Dina Sharifi

Conway's Game of Life

Object-Oriented Programming in C++

ENGS271 Systems Engineering

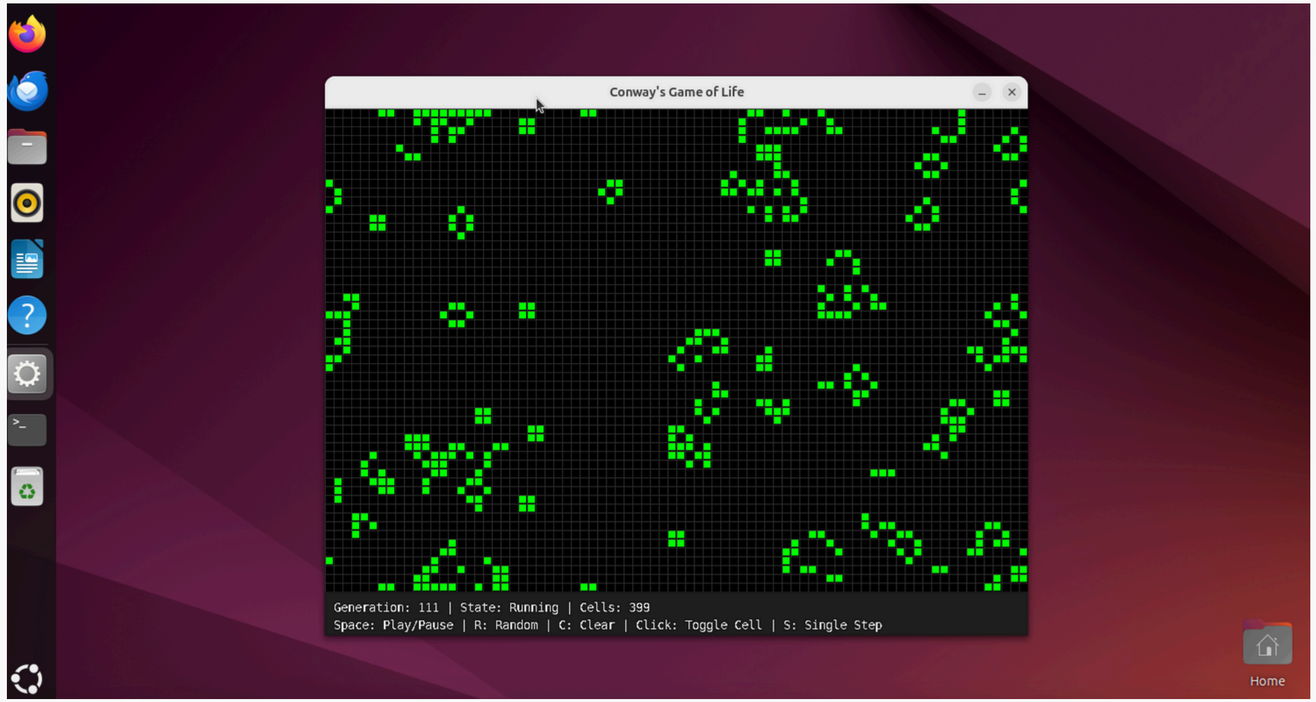
What is Conway's Game of Life

A Cellular Automaton

A zero-player game; once started, patterns evolve automatically.

1) **Basic Concepts**:

- **Grid**: 80x55 board of squares (cells)
- **Cell**:
  - Green = Alive
  - Black = Dead
- **Neighbors:** The 8 squares surrounding any cell
- **Generation:** Each step forward in time



2) **Rules**:

Every cell checks its neighbors:

- **Live** cell with **2-3 live neighbors** = Alive
- **Dead** cell with exactly **3 live. neighbors** = Alive
- All **other** cells = Die or Stay Dead

Why Object-Oriented Programming?

The Problem:

- Complex system with many interacting parts
- Multiple states to manage over time
- Need clean organization for maintenance

The OOP Solution:

- Natural Mapping:

  Grid → Object

  Cells → Data
- Encapsulated game logic and rules
- Clean separation between logic and display

Real-World Analogy:

Think of a car as an object:

Data = fuel level, speed, engine temperature

Actions: start(). stop(), accelerate(), turn()

You use the car without knowing engine mechanics

The four OOP Pillars

Encapsulation, ‘Everything in One Box’

- Our GameOfLife class contains ALL game data and logic
- Private variables: grid, running, generation
- Public methods: update(), render(), randomize()
- Forms self-contained organized code structures

Abstraction, ‘Simple Controls’

- Simplifies complex work into simple buttons,
- update() = one button that handles all math rules
- render() = one button that draws everything, handles all graphics complexity
- Users interact with simple methods, not internal complexity

Inheritance, ‘Family Tree’

- Not used much in this case, but creates new classes from existing ones
- Inherits all existing features and adds new ones
- Could extend GameOfLife for specialized variants

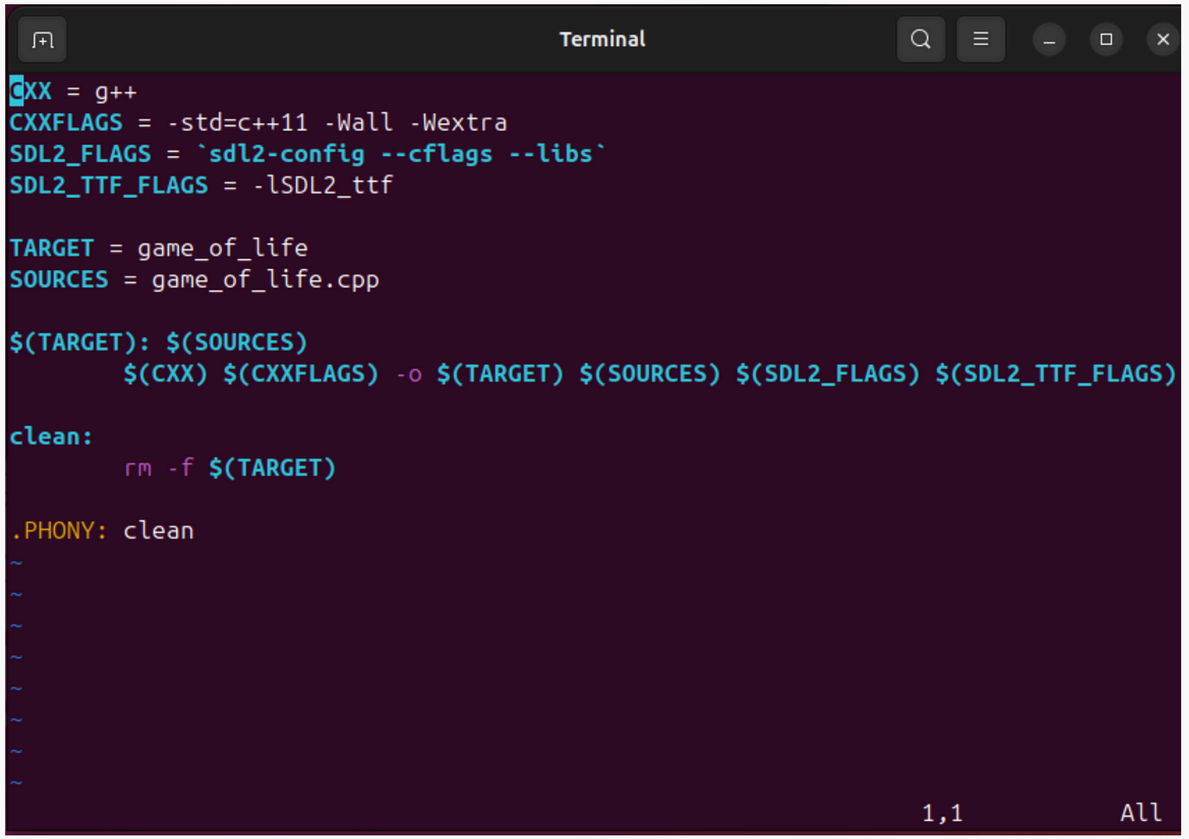
Polymorphism, ‘Same button, different Jobs

- Not used much here either, but provides the same interface with different implementations
- Could allow different rendering styles with the same method calls
- Same method name, with different behaviors

Makefile

What Makefile does:

- Compiles game_of_life.cpp into **executable** game_of_life
- Makes compilation simple: just type ‘make’
- Automatically links **SDL2 libraries** for graphics and text rendering
- Uses C++11 standard with all warning enabled
- ‘make clean’ removes the compiled program

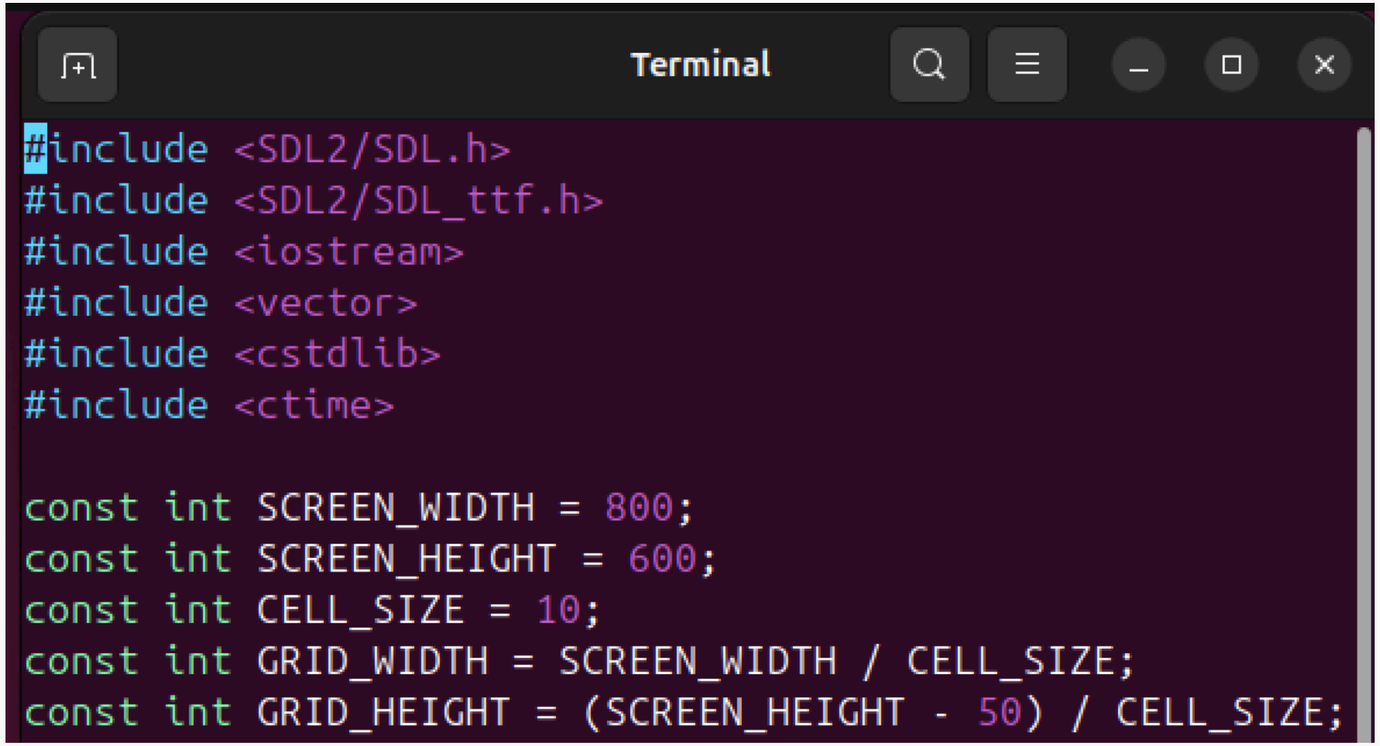


game_of_life.cpp

Header Includes & Game Setup

Libraries Explained:

- **SDL2** handles windows, graphics, and user input
- **vector** stores our grid as a 2D array that can grow/shrink
- **cstdlib** — enables random number generation
- **ctime** — provides time functions for random seeding



game_of_life.cpp

Class Structure

classes:

private: internal data

- Current, Next generations
- Game state (playing/paused)
- Time counter

public: public interface

- Fill cells with random patterns
- Clear all cells
- Click to toggle cell alive/dead
- Count neighbors
- Compute next generations
- Play/pause control
- etc

OOP actions:

- **Encapsulation:** All game data bundled together, marked ‘private’
- **Abstraction:** Clean public methods hide complex internal logic

```
class GameOfLife {
private:
    std::vector<std::vector<bool>> grid;
    std::vector<std::vector<bool>> nextGrid;
    bool running;
    bool stepRequested;
    int generation;

public:
    GameOfLife() : running(false), stepRequested(false), generation(0) {
        grid.resize(GRID_HEIGHT, std::vector<bool>(GRID_WIDTH, false));
        nextGrid.resize(GRID_HEIGHT, std::vector<bool>(GRID_WIDTH, false));
    }

    void randomize() {
        std::srand(std::time(nullptr));
        for (int y = 0; y < GRID_HEIGHT; y++) {
            for (int x = 0; x < GRID_WIDTH; x++) {
                grid[y][x] = (std::rand() % 4) == 0;
            }
        }
        generation = 0;
    }

    void clear() {
        for (int y = 0; y < GRID_HEIGHT; y++) {
            for (int x = 0; x < GRID_WIDTH; x++) {
                grid[y][x] = false;
            }
        }
        generation = 0;
    }
```

game_of_life.cpp

Constructor & Initialization

Blocks

Constructer GameOfLife():

Creates a new GameOfLife object

Sets initial state, paused at 0 generation

Creates two (current and next generation) 80x55 grids

All cells start dead (false)

randomize():

Fills the grid with random living cells

Each cell has 25% chance to be alive

Resets generation counter to 0

Creates a new random starting pattern

Clear():

Kills all cells in the grid

Sets every cell to dead (false)

Resets generation counter to 0

Creates completely blank canvas

Boolean Values Explained

true = yes/on/alive

false = no/off/dead

In our case:

grid[y][x] = **true** means cell at (x,y) is **alive**

grid[y][x] = **false** means cell at (x,y) is **dead**

Why this matters

They provide clean, simple interfaces (abstraction) for complex operations.

Users don't need to know 'how' the grids work internally they can just call randomize() or clear(),

```
class GameOfLife {
private:
    std::vector<std::vector<bool>> grid;
    std::vector<std::vector<bool>> nextGrid;
    bool running;
    bool stepRequested;
    int generation;

public:
    GameOfLife() : running(false), stepRequested(false), generation(0) {
        grid.resize(GRID_HEIGHT, std::vector<bool>(GRID_WIDTH, false));
        nextGrid.resize(GRID_HEIGHT, std::vector<bool>(GRID_WIDTH, false));
    }

    void randomize() {
        std::srand(std::time(nullptr));
        for (int y = 0; y < GRID_HEIGHT; y++) {
            for (int x = 0; x < GRID_WIDTH; x++) {
                grid[y][x] = (std::rand() % 4) == 0;
            }
        }
        generation = 0;
    }

    void clear() {
        for (int y = 0; y < GRID_HEIGHT; y++) {
            for (int x = 0; x < GRID_WIDTH; x++) {
                grid[y][x] = false;
            }
        }
        generation = 0;
    }
```

game_of_life.cpp

Core Logic - Neighbor Counting

What it does

Counts how many of the 8 surrounding cells are alive.

Skips the center cell itself.

edges connect to the opposite sides (Toroidal Wrapping)

```
int countAliveNeighbors(int x, int y) {
    int count = 0;
    for (int dy = -1; dy <= 1; dy++) {
        for (int dx = -1; dx <= 1; dx++) {
            if (dx == 0 && dy == 0) continue;

            int nx = (x + dx + GRID_WIDTH) % GRID_WIDTH;
            int ny = (y + dy + GRID_HEIGHT) % GRID_HEIGHT;

            if (grid[ny][nx]) {
                count++;
            }
        }
    }
    return count;
}
```

Toroidal Wrapping:

Cells on the right edge check left edge as neighbors

Cells on the bottom edge check top edge as neighbors

Creates infinite wrapping world

e.g. Cell [79,5] checks [0,4] [0,5] [0,6] as neighbors

game_of_life.cpp

Core Logic - Update Method

‘brain’ of the simulation

Applies Conway’s Game of Life rules to every cell.

Only runs when game is playing or step requested.

For each cell: counts neighbors, applies rules, updates state.

Advances to next generation.

Updates the generation counter.

```
    void update() {
        if (!running && !stepRequested) return;

        for (int y = 0; y < GRID_HEIGHT; y++) {
            for (int x = 0; x < GRID_WIDTH; x++) {
                int aliveNeighbors = countAliveNeighbors(x, y);

                if (grid[y][x]) {
                    nextGrid[y][x] = (aliveNeighbors == 2 || aliveNeighbors
== 3);
                } else {
                    nextGrid[y][x] = (aliveNeighbors == 3);
                }
            }
        }

        for (int y = 0; y < GRID_HEIGHT; y++) {
            for (int x = 0; x < GRID_WIDTH; x++) {
                grid[y][x] = nextGrid[y][x];
            }
        }

        generation++;
        stepRequested = false;
    }
```

Key Rules

Live cell with 2-3 neighbors survives,

Dead cells with exactly 3 neighbors become alive

All other cells die or stay dead.

game_of_life.cpp

User Interaction & Control Methods

toggleCell(x, y)

- Flips cell state between alive/dead when user clicks
- Converts mouse coordinates to grid coordinates
- Provides direct user editing of the simulation

```
void toggleCell(int x, int y) {
    if (x >= 0 && x < GRID_WIDTH && y >= 0 && y < GRID_HEIGHT) {
        grid[y][x] = !grid[y][x];
    }
}
```

step()

- Advances the simulation by one generation
- Uses stepRequested flag to force a single update
- Allows careful observation of pattern evolution

```
void step() {
    stepRequested = true;
    update();
}
```

```
void setRunning(bool run) { running = run; }
bool isRunning() const { return running; }
```

countLivingCells()

- Counts total number of alive cells in the grid
- Loops through every cell and counts true values
- Provides population data for UI display

```
int countLivingCells() {
    int count = 0;
    for (int y = 0; y < GRID_HEIGHT; y++) {
        for (int x = 0; x < GRID_WIDTH; x++) {
            if (grid[y][x]) count++;
        }
    }
    return count;
}
```

setRunning(run)

- Controls play/pause state of the simulation
- Takes boolean parameter: true=play, false=pause
- Simple setter method for the running variable

isRunning()

- Checks if simulation is currently playing or paused
- Returns boolean: true=playing, false=paused
- Used by main loop to decide when to update

game_of_life.cpp

Graphics Rendering - Visualizing the Game

Provides a black background

Draws gray grid lines (every 10 pixels)

```
    void render(SDL_Renderer* renderer, TTF_Font* font) {
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
        SDL_RenderClear(renderer);

        SDL_SetRenderDrawColor(renderer, 50, 50, 50, 255);
        for (int x = 0; x <= GRID_WIDTH; x++) {
            SDL_RenderDrawLine(renderer, x * CELL_SIZE, 0, x * CELL_SIZE, GR
ID_HEIGHT * CELL_SIZE);
        }
        for (int y = 0; y <= GRID_HEIGHT; y++) {
            SDL_RenderDrawLine(renderer, 0, y * CELL_SIZE, GRID_WIDTH * CELL
_SIZE, y * CELL_SIZE);
        }

        SDL_SetRenderDrawColor(renderer, 0, 255, 0, 255);
        for (int y = 0; y < GRID_HEIGHT; y++) {
            for (int x = 0; x < GRID_WIDTH; x++) {
                if (grid[y][x]) {
                    SDL_Rect cellRect = {x * CELL_SIZE + 1, y * CELL_SIZE +
1, CELL_SIZE - 2, CELL_SIZE - 2};
                    SDL_RenderFillRect(renderer, &cellRect);
                }
            }
        }
```

Colors living cells green

Draws cell slightly smaller for border effect

game_of_life.cpp

UI panel and Text Display

UI Panel Creation:

- Creates dark gray panel at the bottom
- Positions it at the bottom 50 pixels of the screen
- Provides background for text display

Status Text

- Shows real-time game information
- Generation counter, play/pause state, living cell count
- Uses ternary operator for dynamic “Running”/”Paused” text

```
        SDL_SetRenderDrawColor(renderer, 30, 30, 30, 255);
        SDL_Rect uiRect = {0, GRID_HEIGHT * CELL_SIZE, SCREEN_WIDTH, 50};
        SDL_RenderFillRect(renderer, &uiRect);

        SDL_Color textColor = {255, 255, 255, 255};
        std::string statusText = "Generation: " + std::to_string(generation)
 +
                                 " | State: " + (running ? "Running" : "Pause
d") +
                                 " | Cells: " + std::to_string(countLivingCel
ls());

        SDL_Surface* textSurface = TTF_RenderText_Solid(font, statusText.c_s
tr(), textColor);
        if (textSurface) {
            SDL_Texture* textTexture = SDL_CreateTextureFromSurface(renderer
, textSurface);
            SDL_Rect textRect = {10, GRID_HEIGHT * CELL_SIZE + 10, textSurfa
ce->w, textSurface->h};
            SDL_RenderCopy(renderer, textTexture, nullptr, &textRect);
            SDL_DestroyTexture(textTexture);
            SDL_FreeSurface(textSurface);
        }

        std::string instructions = "Space: Play/Pause | R: Random | C: Clear
 | Click: Toggle Cell | S: Single Step";
        SDL_Surface* instSurface = TTF_RenderText_Solid(font, instructions.c
_str(), textColor);
                                                          132,1         44%
```

```
        if (instSurface) {
            SDL_Texture* instTexture = SDL_CreateTextureFromSurface(renderer
, instSurface);
            SDL_Rect instRect = {10, GRID_HEIGHT * CELL_SIZE + 30, instSurfa
ce->w, instSurface->h};
            SDL_RenderCopy(renderer, instTexture, nullptr, &instRect);
            SDL_DestroyTexture(instTexture);
            SDL_FreeSurface(instSurface);
        }
    }

    int countLivingCells() {
                                                          141,1         48%
```

Text Rendering

- Converts text string to visual texture
- Positions text on the UI panel
- Handles SDL’s text rendering pipeline

Instructions

- Shows user control guide
- Explains all keyboard shortcuts
- Positioned below status text

game_of_life.cpp

Main Program - Setup & Game Loop

what do these do?

- Initialize all SDL systems (graphics, text)
- Creates an 800x600 window with title
- Sets up rendering context for drawing
- Loads fonts for UI display
- Creates the main GameOfLife object that runs the simulation

Instructions

- Shows user control guide
- Explains all keyboard shortcuts
- Positioned below status text

```
int main(int argc, char* argv[]) {
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        std::cerr << "SDL could not initialize! SDL_Error: " << SDL_GetError
() << std::endl;
        return 1;
    }

    if (TTF_Init() == -1) {
        std::cerr << "SDL_ttf could not initialize! TTF_Error: " << TTF_GetE
rror() << std::endl;
        SDL_Quit();
        return 1;
    }

    SDL_Window* window = SDL_CreateWindow("Conway's Game of Life",
                                          SDL_WINDOWPOS_UNDEFINED,
                                          SDL_WINDOWPOS_UNDEFINED,
                                          SCREEN_WIDTH,
                                          SCREEN_HEIGHT,
                                          SDL_WINDOW_SHOWN);

    if (!window) {
        std::cerr << "Window could not be created! SDL_Error: " << SDL_GetEr
ror() << std::endl;
        TTF_Quit();
        SDL_Quit();
        return 1;
    }

    SDL_Window* window = SDL_CreateWindow("Conway's Game of Life",
                                          SDL_WINDOWPOS_UNDEFINED,
                                          SDL_WINDOWPOS_UNDEFINED,
                                          SCREEN_WIDTH,
                                          SCREEN_HEIGHT,
                                          SDL_WINDOW_SHOWN);

    if (!window) {
        std::cerr << "Window could not be created! SDL_Error: " << SDL_GetEr
ror() << std::endl;
        TTF_Quit();
        SDL_Quit();
        return 1;
    }

    SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACC
ELERATED);
    if (!renderer) {
        std::cerr << "Renderer could not be created! SDL_Error: " << SDL_Get
Error() << std::endl;
        SDL_DestroyWindow(window);
        TTF_Quit();
        SDL_Quit();
        return 1;
    }

    TTF_Font* font = loadFont();

     GameOfLife game;
```

game_of_life.cpp

Main Program - Input Handling & Game Loop

```
bool quit = false;
SDL_Event e;

Uint32 lastUpdate = 0;
const Uint32 UPDATE_INTERVAL = 100;

while (!quit) {
    Uint32 currentTime = SDL_GetTicks();

    while (SDL_PollEvent(&e) != 0) {
        if (e.type == SDL_QUIT) {
            quit = true;
        }
        else if (e.type == SDL_KEYDOWN) {
            switch (e.key.keysym.sym) {
                case SDLK_SPACE:
                    game.setRunning(!game.isRunning());
                    break;
                case SDLK_r:
                    game.randomize();
                    break;
                case SDLK_c:
                    game.clear();
                    break;
                case SDLK_s:
                    game.step();
                    break;
                case SDLK_ESCAPE:
                    quit = true;
```

Terminal

Runs continuous loop until user quits
Handles keyboard/mouse input events

Updates game state at fixed intervals (100ms)
Renders frames at 60FPS
Converts mouse coordinates to grid positions

```
                case SDLK_s:
                    game.step();
                    break;
                case SDLK_ESCAPE:
                    quit = true;
                    break;
            }
        }
        else if (e.type == SDL_MOUSEBUTTONDOWN) {
            if (e.button.button == SDL_BUTTON_LEFT) {
                int mouseX = e.button.x / CELL_SIZE;
                int mouseY = e.button.y / CELL_SIZE;
                game.toggleCell(mouseX, mouseY);
            }
        }
    }

    if (currentTime - lastUpdate > UPDATE_INTERVAL) {
        game.update();
        lastUpdate = currentTime;
    }

    game.render(renderer, font);
    SDL_RenderPresent(renderer);

    SDL_Delay(16);
}
```

game_of_life.cpp

Main Program - Cleanup & OOP Benefits

```
    if (font) TTF_CloseFont(font);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    TTF_Quit();
    SDL_Quit();

    return 0;
}
```

- Properly closes font resources
- Destroys renderer and window objects
- Shuts down SDL and TTF systems
- Returns success code to operating system

In Short,

From Concept to Program

Our complete journey:

1. **Conway's Rules:** simple 3 rules for cell life/death
2. **OOP Design:** GameOfLife class with encapsulated data
3. **Implementation:** Grid management, rule application, rendering
4. **User Interface:** Keyboard/mouse controls and visual display
5. **Build System:** Makefile for easy compilation

Key OOP Benefits Demonstrated

- **Organization:** All game logic contained in one class
- **Maintainability:** Easy to modify rules or rendering separately
- **Reusability:** Game logic independent of graphics library
- **Readability:** Clean structure that matches mental model

Game of Life shows how OOP transforms abstract concepts into organized, maintainable code that models real systems.

Dina Sharifi

Thank you

ENGS271 Systems Engineering