setting is covered in [Text Settings](#).



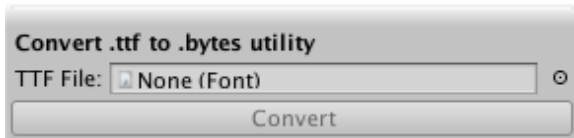
## GameObject settings

These are the settings that apply to the GameObject that’s created from your text. You can change the anchor point, and add colliders and rigidbodies as desired. These settings are covered in detail in the [GameObject Settings](#) section.



## Create FlyingText3D object utility

This is a utility for creating a 3D text object directly in the scene (as opposed to being dynamically generated at runtime; see [FlyingText3D Coding](#) for that). Just type your text in the textfield, set the location in world space if desired (you can always move it later), then click “Create”. It will use the default text settings as mentioned above. You can use tags here if you like (see the [Tags](#) section). Once created, the mesh used for the 3D text will be saved in a folder called 3DTextMeshes in your project. You can make prefabs out of 3D text objects; the easiest way is to drag the 3D text object from the Hierarchy pane to the Project pane.



## Convert .ttf to .bytes utility

This is just a quick utility for renaming .ttf files to .bytes, in case you don’t want to do it outside Unity. Drag a TrueType font file onto the slot (or use the popup selector), then click Convert. The file will be copied to a folder called ConvertedFonts in your project, and can then be used with FlyingText3D.

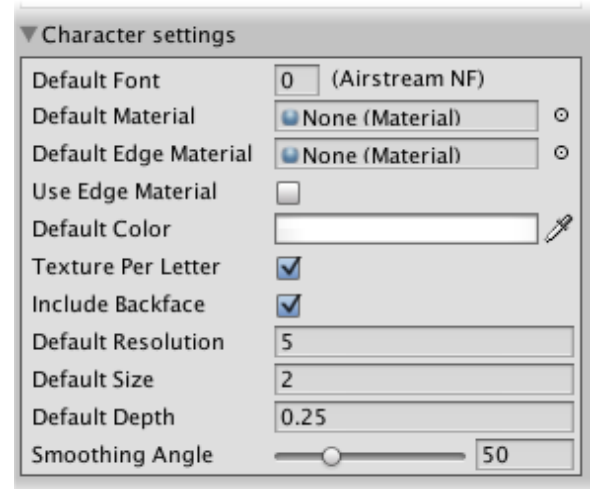
First up is the **Default Font**. This is an index that corresponds to the appropriate font, as set up in the Font Settings list. If you only have one font, then the only number you could use is 0. Unless overridden with tags, all text will use this font.

Next is the **Default Material**. You can use any kind of material that you'd normally use with Unity objects. The only thing to be aware of is that colors in FlyingText3D are implemented with vertex colors, which aren't typically used in the default Unity shaders (outside of the particle shaders, which aren't really appropriate to use for 3D text). You can create custom shaders which include vertex colors, plus there's a shader included with FlyingText3D called VertexLitColor, which uses vertex lighting combined with vertex colors. If you don't select a material and just leave it at "None", then the included VertexColored material is used by default — it must be in the Resources folder for this to work. You can of course create your own materials using the VertexLitColor shader. If you use a material which doesn't make use of vertex colors, you can always change the color of the material itself, if desired. In this case, the only thing you won't be able to do is use more than one color in the same text object.

After that is the **Default Edge Material**. This is used for the extruded edges of characters, if desired (see "Use Edge Material" below). Everything about the default material applies here as well.

If **Use Edge Material** is checked, then all characters will have two materials, the standard material and the edge material. The standard material is for faces (front and back), and the edge material is for the extruded edges. The default depth must be greater than 0 for this to work—if there's no depth, there's nothing to put the edge material on. If "Use Edge Material" is unchecked, then only the default material will be used for the entire character, and the edge material is ignored.

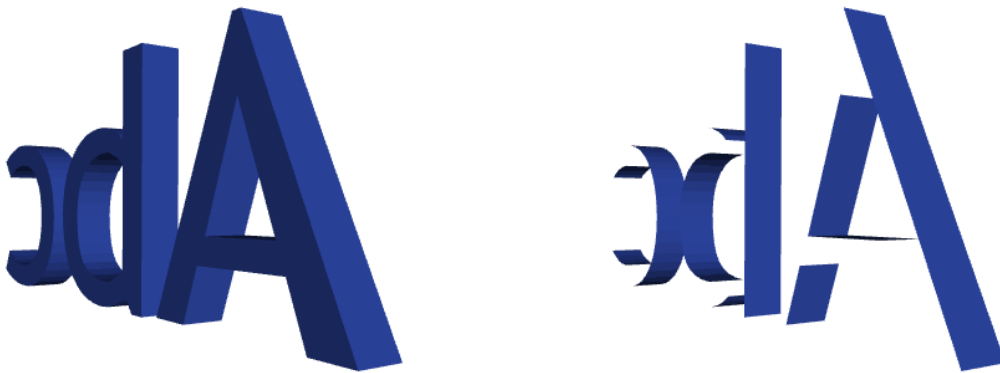
Now we get to **Default Color**. As mentioned above, this requires a shader that uses vertex colors to have any effect. If you're using a material with a texture, it's fairly likely you'd want to leave this as white anyway.



**TexturePerLetter** controls whether any texture you might be using in a material is stretched over the entire block of text (when unchecked) or whether it's constrained to individual letters (when checked). Note that if you use letters as separate GameObjects (see [FlyingText3D Coding](#)), then textures are always computed per-letter regardless of this setting.



You can uncheck **Include Backface** as an optimization for those cases where you're using extruded text, but won't ever see the back. When it's checked, the backface is drawn, but if it's not checked, then it won't be computed, and therefore uses fewer vertices. Note that if the default depth is 0, then text won't have a backface even if this is checked.



The **Default Resolution** affects how smooth text appears. This can be as low as 1, and has no set upper limit, though the higher the resolution, the more vertices are typically generated, and there is a limit to how many vertices can be in a letter (namely, 65,534). What number you should use here depends on several factors, such as how a font is constructed and how close to the camera you intend for it to be, but typically, the higher the number, the more smooth the text. Generally you'd want to try keeping it as low as possible while still looking as smooth as you need. Here's an example font at resolutions 1, 5, 10, and 20:

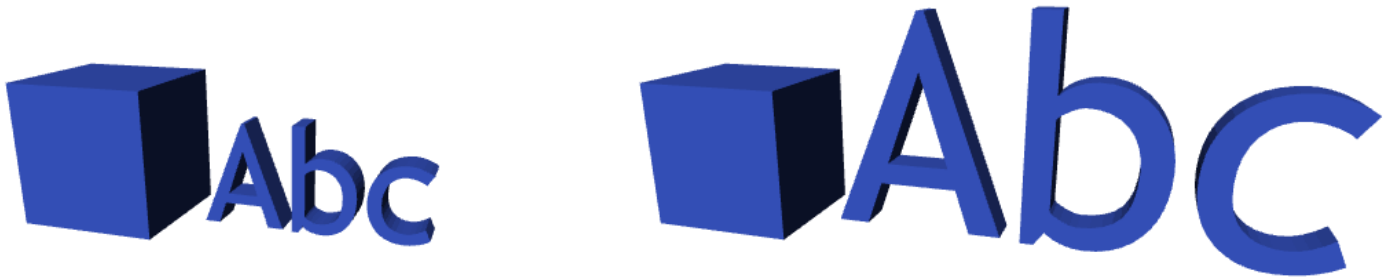


Note that straight lines are never subdivided, no matter how high the resolution. Also, FlyingText3D attempts to limit subdivision where it wouldn't add any noticeable visual quality. For example, tight inside curves are subdivided less often than longer outside curves. Here's a letter O at resolutions 10 and 15...you can see that the interior stays the same while the exterior has more points:



Keep in mind that particularly complex fonts won't appear much different, if at all, at lower resolutions.

**Default Size** is, of course, how big the text is. Fonts use a measurement known as an “em”, and when using a size of 1, that means 1 unit = 1 em. At a size of 2, 1 unit = .5 em, and so on. The size can be as low as .001, with no particular upper limit. Here’s the default Unity cube next to a font using size 1, and the same font using size 2:



**Default Depth** controls how far the text is extruded. If you use a depth of 0, then the text isn’t extruded at all, and uses fewer vertices, since the sides and back face aren’t included. Here’s an illustration of some size 2 text using a depth of 0, .25, and 1.5:



Finally, **Smoothing Angle** controls the look of the edges when text is extruded (it has no effect if the extrude depth is 0). It works basically the same way that the smoothing angle for imported meshes works in Unity. Namely, any two connecting line segments with an angle less than the smoothing angle are hard-edged, and if the angle is greater, they use smoothed lighting. The default of 50 generally works well for most fonts, but can be adjusted as needed — if the angle is too high for some edges, the lighting may look odd, or if it's too low, then some segments will be hard-edged when they should be smooth. If the angle is 0, then all edges are hard, and if the angle is 180, then all are smoothed. Below is an illustration of angles of 50, 0, and 180. You can see that using 180, since everything is smoothed, there's some strange lighting on what should be the flat ends of the character. Note that the smoothing angle has no effect on the resolution — it only affects the lighting of the edges. Also, having smoothed edges also reduces the number of vertices, so it's generally more efficient as well as better-looking.



50°

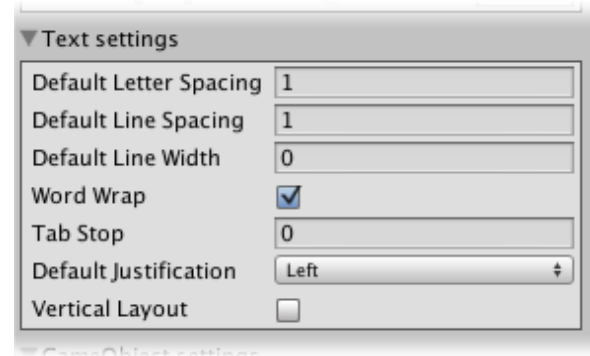


0°



180°

The first of the text settings is **Default Letter Spacing**, which is used to manually control how close together or far apart letters appear. The default of 1 uses the font's built-in spacing, but using less than 1 squishes the letters together (the lower limit is 0, in which case all letters appear on top of each other), and using more than 1 spreads them apart. Here's some text using the default of 1, followed by 1.1, then 1.5:



Abc      Abc      A b c

As you would expect, the **Default Line Spacing** controls how far apart lines of text appear, in those cases where you're using multiple lines in the same text object. (You can do this by using the standard “\n” linebreak character, or a <br> tag.) The line spacing can be negative, in which case lines will go up instead of down. For example, line spacing of 1, 1.5, and -1:

Abc  
def

Abc  
def

def  
Abc

The **Default Line Width** setting is how long, in world units, a line of text can be before it flows to a new line. If the line width is set to 0, that means there's no limit, and text will never wrap (unless you tell it to by using a line break character). Here's a string of text with the line width set to 6, and the same text with the line width set to 8:

Abcde  
fghijkl

Abcdefg  
hijkl

**Word Wrap** is only used if the line width (as detailed above) is not 0. If it's checked, then text will break between words rather than between characters. Although if a word is too long to fit on one line by itself in the specified width, it will necessarily be broken between characters. Only the space character is used for word wrapping, so you can prevent word wrap between specific words by using a non-breaking space character, such as typing alt-space on OS X. Here's a string of text with a line width of 8, with word wrap and without it:

Hello  
there!

Hello th  
ere!

You can use **Tab Stop** to align columns of text. If the number entered here is more than 0, then a tab character (“\t”) will make the next character be positioned at the next available space that's a multiple of the tab stop value. The string “Abcd\tef\nAbc\tdef” with a tab stop of 6, for example, will result in this:

Abcd      ef  
Abc      def

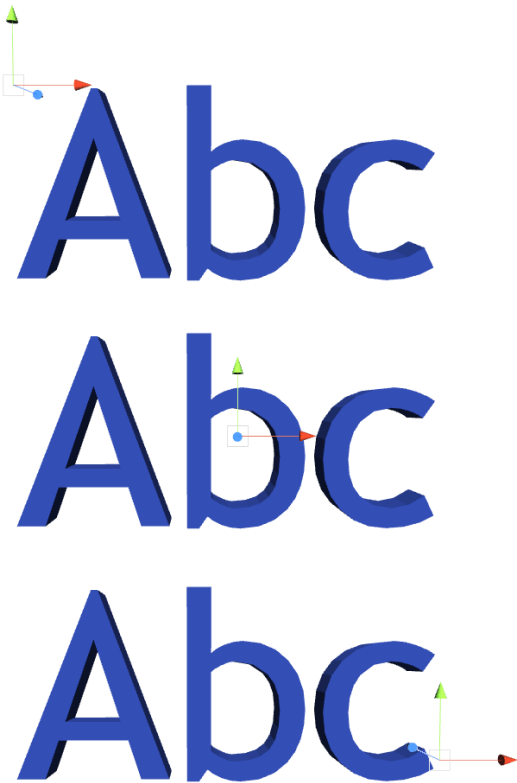
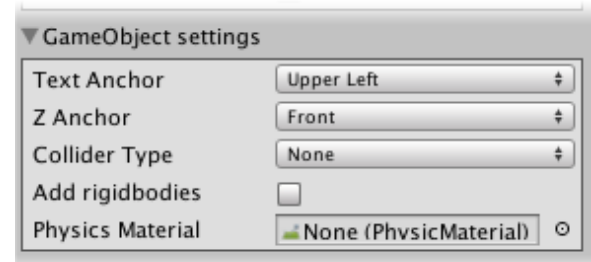
**Default Justification** is whether multiple lines of text are left, center, or right justified (unless overridden with tags). If there's only one line of text, justification only has an effect if the line width is greater than 0.

abc abc abc  
defghi defghi defghi

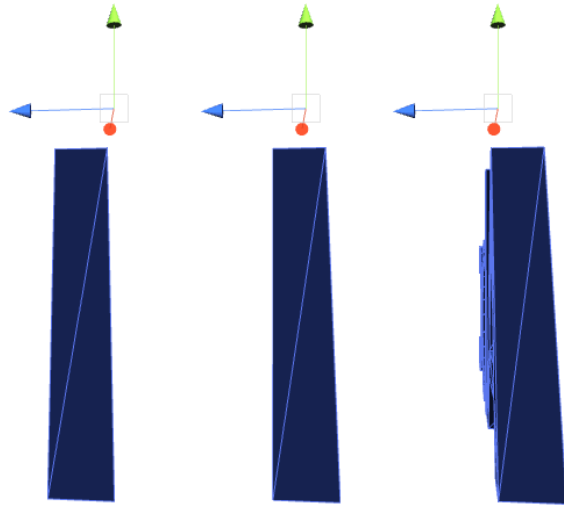
You can use **Vertical Layout** to make text flow vertically instead of horizontally. In this case, line width and word wrap are not used, and a newline character (“\n” or “<br>”) will start a new column.

A  
b  
c

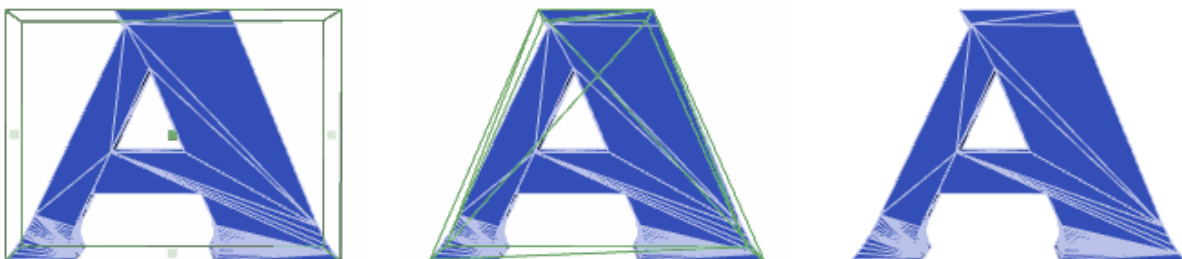
The **Text Anchor** works essentially the same way as the anchor in a Unity GUIText or TextMesh object. It's used to position the pivot point in a given location. (If it doesn't seem to be working correctly, make sure the pivot in Unity is set to Pivot rather than Center.) Below are illustrations of upper left, middle center, and lower right. When using letters as separate GameObjects, the anchor applies to the parent object that groups the separate letters together.



The **Z Anchor** is similar, but it applies to the depth of the 3D text object. It can be front, middle, or back. If the default depth is set to 0, then the Z Anchor option doesn't apply.



The **Collider Type** is the type of collider that's applied to the 3D text object. This can be box, convex mesh, or mesh. All three work with both individual letters and groups of text, though keep in mind that mesh colliders can be expensive to calculate, so long lines of text may experience a delay when they're created.



Remember that non-convex mesh colliders can't collide with other non-convex mesh colliders, so you'd normally only use them for characters that don't move and really need to have accurate collision.

**Add Rigidbodies** will do just that, so that the 3D text can be affected by physics. You can drag a physics material from your project onto the **Physics Material** slot (or use the popup selector), in order to control bounciness, friction, etc.

When writing text for use as 3D text objects, it's possible to insert HTML-like tags to change the appearance of text within the same object. There are currently 8 tags: Size, Color, Font, Zpos, Depth, Justify, Space, and a linebreak tag, namely `<br>`, which you can use instead of `"\n"` if desired. Note that closing tags are not used or recognized; instead, just use another tag to change back if needed. For example, `"<size=1>abc<size=2>def<size=1>ghi"` will create "abc" at size 1, "def" at size 2, then reset to 1 for "ghi". Capitalization and spacing are unimportant, so `<Size = 1>` is the same as `<size=1>`. Quotes around values are also not needed, and are ignored. If you need to use an actual `"<"` or `">"` symbol in the text, use `"<<"` and `">>"`. In other words, `"abc<<def"` will appear as `"abc<def"`. Any number of different tags can be used in a given string.

**Size:** `<size=float>`

The size of the text. This is a float value, and can be as small as .001, with no set upper limit (other than `float.MaxValue`).



`<size=2>Abc<size=1.4>def`

**Color:** `<color=#000000>` or `<color=string>`

The color of the text. This can be specified as a name (currently supported are the built-in Unity color names such as black, white, red, blue, etc.), or as a 6-digit hex code. With a hex code, the first two digits are red, the next two green, then blue. For example, 100% purple would be `#ff00ff`.



`<color=#344fb9>Abc<color=green>def`

**Font:** `<font=int>` or `<font=string>`

The font used, as an index of the fonts supplied in the inspector font settings, or as the name of the font. This is the actual font name, not the file name, which may differ from the real name. Spacing and capitalization aren't important, so `<font=Times New Roman>` is the same as `<font=timesnewroman>` (but easier to read!).



`<font=0>Abc<font=Apple Chancery>def`

**Zpos:** <zpos=float>

Normally all letters are placed at a local z position of 0, but you can control the z position of individual words or letters with this tag. It can be positive or negative to move the letters forward or back.



Abc<zpos=1.0>def

**Depth:** <depth=float>

The depth of the 3D text object, as described in the Text Settings section. This can range from 0 to float.MaxValue, though note that the default depth affects whether this tag has any effect. If the default is 0, then no letters can be extruded even if specified by a tag. If the default is greater than 0, then any depth can be specified, though letters set to 0 by tag in this case technically will still have edges, but they'll be really really thin.



<depth=.25>Abc<depth=1.5>def

**Justify:** <justify=left> <justify=right> <justify=center>

The justification of this line of text and any following lines, or until another justify tag is found. Since it affects entire lines of text, it doesn't matter where in the line this tag is actually located. If there are multiple justify tags in one line, the last one is used.



<justify=left>abc<br>defghi<br>  
<justify=right>jkl

**Space:** <space=float>

This can be used to manually control the precise amount of spacing between two letters. The units are in terms of ems. The number specified can be either positive or negative, and if negative, the letters are squished together. You might use this, for example, to do manual kerning for fonts that don't include any kerning information. Note that for the purposes of word wrapping, spaces done in this way are considered to be non-breaking spaces.

Abc def

Abc<space=.25>def

*Flying*Text3D

```
<font=Airstream><color=5a7ae3><depth=.01><size=2.25>Flying<color=#e0e0e0><depth=.2>  
<font=ArialBlack><size=1.5><space=.1>Text<depth=.75><zpos=-.25>3D
```

In addition to creating 3D text objects from the inspector utility (see [Create FlyingText3D object](#)), you can also create text objects dynamically at runtime. This is done using the **FlyingText.GetObject**, **FlyingText.GetObjects**, and **FlyingText.GetObjectsArray** commands, which return a single GameObject in the first case, or separate GameObjects for each character in the other two cases.

In order for these commands to work, the FlyingText script must be attached to some object in the scene (as described on [page 2](#)). If you use Application.LoadLevel to change scenes, the FlyingText script will not be destroyed, so you can switch to other scenes that haven't had the FlyingText script attached to an object originally. FlyingText runs some initialization in an Awake function, so you'd want to make sure any calls you make to FlyingText functions run after this initialization, or else you'll get an error message informing you of this. The easiest way to do this is to ensure that any FlyingText calls are made in Start or later, though you can also use the script execution order project setting.

Note that if you're using characters in strings outside the ASCII range and you're using C#, your script must be saved with UTF-16 encoding. Unityscript users can use UTF-8 or UTF-16.

## GetObject

```
function GetObject (text : String  
                    position : Vector3 = Vector3.zero,  
                    rotation : Quaternion = Quaternion.identity) : GameObject
```

Only the **text** parameter is required, so you can supply just the text, and the position and rotation will be the default values specified. All the other options, such as material, size, and so on, will get their values from the defaults supplied in the FlyingText3D inspector, or from default values you set in code (see [Setting defaults](#) below). You can also use tags in the text, as described in the [Tags](#) section. The GetObject function returns a single GameObject, so you can assign that to a variable and work with it as you would any other GameObject:

```
var textObject = FlyingText.GetObject ("Hello");  
textObject.tag = "TextObject";
```

You can set the **position** and **rotation** in the same way that you do when using Unity's Instantiate command:

```
var textObject = FlyingText.GetObject ("Hello",  
                                       Vector3(0, 5, 10), Quaternion.Euler(0, 90, 0));
```

You can also change the position and rotation later in the same way that you would for any other GameObject, by referring to the Transform component. You can also supply additional parameters:

[illegible]

## GetObjects

This has the exact same parameters as `GetObject`, except that the `GameObject` that's returned is actually an empty `GameObject` that serves as a parent to a number of separate `GameObjects`, one for each character in the text. You can use the parent to move the children objects around as a group, or detach them, or whatever else comes to mind. For example, assuming that “Add Rigidbodies” is checked in the `FlyingText` inspector:

```
var objectParent = FlyingText.GetObjects ("Hello", null, null, 2.0, .25, 5);
var rigidbodies = objectParent.GetComponentInChildren.<Rigidbody>();
for (var rb in rigidbodies) {
    rb.useGravity = false;
    rb.rigidbody.AddExplosionForce (50, Vector3(4, 0, 0), 10, 3);
}
```

## GetObjectsArray

This also results in separate `GameObjects`, as with `GetObjects`, but the difference is that instead of a single `GameObject` with children, an array of `GameObjects` is returned. In other words, this function returns `GameObject[]` rather than `GameObject`. Otherwise it works the same as `GetObject` and `GetObjects`. The order of objects in the array corresponds to the order of the characters in the string that you passed in. So if you need to refer to individual 3D letters in a defined order, this is the function to use. In this example, the letter “e” is moved .5 units below the other letters:

```
var textObjects = FlyingText.GetObjectsArray ("Hello");
textObjects[1].transform.Translate (Vector3.up * -.5);
```

## UpdateObject

```
function UpdateObject (gameObject : GameObject,  
                        text : String) : void
```

At times you may want to update an existing 3D text object with some new text. Just pass in an appropriate `GameObject` to the `UpdateObject` with the replacement text, and it will change.

Note that any parameters that you might have specified when creating the text object, such as material, size, resolution, and so on, are stored with that object, in the form of a `TextObjectData` component. As such, you only need to specify the new text, and the existing parameters will be re-used. This also means that any object you pass in to `UpdateObject` must not have the `TextObjectData` component removed, or the function won't work. An example of text changing over time:

```
function Start () {  
    var textObject = FlyingText.GetObject ("One");  
    yield WaitForSeconds (1);  
    FlyingText.UpdateObject (textObject, "Two");  
    yield WaitForSeconds (1);  
    FlyingText.UpdateObject (textObject, "Three");  
}
```

## PrimeText

```
function PrimeText (text : String,  
                    size : float,  
                    extrudeDepth : float,  
                    resolution : int) : void
```

When FlyingText3D builds characters, it first needs to parse the TrueType file and render the outlines as meshes. This process takes several times longer the first time a character is used (depending on factors such as the complexity of the font and the resolution) compared to subsequent uses of the same character, where the already-generated data merely has to be copied over to the mesh. Therefore, depending on hardware speed, how many unique characters you're using, and so on, there may be a brief hiccup in framerate when a text object is first created. On faster hardware, this may not actually be noticeable anyway, particularly with lower text resolutions and fewer characters. But on slower hardware, you may in some cases want to “prime” the text first during initialization with characters you're planning on using, so that they're built faster when they're used later “for real”.

Only the text parameter is strictly necessary, in which case the defaults will be used for the size, extrude depth, and resolution. For example:

```
function Start () {  
    FlyingText.PrimeText ("abcdefghijklmnopqrstuvwxyz");  
}
```

Note that changing the size, extrude depth, and resolution during runtime (through the use of tags or by changing the defaults) will also cause characters to be rebuilt when they're next used, though changing the size or extrude depth is quite a bit faster than changing the resolution.

## Setting defaults

Aside from setting defaults in the FlyingText inspector, you can also set them at runtime.

```
FlyingText.defaultFont = int
FlyingText.defaultMaterial = Material
FlyingText.defaultEdgeMaterial = Material
FlyingText.useEdgeMaterial = boolean
FlyingText.defaultColor = Color
FlyingText.texturePerLetter = boolean
FlyingText.includeBackface = boolean
FlyingText.defaultResolution = int
FlyingText.defaultSize = float
FlyingText.defaultDepth = float
FlyingText.smoothingAngle = float
FlyingText.defaultLetterSpacing = float
FlyingText.defaultLineSpacing = float
FlyingText.defaultLineWidth = float
FlyingText.wordWrap = boolean
FlyingText.tabStop = float
FlyingText.defaultJustification = Justify
FlyingText.verticalLayout = boolean
FlyingText.anchor = TextAnchor
FlyingText.zAnchor = ZAnchor
FlyingText.colliderType = ColliderType
FlyingText.addRigidbody = boolean
FlyingText.physicsMaterial = PhysicMaterial
```

ZAnchor is an enum with values of Front, Middle, and Back. For example:

```
FlyingText.zAnchor = ZAnchor.Middle;
```

ColliderType is an enum with values of None, Box, ConvexMesh, and Mesh. For example:

```
FlyingText.colliderType = ColliderType.Box;
```

Justify is an enum with values of Left, Center, and Right. Note that these three enums are in the FlyingText3D namespace, so you should import that namespace if using ZAnchor, ColliderType, or Justify. That's "import FlyingText3D;" in Unityscript, and "using FlyingText3D;" in C#. For example:

```
import FlyingText3D;

function Start () {
    FlyingText.defaultJustification = Justify.Center;
}
```

TextAnchor is a Unity enum and can be found in the Unity documentation.

FlyingText3D does a lot, but it doesn't do everything (yet!). Here's a list of stuff that may have some limitations.

- First and foremost, it's possible to find characters that won't render. One problem is characters made with contours that cross their own paths. This sort of thing is discouraged in the TrueType specifications, but not forbidden, so occasionally you may come across characters that do this. It's less of a problem when using TrueType fonts to render bitmaps, but is fairly hard to deal with as vector outlines. Fortunately it's not very common, and seems mostly limited to lesser-quality, complicated decorative fonts when it does happen. It's actually possible that FlyingText3D may be able to render at least some of these characters anyway just by luck — if you come across this, try different resolutions; a character that doesn't work at resolution 15 might work at resolution 20, say, or vice versa. This limitation may be addressed in a future update.
- Compound glyphs currently only support simple combining. Compound glyphs are most often used with accented characters such as á and ï, where the accent marks and letters may be stored separately, then mixed and matched when output, in order to save space. These characters should work, but it's possible for compound glyphs to be combined in ways that involve things like scaling and rotation. These things aren't implemented yet because it seems very hard to find any fonts which actually do this, but it's possible you may find characters composed of separate parts that don't look right. In the future, improved compound glyph support will probably happen if an appropriate font can be found.
- Only one type of kerning table is implemented at this time. It seems to be by far the most common type — if fonts have kerning at all they always seem to use this table — but it's possible you may run across fonts that should be kerned but aren't. Additional kerning tables may be implemented in the future if warranted.
- Not all character mapping tables are implemented. The most common types are, but it's possible you might encounter a font that won't render at all. As above, additional character mapping tables may be implemented in the future if warranted.
- Non-Western fonts have limited support right now. They will probably render, but right-to-left support isn't implemented yet. You can use the option for vertical layout (useful for Chinese fonts), however this isn't "true" vertical layout support as defined by the TTF file, so it's possible it may not produce ideal results.