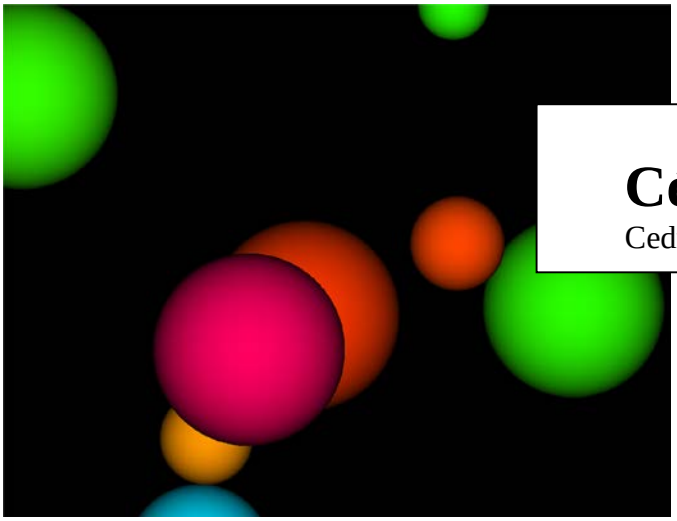
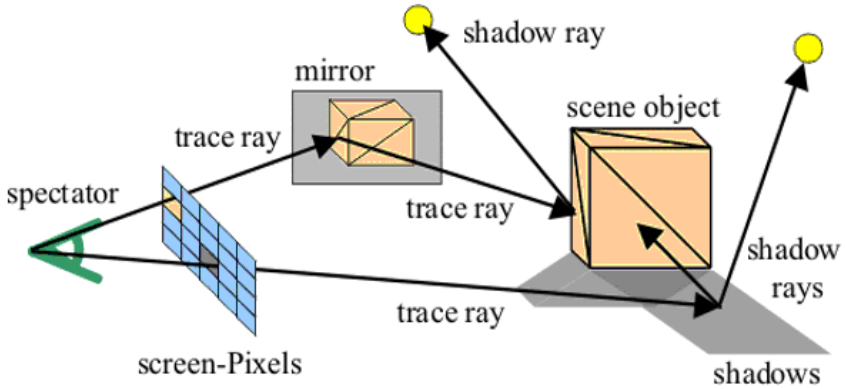
Problèmes Parallelisation



By Professeur
Cédric Bilat
Cedric.Bilat@he-arc.ch



Ray Tracing

Contexte

Contexte

Soit $S \subset \mathbb{R}^3$ une sphère colorié (HSB) de rayon *r* et de centre (C_x, C_y, C_z). Soit U un univers remplie d'un ensemble E de sphères. Soit $(x, y) \in \mathbb{R}^2$ un pixel de l'image représentant la scène 3D.

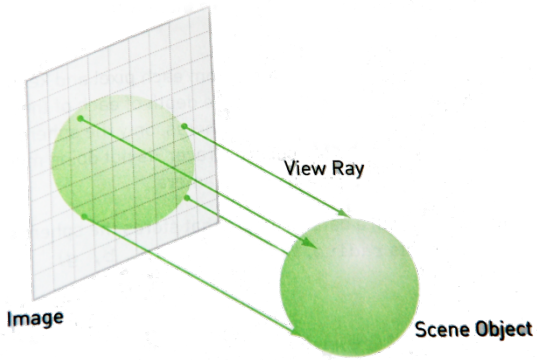
Problème :

Couleur de (x, y) ?

Solution :

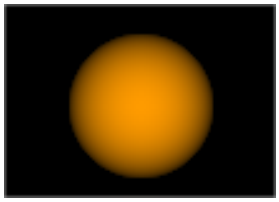
Naïve : *couleur uniforme*

Trouver la sphère S la plus proche du pixel (x, y).
Colorier (x, y) en fonction de la couleur de la sphère S.



Mieux : *couleur smoothing*

Pour avoir un effet visuel plus réaliste, la sphère ne doit pas avoir une couleur uniforme, mais une couleur dont l'intensité dépend de la distance au pixel (x, y). On aimerait que le centre de la sphère soit plus lumineux et que ces bords soient plus sombres. On va jouer avec le paramètre $B \in [0,1]$ B de la couleur HSB. On utilisera $B = 1$ pour le point correspondant au centre de la sphère et $B = 0$ pour les points se trouvant sur le bord. Une décroissance linéaire sera utilisée entre les 2.



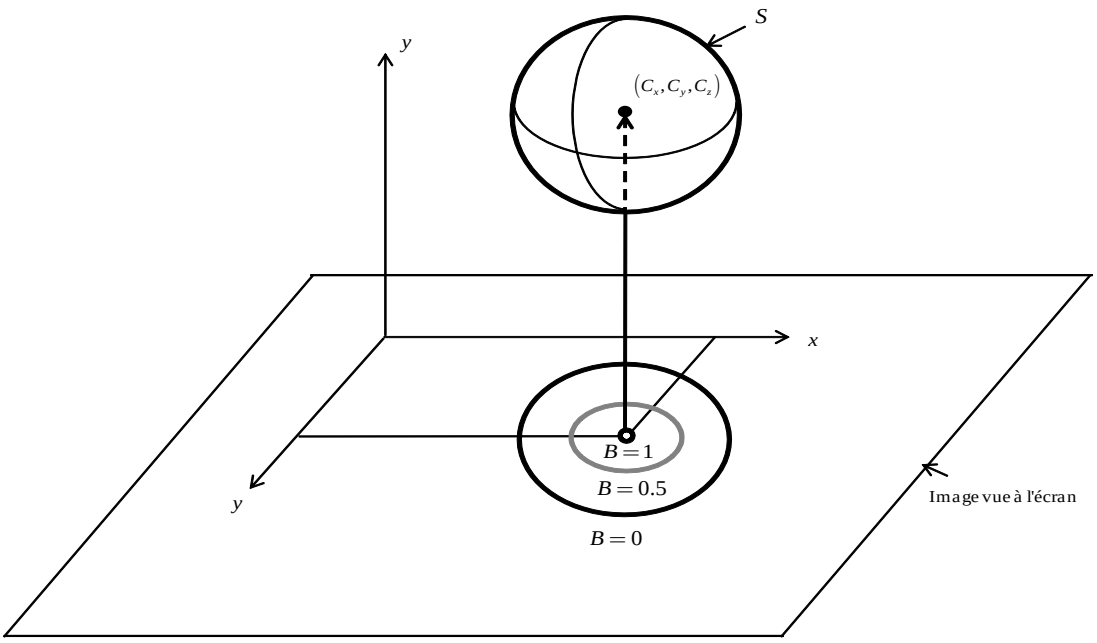


Figure 1: schéma solution

Observation

On se contente ici d'un modèle qui travaille en perspective cavalière. L'accent est mis sur la *programmation parallèle* et non sur la mise au point d'un modèle réaliste de *ray tracing.* En *perspective cavalière* un objet lointain de taille identique à un objet proche aura lors du rendu la même taille à l'écran, ce qui peut donner des effets « non désirables »

Math

Algorithme distance

La seule difficulté est de savoir calculer la distance z de notre pixel (x, y) et une sphère S de rayon r et de centre (C_x, C_y, C_z).

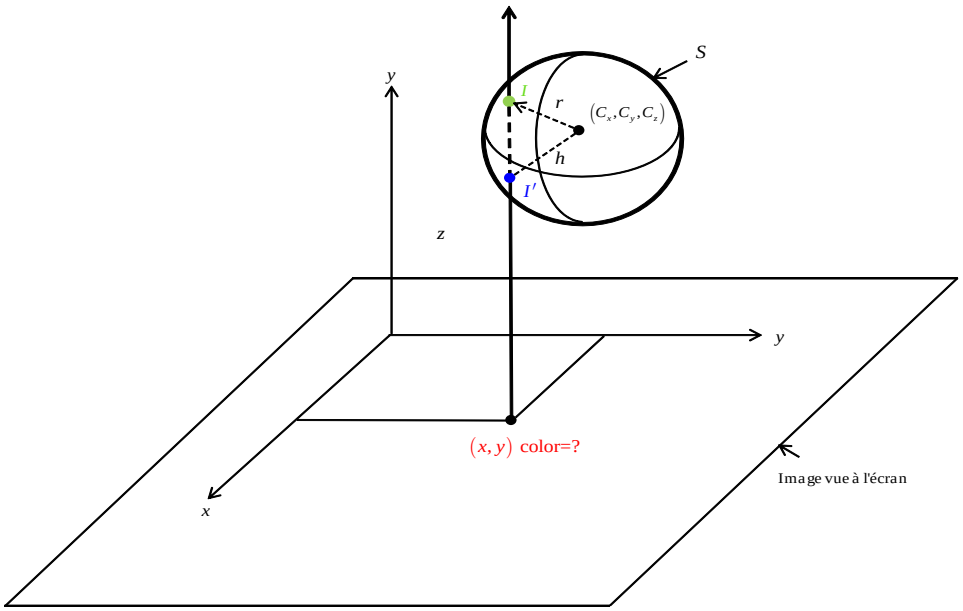


Figure 2 Algorithme distance

On cherche la distance z du pixel (x, y) à la sphère S.

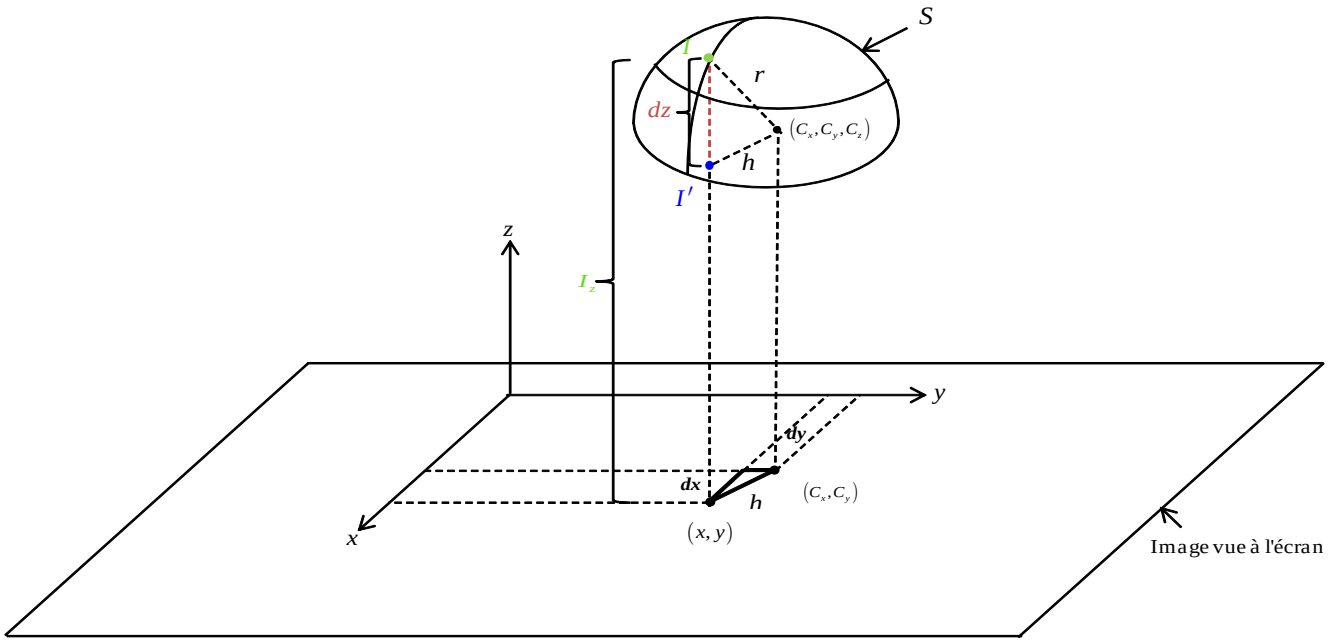


Figure 3 : calcul de la distance z du pixel

Par Pythagore, en utilisant le triangle sur le sol, on a :

$$\boxed{h^2 = (c_x - x)^2 + (c_y - y)^2}$$

Par Pythagore, en utilisant le triangle sur la sphère, on a :

$$r^2 = dz^2 + h^2$$

En regroupant les deux on obtient :

$$\boxed{dz = \sqrt{r^2 - h^2}}$$

Finalement le schéma nous laisse pensait que:

$$I_z = c_z + dz$$

Mais attention, en pratique

$$\boxed{I_z = c_z - dz}$$

En effet, on cherche le point de la sphère le plus proche du sol, donc non pas I mais I' (voir Figure 2) :

$$\boxed{I' = c_z - \sqrt{r^2 - h^2}} \quad (\#)$$

Algorithme color smoothing

Version 1 :

Le B de HSB doit varier dans [0,1].

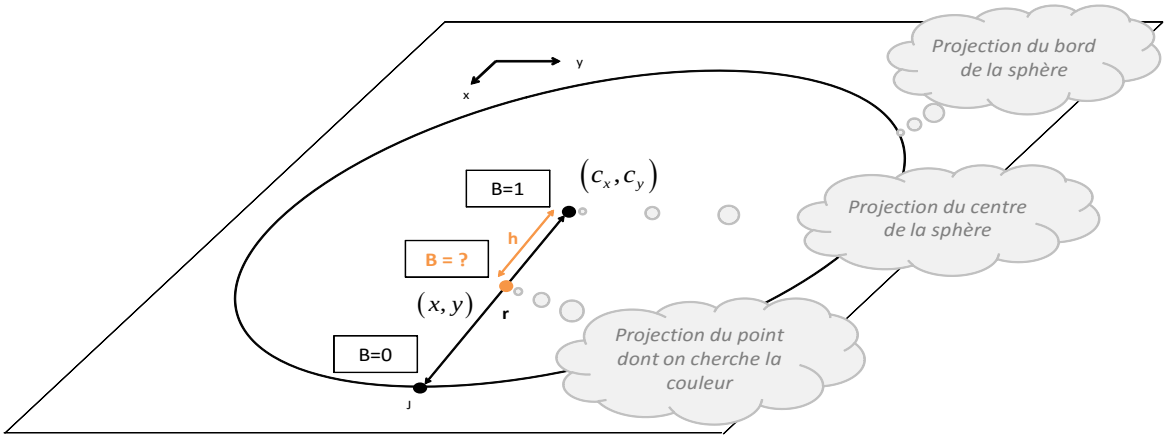
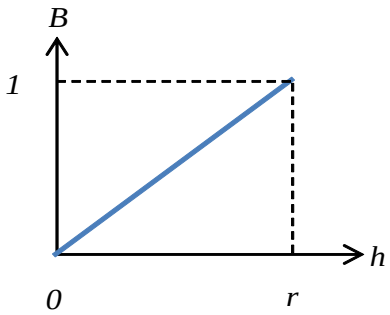


Figure 4 : calcul de B

On calcule B au pro rata de la position (x, y) entre (c_x, c_y) et J.

$$h = h(x, y) = \sqrt{(c_x - x)^2 + (c_y - y)^2}$$



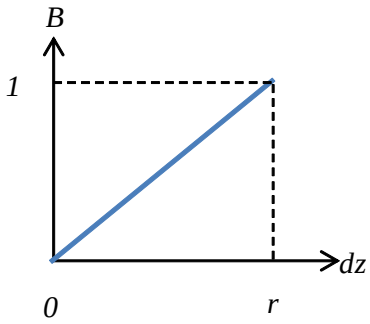
$$B = B(h) = \frac{1}{r}h$$ (de HS**B**)

appication linéaire

Version 2 :

Au lieu de calculer $B = B(h)$ on va calculer $B = B(dz)$. Cette version est peut-être plus performante que la précédente, car *dz* a déjà été calculée par l'algorithme de la distance. On observe que sur la figure 3 que :

- Quand h vaut r, $dz = 0$.
- Quand h vaut 0, $dz = r$.



$$\boxed{B = B(h) = \frac{1}{r}h}$$ (de HS**B**)

appication linéaire

Conclusion :

La couleur de (x, y) est HSB avec :

H	=	celui de la sphère
S	=	1
B	=	$\frac{1}{r}dz$

Algorithme positionnement

Un point $P(x, y)$ de l'image est en dessous d'une sphère de centre $C(cx, cy, cz)$ et de rayon r ssi

$$\boxed{\text{distance}(P, C) < r}$$

Par Pythagore (au carré) :

$$\boxed{P(x, y) \text{ en dessous sphère ssi } h^2(x, y) < r^2}$$

Implémentation

On utilisera la structure à 3 champs (x,y,z) de *float3* de Cuda, pour stocker le centre d'une sphère. Les propriétés de la sphère (centre, rayon, couleur, …) seront encapsulé dans la classe *Sphere*. Cette classe possédera aussi les formules de math présentées précédemment dans ce document.

Sphere

```
#ifndef SPHERE_H
#define SPHERE_H

#include "cudaTools.h"

#ifndef PI
#define PI 3.141592653589793f
#endif

class Sphere
      {
      public:
            __host__
            Sphere(float3 centre, float rayon, float hue)
                  {
                  // Inputs
                  this->centre = centre;
                  this->r = rayon;
                  this->hue = hue;
                  // Tools
                  this->rCarre = rayon * rayon;
                  }
```

```
/**
 * required by example for  new Sphere[n]
 */
__host__
Sphere()
      {
      // rien
      }

__device__
float hCarre(float2 xySol)
    {
   float a = (centre.x - xySol.x);
   float b = (centre.y - xySol.y);
    return a * a + b * b;
    }

__device__
bool isEnDessous(float hCarre)
    {
    return hCarre < rCarre;
    }

__device__
float dz(float hCarre)
    {
    return sqrtf(rCarre - hCarre);
    }

__device__
float brightness(float dz)
    {
    return dz / r;
    }

__device__
float distance(float dz)
    {
    return centre.z - dz;
    }
```

```
            __device__
            float getHueStart()
                {
                return hueStart;
                }

            __device__
            float hue(float t)// usefull for animation
                {
                return 0.5 + 0.5 * sin(t + T + 3 * PI / 2);
                }

    private:
            // Inputs
            float r;
            float3 centre;
            float hueStart;

            // Tools
            float rCarre;
            float T ; // usefull for animation
    };

#endif
```

Sphere.h

La classe *Sphere* sera

(E1) Instancier sur le host

(E2) Copier sur la mémoire du *Device* avec une action de memory management de type
- ***cudaMalloc***
- ***cudaMemcpy***

Note : Comme la classe *Sphère* ne possède pas d'attribut de type pointeur, elle occupe une suite de byte continue en mémoire, elle peut sans autre être transférer octet par octet en connaissant l'adresse de départ et la taille.

Séquence d'utilisation des formules :

Afin d'effectuer le moins de calcul redondant, il est proposé d'utiliser les formules dans l'ordre suivant :

1. Calculer h^2
2. Calculer *isEndessous*
3. Calculer *dz*
4. Calculer *distance*
5. Calculer *brightnesse*

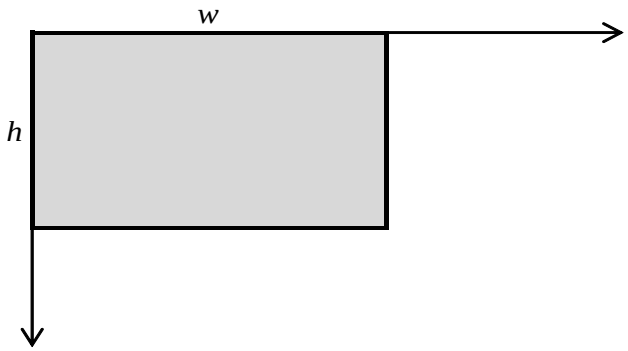
API Image

Utiliser la version *bitmap* de l'API Image fournie. Il n'est pas nécessaire ici d'utiliser la version *fonctionnelle*.

Point de départ

Contexte :

Notre scène 3D va être représentée par une image $[0, w[\times [0, h[$



Chaque points de l'image représente soit

- Un point projeté d'une sphère
- Rien (le noir de l'univers)

Version : *Image Bitmap*

En utilisant une image bitmap, les coordonnées x,y des centres des sphères de notre univers doivent se trouver dans $[0, w[\times [0, h[$. On les tirera aléatoirement en entier, mais seront stocké en *float*.

Version : *Image Fonctionnelle*

En utilisant une image fonctionnelle, on peut spécifier un domaine mathématique sur l'image. Donc les centres des sphères de notre univers doivent se trouver dans $[x_0, x_1[\times [y_0, y_1[$. Ces valeurs seront tirées aléatoirement en float.

Proposition :

On tire 50 sphères aléatoirement avec :

- rayon $\in [20, w/10[$
- x $\in [0 + bord, h - bord[$
- y $\in [bord, w - bord[$
- z $\in [10, 2w[$
- hue $\in [0,1]$

avec $bord = 200$. Il est laissé au lecteur de trouver un point de départ plus intéressant!

Indications Cuda

Il serait intéressant d'effectuer les trois implémentations suivantes :

- Mettre les sphères en *Global Memory*
- Mettre les sphères en *Constant Memory*
- Mettre les sphères en *Shared Memory*

et comparer les différences de performances.

CM

Tip 1 *Limite Mémoire*

Pour remplir complètement la *constant Memory* qui est de **64 Ko**, il faut tirer environ entre 2000 et 3000 sphères selon ce que vous mettez dans la classe *Spheres* !

$$nbSphereMax = \frac{1024 * size(CM)}{sizeof(Sphere)}$$

Pour les cartes de génération *Maxwell*, on a normalement

$$size(CM) = 64\ Ko$$

Le poids d'une *Sphere* est d'environ 28 Octets, ce qui laisse la place pour environ

2300 sphères

Parcimonie

En cas de débordement de la SM, le code plante et aucun message d'erreur pertinent ne vous indique la raison. Commencez donc pas mettre peu de *Sphere* pour valider votre code, puis augmentez !

Fausse Amélioration

Sortez de la classe *Sphere* les méthodes, pour que celle-ci ne contiennent que des données ! Vous pourrez ainsi mettre beaucoup plus de *Sphere* en *Constant Memory CM* ! Non, le code est stocké ailleurs !

SM

Tip 1 *Copie Octet par Octet*

Au lieu de copier

« sphere par sphere »

essayer de copier

« octets par octets »

en castant vos tableaux

Sphere* tabSphereGM
Sphere* tabSM

en

int* tabIntGM
int* tabIntSM

Copier ensuite le tableau de *int*. Attention d'itérer le bon nombre de fois !

Tip 2 *Deobjeification*

Réaliser une implémentation sans objet *Sphere*.
Travailler à la place avec plusieurs tableaux :

float* tabRayon
float3* tabCentre
…

Animation

Objectif :

(O1) On aimerait toujours pouvoir utiliser la *constant memory* pour comparer ses performances par rapport à la *global memory* et par rapport à la *share memory*.

(O2) On aimerait que l'animation soit continue, c'est-à-dire sans saut ni à-coups.

Principe :

La géométrie des sphères reste fixe.
Seule la couleur va varier au cours du temps.

Algorithme :

La couleur d'une sphère sera donné comme initialement en *HSB*. Sa couleur dépendra de la couleur de départ *hStart* et du temps écoulé t.

$$\text{hueSphere}(t) = h(hStart, t)$$

Contraintes fonction h :

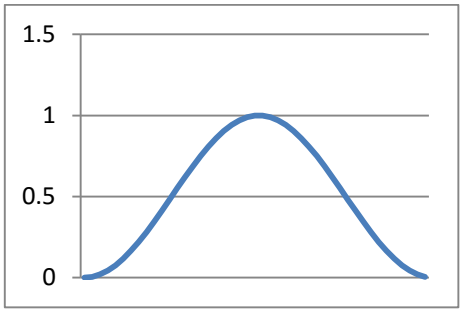
Tout est permis, mais la fonction *h* doit donner des *hue* entre $[0,1]$. Par ailleurs pour la beauté de l'animation, la variation du hue doit être continue.

Construction fonction h : *Proposition*

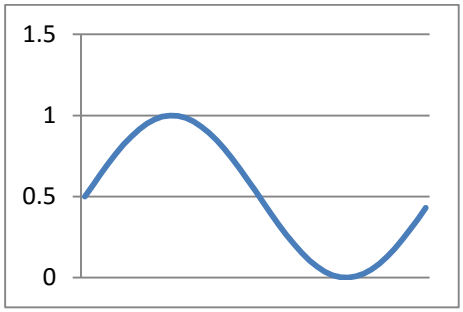
Travaillons avec les quatre cas représentatif suivants :

$$hStart = 0 \quad \in [0,1]$$
$$hStart = 0.5 \quad \in [0,1]$$
$$hStart = 0.75 \quad \in [0,1]$$
$$hStart = 1 \quad \in [0,1]$$

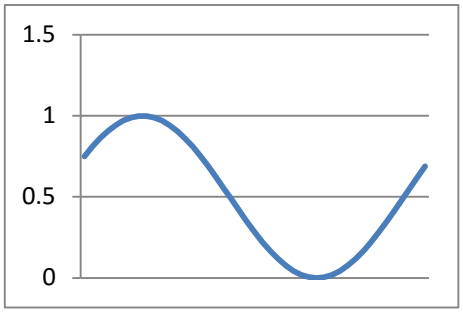
La continuité et périodicité de la fonction *h* recherché sera amené par un *sinus* d'une amplitude $1/2$ et déphasé comme suit :



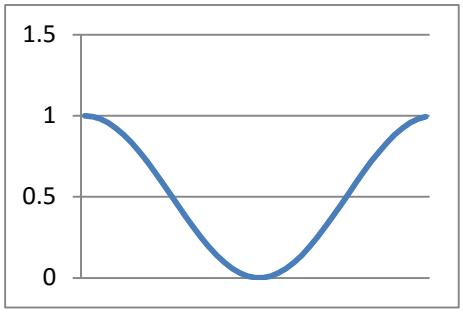
hStart = 0



hStart=0.5



hStart = 0.75



hStart=1

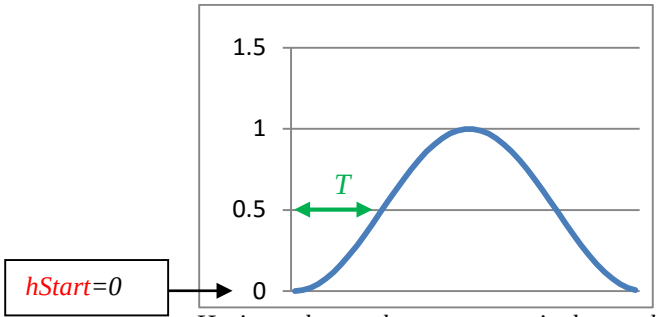
On observe sur les graphes ci-dessus que :

- *h* varie entre 0 et 1.
- *h* est continue.
- *h* est périodique (nécessaire pour une animation).

Construction analytique de h :

Etape 1 : Cas simple *h=0* :

L'objectif est de trouver la fonction *hue* ci-dessous en fonction du temps *t*:



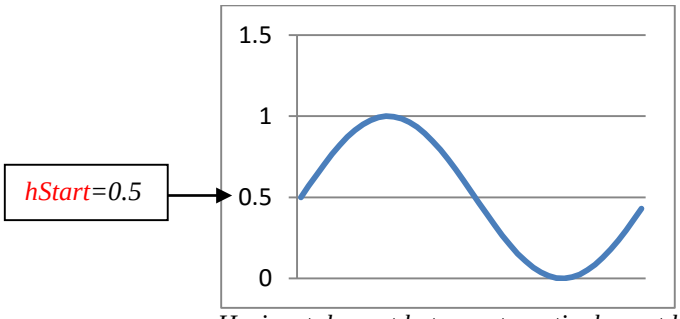
Horizontalement le temps t, verticalement le hue h.

Il est trivial que la fonction ci-dessus est donné par :

$$f(t)=\frac{1}{2}+\frac{1}{2}\sin\left(t+\frac{3\pi}{2}\right) \qquad (*)$$

Etape 2 : **C**as général *h=0.5* (par exemple) :

L'objectif est de trouver le *hue* ci-dessous en fonction du temps *t*:

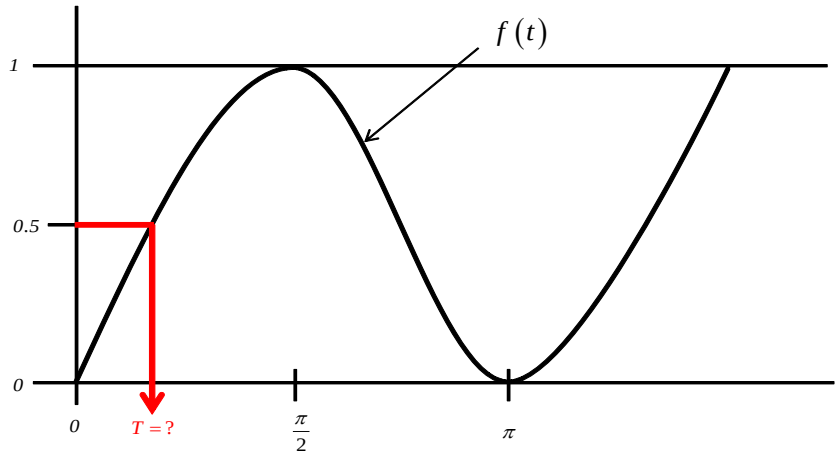


Horizontalement le temps t, verticalement le hue h.

Pour obtenir ce graph, il suffit de translater le graph $f(t)$ mise au point à l'étape 1, de T.
Le *hue* est

$$h(t)=f(t+T)$$

Pour obtenir T on observe



$$f(T) = hStart \quad ssi \quad \frac{1}{2} + \frac{1}{2}\sin\left(T + \frac{3\pi}{2}\right) = hStart$$

$$ssi \quad T = \arcsin(2 * hStart - 1) - \frac{3\pi}{2} \in [0,1]$$

Attention, $f(t)$ n'étant pas linéaire mais trigonométrique $T \neq \frac{\pi / 2}{2}$, en effet la réduction n'a aucune raison d'être proportionnelle.

Conclusion :

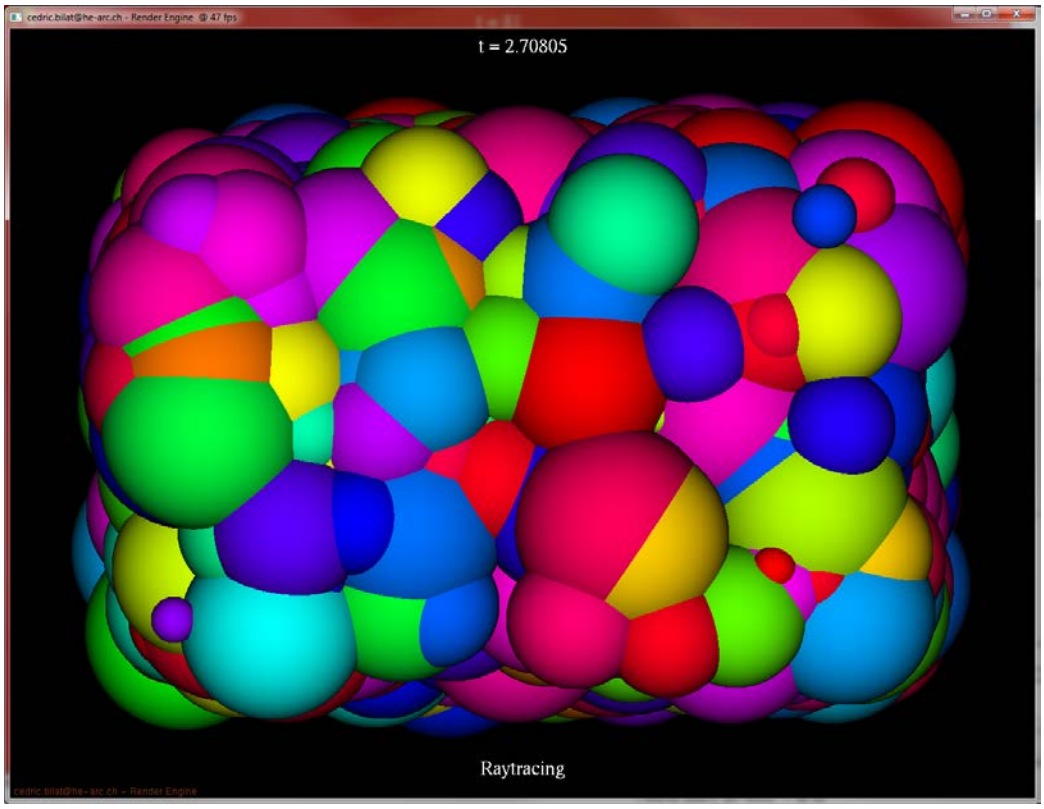
$$hueSphere(t, hStart) = \frac{1}{2} + \frac{1}{2}\sin\left(t + \frac{3\pi}{2} + T\right)$$

avec

$$T(hStart) = \arcsin(2 * hStart - 1) - \frac{3\pi}{2}$$

On fera varier au cours du temps t par *dt* , *avec* un *dt* petit. Le choix de celui-ci est laissé au lecteur!

Annexe



End