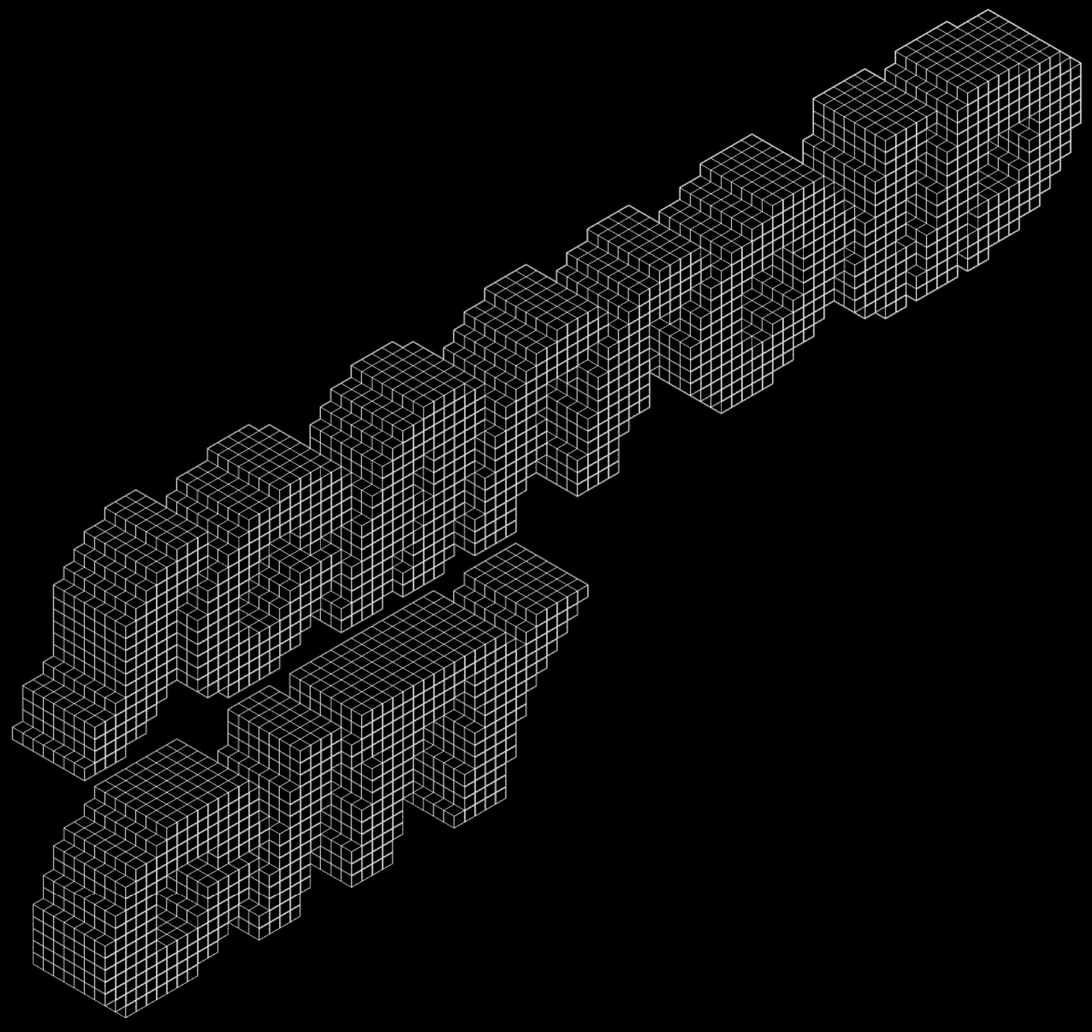


Marco Juliani
AC Studio 2 Term 2
Page Count : 19

Overall Objectives

- Simulate the behavior of a simplified, discrete fleet of elements working togethers toward common goal(s) with little to no supervision, while allowing for user/player override.

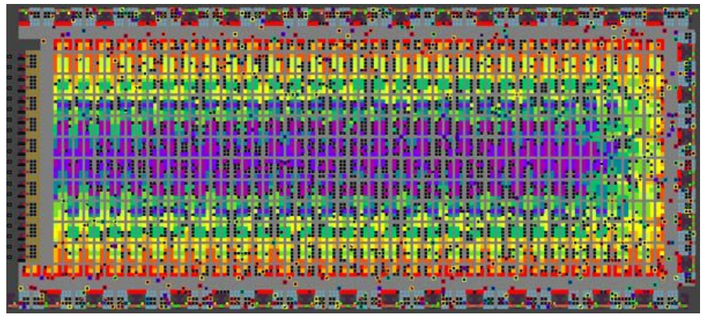
- Create a 'business-like' game where the objective is to take a lean scaffolding up-start from 'loss' to 'break-even' by servicing a number of buildings across a city while forcing the player to think about opportunity cost of every decision (job pursued or overlooked) made.

- Understand bottle-necks of a simplified, isolated, ruled-based dynamic system like this and how those can be addressed through better algorithms.

- How can a hypothetical scaffolding system similar to the one proposed prompt discussion about how current scaffolding systems are used?

Index

Inspiration



A schematic warehouse floor. Shelves with fast-selling items = red. Blue =slow-selling items. Robots rearrange the shelves to keep the fast-selling items at the perimeter, close to packing stations.
Source: https://www.technologyreview.com/s/409020/random-access-warehouses/

'Kilobots' self-organize into shapes by using simple sets of programmed rules.
Source: https://wyss.harvard.edu/a-self-organizing-thousand-robot-swarm/

Picture taken by my at Kensington High Street

KIVA Systems

The distributed fleet paradigm of KIVA to respond in real-time to the logistical needs of market demand is very interesting. It inspired the resource reallocation present in ScaffoldCity every time there is a change in the number of active jobs.

Wyss's Kilobots

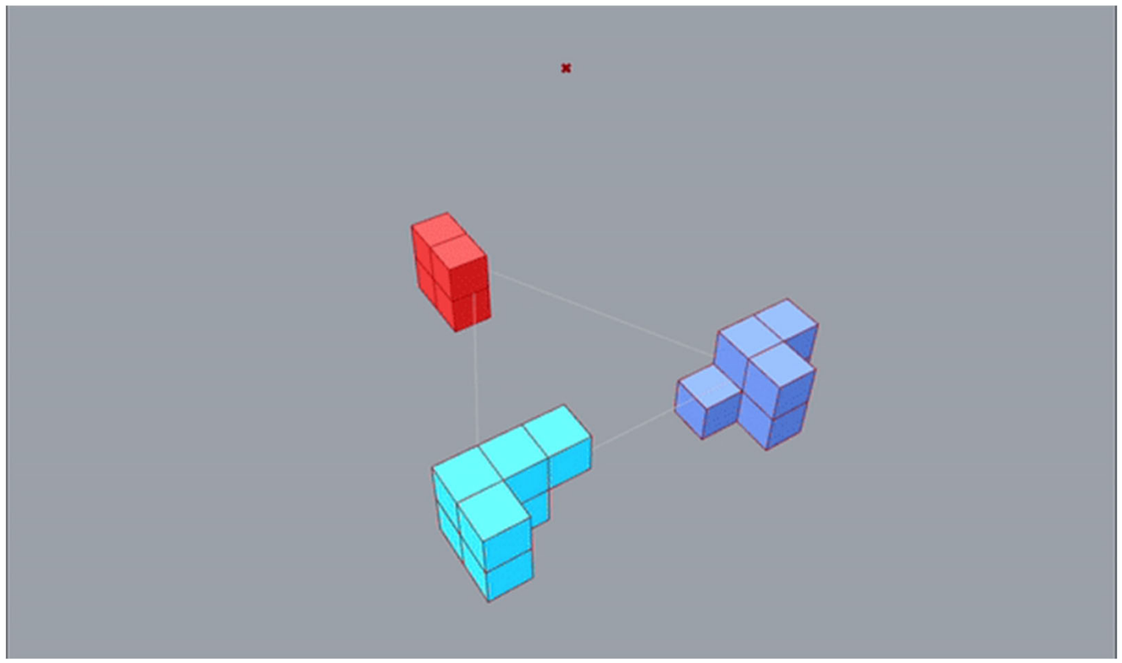
Was interested in creating a much simpler digital analogue, a kind of 'fleet intelligence' whose decision making could be overwritten by inputs.

Ubiquitous scaffolding

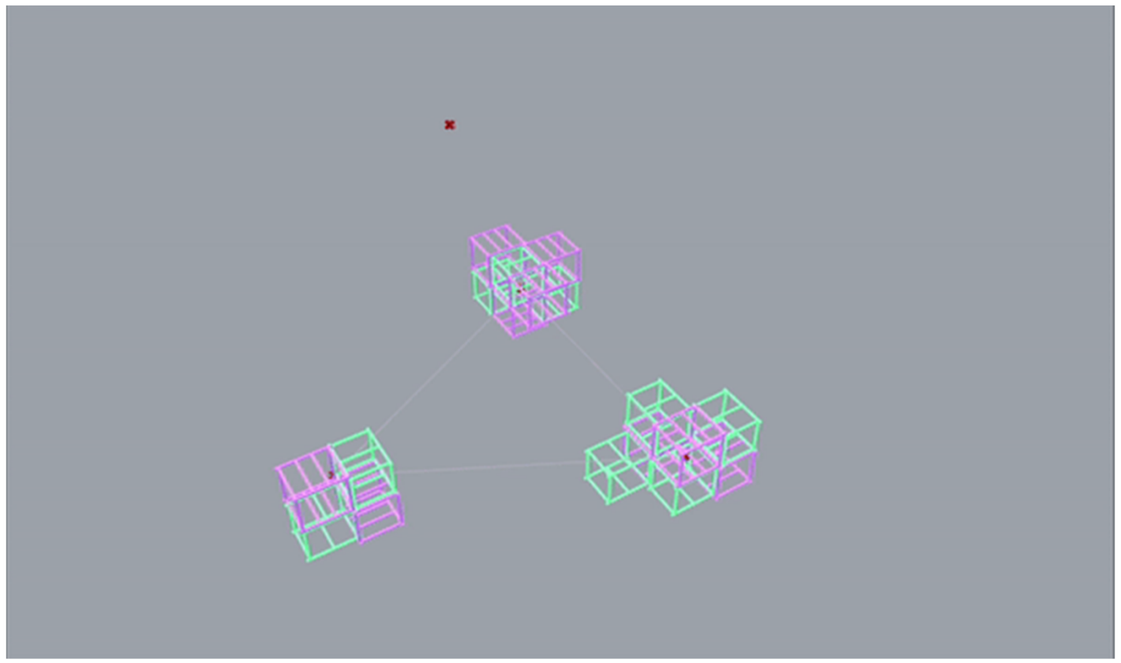
Reconceptualizing what a hypothetical model for modular 'intelligent frames' would look like in contrast to the 'stick system' prevalent today. Upending scaffolding would have significant implications to urban systems/ form given the sheer amount of it used in cities.

Early Fleet Behavior

'Pick-neighbor-closest-to-target-until-target-reached'



Cooperative assembly using three starting points (and corresponding teams) working to reach one target point.



A visualization of the same cooperative assembly in terms of the interlocking of A and B modules. Effectively by checkerboarding a 3d voxel space, you can ensure that B modules will always interlock with A modules. As the structure layers vertically, every 'layer' corresponds to a 90 degree rotation of the A modules (purple) while B modules are free to rotate depending on what corner condition they encounter.

Game Objectives/ Constraints

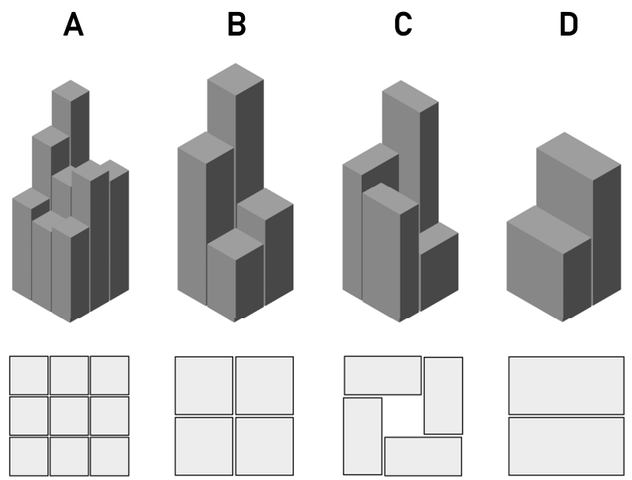
- OBJECTIVE:
  - When the player/fleet has completed enough jobs to collect enough revenue to offset overall costs, the player breaks even and wins the game.

- CONSTRAINTS:
  - Limited fleet size and city arrangement (can 'reset' to start in a new city). City arrangement includes a randomized location for your HQ.
    - Can only take on so many jobs at the same time.
  - Unit cost of 'idle' soldier significantly lower than that of 'active' soldier.
  - 'Deltatime' is a multiplier for these costs, so for every second(24 hours gametime = 1 minute) cost is accrued.
  - Every 'potential job' has its statistics drawn up, leaving the player to decide whether pursuing that job is worth the cost/payout.

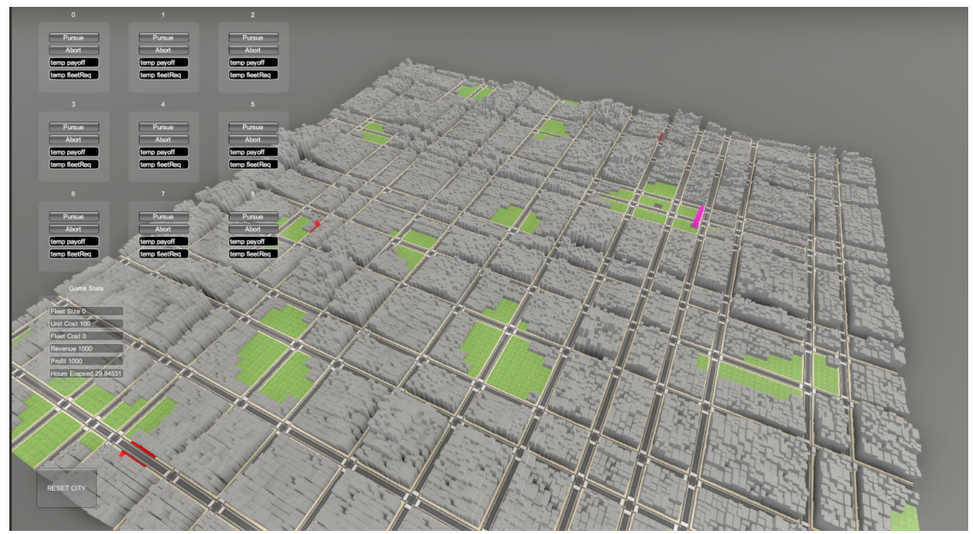
Procedural Buildings /Cities

Pseudocode[1]:

- Pixel[,] = new Pixel[w,h];
- mapPixels[w, h].Type = (int)(Mathf.PerlinNoise(w / 10.0f + seed, h / 10.0f + seed) * 10);
- Assign random x streets, assign random y streets, find their intersections (types -3, -2, -1).



Building Types



Process image

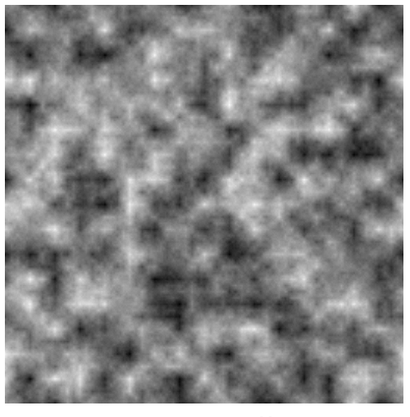
- Intantiate corresponding 'street' gameobjects
- For leftover pixels, ChooseBuilding() with a corresponding height range corresponding to its type(types 0 – 10). This ensured a gradient of heights even though the chosen building types is random between 4 options.

```
public GameObject ChooseBuilding(int min, int max, Vector3 pos, Color color)
{
    var rand = Random.Range(0, 4);
    GameObject output = new GameObject();

    if (rand == 0)
        output = pBuilding.BuildingA(min, max, pos, buildingMat, color);
    else if (rand == 1)
        output = pBuilding.BuildingB(min, max, pos, buildingMat, color);
    else if (rand == 2)
        output = pBuilding.BuildingC(min, max, pos, buildingMat, color);
    else if (rand == 3)
        output = pBuilding.BuildingD(min, max, pos, buildingMat, color);

    return output;
}
```

'Choose Building' function

```
else if (result < 2)
{
    GameObject bldg = ChooseBuilding(30, 45, pos, lerpedColor);
    bldg.tag = "isBuilding";
    bldg.layer = 11;
    mapPixels[w, h].DisplayBuilding = bldg;
    pBuildings.Add(bldg);
    //mapPixels[w, h].IsActive = false;
}
else if (result < 4)
{
    GameObject bldg = ChooseBuilding(20, 30, pos, lerpedColor);
    bldg.tag = "isBuilding";
    bldg.layer = 11;
    mapPixels[w, h].DisplayBuilding = bldg;
    pBuildings.Add(bldg);
}
else if (result < 5)
{
    GameObject bldg = ChooseBuilding(15, 20, pos, lerpedColor);
    bldg.tag = "isBuilding";
    mapPixels[w, h].DisplayBuilding = bldg;
    pBuildings.Add(bldg);
}
```

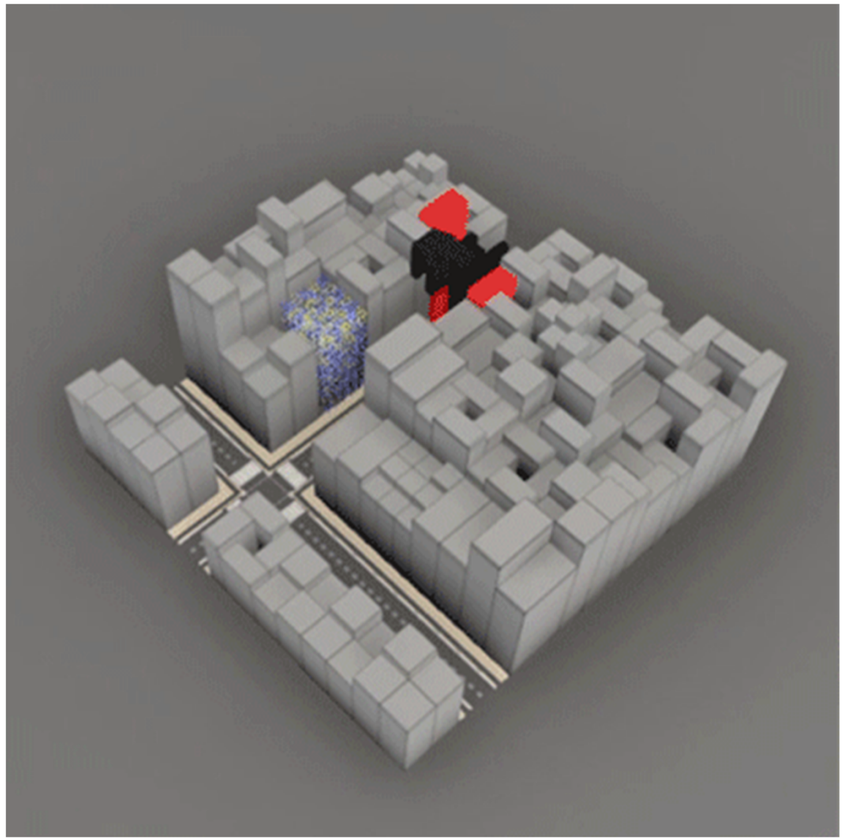
Height noise 'gradient'



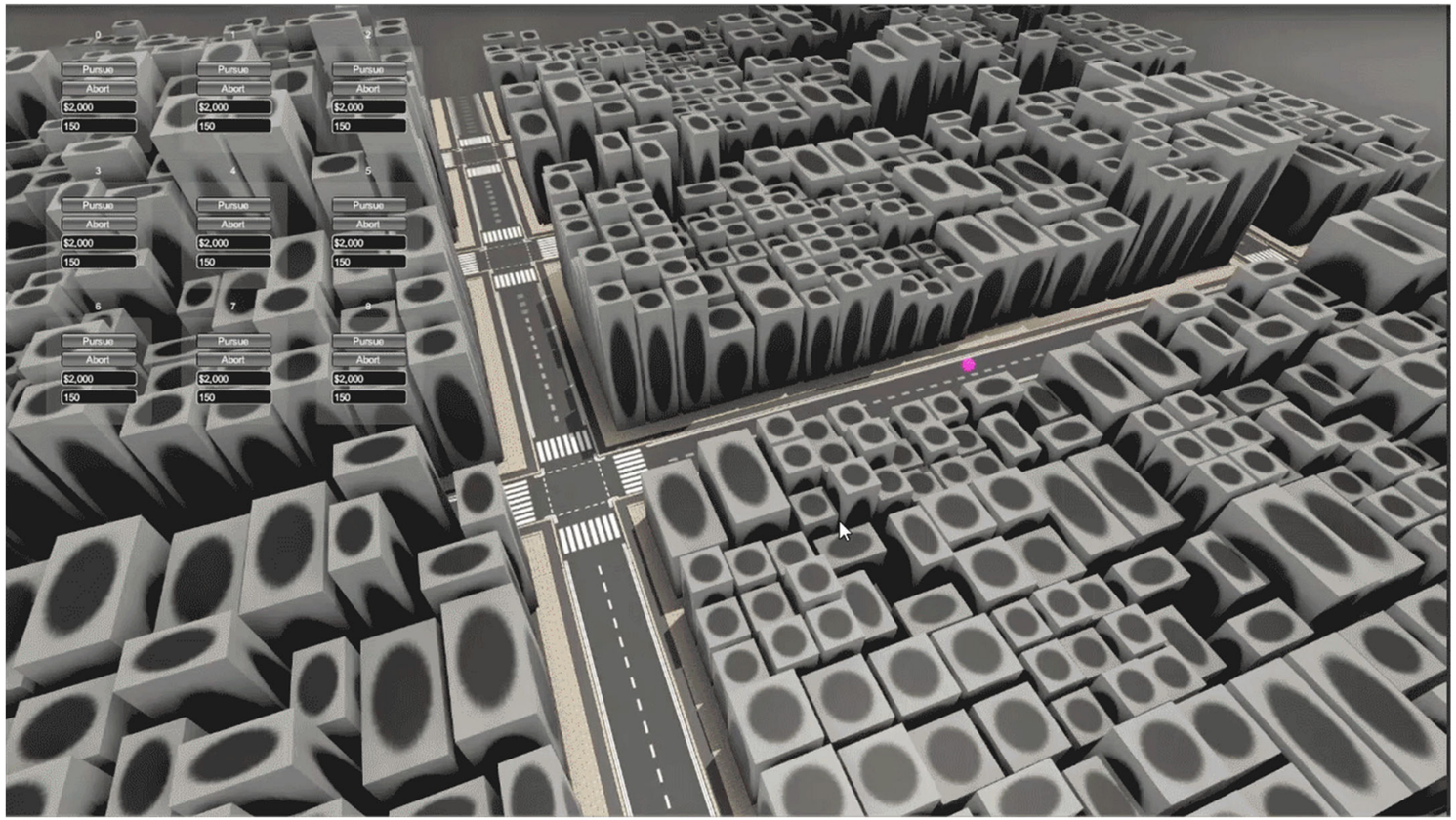
http://www.java-gaming.org/topics/generating-2d-perlin-noise/31637/view.html

[1] This framework of instantiating items in a 2d array to generate a city using Perlin noise was taken from a 'Holistic3d' tutorial https://www.youtube.com/watch?v=xkuniXI3SEE.

Procedural Buildings/Cities



Level ‘reset’ and randomized HQ assignment



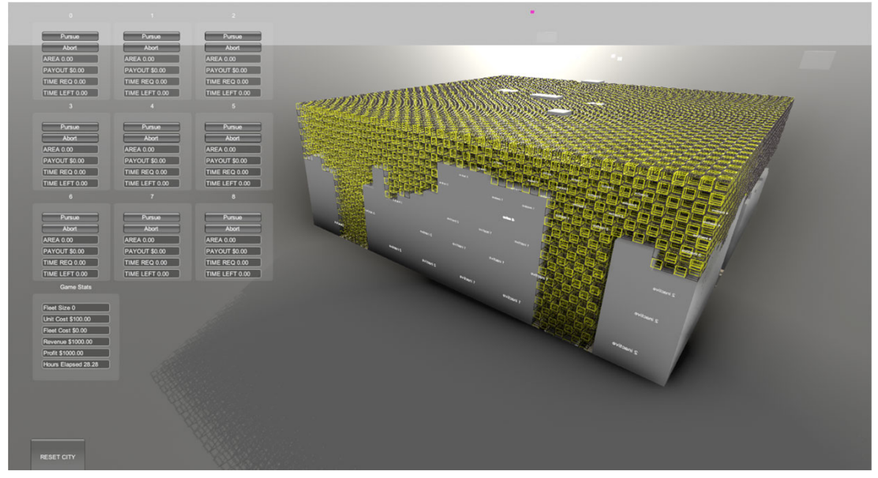
Identifying ‘pixels’ and the corresponding GameObject children that have street frontage and could potentially need servicing ie. ‘Jobs’

Dense Grid *(initially)*

- Array implementation:
  - Voxel[,,] = new Voxel[x,y,z];

- Storing array of (x*y*z) size in memory, even though most of it is 'inactive' never to be used (in this case the space occupied by the buildings).

- This led to really poor overall game performance – constraining the size of the city to be really small. Especially once I tried implementing pathfinding capabilities using Dijkstra's, given that Dijkstra's searches exhaustively rather than heuristically.



Early dense grid implementation. The model was very slow because the hidden voxels are still being stored in memory.

```
//selecting pixels that have street frontage
//pixel.active means you are a street, not a building
public List<Pixel> getOrthoNeighbors(Pixel[,] mapGrid, int mapWidth, int mapHeight, int meX, int meZ)
{
    List<Pixel> neighbors = new List<Pixel>();

    var lx = meX == 0 ? 0 : -1;
    var ux = meX == mapWidth - 1 ? 0 : 1;
    var lz = meZ == 0 ? 0 : -1;
    var uz = meZ == mapHeight - 1 ? 0 : 1;

    for (int i = meX + lx; i <= meX + ux; i++)
    {
        for (int j = meZ + lz; j <= meZ + uz; j++)
        {
            //ensures only ortho neighbors
            if ((i == meX || j == meZ))
                if (mapGrid[i, j].IsActive)
                    neighbors.Add(mapGrid[i, j]);
        }
    }
    return neighbors;
}
```

Sample code used to retrieve 'neighbors', in this case Pixel(building lot) neighbors. While in this case you are asking the function to look at a specific neighborhood of indices in the array, all the others are still stored in memory.

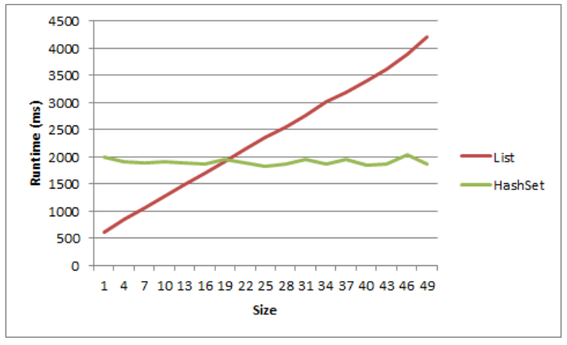
Sparse Grid

- **Dictionary Implementation:**
  - Dictionary <Vector3Int, Voxel>
- **Dictionaries** (hashsets) have O(1) + overhead access speeds *(virtually the same access speed regardless of the number of items you are working with)*.
- **Arrays** have O(n) access speeds *(speed increases in proportion to number of items),*

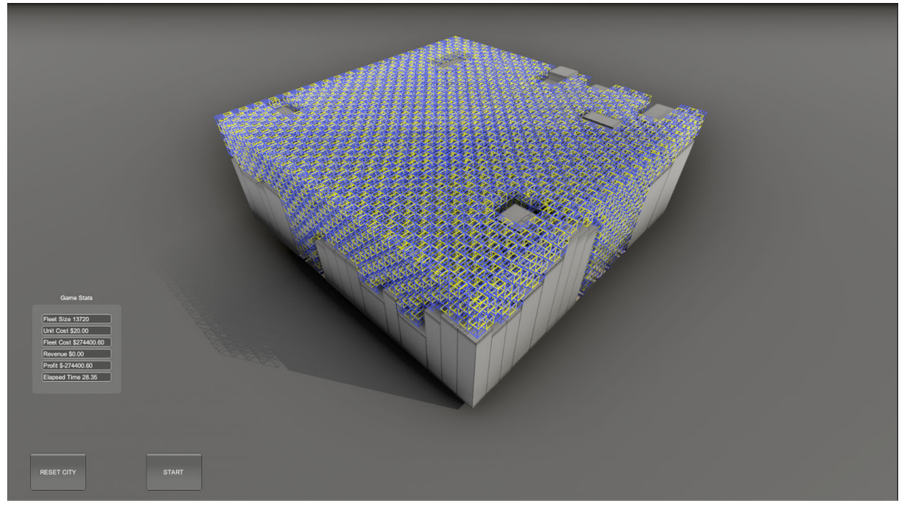
* A dictionary was the data structure of choice due to the large number of items I am working with.

```
//excludes bottom
public IEnumerable<SparseGrid.Voxel> GetFaceNeighbours()
{
    int x = Index.x;
    int y = Index.y;
    int z = Index.z;

    var indices = new[]
    {
        new Vector3Int(x - 1, y, z),
        new Vector3Int(x + 1, y, z),
        new Vector3Int(x, y + 1, z),
        new Vector3Int(x, y, z - 1),
        new Vector3Int(x, y, z + 1),
    };

    foreach (var i in indices)
        if (_grid.Voxels.TryGetValue(i, out var voxel))
            yield return voxel;
}
```

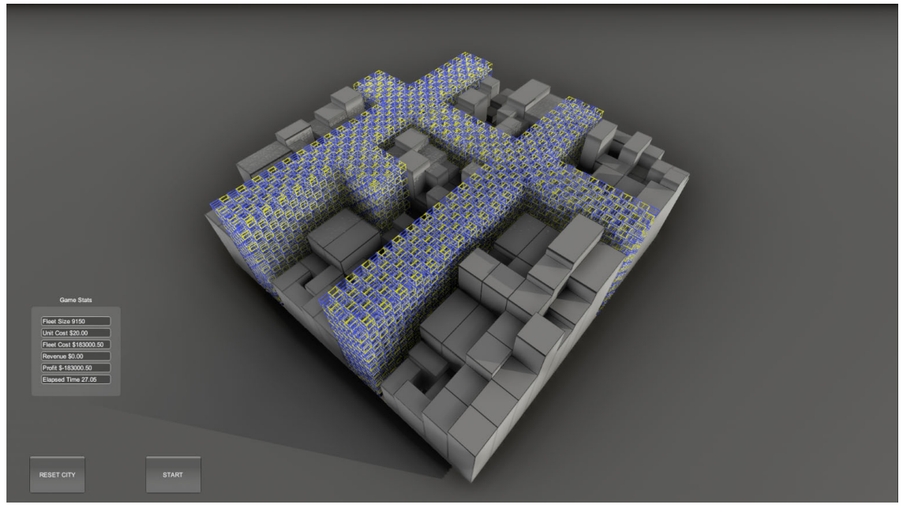
Sample code to retrieve neighbors with Dictionary implementation. Similar to array implementation but much faster query speed as the size of the set grows.



https://stackoverflow.com/questions/150750/hashset-vs-list-performance



Sparse grid visualization



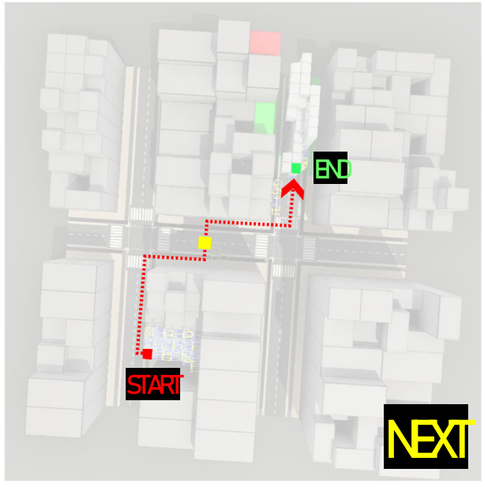
'Optimized' sparse grid by removing much of the void space above buildings that would not be used for pathfinding due to a notion of gravity/ privacy constraints.

Pathfinding - Dijkstra's Shortest Path[2]

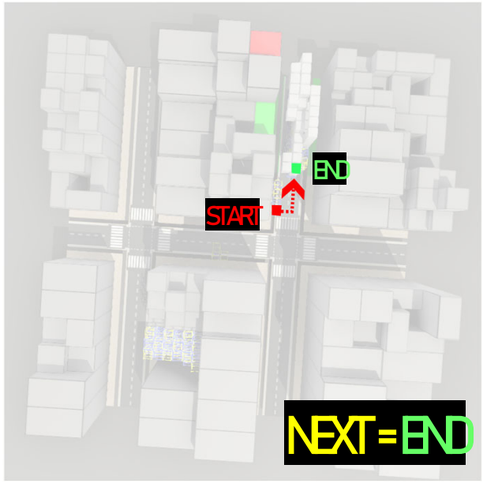
Used this implementation to generate the *'job structures'* as well as for pathfinding from different points in the city to different targets/back to HQ.

```
public void InitGraph()
{
    var faces= GetFaces();
    graphEdges = faces.Select(f => new TaggedEdge<SparseGrid.Voxel, SparseGrid.Face>(f.Voxels[0], f.Voxels[1], f));
    graph = graphEdges.ToUndirectedGraph<SparseGrid.Voxel, TaggedEdge<SparseGrid.Voxel, SparseGrid.Face>>();
}
```

Voxel faces = graph edges
Voxels = graph nodes



In this case, path length of the shortest path between START and END > 10, thus the path length of the shortest path between the two gives us our NEXT, given that it is a vacant voxel.



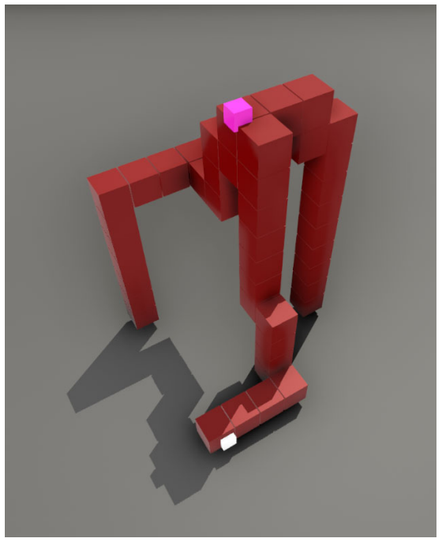
In this case, path length of the shortest path between START and END < 10, thus given the rule established in the code to the right, NEXT = END.

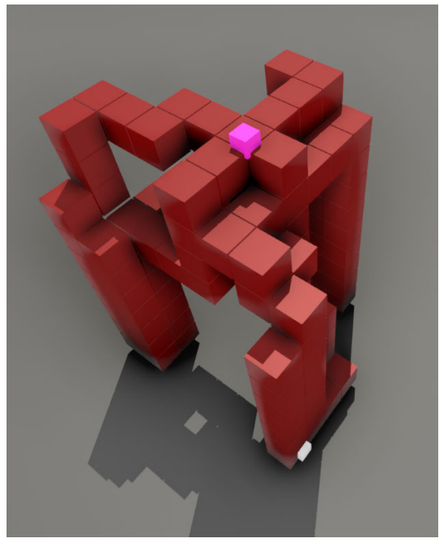
```
if (grid.Voxels.TryGetValue(endIndex, out var end))
{

    var shortest = graph.ShortestPathsDijkstra(_ => 1, start);
    SparseGrid.Voxel next = null;
    List<SparseGrid.Voxel> pathList = new List<SparseGrid.Voxel>();

    if (shortest(end, out var path))
    {
        var current = start;
        foreach (var edge in path)
        {
            current = edge.GetOtherVertex(current);
            pathList.Add(current);
        }

        var indNex = pathList.Count() <= 10 ? pathList.Count() - 1 : (int)(pathList.Count() * 0.5f);

        if (indNex <= 10) next = end;
        else next = pathList.ElementAt(indNex);
    }
```

In this case I am finding a shortest path between a 'start' voxel and a target voxel of index 'endIndex'. If the path length is less than 10, I am asking the 'start' voxel to move to the end index, if larger than 10, I am asking the voxel to move to half the distance to the end voxel.

[2] Used the 'QuickGraph' library implementation of Dijkstra's. Library reference: https://github.com/YaccConstructor/QuickGraph.

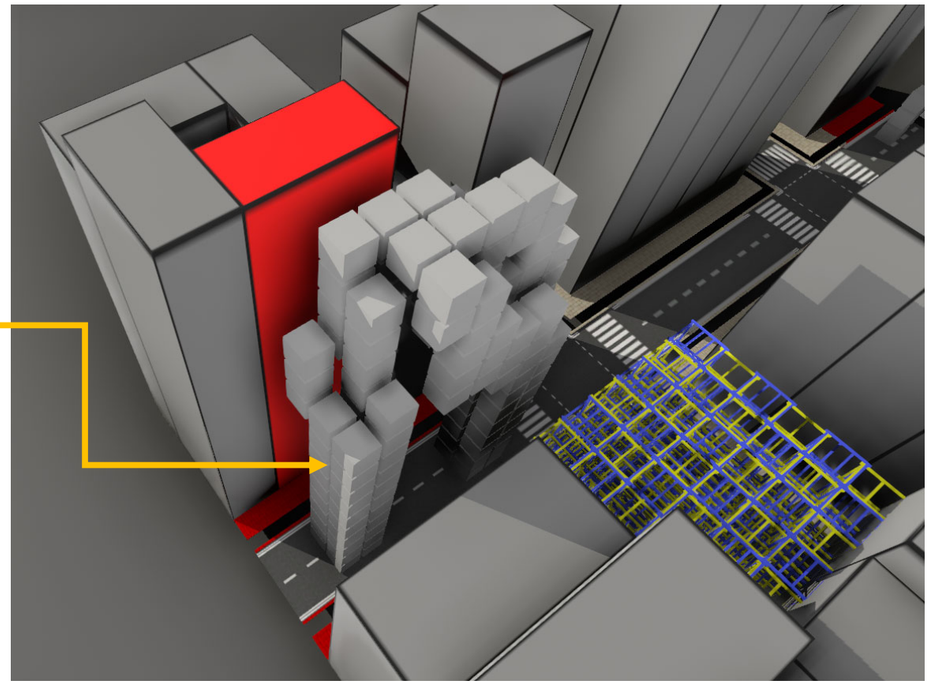
Structure Generation – Dijkstra's Shortest Path[2]





Initial shortest path studies between 3 starting points and a target point. On the left you can see a starting population of 1 per point, on the right the starting population is 3 per point.

Gray cubes are a 'ghost structure' intended to provide the player with a visual cue of what that job will look like when built/ complete. As the scaffolds fill in 'ghost voxels', they disappear.



'Job' class pseudocode:

Job job = new Job();

1) Pick Building()
2) Find two closest street pixels
3) Displace building centroid by transform.position.y/2 to arrive at top of building
4) Displace that point by the distance required to arrive at the face of the building.
5) Draw a shortest path between selected street centroids and displaced point using a starting population that is proportional to the relative size of the building.
6) Record the indices of voxel locations comprising shortest path and sort them by their 'y' axis so that the fleet knows to pursue them in order.

[2] Used the 'QuickGraph' library implementation of Dijkstra's. Library reference: https://github.com/YaccConstructor/QuickGraph.

Fleet Resource Allocation – *Pseudocode*

'JobHandler()' checks to see if a job has timed out, has been pursued , completed. or aborted. It keeps track of potential and active jobs.

```
public void Update()
{

    JobHandler();//if job times out, vox
    JobCount();
    if (CheckCurrentJobs() == true)
    {

        RedistributeJobs();//every time
    }
    UpdateJobTarget();
}
```

This if-statement checks to see if new the number of 'current jobs' has changed (if the user has clicked 'pursue' or 'abort). If so, voxels are redistributed amongst whatever current jobs there are. Scaffolds not assigned to jobs are assigned to return to HQ.

This function takes care of checking to see which voxels have arrived at the job's target voxels. A bucket of voxels assigned to a job will pursue targets[]. As targets[] fill up the remaining fleet members pursuing that job need to be updated on where to go. The voxel that arrived, also has to be told to stay put until the structure is complete.

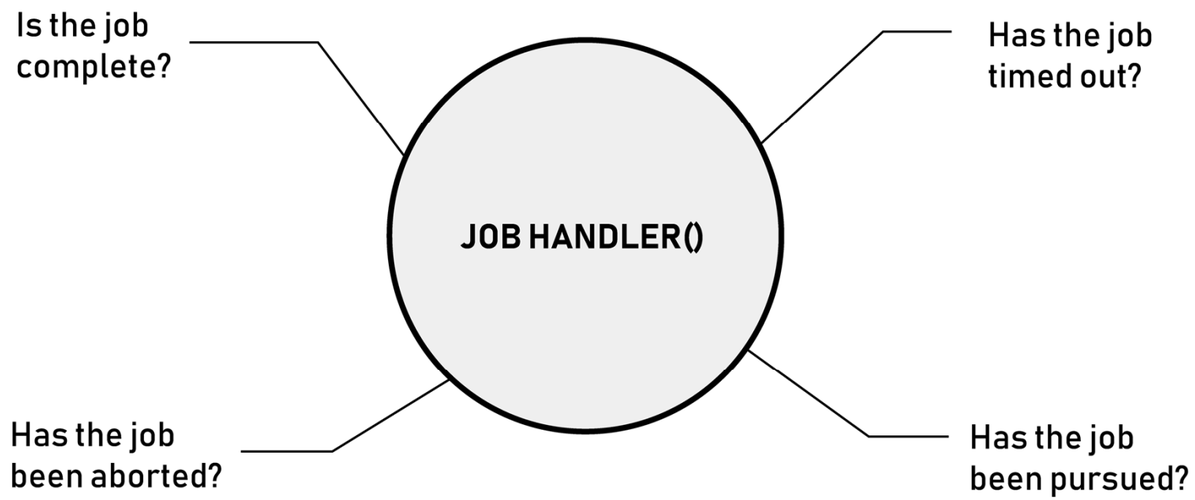
If there are no current jobs, send active fleet back to HQ.

If there is a change in the number of current jobs, re-assign buckets of voxels as to each job as required(job.indices.Count) by the job.

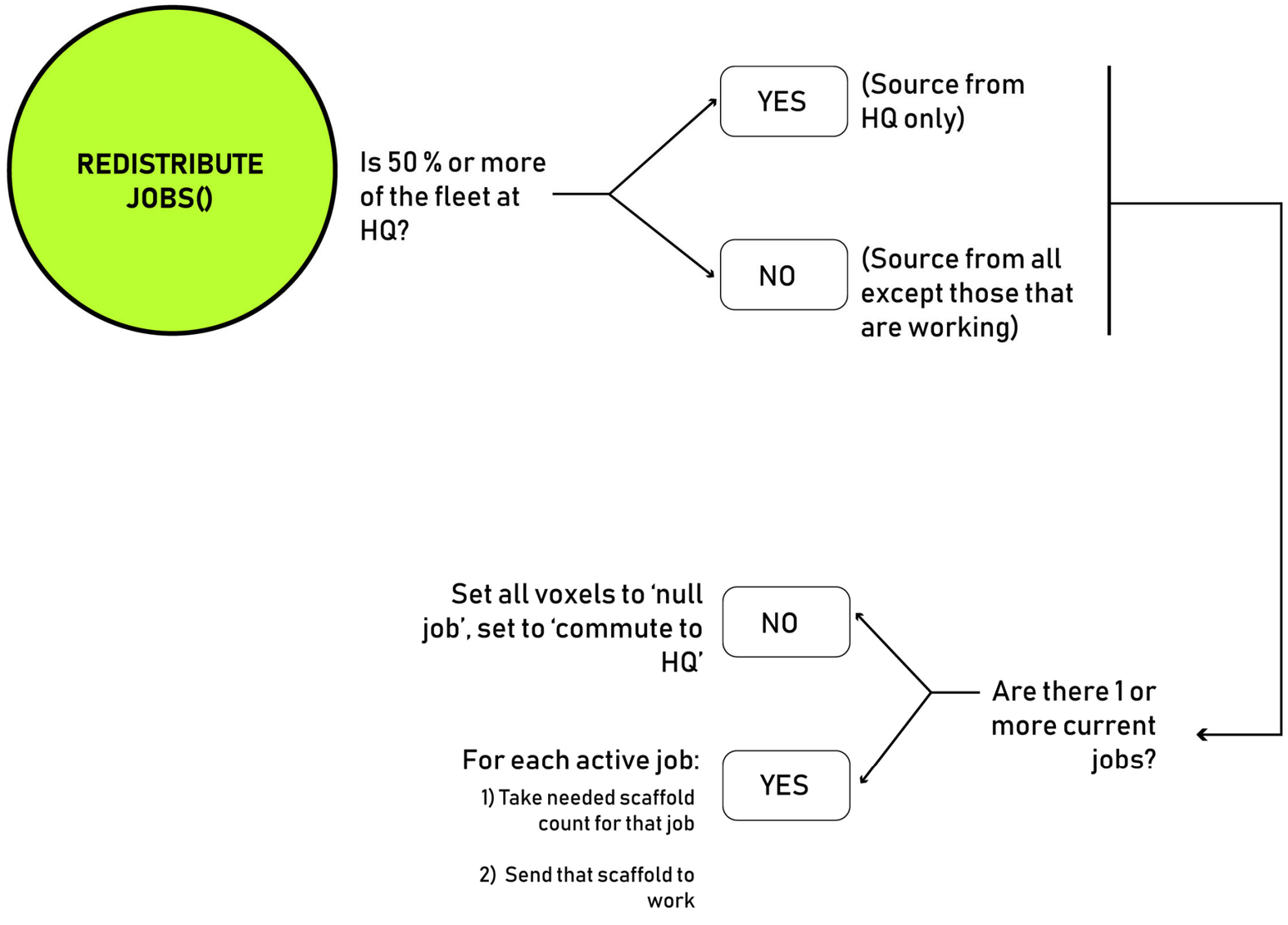
If more than 50 % of the fleet is at HQ, redistribute from only 'idle' voxels, Otherwise, redistribute from the entire fleet.

```
public void RedistributeJobs()
{
    IEnumerable <SparseGrid.Voxel> vox;

    if (army.armyVacancyRate < 0.5f)
        vox = army.grid.GetVoxels().Where(v => v.On).Where(v => v.Idle == true);
    else
        vox = army.grid.GetVoxels().Where(v => v.On);

    if (currentJobs.Count < 1)
    {
        foreach (var v in vox)
        {
            v.Job = null;
            v.MakeCommute();
        }
    }
    else
    {
        foreach (var job in currentJobs)
        {
            var jobSpecVox = vox.Take(job.indices.Count);
            foreach (var j in jobSpecVox)
            {
                j.Job = job;
                j.Job.indices = job.indices;
                j.MakeCommute();

            }
        }
    }
}
```

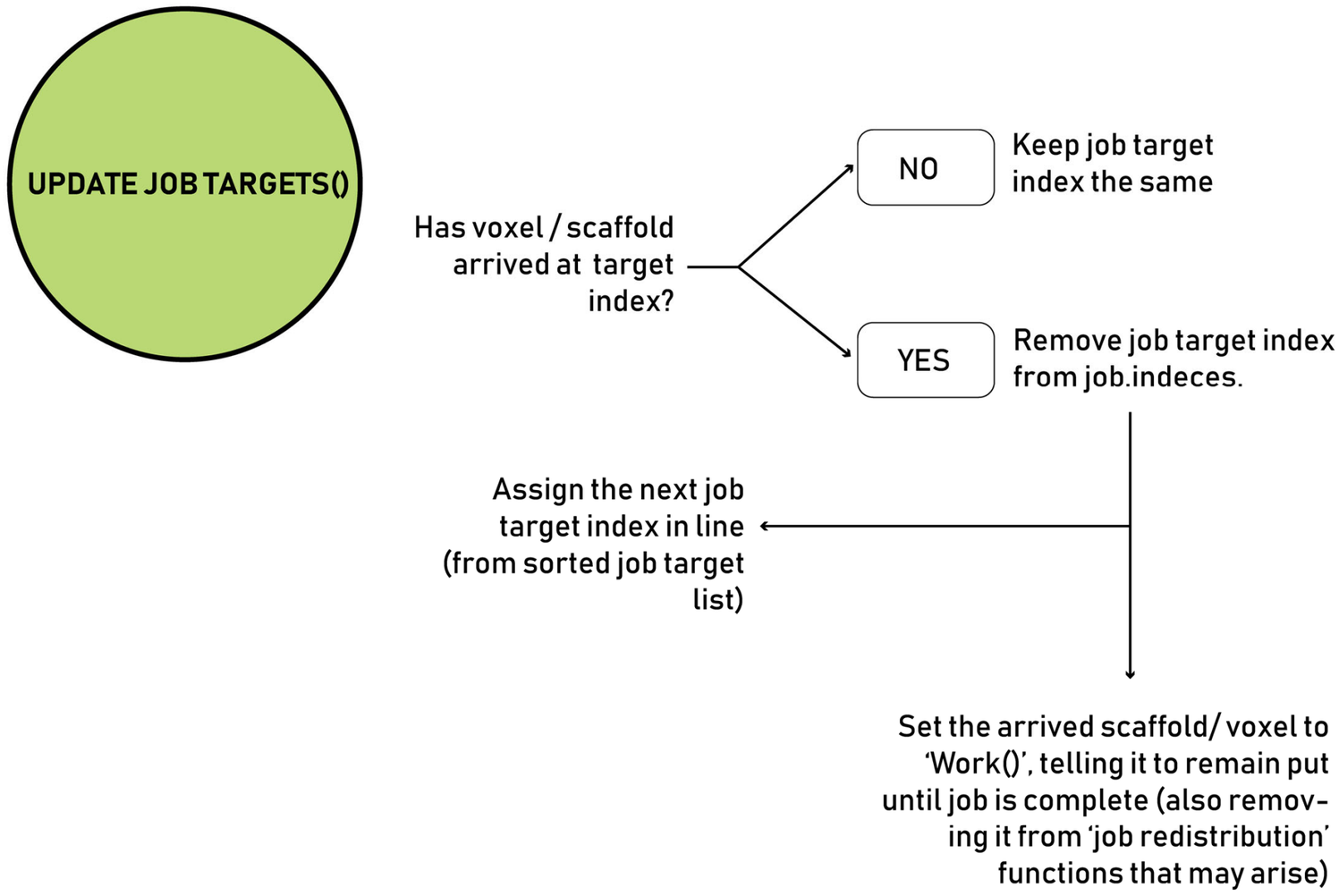
Fleet Resource Allocation - *Pseudocode*

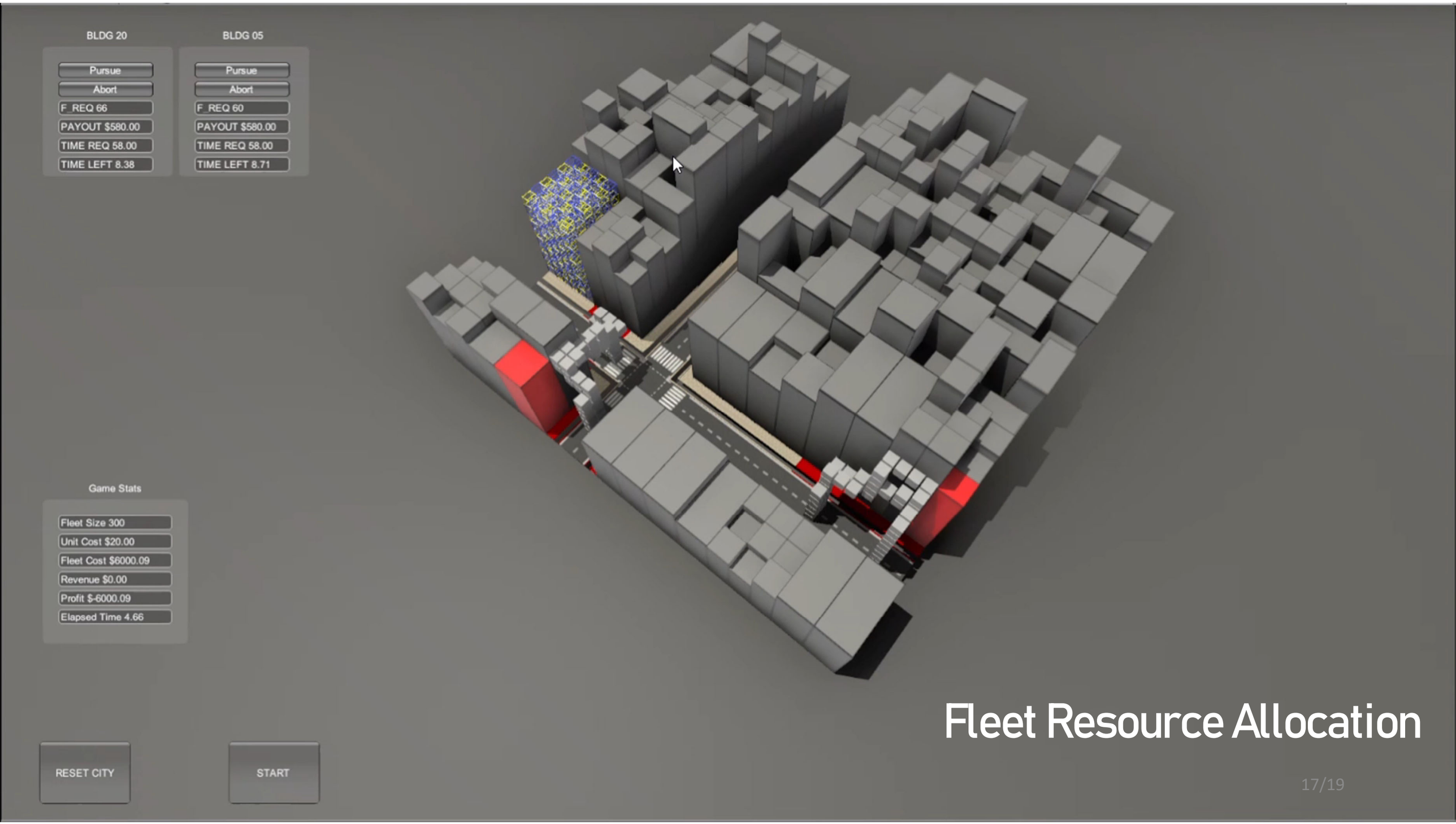


Fleet Resource Allocation – *Pseudocode*



Fleet Resource Allocation - *Pseudocode*



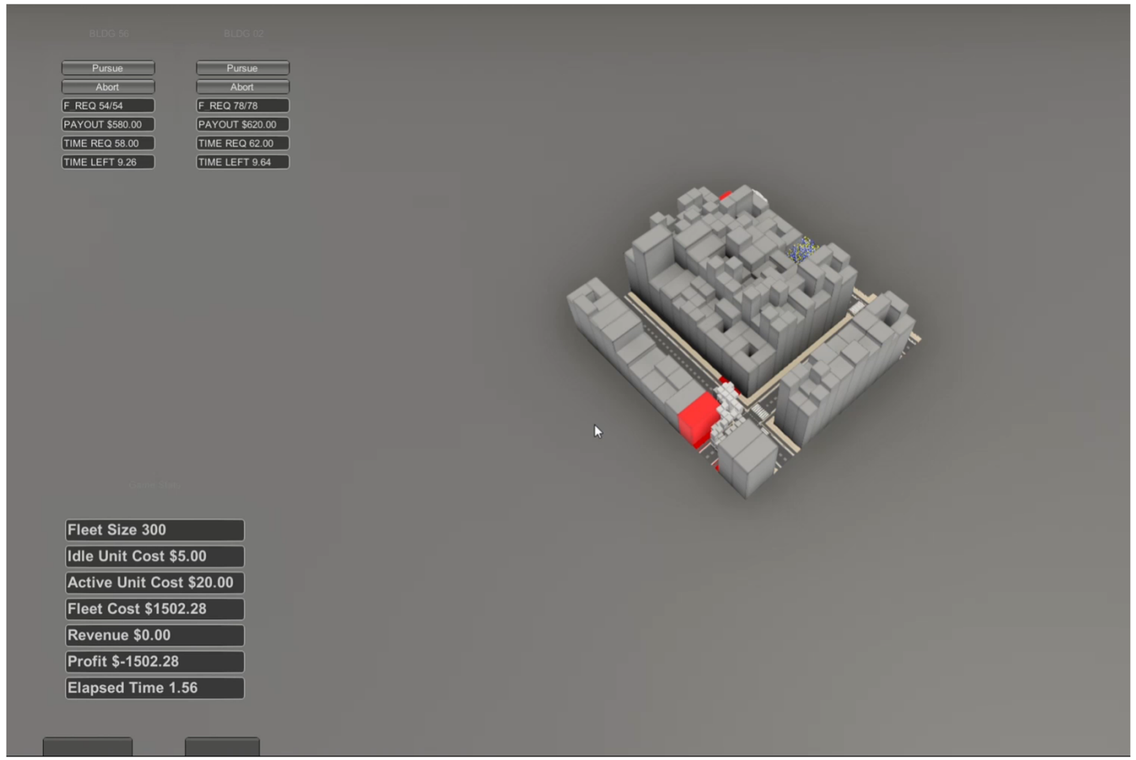


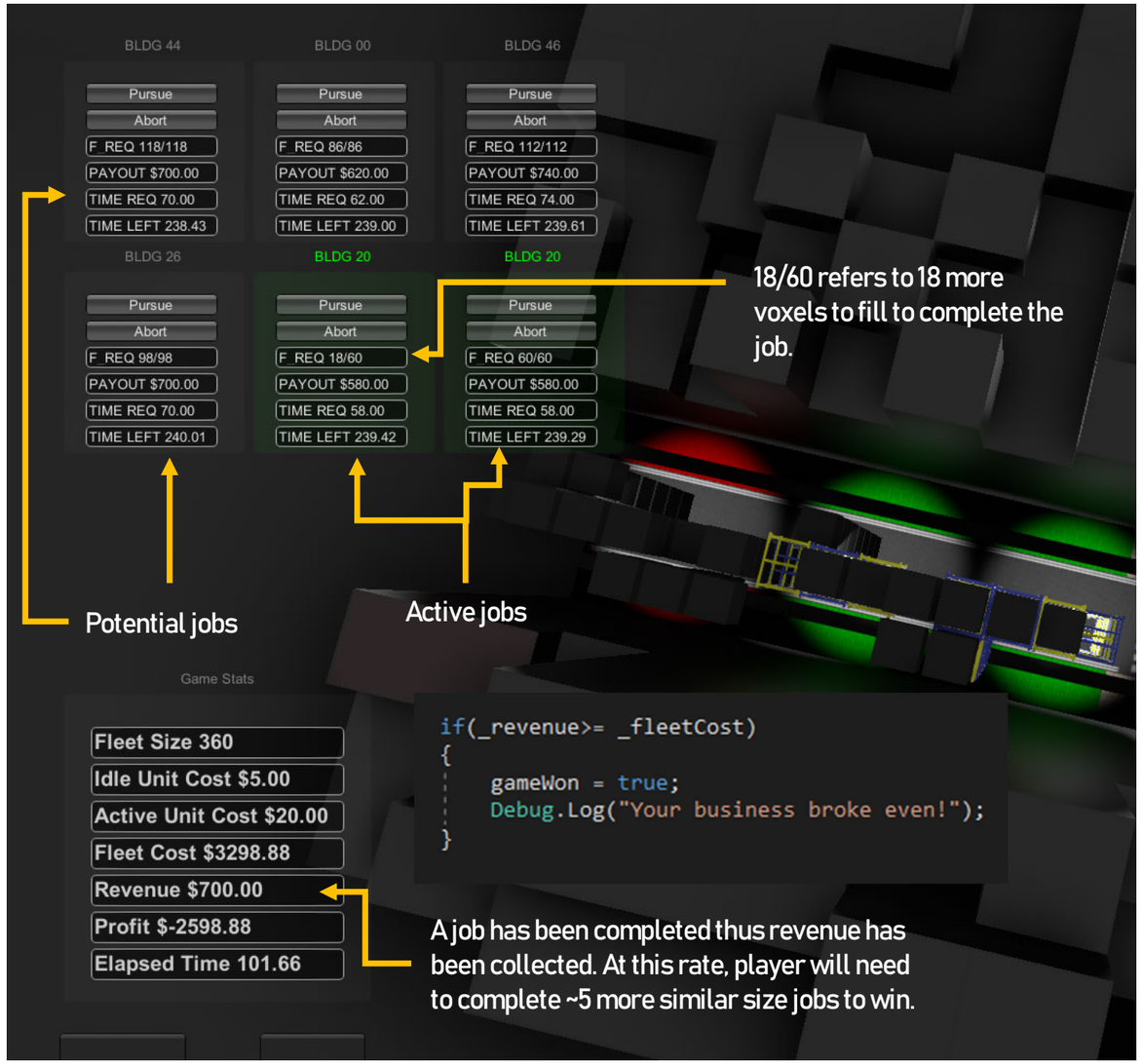
Fleet Resource Allocation

‘Business Logic’

```
public void Update()
{
    city.UpdateFleetStatus();
    fleetSize = city.getFleetSize();
    initFleetCost = fleetSize * unitCostIdle;
    fleetCost = initFleetCost + (((unitCostIdle * city.fleetCountIdle) + (unitCostActive * city.fleetCountActive)) * Time.fixedTime*0.001f);
    revenue = city._revenue;
    profit = revenue - fleetCost;

}
```

The method above keeps a live count of IDLE and ACTIVE voxels and multiplies the corresponding amounts by the corresponding cost of each state.





BLDG 24

Pursue

Abort

F_REQ 58/58

PAYOUT $780.00

TIME REQ 78.00

TIME LEFT 90.48

BLDG 45

Pursue

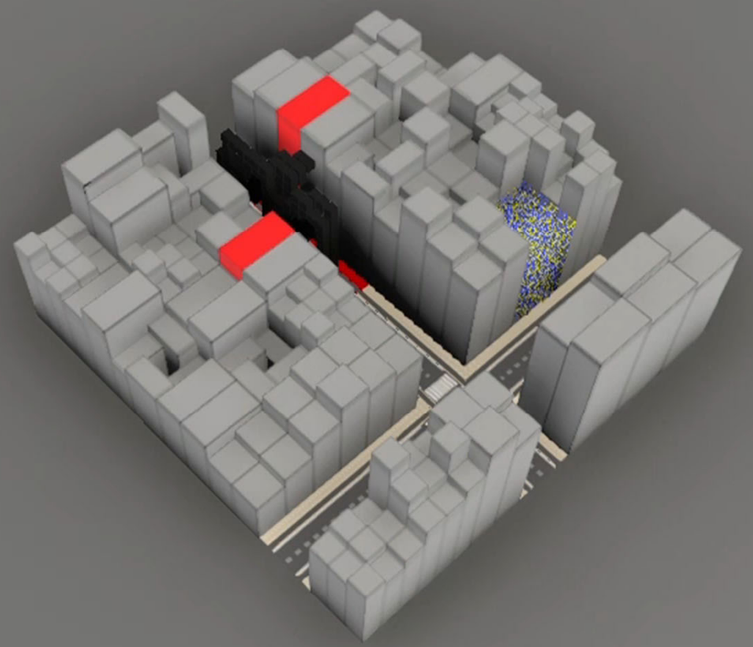
Abort

F_REQ 108/108

PAYOUT $900.00

TIME REQ 90.00

TIME LEFT 90.69



Game Stats

Fleet Size 300

Idle Unit Cost $5.00

Active Unit Cost $20.00

Fleet Cost $1620.79

Revenue $0.00

Profit $-1620.79

Elapsed Time 0.19

RESET CITY

START

Level of Detail (LOD)