Your project has two main parts that need to communicate with each other:

1. **The TeenyAT Program (The "Game Brain" 🧠):** This is the code you will write in **C and Assembly** that runs *inside* the TeenyAT virtual machine. Its only job is to run the game's logic: read player input and "tell" the hardware what to do. It interacts with the outside world by reading from and writing to specific memory addresses (peripherals).
2. **The C++ Host Application (The "Hardware" & "Renderer" 🖥️):** This is the main.cpp file. It runs the TeenyAT virtual machine and simulates the game's hardware. It is responsible for:
   * Drawing everything on the screen (the player, the level, etc.).
   * Detecting real keyboard presses from the user.
   * Implementing the physics engine (gravity, movement).
   * Managing the game world (collisions, level layout).

The two parts communicate through the bus\_read and bus\_write functions. The TeenyAT program writes to an address (e.g., 0x9016 to jump), and your bus\_write function in main.cpp catches it and makes the character jump on screen.

## **Phase 1: Setting Up the Host Application**

Your first step is to turn main.cpp from a simple console application into a graphical game window that can handle player input and simulate your game's peripherals.

### **Step 1: Implement a Graphics Layer**

Your main.cpp currently uses rogueutil to print characters to a console. You need to replace this with a simple 2D graphics library to draw sprites and tiles.

* **Action:** Choose a simple graphics library like **SFML**, **SDL**, or use the **TIGR** library you mentioned as a reference to build your own drawing functions.

**Goal:** Create functions to:

* Open a graphical window.
* Draw a pixel sprite (your player) at a specific (x, y) coordinate.
* Draw your level using tiles.
* The for(;;) loop in main.cpp just runs the TeenyAT clock as fast as possible. You need a standard game loop that manages timing, updates, and rendering.
  + Open a graphical window.
  + Draw a pixel sprite (your player) at a specific (x, y) coordinate.
  + Draw your level using tiles.

### **Step 2: Create a Proper Game Loop**

* **Action:** In main.cpp, restructure your main function into a game loop.

**Code Structure:**C++  
// In main.cpp

while (window.isOpen()) {

// 1. Handle Events (like closing the window, keyboard input)

// 2. Clock the TeenyAT CPU a number of times

tny\_clock(&t);

// 3. Update Game State (apply physics, check collisions)

// 4. Render Graphics (clear screen, draw level, draw player)

}

### **Step 3: Implement the bus\_read Callback for Keyboard Input**

Your TeenyAT program needs to read from address 0x4000 to get key presses. You must provide this data from your C++ host.

* **Action:**
  1. In your C++ game loop, check for keyboard presses (e.g., A, D, W) and store their state (pressed or not pressed).
  2. Create a bus\_read function (the counterpart to your existing bus\_write).
  3. When the addr in bus\_read is 0x4000 (KEY\_INPUT), package your key states into a single tny\_uword bitmask as specified in your slides (Bit 0 for Left, Bit 1 for Right, Bit 2 for Jump).
  4. Pass this tny\_uword back to the TeenyAT through the data parameter of the bus\_read function.

## **Phase 2: Programming the TeenyAT Game Logic**

Now, you'll write the Assembly/C code that runs on the TeenyAT. This code will read the inputs you just prepared and trigger the movement peripherals.

### **Step 1: Create the Main Loop in Assembly/C**

This will be the core logic of your game running on the virtual machine.

* **Action:** Create a simple infinite loop in your TeenyAT assembly file.
* **Goal:** This loop will continuously check for input and update the game.

### **Step 2: Read from the Keyboard Peripheral**

Inside the loop, you need to read the button states from the address you set up in Phase 1.

* **Action:** Use the lod instruction to load the value from the BUTTONS address (0x4000) into a register (e.g., rA). This will give you the bitmask of currently pressed keys.

### **Step 3: Check for Specific Key Presses**

You need to determine which action the player wants to take based on the bits in the register.

* **Action:** Use bitwise and and comparison cmp instructions to check if specific bits are set (e.g., bit 2 for jump). Your pseudo-code in the presentation is the perfect guide for this .

### **Step 4: Trigger Movement Peripherals**

If a key is pressed, write to the corresponding address to tell the C++ host to perform an action.

* **Action:** Use the str [ADDRESS], rZ instruction to "strobe" the write-only peripherals.
  + If the "jump" bit is set, write to MOVE\_N (0x9016).
  + If the "left" bit is set, write to MOVE\_W (0x9014).
  + If the "right" bit is set, write to MOVE\_E (0x9017).

## **Phase 3: Building the Physics and World**

This brings you back to main.cpp. The TeenyAT program is now telling you *when* the player wants to move; it's your job to actually make it happen on screen with physics.

### **Step 1: Create a Player Object/Struct**

You need variables in main.cpp to keep track of the player's state.

* **Action:** Create a struct or class in C++ to hold the player's properties:
  + float x, y; // Position
  + float velocityX, velocityY; // Velocity

### **Step 2: Handle Movement Writes in bus\_write**

Expand your bus\_write function to handle the movement strobes from the TeenyAT.

* **Action:** In your switch(addr) statement, add cases for MOVE\_W, MOVE\_E, and MOVE\_N.
* **Logic:**
  + case MOVE\_W: Apply a westward acceleration (e.g., player.velocityX -= 5.0f;).
  + case MOVE\_E: Apply an eastward acceleration (e.g., player.velocityX += 5.0f;).
  + case MOVE\_N: Apply an instantaneous upward impulse for the jump (e.g., player.velocityY = -15.0f;).

### **Step 3: Implement the Physics Engine**

In your main C++ game loop, you need to update the player's position based on their velocity and other forces.

* **Action:** In the "Update Game State" part of your loop, add the following logic:
  1. **Apply Gravity:** player.velocityY += GRAVITY \* deltaTime;
  2. **Apply Friction/Drag:** player.velocityX \*= 0.9f;
  3. **Update Position:** player.x += player.velocityX \* deltaTime; and player.y += player.velocityY \* deltaTime; *(Note: deltaTime is the time elapsed since the last frame. It ensures smooth movement regardless of framerate.)*

### **Step 4: Create the Level and Handle Collisions**

The player needs a world to interact with.

* **Action:**
  1. Create a 2D array or vector to represent your static level, as planned. Each element will represent a tile (e.g., 0 for air, 1 for solid ground).
  2. In your physics update, after calculating the player's new potential position, check if that new position overlaps with a solid tile in your level array.
  3. If a collision occurs, stop the player's movement in that direction (e.g., if they hit the ground, set player.velocityY = 0; and snap their y-position to the top of the tile).

## **Phase 4: Bringing It All Together**

This is the final stage where you add the remaining game elements and polish the experience.

* **Rendering:** In the "Render Graphics" part of your C++ game loop, draw the level tiles based on your 2D array and then draw the player sprite at its final, updated player.x and player.y position.
* **Win Condition:** Add the flagpole to your level map. In your update logic, simply check if the player's coordinates overlap with the flagpole's coordinates to trigger a win state.
* **Optional Features:** To implement SPRITE\_SET, have your TeenyAT program write a number to 0x9024. Your bus\_write in C++ can catch this and change the texture it uses to draw the player.

By following these phases, you can systematically build your game, ensuring that the TeenyAT logic and the C++ host work together correctly. Good luck!