UV Sim – CS 2450

Danielle Paje
Hayden Christenson
Luis Vilchez
Ryan Lindsay

December 5 2025

Table of Contents

Introduction

UVSim is a simulator that runs BasicML programs. It helps students understand how a basic computer works by letting them load, edit, and run machine-level instructions. The program now supports both 4-digit and 6-digit BasicML formats, expanded memory (250 lines), and a user-friendly GUI.

UVSim supports I/O, arithmetic, load/store, branching, file loading, file saving, format conversion, color themes, and strong input validation. Programs can be executed through a Tkinter GUI.

Use Cases

To start, consider the following user stories:

- As a student, I want to run a BasicML program on UVSim so I can practice and see how the instructions work.
- As a professor, I want UVSim to show the result of the program so students can understand how the instructions change the memory and the accumulator.

Use Case #1: READ Input

- Actor: User
- System: UVSim I/O + Memory
- Goal: Store a user-entered number at the target memory location. • Steps:
    1. Prompt user for input.
    2. Validate input (must be between −9999 and +9999).
    3. Store the value at memory[xx].
    4. Advance instruction pointer.

Use Case #2: WRITE Output

- Actor: Processor
- System: UVSim I/O + Memory
- Goal: Display a value from memory.
- Steps:
    1. Fetch value from memory[xx].
    2. Send value to output (console or GUI output box).
    3. Advance instruction pointer.

Use Case #3: LOAD

- Actor: Processor
- System: Memory + Accumulator
- Goal: Load a value from memory into the accumulator. • Steps:
    1. Parse instruction (20xx).
    2. Fetch value from memory[xx].
    3. Copy value into accumulator.
    4. Advance instruction pointer.

Use Case #4: STORE

- Actor: Processor
- System: Memory + Accumulator
- Goal: Save the accumulator into memory.
- Steps:
    1. Parse instruction (21xx).
    2. Copy accumulator value.
    3. Write value to memory[xx].
    4. Advance instruction pointer.

Use Case #5: ADD

- Actor: Processor
- System: ALU + Memory + Accumulator
- Goal: Add a value in memory to the accumulator.
- Steps:
    1. Parse instruction (30xx).
    2. Fetch value from memory[xx].
    3. Add value to accumulator.
    4. Truncate to 4 digits if overflow.
    5. Advance instruction pointer.

Use Case #6: SUBTRACT

- Actor: Processor
- System: ALU + Memory + Accumulator
- Goal: Subtract a value in memory from the accumulator.
- Steps:
    1. Parse instruction (31xx).
    2. Fetch value from memory[xx].
    3. Subtract value from accumulator.
    4. Truncate to 4 digits if overflow.
    5. Advance instruction pointer.

Use Case #7: MULTIPLY

- Actor: Processor
- System: ALU + Memory + Accumulator
- Goal: Multiply accumulator by a memory value.
- Steps:
    1. Parse instruction (33xx).
    2. Fetch value from memory[xx].
    3. Multiply accumulator by the value.
    4. Truncate to 4 digits if overflow.
    5. Advance instruction pointer.

Use Case #8: DIVIDE

- Actor: Processor
- System: ALU + Memory + Accumulator
- Goal: Divide accumulator by a memory value.
- Steps:
    1. Parse instruction (32xx).
    2. Fetch divisor from memory[xx].
    3. If divisor = 0 → raise error.
    4. Otherwise, accumulator = accumulator // divisor.
    5. Advance instruction pointer.

Use Case #9: BRANCH

- Actor: Processor
- System: Instruction Pointer
- Goal: Jump to a memory address unconditionally.
- Steps:
    1. Parse instruction (40xx).
    2. Set instruction pointer = operand.
    3. Continue execution from that address.

Use Case #10: BRANCHNEG

- Actor: Processor
- System: Instruction Pointer + Accumulator
- Goal: Conditionally branch if accumulator is negative.
- Steps:
    1. Parse instruction (41xx).
    2. If accumulator < 0 → set instruction pointer = operand.
    3. Otherwise, increment instruction pointer.

Use Case #11: BRANCHZERO

- Actor: Processor
- System: Instruction Pointer + Accumulator
- Goal: Conditionally branch if accumulator = 0.
- Steps:
    1. Parse instruction (42xx).
    2. If accumulator == 0 → set instruction pointer = operand.
    3. Otherwise, increment instruction pointer.

Use Case #12: HALT

- Actor: Processor
- System: Execution Controller
- Goal: Stop program execution.
- Steps:
    1. Parse instruction (43xx).
    2. Set running = False.
    3. Program terminates.

Use Case #13: Handle Invalid Opcode

- Actor: Processor
- System: Error Handler
- Goal: Gracefully handle unknown instructions.
- Steps:
    1. Detect invalid opcode.
    2. Raise error message (console or GUI popup).
    3. Stop program execution.

Use Case #14: Handle Out-of-Range Input

- Actor: User
- System: Input Validator
- Goal: Ensure entered values are within range.
- Steps:
    1. Prompt user for input.
    2. If value < −9999 or > +9999 → show warning. 3. Ask user to re-enter until valid.
    3. Store valid input in memory.

Use Case #15: Execute Full Program

- Actor: User (initiates), Processor (executes)
- System: Loader + Execution Loop
- Goal: Run a BasicML program to completion.
- Steps:
    1. Load program lines into memory[00–99].
    2. Fetch instruction at instruction pointer.
    3. Decode opcode and operand.
    4. Execute instruction.
    5. Repeat until HALT or error.

Specifications

Use of this program is only functional with the following installed before use:

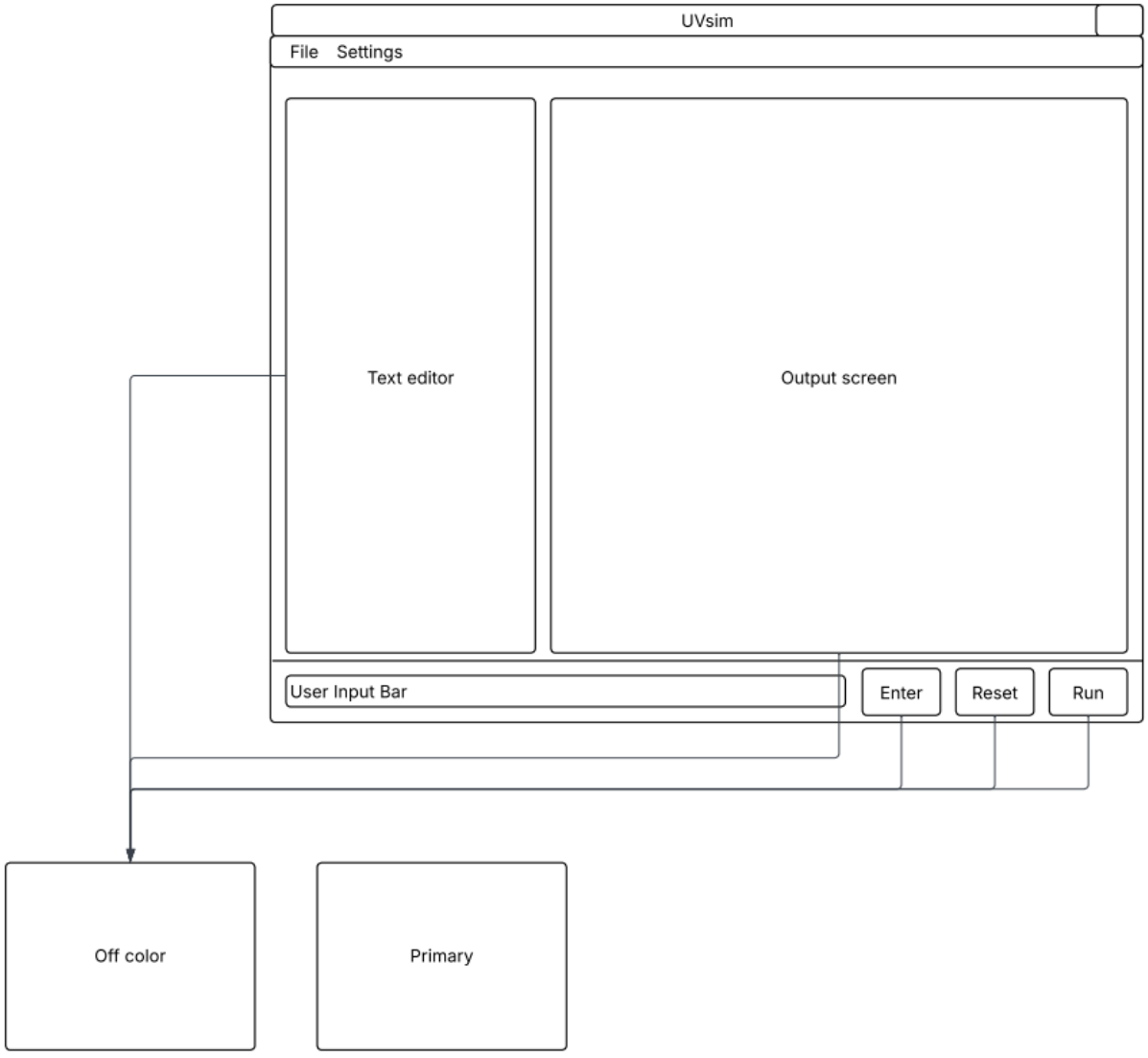
- Python 3.10 or newer
- Tkinter (comes with most Python installs)
- BasicML text files (like Test1.txt or Test2.txt)

Diagrams

Module Descriptions:

- uvsim.py – Implements the core simulation logic / backend functionality (e.g. the “engine” of the application), including simulation models, state updates, data processing, and the underlying non-UI computations.
- gui.py – Defines the graphical user interface: the windows, dialogs, controls, and user interactions that allow users to drive the simulation visually, configure parameters, and view outputs in a user-friendly way.
- main.py – This serves as the entry point for the application. it coordinates initialization, sets up configuration (from config.json), instantiates core simulation and GUI modules, and launches the main workflow.
- config.json –Encapsulates configurable parameters and settings for the application (e.g. simulation parameters, default values, UI options), enabling flexibility and external configuration without code changes.

GUI Wireframe:



Unit Tests

- test_read_test1 – Tests READ method for Test1.txt. Passes if memory contains correct values.
- test_write_test1 – Tests WRITE method for Test1.txt. Passes if last output matches expected memory value.
- test_read_test2 – Tests READ method for Test2.txt. Passes if memory contains correct values.
- test_write_test2 – Tests WRITE method for Test2.txt. Passes if last output matches expected memory value
- test_load – Tests LOAD method. Passes if accumulator matches memory[0].
- test_store – Tests STORE method. Passes if memory[1] matches accumulator.
- test_add – Tests ADD method. Passes if accumlator == 15.
- test_subtract – Tests SUBTRACT method. Passes if accumlator == 5
- test_divide – Tests DIVIDE method. Passes if accumlator == 3
- test_divide_by_zero – Tests division by ZERO. Passes if an exception is thrown.
- test_multiply – Tests MULTIPLY method. Passes if accumulator == 12.
- test_branch – Tests BRANCH method. Passes if accumulator loads the correct memory value.
- test_branchneg – Tests BRANCHNEG method. Passes if accumulator is updated after a branch occurs.
- test_branchzero – Tests BRANCHZERO method. Passes if accumulator is updated after a branch occurs.
- test_halt – Tests HALT method. Passes if halted after execution and accumulator remains unchanged.
- test_branchneg_no_branch – Tests that a negative-branch instruction does not branch when accumulator is positive. Passes if execution continues without branching.
- test_branchzero_no_branch – Tests that a zero-branch instruction does not branch when accumulator is nonzero. Passes if execution continues without branching.
- test_add_overflow – Tests that addition handles overflow correctly. Passes if value truncated correctly to word size.
- test_multiply_overflow – Tests that multiplication handles overflow correctly. Passes if value truncated correctly to word size.

Application Instructions

How to Install and Run

Make sure you have Python 3.10 or newer installed

Run the GUI

- python3 main.py

Run a BasicML program in the terminal

- python3 main.py Test1.txt

Recommendation folder structure

- main.py
- gui.py
- uvsim.py
- config.json
- tests/
  - ⇒ Test1.txt
  - ⇒ Test2.txt
  - ⇒ InvalidFileTest.txt
  - ⇒ Invalid0pcode.txt

Key Features

- Supports 250 lines of memory (000–249)
- Supports 4-digit or 6-digit BasicML formats
- Automatic format detection
- Built-in 4-digit → 6-digit conversion tool

- Input, output, load/store, arithmetic, and branching instructions
- Full Tkinter GUI with editable instruction window
- Custom color themes saved to config.json
- Strong error handling for invalid opcodes, file formats, and input
- Compatible with all provided test files (Test1, Test2, etc.)
- Includes unit tests for arithmetic, branching, load/store, and I/O

How the GUI Works

The UVSim window is simple and easy to use.

Menu	Function
File → Open	Opens a .txt BasicML file from any folder. The file's contents appear in the large center text box (instruction editor).
File → Save / Save As	Saves your current program. "Save As" lets you choose a new name or location.
File → Close	Exits the program safely.
Settings → Change Color Scheme	Opens the color picker so you can choose a primary and off color for the app. Your colors are saved to config.json.
Settings → Reset to Default Colors	Resets the interface to UVU's green (#4C721D) and white (#FFFFFF) color scheme.
Instruction Editor (Center Box)	The main text area where all BasicML instructions are shown. You can type new lines, edit commands, or delete lines here.

Output Panel (Right Side)	Displays messages, errors, and program output during and after execution.
Input Box (Bottom)	Used when a program asks for input (READ command). Type a number and press Enter or click Enter Button..
Run Button	Runs the program loaded in the Instruction Editor. Shows results in the Output Panel.
Reset Button	Clears both the Instruction Editor and Output Panel so you can start fresh.
Enter Button	Sends your input value to the program. Works the same as pressing the Enter key.

Output Box:

The Output Panel shows all printed results from WRITE commands. It also displays error messages and any status updates while the program is running.

If your test files are in a different folder, you can either move them next to main.py or open them manually using **File → Open**.

Output Panel (Right Side)

- Shows the results of WRITE commands
- Shows errors and status messages

Input Box (Bottom)

- Appears when a READ command needs your input
- Type a number and press Enter or the Enter button

Run / Reset / Enter Buttons

- Run: loads and runs your program

- Reset: clears the editor and output boxes
- Enter: sends your input to the running program

How to Run UVSim

1. Open your terminal and run:

   - python3 main.py

2. The GUI window will open.
3. Click Open File and pick a BasicML text file (like Test1.txt).
4. Your program will appear in the Instruction Editor.
5. Click **Run** to start executing the program.
6. If the program uses a **READ** instruction, type your number in the input box and press **Enter** or click the **Enter Button**.
7. You will see the program's output on the right side in the Output Panel.
8. Click **Reset** to clear the editor and output so you can start over.
9. Click **File → Close** when you are done to safely exit the program.

BasicML Commands

Each instruction takes one line in the text file and is read from top to bottom.

Code	Command	What it does
10xx / 010xxx	READ	Ask the user for a number and save it in memory[xx]
11xx / 011xxx	WRITE	Print value from memory[xx]

20xx / 020xxx	LOAD	Load memory[xx] into the accumulator
21xx / 021xxx	STORE	Save the accumulator into memory[xx]
30xx / 030xxx	ADD	Add memory[xx] to the accumulator (truncated if too big)
31xx / 031xxx	SUBTRACT	Subtract memory[xx] from the accumulator (truncated if too big)
32xx / 032xxx	DIVIDE	Divide accumulator by memory[xx] (error if divisor = 0)
33xx / 033xx	MULTIPLY	Multiply accumulator by memory[xx] (truncated if too big)
40xx / 040xxx	BRANCH	Jump to memory[xx]
41xx / 041xxx	BRANCHNEG	Jump to memory[xx] if accumulator < 0
42xx / 042xxx	BRANCHZERO	Jump to memory[xx] if accumulator == 0
43xx / 043xxx	HALT	Stop the program

Works for both **4-digit** and **6-digit** formats.

Supported Formats

UVSim supports 4-digits and 6-digits BasicML formats.

4-digit example

1007 READ to address 07 2109 STORE accumulator into address 09

6-digit example

010007 READ to address 007 021009 STORE accumulator into address 009

4-digit → 6-digit Conversion Rules

- If first two digits are opcode → becomes 3-digit opcode + 3-digit address
- If not an opcode → treat as numeric value

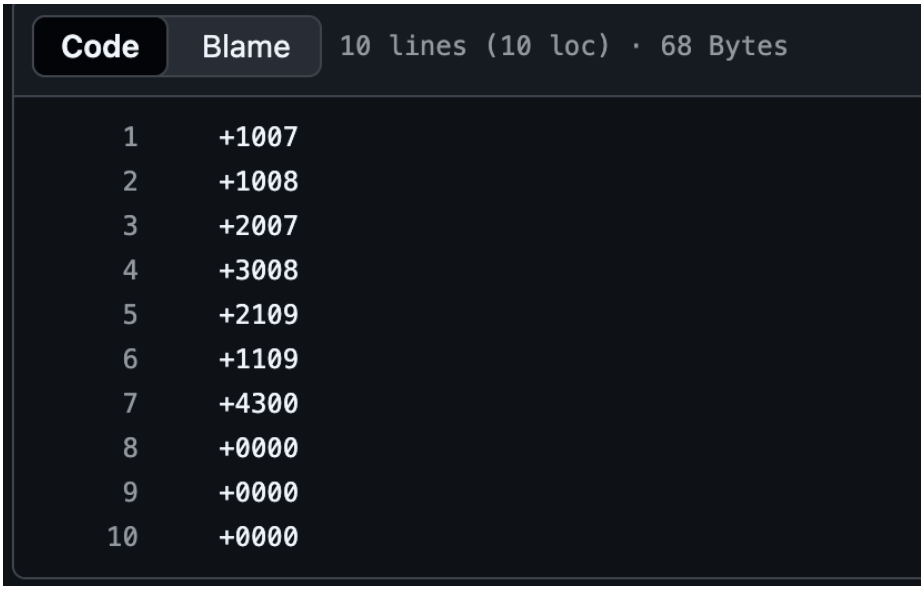
Error Handling

UVSim includes error handling so it doesn't crash when something goes wrong.

Error Type	What Happens
Malformed File	UVSim shows an error message
Mixed Format	File is rejected
Address > 249	UVSim stops execution
Too Many Lines	File cannot be loaded
Division by Zero	UVSim stops safely
Overflow	UVSim keeps the last digits
Invalid Input	UVSim asks again

Example Programs

Test1.txt



This program asks for two numbers, adds them, and prints the result.

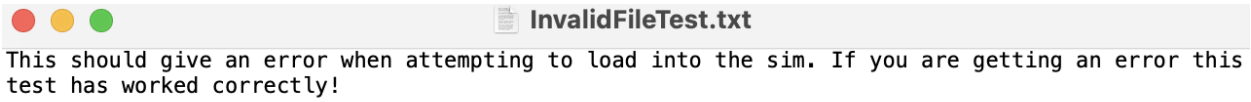
Test2.txt

Code Blame 11 lines (11 loc) · 75 Bytes

```
+1009
+1010
+2009
+3110
+4107
+1109
+4300
+1110
+4300
+0000
+0000
```

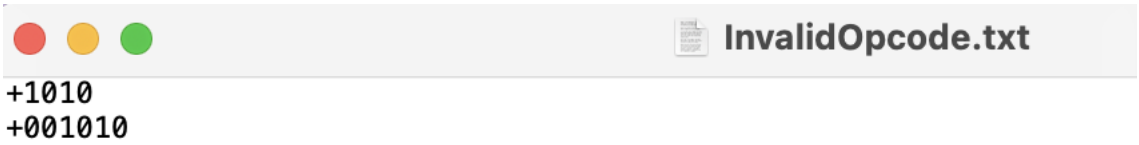
This program asks for two numbers and prints the bigger one.

InvalidFileTest.txt

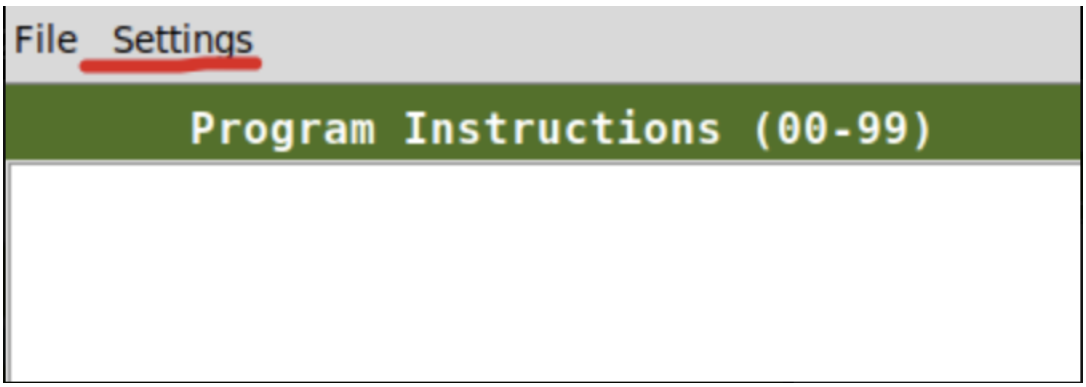


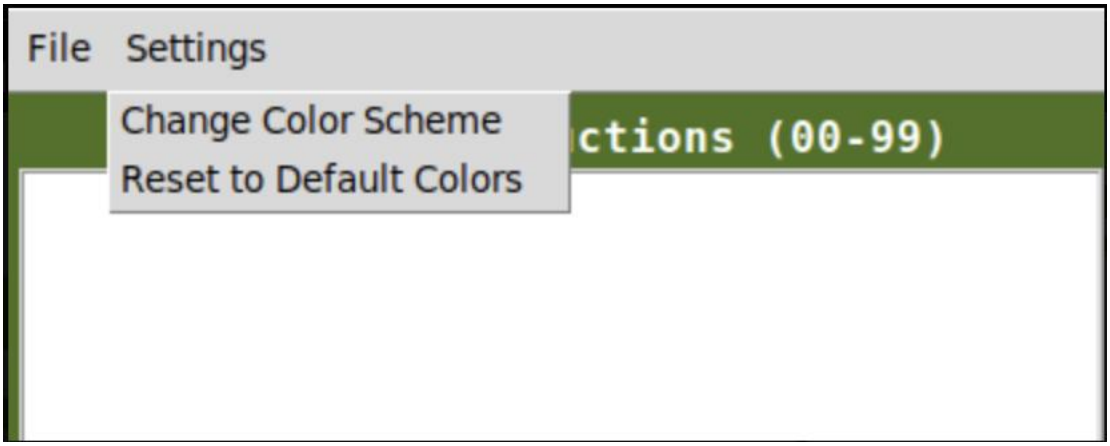
Used to test UVSim's ability to handle broken or incomplete files.

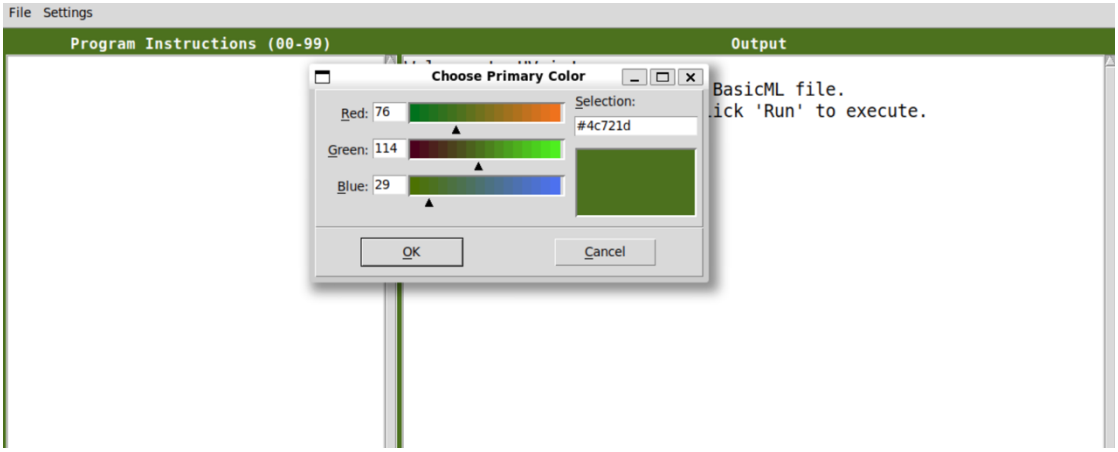
InvalidOpcode.t



Color Customization







Color Settings

- UVSim allows you to change the colors of the interface.
- You can find these options under **Settings** in the top menu bar.
    - Settings Menu
        - **Change Color Scheme** – lets you choose a new primary or off-color
        - **Reset to Default Colors** – returns UVSim to UVU's green (#4C721D) and white

Choosing a Color

- When you click **Change Color Scheme**, a color picker window appears.
- You can adjust:
    - **Red**
    - **Green**
    - **Blue**
- You can also type in a **hex value** (like #4C721D).
- A preview box shows what the color will look like.

Click **OK** to apply the color, or **Cancel** to exit without saving.

How Colors Are Saved

Your chosen colors are saved in a file named **config.json**.

The next time you open UVSim, it will load your custom colors automatically.

Buttons (Left → Right order)

On the bottom of the window, the buttons appear in this order (left to right):

1. **Enter**
2. **Reset**
3. **Run**

Editing and Saving Files

- You can edit instructions directly in the GUI
- Save and Save As allow exporting anywhere on your computer

Future Road Map

First and foremost, the plan for the near future (1-3 months) is to address and improve these known limitations:

- Only one file can be opened at a time
- Multi-file tabs planned for future updates
- Some copy/paste operations may be limited

Building on the current limitations, future versions of the application could introduce a more robust and flexible file-handling system, including support for multiple open files, persistent tab states, and smooth transitions between documents. Enhancing copy/paste behavior across all UI components would improve usability and bring the editor closer to a full-featured professional tool.

Additional improvements could include UI refinements such as customizable themes, keyboard shortcuts, and improved accessibility options. Performance could also be enhanced through incremental rendering, smarter background processing, and stronger error-handling mechanisms. Longer-term goals may involve integrating cloud-based storage, collaboration features, export options, or a plug-in system that allows users to extend functionality over time.

Updates Since Last Milestone

- Fixes to README file have been made.
- Use cases have had some tweaking.
- The focus on 4-digits has been shifted to the requested 6-digit focus.
- Team has met three times to make adjustments.
- Project video has been completed.
- Project report has been completed.