How to Make a Match 3 Game in Unity

Learn how to make a Match 3 game in this Unity tutorial!

By Jeff Fisher Apr 12 2017 · Article (30 mins) · Beginner

In a Match 3 game, the goal is simple: swap pieces around till there's 3 or more in a row. When a match is made, those tiles are removed and the empty spaces are filled. This lets players rack up potential combos and tons of points!

In this tutorial you'll learn how to do the following:

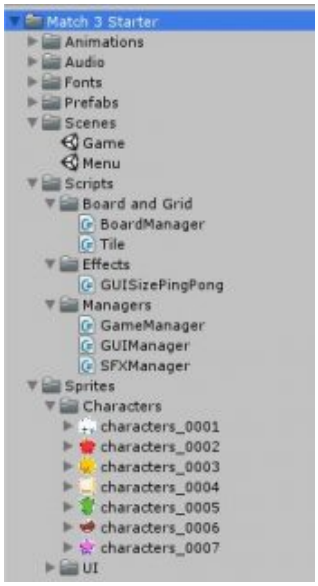
- Create a board filled with game tiles
- Select and deselect tiles with mouse clicks
- Identify adjacent tiles with raycasts
- Swap tiles
- Detect a match of 3 or more using raycasts
- Fill empty tiles after a match has been made
- Keep score and count moves

> ***Note:*** This tutorial assumes you know your way around the Unity editor, that you know how to edit code in a code editor and that you have a basic knowledge of C#. Check out some of our other Unity tutorials(https://www.raywenderlich.com/category/unity) first if you need to sharpen your skills on Unity.

Getting Started

Download the Match 3 how-to starter project (https://koenig-media.raywenderlich.com/uploads/2017/01/Match-3-Game-Starter.zip) and extract it to a location of your choosing.

Open up the Starter Project in Unity. The assets are sorted inside several folders:



Animations: Holds the game over panel animation for when the game ends. If you need to brush up on animation, check out our Introduction to Unity Animation (https://www.raywenderlich.com/116652/introduction-unity-animation-system) tutorial.

Audio: Contains the music and sound effects used in the game.

Fonts: Holds the fonts used in the game.

Prefabs: Contains various managers, UI, and tile prefabs.

Scenes: Holds the menu and game scene.

Scripts: Contains the scripts used in the game. ***BoardManager.cs*** and ***Tile.cs*** are the ones you'll be editing.

Sprites: Contains the UI assets and various character sprites that will be used as tile pieces on the board.

Creating the Board

If it's not opened yet, open up the ***Game*** scene and click play. It's simply a plain blue background with a score and move counter. Time to fix that!

First, create an empty game object and name it ***BoardManager***.

The ***BoardManager*** will be responsible for generating the board and keeping it filled with tiles.

Next, locate ***BoardManager.cs*** under ***Scripts\Board and Grid*** in the Project window. Drag and drop it onto the ***BoardManager*** empty game object in the Hierarchy window.

You should now have this:

It's time to dive into some code. Open up ***BoardManager.cs*** and take a look at what's already in there:

```
public static BoardManager instance;     // 1
    public List<Sprite> characters = new List<Sprite>();     // 2
    public GameObject tile;      // 3
    public int xSize, ySize;     // 4

    private GameObject[,] tiles;      // 5

    public bool IsShifting { get; set; }     // 6

    void Start () {
        instance = GetComponent<BoardManager>();     // 7

        Vector2 offset = tile.GetComponent<SpriteRenderer>().bounds.size;
        CreateBoard(offset.x, offset.y);     // 8
    }

    private void CreateBoard (float xOffset, float yOffset) {
        tiles = new GameObject[xSize, ySize];     // 9

        float startX = transform.position.x;     // 10
        float startY = transform.position.y;

        for (int x = 0; x < xSize; x++) {      // 11
            for (int y = 0; y < ySize; y++) {
                GameObject newTile = Instantiate(tile, new Vector3(startX + (xOffset * x),
                tiles[x, y] = newTile;
            }
        }
    }
```

. Other scripts will need access to `BoardManager.cs`, so the script uses a ***Singleton*** pattern with a static variable named `instance`, this allows it to be called from any script.

. `characters` is a list of sprites that you'll use as your tile pieces.

. The game object prefab `tile` will be the prefab instantiated when you create the board.

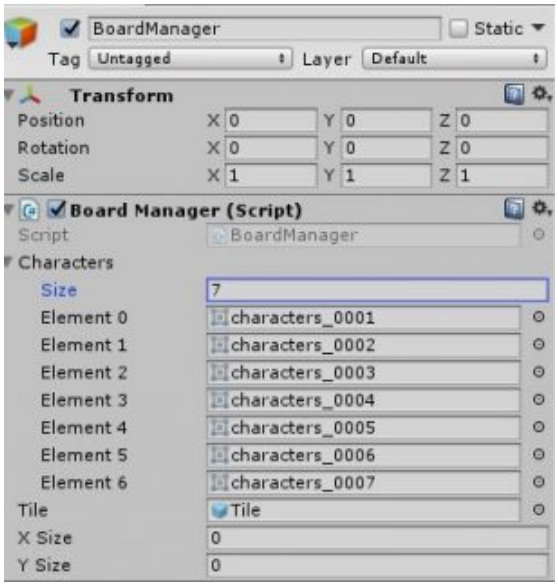
- `xSize` and `ySize` are the X and Y dimensions of the board.
- There's also a 2D array named `tiles` which will be used to store the tiles in the board.
- An encapsulated bool `IsShifting` is also provided; this will tell the game when a match is found and the board is re-filling.
- The `Start()` method sets the singleton with reference of the `BoardManager`.
- Call `CreateBoard()`, passing in the bounds of the `tile` sprite size.
- In `CreateBoard()`, the 2D array `tiles` gets initialized.
- Find the starting positions for the board generation.
- Loop through `xSize` and `ySize`, instantiating a `newTile` every iteration to achieve a grid of rows and columns.

Next, locate your character sprites under ***Sprites\Characters*** in the Project window. Select the ***BoardManager*** in the Hierarchy window.

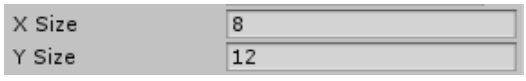
In the Inspector window, change the Character Size value for the ***BoardManager*** script component to *7*. This will add 7 elements to the `Characters` array and display the slots for them in the Inspector window.

Now drag each character into the empty slots. Finally, locate the ***Tile*** prefab under the ***Prefabs*** folder and drag it into the Tile slot.

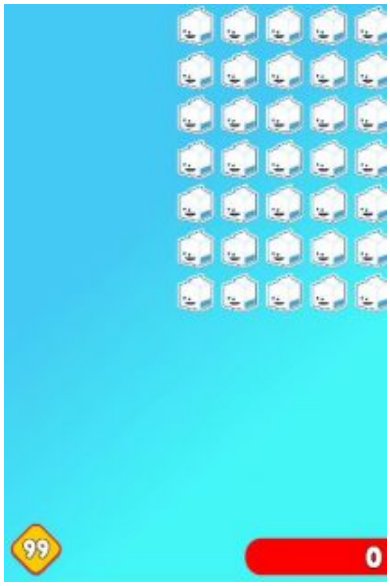
When finished, your scene should look like this:



Now select ***BoardManager*** again. In the ***BoardManager*** component in the Inspector window, set ***X Size*** to ***8*** and ***Y Size*** to ***12***. This is the board size you'll be working with in this tutorial.

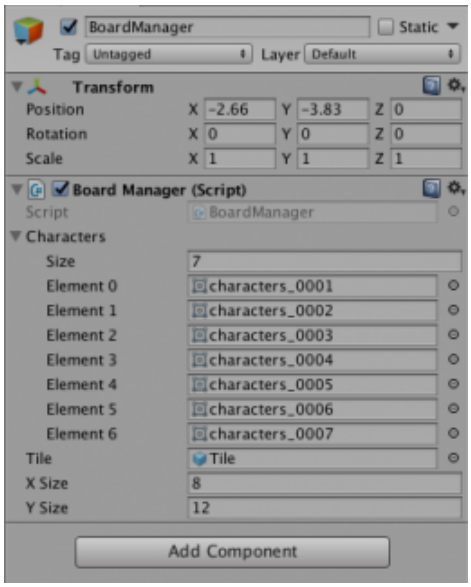


Click play. A board is generated, but it's strangely going offscreen:

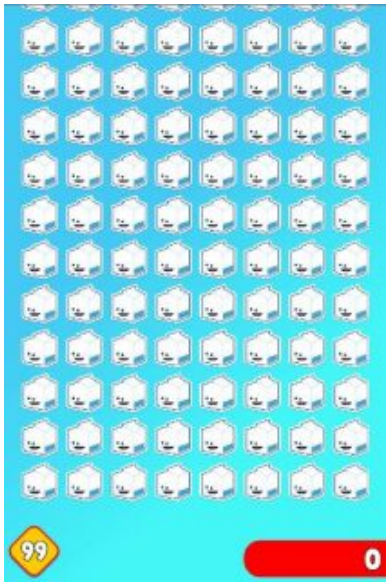


This is because your board generates the tiles up and to the right, with the first tile starting at the BoardManager's position.

To fix this, adjust the ***BoardManager***'s position so it's at the bottom left of your camera's field of view. Set the BoardManager's ***X*** position to ***-2.66*** and the ***Y*** position to ***-3.83***.



Hit play. That looks better, but it won't be much of a game if all the tiles are the same. Luckily, there's an easy way to randomize the board.



Randomizing the Board

Open the BoardManager script and add add these lines of code to the `CreateBoard` method, right underneath `tiles[x, y] = newTile;`:

```
newTile.transform.parent = transform; // 1
Sprite newSprite = characters[Random.Range(0, characters.Count)]; // 2
newTile.GetComponent<SpriteRenderer>().sprite = newSprite; // 3
```

These lines do three key things:

. Parent all the tiles to your BoardManager to keep your Hierarchy clean in the editor.

. Randomly choose a sprite from the ones you previously dragged in earlier.

. Set the newly created tile's sprite to the randomly chosen sprite.

Run the game, and you should see a randomized board:

As you may have noticed, your board can generate a matching 3 combo at the start, and that kind of takes the fun out of it!

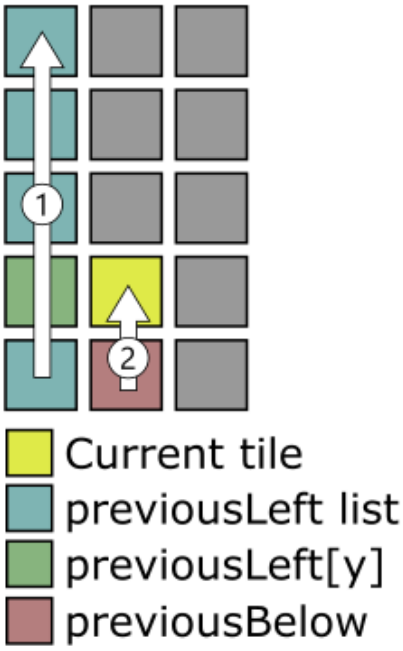
Prevent Repeating Tiles

The board generates up and then right, so to correct the “automatic” matching 3 combo you'll need to detect what sprite is to the left of your new tile, and what sprite is below your new tile.

To do this, create two ***Sprite*** variables in the `CreateBoard` method, right above the double for-loops:

```
    Sprite[] previousLeft = new Sprite[ySize];
    Sprite previousBelow = null;
```

These variables will be used to hold a reference to adjacent tiles so you can replace their characters.
Take a look at the image below:



The loop iterates through all tiles from the bottom left and goes up one tile at a time. Every iteration gets the character displayed left and below the current tile and removes those from a list of new possible characters.
A random character gets pulled out of that list and assigned to both the left and bottom tiles.
This makes sure there'll never be a row of 3 identical characters from the start.

To make this happen, add the following lines right above `Sprite newSprite = characters[Random.Range(0, characters.Count)];`:

```
List<Sprite> possibleCharacters = new List<Sprite>(); // 1
possibleCharacters.AddRange(characters); // 2

possibleCharacters.Remove(previousLeft[y]); // 3
possibleCharacters.Remove(previousBelow);
```

. Create a list of possible characters for this sprite.

. Add all characters to the list.

. Remove the characters that are on the left and below the current sprite from the list of possible characters.

Next, replace this line:

```
Sprite newSprite = characters[Random.Range(0, characters.Count)];
```

with

```
Sprite newSprite = possibleCharacters[Random.Range(0, possibleCharacters.Count)];
```

This will select a new sprite from the list of possible characters and store it.

Finally, add these lines underneath `newTile.GetComponent().sprite = newSprite;`:

```
previousLeft[y] = newSprite;
previousBelow = newSprite;
```

This assign the `newSprite` to both the tile left and below the current one for the next iteration of the loop to use.

Run the game, and check out your new dynamic grid with non-repeating tiles!

Swapping Tiles

The most important gameplay mechanic of Match 3 games is selecting and swapping adjacent tiles so you can line up 3 in a row. To achieve this you'll need to do some additional scripting. First up is selecting the tile.

Open up ***Tile.cs*** in a code editor. For convenience, this script has already been laid out with a few variables and two methods: `Select` and `Deselect`.

`Select` tells the game that this tile piece has been selected, changes the tile's color, and plays a selection sound effect. `Deselect` returns the sprite back to its original color and tells the game no object is currently selected.

What you don't have is a way for the player to interact with the tiles. A left mouse click seems to be a reasonable option for controls.

Unity has a built-in MonoBehaviour method ready for you to use: `OnMouseDown`.

Add the following method to ***Tile.cs***, right below the `Deselect` method:

```
void OnMouseDown() {
        // 1
    if (render.sprite == null || BoardManager.instance.IsShifting) {
        return;
    }

    if (isSelected) { // 2 Is it already selected?
        Deselect();
    } else {
        if (previousSelected == null) { // 3 Is it the first tile selected?
            Select();
        } else {
            previousSelected.Deselect(); // 4
        }
    }
}
```

. Make sure the game is permitting tile selections. There may be times you don't want players to be able to select tiles, such as when the game ends, or if the tile is empty.

. `if (isSelected)` determines whether to select or deselect the tile. If it's already been selected, deselect it.

. Check if there's already another tile selected. When `previousSelected` is null, it's the first one, so select it.

. If it wasn't the first one that was selected, deselect all tiles.

Save this script and return to the editor.
You should now be able to select and deselect tiles by left clicking them.

All good? Now you can add the swapping mechanism.

Swapping Tiles

Start by opening ***Tile.cs*** and adding the following method named `SwapSprite` underneath the `OnMouseDown` method:

```
public void SwapSprite(SpriteRenderer render2) { // 1
    if (render.sprite == render2.sprite) { // 2
        return;
    }

    Sprite tempSprite = render2.sprite; // 3
    render2.sprite = render.sprite; // 4
    render.sprite = tempSprite; // 5
    SFXManager.instance.PlaySFX(Clip.Swap); // 6
}
```

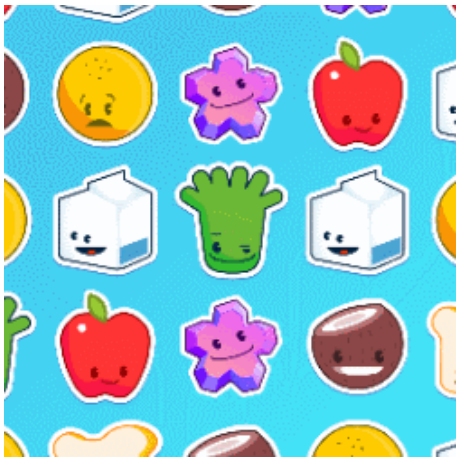
This method will swap the sprites of 2 tiles. Here's how it works:

1. Accept a ***SpriteRenderer*** called `render2` as a parameter which will be used together with `render` to swap sprites.
2. Check `render2` against the ***SpriteRenderer*** of the current tile. If they are the same, do nothing, as swapping two identical sprites wouldn't make much sense.
3. Create a `tempSprite` to hold the sprite of `render2`.
4. Swap out the second sprite by setting it to the first.
5. Swap out the first sprite by setting it to the second (which has been put into `tempSprite`.
6. Play a sound effect.

With the `SwapSprite` method implemented, you can now call it from `OnMouseDown`.
Add this line right above `previousSelected.Deselect();` in the else statement of the `OnMouseDown` method:

```
SwapSprite(previousSelected.render);
```

This will do the actual swapping once you've selected the second tile.
Save this script and return to the editor.
Run the game, and try it out! You should be able to select two tiles and see them swap places:



Finding Adjacent Tiles

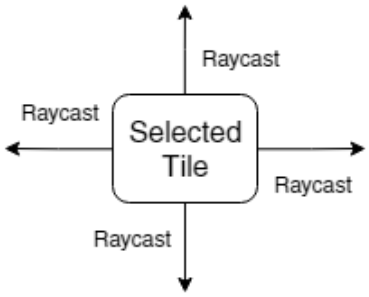
You've probably noticed that you can swap *any* two tiles on the board. This makes the game *way* too easy. You'll need a check to make sure tiles can only be swapped with adjacent tiles.

But how do you easily find tiles adjacent to a given tile?

Open up ***Tile.cs*** and add the following method underneath the `SwapSprite` method:

```
private GameObject GetAdjacent(Vector2 castDir) {
    RaycastHit2D hit = Physics2D.Raycast(transform.position, castDir);
    if (hit.collider != null) {
        return hit.collider.gameObject;
    }
    return null;
}
```

This method will retrieve a single adjacent tile by sending a raycast in the target specified by `castDir`. If a tile is found in that direction, return its GameObject.



Next, add the following method below the `GetAdjacent` method:

```
private List<GameObject> GetAllAdjacentTiles() {
    List<GameObject> adjacentTiles = new List<GameObject>();
    for (int i = 0; i < adjacentDirections.Length; i++) {
        adjacentTiles.Add(GetAdjacent(adjacentDirections[i]));
    }
    return adjacentTiles;
}
```

This method uses `GetAdjacent()` to generate a list of tiles surrounding the current tile. This loops through all directions and adds any adjacent ones found to the `adjacentDirections` which was defined at the top of the script.

With the new handy methods you just created, you can now force the tile to only swap with its adjacent tiles.

Replace the following code in the `OnMouseDown` method:

```
else {
    SwapSprite(previousSelected.render);
    previousSelected.Deselect();
}
```

with this:

```
else {
    if (GetAllAdjacentTiles().Contains(previousSelected.gameObject)) { // 1
        SwapSprite(previousSelected.render); // 2
        previousSelected.Deselect();
    } else { // 3
        previousSelected.GetComponent<Tile>().Deselect();
        Select();
    }
}
```

. Call `GetAllAdjacentTiles` and check if the `previousSelected` game object is in the returned adjacent tiles list.

. Swap the sprite of the tile.

. The tile isn't next to the previously selected one, deselect the previous one and select the newly selected tile instead.

Save this script and return to the Unity editor.
Play your game and poke at it to make sure everything's working as intended. You should only be able to swap two tiles that are adjacent to one another now.

Now you need to handle the *real* point of the game — matching!

Matching

Matching can be broken down into a few key steps:

1. Find 3 or more of the same sprites next to each other and consider it a match.
2. Remove matching tiles.
3. Shift tiles down to fill the empty space.
4. Refill empty tiles along the top.
5. Check for another match.
6. Repeat until no more matches are found.

Open up ***Tile.cs*** and add the following method below the `GetAllAdjacentTiles` method:

```
private List<GameObject> FindMatch(Vector2 castDir) { // 1
    List<GameObject> matchingTiles = new List<GameObject>(); // 2
    RaycastHit2D hit = Physics2D.Raycast(transform.position, castDir); // 3
    while (hit.collider != null && hit.collider.GetComponent<SpriteRenderer>().sprite == r
        matchingTiles.Add(hit.collider.gameObject);
        hit = Physics2D.Raycast(hit.collider.transform.position, castDir);
    }
    return matchingTiles; // 5
}
```

So what's going on here?

1. This method accepts a `Vector2` as a parameter, which will be the direction all raycasts will be fired in.
2. Create a new list of GameObjects to hold all matching tiles.
3. Fire a ray from the tile towards the `castDir` direction.
4. Keep firing new raycasts until either your raycast hits nothing, or the tiles sprite differs from the returned object sprite. If both conditions are met, you consider it a match and add it to your list.
5. Return the list of matching sprites.

Keep up the momentum, and add the following boolean to the top of the file, right above the `Awake` method:

```
private bool matchFound = false;
```

When a match is found, this variable will be set to `true`.
Now add the following method below the `FindMatch` method:

```
private void ClearMatch(Vector2[] paths) // 1
{
    List<GameObject> matchingTiles = new List<GameObject>(); // 2
    for (int i = 0; i < paths.Length; i++) // 3
    {
        matchingTiles.AddRange(FindMatch(paths[i]));
    }
    if (matchingTiles.Count >= 2) // 4
    {
        for (int i = 0; i < matchingTiles.Count; i++) // 5
        {
            matchingTiles[i].GetComponent<SpriteRenderer>().sprite = null;
        }
        matchFound = true; // 6
    }
}
```

This method finds all the matching tiles along the given paths, and then clears the matches respectively.

1. Take a `Vector2` array of paths; these are the paths in which the tile will raycast.
2. Create a GameObject list to hold the matches.
3. Iterate through the list of paths and add any matches to the `matchingTiles` list.
4. Continue if a match with 2 or more tiles was found. You might wonder why 2 matching tiles is enough here, that's because the third match is your initial tile.
5. Iterate through all matching tiles and remove their sprites by setting it `null`.
6. Set the `matchFound` flag to `true`.

Now that you've found a match, you need to clear the tiles. Add the following method below the `ClearMatch` method:

```
public void ClearAllMatches() {
    if (render.sprite == null)
        return;

    ClearMatch(new Vector2[2] { Vector2.left, Vector2.right });
    ClearMatch(new Vector2[2] { Vector2.up, Vector2.down });
    if (matchFound) {
        render.sprite = null;
        matchFound = false;
        SFXManager.instance.PlaySFX(Clip.Clear);
    }
}
```

This will start the domino method. It calls `ClearMatch` for both the vertical and horizontal matches. `ClearMatch` will call `FindMatch` for each direction, left and right, or up and down.

If you find a match, either horizontally or vertically, then you set the current sprite to `null`, reset `matchFound` to `false`, and play the "matching" sound effect.

For all this to work, you need to call `ClearAllMatches()` whenever you make a swap.

In the `OnMouseDown` method, and add the following line just before the `previousSelected.Deselect();` line:

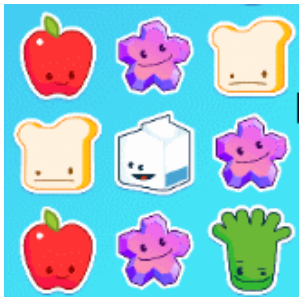
```
previousSelected.ClearAllMatches();
```

Now add the following code directly after the `previousSelected.Deselect();` line:

```
ClearAllMatches();
```

You need to call `ClearAllMatches` on `previousSelected` as well as the current tile because there's a chance *both* could have a match.

Save this script and return to the editor. Press the play button and test out the match mechanic, if you line up 3 tiles of the same type now, they'll disappear.



To fill in the empty space, you'll need to shift and re-fill the board.

Shifting and Re-filling Tiles

Before you can shift the tiles, you need to find the empty ones.
Open up ***BoardManager.cs*** and add the following coroutine below the `CreateBoard` method:

```
public IEnumerator FindNullTiles() {
    for (int x = 0; x < xSize; x++) {
        for (int y = 0; y < ySize; y++) {
            if (tiles[x, y].GetComponent<SpriteRenderer>().sprite == null) {
                yield return StartCoroutine(ShiftTilesDown(x, y));
                break;
            }
        }
    }
}
```

Note: After you've added this coroutine, you'll get an error about `ShiftTilesDown` not exisiting. You can safely ignore that error as you'll be adding that coroutine next!

This coroutine will loop through the entire board in search of tile pieces with ***null*** sprites. When it does find an empty tile, it will start another coroutine `ShiftTilesDown` to handle the actual shifting.

Add the following coroutine below the previous one:

```
private IEnumerator ShiftTilesDown(int x, int yStart, float shiftDelay = .03f) {
    IsShifting = true;
    List<SpriteRenderer>  renders = new List<SpriteRenderer>();
    int nullCount = 0;

    for (int y = yStart; y < ySize; y++) {  // 1
        SpriteRenderer render = tiles[x, y].GetComponent<SpriteRenderer>();
        if (render.sprite == null) { // 2
            nullCount++;
        }
        renders.Add(render);
    }

    for (int i = 0; i < nullCount; i++) { // 3
        yield return new WaitForSeconds(shiftDelay);// 4
        for (int k = 0; k < renders.Count - 1; k++) { // 5
            renders[k].sprite = renders[k + 1].sprite;
            renders[k + 1].sprite = null; // 6
        }
    }
    IsShifting = false;
}
```

`ShiftTilesDown` works by taking in an X position, Y position, and a delay. X and Y are used to determine which tiles to shift. You want the tiles to move down, so the X will remain constant, while Y will change.

The coroutine does the following:

1. Loop through and finds how many spaces it needs to shift downwards.
2. Store the number of spaces in an integer named `nullCount`.
3. Loop again to begin the actual shifting.
4. Pause for `shiftDelay` seconds.
5. Loop through every ***SpriteRenderer*** in the list of `renders`.
6. Swap each sprite with the one above it, until the end is reached and the last sprite is set to `null`

Now you need to stop and start the `FindNullTiles` coroutine whenever a match is found.

Save the ***BoardManager*** script and open up ***Tile.cs***. Add the following lines to the `ClearAllMatches()` method, right above `SFXManager.instance.PlaySFX(Clip.Clear);`:

```
StopCoroutine(BoardManager.instance.FindNullTiles());
StartCoroutine(BoardManager.instance.FindNullTiles());
```

This will stop the `FindNullTiles` coroutine and start it again from the start.

Save this script and return to the edior. Play the game again and make some matches, you'll notice that the board runs out of tiles as you get matches. To make a never-ending board, you need to re-fill it as it clears.

Open ***BoardManager.cs*** and add the following method below `ShiftTilesDown`:

```
private Sprite GetNewSprite(int x, int y) {
    List<Sprite> possibleCharacters = new List<Sprite>();
    possibleCharacters.AddRange(characters);

    if (x > 0) {
        possibleCharacters.Remove(tiles[x - 1, y].GetComponent<SpriteRenderer>().sprite);
    }
    if (x < xSize - 1) {
        possibleCharacters.Remove(tiles[x + 1, y].GetComponent<SpriteRenderer>().sprite);
    }
    if (y > 0) {
        possibleCharacters.Remove(tiles[x, y - 1].GetComponent<SpriteRenderer>().sprite);
    }

    return possibleCharacters[Random.Range(0, possibleCharacters.Count)];
}
```

This snippet creates a list of possible characters the sprite could be filled with. It then uses a series of `if` statements to make sure you don't go out of bounds. Then, inside the `if` statements, you remove possible duplicates that could cause an accidental match when choosing a new sprite. Finally, you return a random sprite from the possible sprite list.

In the coroutine `ShiftTilesDown`, replace:

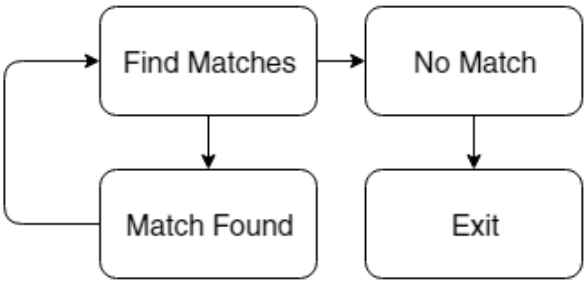
```
renders[k + 1].sprite = null;
```

with:

```
renders[k + 1].sprite = GetNewSprite(x, ySize - 1);
```

This will make sure the board is always filled.

When a match is made and the pieces shift there's a chance another match could be formed. Theoretically, this could go on forever, so you need to keep checking until the board has found all possible matches.



Combos

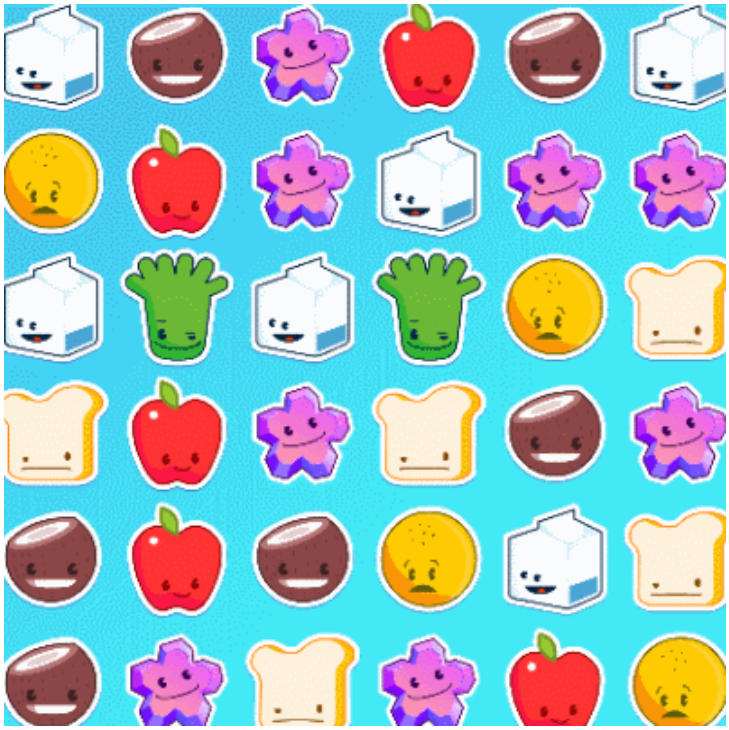
By re-checking all the tiles after a match is found, you'll be able to locate any possible combos that may have been created during the shifting process.
Open up ***BoardManager.cs*** and find the `FindNullTiles()` method.

Add the following `for` at the bottom of the method, below the for loops:

```
for (int x = 0; x < xSize; x++) {
    for (int y = 0; y < ySize; y++) {
        tiles[x, y].GetComponent<Tile>().ClearAllMatches();
    }
}
```

After all of this hard work, it's time to make sure everything works as intended.

Save your work and ***Run*** the game. Start swapping tiles and watch the endless supply of new tiles keep the board filled as you play.



Moving the Counter and Keeping Score

It's time keep track of the player's moves and their score.

Open ***GUIManager.cs*** located under ***Scripts\Managers*** in your favorite code editor. This script handles the UI aspects of the game, including the move counter and score keeper.

To get started add this variable to the top of the file, below `private int score;`:

```
private int moveCounter;
```

Now add this at the top of the `Awake()` method to initialize the number of moves the player can do:

```
moveCounter = 60;
moveCounterTxt.text = moveCounter.ToString();
```

Now you need to encapsulate both integers so you can update the ***UI Text*** every time you update the value. Add the following code right above the `Awake()` method:

```
public int Score {
    get {
        return score;
    }

    set {
        score = value;
        scoreTxt.text = score.ToString();
    }
}

public int MoveCounter {
    get {
        return moveCounter;
    }

    set {
        moveCounter = value;
        moveCounterTxt.text = moveCounter.ToString();
    }
}
```

These will make sure that every time the variables `Score` or `MoveCounter` are changed, the text components representing them get updated as well. You could've put the text updating in an `Update()` method, but doing it this way is much better for performance, especially as it involves handling strings.

Time to start adding points and tracking moves! Every time the player clears a tile they will be rewarded with some points.

Save this script and open up ***BoardManager.cs***. Add the following to the `ShiftTilesDown` method, right above `yield return new WaitForSeconds(shiftDelay);`:

```
GUIManager.instance.Score += 50;
```

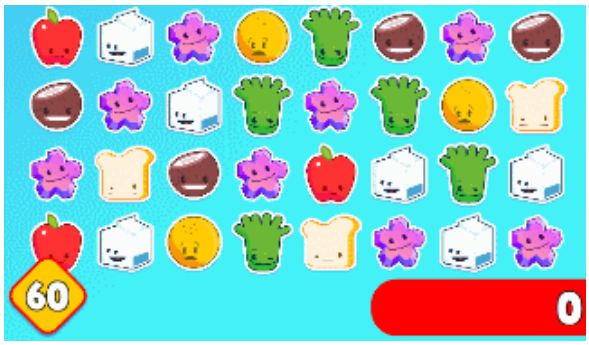
This will increase the score every time an empty tile is found.

Over in ***Tile.cs***, add the following line below `SFXManager.instance.PlaySFX(Clip.Swap);` in the `ClearAllMatches` method:

```
GUIManager.instance.MoveCounter--;
```

This will decrement `MoveCounter` every time a sprite is swapped.

Save your work and test out if the move and score counters are working correctly. Each move should substract the move counter and each match should award some points.



Game Over Screen

The game should end when the move counter reaches 0. Open up ***GUIManager.cs*** and add the following `if` statement to `MoveCounter`s's setter, right below `moveCounter = value;`:

```
if (moveCounter <= 0) {
    moveCounter = 0;
    GameOver();
}
```

This will work — mostly. Because `GameOver()` is called right on the final move, combos won't count towards the final score. That would get you a one-star review for sure!

To prevent this, you need to create a coroutine that waits until ***BoardManager.cs*** has finished all of its shifting. Then you can call `GameOver()`.

Add the following coroutine to ***GUIManager.cs*** below the `GameOver()` method:

```
private IEnumerator WaitForShifting() {
    yield return new WaitUntil(()=> !BoardManager.instance.IsShifting);
    yield return new WaitForSeconds(.25f);
    GameOver();
}
```

Now replace the following line in the `MoveCounter` setter:

```
GameOver();
```

With:

```
StartCoroutine(WaitForShifting());
```

This will make sure all combos will get calculated before the game is over.
Now save all scripts, play the game and score those combos! :]

Where to Go From Here?

You can download the final project here (https://koenig-media.raywenderlich.com/uploads/2017/02/Match-3-Game-Final.zip).

Now you know how to make a basic Match 3 game by using Unity! I encourage you to keep working and building on this tutorial! Try adding the following on your own:

- Timed mode
- Different levels with various board sizes
- Bonus points for combos
- Add some particles to make cool effects

If you're enjoyed this tutorial want to learn more, you should definitely check out our book Unity Games by Tutorials(https://www.raywenderlich.com/store/unity-games-by-tutorials), which teaches you to make 4 full games, scripting in C# and much more.

Here are a few resources to learn even more:

- Unity scripting tutorials(https://unity3d.com/learn/tutorials/topics/scripting): Video tutorials teaching you the basics of scripting in Unity.
- Introduction to Unity UI(https://www.raywenderlich.com/149464/introduction-to-unity-ui-part-1): Master Unity's UI system in this tutorial series.
- Introduction To Unity Particle Systems(https://www.raywenderlich.com/113049/introduction-unity-particle-systems): Learn how particle systems work and spruce up your game with some sweet effects!

Can't wait to see what you all come up with! If you have any questions or comments you can post them in the ***Comments*** section.

raywenderlich.com Weekly

The raywenderlich.com newsletter is the easiest way to stay up-to-date on everything you need to know as a mobile developer.

stevewozniak@apple.com

Get a weekly digest of our tutorials and courses, and receive a free in-depth email course as a bonus!

Reviews

There aren't any reviews yet!

Write the first review and let us know what you think.