Lesson 12: Keyboard Input

3D Game Programming With C++

Digital Media Academy (Summer 2011)

Written by: Andrew Uzilov (andrew.uzilov@gmail.com)
Feel free to contact me with any questions.

In this lesson, we're going to learn how to make keyboard keys control what happens in our game.

The DMA framework comes with a special object called the Key Press Manager, or `KPM` for short. It listens to keystrokes on the keyboard and stores useful things about them, such as:

- Is a certain key currently pressed?
- How long has a certain key been pressed?
- Have we already done something to process the signal from this key?

`KPM` is a global object that is already defined for us, meaning we can use it in any file that begins with the line:

```
#include "../core/CommonDma.h"
```

The `CommonDma.h` file contains the definition of `KPM`. Whenever you use the word `KPM` in the code, is actually a shortcut for a function that returns a pointer to the Key Press Manager object, but we usually don't have to worry about that in practice. Just keep in mind that **`KPM` is effectively a pointer to an object**, meaning that we use an arrow (→) after it, not a dot, when we want to call its methods.

Initially, `KPM` doesn't listen to anything – we have to tell it specifically which keys it should track. This is done in the file `DmaPandaGame.cpp` – there is a TODO in its `main()` function where `KPM` **handlers** (things that listen for keypresses) are supposed to be set up. Here is how we tell KPM to start listening for keystrokes from the space bar:

```
        // Set up key press handlers here.
        KPM->addKeyHandler ("space");
```

FLASHBACK: Look back over Lesson 7. Notice that arrow notation (->) above? That means KPM is a **pointer to an object**, not an object itself. The above calls the method `addKeyHandler()` on that object. For pointers, we have to use arrows (->) instead of dots (`.`) to call a method.

After adding that line to `DmaPandaGame.cpp`, go back to the `logic()` method in `LevelOneState.cpp` – now we can add logic that responds to when the "space" key is pressed.

For example, let's say that we want to rotate our panda 1 degree for every frame that "space" is pressed. We could do it like this:

```
void LevelOneState::logic (void) {
	if (KPM->isKeyPressed ("space")) {
		player.set_hpr (player.get_hpr() + HPR (1, 0, 0));
	}
}
```

Run the above game and try holding down the space bar. The panda is spinning!

Now, let's try something different. Let's say we want to rotate the panda 90 degrees every time the key is **pressed and released.** So basically, it shouldn't matter how long we hold down the space bar – the panda should rotate 90 degrees per one keypress (pushing key down and releasing it), and that's it. We could try this:

```
void LevelOneState::logic (void) {
	if (KPM->isKeyPressed ("space")) {
		player.set_hpr (player.get_hpr() + HPR (90, 0, 0));
	}
}
```

Try running that code and see what happens. Something is wrong here – the panda is moving too much! That's because its heading will change by 90 degrees **for every frame during which the key is held down**. But we only want the panda to rotate on the first frame of the keypress , and that's it. How do we do it?

That's where the `isKeyProcessed()` method comes in. The Key Press Manager can store whether you've dealt with (or "processed") input from a certain key. So, on the first frame during which a key is pressed, we can take an action, like rotating the panda. Then, we tell the `KPM` that we have processed that key, and it saves that information. On all the remaining frames that the key is pressed, we check the `KPM` and if it tells us our key has been processed, we simply don't do anything.

When the key is released, `KPM` automatically resets the "key processed" state – it marks the key as not processed, so the next time it is pressed we can rotate the panda again.

Here is the code for putting it all together:

```
void LevelOneState::logic (void) {
     if (KPM->isKeyPressed ("space")) {
          // If the "space" key is pressed during this frame,
          // we are here.
          if (!KPM->isKeyProcessed ("space")) {
               // If the "space" key has NOT been processed, AND
               // if it is pressed, we are here.
               // This means that we are in the FIRST frame during
               // which the "space" key is held down.
               // Take action!
               player.set_hpr (player.get_hpr() + HPR (90, 0, 0));
               // This method call will tell KPM that the "space"
               // key has been processed by us.
               // This means that on following frames, the condition
               //     !KPM->isKeyProcessed ("space")
               // will be FALSE (NOT of "key is processed").  So, we
               // are not going to enter this code block, and the
               // panda will not rotate.
               KPM->setKeyProcessed ("space");
          }
     }
}
```

Think about it carefully and deeply – if the space bar is held down, what is going to happen on the first frame? Which lines are going to execute? What about the second frame and beyond? What about when you release the space bar?

Try running the code. Notice that now, the panda rotates 90 degrees only once per keypress, no matter how long you hold down the space bar! This is useful for **toggling things in your game between two states**, like:

- Making your game pause when you press a key.
- Changing the camera position between two different views.
- Turning a button or a lever off and on.
- And so forth.

Now, let's say that we want to make the panda rotate as long as the space bar is held down, but once every two seconds. Here is how we would do that:

```
void LevelOneState::logic (void) {
     if (KPM->isKeyPressed ("space")) {
          if (KPM->timeSinceKeyPress ("space") > 2) {
               player.set_hpr (player.get_hpr() + HPR (90, 0, 0));
               KPM->resetKeyPressTimer ("space");
          }
     }
}
```

Try running that code – hold down the space bar for a really long time, and notice how the panda behaves.

The `timeSinceKeyPress()` method returns the number (as the type double) of seconds that this key has been pressed. Inside `KPM`, there is a timer that keeps track of how each key has been held down, as long as that key has a handler (created by the `addKeyHandler()` method). The "if" conditional checks if the key has been held down for more than 2 seconds; if so, we update the player's HPR and reset the timer for this specific key.

This kind of code can be used anytime you want something to happen in your game once every X number of seconds. The biggest example of this is **controlling the firing rate for bullets** – let's say in your game, your player fires bullets at an enemy. Well, you probably don't want to fire a bullet every frame (that would be 50-60 bullets per second!), so we can use the above code to fire only one bullet per some slice of time.

One last very important thing. How we know the Panda3D names for various keyboard keys? The space bar is clearly called "space" – but what about the arrow keys, the enter key, insert, delete, and so forth? Here are the rules:

1) Keys that type a character are named after that character. So, to see if the letter "a" is pressed, you would use `KPM->isKeyPressed ("a")` for example.

2) Keys that don't type a character are named as follows:

```
"escape", "f"+"1-12" (e.g. "f1","f2",..."f12"), "print_screen"
"scroll_lock" "backspace", "insert", "home", "page_up", "num_lock"
"tab",  "delete", "end", "page_down" "caps_lock", "enter",
"arrow_left", "arrow_up", "arrow_down", "arrow_right" "shift",
"lshift", "rshift", "control", "alt", "lcontrol", "lalt", "space",
"ralt", "rcontrol"
```

More details (probably beyond what you need to know) are here:
http://www.panda3d.org/manual/index.php/Keyboard_Support

Exercise 12: Controlling the game with the keyboard

1) Take any previous exercise from Lesson 10, copy it to a new project, and make it so that you can quit the program any time by hitting the "q" key on the keyboard. You'll need to add a conditional to `logic()` that quits the game when "q" is pressed. **Hint:** to quit the game, invoke this special magic:

```
DMAGame::instance()->getFramework()->close_framework();
```

2) Make the panda move back, forward, left, and right using the arrow keys. Only the XYZ coords of the panda should change, not the HPR.

3) Make the panda move back and forth using "up" and "down" arrow keys (change its XYZ coords), and rotate left and right using "left" and "right" arrow keys (change its heading). The panda should always move back and forth **relative to where it is facing**.

4) Position the camera right behind the panda and looking down slightly, like this:

Now, using the same controls as in Question #2, have the camera follow the panda around as you move it with the arrow keys. **Hint:** The above starting view is obtained using these coordinates in `init()`:

```
player.set_pos (0, 0, 0);
player.set_hpr (90, 0, 0);
camera.set_pos (-120, 0, 50);
camera.look_at (120, 0, 10);
```

5) Write a first-person POV game – a program where you simply fly around the level. On each frame, you (your camera) should move forward a few units – so, you never stop moving. Use arrow keys to pitch up or down, or change heading (left or right). Create handlers for additional keys to rotate the camera left and right. Your game should basically allow the player to change every possible orientation parameter of the camera: **heading, pitch, and rotation**. For bonus points, add keys that increase and decrease your speed.

6) **Extra credit:** Take your solution from Question #3 and add a key to make the panda jump up.

   Hint: Think about the physics involved. On the first, and **only** the first frame during which the "jump" key is pressed, the panda launches into the air with a high **upward velocity**. On each following frame, gravity pulls the panda in the downward direction, and the panda's upward velocity (positive) gradually decreases (slows down), and eventually turns around and becomes negative (the panda starts to fall down). When the panda hits on the ground, it stops falling and returns to normal (the velocity along the Z axis is back to 0). You will want to create some new variables and add their declarations to `LevelOneState.h` (and their definitions to `init()`) that track whether the panda's state, such as its velocity.