

# Tutorial 2: Using The Keyboard

by Victor Saar

## Content

In this tutorial we do a bit more than before. We have the beginning of the input class, which inits DirectInput for now. Now we add keyboard support to the class. We have to create an input device for the main system keyboard and to set the behaviour. Then we need two more functions. One to update the status of the keyboard and one to ask for a pressed key. We use the code of Direct3D tutorial four. With the keys we can then modify the pyramide. Please read and understand the code of the Direct3D tutorial before you start with this one. I will not explain the parts concerning that, but only the new input class specific parts.

We don't need to add any files, because we already have the input class files.

## cinput.h

We have to add three functions to the input class. The first inits all keyboard related stuff. The second updates the current status of the keys and with the last we can check, which key was pressed.

```
class CInput
{
public:
    CInput(void);
    ~CInput(void);

    bool InitDirectInput(void);
    bool InitKeyboard(void); //inits keyboard stuff (NEW)

    //updates keyboard status (NEW)
    bool Update(void);
    bool KeyPressed(int); //ask for pressed key (NEW)
```

We also have to add some variables to the class. The first is the keyboard device and the second the buffer that holds the status of each key.

```
private:
    LPDIRECTINPUT8 m_pDIObject;
    LPDIRECTINPUTDEVICE8 m_pDIKeyboardDevice; //keyboard device (NEW)

    char m_KeyBuffer[256]; //buffer for keys (NEW)
};
```

## cinput.cpp

At the beginning some simple things. First we set the pointer to the keyboard device to NULL. Then we add the init function for the keyboard to the standard constructor and also add the clean-up to the destructor.

```
CInput::CInput(void)
{
    m_pDIObject = NULL;
    m_pDIKeyboardDevice = NULL; //pointer to NULL (NEW)

    if(!InitDirectInput()) g_App.SetD3DStatus(false);
    else if(!InitKeyboard()) g_App.SetD3DStatus(false); //init keyboard (NEW)
```

```

} //CInput

CInput::~CInput(void)
{
if(m_pDIKeyboardDevice != NULL)
{
m_pDIKeyboardDevice->Unacquire(); //unacquire device (NEW)
m_pDIKeyboardDevice->Release(); //release keyboard stuff (NEW)
m_pDIKeyboardDevice = NULL; //pointer to NULL (NEW)
}

if(m_pDIObject != NULL)
{
m_pDIObject->Release();
m_pDIObject = NULL;
}
} //~CInput

```

Now we have to implement the new input class functions. We start with `InitDirectInput()`, which has no parameters. At the beginning of it we must create the keyboard device with the help of the main object. The first parameter says that we want the standard system keyboard and the second is the pointer to the device. The last is always set to `NULL`. If one of the functions fails we close the program.

```

bool CInput::InitKeyboard(void)
{
if(FAILED(m_pDIObject->CreateDevice(GUID_SysKeyboard,
&m_pDIKeyboardDevice,
NULL)))
{
MessageBox(g_App.GetWindowHandle(),
"CreateDevice() failed!",
"InitKeyboard()",
MB_OK);
return false;
}
}

```

After that we have to set the device's data format. We give over the global variable `c_dfDIKeyboard`.

```

if(FAILED(m_pDIKeyboardDevice->SetDataFormat(&c_dfDIKeyboard)))
{
MessageBox(g_App.GetWindowHandle(),
"SetDataFormat() failed!",
"InitKeyboard()",
MB_OK);
return false;
}
}

```

Now we have to set the keyboard behaviour with `SetCooperativeLevel()`. We set the cooperative level to background access (`DISCL_BACKGROUND`) and non-exclusive access (`DISCL_NONEXCLUSIVE`).

```

if(FAILED(m_pDIKeyboardDevice->SetCooperativeLevel(g_App.GetWindowHandle(),
DISCL_BACKGROUND |
DISCL_NONEXCLUSIVE)))
{
MessageBox(g_App.GetWindowHandle(),
"SetCooperativeLevel() failed!",
"InitKeyboard()",
MB_OK);
return false;
}
}

```

At the end we have to acquire the keyboard device so that we can retrieve data from it.

```

if (FAILED(m_pDIKeyboardDevice->Acquire()))
{
    MessageBox(g_App.GetWindowHandle(),
        "Acquire() failed!",
        "InitKeyboard()",
        MB_OK);
    return false;
}

return true;
} //InitKeyboard

```

Now we have inited the keyboard stuff we need a function that updates the current keyboard status. This is done in the next function. Here we fill the `KeyBuffer` with these information. If the function failes we close the program.

```

bool CInput::Update(void)
{
    if (FAILED(m_pDIKeyboardDevice->GetDeviceState(sizeof(KeyBuffer),
        (LPVOID)&KeyBuffer)))
    {
        MessageBox(g_App.GetWindowHandle(),
            "GetDeviceState() failed!",
            "Update()",
            MB_OK);
        return false;
    }

    return true;
} //Update

```

The last function is used to see if a key was pressed. We take the value at the position stored in the variable `Key` and make with that a bitwise AND with `0x80`. The result is a new value, which is true or false. If the result is true we return true, else we return false.

```

bool CInput::KeyPressed(int Key)
{
    if (KeyBuffer[Key] & 0x80)
    {
        return true;
    }

    return false;
} //KeyPressed

```

## main.cpp

At the end we have to include the keyboard input into the program. The first thing we do is to add some variables. We add three for the x, y and z coordinate of the pyramide and one for the rotation.

```

//new variables (NEW)
float fRotation = 0.0f, fX = 0.0f, fY = 0.0f, fZ = 10.0f;

```

Now we have to change the main loop. Before the render part we first update the keyboard status. Then we change the new variables after the appropriate keyboard input, which we test with the class function `KeyPressed()`. If the return value is `true` the key was pressed. After that we make the matrix operations, which we need and set the transformation.

```

if(g_App.CheckDevice())
{
g_pInput.Update();
if(g_pInput.KeyPressed(DIK_RIGHT)) fX += 0.01f; //right arrow: x+ (NEW)
if(g_pInput.KeyPressed(DIK_LEFT)) fX -= 0.01f; //left arrow: x- (NEW)
if(g_pInput.KeyPressed(DIK_UP)) fZ += 0.01f; //up arrow: z+ (NEW)
if(g_pInput.KeyPressed(DIK_DOWN)) fZ -= 0.01f; //down arrow: z- (NEW)
if(g_pInput.KeyPressed(DIK_PRIOR)) fY += 0.01f; //prior page: y+ (NEW)
if(g_pInput.KeyPressed(DIK_NEXT)) fY -= 0.01f; //next page: y- (NEW)
if(g_pInput.KeyPressed(DIK_HOME))
    fRotation += 0.001f; //home: rotation+ (NEW)
if(g_pInput.KeyPressed(DIK_END))
    fRotation -= 0.001f; //end: rotation- (NEW)

g_App.GetDevice()->Clear(0,
                        NULL,
                        D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        D3DCOLOR_XRGB(0,0,0),
                        1.0f,
                        0);
g_App.GetDevice()->BeginScene();

//rotation matrix
D3DXMatrixRotationY(&matRotationY,fRotation);
//translation matrix
D3DXMatrixTranslation(&matTranslation,fX,fY,fZ);
g_App.GetDevice()->SetTransform(D3DTS_WORLD,
                               &(matRotationY * matTranslation));

g_App.GetDevice()->SetStreamSource(0,pTriangleVB,0,sizeof(D3DVERTEX));
g_App.GetDevice()->DrawPrimitive(D3DPT_TRIANGLELIST,0,4);

g_App.GetDevice()->SetStreamSource(0,pQuadVB,0,sizeof(D3DVERTEX));
g_App.GetDevice()->DrawPrimitive(D3DPT_TRIANGLESTRIP,0,2);

g_App.GetDevice()->EndScene();
g_App.GetDevice()->Present(NULL,NULL,NULL,NULL);
}

```

That is all for now. You can now translate the pyramide on the x, y and z axis and rotate it around the y axis. In the next tutorials mouse and joystick support will follow.