# Particle systems:
# They’re cool and we need and excuse to implement quicksort

**Out: Friday May 6th**
**Due: Friday May 13th, noon**

## Overview

In this assignment, you will implement depth-sorting of sprites for transparent rendering in a particle system. We’ll explain what that means shortly, but short version is that you’ll be implementing insertion sort and quicksort in a real application and use metering tools to measure their empirical average- and worst-case performance. As before, we’ve also provided a set of automated tests to ensure that your sort algorithm is properly sorting.

## Particle System Overview

Particle systems are a type of very simple physics simulation used in computer graphics. They were first used commercially by the Industrial Light and Magic to render the “genesis effect” scene from the film *Star Trek 2: The Wrath of Khan*. They are commonly used to produce effects such as explosions, clouds, rain or snow, hair, or realistic fluid flow.

A particle system animates a large number of very simple “particles” which obey very simple laws of motion, typically not including any physical interactions between the particles themselves. For this demo, the particles obey basic physics as if they were in a vacuum (i.e. they have position and initial momentum but never slow down or lose energy). In a perfect world, the particles would also collide realistically with the different obstacles in the world (such as amongst themselves), however, that would be too computationally expensive given the number of particles, so the particles currently pass freely through objects. It’s simple, but it’s enough to provide some simple, engaging motion.

Particle systems are typically rendered (drawn on the screen) by drawing simple 2D images called *sprites* in the appropriate location, rather than drawing a full triangle mesh, as is done for true 3D objects. Graphics cards draw everything in terms of triangles; a rectangle is drawn as two triangles. And so a simple 2D image (a rectangle) can be drawn with just two triangles. So drawing a particle as a 2D image allows it to be drawn very efficiently.

## The painter's algorithm

Particularly for applications such as smoke or flame, these sprites are typically drawn using *transparency*. This means that when we draw a sprite in a particular location, its pixel values don't simply replace the existing pixel values in that location, but instead are composited with the image already drawn in that location, so that the sprite acts as both as a new image, and as a filter through which the previous image is viewed. Complex effects can be achieved by changing the degree of transparency or opacity of different parts of the sprite.

The problem here comes when several particles might potentially be drawn in the same location of the screen. Because of the nature of the compositing, the final pixel color achieved depends on the order in which the sprites are drawn. To properly render the particles, we must use the *painter's algorithm*: objects are drawn back-to-front: with the objects farthest from the camera being drawn first, and then the nearer objects being drawn on top of the farther ones, then the next farthest ones, and so on. This involves sorting the particles by their depth (distance from the simulated camera) so that they can be rendered back to front.

## Depth sorting

We've implemented the particle system for you. The basic objects are already defined, along with the physics simulation and the rendering code. All you need to deal with is in the file Sorting.cs.

Although you should feel encouraged to read the rest of the code, the only thing you really need to know about the Particle class, which not surprisingly represents a particle, is that it has a `depth` field, which gives its distance from the camera. The Particle class is defined in Particle.cs, if you want to take a look at it. But again, you'll only be modifying code in the Sorting.cs file.

All you need to do is to fill in the code for the procedures InsertionDepthSort(), and QuickSortDepthSort() in Sorting.cs. These procedures should sort the particles in the `particles` array by their `depth` fields. You can get the pseudocode for these procedures from either book or from the lecture slides. However, note that:

- The pseudocode for lecture slides assumes you're sorting on the values of the elements themselves (so you say array[i]<array[j]), whereas in this case, you'll be sorting on the values of their depth fields (so you say particles[i].depth < particles[j].depth).

- When swapping elements of an array (e.g. for quicksort), you can do it by saying (assuming we want to swap element i with element j):

  ```
  var temp = array[i];
  array[i] = array[j];
  array[j] = temp;
  ```

  Your initial inclination would be to just say:

```
array[i] = array[j];
array[j] = array[i];
```

However, this won't work because the first line would erase the original value of array[i] before the second line can copy it into array[j].

## Getting started

To get started:

- Unzip the code from the assignment.
- Delete the zip file
- Open up the solution file from within the unzipped directory.
- In the ParticleSorting project, open the Sorting.cs file.

## Implementing the sorts

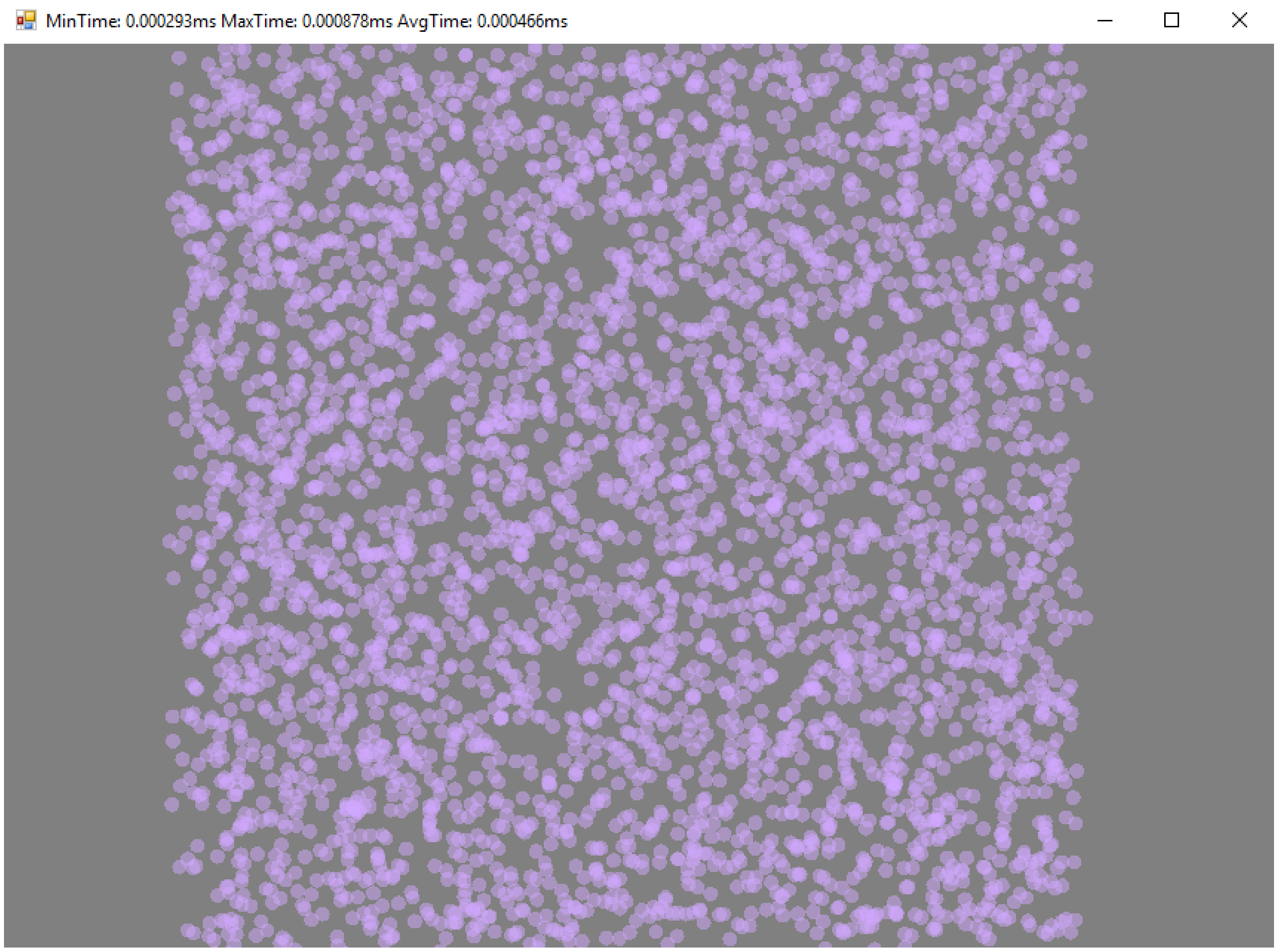
Fill in the code for the methods InsertionDepthSort() and QuicksortDepthSort() inside the Sorting class. The methods are present in the class, but are currently blank. You can test your methods by running the test project, which will run both your insertion sort and quicksort and check that the resulting particle system is properly depth sorted.

For the quicksort routine, you will need to have some way of choosing the pivot element. Normally, you would most likely use randomized quicksort, in which pivot elements are chosen randomly. However, picking a random number can be relatively slow unless specifically optimized to run fast, and using the built-in .NET random-number generator actually slowed my implementation of quicksort down by a factor of 3. Given that the particles will usually be *mostly sorted* already (because the particles move relatively little from frame to frame of the animation), you will probably do fine by always choosing a pivot element in the middle of the range being sorted, that is, by choosing (start+end)/2 as the position of the pivot. You should feel free to experiment with other pivot choices, however, you are not required to do so as part of this assignment.

## Measuring Performance

Once you've debugged your sort algorithms, find the method DepthSort() in the ParticleSystem class, and modify it to use the InsertionSortDepthSort.

When you run the file, you should see something like this:


MinTime: 0.000293ms MaxTime: 0.000878ms AvgTime: 0.000466ms

The purple blob you see on screen is a cloud of 4000 particles. Each particle will have a distance from your viewpoint (your eye). The purpose of this assignment will be to sort these distances in ascending order (from smallest to largest).

At the top of this window should be three numbers: the first number shows the minimum time of sorting all the particles. The second number gives the maximum time of sorting all the particles. The last number gives you the average of the sorting times of all the frames.

Try running the program, let it run for a minute or so and look at the performance. Record the average and maximum execution times using the insertion sort in the table below (Just so you know, the times in the example above are not what you should see!). Now change the code in DepthSort() to use QuicksortDepthSort() instead, and try running the game and examining the timing data again. Again, record the average and maximum execution times. Which sort is faster? Normally, quicksort should stomp insertion sort, however, this is an application where we start each new sort with the particles "mostly" sorted from the previous sort, so you may find that insertion sort works well, at least part of the time. One the other hand, insertion sort may break down seriously if large numbers of particles are changing their depth orderings from frame to frame.

## Table of measured performance data

Fill in the table below with your measured execution times

| | Average ms | Max ms |
|---|---|---|
| Insertion sort | | |
| Quicksort | | |

## Turning it in

First, fill in the table of performance data above and save the file.  Then, as before, to turn the assignment in, you should:

- Make a zip file containing:
  - This file (with the performance data filled in)
  - Your completed Sorting.cs file
- Upload the zip file to Canvas.