UNIVERSITÉ DE
SHERBROOKE

Projet de session

Corps humain fonctionnel

présenté à :

Gilles Brunet

Processus concurrents et parallélisme
(IFT630)

par :

Jérémi Panneton
(16 110 842)
Charles Denicourt
(16 008 378)

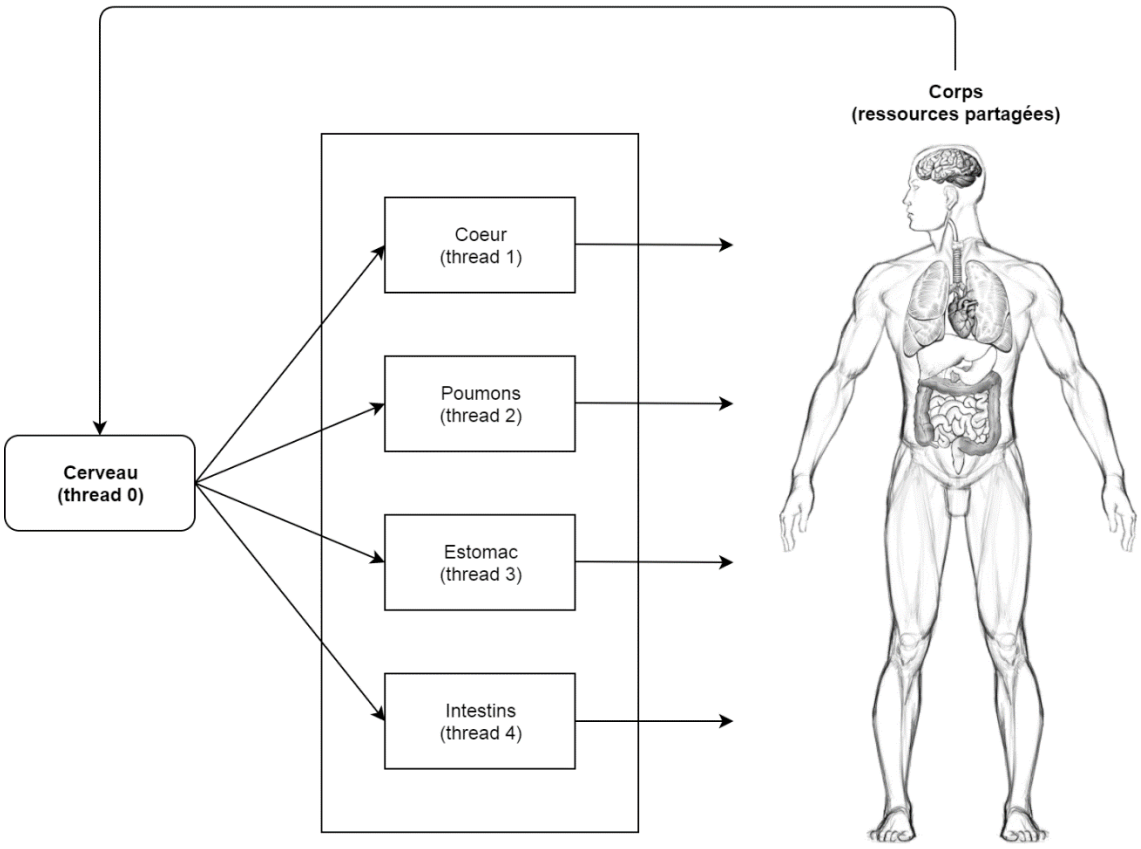
12 août 2018

Table des matières

Démarche et recherches 3
Signaux 5
Ressources partagées 8
Cerveau 8
Système digestif 9
Mode de déploiement 11
Compilation 11
Correction 11
Exécution 12
Références 15

Démarche et recherches

Dans le cadre du cours en parallélisme, nous avons décidé de faire un corps humain minimaliste fonctionnel comme projet final. En effet, nous cherchions une idée originale qui venait de notre cru qui tirerait bien profit de la matière du cours. Ainsi, après y avoir songé pendant plusieurs jours, nous avons réalisé que le cerveau humain envoyait des signaux sous forme d'influx nerveux à différentes parties du corps afin de leur faire effectuer des actions. Les organes, qui sont les principaux acteurs du corps humain, fonctionnent en harmonie mais de manière autonome. Ainsi, chaque organe effectue une tâche bien spécifique qui peut avoir des conséquences ou non sur le fonctionnement des autres organes, mais ce travail s'effectue en même temps et non de manière séquentielle. Autrement dit, chaque organe attend un signal du cerveau pour effectuer sa tâche sans dépendre directement des autres organes. Après avoir fait cette réalisation, nous en sommes venus à la conclusion que c'était un exemple intéressant et quasi-parfait de système parallèle.



En effet, le cerveau lit les données du corps, et envoie aux organes des signaux leur permettant de fonctionner et de réguler les données. Ainsi, chaque organe modifie au sein de leur tâche respective différentes données du corps de manière asynchrone, et le cerveau les lit pour ajuster l'envoi des signaux.

Voici la liste des éléments pris en compte dans le fonctionnement du corps (attributs partagés) ainsi que leur acteur concerné :

- Cœur : rythme cardiaque
- Poumons : rythme respiratoire et taux d'oxygène dans le sang
- Corps : niveau d'énergie et de bonheur
- Intestin : niveau d'excréments

Pour chacune de ces informations, des bornes ont été fixées afin d'assurer la stabilité du système. De plus, une valeur optimale pour chacune d'elles a été déterminée pour permettre la régularisation du système. Voici les valeurs que nous avons choisies (à partir de différentes statistiques trouvées en ligne) pour un sujet âgé d'environ 20 ans :

Donnée	Min	Optimal	Max
Rythme cardiaque	40	100	200
Rythme respiratoire	10	15	65
Taux d'oxygène	0.9	0.95	1
Niveau d'énergie	0.05	0.5	1
Niveau de bonheur	0.05	-	1
Niveau d'excréments	0	0.5	1

Vert : données passives (celles qui sont perturbées et qu'il faut réguler)
Orange : données actives (celles qui permettent de réguler les passives)

À noter que ces valeurs sont approximatives et ne devraient pas être considérées comme étant 100% réalistes. L'essence du travail n'étant pas de nature anatomique, ces valeurs ont peu d'importance dans le cadre du projet.

Cela étant dit, le but principal programme est de réguler les données passives de telle sorte qu'elles s'approchent le plus possible de leur valeur optimale. Ainsi, le cerveau envoie judicieusement des signaux aux organes en fonction des données à réguler, et les organes rétablissent l'ordre progressivement. Toutefois, tout cela ne pourrait arriver sans perturber les données passives. Alors, le programme a été conçu de telle sorte qu'il est possible de les perturber et d'observer les changements

dans les données actives. D'un point de vue graphique, les données actives sont directement liées aux organes et peuvent être visualisées (e.g. cœur qui bat plus ou moins vite).

Signaux

Tel que mentionné précédemment, chaque organe s'exécute sur un fil d'exécution différent, à l'exception du cerveau qui est un cas à part (nous en discuterons plus loin). Ainsi, chacun d'entre eux attendent un signal de la part du cerveau avant d'exécuter leur tâche bien précise et se remettent en attente ensuite (jusqu'à la fermeture du programme). Pour faire parvenir les signaux entre le cerveau et les autres organes, la classe ThreadSignaler a été implémentée. Ainsi, chaque organe à l'exception du cerveau possède son propre fil d'exécution et son propre ThreadSignaler, permettant ainsi l'envoi de signaux et l'exécution des tâches. Cette classe fait usage des variables de condition en C++ (std::condition_variable), qui permettent de bloquer un fil d'exécution jusqu'à ce qu'un signal soit reçu. Ainsi, la classe ThreadSignaler est implémentée comme suit :

```
void ThreadSignaler::notify()
{
    std::lock_guard<std::mutex> lock(m_mutex);
    m_ready = true;
    m_conditionVariable.notify_one();
}

void ThreadSignaler::wait()
{
    std::unique_lock<std::mutex> lock(m_mutex);
    m_conditionVariable.wait(lock, [&] { return m_ready; });
    m_ready = false;
}
```

Comme on peut le constater, trois éléments ont été utilisés :

1- Un mutex : permet d'assurer l'atomicité du signalement
2- Une variable de condition : permet de notifier un fil d'exécution en attente sur la même variable de condition
3- Une variable booléenne : permet d'éviter les faux réveils (voir plus loin)

Il est important de mentionner qu'en C++, les variables de conditions sont implémentées de telle sorte qu'un appel à notify_one() résulte en un signal perdu lorsqu'aucun fil d'exécution n'attend. De plus, l'appel à cette méthode n'est pas atomique, tout comme Monitor.Pulse() en C#. Ainsi, il faut protéger les appels à l'aide d'un mutex. Pour comprendre cette notion, voici comment s'exécute un signalement en C++ avec des variables de condition :

Attente d'un signal :

1- Acquisition du mutex / verrou
2- Mise en attente de la variable de condition. Lors d'un appel à wait(), le verrou est libéré pour permettre le signalement. Lorsqu'un signal est reçu, le verrou est repris.
3- Réinitialisation de la variable booléenne pour indiquer que le signal a été reçu
4- Libération du mutex / verrou

Envoi d'un signal :

1- Acquisition du mutex / verrou
2- Initialisation de la variable booléenne pour indiquer qu'un signal a bel et bien été envoyé
3- Notification de la variable de condition (appel à notify_one())
4- Libération du mutex / verrou

À l'exception de la variable booléenne, ce procédé ressemble drôlement à l'utilisation du mécanisme de Monitor.Pulse / Monitor.Wait en C#. Ainsi, les variables de condition en C++ sont à peu près équivalentes aux moniteurs en C# (d'un point de vue émission et réception de signaux). En effet, une implémentation similaire de ThreadSignaler en C# pourrait ressembler à ceci :

```
void Notify()
{
	lock(obj)
	{
		Monitor.Pulse(obj);
	}
}

void Wait()
{
	lock(obj)
	{
		Monitor.Wait(obj);
	}
}
```

Quant aux faux réveils (*spurious wakeup*), c'est un phénomène qui se produit lorsqu'un fil d'exécution en attente reçoit un signal ne provenant pas du programme lui-même, le réveillant ainsi de manière fautive. Cela se produit parfois avec les fils d'exécution POSIX et Windows afin d'accélérer le traitement des variables de condition. Même si ce phénomène est rare, il n'est généralement pas souhaitable et peut être évité en utilisant une variable booléenne qui indique si le signal provient vraiment du programme lui-même ou non. Ainsi, lors de l'appel à notify(), le flag est levé, et lorsque la variable de condition est réveillée, on vérifie si le flag est bel et bien levé, sinon on relance une attente.

Ressources partagées

Quant aux ressources partagées, ces dernières doivent être synchronisées afin d'assurer un fonctionnement optimal et fiable du corps. Pour ce faire, les données du corps ont été déclarées comme étant atomiques dans une structure nommée BodyInfo. En effet, la librairie standard C++ fournit une interface qui permet d'utiliser et de manipuler atomiquement certains types de données (std::atomic<T>). Malheureusement, les seuls types pouvant être manipulés nativement de manière atomique sont les types intégraux (char, int, etc.). Dans le cas d'une classe T quelconque, seule l'affectation est atomique. Dans le cas des nombres en virgule flottante (float, double), les opérations atomiques ne sont supportées par le langage qu'à partir de C++20. Heureusement, il est possible d'implémenter ces opérations nous-mêmes pour des versions antérieures. C'est pourquoi nous avons créé une classe nommée AtomicNumber permettant de le faire. La classe std::atomic<T> fournit deux méthodes permettant d'échanger la valeur de la variable pour une autre valeur. Ces méthodes s'appellent compare_exchange_weak et compare_exchange_strong. Tout comme pour les variables de condition, ces échanges sont soumis à un problème similaire aux faux réveils où la valeur attendue n'est pas exactement la bonne même si la méthode indique le contraire (rare mais possible). Ainsi, il faut boucler pour s'assurer que la valeur est bel et bien la bonne avec compare_exchange_weak, ou tout simplement utiliser compare_exchange_strong qui fait exactement le même travail (plus lent). Dans notre cas, l'opération est effectuée jusqu'à ce que la valeur retournée soit la bonne, rendant ainsi l'opération atomique.

Cerveau

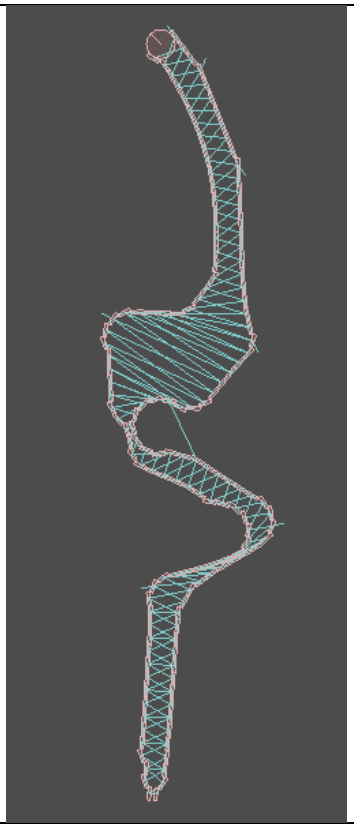
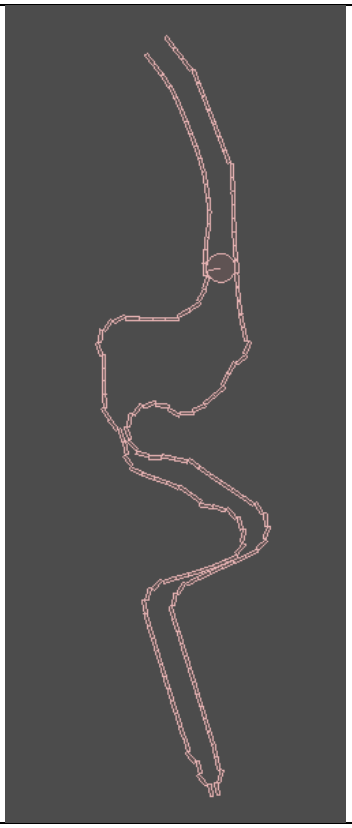
Tel que mentionné précédemment, chaque organe s'exécute sur un fil d'exécution différent, à l'exception du cerveau qui est un cas à part. En effet, le cerveau s'exécute sur le fil d'exécution principal du programme. Dans une application de type jeu, la boucle d'exécution principale effectue trois tâches précises :

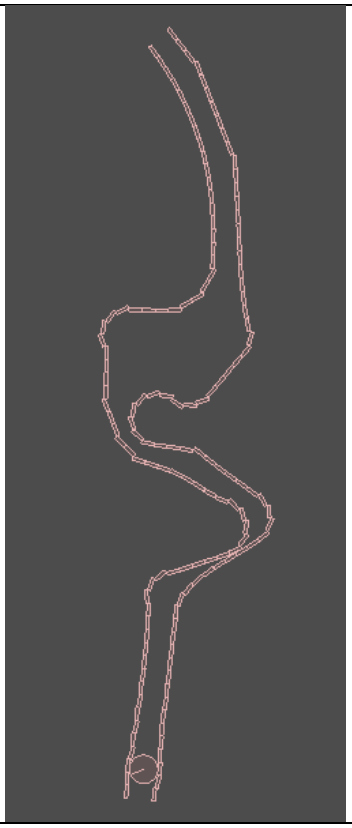
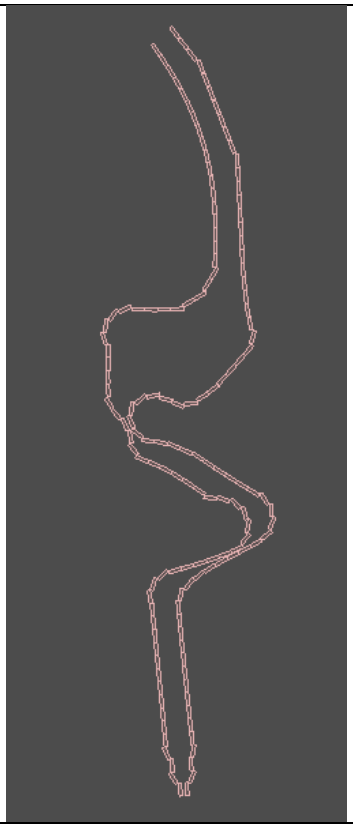
1. Gérer les événements (clavier et autres)
2. Mettre à jour la logique du jeu
3. Afficher les éléments du jeu à l'écran

Étant donné que le cerveau doit envoyer des signaux à certains instants précis, par exemple pour faire battre le cœur à son rythme, il doit calculer le temps écoulé et c'est pourquoi avoir des mises à jour à temps fixe facilite ce processus.

Système digestif

Au départ, nous voulions créer le système digestif de telle sorte que l'on puisse observer le processus de digestion en temps réel. Autrement dit, nous voulions créer un système *physics-based* dans lequel on verrait des corps mous (*soft body*) se faire réduire en particules lors de la digestion et expulser par l'intestin. Évidemment, nous avions l'intention de paralléliser ce système. Nous avons réussi à créer ce système et nous pouvions y insérer des objets. Voici de quoi avait l'air le système :

Instant 1	Instant 2	Instant 3
		

Instant 4	Instant 5	Instant 6
		

L'instant 1 expose les différents joints (en bleu) du système physique. Quant aux blocs rouges, ce sont les corps durs (*rigid body*) liés par les joints du système. Nous avions aussi intégré deux valves / joints qui permettaient de refermer la sortie de l'estomac et de l'intestin. Toutefois, il restait beaucoup trop de travail à faire pour implémenter le système de particules et nous avons décidé de tout laisser tomber. Le système utilisait Box2D comme engin graphique et SFML comme bibliothèque d'affichage. Le projet ne comporte donc aucun élément physique, contrairement à ce qui était prévu au départ.

Mode de déploiement

Compilation

Comme bibliothèque d'affichage, nous avons utilisé SFML (« Simple and Fast Multimedia Library »), qui est un wrapper multiplateforme autour d'OpenGL permettant l'affichage en 2D. Contrairement à SDL (« Simple DirectMedia Layer »), qui fournit des fonctionnalités similaires, SFML adopte un modèle orienté objet et est entièrement programmé en C++ plutôt qu'en C. Cela étant dit, ce genre de wrapper n'offre aucune fonctionnalité superflue, dans le sens où seules les fonctionnalités d'entrée-sortie et d'affichage sont fournies. Ainsi, il fallait développer l'engin de l'application de zéro (boucle de jeu et autres) et programmer tous les mécanismes nécessaires par nous-mêmes. Heureusement, j'avais (Jérémi) déjà programmé une multitude de programmes similaires auparavant, ce qui nous a permis de prendre une longueur d'avance et de sauver du temps.

SFML n'étant évidemment pas « installé » sur les postes de travail de l'université, nous avons décidé de vous fournir le nécessaire pour pouvoir compiler et rouler le programme en 64-bit sur Windows. Pour compiler le programme, il suffit d'ouvrir la solution IFT630.Final.sln et de lancer le programme, tout simplement (nous avons configuré la solution pour inclure et linker le nécessaire).

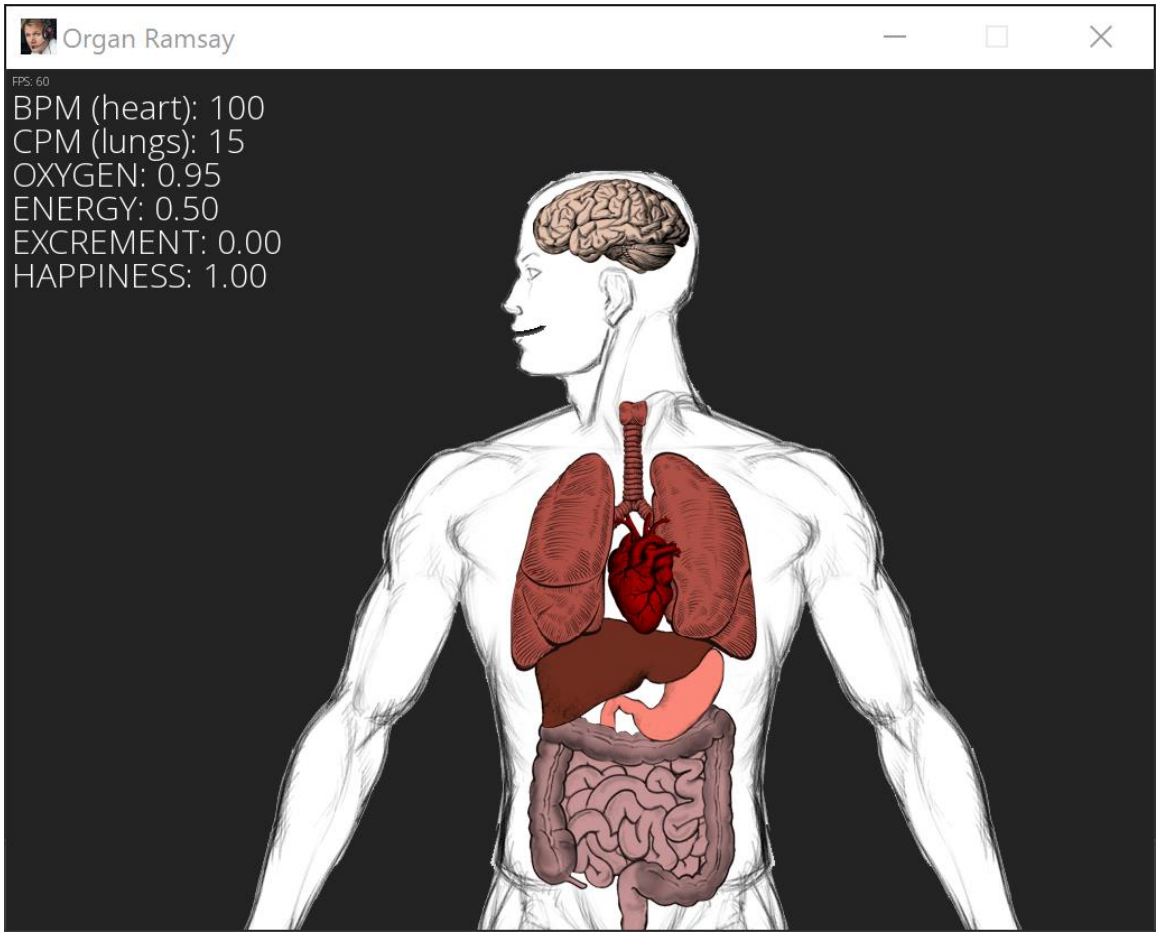
Sinon, si la compilation vous importe peu, nous vous avons aussi fourni l'exécutable 64-bit prêt à être utilisé. Si jamais vous tenez à la version 32-bit, nous pouvons vous l'envoyer aussi sans problème. Toutefois, afin de sauver de l'espace, nous avons décidé de l'omettre.

Correction

Pour votre correction, le code intéressant se trouve dans les répertoires Game/Human et dans Engine/ (AtomicNumber et ThreadSignaler). Plus particulièrement, Brain.cpp contient la logique de régulation des organes, BodyInfo.h contient les données actives et passives, et les classes dérivées de Organ contiennent le code parallèle.

Exécution

Lorsque le programme est exécuté, la fenêtre suivante se lance :



Pour déplacer la caméra, il suffit de maintenir le clic gauche de la souris et de déplacer. Pour zoomer la caméra, il suffit de bouger la molette de la souris.

En haut à gauche, les données passives et actives du corps sont affichées en temps réel. Pendant ce temps, il est possible d'observer le fonctionnement des organes dans l'affichage.

- Cerveau : clignote à chaque signal envoyé
- Cœur : gonfle et change de couleur à chaque battement
- Poumons : gonflent et change de couleur à chaque respiration
- Estomac : ne fait rien du tout
- Intestin : change de teinte plus le niveau d'excréments est élevé
- Sourire : monte ou descend en fonction du niveau de bonheur

Quant aux données passives, voici l'impact des organes sur chacune d'elles :

- À chaque fois que le cœur bat ou que les poumons respirent, 0.5% d'énergie est utilisée (0.005)
- À chaque fois que les poumons respirent, le taux d'oxygène augmente de 0.5% (0.005)
- À chaque seconde, le taux d'oxygène baisse de 0.2% (0.002)
- À chaque fois que l'estomac est sollicité, une quantité aléatoire de nourriture est digérée en énergie et le reste en excréments
- À chaque fois que l'intestin est sollicité, il prend 5 secondes pour se préparer et sort tous les excréments qu'il contient

Ainsi, voici le comportement par défaut du programme :

1- Le niveau d'énergie baisse à cause du cœur et des poumons
2- Une quantité aléatoire de nourriture est envoyée à l'estomac lorsque le niveau d'énergie devient trop bas
3- Le niveau d'énergie monte et l'intestin se remplit
4- Lorsque l'intestin est trop plein, il se vide

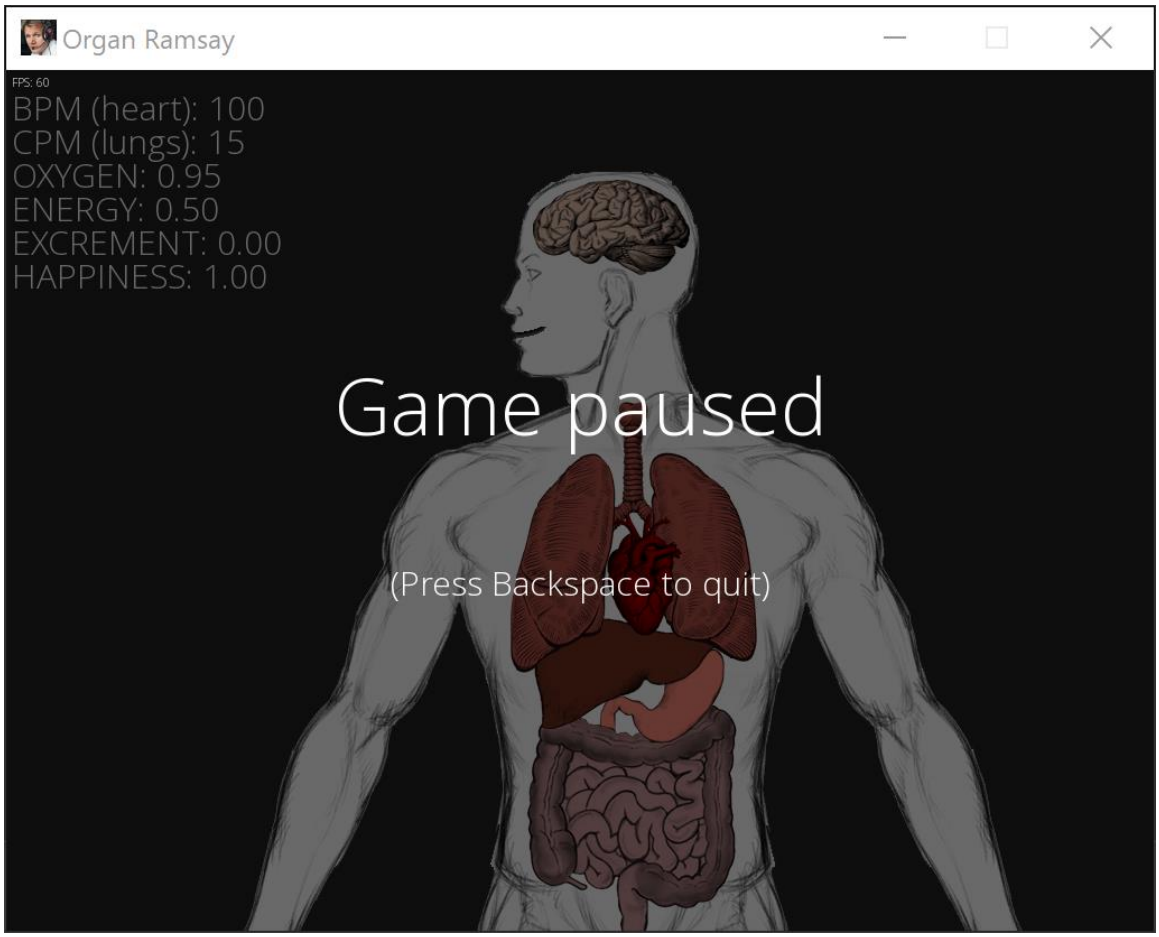
Quant au niveau d'oxygène, le cerveau ajuste le rythme des poumons de telle sorte qu'il gravite toujours autour de 0.95. Quant au cœur, il n'a pas d'impact particulier sauf pour le niveau d'énergie. Pour savoir plus exactement quand ces événements se produisent, veuillez vous référer au code du cerveau et au tableau des valeurs minimales / maximales précédemment énuméré.

Afin de pouvoir mieux observer le comportement du programme, il est aussi possible d'ajuster les données actives et passives manuellement. Au départ, nous voulions ajouter un autre fil d'exécution qui permettrait de lire les entrées au clavier dans la console. Toutefois, en C++, il n'existe aucun moyen portable de vérifier les entrées clavier de manière non bloquante ou de définir un *timeout* sur *std::cin*. Ainsi, il n'aurait pas été possible de mettre fin au fil d'exécution proprement. Alors, nous avons décidé de plutôt gérer les entrées clavier au sein du programme (comme dans un jeu), et de les afficher dans la console. Pour ce faire, il suffit de focus la fenêtre du programme (et non la console!), et d'entrer du texte. Le texte apparait alors en temps réel dans la console, et appuyer sur la touche Enter permet d'envoyer la commande. Voici le format attendu :

[bpm | cpm | oxygen | energy | excrement] = valeur

Veuillez noter que toute commande invalide sera rejetée et que les valeurs sont automatiquement bornées par les valeurs minimales / maximales du tableau.

Pour quitter le jeu, il suffit d'appuyer sur Escape ou tout simplement de le fermer à partir du X en haut à droite de la fenêtre.



Références

https://en.cppreference.com/w/cpp/atomic/atomic

https://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange

https://en.cppreference.com/w/cpp/thread/condition_variable