Rapport INF 443

Relativity Simulator : A Variable Speed of Light

Thibaud Gloaguen - Thomas Leplumey

6 juin 2022

Table des matières

1 Introduction **2**
1.1 Objectif du projet 2
1.2 Quelques notions de relativité restreinte [4] 2
1.3 Notations 3

2 Description et implémentation des effets relativistes **4**
2.1 Aberration des angles 4
2.2 Effet Doppler et luminosité relativiste 6
2.3 Dilatation du temps 8
2.4 Gestion des évènements et délai de l'information 8
2.5 Dynamique relativiste du joueur 10

3 Mise en forme du projet **12**
3.1 Gestion des inputs 12
3.2 Contrôle du joueur 12
3.3 Contrôle de la vitesse de la lumière 14
3.4 Heads-up display 14
3.5 Terrain infini adaptatif 15
3.6 Ciel étoilé 16
3.7 Reconstitution d'une partie du "Platâl" 16
3.8 Lampadaires 17
3.9 Objets mobiles (voitures) 18

4 Conclusion **19**
4.1 Points d'amélioration 19
4.2 Limites de la modélisation 20
4.3 Perspectives futures 20

Chapitre 1

Introduction

1.1 Objectif du projet

Le projet a pour objectif de proposer une simulation dans laquelle l'utilisateur pourra, à sa guise, décroître la vitesse de la lumière et en observer les conséquences dans un environnement simple et interactif. Outre l'intérêt pédagogique de concevoir une telle simulation, celle-ci permet aussi de mieux comprendre les effets de la relativité restreinte qui sont parfois abstraits et difficiles à appréhender.

Les contraintes sont donc de proposer une simulation physique la plus cohérente possible, une interface intuitive et un monde illustrant bien les thématiques physiques. Pour le choix du monde, nous avons donc naturellement décider de représenter une partie du *Platâl*, un monde que l'on connaît, afin de pouvoir l'explorer avec des vitesses de la lumière différente et de voir comment il est modifié. Aussi, de tels objectifs nécessitent de mélanger plusieurs domaines : l'informatique graphique, la physique, le jeu vidéo et les mathématiques appliquées.

Lors de la conception du projet, une attention particulière a été portée à la modularité du code. Dans une optique d'utiliser nos travaux pour d'autres projets plus aboutis nous avons créer différentes classes qui gèrent les contraintes physiques, de sorte qu'implémenter n'importe quelle fonctionnalité non relative directement à la physique ne demande que peu d'efforts comparés à la même implémentation dans un monde "classique". Le rapport a donc pour objectif d'introduire brièvement les concepts physiques nécessaires à la compréhension du projet, et ensuite détailler les technicités de l'implémentation.

1.2 Quelques notions de relativité restreinte [4]

La relativité restreinte (*special relativity*), présentée par Einstein en 1905 [1], repose sur le postulat que la vitesse de la lumière c est invariante par changement de référentiel inertiel (c'est-à-dire un changement de référentiel dont la vitesse relative est constante). Ce postulat, vérifié expérimentalement à de très nombreuses reprises, a des conséquences considérables sur notre manière de décrire la physique.

Ainsi, on peut démontrer que lorsqu'un observateur se déplace à une fraction non négligeable de la vitesse de la lumière par rapport à un monde immobile, des effets de dilatation du temps (l'observateur a l'impression de voir le monde autour de lui évoluer plus vite que lui), ou de contraction des longueurs (le monde autour de l'observateur se retrouve contracté dans la direction de déplacement de l'observateur) se manifestent.

La loi de composition des vitesse se retrouve également modifiée, conduisant notamment à des effets visuels étranges tels que l'aberration des angles qui déforme considérablement le champ de vision de l'observateur, ou l'effet Doppler qui se trouve modifié et conduit à une modification drastique des couleurs perçues du monde environnant.

Les équations de la dynamique changent également considérablement, et se traduisent par une inertie largement accrue d'autant plus que l'on se déplace proche de la vitesse de la lumière, empêchant ainsi par principe un objet de dépasser la vitesse de la lumière, quelles que soient les forces auxquelles il est soumis.

Enfin, de manière plus terre à terre, la finitude même de la vitesse de la lumière (et de la vitesse de l'information plus généralement) implique que nous voyons sans cesse une version retardée du monde autour de nous, d'autant plus âgée qu'elle est loin de nous (l'information mettant du temps à parvenir jusqu'à nous).

Toutefois, du fait du caractère très élevé de la vitesse de la lumière ($c = 299\,792\,458\,m.s^{-1}$), tous ces effets sont imperceptibles à notre échelle. C'est ainsi que vient l'idée de rendre la vitesse de la lumière variable dans notre simulation, permettant au joueur d'atteindre des fractions très élevées de la vitesse de la lumière tout en restant dans un environnement connu à une échelle humaine, et ainsi de rendre perceptibles et appréhendables tous ces effets.

1.3 Notations

Quelques notations seront récurrentes tout au long de ce rapport. Ainsi, on définit :

- La vitesse de la lumière : c
- La vitesse adimensionnée : $\vec{\beta}(\vec{v}) = \vec{v}/c$, et sa norme : $\beta(v) = v/c$
- Le facteur de Lorentz : $\gamma(v) = \dfrac{1}{\sqrt{1 - v^2/c^2}}$

Pour alléger les notations, lorsqu'il n'y aura pas d'ambiguïté, la dépendance en $\vec{v}$ ne sera pas toujours explicitée et on notera ainsi simplement β et γ.

Chapitre 2

Description et implémentation des effets relativistes

2.1 Aberration des angles

Détails physiques

Un premier effet relativiste qui altère considérablement la vision du joueur est l'aberration des angles (*stellar aberration*), qui déforme le champ de vision du joueur.

En effet, considérons un rayon lumineux arrivant vers le joueur avec un vecteur vitesse $\vec{k}$ (on a donc $\|\vec{k}\| = c$), alors que le joueur se déplace lui-même à une vitesse $\vec{v}$. Notons θ l'angle entre le rayon lumineux incident et la vitesse du joueur. La composition des vitesses en relativité affirme que, dans le référentiel du joueur, le rayon incident sera perçu avec un angle θ' différent, déterminé par la formule :

$$\cos\theta' = \frac{\cos\theta + \beta}{1 + \beta\cos\theta}$$

$$\sin\theta' = \frac{\sin\theta}{\gamma(1 + \beta\cos\theta)}.$$

Ainsi, lorsque la vitesse du joueur est proche de celle de la lumière, le champ de vision du joueur sera très élargi lorsqu'il avance, alors qu'à l'inverse il sera très resserré lorsqu'il recule.

Détails d'implémentation

L'implémentation de ce phénomène se fait au niveau du *vertex shader*. Après l'application des matrices *model* et *view*, la position de chaque point est modifiée en accord avec les formules ci-dessus avant d'appliquer la matrice *projection*.

Plus précisément, notons p la position de l'objet dans le repère de la caméra, et v la vitesse du joueur dans ce même repère (c'est-à-dire après application de *model* et *view*). Le cosinus $\cos\theta$ est calculé avec la formule suivante :

$$\cos\theta = \frac{\langle \vec{v}, \vec{p} \rangle}{\|\vec{v}\|\|\vec{p}\|}$$

ce calcul étant conditionné par ($\|\vec{v}\| > \varepsilon$) avec ε petit, dans le cas contraire il n'y a aucun effet relativiste à prendre en compte pour l'aberration des angles.

En revanche, le sinus $\sin\theta$ nécessite la connaissance d'une base orthonormée directe $(\vec{u}_v, \vec{u}_\theta, \vec{u}_z)$. Plus précisément, $\vec{u}_v$ est colinéaire à v, et $\vec{u}_\theta$ est obtenu par le procédé d'orthonormalisation de Schmidt sur $(\vec{u}_v, \vec{p})$. Enfin, $\vec{u}_z$ complète la base orthonormée directe :

$$\vec{u}_v = \frac{\vec{v}}{\|\vec{v}\|}$$

$$\vec{u}_\theta = \frac{\vec{p} - \langle \vec{p}, \vec{u}_v \rangle \vec{u}_v}{\|\vec{p} - \langle \vec{p}, \vec{u}_v \rangle \vec{u}_v\|}$$

$$\vec{u}_z = \vec{u}_v \times \vec{u}_\theta.$$

On calcule alors le sinus $\sin\theta$ comme suit :

$$\sin\theta = \left\langle \frac{\vec{v} \times \vec{p}}{\|\vec{v}\|\|\vec{p}\|}, \vec{u}_z \right\rangle.$$

Les sinus et cosinus de l'angle θ' peuvent finalement être calculés à l'aide des formules données précédemment. La position apparente du point (qui sera affichée à l'écran) est alors calculée comme suit :

$$\vec{p'} = \|\vec{p}\|(\cos\theta' \vec{u}_v + \sin\theta' \vec{u}_\theta).$$

La position $\vec{p'}$ remplace alors $\vec{p}$ dans l'espace projectif (la coordonnée w est laissée à 1), et subit la transformation *projection* pour être affichée à l'écran.

La donnée initiale de $\vec{p}$ est tout de même conservée pour être communiquée au *fragment shader* qui en nécessitera pour calculer d'autres effets.

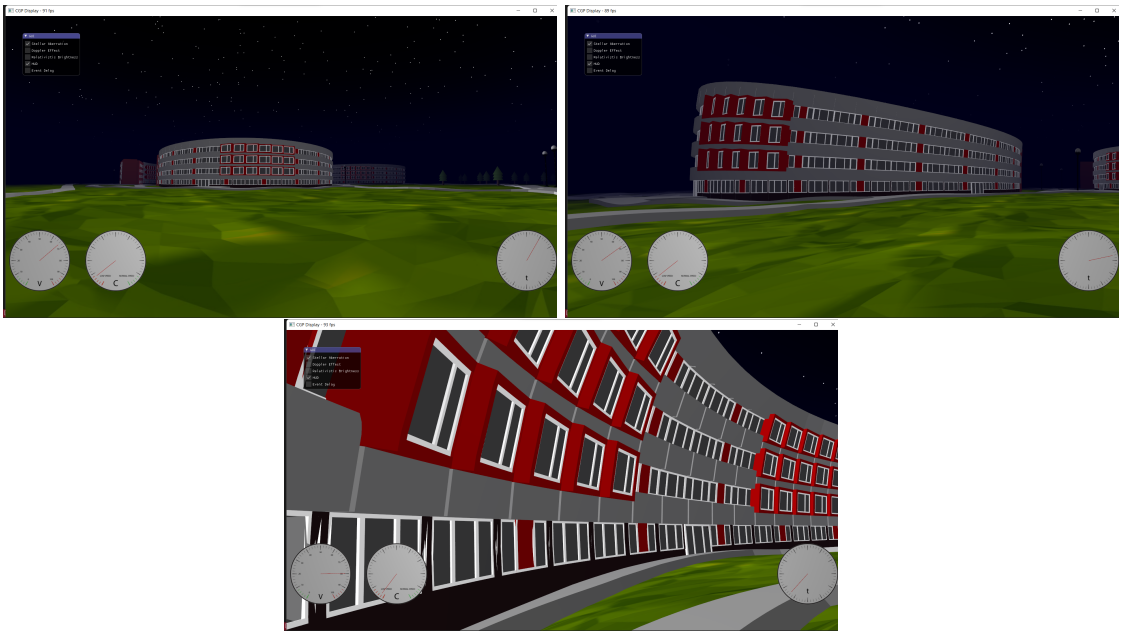

FIGURE 2.1 – Aberration des angles, lorsque le joueur se déplace vers l'avant (à gauche), sur le côté (au milieu), et une combinaison des deux (en bas)

2.2 Effet Doppler et luminosité relativiste

Détails physiques

Un second effet très visuel est l'effet Doppler relativiste. En effet, lorsqu'une source de lumière est en mouvement par rapport à un observateur, la fréquence ν_r reçue par l'observateur dans son référentiel est différente de la fréquence ν_e émise par la source, et donc la couleur perçue par l'observateur est différente de la couleur réelle de l'objet. La relation reliant ces deux fréquences est :

$$\nu_r = \frac{\nu_e}{\gamma(1 + \beta \cos\theta)},$$

où θ est l'angle entre la vitesse relative entre la source et le joueur, et le vecteur reliant la source et le joueur, le tout dans le référentiel du joueur. β et γ sont également calculés par rapport à la vitesse relative entre la source et le joueur.

L'effet Doppler impacte également la quantité de lumière reçu par l'observateur en mouvement. En effet, un observateur se déplaçant en direction de la source de lumière va recevoir plus de photons qu'un observateur s'éloignant de la source de lumière. Ceci aura alors comme conséquence pour l'observateur un changement de luminosité en fonction de sa vitesse et de sa direction de déplacement. Ce changement est traduit par la relation suivante :

$$\mu = \gamma(1 + \beta \cos\theta)\mu_0,$$

où μ et la quantité de lumière reçu par l'observateur et μ_0 la quantité de lumière reçu au repos.

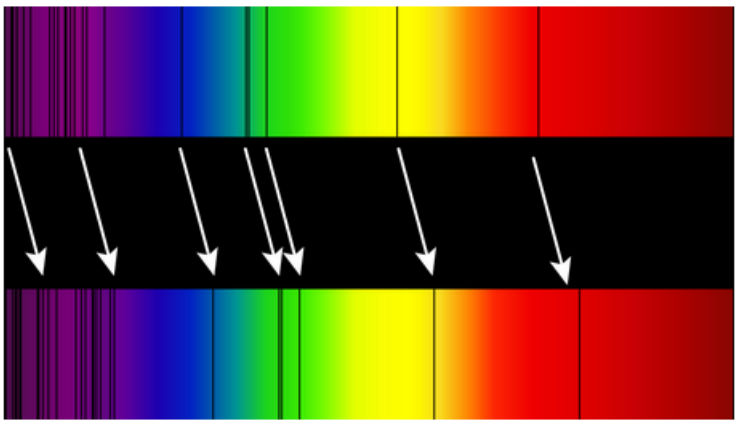

FIGURE 2.2 – Illustration de l'effet Doppler : décalage des fréquences lumineuses vers le rouge (la source de lumière s'éloigne de l'observateur)

Détails d'implémentation

L'implémentation de ce phénomène se fait au niveau du *fragment shader*. Une difficulté majeure pour implémenter l'effet Doppler est de traduire le concept de spectre en un triplet RGB. En effet, la formule précédente nous donne des informations sur la fréquence de la lumière mais l'implémentation dans OpenGL ne nous permet que de travailler avec des triplets RGB. Nous allons donc faire quelques hypothèses simplificatrices sur notre spectre de lumière, comme détailler dans [5]. Nous considérerons que le spectre d'origine est composé de trois rectangles juxtaposés, un rouge, un vert et un bleu, de même épaisseur et dont la hauteur détermine l'intensité lumineuse. Aussi, la caméra possède trois rectangles de sensibilités : un rouge, un bleu et un vert qui sont positionnées aux même emplacements que les rectangles du spectre non déplacé. La couleur est ainsi calculée en tenant compte du pourcentage de la surface de chaque rectangle de la caméra recouvert par le spectre de l'objet.

Ainsi, pour un objet en déplacement, il nous suffit de déplacer les trois rectangles de son spectre à l'aide de la formule de l'effet Doppler et ensuite de faire les calculs d'aire pour obtenir les nouvelles valeurs RGB. De plus, comme les aires en question sont de simples intersections de rectangles, on peut obtenir une formule analytique pour le calcul d'aire ce qui permet un calcul rapide, un aspect crucial puisque ce calcul est effectué en chaque pixel de l'écran.

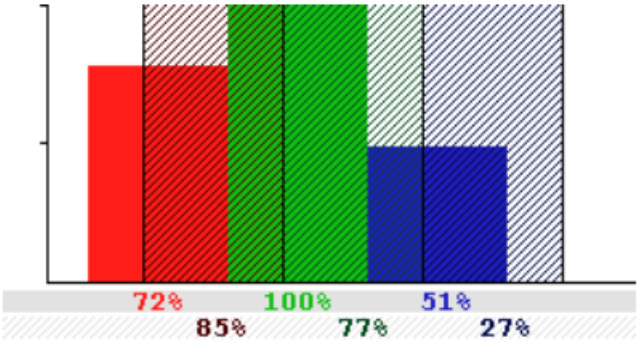

FIGURE 2.3 – Modélisation du spectre des objets par trois rectangles centrés autour des couleurs RGB, et détermination du nouveau spectre en calculant des aires d'intersections de rectangles

Aussi, l'implémentation de la luminosité relativiste est simple. Il suffit d'appliquer le facteur : $\gamma(1+\frac{v}{c})$ à chacun des termes du triplet définissant la couleur. Ce calcul est donc également effectué au niveau du *fragment shader*, avant le calul d'effet Doppler.

Un pré-requis pour appliquer la formule précédente est de calculer le vecteur reliant la source et le joueur dans le référentiel du joueur. En effet, contrairement à la cinétique classique, la composition des vitesses n'est pas une simple addition mais se calcule, en notant w' la vitesse relative reliant la source et le joueur dans le référentiel du joueur, w la vitesse de la source et v la vitesse du joueur, selon la formule :

$$\overrightarrow{w'_{\parallel}} = \frac{\overrightarrow{w_{\parallel}} - \vec{v}}{1 - \vec{v} \cdot \vec{w}/c^2}$$

$$\overrightarrow{w'_{\perp}} = \frac{\overrightarrow{w_{\perp}}}{\gamma(w)(1 - \vec{v} \cdot \vec{w}/c^2)},$$

où $\overrightarrow{w_{\parallel}}$ correspond à la composante parallèle de $\vec{w}$ au mouvement relatif de l'objet et du joueur alors que $\overrightarrow{w_{\perp}}$ correspond aux composantes orthogonales.

Ainsi, l'application de cette formule nécessite de connaître une base du plan orthogonal au mouvement relatif entre l'objet et le joueur. Pour cela on utilise la formule proposée dans [2]. En notant $\vec{n}$ notre premier vecteur de base, on a donc :

$$\vec{b_1} = (1 - \frac{n_x^2}{1+n_z}, -\frac{n_x n_y}{1+n_z}, -n_x)$$

$$\vec{b_2} = (-\frac{n_x n_y}{1+n_z}, 1 - \frac{n_y^2}{1+n_z}, -n_y).$$

La singularité $n_z = -1$ se traite facilement puisque cela implique que $\vec{n} = -\vec{z}$.

Ces calculs sont effectués dans le *vertex shader* et la nouvelle vitesse est ensuite communiquée au *fragment shader*.

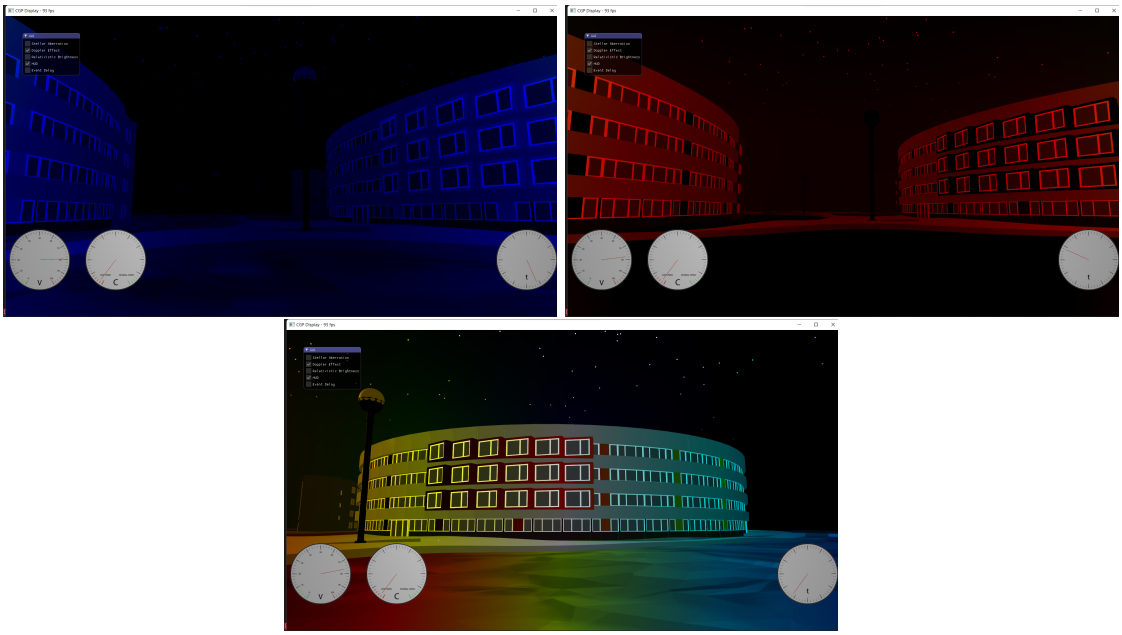

FIGURE 2.4 – Effet Doppler lorsque le joueur avance (*blueshift*, à gauche), recule (*redshift*, à droite), ou se déplace latéralement (en bas)

2.3 Dilatation du temps

Détails physiques

Un phénomène notable en relativité est le caractère relatif de l'écoulement du temps. En effet, un observateur en mouvement par rapport à un certain référentiel (dans notre cas, le joueur en mouvement par rapport au monde), percevra une désynchronisation de son temps par rapport au temps du référentiel.

Plus précisément, quand l'observateur ressentira un certain intervalle de temps dt s'écouler (il s'agit de son temps propre), il s'écoulera dans l'autre référentiel un intervalle de temps γdt.

Détails d'implémentation

Pour rendre compte de ce phénomène, nous avons implémenté une classe *relativistic_timer*. Par souci de cohérence, son utilisation est très similaire à celle du *timer_basic* de *cgp*. Elle est dotée d'un attribut *c_timer* de type *timer_basic*, ainsi que d'un attribut t de type *float*, initialisé à 0.

Son unique méthode *update(vec3 speed, float c)* prend en paramètre la vitesse du joueur et la vitesse de la lumière à l'instant courant, et actualise le temps en multipliant par γ le dt fourni par la méthode *update()* de l'attribut *timer*.

2.4 Gestion des évènements et délai de l'information

Comme la vitesse de la lumière devient faible devant les distances et temps caractéristiques du monde, l'information lorsqu'un objet change d'état mets du temps avant de parvenir jusqu'au

joueur. Pour prendre un exemple simple, si une lampe s'allume à un instant t et qu'elle est suffisamment loin du joueur, celui-ci n'aura l'information qu'à un instant $t + \delta t$. Pour implémenter la notion d'évènements dans notre monde, nous avons du prendre en compte ce délai de l'information.

Dans l'optique du jeu vidéo, et pour pouvoir facilement implémenter des évènements de nature très différent (des lampadaires et des objets animés), nous avons conçu une classe abstraite nommée *events* qui gère de façon très modulaire le délai de l'information, de sorte qu'implémenter n'importe quel évènement devient alors facile en héritant de cette classe. Les deux attributs principaux de cette classe sont : un *basic timer* et une file d'attente d'*event*. La classe *event* est une classe possédant comme attributs une date de création, un identifiant, une position et une vitesse. Ces attributs fournissent toutes les informations nécéssaires pour caractériser un évènement (rôle de l'identifiant) et le repérer. Les méthodes de la classe *events* sont : une méthode pour ajouter des *event* à la file d'attente, une méthode *update* pour actualiser la file d'attente et une méthode abstraite *activate* pour "activer" les évènements.

Les éléments centraux dans la conception de cette classe sont la file d'attente et la méthode *update* associée. L'idée est qu'à chaque frame chaque évènement pousse un statut (soit un *event*), qui correspond à son statut actuel dans son référentiel, dans la file d'attente et qu'il retire de la file d'attente tous les éléments qui ont atteint le joueur. Une bonne image pour se représenter ce qui se passe est d'imaginer que chaque évènement émet à chaque frame une onde qui se propage à la vitesse de la lumière et qui contient toutes les informations sur l'évènement au moment de son émission. Ainsi, le joueur, dès qu'il rencontre une de ses ondes, va voir l'évènement selon l'information portée par cette onde.

Lorsqu'un élément est retiré de la fille d'attente, c'est-à-dire qu'il a atteint le joueur, il est activé, c'est-à-dire envoyé en argument à la méthode *activate*. Contrairement à la méthode *update*, la méthode *activate* est abstraite. C'est celle-ci qui va déterminer le comportement particulier de chaque évènement et c'est ce qui fait la modularité de la classe *events*. La classe *events* gère le "transport" de l'information tandis que les classes qui en héritent explicitent le contenu de cette information. Ainsi pour une lampe, la méthode *activate* allumera ou éteignera la lampe suivant l'identifiant de l'évènement, pour la voiture elle mettra à jour les coordonnées et la vitesse de la voiture. Ces quelques exemples illustrent les possibilités offertes par la classe *events*.

Revenons sur la méthode *update*. Pour savoir quand retirer le premier élément de la file d'attente, elle regarde la date de création de l'event et l'heure actuelle pour connaître le rayon de l'onde d'information et la compare à le distance entre le lieu d'émission de l'event et la position actuelle du joueur. Or, si l'event ou le joueur sont en mouvement, la position de l'event dans le référentiel du joueur sera différente que celle dans le référentiel du monde. La fonction event doit prendre en compte cela. Cela implique d'une part de faire un calcul de composition des vitesses relativistes comme expliqué dans la partie consacrée à l'effet Doppler. D'autre part, il faut aussi prendre en compte la contraction des longeurs dans la direction du vecteur vitesse selon la formule :

$$\overrightarrow{p_{effectif}} = \overrightarrow{p} + (\frac{1}{\gamma} - 1)(\overrightarrow{p} \cdot \overrightarrow{u_v})\overrightarrow{u_v},$$

avec $\overrightarrow{p_{effectif}}$ la position de l'évènement dans le référentiel du joueur, $\overrightarrow{p}$ la position de l'évènement dans le référentiel du monde et $\overrightarrow{u_v}$ la composée normalisée du vecteur vitesse du joueur et de l'objet dans le référentiel du monde.

Un dernier scénario à prendre en compte pour la fonction *update* est lié au changement de c. En effet, les formules relativistes employés pour faire les différents calculs de composition des

vitesses sont faits de tels sorte que la vitesse ne dépasse pas c. Or, il se peut qu'un évènement passé se soit enclenché à un moment ou c' étant plus grand que le c actuel et ainsi avoir une vitesse plus grande que c. Dans ce cas, nous normalisons simplement la vitesse de l'évènement à c pour éviter des résultats aberrants (les formules n'étant pas conçues pour une vitesse supérieure à c).

Pour résumer simplement la classe *events*, elle est composée d'une file d'attente d'évènements et d'une méthode *update* associée qui gère, en prenant en compte tous les effets relativistes associés, la propagation de l'information à la vitesse de la lumière, ainsi que d'une méthode abstraite *activate* qui assure la modularité et permet de traiter de façons variés l'information.

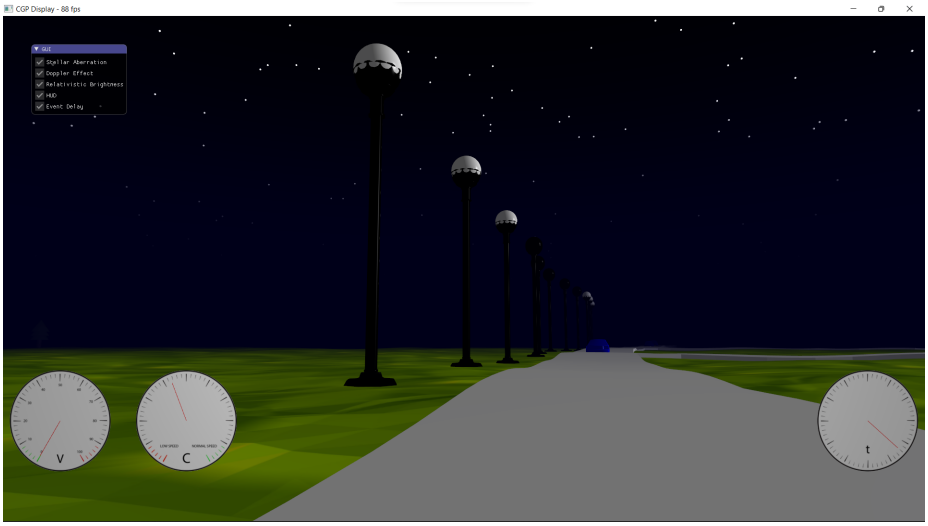

FIGURE 2.5 – Illustration du délai de l'information : la lumière des lampadaires, tous simultanés initialement, nous parvient en décalé lorsque la vitesse de la lumière est relativement faible

2.5 Dynamique relativiste du joueur

Détails physiques

En relativité restreinte, de même que la composition des vitesses est modifiée, la composition des accélérations l'est aussi. C'est entre autres cela qui permet à un objet de ne jamais dépasser la vitesse de la lumière, quelle que soit la force à laquelle il est soumis.

Précisément, nous avons fait le choix de contrôler le joueur en lui appliquant des forces (c'est-à-dire en agissant sur son accélération), et non en contrôlant sa vitesse (comme cela peut être fait dans de nombreux jeux), et ce pour plusieurs raisons :

— L'hypothèse est raisonnable d'un point de vue physique (lorsque nous marchons ou contrôlons un véhicule, c'est notre accélération que nous contrôlons). Alors que dans un jeu standard, l'ergonomie prime souvent sur le réalisme, nous sommes prêts ici à faire de légers compromis dans le but d'illustrer au mieux la relativité.

— Cela permet très facilement d'éviter au joueur de dépasser la vitesse de la lumière. En effet, en laissant par exemple une vitesse constante au joueur, avec une vitesse de la lumière variable, il peut y avoir des risques que la vitesse du joueur dépasse c. En revanche, en utilisant les équations de la dynamique relativiste, cela est simplement impossible.

— Il est nécessaire que la vitesse du joueur soit continue, car l'affichage du monde à l'écran dépend grandement de la vitesse du joueur (à cause de l'aberration des angles et de l'effet Doppler). Une vitesse discontinue causerait des changements brusques de l'affichage, qui

sont assez désagréables. Ainsi, contrôler l'accélération est relativement commode pour assurer une vitesse continue.

Ainsi, à partir des inputs du joueur, on en déduit son accélération $\vec{a}$ à chaque frame (voir partie "Contrôle du joueur"), dans son référentiel. On convertit alors l'accélération dans le référentiel du monde par la formule :

$$\vec{a}' = \gamma^{-3}\vec{a}$$

Puis, on actualise sa vitesse et sa position dans le monde, en prenant en compte le fait que le temps du monde est dilaté par rapport au temps propre du joueur (voir la partie "Dilatation du temps") :

$$\vec{v} = \vec{v} + \vec{a}'\,\gamma dt$$

$$\vec{p} = \vec{p} + \vec{v}\,\gamma dt$$

Ainsi, lorsque l'on est proche de la vitesse de la lumière, l'inertie est bien plus élevée. Cela se ressent bien en jouant (on met plus de temps à accélérer ou à changer de direction, les sauts durent plus longtemps...)

Chapitre 3

Mise en forme du projet

3.1 Gestion des inputs

La gestion des inputs a été implémentée avec les méthodes standards de la bibliothèque *GLFW*. Lors de l'initialisation de la fenêtre, des *callbacks* sont associés aux différents évènements utilisateur. Ainsi, différentes fonctions sont appelées lors de l'appui sur une touche du clavier, le déplacement du curseur ou une rotation de la molette.

Ces fonctions communiquent ensuite avec un objet de la classe *PlayerMover* (que nous avons implémenté), en modifiant ses attributs ou en appelant ses méthodes.

Ainsi, l'appui sur les touches "Z", "Q", "S", "D" et "SPACE" modifie l'état des booléens *isMovingForward*, *ismovingBackward*, *isMovingLeftward*, *isMovingRightward* et *isJumping* respectivement. Un déplacement du curseur appelle la méthode *update_mouse()*, qui se charge de mettre à jour les attributs de position et vitesse du curseur. Enfin, une rotation de la molette appelle la méthode *update_c()*.

3.2 Contrôle du joueur

La fonction *update_camera()* de la classe *PlayerMover* est appelée à chaque frame dans la boucle d'animation de la fonction *main()*, et se charge de mettre à jour la position et la vitesse du joueur.

Rotation de la caméra

En premier lieu, l'orientation de la caméra est mise à jour à partir de la vitesse du curseur. Plus précisément, la fonction *camera.manipulator_rotate_roll_pitch_yaw()* permet de gérer la rotation haut/bas de la caméra, en fonction de la vitesse verticale du curseur. La rotation autour de l'axe z du monde est effectuée par la multiplication de la rotation de la caméra par un quaternion, représentant précisément une rotation autour de l'axe z, d'angle proportionnel à la vitesse horizontale du curseur.

Déplacements horizontaux de la caméra

La direction du vecteur accélération dans le plan horizontal est calculée en fonction des booléens *isMovingForward*, *ismovingBackward*, *isMovingLeftward* et *isMovingRightward*. L'accélération horizontale est alors renormalisée pour avoir une norme constante (définie comme paramètre physique constant de la classe *PlayerMover*).

Gestion du saut

La gestion du saut a été ici un léger enjeu technique comparé à une gestion "basique" de saut. En effet, nous avons voulu garantir un saut fluide et particulièrement une vitesse continue en fonction du temps (pour les mêmes raisons que précédemment, une vitesse discontinue mène à une variation très brusque de l'affichage due à l'aberration des angles et l'effet Doppler, qui rend le saut très désagréable).

Ainsi, pour garantir un tel saut fluide avec une vitesse continue, le saut est décomposé en trois phases :

1. Une phase d'accélération verticale ascendante, à accélération constante relativement élevée, permettant d'atteindre continûment et relativement rapidement une certaine vitesse de saut.
2. Une phase de chute libre, durant laquelle le joueur est simplement soumis à la gravité, permettant une chute parabolique durant la majeure partie du saut.
3. Une phase de décélération verticale, à accélération constante ascendante, calculée précisément pour que le "sommet" de la parabole soit pile au sol (de sorte que le joueur arrive au sol avec une vitesse nulle, causant ainsi presque aucune discontinuité de vitesse).

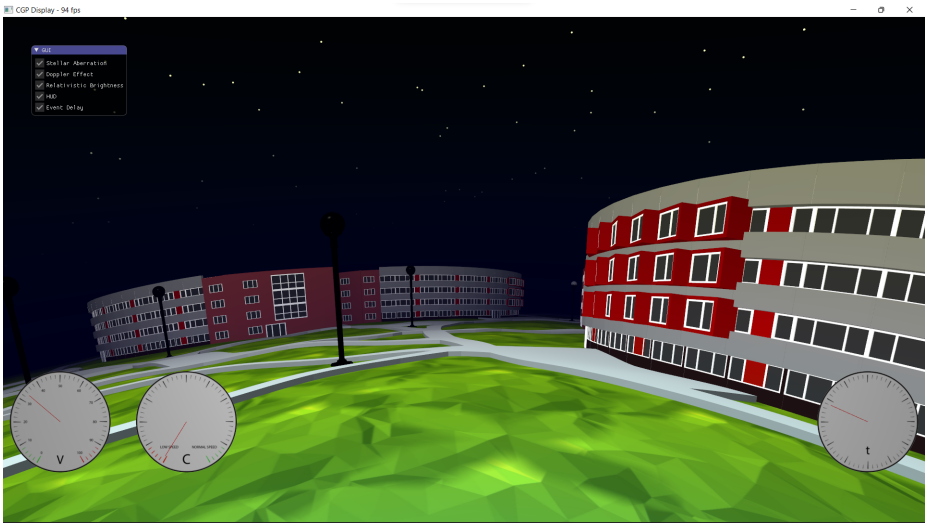

FIGURE 3.1 – Saut et effets visuels engendrés par celui-ci

Méthode d'Euler semi-implicite

L'actualisation de la vitesse et de la position de la caméra se fait via la méthode d'Euler semi-implicite, en utilisant les équations de la dynamique relativiste (détaillées dans la section "Dynamique relativiste du joueur").

Gestion du joueur au sol

Par soucis de commodité, nous avons décidé de ne pas prendre en compte les variations de hauteur du terrain dans le calcul de la vitesse du joueur, mais simplement en actualisant sa position a posteriori. Ce choix réside dans le fait que des déplacements verticaux incessants induisent des effets nauséeux à l'écran du fait de l'aberration des angles et de l'effet Doppler, qui rendent l'expérience de jeu relativement désagréable.

Ainsi, après le calcul de la dynamique du joueur par les équations relativistes, la position verticale du joueur est modifiée pour le mettre au niveau du sol si il est détecté comme étant suffisamment proche du sol (ce qui se calcule aisément à partir de la méthode *evaluate_hills_height()* de la classe *Terrain*, expliquée plus en détail dans la section "Terrain infini adaptatif").

3.3 Contrôle de la vitesse de la lumière

Lors de l'appel de la méthode *update_c()* de la classe *PlayerMover*, la vitesse de la lumière c de la classe *scene* est actualisée, selon les équations suivantes :

— Si la rotation de la molette est vers le bas, alors : $c = c - 0.1 \times (0.97c - v_{max})$

— Si la rotation de la molette est vers le haut, alors : $c = c + 0.1 \times (c - v_{max})$

où v_{max} est la vitesse de marche maximale du joueur.

3.4 Heads-up display

Afin de communiquer les informations essentielles à la bonne compréhension du joueur, nous avons mis en place un *Headsup display (HUD)*, c'est à dire un ensemble d'objet 2D affiché en premier plan de la fenêtre du joueur.

L'affichage des éléments du HUD est géré en définissant une deuxième caméra qui applique une projection orthogonale aux éléments de la scène qu'elle reçoit. Cet environement se voit également attribué une vitesse constante nulle pour que les *shaders* ne calculent pas les effets relativistes sur les éléments du HUD. Les éléments du HUD sont ensuite créés avec des primitives de maillage 2D et sont dessinés avec la fonction *draw* prenant en argument l'environnement du HUD.

Nous avons fait le choix d'afficher 3 compteurs distincts : une horloge, un compteur indiquant la valeur de c et un compteur indiquant la vitesse du joueur en pourcentage de c.

— **Horloge :** Il s'agit de l'horloge attaché au référentiel du monde. Ainsi, lorsque le joueur se déplace lentement par rapport à c, celle-ci s'écoule à la même vitesse que l'horloge du joueur, c'est-à-dire qu'une seconde dans le monde virtuel passe aussi vite qu'une seconde dans le monde réel. Dès que la vitesse du joueur atteint des ordres de grandeur similaire à c, sous les effets de la dilatation du temps, son temps propre (le temps dans le monde réel) s'écoule plus lentement que le temps du monde virtuel. Ainsi, l'horloge tournera plus vite. Cette horloge illustre donc bien le phénomène de dilatation du temps, et le concept primordial en relativité d'attacher une horloge à un référentiel en plus des trois dimensions d'espace. D'un point de vue technique, on utilise pour l'horloge un *relativistic timer*.

— **Compteur de c :** Celui-ci est représenté par un compteur ressemblant à un cadran de vitesse à aiguille automobile. Il permet au joueur d'avoir une idée intuitive de la valeur de c à tout moment, ce qui permet d'expérimenter facilement différentes valeurs de c pour étudier l'effet Doppler lié aux objets en mouvement ou bien au déphasage des lampes liée à la propagation de l'information. L'échelle utilisée est une échelle logarithmique pour explorer une plus grande gamme de valeurs de c.

— **Compteur de vitesse :** Il se présente lui aussi comme un compteur de vitesse à aiguille automobile. Il affiche la vitesse en pourcentage de c car il s'agit de l'information pertinente compte-tenu des objectifs du projet. En effet, cela permet au joueur de savoir à partir de quelle valeur de la vitesse par rapport à c les effets de la relativité restreinte ne sont plus négligeables, ce qui peut fournir ensuite des intuitions dans des calculs physiques par exemple. La connaissance de la vitesse à laquelle les effets se manifestent n'aurait que très peu d'intérêt car la valeur exacte de c n'est pas explicitement fournie au joueur.

3.5 Terrain infini adaptatif

Pour accentuer l'aspect jeu vidéo du projet, la génération du terrain est infinie de telle sorte qu'un utilisateur pourrait agrandir le monde indéfiniment en rajoutant d'autres objets 3D.

La génération du terrain repose sur un ensemble élémentaire de 4 *chunk*. Un *chunk* est une brique élémentaire de base de terrain, ici un terrain de 100×100 dont la hauteur est déterminé par un bruit de Perlin et est ensuite donnée par la méthode *evaluate_hills_height()*. Pour assurer la continuité du terrain, celui-ci est ensuite reproduit 3 fois :

- une fois en étant miroité selon l'axe x
- une fois en étant miroité selon l'axe y
- une fois en étant miroité selon les axes x et y

Ces 4 *chunks*, assemblés correctement, forme ainsi un système à géométrie torique. L'espace est donc maillé avec une succession de cet ensemble de 4 *chunks* dans toutes les directions.

Cependant, l'intégralité de l'espace n'est pas affiché autour du joueur mais seulement un carré de taille 3×3 *chunks* centré autour du joueur. Lorsque le joueur se déplace et sort du *chunk* central, le nouveau *chunk* dans lequel le joueur est entré devient le *chunk* central et l'affichage est actualisé en conséquences. L'implémentation technique de cette étape passe par la mise en place d'un deuxième système de coordonnée qui permet à une position donnée du joueur d'afficher la position du *chunk* correspondant. Ensuite, un calcul de modulo permet de déterminer quelle rotation le chunk a subi et donc d'actualiser les coordonnées en conséquence pour la méthode *evalute_hills_height()* par exemple. Un tel système de coordonné permettrait également de ne pas afficher les objets qui sont présent sur un *chunk* que le joueur ne voit pas (nous avons fait cela pour les arbres mais pas pour les demis-lunes,les lampadaires et les routes).

Le monde étant dorénavant infini, afin que le joueur ne s'égare pas, nous avons peuplé les chunks non centraux d'une forêt, dont la densité en arbre croît avec la distance au centre du monde (le *chunk* (0,0)). La fonction densité utilise les coordonnées des chunks comme argument.

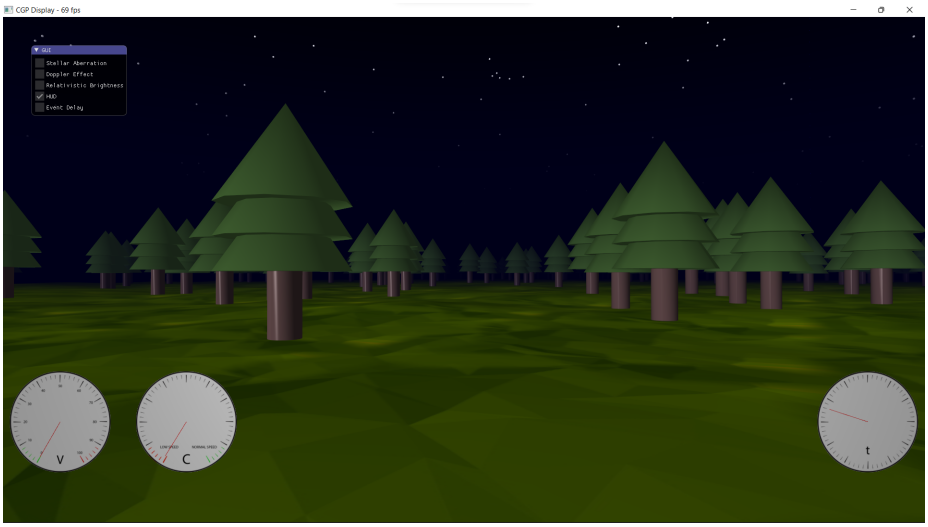

FIGURE 3.2 – Génération aléatoire des arbres sur le terrain : la densité est relativement élevée lorsqu'on est loin du *chunk* d'origine

3.6 Ciel étoilé

L'implémention actuelle du ciel étoilé, certes largement sous-optimale, consiste en une grande sphère noire centrée sur le joueur, ainsi que de nombreuses petites sphères blanches disposées selon une distribution aléatoire uniforme sur la grande demi-sphère supérieure.

Afin de générer une position aléatoire tirée selon une distribution uniforme sur la demi-sphère (représentée par des coordonnées (θ, φ), où θ est par rapport au plan horizontal), nous utilisons la méthode de rejet. Concrètement, nous tirons aléatoirement trois réels selon des distributions uniformes :

- $\theta \sim \mathcal{U}(0, \pi/2)$
- $\varphi \sim \mathcal{U}(0, 2\pi$
- $u \sim \mathcal{U}(0, 2\pi)$

Si $u \in [0, 2\pi \cos\theta]$, alors on garde la position tirée, sinon on rejette ce tirage et on en effectue un nouveau. Cette implémentation est effectuée dans la méthode *generate_positions_on_sky()* de la classe *Sky*.

On obtient ainsi une liste de positions distribuées uniformément sur la demi-sphère supérieure, qui sont ensuite converties en coordonnées cartésiennes pour pouvoir finalement afficher les petites sphères blanches à ces positions.

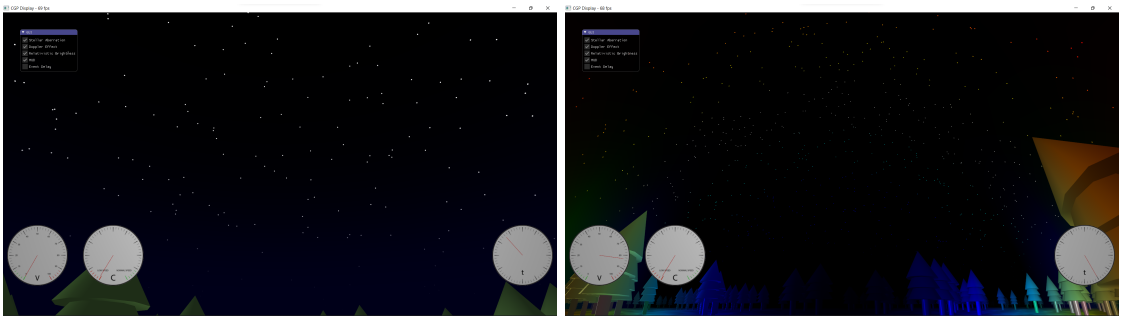

FIGURE 3.3 – Ciel étoilé, lorsque le joueur est fixe (à gauche) et lorsqu'il est en mouvement (à droite)

3.7 Reconstitution d'une partie du "Platâl"

Une composante importante du projet est de retrouver des éléments de notre environnement quotidien qui puissent être vus sous l'angle de la relativité. Ainsi, nous avons reconstitué notre lieu de vie à l'X, à savoir les demi-lunes et alentours.

Les demi-lunes ont été entièrement reconstituées par nos soins à l'aide du logiciel libre *Blender*, dans le cadre de notre travail au JTX (l'association audiovisuelle de l'X). Nous avons ainsi réutilisé ce modèle 3D, moyennant toutefois un travail préalable.

Tout d'abord, il a été nécessaire de subdiviser le maillage. En effet, les effets d'aberration des angles (voir la section "Aberration des angles") sont non linéaires. Ainsi, des arrêtes trop grandes conduisent d'une part à un rendu non réaliste, et d'autre peuvent causer des effets visuels néfastes comme des faces qui se superposent avec d'autres.

Ensuite, il a été nécessaire d'exporter le maillage en plusieurs fichiers *.obj*, en fonction de la texture utilisée. En effet, étant donné qu'il n'est possible d'utiliser qu'une texture par *mesh*, les différents maillages sont importés séparément afin de pouvoir être texturés différemment.

Enfin, par souci d'efficacité, les maillages ne sont importés qu'une fois chacun. Ils sont alors rassemblés dans une *hierarchy* pour reformer une demi-lune. Enfin, au moment de l'affichage, cette *hierarchy* est affichée quatre fois aux emplacements adéquats et avec les rotations adéquates.

Tout cela est empaqueté dans une classe *Demilunes*, contenant essentiellement deux méthodes : une méthode *initialize()* qui crée la hiérarchie et affecte les différentes textures aux bons éléments, et une méthode *display()* qui affiche les quatres demi-lunes aux bons emplacements et avec les bonnes rotations.

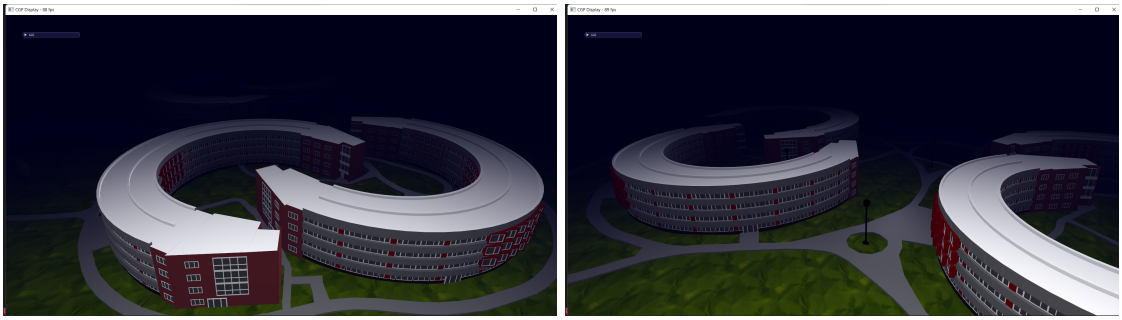

FIGURE 3.4 – Vues aériennes du "Platâl"

3.8 Lampadaires

Les lampadaires sont hérités de la classe *events* présentée précédemment. L'avantage de ce système de classe abstraite est de permettre une implémentation relativement facile de pleins d'éléments distincts. Nous avons fait le choix pour ce projet d'implémenter un lampadaire qui clignote à une fréquence donnée, car celui-ci illustre bien les phénomènes physiques que l'on souhaite illustrer. En effet, le fait que tous les lampadaires du monde clignotent en phase permet de voir, lorsque c décroît suffisamment, le temps que met la lumière à parvenir jusqu'au joueur au travers d'un déphasage des différentes lampes. Aussi, le fait que la fréquence soit fixée cela illustre également l'effet Doppler : lorsqu'on s'approche de la lampe, celle-ci clignote plus vite, et lorsqu'on s'en éloigne celle-ci clignote plus lentement. Enfin, un autre effet, moins visible car masqué par l'effet Doppler, est la dilatation du temps avec la fréquence de la lampe qui augmente avec la vitesse du joueur car pour le joueur le temps dans le monde virtuel s'écoule plus vite que dans son propre référentiel ; c'est aussi le phénomène illustré par l'horloge du HUD. Dans notre simulation, les lampes ont une fréquence de 2Hz.

La classe *Lamp*, qui gère les lampadaires, possède donc comme attributs un *relativistic timer* et une période qui gère le clignotement de la lampe, une variable booléenne indiquant si celle-ci est actuellement éteinte ou allumée, et les maillages correspondant au visuel du lampadaire. La méthode *activate* de *Lamp* change juste la couleur du lampadaire, qui correspond à éteint ou allumé, en fonction de l'event donné en argument, et la variable booléenne, qui s'inverse après une période, permet d'envoyer un évènement correspondant à éteint ou allumé dans la file d'attente.

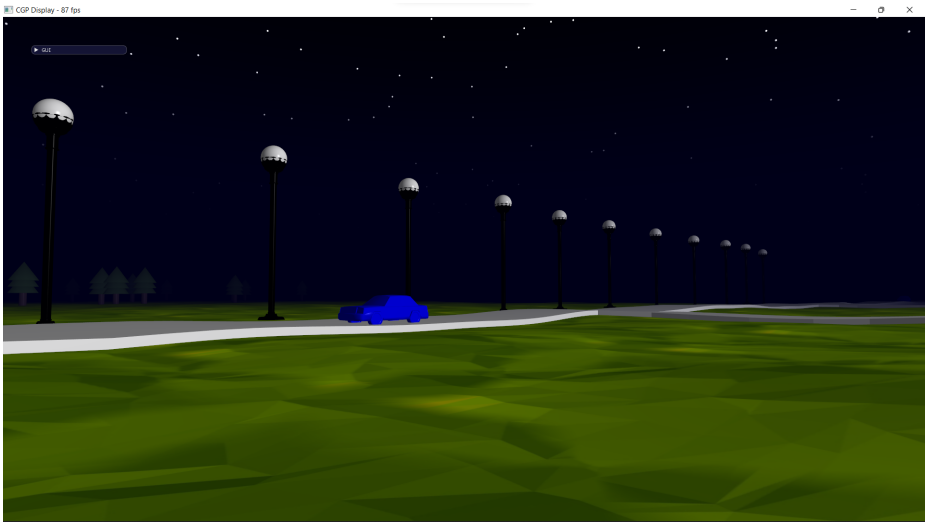

FIGURE 3.5 – Rangées de lampadaires sur le boulevard des maréchaux

3.9 Objets mobiles (voitures)

Les objets mobiles sont également hérités de la classe *events* et donc cela permet de les implémenter plus facilement. La classe implémentée ici ne permet de gérer que des objets animés par animation descriptive. La classe possède comme attribut une *keyframe strucuture*, c'est-à-dire un tableau de positions et de temps associés, un *timer interval* pour gérer la durée de l'animation, deux variables indiquant la vitesse et la position actuelle (qui jouent un rôle similaire à la variable booléenne de la lampe) et enfin un maillage correspondant au visuel de l'objet.

La méthode *activate* de cette classe se contente d'attribuer aux attributs de position et de vitesse actuels la position et la vitesse de l'*event* reçu en argument. La méthode *update*, en plus d'afficher la voiture à la position et à la vitesse de ses attributs, interpole linéairement la position à l'aide de son *timer interval* pour l'ajouter à la fin de la queue d'*events*. Comme on peut encore le constater, avec la modularité de la classe *events*, implémenter la classe d'ojets mobiles devient relativement aisé.

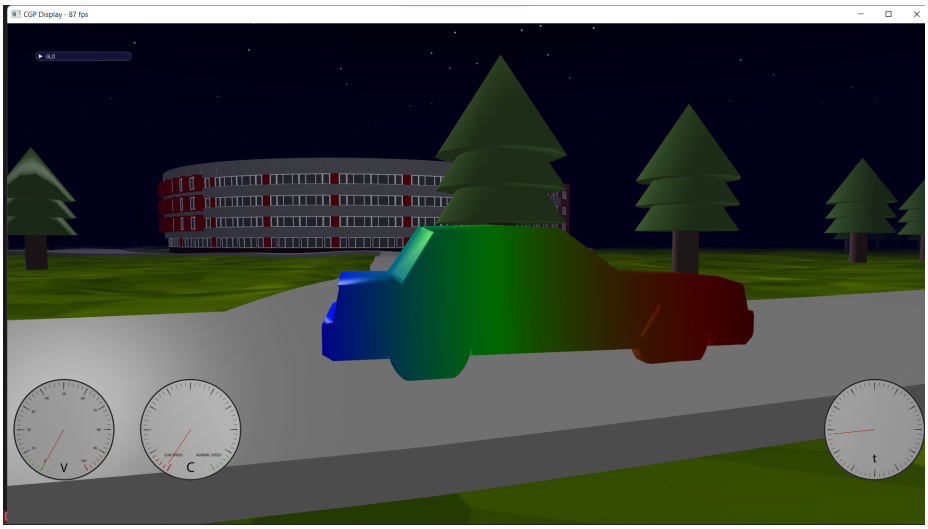

FIGURE 3.6 – Effet Doppler sur une voiture en mouvement (joueur immobile)

Chapitre 4

Conclusion

4.1 Points d'amélioration

Non-linéarité des effets relativistes

Comme expliqué précédemment, les effets relativistes d'aberration des angles et d'effet Doppler sont non linéaires. Or, lors du rendu projectif, les positions et couleurs sur les différentes faces sont interpolées linéairement entre les sommets du maillage. Nous avons résolu partiellement ce problème en subdivisant suffisamment les maillages pour que ces effets soient peu visibles, mais quelques artéfacts résident.

Une solution pour résoudre ce problème pourrait être de retravailler l'image après le rendu, en ayant conservé les informations de position et de vitesse des pixels de l'image dans des buffers, et ainsi de corriger les positions des pixels interpolés pour les repositionner correctement en fonction de leur position et vitesse.

Couleur des objets

Dans notre implémentation actuelle de l'effet Doppler, nous ne considérons que le spectre d'émission des objets dans le visible, déjà lui-même largement approximé en seulement trois longueurs d'onde. Plusieurs améliorations pourraient être apportées à cela :

— Premièrement, on pourrait considérer un modèle de spectre d'émission et d'absorption continu des objets. Il faudrait pour cela tout d'abord trouver un modèle physique décrivant plus ou moins fidèlement ces spectres. Ensuite, pour tirer partie de ce spectre continu, il faudrait représenter les couleurs non plus par des vecteurs à quatre composantes mais beaucoup plus, afin d'échantillonner le spectre et appliquer le modèle de Phong (ou autre modèle d'illumination) à chacune de ces longueurs d'onde séparément, pour les recombiner à la fin.

— Une telle description de spectre continu pourrait également nous permettre éventuellement de représenter le spectre hors du domaine visible (dans l'ultraviolet et l'infrarouge). Ainsi, lorsque l'effet Doppler devient trop intense, au lieu de simplement voir du noir, nous verrions l'effet des longueurs d'ondes infrarouges ou ultraviolettes qui entrent dans le spectre visible à cause du décalage induit par l'effet Doppler.

Ciel étoilé

Une solution plus adaptée que la notre pour rendre un ciel étoilé serait d'utiliser une *skybox* de ciel étoilé, ou bien de générer une texture procédurale à plaquer sur la sphère. Toutefois, la nature des coordonnées (u, v) de la sphère et les singularités aux pôles rendent cela plus délicat, d'où notre implémentation que nous avons jugée plus simple.

4.2 Limites de la modélisation

D'un point de vue physique, la modification de la vitesse de la lumière aurait bien d'autres conséquences fondamentales sur notre univers. On peut par exemple citer deux d'entre elles :

— Les constantes de permittivité du vide ε_0 et de perméabilité du vide μ_0 sont reliées à la vitesse de la lumière par la relation $\varepsilon_0\mu_0 = \frac{1}{c^2}$. Ainsi, une modification de la vitesse de la lumière altérerait également ces constantes, qui régissent l'électromagnétisme. Une telle modification de cette interaction fondamentale pourrait avoir des effets sur la matière elle-même.

— La constante de structure fine est la constante de couplage de l'interaction électromagnétique. Son expression est $\alpha = \frac{e^2}{4\pi\varepsilon_0\hbar c}$. Une diminution de c pourrait donc rendre cette constante largement plus grande que sa valeur actuelle dans notre monde ($\alpha \approx \frac{1}{137}$), et modifier considérablement le profil de l'interaction électromagnétique, et donc la structure de la matière entre autres choses.

Bien évidemment, ces considérations n'ont clairement pas à être prises en compte dans notre projet étant donné son objectif. Le projet conserve très bien sa portée éducative dans sa forme actuelle et de telles considérations, en plus d'être potentiellement impossibles à modéliser physiquement, n'apporteraient rien à son objectif. Toutefois, il reste intéressant d'avoir conscience de ces limites.

4.3 Perspectives futures

Jeu interactif autour de la relativité restreinte

Une évolution intéressante du projet pourrait être la création d'un jeu interactif mettant en jeu les phénomènes de relativité restreinte illustrés ici. L'idée pourrait être d'imbriquer les mécaniques du jeu et les effets de la relativité restreinte, obligeant ainsi le joueur à comprendre et appréhender ces effets afin d'atteindre ses objectifs.

Un exemple de jeu déjà réalisé autour de concepts très similaire est le jeu *A Slower Speed of Light* [3], dans lequel le joueur évolue dans un monde limité et doit ramasser des orbes pour faire diminuer la vitesse de la lumière (le jeu finissant au bout de 100 orbes ramassées lorsque la vitesse de la lumière atteint celle du joueur). Toutefois, dans un tel jeu, le joueur est presque uniquement spectateur et n'est pas réellement amené à interagir avec les effets relativistes. En outre, le jeu n'a que peu d'intérêt propre, autre qu'éducatif, ce qui limite en soi le public touché.

On pourrait par exemple imaginer un jeu d'énigmes et de réflexion dans lequel le joueur pourrait faire varier la vitesse de la lumière via certains mécanismes, et jouer ainsi sur les changements de couleur ou la déformation du champ de vision pour mettre en évidence des éléments, ou bien sur la dilatation des durées pour jouer sur des contraintes de temps, ou encore les changements de référentiel pour rendre simultanés des éléments qui ne le sont initialement pas (ou inversement...)

Une autre possibilité pourrait être de placer la modification de la vitesse de la lumière comme une contrainte du monde, qui obligerait le joueur à s'adapter en permanence (ce qui peut s'avérer être une contrainte importante si par exemple le joueur doit se battre contre d'autres entités ou s'il est limité par le temps...)

Vers la relativité générale

Une autre perspective future d'évolution du projet pourrait être la prise en compte d'effets de relativité générale dans cette simulation. De nombreux effets de relativité générale pourraient être très intéressants visuellement, tels que des effets de lentille gravitationnelle, ou la physique aux abords d'un trou noir.

Toutefois, de tels effets semblent de moins en moins adaptés à la méthode de rendu projectif que nous utilisons actuellement. Une méthode de rendu par lancer de rayons (*raytracing*) semble bien plus adaptée à la description de la relativité générale, permettant ainsi de courber les rayons lumineux et d'obtenir assez facilement les effets désirés.

Toutefois, le calcul de la courbure des rayons lumineux requerrait, dans des cas complexes, le calcul progressif des géodésiques par transport parallèle, nécessitant lui-même le calcul de la métrique tout le long des trajectoires, qui peut s'avérer très coûteux et inadapté à une simulation en temps réel. Ainsi, se limiter à des objets uniques et simples (typiquement sphériques, des étoiles ou des trous noirs), pour lesquels il serait possible de calculer analytiquement toutes ces quantités, semble plus raisonnable dans un premier temps.

Bibliographie

[1] Albert Einstein. « On the special theory of relativity ». In : *Ann Phys* 17 (1905), p. 891-921.

[2] Jeppe Revall Frisvad. « Building an orthonormal basis from a 3D unit vector without normalization ». In : *Journal of Graphics Tools* 16.3 (2012), p. 151-159.

[3] Gerd Kortemeyer, Philip Tan et Steven Schirra. « A Slower Speed of Light: Developing intuition about special relativity with games. » In : *FDG*. 2013, p. 400-402.

[4] David Langlois. *Introduction à la relativité*. 2020.

[5] Fons Van der Plas et Just Verlinden. *Simulating the Theory of Relativity*. 2015. url : `https://docs.google.com/document/d/1WJxNgcMp53MxmJopYbQ6nQ-feFoi8JF5OUCOixQZHZO/pub`.