Comparison of Traffic Simulation Algorithms

CSC212 – Introduction to Software Engineering

Instructor Professor Loi

Luke LaBonte

Development Cycles

- Abstract: What is the best algorithm for traffic simulation?
- Analysis: Compare two popular traffic simulation algorithms, Monte Carlo and Cellular Automata
- Design: Create implementation for the algorithms, and compare relevant statistics
- Implementation: Use Godot engine for graphics and C# for logic
- Deployment: Run simulations in application, collecting relevant statistics

Abstract

- What is the best way to analyze traffic?
  - Raw data is often too complex for numerical analysis
  - Simulation is a better approach
- Which simulation algorithm do we choose?
  - Two popular algorithms are Monte Carlo and Cellular Automata
  - Different in how they function, but both produce similar data
- How do we compare these algorithms?
  - Computation time, memory usage, and accuracy to real world data are good measuring points to compare the algorithms.

Analysis

- How do the algorithms work?
    - Monte Carlo
        - Essentially a probability-based system dynamics model. Flow is restricted by road capacity and random delay, while stock is increased by a constant flow of cars entering the system.
    - Cellular Automata
        - Each car can drive itself and make its own decisions. In essence, cars will accelerate if able to do so, and slow down if there are other cars around them trying to merge.
- How will this be implemented?
    - The Godot engine is used for graphics, while C# is used to create an implementation for the algorithms.

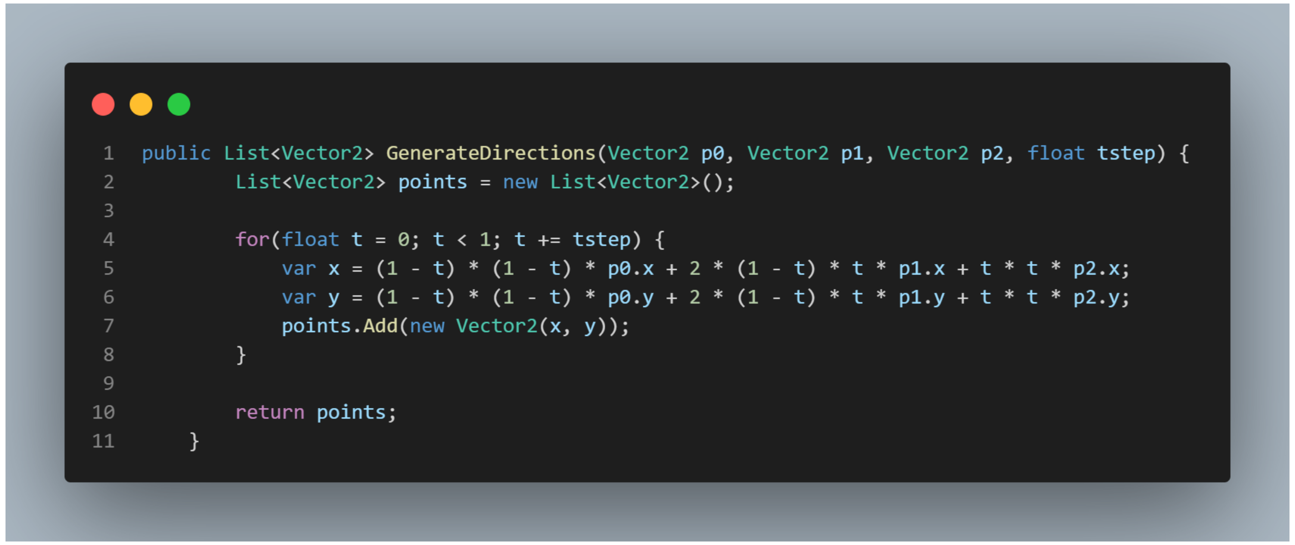
Design

- Overall design is very simple
  - Intersections are represented by graphs, where edges are roads and nodes are where those roads connect
  - Both algorithms can be run efficiently with this data structure
  - Intersections can be easily serialized into a JSON file
- This raises a few complications
  - How do we run pathfinding for cars? With bigger intersections or more cars, this will be very expensive
  - How do we deal with curved roads? Impractical to store all positions along a curve
  - For Cellular Automata, how do we get positions of all other cars?

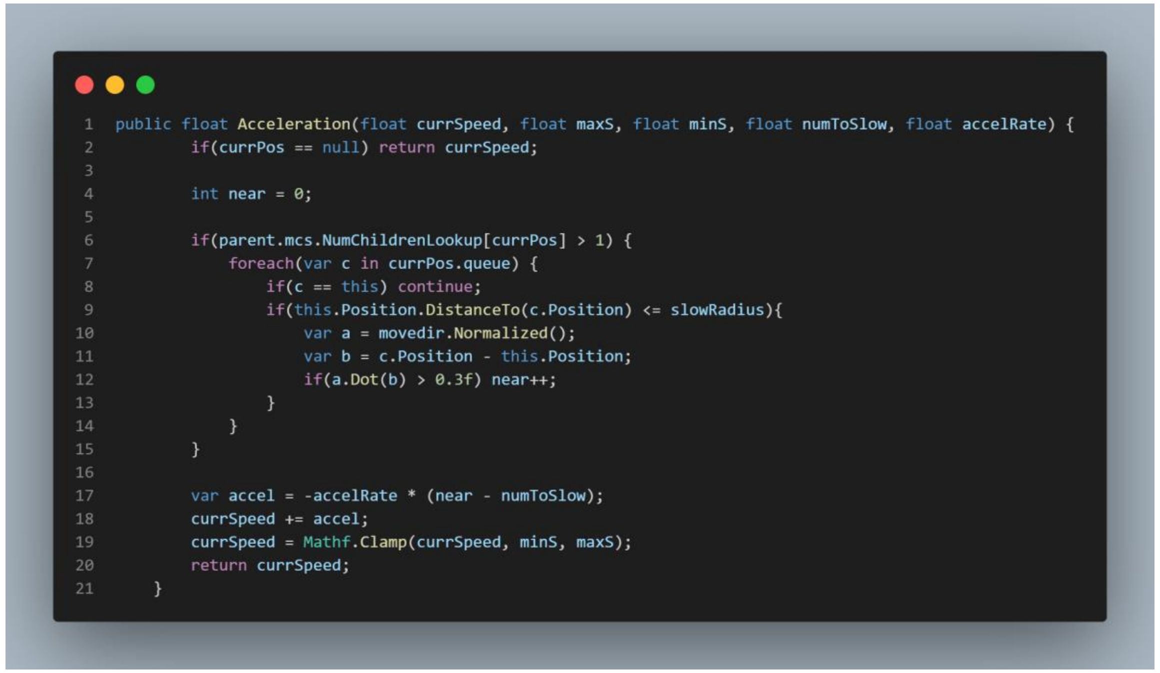
Design (contd.)

- Pathfinding
  - The best solution is to precompute all paths
    - For every input node, calculate shortest path to every output node
    - Store paths in dictionary for fast lookup
- Curves
  - Two solutions: Either precompute all path points, or use Bezier curve formula to dynamically compute points
    - Computing points dynamically is fairly cheap, and only requires 10 iterations per run
    - Precomputing points would be faster, but tradeoff isn’t worth memory usage
- Getting position of other cars
  - Running through every car in the intersection is extremely expensive and impractical
  - Instead, each road can have a list of references to which cars are on it
    - Low memory overhead (just storing pointers) and greatly reduced computing power, as cars only need to check the positions of the cars on the same road

Implementation

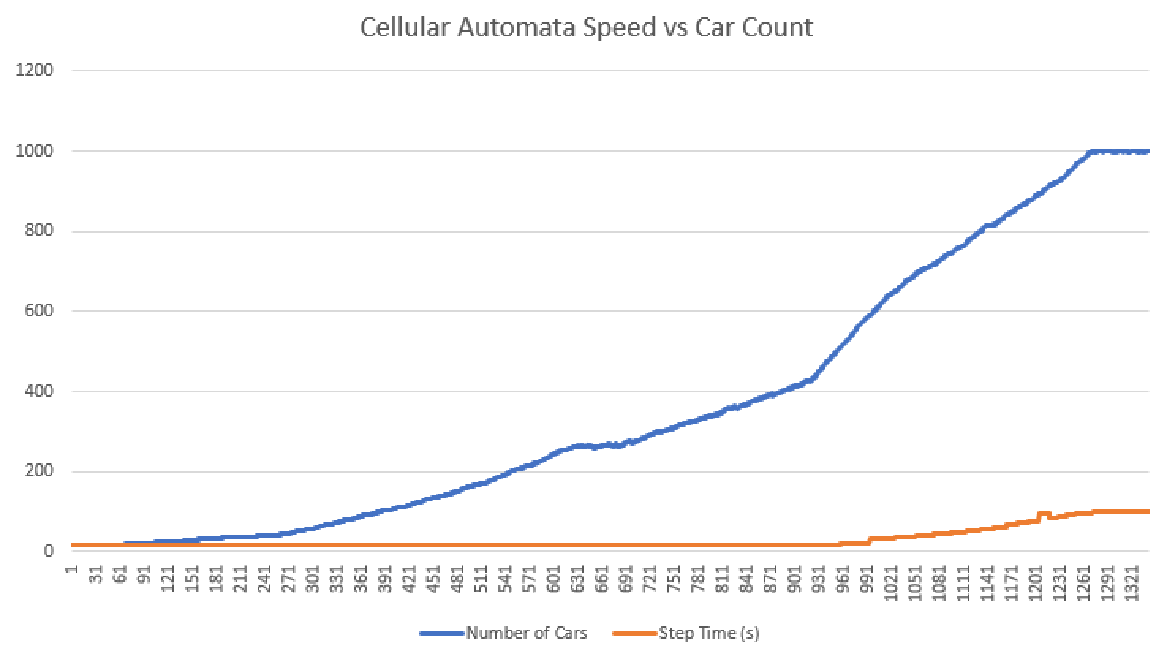


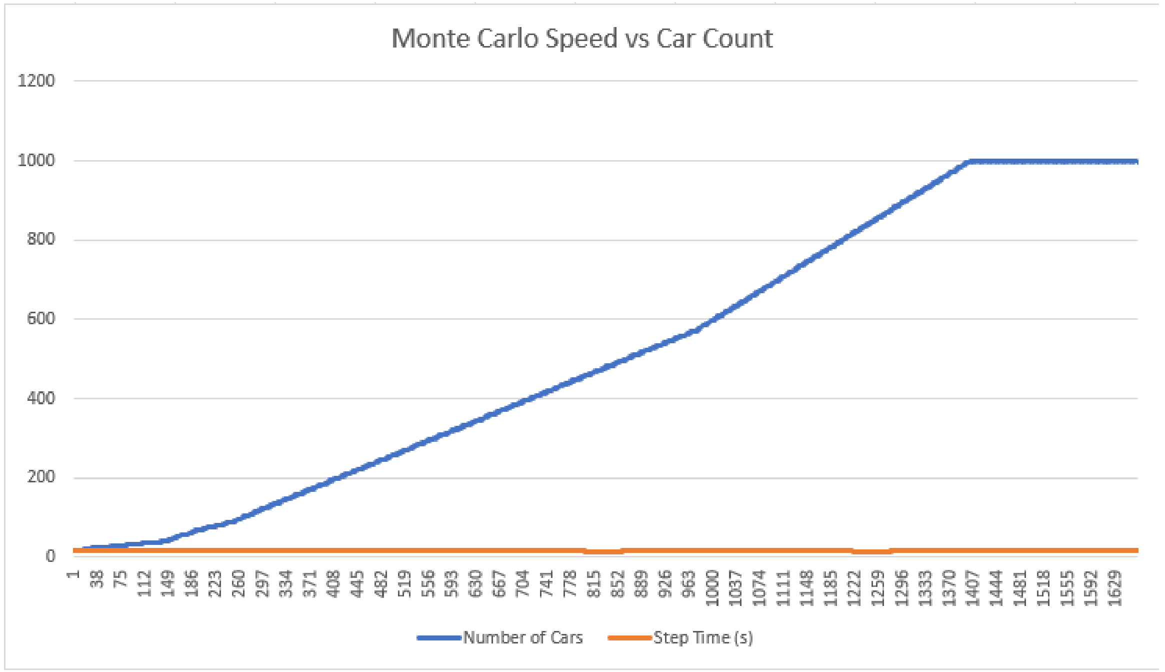
```
public List<Vector2> GenerateDirections(Vector2 p0, Vector2 p1, Vector2 p2, float tstep) {
        List<Vector2> points = new List<Vector2>();

        for(float t = 0; t < 1; t += tstep) {
            var x = (1 - t) * (1 - t) * p0.x + 2 * (1 - t) * t * p1.x + t * t * p2.x;
            var y = (1 - t) * (1 - t) * p0.y + 2 * (1 - t) * t * p1.y + t * t * p2.y;
            points.Add(new Vector2(x, y));
        }

        return points;
    }
```



```
public float Acceleration(float currSpeed, float maxS, float minS, float numToSlow, float accelRate) {
        if(currPos == null) return currSpeed;

        int near = 0;

        if(parent.mcs.NumChildrenLookup[currPos] > 1) {
            foreach(var c in currPos.queue) {
                if(c == this) continue;
                if(this.Position.DistanceTo(c.Position) <= slowRadius){
                    var a = movedir.Normalized();
                    var b = c.Position - this.Position;
                    if(a.Dot(b) > 0.3f) near++;
                }
            }
        }

        var accel = -accelRate * (near - numToSlow);
        currSpeed += accel;
        currSpeed = Mathf.Clamp(currSpeed, minS, maxS);
        return currSpeed;
    }
```

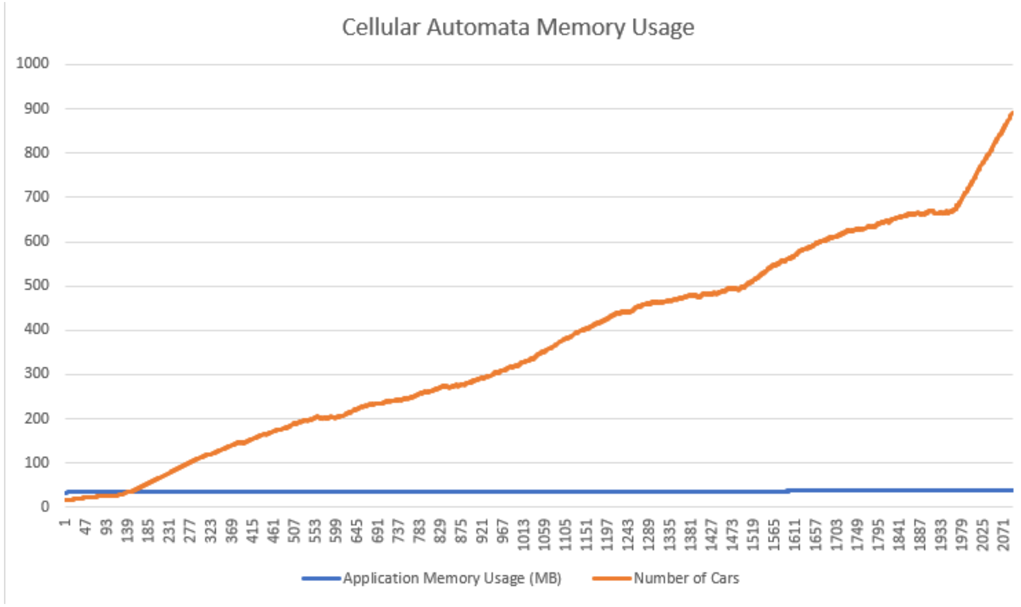
Demo

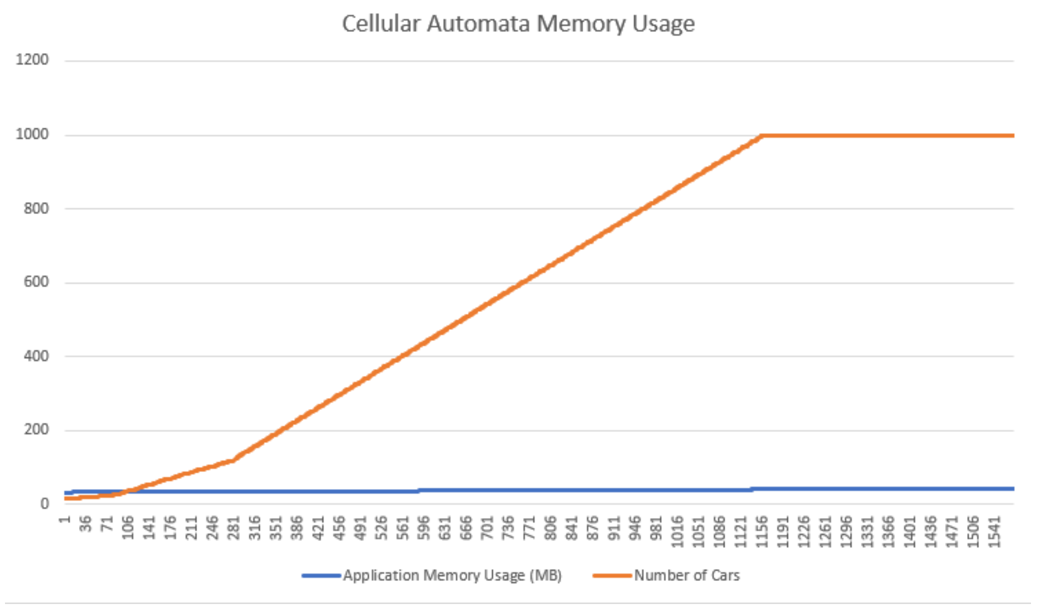




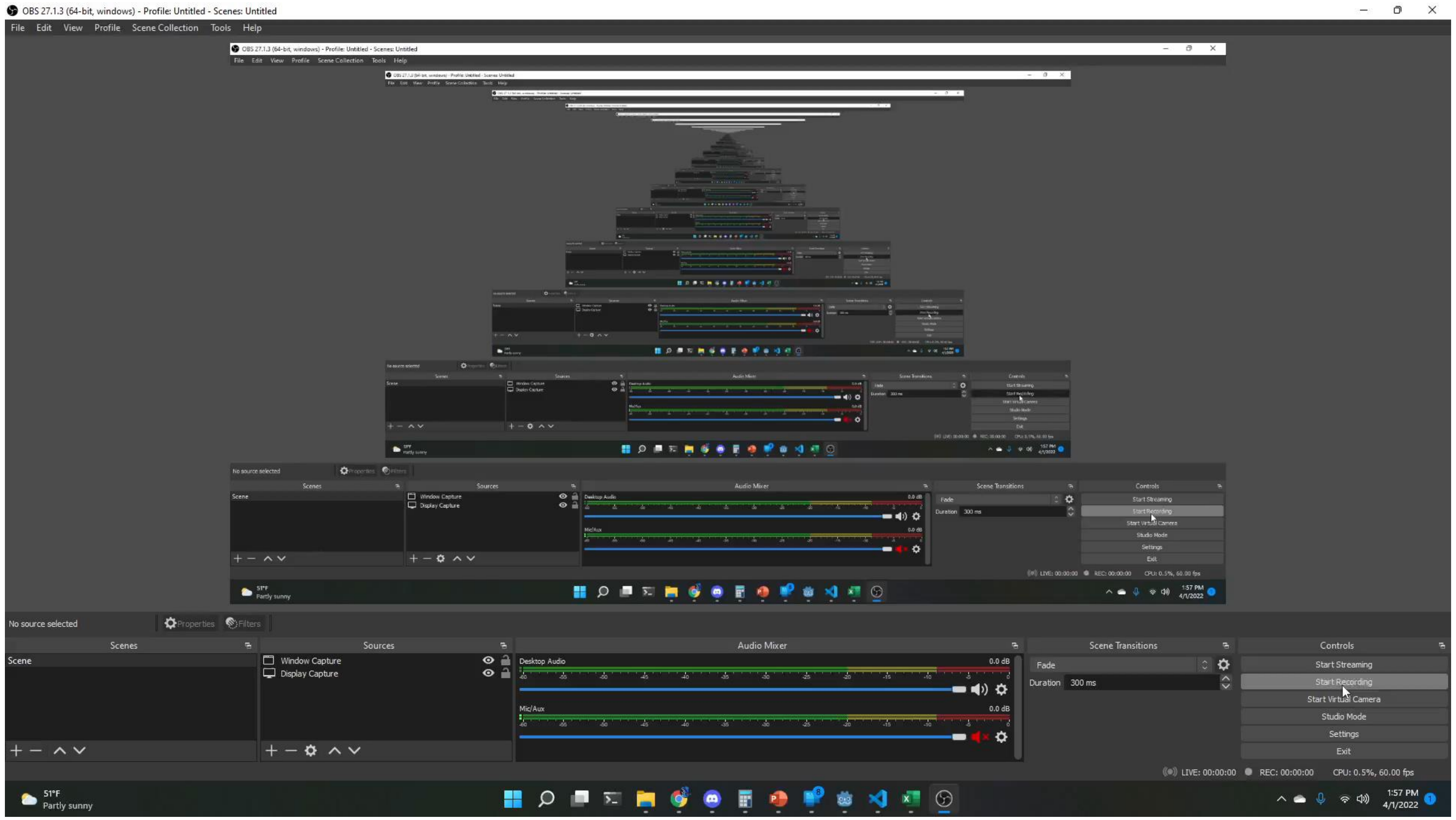
Clearly, Monte Carlo has significantly higher performance, because AI does not have to be calculated for every car on every frame.

Demo (contd.)





Both algorithms have similar memory usage, each only needing to allocate a maximum of 7MB.


OBS 27.1.3 (64-bit, windows) - Profile: Untitled - Scenes: Untitled

Scene Transitions: Fade, Duration 300 ms

Desktop Audio 0.0 dB

Mic/Aux 0.0 dB

LIVE: 00:00:00 REC: 00:00:00 CPU: 0.5%, 60.00 fps

Questions