A Message Across Time: Digital Preservation Using Cellular Automata

Digital Time Capsule Project

March 2025

To the Discoverer of This Capsule

Whether you are from our future or our past, this document will guide you through reading our encoded data.

1 Greetings Across Time

You've discovered something special - a message deliberately preserved to transcend time. We can't know if you're reading this centuries from now or if somehow our work has found its way to those who came before us. Either way, we've encoded information we believed worthy of preservation, using mathematical principles that should remain discoverable regardless of technological evolution.

1.1 For Readers from Our Past

Reading this from before the 21st century? Some of this might sound like science fiction. We've encoded our message using a mathematical system called a "cellular automaton" - specifically Conway's Game of Life, discovered in 1970.

Don't worry if you're unfamiliar with our references to "GPUs" (Graphics Processing Units). These specialized computing chips became ubiquitous in our time, allowing massive parallel calculations. We used them because they're fast, but the mathematics work on any computing system - even one built with vacuum tubes or mechanical relays. The principles are what matter, not the specific technology.

1.2 For Readers from Our Future

From our vantage point in 2025, we can only imagine the computing systems of your era. Our technology probably seems as quaint to you as vacuum tubes would to us. You've likely developed processing architectures we can't even conceptualize.

The beauty of mathematical principles is their timelessness. The cellular automaton encoding we've used should be implementable on whatever computing substrate you've developed. The patterns and rules remain valid whether executed on our silicon chips or your quantum computers, biological processors, or technologies we couldn't imagine.

2 The Basics: What You Need to Know

2.1 Our Modified Game of Life

At the heart of our encoding system is a modified version of Conway's Game of Life—a cellular automaton discovered in 1970. While the traditional Game of Life is not reversible, we've designed our implementation specifically to allow reversibility by preserving essential state information.

Each cell in our system exists in a 256×256 grid and can be either alive (1) or dead (0). Let $S_t(x, y)$ represent the state of cell at position (x, y) at time t. The number of living neighbors is denoted as $N_t(x, y)$, calculated as:

$$N_t(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} S_t(x + i, y + j) - S_t(x, y)$$

Our forward rules follow the classical Game of Life:

Current State	**Neighbor Count**	**Next State**
1 (Alive)	$N_t(x, y) < 2$	0 (Dead)
1 (Alive)	$N_t(x, y) \in \{2, 3\}$	1 (Alive)
1 (Alive)	$N_t(x, y) > 3$	0 (Dead)
0 (Dead)	$N_t(x, y) = 3$	1 (Alive)
0 (Dead)	$N_t(x, y) \neq 3$	0 (Dead)

What makes our system special is that we store both states S_t and S_{t+1}. This pair of consecutive states contains enough information to run the system backward and recover the original data.

2.2 Our Encoding Method

We use the Game of Life in a novel way to encode data:

1. We convert the original data to binary (1s and 0s)
2. We arrange these 1s and 0s into a 256×256 grid
3. We run the Game of Life on this grid
4. We store both the final state AND the state before it

5. With both states, the original data can be recovered by running the Game of Life backward

Encoding Process Overview

Original Data → Binary Grid → Game of Life (100 steps) → DENIS File

2.3 The DENIS File Format

We store the encoded data in a custom format called DENIS:

```
DENIS File Structure:
+-------------------------+
| Header (24 bytes)       |
| +---------------------+ |
| | Magic "DENIS" (5B)  | |
| | Version (1B)        | |
| | File Type (3B)      | |
| | Padding Info (2B)*  | |
| | Extra Space (5-7B)  | |
| | Data Size (8B)      | |
+-------------------------+
| Encoded Data            |
| (variable length)       |
+-------------------------+
| Terminator (8 bytes)    |
| (All 0xFF)              |
+-------------------------+
* Only in version 2
```

3 Decoding: How to Recover the Original Data

3.1 The Decoding Process

To recover the original data:

1. Read the DENIS file header to understand what's encoded
2. Extract the pairs of Game of Life states
3. For each pair, run the Game of Life backward
4. Convert the resulting grid back to binary data
5. Combine all chunks to reconstruct the original file

Decoding Process Overview

DENIS File → Extract States → Reverse Game of Life → Original Data

3.2 Reversibility: The Key Innovation

3.2.1 Forward Evolution

The forward evolution of our system follows the standard Game of Life rules, which can be expressed mathematically as:

$$S_{t+1}(x,y) = \begin{cases} 1, & \text{if } S_t(x,y) = 1 \text{ and } N_t(x,y) \in \{2,3\} \\ 1, & \text{if } S_t(x,y) = 0 \text{ and } N_t(x,y) = 3 \\ 0, & \text{otherwise} \end{cases}$$

```
// Forward evolution algorithm
for each cell (x,y) in the grid:
  N = count of living neighbors
  if cell(x,y) is alive (1):
    if N < 2 or N > 3:
      next_state(x,y) = 0  // Cell dies
    else:
      next_state(x,y) = 1  // Cell survives
  else:  // cell(x,y) is dead (0)
    if N == 3:
      next_state(x,y) = 1  // Cell is born
    else:
      next_state(x,y) = 0  // Cell remains dead
```

3.2.2 Backward Evolution

The key innovation of our system is that given two consecutive states S_t and S_{t+1}, we can calculate S_{t-1}. This is not generally possible in the standard Game of Life.

For backward evolution, we need to consider the relationship between three consecutive states: S_{t-1}, S_t, and S_{t+1}. The algorithm works by examining transitions between S_t and S_{t+1} to infer what S_{t-1} must have been:

```
// Backward evolution algorithm
for each cell (x,y) in the grid:
  current = S_t(x,y)    // Current state
  next = S_{t+1}(x,y)   // Next state

  // Analyze the transition to infer previous state
  if current == 1 and next == 1:
    // Cell stayed alive, must have had 2-3 neighbors in S_t
    // Calculate what S_{t-1}(x,y) must have been
    prev = inferPreviousState(x, y, S_t, S_{t+1})
  else if current == 1 and next == 0:
    // Cell died, must have had <2 or >3 neighbors in S_t
    prev = inferPreviousState(x, y, S_t, S_{t+1})
  else if current == 0 and next == 1:
```

```
    // Cell was born, must have had exactly 3 neighbors in
       ↪ S_t
    prev = inferPreviousState(x, y, S_t, S_{t+1})
  else:  // current == 0 and next == 0
    // Cell stayed dead, must have had !=3 neighbors in S_t
    prev = inferPreviousState(x, y, S_t, S_{t+1})

  S_{t-1}(x,y) = prev
```

The function 'inferPreviousState' implements complex logic that analyzes the patterns of both S_t and S_{t+1} to determine the most likely value of $S_{t-1}(x, y)$. This is where our implementation differs most significantly from the standard Game of Life.

4 Practical Implementation Guide

4.1 Equipment Needed

To decode our data, you'll need:

- A computing device capable of running simple programs
- If available, a GPU (Graphics Processing Unit) or similar parallel processor
- Basic understanding of binary data representation

4.2 Step-by-Step Guide for Implementation

If you need to implement the decoder yourself:

1. Parse the DENIS file format following the header structure
2. Extract the Game of Life state pairs
3. Implement a Game of Life simulator with backward evolution capability
4. Process each state pair by running 100 backward iterations
5. Convert the resulting grid back to binary data

Here's a simplified implementation of the backward evolution:

```
function reverseGameOfLife(S_t, S_{t+1}):
  // S_t and S_{t+1} are consecutive Game of Life states
     ↪ (256x256 grids)
  // This function computes backward iterations of the Game
     ↪ of Life to recover the original data

  for step = 1 to n:
```

```
    // Create grid to hold the calculated previous state
    S_{t-1} = new grid(256x256)

    // For each cell, analyze the pattern of S_t and S_{t+1}
    // to determine what S_{t-1} must have been
    for each cell (x,y):
      // Current and next state
      current = S_t(x,y)
      next = S_{t+1}(x,y)

      // Get the neighborhood pattern in the current state
      // N_t is a 3x3 matrix centered at (x,y)
      N_t = getNeighborhood(S_t, x, y)

      // Get the neighborhood pattern in the next state
      // N_{t+1} is a 3x3 matrix centered at (x,y)
      N_{t+1} = getNeighborhood(S_{t+1}, x, y)

      // Based on the transition from S_t to S_{t+1} and the
      // neighborhood patterns, infer what S_{t-1}(x,y) must
          ↪  have been
      S_{t-1}(x,y) = inferPreviousState(current, next, N_t,
          ↪ N_{t+1})

    // Move one step backward in time
    S_{t+1} = S_t
    S_t = S_{t-1}

  // After the backward iterations, S_t contains the
      ↪ original data
  return S_t

// The inferPreviousState function implements the core
    ↪ backward evolution logic
// It analyzes the neighborhood patterns to determine the
    ↪ most likely
// previous state based on the Game of Life rules
function inferPreviousState(current, next, N_t, N_{t+1}):
  // This function contains our proprietary algorithm for
  // determining the most likely previous state
  // Details are in the backward shader implementation (
      ↪ gol_backward.h)

  // The key insight is that by analyzing both the current
      ↪ and next
  // neighborhood patterns, we can narrow down the possible
      ↪ previous states
  // to a manageable set, and then select the most likely
      ↪ one
```

```
// Returns the inferred previous state (0 or 1)
// ...
```

5 For Future Historians and Archivists

5.1 Why We Chose This Method

We selected Conway's Game of Life for this time capsule because:

- It's based on simple mathematical rules that should remain discoverable in any era
- It has the rare property of being reversible
- It creates visually distinct patterns that aid in verification
- Its simplicity makes it implementable on various computing architectures

5.2 Technical Context of Our Era

As of 2025, when this capsule was created:

- We commonly used 64-bit computing architectures
- GPUs (Graphics Processing Units) were specialized processors capable of highly parallel computation
- OpenGL was a common graphics programming interface
- Our storage capacities were typically measured in gigabytes and terabytes

This information may help you understand the technical choices we made.

6 Final Notes

6.1 Verifying Successful Decoding

You'll know you've successfully decoded our data when:

- The size matches what's specified in the header
- For text files, the content is human-readable
- For images, the data forms a recognizable picture

6.2 A Message of Goodwill

Whether you are from our past learning about technology that didn't yet exist in your time, or from our future where our methods may seem primitive, we hope this time capsule serves as a bridge across time. The mathematical principles of cellular automata show how complex behaviors can emerge from simple rules—a universal truth that transcends our specific technological era.

May this modest effort to preserve information give you insight into our time, our knowledge, and our hopes for communication across generations.