Properly handling keyboard input

Posted on **September 5, 2011**

Working with Windows, such a seemingly simple task such as handling keyboard input can turn into a small engineering nightmare. There is a myriad of deceitful APIs for basically doing the same thing, yet every single one of them has its own flaws. This post details how this was solved in Molecule – read on for a journey through Windows hell.

In order to get everybody on the same page, make sure to read this article about keyboard scan codes – it is a good introduction to the underlying hardware (mess).

Essentially, those scan codes (which can depend on the keyboard being used) will be turned into so-called virtual key codes which make sure that keys pressed on different manufacturers' keyboards will result in the same code.

So all is well – or not.

In addition to the above, virtual key codes were designed in a way such that different physical keys will carry out the same operation in a program, e.g. the default "Enter" key and the numpad "Enter" or the left-hand and right-hand "Shift" and "Control" keys. But more than often, we want to be able to map different **physical keys** to different inputs in a game, like e.g. in a FPS. The problem here is that **virtual** key codes and **physical** keys do not share a 1:1 relationship, which means that different physical keys can end up having the same virtual key code (like in the examples above). Professional gamers often need to map different actions to certain keys, hence the above is a problem which needs to be solved.

In addition to the obvious contenders for problematic keys, there are a few others which will cause problems if we only deal with virtual key codes. The following is a list I assembled while implementing Molecule's keyboard handling:

- Left-hand / right-hand “Shift”, “Control” and “Alt” keys – for obvious reasons.
- “Enter” / “Numpad Enter” – for obvious reasons.
- “Insert”, “Delete”, “Home”, “End”, “Prior”, “Next”, Arrow keys / corresponding keys on the numpad – the virtual key codes of the latter depend on the state of “Numlock”.
- “Alt Gr” on german (and other) keyboards – “Alt Gr” is not the same as right-hand “Alt” on US keyboards, it essentially is a short-cut for “Control” and right-hand “Alt”.
- “Print/SysRequest” – “Print” is internally handled as a hotkey in Windows because it has a special meaning.
- “Pause/Break” – this key is a mess in itself because “Pause” and “Break” act as if they are different physical keys, which of course they are not.
- “Numlock” – again, because it has a special meaning and additionally triggers something inside the hardware internally.

Having identified the culprits we have to deal with, let’s look at our options for capturing keyboard input using Windows:

- DirectInput
- WM_CHAR, WM_KEYDOWN, WM_SYSKEYDOWN messages (read up on message queues if you’re not familiar with this topic)
- GetKeyState, GetAsyncKeyState, GetKeyboardState APIs
- Low-level hooks
- Raw input

As stated in the beginning, everyone of the above has its own quirks and shortcomings, at which we will look now.

DirectInput

Quoting MSDN, “The use of DirectInput for keyboard and mouse input is not recommended, Windows messages should be used instead” already says a lot. Microsoft advises against using their own API for keyboard and mouse input, and rightfully so for several reasons:

- No support for keyboard repeat at the rate the user has set in the control panel – you have to re-invent the wheel.
- No support for keyboard layouts other than US-English.
- Others.

Not supporting any other layout than US-English is a big WTF in my book, and immediately disqualifies DirectInput as an option for handling keyboard input. If you’re still using DirectInput for this purpose and are reading this, make sure to never use e.g. “Y” or “Z” (and non-US keys) for any of your input, or you will piss people of.

WM_* messages

Listening to WM_CHAR messages is perfect for handling input for text fields (e.g. player name entry), because it automatically handles the state of the “Shift”-keys, IMEs, software-generated keys, and everything else. But it is the wrong option for high-speed, undelayed keyboard input often used in games – this is what WM_KEYDOWN is for.

Looking at the WM_KEYDOWN message, we can see that it will be sent for non-system keys (“Alt” is not pressed at the same time), and that in addition to the scan code it exhibits several additional bits such as bit 24 (extended bit), indicating whether the key is an extended key or not. Distinguishing the left-hand/right-hand “Shift” keys cannot be done by looking at the extended bit, but there’s the MapVirtualKey API for that. Additionaly listening to the WM_SYSKEYDOWN message enables us to correctly distinguish between left-hand and right-hand versions of the “Shift”, “Control” and “Alt” keys.

Unfortunately, when pressing “Alt Gr” on a german keyboard, Windows will send two messages – one for “Control”, and one for “Alt”, because that is what it originally was intended for. Therefore, it is impossible to distinguish between the “Alt” and “Alt Gr” keys on a german keyboard, which is a bit of a letdown. Other applications suffer from the same problem – try customizing any short-cut in Visual Studio with e.g. “Alt Gr + O”, and it will show up as “Ctrl + Alt + O”.

Additionally, some special keys such as “Print” will neither send a WM_KEYDOWN nor a WM_SYSKEYDOWN message, hence we have to try a different API.

GetKeyState and other APIs

In theory, you could call GetKeyboardState once a frame, grab the status of each of the keys, and forget about the rest. The problem with this API is that like the Windows messages, it isn’t low-level enough to provide us with a single key for “Alt Gr”, but rather notifies us about two key-presses instead. Bummer.

Low-level hooks

One would think that with something as low-level as event or system hooks, the above problems could be solved. Alas, “Alt Gr” will still result in two messages being sent. In addition to that, using low-level hooks for handling keyboard input is considered bad practice anyway, hence we’re not going down that road any further. In case you want to try it for yourself, here’s the code:

```
LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM lParam)
{
  PKBDLLHOOKSTRUCT p = (PKBDLLHOOKSTRUCT) (lParam);

  if ((wParam == WM_KEYDOWN) || (wParam == WM_SYSKEYDOWN))
```

```
  {
    ME_LOG0("Hook", "Virtual key code: %d", p->vkCode);
  }

  return CallNextHookEx(NULL, nCode, wParam, lParam);
}

// application start
HHOOK hook = SetWindowsHookEx(WH_KEYBOARD_LL, HookProc, instance,

// make sure to have a message pump somewhere

// application shutdown
UnhookWindowsHookEx(hook);
```

Raw input

So here we are, at the lowest possible level of capturing input events. The raw input API is relatively new, therefore many programmers are still unfamiliar with it. MSDN has an introduction on the topic, and here's some code for enabling raw input in your application:

```
RAWINPUTDEVICE device;
device.usUsagePage = 0x01;
device.usUsage = 0x06;
device.dwFlags = RIDEV_NOLEGACY;        // do not generate legacy
device.hwndTarget = hWnd;
RegisterRawInputDevices(&device, 1, sizeof(device));
```

The above will enable raw input for keyboard devices in your application, which can in turn be received by listening to WM_INPUT messages. The different usages and usage pages of HID devices are described in the USB HID Usage Tables document, which is where I got the above numbers from.

Getting input data from WM_INPUT messages is simple:

```
case WM_INPUT:
{
  char buffer[sizeof(RAWINPUT)] = {};
  UINT size = sizeof(RAWINPUT);
  GetRawInputData(reinterpret_cast<HRAWINPUT>(lParam), RID_INPUT,

  // extract keyboard raw input data
```

```
    RAWINPUT* raw = reinterpret_cast<RAWINPUT*>(buffer);
    if (raw->header.dwType == RIM_TYPEKEYBOARD)
    {
      const RAWKEYBOARD& rawKB = raw->data.keyboard;
      // do something with the data here
    }
}
```

A RAWKEYBOARD packet provides us with the following data:

```
    /*
     * The "make" scan code (key depression).
     */
    USHORT MakeCode;

    /*
     * The flags field indicates a "break" (key release) and othe
     * miscellaneous scan code information defined in ntddkbd.h.
     */
    USHORT Flags;

    USHORT Reserved;

    /*
     * Windows message compatible information
     */
    USHORT VKey;
    UINT   Message;

    /*
     * Device-specific additional information for the event.
     */
    ULONG ExtraInformation;
```

Scan code, flags, virtual key code – everything is at our disposal. And because we are at the lowest level now, pressing “Alt Gr” will result in one single message being sent – perfect!

Unfortunately, all is not well yet. Some keys still cause troubles, even with the WM_INPUT model:

- The virtual key code doesn’t distinguish between left-hand and right-hand keys such as “Shift”, “Control” and “Alt”, and the scan code depends on the keyboard hardware. Hence, we still have to use MapVirtualKey for distinguishing between left-hand and right-

hand versions. Surprisingly, MapVirtualKey will not work correctly for “Control” and “Alt” (who would have thought), but luckily we can decipher those by looking at the e0 and e1 bits.

- There’s still problems regarding the numpad keys, as stated above.
- Keys like “Pause/Break” and “Numlock” act strangely, sometimes like they are not even the same physical key – they use so-called escaped sequences which we have to decipher.

Assuming we have our “RAWKEYBOARD” packet ready, let’s start fixing this mess:

```
UINT virtualKey = rawKB.VKey;
UINT scanCode = rawKB.MakeCode;
UINT flags = rawKB.Flags;

if (virtualKey == 255)
{
  // discard "fake keys" which are part of an escaped sequence
  return 0;
}
else if (virtualKey == VK_SHIFT)
{
  // correct left-hand / right-hand SHIFT
  virtualKey = MapVirtualKey(scanCode, MAPVK_VSC_TO_VK_EX);
}
else if (virtualKey == VK_NUMLOCK)
{
  // correct PAUSE/BREAK and NUM LOCK silliness, and set the exte
  scanCode = (MapVirtualKey(virtualKey, MAPVK_VK_TO_VSC) | 0x100)
}
```

Some of the keys using escaped sequences will send “fake” keys with a virtual key code of 255, which we simply discard because there’s not much information to gather from them. The “Shift” key is handled using MapVirtualKey, which takes care of mapping the hardware scan code to the correct virtual key code constant such as VK_LSHIFT or VK_RSHIFT. The last if-clause corrects the **scan code** of VK_NUMLOCK messages, because believe it or not, the “Numlock” key will send the same scan code as the “Pause/Break” key, but a different virtual key code!

In addition to that, the MapVirtualKey API is knowingly buggy, and forgets about setting the extended bit for the scan code. In case you wondered what the scan code is used for, have a look at the GetKeyNameText API, which returns a human-readable string for each key on the keyboard while still correctly handling different keyboard layouts. Such an API is extremely handy and can be used for e.g. displaying input mappings in an option screen, but it needs

the scan code (and extended bits) to do its job.

Having the correct scan code and virtual key code ready, let's try to decipher the rest:

```
// e0 and e1 are escape sequences used for certain special keys, s
// see http://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html
const bool isE0 = ((flags & RI_KEY_E0) != 0);
const bool isE1 = ((flags & RI_KEY_E1) != 0);

if (isE1)
{
  // for escaped sequences, turn the virtual key into the correct
  // however, MapVirtualKey is unable to map VK_PAUSE (this is a
  if (virtualKey == VK_PAUSE)
    scanCode = 0x45;
  else
    scanCode = MapVirtualKey(virtualKey, MAPVK_VK_TO_VSC);
}
```

Again, for some keys like VK_PAUSE, their scan code needs to be corrected because they are part of an escaped sequence (identified by e1). It doesn't really help that MapVirtualKey is unable to map VK_PAUSE because of an API bug.

Luckily, the rest of the bunch can be deciphered easily:

```
switch (virtualKey)
{
  // right-hand CONTROL and ALT have their e0 bit set
  case VK_CONTROL:
    if (isE0)
      virtualKey = Keys::RIGHT_CONTROL;
    else
      virtualKey = Keys::LEFT_CONTROL;
    break;

  case VK_MENU:
    if (isE0)
      virtualKey = Keys::RIGHT_ALT;
    else
      virtualKey = Keys::LEFT_ALT;
    break;

  // NUMPAD ENTER has its e0 bit set
  case VK_RETURN:
```

```
  if (isE0)
    virtualKey = Keys::NUMPAD_ENTER;
  break;

// the standard INSERT, DELETE, HOME, END, PRIOR and NEXT keys w
// corresponding keys on the NUMPAD will not.
case VK_INSERT:
  if (!isE0)
  virtualKey = Keys::NUMPAD_0;
  break;

case VK_DELETE:
  if (!isE0)
    virtualKey = Keys::NUMPAD_DECIMAL;
  break;

case VK_HOME:
  if (!isE0)
    virtualKey = Keys::NUMPAD_7;
  break;

case VK_END:
  if (!isE0)
    virtualKey = Keys::NUMPAD_1;
  break;

case VK_PRIOR:
  if (!isE0)
    virtualKey = Keys::NUMPAD_9;
  break;

case VK_NEXT:
  if (!isE0)
    virtualKey = Keys::NUMPAD_3;
  break;

// the standard arrow keys will always have their e0 bit set, bu
// corresponding keys on the NUMPAD will not.
case VK_LEFT:
  if (!isE0)
    virtualKey = Keys::NUMPAD_4;
  break;

case VK_RIGHT:
  if (!isE0)
    virtualKey = Keys::NUMPAD_6;
  break;
```

```
    case VK_UP:
      if (!isE0)
        virtualKey = Keys::NUMPAD_8;
      break;

    case VK_DOWN:
      if (!isE0)
        virtualKey = Keys::NUMPAD_2;
      break;

    // NUMPAD 5 doesn't have its e0 bit set
    case VK_CLEAR:
      if (!isE0)
        virtualKey = Keys::NUMPAD_5;
      break;
  }
```

In addition to distinguishing between left-hand and right-hand versions of certain keys, the above switch-statement also makes sure that keys on the numpad will always be reported as such, no matter if “Numlock” is enabled or not. Otherwise, input mappings might suddenly not work any more (depending on the state of the “Numlock” key).

And for completeness sake, here’s code for detecting whether a key was up or down, and for getting a human-readable string:

```
// a key can either produce a "make" or "break" scancode. this is
// see http://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html
const bool wasUp = ((flags & RI_KEY_BREAK) != 0);

// getting a human-readable string
UINT key = (scanCode << 16) | (isE0 << 24);
char buffer[512] = {};
GetKeyNameText((LONG)key, buffer, 512);
```

Finally, we have a system in place which can detect keypresses of every single button on the keyboard (even the “weird” ones), distinguish between certain physical keys resulting in the same virtual key code, and even correctly handles exceptions to the rule like the “Break” key, which e.g. could be used to immediately trigger a breakpoint in the running program.

Reading my own post makes me realize again why it took me 1 1/2 days to figure all of that out. Talk about messed up APIs.

This entry was posted in **Input** and tagged **alt gr**, **API bug**, **C++**, **DirectInput**, **GetKeyNameText**, **keyboard hook**, **MapVirtualKey**, **raw input**, **window messages**, **windows**, **WM_INPUT** by **Stefan Reinalter**. Bookmark the **permalink [http://molecularmusings.wordpress.com/2011/09/05/properly-handling-keyboard-input/]** .

