Tanque de Provas Numérico da USP

Eric Henrique Favero

Rubens A. Amaro Junior

Manual Prático BURNER

Prof. Dr. Cheng Liang Yee

São Paulo

2018

Pré-processador BURNER[1]

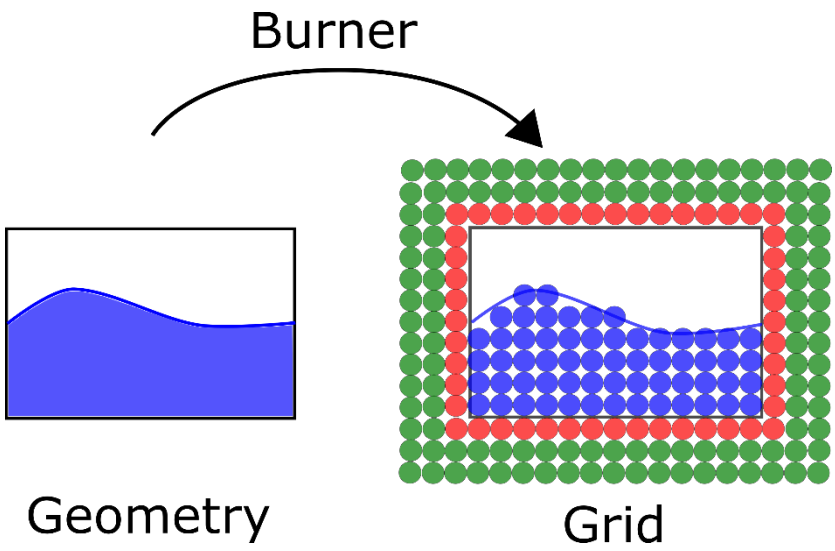
Sumário

Burner 3
main.cpp 6
ioside.cpp 9
Point.h 12
ioflow.cpp 14
Particle.h 16
define.h 17
Arquivo de entrada .txt 19
Arquivo GRID 21
Exemplos 22
Visualização no ParaView 26
Observações e Cuidados importantes 30

[1] Software desenvolvido por Eric Henrique Favero.
eric.favero@gmail.com
Versão 1.0

Burner

Pré-processador gerador das posições espaciais (x, y, z) definidas para diferentes materiais (ID's). Através de funções matemáticas no código computacional, representadas por comandos (*if, while, for,* ...), o programa gera o domínio computacional discreto (geometria do problema) utilizado pelo *solver* baseado no método de partículas *Moving Particle Semi-implicit* (MPS).



O **Burner** é um programa escrito em linguagem C++, composto por seis arquivos:

Cabeçalho (*Header*)	**Fonte (*Source*)**
define.h	main.cpp
Particle.h	ioside.cpp
Point.h	ioflow.cpp

Os arquivos podem ser editados e o programa compilado com diversos editores, como por exemplo:

CodeBlocks - http://www.codeblocks.org/

NetBeans - https://netbeans.org/

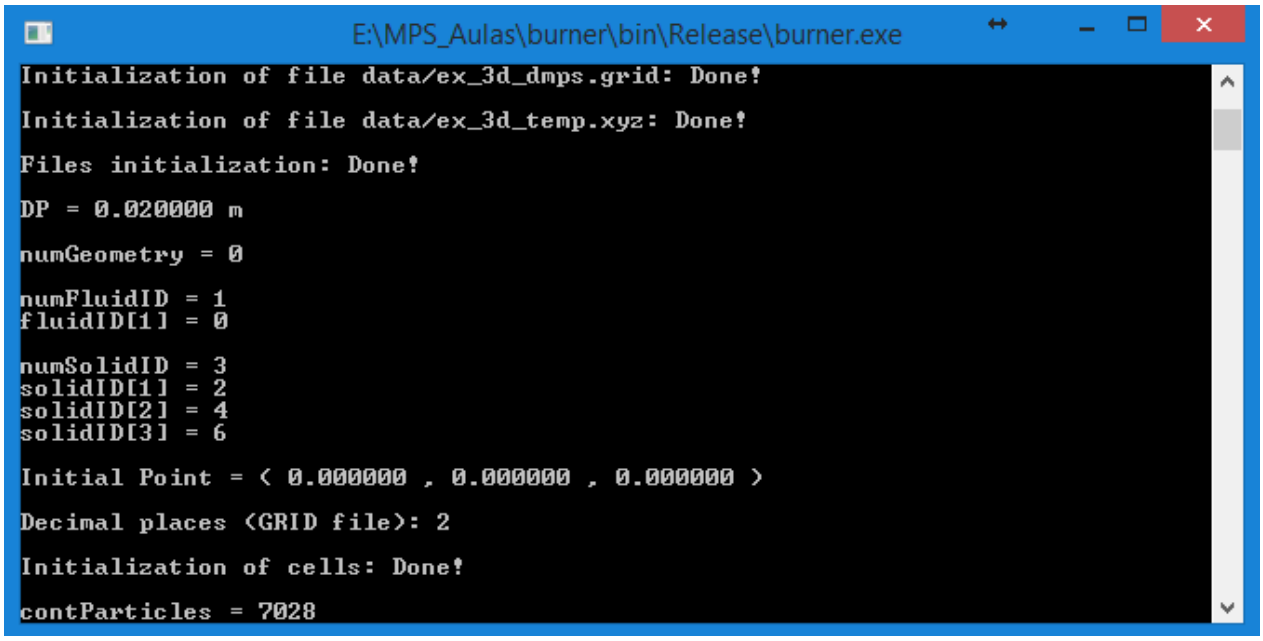
Dev-C++ - http://orwelldevcpp.blogspot.com.br/

Na pasta do **Burner**, já existe um **project** pré compilado para o **CodeBlocks**, o arquivo *burner.cbp*.

Após compilação do código, através da sua execução é aberto um **prompt** de comando onde deve ser passada a pasta e **nome do arquivo (txt) com dados de entrada** (geralmente **data/nome do arquivo sem .txt**):

Após ENTER, o programa apresenta os nomes dos arquivos de saída, dados do arquivo de entrada, e o **número de casas decimais** para impressão das posições geométricas do modelo. O programa começa a gerar o modelo, mostrando os arquivos de saída à medida que são finalizados.

Caso seja necessário gerar um caso com mais de 10 casas decimais de precisão das posições dos pontos (partículas), o usuário deve realizar modificações no arquivo **Particle.h**, conforme explicado no final desse manual.



Para visualização das geometrias geradas pelo **Burner** (arquivo .xyz), pode-se utilizar o *software*:

ParaView - http://www.paraview.org/

Basicamente, o **USUÁRIO** terá de alterar, os arquivos **ioside.cpp** e **ioflow.cpp** para gerar as geometrias.

Dependendo da geometria, os arquivos **define.h** e **Particle.h** também poderão ser alterados.

A seguir são apresentadas as principais funcionalidades dos arquivos que compõem o **Burner**.

main.cpp

O arquivo **main.cpp** contem a estrutura principal do código, ou seja, os principais comandos realizados durante a execução. Abaixo são apresentadas algumas das principais partes do arquivo. Esse arquivo, praticamente, não deve ser alterado.

Nas primeiras linhas do arquivo, são declaradas algumas variáveis do *int main*.

```
int main ()
{

    time_t start, end;

    //createFiles(paraview_xyz, mps_grid, dmps_grid, input, temp);

    FILE *paraview_xyz, *paraview_2D_xyz, *mps_grid, *mps_2D_grid, *dmps_grid, *input, *temp;
    int mode;
    long section;
    char d;
    char file_name[128];
    char file_name_temp[128];
```

Um arquivo de entrada (extensão txt), é acessado pelo programa. Através do comentário de algumas linhas (//), pode-se escolher entre:

- Programa sempre lê um arquivo nomeado “burner.txt”

```
printf ("Opening default input: burner.txt ");
//printf ("Enter the name of the case: ");
//scanf ("%s", file_name);
```

- Programa lê arquivo com o nome fornecido pelo usuário

```
//printf ("Opening default input: burner.txt ");
printf ("Enter the name of the case: ");
scanf ("%s", file_name);
```

O **nome do arquivo de entrada** é usado como nome base em todos os **arquivos de saída** gerados no programa.

Após a passagem do nome do arquivo, o programa verifica se o modelo é bi (2D) ou tridimensional (3D). Vale salientar que modelos 2D são gerados no plano XY, ou seja, coordenadas em Z são definidas como zero pelo Burner.

```
if (mode%2 == 0) {

    //fscanf (input, "\n%c", &d);
    d = 'z';
    printf ("The 2D cases are have to be sketched in XY\n", d);
    //fscanf (input, "%ld", &section);
    section = 0;
    //printf ("section = %ld\n\n", section);

    //printf ("2D case in section %ld of the axis %c.\n\n", section, d);

    strcpy (file_name_temp, file_name);
    strcat (file_name_temp, ".xyz");
    paraview_2D_xyz = fopen(file_name_temp, "w");
    printf("Inicialization of file %s: Done!\n", file_name_temp);

    strcpy (file_name_temp, file_name);
    strcat (file_name_temp, "_mps.grid");
    mps_2D_grid = fopen (file_name_temp, "w");
    printf("Inicialization of file %s: Done!\n", file_name_temp);
```

Criando arquivos nos seguintes formatos:

.xyz: arquivo com tipo de partícula (ID) e posições (x,y,z) para visualização no PARAVIEW.

.grid: arquivo com tipo de partícula (ID), posições (x,y,z), vdelocidades iniciais (vx, vy, vz), pressão inicial e PND inicial. Arquivo passado como entrada do *solver* MPS.

Um arquivo com sufixo _temp.xyz também é gerado. Caso ocorra algum problema ou o programa não tenha terminado sua execução, esse arquivo pode ser visualizado no PARAVIEW (http://www.paraview.org/), contendo a parte do modelo gerada até o momento.

Novas variáveis são definidas e recebem valores do arquivo de entrada:

```
double DP;
int numGeometry;
double geometry[NUMMAXGEOMETRY];
int numFluidID;
int fluidID[NUMMAXID];
int numSolidID;
int solidID[NUMMAXID];
double r[3];
Particle zeroPoint;

{
    fscanf (input, "%lf", &DP);
    printf ("\nDP = %lf m\n", DP);

    fscanf (input, "%d", &numGeometry);
    printf ("\nnumGeometry = %d\n", numGeometry);
    for (int n = 0; n < numGeometry; n++) {
```

O pograma chama a estrutura *Particle*, que possui as principais funções para construção do modelo, define a variável *countPartcles* (número de partículas) e realiza a escrita dos arquivos de saída:

```
Particle *particle = new Particle[NUMMAXPARTICLES];

//initializationParticles ();

long contParticles = zeroPoint.burnCells(temp, particle, DP, geometry, fluidID, numFluidID, solidID, numSolidID);

Particle print;
if (mode%2 == 0) {
    print.print_Paraview_2D(paraview_2D_xyz, contParticles, particle, DP, geometry, numGeometry, section, d);
    print.print_MPS_2D(mps_2D_grid, contParticles, particle, DP, section, d);
    if (DMPS == 1) print.print_dMPS(dmps_grid, contParticles, particle, DP);
}

if (mode%3 == 0) {
    print.print_Paraview(paraview_xyz, contParticles, particle, DP, geometry, numGeometry);
    print.print_MPS(mps_grid, contParticles, particle, DP);
    if (DMPS == 1) print.print_dMPS(dmps_grid, contParticles, particle, DP);
}

if (ALT == 1)
{   //condition, random position, random velocity
    strcat (file_name, "_alt.grid");
    FILE *alt = fopen(file_name, "w");
    printf("\nInicialization of file %s: Done!\n", file_name);
    print.print_alt(alt, contParticles, particle, DP);
}
```

ioside.cpp

Arquivo onde o **USUÁRIO** define **dimensões** e **equações** que delimitam as geometrias.

Primeiro são definidos pontos e variáveis. (NÃO ALTERAR ESSA PARTE !!!)

```
int Particle::ioside (double DP, double geometry[])
{
    Point P (r, DP);

    double x = P.x;
    double y = P.y;
    double z = P.z;

    Point O;

    Point X (1,0,0);
    Point Y (0,1,0);
    Point Z (0,0,1);
```

DP representa a distância entre pontos, ou **distância entre partículas**, lido pelo **Burner** do **arquivo de entrada**.

Variáveis que serão utilizadas para gerar as geometrias são **definidas** pelo **usuário**.

```
///Variaveis (double, Point, etc..)

double var1 = 0.6;
double var2 = 0.3;
```

Regiões representando as principais geometrias são **definidas** pelo **usuário**.

```
///Regiões (Region)

Region paredes  = P.parallelepiped (var1, var2, var3);
Region fluido   = P.parallelepiped (var4, var5, var6);
Regio solido    = P.cylinder (raio, altura);
```

O arquivo **Point.h** contém algumas funções ***geometrias*** e ***transformações*** (translação e rotação) prontas para serem utilizadas. Outras geometrias e transformações podem ser criadas pelo usuário e colocadas no arquivo **Point.h** ou então, colocadas diretamente no arquivo ioside.cpp, com utilização de *if*, *else*, *for,* ....

Operações para construção das geometrias são **definidas** pelo **usuário**.

```
///Operações
if (paredes)
{
    if (fluido)
        return 0;

    return -1;
}
```

Cada ***return ID*** se refere a um tipo de material diferente (fluido ou sólido), conforme passado pelo arquivo de entrada. Para definirmos uma **região vazia**, colocamos ***return* -1**.

ATENÇÃO, TODAS AS GEOMETRIAS SÃO GERADAS SEGUINDO A PRIORIDADE EM QUE FOI ESCRITA NO ARQUIVO. Por exemplo, para gerar um sólido com furo, dentro de um fluido, as chamadas devem ser definidas na seguinte ordem:

```
///Operações

if (paredes)
{
    if (furo)
        return -1;

    if (solido)
        return 4;

    if (fluido)
        return 0;

    return -1;
}
```

Caso o usuário deixe as chamadas em ordem diferente, como por exemplo:

```
///Operações

if (paredes)
{
    if (solido)
        return 4;

    if (furo)
        return -1;

    if (fluido)
        return 0;

    return -1;
}
```

O sólido estará dentro do fluido, porém o furo no sólido não será feito, pois o programa já terá gerado todo o sólido, não permitindo a criação do furo (espaço vazio).

Os sólidos terão a parede externa com ID = número passado no arquivo de entrada e pelo *return*, e 2 camadas de paredes internas (*dummies*) com ID = número passado no arquivo de entrada + 1.

Caso seja necessária a geração de um sólido com paredes internas e externas em contato com fluido, e portanto, mínimo de 4 camadas intermediárias de *dummies* nos casos 2D e 2 camadas intermediárias de *dummies* nos casos 3D, operações de união ou subtração podem ser utilizadas. O **Burner** não gera esse tipo de parede automaticamente.

O **BURNER** REALIZA A CONSTRUÇÃO DE SÓLIDOS SEMPRE CONSIDERANDO O ESPAÇO 3D, MESMO PARA GEOMETRIAS 2D. Dessa forma, podemos ter certas dificuldades para criar sólidos internos de geometrias 2D. Uma das formas de contornar isso é:

- O usuário cria regiões de fluido, ou seja, completamente preenchida com partículas, e realiza operações de subtração para gerar as paredes *dummies* e espaços vazios no sólido interno.
- O usuário cria o sólido interno como uma geometria 3D, mas desloca o sólido no eixo Z, ou então estende as dimensões do sólido para valores de Z negativo e positivo. A geometria resultante será a seção do sólido que passa pelo plano XY.

Após as operações, deixar ***return 2*** para construção de paredes que delimitam todo o domínio físico. Automaticamente, o **Burner** irá gerar a parede interna com ID = 2 e parede externa com ID = 3 (*dummies*).

```
///Return padrão (constrói a parede externa)
return 2;
```

Point.h

Principais geometrias e transformações pré-definidas no programa. O usuário pode criar uma geometria nova e adicionar no arquivo.

Exemplos:

```
Point transformation (Point O, Point Vector = Point (0,0,1), double rotation = 0)
{
    Point P;

    double senTeta, cosTeta, senFi, cosFi;
    double projXY, norma;

    norma = sqrt(pow(Vector.x, 2) + pow(Vector.y, 2) + pow(Vector.z, 2));
    if (fabs(norma) <= EPS) {
        printf ("Error: ||Vector|| = 0!\n");
    }
    projXY = sqrt(pow(Vector.x, 2) + pow(Vector.y, 2));
    if (!(fabs(projXY) <= EPS)) {

        senTeta = (Vector.y)/projXY;
        cosTeta = (Vector.x)/projXY;

        senFi = (Vector.z)/norma;
        cosFi = projXY/norma;
```

Operações de translação e rotação sobre uma geometria.

Point O – Translação.

Point Vector – Vetor onde será realizada a rotação.

double rotation – ângulo de rotação em graus.

```
Region parallelepiped (double X, double Y, double Z)
{
    if (x > 0 && x < X)
        if (y > 0 && y < Y)
            if (z > 0 && z < Z)
                return true;

    return false;
}
```

Paralelepípedo com dimensões X, Y e Z.

```
Region cylinder (double R, double H)
{
    if (z > 0 && z < H)
        if (x*x + y*y < R*R)
            return true;

    return false;
}
```

Cilindro de raio R e comprimento H.

ioflow.cpp

Eventuais cortes e transformações sobre as geometrias e/ou paredes externas.

Primeiro são definidos pontos e variáveis. (NÃO ALTERAR ESSA PARTE !!!)

```
int Particle::ioflow (double DP, double geometry[])
{
    Point P (r, DP);

    double x = P.x;
    double y = P.y;
    double z = P.z;

    Point O;
    Point X (1,0,0);
    Point Y (0,1,0);
    Point Z (0,0,1);
```

Variáveis que serão utilizadas para cortes nas geometrias são definidas pelo usuário.

```
  ///Variaveis (double, Point, etc..)

  double var1 = 0.6;
  double var2 = 0.3;
```

Os cortes podem ser definidos por regiões.

```
  ///Regiões (Region)

  Region paredes  = P.parallelepiped (var1, var2, var3);
  Region fluido   = P.parallelepiped (var4, var5, var6);
  Regio solido    = P.cylinder (raio, altura);
```

E colocados nas operações:

```
  ///Operações
  if( id == 2 || id == 3 ) // Somente paredes externas
  {
      if (paredes) return -1;
  }
```

ou então, colocados diretamente nas operações com utilização de *if*, *else*, ...:

```
///Operações
if( id == 2 || id == 3 ) // Somente paredes externas
{
    if (y > 0.5 || x > 1.5) return -1;
}
```

Além de paredes externas, como apresentadas nos exemplos acima, outros materiais podem ser removidos.

Após as operações, deixar ***return id***.

```
///Return padrão (caso nenhuma operação seja feita)

return id;
```

Particle.h

Além das funções detalhadas anteriormente como ***ioside*** e ***ioflow***, dentro da estrutura *struct Particle* existem outras funções:

- Responsáveis pelo **sistema de busca** e **construção do modelo**.

```
void getMatrix

long getParticles

int cellNeighbor

int comparesCells

long burnCells
```

- Responsáveis pela **escrita** nos **arquivos de saída**.

```
void print_Paraview

void print_Paraview_2D

void print_MPS

void print_MPS_2D

void print_dMPS

void print_alt
```

define.h

Parâmetros básicos como dp, ids, ponto zero.

```
typedef bool Region;

#define NUMMAXID 30
#define NUMMAXGEOMETRY 30
#define NUMMAXPARTICLES 20000000
#define NUMMAXCELLS 150000

#define CELLDIAMETER 25
#define CELLRADIUS 12
#define CELLBORDER 3

#define SIMPLECUBE 1

#define POSITIONX 1
#define POSITIONY 1
#define POSITIONZ 1

#define EPS 0.00000001

#define PI 3.14159265

#define DMPS 0
#define ALT 0

#endif
```

Variável booleana Region.

NUMMAXID - Número máximo de materiais (fluidos ou sólidos diferentes).

NUMXAGEOMETRY - Número máximo de geometrias (Ainda não implementado).

NUMMAXPARTICLES - Número máximo de pontos que o programa pode gerar.

NUMMAXCELLS - Número máximo de células utilizado pelo programa para geração do modelo.

CELLDIAMETER –

CELLRADIUS –

CELLBORDER –

SIMPLECUBE –

POSITIONX - Limite mínimo do domínio no eixo X (1 -> Partícula ao lado da parede é deslocada 1*DP/2)

POSITIONY - Limite mínimo do domínio no eixo Y (1 -> Partícula ao lado da parede é deslocada 1*DP/2)

POSITIONZ - Limite mínimo do domínio no eixo Z (1 -> Partícula ao lado da parede é deslocada 1*DP/2)

EPS - Precisão das variáveis.

PI – Valor de PI.

DMPS - Gera (1) ou não (0) arquivo de saída para o DMPS (versão paralelizada do MPS, geralmente utilizada para casos com uma quantidade superior a 10 milhões de partículas). Somente para casos 3d.

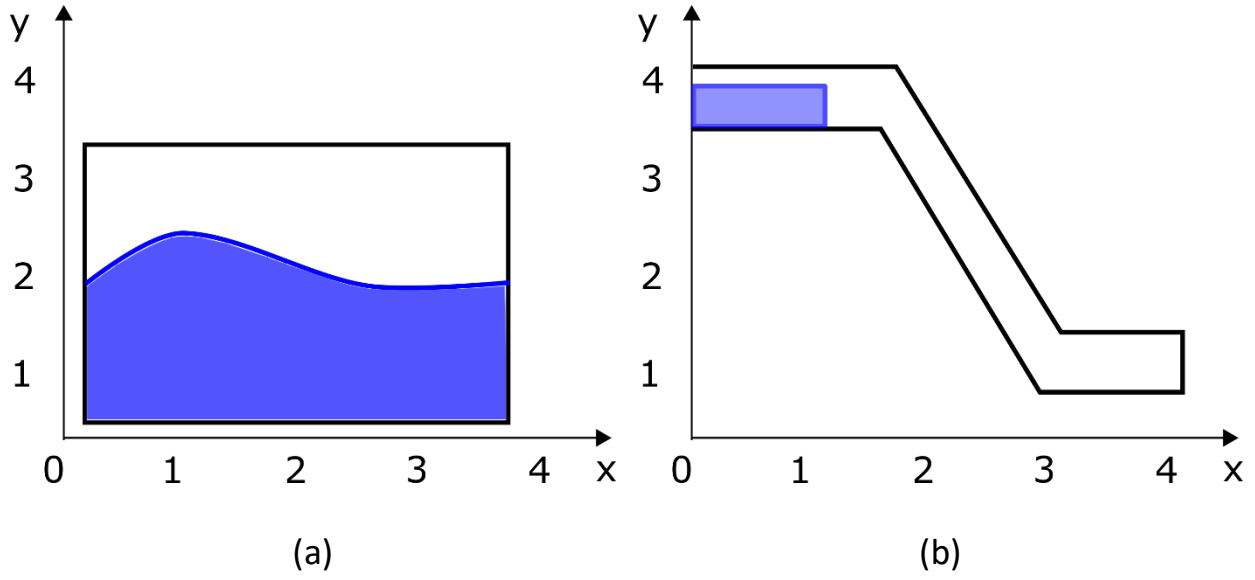
ALT - Gera (1) ou não (0) arquivo .grid para inclusão de condições no xml que serão utilizados no solver MPS. Útil para automatizar a inclusão de condições de um número grande de partículas, como por exemplo, definir os ID de vários sensores.

Arquivo de entrada .txt

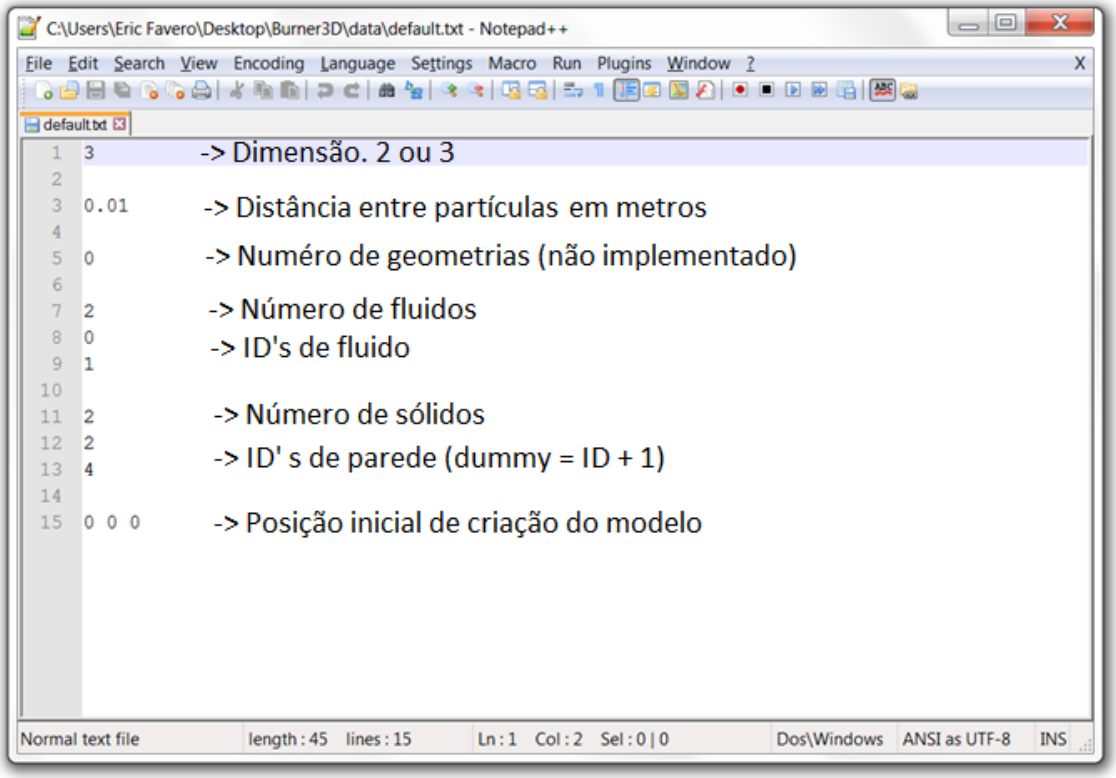
O arquivo de entrada ***nome_do_arquivo.txt***, possui os principais dados de entrada passados para o **Burner**.

O NOME DO ARQUIVO **NÃO** PODE CONTER ESPAÇO, como por exemplo "*meu arquivo.txt*". Ao invés disso, podemos nomear o arquivo utilizando *underscore* "meu_arquivo.txt".

ATENÇÃO ESPECIAL PARA A POSIÇÃO INICIAL DE CRIAÇÃO DA GEOMETRIA!!! O USUÁRIO DEVE PASSAR A COORDENADA DE UM PONTO PRÓXIMO A UMA REGIÃO ONDE EXISTA A GEOMETRIA (FLUIDO OU SÓLIDO) QUE SERÁ GERADO. CASO NÃO EXISTA UM PONTO DO GEOMETRIA PRÓXIMO DESSA POSIÇÃO, O BURNER NÃO GERA O MODELO. Como exemplo, podemos tomar as geometrias abaixo como referência. O tanque (a) possui parede e fluido próximos do ponto (0,0). Neste caso, podemos deixar a última linha do arquivo de entrada txt com valores 0 0 0 e o **Burner** irá gerar a geometria. Já no caso da tubulação (b), não existe parede ou fluido próximo do ponto (0,0), e caso deixemos o ponto 0 0 0 no arquivo de entrada txt, o **Burner** pode não conseguir gerar a geometria. Neste caso, o usuário deve fornecer outro ponto no arquivo de entrada. Por exemplo, poderiam ser fornecidos um dos seguintes pontos na última linha do arquivo txt: 0 4 0, 2 3 0, 3 1 0 ou 4 1 0.



A figura abaixo apresenta o que cada linha do arquivo de entrada irá passar para o **Burner**.


C:\Users\Eric Favero\Desktop\Burner3D\data\default.txt - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
default.txt
3 -> Dimensão. 2 ou 3
0.01 -> Distância entre partículas em metros
0 -> Numéro de geometrias (não implementado)
2 -> Número de fluidos
0
1 -> ID's de fluido
2 -> Número de sólidos
2
4 -> ID' s de parede (dummy = ID + 1)
0 0 0 -> Posição inicial de criação do modelo
Normal text file length : 45 lines : 15 Ln : 1 Col : 2 Sel : 0 | 0 Dos\Windows ANSI as UTF-8 INS

Arquivo GRID

Arquivo de texto gerado pelo **Burner** e que servirá como entrada do solver MPS.

1ª linha: Número zero (default)

2ª linha: Número de pontos (partículas)

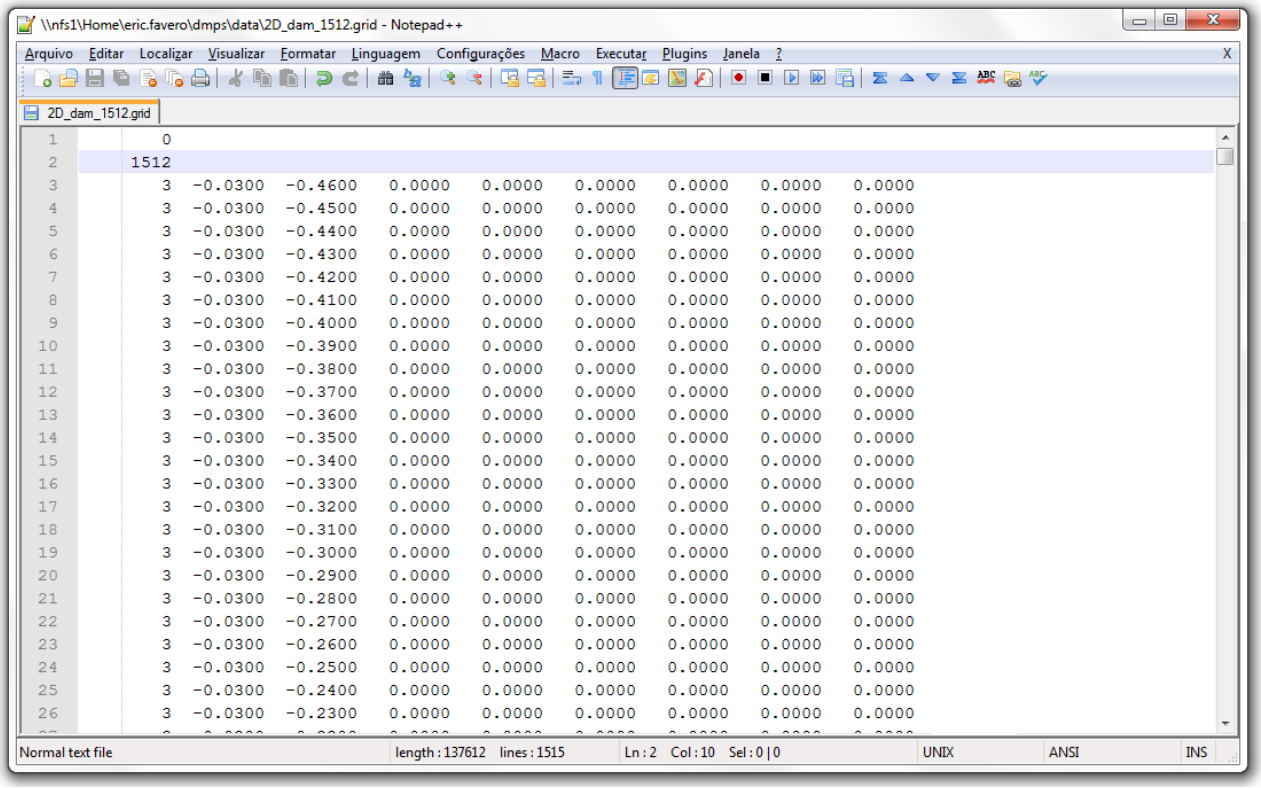
1ª coluna: tipo de material (ID)

2ª, 3ª, 4ª colunas: posições (x,y,z)

5ª, 6ª, 7ª colunas: velocidades iniciais (vx,vy,vz)

8ª coluna: pressão inicial (atualmente é desconsiderado pelo MPS)

9ª coluna: PND inicial (atualmente é desconsiderado pelo MPS)



Exemplos

Neste exemplo será gerada uma esfera de raio 2 m ($x^2 + y^2 + z^2 <= 2^2$) completamente preenchida com fluído de ID = 0.

```
#ifndef H_IOSIDE
#define H_IOSIDE

#include "Point.h"
#include "Particle.h"

int Particle::ioside (double DP, double geometry[])
{
    Point P (r, DP);

    double x = P.x;
    double y = P.y;
    double z = P.z;

    Point O;

    Point X (1,0,0);
    Point Y (0,1,0);
    Point Z (0,0,1);

    ///Variaveis (double, Point, etc..)
    double raio = 0.2;

    ///Operações
    if (x*x + y*y + z*z < raio*raio)
        return 0;

    ///Return padrão (constrói a parede externa)
    return 2;
}
    #endif
```

O mesmo exemplo poderia ser feito com uma função de geometria *sphere*

```
#include "Point.h"
#include "Particle.h"

int Particle::ioside (double DP, double geometry[])
{
    Point P (r, DP);

    double x = P.x;
    double y = P.y;
    double z = P.z;

    Point O;

    Point X (1,0,0);
    Point Y (0,1,0);
    Point Z (0,0,1);

    ///Variaveis (double, Point, etc..)
    double raio = 0.2;

    ///Regiões (Region)
    Region esfera = P.sphere(raio);

    ///Operações
    if (esfera)
        return 0;

    ///Return padrão (constrói a parede externa)
    return 2;
}
    #endif
```

Neste exemplo será gerado um toróide vazio (ID = -1).

```
#include "Particle.h"

int Particle::ioside (double DP, double geometry[])
{
    Point P (r, DP);

    double x = P.x;
    double y = P.y;
    double z = P.z;

    Point O;

    Point X (1,0,0);
    Point Y (0,1,0);
    Point Z (0,0,1);

    ///Variaveis (double, Point, etc..)
    double Rext = 0.3;
    double Rint = 0.1;

    ///Regiões (Region)
    Region toroide = P.toroid(Rext,Rint);

    ///Operacões
    if (toroide)
        return -1;

    ///Return padrão (constrói a parede externa)
    return 2;
}
    #endif
```

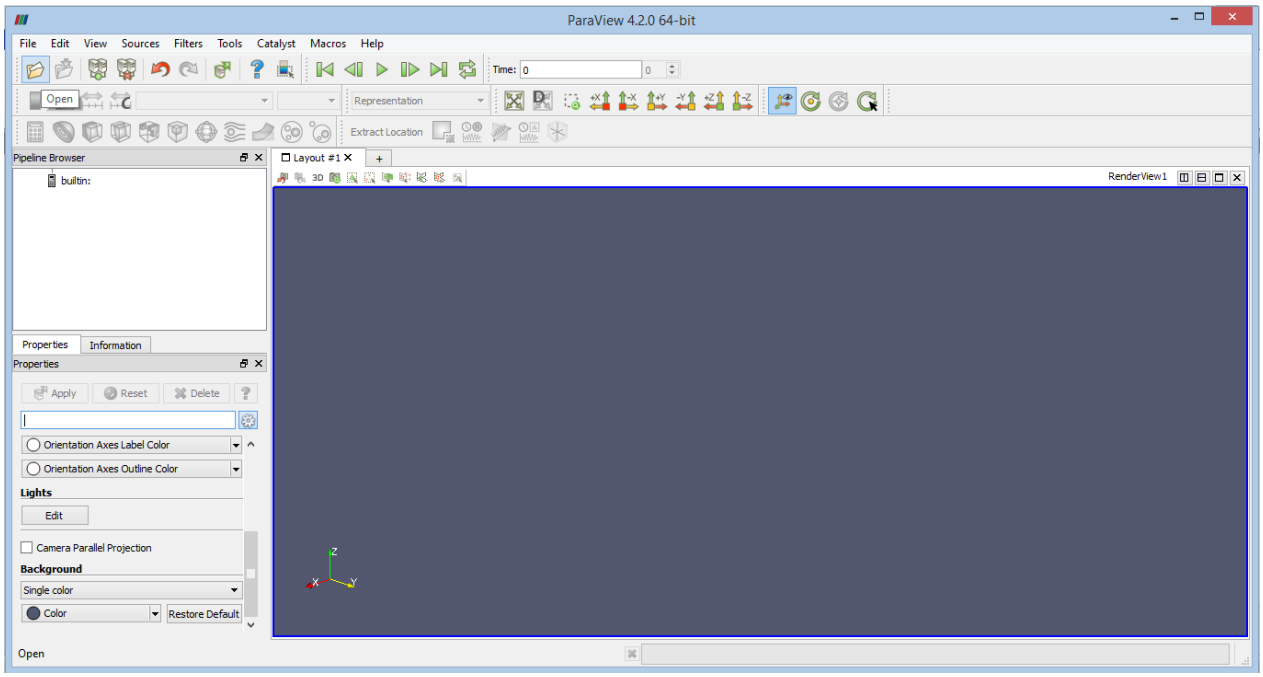
Neste exemplo será gerado uma caixa retangular deslocada da origem (0,0,0), uma distância P1 = (0.1, 0.3, 0) e rotacionada - 45^0, ou seja, base paralela ao vetor P2 (1, -1, 0). Ao invés do vetor P2, poderia ser informado o ângulo de - 45^0. A caixa será preenchida com uma coluna de fluido (ID = 0), de dimensões (0.6, 0.3, 0.2).

```
int Particle::ioside (double DP, double geometry[])
{
    Point P (r, DP);

    double x = P.x;
    double y = P.y;
    double z = P.z;

    Point O;

    Point X (1,0,0);
    Point Y (0,1,0);
    Point Z (0,0,1);

    ///Variaveis (double, Point, etc..)
    double Wx = 0.6;
    double Wy = 0.3;
    double Wz = 0.2;

    double Lx = 1.6;
    double Ly = 0.6;
    double Lz = 0.2;

    Point P1 (0.1, 0.3, 0);
    Point P2 (1, -1, 0);

    ///Regiões (Region)
    Region paredes = P.transformation(P1,P2).parallelepiped (Lx, Ly, Lz);
    Region fluido  = P.parallelepiped (Wx, Wy, Wz);

    ///Operações
    if (paredes)
    {
        if (fluido)
            return 0;

        return -1;
    }

    ///Return padrão (constrói a parede externa)
    return 2;

}

#endif
```

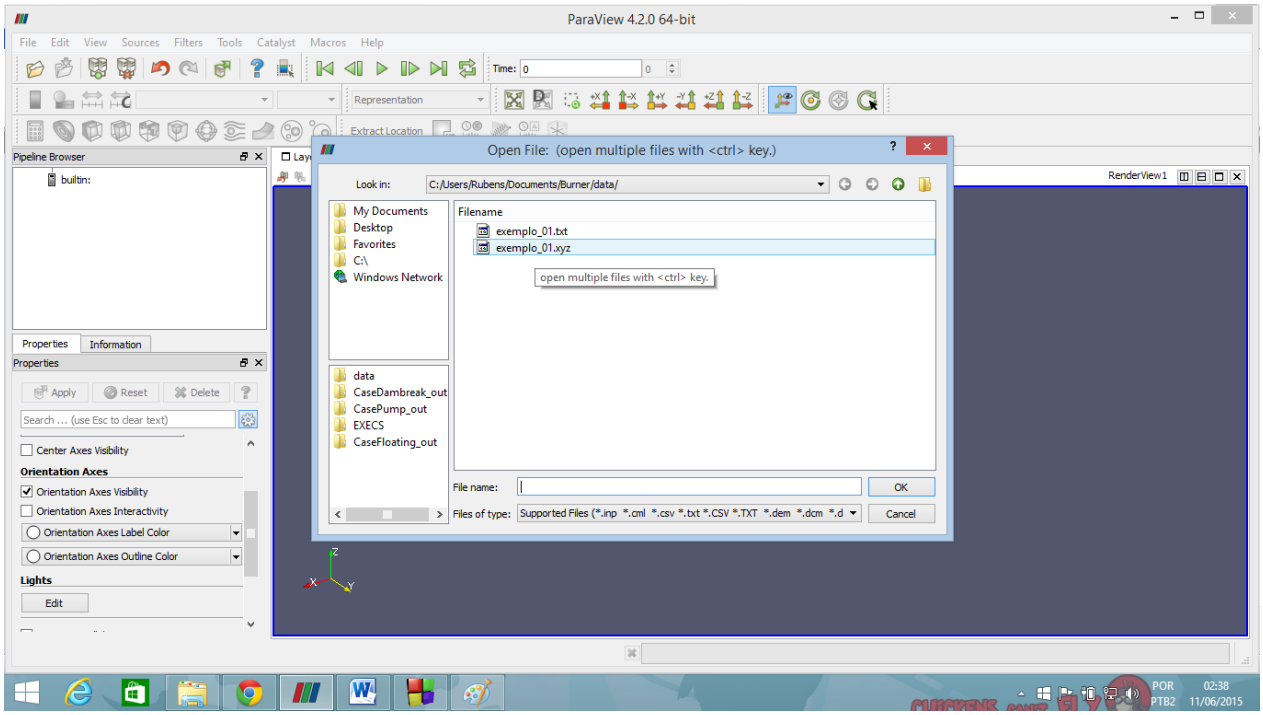
Visualização no ParaView

Para visualizar um modelo no Paraview (versão 4.2.0. Para versões atuais, consulte o tutorial do Paraview), siga os seguintes passos:

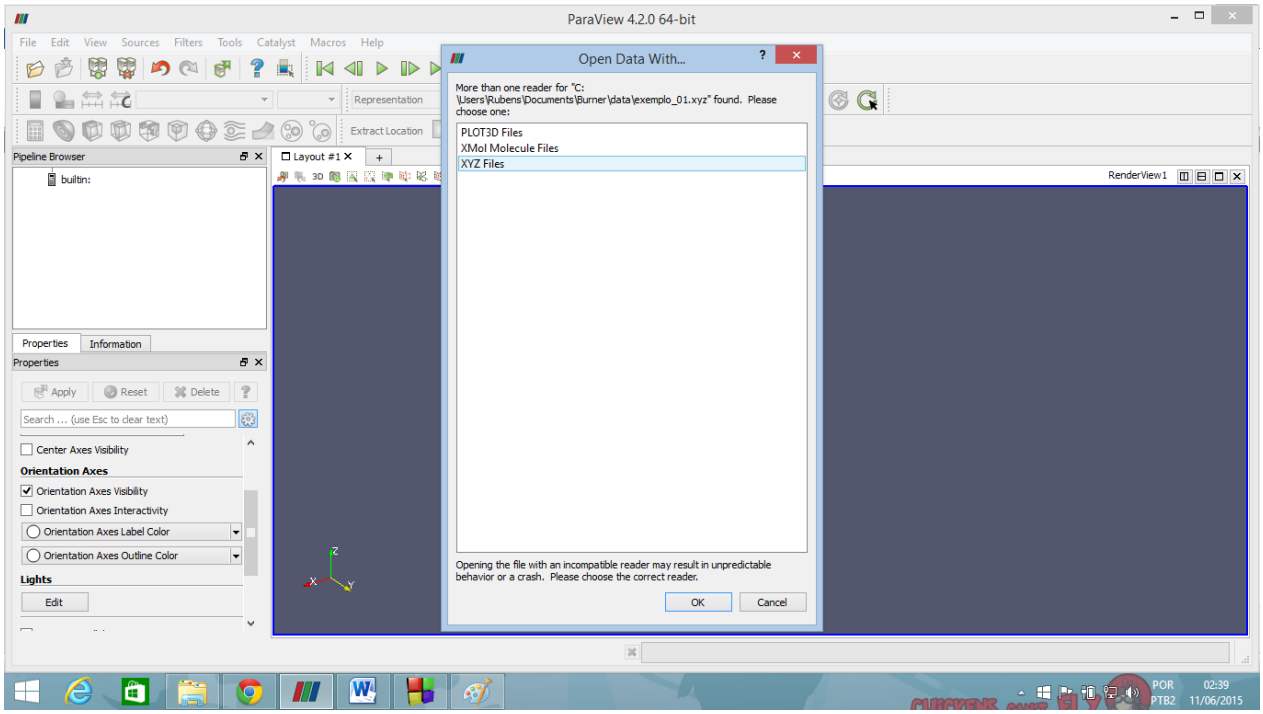
1. Selecione a pasta *Open* no canto superior esquerdo.



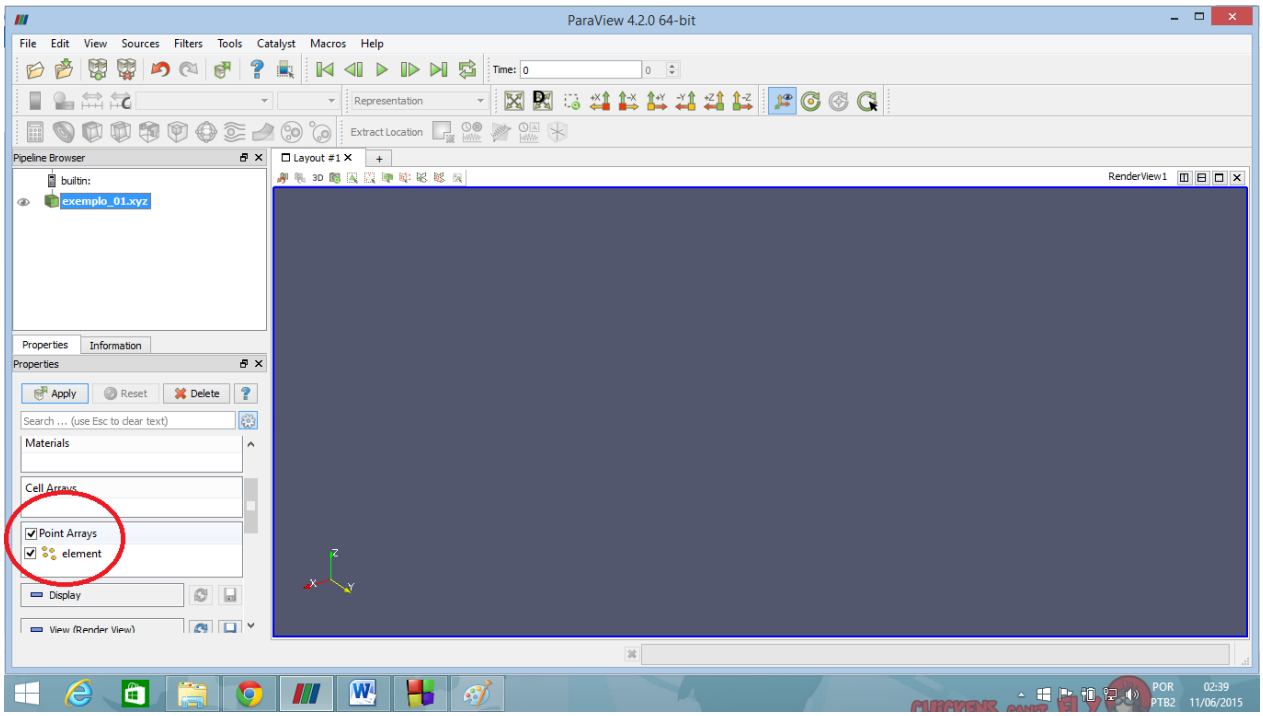
2. Selecione o arquivo .xyz.



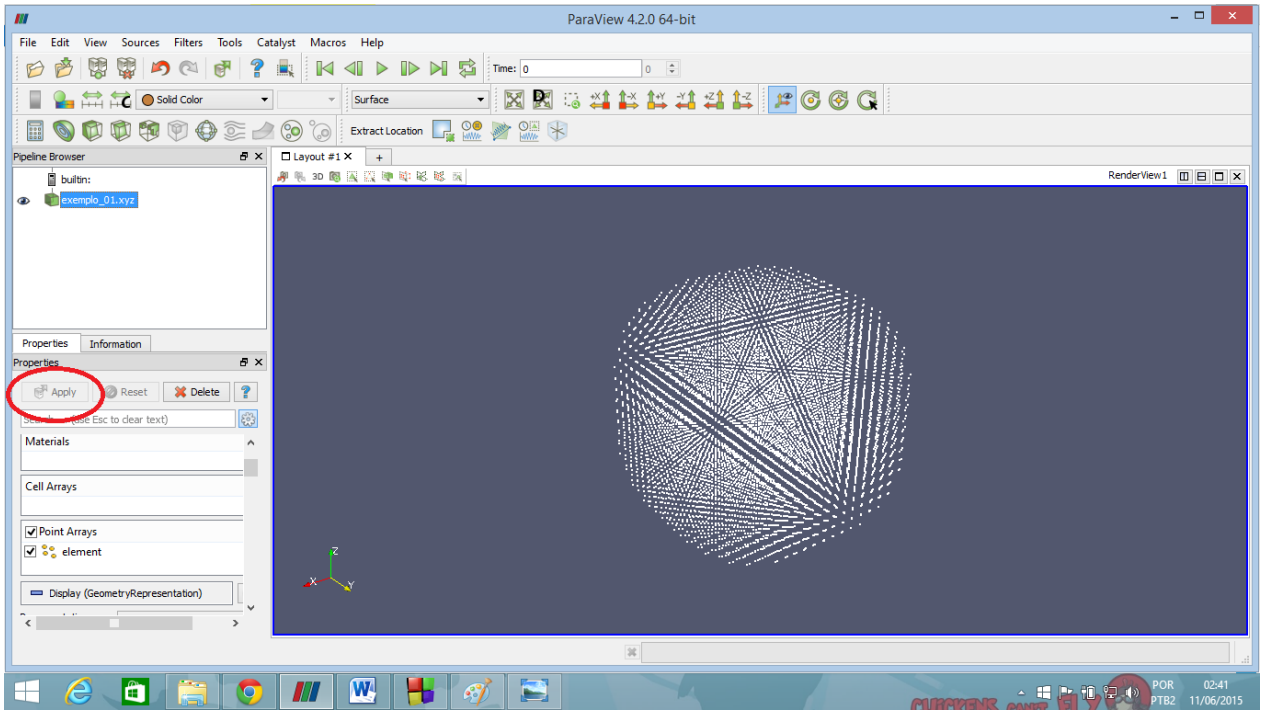
3. Selecione o formato XYZ *Files*.



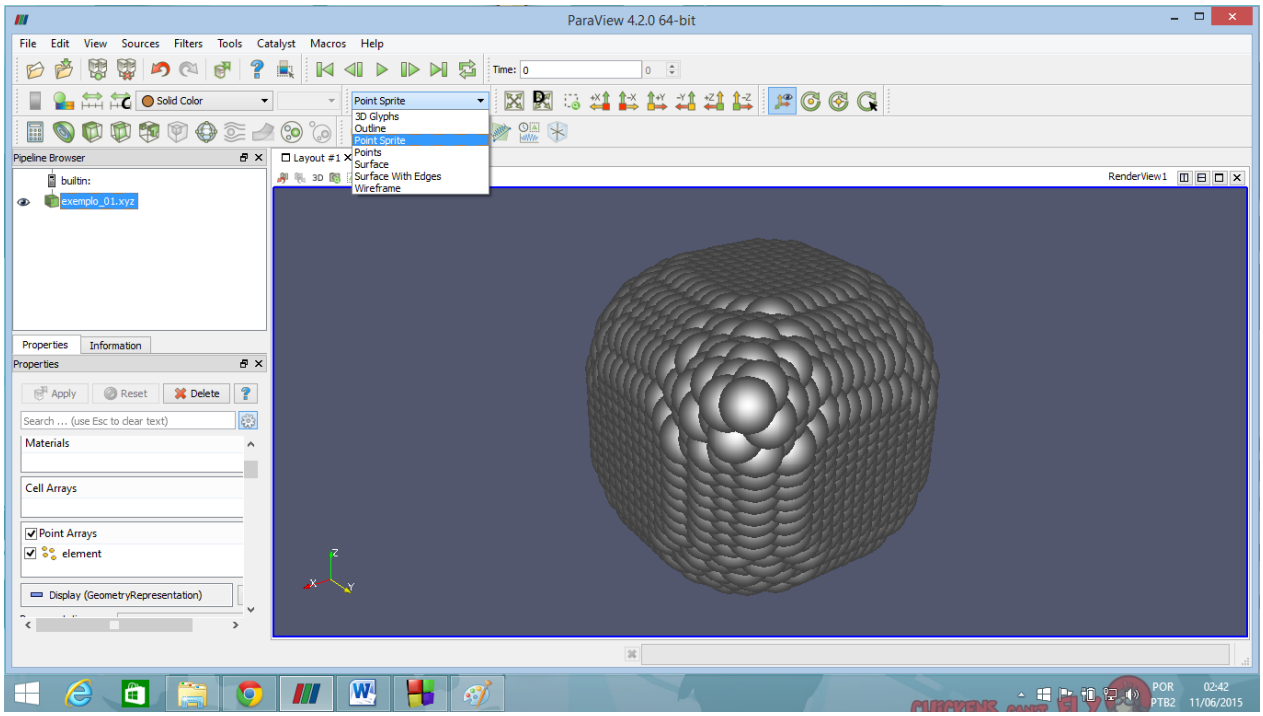
4. Habilite o campo *element*, para visualização de materiais em cores diferentes.



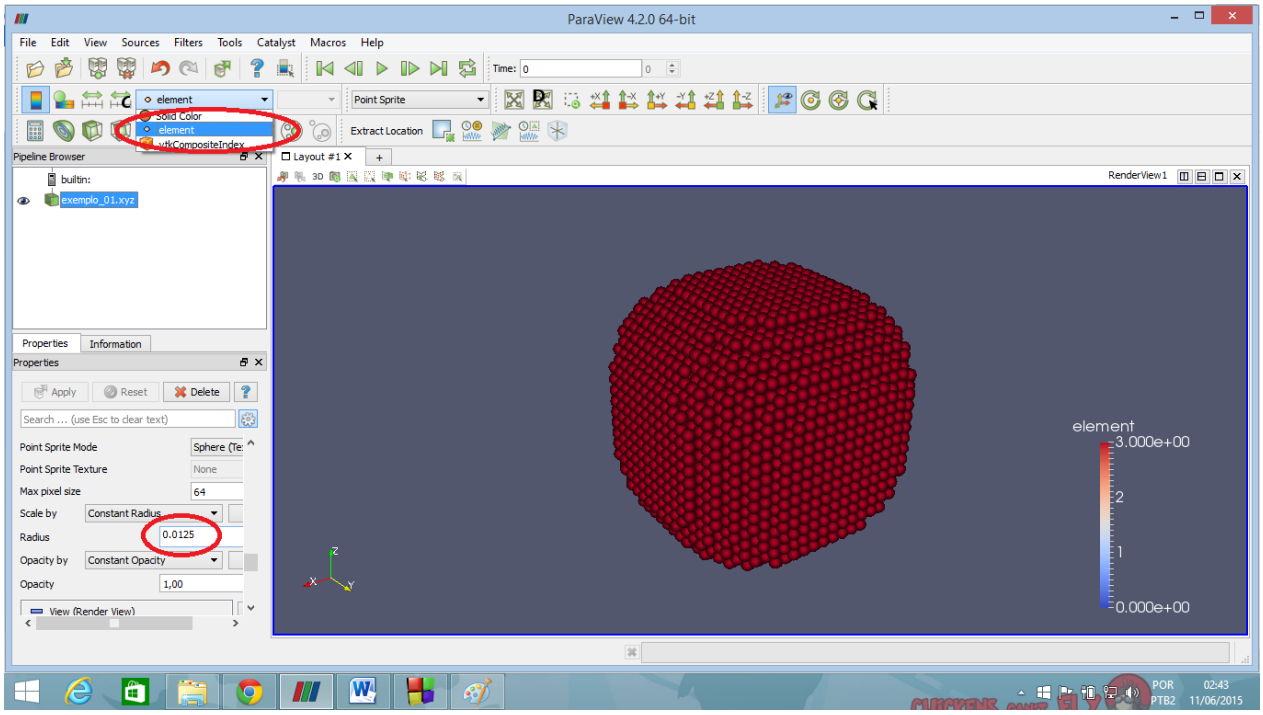
5. Click em *Apply*.



6. Para uma visualização melhor, podemos selecionar o modo *Point Sprite.*



7. Arrumar o raio das esferas, no campo *radius*, em função da distância entre partículas do modelo e selecionar o campo *element*.



Caso o modo ***Point Sprite*** não esteja habilitado, faça os seguintes passos:

Tools -> Manage Plugins ... -> PointSprite_Plugin

Clique no botão ***Load Selected*** e marque o campo ***Auto Load.***

Reinicie o ParaView, abra um modelo .xyz e verifique se a opção Point Sprite está habilitada.

Observações e Cuidados importantes

Posição inicial no arquivo txt

A POSIÇÃO INDICADA NA ÚLTIMA LINHA DO ARQUIVO DE ENTRADA TXT deve indicar um PONTO PRÓXIMO de uma região com FLUIDO ou SÓLIDO do modelo que será gerado. Caso não exista um ponto que represente fluido ou solido próximo dessa posição, o BURNER NÃO GERA O MODELO.

Sólido interno nos modelos 2d

O **BURNER** REALIZA A CONSTRUÇÃO DE SÓLIDOS SEMPRE CONSIDERANDO O ESPAÇO 3D, MESMO PARA GEOMETRIAS 2D. Dessa forma, podemos ter certas dificuldades para criar sólidos internos de geometrias 2D. Uma das formas de contornar isso é:

- O usuário cria regiões de fluido, ou seja, completamente preenchida com partículas, e realiza operações de subtração para gerar as paredes *dummies* e espaços vazios no sólido interno.
- O usuário cria o sólido interno como uma geometria 3D, aumentando sua dimensão no eixo Z, ao menos cerca de 8*DP, mas desloca o sólido no eixo Z, ou então estende as dimensões do sólido para valores de Z negativo e positivo. Também é necessário transladar e aumentar a dimensão no eixo Z da geometria externa. A geometria resultante será a seção do sólido que passa pelo plano XY.

Precisão das casas decimais

O **Burner** irá gerar as geometrias com um número de casas decimais em função da metade da distância entre partículas, ou seja, caso seja passado uma distância entre partículas de 0.005 m, o programa irá trabalhar com distâncias da ordem de 0.0025 m, 4 casas decimais.

O **Burner** está programado para trabalhar com valores entre 2 e 10 casas decimais. Se necessário, o usuário pode alterar esses valores nas funções de escrita dos arquivos de saída, localizadas no arquivo **Particle.h**. A precisão indicada nessas funções deve estar de acordo com a precisão que o **Burner** estará trabalhando.

Por exemplo, para uma distância entre partículas de 0.00005 m, (precisão de 0.000025 m, **6 casas decimais**) as funções devem estar no seguinte formato:

```
else
   fprintf (paraview_xyz, "\n%d %9.6lf %9.6lf %9.6lf", particle[i].id, X.x, X.y, X.z);
```

Ou seja, o valor após o ponto define a precisão dos números que serão escritos nos arquivos de saída. No exemplo, **%9.6lf**, indica seis casas decimais. A medida que utilizamos modelos com maior resolução, menores distâncias entre partículas, precisamos aumentar esses valores no **Burner**.

OBS: Não convém deixar um valor maior do que o necessário, pois isso acaba gerando arquivos com tamanhos cada vez maiores.