Clock Portal S3



Science, Technology, Engineering, and Mathematics (STEM) Guide

tg tech

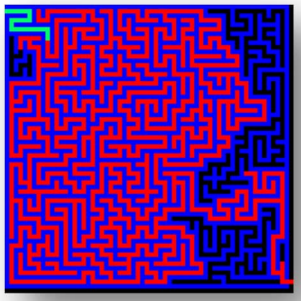
Table of Contents

Table of Contents 1
Introduction and Background 2
Maze Generation & Solving 3
Sorting Demonstrations 6
Wall Calendar 9
Hourglass 12
Game of Life 14
Day/Night Map 16
Zen Modes 21
Analog Clock 23
STEM Guide — Student Worksheet Version 25
STEM Guide — Educator Lesson Plan Version 29

Introduction and Background

- The Clock Portal S3 is an instrument which displays results of mathematical calculations into flowing, colorful, animated artwork.
- Beyond its value as a timekeeping instrument, the device offers a visual learning experience demonstrating how simulations describe the real world.
- In this document you'll find a deeper look at the concepts, mathematics, and algorithms that drive the device.

Maze Generation & Solving



- The maze is generated using an iterative Depth-First Search (DFS) recursive backtracker with an explicit stack held 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.

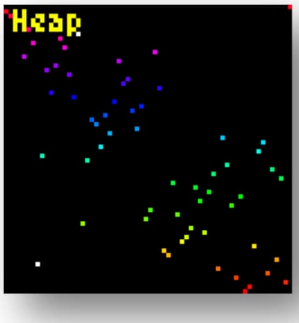
v. Push the updated path to the display.

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 the 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.

Practical Applications

- **Robotics navigation:** DFS/BFS searches mirror how robots explore unknown spaces, maps, and find transportation paths.
- **Network routing:** Graph traversal is essentially maze solving. Routers use BFS logic to establish network paths.
- **Printed Circuit Board and chip routing:** Very Large Scale Integration chip signal routing uses maze algorithms (Lee's algorithm).
- **Game development:** Procedural dungeon generation, AI pathfinding, level design.
- **Security analysis:** Attack-path discovery in network graphs.
- **Optimization research:** Benchmarking search algorithms, heuristics, and backtracking strategies.

Sorting Demonstrations



- Visually demonstrates seven widely known sorting algorithms on a set of 64 randomly selected numbers.
- The algorithms 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 algorithm(s) 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 values being compared and expose pivot points or segmentation selected by the algorithm.
- When a round begins, the display is initialized with vertical bars (or dots) whose heights are generated as follows: If **No Repeat** is enabled, the heights are a shuffled permutation of the numbers 1 through 64, each used exactly once. If **No Repeat** is disabled, the system generates 64 random values between 1 and 64, allowing duplicates. The bars (or dots) are then drawn using these values as their respective 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

It is easy to see that different sorting algorithms can outperform others depending on the initial sequence order.

Practical Applications

- **Data science foundations:** Sorting is the backbone of indexing, searching, and database operations.
- **Algorithm performance analysis:** Visualizing comparisons/swaps reveals time complexity behavior.
- **Embedded system benchmarking:** Sorting stresses memory, stack, and CPUs for microcontroller testing.
- **Computer science education:** Sorting visualizers are used everywhere to teach algorithmic thinking.

Wall Calendar



This function draws the calendar much the same way a person would lay out a paper calendar.

First, it determines which month and year should be displayed by reading the real-time clock and prints the month title centered at the top of the screen. If the user is viewing November 2026, for example, it writes "November 2026" across the top.

The next step is figuring out where to place day one. A calendar can't simply start counting at the upper-left corner because every month begins on a different day of the week. To solve this, the code creates a temporary date representing the first day of the selected month and asks the C runtime library (mktime) to determine what weekday that date falls on. Internally, mktime() converts the calendar date into a continuous count of days from a fixed reference date and then computes the weekday by dividing by 7 and taking the remainder.

Although the calendar code only provides a year, month, and day, the mktime() library performs a surprisingly complex series of calculations behind the scenes. To determine the weekday for a date such as November 1, 2026, it first evaluates whether each year between a known reference date and the target date is a leap year. Under the Gregorian calendar, a year is a leap year if it is divisible by 4, except for century years that are divisible by 100. However, century years divisible by 400 remain leap years. This means 2000 was a leap year, while 1900 and 2100 are not.

After determining which years contain an extra day, mktime() calculates the total number of days that have elapsed from a reference point up to the requested date. It adds the appropriate number of days for each year, accounts for the varying lengths of the months, and inserts an additional day whenever a leap year includes February 29.

Once the total day count has been calculated, determining the day of the week becomes a simple mathematical problem. Because weekdays repeat every seven days, the library divides the total elapsed days by seven and examines the remainder. That remainder identifies whether the date falls on a Sunday, Monday, Tuesday, and so forth.

By combining leap-year rules, month-length calculations, and modulo-seven arithmetic, mktime() can accurately determine the weekday for virtually any valid date. The calendar program then uses this result to place the first day of the month into the correct column, ensuring that all remaining dates align properly beneath their corresponding weekday headings.

The result is a number from 0 to 6, where 0 represents Sunday, 1 represents Monday, and so on through Saturday.

Once it knows the weekday of the first day of the month, it knows how many empty spaces must appear before the number 1. For example, if the first day of the month is a Thursday, the Sunday, Monday, Tuesday, and Wednesday positions in the first row must be left blank. If the month starts on a Sunday, no blank spaces are needed.

The code then draws the weekday headers ("S M T W T F S") across the top of the calendar grid. These headers establish the seven columns of the calendar, one column for each day of the week.

With the grid established, the function begins placing date numbers. It starts with day 1 and walks through the calendar one cell at a time, moving left to right across each row and then continuing onto the next row. Before it reaches the cell where day 1 belongs, it skips over the cells that should remain empty. Once it reaches the correct starting position, it begins drawing day numbers sequentially: 1, 2, 3, 4, and so on.

As each day is drawn, the code calculates the exact pixel position for that calendar cell and prints the day number there. Single-digit days are shifted slightly to keep them visually centered relative to two-digit days.

The function continues filling the grid row by row until it reaches the last valid day of the month. The total number of days comes from daysInMonth(), so the calendar automatically stops at 28, 29, 30, or 31 depending on the month and leap-year status.

While drawing the dates, the code also checks whether the day being rendered is today's actual date. If the displayed month and year match the current month and year, and the day number matches today's day of the month, a rectangle is drawn around that cell. This creates the "today" highlight without affecting any other dates.

The end result is a traditional month-view calendar: the month title appears at the top, weekday headers define the columns, enough blank cells are inserted so the first day lands under the correct weekday, the day numbers are filled in sequentially across the grid, and the current date is outlined for emphasis. After the calendar is drawn, the function continues on to render the small digital clock at the bottom of the display.

Practical Applications

- **Wall displays and smart clocks:** A compact monthly calendar gives users an at-a-glance view of upcoming dates without needing to check a phone or computer.
- **Embedded date visualization:** Displaying dates on LED matrices demonstrates how microcontrollers convert raw date information into a familiar calendar layout.
- **Date and time programming:** Calendar rendering is a practical example of working with weekdays, leap years, month lengths, and date calculations.
- **User interface design:** The project illustrates how limited display space can be organized into headers, grids, highlighted elements, and status information.
- **Real-time information systems:** Highlighting the current day and showing a live clock combines static calendar data with continuously updated real-time data.
- **Home automation dashboards:** Calendar widgets are commonly integrated into smart home displays, information kiosks, weather stations, and office status boards.

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 equal to 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.

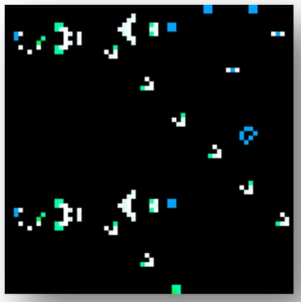
3. Simulate settling grains in the bottom chamber

   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.

Hourglass — Practical Applications

- **Time-based state visualization:** Shows countdowns, timers, and progress in an intuitive display.
- **User Experience metaphors:** Hourglass animations are used in loading indicators and wait states to assure users that progress is being made.
- **Physics simulation:** Sand flow mimics granular material behavior.

Game of Life



- Shows cellular automata 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.

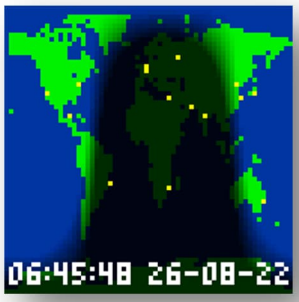
8. Three methuselahs blooming independently; P54 anchors the bottom.
9. Three spaceship lanes with oscillators underneath.
10. Queen Bee in corner; gliders feed a Snark; pulsar stabilizes center.
11. Gun lane with oscillator triad and a drifting methuselah.
12. Chaos basin on left; Snark reflector; HWSS 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 repeating 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 repeated oscillations. Population counts alone miss detecting stagnation since blinkers and pulsars keep the cell count changing.

Practical Applications

- **Cellular automata research:** Used in complexity theory, emergent behavior, and artificial life studies.
- **Biology modeling:** Simplified population dynamics, growth/decay patterns.

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 (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.

Practical Applications

- **Astronomy and Earth-science visualization:** Shows solar position, declination, and illumination in real time.
- **Aviation and maritime navigation:** Pilots and ships use similar maps for planning and situational awareness.
- **Solar energy modeling:** Solar systems use sun position and illumination calculations.
- **Geospatial dashboards**: Geographic information systems use day/night overlays for global monitoring of land, vegetation, and seas.

1 Introduction

This document provides the complete mathematical foundation for the world-map solar renderer used in the LED matrix clock. It describes how pixel coordinates are mapped to geographic latitude and longitude, how solar geometry is computed from UTC time, how illumination is determined using vector dot products, and how twilight blending and color shading are applied. The equations here correspond exactly to the behavior of the rendering engine.

2 Pixel $\leftrightarrow$ Geographic Coordinate Mapping

Longitude from pixel X

$$\text{lon}(x) = \frac{x}{63} \cdot 360^\circ - 180^\circ$$

Latitude from pixel Y

$$\text{lat}(y) = 90^\circ - \frac{y}{63} \cdot 180^\circ$$

Pixel X from longitude

$$x(\text{lon}) = \left\lfloor \frac{\text{lon} + 180^\circ}{360^\circ} \cdot 63 \right\rfloor$$

Pixel Y from latitude

$$y(\text{lat}) = \left\lfloor \frac{90^\circ - \text{lat}}{180^\circ} \cdot 63 \right\rfloor$$

3 Calendar Calculations

Leap year

$$\text{isLeap}(Y) = \begin{cases} \text{true}, & (Y \bmod 4 = 0 \wedge Y \bmod 100 \neq 0) \\ \text{true}, & (Y \bmod 400 = 0) \\ \text{false}, & \text{otherwise} \end{cases}$$

Days in month

$$\text{daysInMonth}(Y, M) = \begin{cases} 29, & M = 2 \wedge \text{isLeap}(Y) \\ \text{mdays}[M-1], & \text{otherwise} \end{cases}$$

Day of year

$$\text{doy}(Y, M, D) = D + \sum_{m=1}^{M-1} \text{daysInMonth}(Y, m)$$

4 Solar Declination

Earth axial tilt

$$\varepsilon = 23.44^\circ$$

Declination

$$\delta(\text{doy}) = \varepsilon \cdot \sin\left(\frac{2\pi}{365}(\text{doy} - 81)\right)$$

Radians

$$\delta_{\text{rad}} = \delta \cdot \frac{\pi}{180}$$

5 NOAA-Based Subsolar Longitude

Fractional year

$$\gamma = \frac{2\pi}{365}\left(\text{doy} - 1 + \frac{H + M/60}{24}\right)$$

Equation of time

$$\text{EOT} = 229.18\,(0.000075 + 0.001868\cos\gamma - 0.032077\sin\gamma - 0.014615\cos(2\gamma) - 0.040849\sin(2\gamma))$$

True solar time

$$\text{TST} = 60H + M + \text{EOT}$$

Hour angle

$$\text{HA} = \frac{\text{TST}}{4} - 180^\circ$$

Subsolar longitude

$$\lambda_{\text{sub}} = -\text{HA}$$

6 Solar Illumination (Dot Product)

Latitude trig

$$\sin\phi = \sin\left(\phi \cdot \frac{\pi}{180}\right), \quad \cos\phi = \cos\left(\phi \cdot \frac{\pi}{180}\right)$$

Longitude difference

$$\Delta\lambda = \lambda_{\text{sub}} - \lambda$$

Radians

$$\Delta\lambda_{\text{rad}} = \Delta\lambda \cdot \frac{\pi}{180}$$

Solar cosine

$$\cos A = \sin\phi \cdot \sin\delta_{\text{rad}} + \cos\phi \cdot \cos\delta_{\text{rad}} \cdot \cos(\Delta\lambda_{\text{rad}})$$

7 Twilight Blend Function

Clamp

$$c = \min(1, \max(-1, \cos A))$$

Zenith angle

$$\theta = \arccos(c)$$

Degrees

$$\theta^\circ = \theta \cdot \frac{180}{\pi}$$

Blend

$$T(\cos A) = \begin{cases} 1.0, & \theta^\circ < 80 \\ 0.18, & \theta^\circ > 100 \\ \max\left(\dfrac{100 - \theta^\circ}{20},\ 0.18\right), & 80 \le \theta^\circ \le 100 \end{cases}$$

8 Color Mapping

Land

$$G_{\text{land}} = \lfloor 180T \rfloor$$

$$(R, G, B) = (0, G_{\text{land}}, 0)$$

Ocean

$$G_{\text{ocean}} = \lfloor 40T \rfloor$$

$$B_{\text{ocean}} = \lfloor 120T \rfloor$$

$$(R, G, B) = (0, G_{\text{ocean}}, B_{\text{ocean}})$$

9 City and Sun Icon Coordinates

City

$$x_c = \left\lfloor \frac{\lambda_c + 180^\circ}{360^\circ} \cdot 63 \right\rfloor + d_x$$

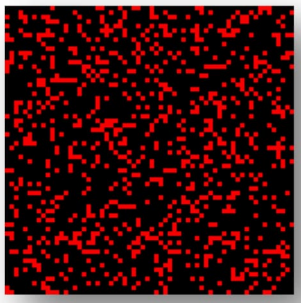
$$y_c = \left\lfloor \frac{90^\circ - \phi_c}{180^\circ} \cdot 63 \right\rfloor + d_y$$

Sun

$$x_{\odot} = \left\lfloor \frac{\lambda_{\text{sub}} + 180^{\circ}}{360^{\circ}} \cdot 63 \right\rfloor$$

$$y_{\odot} = \left\lfloor \frac{90^{\circ} - \delta}{180^{\circ}} \cdot 63 \right\rfloor$$

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.

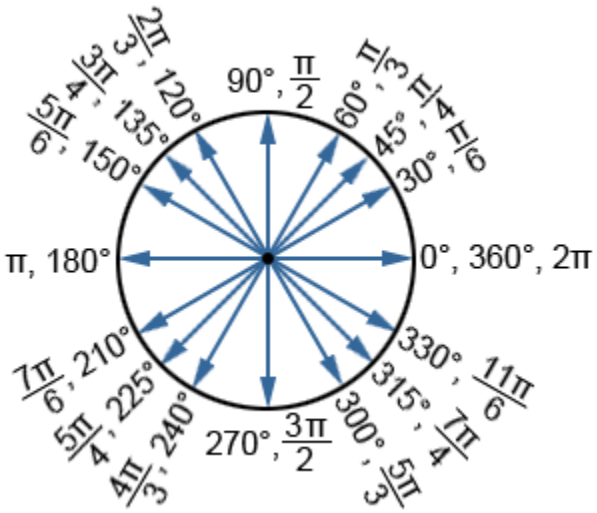
Practical Applications

- **Burn-in prevention:** Subtle motion prevents OLED/LED ghosting and damage.
- **Stress testing:** Continuous animation tests power draw, thermal behavior, and frame pacing.
- **Meditation / focus tools:** Visuals can improve attention and reduce cognitive load.

Analog Clock



- Hand positions are determined by computing the angles in coordinate system where the reference angle is 0 ("straight up"), which is also referred to as "12 o'clock." 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 :**

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

  Which corresponds to “straight up” or “12 o’clock.”

- **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.

Practical Applications

- **Human-friendly time representation**: Analog clocks convey time trends (how much time is left or how much time has passed) better than digital clocks.
- **User interfaces:** Used in dashboards, wearables, and embedded displays.
- **Embedded graphics testing:** Rotating hands test trigonometry, rendering, and animation timing.
- **Teaching time concepts:** Analog clocks are still used in education for time comprehension.

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. Big-O notation is a mathematical shorthand used in programming to describe how an algorithm's runtime or memory requirements grow as the size of the input data increases.
- **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?

Calendar Rendering

Student Exercises

- **First Day Calculation**: Explain how the program determines which day of the week the 1st of the month falls on.
- **Weekday Alignment**: If a month begins on a Thursday, describe how the program positions day 1 in the correct calendar column.
- **Date Placement**: Explain how the calendar fills in dates from 1 through the last day of the month using rows and columns.
- **Highlighting Today**: How does the program identify and highlight the current date while leaving all other dates unchanged?
- **Leap Year Investigation**: Why does February sometimes have 29 days, and how must the calendar algorithm account for this?
- **Manual Simulation**: Draw a calendar for a 30-day month that begins on Tuesday and show where each date would be placed.

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 40 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.

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.

Calendar Rendering

Learning Objectives

1. Understand how software determines the day of the week for a given date.
2. Apply grid-based thinking to map dates into rows and columns.
3. Analyze how algorithms use loops and conditional logic to generate a monthly calendar.
4. Explore real-world challenges such as varying month lengths and leap years.

Discussion Prompts

1. Why must a calendar know the weekday of the first day of the month before it can display any dates?
2. How does the calendar decide where to place day 1, day 2, and all subsequent dates?
3. What problems would occur if every month were assumed to have 30 days?
4. Why are leap years necessary, and how do they affect calendar software?
5. How does highlighting the current date improve the usability of the calendar?
6. What other information could be added to a digital calendar display to make it more useful?

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 hand 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.