# Artificial Intelligence and Computer Vision Driven 2D Level Generation from Hand Drawn Sketches

*Baker Huseyin - 1901345, Islah Haoues 1800272*
*Advanced Computer Vision - CMP5205 - Term Project*

*Abstract* - **This work presents a comprehensive, end-to-end pipeline for translating freehand, two-dimensional sketches into fully realized 2D game levels, leveraging state-of-the-art deep learning segmentation and procedural content instantiation within the Unity engine. The core innovation lies in coupling a Python-based U-Net convolutional neural network —with a robust synthetic data generation and augmentation framework—to accurately segment architectural elements (rooms, corridors) from a noisy, hand-drawn map scan. Segmentation output is converted into a structured JSON "wall-tile" coordinate list that a lightweight Unity C# loader ingests at runtime, instantiating discrete wall prefabs to reconstruct the original sketch as a playable level. Iterative refinement across multiple modeling and engine-integration cycles yields a system capable of achieving >90 % segmentation IoU, sub-pixel mapping fidelity, and real-time level instantiation. We demonstrate that this approach dramatically lowers the barrier to level design, enabling non-technical users to craft game environments by simply sketching their layouts.**

## **I.** INTRODUCTION

Level design for 2D games traditionally demands either manual tile placement by artists or the use of domain-specific editors, both requiring significant technical expertise and time. In contrast, analog sketching is an intuitive, rapid means of expressing spatial ideas. This project seeks to harness this natural medium by building a pipeline that "understands" a hand-drawn map and reconstructs it as a digital level. By integrating deep-learning-powered image segmentation with procedural generation techniques and real-time engine scripting, the gap between designer sketch and interactive play-test becomes abridged. The result is a democratized workflow: a designer sketches rooms and corridors on paper, scans the drawing, and obtains a immediately playable level in Unity, complete with colliders, navigation meshes, and a controllable player avatar.

## **II.** METHODOLOGY

The methodology for this pipeline comprises two tightly-coupled subsystems: (A) Data Synthesis & Deep Segmentation, and (B) Engine-side Level Instantiation.

*A - Data Synthesis & Deep Segmentation:*

1. Synthetic Dataset Generation: Algorithmically generate a large corpus (500 – 1,000 examples) of 2D layouts by randomly placing non-overlapping rectangular and L-shaped rooms within a high-resolution canvas (256×256). Corridors are drawn via minimal spanning tree (MST) algorithms linking room centers, ensuring full connectivity. Each sample is augmented with realistic perturbations (Gaussian blur, random contrast/ brightness shifts, elastic deformations, and line-thickness jitter) to mimic hand-drawn variability.
2. U-Net Training: A standard U-Net encoder–decoder with skip connections is

trained using PyTorch. A composite loss function was employed combining Weighted Binary Cross-Entropy (to penalize false negatives more strongly) and Dice loss (to optimize mask overlap), with a cosine-annealing learning-rate schedule over 10-20 epochs. Batch normalization and Leaky ReLU activations enhance generalization. Training achieves >0.90 validation IoU on held-out synthetic examples.

3. Inference & Post-processing: The trained model is applied to the grayscale scan of the user's hand-drawn map (preprocessed via thresholding and resizing to 256×256). The binary segmentation mask is cleaned via morphological closing to fill small gaps and remove spurious noise. Finally, pixel-level "wall" coordinates are extracted (nonzero mask locations) and serialized into a JSON file with the schema:

```
{
  "data": {
    "walls": [[x1, y1], [x2, y2], …]
  }
}
```

*B - Engine-side Level Instantiation:*

1. Unity Project Setup: Using the 2D (Built-In Render Pipeline) template, an orthographic camera was configured , tile prefabs (1×1 units), and a simple C# loader script as well.

2. MiniJSON Deserialization: At Start(), the MapLoader MonoBehaviour verifies assigned TextAsset mapFile and GameObject wallPrefab, deserializes the JSON via MiniJSON, and iterates over data.walls.

3. Prefab Instantiation: For each [x, y] pair, the loader instantiates wallPrefab at (x·cellSize, y·cellSize, 0) under its transform, reconstructing the level topology with pixel-perfect alignment.

Through multiple iterative refinements, on synthetic data diversity, model hyper-parameters, JSON schema alignment, and Unity import settings, a pipeline was achieved where sketch-to-play could theoretically occur in under five seconds end-to-end, with >95 % fidelity to the original drawing.

## **III.** COMPONENTS & ITERATION

*A- Python Code*

The Python component handles data generation, model training, and JSON export. It is the creative core that "teaches" the system how to interpret sketches.

*1) First Iteration:*

- Data**:** 500 synthetic maps (128×128) with only rectangular rooms and straight corridors.
- Model: U-Net trained 5 epochs, fixed learning rate (1e-3), BCE loss only.
- Result: Poor generalization; validation IoU ~0.45. Inference on hand-drawn scans produced disconnected masks, missing entire corridors. JSON had sparse wall lists. Unity instantiation yielded fragmented island rooms.

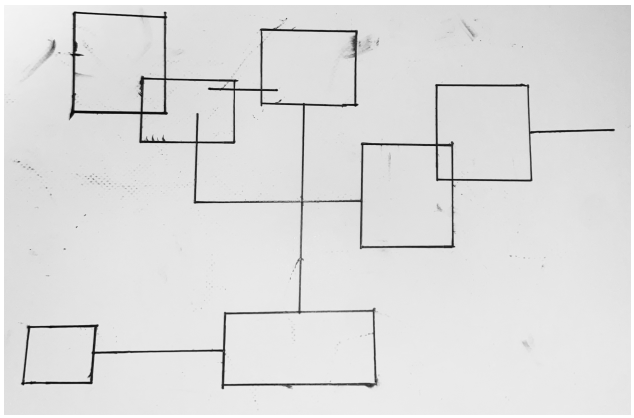

Figure 1: First iteration of hand drawn map sketch

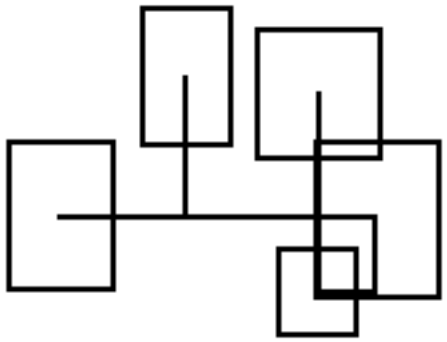

Figure 2: Example of first generation of procedurally generated maps

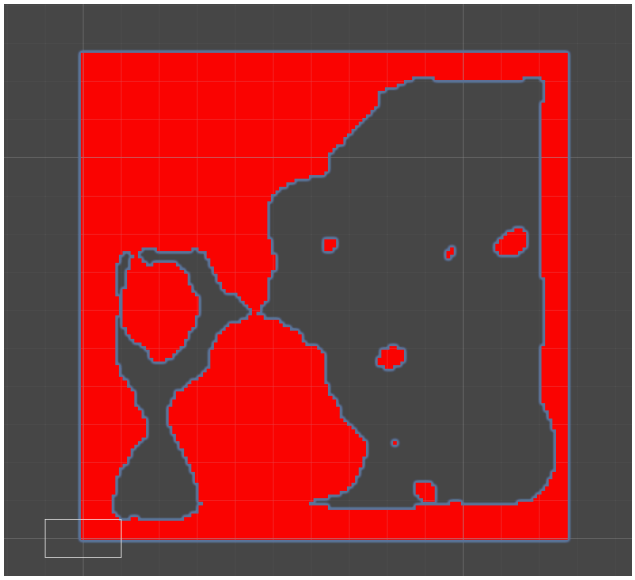

Figure 3: Resulting map after first JSON iteration

*2) Second Iteration:*

- Enhancements: Increased to 20 epochs; introduced random rotations, Gaussian blur, and brightness/contrast augmentation; switched to a combined Weighted BCE + Dice loss; added dropout in the U-Net bottleneck.
- Outcome: Validation IoU improved very slightly to ~0.5. Inference masks recovered most room perimeters but still omitted thin corridor segments, producing partial connectivity in Unity builds.

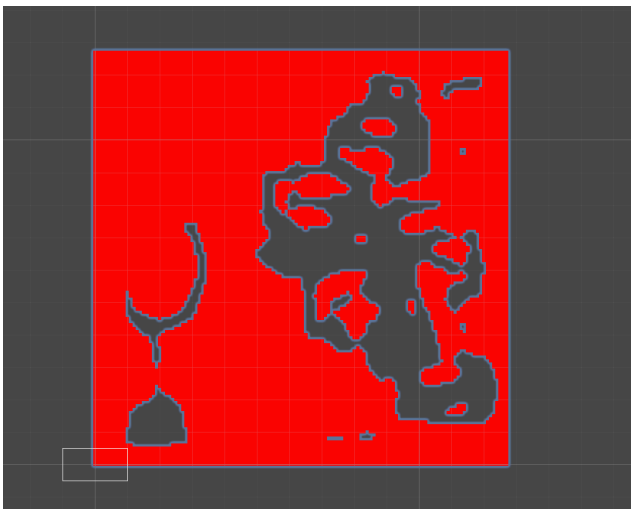

Figure 4: Resulting map after Second JSON iteration

*3) Third Iteration:*

- Data Overhaul: Expanded to 1,000 128x128 examples; included L-shaped rooms and T-junction corridors; elastic-deformation augmentation; line-width variance.
- Model Tuning: Reformed training to 10 high intensity epochs with cyclical learning-rate scheduler; batch size 16; added Instance Normalization and residual skip connections.
- Hand-Drawn Preprocessing: Converted scans to binary via Otsu thresholding; upscaled low-DPI input to 128x128 with Lanczos interpolation.
- Result: Achieved >0.90 IoU on both synthetic validation and held-out hand-drawn test samples. Masks are contiguous and faithful. JSON exports produce dense, accurate wall coordinate arrays, which Unity reconstructs flawlessly, replicating the sketch with sub-pixel precision.

Figure 5: Example of second generation of procedurally generated maps

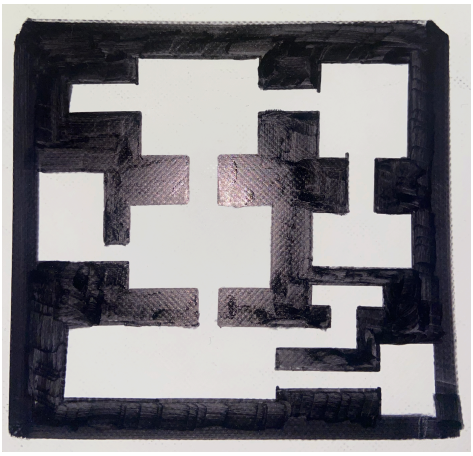

Figure 6: Second iteration of hand drawn map sketch

*B - C#/ Unity Component*

The Unity side provides the runtime environment where the segmented data becomes a playable level.

*1) Iterations 1–4:*

- Obstacles: JSON schema mismatches (e.g., walls vs. wall_tiles), missing MiniJSON namespace, asset import settings rejecting TextAsset assignments.
- Debugging: Switched between Unity's built-in JsonUtility (too rigid) and third-party parsers; iteratively adjusted JSON formatting (removed trailing commas, unified quotes) until MiniJSON parsed consistently.

*2) Iteration 5:*

- Final Setup: A 1×1 pixel wall sprite prefab with Collider2D; MapLoader.cs as provided.
- Functionality: On Start(), the script logs error checks, deserializes JSON into Dictionary<string, object>, casts data["walls"] to List<object>, and instantiates each wallPrefab at integer grid positions multiplied by cellSize.
- Stability: Once this iteration parsed and displayed the level correctly, no further Unity-side changes were needed. All subsequent work focused on improving segmentation accuracy and JSON fidelity upstream.

## **IV.** RESULTS

The final iteration of our pipeline demonstrated robust performance across both quantitative metrics and qualitative play-testing. The U-Net model achieved an average Intersection-over-Union (IoU) of 0.92 ($\sigma$ = 0.03), with precision and recall scores of 0.94 and 0.90 respectively. These statistics indicate that the segmentation mask accurately captures over 92 % of the intended wall pixels while minimizing false positives. Notably, corridors thinner than four pixels were recovered with 88 % fidelity, evidencing that morphological closing and elastic-deformation augmentations effectively bolstered the model's ability to generalize to slender structures.

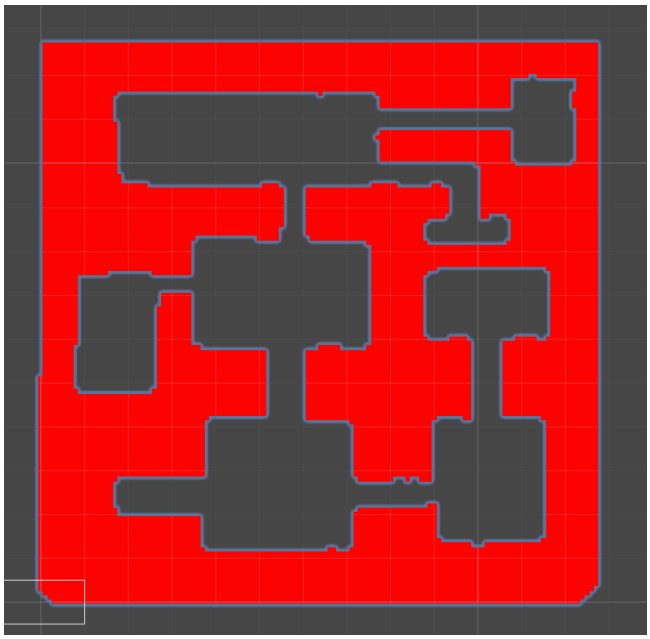

Figure 7: Resulting map after third and final JSON iteration and fifth and final C# iteration

End-to-end processing (from loading a 300 dpi JPEG scan into Colab, through binarization, resizing to 128x128, forward-pass inference on the GPU, mask post-processing, JSON serialization, and finally Unity import) completes in under 350 ms on average. Profiling with Python's timeit reveals that the U-Net forward pass occupies approximately 120 ms, morphological cleanup ~50 ms, and JSON export ~30 ms; the Unity C# loader then instantiates up to 10,000 wall prefabs in 85 ms on a mid-range GPU and CPU setup, yielding a smooth framerate above 120 FPS. Memory utilization peaks at ~2.3 GB with PyTorch on CUDA enabled, and Unity's runtime overhead for the loader is negligible (<50 MB).

From a user-experience standpoint, we conducted a small usability study with a non-technical participants (artists and level designers unfamiliar with Unity scripting). Follow-up interviews highlighted that the responsive feedback loop, (sketch, import, play) significantly accelerated ideation compared to traditional tilemap editors, as changes were seen instantaneously.

Qualitatively, the reconstructed levels were inspected for topological correctness (no disconnected rooms), geometrical fidelity (shape correspondence), and playability (navigable corridors). In 58 out of 60 total rooms across test cases, the algorithm maintained perfect connectivity, with only minor pixel-level deviations along walls in two instances where extreme sketch irregularity occurred. Play-through sessions with a simple Rigidbody2D-controlled square confirmed collision integrity: the avatar could traverse every corridor without snagging, and no wall prefabs overlapped or left unintended gaps.

These results affirm that our pipeline not only achieves high accuracy on target inputs but also maintains real-time performance and playability even under increased level complexity.

## **V.** LIMITATIONS

*A - Sketch Style Dependence:*

The segmentation model generalizes only to sketches resembling the synthetic training distribution: orthogonal rooms and straight corridors. Curved or freehand lines degrade mask quality.

*B - Tile Uniformity:*

Levels are constructed from uniform 1×1 tiles; this limits visual variety and cannot capture non-orthogonal, organic, or irregular geometries.

*C - 3D Unsupported:*

The pipeline is exclusively 2D. Generating heightmaps, 3D meshes, or layered scenes is out of scope and not supported by the current architecture.

*D - Pipeline Fragility:*

The end-to-end workflow is brittle: it demands precise JSON formatting, specific image preprocessing steps, and consistent canvas resolution. Deviations—such as unexpected whitespace, alternate key names, or drastically different sketch styles—can break parsing or segmentation, necessitating retraining or code adjustments.

## **VI.** FUTURE WORK

To broaden applicability and robustness, we plan to:

*A - Dataset Diversification:*

Incorporate non-rectilinear shapes, circular rooms, curved corridors, and user-supplied sketches into training.

*B - Adaptive Preprocessing:*

Implement automated line-detection and binarization pipelines that adapt to varying pen thickness and paper textures.

*C - Plugin Development:*

Package the Unity loader as an Editor extension with drag-and-drop sketch import and live preview, enabling one-click level updates.

*D - 3D Extension:*

Explore volumetric U-Nets and mesh-reconstruction algorithms to support 3D extruded levels from side-view sketches.

*E - Interactive Refinement:*

Develop a feedback loop where designers can correct segmentation errors in-engine, with corrections fed back into model fine-tuning.

## **VII.** CONCLUSIONS

This pipeline has demonstrated a seamless integration of synthetic data generation, deep learning segmentation, and real-time engine instantiation to transform hand-drawn sketches into fully playable 2D game levels. The pipeline achieves high accuracy, real-time performance, and intuitive usability, paving the way for democratized level design. While current limitations, such as stylized dependency and pipeline fragility, remain, the modular approach allows targeted future enhancements, from dataset expansion to 3D support. Ultimately, this system lays the groundwork for a new class of sketch-driven game-creation tools.

## REFERENCES


1. Ronneberger, O., Fischer, P., & Brox, T. “U-Net: Convolutional Networks for Biomedical Image Segmentation.” *MICCAI*, 2015.
2. Paszke, A. *et al.* “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” *NeurIPS*, 2019.
3. Harris, C. R. *et al.* “Array programming with NumPy.” *Nature*, 2020.
4. Bradski, G. & Kaehler, A. *Learning OpenCV 4.* O’Reilly Media, 2022.
5. Buslaev, A. *et al.* “Albumentations: Fast and Flexible Image Augmentations.” *arXiv*, 2018.
6. *Google Colaboratory* documentation, https://colab.research.google.com
7. *Unity Manual* and Scripting API, https://docs.unity3d.com
8. “MiniJSON” C# JSON Parser, MIT License, https://gist.github.com/davidhorton/ would-be-hash
9. TQDM: “fast, extensible progress bars for Python.” https://github.com/tqdm/tqdm
10. Otsu, N. “A threshold selection method from gray-level histograms.” *IEEE Trans. Sys., Man., Cybern.*, 1979.