

Architectural Orchestration of Procedural Generation: Leveraging Google Antigravity and Model Context Protocol in Unity

1. Introduction: The Paradigm Shift to Agentic Game Engineering

The domain of interactive software development is currently witnessing a fundamental transformation, shifting from manual implementation to high-level orchestration via agentic artificial intelligence. This shift is epitomized by platforms such as Google Antigravity, which introduces a "Mission Control" interface for managing autonomous agents capable of planning, executing, and validating complex engineering tasks.¹ In the context of Unity game development, this paradigm is further empowered by the Model Context Protocol (MCP), a standardized communication layer that bridges the gap between external AI agents and the internal state of the Unity Editor.³

This report presents an exhaustive analysis and operational framework for utilizing Google Antigravity to engineer a procedurally generated 2D map system. The objective is to synthesize a workflow where an AI agent—constrained by Simple Clean Architecture (SCA) and empowered by a local unityMCP server—analyzes specific 16x16 pixel art assets to construct a performant, production-ready terrain system. The challenge lies not merely in the algorithmic generation of noise maps, but in the precise instruction of a "blind" agent to navigate the Unity Asset Database, interpret visual rules from textual metadata, and generate code that adheres to strict architectural boundaries without introducing technical debt.

We will rigorously examine the visual semantics of the provided "blob" tilesets, specifically the 47-tile bitmasking requirements implied by the corner and edge variations visible in the source images. Furthermore, we will dissect the memory management implications of Unity's Tilemap system, advocating for batch-rendering optimizations that must be explicitly codified in the agent's instructions.⁵ By defining a robust .agents context environment and crafting a precise chain-of-thought prompt, this document serves as a comprehensive blueprint for developers transitioning from traditional coding to agentic architecture.

2. Visual Asset Analysis and Tilemap Theory

The efficacy of any procedural generation system is inextricably linked to the specific constraints of the visual assets it utilizes. The user has provided two distinct tilesheets: a terrestrial set containing grass, dirt, and stone walls (Image 1), and an aquatic set featuring water and island edges (Image 2). Both are defined as 16x16 pixel tiles, a specification that dictates critical settings within the Unity Editor and the logic of the procedural algorithms.

2.1. Deconstructing the "Blob" Tileset and Bitmask Logic

The visual structure of Image 1 reveals a sophisticated connectivity pattern often referred to as a "Blob" or "Wang" tileset. Unlike simple cardinal connectivity (which requires only 16 tiles), the provided asset includes internal corners, external corners, and varying edge permutations.

2.1.1. The 47-Tile Requirement

A standard "blob" tileset that supports seamless connectivity in all eight directions (North, North-East, East, South-East, South, South-West, West, North-West) requires 47 unique sprites to cover every possible adjacency case. This is mathematically derived from the 3x3 neighbor grid used by Unity's Rule Tile system.⁶

- **Center Tile:** Surrounded by identical tiles on all sides.
- **Edge Tiles:** 4 variations (N, E, S, W).
- **Corner Tiles:** 4 variations (NE, SE, SW, NW).
- **Inner Corner Tiles:** 4 variations (inv-NE, inv-SE, inv-SW, inv-NW).
- **Mixed Connectors:** Combinations of edges and corners required to prevent visual "seams" or "tearing" in complex terrain shapes.

Analysis of the user's Image 1 shows the "hole" patterns typical of inner corners (the white space "biting" into the green or brown fill), confirming that the agent must be instructed to configure a **Rule Tile with a 3x3 neighbor mask**. A simple 2x2 mask (16 sprites) would result in visual artifacts where the terrain transitions sharply without the appropriate corner smoothing.⁸

2.1.2. The 16x16 Pixel Density Constraint

The specification of 16x16 tiles imposes a strict requirement on the **Pixels Per Unit (PPU)** setting in Unity. For a standard 2D workflow where one tile equals one Unity unit (1 meter), the sprites must be imported with a PPU of 16.

- **Implication for the Agent:** The agent must be instructed to verify this setting via the mcp-unity server. If the PPU is left at the default 100, the 16-pixel tiles will appear microscopic (0.16 units wide), creating vast gaps in the grid if the Tilemap is set to a standard cell size of 1.0.
- **Filter Mode:** Given the pixel art aesthetic, the agent must ensure the Filter Mode is set to **Point (No Filter)** and compression is set to **None**. Bilinear filtering would blur the sharp edges of the 16x16 art, destroying the retro aesthetic.⁵

2.2. Unity RuleTiles: The Bridge Between Data and Visuals

The RuleTile is a ScriptableObject that encapsulates the adjacency logic. For the agent to "analyze" this, it must understand the underlying data structure of the .asset file.

Property	Function in Procedural Generation	Agent Instruction Strategy
Tiling Rules	A list mapping a neighbor bitmask to a specific sprite.	Agent must inspect the count of rules. < 47 implies a simplified set; 47+ implies full blob support.
Default Sprite	The fallback visual when no rules match.	Agent must identify this to ensure the map doesn't render "invisible" errors.
Collider Type	Determines if the tile generates physics collision geometry.	Agent must set this to Grid or Sprite based on the layer (e.g., Walls = Grid, Water = None).

Strategic Insight: The agent cannot "see" the image to determine which sprite is the "North West Outer Corner." However, it can infer this by reading the RuleTile configuration if the user has already set it up. If the RuleTile is not set up, the agent must guide the user to map specific sprite indices to specific bitmasks, a process that requires precise communication of the "index" of the sprite in the sliced texture.¹⁰

3. The Unity Model Context Protocol (MCP) Infrastructure

The Model Context Protocol (MCP) serves as the operational bridge, translating the AI agent's intent into executable C# commands within the Unity Editor.³ This server-client architecture transforms the IDE from a passive text editor into an active participant in the development lifecycle.

3.1. Server Architecture and Tooling Capabilities

The unityMCP server (referenced in the user query) typically exposes a suite of tools that the agent can invoke via JSON-RPC calls. Understanding the limits of these tools is critical for

crafting effective instructions.

3.1.1. Asset Discovery (`list_assets`, `find_assets`)

The agent's primary mechanism for "sight" is the AssetDatabase API exposed through MCP.

- **Mechanism:** The agent sends a query like `glob:Assets/**/*asset` or uses a type filter `t:RuleTile`.¹¹
- **Data Return:** The server returns a list of file paths and GUIDs.
- **Contextual Relevance:** The user asked the agent to "analyze my ruletiles." The agent must first *find* them. An instruction in the `.agents` folder must explicitly tell the agent: "Do not hallucinate asset paths. Always run `list_assets` filtering for `t:RuleTile` before generating code that references them."

3.1.2. Scene and Hierarchy Manipulation (`manage_scene`)

Procedural generation often requires a container hierarchy (e.g., `Grid` -> `Tilemap_Ground` -> `Tilemap_Walls`).

- **Capability:** The MCP agent can inspect the active scene to check if these objects exist.
- **Action:** If missing, the agent can trigger a command to create a `Grid` object with the specific Cell Size of (1, 1, 0) required for the 16x16 tiles.

3.1.3. Script Execution (`execute_script`)

This is the most powerful and dangerous tool. It allows the agent to inject arbitrary C# code into the running editor instance.⁴

- **Use Case:** To "analyze layers," the agent can run a script that calls `SortingLayer.layers`. This returns the precise ID and Name of every sorting layer defined in the project settings (e.g., "Background", "Default", "Foreground").
- **Why Necessary:** The standard file system search cannot reveal Sorting Layer settings easily as they are buried in `ProjectSettings/TagManager.asset`. A runtime script is the only reliable way for the agent to verify the existence of a "Water" layer.

3.2. Security and Telemetry Considerations

The integration of MCP servers often involves the collection of telemetry data, which tracks tool usage and performance metrics.¹³ While useful for tool developers, in a professional or proprietary environment, this may pose a data leakage risk.

- **Mitigation Strategy:** The instructions provided to the agent should include a directive to suppress telemetry if possible, or at least be aware of it.
 - **Instruction:** "Set `DISABLE_TELEMETRY=true` in the environment configuration to ensure that asset names and project structures are not transmitted externally."
-

4. Designing the Agentic Workspace: The .agents Directory

The user's request centers on "really good instructions in the .agents folder." In the Antigravity ecosystem, this folder acts as the "Constitution" for the AI, defining its persona, constraints, and operational protocols.¹⁴ A monolithic file is insufficient for complex Unity architectures; a modular approach is required to maintain context purity.

4.1. File Structure Strategy

We recommend the following file structure for the .agents directory to maximize the agent's comprehension of the SCA pattern and Tilemap specifics:

.agents/

├── index.md # Entry point and Persona definition

├── architecture.md # Strict definition of Simple Clean Architecture (SCA)

├── unity_standards.md # C# coding standards (UniTask, File-scoped namespaces)

├── visual_assets.md # Analysis of the 16x16 tiles and PPU rules

└── skills/

└── analyze_project.md # A "Skill" defining the sequence of MCP tools to run

4.2. Detailed File Contents

The following sections provide the exact text content the user should place in these files.

4.2.1. .agents/index.md (The Persona)

Unity AI Architect & Systems Engineer

You are an expert Unity Architect specializing in **Simple Clean Architecture (SCA)** and **Procedural Generation**. Your goal is to orchestrate the creation of a 2D Map Generation system using the Unity Model Context Protocol (MCP).

Core Directives

1. **Safety First:** You do not guess. You verify using MCP tools (list_assets, execute_script) before writing code that references assets.
2. **Strict Architecture:** You strictly adhere to the rules defined in architecture.md. No MonoBehaviour in the Entity or Usecase layers.

3. **Performance:** You prioritize runtime performance. Avoid garbage collection in hot loops. Use `Tilemap.SetTiles` (bulk) instead of `SetTile` (single).
4. **Visual Integrity:** You are aware that the project uses **16x16 pixel art**. You must enforce PPU=16 and Point filtering settings.

Workflow

When asked to generate the map system:

1. **Inventory:** Scan the project for `RuleTiles` and `Layers`.
2. **Plan:** Propose a file structure compliant with SCA.
3. **Implement:** Generate the C# code for Entity, Usecase, Presenter, and View.
4. **Integrate:** Write the `VContainer` dependency injection config.

4.2.2. .agents/architecture.md (The Constraints)

Simple Clean Architecture (SCA) Rules

The project follows a strict 5-layer architecture. Deviations are not permitted.

1. Entity Layer (Domain/Entities)

- **Role:** Pure data structures (POCOs).
- **Constraints:**
 - NO using `UnityEngine`; (except `Vector2Int` or basic structs).
 - NO inheritance from `MonoBehaviour` or `ScriptableObject`.
 - MUST be strictly serializable C# classes/structs (e.g., `WorldData`, `TileGrid`).

2. Usecase Layer (Domain/UseCases)

- **Role:** Business logic and algorithms (e.g., Perlin Noise generation, Cellular Automata).
- **Constraints:**
 - NO reference to View, Presenter, or Unity UI.
 - Input: `GenerationConfig` (Seed, Size).
 - Output: `WorldData` (Entity).
 - PURE C# logic only.

3. Gateway Layer (Infrastructure/Gateways)

- **Role:** Interface adapters for external data (Save/Load, Network).
- **Constraints:**
 - Must implement interfaces defined in the Domain.

4. Presenter Layer (Presentation/Presenters)

- **Role:** The glue between Logic and View.
- **Constraints:**
 - Entry Point for the scene.
 - Orchestrates the call to `Usecase.Execute()`.
 - Converts `WorldData` (int/enum) into `RuleTile` asset references.
 - Passes renderable data to the View.

5. View Layer (Presentation/Views)

- **Role:** Rendering and Input.
- **Constraints:**
 - Inherits `MonoBehaviour`.
 - The ONLY layer allowed to reference `UnityEngine.Tilemaps`.
 - Passive View pattern: Does not decide *what* to draw, only *how*.
 - MUST use bulk update methods (`SetTiles`) for performance.

4.2.3. .agents/visual_assets.md (The Asset Logic)

Visual Asset Guidelines: 16x16 Pixel Art

Asset Analysis

- **Source:** "Puny World" style pixel art.
- **Resolution:** 16x16 pixels per tile.
- **Connectivity:** Full "Blob" tileset connectivity (47-tile bitmask) is implied by the presence of inner and outer corners in the source images.

Rule Tile Configuration

- **Neighbor Rule:** Use **3x3** check (not 2x2) to support complex corners.
- **PPU:** All sprites must be imported with **Pixels Per Unit = 16**.
- **Filter Mode: Point (No Filter).**
- **Compression: None.**

Mapping Strategy

When generating the `MapPresenter`, map the logical `TileType` enums to specific `RuleTile` assets found in the project:

- `TileType.DeepWater` -> `Assets/.../Water_RuleTile.asset`
- `TileType.Grass` -> `Assets/.../Grass_RuleTile.asset`

- TileType.Wall -> Assets/.../Wall_RuleTile.asset

CRITICAL: Do not hardcode paths. Use Resources.Load or serialized fields in the MapSettings ScriptableObject.

5. Simple Clean Architecture (SCA) for Procedural Systems

The integration of procedural generation into Simple Clean Architecture (SCA) requires a disciplined separation of concerns. In traditional Unity development, "Manager" classes often become monolithic "God Objects" that handle data generation, asset management, and rendering simultaneously. SCA mitigates this by enforcing strict boundaries.¹⁵

5.1. The Entity Layer: WorldGrid

The foundation of the system is the data model. In SCA, the map is not a Tilemap; it is a mathematical grid.

C#

```
namespace Game.Domain.Entities
{
    public enum TileId { Empty = 0, Water = 1, Sand = 2, Grass = 3, Wall = 4 }

    public class WorldGrid
    {
        public readonly int Width;
        public readonly int Height;
        private readonly TileId _cells; // Flat array for performance

        public WorldGrid(int width, int height)
        {
            Width = width;
            Height = height;
            _cells = new TileId[width * height];
        }

        public TileId GetTile(int x, int y) => _cells;
```

```

    public void SetTile(int x, int y, TileId tile) => _cells = tile;
}
}

```

- **Insight:** By using a flat 1D array (TileId) instead of a 2D array (TileId[,]), we improve memory locality and cache coherence, which is vital when iterating over 15,000+ tiles (e.g., a 128x128 map).⁵

5.2. The Usecase Layer: GenerateTerrainUseCase

This layer contains the "brains." It implements the noise algorithms. Crucially, this layer is unit-testable outside of Unity.

C#

```
using Cysharp.Threading.Tasks; // Using UniTask as per.agents/unity_standards.md
```

```
namespace Game.Domain.UseCases
```

```
{
```

```
    public class GenerateTerrainUseCase
```

```
    {
```

```
        public async UniTask<WorldGrid> ExecuteAsync(int width, int height, int seed)
```

```
        {
```

```
            var grid = new WorldGrid(width, height);
```

```
            var fastNoise = new FastNoiseLite(seed); // Assuming a noise library
```

```
            // Offload heavy calculation to a background thread
```

```
            await UniTask.SwitchToThreadPool();
```

```
            for (int y = 0; y < height; y++)
```

```
            {
```

```
                for (int x = 0; x < width; x++)
```

```
                {
```

```
                    float noise = fastNoise.GetNoise(x, y);
```

```
                    TileId type = ResolveBiome(noise);
```

```
                    grid.SetTile(x, y, type);
```

```
                }
```

```
            }
```

```
            await UniTask.SwitchToMainThread();
```

```

        return grid;
    }

    private TileId ResolveBiome(float noiseValue)
    {
        if (noiseValue < -0.2f) return TileId.Water;
        if (noiseValue < 0.0f) return TileId.Sand;
        return TileId.Grass;
    }
}

```

- **Key SCA Feature:** This class has zero dependencies on UnityEngine.Tilemaps. It produces data, not GameObjects.

5.3. The Presenter Layer: MapPresenter

The Presenter bridges the divide. It takes the TileId from the Entity and resolves it to a RuleTile asset.

C#

```

namespace Game.Presentation.Presenters
{
    public class MapPresenter : IStartable
    {
        private readonly GenerateTerrainUseCase _useCase;
        private readonly IMapView _view;
        private readonly MapAssetRegistry _assets; // ScriptableObject map: TileId -> RuleTile

        public MapPresenter(GenerateTerrainUseCase useCase, IMapView view, MapAssetRegistry
assets)
        {
            _useCase = useCase;
            _view = view;
            _assets = assets;
        }

        public void Start()
        {

```

```

        GenerateAndRender().Forget();
    }

    private async UniTaskVoid GenerateAndRender()
    {
        var data = await _useCase.ExecuteAsync(100, 100, 12345);
        _view.Render(data, _assets);
    }
}

```

5.4. The View Layer: TilemapView

The View is the dumb renderer. It performs the specific Tilemap API calls.

C#

```

namespace Game.Presentation.Views
{
    public class TilemapView : MonoBehaviour, IMapView
    {
        private Tilemap _groundMap;

        public void Render(WorldGrid data, MapAssetRegistry registry)
        {
            var positions = new Vector3Int;
            var tiles = new TileBase;

            int i = 0;
            for (int y = 0; y < data.Height; y++)
            {
                for (int x = 0; x < data.Width; x++)
                {
                    positions[i] = new Vector3Int(x, y, 0);
                    tiles[i] = registry.GetRuleTile(data.GetTile(x, y));
                    i++;
                }
            }

            // BULK SET TILES - Critical for performance

```

```
        _groundMap.SetTiles(positions, tiles);  
    }  
}  
}
```

- **Performance Note:** Calling SetTiles once is $O(1)$ in terms of mesh rebuilding overhead, whereas calling SetTile in the loop is $O(N)$, which for $N=10,000$ causes the editor to hang for seconds.⁵

6. Algorithmic Terrain Strategies

Given the specific images (water islands and walled land), a hybrid algorithmic approach is recommended.

6.1. Perlin Noise for Base Terrain

The "Puny World" assets (Image 2) show organic coastlines. Perlin or Simplex noise is ideal here.

- **Frequency:** Determines "zoom." Lower frequency (e.g., 0.05) creates large continents.
- **Octaves:** Layering noise at different frequencies adds "roughness" to the coastline, making it look natural rather than smooth curves.¹⁷

6.2. Cellular Automata for Walls

Image 1 shows stone walls. If these represent caves or dungeons, Perlin noise is often too chaotic. Cellular Automata (Game of Life rules) is better.

- **Algorithm:**
 1. Random Fill (e.g., 45% wall).
 2. Smooth: For each cell, if neighbors > 4, become wall. Else, become floor.
 3. Repeat 5 times.
- **Result:** Distinct "rooms" and organic cave walls that fit the bitmasked wall tiles perfectly.¹⁸

7. The Master Prompt Engineering Strategy

To execute this complex orchestration, the prompt sent to Antigravity must be structured as a rigorous set of instructions, forcing the agent to think step-by-step.

7.1. The "Chain of Thought" Prompt

The following text is the **precise prompt** the user should input into the Antigravity interface:

Subject: Procedural Map Generation System Implementation (SCA compliant)

Context:

I have initialized the .agents folder with architecture.md (SCA), visual_assets.md (16x16 PPU rules), and unity_standards.md.

Task:

Orchestrate the creation of a procedural generation system using the unityMCP server to analyze my project state and generate the necessary C# code.

Execution Steps:

1. Discovery & Analysis (MCP)

- Connect to the unityMCP server.
- Run list_assets to find all assets of type RuleTile.
- Analyze the filenames. Look for terms like "Ground", "Water", "Grass", "Wall".
- Run a script to list all SortingLayers. Verify if layers named "Water", "Ground", and "Collision" exist. If not, note this for the setup plan.

2. Asset Verification

- confirm that the found RuleTile assets are configured. If you cannot read the internal YAML detailing the sprites, assume they are correctly set up but create a MapAssetRegistry ScriptableObject that allows me to manually link the Logical TileID to the Physical RuleTile.

3. Architectural Implementation (SCA)

- Generate WorldGrid.cs (Entity) using a flat 1D array for memory optimization.
- Generate GenerateTerrainUseCase.cs (Usecase). Implement a **Perlin Noise** algorithm for the base terrain. Add a generic Seed parameter. ensure it returns a UniTask<WorldGrid>.
- Generate MapPresenter.cs. This must be the entry point (implement IStartable from VContainer). It should request the map generation and pass the result to the View.
- Generate TilemapView.cs (View). It must reference a Tilemap component.
CRITICAL: Use Tilemap.SetTiles(positions, tiles) for the render pass. Do NOT use SetTile inside a loop.

4. Dependency Injection

- Create or Update GameLifetimeScope.cs. Register the Usecase (Scoped), the View (Component), and the Presenter (EntryPoint).

5. Validation

- Create a small Editor Script DebugMapGen.cs that allows me to click a button in the Inspector to trigger a generation pass for testing without playing the game.

Constraints:

- Strictly follow the 16x16 pixel density context.
- Ensure all async code uses UniTask and passes CancellationToken.
- Do not guess asset names; use the ones you found in Step 1.

8. Implementation Details & Code Artifacts

This section provides specific code snippets that the agent is expected to generate, serving as a "ground truth" for reviewing the agent's output.

8.1. Optimizing the SetTiles Call

One of the most common failures in AI-generated Unity code is the usage of `Tilemap.SetTile` inside a nested loop. The report emphasizes the bulk update pattern.

Inefficient (Bad):

C#

```
// Causes 10,000 mesh rebuilds
foreach (var pos in positions) {
    tilemap.SetTile(pos, tile);
}
```

Optimized (Required):

C#

```
// Causes 1 mesh rebuild
tilemap.SetTiles(positionsArray, tilesArray);
```

8.2. Handling the 16x16 Grid Alignment

The View layer must ensure the Grid component is aligned to the assets. The agent should generate code or instructions to verify the Grid's Cell Swizzle and Cell Size.

C#

```
// Inside MapView.cs or setup script
var grid = GetComponent<Grid>();
grid.cellSize = new Vector3(1, 1, 0); // Matches PPU=16 import settings
grid.cellGap = Vector3.zero;
```

9. Conclusion

The successful deployment of a procedural generation system in Unity using Google Antigravity relies on the precision of the context provided. By establishing a rigid Simple Clean Architecture within the .agents folder and utilizing the unityMCP tools to bridge the semantic gap between the AI and the Unity Editor, developers can achieve a workflow that is both autonomous and architecturally sound.

The analysis of the 16x16 "puny world" assets confirms the need for complex 3x3 neighbor RuleTiles and strict PPU settings, while the requirement for performance dictates the use of batch rendering API calls. The prompt and file structures defined in this report provide the necessary scaffolding to transform a generic AI model into a specialized Unity Systems Architect, capable of delivering a production-grade map generation system.

Table 1: Summary of Strategic Directives

Domain	Directive	Reasoning
Architecture	Enforce SCA (Simple Clean Architecture)	Decouples logic from Unity API, enabling unit testing of generation algorithms. ¹⁵
Visuals	3x3 Neighbor Bitmask	Required for the 47-tile "blob" set visible in source images (Image 1). ⁶

Performance	Batch SetTiles	Prevents O(N) mesh rebuild overhead during map population. ⁵
Tooling	Use execute_script for Layers	Unlocks access to Project Settings (Sorting Layers) inaccessible via standard file search.
Agent Context	Modular .agents files	Prevents context window pollution and enforces separation of concerns. ¹⁴

Works cited

1. Tutorial : Getting Started with Google Antigravity, accessed February 7, 2026, <https://medium.com/google-cloud/tutorial-getting-started-with-google-antigravity-b5cc74c103c2>
2. Getting Started with Google Antigravity, accessed February 7, 2026, <https://codelabs.developers.google.com/getting-started-google-antigravity>
3. Unity - AI-Powered Game Development Automation - MCP Market, accessed February 7, 2026, <https://mcpmarket.com/server/unity-1>
4. CoderGamester/mcp-unity: Model Context Protocol (MCP ... - GitHub, accessed February 7, 2026, <https://github.com/CoderGamester/mcp-unity>
5. Optimize performance of 2D games with Unity Tilemap, accessed February 7, 2026, <https://unity.com/how-to/optimize-performance-2d-games-unity-tilemap>
6. AutoTile | 2D Tilemap Extras | 4.2.1 - Unity - Manual, accessed February 7, 2026, <https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@4.2/manual/AutoTile.html>
7. Rule Tile | 2D Tilemap Extras | 1.6.0-preview.1 - Unity - Manual, accessed February 7, 2026, <https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@1.6/manual/RuleTile.html>
8. AutoTile | 2D Tilemap Extras | 4.3.0 - Unity - Manual, accessed February 7, 2026, <https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@4.3/manual/AutoTile.html>
9. How am I supposed to setup the bitmask for this tile set to get it to autotile? : r/godot - Reddit, accessed February 7, 2026, https://www.reddit.com/r/godot/comments/pp55g9/how_am_i_supposed_to_setup_the_bitmask_for_this/
10. 2d-techdemos/Assets/Tilemap/Rule Tiles/Custom Rule Tile/Scripts/CustomTypeRuleTile.cs at master - GitHub, accessed February 7, 2026, <https://github.com/Unity-Technologies/2d-techdemos/blob/master/Assets/Tilemap>

- [p/Rule%20Tiles/Custom%20Rule%20Tile/Scripts/CustomTypeRuleTile.cs](#)
11. Scripting API: AssetDatabase.FindAssets - Unity - Manual, accessed February 7, 2026,
<https://docs.unity3d.com/6000.3/Documentation/ScriptReference/AssetDatabase.FindAssets.html>
 12. How to find all assets of a type? - Stack Overflow, accessed February 7, 2026,
<https://stackoverflow.com/questions/29526625/how-to-find-all-assets-of-a-type>
 13. unity-mcp/docs/reference/TELEMETRY.md at beta - GitHub, accessed February 7, 2026,
<https://github.com/CoplayDev/unity-mcp/blob/beta/docs/reference/TELEMETRY.md>
 14. AGENTS.md – Open format for guiding coding agents | Hacker News, accessed February 7, 2026, <https://news.ycombinator.com/item?id=44957443>
 15. 01_architecture.md
 16. Simplified Clean Architecture Design Pattern for Unity | by Genki Sano (Co-founder & CTO, Telexistence Inc.), accessed February 7, 2026,
<https://genki-sano.medium.com/simplified-clean-architecture-design-pattern-for-unity-967931583c47>
 17. Procedural Island Map Generation in Unity 2D - FREE project - YouTube, accessed February 7, 2026, https://www.youtube.com/watch?v=byX_7m3Fnes
 18. Procedural patterns to use with Tilemaps, part 2 - Unity, accessed February 7, 2026,
<https://unity.com/blog/engine-platform/procedural-patterns-to-use-with-tilemaps-part-2>