Tanks

Multithreaded Networking

Bradley Rawson and Andrew Mellen

Department of Computer Science and Networking

Wentworth Institute of Technology

Boston, MA 02115, USA

[rawsonb@wit.edu](mailto:rawsonb@wit.edu) [mellena1@wit.edu](mailto:mellena1@wit.edu)

***Abstract—*** Tanks is a game where someone hosts a server, anyone with the game can connect, and all clients on the server can play against each other. The main objective is to shoot the other players’ tanks.

***Keywords—*** volatile, synchronized

1. **Introduction**

Tanks is a project that took everything that we learned in class and took it to a new level. Creating a network program with so many different pieces and clients requires multithreading, and a lot of it. Dealing with so many processes happening at the same time requires a lot of forethought (and future debugging) that is nowhere close to a single threaded application. Tanks uses threads to read and write data on every client and the server, which is complicated enough at times. But adding in the complexity of writing the images to the pane and moving around the images in a chaotic mess of information intake makes it very difficult.

Object Oriented Programming helps a lot with breaking up this complexity, along with a lot of built in features of Java. Object Oriented Programming breaks the problem up into multiple pieces, which helps organize the different tasks and issues very well.

One major lesson to take away from this project is the usefulness of the volatile and synchronized keywords in Java. Multithreading without them is almost impossible, due to the nature of having multiple threads.

One other new topic that was used extensively in this project was Java’s ability to send objects over a network. This is accomplished using ObjectOutputStreams and ObjectInputStreams between the Server and Clients. The idea was to make the sending of information back and forth easier, but in some occassions it had the opposite effect.

1. **Problem**
2. *Definitions/Problem Formulation*

Create a networked tanks game that has the ability to support multiple players. The game should be aesthetically pleasing and as bug free as it can be. The game should be enjoyable to play and feel like a finished product, not something thrown together. It should have nice finishing touches that give the game more customization options that give it the finished “feel”.

1. *Sub-Problems*
2. *Multithreading* – Keeping all of the threads from messing each other up was a challenge, something that was not expected as much going into the project.
   1. *Looping in threads* – having loops created too much CPU usage in certain situations. One situation caused each individual client to use around 40% CPU usage
   2. *Using the same streams simultaneously accidentally* – Using a stream in two different threads to send and receive data at the same time corrupts the data, therefore ruining the server/clients
   3. *Emptying of queues before they were ready* – two threads simultaneously trying to pull from a queue can cause the queue to empty out when it’s not expected to
3. *Sending and receiving objects* – Going in to the project, sending of objects was going to be a feature that would make the networking much simpler and more concise code. Once it was coded up, it created its own problems that were unexpected.
4. *Updating the tanks on the pane* – Any missed leave message would have a tank stick around on the pane forever
5. *Collision* – Java’s built in collision mechanisms for JavaFX are subpar. Custom methods had to be made to determine whether or not an object was colliding with something else
6. **Solution**
7. *Abbreviations and Acronyms*
   1. CPU – Computer Processing Unit
8. *Figures and Tables*



Fig 1. Utilizing volatile to add something new to the sendingQueue. addNextNode is also synchronized to ensure that two items aren’t be added simultaneously

Fig. 2 Sleeping a thread to make sure a loop doesn’t continuously run when it is not needed



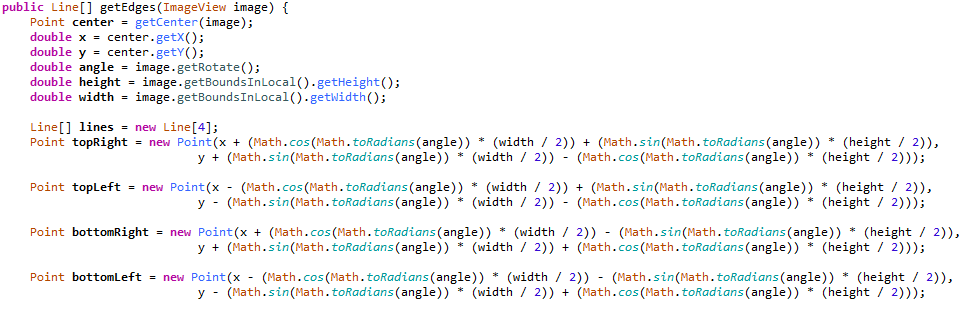


Fig. 3 Calculating the points of a rectangle knowing the center point

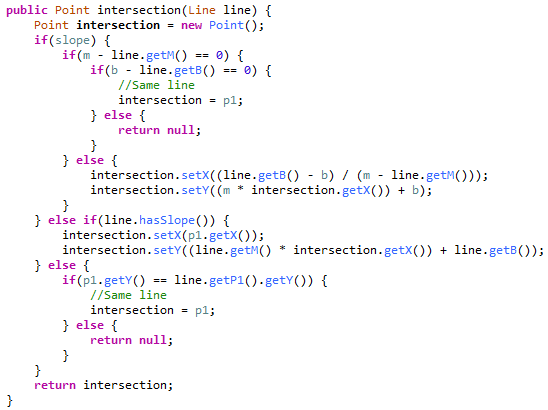


Fig.4 Finding the intersection of two lines

1. *Sub-Problem Solutions*
2. *Multithreading* – The multithreading issues were solved with a lot of debugging and the newfound use of the Java keywords volatile and synchronized.

Volatile ensures that the Java runtime will read whatever variable is set as volatile from the ram every time it is used. This solves the issue of a different core holding the value in cache and not reading in a change from a different thread. This was a major problem at first, but once volatile was used, it fixed a lot of the problems. (See fig.1)

Synchronized makes sure that two methods aren’t used simultaneously in two threads, which fixed Sub-Problem i.c. Synchronized guaranteed that the queue didn’t try to pull multiple items simultaneously.

* + 1. *Looping in threads* – This was fixed by sleeping the threads (see fig.2) when it didn’t have anything to do. This stops them for doing something every clock cycle for no reason and taking up too much CPU usage.
    2. *Using the same streams simultaneously accidentally* – This was fixed by adding everything that had to be sent into a queue, as the thread sending data will just go through that queue to send packets. Trying to call it to send something will cause it to send two things to the same client at once without a queue.
  1. *Sending and receiving objects* – Sending and receiving objects using Java rather than sending primitives proved to be quite the challenge at first. Objects only send their data if the class implements Serializable, which proved to be a problem for some built in classes that the program needed to have sending. It was fixed by having some of the parent class’s variables stored in the child class as well.
  2. *Updating the tanks on the pane* – This problem was a very inconsistent one which made it difficult to pin down to one key problem. The end solution was the same as in i.b, as not receiving bad packages should keep the updating of tanks stable.
  3. *Collision* – As mentioned above, JavaFx’s intersection method was very lacking. It didn’t deal with the rotation of a Node at all, which was very important in a game that you are rotating constantly. The solution was to implement a new collision and intersection method for the Tank. It calculates the corners of the rectangle based on the rotation using trigonometry (see Fig.3), and then makes lines for all of the edges. From there it calculates intersects between one tank’s edges and another’s edges (see Fig.4), and then sees if the intersection point is on both line. This proved to be a great collision method, much better than the built in one, which just made bigger hit boxes on rotate.

1. **Conclusion**

This was a very enjoyable project to work on. It was something that both Brad and I were very passionate about and wanted to see working well. The greatest accomplishment we can have coming out of this is if people actually play it and want to play it, which hopefully they will.

Although there were problems throughout the creation of this game, they were problems that taught us a lot about multithreading and networking, which made the project worth it. We both have countless takeaways from this project, and because of that we consider this project a huge success.