09 Buttons and Timers

Functional and Implementation Guidelines

Functional Guidelines

1. Implement a new set of functionalities that supports Button objects. Those object should support
   1. self-drawing API
   2. event handling
   3. callback invocation on button click
   4. important - the button should have absolutely zero knowledge about the entity it contacts for the click
   5. Example: there is a panel, which contains buttons. The button should signal the panel that it was clicked. But it is quite important for the button to not have the knowledge (and control) over the panel
2. Implement an Engine level functionality for supporting timers. It should support
   1. tracking elapsed frame time
   2. updating all started timers
   3. invoking any callbacks associated with the timers
3. Implement a Timer set of functionalities. Each Engine/Game object should be able to   
   use the Timer API
   1. start timer
   2. stop timer
   3. attach callback when a timer “tick”
   4. the timer should not have the knowledge (and control) over it’s holder
4. Implement some basic animations in order to test the new features are working as expected
5. Implement graphical debug console
   1. You can add custom information that could be visualized at run-time
   2. Have the option to turn the graphical debug console on/off

Implementation Guidelines

1. Implement ButtonBase class
   1. Structure

| enum ButtonStates {  UNCLICKED, CLICKED, DISABLED  };  class ButtonBase {  Image \_buttonTexture;  Rectangle \_boundRect = Rectangle::ZERO;  bool \_isInputUnlocked = true;  }; |
| --- |

* 1. Public API

| virtual void handleEvent(const InputEvent &e) = 0;  void create(int32\_t rsrcId, const Point &pos = Point::ZERO);  virtual void draw();  void lockInput();  void unlockInput();  bool isInputUnlocked() const;  bool containsEvent(const InputEvent &e); |
| --- |

* 1. Implementation

| void ButtonBase::lockInput() {  \_isInputUnlocked = false;  \_buttonTexture.setFrame(DISABLED);  }  void ButtonBase::unlockInput() {  \_isInputUnlocked = true;  \_buttonTexture.setFrame(UNCLICKED);  } |
| --- |

1. Delete the mouseTargetText
2. Implement GameProxy
   1. Signature provided in different header

| class GameProxy { public:  virtual ~GameProxy() = default;  virtual void onButtonPressed(int32\_t buttonId) = 0;  }; |
| --- |

* 1. Implement given in the Game.c file

| class Game : public GameProxy {  //…  private:  void onButtonPressed(int32\_t buttonId) final;  } |
| --- |

1. Implement WheelButton class
   1. Structure

| class WheelButton : public ButtonBase {  GameProxy \* \_gameProxy = nullptr;  int32\_t \_buttonId { 0 };  }; |
| --- |

* 1. Most important Public API

| int32\_t init(GameProxy \* gameProxy, int32\_t buttonId); |
| --- |

* 1. Attach the game address as a game proxy

| struct Game {  struct Hero hero;  struct Wheel wheel;  struct WheelButton wheelBtns[WHEEL\_BUTTON\_COUNT];  };  int32\_t Game::init(const GameConfig &cfg) { ...  if (EXIT\_SUCCESS != \_wheelBtn[0].init(this, 0)) {  //error  } ... } |
| --- |

* 1. This way the WheelButton can not access the functionality behind the game

| \_gameProxy->onButtonPressed(\_buttonId); |
| --- |

* 1. The button can only signal that some action needs to be taken by the game - nothing else

1. Implement startRotation/stopRotation methods that are activated with the button clicks.
2. Implement process() methods in the Engine, ManagerHandler, Wheel
   1. Process the animation in the Wheel when it is started
3. How can we implement this functionality without binding to the FPS of the game? Timers!
4. **Timers - Lecture**
   1. SDL\_Timers and why not use them
5. TimerMgr class
   1. Structure

| class TimerMgr: public MgrBase {  Time \_elapsedTime;  std::map<int32\_t, TimerData> \_timerMap;  std::set<int32\_t> \_removeTimerSet;  }; |
| --- |

* 1. Public API

| void process() final;  void startTimer(TimerClient \*tcIstance, int64\_t interval, int32\_t timerId,  TimerType timerType);  void stopTimer(int32\_t timerId);  void detachTimerClient(int32\_t timerId);  bool isActiveTimerId(int32\_t timerId) const;  void onInitEnd(); |
| --- |

* 1. Implementation

| void TimerMgr::process() {  const int64\_t msElapsed = \_elapsedTime.getElapsed().toMilliseconds();  for (auto it = \_timerMap.begin(); it != \_timerMap.end(); ++it) {  it->second.remaining -= msElapsed;  if (0 > it->second.remaining) {  onTimerTimeout(it->first, it->second);  }  }  removeTimersInternal();  }  void TimerMgr::removeTimersInternal() {  for (const int32\_t timerId : \_removeTimerSet) {  auto mapIt = \_timerMap.find(timerId);  if (mapIt != \_timerMap.end()) {  \_timerMap.erase(mapIt);  }  }  //clear the removeTimerSet  \_removeTimerSet.clear();  }  bool TimerMgr::isActiveTimerId(int32\_t timerId) const {  return (\_removeTimerSet.end() == \_removeTimerSet.find(timerId))  && (\_timerMap.end() != \_timerMap.find(timerId));  } |
| --- |

1. TimerClient class
   1. Structure

| class TimerClient {  //no members  }; |
| --- |

* 1. Public API

| virtual void onTimeout(int32\_t timerId) = 0;  void startTimer(int64\_t interval, int32\_t timerId, TimerType timerType);  void stopTimer(int32\_t timerId);  bool isActiveTimerId(int32\_t timerId) const; |
| --- |

1. Implement a timer in the Wheel struct (the rotation anim)

| class Wheel : public TimerClient {  Image \_wheel;  int32\_t \_rotTimerId = 0;  double \_currRotAngle = 0.0;  bool \_isRotStarted = false;  }; |
| --- |

1. startButton starts the wheel animation timer, stopButton stops it
2. Implement an Engine DebugConsole struct
   1. Structure

| //Note: this information is chosen as an example. Fill it as you see it  class DebugConsole {  Text \_fpsText;  uint32\_t \_maxFrames = 0; //used to determine the current FPS  int32\_t \_updateCounter = 0; //used for skipping turns  bool \_isActive = false;  }; |
| --- |

* 1. Public API

| void handleEvent(const InputEvent &e);  void update(int64\_t elapsedMicroseconds);  void draw();  bool isActive() const; |
| --- |

* 1. It should display debug information at run-time using texts in the upper left corner. Example: current FPS, active timers, etc …
  2. Some key (for Example ‘~’ Tilda) should toggle the DebugConsole on/off