Final Report for EAS 6138 (RCCD) Rounded cell collision detection and Regions of Interest.

Jackie Bell
Department of Mechanical and Aerospace Engineering
University of Florida
Gainesville Florida,United States
0009-0001-4291-2230

***Abstract*—In this paper Rounded Cell Collision Detection (RCCD) was utilized to visualize and evaluate real flow of autonomous particles through a converging-diverging nozzle. We cover the mathematics of collision resolution with boundaries, represented as boundary particles, and elastic collisions between particles. The results are HSV colormapped and are then converted to velocity angle vector fields without interrupting the GPU.**

***Index Terms*—particle system, image space, collision detection, nearest neighbor, region of interest, GPU, GPGPU.**

I. INTRODUCTION

The paper is an extension to a paper entitled *Rounded cell collision detection on the GPU* [6], where Rounded Cell Collision Detection was performed on a set of static particles in various states of collision. In this paper we discuss RCCD while particles are in motion. There will now be time and distance involved in capturing collisions and certain types of speed limits. This will also involve exploring Regions of Interest (RI) more fully as they can be configured to provide any number of properties required to fulfill specific application requirements. Some applications may require hard or soft shell collisions, inelastic or elastic responses, and/or repulsion/attraction dynamics. Many applications will require the compute stage to execute on data a number of times so to balance energy before rendering and building a new potentially colliding set (PCS).

Nodal type particle type systems suffer some advantages and disadvantages compared to Computational Fluid Dynamics.

Particle methods are appealing in that they can simulate physical phenomena much more realistically than those generated by smooth equations. Real thermo-fluid flows exhibit a great deal of randomness and turbulence [7]. CPM also provides more information, not least of which is the varying direction of particles in the flow because of collision reactions with other particles and boundaries. Some authors see the movement away from *continua* as a paradigm shift towards *discrete populations* [4].

This nodal nature of SPH restricts real freedom of motion, and techniques have been developed to overcome this. For example, the *particle finite element method* (PFEM), or the *discrete element method* (DEM), enhance separation, such as water droplets developing off of a solid, while maintaining node connections [5]. A popular particle combination is that of SPH with DEM. Categories of DEM are classified based on the governing dynamic. Discrete element methods can be statistical mechanical, Newtonian dynamical, or both (hybrid) [3].

Particles in DEM are discrete and exhibit freedom of movement but are not truly autonomous, they depend on integral of force or energy [3] to calculate displacement. They are effective with granular flows, powder rock, and soil mechanics [1].

Some of this study is based on what will be defined as an *autonomous particle* - that is a locally discrete kinetic particle as opposed to a nodal type particle as is the case of SPH and DEM for example. Also explored is a method of retrieving data for analysis called HSV color mapping. HSV color mapping provides two fold advantage over traditional methods. First, it does not require a transfer of data from the GPU to the CPU as the data is in the form of captured images from the CPU frame buffer. Analysis can then be performed by processing images. Second, the color map can be used to produce angle, velocity, and temperature flow fields - among others.

The is a coding assignment so the important codes are provided and discussed. All of the code provided here was developed for this paper as the previous codes only provided collision detection. Specifically, all motion, collision resolution, boundary conditions, and applications for evaluation were developed here.

One might wonder why there is not an emphasis on kinetic simulations in engineering and physics while it dominates in computer graphics. Both computer science and mechanical engineering require many years of study to master. Computer

graphics engineers are highly skilled in multi-dimensional calculus whereas thermo-fluid engineers utilize energy as the basis for their profession. That is not to say that computer graphics does not entail energy, it is just that the basis is force where mass and momentum provide the mathematics needed to make realistic graphics animations. Without energy there is no heat transfer, internal energy, or other thermodynamic phenomena - but again, these are not required as the emphasis is on speed and realism. This overall effort is a multi-disciplinary effort to close this gap and to find ways to incorporate advanced physics like heat transfer. We have successfully experimented with strategies to perform these calculations but this is not the subject matter here.

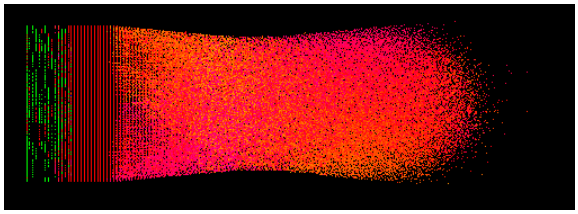


Fig. 1: *Flow thorugh a Converging-Diverging Nozzle HSV colorized to reflect the velocity angle.*

II. Problem Specification- The Converging Diverging Nozzle

The converging-diverging nozzle (CDN) is a passive device which exhibits a vast range of thermo-fluid behavior. Flow is substantially impacted by velocity magnitude but to a greater extent the velocity direction. Buffers form in the compressive section of the nozzle as a result of eddy's which choke the flow and which are far too complex and random to be described by smooth functions. Modeling the CDN also provides an opportunity to preview a number of RCCD features with Regions of Interest.

The objective of this demonstration is to simulate flow trough a CD nozzle with particle-particle, particle-boundary collision detection, and resolution while extracting and reporting velocity angles without CPU-GPU data transmission.

RCCD utilizes Regions of Interest (RI) to provide intersection detection where the phenomena resulting from the overlap can vary. Any modeled entity can be represented by many regions of interest. Most often the region is the *hard sphere* (distance basis) boundary of a particle but can also be the boundary of an attraction or repulsion field. The RI can can represent a neighbor region to ascertain the number of close neighbors. RI can represent an additional region around a particle utilized for Time of Impact (TOI) calculations in *soft-sphere* (time basis) adaptations allowing for higher speed simulations. Regions of interest can also represent *boundary particles* which not only alert to boundaries, but contain additional information about them.

This demonstration is about the computational mechanics of RCCD with RI more than it is about accurate thermo-fluid flow, so it incorporates some simplifying assumptions. The goal is to illustrate some kinetics of locally discrete *autonomous* particles, how these might be implemented on the GPU, and how information can be extracted from the simulation without CPU-GPU data transfer.

This demonstration is also a step forward in the development of this system and as such the dynamics are only as advanced as required to achieve the desired goals. The simplifying assumptions that have been implemented are as follows.

The particles are hard-sphere and are also perfectly elastic where contact determination is made by distance. The time step and particle speed are 'hard-coded' to insure particles have time to react with each other and boundaries. As the approach matures the equations for speed limits will be derived and would be sourced from configuration files.

The particle should be traveling at a high enough speed to completely rebound from interactions but not so fast that the particles *tunnel*. The speed cannot be so slow that particles spend more than a few time steps in a collision but they must also have a buffer of time to spend in contact. In short, even though collision-resolution is elastic-hard-sphere there is a small buffer of time regions remain in contact. This allows the collision to remain conservative with some buffering.

Conservation is another simplification in that energy is not exchanged with other particles, borders, or heat. This has a significant impact on the range of flow that can be simulated since particles will always maintain velocity magnitude. Particles cannot slow by transferring kinetic energy into potential (internal) energy. If particle density is too high particles can be pushed outside boundaries, clog the nozzle, or they can bind and rotate. Bounded rotation is a highly desirable feature except in current instance where it can disrupt the expected flows. The result of these considerations is that the density of particles cannot be so high that particles do not have space to react but not so low that they do not accurately represent the expected flow.

The simulation could be considered to represent the flow of spherical abrasive in sandblasting equipment and in the future these can be mixed with inelastic gaseous collisions.

The code is written in C++, Vulkan, and Matlab.

III. Numerical methods and experimental setup

This section covers the theory of 3-d collisions between particles and particles with boundaries. These collisions change velocity so the final section discusses position update.

A. *Particle-Particle Collision Resolution*

Particle-particle collision is a vast subject, but as referenced earlier, these collisions are elastic. Although not very realistic, these collisions require only an equation for momentum exchange in three dimensions. Rounded cell collision detection processes collisions for each particle, not each particle pair. Therefore, each particle in a collision is resolved separately. This requires the same calculation twice. The particle being processed, and all particles against which it is being processed. The particle being processed is called the *source particle* and the particle, or particles, against which it is being compared are called the *target particles*.

Collision resolution is processed in the compute pipeline as shown in the code of fig. 2. The *potentially colliding set* is delivered to the compute kernel by the graphics pipeline. Starting at line 2, the code iterates over the eight corners of the *axis aligned bounding box* (AABB) encasing the particle sphere, seeking other particle corners that occupy the same cell. The code traverses the length of the array which contains all of the particles in the cell at line 12. When a particle is found, it is registered in a duplicates array, since two or more particle corners can span the same cells. If it is not a duplicate the `ProcessParticleContact(..)` function is called in fig. 3.

```
1  ///.........
2      for(uint ii = 0; ii < 8;ii++)
3      {
4          // Set location to local variable.
5          uint loc = P[Findex].zlink[ii].ploc;
6
7          // If the lcation is not zero..
8          if (loc != 0)
9          {
10             // Use the linked particle location to index into the particle-cell hash table
11             // And compare this particle with all of the paricles at this location.
12             for(uint jj = 0; jj< MAX_ARY;jj++)
13             {
14                 Tindex = clink[loc].idx[jj];
15
16                 // If the linked particle is zero teminate
17                 if(Tindex == 0)
18                     break;
19
20
21
22
23                 for(uint ii = 0; ii <= dupcnt && dupcnt < 256; ii++)
24                 {
25                     if(duplst[ii] == Tindex)
26                     {
27                         dupflg = true;
28                         break;
29                     }
30                     if(duplst[ii] == 0)
31                     {
32                         dupflg = false;
33                         duplst[ii] == Tindex;
34                         dupcnt = ii;
35                         break;
36                     }
37                 }
38
39                 if(dupflg == false)
40                 {
41                     ProcessCDBoundary(Findex,Tindex,OutVel);
42                     ProcessParticleContact(ii,Findex, Tindex, OutVel);
43                 }
44             }
45         }
46     }
47 ///.........
```

Fig. 2: A snippet of the compute kernel which processes the PCS

The code in fig. 3 performs *contact determination* by using a squared distance determination between centers (line 28) and a squared distance of the sum of their radii (line 33). If the squared the distance between centers is less than the squared distance of the sum of their radii then they are in contact. The squared distances are used because the square root function is expensive on the GPU. If the particles are in contact then the resulting momentum reaction is calculated.

```
1  // Takes the index of two particles and detemines the distance between them
2  // If the distance is less than the sum of radii squared the are in comllsion.
3  // If collsiong increment the collsion counter.
4  uint ProcessParticleContact(uint crnr, uint Findex, uint Tindex, in out vec3 OutVel)
5  {
6      if(Findex == Tindex || Tindex <= bbound)
7          return 0;
8
9      vec3 U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
10
11     float xT = P[Findex].PosLoc.x;
12     float yT = P[Findex].PosLoc.y;
13     float zT = P[Findex].PosLoc.z;
14
15     float xP = P[Tindex].PosLoc.x;
16     float yP = P[Tindex].PosLoc.y;
17     float zP = P[Tindex].PosLoc.z;
18
19     float Fm    = P[Findex].MolarMatter;
20     float Ft    = P[Tindex].MolarMatter;
21     vec3 InPosF = P[Findex].PosLoc.xyz;
22     vec3 InPosT = P[Tindex].PosLoc.xyz;
23     vec3 InVelF = P[Findex].prvvel.xyz;
24     vec3 InVelT = P[Tindex].prvvel.xyz;
25     vec3 newVel;
26
27     // Get squared distance between centers
28     float dsq = ((xP-xT)*(xP-xT)+
29                  (yP-yT)*(yP-yT)+
30                  (zP-zT)*(zP-zT));
31
32     // Get sqaured diameter
33     float rsq = ((P[Findex].PosLoc.w+P[Tindex].PosLoc.w)*(P[Findex].PosLoc.w+P[Tindex].PosLoc.w));
34
35     // If square of distance is less than square of radii there is a collision.
36     if (dsq <= rsq )
37     {
38         // This particle has collision
39         P[Findex].ColFlg = 1;
40
41         // Make sur this is not a duplicate collision.
42         if(inColl(Findex,Tindex))
43         {
44             return 0;
45         }
46         // Calulate the resolution
47         CalcMomentum(Findex,Fm,Ft,InPosF,InPosT,InVelF,InVelT,newVel);
48         P[Findex].VelRad.xyz = newVel;
49         // Count collisions
50         atomicAdd(collOut.CollisionCount,1);
51         return -1;
52     }
53     else
54     {
55         // Not in collsion anymore
56         P[Findex].ColFlg = 0;
57         // Clear dup flags
58         ClearCflg(Findex,Tindex);
59     }
60
61
62     return 0;
63 }
```

Fig. 3: The code for `ProcessParticleContact(...)` does a distance based contact determination.

The momentum equations use the dot product to project velocity onto the the line of impact, which is the tangent plane bisecting the line between the particle centers, as illustrated in Fig. 4.

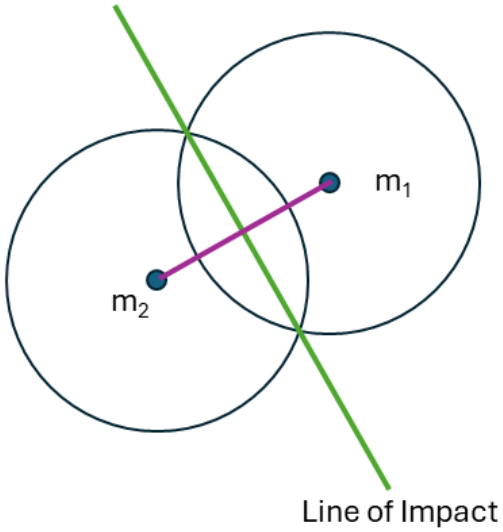


Fig. 4: *Gradient Field Appication.*

Equations (1-4) calculate parameters for the source particle.

$$\hat{x} = \left(\vec{P_F} - \vec{P_T}\right)/||\vec{x}|| \tag{1}$$

$$\vec{x_f} = \hat{x} \bullet \vec{v_f} \tag{2}$$

$$\vec{v_{fx}} = \hat{x}\,(\vec{x_f}) \tag{3}$$

$$\vec{v_{fy}} = \vec{x_f} - \vec{v_{fx}} \tag{4}$$

Equations (5-8) calculate parameters for the target particle.

$$\hat{x} = -\hat{x} \tag{5}$$

$$\vec{x_t} = \hat{x} \bullet \vec{v_t} \tag{6}$$

$$\vec{v_{tx}} = \hat{x}\,(\vec{x_t}) \tag{7}$$

$$\vec{v_{ty}} = \vec{x_f} - \vec{v_{tx}} \tag{8}$$

The velocity for the source particle is calculated as in 9.

$$\vec{V_f} = \vec{v_{fx}}\frac{(m_1 - m_2)}{m_1 + m_2} + \vec{v_{tx}}\frac{2(m_2)}{m_1 + m_2} + \vec{v_{ty}} \tag{9}$$

Fig. 6 shows this code in glsl a language that is 'built' for multidimensional calculus. For example, like templates in C++, variables types, like arrays and matrices, can be mulitplied, divided, added, subrtacted, compared, etc, just like scalars. The GPU is also designed to process these type very efficiently.

```
uint CalcMomentum(  uint Findex,            // Source particle index
                    float Fm,               // Source mass
                    float Ft,               // Target mass
                    vec3 InPosF,            // Source position
                    vec3 InPosT,            // Target position
                    vec3 InVelF,            // Source velocity
                    vec3 InVelT,            // Target velocity
                    in out vec3 newVel)     // Returned new velocity
{

    float m1, m2, x1, x2;
    vec3 v1temp, v1, v2, v1x, v2x, v1y, v2y;
    vec3 x = InPosT-InPosF;

    //Process source particle
    x = normalize(x);
    v1 = InVelF;
    x1 = dot(x,v1);
    v1x = x * x1;
    v1y = v1 - v1x;
    m1 = Fm;

    //Process target particle
    x = x*-1;
    v2 = InVelT;
    x2 = dot(x,v2);
    v2x = x * x2;
    v2y = v2 - v2x;
    m2 = Ft;

    //Return velocity for source particle
    newVel = vec3( v1x*(m1-m2)/(m1+m2) + v2x*(2*m2)/(m1+m2) + v1y );
    return 0;
}
```

Fig. 5: The code for `CalcMomentum` which performs a 3-d reflection around the line of impact between two particles.

B. Particle-Particle Boundary Resolution

Fig. 6 illustrates the CD nozzle modeled in this application. When displayed in the application it is a facade to provide the viewer a sense of proportions.

Fig. 6: *Actual Converging-Diverging Nozzel design.*

Fig. 7 shows the actual dimension of the CD nozzle. It has a very wide throat so to to insure low density flow. The measures are dimensionless and can be mm, cm, etc.

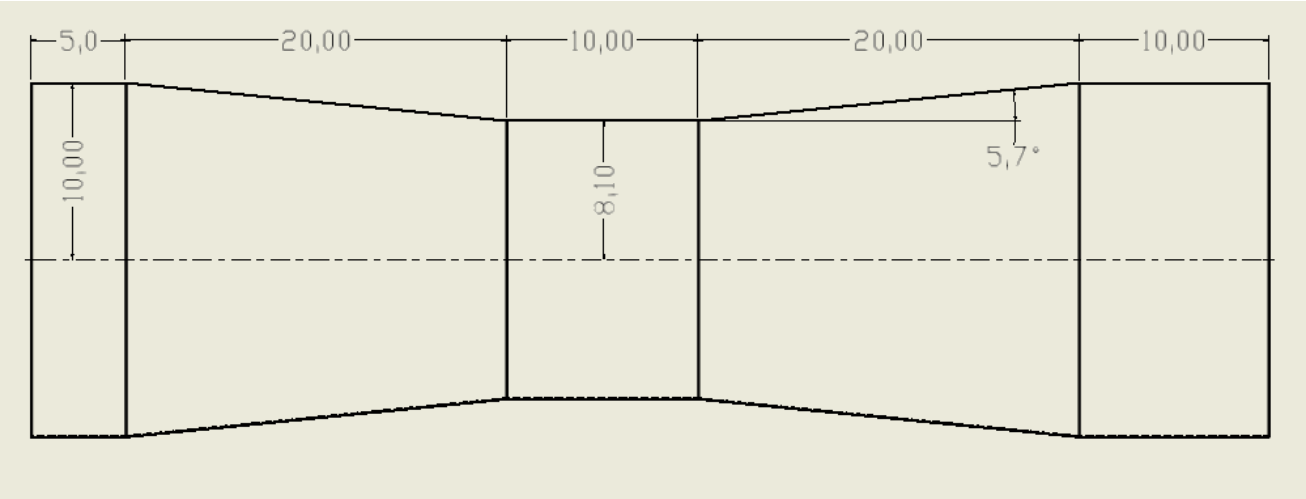


Fig. 7: *Converging-Diverging Nozzel design specifications.*

A number of Matlab applications were developed to analyze and debug logic. Fig. 8 shows the plot of the CD nozzle with points of intersection on the tangent plane. Fig. 8

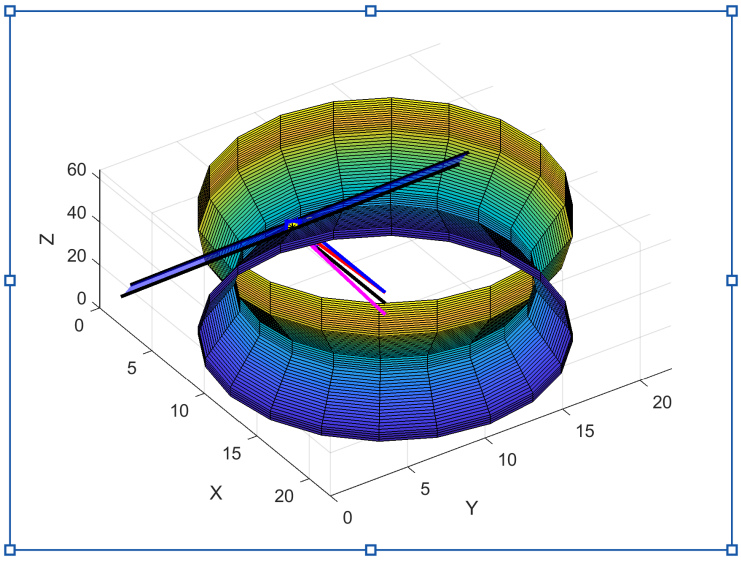


Fig. 8: *A converging-diverging nozzle showing a boundary collision points tangent plane and vectors.*

The challenge in boundary calculations is to establish the plane tangent to the boundary at the point of contact. This requires the establishment of three independent vectors terminating in the plane. Once the tangent plane is formed the normal vector is determined and the collision reaction is calculated by reflecting the particle around it.

The black asterisk represents point A (P_a), at the same angle as the particle, but at the radius of the boundary. The blue point B (P_b), represents a point at the same angle as the particle, but down the length of the nozzle at a lower radius. The red point C (P_c), represents a point at the same radius as point B but at a slightly different angle and therefore parallel to it. These three points constitute independent vectors terminating on a plane tangent to the slope of the nozzle at that point. From these points the normal of the tangent plane is calculated, $\vec{N}$, which is the mauve dotted line. The green vector is the incoming velocity (V_{in}), which is reflected around the normal to produce the output velocity (V_{out}), represented by the yellow line.

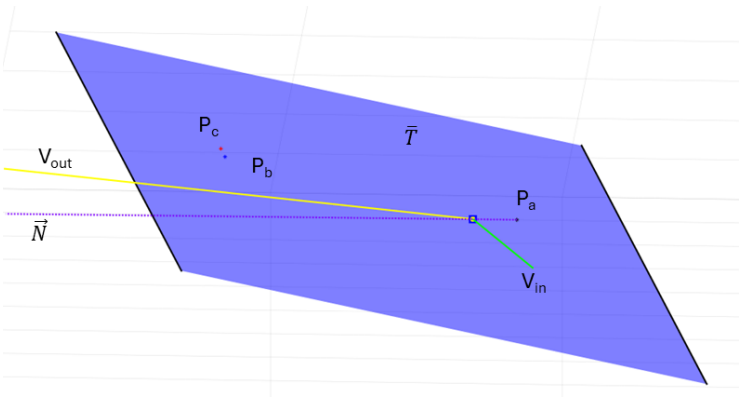


Fig. 9: *A plane tangent to a converging-diverging nozzle showing a boundary collision points, tangent plane and vectors. The black astrisk represents point A (P_a), at the same angle and depth as the particle, but at the radius of the boundary. The blue point B (P_b), represents a point at the same angle as the particle, but down the length of the nozzel at a lower radius. The red point C (P_c), represents a point at the same radius as point B but at a slighly different angle and therefore paralell to it. These three points consitute independent vectors terminating on a plane tangent to the slope of the nozzel at that point. From these points the normal of the tangent plane is calculated, $\vec{N}$, which is the mauve dotted line. The green vector is the incoming velcoity (V_{in}), which is reflected around the normal to produce the output velocity (V_{out}) represented by the yellow line.*

Where the boundary is flat the negative of the *xy* components of velocity are taken. When the three independent points are found the unit normal is calculated according to (10).

$$\hat{N} = \frac{\left(\vec{P}_a - \vec{P}_b\right) \otimes \left(\vec{P}_a - \vec{P}_c\right)}{||\vec{N}||} \tag{10}$$

With the unit normal in hand the reflected velocity out, ($\vec{V_{out}}$) is determined by (11)

$$\vec{V_{out}} = \vec{V_{in}} - 2.0(\vec{V_{in}} \bullet \hat{N})\hat{N} \tag{11}$$

Boundary particles are placed in every cell which contains a portion of the boundary. Fig. 10 shows a slice of the CDN with the boundary in black and boundary particles in red. Notice that there is a red particle in every cell through which the boundary travels.

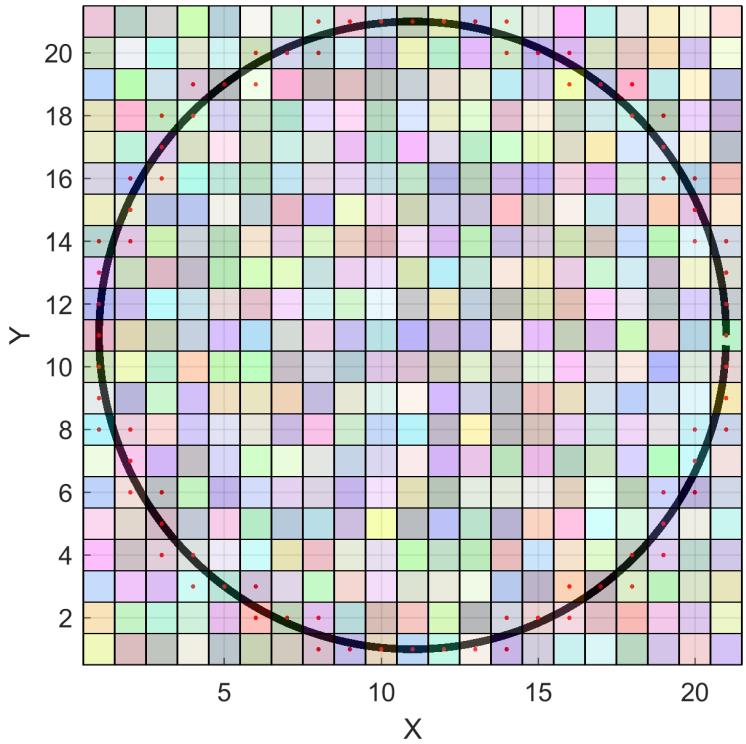


Fig. 10: *A slice of the CD nozzle showing the boundary passing through cells. A boundary particle, in red, occupioes each cell containing the boundary.*

Fig. 11 shows the complete nozzle with all the boundary particles in place.

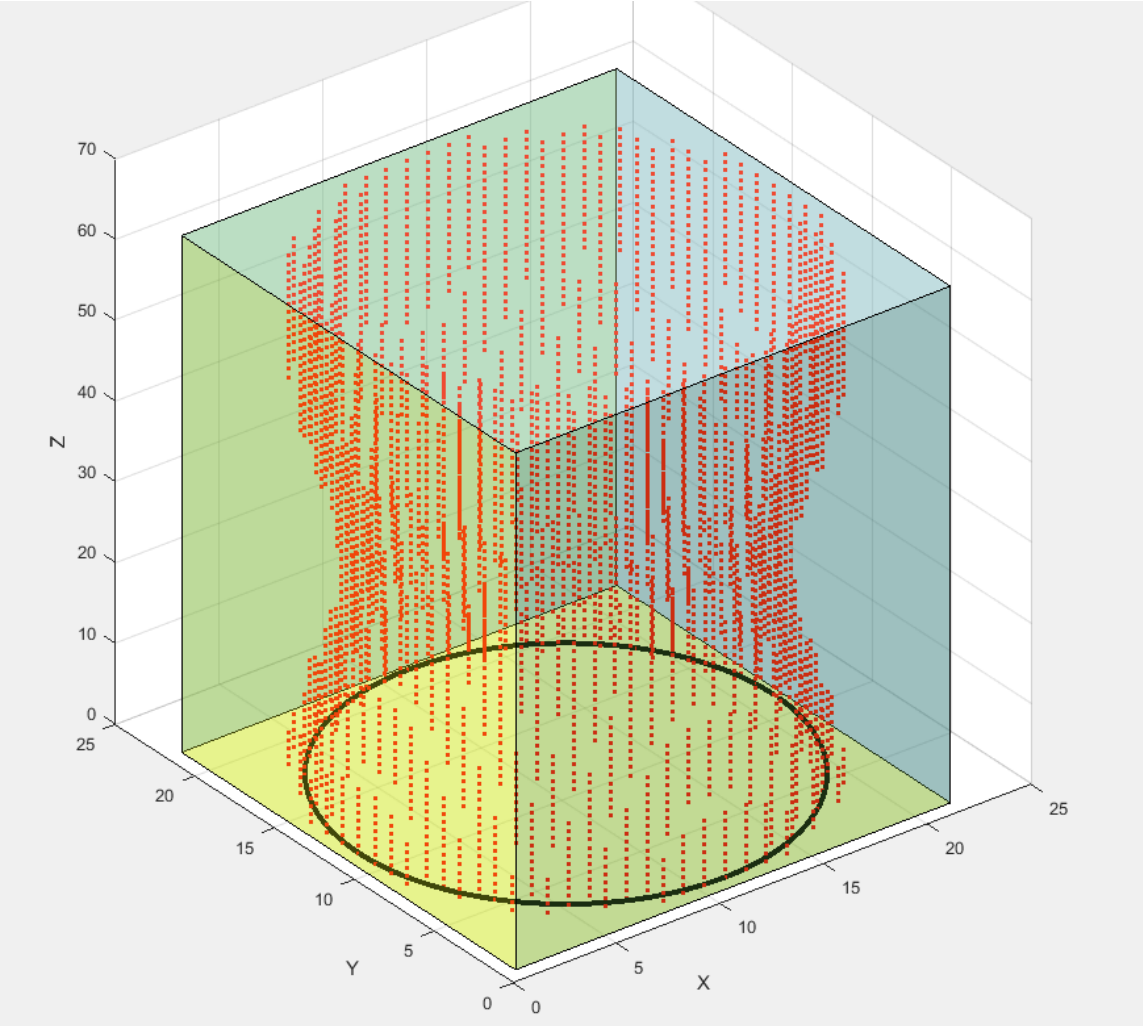


Fig. 11: *The CDN showing all boundary particles.*

Processing boundaries in this manner means that a particle will only check for a boundary if it is in a cell with one - this saves calculating boundary proximity for every particle every frame. Boundary particles are convenient in another manner. Notice that the equation (10) requires a normalized normal vector which in turn requires a square root. If the range of boundary that passes through the cell is considered infinitesimal, then the unit normal can be calculated and stored on the CPU before transfer to the GPU. This is a concept borrowed from computer graphics where normals are sent with triangle primitives.

The process of boundary collision resolution is a little bit more complicated. Fig. 12 shows the code for processing the boundary. It begins by insuring boundary particles are not considered since they are stationary. When the simulation setup is performed the boundary particles are added first, then the actual particles. Any particle number greater than the last boundary particle is then an moving particle (line 6).

The *active location* of the boundary is stored in the boundary particle velocity components. In this case they will all be non-zero if it is a particle boundary.

But, imagine a cube with boundary particles. If the boundary lay in the middle of a cube wall then the only location that will have a collision will be the single dimension. In a particle is in the corner of the cube the particle can reflect off of three sides. In this case, the three velocity components will be nonzero.

Next, in the same manner as particle-particle collision the squared distance between the particle and the wall is determined and if the the particle lay within it the boundary collision it is processed (line 33) by calling the `CalcBoundaryVel(..)` function, fig. 13.

Fig. 13 shows the code for the calculation of equations (10,11). There are some other considerations, such as contact with a flat.

```
1  void ProcessCDBoundary(uint Findex, uint Bindex, in out vec3 OutVel)
2  {
3
4      // Process this particle only if it is a particle, not a boundary particle.
5      // particle
6      if(Bindex > bbound)
7          return;
8
9
10     float tol = 0.5;
11     vec3 InPosF = P[Findex].PosLoc.xyz;
12     vec3 InVelF = P[Findex].VelRad.xyz;
13     vec3 InPosB;
14     vec3 InVelB;
15
16     // The positon of the target boundary particle is stored in the velocity.
17     // If any of these components are non-zero then we are in a cell with a boundary.
18     if(P[Bindex].VelRad.x != 0.0 || P[Bindex].VelRad.y != 0.0 || P[Bindex].VelRad.z != 0.0 )
19     {
20         float xT = P[Findex].PosLoc.x;
21         float yT = P[Findex].PosLoc.y;
22         float zT = P[Findex].PosLoc.z;
23         // Get the boundary radius at this point.
24         float radius = GetCDRadius(P[Findex].PosLoc.z);
25         // Square it.
26         float dsq = radius*radius;
27         // Get the squared position of the source particle.
28         float yr = abs(yT-CENTER)+2*P[Findex].PosLoc.w;
29         float xr = abs(xT-CENTER)+2*P[Findex].PosLoc.w;
30         float psq = ((yr*yr)+(xr*xr));
31
32         // If we are within the squared radius of the bpoundary process it.
33         if(psq >= dsq && P[Findex].bcs[0].clflg == 0)
34         {
35
36             newVel = CalcBoundaryVel(Findex,InPosF,InVelF,CENTER);
37             // Set the new velocity.
38             P[Findex].VelRad.xyz = newVel;
39             // Set the "in collsion" flag.
40             P[Findex].bcs[0].clflg = 1;
41         }
42         else
43         {   // Give it 6 frames to get out of range of the bouddary so it does not
44             // repeat the collsion. We could 'impulse' out of the boudary but then we
45             // may knock another particle out of the boudary. THis is hardcoded for this
46             // demonstration only.
47             if(P[Findex].bcs[0].clflg++ == 6)
48                 P[Findex].bcs[0].clflg = 0;
49         }
50
51     }
52
53 }
```

Fig. 12: The code for `ProcessCDBoundary(...)` which performs a distance calculation for a boundary particle.

If the collision is with a flat, then the `GetCDRadius(...)` function returns the negative of its length (line 19). If this is the case, only the sign of the xy velocity components need be changed (lines 25-28). If the particle is on the slope of the nozzle then the three independent points are determined as described. Care must be taken when getting independent points on the slope, to insure that the reach does not cross over into another section of the nozzle that is off the slope in question (lines 40-50).

The final component of boundary detection is the function that returns the radius at any point along the z direction of the nozzle fig. 14. This is a piecewise function that returns a positive value if the particle is on the slope of the boundary and a negative value if it is on a flat. If this function represented an equation then it could not only return the radius of the boundary at any point, but the tangent plane by taking the derivative.

```
vec3 CalcBoundaryVel(uint index, vec3 Pos, vec3 Vel, uint Center)
{
    uint startf = 4465;
    uint endf = 4470;
    uint particl = 4704;

    vec3 pointA;
    vec3 pointB;
    vec3 pointC;
    float radA;
    float radB;
    float lowZ;
    vec3 rvel;

    Get the angle of the particle in the XY plane.
    float angxy = atan2piPt(vec2(Pos.x-Center,Pos.y-Center));

    // Get the radius at this point.
    radA = GetCDRadius(Pos.z);

    // If the radius is negative we are on a flat
    // so just revers xyvelocity
    if(radA < 0.0)
    {
        rvel.x = -Vel.x;
        rvel.y = -Vel.y;
        rvel.z = Vel.z;
        return rvel;
    }

    // Get point A at the same angle of the particle but
    // on the boundary
    pointA.x = radA*cos(angxy)+Center;
    pointA.y = radA*sin(angxy)+Center;
    pointA.z = Pos.z;

    // For the next point we need a vector up or doen the slope
    // parallel to Point A. THis code makes sure we always get
    // that point from another part of the nozzle that is sloping.
    if(Pos.z >= secx2_beg && Pos.z  < secx2_end - 10)
        lowZ = pointA.z+4.0;

    else if(Pos.z >= secx2_beg+10.0 && Pos.z < secx2_end)
        lowZ = pointA.z-4.0;

    if(Pos.z >= secx4_beg && Pos.z < secx4_end - 10 )
        lowZ = pointA.z+4.0;

    else if(Pos.z >= secx4_beg+10.0 && Pos.z < secx4_end )
        lowZ = pointA.z-4.0;

    // Get the new radius for point B and calulate the vector.
    radB = GetCDRadius(lowZ);
    pointB.x = radB*cos(angxy)+Center;
    pointB.y = radB*sin(angxy)+Center;
    pointB.z = lowZ;

    // These two previous point are not independent since they
    // are parallel we need a point off to the side. Rotate
    // the point B vactor a little and esablish the last
    // indepented point on the plane
    pointC.x = radB*cos(angxy+PI/64)+Center;
    pointC.y = radB*sin(angxy+PI/64)+Center;
    pointC.z = lowZ;

    // Get the plane normal
    vec3 param1 = pointA-pointB;
    vec3 param2 = pointA-pointC;
    vec3 normvec = cross(pointA-pointB, pointA-pointC);
    float D = -dot(normvec, pointA);
    vec3 nnormvec = normalize(normvec);

    // Caclulate velocity reflection.
    rvel = Vel - 2.0*(dot(Vel,nnormvec)*nnormvec);

    return rvel;

}
```

Fig. 13: The code for `CalcBoundaryVel(...)` which performs the calculates required to reflect the particle around the boundary norm.

```
const float sec24slp = 0.1f;

const float secx1_beg = 1.0f;
const float secx1_end = 5.0f;
const float secz1     = 10.0f;

const float secx2_beg = 5.0f;
const float secx2_end = 25.0f;
const float secz2     = 10.0f;

const float secx3_beg = 25.0f;
const float secx3_end = 30.0f;
const float secz3     = 8.1f;


const float secx4_beg = 30.0f;
const float secx4_end = 50.0f;
const float secz4     = 10.0f;

const float secx5_beg = 50.0f;
const float secz5     = 10.0f;

float GetCDRadius(float Z)
{
    // SEC1 flat return negative of radius
    if(Z >= secx1_beg && Z <= secx1_end)
       return -secz1;

    // SEC2 slope down
    // 5.0 included to less than 25
    else if(Z > secx1_end && Z <= secx2_end)
       return secz2+(Z-secx2_beg)*(-sec24slp);

    // SEC3 flat return negative of radius
    else if(Z > secx2_end && Z <= secx3_end)
       return -secz3;

    // SEC4 slope up
    else if(Z > secx3_end && Z <= secx4_end)
        return secz3+(Z-secx3_end)*(sec24slp);

    // SEC5 flat
    else if(Z > secx4_end)
        return -secz5;

    else
        return 0;

    return 0;
}
```

Fig. 14: The code for `GetCDRadius(...)` which returns the radius of the nozzle at any point along its z-length.

C. Motion update

The position of the particle cannot be updated in the compute kernel. If the particle position is changed there it changes the distance between particles, so that collisions may be missed. It could also have undefined consequences as particles may get caught in a collision loop. Therefore the position is changed in the graphics pipeline, by the vertex kernel, before the particle corners are located again in the PCS.

Fig. 15 shows the code for position change. A number of checks are performed here, starting with determining if the particle exceeded the boundary(lines 21-25). If this happens the particle is disabled. The next checks are to determine if this run has motion enabled by configuration file, if the particle has been disabled, or if the simulation has been stopped or started by pressing the 'S' key (line 29).

The angle of the yz components of velocity are taken (lines 34-36) and are stored in the particle variable (line 37). This value is inserted in the first component of the HSV color map. It should be noted that the spherical angles of the velocity can also be determined and inserted into the first two components of the HSV color map. Additionally, even though the angles are taken in a slice of the yz plane, the 3-D dimension of the particle velocity can be plotted. For this demonstration we are only using the yz plane.

With all of the calculus being performed *not a number* (nan) can be returned. This is a major error and causes the simulation to shut down. Finally, if the particle has exited the nozzle it is disabled.

In this demonstration a total of 156,924 particles exhibited no boundary violations or nan's.

```
uint ChangePosCDNoz(uint index)
{

    // There was an error with this particle do not process it anymore
    if(P[0].PosLoc.w == 1.0)
        return 1;

    // Get radius squared to check to see if the
    // particle has violated a boundary
    float radius = GetCDRadius(P[index].PosLoc.z);

    float dsq = radius*radius;
    float yT = P[index].PosLoc.y;
    float xT = P[index].PosLoc.x;
    float yr = (yT-CENTER);
    float xr = (xT-CENTER);
    float psq = ((yr*yr)+(xr*xr));

    // If the positon of the particle is beyond a boundary
    // and it has not already been reported  disable it
    if(psq > dsq && uint(P[index].prvvel.w) == 0)
    {
        P[index].prvvel.w = 1.0;
        return 1;
    }

    // Calulate change in position if we are configured for motion,
    // and the particle is active, and the stop flag is not engaged
    if(doMotion == 1 && uint(P[index].prvvel.w) == 0 && uint(ShaderFlags.StopFlg) == 0)
        P[index].PosLoc.xyz += P[index].VelRad.xyz*dt;


    // Calculate the yz angle
    vec2 angnorm = normalize(P[index].VelRad.zy);
    float angletmp = atan2piPt(angnorm);
    // HSV Normalize and assign to particle angle variable
    P[index].FrcAng.w = atan2piPt(angnorm)/(2*PI);

    // Can't have nan's, abort simulation.
    if(isnan(P[index].FrcAng.w) || isnan(P[index].VelRad.x) || isnan(P[index].VelRad.y) || isnan(P[index].
          VelRad.z))
    {
        collIn.ErrorReturn = 7;
        collIn.particleNumber = index;
    }
    // Stop once the particle is beyond the nozzle
    if(P[index].PosLoc.z > 64.4)
        P[index].prvvel.w = 1.0;
    return 0;

}
```

Fig. 15: The code for `ChangePosCDNoz(...)` which calculates displacement after the PCS has been evaluated for collisions in the compute kernel.

D. Setup

In most applications the evaluation data is extracted from the source, in this case the GPU. But GPU-CPU communications is very expensive. Collecting data from a parallel system is also difficult and complicates the codes. When the GPU if finished it transfers a *swap frame* image to a frame buffer in video memory which is accessible by the CPU. When capturing a frame buffer with a simple screen capture program, the GPU is not impacted. The task then is to provide color information that is meaningful and to then extract the required data from that.

There is a source of high speed communication between the GPU and CPU called *push constants*. These would allow a built-in screen capture program to be configured to capture a series of frames. This would be convenient when there is the need to analyze changes frame by frame.

```
index = 0;
Sidelen = 64;
PipeCenter = 11.0000;
PipeRadius = 10.0000;
CellAryW = 20;
CellAryH = 20;
CellAryL = 64;
radius = 0.2;
PartPerCell = 8;
pcount = 5304;
colcount = 0;
dataFile = "J:/RCCDData/perfdataM/CDNozBoundaryTest.bin";
aprFile = "J:/RCCDData/perfdataM/CDNozBoundaryTest.csv";
density = 0.0;
pdensity = 0.0;
dispatchx =  5305;
dispatchy = 1;
dispatchz = 1;
workGroupsx = 1;
workGroupsy = 1;
workGroupsz = 1;
ColArySize = 128;
MaxSingleCollisions = 8;
```

Fig. 16: The benchset test configuration file provides information on each test to be run by ***rccdApp***

In order to show the process of evaluating the flow a simple experiment is set up. Fig. 17 shows the configuration. The image is misleading in that is shows only 6 blue particles on each side of the nozzle. In reality there are 720 particles which will be activated in sequence. Each set of particles have a sequence number stored in the particle structure. This represents the frame number at which the particle will become active which provides the means to stream or flow.

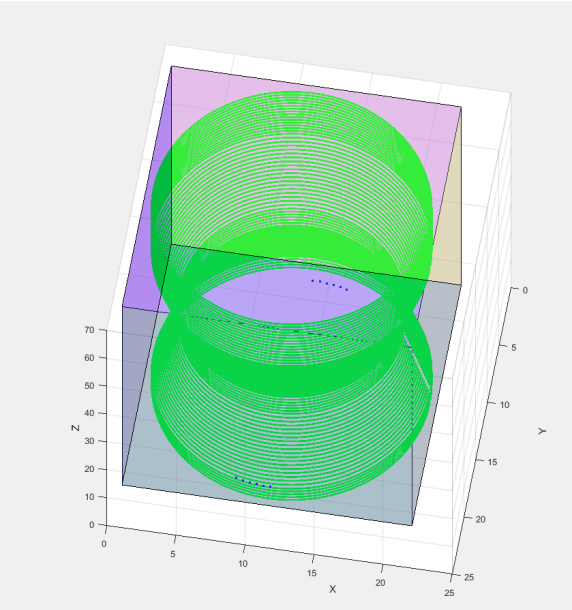


Fig. 17: *The setup for a simple demonstration. The particles in blue are arranged at the top and bottom of the starting point of the nozzle and aimed at the sloping sections.*

When the Matlab application generates the data for the simulation it creates two files, one binary holding the actual particles, and one configuration file, fig. 16. This file communicates the height, width, and length of the cell array (lines 5-7), the radius of the particle, the count of actual particles (line 10), the location of the binary data file to load (line 13), the thread dispatch configuration for the CPU (lines 16-18) and the workgroup threads for the GPU (lines 19-21).

Each cell can contain any number of particles which requires an array to store them. The *ColArySize* parameter (line 22) sets the size of this array. This parameter is determined at this point by experience since if the number of particles exceeds the number of slots available in the array the simulation will error and exit. All other parameters are not used.

The simple sequence is run and results of the flow are illustrated in fig. 18. In fig. 21a the boundary particles are enabled but interfere with the view. In fig. 21b they are disabled.

This view of this flow is in 3-d, but in a upcoming section we will take a slice of the flow in 2-d. Take note of the colors in fig. 18e where the top appears to be a purplish color while the bottom tends more towards the yellow.

Two Matlab applications were developed to evaluate the flow. The first, ***hsvmain***, fig. 19 provides an interface to view and adjust settings. Entering an angle in the *Angle Input field* plots an *Angle Vector* on the *HSV Map* color map indicating the direction in the color field. The color of that angle is also displayed in the *HSV Color Output* image box. The image being processed is illustrated in the *Image Evaluation Field*.

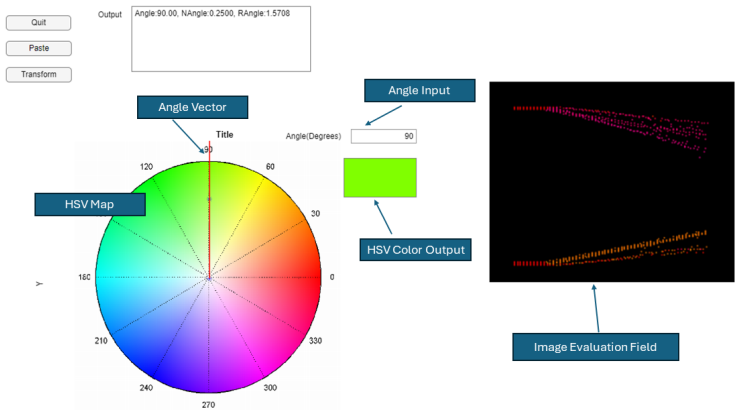
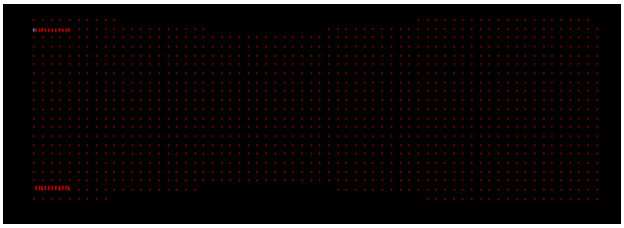
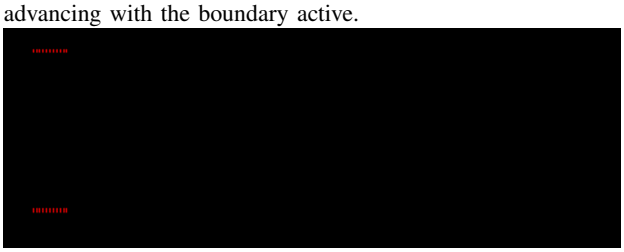


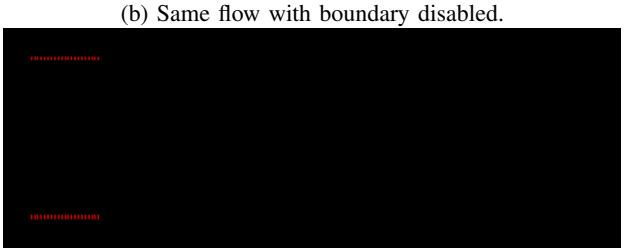
Fig. 19: *The setup for a simple demonstration. The particles in blue are arranged at the top and bottom of the starting point of the nozzle and aimed at the sloping sections.*



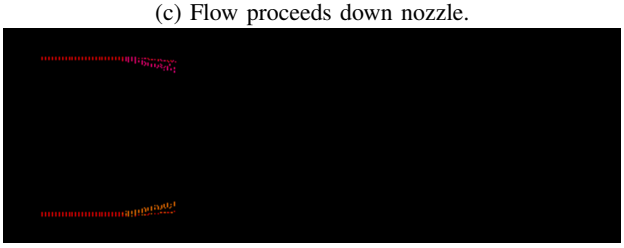
(a) Starting flow for the simple sequence shows the particles advancing with the boundary active.



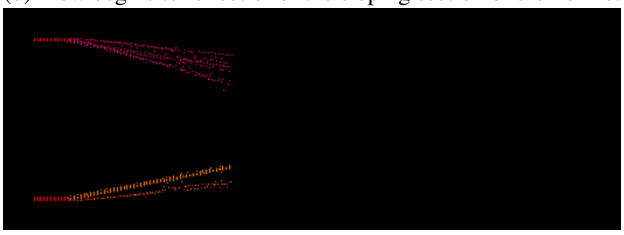
(b) Same flow with boundary disabled.



(c) Flow proceeds down nozzle.



(d) Flow begins to reflect of of the sloping section of the nozzle.



(e) Flow reflects off of sloping section.

Fig. 18: Series of images for the simple flow.

Activating the *Transform* button launches a child application illustrated in fig. 20. This application transforms the color field into a directional vector field as shown in the *Velocity Angle Gradient Field*.

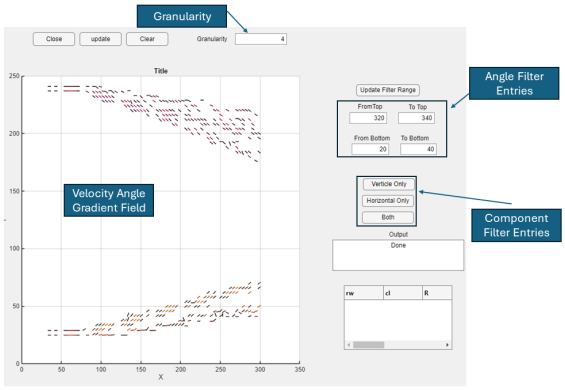
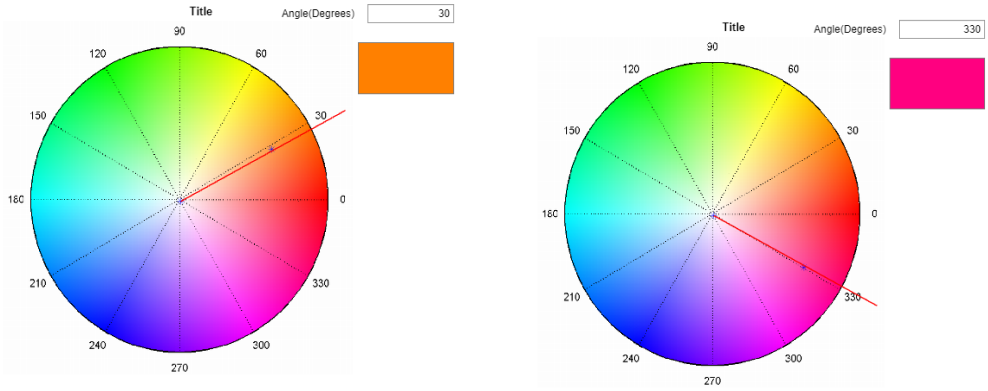


Fig. 20: *The setup for a simple demonstration. The particles in blue are arranged at the top and bottom of the starting point of the nozzle and aimed at the sloping sections.*

Images can be very complex and noisy, or the viewer may want to see specific information, so there needs to be a way to filter results. The granularity of the plot can be adjusted with the *Granulatiry* text box. Pixels are square in an image file, not round, but through a series of steps on the GPU and video system they are rounded. The granularity sets a square box of the side length which will evaluate every NxN set of pixels in the image.



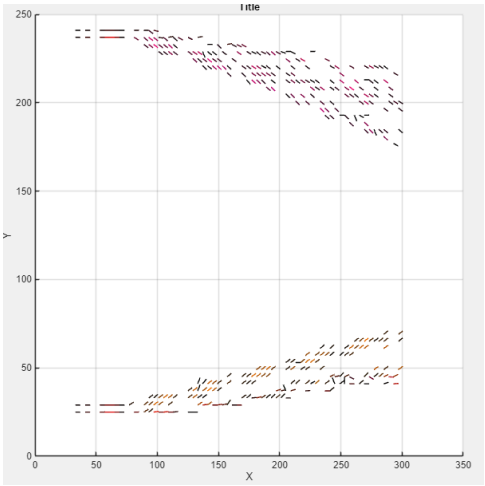
(a) HSV Color map of 30 degrees. The blue arrow is a vector pointing the same direction

(b) HSV Color map of 330 degrees. The blue arrow is a vector pointing the same direction

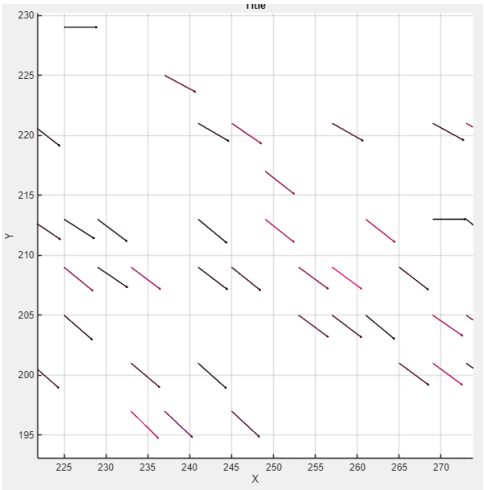
Fig. 21: Example of the HSV color map showing 30 and 330 degress angle.

The *Angle Filter Entries* filter for a rage of directions on the top and bottom of the flow. The *Component Filter Entries* filter for the horizontal or vertical components of velocity angles. Fig. 21 shows two HSV color maps representing 30 and 330 degrees. These colors seem to match the dominate colors in the *Image Evaluation Field* of fig. 20.

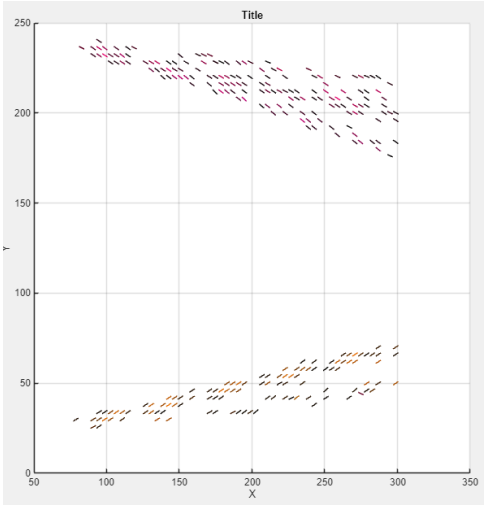
Performing an update in the plotting app (fig. 20) results in the vector plot of fig. 22a. To see the vectors more clearly we scale the top of the vector field in fig. 22b. Fig. 22c shows the angles filtered from 320 to 340 and 20 to 40 degrees while fig. 22d shows the same filtering but the gradient is set to its lowest value.



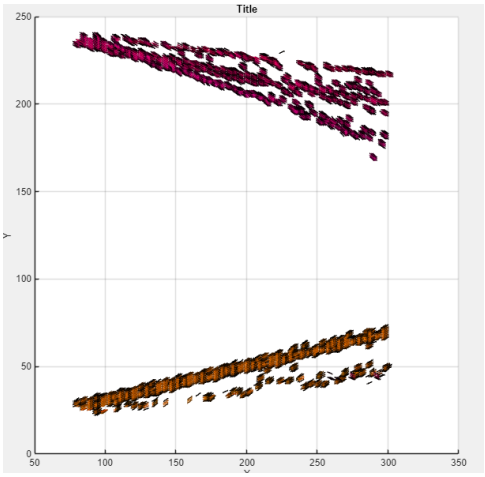
(a) Simple flow plot showing all particles



(b) Simple flow plot focued on the top of the flow with vectors scaled.



(c) Simple flow plot with vectors filtered from 320 to 340 degrees and from 20 to 40 degrees are filtered.



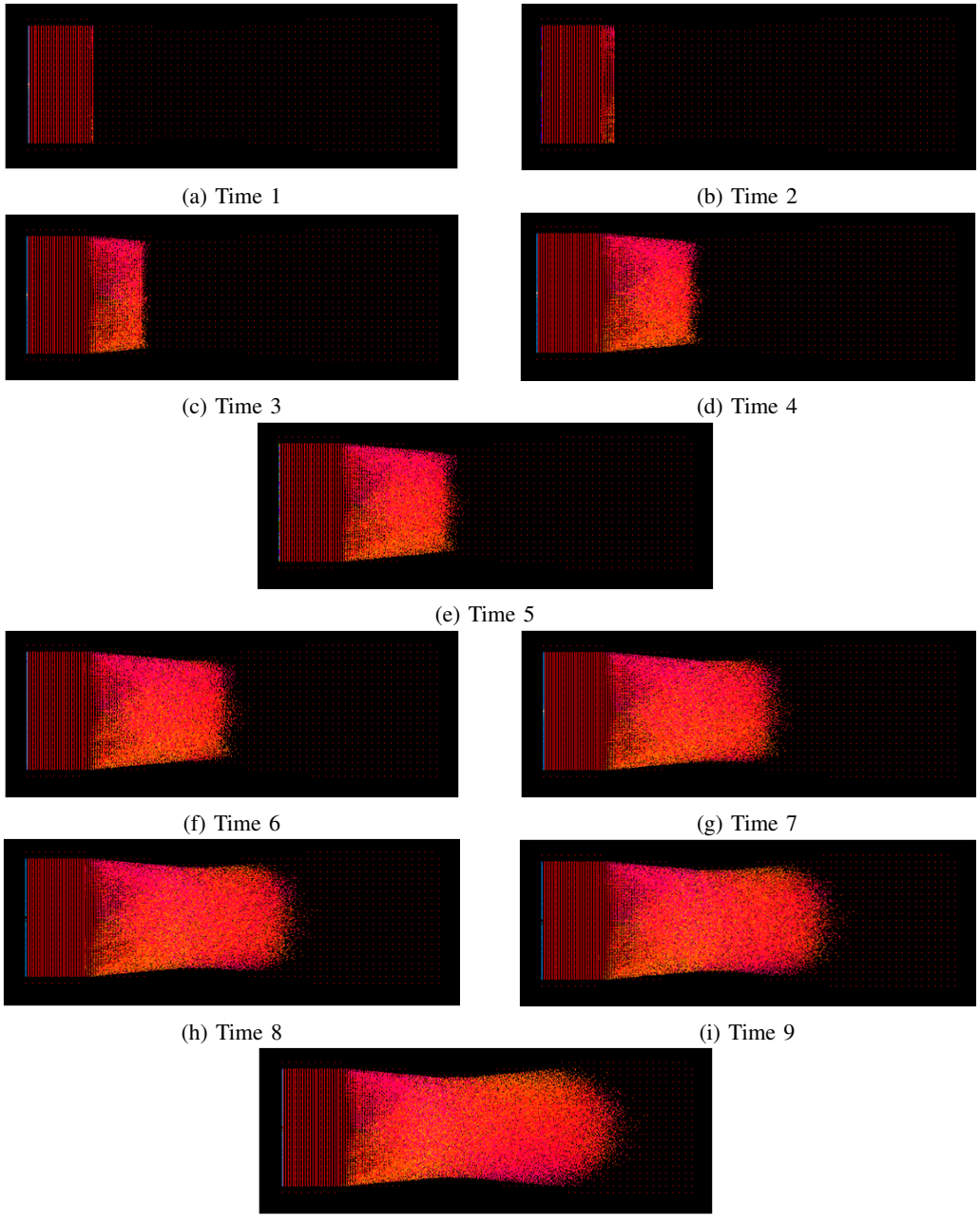
(d) Simple flow plot with vectors filtered from 320 to 340 degrees and from 20 to 40 degrees are filtered. The plot shows only these vectors but in this plot the granularity has been set to 1.0.

Fig. 22: A series of vector fields with various views and filters.

IV. Results and discussion

In this section we perform full flow through a CD nozzle with 161,508 particles in realtime. The rate of flow in this demonstration requires some time to develop but this is consequence of the infancy of this application. The demonstration is in relative dimensionless units, but for the sake of this argument assume units of meters. The nozzle then is 64 meters long with a particle speed (velocity magnitude) of 0.5 m/s and a time step of 0.01s. The particle then changes position at 50 mm per second. The full flow demonstration starts at almost 400 *fps* and slows to about 70 *fpos*. This frame rate for 161,508 particles aligns with the performance found in [2], Table III for between 115,712 and 246,784 particles which may be interpreted to mean that the additional load of collision resolution has little impact.

Fig. 23 shows 10 time slices of full flow notice how the purple and yellow areas form distinctive patterns.



(a) Time 1

(b) Time 2

(c) Time 3

(d) Time 4

(e) Time 5

(f) Time 6

(g) Time 7

(h) Time 8

(i) Time 9

(j) Time 10

Fig. 23: A full flow example image caputre at 10 different times.

We can analyze the flow by slicing the nozzle for a 2-d view. Fig. 24 shows a center yz slice of image fig. 23j. The viewer (camera, or eye, in computer graphics) is 240 units from the center of the nozzle. The nozzle is sliced from 240 to 241 units. Unfortunately there is not the particle density required to see patterns that are obvious in the full flow series (fig. 23).

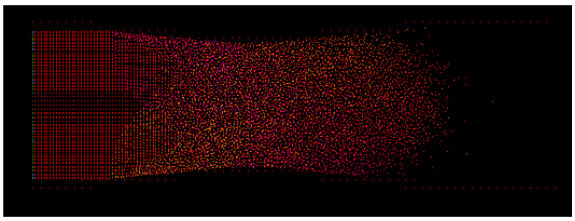


Fig. 24: *This shows a center yz slice of image fig. 23j. The viewer (cmaera,eye) is 240 units from the center of the nozzle. The nozzle is sliced from 240 to 241 units.*

A. Discussion -What can we learn?

Fig. 25 is a vector plot of the flow. Notice how the velocity angle form boundaries which are mirror images of the physical boundaries. As the flow increases in velocity magnitude we can envision these growing and clamping the flow.

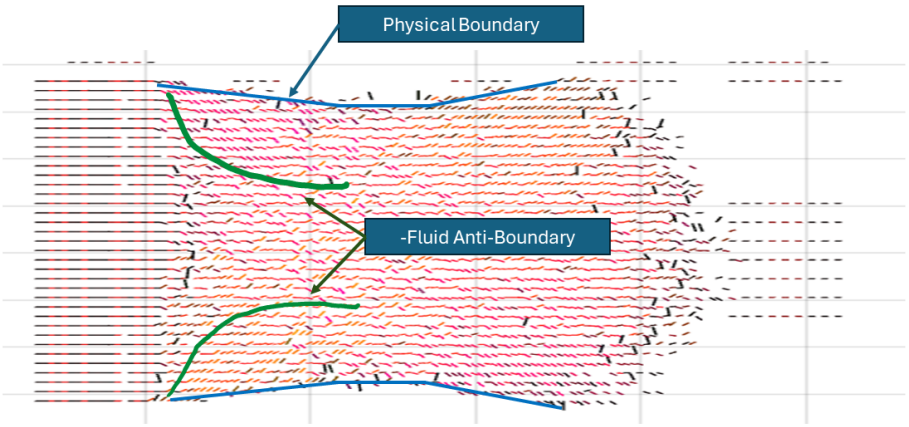


Fig. 25: *The vector plot of the flull flow example. The vetors on the top and bottom of the convering section seem to form a fluid boundary opposite in the direction of the physical boundary forming a kind of "Fluid Anti-Boundary".*

Fig. 26 is an illustration of a de Laval nozzle and shows what may happen as the back pressure of the flow is increased in an actual nozzle. At pressure 1 (P1) a small boundary is formed mirroring the physical boundary. At pressure 2 (P2) the fluid boundary grows toward the throat of the nozzle. At pressures above P3 the flow and this fluid boundary establish an equilibrium and the boundary restricts the flow down the throat. Any increase of pressure from P3 increases the pressure of the boundary and flow but both remain im equilibrium.

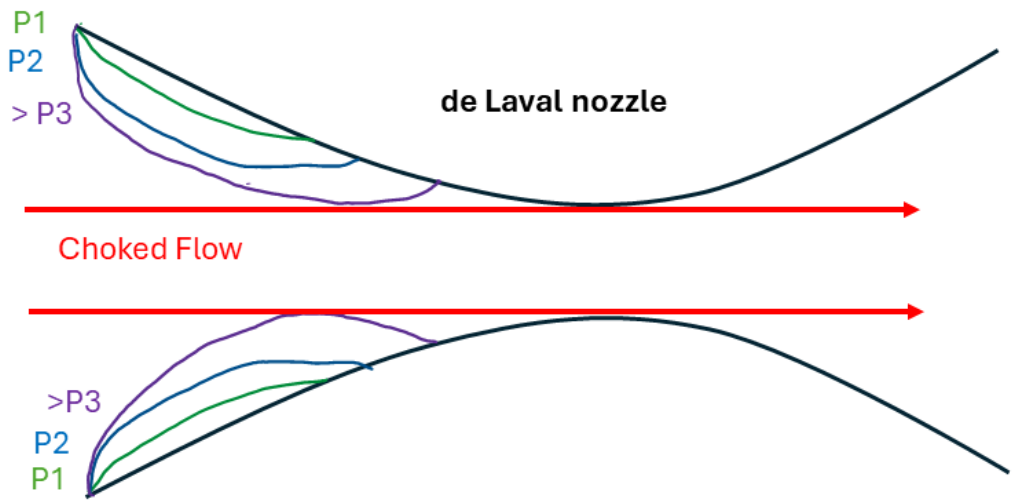


Fig. 26: *This illustration of a de Laval nozzle shows what may happen as the back pressure of the flow is increased in an actual nozzle. At pressure 1 (P1) a small boundary is formed mirroring the physical boundary. At pressure 2 (P2) the fluid boudary grows towayrds thw throat of the nozzle. At pressures above P3 the flow and boundary estaclidh an equilibriaum and the boudnary restricts the flow down the throat. Any increase of pressure from P3 increases the pressure of the bondary and flow but both remain im equilibrium.*

V. Conclusions

This very basic demonstration how powerful autonomous particle simulations can be. There is a great deal of mathematics than can be performed on vector fields both ans snapshots and changes in parameters. Simulation results can be evaluated for curl which can show how eddys form. The simulation can be rotated for slices in 3-d to evaluate specific phenomena such as diverging section shock. This also means that shocks are no longer infinitesimal entities that must be skipped over.

The project is multi-disciplinary and aspires to bring computer graphics concepts to thermo-fluid flow. Computer graphics experts have knowledge of a great deal of different types of math such as splines, barycentric coordinates, etc. They are also experts on the GPU which is essential for performance tuning and software design.

This approach is also the reverse of CFD approaches where the profile of the nozzle is determined before the study. This methods allows the researcher to start with a profile that can be modified in real time so to evaluate the effect on the flow. This may also allow the training of Artificial Intelligence so to produce novel forms.

References

[1] M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. J. Peters, Eds., *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing. Amsterdam: IOS Press, 2014, no. v. 25.

[2] J. Bell, "A SURVEY OF GPU-ACCELERATED PARTICLE COLLISION DETECTION."

[3] Y. J. Huang, O. J. Nydal, C. Ge, and B. Yao, "An Introduction to Discrete Element Method: A Meso-scale Mechanism Analysis of Granular Flow," *Journal of Dispersion Science and Technology*, vol. 36, no. 10, pp. 1370–1377, Oct. 2015.

[4] A. A. Munjiza, *Computational Mechanics of Discontinua*, ser. Wiley Series in Computational Mechanics. Chichester, West Sussex, U.K.: Wiley, 2012.

[5] E. Oñate and R. Owen, Eds., *Particle-Based Methods: Fundamentals and Applications*, ser. Computational Methods in Applied Sciences. Dordrecht: Springer Netherlands, 2011, vol. 25.

[6] J. Peters, SA. Sherif, and J. Bell, "Rounded cell collision detection on the GPU."

[7] G. Wang, F. Yang, K. Wu, Y. Ma, C. Peng, T. Liu, and L.-P. Wang, "Estimation of the Dissipation Rate of Turbulent Kinetic Energy: A Review," *Chemical Engineering Science*, vol. 229, p. 116133, Jan. 2021.