WebGL 2D Crowd Simulation Renderer

Course Name: CSE 606, Computer Graphics
Student Name: Gathik Jindal
Student ID: IMT2023089

21 September 2025

Abstract

This project simulates a 2D crowd simulation. A square object in the center, multiple vertices around the canvas and people scattered around. Triangles are formed with the the canvas borders, vertices and obstacle's corners, each triangle is given a color based on density. User can add and delete triangles and also move people around. Triangulation is recomputed every time the obstacle is moved.

1 Features

This project implements a variety of features required for a 2D crowd simulation, focusing on real-time interactivity and WebGL rendering.

- Dynamic Scene Generation: The application initializes with a random distribution of "dots" (for triangulation) and "people" within a defined space.
- Constrained Delaunay Triangulation: The 2D space is partitioned into triangles using the cdt2d library. A central obstacle is integrated as a constraint, ensuring its edges are part of the triangulation mesh.
- Interactive Obstacle: Users can manipulate a central obstacle in real-time:
  - Translation: Move the obstacle up, down, left, or right.
  - Rotation: Rotate the obstacle around its center.
  - Scaling: Scale the obstacle uniformly up or down.
- Real-time Triangulation Updates: Any transformation applied to the obstacle automatically triggers a recalculation of the triangulation mesh to adapt to the new layout.
- Population Density Visualization: Triangles are color-coded based on the number of "people" they contain, providing an immediate visual representation of crowd density.
  - Blue: Underpopulated (below the density threshold).
  - Green: Correctly populated (at the density threshold).
  - Red: Overpopulated (above the density threshold).
- Collision Handling: People and dots that collide with the obstacle are removed and regenerated in valid, non-colliding locations.
- Interactive Mesh Editing: The triangulation mesh can be manually edited:
  - Add Triangle: Users can select three vertices (dots or corners) to manually create a new triangle.
  - Delete Edge: Users can click on an existing edge to remove it from the triangulation.
  - Interactive Crowd Movement: Individual "people" can be clicked and dragged to new locations, which updates the density coloring of the affected triangles.
- Dynamic Controls: Sliders allow for real-time adjustment of the number of dots, the number of people, and the population density threshold.

2 Controls

Table 1: Obstacle Manipulation

Key	Action
‘W‘	Move Obstacle Up
‘S‘	Move Obstacle Down
‘A‘	Move Obstacle Left
‘D‘	Move Obstacle Right
‘Q‘	Rotate Obstacle Counter-Clockwise
‘E‘	Rotate Obstacle Clockwise
‘O‘	Scale Obstacle Up
‘P‘	Scale Obstacle Down

3 Methodology

The 2D crowd simulation renderer was developed using modern JavaScript (ES6 Modules) and the native WebGL 1.0 API, structured around a central animation loop managed by ‘requestAnimationFrame‘. The application’s architecture is event-driven; user inputs or changes in the simulation state trigger a central ‘update()‘ function. This function is responsible for recalculating the simulation’s geometry and density, and then updating the necessary GPU buffers. This approach cleanly separates state management from the continuous rendering process.

3.1 Rendering Pipeline

A single, generic GLSL shader program is used to render all geometric objects in the scene. The rendering pipeline employs a 2D *orthographic projection* to correctly handle the canvas’s aspect ratio and to map a custom world coordinate system to WebGL’s native clip space.

All object transformations (*translation, rotation, and scale*) are calculated on the CPU using the ‘gl-matrix‘ library and are applied on the GPU by passing a final ‘mat4‘ transformation matrix as a uniform to the vertex shader. A generic ‘drawObject‘ function handles the rendering of all entities by binding the appropriate attribute buffers (position, color) and intelligently choosing the correct draw call—‘gl.drawElements‘ for indexed geometry like triangles and ‘gl.drawArrays‘ for non-indexed points. To support the visualization of semi-transparent density zones, *alpha blending* is enabled for the WebGL context.

3.2 Spatial Partitioning via Triangulation

The core of the spatial partitioning is a *Constrained Delaunay Triangulation (CDT)*, which is generated using the external ‘cdt2d‘ library. The vertices for this triangulation are sourced from a combination of user-configurable random ”dots” and the four corners of the canvas, ensuring the entire visible area is partitioned.

The edges of the central obstacle are enforced as *constraints* within the triangulation. This guarantees that no triangle edges intersect the obstacle, effectively treating it as a solid object within the partitioned space. The entire triangulation is dynamically recalculated in real-time whenever the obstacle is transformed, adapting the mesh to the new geometric constraints.

3.3 Crowd Simulation and Density Visualization

The ”crowd” is represented by a collection of ”people” points, the number of which is controlled by a UI slider. A collision detection system, implemented in ‘math.js‘, prevents these points from spawning or existing inside the obstacle. This check is performed in the obstacle’s local coordinate space to accurately account for its rotation and scale.

The primary simulation feature is the visualization of population density. The ‘getTriangleDensity‘ function iterates through every triangle in the mesh and, for each one, counts the number of ”people” points contained within its boundaries. This point-in-polygon test is implemented using a robust *Barycentric coordinate* method.

Based on the count, the triangle's geometry is sorted into one of three categories (overpopulated, underpopulated, or correctly populated) and rendered with a corresponding color, providing an immediate visual summary of the crowd's distribution.

3.4 User Interaction and Scene Editing

The application supports a rich set of user interactions for manipulating the simulation.

- **Mouse Picking:** All direct interactions, such as dragging points or editing the mesh, rely on a mouse-picking system. User clicks in screen-space pixel coordinates are converted to world-space coordinates by multiplying them with the inverse of the projection matrix.
- **Agent Manipulation:** In the default mode, users can click and drag any "person" point. The point's position is updated in real-time in its GPU buffer for smooth visual feedback. Upon releasing the mouse, the main 'update()' function is called to recalculate the density colors for the affected triangles.
- **Interactive Triangulation Editing:** An "Edit Mode" system allows for manual refinement of the triangulation mesh, as suggested by the assignment.
- **Adding Triangles:** In "Add Triangle" mode, the user can select any three triangulation vertices. Upon the third selection, a new triangle is formed by adding its indices to the main index buffer, and the scene is updated.
- **Deleting Triangles:** In "Delete Edge" mode, the 'findClosestEdge' helper function in 'math.js' is used to determine which edge is nearest to the user's click. All triangles that share this edge are then removed from the index buffer, and the scene is redrawn.

4 File Structure

The project is organized into several JavaScript modules to separate concerns.

- *index.html*: The main HTML file that sets up the canvas and UI controls. index.js: The core application logic. It initializes WebGL, manages the render loop, handles user input, and coordinates all other modules.
- *math.js*: Contains the key computational logic, including collision detection, triangulation, density calculation, and mesh manipulation functions.
- *utility.js*: Provides utility functions for handling keyboard-driven obstacle transformations (movement, rotation, scaling) and clamping it within bounds.
- *draw-scene.js*: A generic module for drawing objects in WebGL. It sets up shader attributes and executes the appropriate draw calls.
- *init-buffers.js*: Helper functions for creating and initializing WebGL buffers (position, color, indices).
- *gl-utility.js*: Low-level WebGL helper functions for shader compilation and linking, and for updating buffer data.
- *DOM.js*: Contains functions for interacting with the HTML DOM, such as handling slider inputs and converting mouse coordinates.

5 Run Locally

This project uses ES6 modules, which requires it to be run from a local web server to function correctly.

1. **Clone the repository or download the source files.**

   ```
   git clone https://github.com/gathik-jindal/2D-Crowd-Simulation-Renderer.git
   ```

2. **Navigate to the project directory.**

   cd 2D–Crowd–Simulation–Renderer

3. **Start a local web server.** A simple way is to use Python's built-in HTTP server.

   - If you have Python 3:

     python –m http.server

   - If you have Python 2:

     python –m SimpleHTTPServer

   Alternatively, you can use a development tool like the Live Server extension for VS Code.

4. **Open your web browser** and navigate to the local server's address (e.g., `http://localhost:8000`).

6 Dependencies

This project relies on two external libraries included via CDN:

- *gl-matrix*: A JavaScript library for high-performance vector and matrix math.
- *cdt2d*: A library for performing 2D Constrained Delaunay Triangulation.

7 Questions To Be Answered

7.1 Explain your strategy for user interactions?

The user interaction strategy for this project is designed to be both comprehensive and intuitive, providing clear and distinct controls for each of the application's core functionalities. The overall approach is event-driven, responding to user input from the keyboard, mouse, and HTML sliders to dynamically update the simulation state and rendering in real-time.

The strategy is broken down into four main components:

- *Obstacle Manipulation (Keyboard)*: Transformations of the central obstacle are mapped to standard keyboard inputs familiar from gaming, ensuring ease of use. The 'W', 'A', 'S', and 'D' keys control translation, 'Q' and 'E' handle rotation, and 'O' and 'P' manage uniform scaling. This input is processed in each frame of the main render loop, allowing for smooth and continuous transformations. A 'clamp' function ensures the obstacle correctly remains within the screen boundaries, even when rotated.

- *Simulation Parameter Control (UI Sliders)*: Key simulation parameters—the number of triangulation vertices ("dots"), the number of agents ("people"), and the population density threshold—are controlled via HTML range sliders. This provides a constrained and visual method for users to modify the scene's complexity and the rules for density visualization. An 'input' event listener on each slider triggers an immediate and complete regeneration of the relevant points and a full update of the simulation, providing instant feedback.

- *Direct Agent Manipulation (Mouse Drag)*: To fulfill the requirement of moving agents between triangles, a direct manipulation approach was implemented. The user can click and drag any yellow "person" point. This is managed by a state machine ('isDragging', 'draggedPointIndex') and a set of 'mousedown', 'mousemove', and 'mouseup' event listeners. Mouse coordinates in screen space are converted to world space to accurately pick and move the agents in real-time. Upon releasing the mouse, the simulation's 'update()' function is called to recalculate the density visualization, showing the immediate consequence of the user's action.

- *Triangulation Editing (Mode-Based Interaction)*: To handle the complex task of manually editing the triangulation mesh, an explicit "Edit Mode" system was implemented, controlled by UI buttons. This separates the functionality of agent dragging from mesh editing, preventing user error and ambiguity. This approach follows the assignment's hint that interactive editing is a practical alternative to full automation.
  - In *"Add Triangle"* mode, clicks select vertices. Once three vertices are selected, a new triangle is added to the mesh's index buffer.
  - In *"Delete Edge"* mode, a 'findClosestEdge' helper function is used to identify the edge nearest to the user's click. All triangles sharing this edge are then removed from the index buffer. In both cases, a call to the central 'update()' function synchronizes the visual representation with the new mesh structure.

7.2 How is scaling about a corner of the quadrilateral different from that of the center?

Scaling about a corner is fundamentally different from scaling about the center because *all scaling transformations are performed relative to the origin (0,0) of the current coordinate system.* To scale an object around a point other than its origin (like a corner), the object must first be moved so that the desired fixed point is at the origin.

Here is a breakdown of the two processes:

7.2.1 Scaling About the Center

This is the simpler operation and the one implemented for the obstacle in this project. It assumes the object's model is defined such that its geometric center is at the origin '(0,0)'.

- *Scale*: The object is scaled by the desired factor. Since its center is at the origin, all its vertices move proportionally away from (or towards) the center. The object grows or shrinks in place.
- *Translate*: The correctly scaled object is then moved to its final position in the world.

The composite transformation matrix would be $M = \text{Translation} \times \text{Scale}$. When applied to a vertex V, the operation is $V' = T \cdot S \cdot V$. The scaling happens first, followed by the translation.

7.2.2 Scaling About a Corner

This is a multi-step process that uses a specific point on the object (a corner, P) as the fixed point of the transformation.

- *Translate to Origin*: The entire object is moved so that the corner P is at the coordinate system's origin. This is achieved by translating the object by the vector $-P$.
- *Scale*: With the corner now at the origin, the scaling operation is applied. All other vertices will move relative to this corner, which remains fixed at '(0,0)'.
- *Translate Back*: The object is moved back to its original position by translating by the vector $+P$.

This three-step process ensures that the corner's world position remains unchanged while the rest of the object scales around it. The composite transformation matrix would be $M = \text{Translate}_{(\text{back})} \times \text{Scale} \times \text{Translate}_{(\text{to origin})}$. The operation on a vertex V is $V' = T_P \cdot S \cdot T_{-P} \cdot V$.

8 Demo

You can view a demo of the application here: demo

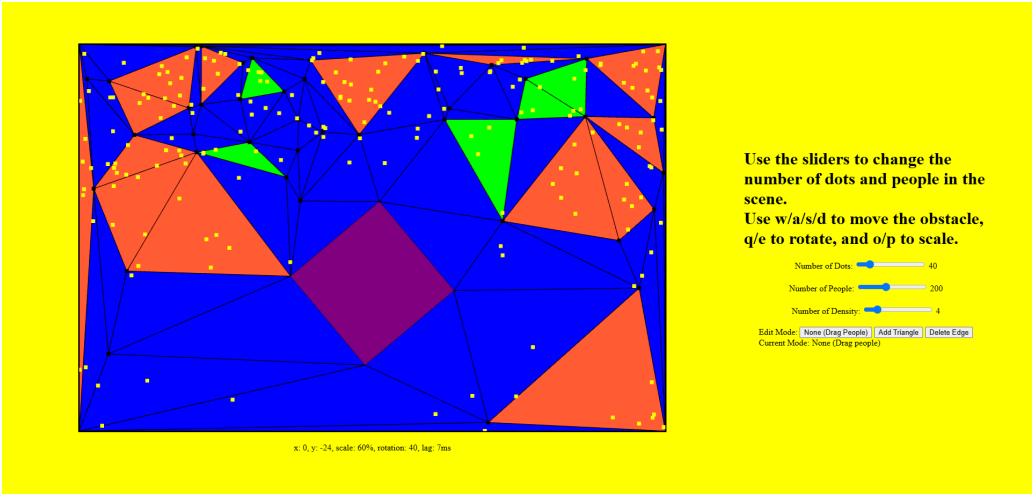

Figure 1: State after Transformation, scaling and rotation.

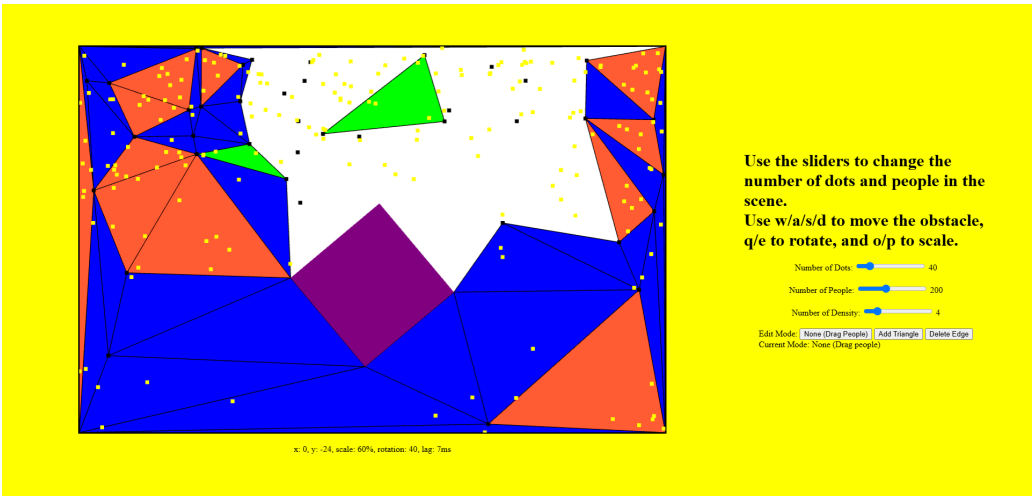

Figure 2: State after deleting edges and adding triangles.

9 Acknowledgments

I would like to acknowledge the resources that were critical to the completion of this project. The Mozilla Developer Network (MDN) Web Docs, OpengGL docs, GLMatrix docs all served as an essential technical reference for the WebGL API and most of the mathematical functions I needed.

Furthermore, I would like to mention Google's Gemini AI, which was instrumental as an interactive assistant throughout the development process. Its role included helping to debug complex WebGL issues, providing clear explanations of core computer graphics concepts, and assisting in the design and refactoring of the application's code. In particular, the re-triangulation function and some of the helper functions. Gemini was very helpful in providing snippets of code that I knew the logic of but not the correct way to write in javascript.