Sprite sheet packer: performance testing

The two relevant metrics for determining the viability of a packing algorithm are computational time and packing ratio. Packing ratio is defined here as the sum of the individual areas of the rectangles divided by the area of the bounding rectangle. As an approximation for the computational time, the real world time spent on the packing function was measured.
For testing purposes, a special benchmark package was created. The benchmark can be accessed by running the program itself with the command line argument ”-benchmark”, which causes the program to change its normal behavior and run the PackerComparison.runTest method from the performancetest package instead. This method creates a pseudo-randomly generated set of rectangles whose maximum dimensions are determined by the specified width of the sprite sheet, and uses it to generate a layout with each of the packing algorithms. While generating the layouts, it records the time used. It also calculates and reports the resulting packing ratio.

Results

The packing ratio can depend surprisingly much on random chance. To combat this, the same test was performed eight times with different seeds. Results become more consistent as the number of rectangles increases. However, since the computational time for the Scanline algorithm quickly becomes prohibitively long, 250 rectangles was chosen as a suitable compromise.

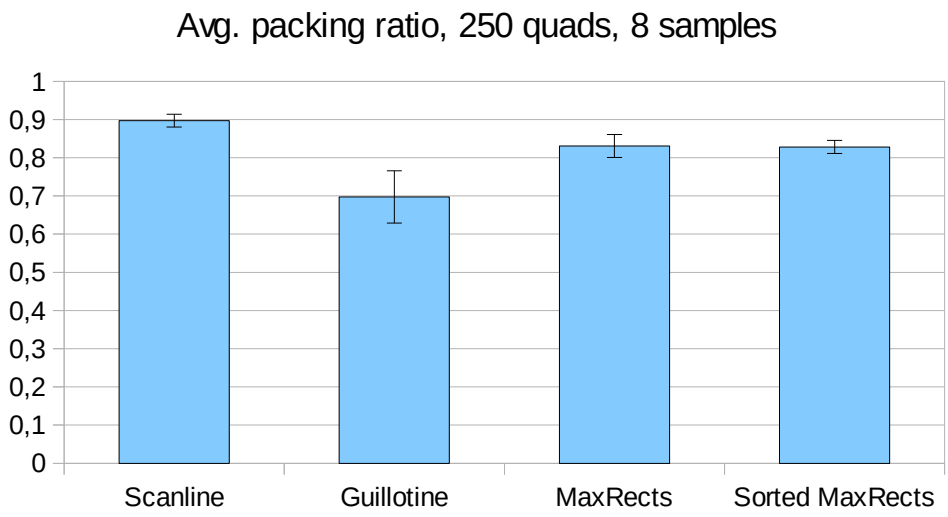

Figure 1: Averages and standard deviations of packing ratio for the algorithms tested.

The timing tests were done with a varying number of sprites, while keeping the seed constant at 0. As mentioned, the Scanline algorithm was notably slower than the other algorithms and was therefore left out for rectangle counts above 500. The Guillotine algorithm was the fastest one by far and shows an almost linear time requirement, despite it theoretically being O(N^2).

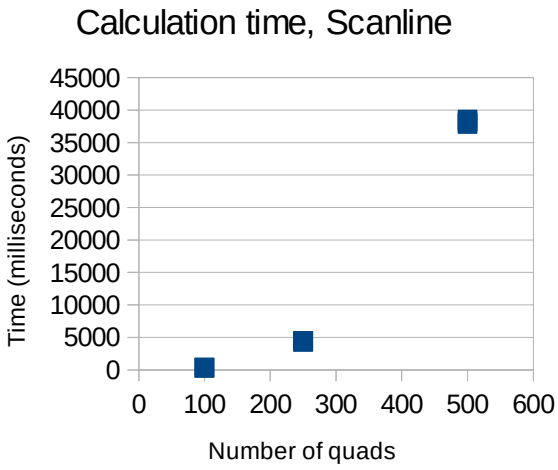

Figure 2: Time requirement for the scanline algorithm

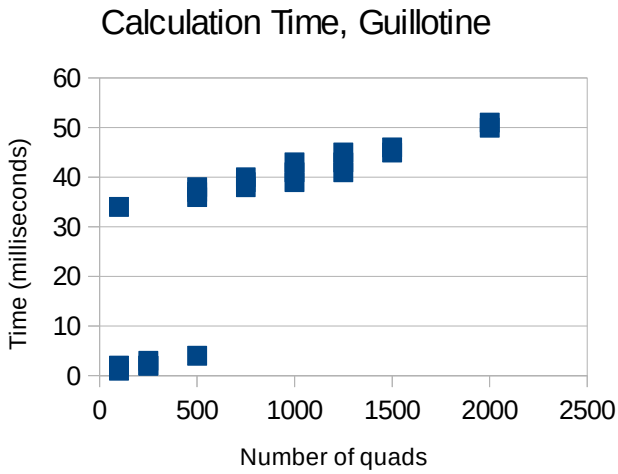

Figure 3: Time requirement for the Guillotine algorithm. The offset depends on Java's dynamic optimisation

While the performance of the MaxRects algorithm is good enough for almost all practical applications, we can still clearly see that the time requirement grows very fast for layouts with over 1000 rectangles. Sorting the array before calculating the layout increases the time slightly, but does not change the profile of the curve.

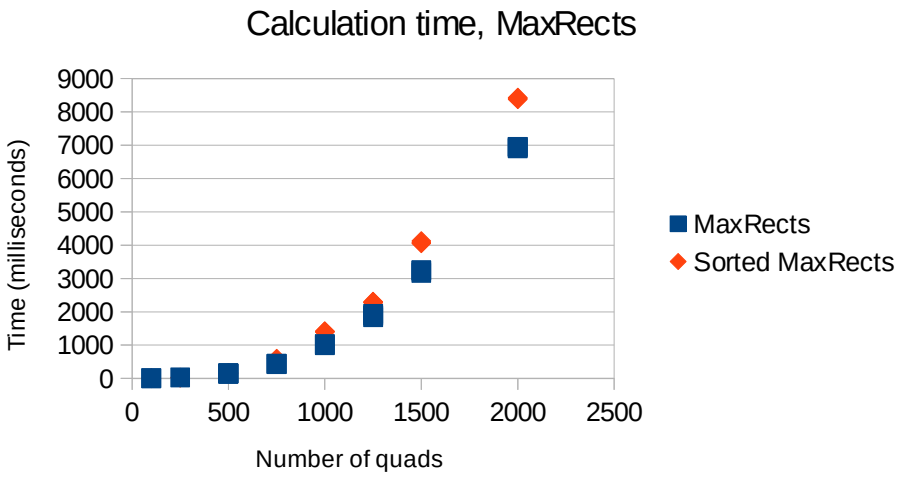

Figure 4: Time requirement for the MaxRects algorithms

Conclusions

As expected, the time requirement of the scan line algorithm increases very quickly, rendering it almost useless for generating large layouts. However, its consistent and good packing ratio at low rectangle counts makes it useful for optimizing small sprite sheets. The MaxRects algorithm remains viable on any layout up to a few thousand rectangles, where its O(N^3) requirement starts to show. Sorting the quads in descending order according to their area before generating the layout improves the consistency of the results, with a minimal impact on computational time. The Guillotine algorithm uses the available space inefficiently at low rectangle counts, but it is able to handle extremely large layouts without problems. Sorting could improve its performance for smaller layouts, but will produce a notable increase in the time requirement since the algorithm itself is so efficient.