# CS 3020 – Final Exam

## Description

For your final exam, you will be modifying a provided project to make it more maintainable and usable. There are two opportunities to earn extra credit by implementing threads and improved collision resolution.

The application is remarkably simple. When the user clicks on the panel, a square will appear with a random velocity and will start moving in that direction. Squares use simple (but buggy, you’ll see!) collision detection to avoid each other and hang out inside the panel. The user can also select the “Add 100” button that instantly adds 100 randomly placed moving squares to the panel. The count indicates how many are currently being rendered.

It is your responsibility to **first refactor** the code and then **add some minor functionality**. It is NOT your responsibility to find or fix ANY of the bugs (except for extra credit) – the collision resolution strategy is remarkably simple, has a few issues and does not represent a production-level algorithm, but you may fix it for extra credit. Other than the “new features” you are adding, the existing functionality should not be changed, just refactored. That being said, if you run into a bug that gets in the way of your development, you should feel free to fix it (but I didn’t purposefully put any of these in).

## Requirements

When you submit your application, it must continue to:

* Compile and run without errors or warnings (it already does, so keep it that way!)
* Retain the same functionality it currently has:
  + User clicks in the panel to place individual squares
  + User clicks “Add 100” to place 100 new squares instantly
  + Count displayed
  + Squares collide and “bounce” off of each other and the walls of panel
  + Color changes to red when squares are in collision
* Use the same GUI components in the same layout as is currently implemented

### Refactoring (30 pts)

Currently, the application was built quickly and poorly – it is essentially unreadable as the code is poorly formatted, has no comments, and uses poorly named variables, methods, and classes.

Refactor the application by (apply ALL where appropriate):

* Reformatting the code to be more readable
* Renaming variables, methods, and classes to be more readable and more easily understood
* Adding methods to modularize reusable code (hint: collision detection and resolution)
* Reduce the length and complexity of methods (hint: do one task and do it very well)

### Object-Oriented Design (30 pts)

Currently, the application is structured essentially in a single file (Form1.cs), uses no modularity or other classes/objects, and has several instances of repeated code.

Modularize the application by introducing classes to represent objects (hint: squares) and modularize Form1.cs as much as possible by moving functionality to other methods or classes (maybe a utility class?) to reduce the file to only immediate GUI-handling activities. The majority of the functionality should be moved into other classes/methods to promote readability and reuse (for example, collision detection could easily be modularized and removed from Form1 – won’t we need to collide other objects in other applications?).

### Improved Functionality (40 pts)

In addition to the above, your application must add new:

* Extend the system to support ANY NUMBER of squares (hint: dynamic collection)
* User clicks on the “+” button to increase the size of the squares by 1 pixel
* User clicks on the “-“ button to decrease the size of the squares by 1 pixel
* User LEFT clicks to add an individual square to the panel at the point where the user clicks
* User RIGHT clicks to remove an individual square from the panel. The square the user clicked should be removed. If more than one square was clicked (i.e. overlapping squares), ALL of the colliding squares must be removed.

### EXTRA CREDIT: Threading (20 pts)

Currently, the application is no longer interactive when the user puts over 1000 or so squares on the panel – the collision detection and resolution algorithm grinds the GUI to a halt, preventing the user from continuing to use it as expected.

Add threads to the application to improve performance by moving the collision detection and resolution loop to its own thread. You should launch only one thread per tick event (not one thread per pair of rectangles). Once threads are implemented, your GUI should be more interactive when there are 1000s of squares in the panel.

You will be evaluated on your use of threads in terms of minimizing the overhead in constructing additional threads, maximizing the usefulness of the thread, and correctly handling shared memory between threads (i.e. locks, barriers, mutexes, etc.).

### EXTRA CREDIT: Fix Collision Resolution (20 pts)

Currently, the application’s collision resolution strategy is poorly implemented. Squares are added to the application without regard for whether they overlap existing squares or not, nor for whether they could even fit within the panel without overlapping (i.e. the sum of areas of the squares would exceed the panel’s area).

Fix the collision issues by:

* Implement any decent collision resolution strategy of your choice.
* Fix the “add 100” square placement to prevent new squares from overlapping existing squares.
* Adjust the size of the squares automatically if the user adds too many squares to fit into the panel. There is still a physical limit to this, but the number of squares will be very high.
* If the user clears all of the squares, they should return to their original size.
* Fix the user-directed square placement by adjusting the position(s) of the overlapping squares to prevent collision.

## Hints

* There are several tools in the Visual Studio package that will help you quickly reformat your code and rename variables. Find them!
* You may find some helpful tips in the course Style Guide