Clock Portal S3

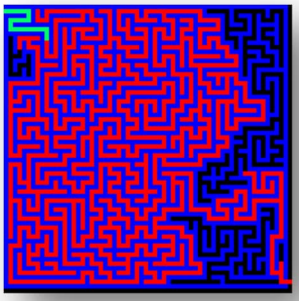
STEM Guide

The Clock Portal S3 is an instrument which turns the mathematics of processes into flowing, animated artwork.

Beyond its technical value, the device offers a striking visual learning experience demonstrating how simulations describe the real world.

In this document you'll find a deeper look at the algorithms that drive the visuals.

Maze Generation & Solving



- The maze is generated using an iterative Depth-First Search (DFS) recursive backtracker with an explicit stack in static memory. The maze spans a 31×31 cell grid

    1. **Initialize the maze grid**

    Every cell starts with all four walls present and is marked unvisited. A stack is prepared for DFS maze generation. The algorithm begins at the top left cell (0,0), marks it visited, and pushes it onto the stack.

    2. **Depth-First Search backtracking (maze carving)**

    The algorithm repeatedly looks at the current cell on top of the stack, checks all four directions (N, S, E, W) for unvisited neighbors, and:

    - If some unvisited neighbors exist, it randomly selects a direction, removes the wall between the current cell and the neighbor, marks the neighbor visited, and pushes it onto the stack.
    - If no unvisited neighbors exist, it pops the stack to backtrack. This continues until the stack is empty, producing a perfect maze with no loops and no unreachable cells.

    3. **Render the maze background**

    The screen is cleared to black to represent open corridors. Corner pillars are drawn at every grid intersection and outer border walls are rendered around the maze perimeter.

    4. **Draw internal maze walls**

For each cell:

- If the east wall is present, draw a vertical wall pixel.
- If the south wall is present, draw a horizontal wall pixel. Walls are drawn in vibrant blue.

5. Display the final maze

The fully rendered maze is rendered to the LED panel, showing dark paths, blue walls, a cyan entrance marker, and a gold exit marker.

- **Solving**

The maze is solved using an animated Breadth-First Search (BFS) algorithm, with the search paths highlighted in red.

1. Solver initialization

All solver-related flags and parent pointers are cleared. The BFS queue is reset. The upper left start cell (0,0) is marked visited and queued.

2. BFS solver steps

i. If the queue is empty the solver is finished.

ii. Pop the next cell, draw its position in red, and draw a connecting pixel to its parent (or the entrance if starting).

iii. If the current cell is the exit:

- Draw the exit indicator.
- Trace parent pointers backward to build the final path array.
- Switch to path-drawing mode and stop BFS.

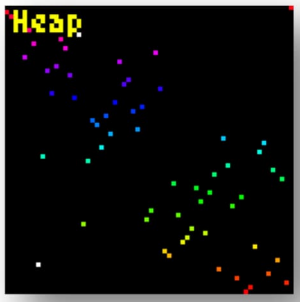
iv. Otherwise, for each direction:

- If there is no wall blocking movement…
- And the neighbor is inside bounds…
- And not yet visited…
  - Mark it as visited, store the parent, and queue the neighbor.

v. Push the updated path to the display.

3. **Path-drawing step (reverse animation of final route)**
    i. If all path nodes have been drawn, the solver is done.
    ii. Draw the current path cell in green and connect in to the next cell.
    iii. Highlight entrance and exit if applicable.
    iv. Show the frame and move to the previous node in the path.
    v. When the index reaches zero, the solver enters the done state.
4. **Restore maze display**
    i. Redraw all maze walls.
    ii. Re-draw BFS-visited cells in red including parent connectors.
    iii. If BFS reached the goal, redraw the exit indicator.
    iv. Re-draw the solved path in green.
    v. Display the reconstructed frame.

Sorting Demo



- Visually demonstrates seven widely known sorting algorithms on a set of 64 random numbers.
- The algorithms cycled through are: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, Heap Sort, and Shell Sort. Each “round” of sorting uses the same list of random numbers for the selected algorithms so the sorting times and method can be directly compared.
- After a “round” is completed, a horizontal bar flows from top to bottom to erase the panel and a new round is started with a different selection of random numbers. White or red highlights show the value being compared showing pivot points or segmentation selected by the algorithm.
- 64 random height vertical bars (or dot heights) are shown at the start of each algorithm. If “No Repeat” is selected, every number between 1 to 64 are shuffled and used as the bar (or dot) heights. If “No Repeat” is **not** selected, 64 random numbers, with possible repeats, are generated and used for the heights.
- The same bar order is used in each “round” for the selected algorithms. When the last of the selected algorithms completes, a new set of random numbers is generated and the sequence repeats.
- The algorithms are:

  1. **Bubble Sort** - Repeatedly compares adjacent elements and swaps them if they are out of order.

     - Scan left → right
     - Swap neighbors when needed
     - Largest values drift toward the end for each pass

2. **Insertion Sort** - Builds a sorted region at the left by inserting each new element into its correct position.

- Take the next element (“key”)
- Shift larger elements right
- Insert the key into the gap

3. **Selection Sort** - Finds the smallest element in the unsorted region and places it at the next sorted position.

- Scan entire unsorted range
- Track the index of the minimum
- Swap the minimum into position

4. **Merge Sort (Bottom-Up)** - Sorts by repeatedly merging small sorted runs into larger sorted runs until the entire list is sorted.

- Start with runs of size 1
- Merge adjacent runs
- Double the run size each iteration

5. **Quick Sort (Iterative Partitioning)** - Partitions the list around a pivot so smaller elements move left and larger elements move right.

- Choose a pivot
- Partition a sub-range
- Push left/right partitions onto the stack
- Process until stack empty

6. **Heap Sort** - Uses a max-heap to repeatedly extract the largest element and place it at the end of the array.

- Build max-heap
- Swap root with last element
- Sift down to restore heap
- Shrink heap and repeat

7. **Shell Sort** - Generalizes insertion sort by comparing elements far apart using a gap that shrinks over time.

- Start with large gap
- Perform gap-based insertion sort
- Reduce gap until 1 is reached
- Final pass is standard insertion sort

One can see that different sorting algorithms can “beat” others depending on the initial sequence ordering.

Hourglass

- Shows the passage of time by virtual sand flowing from the top chamber to the bottom chamber.
- The **Hour mode** shows the passage of seconds in the current hour. Since an hour is 3,600 seconds, the bottom chamber will be ¼ full at 900 seconds (15 minutes), ½ full at 1,800 seconds, etc.
- The **Day mode** shows the passage of seconds in the current day. One day is 86,400 seconds long, so the bottom chamber will be ¼ full at 21,600 seconds past midnight, ½ full at 43,200 seconds past midnight, etc.
- Process

    1. Compute how much sand belongs in each chamber

    The code first determines how many interior pixels exist in the top and bottom halves of the hourglass. It uses these counts as “capacity” values. It then computes a scaled fraction from 0–10,000 based on either the seconds past the hour or the seconds past midnight depending on the setting.

    Using that fraction, it calculates: topSand = how many grains remain in the upper chamber and bottomSand = how many grains have fallen into the lower chamber.

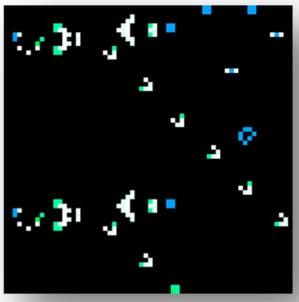
    2. Fill the top chamber

    Starting from the throat and moving upward, it fills rows of the top chamber. For each row, it computes the hourglass width at that Y coordinate. It marks interior pixels as “sand” until topSand is exhausted. This produces a clean, level sand surface that rises or falls smoothly.

    3. Simulate settling grains in the bottom chamber

For each grain that belongs in the bottom chamber, it calls a function that simulates a grain falling and settling naturally, alternating left/right bias. This gives the bottom pile an organic, sloped shape instead of a flat fill.

Game of Life

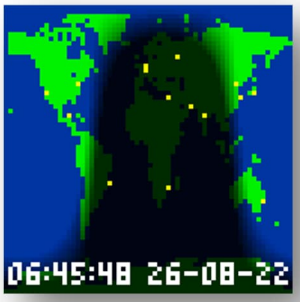


- Shows cellular automata patterns patterns using the classic “Conway’s Game of Life” rules.
- Each pixel (or "cell") can be either alive or dead, and each evolves according to the following rules:
    1. Any live cell with fewer than two live neighbors dies (underpopulation)
    2. Any live cell with more than three live neighbors dies (overpopulation).
    3. Any live cell with two or three live neighbors lives, unchanged, to the next generation.
    4. Any dead cell with exactly three live neighbors comes to life.
- Pixels take on different colors depending on their “age.”
- There are several initial starting patterns that are chosen at random. The terminology (e.g. Glider) used below can be looked up easily on the internet:
    1. Gun + oscillator + methuselah fires gliders into open space.
    2. Spaceship traffic with three westbound LWSS lanes plus two southeast gliders.
    3. Chaos garden: three methuselahs bloom around a stable pulsar.
    4. Twin guns stacked vertically; both glider streams exit the right edge.
    5. Glider squadron sweeping southeast, pulsar tucked into bottom-left.
    6. Methuselah row across the top, spaceship and glider underneath.

7. Gun firing into a Snark reflector; gliders bend downward.
8. Three methuselahs blooming independently; P54 anchors the bottom.
9. Three spaceship lanes with oscillators underneath.
10. Queen Bee in corner; gliders feed a Snark; pulsar stabilizes center.
11. Gun lane with oscillator triad and a drifting methuselah.
12. Chaos basin on left; Snark reflector; HWSS sweeps right edge.
13. Pulsar hub with gun corridor and T-pentomino bloom.
14. Snark relay chain with glider feed and P30 stabilizer.
15. Pulsar + oscillator ring + HWSS sweep.
16. Gun + Snark + Pi-heptomino bloom.
17. Triple methuselah bloom with spaceship traffic.
18. Snark gate with pulsar hub and drifting diehard.
19. Oscillator triad + HWSS + glider stream.
20. Gun + Queen Bee + Snark + T-pentomino bloom.
21. Pulsar + spaceship highway + Pi-heptomino bloom.

- After a pattern reaches a certain number of generations **or** settles into a stagnant pattern, another starting pattern is chosen and the sequence begins anew.
- Stagnation is detected by generating a hash of the panel each generation and looking for a repeat within the last 32 states. This catches still life patterns and any oscillators up to a selected period. Population counts alone miss detecting stagnation since blinkers and pulsars keep the cell count moving.

Day/Night Map



- Shows which portions of the earth are in daylight, twilight, or night.
- If you select Show Cities, yellow dots will mark New York, Los Angeles, London, Paris, Moscow, Cairo, Johannesburg, Dubai, Mumbai, Beijing, Shanghai, Tokyo, Sydney, São Paulo, and Mexico City.
- You may also choose to display the sub-solar position which is the point on earth where the sun is directly overhead.
- The shape of the day/night termination areas slowly and continually change with time as the Earth orbits the sun.
- Shape changes are due to the Earth's 23.5° tilt relative to the sun and the effect of projecting the spherical Earth as a 2D map.
- This is a great educational aid to demonstrate why daylight is shorter in the winter in the northern hemisphere and longer in the southern hemisphere.
- A 2D using a Equirectangular (Plate Carrée) projection of the earth is drawn. Then, computations involve:

1. Calendar math

   Determine whether a given year is a leap year. Looks up how many days are in each month (with February adjusted for leap years). Converts a date (year, month, day) into a day-of-year index (1–365/366) by summing all days in earlier months and adding the current day.

2. Solar geometry

   Compute the solar declination (the tilt of Earth relative to the sun on that day). Compute the sun's longitude based on UTC time. Computes an accurate subsolar longitude (where the sun is directly overhead) using

NOAA's "equation of time," which accounts for Earth's orbital eccentricity and axial tilt.

3. Global solar state

   Stores the current UTC date/time. Converts that into: day of year, solar declination (degrees + radians), and corrected subsolar longitude. These values are reused during rendering.

4. Precompute latitude trig tables

   For each screen row (0–63), convert pixel Y → latitude. Precomputes sin $(latitude)$ and cos $(latitude)$ for fast shading calculations.

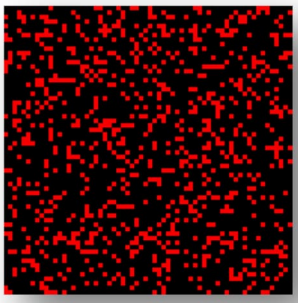
5. Compute sunlight at each pixel

   For each pixel (x,y): Convert pixel X → longitude. Use the solar dot-product formula to compute cos (A), which represents how directly sunlight hits that location.

   Converts cos (A) into a twilight blend factor (full daylight → twilight → night).

   Checks whether the current pixel is land or ocean. Applies different color shading for land vs ocean based on the twilight blend factor. Draws the pixel.

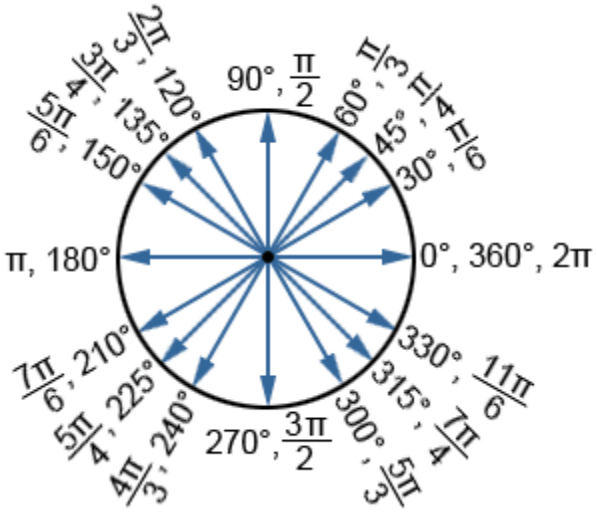
Zen Modes



- Displays pixels that are randomly turned on and off.
- A density is defined to choose the percentage of pixels that should be in the ON state vs. the OFF state.
- Bounded random delays are introduced to display the pixels for a random ON and OFF time to generate the effect.

Analog Clock

- Hand positions are determined by computing the angles in coordinate system where the reference angle is 0 ("straight up"). Angles are computed in radians.



Examples:

Hand angle formulas:

- $Minute\ Hand\ Angle = \left(\frac{Current\ Minutes}{60}\right) \cdot 2\pi - \frac{\pi}{2}$

  Note that subtracting $\frac{\pi}{2}$ rotates the coordinate system counterclockwise by 90 degrees (or, equivalently, $\frac{\pi}{2}$ radians).

- **For minutes = 0 :**

$$Minute\ Hand\ Angle = \left(\frac{0}{60}\right) \cdot 2\pi - \frac{\pi}{2} = -\frac{\boldsymbol{\pi}}{\mathbf{2}}$$

Which corresponds to “straight up”

- **For minutes = 15 :**

$$Minute\ Hand\ Angle = \left(\frac{15}{60}\right) \cdot 2\pi - \frac{\pi}{2} = \mathbf{0}$$

Which corresponds to pointing right (3 o’clock).

- The hour hand angle is computed similarly:

$$Hour\ Hand\ Angle = \left(\frac{Current\ Hours}{12}\right) \cdot 2\pi - \frac{\pi}{2}$$

- A line is then drawn at the angle beginning at the center of the face of the clock.

STEM Guide — Student Worksheet Version

This worksheet is designed for students to explore algorithms, simulations, and mathematical models through hands-on thinking tasks. Each section includes short-form questions, challenges, and reflection prompts.

Maze Generation & Solving

Student Exercises

- **Understanding DFS:** Explain in your own words how the DFS backtracking algorithm creates a maze.
- **Wall Removal:** Draw a 4×4 grid and simulate removing walls using DFS for at least five steps.
- **BFS Pathfinding:** Why does BFS always find the shortest path in a maze? Write a one-paragraph explanation.
- **Compare Algorithms:** Describe one difference between DFS (generation) and BFS (solving).
- **Reflection:** How does randomness affect the shape of the maze?

Sorting Demo

Student Exercises

- **Identify Patterns:** Watch two sorting algorithms and describe how the bars move differently.
- **Time Complexity:** Match each algorithm to its Big-O time complexity.
- **Stability Check:** Give an example of a dataset where stability matters.
- **Hands-On:** Sort a list of 10 numbers using Insertion Sort by hand.
- **Reflection:** Which algorithm seemed fastest visually? Why?

Hourglass

Student Exercises

- **Fraction Math:** If 1,800 seconds have passed in an hour, what fraction of the sand should be in the bottom chamber?
- **Geometry:** Sketch the hourglass shape and label the top and bottom capacities.
- **Simulation:** Describe how the sand "settles" in the bottom chamber.
- **Reflection:** Why does the bottom pile form a slope instead of a flat surface?

Game of Life

Student Exercises

- **Rule Summary:** Write the four rules of Conway's Game of Life.
- **Predict Behavior:** Draw a small 5×5 grid and predict the next generation.
- **Pattern Types:** What is a glider? What is an oscillator?
- **Reflection:** Why do some patterns grow while others die out?

Day/Night Map

Student Exercises

- **Earth Geometry:** What causes the day/night boundary to change shape throughout the year?
- **Latitude/Longitude:** Convert pixel coordinates to latitude and longitude using simple proportional reasoning.
- **Solar Position:** Explain what the "sub-solar point" means.
- **Reflection:** Why are winter days shorter in the northern hemisphere?

Zen Modes

Student Exercises

- **Probability:** If density is 40%, how many pixels out of 100 should be ON?

- **Randomness:** Describe how randomness affects the visual pattern.
- **Reflection:** Why do some frames look more chaotic than others?

Analog Clock

Student Exercises

- **Angle Practice:** Compute the minute hand angle at 30 minutes.
- **Coordinate Sketch:** Draw a clock and show where the hour hand points at 3:00.
- **Reflection:** Why is the minute hand moving even between whole minutes?

STEM Guide — Educator Lesson Plan Version

This lesson-plan version provides objectives, discussion prompts, and assessment ideas for each section.

Maze Generation & Solving

Learning Objectives

- Understand DFS and BFS algorithms.
- Explore pathfinding and maze structure.

Discussion Prompts

- How does randomness influence maze complexity?
- Why is BFS optimal for shortest-path discovery?

Activities

- Students simulate DFS on paper.
- Compare BFS vs A* in a classroom demo.

Assessment

- Short written explanation of DFS and BFS differences.

Sorting Demo

Learning Objectives

- Compare sorting algorithms and their efficiencies.
- Interpret visual sorting behavior.

Discussion Prompts

- What does “Big-O” tell us about algorithm performance?
- Why do some algorithms perform better on nearly sorted data?

Activities

- Students hand-sort lists using different algorithms.
- Class debate: “Which sorting algorithm is most intuitive?”

Assessment

- Students label sorting algorithm diagrams.

Hourglass

Learning Objectives

- Connect time measurement to visual simulation.
- Understand fractional mapping and geometry.

Discussion Prompts

- Why does the bottom sand pile form a slope?
- How does pixel capacity relate to real-world volume?

Activities

- Students compute sand fractions for different times.

Assessment

- Worksheet on fraction-to-pixel mapping.

Game of Life

Learning Objectives

- Explore cellular automata and emergent behavior.
- Understand rule-based systems.

Discussion Prompts

- What makes a pattern stable or chaotic?
- How do simple rules create complex behavior?

Activities

- Students simulate two generations of a pattern.
- Identify gliders, oscillators, and still lifes.

Assessment

- Students classify patterns from screenshots.

Day/Night Map

Learning Objectives

- Understand Earth-Sun geometry.
- Interpret global daylight patterns.

Discussion Prompts

- Why does daylight vary by season?
- How does Earth's tilt affect solar declination?

Activities

- Students map pixel coordinates to lat/long.
- Compare twilight zones at different times of year.

Assessment

- Short quiz on solar geometry.

Zen Modes

Learning Objectives

- Explore randomness and probability.
- Interpret stochastic visual systems.

Discussion Prompts

- What does density mean in a random system?
- How do ON/OFF durations affect perceived motion?

Activities

- Students compute expected ON pixels for various densities.

Assessment

- Probability worksheet.

Analog Clock

Learning Objectives

- Connect angles to time measurement.
- Interpret coordinate-based rendering.

Discussion Prompts

- Why does the minute hand move continuously?
- How do radians relate to clock positions?

Activities

- Students compute angles for various times.

Assessment

- Angle-calculation quiz.