03 Input Events

Functional and Implementation Guidelines

Functional Guidelines

1. Your application should implement a set of cross-platform functionalities
   1. time utils - availability to measure elapsed time
   2. sleep mechanism - a way to sleep the current execution of your thread
2. Your application should implement OS event handling
   1. mouse events - moves, clicks, mouse wheel
   2. keyboard events - presses, releases
   3. touchscreen events - click, release
3. The application should be closed on window X button, ESC keyboard symbol or termine signal
4. Play with the input event and implement some basic gameplay
5. Your application should implement the main loop design pattern
   1. ability to limit the FPS
   2. multiple event handling support for a single frame

Implementation Guidelines

1. **Input Events - Lecture**
2. Copy the EventDefines.h inside utils/input folder
3. implement InputEvent class
   1. Structure

|  |
| --- |
| //Do not expose SDL\_Event internals to the system  union SDL\_Event;  class InputEvent {  /\* Holds event position on the screen \*/  Point pos = Point::UNDEFINED;  /\* See EventDefines.h for more information \*/  int32\_t key = Keyboard::KEY\_UNKNOWN;  uint8\_t mouseButton = Mouse::UNKNOWN;  TouchEvent type = TouchEvent::UNKNOWN;  /\*\* Holds the OS communication event system \*/  SDL\_Event \*\_sdlEvent = nullptr;  }; |

* 1. API

|  |
| --- |
| int32\_t init();  void deinit();  bool pollEvent();  bool checkForExitRequest() const; |

* 1. Internal SDL\_Methods

|  |
| --- |
| #include <SDL\_events.h>  SDL\_PollEvent()  SDL\_GetMouseState() |

* 1. Internal state

|  |
| --- |
| void InputEvent::setEventTypeInternal() {  switch (\_sdlEvent->type) {  case EventType::KEYBOARD\_PRESS:  break;  case EventType::KEYBOARD\_RELEASE:  break;  //NOTE: the fall-through is intentional  case EventType::MOUSE\_PRESS:  case EventType::FINGER\_PRESS:  break;  case EventType::MOUSE\_RELEASE:  case EventType::FINGER\_RELEASE:  break;  //X is pressed on the window (or CTRL-C signal is sent)  case EventType::QUIT:  break;  default:  break;  }  } |

1. Implement an Engine class

|  |
| --- |
| class Engine {  MonitorWindow \_window;  InputEvent \_event;  SDL\_Surface \*\_screenSurface = nullptr;  }; |

* 1. Public API

|  |
| --- |
| int32\_t init();  void deinit();  void start(); |

* 1. **The Main Loop - Lecture**
  2. Private API

|  |
| --- |
| void mainLoop();  void drawFrame();  bool processFrame();  void handleEvent(); |

* 1. Exit program on user request
  2. Refactor the main.c file to use the new Engine class

1. Use the new resources from the resource folder
2. Expand the Engine class

|  |
| --- |
| enum Images {  UP, DOWN, LEFT, RIGHT, ALL\_KEYS, COUNT  };  class Engine {  MonitorWindow \_window;  InputEvent \_event;  SDL\_Surface \*\_screenSurface = nullptr;  SDL\_Surface \*\_currChosenImage = nullptr;  SDL\_Surface \*\_imageSurfaces[COUNT] {};  }; |

1. Use keyboard arrows to change, which image is being selected
   1. populate the handleEvent() method
2. Check your CPU usage while the program is active :)
3. Implement limitFPS method()

|  |
| --- |
| #include <thread> //temporary include, because it is not supported on MinGW  void limitFPS() {  using namespace std::literals;  std::this\_thread::sleep\_for(15ms);  } |

1. Implement Time set of functionalities
   1. TimerDefines

|  |
| --- |
| //using the metric system to determine time measurements  1 second = 1 \* 10^3 milliseconds  1 second = 1 \* 10^6 microseconds  1 second = 1 \* 10^9 nanoseconds |

* 1. Time class

|  |
| --- |
| #include <chrono>  class Time {  std::chrono::time\_point<std::chrono::steady\_clock> \_now = std::chrono::steady\_clock::now();  std::chrono::time\_point<std::chrono::steady\_clock> \_elapsedLastCall;  }; |

* 1. Time Public API

|  |
| --- |
| void initTime(struct Time\* time);  /\*\* @brief used to measure elapsed time since last invoke of this method  \* NOTE: .getElapsed() method simply measures time between 2  \* points of time.  \* \*/  Time& getElapsed();  int64\_t toSeconds() const;  int64\_t toMilliseconds() const;  int64\_t toMicroseconds() const;  int64\_t toNanoseconds() const; |

* 1. Time implementation

|  |
| --- |
| Time& Time::getElapsed() {  \_elapsedLastCall = \_now;  \_now = std::chrono::steady\_clock::now();  return \*this;  }  int64\_t Time::toSeconds() const {  return std::chrono::duration\_cast<std::chrono::seconds>(\_now - \_elapsedLastCall).count();  } |

1. Implement cross-platform sleeping mechanism
   1. By default everything in the standard should be cross-platform
   2. This is true only if the compiler you are using have implemented all of the standard features
   3. MinGW compiler does not fully support <thread> implementation

|  |
| --- |
| class Threading {  public:  Threading() = delete;  static void sleepFor(int64\_t microseconds);  }; |

|  |
| --- |
| #ifdef \_WIN32  #include <windows.h>  #else  #include <thread>  #endif  void Threading::sleepFor(int64\_t microseconds) {  #ifdef \_WIN32  const DWORD milliseconds = static\_cast<DWORD>(microseconds / 1000);  Sleep(milliseconds); //sleep uses milliseconds  #else  std::this\_thread::sleep\_for(std::chrono::microseconds(microseconds));  #endif  } |

1. Keep track of global time in the mainLoop method
   1. use the new Time class as part of the Engine
   2. measure time at the beginning of the frame and on the end of the frame
2. Limit the FPS based on the elapsed time and max frame rate

|  |
| --- |
| void Engine::limitFPS(const int64\_t elapsedMicroseconds) {  constexpr auto maxFrames = 60;  constexpr auto microsecondsInASeconds = 1000000;  constexpr auto maxMicrosecondsPerFrame = microsecondsInASeconds / maxFrames;  const int64\_t microSecondsFpsDelay = maxMicrosecondsPerFrame - elapsedMicroseconds;  if (0 < microSecondsFpsDelay) {  Threading::sleepFor(microSecondsFpsDelay);  }  } |