Projet de spécialité 2010

Conception d'un modèle de feu 3D temps réel

Étudiants impliqués :
Benjamin Aupetit - IRVM - benjamin.aupetit@ensimag.imag.fr
Julien Champeau - IRVM - julien.champeau@ensimag.imag.fr
Arnaud Emilien - IRVM - arnaud.emilien@ensimag.imag.fr

Encadrants :
Marie-Paule Cani - Marie-Paule.Cani@inrialpes.fr
Aurélie Catel - aurelie.catel@grenoble-inp.fr

Ensimag 2010

Table des matières

1 Présentation du projet

1.1 Introduction

Il existe de nombreux travaux de modélisation du feu. Cependant, peu d'entre eux modélisent la totalité du processus de combustion. Nous avons trouvé interessant de répondre à la problématique : comment modéliser une combustion réaliste et temps réel ?

1.2 Objectifs initiaux

Le but principal de notre projet est de réaliser un modèle de combustion en 3D temps réel.
Cette modélisation devra être la plus réaliste possible : en effet nous voulons permettre d'intégrer notre solution dans un environnement temps réel qui requiert un comportement proche de la réalité en restant interactif. Dans ce sens notre solution ne sera pas adaptée à tous les jeux vidéos, car ceux ci ne nécessitent pas une approche réaliste mais juste des effets visuels impressionnants.

Notre solution devra ainsi répondre aux critères suivants :
- Modèle de flamme convainquant.
- Génération de fumée.
- Propagation réaliste sur l'objet.
- Prise en compte des flux d'air (vent, advection du feu...)
- Interaction du système avec les objets.
- Destruction procédurale progressive d'un objet.
- Propagation à l'environnement.

1.3 Moyens utilisés

Sachant que nous voulons réaliser une application temps réel, nous avons décider dans un premier temps de le réaliser sur **CPU** en utilisant **OpenGL** pour valider le modèle, puis de l'adapter en **Shader GLSL**. Les calculs réalisés étant hautement parallélisables, la programmation sur GPU est indispensable.

1.4 La démarche

Notre démarche a été similaire à celle d'un chercheur effectuant une nouvelle publication. En effet, nous avons commencé par effectuer une série de recherche bibliographique pour saisir au mieux les avantages et inconvénients de différentes méthodes, et ainsi d'en tirer le meilleur pour réaliser notre modèle.
Nous avons effectué un compte rendu de la plupart des articles lu, ceux-ci se trouvent dans la partie **recherche bibliographique** de ce document.
Nous avons ensuite implémenté notre modèle dont nous justifions les choix dans la suite de ce document.

1.5 Résultats

Nous avons ainsi implémenté totalement le modèle de fluide et d'objet sur CPU, et le modèle de fluide sur GPU. Le modèle d'objet nécessite une adaptation plus importante aux fonctionalités de la carte graphique, que nous expliquerons dans la suite de ce document.
Le modèle CPU présente un taux d'image par secondes suffisant pour montrer le bon fonctionnement du modèle. Toutes les fonctionalités du modèles sont implémentées et opérationelles. (environ 9 fps avec une grille 40x40x40)
La version GPU est comme prévu beaucoup plus rapide et permet d'utiliser une taille de grille plus importante. (environ 9 fps avec une grille 100x100x100)
Nous sommes donc satisfaits de notre travail, nous avons réussi à obtenir un modèle de combustion complet et intéractif en CPU, et un modèle de fluide complet en GPU. L'ajout d'un modèle d'objet au GPU de devrait pas trop affecter les performances. Les performances sont donc suffisamment bonnes pour valider notre contrainte de temps réel.

2 Aperçu de notre modèle

Ici nous présentons un résumé de notre modèle global. Ce modèle a été établi pour unifier les différents modèles du processus de combustion. En effet nous devons nous assurer de la compatibilité de ceux ci, pour éviter les incohérences entre les différentes étapes, ou des calculs de synchronisation entre les phases.

2.1 Les flammes

Pour modéliser les flammes il faut différencier deux aspects. Le rendu des flammes d'une part, et la modélisation du phénomène associé d'autre part.

2.1.1 Un modèle de fluide pour le feu

Le modèle de fluide est basé sur les travaux de **Jos Stam, stable fluids**. Ce modèle inclue la diffusion de la densité de gaz, la densité de fumée, la chaleur, la vitesse de l' air. Il permet aussi de gérer plus simplement la diffusion du feu entre les objets dans tout son environnement.

2.1.2 Le rendu des flammes et de la fumée

Pour rendre les flammes nous avons besoin d'une méthode rapide mais donnant une impression de réalisme. Pour faire ceci nous utilisons une méthode inspirée de celle de **Zeki Melek**. Notre méthode consiste à transformer la matrice du fluide en texture3D, puis de découper le fluide en "tranches" orthogonales à l'orientation de la camera. Pour chaque tranche, nous calculons les coordonnées de textures des points la constituant. Ainsi notre méthode de **rendu de texture3D** permet de visualiser le fluide à 360 degrés.
Nous utilisons de plus un shader GLSL de **multitexturing**. Ce shader permet dans un premier temps de choisir si nous devons afficher la flamme ou la fumée, en fonction qui rajoute un **bruit de Perlin** aux coordonnées de texture, multiplié par un **dégradé vertical**. Ceci permet d'avoir une flamme réaliste, dont les flammes sont plus violentes dessus que dessous.
Enfin, nous déterminons la couleur de la flamme en nous basant sur le **principe du corps noir**, qui détermine la coloration de la flamme en fonction de la chaleur.

2.2 Les objets

2.2.1 L'ajout des objets au modèle de fluide

Nos objets sont représentés sous formes de **grilles de voxels**, ce qui permet d'interagir facilement avec la grille uniforme du fluide. Ces grilles de voxels sont indépendantes du fluide, et propres à chaque objets. Nous réalisons le liens entre les objets et le fluide en nous basant sur les positions des boites englobantes des objets et du fluide, puis en convertissant les coordonnées dans un modèle en coordonnées dans un autre.
Les voxels de l'objet ont tous des paramètres propres, ce qui permet de dégrader progressivement l'objet, de générer du gaz et de la chaleur, d'influencer le champs de vitesse... Ces paramètres offrent la possibilité d'avoir plusieurs types de matériaux : citons par exemple des matériaux insensibles à la chaleur, d'autres prenant feu, et encore d'autres qui se dégradent sans bruler...

Les objets peuvent être chargés à partir d'un mesh au format **.obj**. Nous convertissons ce mesh en grille de voxel grâce à notre calculateur de **champs de distance**.

2.2.2 Interactions avec le fluide

Lorsqu'un objet rentre en contact avec le fluide, il y a **échange de chaleur aux bords de l'objet**. Puis nous avons modélisé la diffusion de la chaleur au sein de l'objet en nous servant de **l'équation de la chaleur**.
Nous avons ajouté autour de l'objet un champs de répulsion qui force le fluide à contourner l'objet.

2.2.3 Le rendu des objets

Au niveau du rendu, nous utilisons un **marching cube adaptatif**. Cela nous permet d'afficher simplement et rapidement la grille de voxels. De plus, lorsque nous brulons un voxel, nous signalons ce voxel brulé à l'algorithme, qui recalcule la surface uniquement là ou c'est nécessaire.

3 Modèle de fluide pour la flamme, la fumée et la chaleur

Comme décris dans l'aperçu, nous nous sommes focalisé sur un modèle "réaliste" qui peut être exploité en temps réel. Pour cela nous avons choisit un modèle de fluide pour représenter les flammes et la fumée. Nous avons préféré le modèle de fluide au modèle de particules car c'est un modèle qui s'adapte parfaitement à la combustion d'un objet.

Le principe de base est le suivant : lorsqu'un objet brûle il libère un fluide inflammable, si la température est suffisante ce gaz s'enflamme, sinon il devient en partie de la fumée. Quand le combustible enflammé s'éloigne de la source de chaleur, il refroidi et lorsqu'il atteint une température suffisamment faible il se transforme en un autre gaz(invisible) et en fumée.

Pour faire cela nous avons besoin de quatre informations en permanence : le champ de vitesse de l'air, le champs de température, le champs de combustible et le champs de fumée. En nous basant sur le modèle développé par **Jos Stam** dans **Stable Fluid** nous pouvons faire évoluer le système.

3.1 Plus de détails :

Le modèle de fluide qui semble être bien adapté à la représentation du feu est basé sur le principe des équations de Navier-Stokes pour la dynamique des fluides indéformables :

$$\frac{\partial x}{\partial t} = -(\vec{u}.\nabla)x + \nu\nabla^2 x + \vec{f} \tag{1}$$

Ou u représente le champ de vitesse du milieu et x la valeur transporté par ce champ (densité par exemple)

Cette équation est constitué de 3 termes principaux (4 en théorie mais on a pas introduit l'effet de la pression ambiante) :

- Un terme d'advection : $-(\vec{u}.\nabla)x$
  Le champ de vitesse va servir à transporter la quantité de fluide dans l'espace. L'auto-advection du champ de vitesse ($-(\vec{u}.\nabla)\vec{u}$) est ce qui va permettre le mouvement des particules de gaz dans le milieu.

- Un terme de Diffusion : $\nu\nabla^2 x$
  Le terme ν représente la viscosité du fluide et le terme de diffusion de l'équation de Navier-Stokes sert en fait à modéliser la capacité d'un fluide à se mouvoir dans l'espace qui l'entoure. Plus ν est élevé et moins le fluide pourra se mouvoir dans le milieu.

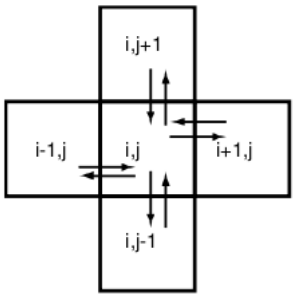
- Un terme représentant les forces extérieures : $\vec{f}$
  Ce terme sert a modifier le comportement de la valeur x dans le champ de vitesse lorsqu'une force extérieure rentre en jeu (du vent par exemple).

Pour construire notre modèle de feu, nous allons considérer 2 variables : $\vec{u}$ le champ de vitesse et ρ la densité du fluide. Notre feu sera alors un champ de densité en mouvement dans un champ de vitesse.

La résolution se fera par étapes : l'ajout des forces externes, la diffusion du fluide et l'impacte de l'advection.

1. Les forces :
   L'ajout des forces externes dans le cas de la densité consiste en fait a ajouter une source de densité la ou on le souhaite.

2. La diffusion :



Echange d'informations dans la grille - Jos Stam

Cela consiste à échanger la densité d'une case de notre grille de l'espace avec ses 6 voisins (haut, bas, gauche, droite, devant, derrière). Ce qu'il se passe en fait c'est que la case va perdre de la densité en la cédant à ses voisins mais va aussi en gagner car ses voisins aussi sont soumis a cela et lui transfère aussi du fluide.

Résolution :

$$\frac{\partial \vec{u}}{\partial t} = \nu \nabla^2 \vec{u} \tag{2}$$

$$\vec{u}(\vec{x}, t + \delta t) = \vec{u}(\vec{x}, t) + \nu \delta t \nabla^2 \vec{u}(\vec{x}, t) \tag{3}$$

D'où :

$$(I - \nu \delta t \nabla^2)\vec{u}(\vec{x}, t + \delta t) = \vec{u}(\vec{x}, t) \tag{4}$$

Calcul du laplacien :

$$\nabla^2 p = \frac{p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k}}{(\delta x)^2} + \frac{p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j+1,k}}{(\delta y)^2} + \frac{p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}}{(\delta z)^2} \tag{5}$$

Qui dans le cas où $\delta x = \delta y = \delta z$ nous donne :

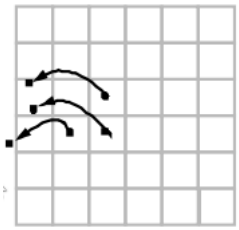
$$\nabla^2 p = \frac{p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - 6p_{i,j,k}}{(\delta x)^2} \tag{6}$$

Grâce a cela on trouve facilement la formule pour la résolution :

$$u^{l+1}_{i,j,k} = \frac{u^l_{i+1,j,k} + u^l_{i-1,j,k} + u^l_{i,j+1,k} + u^l_{i,j-1,k} + u^l_{i,j,k+1} + u^l_{i,j,k-1} + \frac{(\delta x)^2}{\nu \delta t} u^l_{i,j,k}}{6 + \frac{(\delta x)^2}{\nu \delta t}} \tag{7}$$

La résolution linéaire de cette équation combinée à une relaxation de Gauss-Seidel nous permet de considérer une méthode stable, même pour des taux de diffusion ($\frac{\nu \delta t}{(\delta x)^2}$) grand.

3. L'advection :



Principe de l'advection - Jos Stam

L'advection est le procédé par lequel le champ de vitesse du fluide transporte la matière d'un endroit a un autre. Le plus simple est de voir la densité de matière comme une quantité de particules et qu'il faut voir a chaque pas de temps. Et ce que l'on fait c'est que l'on considère les centres de nos cellules comme étant des particules, et plutôt que de regarder ou vont se rendre ces particules, on regarde d'ou elle proviendrait si on faisait un pas de temps en arrière. On interpole ainsi les densités depuis leur position d'avant avec celles de leurs plus proches voisins.

3.2 Nos améliorations :

Par rapport au modèle de **Stam**, nous avons rajouté des éléments en plus nous permettant de gérer plusieurs phénomènes pour notre feu :

3.2.1 La Température :

La température est un facteur important pour le feu car la température va générer une force (**buoyancy**) qui va modifier le champs de vitesse (l'air chaud qui monte) et donc cela va avoir un impact sur le mouvement de notre flamme.
Cette force a pour forme :

$$\vec{f} = \sigma T \vec{k} \tag{8}$$

Nous avons donc un paramètre de température en plus de la densité de gaz, qui est géré par les mêmes équations, mais a cela nous avons rajouté un coefficient de refroidissement qui fait baisser la température lorsque l'on s'éloigne de la source.

3.2.2 Vorticity confinement :

Le fait d'utiliser une méthode de relaxation pour rendre le système stable a pour conséquence de provoquer une dissipation d'énergie dans notre calcul et donc une perte de donnée.

Le moyen de palier à ce problème est d'utiliser la technique de **Ron Fedkiw : "vorticity confinement"** qui redonne au fluide l'énergie perdue par le calcul du laplacien.

3.2.3 Génération de fumée :

Le feu (ainsi que les objets qui brûlent) produit de la fumée, nous avons donc géré un deuxième fluide évoluant dans le même champ de vitesse avec des paramètres qui lui sont propres.

Nous avons donc introduit un paramètre de **consommation de la matière** qui a chaque pas de temps va diminuer la densité de fluide "feu". En parallèle nous produisons de la densité "fumée" la où on consume le feu , modulé avec un paramètre de conversion (ex : 10 particules de feu forment 3 particules de fumée).

3.2.4 Résumé :

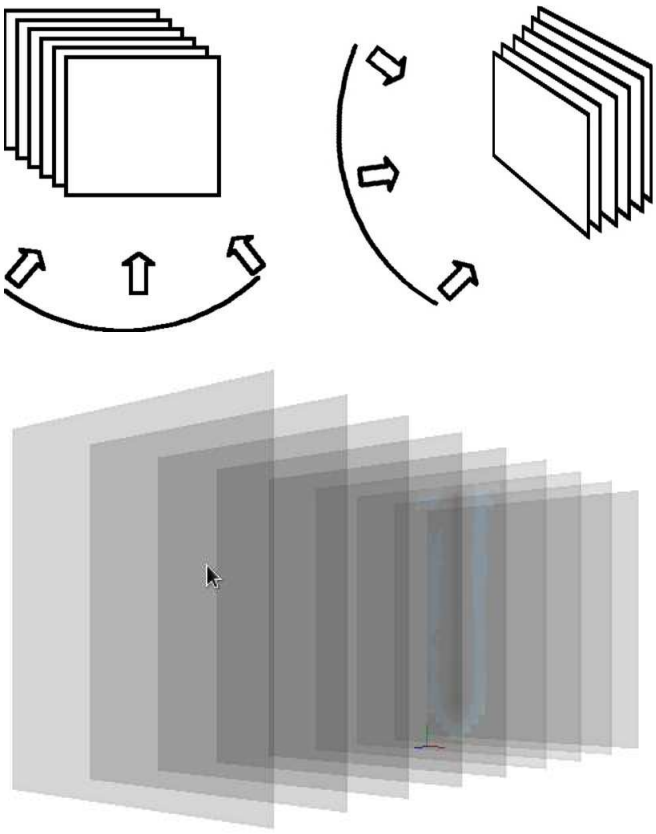
Notre modèle de fluide est donc constitué de 4 grilles à pas constant, une grille pour le gaz, une grille pour la fumée, une grille pour la chaleur, et une dernière grille pour la vitesse. Nous traitons ces grilles selon le principe de résolution de Jos Stam, avec des traitements supplémentaires qui ont été détaillés ci dessus.

3.3 Méthode de rendu

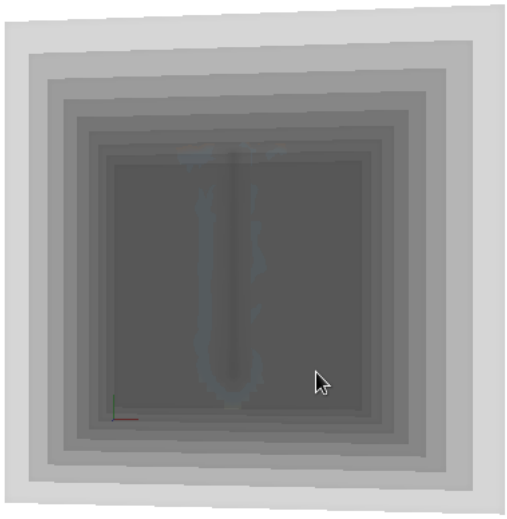
3.3.1 Affichage volumique

Dans notre cas nous avons à faire à des textures3D, qui peuvent impliquer des problèmes lors de l'affichage. Nous avons donc opté pour la solution suivante :

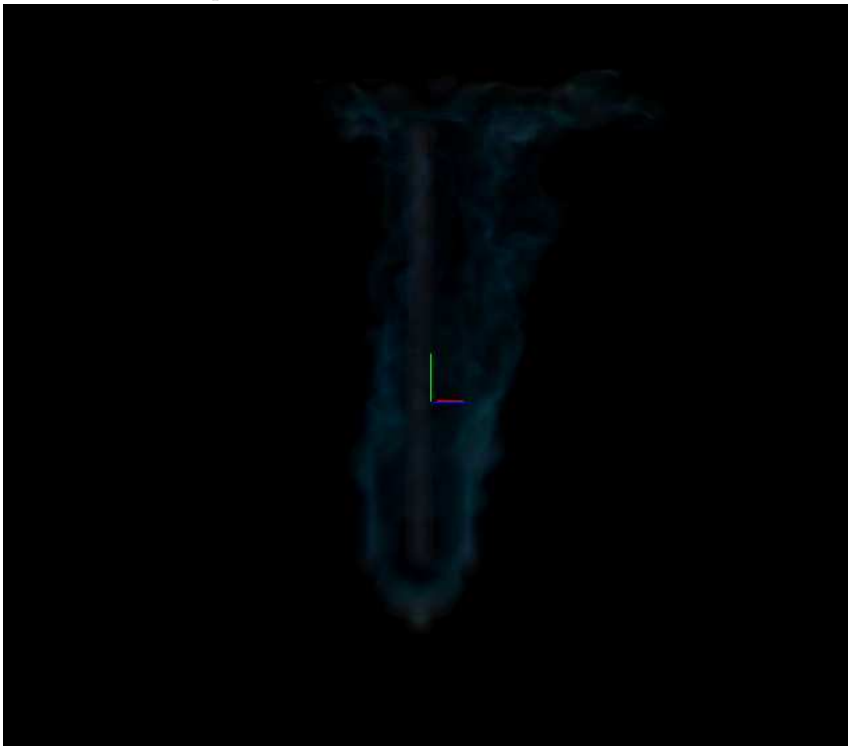
Nous affichons N plans de coupe de notre texture3D, chaque plan étant orthogonal a la direction de la caméra. Pour cela nous avons du implémenter tout un ensemble d'opérations faisant en sorte que les coordonnées de texture des plans orthogonaux à la caméra correspondent à l'orientation de la flamme dans l'espace. Ceci nous permet alors de visionner notre texture volumique depuis n'importe quel point de l'espace.



Plans de coupes dans la texture3D.



Plans de coupes dans la texture3D, orientés faces à la camera.



Plans de coupes dans la texture3D, orientés faces à la camera, affichés normalement.

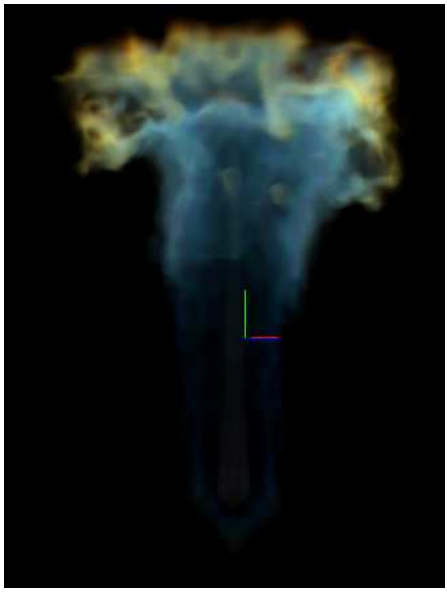
3.3.2 Rendu de la flamme

Une flamme ne saurait être jolie si l'on ne prennait pas en compte la variation de la couleur en fonction de la température. Pour cela nous avons considéré notre flamme comme un modèle de corps noir, c'est à dire une entité qui rayonne à des longueurs d'ondes différentes en fonction de sa température, cela en utilisant la **formule de Planck** pour le corps noir :

$$L_\lambda = \frac{2hc^2\lambda^{-5}}{exp(\frac{hc}{k\lambda T}) - 1} \tag{9}$$

Où h est la constante de planck, c la vitesse de la lumière, et k la constante de Boltzmann.

On obtient donc une relation entre la température de notre flamme et sa luminance, ce qui nous permet d'avoir un rendu fonction de la température. De plus nous avons ajusté la transparence de la flamme en réglant l'α en tenant compte de la densité de gaz qui brule et de la température de la flamme (les flammes bleus sont plus transparentes que les rouges et jaunes).
Pour éviter de calculer cette formule assez complexe à chaque fois, nous **stockons** un grand nombre de valeurs dans un tableau.



Rendu de la couleur de la flamme en fonction de la température de l'air.

3.3.3 Rendu de la fumée

Pour le rendu de la fumée, il s'agit simplement d'un niveau de gris avec une transparence α fonction de la densité de fumée en un point.

3.3.4 Multitexturing

Un problème d'affichage qui se posa rapidement à nous fut le moment ou nous devions afficher la flamme ET la fumée avec une texture volumique. Le fait d'afficher sous forme de plans posa des problème au niveau de la précision du Z-Buffer d'OpenGL qui ne savait plus quelle texture était devant l'autre et cela provoquait un effet de clignotement indésirable.

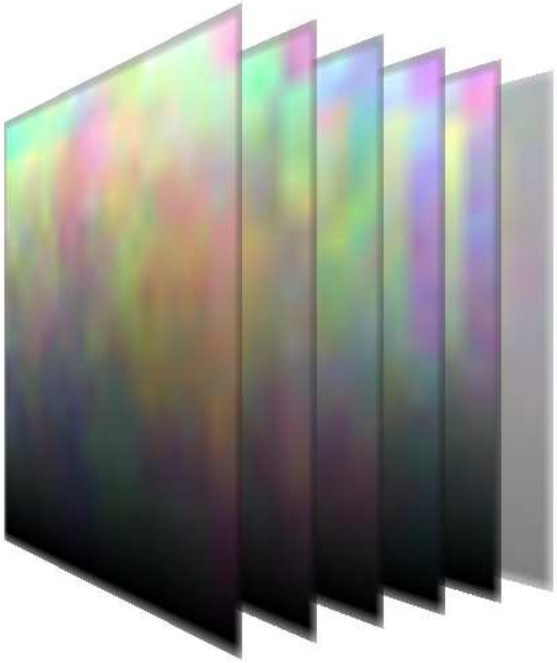
La solution que nous avons retenue pour palier à ce problème était de réaliser un **fragment shader** pour faire un affichage Multitexturé prenant en compte l'alpha des 2 texture.

Le principe en lui même est assez simple :
Pour chaque pixel de notre texture
si $\alpha_{texture1} > \alpha_{texture2}$ alors
affichage texture1 ;
sinon
affichage texture2 ;
fin

3.3.5 amélioration de rendu : Bruit de Perlin 4D

Dans la réalité il y a toujours un "vent ambiant" autour d'une flamme, ainsi que de nombreux phénomènes macroscopiques, ce qui à pour conséquence de multiples effets de mouvements au niveau de

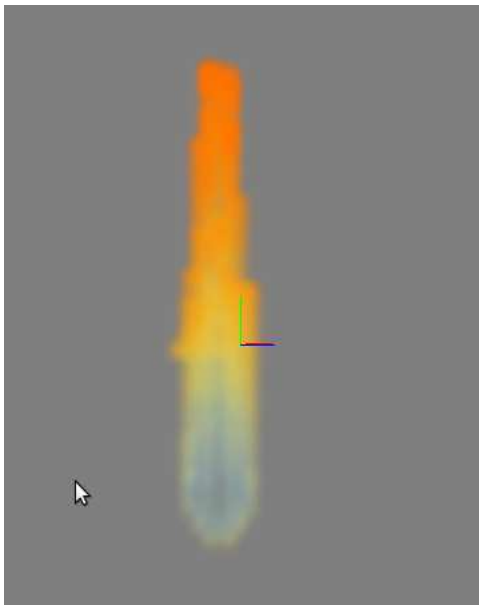
sa surface. Pour simuler cet effet nous ajoutons lors du rendu par le fragment shader un **bruit de perlin 4D** (dont une dimension temporelle pour conserver la cohérence du mouvement).



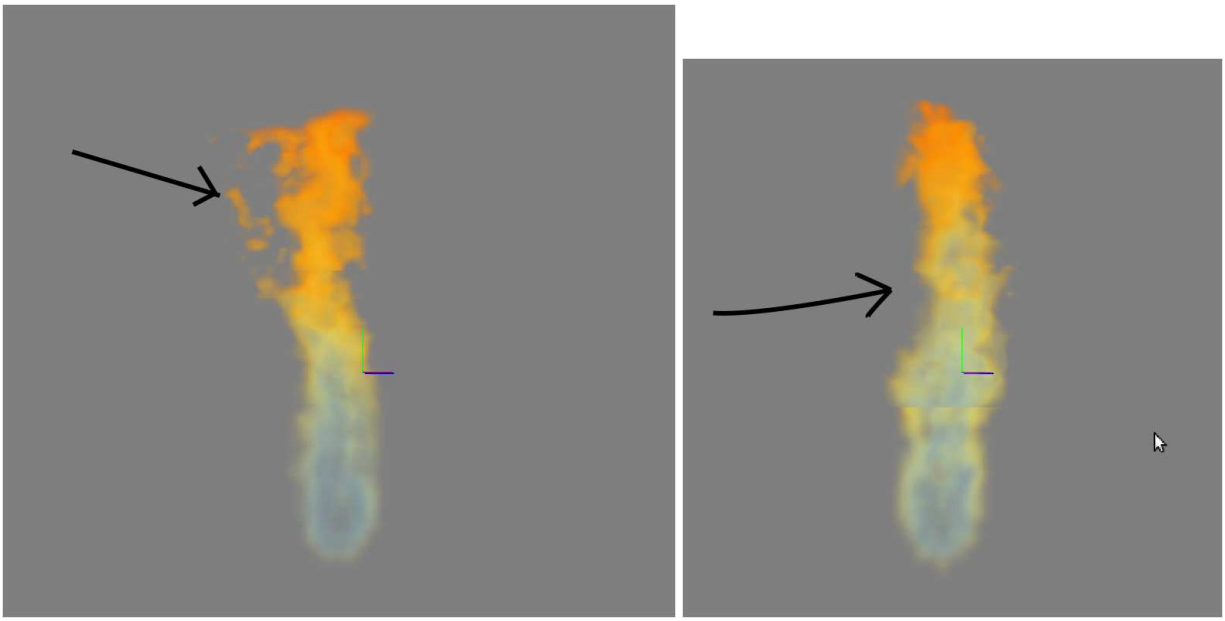
Le fait d'ajouter ce bruit lors du rendu dans le shader ne modifie pas la position spatiale des densités, des températures et de la fumée et ne perturbe donc pas les calculs de l'équation de dynamique des fluides.

Les rendus obtenus étant plutôt convaincant :

Sans bruit de Perlin :

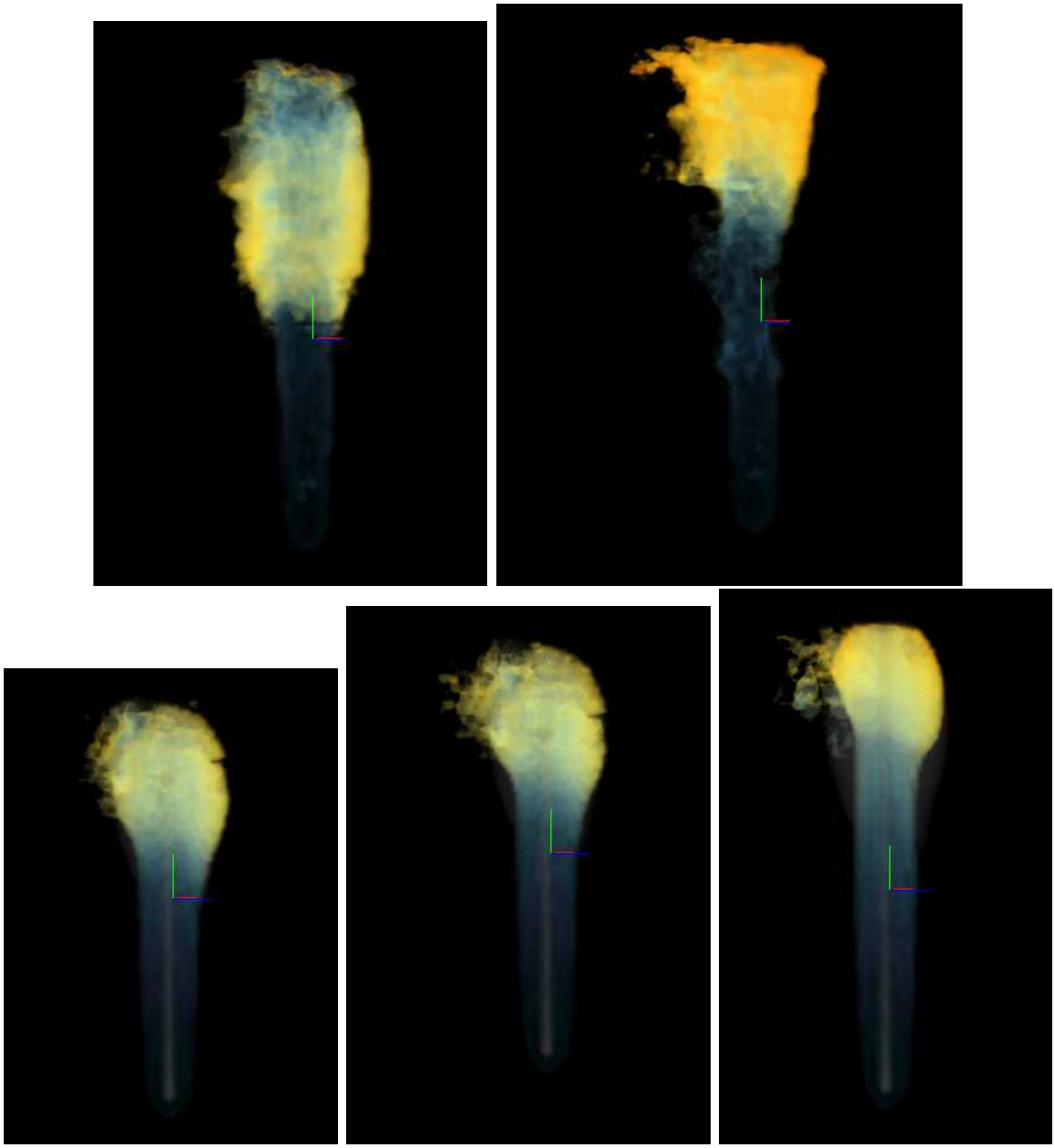


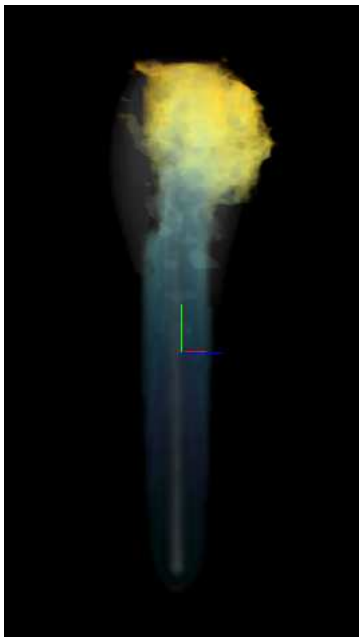
Avec Bruit de Perlin :



L'expression mathématique du bruit de Perlin est la suivante :
$P_n(x) = \sum_{i=0}^{n-1} p^i bruitPerlin(2^i.x)$.
C'est une série (absolument) convergente, majorée par la série géométrique :
$\sum_{i=0}^{n-1} p^i = \frac{1-p^n}{1-p}$.

3.4 Résultats





4 Modèle de flamme et de fumée GPU

Nous utilisons le langage **GLSL** (Open**GL S**hading **L**anguage). Ce langage de shader est utilisé en général pour le rendu de scène 3D. Le shader utilisé est un **fragment shader**. Il est (très grossièrement) destiné à changer la couleur d'affichage d'un objet.
D'autres outils de programmation existent (comme OpenCL) mais nous n'étions pas tous compatibles avec cette technologie. Nous avons donc choisi le GLSL qui est suporté par un bien plus grand nombre de carte graphiques.

Nous avons donc du mettre en place des outils permettant de résoudre notre système de fluide avec des outils qui sont à priori pas destinés à ce genre de problèmes.

4.1 Transformation du problème pour une résolution GPU

4.1.1 Matrice 3 dimensions == Texture 3D

Notre matrice 3 dimensions dans lequel nous traitons le modèle de fluide peut se rapprocher d'une **Texture 3D** d'OpenGL.
Il faut cependant noter que les couleurs d'une matrice sont stockées entre 0.0 et 1.0, ce qui implique un changement de repère entre les résultats des calculs et la façon de les stocker.

4.1.2 Operation sur le point (i,j,k) de la matrice == FragmentShader

Un shader c'est un programme qui s'execute sur une carte graphique. Un fragment shader est un programme qui s'applique sur la couleur d'un pixel d'un objet.

Le principe de fonctionnement d'un shader est simple. On active le shader, on dessine, on désactive le shader. Tout ce qui est dessiné pendant qu'il est actif subit toutes les modifications effectuées par le shader.

Ainsi lorsqu'on fait une triple boucle pour parcourir l'ensemble des éléments du tableau, il suffit d'appliquer un shader sur notre texture3D et toutes les couleurs de celle-ci seront modifiées.

La première difficulté est de pouvoir traiter l'ensemble de la Texture 3D avec le shader. On peut le faire en dessinant plusieurs fois un plan et en changeant ses coordonnées de texture. Il ne faut pas oublier de regler la taille du **viewport** du rendu qui doit être égal à la taille de la grille.

4.1.3 Itération == Framebuffer (render to texture 3D)

Un **shader** ne s'applique qu'au moment du dessin. C'est à dire qu'on active le shader, on dessine, on désactive le shader. Tout ce qui est dessiné lorsqu'il est actif subit ses modifications. Mais du coup, la texture 3D initiale n'est jamais modifiée. Or lorsqu'on fait une itération dans le calcul il est essentiel de conserver le résultat dans une texture 3D.

Avec une méthode de **render to texture** il est possible d'enregistrer ce qui s'affiche à l'écran dans une texture.

Principe du render to texture 3D : Nous avons un **FrameBuffer**. Lorsqu'il est actif, le dessin ne s'affiche pas sur l'écran, mais est conservé en mémoire. Nous pouvons rediriger la sortie du framebuffer vers un tranche (un layer) de la texture3D.

Il nous faut donc deux textures : la texture courante, et la texture dans lequel on enregistrera les résultat du traitement par le shader.

Si notre texture 3D fait 128 x 128 x 128, nous allons dessiner dans un viewport de 128x128 chaque tranche (les 128 tranches de profondeur) de la texture.

Lors du traitement de la tranche i, nous :

- activons le shader (qui applique le calcul de résolution) (en fait on l'active une seule fois au début)
- redirigeons la sortie du framebuffer vers la tranche i de la texture de sortie,
- dessinons un plan (dont les coordonnées de texture correspondent à la tranche i) et dont la texture est la texture d'entrée

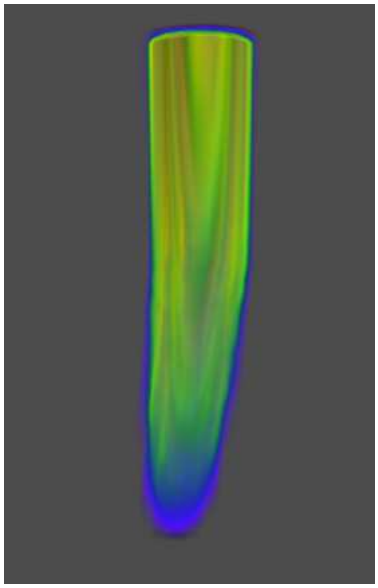
Ainsi en sortie de l'itération nous avons dans la seconde texture le résultat du shader appliqué à la première texture.

Remarque importante : Sur le papier cette astuce est très compréhensible. La mettre en place n'est pas si facile puisqu' un simple paramètre mal configuré entraine un résultat faux, et en trouver la cause n'est pas évident...

4.2 Détail de l'adaptation

Les matrices de gaz, fumée et chaleur sont maintenant stockées dans les couleurs RGB d'une texture3D. Les trois composantes de la vitesse sont aussi stockées dans les couleurs RGB d'une matrice3D. Les sources de densité et de vitesse sont aussi stockées dans des texture3D. Ainsi la première étape consiste à charger en mémoire les textures qui nous serviront par la suite.

Chaque étape de calcul a un shader associé. Le code des fragment shader est quasi identique à ce qu'on peut trouver à l'intérieur de la boucle des fonctions de la partie CPU. Il a fallut néanmoins rajouter des modificateurs de repères car le résultat final est stocké dans une texture3D, dont les composantes sont entre 0.0 et 1.0. Pour effectuer un calcul il faut charger le shader associé, lier les textures3D aux entrées et sorties du shader, lier les variables intervenant dans les calculs, et enfin appliquer le rendu dans le framebuffer.



Coloration de la flamme sur le modèle GPU.

4.3 Rendu

Au niveau du rendu, nous utilisons exactement le même principe que pour la partie CPU, à savoir un bruit de perlin et une coloration en fonction de la température. Cependant, nous avons transformé notre tableau de valeurs de coloration en texture, afin d'exploiter au mieux la capacité du shader à

manipuler les textures (la manipulation d'un tableau en shader étant horriblement lente). Notons que pour le rendu nous ne faisons plus de phase "matrice to texture3D" ce qui permet un **gain de temps important**.

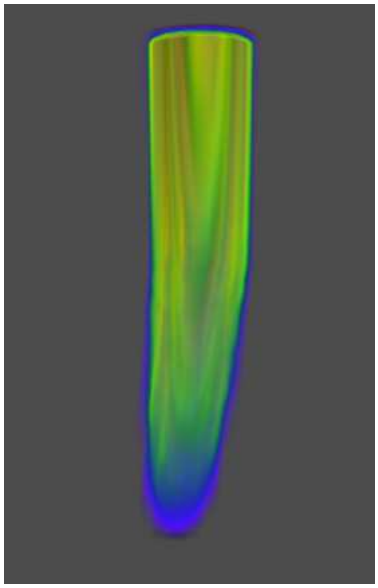
4.4 Tentative d'amélioration

Un framebuffer est capable de dessiner dans plusieurs tranches de la texture en même temps (en général 8). Par défaut il dessinera la même chose dans les 8 tranches. Nous avons donc créé un shader qui dessine dans les 8 tranches de sortie les 8 tranches de l'entrée correspondantes.
Malheureusement il n'y a aucun gain de temps, et les shaders sont un poil plus compliqués à réaliser. Nous avons donc enlevé cette méthode.

4.5 Résultats - Comparatif de performances

Les résultat en terme de perfomance sont plutôt bons. Cependant il faut bien avoir conscience qu'ils dépendent totalement de la carte graphique utilisée, contraitement au modèle CPU qui tourne à peut près à la même vitesse sur les processeurs actuels.

2x5600+, Geforce9800 CPU 20x20x20 : 9 fps CPU 30x30x30 :



Coloration de la flamme sur le modèle GPU.

4 fps CPU 40x40x40 : 2 fps CPU 50x50x50 : 1 fps CPU 60x60x60 : ¡ 1 fps CPU 70x70x70 : ¡ 1 fps

GPU 20x20x20 : 26 fps GPU 30x30x30 : 15 fps GPU 40x40x40 : 10 fps GPU 50x50x50 : 6 fps GPU 60x60x60 : 6 fps GPU 70x70x70 : GPU 100x100x100 :

Athlon QL-62, HD3200 CPU 20x20x20 : CPU 30x30x30 : CPU 40x40x40 : CPU 50x50x50 : CPU 60x60x60 : CPU 70x70x70 :

GPU 20x20x20 : GPU 30x30x30 : GPU 40x40x40 : GPU 50x50x50 : GPU 60x60x60 : GPU 70x70x70 : GPU 100x100x100 :

pi7, HD5730 CPU 20x20x20 : CPU 30x30x30 : CPU 40x40x40 : CPU 50x50x50 : CPU 60x60x60 : CPU 70x70x70 :

GPU 20x20x20 : GPU 30x30x30 : GPU 40x40x40 : GPU 50x50x50 : GPU 60x60x60 : GPU 70x70x70 : GPU 100x100x100 :

Benjamin CPU 20x20x20 : CPU 30x30x30 : CPU 40x40x40 : CPU 50x50x50 : CPU 60x60x60 : CPU 70x70x70 :

GPU 20x20x20 : GPU 30x30x30 : GPU 40x40x40 : GPU 50x50x50 : GPU 60x60x60 : GPU 70x70x70 : GPU 100x100x100 :

4.6 Conclusion

Le modèle GPU est donc totalement justifié. Son gain énorme de performance valide l'idée qu'un modèle de fluide réaliste peut être utilisé en temps réel avec une grande grille. Même si notre modèle CPU est interactif, la grille 40x40x40 et ses 8 fps est trop petite pour réaliser un logiciel de simulation. Cependant, notre version GPU nous montre que notre modèle de fluide peut être utilisé dans une application temps réel de grande envergure. Avec 8 fps en 100x100x100, dans une version pouvant encore être optimisée, nous pensons que tous les efforts fournis sur l'implémentation en GPU n'ont pas été vains.

5 Modèle d'objet

5.1 Détail du modèle

Nous utilisons une **grille de voxels** pour représenter les objets. Chaque voxel a plusieurs attributs : comme sa température de combustion, sa quantité de matière, son taux de combustion, son taux de conversion en gaz, son taux de conversion en fumée, sa température, sa vitesse de répulsion.
Nous avons choisi ce modèle car le fait de représenter les objets en voxels permet de faire le lien rapidement entre la grille de l'objet et la grille du fluide, avec une simple fonction prenant en compte la position des objets et la valeur d'une division de l'espace.

Chaque objet est initialisé avec un voxel de base. Nous pouvons donc jouer sur les propriétés de notre objet pour optenir tout un tas d'effet. Par exemple, nous pouvons réaliser des explosifs, des grenades fumigènes, des bout de bois qui brulent, des objets qui se décomposent sans bruler, des objets insensibles à la chaleur...

5.2 Rendu

Pour chaque grile de voxel, nous calculons son maillage avec un **marching cube adaptatif**. Nous stockons le maillage calculé dans des tableaux, ces tableaux étant associés aux voxels qu'ils représentent. Lors du rendu, nous parcourons la liste des triangles calculés et les affichons. Lorsqu'un voxel est modifié, nous recalculons le marching cube uniquement là ou la modification intervient. Ainsi nous disposons d' un modèle d'objets déformables dont le rendu est effectué de manière efficace.

5.3 Chargement des objets

Nous avons réalisé un **chargeur de fichier .obj**. A l'aide d'un **calculateur de champs de distance**, nous construisons l'équivalent de notre objet maillé en grille de voxel.

5.4 Principe d'intéraction avec le fluide

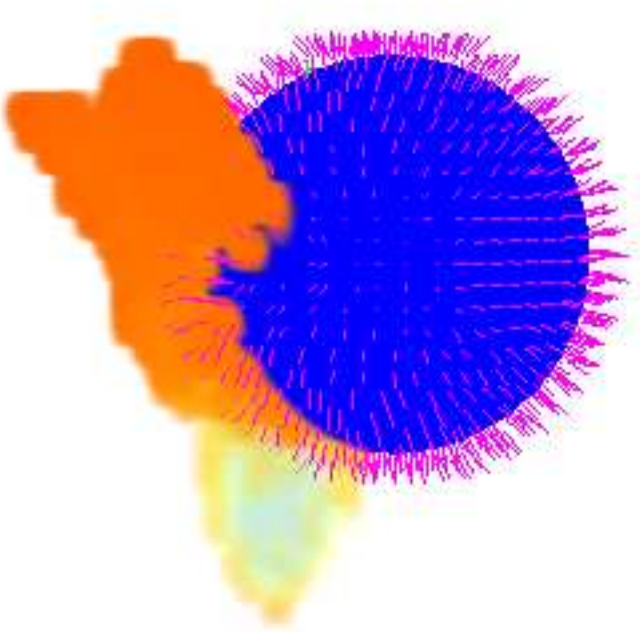
Concrètement, nous avons englobé notre fluide et nos objets par une **boite englobante**. Lorsqu'il y a intéraction, nous convertissons les coordonnées (i,j,k) de la grille de l'objet en coordonnées (i',j',k') dans le modèle de fluide. Si ces coordonnées sont valides, alors il y a intéraction du voxel avec le fluide.

5.5 Propagation de la chaleur au sein de l'objet

5.5.1 Champs de répulsion et déplacement de l'objet

Lorsqu'on place un objet dans le fluide, la vitesse du fluide est changée de deux manières :
La vitesse des voxels du fluide dans lesquels se trouve l'objet sont affecté : leur vitesse **devient égale à la vitesse du déplacement de l'objet**. Ainsi lorsqu'on déplace l'objet le fluide est emporté par son mouvement.
Les voxels se situant sur la frontière sont affecté d'un terme de **répulsion**.

Une première version du champs de répulsion consistait à calculer la normale aux surfaces de l'objet.



A gauche : affichage en marching cube de l'objet. A droite : affichage en voxel avec l'affichage des vecteurs vitesse de repulsions.

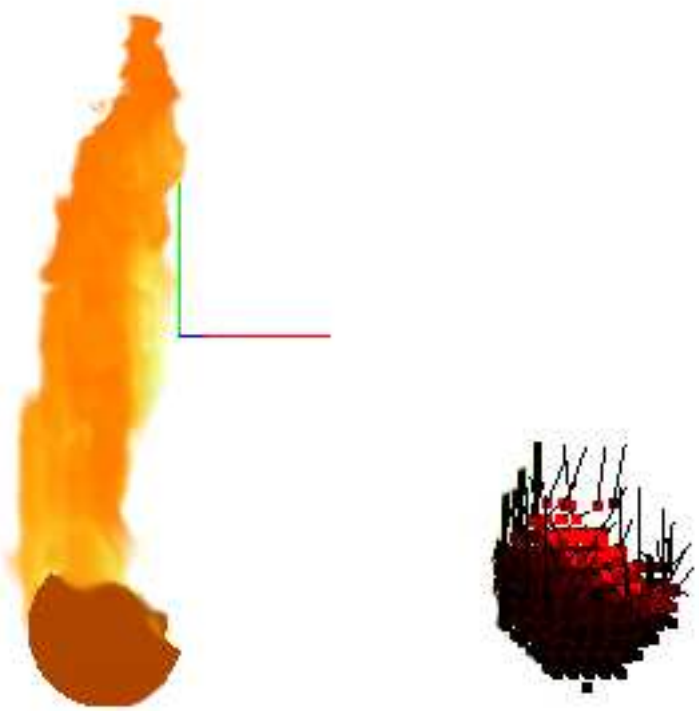
Puis nous avons opté pour une version "truquée" dans laquelle nous orientons les vitesses de répulsions de manière à ce que le fluide suive les contours de l'objet.

[IMAGE]

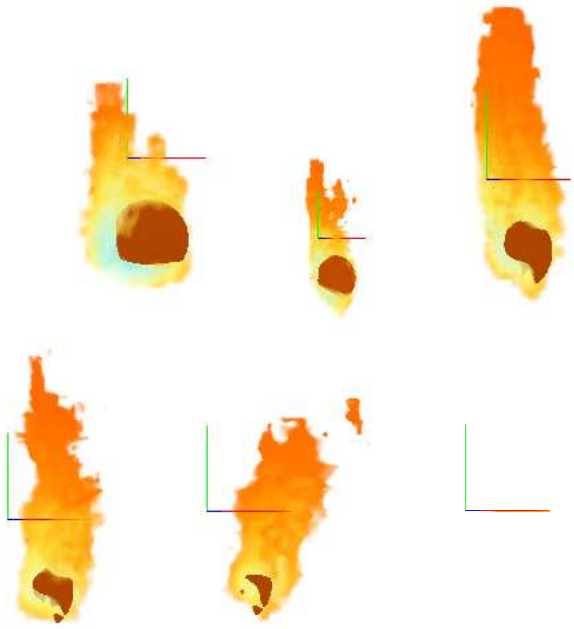
5.6 Pyrolise

Dans un premier temps, c'est la chaleur qui est échangé entre le fluide et le contour extérieur d'un objet.
Lorsque la température d'un voxel situé sur la frontière de l'objet dépasse un seuil critique, la pyrolise commence.
L'objet perds de la matière dans des proportions fixées par les paramètres du voxel. En effet, en fonction des paramètres, le voxel va generer plus ou moins de gaz, de fumée, et de température.

5.7 Combustion



A gauche : affichage en marching cube de l'objet. A droite : affichage en voxel avec l'affichage des vecteurs vitesse de repulsions.



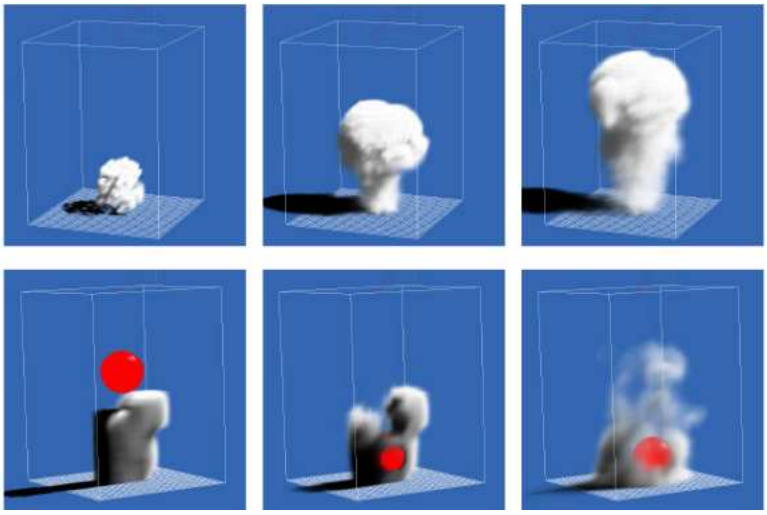
Combustion progressive de l'objet.

5.8 Résultats

6 Analyses Bibliographiques

Cette section regroupe les différents articles lus, concernant les travaux déjà effectués dans ce domaine. Nous allons expliquer brièvement de quoi ils parlent, ce que nous en avons retenu de bien ou de mal, ce que nous allons utiliser...

6.0.1 Real-Time Fluid Dynamics for Games

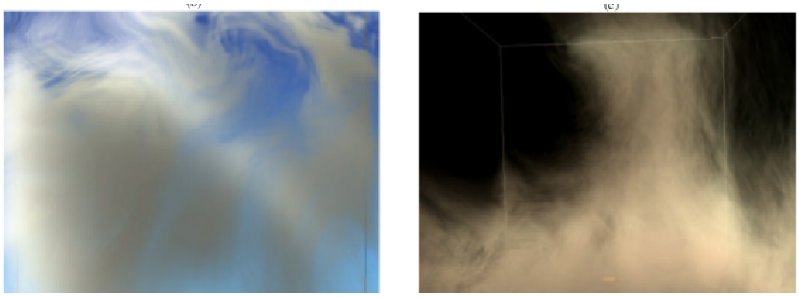


Auteur(s) : Jos Stam.
Publication : Proceedings of the Game Développer Conférence, March 2003
Sujet(s) abordé(s) :
Modèle de fluide temps réel, pouvant être utilisé pour le feu, la fumée
Le modèle gère : le déplacement du fluide, la gestion de combustible,les forces appliquées sur le fluide.
Principe :
Le modèle a été décomposé sur plusieurs étapes : génération du fluide par des sources, ajout des forces, diffusion du fluide, puis résolution de l'équation de la densité et de l'équation de la vitesse (équations de Navier-Stockes). Pour résoudre les deux équations, il utilise une astuce permettant de résoudre le système très rapidement. Enfin, il utilise le principe de conservation de la masse, qui rajoute des effets réalistes de vortex, avec la décomposition de Hodge.
Point(s) positif(s) :
- Ce modèle est un des plus connus dans le monde de la modélisation de fluide temps réel
- Temps réel.
- C'est facile à implémenter (moins de 100 lignes pour la version 2D)
- Peut être adapté pour la propagation du feu.
- Peut être adapté pour qu'un objet influe sur le modèle.
- Résultat réellement joli et réaliste.
- Très bien expliqué.
- Stam a fournit un exemple d'implémentation pour la version 2D pour de la fumée, plutôt impressionnant.
- Le modèle peut s'adapter tout types de fluides.

Point(s) négatif(s) :
- La zone d'influence est assez petite, il faut voir si c'est possible de l'étendre sans trop alourdir les calculs. (Octree ?)
- Pas trop de détails sur la représentation graphique du feu.
- La dissipation numérique implique que le résultat n'est pas exact.

Conclusion :
C'est sur ce modèle que nous avons construit notre modèle de fluide.

6.0.2 Stable Fluids



Auteur(s) : Jos Stam.
Publication : SIGGRAPH 99 Conference Proceedings
Sujet(s) abordé(s) :
Résolution de l'équation des fluides (Navier-Stockes) avec, pour la première fois, un algorithme inconditionnellement stable.
Principe :
C'est une résolution de Navier-Stockes orientée "Computer Graphic", dans le sens o'u la résolution n'est pas exacte, et ne serait pas utilisable pour des vraies simulations de fluides, mais est "graphiquement adaptée" au problème de fluides.
Résolution en quatre étapes : ajout des forces, advection, diffusion, conservation de la masse.
Point(s) positif(s) :
- 2D et 3D.
- Facile à implémenter.
- Résolution stable et temps réel.
- Résultat réaliste.

Point(s) négatif(s) :
- La dissipation numérique implique que le résultat n'est pas exact.

Conclusion :
Ce modèle de fluide semble le plus adapté si nous choisissons d'utiliser un modèle de fluide pour le feu, la fumée et la propagation.

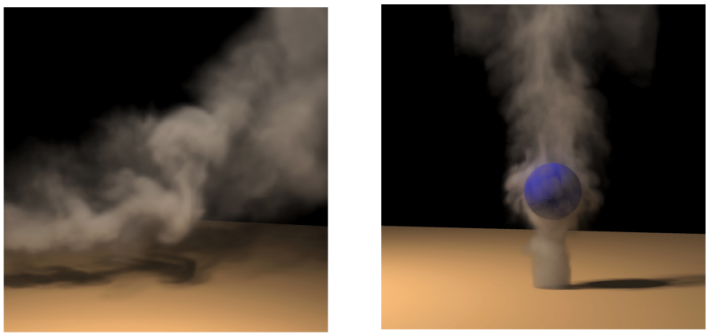
6.0.3 An Interactive Simulation Framework for Burning Objets

Auteur(s) : Zeki Melek, John Keyser.
Publication : Technical Report 2005 3 1, Texas AM Department of Computer Science, 2005.
Sujet(s) abordé(s) :
Premier modèle essayant de réaliser en même temps une propagation de feu réaliste avec un modèle de fluide, de la propagation sur un objet et entre les objets, et de la destruction d'objets.
Principe :
Le modèle de fluide utilise Stable Fluid de **Stam**.
Point(s) positif(s) :
- Temps réel
- Modèle de fluide, propagation, gestion des objets, destruction des objets.

Point(s) négatif(s) :
- L'implémentation CPU est lente (4fps pour une grille 20*20*20)

Conclusion :

6.0.4 Visual Simulation of Smoke



Auteur(s) : Ronald Fedkiw, Jos Stam, Henrik Wann Jensen.
Publication : SIGGRAPH 2001 Conference Proceedings
Sujet(s) abordé(s) :
Création d'un modèle de fumée, basé sur le travail de Stam (Stable Fluids).
Rendu réaliste de la fumée.
Principe :
L'équation du fluide est résolue comme dans "Stable Fluids". Cette fois la chaleur est prise en compte, et traitée de la même manière que la densité. Il en résulte une force de préssion qui s'ajoute simplement au modèle. Le modèle prend en compte les petits phénomènes de vortex qui se créent dans le fluide, basée sur la méthode de Steinhoff ("Vorticity confinement"). Ainsi ils rajoutent une force de rotation pour créer les minis vortex.
Pour le rendu : ils découpent la grille de voxels en plusieurs plans, rendus comme une superposition de plans transparents. Une autre méthode de rendu, plus réaliste, utilise une méthode de lancé de rayons.
Point(s) positif(s) :
- Rendu très réaliste de la fumée.
Point(s) négatif(s) :
- Pas en temps réel.
Conclusion :
La méthode de rendu semble interessante. De plus la création de minis vortex est un atout pour le côté réaliste du modèle.

6.0.5 Simulating Water and Smoke with an Octree Data Structure

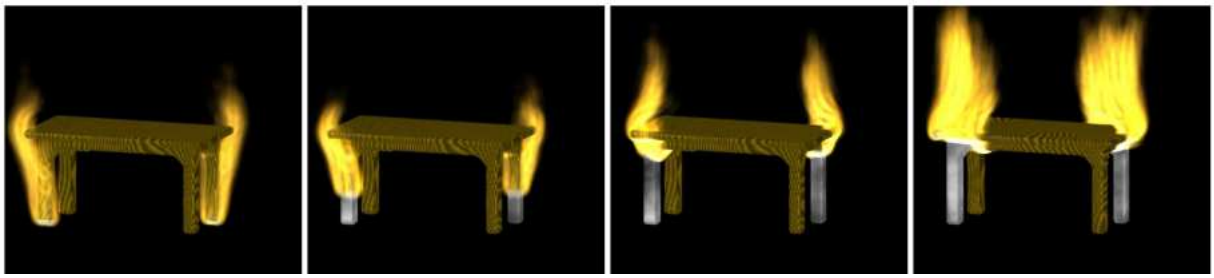
Auteur(s) : Frank Losasso, Frédéric Gibou, Ron Fedkiw.
Publication : SIGGRAPH 2004
Sujet(s) abordé(s) :
Simulation d'eau et de fumée. L'équation de Navier-Stockes est résolue sur une grille Octree.
La grille du modèle de fluide s'adapte selon le niveau de détail du phénomène (par exemple plus de détails là o'u il y a des minis vortex)
Principe :
Le calcul de l'équation des fluides est effectué sur un Octree.
Point(s) positif(s) :

- L'octree n'est pas restreint (ce qui était le cas des travaux précédents).
- Réduction des calculs de 75% pour un résultat équivalent avec une grille à pas constant.

Point(s) négatif(s) :
- Compliqué à mettre en place.

Conclusion :
La complexité de la programmation est assez importante. L'avantage de cette méthode est d'avoir un très haut niveau de détail en effectuant plus de calculs là o'u cela est nécessaire. Cependant nous comptons sur la version GPU pour gagner en fps.

6.0.6 Voxels On Fire

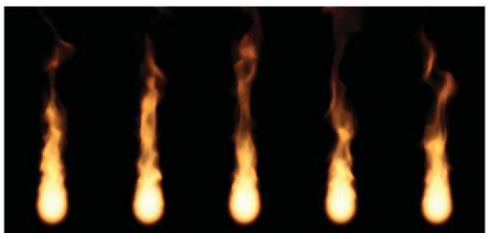


Auteur(s) : Ye Zhao, Xiaoming Wei, Zhe Fan, Arie Kaufman, Hong Qin.
Sujet(s) abordé(s) :
Animation et propagation de flammes, avec brulure de l'objet.
Principe :
L'animation et la propagation sont gérés par un modèle de fluide (Lattice Boltzmann Model), avec une grille régulière. L'objet est représenté sous forme de voxels, sur lequel est calculé un champs de distance. Le voxel de l'objet est considéré comme un combustible. Pour le rendu ils utilisent des particules et le splatting (**Westover**).
Point(s) positif(s) :
- temps réel (24fps pour une grille 64*64*64)
- propagation réaliste
- l'objet est brulé progressivement

Point(s) négatif(s) :
- le rendu utilise un grand nombre de particules pour être réaliste
- l'objet n'est pas détruit seule la texture est altérée

Conclusion :

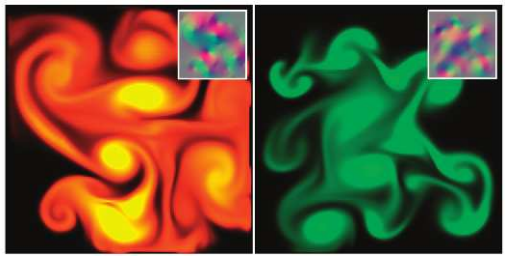
6.0.7 Real-Time Simulation and Rendering of 3D Fluids



Auteur(s) : Keenan Crane, Ignacio Llamas, Sarah Tariq.
Publication : GPU GEMS
Sujet(s) abordé(s) :
Résolution des équations de Navier-Stokes grace a la programmation GPU pour atteindre le temps réel. Les travaux étant réalisé par NVIDIA sur des cartes graphiques performantes et utilisant le shader langage GS ou CUDA.
La Technique de collision entre gaz et objets est aussi détaillée ainsi que la voxelisation de certains objets.

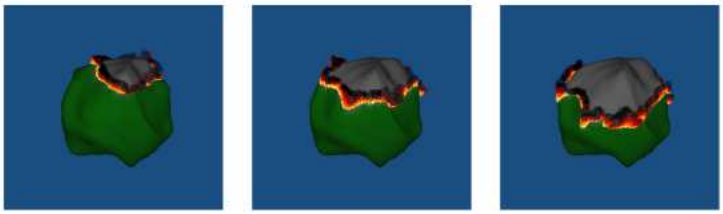
Principe :
Utilisation des langages de programmation utilisant les shaders GPU pour permettre un calcul bien plus rapide que par CPU et donc atteindre le temps réel bien plus facilement.
Point(s) positif(s) :
- Les temps de calcul atteint par les GPU sont néttement plus faibles que ceux du CPU. On gagne donc a traiter les equations de dynamique des fluides par la carte graphique et non par le processeur. Les equations de la dynamique des fluides sont traitées de la meme maniere que dans les différents documents de Stam vus précédement.
Point(s) negatif(s) :
- Nous n'allons pas utiliser le shading langage du document mais plutot le GLSL (openGL Shading Langage) qui est légérement différent dans la facon de coder. Il nous faut donc chercher un peu plus de documentation sur le codage GLSL.
Conclusion :
L'utilisation des shaders GPU semble être la clé pour nous permettre de réaliser quelquechose de réaliste en temps réel.

6.0.8 Fast Fluid Dynamics Simulation on the GPU



Auteur(s) : Mark J. Harris.
Publication : GPU GEMS
Sujet(s) abordé(s) :
Chapitre du livre GPU Gems de NVIDIA traitant d'une méthode de resolution des équations de "stable fluids" entierement réalisées sur GPU.
Principe :
Tous les calculs pour les champs de vitesses et de densités pour les fluides (diffusion, advection, forces, conservation de la masse) sont réalisé sur GPU pour augmenter la vitesse et la puissance de calcul.
Point(s) positif(s) :
- Les termes de l'équation de Navier stokes sont tous décortiqués et trés bien expliqués.
- Facilement Compréhensible et tous les calculs mathématiques sont bien détaillés.
- Modéle 2D pour que la compréhension soit facile.
- Prise en compte d'un terme de pression.
- Utilisation de la décomposition de Helmholtz-Hodge
- Exemple de programmation GPU (type CUDA ou GS)
Point(s) negatif(s) :
- Toujours pas de "cours" GLSL.
Conclusion :
Document qui peut servir de référence pour analyser et comprendre les equations de la dynamique des fluides et pour initier les bases de la programmation GPU.

6.0.9 Meshes On Fire



Auteur(s) : Haeyoung Lee, Laehyun, Mark Meyer, Mathieu Desbrun.
Publication : Eurographics Workshop on Computer Animation and Simulation '2001
Sujet(s) abordé(s) :
Propagation des flammes à la surface d'un objet.
Principe :
La propagation est un parcours géodésique de la surface, et subit le vent environnant.
Les flammes sont rendues sont formes de particules avec des blobs.
Les flammes générées subissent le champ de vent ce qui les rend plus réaliste.
Point(s) positif(s) :
- Temps réel.
- Prise en compte de plusieurs feu.
- Rapide à calculer.
- Très efficace pour changer la texture d'un objet brulé.
- Propagation fonction d'un champ de vent.
- Les particules de feu générées peuvent être utilisées pour générer un feu sur un autre objet ou sur une autre partie de l'objet.

Point(s) négatif(s) :
- Ne peut pas s'adapter à la destruction d'un objet.
- Peut être une perte de performance si on utilise les particules de feu pour allumer des foyer aux autres endroits du mesh.
- La propagation surfacique n'est pas toujours adaptée.

Conclusion :
Ce modèle est intéressant, mais nécessite d'être beaucoup modifié pour correspondre à notre but. Il n'est pas très adapté à la destruction des objets. (Sauf si nous voulons uniquement dégrader la texture de l'objet)

6.0.10 Real-Time Simulation of Deformation and Fracture of Stiff Materials

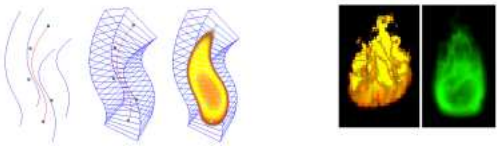
Auteur(s) : Matthias Müller, Leonard McMillan, Julie Dorsey, Robert Jagnow.
Publication : EUROGRAPHICS 2001 Computer Animation and Simulation Workshop
Sujet(s) abordé(s) :
Destruction réaliste et temps réel d'un objet.
Principe :
C'est la simplification d'un problème de propagation en négligeant les effets microscopiques, qui ne sont pas vraiment visibles en temps réel, mais coûtent énormément en calcul.
Les meshs sont représentés par des "tetrahedral meshes". Le modèle de propagation continu est transformé en modèle discret. L'élasticité des objets est prise en compte.
Point(s) positif(s) :

- Temps réel.
- Le système est stable est rapide.
- La méthode de destruction de l'objet basée sur des tétrahèdres est très intéressante.
- Propagation des effets de la destruction à l'intérieur de l'objet.

Point(s) négatif(s) :
- Peut être lourd à utiliser si il y a trop d'objets à l'écran.

Conclusion :
Le principe de déformation sur un mesh tétrahédral est très intéressant, et peut facilement s'adapter à un modèle de fluide.

6.0.11 Real-time Procedural Volumetric Fire

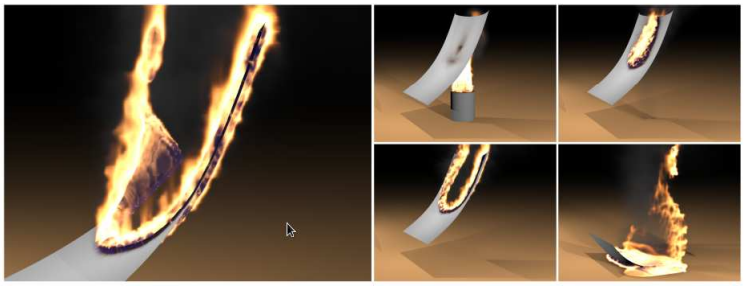


Auteur(s) : Alfred R. Fuller, Hari Krishnan, Karim Mahrous, Bernd Hamann, Kenneth I. Joy.
Publication : Proceedings of the 2007 symposium on Interactive 3D graphics and games
Sujet(s) abordé(s) :
Rendu de feu réaliste en temps réel, utilisant le bruit de perlin.
Principe :
La flamme est calculée en 3 dimensions, et le rendu est effectué par le GPU.
L'animation de la texture est procédurale.
Point(s) positif(s) :
- Temps réel.
- Réaliste.

Point(s) négatif(s) :
- Représentation du feu uniquement, pas de propagation et de fumée.
- Pas d'utilisation d'équation de dynamique des fluides, le feu est manuellement - confiné dans un volume controlable grace à des points de controle.

Conclusion :
La qualité du feu est importante, mais ce modèle de flamme ne concerne que la partie "affichage". Il faudra voir si le modèle de propagation choisi peut utiliser un tel modèle de flammes.

6.0.12 Melting and Burning Solids into Liquids and Gases



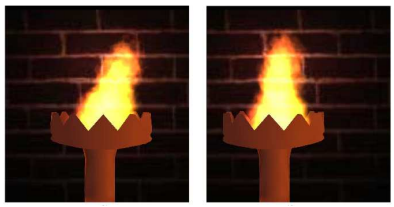
Auteur(s) : Frank Losasso, Geoffrey Irving, Eran Guendelman, Ron Fedkiw.
Publication : IEEE TVCG 12, 343-352 (2006).
Sujet(s) abordé(s) :
Modification des propriétés physiques d'un matériau tel que l'eau sous différentes phases, et intéraction entre les solides et les liquides et gas.
Point(s) négatif(s) :
- Peu de détails mathématiques, juste des explications.
- Pas assez de lien avec le feu et la fumée.

Conclusion :
Peu utilisable

6.0.13 FireStarter – A Real-Time Fire Simulator

Auteur(s) : Marc de Kruijf.
Sujet(s) abordé(s) :
Description du feu réel, des différents modèles utilisés, et réalisation d'un modèle avec des particules.
Principe :
Le feu est un système de particules, les flammes sont générées avec une vie aléatoire, une vitesse verticale dépendant de la chaleur, et une vitesse horizontale aléatoire. De plus, un test aléatoire fonction de la durée de vie de la particule est effectué pour qu'elle se transforme en fumée. Le tout est effectué dans une grille cylindrique 3D. Au niveau de l'affichage, les particules sont reliées pour former des faces qui seront ensuite texturées et colorées en fonction de la chaleur.
Point(s) positif(s) :
- Temps réel.
- Calcul très rapide et très simple.
Point(s) négatif(s) :
- Pas très réaliste actuellement.
- De gros défauts sont visibles au niveau du rendu.
- Les particules suivent une pseudo mécanique des fluides, qui n'est pas détaillée ici.
- Le feu n'est pas adapté à plusieurs cas de figures, il faudrait un système de génération des générateurs de particules, ce qui n'est pas forcement permis par le système.
- Pas de prise en compte des objets (à moins d'ajouter des détections de collisions)
Conclusion :
La bonne idée de transformer une particule de feu en fumée à partir d'un tirage aléatoire est une bonne idée. Ce modèle permet de représenter du feu d'une manière qui pourrait être suffisamment réaliste pour ne pas se lancer dans une résolution de mécanique du fluide.

6.0.14 Simulating Fire With Texture Splats



Auteur(s) : Xiaoming Wei, Wei Li, Klaus Mueller1 and Arie Kaufman.
Publication : IEEE Visualization 2002.
Sujet(s) abordé(s) : Méthode d'affichage de feu avec des particules et une texture "Splat"
Principe : C'est une méthode de rendu efficace adaptée aux particules.
Conclusion : Nous n'utiliserons pas cette technique, car nous avons trouvé des procédés plus adaptés pour le modèle de fluide.

6.0.15 Physically Based Modeling and Animation of Fire

Auteur(s) : Duc Quang Nguyen, Ronald Fedkiw, Henrik Wann Jensen.
Publication :
Sujet(s) abordé(s) :
Modélisation du feu comme le comportement d'un fluide suivant les équation de Navier-Stokes de la dynamique des fluide.
Principe :
De même que les articles de Stam, Le principe est de considérer le feu comme un fluide auquel on applique les équations de Navier Stokes pour les fluides incompressibles. On prend aussi en compte la densité du fluide mais aussi l'impact de la température sur celui ci. Cela permet d'avoir un champ de mouvement pour le fluide qui réagit avec la température. De plus l'utilisation de la température pour ce modéle permet d'être utilisé pour le rendu en considérant le feu comme un "corps noir" et de déterminer la coleur de la flamme grace au modéle du corps noir de planck en utilisant un spectre de couleur adapté.
Point(s) positif(s) :
- Plus de réalisme dans le modéle grace à l'impact de la température pour l'advection et la gestion du spectre lumineux du feu.
Point(s) negatif(s) :
- rajoute du temps de calcul pour la prise en compte de la température.
Conclusion :
Certains points comme la température peuvent être à rajouter pour un effet de réalisme intéressant.

6.0.16 Adaptative Physics Based Tetrahedral Mesh Generation Using Level Sets

Auteur(s) : Robert Bridson, Joseph Teran, Neil Molino, Ronald Fedkiw.
Publication : Engineering with Computers (2005)
Sujet(s) abordé(s) :
Présentation d'un algorithme de génération de Mesh Tetrahédral, dont l'entrée est une fonction distance, une grille cartésienne ou un octree.