Travaux dirigés de Simulation n°4

Cours de simulation et animation

—IMAC troisième année—

Finitions et projet

Objectifs

- Finitions de la simulation générique,
- Préparation au projet.

Programme

- Améliorations et corrections,
- Adaptations pour la cohabitation du fluide, des solides et des espaces vides,
- Pistes pour le perfectionnement du simulateur (fumée, 3D...),
- Outils pratiques et consignes pour le projet,
- Brainstorming pour la création de formes.

▸ Exercice 1. Améliorations et corrections

Quelques modifications doivent être apportées au programme.

Une liste non exhaustive d'erreurs à corriger est également fournie.

A - Capitalisation de code

a) A la fin des méthodes `advectVelocities()` et `applyForces()`, on dispose d'un tableau temporaire de nouvelles vitesses, calculées aux centres des cellules. Ce tableau sert à mettre à jour `velocitiesX` et `velocitiesY` (soit les vitesses finales, sur les bords des cellules). Une fois ceci fait, afin que l'affichage des vitesses soit juste, il faut faire la mise à jour des VBOs de `objectVelocitiesBorders` et `objectVelocitiesCenters`.

L'ensemble de ces trois étapes est long en lignes de code et gagne à être capitalisé en la fonction suivante.

```
void Simulation::updateVelocitiesFromCenters(GLfloat * centeredVel)
{
    // Resets velocities components to 0
    for (GLuint iBordersX=0 ; iBordersX<nbBordersX ; iBordersX++)
        velocitiesX[iBordersX]=0.0;
    for (GLuint iBordersY=0 ; iBordersY<nbBordersY ; iBordersY++)
        velocitiesY[iBordersY]=0.0;

    // Uses the new centers velocity to update the separated velocity components
    for (GLuint iY=0 ; iY<nbSamplesY ; iY++)
    {
        for (GLuint iX=0 ; iX<nbSamplesX ; iX++)
        {
            GLuint iSamples=iY*nbSamplesX+iX;
            GLuint indexVelocitiesXLeft =iY *(nbSamplesX+1)+ iX;
            GLuint indexVelocitiesXRight =iY *(nbSamplesX+1)+(iX+1);
            GLuint indexVelocitiesYBottom=iY * nbSamplesX + iX;
            GLuint indexVelocitiesYTop =(iY+1) * nbSamplesX + iX;

            GLfloat coefLeft=0.5;
            GLfloat coefRight=0.5;
```

```
            GLfloat coefBottom=0.5;
            GLfloat coefTop=0.5;
            if (iX==0) coefLeft =1.0;
            if (iX==(nbSamplesX-1)) coefRight =1.0;
            if (iY==0) coefBottom=1.0;
            if (iY==(nbSamplesY-1)) coefTop =1.0;

            velocitiesX[indexVelocitiesXLeft] +=coefLeft *centeredVel[iSamples*4+0];
            velocitiesX[indexVelocitiesXRight] +=coefRight *centeredVel[iSamples*4+0];
            velocitiesY[indexVelocitiesYBottom]+=coefBottom*centeredVel[iSamples*4+1];
            velocitiesY[indexVelocitiesYTop] +=coefTop *centeredVel[iSamples*4+1];
        }
    }
    enforceVelocitiesBorders();

    // Updates the corresponding data if stored on GPU
    if (objectVelocitiesCenters!=NULL)
    {
        GLfloat vertices[nbSamples*4*2];
        for (GLuint iSamples=0 ; iSamples<nbSamples ; iSamples++)
        {
            for (GLuint iCoord=0 ; iCoord<4 ; iCoord++)
            {
                vertices[(iSamples*2+0)*4+iCoord]=samples[iSamples*4+iCoord];
                vertices[(iSamples*2+1)*4+iCoord]=samples[iSamples*4+iCoord]-centeredVel[iSamples*4+iCoord]*vectorScale;
            }
        }

        glBindBuffer(GL_ARRAY_BUFFER, objectVelocitiesCenters->vboId);
        glBufferData(GL_ARRAY_BUFFER, objectVelocitiesCenters->nbVertices*4*sizeof(GLfloat), vertices, GL_DYNAMIC_DRAW);
    }

    if (objectVelocitiesBorders!=NULL)
    {
        GLfloat colors[objectVelocitiesBorders->nbVertices*4];
        GLuint iBorders=0;
        for (GLuint iY=0 ; iY<nbSamplesY ; iY++)
            {
                for (GLuint iX=0 ; iX<(nbSamplesX+1) ; iX++)
                {
                GLfloat normVelocityX=velocitiesX[iY*(nbSamplesX+1)+iX];
                // if (normVelocityX<0.0) normVelocityX=-normVelocityX;
                colors[iBorders*4+0]=normVelocityX;
                colors[iBorders*4+1]=0.0;
                colors[iBorders*4+2]=-normVelocityX;
                colors[iBorders*4+3]=1.0;
                iBorders++;
            }
        }

        for (GLuint iY=0 ; iY<(nbSamplesY+1) ; iY++)
            {
                for (GLuint iX=0 ; iX<nbSamplesX ; iX++)
                {
                GLfloat normVelocityY=velocitiesY[iY*nbSamplesX+iX];
                //if (normVelocityY<0.0) normVelocityY=-normVelocityY;
                colors[iBorders*4+0]=0.0;
                colors[iBorders*4+1]=normVelocityY;
                colors[iBorders*4+2]=-normVelocityY;
                colors[iBorders*4+3]=1.0;
                iBorders++;
            }
        }
        glBindBuffer(GL_ARRAY_BUFFER, objectVelocitiesBorders->colorsVboId);
        glBufferData(GL_ARRAY_BUFFER, objectVelocitiesBorders->nbVertices*4*sizeof(GLfloat), colors, GL_DYNAMIC_DRAW);
    }
}
```

Recopiez la et appellez la dans `advectVelocities()` et `applyForces()`.

b) De même, lorsque ce sont les velocités aux bords qui sont calculées, il faut mettre à jour à la fois les deux VBOs de `objectVelocitiesBorders` et `objectVelocitiesCenters`.

La fonction suivante réalise ces deux tâches.

```
void Simulation::updateVelocitiesFromBorders()
{
    if (objectVelocitiesCenters!=NULL)
    {
        GLfloat vertices[nbSamples*4*2];
        for (GLuint iY=0 ; iY<nbSamplesY ; iY++)
        {
            for (GLuint iX=0 ; iX<nbSamplesX ; iX++)
            {
                GLuint iSamples=iY*nbSamplesX+iX;

                GLfloat sampleVelocity[]={0.0, 0.0, 0.0, 0.0};

                // Interpolates the velocity at cell center (sampleVelocity)
                GLuint indexVelocitiesXLeft =iY *(nbSamplesX+1)+ iX;
                GLuint indexVelocitiesXRight =iY *(nbSamplesX+1)+(iX+1);
                GLuint indexVelocitiesYBottom=iY * nbSamplesX + iX;
                GLuint indexVelocitiesYTop =(iY+1) * nbSamplesX + iX;
                sampleVelocity[0]=(velocitiesX[indexVelocitiesXLeft]
                                   +velocitiesX[indexVelocitiesXRight])/2.0;
                sampleVelocity[1]=(velocitiesY[indexVelocitiesYBottom]
                                   +velocitiesY[indexVelocitiesYTop])/2.0;

                for (GLuint iCoord=0 ; iCoord<4 ; iCoord++)
                {
                    vertices[(iSamples*2+0)*4+iCoord]=samples[iSamples*4+iCoord];
                    vertices[(iSamples*2+1)*4+iCoord]=samples[iSamples*4+iCoord]-sampleVelocity[iCoord]*vectorScale;
                }
            }
        }

        glBindBuffer(GL_ARRAY_BUFFER, objectVelocitiesCenters->vboId);
        glBufferData(GL_ARRAY_BUFFER, objectVelocitiesCenters->nbVertices*4*sizeof(GLfloat), vertices, GL_DYNAMIC_DRAW);
    }


    if (objectVelocitiesBorders!=NULL)
    {
        GLfloat colors[objectVelocitiesBorders->nbVertices*4];
        GLuint iBorders=0;
        for (GLuint iY=0 ; iY<nbSamplesY ; iY++)
            {
                for (GLuint iX=0 ; iX<(nbSamplesX+1) ; iX++)
                {
                GLfloat normVelocityX=velocitiesX[iY*(nbSamplesX+1)+iX];
                if (normVelocityX<0.0) normVelocityX=-normVelocityX;
                colors[iBorders*4+0]=normVelocityX*10.0;//new
                colors[iBorders*4+1]=0.0;
                colors[iBorders*4+2]=0.0;
                colors[iBorders*4+3]=1.0;
                iBorders++;
                }
        }

        for (GLuint iY=0 ; iY<(nbSamplesY+1) ; iY++)
            {
                for (GLuint iX=0 ; iX<nbSamplesX ; iX++)
                {
                GLfloat normVelocityY=velocitiesY[iY*nbSamplesX+iX];
                if (normVelocityY<0.0) normVelocityY=-normVelocityY;
                colors[iBorders*4+0]=0.0;
                colors[iBorders*4+1]=normVelocityY*10.0;//new
                colors[iBorders*4+2]=0.0;
                colors[iBorders*4+3]=1.0;
                iBorders++;
                }
        }
        glBindBuffer(GL_ARRAY_BUFFER, objectVelocitiesBorders->colorsVboId);
        glBufferData(GL_ARRAY_BUFFER, objectVelocitiesBorders->nbVertices*4*sizeof(GLfloat), colors, GL_DYNAMIC_DRAW);
    }
}
```

Recopiez la fonction.

Vous pouvez à présent l'appeller depuis `project()`, juste avant l'update du VBO de `objectPressures`.

c) A plusieurs reprises dans le programme, nous allons devoir retrouver, à partir d'une position donnée, les indices iX et iY de la cellule superposée.

La fonction suivante réalise cette opération :

```
// Sets integer indices of the cell position pos overlaps
// 3D not implemented
void Simulation::getCell(GLfloat * pos, int * iX, int * iY, int * iZ)
{
    (*iX)=int(floor((pos[0]-offset[0])/h));
    (*iY)=int(floor((pos[1]-offset[1])/h));
}
```

Ajoutez la à votre programme.

B - Corrections

a) Corrigez ainsi la cible `bigclean` du **makefile** (en mettant bien une tabulation et pas des espaces) :

```
bigclean : clean
	rm -f $(EXE)
```

b) Corrigez le calcul de l'offset du coin inférieur gauche de la grille dans `Simulation::Simulation(...)` par :

```
// in Simulation::Simulation(...)
    // Start point left/bottom/back of the grid
    this->offset[0]=-size/2.0;
    this->offset[1]=-size*(GLfloat(nbSamplesY)/GLfloat(nbSamplesX))/2.0;
    this->offset[2]=-size*(GLfloat(nbSamplesZ)/GLfloat(nbSamplesX))/2.0;
```

c) Corrigez les deux premières lignes de `interpolateFromCenters(...)` :

```
Simulation::interpolateFromCenters(GLfloat * data, GLfloat * position, GLfloat * result)
{
    GLfloat xCorner=(position[0]-offset[0])/h;
    GLfloat yCorner=(position[1]-offset[1])/h;
    (...)
}
```

d) Corrigez les deux premières lignes de `interpolateFromBorders(...)` :

```
void Simulation::interpolateFromBorders(...)
{
    GLfloat xCorner=(position[0]-offset[0])/h;
    GLfloat yCorner=(position[1]-offset[1])/h;
    (...)
}
```

e) Une fois ceci fait, il est possible d'utiliser une grille non carrée, comme par exemple une grille aux proportions de l'écran :

```
// in main.cpp
    nbSamplesY=nbSamplesOneDir*application->height/application->width;
```

f) Retirez les lignes suivantes, inutiles et fausses, à la fin de la fonction `initParticles(...)` :

```
        int iX=(int)floor((particles[iParticles*4+0]+size/2.0)/h);
        int iY=(int)floor((particles[iParticles*4+1]+size/2.0)/h);
        if ( (iX>=0) && (iX<(int)nbSamplesX) && (iY>=0) && (iY<(int)nbSamplesY) )
```

g) Des modifications sont à apporter à la méthode `Simulation::conjugateGradient(...)`, donnée dans l'énoncé du TD03.

Les modifications sont l'initialisation des `doubles alpha` et `beta` à 0, un commentaire et le `cout` en sortie de fonction.

La méthode `Simulation::MICPreconditioner(...)` a également été modifée pour corriger une erreur d'indice (dans le second test, `iSBottom` au lieu de `iSLeft`).

Le **.pdf** en ligne a été mis à jour (vous pouvez donc indifféremment reprendre les deux fonctions).

▸ Exercice 2. Adaptations pour supporter les zones solides et vides

Malgré que nous ayons prévu l'utilisation de cellules typées, pour le vide ou le solide, nous n'avons pas encore fait toutes les adaptations qui s'imposent pour que ça fonctionne, notamment pour les cellules vides.

Voici les quelques modifications qui sont nécessaires.

A - Particules sur cellules fluides

Pour l'instant, les particules sont réparties sur toutes les cellules de la grille. Pour rendre la visualisation plus cohérente et esthétique, nous devons ne conserver que les particules sur les zones de type fluide.

Le code suivant réalloue le tableau de particules pour qu'il ne contienne que le sous-ensemble superposé aux cellules fluides. Placez le dans `initParticle(...)`.

```
// Init the particles positions, and interpolates colors and velocities from grid
// 3D not implemented
void Simulation::initParticles()
{
    (...)
    GLuint iParticles=0;
    GLfloat * particlesTmp=new GLfloat[this->nbParticles*4]; //new

    // Inits 4 particles in 2D cells (optimal distribution)
    // if (nbParticlesCoef==1) : corners of a losange in the square cell
    // if (nbParticlesCoef==2) : twice as more ...
        for (GLuint iSamples=0 ; iSamples<nbSamples ; iSamples++)
        {
            if (types[iSamples]==0)
            {
                for (GLuint iY=0 ; iY<nbAxis ; iY++)
                {
                    GLfloat offsetY=-halfCell + step/2.0 + iY*step;
                    for (GLuint iX=0 ; iX<nbAxis ; iX++)
                    {
                        GLfloat offsetX=-halfCell + sideOffsetX + iX*step;
                        if (iY%2==1) offsetX+=step/2.0;

                        particlesTmp[iParticles*4+0]=samples[iSamples*4+0]+offsetX;
                        particlesTmp[iParticles*4+1]=samples[iSamples*4+1]+offsetY;
                        particlesTmp[iParticles*4+2]=samples[iSamples*4+2];
                        particlesTmp[iParticles*4+3]=samples[iSamples*4+3];
                        iParticles++;
                    }
                }
            }
        }

    nbParticles=iParticles;
    delete [] particles;
    delete [] particleColors;
    delete [] particleVelocities;
    this->particles=new GLfloat[this->nbParticles*4];
    this->particleColors=new GLfloat[this->nbParticles*4];
    this->particleVelocities=new GLfloat[this->nbParticles*4*2];

    for (GLuint iParticlesCoord=0 ; iParticlesCoord<4*this->nbParticles ; iParticlesCoord++)
        this->particles[iParticlesCoord]=particlesTmp[iParticlesCoord];
    delete [] particlesTmp;

    for (GLuint iParticles=0 ; iParticles<this->nbParticles ; iParticles++)
    {
        // Moves particles randomly and at close range around initial position
        (...)
    }
}
```

B - Avection des velocités depuis les cellules non fluides

Lors de l'advection **semi-Lagrangienne** des vélocités ou couleurs, on interpole la vitesse à la position qu'on suppose être la position précédente des élements du fluide. Toutefois, dès l'instant qu'on travaille avec des cellules

de plusieurs types, il est possible que la position obtenue `virtualParticlePos` n'appartienne pas à une cellule de fluide.

Dans ce cas, la méthode à adopter est d'interpoler la quantitée advectée sur la limite de la zone fluide la plus proche de `virtualParticlePos`.

Dans des fonctions d'interpolation, on a déjà règlé le cas du bord de la grille, en prennant la valeur du bord lorsque la position à interpoler est dehors. Toutefois, le cas de l'interpolation dans une cellule vide ou solide dans la grille est indéterminé : la valeur obtenue sera celle stockée dans le tableau, indifféremment du type de la cellule. Pour bien, il faudrait réaliser cette distinction, mais elle n'est pas d'importance capitale par rapport à l'objectif créatif.

Par contre, nous pouvons adopter une méthode intermediare dans le cas spécifique de l'advection, en précisant que si `virtualParticlePos` est hors d'une zones fluide, nous ne souhaitons pas réaliser d'interpolation. Celà signifie que la quantité supposée advectée ne le sera pas partout, et notamment pas sur tous les points situés en fin de zone fluide, par rapport au déplacement. C'est un comportement physiquement acceptable, qui donne des résultats suffisants cette simulation.

a) Modifiez en conséquence `advectColors(...)`.

b) Modifiez également `advectVelocities(...)`.

C - Application des forces uniquement sur les cellules fluides

La gravité, et les forces en général, ne sont sensées être appliquées que sur les cellules de type fluide.

Modifiez la méthode `applyForces(...)` en conséquence.

D - Résolution uniquement sur les cellules fluides

La résolution du système qui vise à obtenir les pressions n'a de sens que sur les cellules de type fluide.

Par facilité, nous avons choisi de stocker - dans la matrices A, le vecteur rhs et le vecteur p - autant de lignes que de cellules de la grille. Il nous faut donc recourir à un moyen de préciser, lors de la résolution du système, que les lignes des cellules non fluide n'ont pas d'importance (elles ne seront jamais lues). En effet, nous ne voulons pas que ces cellules puissent retarder d'une quelconque manière la résolution. La résolution est terminée lorsque le maximum de la divergence des cellules passe sous un certain seuil. Un moyen d'empecher la méthode de résoudre les lignes vides ou solides est donc de fixer, articiellement et dès le départ, la divergence à 0. Ainsi les pressions des lignes concernées resteront inchangées et le système ne perdra pas de temps dessus.

Réalisez les changement nécessaires dans `project(...)`. Il faut notamment, dans un premier temps, remplir `rhs` à 0 lorque l'on n'est pas dans sur une cellule fluide. Dans un second temps, il ne faut soustraire, des velocités, le gradient de pression, que si l'on est sur une cellule fluide.

E - Utiliser les particules dans la simulation

a) La dernière modification à réaliser avant de pouvoir voir les trois types cohabiter naturellement est de faire passer les particules de vide à fluide et inversement au cours de la simulation. En effet, si un volume d'eau chute dans la grille sous l'effet de la gravité, les cellules du haut, originellement fluides, doivent devenir vides, et les cellules du bas doivent passer fluide.

L'algorithme le plus simple pour ce faire est celui des "**marker particles**". L'idée est d'utiliser des particules en mouvement dans la grille pour marquer les cellules de type fluide. Quand aucune particle n'appartient à la cellule, celle-ci est **vide**. Si la cellule contient au moins une particule, elle est **fluide**.

Les **marker particles** ne sont a priori pas les mêmes que les particules de visualisation, principalement car il n'est pas nécessaire qu'il y ait un grand nombre de **marker particles** (4 par cellules est un bon nombre, ce qui correspond à notre densité `1`). Une autre raison est que le Δt de simulation est sensé, a priori, être largement inférieur à celui du rendu. Pour chaque frame affichée, on aurait pour bien, réalisé plusieurs itérations de simulation. Dans ce cas, on voit bien que les particules de simulation et de visualisation devraient être distinctes.

Toutefois, pour faire simple, nous allons utiliser les particules déjà présentes dans notre programme comme des **marker particles**. Dans la fonction de déplacement des particules, on passe toutes les cellules non solides à vide. Puis pour chaque particule, on passe à fluide la cellule sur laquelle elle arrive.

Réalisez ces changements dans `advectParticles()`.

b) Quite à modifier `advectParticles()`, commentez les trois ensembles de lignes suivants :

```
// What is the color at this new position ?
interpolateFromCenters(colors, &(particles[iParticles*4+0]), &(particleColors[iParticles*4+0]));
for (GLuint iCoord=0 ; iCoord<4 ; iCoord++)
{
    velocityColors[(iParticles*2+0)*4+iCoord]=particleColors[iParticles*4+iCoord];
    velocityColors[(iParticles*2+1)*4+iCoord]=0.0;
}
```

```
glBindBuffer(GL_ARRAY_BUFFER, objectParticles->colorsVboId);
glBufferData(GL_ARRAY_BUFFER, objectParticles->nbVertices*4*sizeof(GLfloat), particleColors, GL_DYNAMIC_DRAW);
```

```
glBindBuffer(GL_ARRAY_BUFFER, objectParticleVelocities->colorsVboId);
glBufferData(GL_ARRAY_BUFFER, objectParticleVelocities->nbVertices*4*sizeof(GLfloat), velocityColors, GL_DYNAMIC_DI
```

On vient de désactiver l'utilisation de la couleur advectée sur la grille. En conséquence, les particules conservent la couleur attribuée lors de leur initialisation. Il s'agit donc d'une advection naturelle (la propriété de couleur de la matière est transportée lors de son déplacement). Ce mode de fonctionnement peut, selon les goûts, donner satisfaction, car il ne subit pas la diffusion (flou et mélange) de l'advection **semi-lagrangienne**.

Pour indication, dans la vidéo montrée en TD, les particules sont initialement colorées en bandes horizontales comme suit, et non par interpolation :

```
// in Simulation::initParticles() : end of last loop
    (...)
    //interpolateFromCenters(colors, &(particles[iParticles*4+0]), &(particleColors[iParticles*4+0]));
    particleColors[iParticles*4+0]=1.0;
    particleColors[iParticles*4+3]=1.0;
    if (iParticles<2*nbParticles/3) particleColors[iParticles*4+1]=1.0;
    if (iParticles<1*nbParticles/3) particleColors[iParticles*4+2]=1.0;
    (...)
```

Comme la couleur n'influe pas la simulation du champ de vitesse, il n'est pas grave qu'elle soit incohérente avec celle stockée sur les cellules.

En revanche, si la quantité advectée était utilisée dans la simulation, il faudrait reprendre l'advection semi-lagrangienne. Ainsi, ce qui serait simulé en terme de mouvement serait bien cohérent avec l'apparence des particules. Des exemples de quantités à advecter qui influent sur la simulation sont la température et la densité de suie dans une simulation de fumée.
Il existe aussi d'autres méthodes que l'advection semi-lagrangienne où l'on utilise les particules pour inscrire la quantité sur la grille, comme dans le cas des **marker particles**. La méthode **Particle-In-Cell** en est un exemple.

▸ **Exercice 3. Perfectionner le simulateur**

Sont données ici quelques explications complémentaires pour vous permettre de spécialiser votre programme, si nécessaire.

A - Solides en mouvement

Vous pourriez souhaiter placer des solides en mouvement dans la simulation.
Trois cas sont possibles :

- Le solide, très léger, est "porté" (ou "advecté") par le fluide sans impacter sa vitesse (ex : particules, neige, suie de fumée, feuilles...).
- Le solide par son mouvement pousse le fluide (ex: ventilateur, récipient en mouvement...) et affecte donc sa vitesse, sans lui-même être affecté.
- Le solide et le fluide s'impactent mutuellement (ex : moulin, hélice...).

Le premier cas est très simple à mettre en place et ne nécessite pas d'explications supplémentaires.

Le troisième cas nécessite une simulation nettement plus avancée (se référer au chapître 11 de [**FSCG**]).

Le second vous est accessible si vous êtes prêts à gérer les conditions aux bordures des solides avec un peu plus de finesse. En effet, la fonction `enforceVelocitiesBorders()` doit être modifiée pour que les vélocités normales aux bords solides soient celles des solides concernés. La friction (ralentissement du aux frottements) dans ce cas n'est pas simulée. Il faut, de plus, faire intervenir les vitesses des solides dans `rhs`. Enfin, nous avons négligé dans la simulation le fait de correctement interpoler aux bords des solides et non à l'interieur. Il faudrait donc améliorer `interpolateFromBorders()` à cet effet.

B - Fumées et vapeurs

Le principe de fumée est qu'une source émet de l'air chaud, chargé en particules de suie, dans un environnment plus frais. Comme l'air chaud est moins dense, il pèse moins lourd et s'élève. L'air froid le remplace en bas. La suie alourdit toutefois légèrement l'air chaud. Les mouvements respectifs de l'air chaud par rapport à l'air froid créent la dynamique d'une colonne de fumée (ascension, volutes), que les particules de suie rendent apparente.
Une simulation de fumée basique n'est en réalité pas beaucoup plus complexe que ce que vous avez déjà réalisé. En voici les étapes (pour plus de détails, se référer au chapître 5 de [**FSCG**]) :

- On travaille sur une large grille de cellules de type fluide, dont la densité est celle associée à de l'air.
- On stocke deux nouvelles valeurs par cellules : la température T [en Kelvins K] et la densité de fumée s.
- On créée une arrivée de fumée dans une cellule ou à la limite basse de la grille. Il s'agit de donner à cette zone une température supérieure à la température ambiante (273 K) et une densité de fumée supérieure à 0 (on adopte la convention que le maximum de s est 1).
- Au moment de la phase d'advection (couleur, vélocité), on advecte de la même manière que la couleur, les quantités s et T. On souhaite en effet que l'air s'élève avec sa température et sa suie.
- On ajoute la force $\vec{b}$ (pour "**buoyancy**"), inverse à celle de la gravité $\vec{g}$:

  $$\vec{b} = [\alpha s - \beta(T - T_{ambiant})]\vec{g}$$

  avec $\alpha = \frac{\rho_{soot} - \rho_{air}}{\rho_{air}}$ et $\beta = \frac{1}{T_{amb}}$ à règler de sorte que $|\alpha s - \beta(T - T_{ambiant})| << 1$.
- Il ne reste plus qu'à remplacer la constante de densité ρ dans la matrice A et dans le coefficient de ∇p soustrait à la vitesse dans `project()`. On interpole/prélève s en chaque point de la grille pour obtenir la valeur locale de ρ par la formule : $\rho = \rho_{air}(1 + \alpha s)$.

Une simulation avancée de fumée devrait garantir, en plus, la conservation de masse, en simulant le changement de volume de l'air selon sa température. Nous n'en sommes pas à ce niveau de détail.

Ajouter du bruit ou de la vorticité artificielle au mouvement pour en accentuer les détails est, par contre, pratique courante, nottamment dans les applications interactives (se référer au chapître 9 de **FSCG**). Il peut s'agir d'un élément intéressant pour améliorer l'apparition de formes dans votre projet.

C - Passage en 3D

Pour le passage de la simulation en 3D, la plupart des modifications à apporter au programme sont intuitives. Seul le calcul du préconditionneur pour la 3D n'est pas évident à deviner.
En voici donc la formule et l'agorithme de calcul :

$$E_{(i,j,k)} = \sqrt{\begin{array}{l} A_{(i,j,k),(i,j,k)} - (A_{(i-1,j,k),(i,j,k)}/E_{(i-1,j,k)})^2 \\ \quad - (A_{(i,j-1,k),(i,j,k)}/E_{(i,j-1,k)})^2 - (A_{(i,j,k-1),(i,j,k)}/E_{(i,j,k-1)})^2 \\ \quad - A_{(i-1,j,k),(i,j,k)} \\ \qquad \times (A_{(i-1,j,k),(i-1,j+1,k)} + A_{(i-1,j,k),(i-1,j,k+1)})/E^2_{(i-1,j,k)} \\ \quad - A_{(i,j-1,k),(i,j,k)} \\ \qquad \times (A_{(i,j-1,k),(i+1,j-1,k)} + A_{(i,j-1,k),(i,j-1,k+1)})/E^2_{(i,j-1,k)} \\ \quad - A_{(i,j,k-1),(i,j,k)} \\ \qquad \times (A_{(i,j,k-1),(i+1,j,k-1)} + A_{(i,j,k-1),(i,j+1,k-1)})/E^2_{(i,j,k-1)} \end{array}}$$

- Set tuning constant $\tau = 0.97$ and safety constant $\sigma = 0.25$
- For i=1 to nx, j=1 to ny, k=1 to nz:
  - If cell (i, j, k) is fluid:
    - Set $e = \texttt{Adiag}_{i,j,k} - (\texttt{Aplusi}_{i-1,j,k} * \texttt{precon}_{i-1,j,k})^2 - (\texttt{Aplusj}_{i,j-1,k} * \texttt{precon}_{i,j-1,k})^2 - (\texttt{Aplusk}_{i,j,k-1} * \texttt{precon}_{i,j,k-1})^2 - \tau [\texttt{Aplusi}_{i-1,j,k} * (\texttt{Aplusj}_{i-1,j,k} + \texttt{Aplusk}_{i-1,j,k}) * \texttt{precon}^2_{i-1,j,k} + \texttt{Aplusj}_{i,j-1,k} * (\texttt{Aplusi}_{i,j-1,k} + \texttt{Aplusk}_{i,j-1,k}) * \texttt{precon}^2_{i,j-1,k} + \texttt{Aplusk}_{i,j,k-1} * (\texttt{Aplusi}_{i,j,k-1} + \texttt{Aplusj}_{i,j,k-1}) * \texttt{precon}^2_{i,j,k-1}]$
    - If $e < \sigma \texttt{Adiag}_{i,j,k}$, set $e = \texttt{Adiag}_{i,j,k}$
    - $\texttt{precon}_{i,j,k} = 1/\sqrt{e}$

D - Rendu

L'aspect visuel de votre simulation est grandement porté par le mode d'affichage des particules. Chaque particule est une sprite (texture par point, projetée parallèlement à l'écran).
Vous pouvez en changer la texture, la taille, la couleur, le mode de transparence, l'illumination...
Rien ne vous empèche de fabriquer une normale (gràce à votre simulation, ou à l'aide d'une forme) et d'éclairer vos sprites selon cette normale.

Si votre simulation est en 3D, le rendu doit permettre de percevoir le volume. Le plus important mais le plus difficile serait d'avoir des ombres volumiques (ex : "**deep opacity map**").

Plus simplement (par quelques ajouts dans les shaders), il est possible d'enrichir le rendu en :

- mélangeant la couleur des particules, selon leur profondeur, à la couleur bleutée d'un fond ("**perspective aérienne**"),
- floutant selon le mouvement ("**flou de mouvement**") : par exemple en surimposant plusieurs rendus,
- floutant selon la profondeur ("**flou de profondeur**") : par exemple en mélangeant des textures plus ou moins floues selon z,
- réduisant la taille des sprites en s'éloignant de la caméra (**perspective**) : par exemple avec la formule $gl\_PointSize = \sqrt{\frac{1}{a+b\times d+c\times d^2}}$ (a, b, c et d au choix, à utiliser dans le vertex shader).

Le rendu d'une surface lisse d'eau est également trop complexe pour être décrit ici mais libre à vous de chercher de la documentation ("**Level Sets**"). Une courte explication est donnée en chapître 6 de [**FSCG**].

E - Optimisation

Si vous êtes interessés par l'optimisation, il est possible de tester et d'optimser les fonctions qui prennent le plus de temps, de paralléliser certaines boucles sur CPU, ou encore de passer tout l'algorithme sur GPU.

a) Benchmark
Pour évaluer les fonctions les plus coûteuses, vous pouvez réaliser un benchmark avec, par exemple, l'un des outils suivants :

- callgrind : durée en cycles processeur de chaque fonction
  `valgrind --tool=callgrind [executable]` → fichier `callgrind.out...`
  `kcachegrind callgrind.out...`

- cachegrind : taux de "**cache-miss**" (faible cohérence donc réutilisabilité des données chargées de la RAM en cache)
  `valgrind --tool=cachegrind [executable]` → fichier `cachegrind.out...`
  `kcachegrind cachegrind.out...`

b) Parallélisation CPU

Pour réaliser une parallélisation automatique entre plusieurs CPU, OpenMP est très simple d'utilisation.

- Ajoutez juste aux **cflags** : `-fopenmp`.
- Devant chaque boucle à paralléliser, ajoutez la ligne suivante :
  `#pragma omp parallel for schedule(dynamic, 1)`

Attention, les boucles parallélisables sont celles qui traitent des données indépendantes entre elles.

Par exemple, la boucle de convergence de l'agorithme de résolution ne peut être que séquentielle (pas de parallélisation possible). En revanche, toutes les boucles sur les particules et les cellules traitent bien des données indépendantes et sont parallélisables.

c) Parallélisation GPU

La parallélisation GPU des simulations de fluide de type grille est assez naturelle. La méthode de résolution vue ici n'est toutefois pas adaptée à la parallélisation.

▸ Exercice 4. Aide technique pour le projet

A - Sauvegarde sur disque

Si vous souhaitez travailler sur le rendu une fois votre simulation terminée, il peut être très interessant d'avoir sauvegardé vos grilles de vélocité à chaque frame dans un fichier texte, que vous pouvez relire à loisir, en temps réel.

Ainsi l'application de lecture peut être focalisée sur le rendu (moteur de rendu à part ou OpenGL, programme actuel ou adaptation). Vous pouvez intégrer votre fluide à une démo temps réel, ou bien juste règler tranquillement tous les paramètres de rendu.

B - Charger la forme

Pour le chargement de la forme, les deux options sont :

- charger direment une image ou une vidéo en image successives, et utiliser les informations des pixels dans la simulation et/ou le rendu,
- faire un premier rendu d'une scene 3D et d'utiliser les images produites (2D, couleur, depth...) en entrée du second rendu.

C - Export vidéo

N'oubliez pas que vous disposez d'un script shell pour générer une vidéo (cf TD01 - Exercice 1 - partie D). Testez le !

D - Cadrage

N'hésitez pas à utiliser le maximum de votre écran pour la simulation, et de vous mettre en plein écran avant l'enregistrement de vos vidéos. Le cadrage sur votre fluide peut être un moyen de créer du sens et de faire apparaître une forme, expérimentez !

E - Règlages

Vous devrez faire de nombreux essais avant d'obtenir la configuration optimale, n'attendez pas le dernier moment pour lancer des rendus de tests, dans lesquels vous apprennez à règler au mieux tous les paramètres (et gérez les éventuelles **seg faults**...).

Δt, Δx, la fréquence de rendu, le nombre de particules par cellules et la taille des particules sont les paramètres les plus importants.

Dans l'exemple vidéo, $nbSamplesX = 100$, $nbParticlesCoef = 3$ et $\Delta t = \frac{1}{30}$.

Il faudrait surement augmenter la fréquence de simulation tout en conservant une telle fréquence de rendu.

▸ **Exercice 5. Aide artistique**

Voici une liste absolument non exhaustive d'idées pour faire apparaître une forme dans la simulation.

Mots d'ordre : intangibilité et poésie !

A - Utilisation des solides
Les zones solides peuvent servir à délimiter une forme (à l'interieur ou à l'exterieur du fluide).

Si la zones solide contient le fluide, des particules aux frontières peuvent être laissées libres avec leur inertie pour enrichir les contours. Les bords solides dans la simulation peuvent être rendu, par le traitement des particules, avec plus ou moins de transparence au fluide.

B - Utilisation des forces
Les forces peuvent suivrent les tangentes de la surface d'une forme, suivrent des directions principales de la surface ou parcourir son squelette (axes internes).

C - Utilisation du rendu
La forme peut n'apparître que par le subterfuge d'un rendement différentié des particules (normales et illumination, densité et taille, trainées plus ou moins réalistes...).

D - Déplacement de la caméra
Une caméra, se déplaçant lentement le long des zones fluides interagissant avec la forme, peut permettre de la faire apparaître doucement.

E - Accumulation de particules
Des particules, de la suie ou d'autres petits objets (feuilles, flocons), peuvent s'accumuler sur la forme avec le temps qui passe dans la simulation.

F - Simulation 2D sur mesh
Une simulation 2D peut être plaquée sur un mesh 3D de la forme.

G - Simulation 2D en rideaux successifs
Des effets de profondeur peuvent être obtenus par plusieurs simulations 2D à différentes profondeurs ou superposées.

H - Solidification ou destructuration/fonte/vaporisation
La forme peut être un solide qui se transforme en fluide, ou l'inverse.

I - Ajout artificiel de mouvement
L'alteration articificielle légère des positions ou vitesses peut faire apparaître une forme un peu subliminale.

▸ Exercice 6. Rendu final du projet

Vous devez rendre une archive de code, un court rapport au format .pdf et imprimé, et votre vidéo. La vidéo est très largement l'élément le plus important. N'hésitez pas à l'enrichir de musique si ça porte mieux votre thème.

Le rapport papier doit être déposé le mercredi 14 décembre à 14h00 au bureau de Sylvie Donard). Tous les autres éléments peuvent être délivrés par clé usb, téléchargement ou mail le jour de la soutenance. Celle-ci aura lieu en présence de votre chargé de TD et enseignant de CM, le jeudi 15 décembre au matin.

A - Mini-rapport

Un rapport de quelques pages doit décrire brièvement :

- votre simulateur (résumé du fonctionnement de la méthode vue en TP et spécialisations),
- la direction artistique et la méthode de rendu que vous avez choisi,
- le choix du thème et le sens d'utiliser un fluide pour le dessiner,
- les diffultés rencontrées,
- les apprentissages et évolutions potentielles de votre projet,
- des images explicites du cours de votre avancement, illustrant vos différents tests et le résultat final.

B - Archive de code

Votre programme doit fonctionner sur les ordinateurs de la fac (linux). Si vous avez besoin de plus de temps pour réaliser un changement de plate-forme, vous disposerez d'un délai supplémentaire pour le rendu de l'archive.

C - Soutenance-démo-projection

Lors de votre soutenance, vous devez expliquer votre process de création, en reprennant les éléments du rapport et en illustrant par des animations.
Vous n'êtes pas tenus de faire fonctionner votre programme devant le jury mais devez par contre mettre en avant votre film.

Les soutenances sont bien sûr publiques. Il serait intéressant que vous assistiez aux rendus des autres groupes. Un visionnage global peut sinon être organisé en fin de matinée.