DTT
APPS WEB GAMES

SINCE 2010

Word Finder

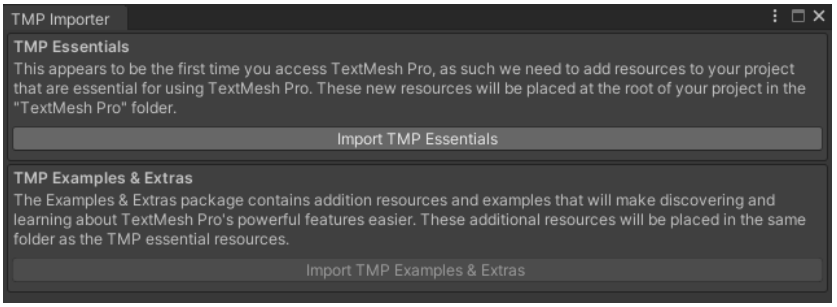
Documentation | V2.0.0 | 25-01-22

Table of Contents

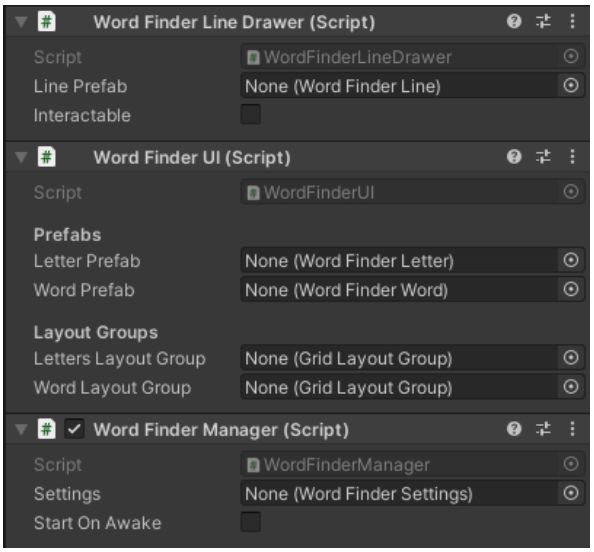
1. Get started quickly

This is a summary to help you get started quickly. More details are provided below.

To get started, you first need to import the **TMP Essentials**. This is a free asset included with Unity. This screen will popup after importing the package. The Word Finder package uses the **TextMeshProGUI** to draw the letters and text on the screen.



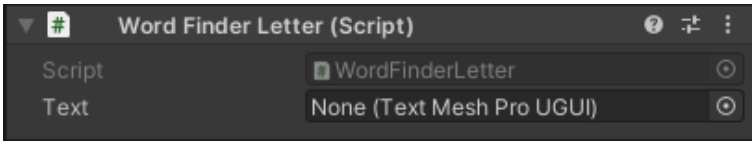
After importing the TMP essentials, you can get started on setting up the game.



- First, place a WordFinderManager component on a gameobject. This will automatically create WordFinderLineDrawer and WordFinderUI components on the same gameobject.
- Secondly, you need to create a set of prefabs. For the quick start, we recommend using the example prefabs found in the demo folder. These are called: DefaultWord, DefaultLetter, and DefaultLine. These prefabs and the **DemoScene** that comes with it can be used as an example on how to implement your own behaviours.

  Alternatively you can create your own prefabs, which we'll cover in the following points.

- To create a letter prefab, make a new prefab and add the WordFinderLetter component to it. This component can be found in the 'Runtime/UI' folder. Then assign a **TextMeshProUGUI** object to the text field. The letter text will be set on this component. You can then assign the prefab to the WordFinderUI component.



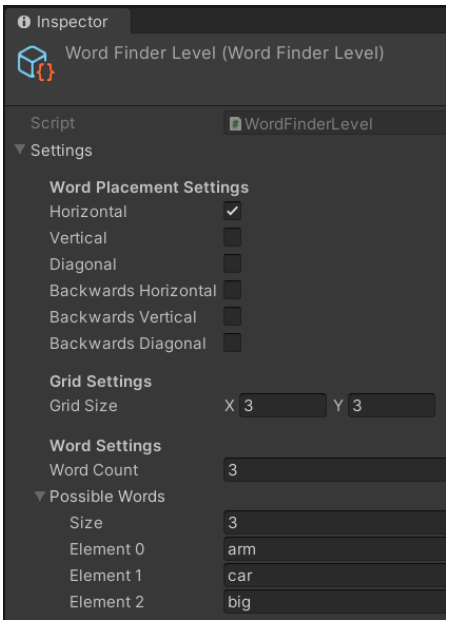
- You then need to make a prefab for the lines. You can add the DefaultLine component to your line prefab, which can be found in the 'Runtime/UI' folder. This component adds a basic snap and fail animation to the line.

  To learn about creating more about your own line implementation, go to the API section of this document about the WordFinderLine component.

  You can customize the style of your line using the **LineRenderer** component that is automatically added to the object. You can then assign the prefab to the Line Prefab field on the WordFinderLineDrawer component.

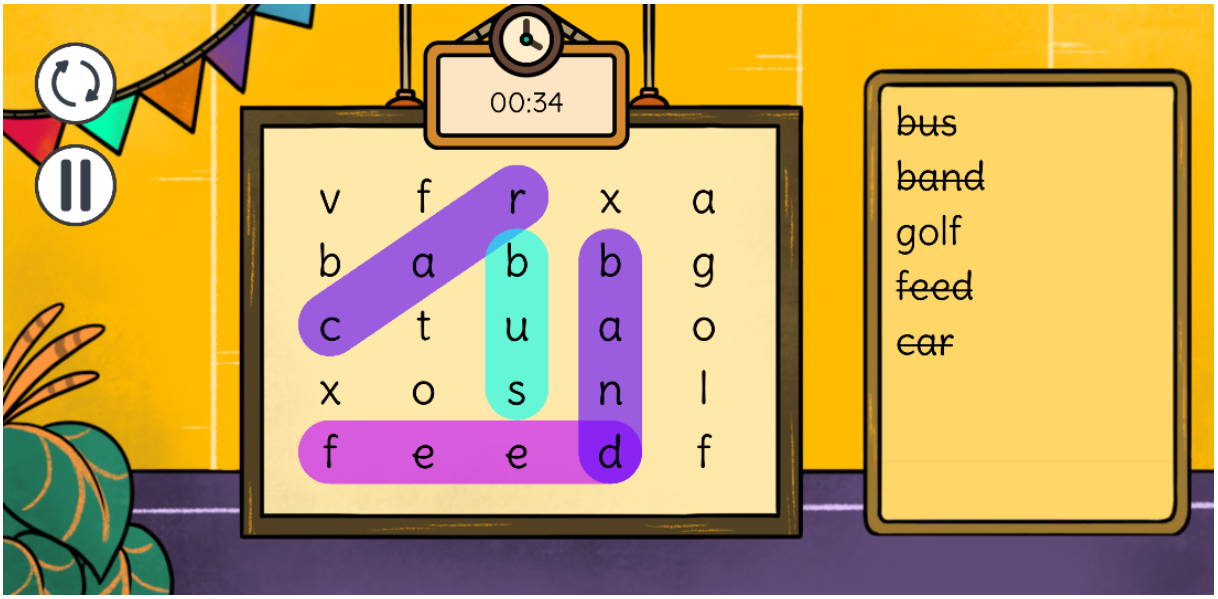
- Lastly, you need to make a prefab for the words. To create a WordFinderWord, you have to place the WordFinderWord component on a prefab, which can be found in the 'Runtime/UI' folder. Assign a **TextMeshProUGUI** component to the Text field, and assign the prefab to the Word Prefab field on the WordFinderUI component.
- The UI of the game requires a canvas with the **Render mode** set to **Screen Space - Camera**. You also need 2 **GridLayoutGroups**. One for the letters and one for the words. These layout groups can then be dragged into the Letters Layout Group and Word Layout Group fields on the WordFinderUI component. The cell size is dynamically set. The rest of the settings on the layout group can be customized. You can create these yourself, or use the examples provided in the demo scene.

- After these steps, you need to make a **WordFinderLevel** object. To create one, go to the project tab in Unity, right click, and go to Create/DTT/MiniGame/WordFinder/Level. This will create a scriptable object on which you can define the rules of your game. You can assign this object to the settings field on the **WordFinderManager** component.
- To specify which words you want in your word finder game, you can simply add words to the **Possible Words** field on the **WordFinderLevel** object.
- Now you can start the game by either calling the **StartGame** method on the **WordFinderManager**, or by setting the start on awake field on the **WordFinderManager** to true.



2. Introduction

DTT WordFinder is a Unity asset that allows you to easily implement a word finder game into your project. The asset allows you to customize each of the components present in a word finder game. Using a Scriptable Object you can change the difficulty of the game, allowing you to have different difficulty settings per level. Furthermore, you can also define your own list of words that you want to use in your word finder. All this allows for an easy and fast way to implement a simple word finder game into your project.

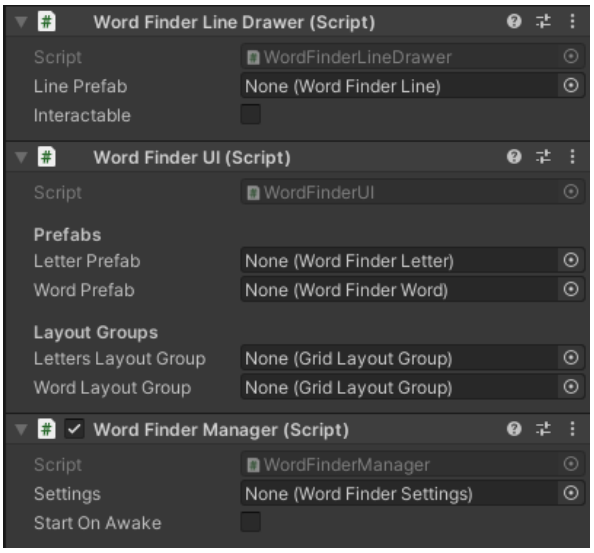


3. Changelog

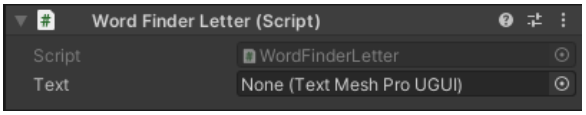
1. version 1.0.0 - Initial release
2. version 2.0.0 - Minigame base implementation
    a. Added
        i. Implemented and added dependency to the base minigame package.
    b. Updated
        i. Dependency to runtime utilities.

4. Set-Up

1. To start off, place the **WordFinderManager** component on an object in your scene. This will also add a **WordFinderLineDrawer** and **WordFinderUI**. The **WordFinderManager** can be called to start and pause the game, and allows you to listen to events for when the game starts or ends. The **WordFinderLineDrawer** handles the creation of the lines on the letter grid. The **WordFinderUI** generates the UI elements of the game, like the letters and a list of the words you need to search for.



2. You then need to create prefabs for each of the components in the game. The first one will be the **letters** displayed on the letter grid. This component only requires you to add the **WordFinderLetter** to a gameobject. You can reference the **WordFinderLetter** class to listen to certain events. *More about the possible events can be found in the API section of this document.*



After creating the prefab and adding the **WordFinderLetter** component, you need to add a **TextMeshProUGUI** component somewhere in your prefab. You can then drag this component to the **Text** field in the **WordFinderLetter**. The letter will be set on this text object. After these steps, you can drag the prefab into the **Letter Prefab** field of the **WordFinderUI**. You'll also need to assign a **GridLayoutGroup** to the **WordFinderUI**. The generated letters will be placed into this layout group.

3. Next up is the prefab for the **words** that indicate which words to look for in the grid. Like the letter, you first need to make a prefab with the **WordFinderWord** component on it. The word also has certain events the user can listen to. *More about these events can be found in the API section of this document.*

   NOTE: The game uses object pooling to prevent an excessive amount of garbage collection, since a lot of gameobjects are used per game. The **WordFinderLetter** and **WordFinderWord** allow you to listen to an event called **Reset** that is called once the object is reused. Use this event to reset the components back to their original state.

   Like the **WordFinderLetter**, the word also requires a **TextMeshProUGUI**. After adding the text to the **Text** field, you can drag the prefab into the **Word Prefab** field of the **WordFinderUI**. Just like the letters, you'll need to assign another **GridLayoutGroup** to the **WordFinderUI**. The generated words will be put into this layout group. The cell size of the grid will dynamically change based on the amount of words and the space they take in.

4. The last required component is the **WordFinderLine**. You have the option to make your own implementation of the abstract **WordFinderLine** class. This class will have you implement an **OnSuccess** method, which is called when the line successfully marks a word. The **OnFailure** method is called when the line unsuccessfully marks a line, and the **ResetObject** method, like the **WordFinderWord** class, is called when the line is reused. You can also override the **SnapToLetter** method to implement your own animation for when the line snaps to a new point on the grid.

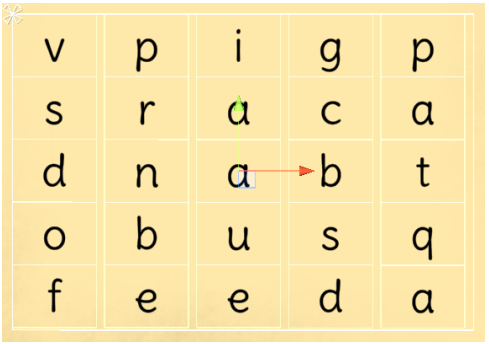
```
public class MyLine : WordFinderLine
{
    public override IEnumerator OnFailure()
    {
        // Add OnFailure behaviour here...
        yield break;
    }

    public override void OnSuccess()
    {
        // Add OnSuccess behaviour here...
    }

    public override void ResetObject()
    {
        // Reset the line to its original state!
    }

    public override void SnapToLetter(Vector2 position)
    {
        // Implement your own animation for when the line snaps
        // to a letter. Base instantly sets it to the given position.
        base.SnapToLetter(position);
    }
}
```

Alternatively, you can use the **DefaultLine** component, which is a simple implementation of the **WordFinderLine** that gives the line a snap animation and fade out animation when the line fails to find a word.

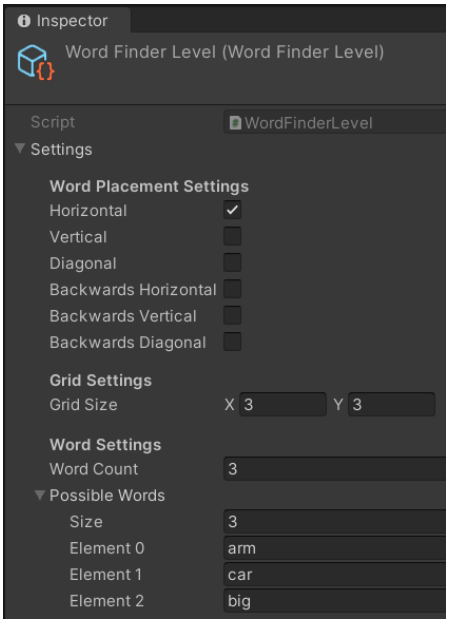
Once you've created your implementation of the **WordFinderLine**, add your behaviour to a prefab. This will automatically add a **LineRenderer**, where you can further customize the look of the drawn line. You can then assign the line prefab to the **Line Prefab** field of the **WordFinderLineDrawer**.

NOTE: The **Order in Layer** field on the **LineRenderer** component should be higher than the order layer of the canvas which the UI will be drawn on, to prevent the lines from being drawn behind the UI.

5. The UI of the game also has a few requirements. The **Render mode** of the canvas must be set to **Screen Space - Camera**. This makes it possible for the lines to be drawn over the Letters on the grid. The UI also requires 2 **Grid Layout Groups**, One for the letters and one for the words. These can be dragged into the Letters Layout Group and Words Layout Group in the WordFinderUI component. The layout group cell sizes are set dynamically, and any other settings aren't needed.



6. After creating all 3 prefabs and assigning them to the correct fields, you can start defining the difficulty of the game using the WordFinderLevel scriptable object. To create this object, right click in the project section of your Unity project and go to Create/DTT/MiniGame/WordFinder/Level. This will create a WordFinderLevel scriptable object. On this object, you can set whether the words will be placed horizontally, vertically, and/or diagonally. You can also set the grid size and max amount of words.



   Furthermore, you can define the possible words by adding words to the PossibleWords list. If a word doesn't fit in the grid, it will give a warning and not use it in the generation process.

   The WordFinderLevel can be dragged into the Level field of the WordFinderManager.

7. After these steps you can finally start the game by either calling the StartGame method from the WordFinderManager, or by setting the Start on Awake field on the manager to true.

5. API

WordFinderManager

The WordFinderManager component handles the state of the game and allows you to start, pause and clear the game.

- Continue method unpauses the game. This will continue the timer and enable the WordFinderLineDrawer again.
- ClearLevel method clears all wordfinder components from the game, and stops the game.
- Pause method pauses the game. This will stop increasing the time of the PlayTime variable and disables the ability to select words using the WordFinderLineDrawer.
- StartGame method can be used to start the game. The method will initiate the generation of the word finder using the given WordFinderLevel. This method also calls ClearLevel, which clears the objects of the last wordfinder game.

There are also some variables you can access that tell you the status of the game.

- EventHandler holds all the game events the player can listen to. These include events for when the game starts and ends. More is explained about this class in the next API section.
- IsGameActive is true when the game is currently active. It stays true even when paused.
- IsGenerating is true when the wordfinder grid is still being generated. While this is true you can't call StartGame, to prevent the game from generating more than once at the same time.
- IsPaused is true when the game is currently paused.
- LineDrawer is a reference to the current WordFinderLineDrawer used by the manager.
- PlayTime is the amount of time the game has been playing. It resets when the game restarts and stops counting when IsPaused is true.

WordFinderEventHandler

The WordFinderEventHandler handles the main game events. Here is a list of all events:

- OnClear is called when the game components are removed from the canvas. This is called when ClearLevel or StartGame is called from the WordFinderGameManager.
- OnDeselectLetter is called once a letter is deselected, and a line has been drawn.
- OnFinish is called when the user finds all words and the game is complete.
- OnSelectedLetter is called once a letter has been dragged on with a line.
- OnStart is called after the word finder has successfully been generated and started.
- OnWordSelected is called once a word has been selected by the line. The event returns whether it was a valid word and the selected string of letters.

WordFinderLineDrawer

The WordFinderLineDrawer allows you to turn off the intractability using the interactable variable.

WordFinderUI

The WordFinderUI allows you to read the letterGrid, containing all active letters.

WordFinderLetter

The **WordFinderLetter** can be put on a prefab and needs a **TextMeshProUGUI** to display the given letter. An example of the implementation can be found in the Demo/Prefab folder.

This component has some events you can listen to:

- **Confirmed** is invoked when the letter is part of a selected string of letters.
- **Failure** is invoked when the letter is part of a selected string of letters that wasn't a valid word.
- **LastSelected** is invoked when the letter is the last selected letter of a newly selected string.
- **Reset** is invoked when the letter gets reused for the next game.
- **Success** is invoked when the letter is part of a selected string of letters that was a valid word.

You can access some values in this component:

- **Completed** is a boolean that is true when the letter was a part of a valid selected word.
- **Data** is the letter that this object displays.
- The **Neighbours** value contains a dictionary with each of the neighbouring letters.
- **GridPosition** is a vector2 that represents the position of the letter in the grid.
- **Text** is the **TextMeshProUGUI** that is used to display the text.

WordFinderWord

The WordFinderWord can be put on a prefab and needs a **TextMeshProUGUI** to display the given word. An example of the implementation can be found in the Demo/Prefab folder.

This component has some events you can listen to:

- Completed is invoked when the word has been found.
- Reset is invoked when the letter gets reused for the next game.

You can access the following values:

- Text is the **TextMeshProUGUI** that is used to display the text.
- Word is the string this component displays.

WordFinderLine

The WordFinderLine is abstract and must be inherited from to implement your own behaviour. The class allows you to implement the OnSuccess, OnFailure, and just like the other objects, the ResetObject method to reset the object for reuse.

```
public class MyLine: WordFinderLine
{
    public override IEnumerator OnFailure()
    {
        // Add OnFailure behaviour here...
        yield break;
    }

    public override void OnSuccess()
    {
        // Add OnSuccess behaviour here...
    }

    public override voif ResetObject()
    {
        // Reset the line to its original state!
    }

    public override void SnapToLetter(Vector2 position)
    {
        // Implement your own animation for when the line snaps
        // to a letter. Base instantly sets it to the given position.
        base.SnapToLetter(position);
    }
}
```

OnSuccess is called once the line successfully marks a valid word. **OnFailure** is an IEnumerator. Once the IEnumerator is done, the line will be set inactive and returned to the pool, so it can be reused. This method is called when the line marks an invalid word.

The **WordFinderLine** also allows you to override the **SnapToLetter** method, which is called whenever a new word is selected. The position parameter is the position the line has to move to, to connect with the last letter of the selected word. The base of this method uses the **SetEndPosition** method, which sets the end position of the line and sets the z of the end position equal to the start position, to prevent visual bugs.

NOTE: The line renderer will only use 2 positions, the start and end point of the line. Currently, multiple positions aren't supported.

You can access the following values:

- **EndPosition** is the position of the last letter the line is connected to.
- **LineRenderer** is a reference to the LineRenderer component of this line.
- **StartPosition** is the position of the first letter the line is connected to.

A basic implementation of the line can also be used, called **DefaultLine**. You can use the **Snap Speed** field to adjust the speed at which the line snaps to a new letter. The **Fade Time** field can be used to adjust the time it takes until the line has fully faded away after an incorrect answer.

WordFinderLevel

The **WordFinderLevel** is a scriptable object that can be created from Create/DTT/MiniGame/WordFinder/Level. This object handles the settings of the game and can be placed into the **Level** field on the **WordFinderManager**.

- **Horizontal**, **Vertical**, **Diagonal** and each of the **Backwards** versions of these orientations, define the orientation the words can be placed in on the grid.
- **Grid Size** defines the width and height of the grid.
- **Word Count** is the max amount of words that will be used in the game. If words no longer fit in the grid, the amount of words can be lower than the word count.
- **Possible Words** is a list of all words that can be used for the word finder game. If the size of a word is bigger than the grid size, you will receive a warning and the word won't be used in the grid.

You can also call some methods using the **WordFinderLevel**.

- **GetPossibleWords** gets all the words from the **Possible Words** field that can fit in the grid.
- **SetSettings** allows you to change the level configuration by passing in a new **WordFinderConfig** struct into the parameter.

6. Known Limitations

- **Canvas Render Mode:** Lines can only be drawn in front of the canvas when the canvas render mode is set to Screen Space – Camera.

7. Support and feedback

If you have any questions regarding the use of this asset, we are happy to help you out. Always feel free to contact us at:

unity-support@d-tt.nl

(We typically respond within 1-2 business days)

We are actively developing this asset, with many future updates and extensions already planned. We are eager to include feedback from our users in future updates, be they 'quality of life' improvements, new features, bug fixes or anything else that can help you improve your experience with this asset. You can reach us at the email above.

Reviews and ratings are very much appreciated as they help us raise awareness and to improve our assets.

DTT stands for Doing Things Together

DTT is an app, web and game development agency based in the centre of Amsterdam. Established in 2010, DTT has over a decade of experience in mobile, game, and web based technology.

Our game department primarily works in Unity where we put significant emphasis on the development of internal packages, allowing us to efficiently reuse code between projects. To support the Unity community, we are publishing a selection of our internal packages on the Asset Store, including this one.

More information about DTT (including our clients, projects and vacancies) can be found here:

https://www.d-tt.nl/en/