Graphics Part II: Code Tutorial for Week 7

Dr P. Perakis

Preparation

We have provided a fully working code for the loading of a terrain and a TerrainNode class to render a Terrain with texture in the `Exercise_06_1.zip` file. This code can be used as a starting point for the following work, or you can use any working code you have up to now. You will also need to use the files provided in the `Exercise_07_Files.zip` file.

Introduction

This week, you should update the Camera class to your framework and see if you can fly around the terrain that you implemented last week. As before, the way I suggest to do it here, is just that – a suggestion. Feel free to implement it in any way you wish.

Updating the Camera Class

The `DirectXFramework` you have been using so far incorporates a very basic Camera class.

A possible updated `Camera.h` and `Camera.cpp` have been provided for you in the file `Exercise_07_Files.zip`.

Replace the existing files in your Visual Studio solution with these new files.

Using the Camera Class

There are two changes you now need to make to your code, for using the new Camera class:

- In `DirectXFramework.cpp`, add the following inclusion clause for using the Camera:

  ```
  #include "Camera.h"
  ```

- In `DirectXFramework.cpp`, add the following line to the `DirectXApp::UpdateSceneGraph()` method:

  ```
  UpdateCamera();
  ```

  This will ensure that the View Matrix is updated with any changes to the camera. We will see this method in the next section.

- In `DirextXApp::CreateSceneGraph`, set the initial position of the camera, e.g.

  ```
  GetCamera()->SetCameraPosition(0.0f, 50.0f, -300.0f);
  ```

Flying a camera around the terrain

Now we can try flying the camera around the terrain.

This tutorial uses a keyboard, but if you want to use a game controller, we have put a class that does this in the files for this week (`GamePadController.h` and `.cpp`). I will leave looking at this as a task for you. You will need to make changes to the class so that you can perform actions when game pad controls are used, but the basic logic is included.

In the case you handle input by keyboard, to read the state of a key, we could put in message handlers for `WM_KEYDOWN`, etc, but that will allow us to determine the state of a particular key at a

particular point in time, and not continuously. Instead, we can make a call to the `GetAsyncKeyState` function for reading the state at any time. This is defined as:

```
short GetAsyncKeyState(int virtualKey);
```

where `virtualKey` is the virtual key code of the key we want to test.

A full list of virtual key codes can be found at: https://msdn.microsoft.com/en-us/library/windows/desktop/dd375731(v=vs.85).aspx

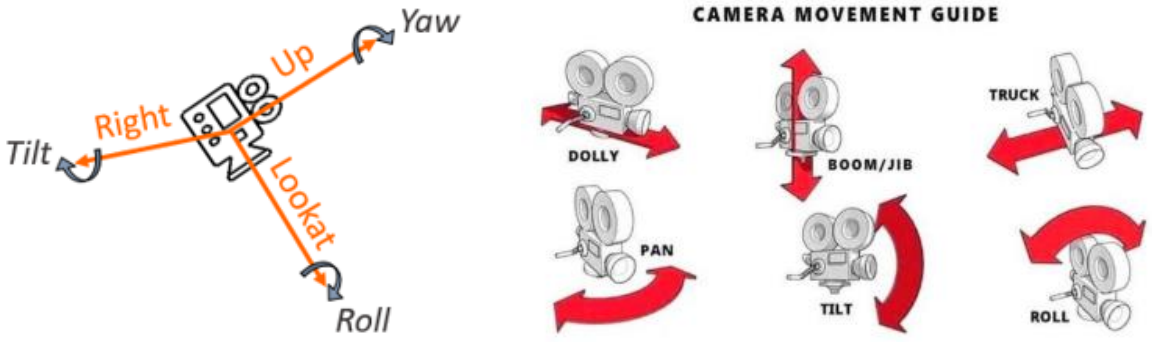
If the function succeeds, the return value specifies whether the key was pressed since the last call to `GetAsyncKeyState`, and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after the previous call to `GetAsyncKeyState`.

So, if we just want to test if a particular key is pressed, we can test for a negative result from the function (since a short is a signed value). For example:

```
if (GetAsyncKeyState(VK_UP) < 0)
{
	GetCamera()->SetForwardBack(1);
}
```

This will move the camera forward by one unit if the up-arrow key is pressed. Of course, you can use whatever keys you want.

To have control of the camera moves, you need adjust the position of the camera by moving it forward or backward along the look-at direction (dolly), the pitch of the camera for it to view up or down (tilt), the yaw of the camera for it to turn left or right (pan), and the roll of the camera to rotate it left or right. No boom neither truck is included.



I have created some code to do this in `DirectXApp::UpdateCamera()` method, using the w, s, a, d, q, e and shift keys:

```
void DirectXApp::UpdateCamera()
{
	float fwdbwdStep = 1.0f;
	float angleStep = 0.5f;

	if ((GetAsyncKeyState('W') < 0) && !(GetAsyncKeyState(VK_SHIFT) < 0))
	{
```

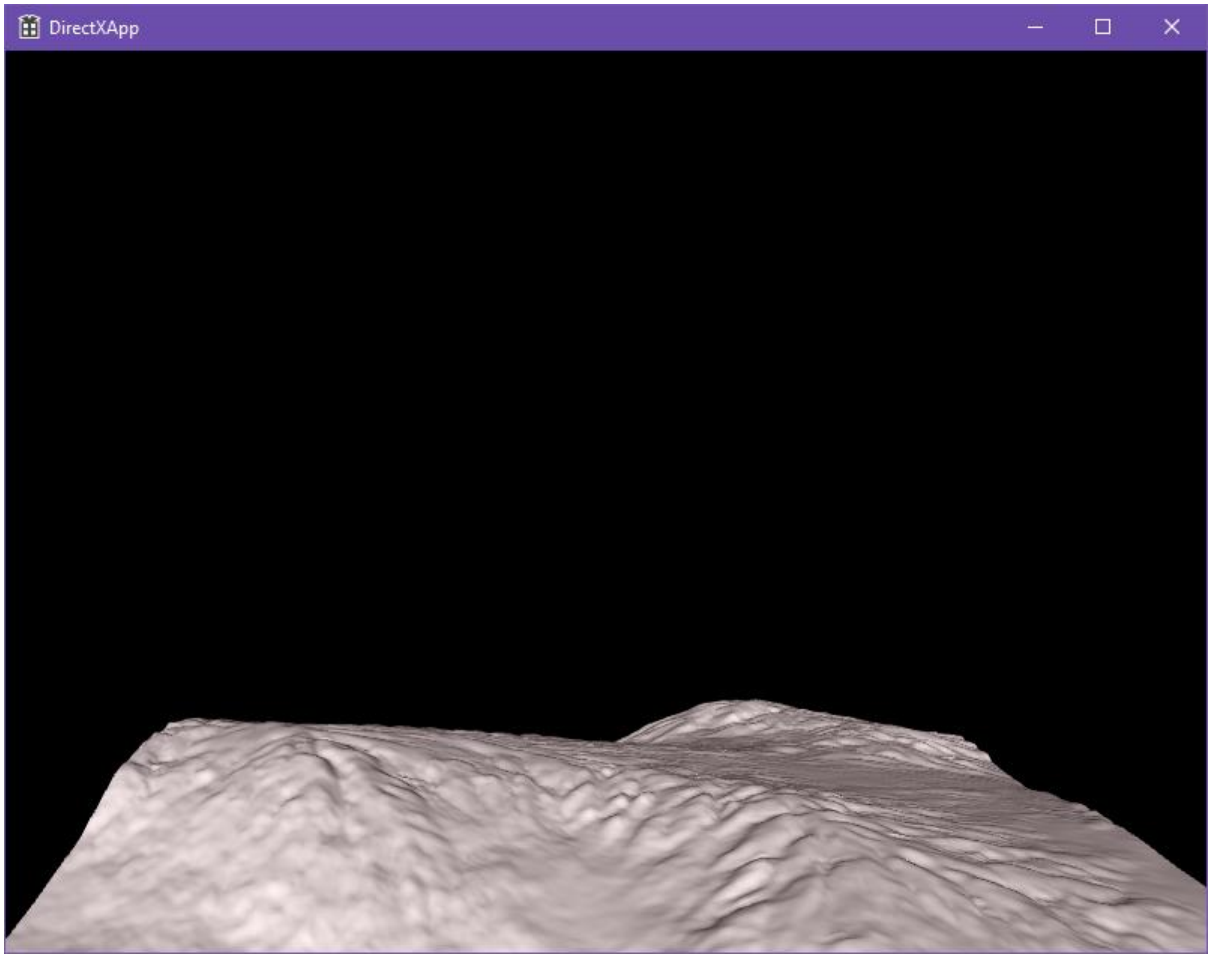
```
            GetCamera()->SetForwardBack(fwdbwdStep);
        }
        if ((GetAsyncKeyState('S') < 0) && !(GetAsyncKeyState(VK_SHIFT) < 0))
        {
            GetCamera()->SetForwardBack(-fwdbwdStep);
        }
        if (GetAsyncKeyState('A') < 0)
        {
            GetCamera()->SetYaw(-angleStep);
        }
        if (GetAsyncKeyState('D') < 0)
        {
            GetCamera()->SetYaw(angleStep);
        }
        if ((GetAsyncKeyState('W') < 0) && (GetAsyncKeyState(VK_SHIFT) < 0))
        {
            GetCamera()->SetPitch(-angleStep);
        }
        if ((GetAsyncKeyState('S') < 0) && (GetAsyncKeyState(VK_SHIFT) < 0))
        {
            GetCamera()->SetPitch(angleStep);
        }
        if (GetAsyncKeyState('Q') < 0)
        {
            GetCamera()->SetRoll(-angleStep);
        }
        if (GetAsyncKeyState('E') < 0)
        {
            GetCamera()->SetRoll(angleStep);
        }

        GetCamera()->Update();
}
```

Right now, as you fly around the terrain, you will see that you can fly into it, i.e. fly underground. In most cases, this would not be desirable, so the first thing we will look at when looking at collision detection is how to know if we are colliding with the ground. To do this, we need to know the height of the terrain at any value of X and Z. This is left as an exercise for you to try.

The output of the executable

After running the code that displays the solid hills of the terrain you see something like the following as an initial view.



After some moves using the keyboard, you may end up with something like this.