"Ghost excitations" near curved boundaries are a common artifact in grid-based fluid simulations. They often stem from inconsistencies in how the simulation handles the discrete representation of the curved surface, especially concerning:

1. **Pressure Solver (`solveIncompressibility`):** The discretization of the pressure Poisson equation ($\nabla \cdot (1/\rho \nabla p) = \nabla \cdot \mathbf{u}^* / \Delta t$) and the subsequent velocity update ($\mathbf{u}^{n+1} = \mathbf{u}^* - \Delta t /\rho \nabla p$) must correctly account for solid boundaries. Standard finite differences can struggle at irregular boundaries. The use of `s` values (0 for solid, 1 for fluid) as weights in the pressure solve (`var s = sx0 + sx1 + sy0 + sy1; ... var p = -div / s;`) is a way to handle this, but it might not perfectly enforce the zero normal velocity condition at the boundary after the pressure update.
2. **Velocity Boundary Conditions:** Ensuring that the velocity components on the faces of solid cells are correctly set to the boundary's velocity (0 for static walls, $v_{obs}$ for moving obstacles) is crucial. This needs to be done:
    - After transferring particle velocities to the grid (P2G).
    - And critically, *after* the pressure projection step, as the pressure solve itself can introduce small, non-zero velocities at boundary faces.
3. **Grid-Particle Transfer (`transferVelocities`):** Interpolating velocities to/from particles near these "stair-stepped" boundaries can also contribute if the grid velocities themselves are not well-behaved.

## Understanding the MAC Grid

The **Marker-And-Cell (MAC) grid** is a foundational concept in grid-based fluid simulations, including the FLIP method discussed here. It employs a "staggered" arrangement for velocity components, which offers significant numerical advantages.

- **Scalar Quantities at Cell Centers:** Variables such as pressure ($p$), density ($\rho_{particle}$), and cell type markers (e.g., `s` indicating solid/fluid, or `cellType`) are defined at the **center** of each grid cell.
- **Vector Components at Cell Faces:** The velocity vector $\mathbf{u} = (u, v)$ is not stored directly at the cell center. Instead, its components are offset:
    - The **x-component of velocity ($u$)** is defined at the center of the **vertical faces** of grid cells (i.e., the left and right sides).
    - The **y-component of velocity ($v$)** is defined at the center of the **horizontal faces** of grid cells (i.e., the bottom and top sides).

**Visual Representation (2D):**

Consider a grid cell indexed by `(i, j)` with dimensions $\Delta x$ and $\Delta y$ (often $\Delta x = \Delta y = h$, the grid spacing):

```
      +--- v(i, j+1) ---+  <-- v-velocity at top face of cell (i,j)
      |                 |      (sometimes noted as v_ij+1/2 or v_i,j+1 in this
context)
      |                 |
u(i,j)--- P(i,j) --- u(i+1,j) <-- u-velocities at left/right faces of cell (i,j)
```

```
      |    s(i,j)       |      (sometimes noted as u_i-1/2,j and u_i+1/2,j or
u_i,j and u_i+1,j)
      |                 |
      +---  v(i,j)  ---+  <-- v-velocity at bottom face of cell (i,j)
                           (sometimes noted as v_i,j-1/2 or v_i,j)
```

In the context of the JavaScript `FlipFluid` simulation, if `f.u` and `f.v` are arrays of size `fNumX * fNumY`, a common interpretation linked to the `solveIncompressibility` pressure updates is:

- `f.u[i*n + j]` represents the u-velocity on the *left* face of cell `(i,j)`.
- `f.v[i*n + j]` represents the v-velocity on the *bottom* face of cell `(i,j)`. Where `n = f.fNumY`.

The divergence calculation for cell `(i,j)` then naturally becomes: $\nabla \cdot \mathbf{u}_{ij} \approx \frac{u(i+1,j) - u(i,j)}{\Delta x} + \frac{v(i,j+1) - v(i,j)}{\Delta y}$

**Advantages of the MAC Grid:**

- **Numerical Stability:** Prevents "checkerboard" pressure oscillations common in collocated grids (where all variables are at cell centers).
- **Accurate Discretization:** Facilitates stable and accurate finite difference approximations for pressure and velocity updates.
- **Conservation:** Helps in formulating schemes that better conserve mass and momentum.

## Theoretical Proposal for Improved Boundary Handling

The most direct way to address ghost excitations from pressure solver issues at boundaries is to **re-enforce the known boundary velocities on the MAC grid *after* the pressure projection step.**

- **What was missing/insufficient:** The original `solveIncompressibility` function updates grid velocities based on the calculated pressure field. While the `s` values (0 for solid, 1 for fluid/air) influence the pressure solve by scaling the pressure gradient's effect, there's no explicit step *after* this velocity update to guarantee that grid velocities at solid faces strictly adhere to the boundary conditions (e.g., $u_n=0$ for a static wall, or $u_n = v_{obstacle} \cdot n$ for a moving one). The "restore solid cell velocities" step in the `transferVelocities` function (during Particle-to-Grid transfer) uses `prevU`/`prevV`, which are the grid velocities *before* the current timestep's pressure solve, so they don't reflect the corrections made by the pressure solver.

- **Proposal:** After the main iteration loop in `solveIncompressibility` (where `this.u` and `this.v` are updated by pressure gradients), a new code section is added. This section iterates over all MAC grid velocity components (`u` and `v`). For each component, it identifies the two grid cells it lies between. If either of these cells is determined to be part of a solid boundary (either the static circular wall or the draggable obstacle), the velocity component is reset to the known velocity of that solid boundary.

- **Relevant Equations/Logic:**

    - The core principle is to enforce the no-slip (for static walls) or no-penetration/specified-velocity (for moving obstacles) condition. For a velocity component $u_{face}$ normal to a boundary surface element, we want $u_{face} = v_{boundary, normal}$.
    - For a $u$-velocity component $u_{i,j}$ (on the left face of cell `(i,j)`, between cell `(i-1,j)` and `(i,j)`):

        - If cell `(i-1,j)` is static solid (e.g., part of the circular wall), then $u_{i,j} = 0$.
        - If cell `(i,j)` is static solid, then $u_{i,j} = 0$.
        - If cell `(i-1,j)` is part of a draggable obstacle (and neither `(i-1,j)` nor `(i,j)` is static solid), then $u_{i,j} = v_{obstacle,x}$.
        - If cell `(i,j)` is part of a draggable obstacle (and neither `(i-1,j)` nor `(i,j)` is static solid), then $u_{i,j} = v_{obstacle,x}$. (Static solid conditions take precedence over draggable obstacle conditions for a given face).
    - Similar logic applies to $v$-velocity components $v_{i,j}$ (on the bottom face of cell `(i,j)`, between cell `(i,j-1)` and `(i,j)`), using $v_{obstacle,y}$.
    - This step ensures that the velocity component normal to any solid boundary face is correctly set, effectively projecting out any velocities that would cause fluid to flow into or out of solid regions after the pressure solve.

## Implementation Details

To implement this, helper functions are introduced to determine the type of a given grid cell:

```
// Helper to check if cell (ix, iy) is part of the static circular wall
function isCellStaticWall(ix, iy, fluidObject, sceneObject) {
    if (ix < 0 || ix >= fluidObject.fNumX || iy < 0 || iy >= fluidObject.fNumY)
return true; // Out of bounds is wall

    var cellRealX = (ix + 0.5) * fluidObject.h;
    var cellRealY = (iy + 0.5) * fluidObject.h;
    var distSqToMainCenter = (cellRealX - sceneObject.circleCenterX) * (cellRealX
- sceneObject.circleCenterX) +
                             (cellRealY - sceneObject.circleCenterY) * (cellRealY
- sceneObject.circleCenterY);
    return distSqToMainCenter > (sceneObject.circleRadius *
sceneObject.circleRadius);
}

// Helper to check if cell (ix, iy) is part of the active draggable obstacle
function isCellDraggable(ix, iy, fluidObject, sceneObject) {
    if (!sceneObject.showObstacle) return false;
    if (ix < 0 || ix >= fluidObject.fNumX || iy < 0 || iy >= fluidObject.fNumY)
return false;

    var cellRealX = (ix + 0.5) * fluidObject.h;
    var cellRealY = (iy + 0.5) * fluidObject.h;
    var dxDraggable = cellRealX - sceneObject.obstacleX;
    var dyDraggable = cellRealY - sceneObject.obstacleY;
    return (dxDraggable * dxDraggable + dyDraggable * dyDraggable) <
(sceneObject.obstacleRadius * sceneObject.obstacleRadius);
}
```

The `solveIncompressibility` function in the `FlipFluid` class is then modified:

```
// Inside class FlipFluid:
solveIncompressibility(numIters, dt, overRelaxation, compensateDrift = true) {
    // ... (existing pressure solve iterations) ...
    // this.p.fill(0.0);
    // this.prevU.set(this.u); // Store u before pressure solve
    // this.prevV.set(this.v); // Store v before pressure solve
    //
    // for (var iter = 0; iter < numIters; iter++) {
    //     // Pressure update loop using s values to weight divergence
contribution
    //     // ... this.u and this.v are updated by pressure gradients ...
    // }

    // --- BEGIN NEW: Re-enforce boundary conditions on u and v after pressure
solve ---
    var currentScene = window.scene; // Access global scene object
    var n = this.fNumY;

    // Enforce u-velocities: u[idx = i*n+j] is velocity on the LEFT face of cell
(i,j).
    // It is horizontally between cell (i-1,j) and cell (i,j).
    for (var i = 0; i < this.fNumX; i++) { // Loop over grid cell columns
        for (var j = 0; j < this.fNumY; j++) { // Loop over grid cell rows
            var u_idx = i * n + j;

            // Cells adjacent to this u-velocity face: (i-1,j) and (i,j)
            var leftCellIsStatic = (i === 0) || isCellStaticWall(i - 1, j, this,
currentScene);
            var rightCellIsStatic = isCellStaticWall(i, j, this, currentScene);

            var leftCellIsDraggable = (i > 0) && isCellDraggable(i - 1, j, this,
currentScene);
            var rightCellIsDraggable = isCellDraggable(i, j, this, currentScene);

            if (leftCellIsStatic || rightCellIsStatic) {
                this.u[u_idx] = 0.0; // Static wall condition
            } else if (leftCellIsDraggable || rightCellIsDraggable) {
                this.u[u_idx] = currentScene.obstacleVelX; // Draggable obstacle
condition
            }

            // Enforce v-velocities: v[idx = i*n+j] is velocity on the BOTTOM face
of cell (i,j).
            // It is vertically between cell (i,j-1) and cell (i,j).
            var v_idx = i * n + j;

            // Cells adjacent to this v-velocity face: (i,j-1) and (i,j)
            var bottomCellIsStatic = (j === 0) || isCellStaticWall(i, j - 1, this,
currentScene);
            var topCellIsStatic = isCellStaticWall(i, j, this, currentScene);

            var bottomCellIsDraggable = (j > 0) && isCellDraggable(i, j - 1, this,
currentScene);
```

```
            var topCellIsDraggable = isCellDraggable(i, j, this, currentScene);

            if (bottomCellIsStatic || topCellIsStatic) {
                this.v[v_idx] = 0.0; // Static wall condition
            } else if (bottomCellIsDraggable || topCellIsDraggable) {
                this.v[v_idx] = currentScene.obstacleVelY; // Draggable obstacle
condition
            }
        }
    }
    // --- END NEW ---
}
```

This explicit enforcement step, performed after the pressure solver has made the velocity field approximately divergence-free, helps to clean up artifacts at boundaries by ensuring the grid velocities respect the physical constraints imposed by solid objects. This is particularly important for curved or complex boundaries where the MAC grid's stair-step approximation can lead to inaccuracies if not carefully handled.