CSOPESY Lecture – I/O, Display Interfaces, Keyboard Polling

Instructor: Neil Patrick Del Gallego, Ph.D.

Goal:

- Design a real-time application: Create a marquee console that supports the interleaving of polling events and CPU events.

Features to design in our CSOPESY emulator

- A means to support keyboard input, both real-time and event-driven.
  - Real-time = The user can always type characters, and this displays on-screen immediately. E.g. Word processor programs.
  - Event-driven = wait for an event, such as an enter command, and then perform an associated operation.
- How an application must behave under real-time/event-driven conditions.
  - Real-time = screen always refreshes even if there's no event/user input.
    - Sample applications: games and interactive applications like streaming websites.
    - Sample OSes: Modern OS like Windows are considered real-time.
  - Event-driven = screen refreshes after user input.
    - Sample applications:
    - Sample OSes: console/command line interfaces. However, there are means to support real-time switching. Example: Concept of tmux**,** pip download interface.

Examples of real-time applications

```cpp
#include <iostream>
#include <conio.h>
#include <Windows.h> // For Sleep()

void keyboardInterruptHandler() {
    if (_kbhit()) {
        char key = _getch();
        std::cout << "Key pressed: " << key << std::endl;
    }
}

int main() {
    while (true) {
        // Simulate other program logic
        Sleep(100); // Sleep for 100 milliseconds

        // Check for keyboard input through the interrupt handler
        keyboardInterruptHandler();
    }

    return 0;
}
```

The example above shows a real-time keyboard polling mechanism.

Another example in CSOPESY emulator, where it shows the main loop. A real-time application would typically have a while loop wherein the application exits the loop when an exit/terminating sequence is triggered.

```cpp
int main()
{
    InputManager::initialize();
    FileSystem::initialize();
    // FileSystem::getInstance()->test_createRandomFiles(1000);
    // FileSystem::getInstance()->saveFileSystem();
    FileSystem::getInstance()->loadFileSystem();
    ConsoleManager::initialize();
    MessageBuffer::initialize();
    ResourceEmulator::initialize();
    MemoryManager::initialize();

    bool running = true;
    while(running)
    {
        ConsoleManager::getInstance()->process();
        ConsoleManager::getInstance()->drawConsole();

        running = ConsoleManager::getInstance()->isRunning();
    }

    // MemoryManager::test_MemoryAllocation();

    MemoryManager::destroy();
    ResourceEmulator::destroy();
    MessageBuffer::destroy();
    ConsoleManager::destroy();
    InputManager::destroy();
    return 0;
}
```

Another example in a game application where real-time interaction is highly valued:

```cpp
// Game Loop
void RunGameLoop(IGameEvent& gameEvent) {
    const float targetFPS = 60.0f;
    const float frameTime = 1.0f / targetFPS;

    while (true) {
        // Handle user input
        if (_kbhit()) {
            char key = _getch();
            gameEvent.OnKeyPressed(key);
        }

        // Update game logic
        gameEvent.OnUpdate(frameTime);

        // Render game state
        gameEvent.OnRender();

        // Sleep to achieve a constant frame rate
        Sleep(static_cast<DWORD>(frameTime * 1000.0f));
    }
}

int main() {
    GameEventHandler gameHandler;

    // Start the game loop
    RunGameLoop(gameHandler);

    return 0;
}
```

Examples of event-driven applications
Event-driven applications are those in which the flow of the program is determined by events such as user actions, sensor outputs, or messages from other programs

- Graphical User Interface (GUI) Applications
    - Example: A simple text editor
    - Events: Mouse clicks, keypresses, menu selections
    - Usage: When the user interacts with the GUI by clicking buttons, typing text, or choosing options from menus, events are triggered, and the application responds accordingly.

The code below shows Qt framework in C++, designed for event-driven applications:
https://www.qt.io/product/framework

```cpp
//The code below shows Qt framework in C++, designed for event-driven applications.
#include <QApplication>
#include <QTextEdit>
#include <QMenuBar>
#include <QMenu>
#include <QAction>
#include <QFileDialog>

class TextEditor : public QTextEdit {
    Q_OBJECT

public:
    TextEditor(QWidget *parent = nullptr) : QTextEdit(parent) {
        createActions();
        createMenus();
    }

private:
    void newFile() {
        clear();
    }

    void openFile() {
        //open file
    }

    void saveFile() {
        //save file
    }

private:
    void createActions() {
        newAction = new QAction(tr("&New"), this);
        newAction->setShortcut(QKeySequence::New);
        connect(newAction, &QAction::triggered, this, &TextEditor::newFile);

        openAction = new QAction(tr("&Open"), this);
        openAction->setShortcut(QKeySequence::Open);
        connect(openAction, &QAction::triggered, this, &TextEditor::openFile);

        saveAction = new QAction(tr("&Save"), this);
        saveAction->setShortcut(QKeySequence::Save);
        connect(saveAction, &QAction::triggered, this, &TextEditor::saveFile);
    }

    void createMenus() {
        fileMenu = menuBar()->addMenu(tr("&File"));
        fileMenu->addAction(newAction);
        fileMenu->addAction(openAction);
        fileMenu->addAction(saveAction);
    }

    QMenu *fileMenu;
    QAction *newAction;
    QAction *openAction;
    QAction *saveAction;
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    TextEditor textEditor;
    textEditor.show();

    return app.exec();
}
```

Web Applications

- Example: An online shopping website
- Events: Button clicks, form submissions, AJAX requests
- Usage: Users interact with the website by clicking buttons to add items to their cart, submitting forms for payment, and the application responds to these events.

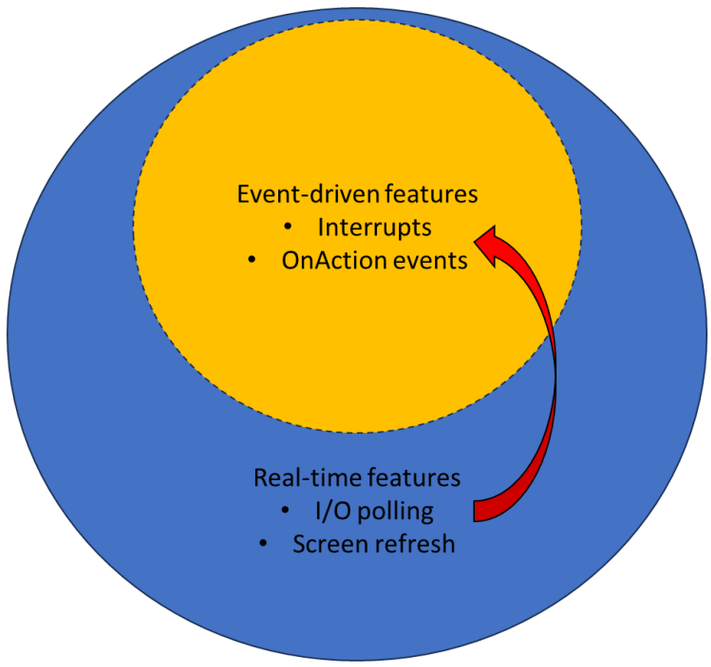
Mobile Applications:

- Example: A social media app
- Events: Taps, swipes, device orientation changes
- Usage: Users interact with the app by tapping on posts, swiping to navigate, and the application responds by displaying relevant content.

Networking Applications:

- Example: Chat application
- Events: Receiving a message, connection status changes
- Usage: When a user sends a message, or when the application receives a message from another user, events are triggered, and the application updates the chat interface.

Observation: All event-driven applications, are derived from real-time applications. All event-driven applications still require functionality that needs to run in real-time.



Family of implementation features

Keyboard polling example shown previously:
Real-time feature → keyboard polling,
Event-driven feature → IKeyboardEvent implementation

```cpp
//Here is an example of a simple C++ application that combines real-time and event-driven handling:
#include <conio.h> // For _kbhit() and _getch()

// Interface for Keyboard Events
class IKeyboardEvent {
public:
    virtual void OnKeyDown(char key) = 0;
    virtual void OnKeyUp(char key) = 0;
};
// Implementation of IKeyboardEvent
class KeyboardEventHandler : public IKeyboardEvent {
public:
    void OnKeyDown(char key) override {
        std::cout << "Key Down: " << key << std::endl;
    }

    void OnKeyUp(char key) override {
        std::cout << "Key Up: " << key << std::endl;
    }
};
// Keyboard Polling
void PollKeyboard(IKeyboardEvent& keyboardEvent) {
    while (true) {
        if (_kbhit()) { // Check if a key has been pressed
            char key = _getch(); // Get the pressed key

            // Check if it's a key down or key up event
            if (GetAsyncKeyState(key) & 0x8000) {
                keyboardEvent.OnKeyDown(key);
            } else {
                keyboardEvent.OnKeyUp(key);
            }
        }

        // Other program logic can continue here
    }
}

int main() {
    KeyboardEventHandler keyboardHandler;
    // Start keyboard polling
    PollKeyboard(keyboardHandler);
    return 0;
}
```

- The IKeyboardEvent class defines an interface with OnKeyDown and OnKeyUp virtual functions.
- The KeyboardEventHandler class implements this interface, printing messages to the console when a key is pressed or released.
- The PollKeyboard function continuously checks for key presses using _kbhit() and retrieves the key using _getch().
- Depending on the state of the key using GetAsyncKeyState, it calls the appropriate event handler.

DISPLAY INTERFACE

- How should we draw on the screen? The goal is to support **real-time handling of I/O.**
- Thus, for every cycle, we must have mechanisms to refresh the screen per frame, while polling for I/O events.

```
bool running = true;
while(running)
{
    ConsoleManager::getInstance()->process();
    ConsoleManager::getInstance()->drawConsole();

    running = ConsoleManager::getInstance()->isRunning();
}
```

- process() should contain handling of logic and other non-drawing operations.
- DrawConsole() refreshes the screen with the updated information.

Each console window must have the following implementations:

```
class AConsole
{
public:
    typedef std::string String;
    AConsole(String name);
    ~AConsole() = default;

    String getName();
    virtual void onEnabled() = 0;
    virtual void display() = 0;
    virtual void process() = 0;

    String name;
    friend class ConsoleManager;
};
```

- onEnabled() is called every time the console first draws to the screen.
- process() is called every frame
- display() is called every frame

Temporary and easy solution to creating an event-driven console: Put the drawing in the process() method, and use std::in to wait for user input in order to pause the screen refresh.

DEMO: show the implementation of the following from the CSOPESY-Emulator code:

- AConsole class
- ConsoleManager

Activities

- Create a means to support multi-window **real-time** drawing of screens.
    - Using the screen -s <name> format, create custom screens: screen1, screen2, screen3, where each screen is created by an individual in your student group.
    - Study the Windows console API → how to move your cursor, set console window size, and set font color.
    - Continue the "marquee" command wherein it moves the marquee screen once recognized. The marquee screen simply displays "Hello, marquee", and then the user can type "exit" to go back to the main menu.
    - Create your own ASCII-animated screen wherein the animation speed is controlled by a refresh rate.

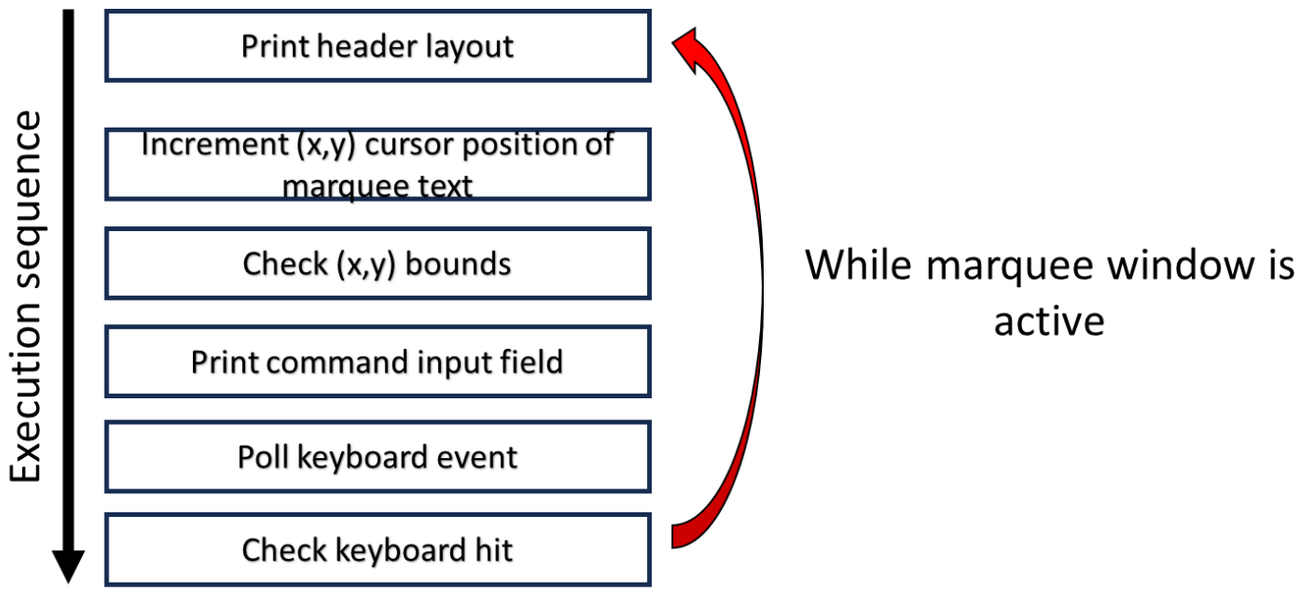
```
static constexpr int REFRESH_DELAY = 10; //screen refresh in marquee console
static constexpr int POLLING_DELAY = 5; //keyboard polling rate. Lower is better.
```

    - Link everything together in your main menu.
    - Be prepared to be called in class to discuss your implementation.

How do we validate the effectiveness of our keyboard polling implementation?
One method to verify this is to create a marquee console → an animated text that moves across the screen per frame while still allowing the user to type in text.

TODO: Demonstrate marquee console implementation.

General flow of the marquee console window is shown below:

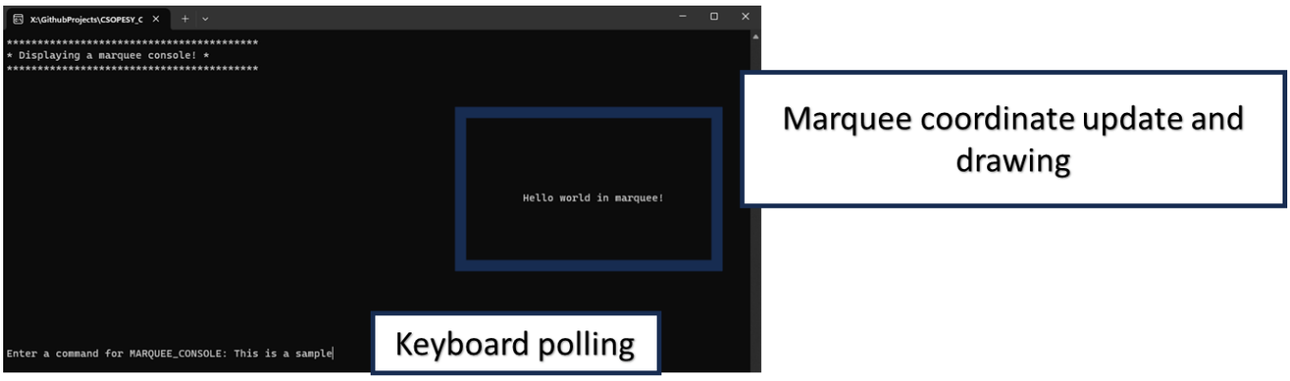


To create a customized console layout, we maximize the functionalities provided by the Windows Console API: https://learn.microsoft.com/en-us/windows/console/console-functions

```
114 void ConsoleManager::setCursorPosition(int posX, int posY) const
115 {
116     COORD coord;
117     coord.X = posX;
118     coord.Y = posY;
119     SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
120 }
```

Task	Classes and functions
Overall UI handler	MarqueeConsole
Print header layout	MarqueeWorkerThread
Cursor increment animation	MarqueeWorkerThread, ConsoleManager
Check (x,y) bounds	MarqueeWorkerThread, ConsoleManager
Poll keyboard event	MarqueeConsole, ConsoleManager
Check keyboard hit	MarqueeConsole, ConsoleManager

Promoting parallelism



- Observation: Marquee behavior is completely different and independent from keyboard polling.
- To further increase responsiveness, we can parallelize this behavior per CPU cycle. Most multiprocessor systems would assign another free logical core for any parallel execution. How do we do this?

Activities

- Implement your own marquee console application. Then, find a balance between having a fluid refresh rate, and polling rate.
    - Identify the recommended values for your current hardware.
    - Identify the limits of your refresh and polling rate - wherein there is the presence of screen tearing and/or there is a noticeable delay in characters being recognized during typing.
    - Report this in a PPT format and be prepared to be called in class.