

Input Handler

(Controller)

TPV2

Samir Genaim



Lecture Notes
Samir Genaim
TPV2 2021/22
Facultad de Informática
Universidad Complutense de Madrid

¿Qué es el Patrón Input Handler?

El Patrón Input Handler

Es una abstracción sobre el manejo de entrada. El objetivo es agregar todos los eventos de entrada en un sólo objeto y proporcionar un punto de acceso global a ese objeto para consultar los eventos, etc.

Un contexto ...

Suponemos que estamos usando una arquitectura de video juegos donde los objetos de juego heredan de la siguiente clase:

```
class GameObject {  
public:  
    GameObject() ...  
    virtual ~GameObject() ...  
  
    virtual void handleInput(const SDL_Event &event) = 0;  
    virtual void update() = 0;  
    virtual void render() = 0;  
  
    // ...  
};
```

El bucle principal en este contexto ...

El bucle principal del juego tiene la siguiente forma

```
while (!exit_) {  
    ...  
    // handle input  
    while (SDL_PollEvent(&event)) {  
        ...  
        for (auto &o : objs_) o->handleInput(event);  
    }  
  
    // update  
    for (auto &o : objs_) o->update();  
  
    // render  
    for (auto &o : objs_) o->render();  
    ...  
}
```

`objs_` es la lista de objetos de juego, es decir de tipo `std::vector<GameObject*>`

El bucle principal de un juego ...

```
while (!exit_) {  
    // handle input  
    while (SDL_PollEvent(&event)) {  
        for (auto &o : objs_) o->handleInput(event);  
    }  
    ...  
}
```

El problema con este diseño:

1. Para cada evento llamamos a `handleInput` de cada `GameObject`, aunque no los usa (o al parse/execute de los comandos)
2. Si queremos proporcionar otro tipo de entrada, tenemos que cambiar el método `handleInput` de `GameObject` (o los parse/execute de los comandos)

El objetivo es quitar el parámetro de `handleInput` y proporcionar otra manera de manejar la entrada: usando **InputHandler**

La clase InputHandler

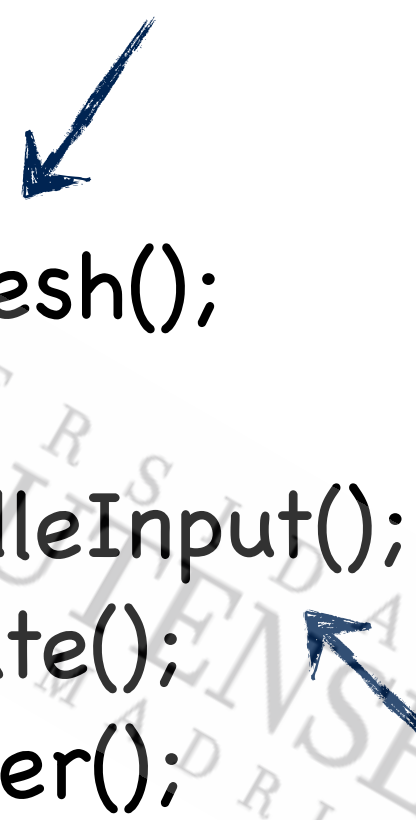
En nuestro diseño, InputHandler es una clase Singleton

- ♦ Mantiene un estado con los eventos de entrada que han ocurrido en la última iteración del bucle principal.
- ♦ Tiene un método `refresh()` para actualizarlo con los eventos actuales.
- ♦ Tiene métodos para consultar el estado (teclas, raton, etc) que se puede usar desde cualquier parte que tiene referencia al InputHandler

InputHandler: El bucle principal

Suponemos que refresh borra los eventos de la iteración anterior y actualiza el estado con los eventos actuales

```
while (!exit_) {  
    ...  
    InputHandler::instance()->refresh();  
    ...  
    for (auto &o : objs_) o->handleInput();  
    for (auto &o : objs_) o->update();  
    for (auto &o : objs_) o->render();  
    ...  
}
```



handleInput de GameObject no recibe el evento. Sólo una llamada por objeto, como el update y el render

InputHandler: ejemplo de uso

```
void Hero::handleInput(const SDL_Event& event) {  
    ...  
    if (event.type == SDL_KEYDOWN) {  
        if (event.key.keysym.sym == SDLK_a) {  
            ...  
        } ...  
    }  
}
```

sin InputHandler

```
void Hero::handleInput() {  
    ...  
    if (InputHandler::instance()->isKeyDown(SDLK_a)) {  
        ...  
    } ...  
}
```

con InputHandler

La Clase InputHandler

```
class InputHandler : public Singleton<InputHandler> {
```

```
...
```

```
inline void refresh();
```

```
inline void clearState();
```

```
inline void update(const SDL_Event &event);
```

```
// keyboard
```

```
inline bool keyDownEvent();
```

```
inline bool keyUpEvent();
```

```
inline bool isKeyDown(SDL_Scancode key);
```

```
inline bool isKeyDown(SDL_Keycode key);
```

```
inline bool isKeyUp(SDL_Scancode key);
```

```
inline bool isKeyUp(SDL_Keycode key);
```

```
// mouse
```

```
inline bool mouseMotionEvent();
```

```
inline bool mouseButtonEvent();
```

```
inline const std::pair< Sint32, Sint32> & getMousePos();
```

```
inline bool getMouseButtonState(MOUSEBUTTON b);
```

```
...
```

```
};
```

Para actualizar el estado

Para consultar el estado del teclado

Para consultar el estado del ratón

```
enum MOUSEBUTTON : uint8_t { LEFT = 0, MIDDLE = 1, RIGHT = 2 };
```

La Clase InputHandler

```
class InputHandler : public Singleton<InputHandler> {
```

```
...
```

```
private:
```

```
InputHandler() {  
    clearState();
```

```
    kbState_ = SDL_GetKeyboardState(0);
```

```
}
```

```
...
```

```
bool isKeyUpEvent_;
```

```
bool isKeyDownEvent_;
```

```
bool isMouseMotionEvent_;
```

```
bool isMouseButtonEvent_;
```

```
std::pair< Sint32, Sint32> mousePos_;
```

```
std::array< bool, 3> mbState_;
```

```
const Uint8 *kbState_;
```

```
};
```

Pedimos a SDL el array que tiene los estados de las teclas — solo una vez

Booleans para indicar el tipo de evento ...

La posición del mouse y el estado de los botones

`kbState_[SDL_SCANCODE_A]` es 0/1 depende del estado de la tecla A. Es un array de SDL


La Clase InputHandler

```
inline void refresh() {  
    SDL_Event event;  
  
    clearState();  
    while (SDL_PollEvent(&event))  
        update(event);  
}
```

Actualizar con todos los eventos en la cola de eventos de SDL, llamando a un método update (ver la siguiente página)

La Clase InputHandler

```
inline void clearState() {  
    isKeyDownEvent_ = false;  
    isKeyUpEvent_ = false;  
    isMouseButtonEvent_ = false;  
    isMouseEvent_ = false;  
    for (int i = 0; i < 3; i++) {  
        mbState_[i] = false;  
    }  
}
```



Simplemente borra el estado actual

La Clase InputHandler

```
inline void update(const SDL_Event &event) {  
    switch (event.type) {  
        case SDL_KEYDOWN:  
            onKeyDown(event);  
            break;  
        case SDL_KEYUP:  
            onKeyUp(event);  
            break;  
        case SDL_MOUSEMOTION:  
            onMouseMotion(event);  
            break;  
        case SDL_MOUSEBUTTONDOWN:  
            onMouseButtonChange(event, true);  
            break;  
        case SDL_MOUSEBUTTONUP:  
            onMouseButtonChange(event, false);  
            break;  
        ...  
    }  
}
```

depende de evento, invocamos a métodos correspondientes para actualizar el estado

La Clase InputHandler

```
inline void onKeyDown(const SDL_Event&) {
    isKeyDownEvent_ = true;
}

inline void onKeyUp(const SDL_Event&) {
    isKeyUpEvent_ = true;
}

inline void onMouseMotion(const SDL_Event &event) {
    isMouseMotionEvent_ = true;
    mousePos_.first = event.motion.x;
    mousePos_.second = event.motion.y;
}

inline void onMouseButtonChange(const SDL_Event &event, bool isDown) {
    isMouseButtonEvent_ = true;
    switch (event.button.button) {
    case SDL_BUTTON_LEFT:
        mbState_[LEFT] = isDown;
        break;
    case SDL_BUTTON_MIDDLE:
        ...
    }
}
```

La Clase InputHandler

```
inline bool keyDownEvent() {  
    return isKeyDownEvent_;  
}
```

```
inline bool keyUpEvent() {  
    return isKeyUpEvent_;  
}
```

```
inline bool isKeyDown(SDL_Scancode key) {  
    return keyDownEvent() && kbState_[key] == 1;  
}
```

```
inline bool isKeyDown(SDL_Keycode key) {  
    return isKeyDown(SDL_GetScancodeFromKey(key));  
}
```

```
inline bool isKeyUp(SDL_Scancode key) {  
    return keyUpEvent() && kbState_[key] == 0;  
}
```

```
inline bool isKeyUp(SDL_Keycode key) {  
    return isKeyUp(SDL_GetScancodeFromKey(key));  
}
```

Se puede quitar la llamada a keyDownEvent, pero hay que estar seguro de que hay un evento correspondiente antes de llamar a isKeyDown porque después de pulsar una tecla, su estado queda igual durante varias iteraciones.

¡Esto no garantiza que la tecla 'key' ha cambiado estado en la última iteración, sólo que su estado actual es UP!

Se puede quitar la llamada a keyUpEvent como en el caso de isKeyDown

La Clase InputHandler

```
inline bool mouseMotionEvent() {  
    return isMouseEvent_;  
}  
  
inline bool mouseButtonEvent() {  
    return isMouseButtonEvent_;  
}  
  
inline const std::pair< Sint32, Sint32 > & getMousePos() {  
    return mousePos_;  
}  
  
inline int getMouseButtonState(MOUSEBUTTON b) {  
    return mbState_[b];  
}
```