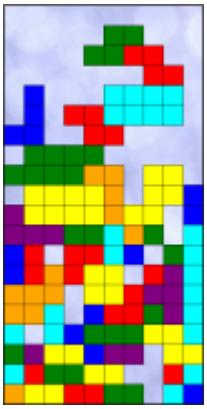
Javascript Tetris

Mon, Oct 10, 2011



My previous game, snakes, took a while to ship. It was a good experience to learn what it means to finish a game, but in reality, these are just side projects for me to play with some basic game mechanics and learn a little game programming. If I spend that amount of time on every minor project, I will still be doing this in 2035...

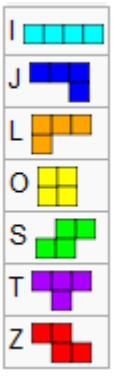
... with that in mind, I needed to ensure my next game was a short, single weekend project, so I chose an easy one, and I didn't spend any time polishing it beyond the primary game mechanic of ...

Tetris

- play the game now
- view the source code
- read more about how it works

Without the polish, it's a little ugly, but its fully functional, and I can show you how to implement it yourself...

Implementation Details



Tetris is a pretty easy game to implement, just a simple html file with some inline css/javascript.

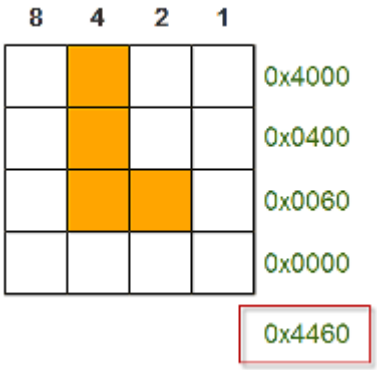
The only tricky part is dealing with rotation of the 7 tetrominoes.

You could try to do mathematical rotations at run time, but you will quickly discover that some of the pieces rotate around a central block (j, l, s, t and z), whilst others rotate around a point between blocks (i and o).

You could special case the 2 different behaviors, or you could accept the fact that Tetris is hard-coded to always have 7 pieces with 4 rotations and simply hard code all 28 patterns in advance.

You can avoid other special case code if you assume that all pieces are, conceptually, laid out on a 4x4 grid, where each cell is either occupied or not. Interestingly 4x4 = 16, and 'occupied or not' = 1 or 0.

That sounds like we can represent each pattern as a simple 16 bit integer to define exactly how we want each piece to rotate:



```
var i = { blocks: [0x0F00, 0x2222, 0x00F0, 0x4444], color: 'cyan'   };
var j = { blocks: [0x44C0, 0x8E00, 0x6440, 0x0E20], color: 'blue'   };
var l = { blocks: [0x4460, 0x0E80, 0xC440, 0x2E00], color: 'orange' };
var o = { blocks: [0xCC00, 0xCC00, 0xCC00, 0xCC00], color: 'yellow' };
var s = { blocks: [0x06C0, 0x8C40, 0x6C00, 0x4620], color: 'green'  };
var t = { blocks: [0x0E40, 0x4C40, 0x4E00, 0x4640], color: 'purple' };
var z = { blocks: [0x0C60, 0x4C80, 0xC600, 0x2640], color: 'red'    };
```

We can then provide a helper method that given:

- one of the pieces above

- a rotation direction (0-3)
- a location on the tetris grid

... will iterate over all of the cells in the tetris grid that the piece will occupy:

```
function eachblock(type, x, y, dir, fn) {
  var bit, result, row = 0, col = 0, blocks = type.blocks[dir];
  for(bit = 0x8000 ; bit > 0 ; bit = bit >> 1) {
    if (blocks & bit) {
      fn(x + col, y + row);
    }
    if (++col === 4) {
      col = 0;
      ++row;
    }
  }
};
```

Valid Piece Positioning

We need to be careful about our bounds checking when sliding a piece left and right, or dropping it down a row. We can build on our `eachblock` helper to provide an `occupied` method that returns true if any of the blocks required to place a piece at a position on the tetris grid, with a particular rotation direction, would be occupied or out of bounds:

```
function occupied(type, x, y, dir) {
  var result = false
  eachblock(type, x, y, dir, function(x, y) {
    if ((x < 0) || (x >= nx) || (y < 0) || (y >= ny) || getBlock(x,y))
      result = true;
  });
  return result;
};

function unoccupied(type, x, y, dir) {
  return !occupied(type, x, y, dir);
};
```

NOTE: assume `getBlock` returns true or false to indicate if that position on the tetris grid is occupied.

Randomizing the Next Piece

Chosing the next *random* piece is an interesting puzzle. If we chose in a purely random fashion, e.g:

```
var pieces = [i,j,l,o,s,t,z];
var next = pieces[Math.round(Math.random(0, pieces.length-1))];
```

... we find that the game is quite frustrating since we often get long sequences of the same type, and sometimes dont get a piece we want for quite some time.

It seems the standard way for picking the next piece in Tetris is to imagine a bag with 4 instances of each piece, we randomly pull one item from the bag until it is empty, then rinse and repeat.

This ensures that each piece will show up at least 4 times in every 28 pieces, it also ensures that the same piece will only repeat in a chain, at most, 4 times... well, technically it can be up to 8 because we could get a chain of 4 at the end of one bag followed by a chain of 4 at the start of the next bag, but the chances of that are quite remote.

This makes for a much more playable game, so we implement our `randomPiece` method something like this:

```
var pieces = [];
function randomPiece() {
  if (pieces.length == 0)
    pieces = [i,i,i,i,j,j,j,j,l,l,l,l,o,o,o,o,s,s,s,s,t,t,t,t,z,z,z,z];
  var type = pieces.splice(random(0, pieces.length-1), 1)[0]; // remove a sing
  return { type: type, dir: DIR.UP, x: 2, y: 0 };
};
```

Once we have our data structure and helper methods in place, the remainder of the game becomes fairly straight forward.

Game Constants and Variables

We declare some constants that never change:

```
var KEY     = { ESC: 27, SPACE: 32, LEFT: 37, UP: 38, RIGHT: 39, DOWN: 40 },
    DIR     = { UP: 0, RIGHT: 1, DOWN: 2, LEFT: 3, MIN: 0, MAX: 3 },
    stats   = new Stats(),
    canvas  = get('canvas'),
    ctx     = canvas.getContext('2d'),
    ucanvas = get('upcoming'),
    uctx    = ucanvas.getContext('2d'),
    speed   = { start: 0.6, decrement: 0.005, min: 0.1 }, // seconds until cur
    nx      = 10, // width of tetris court (in blocks)
    ny      = 20, // height of tetris court (in blocks)
    nu      = 5;  // width/height of upcoming preview (in blocks)
```

and some variables that will change, possibly `reset()` for every game:

```
var dx, dy,         // pixel size of a single tetris block
    blocks,         // 2 dimensional array (nx*ny) representing tetris court -
    actions,        // queue of user actions (inputs)
    playing,        // true|false - game is in progress
    dt,             // time since starting this game
    current,        // the current piece
    next,           // the next piece
    score,          // the current score
    rows,           // number of completed rows in the current game
    step;           // how long before current piece drops by 1 row
```

In a more complex game these should all be encapsulated within one or more classes, but to keep Tetris simple we are using simple global variables. But that doesn't mean we want them modified all over the code, so it still makes sense to write getter and setter methods for most of the variable game state.

```
function setScore(n)            { score = n; invalidateScore(); };
function addScore(n)            { score = score + n; };
function setRows(n)             { rows = n; step = Math.max(speed.min, speed.s
function addRows(n)             { setRows(rows + n); };
function getBlock(x,y)          { return (blocks && blocks[x] ? blocks[x][y] :
```

```
function setBlock(x,y,type)     { blocks[x] = blocks[x] || []; blocks[x][y] =
function setCurrentPiece(piece) { current = piece || randomPiece(); invalidate
function setNextPiece(piece)    { next    = piece || randomPiece(); invalidate
```

This also allows us to have a controlled way to know when a value has changed, so that we can `invalidate` the UI and know that section needs re-rendering. This will allow us to optimize our rendering and only `draw` things when they change.

The Game Loop

The core game loop is a simplified version of the same loop from pong, breakout and snakes. Using requestAnimationFrame (or a polyfill) we simply need to `update` our game state based on the time interval and then `draw` the result:

```
var last = now = timestamp();
function frame() {
  now = timestamp();
  update((now - last) / 1000.0);
  draw();
  last = now;
  requestAnimationFrame(frame, canvas);
}
frame(); // start the first frame
```

Handling Keyboard Input

Our keyboard handler is very simple, we dont actually do anything on a keypress (except for starting/abandoning the game). Instead we simply record the action in a queue to be handled during our `update` process.

```
function keydown(ev) {
  if (playing) {
    switch(ev.keyCode) {
      case KEY.LEFT:   actions.push(DIR.LEFT);  break;
      case KEY.RIGHT:  actions.push(DIR.RIGHT); break;
      case KEY.UP:     actions.push(DIR.UP);    break;
      case KEY.DOWN:   actions.push(DIR.DOWN);  break;
      case KEY.ESC:    lose();                  break;
```

```
    }
  }
  else if (ev.keyCode == KEY.SPACE) {
    play();
  }
};
```

Playing the Game

Having defined our data structures, setup our constants and variables, provided our getters and setters, started a game loop and handled keyboard input, we can now look at the logic that implements the Tetris game mechanics:

The `update` loop consists of handling the next user action, and if the time accumulated is greater than some variable (based on the number of completed rows), then we drop the current piece by 1 row:

```
function update(idt) {
  if (playing) {
    handle(actions.shift());
    dt = dt + idt;
    if (dt > step) {
      dt = dt - step;
      drop();
    }
  }
};
```

Handling user input consists of either moving the current piece left or right, rotating it, or dropping it 1 more row:

```
function handle(action) {
  switch(action) {
    case DIR.LEFT:  move(DIR.LEFT);  break;
    case DIR.RIGHT: move(DIR.RIGHT); break;
    case DIR.UP:    rotate();        break;
    case DIR.DOWN:  drop();          break;
```

```
  }
};
```

`move` and `rotate` simply change the current pieces `x`, `y` or `dir` variable, but they must check to ensure the new position/direction is unoccupied:

```
function move(dir) {
  var x = current.x, y = current.y;
  switch(dir) {
    case DIR.RIGHT: x = x + 1; break;
    case DIR.LEFT:  x = x - 1; break;
    case DIR.DOWN:  y = y + 1; break;
  }
  if (unoccupied(current.type, x, y, current.dir)) {
    current.x = x;
    current.y = y;
    invalidate();
    return true;
  }
  else {
    return false;
  }
};

function rotate(dir) {
  var newdir = (current.dir == DIR.MAX ? DIR.MIN : current.dir + 1);
  if (unoccupied(current.type, current.x, current.y, newdir)) {
    current.dir = newdir;
    invalidate();
  }
};
```

The `drop` method will move the current piece 1 row down, but if thats not possible then the current piece is as low as it can go and it will be broken into individual blocks. At this point we increase the score, check for any completed lines and setup a new piece. If the new piece will also be in an occupied position then the game is over:

```
function drop() {
  if (!move(DIR.DOWN)) {
    addScore(10);
    dropPiece();
    removeLines();
    setCurrentPiece(next);
    setNextPiece(randomPiece());
    if (occupied(current.type, current.x, current.y, current.dir)) {
      lose();
    }
  }
};

function dropPiece() {
  eachblock(current.type, current.x, current.y, current.dir, function(x, y) {
    setBlock(x, y, current.type);
  });
};
```

Rendering

Rendering becomes a fairly straight forward use of the `<canvas>` API, and a DOM helper method `html`:

```
  function html(id, html) { document.getElementById(id).innerHTML = html; }
```

Since tetris moves in a fairly 'chunky' manner, we can recognize that we dont need to redraw everything on every frame at 60fps. We can simply redraw elements only when they change. For this simple implementation we will break down the UI into 4 parts and only rerender those parts when we detect a change in:

- the score
- the completed row count
- the next piece preview display
- the game court

This last part, *the game court*, is quite a broad category. Technically we could track each individual block in the grid and only redraw the ones that have changed, but that would be

overkill. Redrawing the entire grid can be done in only a few ms, and ensuring we only do it when a change has occurred means we only take that small hit a couple of times a second.

```
var invalid = {};

function invalidate()         { invalid.court  = true; }
function invalidateNext()     { invalid.next   = true; }
function invalidateScore()    { invalid.score  = true; }
function invalidateRows()     { invalid.rows   = true; }

function draw() {
  ctx.save();
  ctx.lineWidth = 1;
  ctx.translate(0.5, 0.5); // for crisp 1px black lines
  drawCourt();
  drawNext();
  drawScore();
  drawRows();
  ctx.restore();
};

function drawCourt() {
  if (invalid.court) {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    if (playing)
      drawPiece(ctx, current.type, current.x, current.y, current.dir);
    var x, y, block;
    for(y = 0 ; y < ny ; y++) {
      for (x = 0 ; x < nx ; x++) {
        if (block = getBlock(x,y))
          drawBlock(ctx, x, y, block.color);
      }
    }
    ctx.strokeRect(0, 0, nx*dx - 1, ny*dy - 1); // court boundary
    invalid.court = false;
  }
};

function drawNext() {
  if (invalid.next) {
```

```
    var padding = (nu - next.type.size) / 2; // half-arsed attempt at centerir
    uctx.save();
    uctx.translate(0.5, 0.5);
    uctx.clearRect(0, 0, nu*dx, nu*dy);
    drawPiece(uctx, next.type, padding, padding, next.dir);
    uctx.strokeStyle = 'black';
    uctx.strokeRect(0, 0, nu*dx - 1, nu*dy - 1);
    uctx.restore();
    invalid.next = false;
  }
};

function drawScore() {
  if (invalid.score) {
    html('score', ("00000" + Math.floor(score)).slice(-5));
    invalid.score = false;
  }
};

function drawRows() {
  if (invalid.rows) {
    html('rows', rows);
    invalid.rows = false;
  }
};

function drawPiece(ctx, type, x, y, dir) {
  eachblock(type, x, y, dir, function(x, y) {
    drawBlock(ctx, x, y, type.color);
  });
};

function drawBlock(ctx, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x*dx, y*dy, dx, dy);
  ctx.strokeRect(x*dx, y*dy, dx, dy)
};
```

Room for Improvement

Like I said, this is just a raw Tetris game mechanic. If you were to polish this game you would want to add things like:

- menu
- levels
- high scores
- animation and fx
- music and sound fx
- touch support
- player vs player
- player vs ai
- (etc)

Related Links

You can play the game now or view the source code.

- Official Tetris
- Tetris on Wikipedia
- Tetris Classic

The game plays best on a desktop browser. Sorry, there is no mobile support.

- Chrome, Firefox, IE9+, Safari, Opera

J

J.R

10 years ago

Cool, I played for a while and is good!

However, the code is quite problematic. I'm not talking about the logic you used to solve the problem, I really liked the idea of representing the tetrominoes with 16-bit integers. The question is about the design decisions of the code. Your code is quite hard to read, but you can read it. However, it is impossible to maintain it or extend it. There is no way to add functionality without breaking any thing. I think that's the reason you do not want to revisit it soon. Nobody would want to maintain a code like this!

Do not get me wrong! I know that your intention is not to keep the code of this game as if it were an ongoing open source project, but you really should consider refactoring it to leave it exposed in your blog and your account on Github.

6 0 Reply • Share ›

C

Cassandra Martinez → J.R

8 years ago

i totally agree those hard coded values make for a lot of recoding when it comes to making a change

0 0 Reply • Share ›

A

Adam Nowak

5 years ago

Well done, sir

2 0 Reply • Share ›

S

Sourand Bitter

5 years ago edited

Very cool! That's my first time using bitwise operators and numbers as data structure. I rewrote the code using the original as a template, it was super instructive. Very succint and well written code, thank you.

1 1 Reply • Share ›

N

naing lay

10 months ago

i totally agree those hard coded values make for a lot of recoding when it comes to making a change

0 0 Reply • Share ›

G

geroz@gmx.de

4 years ago edited

Sorry for unearthing this ancient thread. ;-)

It is worth noting that some Tetris players prefer clockwise piece rotation, while some players (as myself) prefer counterclockwise piece rotation. Those players would and actually do utterly underachieve during gameplay when the J, L, and T pieces rotate "the wrong way" for their personal taste.

The blunt approach would be to re-arrange the J,L, and T blocks arrays in reverse order. A more subtle solution is to change the rotate() function from

```
var newdir = (current.dir == DIR.MAX ? DIR.MIN : current.dir + 1);
```

to

```
var newdir = (current.dir == DIR.MIN ? DIR.MAX : current.dir - 1);
```

For maximum flexibility, though, one could imagine to enhance the rotate() method by adding a boolean "clockwise" parameter to perform one or another above mentioned rotations. Like so:

```
function rotate(clockwise) {
var newdir;
if (clockwise) {
newdir = (current.dir == DIR.MAX ? DIR.MIN : current.dir + 1);
}
else {
newdir = (current.dir == DIR.MIN ? DIR.MAX : current.dir - 1);
}
if (unoccupied(current.type, current.x, current.y, newdir)) {
current.dir = newdir;
```

```
invalidate();
}
}
```

Then one could establish a second set of keys (perhaps WASD), so one set of controls would let the player rotate the pieces in clockwise order, and the other one in counterclockwise order. This would require to (i) extend both the KEY array (by 87, 65, 83, and 68, for W, A, S, and D, respectively) and (ii) the DIR array, (by an additional event, like "REVUP" for "reverse up"), and ultimately (iii) improving the keydown(ev) method to handle the additional for keys with the help of the above mentioned rotate(clockwise) method.

0 0 Reply • Share ›

A

Abhimanyu

5 years ago

DIR is not defined. Make sure you're spelling it correctly and that you declared it.
I thought you were going to type '{' but you typed 'pieces'.
I thought you were going to type '{' but you typed 'result'.
nx is not defined. Make sure you're spelling it correctly and that you declared it.
ny is not defined. Make sure you're spelling it correctly and that you declared it.

I ran it through my java checker and it came up with these errors

0 0 Reply • Share ›

O

okadahatch ↗ Abhimanyu

a year ago

its javascript not java maybe thats why

0 0 Reply • Share ›

Y

YOUTH CHANNEL

6 years ago

Hi, what is the algorithm for this game ?

0 0 Reply • Share ›

Bob

7 years ago

Hi Jake, can you tell me how the "upcoming" box doesn't show a preview but an image file that is linked to the variable block generated? So I want people not to see the shape of the next object but a picture. Can you help me?

0 0 Reply • Share ›

박준규

8 years ago edited

Thank you.
I read it several times and print into paper.
Your code is brilliant but hard to understand...
Because it needs deep understanding about javascript.
(Yes.. I'm a novice...)

0 0 Reply • Share ›

Jihad Bagas Cakra

8 years ago

Sir, i have the question.
If we want to add top ten highest score, is anything we can do to add it?

0 0 Reply • Share ›

C

Cassandra Martinez

↗ Jihad Bagas Cakra

8 years ago

yes you could add another module (function) then call it (simple way of putting it)

0 0 Reply • Share ›

tgwaste

9 years ago

very helpful thanks!

0 0 Reply • Share ›

G

gmc

10 years ago

Very nice work dude!

0 0 Reply • Share ›

Jake Gordon Mod ↗ gmc

10 years ago

Thanks!

0 0 Reply • Share ›

D

Darknior

11 years ago

11 years ago

Thanks for this demo :D
Do you will update it one day ?
Will you distribute the full source code ?

0 0 Reply • Share ›

Jake Gordon Mod → Darknior

11 years ago

The code that is up on github (see related links) is the full source code, its just the raw game mechanics without much flair.

I don't have any plans to revisit it any time soon, sorry.

0 0 Reply • Share ›

J

Joe

11 years ago

Hey, thanks for this great post. Is there a way to add image files for the different blocks instead of the css color? E.g: Draw images of each block and each rotation and then use them in the game. Where would I start on doing that?

0 0 Reply • Share ›

Jake Gordon Mod → Joe

11 years ago edited

Glad you like the post.

To draw images you will need to load them first, you do this by creating an IMG tag programatically, you can see an example of this under "Loading Assets" in my boulderdash article

http://codeincomplete.com/p...

Once the images have loaded you can draw them using the canvas drawImage method.

I know this has already been done with the arab spring tetris version (http://karlremarks.blogspot..., you can probably do a view-source and see how Karl modified the drawPiece method.

Good luck.

0 0 Reply • Share ›

S

Smith

12 years ago

Hey, I'm struggling to understand the rationale behind the function 'eachblock', specifically the bit manipulation? You think you can go into more detail regarding that?

0 0 Reply • Share ›

Jake Gordon Mod → Smith

12 years ago edited

I could try :-)

Piece Patterns
==========

Each rotation for each of the 7 pieces can be represented by a 4x4 bit pattern. The pattern for the L piece in the example image in the article is:

```
0100
0100
0110
0000
```

Written alternatively as a sequential 16 bit binary is

```
0100-0100-0110-0000
```

- which gives us our 0x4460 hexadecimal.

see more

3 0 Reply • Share ›

S

Smith → Jake Gordon

12 years ago

Yeah, that really clears things up. As a novice programmer I found using the hexadecimal numbers to represent the blocks extremely novel. I wasn't clear on how they actually represented the blocks until you explained it. Are bitwise operations done for purposes such as this usually?

Thanks.

0 0 Reply • Share ›

Jake Gordon Mod

→ Smith

12 years ago edited

I think this kind of thing is fairly rare in dynamic languages such as javascript and ruby when performance is not the #1 priority.

It's more likely to be seen in native C/C++ code where performance and memory use is usually a priority, particularly on platforms where memory is tight, such as embedded systems and such

If performance is not an issue, it's generally best practice (IMO) to favor more readable, but perhaps more verbose, data/code over bit twiddling such as this. Long term you should favor maintainability and ease of understanding.