CS188 - Final Project Report
Susumu Noda

Tetris Express

Introduction

My final project idea was to implement the addicting, decades-old game, Tetris. From our aquarium lab, I became interested in TiledLayer and Sprite manipulation, as well as the idea of using these to put together a playable game. As simple as the idea behind the game is, it was surprisingly difficult to implement several aspects of the game, and it required a certain level of creativity on my part to make it work. Here I will outline some of my accomplishments and the problems I encountered while recreating this widely popular game.

Getting started

Before I even got started on implementing the actual game, I had to figure out how to make the screen look like a game. For this, I needed a playable game area, a score area, and a level display, among other things. This was relatively simple, as it was only a matter of painting onto the screen and placing some TiledLayers in the write place. I actually created a reusable class called BlockGameArea that has a host of easy-to-use methods that I used to set up my Tetris game area (simple methods like `createBorder(), setGameBackground(), setScore()`, etc.). At this point, I was ready to take on the real challenge of making Tetris work in J2ME.

Where do I even begin?

So the first step in creating this game was to understand what exactly the Tetris blocks (or tetrominoes) are. Having made the aquarium in Lab 4, I was somewhat familiar with Sprites and what they did. I knew that they were made up of an image, or sequence of images, and that they could move, collide with things, and so forth. This is exactly what I was looking for, and, in actuality, the Sprite does much more than what I need a Tetris block to do; for example, I have no desire to animate the blocks. Thus, I had a direction: to use Sprites as my basis for Tetris blocks.

Now what?

For a considerable amount of time, I was at a roadblock where I could not figure out the best way to make a "block" out of sprites. Some options that I considered were using a Sprite that had the different block configurations (orientations) as its sequences. While that would not have been entirely impossible, I foresaw a major setback: how would I make blocks disappear when a line was full? Should a block be one entity in itself or a bunch of smaller components that have been somehow "glued together?"

Tetris block: Array, TiledLayer, or something else?

I pretty much threw out the idea of the Tetris block being a Sprite with several sequences depicting its orientations. There was just no way of figuring out how to remove a single box from the whole block, or two, or three. I wondered, could I somehow make a block out of an array of

Sprites put together? But then, how would I know how big the array should be, and more over, how would I make it displayable? Then I realize that a TiledLayer is already pretty much an array of displayable components. However, a TiledLayer *does not* have the capabilities of a Sprite; for example, there is no collision detection in the TiledLayer class. For another, it cannot rotate if needed. For these reasons and others, I decided against using a TiledLayer for my block implementation. Then, after some (a lot of) thinking, I finally came up with the idea of a SuperSprite class.

SuperSprite Class

The idea behind the SuperSprite class was to somehow bring together a bunch of Sprites and "glue" them together, being able to manipulate the whole thing as a single entity. The simplest way to do this, I realized, was to make a linked list of Sprites that, depending on the direction to which the Sprite was added, would appear to the top, bottom, left, or right of the last Sprite that was added (think of the classic game "Snake" where each block is attached to the next in some direction).

A SuperSprite is constructed as follows:

```
public SuperSprite(Sprite sprite, int start_x, int start_y){
    pos_x = start_x;
    pos_y = start_y;

    iWidth = sprite.getWidth();
    iHeight = sprite.getHeight();
    head = new SpriteNode();
    head.sp = sprite;
    head.id = idCount++;
    setPosition(start_x,start_y);
    tail = head;
    refNode = head;
}
```

As you can see, the Sprite is passed in as a parameter, leaving it possible for the user to manipulate each Sprite individually as well.

Next, we see how more Sprites are appended onto the current SuperSprite:

```
public void append(Sprite sprite, int direction){
    tail.next = new SpriteNode();
    tail.next.sp = sprite;
    tail.next.direction = direction;
    tail.next.prev = tail;
    tail.next.id = idCount++;

    switch(direction){
        case UP:
            tail.next.sp.setRefPixelPosition(tail.sp.getX(),tail.sp.getY()-iHeight);
            break;
        case RIGHT:
            tail.next.sp.setRefPixelPosition(tail.sp.getX()+iWidth,tail.sp.getY());
            break;
        case DOWN:
            tail.next.sp.setRefPixelPosition(tail.sp.getX(),tail.sp.getY()+iHeight);
            break;
        case LEFT:
            tail.next.sp.setRefPixelPosition(tail.sp.getX()-iWidth,tail.sp.getY());
            break;
    }
    tail = tail.next;
    updateWidth();
    updateHeight();
}
```

As explained, each new Sprite is added to the last node in the SuperSprite in some direction, and that new node's position is adjusted to reflect this positioning.

So, after coming up with this idea, it became incredibly easy to create my Tetris blocks (and any other block shape that I wanted to) with as few as 2 or 3 lines of code:

```
blockArray[blockCount].append(new Sprite(newBlockImage),SuperSprite.DOWN);
blockArray[blockCount].append(new Sprite(newBlockImage),SuperSprite.DOWN);
blockArray[blockCount].append(new Sprite(newBlockImage),SuperSprite.DOWN);
```

This code takes a SuperSprite, originally only one Sprite in length, and adds 3 more Sprites to the bottom of it, producing something like this:

Finally, with my blocks created, the next step in my project was to figure out how to move them, make many of them appear on the screen and, ultimately, to have them collide with each other.

Next step: moving, having multiple SuperSprites

The first step was simple enough. Since Sprites already come with `move() and setPosition()` methods, all I had to do when moving a SuperSprite was iterate through the SuperSprite linked list (i.e. from head to tail) and move all nodes by the specified amount. These were then mapped to key functions on the keyboard using GameCanvas's `keyPressed()` method. Now I had moving SuperSprites on the screen, and I was making progress.

To have multiple SuperSprites on the screen at once, I obviously needed to store the previously created ones somewhere, and what easier way to do this than to simply use an array? This proved extremely effective and easy, and, each time a SuperSprite was created, a counter into the SuperSprite array was simply incremented. (To clarify without going into unnecessary detail, a new block or SuperSprite is created whenever the last one has hit the bottom or another block, as per Tetris standards). Thus, in my `paint()` method, in which all of the displayable components of the game are painted onto the screen, all I have to do is to iterate through all of the elements in the SuperSprite array in a for-loop and paint each one. Compared to the rest of my challenges, this was fairly simple to do.

Perhaps the most annoying part of Tetris: Collision detection

As I said before, SuperSprites must know when they collide with either the game's border or another SuperSprite and then *not* continue to move if they do. The Sprite class included in J2ME does in fact come with a `boolean collidesWith()` method, but with one major setback, at least in terms of my needs. The method included in the Sprite class returns true only if two Sprites are already colliding (i.e. sharing coordinates on screen). It will *not* return true if two Sprites share an edge, however. That is, if a Tetris block looks like it is resting at the bottom of the border (i.e. its bottom edge is "touching" the border), the Sprite still has not collided with it, much to my dismay. Were that the case, I would have had far fewer headaches trying to figure out a way to get around this and, once I figured it out, trying to fix all the bugs that came with it.

Days later, I still hadn't come up with a viable solution. Then, after many, many hours of contemplation, I finally came up with a solution that might work. The idea was as follows: given that a SuperSprite's `collidesWith()` method (which uses Sprite's method of the same name) only returns true once it has *already* collided with something, why not create "ghost" SuperSprites that always surround the current Tetris block to let me know if it has collided with something. Basically, I decided to have SuperSprites above, below, to the left, and to the right of my main SuperSprite that were there solely for the purpose of collision detection. For example, if my left "ghost" collides with something, I don't want to move any more to the left. The same thing applies for the rest of my "ghosts." The idea actually turned out wonderfully, and I was thus able to successfully detected if my block was colliding with anything.

For reference, a block with its ghosts visible looks something like this:

One more subtle but cumbersome detail about Sprite collision is that, for some reason, they have to be rendered at odd places in order for collision to be detected properly. This is one thing that I never really truly figured out or debugged, but, for example, I render the ghosts in my `init()` function as well as in my `paint()` function, because otherwise they behaved in an unexpected manner. All said and done, by this point I had moving SuperSprites that would know when they hit something and would stay on screen when the next SuperSprite was created.

The next hurdle: rotating SuperSprites

When I got this point, I felt that I was almost done with the game, except I knew was neglecting the two most important aspects of the game (rotating the pieces around to the fit empty slots and for the lines to disappear when they needed to). The former of these two was a considerable challenge. I again knew that the Sprite class came with a set of transform methods, but I couldn't see how they would come in useful for my purposes. I needed to somehow make the entire *SuperSprite* change in orientation, not just its constituent parts. After debating the mechanics of how I was going to make this work, I finally decided on a couple of things: 1) to make one of the nodes in the SuperSprite a *reference node* around which the rest of the Sprites would rotate, and 2) in order to "rotate," I just need to adjust the direction to which nodes are attached to each other (remember the direction parameter in the `append()` method). With this in mind, I set about choosing for each Tetris shape the node that I figured would be the best "middle point," and then I got to work on my `rotate()` method.

Once the idea was in my head, the actual implementation for this method wasn't too bad. It was tedious cumbersome trying to figure out and imagine where a node would be attached to the previous node, given the degree that it was rotating. However, some pen and paper solved this issue, and it was fairly easy to make it so that, if a SuperSprite were rotated 90 degrees, all Sprites would be "reattached" to the node before it in the direction that is (counterclockwise) one before it. For example, if node B were attached to node A to the right, it would now be attached to the top of A. This, among other details, allowed for successful rotating of my blocks.

As a side note, I was about to panic when the ghost blocks *weren't* rotating around the main block as I expected, but I soon figured out that I was just unknowingly resetting their position every time a key was pressed (as was appropriate *before* rotation was implemented).

CS188 - Final Project Report
Susumu Noda

Rotating blocks that move and collide…next is?

Finally, I came to my last big challenge in implementing Tetris. Now that I have everything else ironed out and working as expected, all I had to do was to make the lines disappear and implement some way to keep a score. This sounded easy enough, and indeed compared to the rest of the troubles I had, it was nowhere near as difficult. My implementation of line-checking was as follows: have an array that essentially tells me how many Sprites (note, individual Sprites) occupy each "line" (in this case, each of 16 rows), and if a line reaches capacity (i.e. 10, the number of columns), then clear that line and move everything above it down one line. To do this, every time a block is dropped or has collided with something at the bottom, I go through each of my SuperSprites and each of their Sprites in two for-loops to record what their coordinates are; these coordinates are used to calculate which row the Sprite is in. If a line is deleted, then another two for-loops are used to move everything *above* that row down one row. While this works correctly, I understand that using so many for-loops is incredibly inefficient, for future optimizations, this is something I would work to improve on.

Wrapping up: Score-keeping, levels, etc.

With everything finally functioning, my last task was to increment the score every time lines were cleared, and to increase the level every so often to keep things interesting. With each level-up, the speed increases slightly, up to a point where it is very difficult. These things are of course displayed on the screen as well.

Some other things that I added were the option to turn on "crazy mode," which brings up a few extra Tetris shapes that don't usually show up in the classic game. Again, this is just something to keep the game interesting and to allow some level of configurability in the game. The other thing that users can configure is the background, which has two options: a cool, galactic experience, or a nice, relaxing aurora borealis.

What I would add

Some things that I would add in future versions of the game are a high score board, perhaps one that can be uploaded to a server and viewed by other players, or at least one that resides persistently on the mobile device. The reason I chose not to do this was that I realized, at least for my machine, that RMS did not work at all when I tried to use it in a previous lab. Thus, I figured it wasn't worth too much of my effort to debug/configure WTK to make this work in an emulator. One final thing I might add is the ability to see the upcoming block. I primarily chose to exclude this due to time constraints and to avoid excessive changes to my working code.