InnoGames

BRING LARGE-SCALE TRAFFIC SIMULATION TO MOBILE WITH DOTS

Hai Nam Pham

AGENDA

01 Traffic Simulation Demo

02 Design in DOTS

03 Path finding & Handing Collision

04 Best Practices & Pitfalls

05 Questions & comments

01

TRAFFIC SIMULATOR DEMO

BASICS: INTERSECTIONS

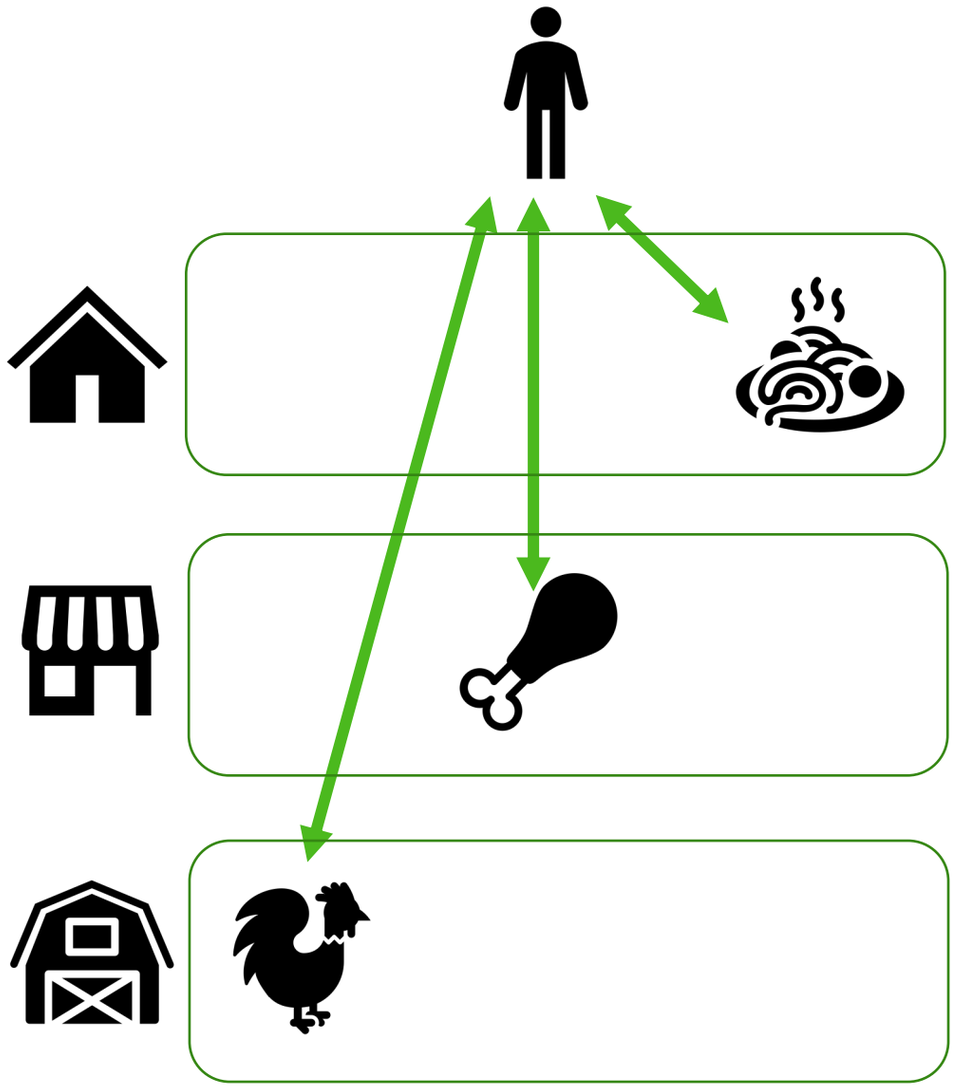
INTERMEDIATE: ROUNDABOUTS

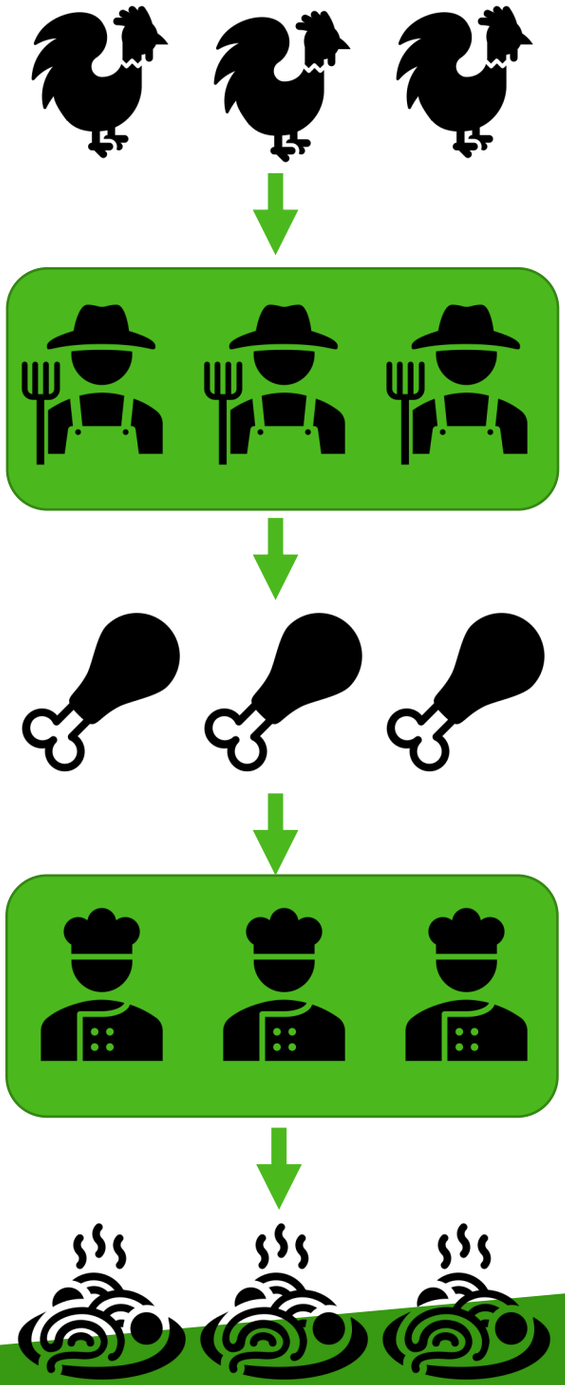
ADVANCED: HIGHWAY BUTTERFLY INTERSECTION

complexity + scale

DESIGN IN DOTS
(DATA ORIENTED TECH STACK)

DOTS: PARALLEL PROCESSING





DOTS: ENTITY COMPONENT SYSTEM

Entity

- Objects in the game world
- Contains components
- Just an index

Component

- Entity's data
- Game state
- Struct of simple types

System

- Filter entities by component types
- Make changes to components & entities

ENTITIES & COMPONENTS EXAMPLES

Vehicle entity

- Length
- Current connection
- Coordinate
- Move distance
- Destination

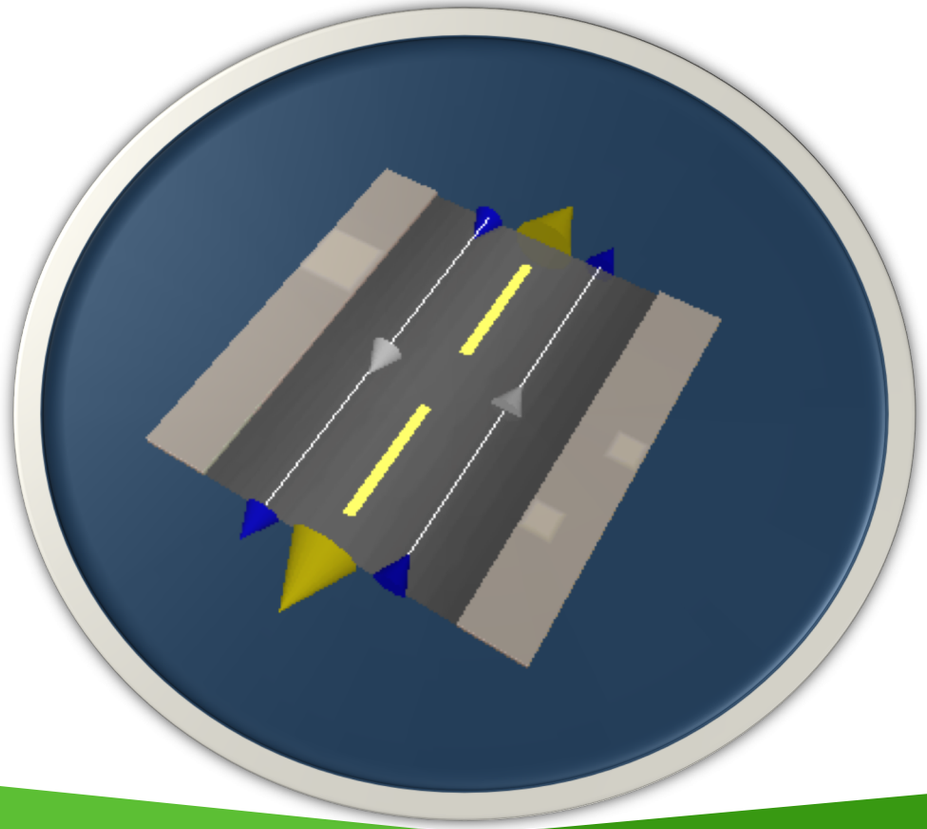
Connection entity

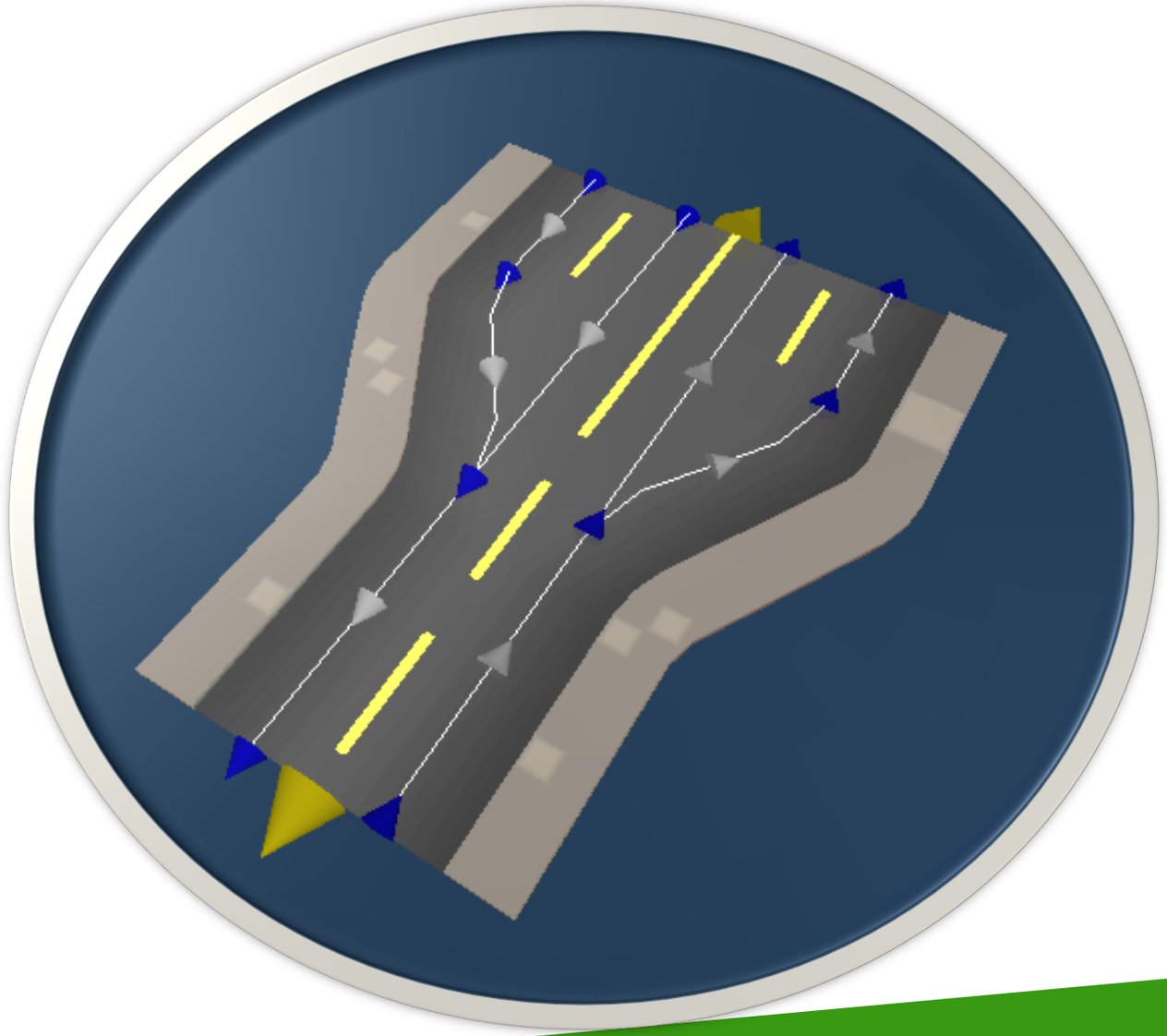
- Start node
- End node
- Length
- Enter space
- Vacate space
- Allow entrance
- Roadsign

ROAD SEGMENT

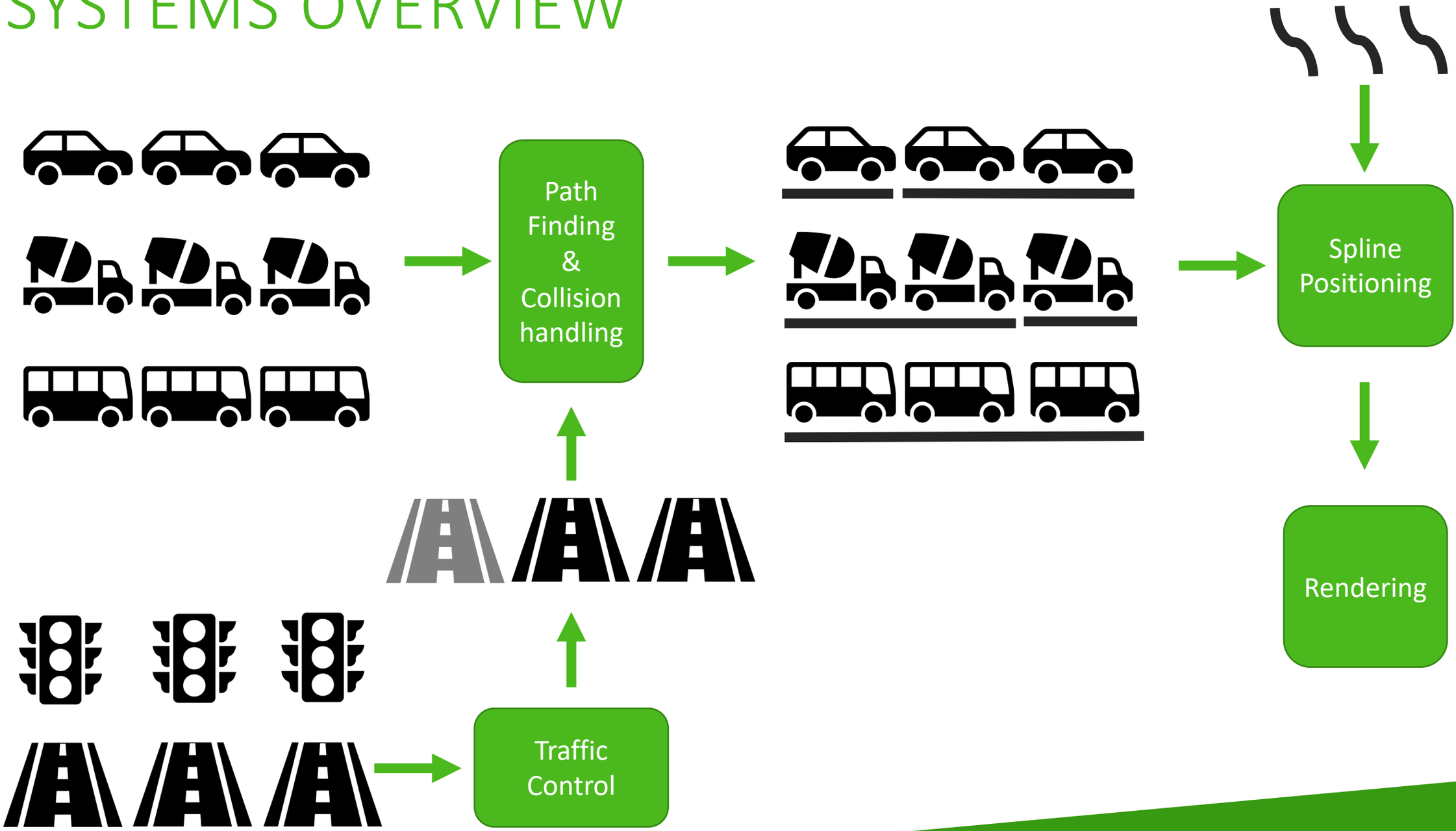
Road segments are modelled as:

- A set of connections
- Each connection is one-way
- Each connection is an entity





SYSTEMS OVERVIEW



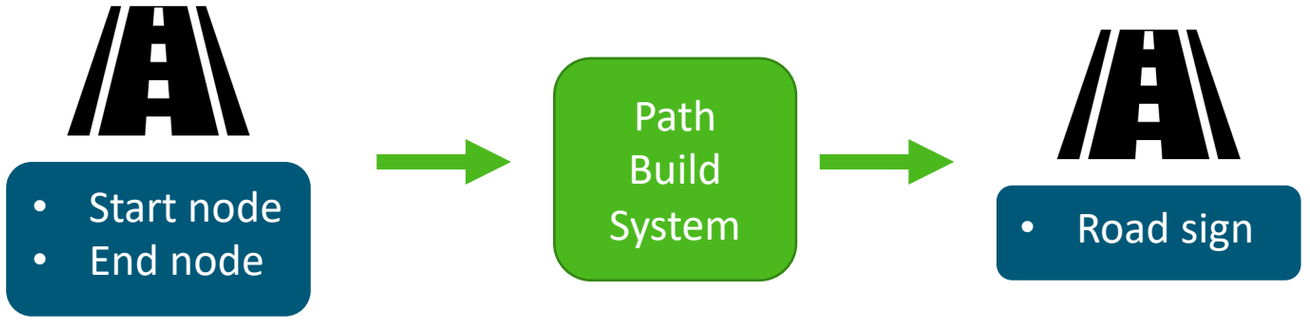
03

Path finding & Handling Collision

PATH FINDING

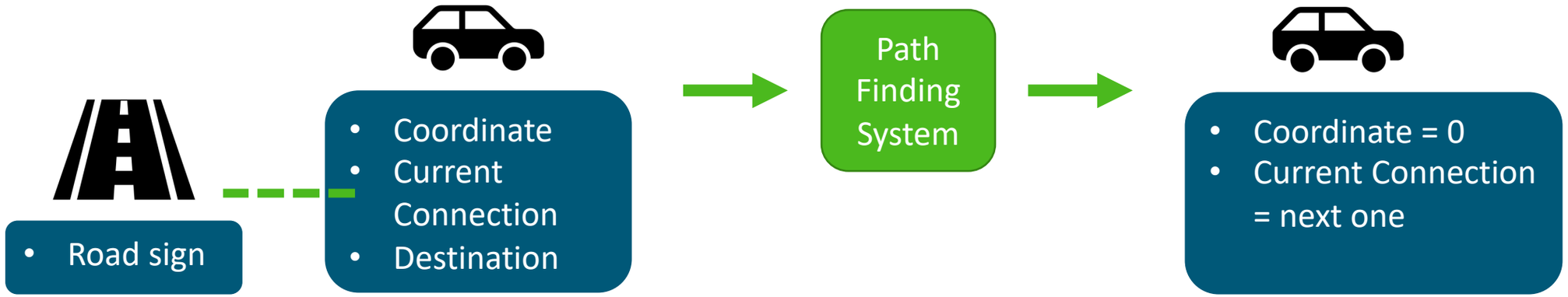
- Coordinate < Length[Current Connection]

PATH FINDING



- Computed once
- Floyd-Warshall algorithm
- Road sign stores
  - For each destination
  - The next connection toward that destination
  - Example: airport: turn left, city-center: turn right

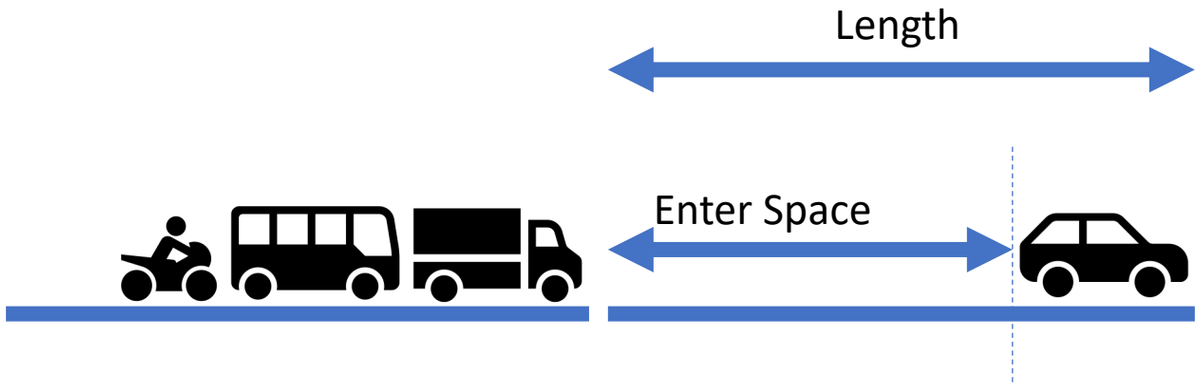
PATH FINDING

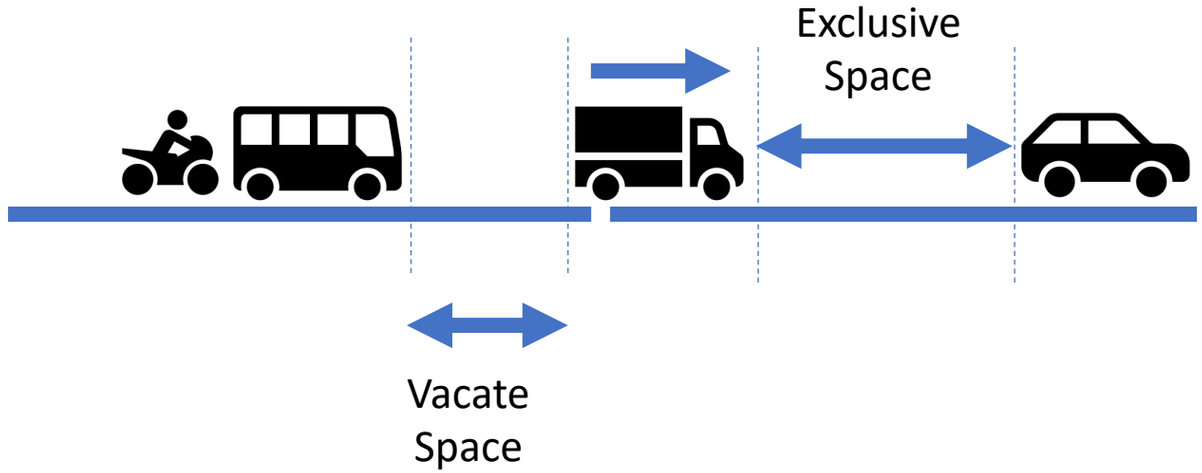


- Coordinate == Length[Current Connection]

COLLISION WHEN ENTERING A CONNECTION

Compute vehicle's Move Distance
→ Make sure it won't collide with the vehicle in front

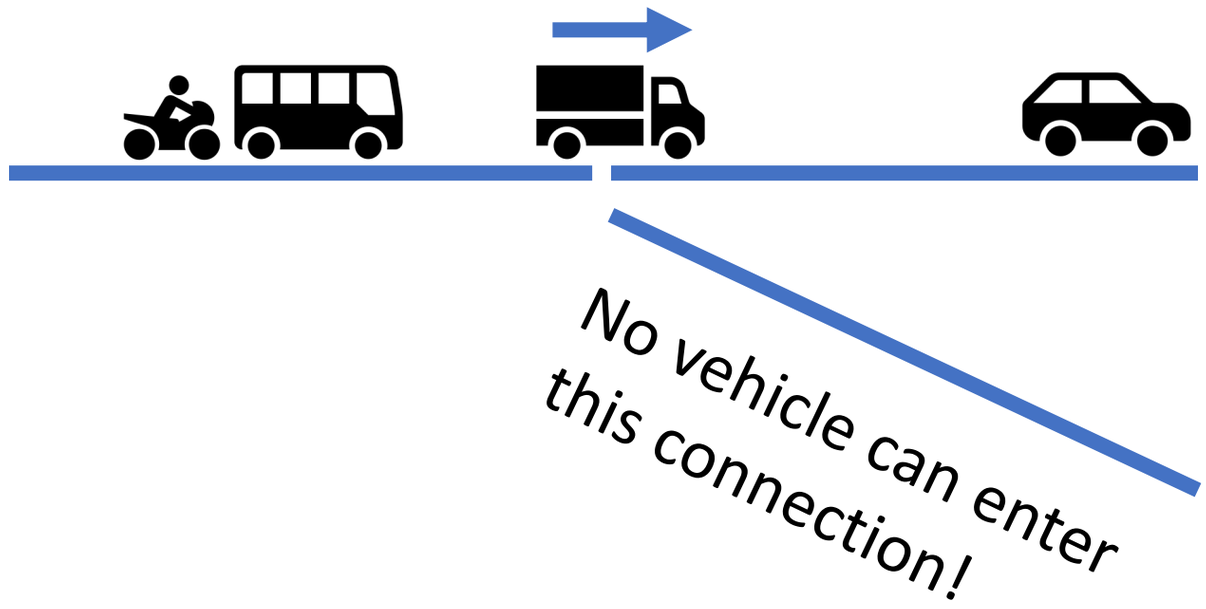




Connection

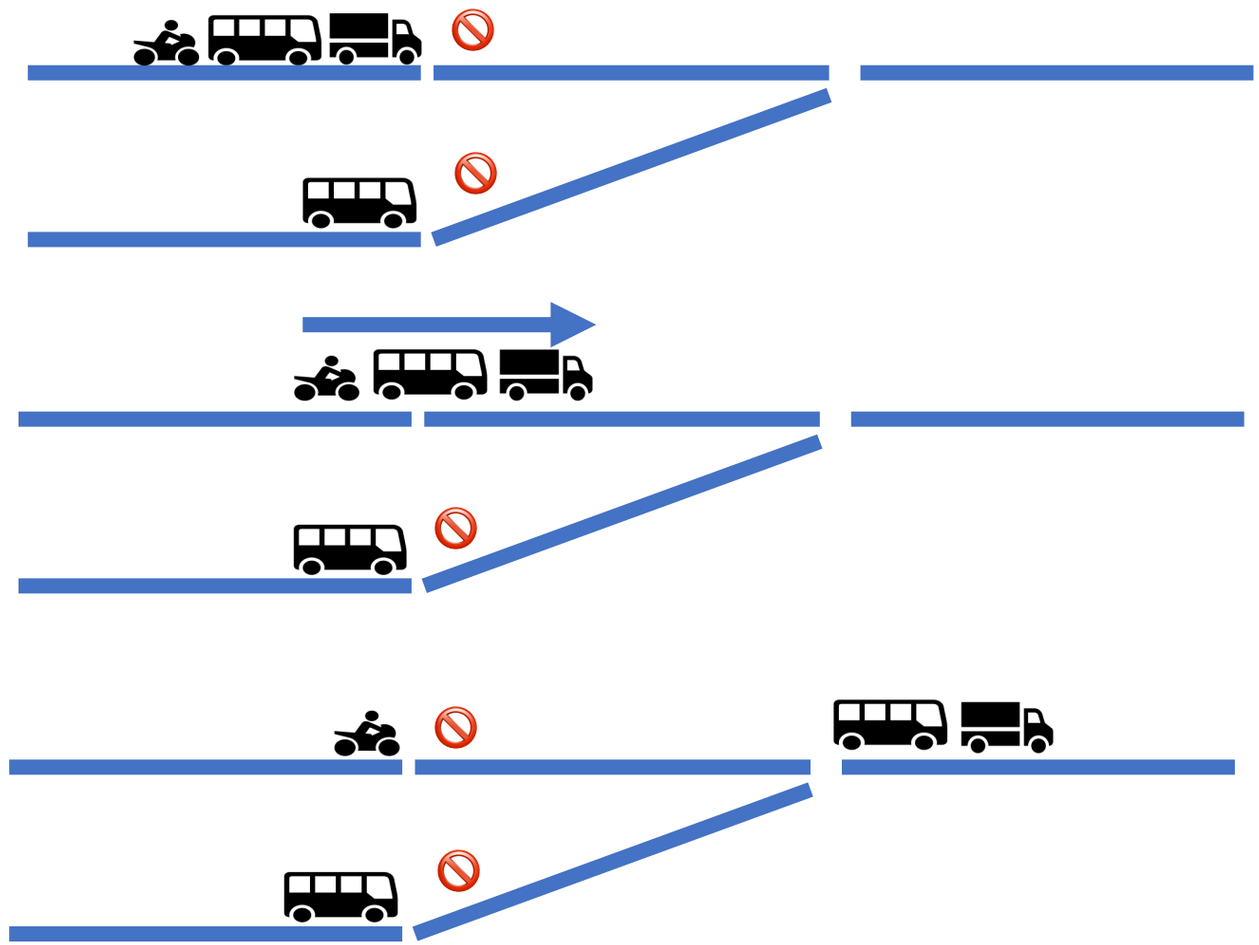
- Length
- Enter Space
- Vacate Space
- ~~Exclusive Space~~

COLLISION WHEN ENTERING A SPLIT

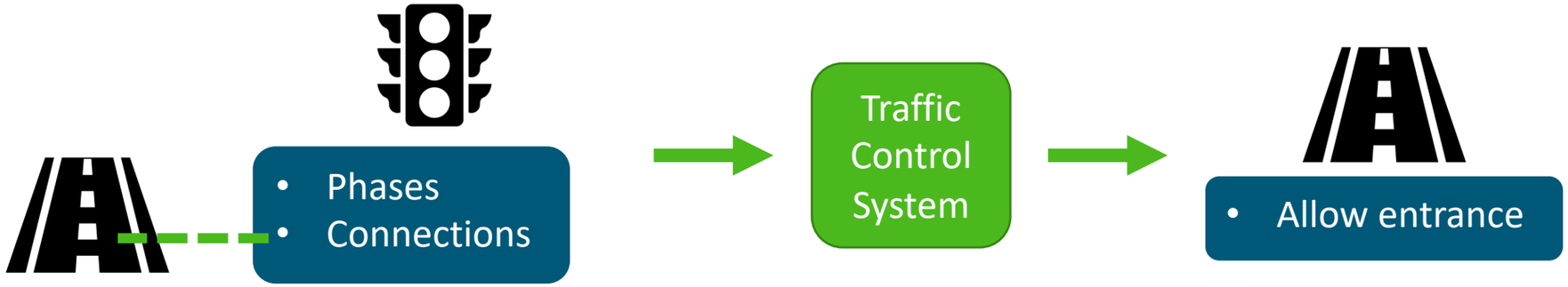


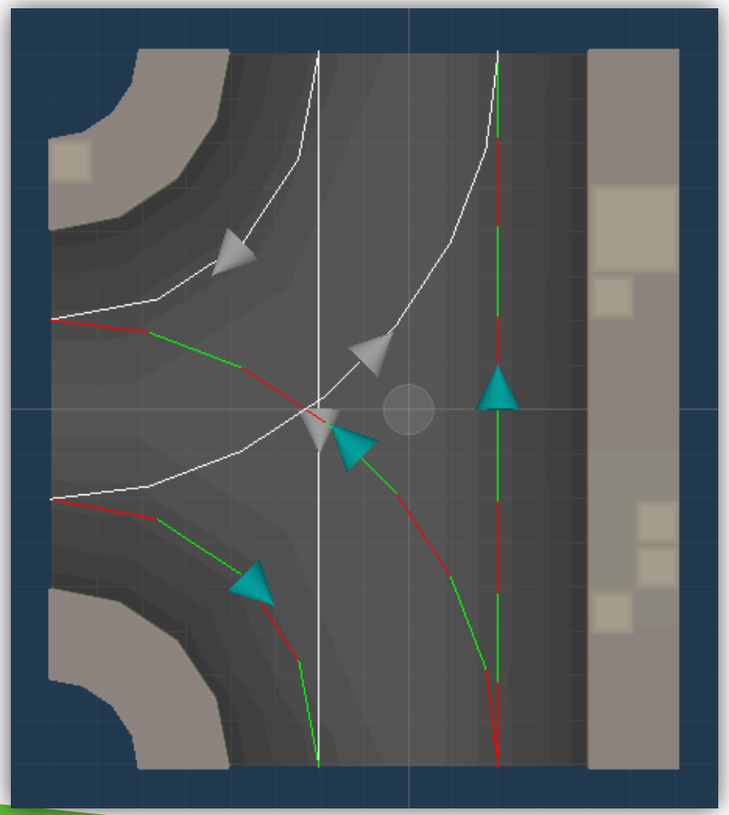
COLLISION BETWEEN MERGING CONNECTIONS

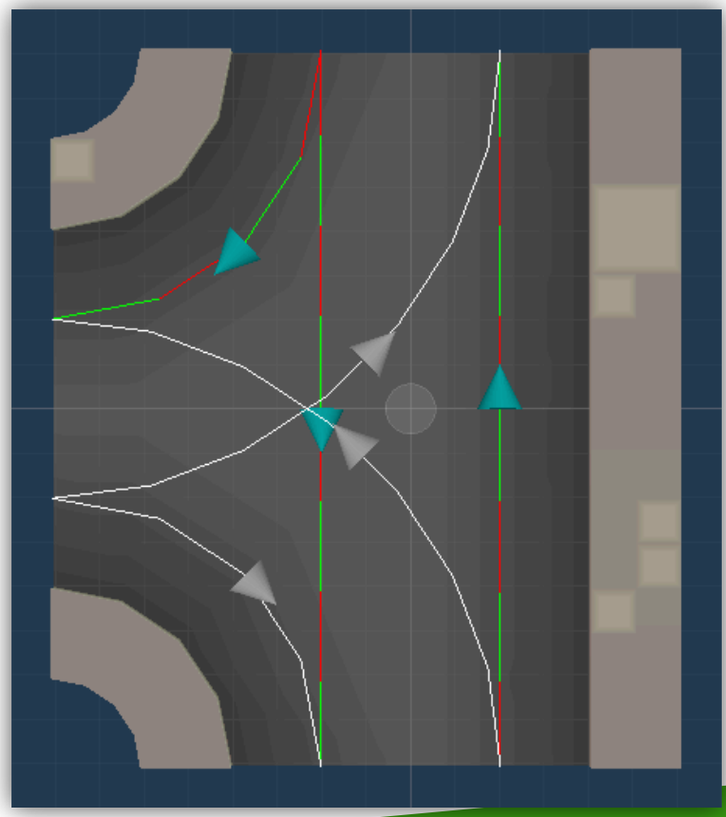
- Use traffic control system
- Multiple phases
- 3 states per phase:
  - Blocking lanes
  - Taking in vehicles
  - Clearing vehicles

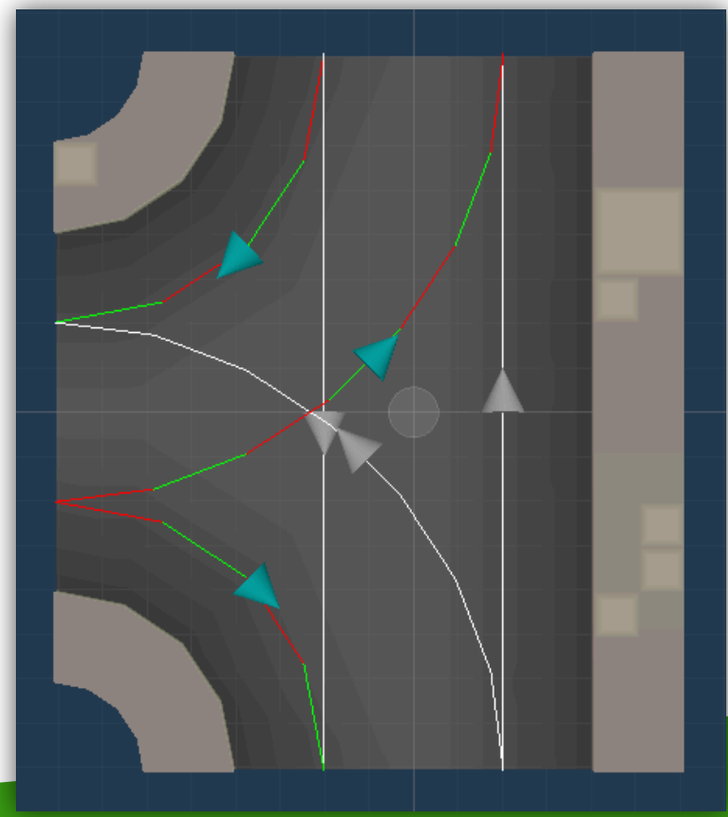


TRAFFIC CONTROL

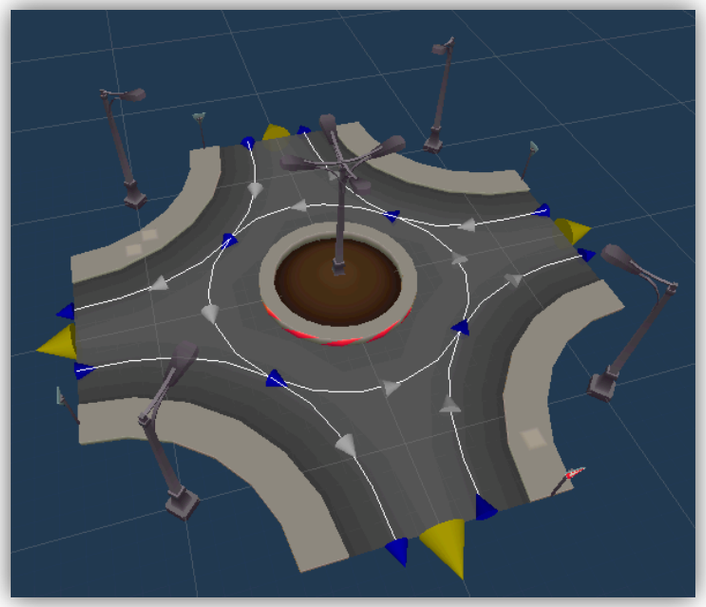


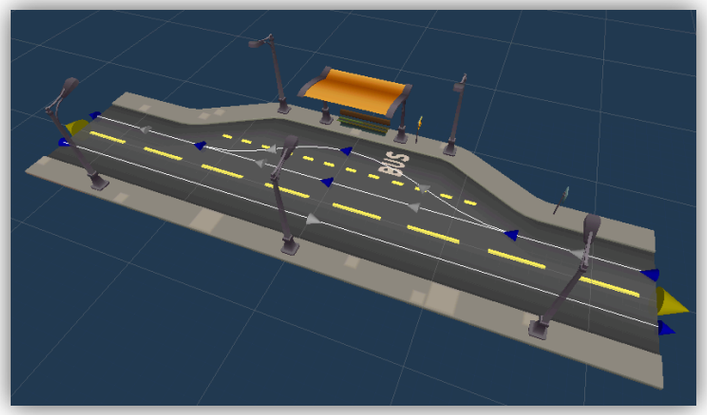


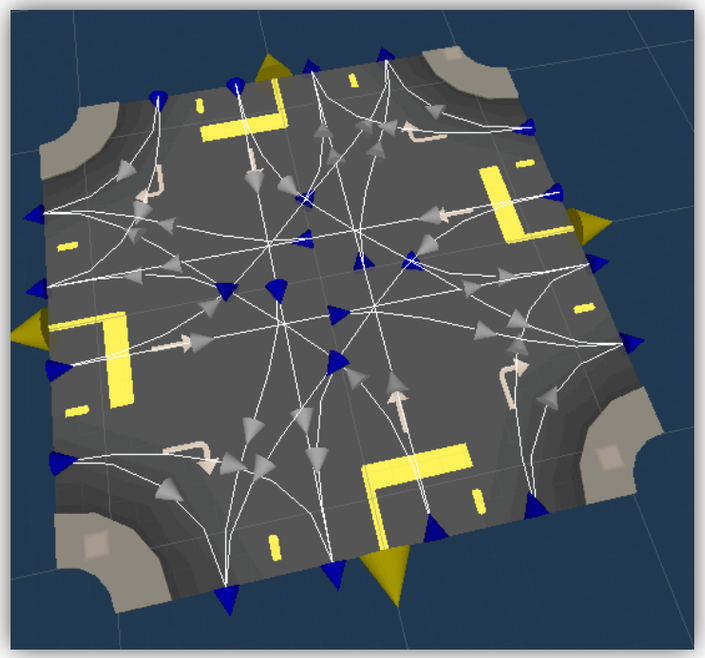




TRAFFIC CONTROL







Best practices & Pitfalls

DOTS: DATA ACCESS

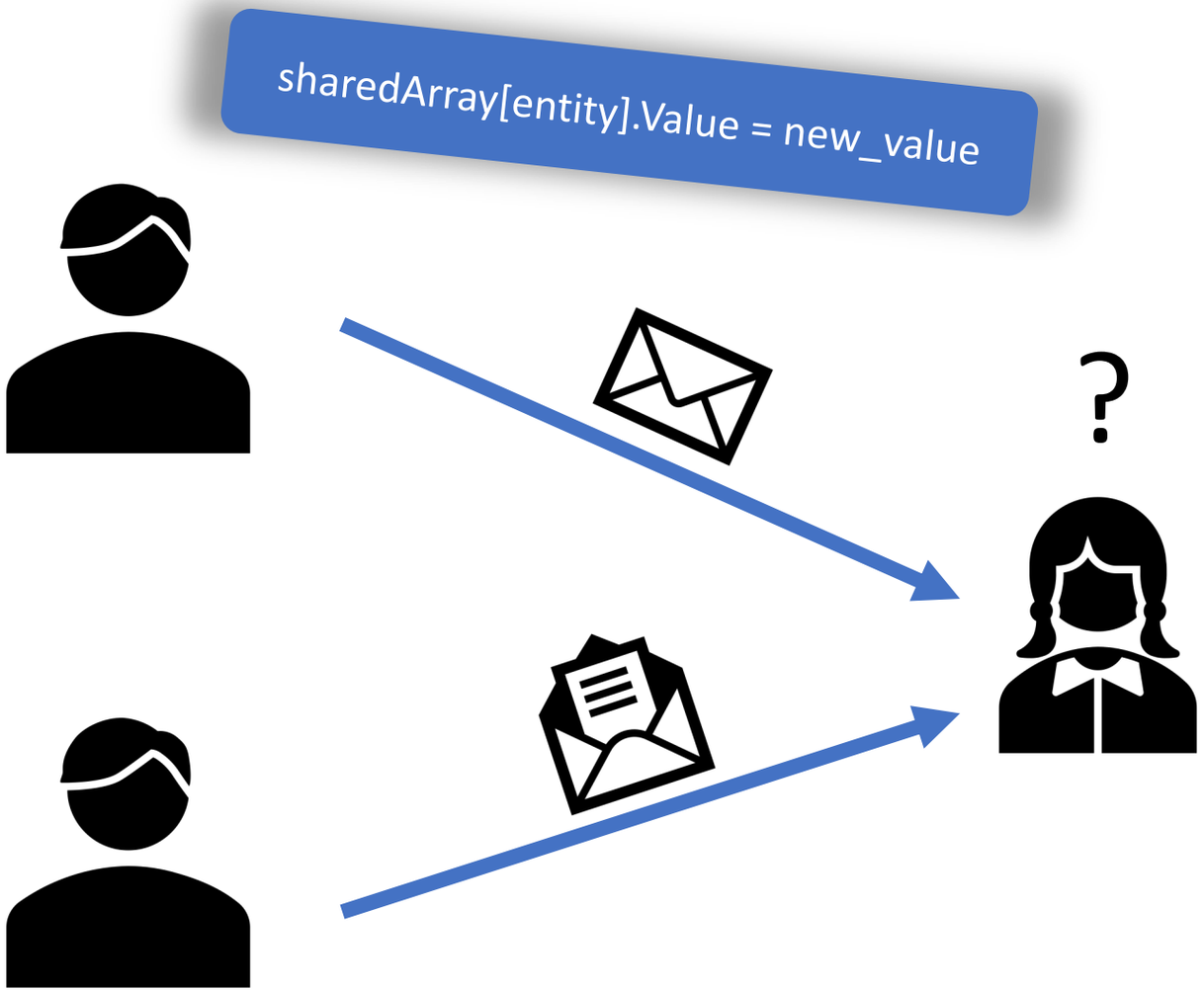
```
private struct VehicleMoveJob : IJobForEachWithEntity<Vehicle, VehicleCoord>
{
    [ReadOnly]
    public ComponentDataFromEntity<RoadLength> RoadLengths;

    [NativeDisableParallelForRestriction]
    public ComponentDataFromEntity<RoadState> RoadStates;

    public void Execute(Entity entity, int index,
        [ReadOnly] ref Vehicle vehicle, ref VehicleCoord coordinate)
    {
        coordinate.Value += vehicle.Speed;
        var roadState = RoadStates[entity];
        roadState.VacateSpace = math.min( x: RoadLengths[entity].Value,
            y: vehicle.Speed);
        RoadStates[entity] = roadState;
    }
}
```

- Exclusive data: jobs will be scheduled around these → no race condition
- Read-only shared data → safe for reading
- Shared data → potential race-condition → the most interesting things happen here

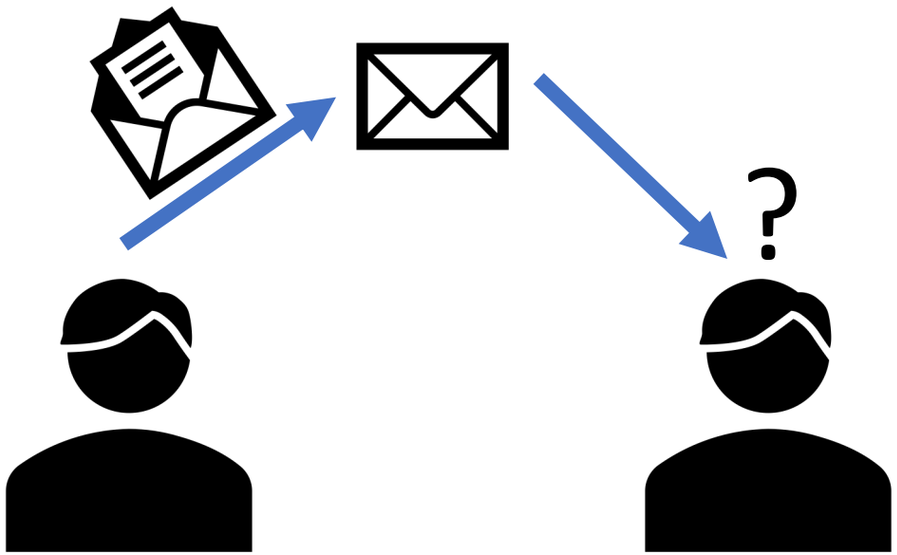
RACE CONDITION: WRITE TO THE SAME ADDRESS



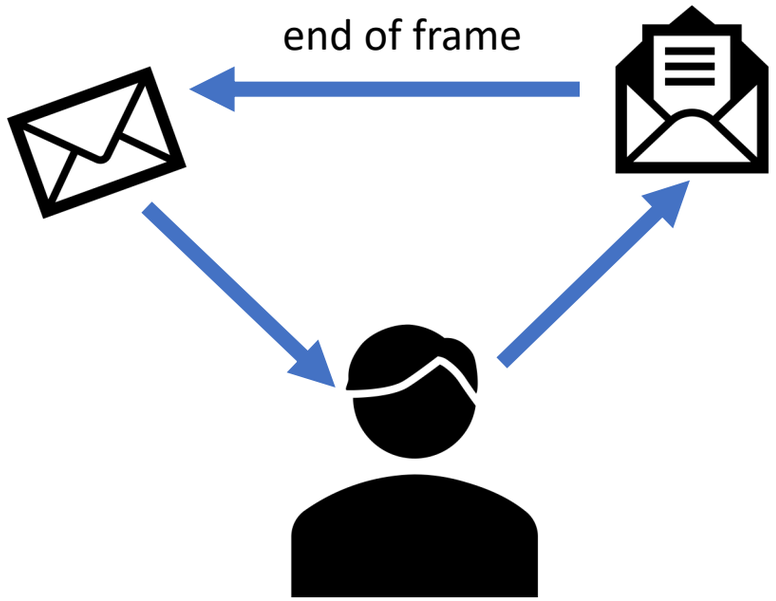
- Find 1-to-1 relationship to exclusive data
- Only one vehicle can enter/exit a connection
- Traffic controllers don’t share connections

RACE CONDITION: READ VS WRITE

sharedArray[entity].Value++



Solution: use a buffer state



TIPS TO AVOID RACE CONDITION

- What works for me:
  - Write pseudo code for parallel algorithm
  - Solve race-condition for all writing instructions
  - Write sequential system based on that pseudo code
  - Iterate until all edge-cases are covered
  - Convert to Job system
  - Iterate until no race-condition remains
- Too many race-condition conflicts:
  - Split the job into multiple jobs
- Floating point is the enemy, use integer
  - Race-condition avoidance rely on strong assumption of exclusivity
  - The assumptions don’t hold when use floating points that accumulate errors over time

DOTS TIPS

- Avoid CommandBuffer:
  - Command Buffer is used for structural changes such as spawn Entity, add/remove Components
  - It’s executed sequentially anyway!
  - To add component, use batch operations from EntityManager, then use jobs to set component values instead
- Use Subscene for faster scene loading!
  - Use GameObjectConversionSystem to supercharge your setup
- Use SharedComponent to group entities together for better cache efficiency
- Use single-field component struct

QUESTIONS & COMMENTS

Source code:

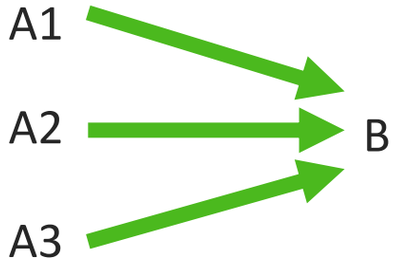
git@gitlab.innogames.de:hainam.pham/ecs_experiments.git

Thank You

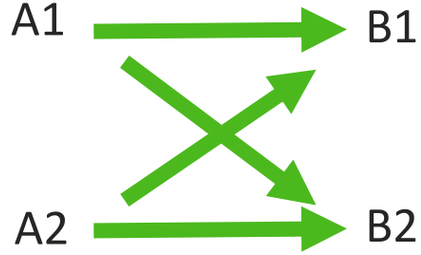
PATH FINDING

PATH FINDING: SEQUENTIAL ALGORITHMS

- A*
- Pros: low memory footprint
- Cons: doesn't scale well



- Flow field
- Pros: scale well
- Cons: not suitable for traffic simulation



- Floyd-Warshall
- Pros: instantly fast
- Cons: huge memory footprint

FLOYD-WARSHALL ALGORITHM

- Network of N nodes
- Complexity for building pathfinding cache: $O(N)^3$
- Cache memory cost: N^2

```
1 let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
2 for each edge (u,v)
3    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
4 for each vertex v
5    dist[v][v] ← 0
6 for k from 1 to |V|
7    for i from 1 to |V|
8       for j from 1 to |V|
9          if dist[i][j] > dist[i][k] + dist[k][j]
10             dist[i][j] ← dist[i][k] + dist[k][j]
11         end if
```

FLOYD-WARSHALL : MEMORY PROBLEM?

- Memory: N^2

  →too expensive for a metropolis!

  →divide the city into smaller sections

  →1000 x 1000: divide to 10 sections of 100x100

  →New memory cost: 10x100x100 / (1000x1000) = 10%!

- Path finding within the same section: classic Floyd-Warshall algorithm
- Path finding between different sections?

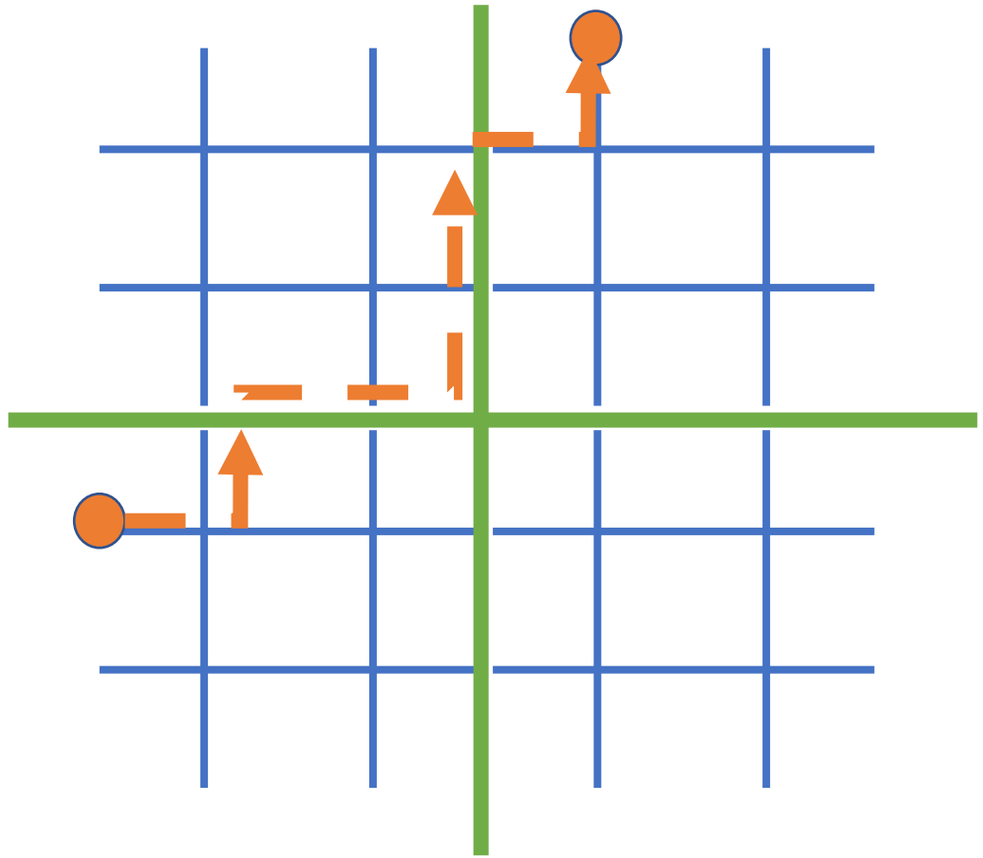
Graph size reduction

- Graph is divided into connected sections
- Arrange sections into hierarchy

Lane exclusion

- Exclude lanes with only one exit from the path finding cache

PATH FINDING: HIERARCHY



Cross-hierarchy path finding

- Demo