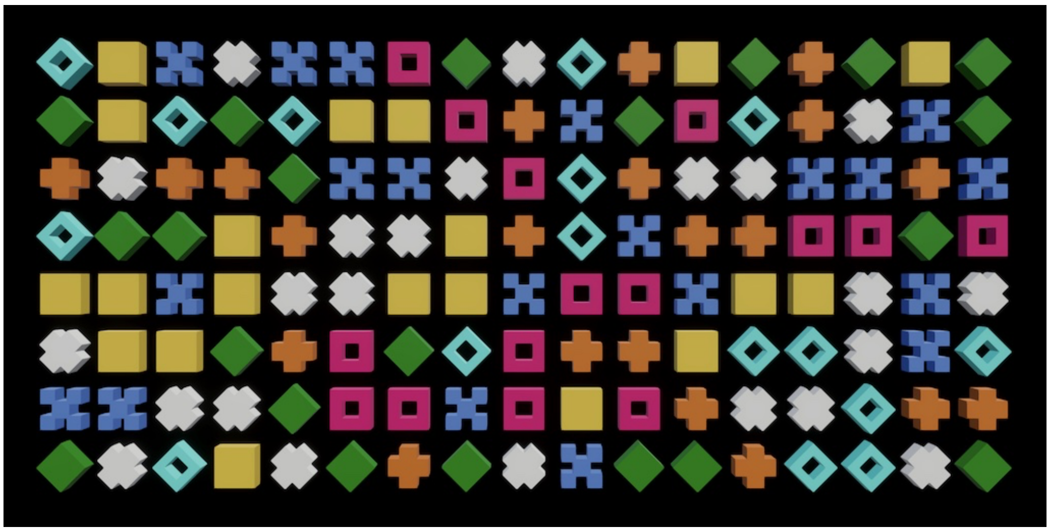
published 2023-06-30

Match 3 Matching Tiles

Match sequences of tiles in a 2D grid.
Keep visualization and game logic separate.
Animate state transitions.
Accumulate match scores.
Search for possible moves.

This is the sixth tutorial in a series about prototypes. In it we will create a simple match-3 game.

This tutorial is made with Unity 2022.3.2f1.



Tiles waiting to be matched.

1 The Game

There is a great variety of match-3 and similar games, like multiple versions of *Bejeweled* and *Puzzle Quest* and others. This time we will make a very simple game in this genre, matching tiles in rows and columns of at least three of the same. We'll simple name it *Match 3*. One obvious way that these games use to distinguish themselves is by differing their visuals, while the gameplay is mostly the same. So this time we'll strictly separate the game logic and visualization. This introduces a layer of indirection, but these games are so small and simple that we don't need to worry about performance much.

Once again we can start with a duplicate of the *Paddle Square* project and remove everything that we do not need. We only keep the global post-FX volume, the camera, and a main light. Disable shadows and clear the rotation of the camera, setting its position to (0, 0, −10) as the game is positioned on the XY axes. Also keep the text prefab.

Also, this time we'll fully support hot reloading so we can change code while the game is playing, to make debugging and testing easier.

1.1 Match 3 Skin

To support easy switching of visualization while keeping the game logic the same, we introduce a `Match3Skin` component type that acts as a proxy for the actual game logic. The main game object will only interact with this skin. To facilitate this interaction give it a public property to indicate whether the match-3 game is playing, one to indicate whether it is currently busy, and methods to start a new game and to do its work. These are initially dummies, indicating that we're always playing and never busy. Create a *Match 3* game object with this component.

```
using TMPro;
using Unity.Mathematics;
using UnityEngine;

using static Unity.Mathematics.math;

public class Match3Skin : MonoBehaviour
{
	public bool IsPlaying => true;

	public bool IsBusy => false;

	public void StartNewGame () {}

	public void DoWork () { }
}
```

Next, create a game object with a `Game` component that acts and the main controller of the game, as usual. Give it a configuration field for the match-3 game, which we treat as the actual game even through it is the skin. Have it start a new game when it awakens.

```
using UnityEngine;

public class Game : MonoBehaviour
{
    [SerializeField]
    Match3Skin match3;

    void Awake () => match3.StartNewGame();
}
```

Each update, if the match-3 game is playing, handle player input if the game isn't already busy, and after that tell it to do its work. If the game is not playing start a new game if space is pressed. Handling input is done is a separate method, initially doing nothing.

```
    void Update ()
    {
        if (match3.IsPlaying)
        {
            if (!match3.IsBusy)
            {
                HandleInput();
            }
            match3.DoWork();
        }
        else if (Input.GetKeyDown(KeyCode.Space))
        {
            match3.StartNewGame();
        }
    }

    void HandleInput () { }
```

Note that with this approach it would be fairly simple to support multiple skins, game modes, or multiple games in a single app, by changing the active game.

1.2 Handling Input

We'll add support for a single input method to `Match3Skin`, by giving it a public `EvaluateDrag` method. This method evaluates an ongoing drag action, given a start and end position. These are `Vector3` values that represent mouse or touch positions in screen space. The method returns whether the drag should be continued or aborted. We don't process input at this point yet so there's no reason to maintain the drag.

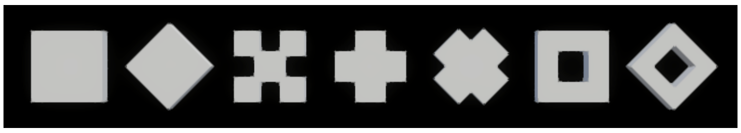
```
	public bool EvaluateDrag (Vector3 start, Vector3 end)
	{
		return false;
	}
```

To support dragging, `Game` needs to keep track the drag start and whether it is dragging. In `HandleInput`, if we're not already dragging and the primary mouse button is pressed down, start a drag. Otherwise if we're dragging and the button is still pressed, evaluate the drag and use this to decide whether to continue dragging. Otherwise end the drag.

```
	Vector3 dragStart;

	bool isDragging;

	…

	void HandleInput ()
	{
		if (!isDragging && Input.GetMouseButtonDown(0))
		{
			dragStart = Input.mousePosition;
			isDragging = true;
		}
		else if (isDragging && Input.GetMouseButton(0))
		{
			isDragging = Match3Game.EvaluateDrag(dragStart, Input.mousePosition);
		}
		else
		{
			isDragging = false;
		}
	}
```

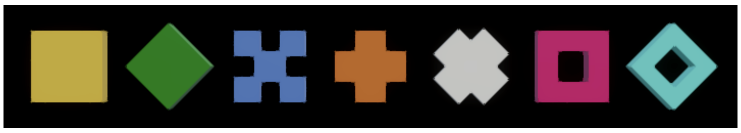
1.3 Tiles

To visualize the match-3 game we need to show tiles. We use small square tiles with a side length of one unit. These tiles are aligned on the XY plane and we'll make them 0.2 units thick. As usual I only use cubes to visualize them. Create seven tile prefabs that are easy to distinguish even when they are all the same color. They should all have a root game object that has the identity transformation, so without rotation and with a scale of 1.



Seven tiles.

Once the tiles are visually distinct enough you can give each a different color. Doing it in this order ensures that colorblind people will also be able tell the tiles apart.



Colored tiles.

We'll be working through lots of tile instances, so we're going to pool them. But we'll pool something else later as well, so to avoid duplicate code let's introduce a generic `PrefabInstancePool` struct type for `MonoBehaviour` prefabs that wraps a stack. Give it a public `GetInstance` method that instantiates a gives prefab and a `Recycle` method that destroys a given instance's game object. This pool isn't serializable.

```
using System.Collections.Generic;
using UnityEngine;

public struct PrefabInstancePool<T> where T : MonoBehaviour
{
	Stack<T> pool;

	public T GetInstance (T prefab)
	{
		return Object.Instantiate(prefab);
	}

	public void Recycle (T instance)
	{
		Object.Destroy(instance.gameObject);
	}
}
```

Adjust the `GetInstance` method so it uses the pool: create it if needed, reuse and instance if available and reactivate its game object, otherwise create a new instance.

```
    public T GetInstance (T prefab)
    {
        if (pool == null)
        {
            pool = new();
        }

        if (pool.TryPop(out T instance))
        {
            instance.gameObject.SetActive(true);
        }
        else
        {
            instance = Object.Instantiate(prefab);
        }
        return instance;
    }
```

To support disabled domain reloading, if the pool exist check if it contains a reference to something that has been destroyed. If so we assume that the pool survived exiting play mode and thus clear it to get rid of the old references. This is only needed in the editor.

```
        if (pool == null)
        {
            pool = new();
        }
#if UNITY_EDITOR
        else if (pool.TryPeek(out T i) && !i)
        {
            // Instances destroyed, assuming due to exiting play mode.
            pool.Clear();
        }
#endif
```

When recycling, if we're in the editor check whether the pool is missing. If so we assume the reference got lost due to a hot reload and only then do we destroy the game object. Otherwise we add the instance to the pool and deactivate its game object.

Note that this only gets rid of game objects that were still in use during the hot reload. Those that were already recycled will never get reused, as the pool that referenced them is gone. So the amount of permanently disabled instances will slowly grow with each hot reload, until play mode is exited.

```
    public void Recycle (T instance)
    {
#if UNITY_EDITOR
        if (pool == null)
        {
            // Pool lost, assuming due to hot reload.
            Object.Destroy(instance.gameObject);
            return;
        }
#endif
        pool.Push(instance);
        instance.gameObject.SetActive(false);
    }
```

Now create a `Tile` component type that uses this pool. Give it a public `Spawn` method that gets an instance of itself, gives it the same pool, and places it at a give position. Also give it a public `Despawn` method that recycles itself.

```
using UnityEngine;

public class Tile : MonoBehaviour
{
    PrefabInstancePool<Tile> pool;

    public Tile Spawn (Vector3 position)
    {
        Tile instance = pool.GetInstance(this);
        instance.pool = pool;
        instance.transform.localPosition = position;
        return instance;
    }

    public void Despawn () => pool.Recycle(this);
}
```

Add this component to the root of all seven tile prefabs. Then add a configuration array for tile prefabs to `Match3Skin` and assign the tiles to it.

```
    [SerializeField]
    Tile[] tilePrefabs;
```

2 Basic Gameplay

The basic match-3 gameplay consists of filling a 2D grid with tiles, then the player swaps tiles to make matches. Matching tiles are cleared, then the holes are filled by dropping the tiles above them, adding new tiles as needed.

2.1 2D Grid

Both the game logic and its skin will have to work with 2D grids, so we introduce a generic serializable `Grid2D` struct to facilitate this. It uses an internal array of cells to store its data and also keeps track of its 2D size as an `int2`, which is passed to its constructor method.

```
using Unity.Mathematics;

[System.Serializable]
public struct Grid2D<T>
{
    T[] cells;

    int2 size;

    public Grid2D (int2 size)
    {
        this.size = size;
        cells = new T[size.x * size.y];
    }
}
```

Why not just use a multidimensional array?

Those cannot be serialized by Unity, so cannot survive hot reloads. Also, we give it a few extra convenient properties and methods.

Add public getter properties for its size, and also for its individual size components.

```
    public int2 Size => size;

    public int SizeX => size.x;

    public int SizeY => size.y;
```

Also make it possible to conveniently check whether the grid is undefined in the context of Unity's serialization, which means that it either lacks an array or the array length is zero.

```
    public bool IsUndefined => cells == null || cells.Length == 0;
```

Give it two indexers to get and set grid elements, either with separate X and Y coordinates or with a single `int2` coordinates pair.

```
	public T this[int x, int y]
	{
		get => cells[y * size.x + x];
		set => cells[y * size.x + x] = value;
	}

	public T this[int2 c]
	{
		get => cells[c.y * size.x + c.x];
		set => cells[c.y * size.x + c.x] = value;
	}
```

Include a method to check whether given coordinates are valid. We only need a version with a single `int2` parameter, but you could add one with separate coordinate parameters as well.

```
	public bool AreValidCoordinates (int2 c) =>
		0 <= c.x && c.x < size.x && 0 <= c.y && c.y < size.y;
```

Finally, add a method to swap two elements, given their coordinates.

```
	public void Swap (int2 a, int2 b) => (this[a], this[b]) = (this[b], this[a]);
```

How does that code work?

The code `(a, b) = (b, a)` is shorthand for `var t = a; a = b; b = t;`.

2.2 Starting the Game

To store the game state we need to represent the tiles, for which we create a `TileState` enum, naming the seven states A through G. Also include a default zero state named `None` to represent an empty tile space.

```
public enum TileState
{
	None, A, B, C, D, E, F, G
}
```

The game state and logic will be taken care of by a new `Match3Game` component type. It can be added to the same game object that also has the `Match3Skin` component. Give it a configuration field for its size, set to 8×8 by default. Also give it a `TileState` grid and getter properties that forward to the grid's indexers and its size. This ensures that only `Match3Game` can change the grid state.

```
using System.Collections.Generic;
using Unity.Mathematics;
using UnityEngine;

using Random = UnityEngine.Random;

using static Unity.Mathematics.math;

public class Match3Game : MonoBehaviour
{
    [SerializeField]
    int2 size = 8;

    Grid2D<TileState> grid;

    public TileState this[int x, int y] => grid[x, y];

    public TileState this[int2 c] => grid[c];

    public int2 Size => size;
}
```

Give it a public `StartNewGame` method that creates an new grid if it is undefined, then invokes `FillGrid` which loops through all grid rows and fills them with random states.

```
    public void StartNewGame ()
    {
        if (grid.IsUndefined)
        {
            grid = new(size);
        }
        FillGrid();
    }

    void FillGrid ()
    {
        for (int y = 0; y < size.y; y++)
        {
            for (int x = 0; x < size.x; x++)
            {
                grid[x, y] = (TileState)Random.Range(1, 8);
            }
        }
    }
```

Next, add a `Match3Game` configuration field to `Match3Skin` and hook it up. Then give it its own grid with `Tile` elements, plus a 2D tile offset to position the tiles in world space.

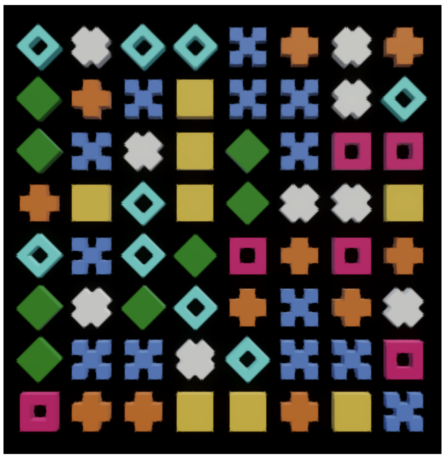
```
    [SerializeField]
    Match3Game game;

    Grid2D<Tile> tiles;

    float2 tileOffset;
```

When starting a new game first forward the invocation to the actual game. Then set the tile offset to center the tiles on the origin. Then create a new tiles grid if it is undefined, otherwise despawn all tiles and clear their references. Clearing the references isn't strictly necessary but makes it easier to detect bugs.

```
	public void StartNewGame () {
		…

		game.StartNewGame();
		tileOffset = -0.5f * (float2)(game.Size - 1);
		if (tiles.IsUndefined)
		{
			tiles = new(game.Size);
		}
		else
		{
			for (int y = 0; y < tiles.SizeY; y++)
			{
				for (int x = 0; x < tiles.SizeX; x++)
				{
					tiles[x, y].Despawn();
					tiles[x, y] = null;
				}
			}
		}
	}
```

After that loop through all rows and spawn the appropriate tile instances and add them to the grid. Create a separate `SpawnTile` method for this that spawns a single tile given a tile state and coordinates.

```
	public void StartNewGame () {
		…

		for (int y = 0; y < tiles.SizeY; y++)
		{
			for (int x = 0; x < tiles.SizeX; x++)
			{
				tiles[x, y] = SpawnTile(game[x, y], x, y);
			}
		}
	}

	Tile SpawnTile (TileState t, float x, float y) =>
		tilePrefabs[(int)t - 1].Spawn(new Vector3(x + tileOffset.x, y + tileOffset.y));
```



Initial random game state.

2.3 Avoiding Immediate Matches

When entering play mode we now see a grid filled with random tiles. As this is completely random it is possible that this initial state already contains horizontal or vertical sequences of three or more matching tiles. To avoid this we have to adjust `Match3Game.FillGrid`.

Each tile that gets placed can generate up to two matches, one horizontal and one vertical. We have to detect these so we can avoid them. So keep track of two tiles states— A and B— and how many potential matches we have detected. If there are at least two tiles to the left then set A to one of them and if both are equal set the potential match count to 1.

```
	void FillGrid ()
	{
		for (int y = 0; y < size.y; y++)
		{
			for (int x = 0; x < size.x; x++)
			{
				TileState a = TileState.None, b = TileState.None;
				int potentialMatchCount = 0;
				if (x > 1)
				{
					a = grid[x - 1, y];
					if (a == grid[x - 2, y])
					{
						potentialMatchCount = 1;
					}
				}

				grid[x, y] = (TileState)Random.Range(1, 8);
			}
		}
	}
```

Follow this by doing the same for the two tiles downward, assigning one to B and incrementing the potential match count. However, if there is only a single match use A instead, and if there are two matches ensure that A and B are ordered lowest to highest.

```
                if (x > 1)
                {
                    …
                }
                if (y > 1)
                {
                    b = grid[x, y - 1];
                    if (b == grid[x, y - 2])
                    {
                        potentialMatchCount += 1;
                        if (potentialMatchCount == 1)
                        {
                            a = b;
                        }
                        else if (b < a)
                        {
                            (a, b) = (b, a);
                        }
                    }
                }
```

Now we can avoid making a match by decreasing the random range by the match count and skipping A and B if needed.

```
                TileState t = (TileState)Random.Range(1, 8 - potentialMatchCount);
                if (potentialMatchCount > 0 && t >= a)
                {
                    t += 1;
                }
                if (potentialMatchCount == 2 && t >= b)
                {
                    t += 1;
                }
                grid[x, y] = t;
```

2.4 Performing a Move

A move in this game consists of picking a tile and swapping it with one of its direct neighbors. Introduce a `MoveDirection` enum for the allowed directions, which are up, right, down, and left, plus a default zero state named `None` to represent an invalid move.

```
public enum MoveDirection
{
    None, Up, Right, Down, Left
}
```

Also create a serializable `Move` struct type to store an entire move action, with properties for its direction, from and to coordinates, plus a property that indicates whether the move is valid, based on whether its direction is set. The properties are privately set via the constructor method, which only requires the starting coordinates and direction and determines the destination coordinates from those.

```
using Unity.Mathematics;

using static Unity.Mathematics.math;

[System.Serializable]
public struct Move
{
    public MoveDirection Direction
    { get; private set; }

    public int2 From
    { get; private set; }

    public int2 To
    { get; private set; }

    public bool IsValid => Direction != MoveDirection.None;

    public Move (int2 coordinates, MoveDirection direction)
    {
        Direction = direction;
        From = coordinates;
        To = coordinates + direction switch
        {
            MoveDirection.Up => int2(0, 1),
            MoveDirection.Right => int2(1, 0),
            MoveDirection.Down => int2(0, -1),
            _ => int2(-1, 0)
        };
    }
}
```

Couldn't this be a readonly struct?

Yes, except that Unity's serialization doesn't support readonly values, so a readonly move wouldn't survive hot reloads.

Now we can add a public `TryMove` method to `Match3Game` that takes a move and returns whether it was successful, meaning that it resulted in a match. For now make it always swap the from and to tiles and indicate success.

```
    public bool TryMove (Move move)
    {
        grid.Swap(move.From, move.To);
        return true;
    }
```

Next, add a private `DoMove` method to `Match3Skin` that tries a given move and if successful swaps both the tile positions and the tiles themselves.

```
	void DoMove (Move move)
	{
		if (game.TryMove(move))
		{
			(
				tiles[move.From].transform.localPosition,
				tiles[move.To].transform.localPosition
			) = (
				tiles[move.To].transform.localPosition,
				tiles[move.From].transform.localPosition
			);
			tiles.Swap(move.From, move.To);
		}
	}
```

To know which tiles we should swap introduce a method that converts from screen space to the skin's tile space.

```
	float2 ScreenToTileSpace (Vector3 screenPosition)
	{
		Ray ray = Camera.main.ScreenPointToRay(screenPosition);
		Vector3 p = ray.origin - ray.direction * (ray.origin.z / ray.direction.z);
		return float2(p.x - tileOffset.x + 0.5f, p.y - tileOffset.y + 0.5f);
	}
```

Now we can convert the start and end of a drag to tile coordinates in `EvaluateDrag`. Add a configuration field for a drag threshold, set to half a tile by default. Use that to determine the drag direction by checking the drag delta. Then if the move and both sets of tile coordinates are valid do the move and return **`false`** to indicate that the drag should stop. Otherwise return **`false`** to indicate that the drag can continue.

```
	[SerializeField, Range(0.1f, 1f)]
	float dragThreshold = 0.5f;

	…

	public bool EvaluateDrag (Vector3 start, Vector3 end)
	{
		float2 a = ScreenToTileSpace(start), b = ScreenToTileSpace(end);
		var move = new Move(
			(int2)floor(a), (b - a) switch
			{
				var d when d.x > dragThreshold => MoveDirection.Right,
				var d when d.x < -dragThreshold => MoveDirection.Left,
				var d when d.y > dragThreshold => MoveDirection.Up,
				var d when d.y < -dragThreshold => MoveDirection.Down,
				_ => MoveDirection.None
			}
		);
		if (
			move.IsValid &&
			tiles.AreValidCoordinates(move.From) && tiles.AreValidCoordinates(move.To)
			)
		{
			DoMove(move);
			return false;
		}
		return true;
	}
```

It is now possible to swap a tile with its neighbor via dragging.

2.5 Finding Matches

To only allow moves that result in a match we have to scan the grid for matches. To facilitate this we introduce a serializable `Match` struct type. This is just a value container so we make all fields public. It has coordinates that indicate the first tile of the match—the bottom left one—its length, and whether it is a horizontal match. Give it a convenient constructor method that sets these fields, using separate parameters for the X and Y coordinates.

```
using Unity.Mathematics;

[System.Serializable]
public struct Match
{
    public int2 coordinates;

    public int length;

    public bool isHorizontal;

    public Match (int x, int y, int length, bool isHorizontal)
    {
        coordinates.x = x;
        coordinates.y = y;
        this.length = length;
        this.isHorizontal = isHorizontal;
    }
}
```

`Match3Game` won't immediately process matches that it finds but will store them in a list. This allows the skin to do other work in between detecting and processing matches. The matches themselves will remain private to `Match3Game`, but it will expose whether it has any matches via a property. Also create the list at the start of a game if needed.

```
    List<Match> matches;

    …

    public bool HasMatches => matches.Count > 0;

    public void StartNewGame ()
    {
        if (grid.IsUndefined)
        {
            grid = new(size);
            matches = new();
        }
        FillGrid();
    }
```

Add a `FindMatches` method that returns whether any matches were found, as a shorthand for invoking the propery afterwards.

```
    bool FindMatches ()
    {
        return HasMatches;
    }
```

We begin by searching for horizontal matches. For each row, set the start tile state to the first tile and set the match length to 1. Then loop through the rest of the row, increasing the length as long as the current tile matches the start. If there isn't a match and the length is at least 3 add the horizontal match to the list, then reset the start. Also check for a 3+ match at the end of the row.

```
    bool FindMatches ()
    {
        for (int y = 0; y < size.y; y++)
        {
            TileState start = grid[0, y];
            int length = 1;
            for (int x = 1; x < size.x; x++)
            {
                TileState t = grid[x, y];
                if (t == start)
                {
                    length += 1;
                }
                else
                {
                    if (length >= 3)
                    {
                        matches.Add(new Match(x - length, y, length, true));
                    }
                    start = t;
                    length = 1;
                }
            }
            if (length >= 3)
            {
                matches.Add(new Match(size.x - length, y, length, true));
            }
        }

        return HasMatches;
    }
```

Then do the same for vertical matches, looping through columns instead of rows.

```
		for (int y = 0; y < size.y; y++)
		{
			…
		}

		for (int x = 0; x < size.x; x++)
		{
			TileState start = grid[x, 0];
			int length = 1;
			for (int y = 1; y < size.y; y++)
			{
				TileState t = grid[x, y];
				if (t == start)
				{
					length += 1;
				}
				else
				{
					if (length >= 3)
					{
						matches.Add(new Match(x, y - length, length, false));
					}
					start = t;
					length = 1;
				}
			}
			if (length >= 3)
			{
				matches.Add(new Match(x, size.y - length, length, false));
			}
		}

		return HasMatches;
```

Make `TryMove` invoke `FindMatches` after swapping and if it indicates success return **`true`**. Otherwise undo the swap and return **`false`**.

```
	public bool TryMove (Move move)
	{
		grid.Swap(move.From, move.To);
		if (FindMatches())
		{
			return true;
		}
		grid.Swap(move.From, move.To);
		return false;
	}
```

If we enter play mode at this point we can initially only successfully make a move that results in a match. However, after that we can move freely, because the matches aren't cleared yet.

2.6 Processing Matches

After matches are found they need to be processed. Processing matches means clearing all matched tiles. To communicate that to the skin we add a public property that is privately set to `Match3Game` exposing a list of cleared tile coordinates. Also add a property that indicates whether the game state needs to be filled, which is privately set.

```
	public List<int2> ClearedTileCoordinates
	{ get; private set; }

	public bool NeedsFilling
	{ get; private set; }

	public bool HasMatches => matches.Count > 0;

	public void StartNewGame ()
	{
		if (grid.IsUndefined)
		{
			grid = new(size);
			matches = new();
			ClearedTileCoordinates = new();
		}
		FillGrid();
	}
```

Doesn't `NeedsFilling` directly depend on the list's count?

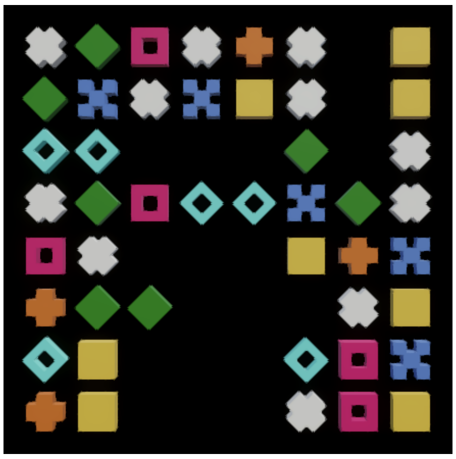
Logically yes, but because the list is publicly accessible it could be modified externally. So we don't depend the list's state, it is solely used to communicate changes.

Create a public `ProcessMatches` method that clears the list of cleared tiles coordinates and then loops through all matches and clears all their tiles, adding their coordinates to the list. Because horizontal and vertical matches can overlap make sure to include each cleared tile only once. When done clear the matches and indicate that filling is needed.

```
	public void ProcessMatches ()
	{
		ClearedTileCoordinates.Clear();

		for (int m = 0; m < matches.Count; m++)
		{
			Match match = matches[m];
			int2 step = match.isHorizontal ? int2(1, 0) : int2(0, 1);
			int2 c = match.coordinates;
			for (int i = 0; i < match.length; c += step, i++)
			{
				if (grid[c] != TileState.None)
				{
					grid[c] = TileState.None;
					ClearedTileCoordinates.Add(c);
				}
			}
		}

		matches.Clear();
		NeedsFilling = true;
	}
```

`Match3Skin` now has potential work to do. If the game has matches invoke a new `ProcessMatches` method that forwards to the game and despawns all cleared tiles and clears their references.

```
	public void DoWork () {
		if (game.HasMatches)
		{
			ProcessMatches();
		}
	}

	void ProcessMatches ()
	{
		game.ProcessMatches();

		for (int i = 0; i < game.ClearedTileCoordinates.Count; i++)
		{
			int2 c = game.ClearedTileCoordinates[i];
			tiles[c].Despawn();
			tiles[c] = null;
		}
	}
```


Matches leave holes.

Holes now start appearing when we make matches. Trying to swap holes will produce a `NullReferenceException` because we cleared those references.

2.7 Filling Holes

The next step is to fill the holes created by matches. We do this by applying gravity, dropping tiles that are now floating. To communicate this we introduce a serializable `TileDrop` struct type. It contains public fields for the coordinates of a tile and a `fromY` coordinate to indicate where it fell from. Give it a convenient constructor method with separate parameters for its coordinates.

```
using Unity.Mathematics;

[System.Serializable]
public struct TileDrop
{
    public int2 coordinates;

    public int fromY;

    public FallenTile (int x, int y, int distance)
    {
        coordinates.x = x;
        coordinates.y = y;
        fromY = y + distance;
    }
}
```

Add a list of dropped tiles to `Match3Game`, just like the list of cleared tiles.

```
    public List<TileDrop> DroppedTiles
    { get; private set; }

    …

    public void StartNewGame ()
    {
        if (grid.IsUndefined)
        {
            grid = new(size);
            matches = new();
            ClearedTileCoordinates = new();
            DroppedTiles = new();
        }
        FillGrid();
    }
```

Then create a public `DropTiles` method that clears the list and then loops through all columns. It goes from bottom to top, keeping track of the hole count. If it encounters a hole, increase the count. Otherwise, if there are holes below it, drop down the tile state the appropriate distance and add an entry to the list. When done indicate that filling is no longer needed.

```
    public void DropTiles ()
    {
        DroppedTiles.Clear();

        for (int x = 0; x < size.x; x++)
        {
            int holeCount = 0;
            for (int y = 0; y < size.y; y++)
            {
                if (grid[x, y] == TileState.None)
                {
                    holeCount += 1;
                }
                else if (holeCount > 0)
                {
                    grid[x, y - holeCount] = grid[x, y];
                    DroppedTiles.Add(new TileDrop(x, y - holeCount, holeCount));
                }
            }
        }

        NeedsFilling = false;
    }
```

This drops down all existing tiles, effectively pushing all holes to the top. Fill these holes with random tiles per column. Also add drop entries for these new tiles, with their origin Y coordinate set appropriately above the grid.

```
		for (int x = 0; x < size.x; x++)
		{
			int holeCount = 0;
			for (int y = 0; y < size.y; y++)
			{
				…
			}

			for (int h = 1; h <= holeCount; h++)
			{
				grid[x, size.y - h] = (TileState)Random.Range(1, 8);
				DroppedTiles.Add(new TileDrop(x, size.y - h, holeCount));
			}
		}
```

`Match3Skin` now has more work to do. If it doesn't have matches to process and the game needs filling then it invokes its own `DropTiles` method, which forwards to the game. Then it loops through all dropped tiles. If the tile fell from within the grid adjust its position. Otherwise spawn a new tile at the appropriate position. Then update the tiles grid.

```
	public void DoWork () {
		if (game.HasMatches)
		{
			ProcessMatches();
		}
		else if (game.NeedsFilling)
		{
			DropTiles();
		}
	}

	void DropTiles ()
	{
		game.DropTiles();

		for (int i = 0; i < game.DroppedTiles.Count; i++)
		{
			TileDrop drop = game.DroppedTiles[i];
			Tile tile;
			if (drop.fromY < tiles.SizeY)
			{
				tile = tiles[drop.coordinates.x, drop.fromY];
				tile.transform.localPosition = new Vector3(
					drop.coordinates.x + tileOffset.x, drop.coordinates.y + tileOffset.y
				);
			}
			else
			{
				tile = SpawnTile(
					game[drop.coordinates], drop.coordinates.x, drop.coordinates.y
				);
			}
			tiles[drop.coordinates] = tile;
		}
	}
```

The fallen tiles can immediately form new matches, so we should invoke `FindMatches` again at the end of `Match3Game.DropTiles`. This will cause the game to keep cascading until it reaches a state without any matches.

```
        NeedsFilling = false;
        FindMatches();
```

3 Animating Transitions

Our game is minimally functional at this point, but it is hard to see what is going on because it immediately switches to the next state. So we're going to slow down the game by introducing state transitions. This won't affect the logic of the game, only its skin.

3.1 Swapping Tiles

The first thing that we'll do is animate the tile swapping caused by a move. To isolate this logic from the skin let's create a serializable `TileSwapper` class for it. Give it a configurable duration set to 0.25 seconds by default, but give it a high maximum like 10 so you can slow it down to easily check the transition. Also give it a configurable max depth offset set to 0.5 units by default, which controls how far tiles move in the Z dimension to avoid penetrating each other.

```
using UnityEngine;

[System.Serializable]
public class TileSwapper
{
    [SerializeField, Range(0.1f, 10f)]
    float duration = 0.25f;

    [SerializeField, Range(0f, 1f)]
    float maxDepthOffset = 0.5f;
}
```

It needs a public `Swap` method to initiate a swap along with an `Update` method to animate it. The `Swap` method has parameters for the tiles to be swapped and a parameter to indicated whether the swap should pingpong back to its original position. It returns the duration of the swap animation.

```
    public float Swap (Tile a, Tile b, bool pingPong) {
        return duration;
    }

    public void Update () {}
```

Add a configuration field for the swapper to `Match3Skin` along with a field for its busy duration, which indicates how long the skin is going to be busy with something. Set it to zero at the start of a game and make `IsBusy` return whether it is greater than zero.

```
	[SerializeField]
	TileSwapper tileSwapper;

	float busyDuration;

	…

	public bool IsBusy => busyDuration > 0f;

	public void StartNewGame () {
		busyDuration = 0f;
		…
	}
```

Now `DoWork` should begin by checking whether the busy duration is greater than zero. If so update the tile swapper, decrease the remaining duration, and if there is still time remaining return. This delays progressing the game state until after the swappper is finished.

```
	public void DoWork () {
		if (busyDuration > 0f)
		{
			tileSwapper.Update();
			busyDuration -= Time.deltaTime;
			if (busyDuration > 0f)
			{
				return;
			}
		}

		…
	}
```

Why not query the swapper instead of using a busy duration?

Using a separate busy duration makes it easier to overlap multiple transitions.

Adjust `DoMove` so it no longer adjust the tile positions itself but instead activates the swapper and sets the busy duration. Make it pingpong is the move didn't succeed.

```
	void DoMove (Move move)
	{
		bool success = game.TryMove(move);
		Tile a = tiles[move.From], b = tiles[move.To];
		busyDuration = tileSwapper.Swap(a, b, !success);
		if (success)
		{
			tiles[move.From] = b;
			tiles[move.To] = a;
		}
	}
```

Moving back to `TileSwapper`, we implement its functionality. Give it fields to store the tiles, their initial positions, a progress set to −1 by default, and whether it should pingpong. Set all these fields in `Swap`, set progress to zero, and return double the duration for a pingpong.

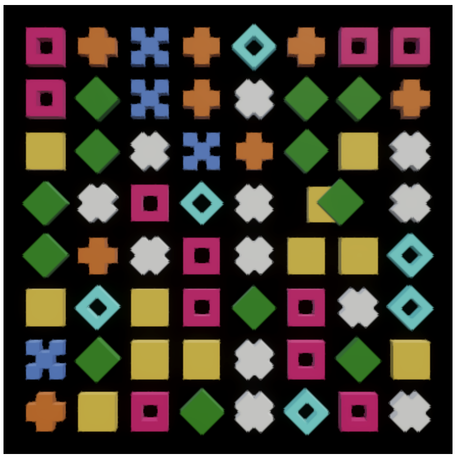
```
	Tile tileA, tileB;

	Vector3 positionA, positionB;

	float progress = -1f;

	bool pingPong;

	public float Swap (Tile a, Tile b, bool pingPong)
	{
		tileA = a;
		tileB = b;
		positionA = a.transform.localPosition;
		positionB = b.transform.localPosition;
		this.pingPong = pingPong;
		progress = 0f;
		return pingPong ? 2f * duration : duration;
	}
```

We use a progress of −1 to indicate that the swapper is inactive. So `Update` should begin checking that and return if inactive. Otherwise it continues and increases its progress. If it exceeded its duration it should do one of two things. In case of a pingpong it decreases progress by the duration, disables pingpong, and swaps the tiles. Otherwise it sets progress to −1, sets the tiles to their final positions, and returns.

```
	public void Update ()
	{
		if (progress < 0f)
		{
			return;
		}

		progress += Time.deltaTime;
		if (progress >= duration)
		{
			if (pingPong)
			{
				progress -= duration;
				pingPong = false;
				(tileA, tileB) = (tileB, tileA);
			}
			else
			{
				progress = -1f;
				tileA.transform.localPosition = positionB;
				tileB.transform.localPosition = positionA;
				return;
			}
		}
	}
```

After all that linearly interpolate both tile positions. We use the sine of π times the interpolator scaled by the max depth offset to displace the tiles in the Z dimension, negative for the first tile and positive for the second tile.

```
	public void Update ()
	{
		…

		float t = progress / duration;
		float z = Mathf.Sin(Mathf.PI * t) * maxDepthOffset;
		Vector3 p = Vector3.Lerp(positionA, positionB, t);
		p.z = -z;
		tileA.transform.localPosition = p;
		p = Vector3.Lerp(positionA, positionB, 1f - t);
		p.z = z;
		tileB.transform.localPosition = p;
	}
```


Tiles getting swapped.

Tile swaps are now animated and the game waits until the animation is finished before showing the results of the move. If it was a successful move the tiles change and the game progresses, otherwise the tiles bounce back. As the skin is busy during this time we cannot initiate a new swap while the current one is still in progress.

3.2 Disappearing Tiles

We're also going to add animations to disappearing tiles. To support different animations for each tile add a configurable disappear duration to `Tile`, set to 0.25 seconds by default. Let it keep track of its disappear progress, initialized to −1 when it spawns. We'll simple shrink the tile's scale to zero, so set it back to 1 when it spawns. Also disable the component so it won't needlessly update itself when it is not transitioning.

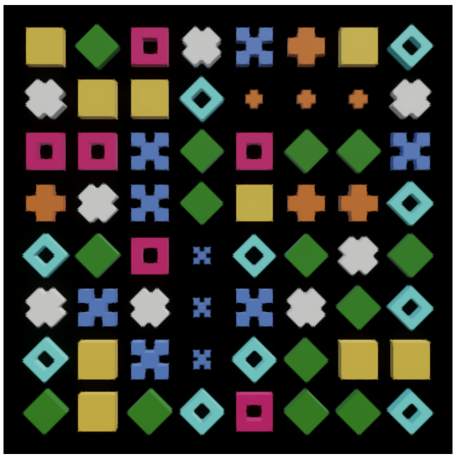
```
	[SerializeField, Range(0f, 1f)]
	float disappearDuration = 0.25f;

	PrefabInstancePool<Tile> pool;

	float disappearProgress;

	public Tile Spawn (Vector3 position)
	{
		Tile instance = pool.GetInstance(this);
		instance.pool = pool;
		instance.transform.localPosition = position;
		instance.transform.localScale = Vector3.one;
		instance.disappearProgress = -1f;
		instance.enabled = false;
		return instance;
	}
```

Give it a public `Disappear` method that sets its progress to zero, enables itself, and returns its duration. Add an **`Update`** method that, if it is progressing, decreases its scale to zero and despawns itself when it is done.

```
	public float Disappear ()
	{
		disappearProgress = 0f;
		enabled = true;
		return disappearDuration;
	}

	void Update ()
	{
		if (disappearProgress >= 0f)
		{
			disappearProgress += Time.deltaTime;
			if (disappearProgress >= disappearDuration)
			{
				Despawn();
				return;
			}
			transform.localScale =
				Vector3.one * (1f - disappearProgress / disappearDuration);
		}
	}
```

All that **`Match3Skin`** has to do to support disappearing tiles is invoke `Disappear` instead of `Despawn` in `ProcessMatches`. To wait until all animations are finished set the busy duration to the maximum of each tile's disappear duration and itself.

```
	void ProcessMatches ()
	{
		game.ProcessMatches();

		for (int i = 0; i < game.ClearedTileCoordinates.Count; i++)
		{
			int2 c = game.ClearedTileCoordinates[i];
			busyDuration = Mathf.Max(tiles[c].Disappear(), busyDuration);
			tiles[c] = null;
		}
	}
```


Disappearing tiles.

3.3 Falling Tiles

The third and final type of transitions that we'll animate are those of falling tiles. We'll let each tile take care of its own falling animation. It needs to keep track of a from and to Y position, the falling duration, and its progress. We could include `falling` in these field names, but let's group them in a inner `FallingState` struct instead. Once again we use a progress of −1 to indicate that the transition is inactive.

```
	[System.Serializable]
	struct FallingState
	{
		public float fromY, toY, duration, progress;
	}

	FallingState falling;

	public Tile Spawn (Vector3 position)
	{
		…
		instance.disappearProgress = -1f;
		instance.falling.progress = -1f;
		instance.enabled = false;
		return instance;
	}
```

Add a public `Fall` method with a destination Y position and speed as parameters and use those to set the falling state. Make it return the duration of the fall.

```
	public float Fall (float toY, float speed)
	{
		falling.fromY = transform.localPosition.y;
		falling.toY = toY;
		falling.duration = (falling.fromY - toY) / speed;
		falling.progress = 0f;
		enabled = true;
		return falling.duration;
	}
```

Now when updating the tile also has to check whether it is falling. If so perform a linear interpolation and stop when it is done.

```
	void Update ()
	{
		…

		if (falling.progress >= 0f)
		{
			Vector3 position = transform.localPosition;
			falling.progress += Time.deltaTime;
			if (falling.progress >= falling.duration)
			{
				falling.progress = -1f;
				position.y = falling.toY;
			}
			else
			{
				position.y = Mathf.Lerp(
					falling.fromY, falling.toY, falling.progress / falling.duration
				);
			}
			transform.localPosition = position;
		}
	}
```

In this case we can disable the component when done falling, but only if it isn't already disappearing at that moment.

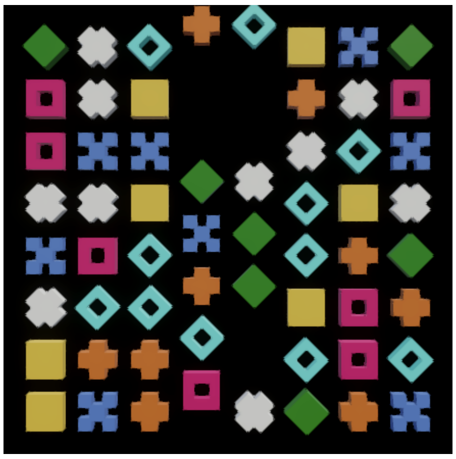
```
				falling.progress = -1f;
				position.y = falling.toY;
				enabled = disappearProgress >= 0f;
```

All tiles will drop with the same speed, so add a configuration field for it to `Match3Skin`, set to 8 by default. Also give it a configuration field for an extra offset to be added to dropped tiles that were newly created, set to 2 by default. If it is set to zero new tiles will spawn directly above the grid. Increasing it makes those tiles spawn higher, so they can spawn out of view and then drop into view from above.

```
	[SerializeField, Range(0.1f, 20f)]
	float dropSpeed = 8f;

	[SerializeField, Range(0f, 10f)]
	float newDropOffset = 2f;
```

In `DropTiles`, no longer adjust the position of tiles that were already in the grid. Add the vertical drop offset to those tiles that fall in from above the grid. Then make the tiles fall and set the busy duration to the longest fall duration.

```
			if (drop.fromY < tiles.SizeY)
			{
				tile = tiles[drop.coordinates.x, drop.fromY];
				~~//tile.transform.localPosition = new Vector3(~~
				~~//	drop.coordinates.x + tileOffset.x, drop.coordinates.y + tileOffset.y~~
				~~//);~~
			}
			else
			{
				tile = SpawnTile(
					game[drop.coordinates], drop.coordinates.x, drop.fromY + newDropOffset
				);
			}
			tiles[drop.coordinates] = tile;
			busyDuration = Mathf.Max(
				tile.Fall(drop.coordinates.y + tileOffset.y, dropSpeed), busyDuration
			);
```


Dropping tiles.

4 Scoring

Now that we can see what is going on, let's add scoring to the game.

4.1 Total Score

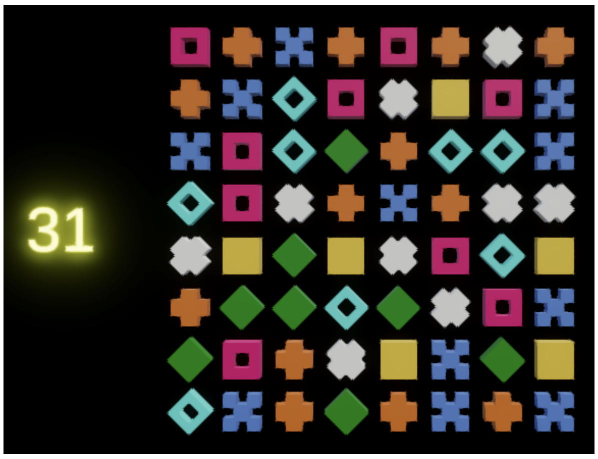
Add a public property for the total score to `Match3Game`, which is privately set. Set it to zero at the start of a new game and increase it by the match length for each match in `ProcessMatches`.

```
	public int TotalScore
	{ get; private set; }

	public bool HasMatches => matches.Count > 0;

	public void StartNewGame ()
	{
		TotalScore = 0;
		…
	}

	…

	public void ProcessMatches ()
	{
		ClearedTileCoordinates.Clear();

		for (int m = 0; m < matches.Count; m++)
		{
			…
			TotalScore += match.length;
		}

		…
	}
```

Add a configuration field for a total score text display to `Match3Skin`, set its text to 0 at the start of a game, and update it after processing matches.

```
	[SerializeField]
	TextMeshPro totalScoreText;

	…

	public void StartNewGame () {
		busyDuration = 0f;
		totalScoreText.SetText("0");
		…
	}

	…

	void ProcessMatches ()
	{
		…

		totalScoreText.SetText("{0}", game.TotalScore);
	}
```

Create a text game object and use it for the total score text. Position it to the left of the game, at −6. Give it a width of 3 and a height of 2. Enable its *Auto Size* option with its min set to 6 and its max set to 12. That way the text shrinks to fit its designated area in case a very high score is achieved.



Total score.

4.2 Floating Scores

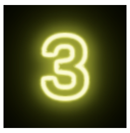
We'll also add scores for individual matches that will temporarily float in front of the grid. To communicate these scores from game to skin introduce a serializable `SingleScore` struct with public fields for its position and its value. The position is a `float2` because it can end up in between two tiles in case of a match length of 4.

```
using Unity.Mathematics;

[System.Serializable]
public struct SingleScore
{
	public float2 position;

	public int value;
}
```

Add a property for a list of scores to `Match3Game`, like the other publicly accessible lists. Clear it at the start of `ProcessMatches` and add a score to it in the middle of each match.

```
	public List<SingleScore> Scores
	{ get; private set; }

	…

	public void StartNewGame ()
	{
		TotalScore = 0;
		if (grid.IsUndefined)
		{
			…
			Scores = new();
		}
		FillGrid();
	}

	…

	public void ProcessMatches ()
	{
		ClearedTileCoordinates.Clear();
		Scores.Clear();

		for (int m = 0; m < matches.Count; m++)
		{
			…

			var score = new SingleScore
			{
				position = match.coordinates + (float2)step * (match.length - 1) * 0.5f,
				value = match.length
			};
			Scores.Add(score);
			TotalScore += score.value;
		}

		…
	}
```

Create a prefab from an empty game object with a text child. Give the child a Z offset of −0.25 so it will float in front of the tiles. Set its width and height to 1 and its font size of 8. To make it easier to see I gave it an adjusted material with a black color and a bright yellow outline instead of the default material that we've been using for all text.



Floating score.

Create a `FloatingScore` component type and assign it to the root of the prefab. Give it a configuration field for its text and hook it up. Also give it a public `Show` method with a position and value as parameters and use those to display and instance from a pool.

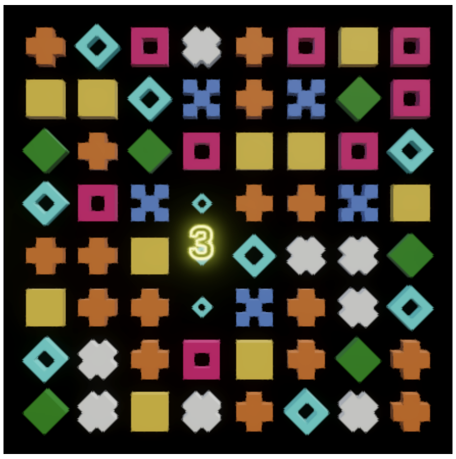
```
using TMPro;
using UnityEngine;

public class FloatingScore : MonoBehaviour
{
    [SerializeField]
    TextMeshPro displayText;

    PrefabInstancePool<FloatingScore> pool;

    public void Show (Vector3 position, int value)
    {
        FloatingScore instance = pool.GetInstance(this);
        instance.pool = pool;
        instance.displayText.SetText("{0}", value);
        instance.transform.localPosition = position;
    }
}
```

We make the text disappear automatically and also rise upward while it is visible. Add configuration fields for the duration and rise speed, set to 0.5 and 2 by default. Give it an age set to zero when it is shown. Have an `Update` method increase its age, recycle itself when its time is up, and move itself upward otherwise.

```
    [SerializeField, Range(0.1f, 1f)]
    float displayDuration = 0.5f;

    [SerializeField, Range(0f, 4f)]
    float riseSpeed = 2f;

    float age;

    PrefabInstancePool<FloatingScore> pool;

    public void Show (int score, Vector3 position)
    {
        …
        instance.age = 0f;
    }

    void Update ()
    {
        age += Time.deltaTime;
        if (age >= displayDuration)
        {
            pool.Recycle(this);
        }
        else
        {
            Vector3 p = transform.localPosition;
            p.y += riseSpeed * Time.deltaTime;
            transform.localPosition = p;
        }
    }
```

Add a configuration field for the floating score prefab to `Match3Skin` and hook it up. Then show all scores at the end of `ProcessMatches`.

```
	[SerializeField]
	FloatingScore floatingScorePrefab;

	…

	void ProcessMatches ()
	{
		…

		for (int i = 0; i < game.Scores.Count; i++)
		{
			SingleScore score = game.Scores[i];
			floatingScorePrefab.Show(
				new Vector3(
					score.position.x + tileOffset.x,
					score.position.y + tileOffset.y
				),
				score.value
			);
		}
	}
```



Single floating score.

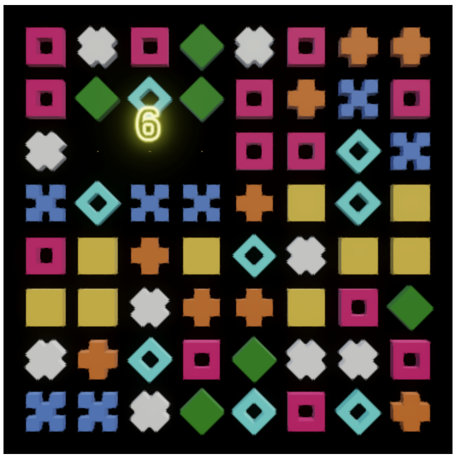
It is possible for floating scores to overlap, because we don't wait for them to disappear, and also because horizontal and vertical matches can overlap. To avoid Z-fighting of overlapping scores add a tiny depth offset to successive scores, pulling each closer by 0.001 units. Reset this offset back to zero once it has passed −0.02.

```
float floatingScoreZ;

…

void ProcessMatches ()
{
    …

    for (int i = 0; i < game.Scores.Count; i++)
    {
        SingleScore score = game.Scores[i];
        floatingScorePrefab.Show(
            new Vector3(
                score.position.x + tileOffset.x,
                score.position.y + tileOffset.y,
                floatingScoreZ
            ),
            score.value
        );
        floatingScoreZ = floatingScoreZ <= -0.02f ? 0f : floatingScoreZ - 0.001f;
    }
}
```

4.3 Score Multiplier

Match 3 games usually reward combos, cascades, or simultaneous matches. Let's do this as well, by adding a score multiplier to `Match3Game`. Set it to 1 each time a move is tried, multiply each single score value with it, and then increment it. Thus the more matches you get from a single move—no matter how—the greater the reward.

```
    int scoreMultiplier;

    …

    public bool TryMove (Move move)
    {
        scoreMultiplier = 1;
        …
    }

    public void ProcessMatches ()
    {
        …

        for (int m = 0; m < matches.Count; m++)
        {
            …

            var score = new SingleScore
            {
                position = match.coordinates + (float2)step * (match.length - 1) * 0.5f,
                value = match.length * scoreMultiplier++
            };
            …
        }

        …
    }
```



Second match after a move scores double points.

5 Finding Moves

The point of the game is to make matches, so it should end when it is no longer possible to do so. To detect this game-over state we have to search the grid for a possible move.

5.1 Searching for a Move

Let's add a public static `FindMove` method to `Move` that returns a move given a game. It runs though all rows of the game, looking for a valid move for the current tile. If it doesn't find such a move it returns the default move instead, which is invalid. We begin with the double loop that only retrieves the current tile. We use an `int2` coordinates variable for the loop, and store the game size in a local `s` variable to keep the code short.

```
	public static Move FindMove (Match3Game game)
	{
		int2 s = game.Size;
		for (int2 c = 0; c.y < s.y; c.y++)
		{
			for (c.x = 0; c.x < s.x; c.x++)
			{
				TileState t = game[c];
			}
		}

		return default;
	}
```

There are various ways to look for a potential match, but we'll take the current tile as the one to move. The first case that we consider is when a match can be made on the same row by moving the tile left. In schematic form, where `X` is the tile and `?` is a potential match to check:

```
??_X
```

This match is possible if the tiles two and three steps to the left both exist and match the tile. If so return the appropriate move.

```
				TileState t = game[c];

				if (c.x >= 3 && game[c.x - 2, c.y] == t && game[c.x - 3, c.y] == t)
				{
					return new Move(c, MoveDirection.Left);
				}
```

If this match isn't possible we can try the same in the opposite direction as well:

```
??_X_??
```

```
if (c.x >= 3 && game[c.x - 2, c.y] == t && game[c.x - 3, c.y] == t)
{
    return new Move(c, MoveDirection.Left);
}

if (c.x + 3 < s.x && game[c.x + 2, c.y] == t && game[c.x + 3, c.y] == t)
{
    return new Move(c, MoveDirection.Right);
}
```

After that we can perform the same checks vertically:

```
      ?
      ?

   ?? X ??

      ?
      ?
```

```
if (c.y >= 3 && game[c.x, c.y - 2] == t && game[c.x, c.y - 3] == t)
{
    return new Move(c, MoveDirection.Down);
}

if (c.y + 3 < s.y && game[c.x, c.y + 2] == t && game[c.x, c.y + 3] == t)
{
    return new Move(c, MoveDirection.Up);
}
```

If we're still going we have to start looking diagonally. Let's first consider all cases involving the tile one step down and left. We begin with the case where that tile and the one to the left of it will form a match after moving down:

```
  X
??
```

First check if we can go down one step. Then check if the down-left tile exists and matches. If so check if the tile to the left of that one matches as well, and if so return the move.

```
if (c.y > 1)
{
    if (c.x > 1 && game[c.x - 1, c.y - 1] == t)
    {
        if (c.x >= 2 && game[c.x - 2, c.y - 1] == t)
        {
            return new Move(c, MoveDirection.Down);
        }
    }
}
```

If that fails a move down could still work with the down-left tile if the down-right tile also matches:

```
 X
?? ?
```

```
if (
    c.x >= 2 && game[c.x - 2, c.y - 1] == t ||
    c.x + 1 < s.x && game[c.x + 1, c.y - 1] == t
)
{
    return new Move(c, MoveDirection.Down);
}
```

If moving down doesn't work we might still be able to match with the down-left tile by moving to the left instead:

```
 ?
  X
?? ?
 ?
```

These checks are the same, but with X and Y swapped.

```
if (c.y > 1)
{
    if (c.x > 1 && game[c.x - 1, c.y - 1] == t)
    {
        if (
            c.x >= 2 && game[c.x - 2, c.y - 1] == t ||
            c.x + 1 < s.x && game[c.x + 1, c.y - 1] == t
        )
        {
            return new Move(c, MoveDirection.Down);
        }
        if (
            c.y >= 2 && game[c.x - 1, c.y - 2] == t ||
            c.y + 1 < s.y && game[c.x - 1, c.y + 1] == t
        )
        {
            return new Move(c, MoveDirection.Left);
        }
    }
}
```

At this point it is clear that there is no match possible with the down-left tile. So let's check the down-right tile, with `o` indicating old checks that we don't have to consider again:

```
 o ?
  X
oo ??
 o ?
```

This requires the same code as for the down-left tile, but now to the right instead of left. Also note that we can skip one case.

```
if (c.y > 1)
{
    …

    if (c.x + 1 < s.x && game[c.x + 1, c.y - 1] == t)
    {
        if (c.x + 2 < s.x && game[c.x + 2, c.y - 1] == t)
        {
            return new Move(c, MoveDirection.Down);
        }
        if (
            c.y >= 2 && game[c.x + 1, c.y - 2] == t ||
            c.y + 1 < s.y && game[c.x + 1, c.y + 1] == t
        )
        {
            return new Move(c, MoveDirection.Right);
        }
    }
}
```

If there still isn't a match we have to perform the same checks again but for the two diagonal tiles up instead of down:

```
 ? ?
?? ??
  X
OO OO
 O O
```

We can do this by duplicating the entire `if (c.y > 1) { … }` code block, flip the vertical direction, and remove two cases. I only marked the changes in the duplicated code.

```
            if (c.y + 1 < s.y)
            {
                if (c.x > 1 && game[c.x - 1, c.y + 1] == t)
                {
                    if (
                        c.x >= 2 && game[c.x - 2, c.y + 1] == t ||
                        c.x + 1 < s.x && game[c.x + 1, c.y + 1] == t
                    )
                    {
                        return new Move(c, MoveDirection.Up);
                    }
                    if (c.y + 2 < s.y && game[c.x - 1, c.y + 2] == t)
                    {
                        return new Move(c, MoveDirection.Left);
                    }
                }

                if (c.x + 1 < s.x && game[c.x + 1, c.y + 1] == t)
                {
                    if (c.x + 2 < s.x && game[c.x + 2, c.y + 1] == t)
                    {
                        return new Move(c, MoveDirection.Up);
                    }
                    if (c.y + 2 < s.y && game[c.x + 1, c.y + 2] == t)
                    {
                        return new Move(c, MoveDirection.Right);
                    }
                }
            }
```

5.2 No More Moves

Now that we can find a move, add a publicly accessible property for a possible move to `Match3Game` that is privately set. Find a new possible move after filling the grid when starting a new game and if no matches were found after dropping tiles.

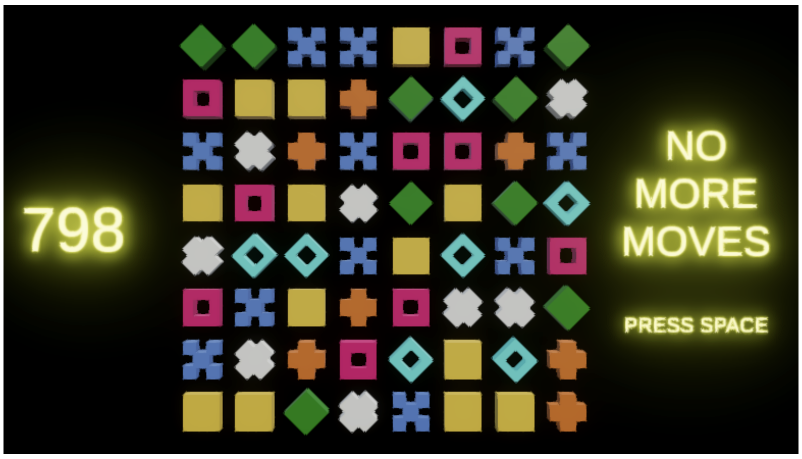
```
    public Move PossibleMove
    { get; private set; }

    …

    public void StartNewGame ()
    {
        …
        FillGrid();
        PossibleMove = Move.FindMove(this);
    }

    …

    public void DropTiles ()
    {
        …

        NeedsFilling = false;
        if (!FindMatches())
        {
            PossibleMove = Move.FindMove(this);
        }
    }
```

This makes it possible for `Match3Skin` to detect the game-over state. Give it a configurable game-over text that is deactivated when a new game is started. Change the `IsPlaying` property so it returns whether we're busy or there is a valid possible move. Then at the end of `DoWork`, when there are neither matches, nor a need for filling, and we're not playing, activate the game-over text.

```
	[SerializeField]
	TextMeshPro gameOverText, totalScoreText;

	…

	public bool IsPlaying => IsBusy || game.PossibleMove.IsValid;

	public void StartNewGame () {
		busyDuration = 0f;
		totalScoreText.SetText("0");
		gameOverText.gameObject.SetActive(false);

		…
	}

	…

	public void DoWork () {
		…

		if (game.HasMatches)
		{
			ProcessMatches();
		}
		else if (game.NeedsFilling)
		{
			DropTiles();
		}
		else if (!IsPlaying)
		{
			gameOverText.gameObject.SetActive(true);
		}
	}
```

Create a game-over text game object and set its X position to 6. Give it width 3, height 5, and font size 8. Use it to indicate that there are no more moves, and also a smaller hint that a new game can be started by pressing space:

```
NO MORE MOVES

<size=50%>PRESS SPACE
```



Game over.

The game will now automatically stop when there are no more valid moves. Although unlikely, it is possible that this is already the case at the very start of the game. To Avoid this make `Match3Game` keep filling the grid in `StartNewGame` until there is a valid possible move.

```
		do
		{
			FillGrid();
			PossibleMove = Move.FindMove(this);
		}
		while (!PossibleMove.IsValid);
```

5.3 Automatic Play

We end this tutorial by adding the ability for the game to play itself. This is done by simply adding a public `DoAutomaticMove` method to `Match3Skin` that uses the possible move.

```
public void DoAutomaticMove () => DoMove(game.PossibleMove);
```

Add a configuration option to `Game` to toggle automatic play. If enabled have `HandleInput` do an automatic move instead of checking for player input.

```
[SerializeField]
bool automaticPlay;

…

void HandleInput ()
{
	if (automaticPlay)
	{
		match3.DoAutomaticMove();
	}
	else if (!isDragging && Input.GetMouseButtonDown(0))
	{
		dragStart = Input.mousePosition;
		isDragging = true;
	}
	…
}
```

The game can now play itself. Its artificial intelligence is very basic as it simply performs the first move that was found. It plays as fast as possible, issuing the move as soon as the skin is not busy.

Our match-3 game prototype is now finished. It could be improved by adding more visuals and animations, by adding special tiles with their own behavior, by introducing entirely new game mechanics, by improving the AI, by making it multiplayer, or in other ways. Such changes would be the basis for a project based on this tutorial.

The next tutorial is Bouncy Ball Shooter.