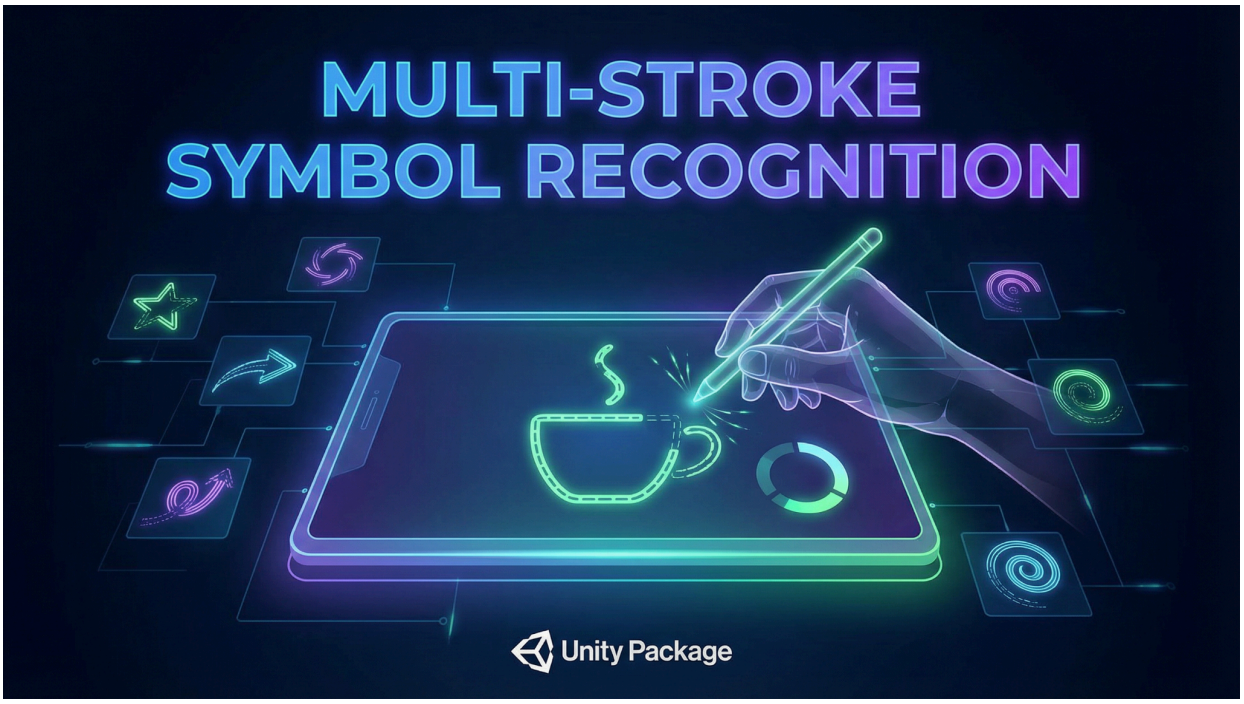
Multi-Stroke Symbol Recognition

Documentation



1. Overview

This package provides a robust system for 2D symbol recognition within Unity. It is designed to identify user-drawn symbols (from mouse/touch strokes) or match template textures.

The system is built on a two-part feature extraction process:

1. **Zernike Moments:** Used to create a rotation-invariant description of the symbol's **shape**.
2. **Angular Histogram:** Used to create a descriptor of the symbol's pixel distribution, allowing for **orientation-sensitive** matching.

This dual approach allows the system to distinguish between symbols that have similar shapes but different orientations (e.g., "6" vs. "9") or to ignore orientation when it doesn't matter (e.g., a "fireball" symbol).

2. Core Concepts

2.1. The Recognition Pipeline: Two Distinct Phases

The recognition system operates in two critical phases to ensure efficiency, precision, and flexibility. This approach clearly separates the one-time, computationally expensive **pre-computation** from the real-time **recognition** process.

A. Pre-computation Phase (Editor Time)

This phase runs once in the Unity Editor (or a dedicated tool) to build the symbol library. Its purpose is to **generate the immutable numerical data** (Zernike Moments and Angular Histograms) from image templates, avoiding this costly calculation at runtime.

Step	Input/Process	Description
1. Input	**Texture2D Template**	The developer provides a Texture2D (an image) as the perfect example (*template*) for each symbol.
2. Processing	**Image Analysis**	The system rasterizes and normalizes the *template* image, ensuring uniform stroke thickness.
3. Extraction	**Descriptor Generation**	**Zernike Moments** (magnitudes for pure **Shape**) and the **Angular Histogram** (for **Orientation**) are calculated.
4. Output	**JSON Serialization**	The generated numerical vectors (momentMagnitudes and distribution), along with metadata (thresholds, Stroke Count), are saved in the ReferenceSymbolGroup object and **serialized to a JSON file** for runtime loading.

B. Recognition Phase (Runtime)

This phase occurs in the game when the user draws. It only requires the pre-calculated data from the JSON file to classify the user's drawing.

Step	Input/Process	Description
1. Load Library	**JSON Deserialization**	On startup, the **ZernikeManager** loads and deserializes the complete symbol library from the **JSON file** into memory.
2. Stroke Acquisition	**List<List<Vector 2>>**	The system receives the user's drawing points and the **Stroke Count**, which is used as a crucial initial filter.
3. Normalizatio n & Rasterization	**Shape Preparation**	Vector strokes are converted into a normalized **Binary Image**. The drawing is **centered and scaled** to eliminate position and size variations.
4. Uniform Thickness	**Standardization**	The binary image undergoes **thinning** (skeletonization) and subsequent **re-thickening** to a uniform TargetThickness. This neutralizes stroke thickness inconsistencies from the user.
5. Feature Extraction	**Descriptor Generation**	The system calculates the **Zernike Moments** and the **Angular Histogram** from the user's drawing, using the exact same process as in Phase A.

6. Matching & Classification	**Distance Calculation**	The recognizer filters the library by **Stroke Count**. It then calculates the **Zernike Distance** (for shape match) and, optionally, the **Histogram Difference** (for orientation match). The symbol with the lowest total distance below the defined thresholds is declared the winner.

2.2. Feature Descriptors

Zernike Moments (Shape)

Zernike Moments are a set of orthogonal polynomials used to characterize 2D shapes. The key feature leveraged is that the **magnitudes** of these moments are mathematically **rotation-invariant**. This ensures that the system recognizes a symbol based purely on its shape, regardless of its orientation when drawn.

Angular Histogram (Orientation)

This custom descriptor models the spatial distribution of the symbol's pixels relative to its geometric center. It divides the image into a polar grid (e.g., 16 **sectors** and 3 **rings**), counting the pixel mass in each bin.

Function: The difference between the user's and the template's Angular Histograms allows the system to distinguish between symbols that are geometrically identical (e.g., a perfect circle) but have specific orientation requirements (e.g., the number "6" vs. the number "9").

3. Key Classes & API

The package is structured into several interconnected C# files, each responsible for a specific stage of the recognition pipeline.

- `Drawer.cs`
    - **The input controller.** This monobehaviour script manages the drawing inputs from the player and sends the result to the Zernike manager.

- **`ZernikeManager.cs`**
    - **The Controller.** This main Unity component (`MonoBehaviour`) manages global settings, handles JSON loading/saving, and orchestrates the entire runtime recognition flow, from receiving user strokes to displaying the final result.
- **`ZernikeProcessor.cs`**
    - **The Pipeline Runner.** This class acts as the intermediary between the Manager and the low-level mathematical/image processing tools. It manages the internal state of the current drawing and sequentially calls the *Rasterizer*, *Normalizer*, and *Zernike Calculator*.
- **`ZernikeRecognizer.cs`**
    - **The Matcher/Classifier.** Contains the core logic for comparing the player's calculated features (Zernike Moments and Angular Histogram) against the loaded symbol library. It calculates the final distance, applies thresholds, and determines the best matching symbol.
- **`SymbolRasterizer.cs`**
    - **The Drawing Tool.** This static class is responsible for converting raw input (either multi-stroke vectors or `Texture2D` pixels) into the initial normalized, centered, and scaled square **Binary Image** matrix (`float[,]`) used by the pipeline.
- **`ImageRasterizer.cs`**
    - **The Normalizer & Feature Prep.** This static utility is crucial for standardizing the image. It implements the **Zhang-Suen thinning algorithm** (skeletonization), re-thickens the lines to a uniform size, and performs the calculation of the **Angular Histogram** feature descriptor.
- **`ZernikeMomentCalculator.cs`**
    - **The Feature Extractor.** This static class contains the complex mathematical implementation required to compute the **Zernike Moments** from the processed binary image. It also handles the necessary coordinate transformation (centroid and normalization) for the moment calculation.
- **`BinaryImage.cs`**
    - **The Image Data Structure.** A simple utility class that wraps the core `float[,]` image matrix. It provides safe accessors and basic operations like clearing the matrix or counting active pixels.
- **`ReferenceSymbol.cs`** & **`ZernikeMoment.cs`**
    - **The Data Containers.** These files define the serializable data structures used to store the pre-calculated features: `ReferenceSymbol` holds the moments and distribution arrays, and `ZernikeMoment` holds the magnitude and phase of a single moment.

4. Data Structures

These serializable classes define the reference library.

`ReferenceSymbolGroup`

Represents a *single logical symbol* (e.g., "Fireball") which may have multiple drawing examples.

- `symbolName`: The human-readable name of the symbol (e.g., "Shield").
- `symbols`: A `List<ReferenceSymbol>` containing one or more pre-computed templates for this symbol.
- `strokes`: **Crucial.** The exact number of strokes required for this symbol. The recognizer uses this as a hard filter.
- `Threshold`: The maximum allowed **Zernike distance**. A higher value is more lenient (0.5 is a good default).
- `useRotation`: If `true`, the `orientationThreshold` and angular histogram will be used to check orientation.
- `orientationThreshold`: The maximum allowed **Histogram difference**.
- `isSymmetric`: A special flag. If `true`, the histogram difference is added to the Zernike distance to create the final distance score.

`ReferenceSymbol`

Represents a *single pre-computed template* (one specific drawing).

- `symbolName`: A descriptive name for this specific template.
- `momentMagnitudes`: `List<double>` of the pre-computed Zernike moment magnitudes.
- `distribution`: `float[]` of the pre-computed angular histogram.
- `symbolID`: A unique ID.

5. Code examples

Code Example 1 — Recognizing a Symbol

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

public class SymbolRecognitionExample : MonoBehaviour
{
    public ZernikeManager zernikeManager;

    void Start()
    {
        // Create some sample stroke data
        List<List<Vector2>> drawnStrokes = new List<List<Vector2>>();
        drawnStrokes.Add(new List<Vector2>()
        {
            new Vector2(0.1f, 0.1f),
            new Vector2(0.2f, 0.2f),
            new Vector2(0.3f, 0.3f)
        });

        int strokeCount = 1;

        // Send the strokes to the Zernike Manager to perform recognition
        zernikeManager.OnDrawingFinished(drawnStrokes, strokeCount);
    }
}
```

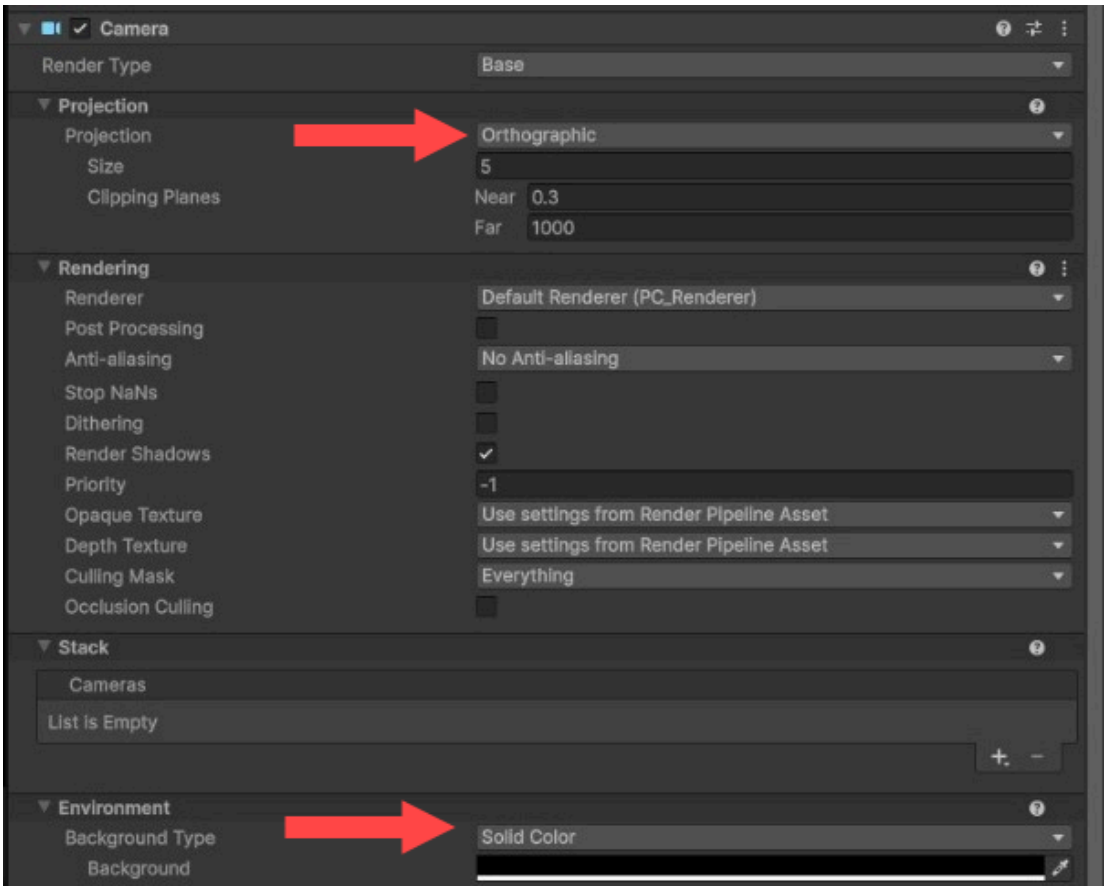
Code Example 2 — Registering a Callback When a Symbol Is Recognized

```
using UnityEngine;

public class SymbolEvents : MonoBehaviour
{
    public ZernikeManager zernikeManager;

    void Start()
    {
        // Subscribe to the recognition callback
        zernikeManager.recognitionAction += OnSymbolRecognized
    }

    private void OnSymbolRecognized(string symbolName)
    {
        Debug Log("Recognized symbol: " + symbolName);

        if (symbolName == "Fireball")
        {
            CastFireball();
        }
    }

    private void CastFireball()
    {
        Debug Log("🔥 Fireball launched!");
    }
}
```

6. How to Use

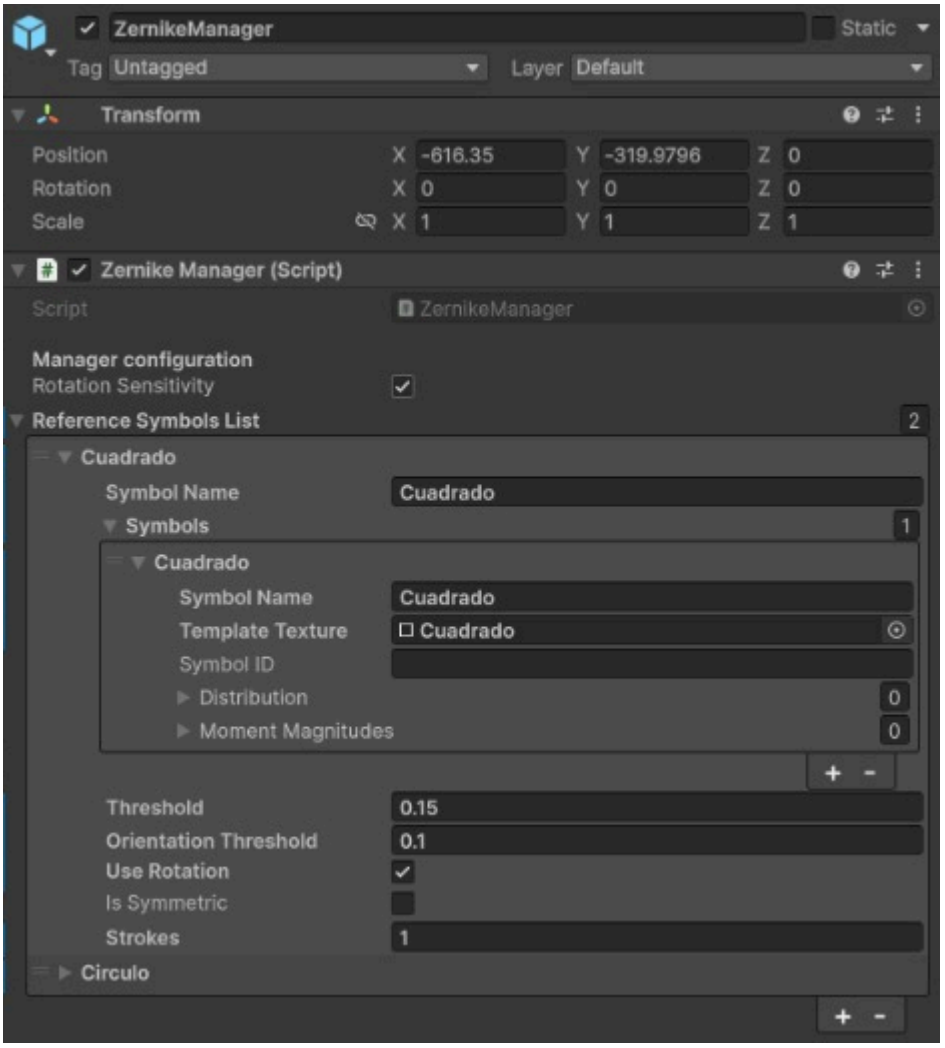
Scene setup:

1. In a new scene set the main camera projection to orthographics (for testing, a dark solid background is recommended)



2. Drag the “Zernike Objects” prefab into the scene.

3. Configure the Zernike manager

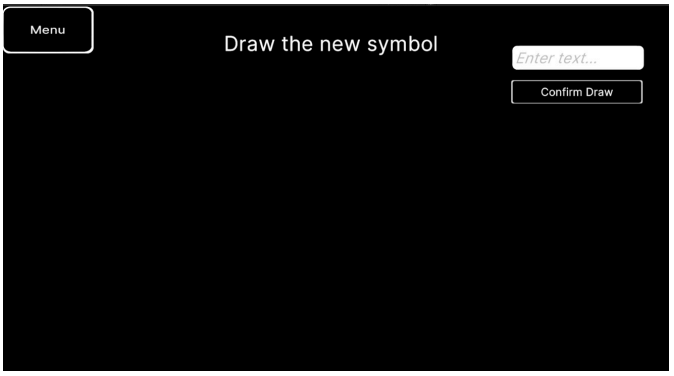


4. Once you hit the play button, the editor list will be saved in a JSON file inside the Resources folder.

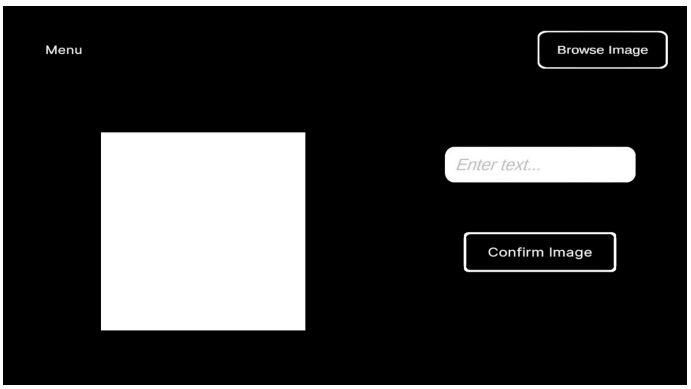
Configuration scenes

After adding the symbols for the first time, you will be able to configure the saved symbol list by adding new symbols or modifying their values in the following scenes:

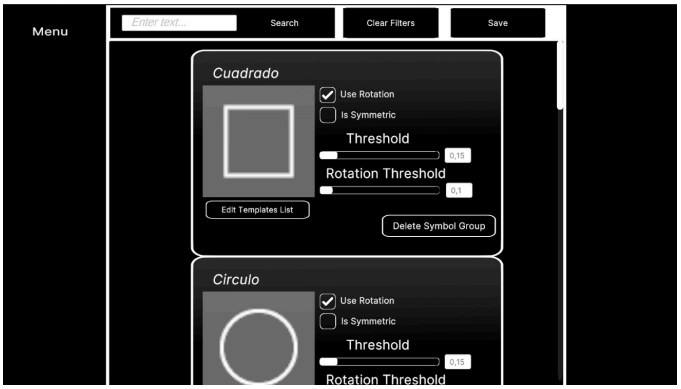
- **UploadDrawing:** in this scene you can add symbols from drawings.



- **UploadImage:** In this scene you can add images from your disk



- **SymbolConfiguration:** in this scene you can configure the symbols values, delete symbols or delete groups



7. Performance

This package was designed so that all the computationally expensive operations—image preprocessing, stroke normalization, line thinning, Zernike moment extraction, and angular histogram generation—are executed **only once during the editor pre-computation phase**.
The resulting descriptors are stored in a JSON library and reused at runtime, keeping gameplay performance extremely efficient.

Runtime Performance

During gameplay, the recognition pipeline consists of:

1. **Stroke rasterization**
2. **Normalization & thinning / re-thickening**
3. **Zernike Moments computation** (rotation-invariant shape descriptor)

4. **Angular Histogram computation** (orientation descriptor)
5. **Template matching** against the symbol library

This process is triggered **only when the player finishes drawing**, not every frame, which makes performance requirements very light.

Measured Results

On a 64×64 image resolution with a library of **15 templates**, the full recognition pipeline completes in **6–10 ms per recognition**

This includes *all* steps: rasterization, feature extraction, and matching.

These timings are well within real-time constraints for any Unity application—desktop or mobile—because recognition is performed only on demand and never during the per-frame update loop.

Scalability

- The runtime cost scales **linearly** with the number of templates in the library.
- Even with significantly larger libraries (40–60 symbols), expected recognition times remain below ~15 ms on most hardware.
- Increasing the image size (e.g., 128×128) will increase processing cost, but the default 64×64 size provides an excellent balance between accuracy and speed.

Editor-Time Preprocessing

When building or updating the symbol library, each template image is processed once:

- normalization
- skeletonization and re-thickening
- Zernike moment extraction
- histogram generation
- JSON serialization

Typical preprocessing cost per template: **5–30 ms**, depending on complexity.

Since this occurs only in editor tools and never during gameplay, it does not affect runtime performance.

Performance Summary

- **Runtime recognition:** 6–10 ms with 15 templates
- **Editor preprocessing:** 5–30 ms per template
- **Matching:** extremely lightweight (microseconds)
- **Memory footprint:** small (JSON + in-memory arrays)

Overall:

The system is efficient, mobile-friendly, and suitable for real-time games or drawing-based interaction without impacting frame rate.

8. Known Limitations

Although the recognition system is robust and flexible, there are a few inherent limitations that developers should keep in mind when designing symbol libraries and creating templates.

1. Very Similar Symbols May Require Additional Templates

The system relies on comparing the user's drawing against precomputed template descriptors (Zernike moments and angular histograms).

When two symbols have **very similar shapes**, a single template per symbol may not sufficiently capture all variations in how players draw them.

Examples:

- Slight curvature differences
- Very similar silhouettes
- Symbols that differ only by a small angle or orientation
- Subtle internal gaps or intersections

In these cases, adding **multiple templates** for the same symbol significantly improves accuracy by expanding the feature-space coverage of that symbol.

2. Drawn Templates Generally Match Drawn Input Better

Templates created from **actual drawings** tend to match much more reliably with user-drawn input than templates generated from clean textures or icons.
This is expected, because:

- Drawn templates have natural imperfections
- Stroke flow and curvature match real player input
- Stroke thickness variations are more representative
- Noise and small inconsistencies better model real usage

For best results, especially with complex symbols, it is recommended to include at least one **hand-drawn version** of each symbol in the library.

3. Highly Symmetric Symbols Can Cause Ambiguous Matches

Symbols with strong symmetry—circles, diamonds, crosses, radial shapes, etc.—may produce nearly identical Zernike descriptors across different orientations.
This can make orientation-based matching overly strict or produce false mismatches if the player's drawing is rotated slightly.

To address this, each symbol group includes an `isSymmetric` flag.

- When **isSymmetric = true**, the angular histogram no longer penalizes small orientation differences.
- This prevents the recognizer from rejecting valid drawings of symmetric symbols simply because they are rotated.

This option is specifically intended for radial or reflective symmetry, and should be enabled whenever rotational ambiguity is expected.

9. Third-Party Licenses & Credits

This asset includes third-party components distributed under the MIT License.
The following notices are reproduced here in compliance with their respective licensing terms.

Unity Standalone File Browser

MIT License
Copyright (c) 2018 **Markus Göbel (Bunny83)**

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Unity Windows File Drag-Drop

MIT License
Copyright (c) 2017 **Gökhan Gökçe**

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

10. Mathematical Algorithms & Academic References

This package relies on well-established algorithms from the fields of image processing and pattern recognition.
The following references are provided for completeness and academic attribution:

Zernike Moments (Shape Descriptor)

Used for rotation-invariant shape analysis and symbol comparison.

- **Teague, M. R. (1980).**
  Image analysis via the general theory of moments.
  Journal of the Optical Society of America, 70(8), 920–930.

- **Khotanzad, A., & Hong, Y. H. (1990).**
  Invariant image recognition by Zernike moments.
  IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(5), 489–497.

Zhang–Suen Thinning Algorithm (Skeletonization)

Used to normalize stroke thickness.

- **Zhang, T. Y., & Suen, C. Y. (1984).**
  A fast parallel algorithm for thinning digital patterns.
  Communications of the ACM, 27(3), 236–239.

Polar / Angular Histogram Descriptor (Orientation Analysis)

Used to distinguish symbols with identical shape but different orientation.

Although the implementation is custom, the technique is based on well-established polar-grid spatial distribution descriptors commonly used in shape recognition:

- **Belongie, S., Malik, J., & Puzicha, J. (2002).**
  Shape Matching and Object Recognition Using Shape Contexts.
  IEEE Transactions on Pattern Analysis and Machine Intelligence, 24(4), 509–522.

11. Conclusion

This package provides a complete and efficient solution for multi-stroke symbol recognition inside Unity.
By combining rotation-invariant Zernike Moments with a polar Angular Histogram, the system achieves both robustness and flexibility, enabling accurate classification even with imperfect user-drawn input.

The pipeline is carefully designed to separate heavy preprocessing from lightweight runtime execution, ensuring consistently fast recognition times suitable for real-time gameplay. Developers can easily extend the symbol library, customize thresholds, or integrate the recognizer into existing interaction systems.

With clear tooling for importing textures, capturing drawings, configuring templates, and inspecting results, this package offers a practical, production-ready framework for any project that relies on gesture-based or symbol-based input.

For support, questions, or feature requests, please feel free to contact the developer.

Contact e-mail: evilcaveinfo@gmail.com