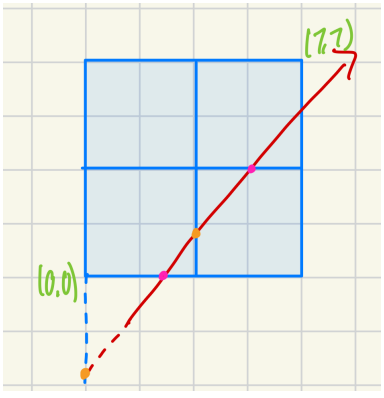
21-10-2024

Samuel Fournier

October 2024

Dans les deux dernières semaines, je n'ai pas eu la chance de travailler autant que j'aurais voulu sur le projet (devoirs, exames et autres...). Par contre, j'ai réussi à réparer certains bug avec le code et j'ai implémenter ma première version de *Ray Marching*.
J'ai passé la semaine du 7 octobre à restructuré mon projet et tenter de faire un gros *cleanse* de mon code. À force de travailler, j'ai rapidement remarqué que le code que j'écrivais laissait un peu à désirer. Les changements principaux sont dans **Medium.h** et **Medium.cpp**. J'ai enlevé toutes les définitions de fonction de **Medium.h** et je les ai mis dans **Medium.cpp** (je vais remettre certaines fonctions dans **Medium.h** comme le constructeur par exemple). J'ai ajouté le **enum TraversalType** pour me permettre de facilement définir quel type d'algorithme de parcours le code doit utiliser. Finalement, j'ai revisiter mon code pour l'algorithme du *DDA*. Mon implémentation s'inspirait de celle de Amanatides & Woo, mais il y avait un concept que je n'avais pas été en mesure de comprendre (le tDeltaX, Y, Z), ce qui m'a forcé à faire ma propre implémentation. Par contre, mon implémentaion avait deux grands problèmes: elle était **TRÈS** lente et elle ne fonctionnait pas la moitié du temps (j'exagère un peu, mais c'est quand même vrai). Grâce à des ressources en lignes et des brouillons d'équation au tableau blanc, j'ai finalement compris ce que le tDelta était.
Supposons un rayon:



Le rayon possède une direction positive en x et en y. L'équation du rayon est:

$$\begin{aligned} r(t) &= \vec{O} + t * \vec{d} \\ &= \begin{cases} r_x(t) = \vec{O_x} + t * \vec{d_x} \\ r_y(t) = \vec{O_y} + t * \vec{d_y} \end{cases} \end{aligned}$$

Le tDelta représente la quantité que t doit changer pour passer d'un plan à un autre. Par exemple, si on observe les points roses sur le dessin on obtient les deux équations suivantes:

$$\begin{aligned} & & 0 &= \vec{O_y} + t * \vec{d_y} \\ 0.5 &= \vec{O_y} + (t + \delta_t) * \vec{d_y} \iff & 0 &= \vec{O_y} + (t + \delta_t) * \vec{d_y} - 0.5 \end{aligned}$$

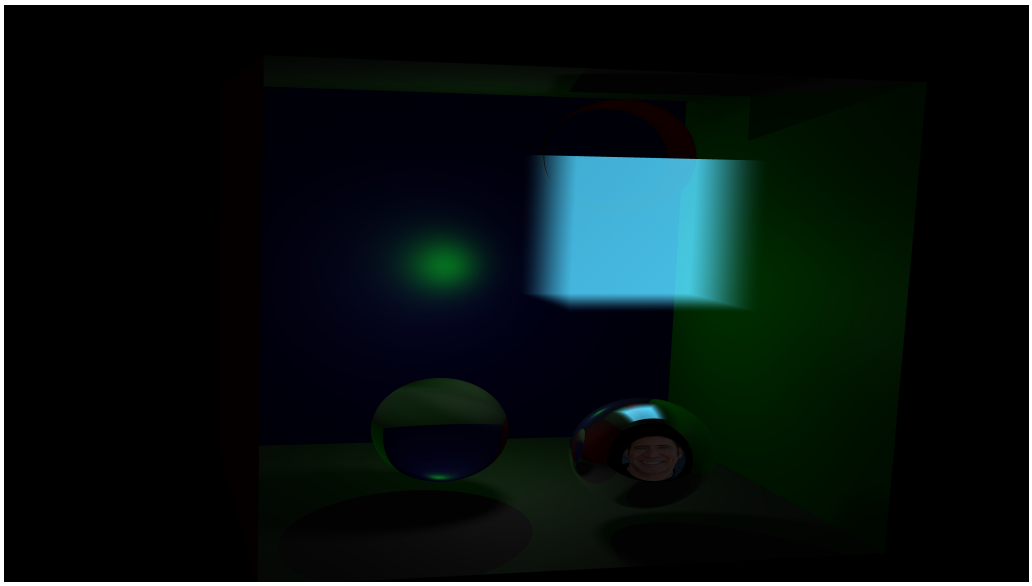
On peut facilement résoudre pour δ_t:

$$\begin{aligned} \vec{O_y} + t * \vec{d_y} &= \vec{O_y} + (t + \delta_t) * \vec{d_y} - 0.5 \\ t * \vec{d_y} &= (t + \delta_t) * \vec{d_y} - 0.5 \\ &= t * \vec{d_y} + \delta_t * \vec{d_y} - 0.5 \\ 0.5 &= \delta_t * \vec{d_y} \\ \frac{0.5}{\vec{d_y}} &= \delta_t \end{aligned}$$

Une petite remarque, dans l'exemple du dessin, comme il y a 2 voxels en x et en y, la taille d'un voxel est 0.5 (et la distance entre deux plans consécutifs sera toujours la taille d'un voxel). Par conséquent, l'équation finale est:

$$\frac{\text{Voxel\_Size}}{\vec{d_y}} = \delta_t$$

Mon exemple est en 2D, mais il est valide en 3D. La preuve est laissé au lecteur. Avec cette nouvelle version du DDA, le parcours de grille se faisait **BEAUCOUP** plus rapidement et il n'y avait plus aucun problème (bug, crash, etc...). De plus, mon code pouvait *render* un **medium** de 1000 x 1000 x 1000 voxels (avant mon max était 15 x 15 x 15) et j'ai pus obtenir une image en 8K de ma scène (juste parce que je peux).



La semaine suivante, j'ai passé mon temps à suivre la ressource en ligne de Scratchapixel. Pierre m'a envoyé cette ressource pour que puisse bien comprendre les différentes équations reliés à l'illumination, le *scattering* et le *Ray Marching*. J'ai réussi à implémenter le *Ray Marching* sans aucune diffculté. Cependant, l'implémentation les calculs d'illumination et de *scattering* on été compliqué pour quelques raisons. Premièrement, ce sont des concepts un peu plus avancé. Deuxièmement, il fallait modifier la structure de code légèrement pour permettre certains comportements ce qui à été plus difficile qu'anticipé.

En ce moment je pense étre légèrement en retard sur mon échéancier, malgré le fait que j'ai une première version du *Ray Marcher*. Lors de la semaine de lecture je vais continuer de travailler sur le *Ray Marcher* et les équations associés.