CItris - Console tetris in C#

Peter Wilkinson <Peter@quanglewangle.com>

Revision History
Revision 1.0 20 July 2011
Draft

Abstract

A simple console implementation of the game tetis in C# using no object oriented features

History, acknowledgements and licence

The source code is derived from **consoletetris**, a C++ program written by bidepan2 and obtained from https://code.google.com/p/consoletetris/ under an MIT licence

The code here has been simplified; had most of the OO aspects removed; translated into C#; and had a number of bugs fixed. I nonetheless owe a debt to bidepan2, which I freely acknowledge. One particular aspect of bidepan2's code I admire is the `transform()` function which rotates a block simply by changing it into another of the same shape but oriented 90° around

Purpose

There were three purposes to writing the software:

1. To remind myself how to program, since I was about to teach a programing course
2. To establish that it was possible to write a reasonably substantial program in C# without using object oriented features. This was because the syllabus I was about to teach excluded OO, more or less explicitly. This is not the place to argue the merits of excluding OO from an programming course: trust me that the syllabus did so exclude.
3. To provide an exemplar program around which to base the course

While the first two points were successful, the last failed: the program was too complex and too long to use to base the course around. I recovered some of the effort by saying that I would make the source code available at the end of the course, for students to do with whatever they liked.

The code

Playing surface

The game takes place on the `surface` two dimensional array

```
static int[,] surface = new int[COL, ROW];
```

Blocks

The blocks themselves are drawn inside a four by four cell bounding box. The painted cells are kept as far up and left in the bounding box as possible. There is only one active block at any one time. Three global variables are important for the active block.

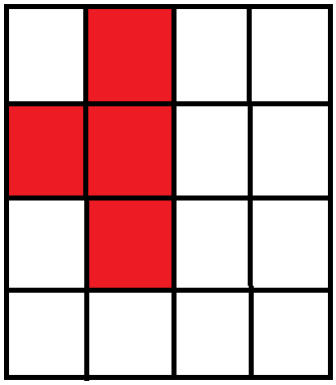

Figure 1. Sideways T in four by four bounding box

```
static int x;
static int y;
static blocktype type;
```

Block position is set by x and y and its type (see below) is kept in `blocktype`.

The sort of tile (Z, T, L, square, I) is held in `blocktype`. This is used as a reference into the array `blocklayouts`, which is an array of four by four bitmaps.

Here is a section showing the first two four by four tiles, for a vertical I and a horizontal I

```
 static int[,,] blocklayouts = {
{ { 1, 0, 0, 0 }, /* IV */
  { 1, 0, 0, 0 },
  { 1, 0, 0, 0 },
  { 1, 0, 0, 0 } },
{ { 1, 1, 1, 1 }, /* IH */
  { 0, 0, 0, 0 },
  { 0, 0, 0, 0 },
  { 0, 0, 0, 0 } },
...initializaion of other blocks ommitted for clarity
```

The hardest thing here is figuring out the syntax for initializing a three dimensional array.

Collisions and landing

The boolean function `collision()` checks (only) coloured cells in the currently active block to determine if

- the coloured cell would (if rendered) be off the edge of the playing surface or
- the coloured cell would (if rendered) overlay a coloured cell of a "landed" blocks

When a block lands it becomes part of the background.

Non-blocking input and "Busy waiting"

This program uses non-blocking input. `Console.KeyAvailable` is tested to see if a key is pressed, then if it is `Console.ReadKey(true)` is called to fetch the key. Performed in a loop this technique is known as *busy waiting*.

Busy waiting can soak up CPU cycles and profoundly effect performance and in the worst case can so starve the operating system of time that the display thread stalls and programs can even get into a state where they can not be stopped. The call to `Sleep()` passes control back to the operating system briefly to allow it to schedule other tasks. `Sleep()` takes as a parameter the number of milliseconds before returning. `Sleep()` sleeps for *at least* this number of milliseconds and may sleep longer so don't use it for timing.

Even if you want program to run stupidly fast don't be tempted to remove the call `Sleep()`. A parameter of zero will make it sleep momentarily, but still allow the operating system a look-in and the game will be so fast as to be unplayable.

```
while (Console.KeyAvailable) { Find out if a key is pressed

    If we get here there is a key pressed, so read it

    ConsoleKeyInfo key = Console.ReadKey(true);

        switch (key.Key){

           Switch cases omitted for clarity

      }
      The call to Sleep is essential.
      See text

      System.Threading.Thread.Sleep(30);

}
```

Compiling the code

The code was developed in Sharp Develop V3.2 but there is nothing about the code that makes it dependent on Sharp Develop. If you are using Sharp Develop the easiest way to complie the code is to start a new Console Mode project and copy and paste *all* of the source code over *all* of the initial default "Hello World" code. Then compile and run.