Bigger Games - Case Report

Tips To Run

There is a GameConfig Scriptable Object at the GameManager. It has two checkbox.

One is for ShouldAnimatePieceCreation. When it is set to true, at every level, tangram pieces created with animation on the board, then scattered to the field.

The other one is for ShouldLoadDynamicLevel. When it is set to true, you can play infinitely many times and all the levels are generated from data on the LevelConfig Scriptable Objects. There are 3 blueprint levelconfigs for each difficulty level. You can change the parameters from Scriptable Object. At every win, seed will be incrementing so at the next level you will see different pieces with different color because seed is changed.

Responsibilities Of Manager Classes

- ServiceProvider → Service Locater Design Pattern, Creation of C# classes
- GameManager → Starting Game
- AssetLibrary → Holding configs and prefabs
- SpawnManager → Responsible for random spawning of the pieces at the bottom of the screen.
- PieceManager → Managing pieces
- LevelManager → Responsible for arranging level by validating, creating using levelReader.

- LevelReader → This system consist of 1 base abstract class and 2 class derived from it.

  DynamicLevelReader → Read level from Scriptable Object

  LocalLevelReader → Read level from JSON

- TangramManager → This system consist of 1 base abstract class and 2 class derived from it.

  AnimatedTangramManager → Pieces created with animation that user can see the process

  InstantTangramManager → Pieces created instantly

What is Procedural Generation ?

I would like to briefly explain the concept of Procedural Generation.

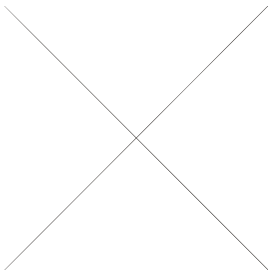
Procedural Generation is the process of determining object properties at runtime. It leverages the fact that computers cannot generate truly random numbers and that if the seed value is known, all random numbers produced throughout the program can be predicted.

For example, in the game *No Man's Sky*, there are quintillions of planets. If all these planets were stored as assets in the game, estimating the game's size would be extremely difficult. Instead, the game generates these planets at runtime using randomized algorithms based on specific parameters, ensuring that the only information that needs to be stored about a planet is its set of parameters to recreate same thing again.

In the videos I watched and the resources I read, I came across *Perlin Noise* and the *Wave Function Collapse Algorithm*. I learned about techniques such as using Perlin Noise to generate smoothly varying neighboring values or using Wave Function Collapse to propagate constraints across neighboring elements. However, I determined that these methods were not suitable for the algorithm I wanted to create.

General Structure Of Core Classes

Block



Triangles

Board consists of square *Block's*, with the dimensions of board size. If we consider these *Block's* as squares divided by diagonal lines, each *Block* contains 4 triangles. We can create all the pieces seen in the reference video using these triangles. Therefore, I choose to take these triangles as our base units.

However, as a rule, a single unit triangle is not enough to create a valid piece. To form a piece, we need at least 2 unit triangles. In this smallest piece configuration, the selected triangles must be either non-diagonal adjacent triangles or must combine with one of the triangles in the opposite position. This was a pattern I observed in the video and wanted to establish as a rule.

Procedural Generation Algorithm Overview

To recap the structure we built: a Board is composed of Blocks as a multidimensional array. Each Block consists of 4 Triangles. The square structure of the Board makes it easier to reach neighboring triangles from any given triangle in this context.

Based on the Board received from the Level, a Block multidimensional array is created. The Piece Count value determines how many pieces we will create. When creating Blocks, all triangles are collected in a list called activeTriangles. This list will allow us to access available locations (triangles within Blocks).

First, for each color in sequence, one triangle is selected from the activeTriangles list. A Piece is initialized with this triangle. Then, the captured triangle is removed from the accessible triangles. Since it needs at least 2 triangles to meet the valid Piece condition, we need to capture one more triangle. To maintain integrity, a triangle is selected from the list of accessible triangle locations that we call neighbors, it is captured and added to the Piece, and this location is removed from the list of visitable triangles. After performing the same action for each piece, we have the base pieces ready.

Now these pieces are value-copied to a new list. We will iterate through our pieces with this list. Starting from the first element of the list, a list of accessible triangles is obtained for a Piece. This is a list that holds suitable neighbors for visiting, checking not only the adjacent triangles of any triangle but also the neighbors of each part of the piece. The selected piece attempts to perform the

Capture operation we're familiar with from the previous step by randomly selecting a triangle from this list of accessible triangles.

This continues in sequence for other pieces as well. If there are no elements left in the list of accessible triangles for any piece, this means that piece has completed its formation phase. In this case, we remove this piece from the list of pieces we're iterating through. We continue this iteration until there are no elements left in the list. Thus, all pieces are initialized.
Continuing with the list of accessible triangles ensures piece integrity, while selecting 2 triangles in the first step ensures we meet the minimum 2 triangle requirement.

Placing Pieces Algorithm Overview

From a logical perspective, the input control for dragging and dropping pieces should be managed by the *Piece* itself. However, since we cannot determine the number of triangles or the exact shape of a *Piece* until the algorithm runs, placing a collider on the *Piece* directly is a challenging problem in the short term. Therefore, I added a Polygon Collider to each *Triangle* unit instead.

We could have handled input by casting a Ray from the *Piece* and detecting the hit *Triangle* child colliders, but using Unity's built-in methods to receive input directly from the *Triangle's* collider seemed more efficient. For this reason, I created a component called *TriangleInteractionHandler*, which communicates with the Piece to handle the drag operation.

When a *Piece* is dragged and dropped onto a location on the *Board*, the system first checks whether each *Piece* unit Triangle has a suitable Block *Triangle*. If at least one unit *Triangle* is not in a valid position, the entire *Piece* is considered unplaceable and vice versa. My triangle sprites are actually squares with just visible triangle shape. So even I placed 4 triangle prefabs, which is leftPrefab, rightPrefab, upperPrefab, bellowPrefab at the same position, they don't overlapped with each other.