B5 - C++ avancé

B-RPC-501

Type R avancé

Un moteur de jeu qui rugit !

4.0-wip

EPITECH.

Type R avancé

nom binaire: r-type_server, r-type_client C++
Langue: +
outil de construction: cmake (+ gestionnaire de paquets optionnel)

- La totalité de vos fichiers sources, à l'exception de tous les fichiers inutiles (fichiers binaires, temp, obj,...), doivent être inclus dans votre livraison.

Ce projet de la**C++ avancé**L'unité de connaissances vous présentera le développement de jeux vidéo en réseau, tout en vous donnant l'occasion d'explorer des sujets de développement avancés en profondeur ainsi que d'apprendre de bonnes pratiques d'ingénierie logicielle.

Votre objectif : implémenter un serveur multithread et un client graphique pour un jeu appelé**Type R**, en utilisant un moteur de votre propre conception.

Dans un premier temps, vous développerez l'architecture de base du jeu et livrerez un prototype fonctionnel, et dans un second temps, vous développerez plusieurs aspects du prototype vers de nouveaux horizons, en explorant des domaines spécialisés de votre choix parmi une liste de possibilités proposées.

R-TYPE, LE JEU

Pour ceux d'entre vous qui ne connaissent peut-être pas ce jeu vidéo à succès, qui représente d'innombrables heures perdues de notre enfance,iciest une petite introduction.

Ce jeu est officieusement appelé unShmup horizontal(par exemple*Tue les*), et bien que R-Type ne soit pas le premier de sa catégorie, celui-ci a été un énorme succès parmi les joueurs dans les années 90, et a eu plusieurs portages, spin-offs et remakes 3D sur des systèmes modernes.

D'autres jeux similaires et bien connus sont les*Gradius*séries et*Étoile flamboyante*sur Néo Géo.

Comme vous le comprenez maintenant, vous devez**créez votre propre version de R-Type**. . . mais avec un twist non présent dans le jeu original (ni dans les remakes d'ailleurs) : votre version sera un**jeu en réseau**, où un à quatre joueurs pourront combattre ensemble le diabolique Bydos !

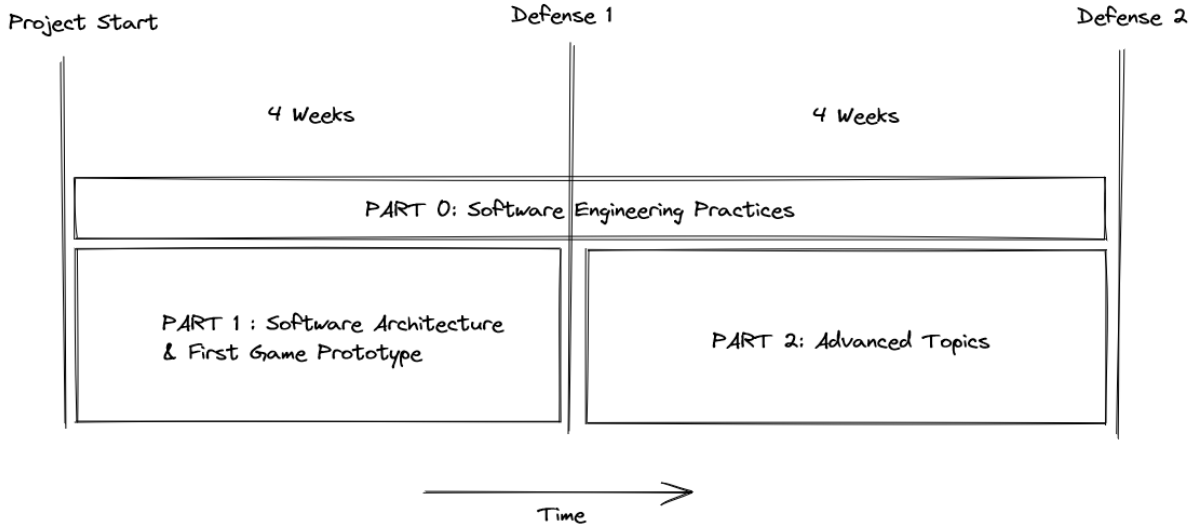
ORGANISATION DU PROJET

Merci de lire attentivement cette page, car il est très important de comprendre le calendrier et l'organisation du projet pour réussir.

Ce projet est divisé en**deux parties**, conduisant chacun à un*Livraison*et évalués dans un cadre dédié*La défense*.

De plus, il y a aussi un**partie commune**dédiée aux pratiques du génie logiciel - appelée « Part 0 » - qui doivent être mises en œuvre en continu au cours du développement, et évaluées aux deux soutenances.

Le schéma du projet est le suivant :



Suivant cette organisation, le plan de ce document est le suivant :

- **Partie 0**: Pratiques de génie logiciel

Cette partie détaille quelles sont les attentes en termes de pratiques de génie logiciel que votre projet doit avoir. Des sujets tels que*documentation du logiciel*,*construire l'outillage du système*,*Gestion des dépendances tierces*,*exigences multiplateformes*,*Flux de travail VCS*, et*emballage*seront abordés.

Ces pratiques doivent être un effort continu, et non quelque chose fait à la toute fin du projet. A ce titre, chaque soutenance de projet tiendra compte des travaux qui ont été réalisés sur ce sujet.

- **Partie 1**: Architecture logicielle & premier prototype de jeu

L'objectif de la première partie est de développer les bases de base de votre moteur de jeu en réseau, vous permettant de créer et de livrer un premier prototype de jeu fonctionnel, ou*Produit minimum viable*(alias.*MVP*).

Évidemment, cette partie doit être faite en premier : vous**DEVOIR**avoir un jeu en réseau qui fonctionne pour pouvoir passer à la seconde partie !

Le délai pour cette première livraison et soutenance est de 4 semaines après le début du projet.

- **Partie 2**: Sujets avancés : Élargissez-vous à de nouveaux horizons

Cette deuxième partie vous permettra d'approfondir certains aspects de votre MVP :*Architecture logicielle avancée*,*Mise en réseau avancée*, et/ou*Gameplay avancé et conception de jeu*.

Vous aurez la possibilité de choisir les aspects que vous souhaitez explorer, ce qui vous conduira à la livraison finale du projet.

Le délai pour cette livraison finale et soutenance est de 4 semaines après la livraison MVP précédente, soit 8 semaines après le début du projet.

CONSEILS GÉNÉRAUX

Pour démarrer le projet en toute sécurité, assurez-vous de bien comprendre les attentes définies par la « Partie 0 », et concentrez-vous sur les**première partie**d'abord (tout en jetant juste un coup d'œil au**deuxième partie**pour vous donner une idée de ce qui pourrait suivre).

Puis, une fois la première livraison terminée, passez à la deuxième partie pour faire passer votre prototype au niveau supérieur !

PARTIE 0 : PRATIQUES DE GÉNIE LOGICIEL

Avoir de bonnes pratiques en génie logiciel est indispensable pour tout type de projet professionnel sérieux. C'est ainsi que vous construisez de vrais produits, et pas seulement des "projets de jouets".

Comme ces pratiques sont censées être un processus continu, les éléments de cette section sont considérés comme des conditions préalables au projet et seront évalués lors des deux soutenances.

SYSTÈME DE CONSTRUCTION, DÉPENDANCES, CROSS PLATEFORME, VCS, PACKAGING

- Le projet**DEVOIR**utilisation**CMake**comme son système de construction. Aucun Makefile n'est autorisé.
- Le projet**DEVOIR**être autonome en ce qui concerne les bibliothèques tierces pouvant être utilisées.

Plus précisément, cela signifie que le projet peut être construit et exécuté**sans pour autant**modifier quoi que ce soit sur le système : il ne repose pas sur des bibliothèques préinstallées à l'échelle du système ou sur des en-têtes de développement (par exemple, installés manuellement avec*apt install*ou*installation dnf*), à l'exception des compilateurs et bibliothèques C++ standard et de certaines bibliothèques évidentes de système graphique ou audio spécifiques à la plate-forme (*X11*,*Wayland*,*OpenGL*la mise en oeuvre,*ALSA*bibliothèques, etc).

En conséquence, le projet**DEVOIR**utiliser une méthode appropriée pour gérer les dépendances tierces, qui pourraient être**UNE**du suivant:

- *Sous-modules Git*;
- *CMake Fetch_Package*ou*Projet_externe*fonctionnalités ;
- Un gestionnaire de packages dédié :*Conan*,*Vcpkg*, etc.

Tu choisis. Il n'y a pas de bonne ou de mauvaise solution, chacune a ses avantages et ses inconvénients.

Copier le code source complet des dépendances directement dans votre référentiel n'est PAS considéré comme une méthode appropriée de gestion des dépendances, même si cela permet toujours au projet d'être autonome.

Au final, l'évaluateur devrait être capable de construire et d'exécuter votre projet "Out of the Box", en utilisant uniquement CMake et éventuellement quelques commandes supplémentaires pour configurer le gestionnaire de packages.

- Le projet**DEVOIR**être**Multiplateforme**

Il doit être compilé et exécuté sous Windows (en utilisant le compilateur Microsoft Visual C++) ET Linux (en utilisant GCC) au moins.

Sous-système Windows pour Linux(alias.*WSL*) n'est PAS une façon de dire qu'un programme prend en charge Windows !

C'est encore mieux si le projet peut être compilé à l'aide d'un compilateur et/ou d'une plate-forme alternatifs. Vous découvrirez probablement de nouveaux avertissements du compilateur et des problèmes dont vous n'étiez peut-être pas conscients.

Exemples:*Bruit*sous Windows ou Linux,*MinGW*ou*Cygwin*sous Windows,*Pomme CLang*sur macOS

De plus, vous avez probablement déjà entendu parler de*Docker*. Il peut être utilisé pour construire et exécuter des projets C++, et en particulier, tout type de programme serveur (y compris un serveur de jeu). Ceci est généralement utilisé dans les workflows CI.

- Adéquat**Flux de travail VCS**devrait être utilisé pour le développement

Vous devez adopter les bonnes méthodologies et pratiques Git : branches de fonctionnalités, demandes de fusion, problèmes, balises pour les jalons importants, petits commits, description correcte des commits, etc.

Méfiez-vous de l'utilisation excessive de Liveshare ouprogrammation de foulméthodologie, qui pourrait conduire à une mauvaise utilisation de Git : un seul committer, des contributeurs fantômes, pas de branches, etc.

C'est encore mieux si une forme de base de workflows CI/CD pour l'automatisation est utilisée.

Exemples:*Actions GitHub*,*CercleCI*,*CI/CD Gitlab*

Un autre aspect à considérer est l'utilisation d'outils spécifiques tels que les linters C++, les analyseurs statiques ou les formateurs, car ils aident beaucoup à appliquer un style de formatage commun ou à rechercher des problèmes supplémentaires non détectés par les compilateurs.

Exemples:*bien rangé*,*format clang*

- Le projet peut être**emballé/livré/installé**autonome, sans nécessiter de construction à partir des sources, et avec une version correcte.

Ceci est définitivement utile pour les utilisateurs finaux ou les passionnés intéressés par le résultat du projet, et non par la source. Tu **DEVOIR**fournir un moyen de créer des archives tar ou des installateurs pour le jeu.

Exemples:*CMake CPackComment*, ou cible personnalisée dans CMake à l'aide des commandes zip/tar

Notez que les fichiers binaires finaux ne doivent rien référencer dans le répertoire source, et en particulier les ressources du jeu : ceux-ci doivent être accompagnés de fichiers binaires.

De plus, des archives tar ou des programmes d'installation prêts à l'emploi peuvent être mis à disposition sur le site en ligne du projet.

Les packages créés avec un pipeline CI peuvent être automatiquement ajoutés aux pages de publication de GitHub

DOCUMENTATION

La documentation n'est pas votre tâche préférée, nous le savons tous ! Cependant, la documentation est également la première chose que vous voudrez rechercher si vous avez besoin de vous plonger dans un nouveau projet.

L'idée est de fournir les éléments de documentation essentiels que vous seriez heureux de voir si vous vouliez contribuer à un nouveau projet.

Tu**DEVOIR**écrire la documentation en anglais.

Ceci comprend:

- D'abord et avant tout, le**LISEZMOI**dossier !

C'est la première chose que chaque développeur verra : c'est la façade publique de votre projet. Mieux vaut donc le faire correctement.

Il doit être court, agréable, pratique et utile, avec des informations typiques telles que l'objectif du projet, les dépendances/exigences/ plates-formes prises en charge, les instructions de construction et d'utilisation, la licence, les auteurs/contacts, des liens utiles ou des informations de démarrage rapide, etc. .

Inspirez-vous des projets existants ! Que trouvez-vous utile lorsque vous recherchez sur Google une nouvelle bibliothèque ou un nouveau projet à utiliser ?

- La**Documentation développeur**

C'est la partie que vous n'aimez pas. Mais pensez-y : son objectif principal est d'aider les nouveaux développeurs à plonger dans le projet et à comprendre son fonctionnement de manière large (et non dans les moindres détails, le code est là pour ça). Pas besoin d'être exhaustif ou verbeux, il faut que ce soit pratique avant toute autre chose.

Demandez-vous : que voudriez-vous savoir si vous vouliez commencer le développement d'un projet de jeu vidéo ?

Vous aurez généralement besoin des types d'informations suivants :

- Diagrammes architecturaux (une vue typique « couche/sous-système » courante dans les jeux vidéo)
- Aperçus et description des principaux systèmes (et comment cela se matérialise dans le code)
- Tutoriels et tutoriels.

Les directives de contribution et les conventions de codage sont également très utiles. Ils permettent aux nouveaux développeurs de connaître les conventions, les processus et les attentes de votre équipe.

Avoir une bonne documentation pour les développeurs démontrera à l'évaluateur que vous avez une bonne compréhension de votre projet, ainsi que la capacité de bien communiquer avec d'autres développeurs.

- La documentation doit être**disponible et accessible de manière moderne**.

Les documents tels que PDF ou .docx ne sont pas vraiment la façon dont la documentation est livrée de nos jours. Il est plus pratique de lire la documentation en naviguant en ligne à travers un ensemble de pages structurées correctement interconnectées, avec un plan d'accès rapide quelque part, une barre de recherche utile et un contenu indexé par les moteurs de recherche.

Des outils générateurs de documentation sont conçus pour cela, permettant de générer un site web statique à partir de fichiers de documentation sources. Les Wikis en ligne sont également une alternative intéressante orientée vers le travail collaboratif.

Exemples:*réduction*,*texte restructuré*,*Sphinx*,*Gitbook*,*Doxygène*, Wiki, etc.
Il existe aujourd'hui de nombreuses possibilités, rendant les documents hérités définitivement obsolètes.

Un exemple de documentation de projet bien faite :https://emscripten.org.

Celles-ci ont été générées à partir de fichiers source reStructuredText, stockés dans le même référentiel que le code du projet. Une action CI génère le site Web statique à partir de fichiers source, puis est finalement poussé vers un serveur ou vers des pages GitHub.

- Documentation du protocole

Ce projet est un jeu en réseau : à ce titre, le protocole de communication est un élément critique du système.

La documentation du protocole réseau doit décrire les différentes commandes et paquets envoyés sur le réseau entre le serveur et le client. Quelqu'un**DEVRAIT**être capable d'écrire un nouveau client pour votre serveur, simplement en lisant la documentation du protocole.

Les protocoles de communication sont généralement plus formels que la documentation habituelle des développeurs, et les documents classiques sont acceptables à cette fin. Rédaction d'unRFCest une bonne idée.

PARTIE 1 : ARCHITECTURE LOGICIELLE ET PROTOTYPE DE JEU

La première partie du projet se concentre sur la construction des fondations de base de votre moteur de jeu et sur le développement de votre premier prototype R-Type (sic).

Le jeu**DEVOIR**sera jouable à la fin de cette partie : avec un joli champ d'étoiles en arrière-plan, les vaisseaux spatiaux des joueurs affrontent des vagues de Bydos ennemis, chacun tirant des missiles pour tenter d'abattre l'adversaire.

Le jeu**DEVOIR**être un jeu en réseau : chaque joueur utilise un client distinct sur le réseau, se connectant à un serveur ayant l'autorité finale sur ce qui se passe réellement dans le jeu.

Le moteur de jeu sous-jacent**DEVOIR**être correctement structuré, avec des sous-systèmes et/ou des couches visibles pour le rendu, la mise en réseau, la logique, etc.

Les conditions préalables à ce projet sont expliquées dans la partie précédente sur les pratiques de génie logiciel, et en particulier les attentes concernant les outils du système de construction, le gestionnaire de packages, les exigences multiplateformes, le packaging et la documentation.

SERVEUR

Le serveur implémente toute la logique du jeu. Il agit comme la source faisant autorité des événements de logique de jeu dans le jeu.

Ce**DEVOIR**être multithread. Le serveur**NE DOIT PAS**bloquer ou attendre les messages des clients, car le jeu DOIT tourner image après image sur le serveur !

La qualité de vos abstractions sera fortement évaluée lors de la soutenance finale, alors payez**proche**attention à eux.

Tu**PEUT**utilisation*Asie*ou*Boost.Asio*pour la mise en réseau, ou s'appuyer sur une couche réseau spécifique au système d'exploitation avec une encapsulation appropriée (en gardant à l'esprit la nécessité pour le serveur d'être multiplateforme).

CLIENT

Le client est le terminal d'affichage du jeu.

Ce**DEVOIR**contenir tout ce qui est nécessaire pour afficher le jeu et gérer les entrées des joueurs.

Il est fortement recommandé que le client exécute également le code logique du jeu, pour éviter d'avoir trop de problèmes dus au décalage du réseau. En tout cas le serveur**DEVOIR**avoir autorité sur ce qui se passe à la fin.

Vous pouvez utiliser le**SFML**pour le rendu/audio/entrée/réseau, mais d'autres bibliothèques peuvent être utilisées (telles que**SDL** par exemple). Cependant, les librairies au périmètre trop large, ou les moteurs de jeux existants (*UE*,*Unité*,*Godot*, etc.) sont interdits.

Voici une description de l'officiel**Type R**filtrer:

- 1 : Joueur
- 2 : Monstre
- 3 : Monstre (qui génère un bonus à la mort)
- 4 : Missile ennemi
- 5 : Missile du joueur
- 6 : Obstacles de scène
- 7 : Tuile destructible
- 8 : Arrière-plan (champ d'étoiles)

PROTOCOLE

Tu**DEVOIR**concevoir un protocole binaire pour les communications client/serveur.

Un protocole binaire, contrairement à un protocole texte, est un protocole dans lequel toutes les données sont transmises au format binaire, soit brutes (telles quelles à partir de la mémoire), soit avec un codage spécifique optimisé pour la transmission de données.

En d'autres termes, avant la transmission, les données ne sont PAS converties en chaînes de texte en clair séparées par un caractère de fin de ligne ou un autre caractère (typique _const char*_ ou*std ::chaîne*). Par exemple, un entier est transmis tel quel, et non comme une représentation textuelle du nombre : c'est ce qu'on appelle le mode « binaire ».

Faites attention à la*Boost :: sérialisation*bibliothèque, et en particulier*booster :: archive :: texte*, car il pourrait être utilisé pour sérialiser les données en données texte avant la transmission, ce qui n'est pas l'objectif que vous souhaitez atteindre avec un protocole en mode binaire.

Tu**DEVOIR**utiliser UDP pour les communications entre le serveur et les clients. Une deuxième connexion utilisant TCP peut être tolérée, mais vous**DEVOIR**fournir une justification solide. Dans tous les cas, TOUTES les communications en jeu **DEVOIR**utiliser UDP.

UDP fonctionne différemment de TCP, assurez-vous de bien comprendre la différence entre la communication orientée datagramme (ou messages limités) et la communication orientée flux.

Pensez à la complétude de votre protocole, et en particulier à la gestion des messages erronés. Ces messages ou paquets malformés**NE DOIT PAS**conduire le client ou le serveur à planter.

Tu**DEVOIR**documentez votre protocole. Voir la section précédente sur la documentation pour plus d'informations sur ce qui est attendu pour la documentation du protocole.

BIBLIOTHÈQUES

Tu**PEUT**Utilisez le**SFML**côté client, ou d'autres bibliothèques similaires telles que**SDL**par exemple. Cependant, les librairies au périmètre trop large, ou les moteurs de jeux existants (*UE*,*Unité*,*Godot*, etc.) sont interdits.

La portée de la bibliothèque tierce utilisée sur le client doit être limitée à : graphique, audio, entrée, mise en réseau et threading.

Tu**PEUT**utilisation*Augmenter*bibliothèques sur le serveur et/ou le client. Cependant, vous devez justifier chaque utilisation spécifique de la bibliothèque Boost.

Pour le réseautage, vous**PEUT**utilisation*Asie*ou*Boost.Asio*pour votre serveur et/ou le client. Comme alternative, sur le client, vous pouvez utiliser les fonctionnalités de mise en réseau fournies par la bibliothèque tierce que vous utilisez comme décrit précédemment, mais UNIQUEMENT pour le client (pas de SFML ou similaire sur le serveur, évidemment).

*Asie*et*Boost.Asio*sont les mêmes bibliothèques, la première est simplement une version autonome.

Vous pouvez également utiliser les fonctionnalités de mise en réseau de bas niveau fournies par le système d'exploitation.

Sachez que les API de mise en réseau de bas niveau ne sont généralement pas portables entre les systèmes d'exploitation et, à ce titre, une encapsulation solide est nécessaire pour obtenir une véritable multiplateforme, avec des backends différents pour chaque système pris en charge.

En règle générale, avant d'utiliser une bibliothèque non mentionnée dans ce document, parlez-en à votre équipe pédagogique pour être sûr que tout ira bien.

MOTEUR DE JEU

Vous expérimentez maintenant le C++ et la conception orientée objet depuis un an. Cette expérience signifie qu'il devrait maintenant être évident pour vous de créer**abstractions**et écris**code réutilisable**.

Par conséquent, avant de commencer à travailler sur votre jeu, il est important que vous commenciez par créer un**moteur de jeu**!

Le moteur de jeu est la base de tout jeu vidéo : il détermine la manière dont vous représentez un objet dans le jeu, le fonctionnement du système de coordonnées et la manière dont les différents systèmes de votre jeu (graphiques, physiques, réseau...) communiquent.

Lors de la conception de votre moteur de jeu,**découplage**est la chose la plus importante sur laquelle vous devriez vous concentrer. Le système graphique de votre jeu n'a besoin que de l'apparence et de la position d'une entité pour le rendre. Il n'a pas besoin de savoir combien de dégâts il peut infliger ou à quelle vitesse il peut se déplacer ! De même, un système physique n'a pas besoin de savoir à quoi ressemble une entité pour mettre à jour sa position. Réfléchissez aux meilleurs moyens de découpler les différents systèmes de votre moteur.

Pour ce faire, nous vous recommandons de jeter un œil à Entity-Component-System**modèle architectural**, aussi bien que**Médiateur**design pattern. Mais il existe bien d'autres manières d'implémenter un moteur de jeu ! Sois créatif !

GÉNÉRAL

Le client**DEVOIR**afficher un arrière-plan à défilement horizontal lent représentant l'espace avec des étoiles, des planètes. . . C'est le champ d'étoiles.

Le défilement du champ stellaire ne doit PAS être lié à la vitesse du processeur. Au lieu de cela, vous**DEVOIR**utiliser des minuteries.

Joueurs**DEVOIR**pouvoir se déplacer à l'aide des touches fléchées.

Le serveur**DEVOIR**être multithread.

Si un client tombe en panne pour une raison quelconque, le serveur**DEVOIR**continuer à travailler et**DEVOIR**avertir les autres clients du même jeu qu'un client a planté.

Type Rles sprites sont disponibles gratuitement sur Internet, mais un ensemble de sprites est disponible avec ce sujet.

Les quatre joueurs dans un jeu**DEVOIR**être distinctement identifiable (via la couleur, le sprite, etc.)

Là**DEVOIR**soyez les esclaves de Bydos dans votre jeu.

- Monstres**DEVOIR**être en mesure d'apparaître au hasard sur la droite de l'écran.
- Le serveur**DEVOIR**avertir chaque client lorsqu'un monstre apparaît, est détruit, tire, tue un joueur, etc. . .

Enfin, pensez à la conception sonore de base de votre jeu. Ceci est important pour une bonne expérience de jeu.

C'est le minimum, vous**DEVOIR**ajoutez tout ce qui, selon vous, rapprochera votre jeu de l'original.

PARTIE 2 : SUJETS AVANCÉS : DÉVELOPPEZ-VOUS VERS DE NOUVEAUX HORIZONS

Maintenant que vous avez un prototype de jeu fonctionnel, il est temps d'explorer de nouveaux terrains et de profiter de l'occasion pour vous plonger dans les sujets de développement de logiciels avancés.

Trois "pistes" sont présentées dans ce document, vous pouvez choisir n'importe quelles pistes et sous-thèmes sur lesquels travailler.

Vous travaillez en équipe et de nombreux sujets présentés sont orthogonaux les uns aux autres. Ainsi, il y a définitivement de la place pour que chaque membre de l'équipe travaille sur la seconde partie, éventuellement en parallèle sur des sujets différents.

Alors, inspirez un bon coup, lisez tout dans cette partie, discutez avec votre équipe, discutez avec votre équipe pédagogique, choisissez vos sujets favoris. . . et résolvez des problèmes du monde réel !

En raison de la portée de cette partie, gardez à l'esprit que tout n'est pas à faire pour valider le projet. Cependant, on s'attend à ce qu'un travail important soit fait sur un ou plusieurs sujets et fonctionnalités.

Ne commencez pas les développements sérieux sur cette partie tant que vous n'avez pas terminé la première partie (première livraison et première soutenance) ! Cependant, vous pouvez certainement lire cette partie au préalable pour avoir une idée de ce qui pourrait suivre : cela peut vous aider à configurer l'architecture initiale de votre moteur de jeu.

ARCHITECTURE AVANCÉE - CONSTRUIRE UN MOTEUR DE JEU

De nos jours, la plupart des jeux sont construits sur un « moteur de jeu ». Cette piste vise à l'extraire de votre prototype actuel, à s'appuyer dessus pour améliorer ses capacités, puis à créer un tout autre jeu à l'aide de votre moteur.

QU'EST-CE QU'UN MOTEUR DE JEU ?

Pour le dire simplement, un moteur de jeu est ce qui reste de votre base de code, une fois que vous avez supprimé les règles du jeu, le monde et les ressources. Un moteur de jeu peut être spécialisé pour un genre spécifique (par exemple, Bethesda Creation Engine a été conçu pour créer des RPG 3D avec une histoire en ramification), ou à usage général (par exemple, Unity).

Tous les moteurs visent à fournir un ensemble d'outils et de blocs de construction aux développeurs de jeux, afin qu'ils puissent réutiliser les fonctionnalités communes, réduisant ainsi le temps de développement.

SUIVRE L'OBJECTIF

Dans cette piste, vous êtes libre de choisir la partie de votre moteur que vous souhaitez améliorer. Cependant, nous vous demanderons de créer un autre exemple de jeu pour prouver que votre moteur et le R-Type sont deux projets différents, et nous utiliserons ce nouveau jeu pour évaluer la réutilisation de votre moteur. Votre moteur doit également être séparé de votre R-Type et suivre les mêmes règles pour l'auto-emballage et la documentation.

Vous n'êtes pas obligé d'implémenter un 2e jeu à part entière. Considérez-le comme une démonstration des fonctionnalités de votre moteur.

Nous ne voulons pas nécessairement que vous développiez toutes les fonctionnalités par vous-même. N'hésitez pas à demander à votre responsable d'unité locale d'utiliser des bibliothèques tierces pour certaines fonctionnalités avancées (rendu de l'interface utilisateur, physique avancée, support matériel, etc.). Tu**DEVOIR**abstrait correctement vos dépendances cependant.

Le reste de cette section présentera les fonctionnalités de la plupart des moteurs. Bien que ceux-ci soient spécifiquement recherchés lors de la soutenance, des fonctionnalités supplémentaires seront également notées.

MODULARITÉ

Un bon moteur ne doit pas prendre plus d'espace mémoire que nécessaire, à la fois sur un disque consommateur et en mémoire pendant l'exécution.

La modularisation est un bon moyen d'y parvenir. Voici quelques manières courantes de construire un moteur de jeu modulaire :

- **temps de compilation**: Le développeur choisit le module qu'il compilera à partir de votre moteur, en utilisant des drapeaux dans son système de construction ou son gestionnaire de paquets.
- **temps de lien**: Le moteur est construit en plusieurs bibliothèques, que le développeur peut ensuite choisir de lier.
- **API de plug-in d'exécution**: Les modules sont construits en tant que bibliothèques d'objets partagés, qui sont soit chargées automatiquement à partir d'un chemin donné, soit configurées au début du jeu, soit chargées/déchargées selon les besoins pendant l'exécution.

Tu**DEVOIR**rendre votre moteur modulaire.

PRINCIPALES CARACTÉRISTIQUES

Moteur de rendu

Comme le moteur de rendu est chargé d'afficher les informations à l'écran, le type de jeux que l'on peut créer dépend étroitement de ses fonctionnalités (module 2.5D ou 3D, système de particules, éléments d'interface utilisateur préfabriqués, . . .).

Moteur physique

L'objectif principal du moteur physique est de gérer les collisions et la gravité. Un moteur plus avancé peut également permettre de faire déformer, casser, rebondir des entités, etc.

Moteur audio

Un moteur audio de base est chargé de lire l'audio de fond pendant le jeu. Les plus avancés incluent certains SFX, du bruit de clic dans l'interface utilisateur au bruit dans le jeu. Ils peuvent également gérer les sons sensibles aux positions dans certains jeux.

Cinématique du jeu

Un système cinématique intégré au jeu prend le contrôle de la caméra et des entités du jeu, pour agir à travers une scène scénarisée. Même s'il pourrait être facilement implémenté dans le code du jeu, il est également courant de le trouver en tant que module de moteur.

Interface Homme-machine

Dans la plupart des cas, le moteur est responsable de la gestion des dispositifs de contrôle, de sorte que le développeur n'a qu'à spécifier lequel il a l'intention d'utiliser. Cela peut aller d'une fonctionnalité simple telle que la configuration du clavier et de la manette de jeu à une fonctionnalité plus avancée ou intégrée, telle que le déclenchement d'un événement en un clic sur un élément de l'interface utilisateur, la prise en charge d'un pavé tactile ou le référencement d'une touche par son emplacement physique au lieu du lettre, pour mieux supporter différentes mises en page.

Interface de passage de messages

Les jeux étant de plus en plus avancés, le nombre de systèmes différents en interaction rend à la fois la synchronisation et la communication plus difficiles à gérer de manière découplée. Une interface de transmission de messages est un moyen de résoudre certains de ces problèmes, car la plupart des interactions sont alors gérées via un système d'événements. La base permet un événement asynchrone simple, tandis que la plus avancée peut prendre la priorité, répondre et même envoyer un message synchrone en cas de besoin.

Gestion des ressources et des actifs

Une bonne gestion des ressources est une tâche fastidieuse, en tant que telle, elle est souvent laissée au moteur, le jeu ne les référençant que par des identifiants ou des noms.
Le schéma commun de gestion des ressources comprend le préchargement (écran de chargement) ou le chargement à la volée, laissant le moteur gérer un cache de ressources.

Script

Dans la plupart des jeux, les comportements d'entité sont reportés à des scripts externes, car ils permettent des cycles de test/développement plus rapides (pas besoin de recompiler). En tant que tel, la plupart des moteurs prennent en charge l'intégration de scripts (soit via un langage personnalisé,

ou intégration d'interprète).

OUTILS

Outre les fonctionnalités susmentionnées, la plupart des moteurs de jeu fournissent également des outils au développeur. Il peut s'agir d'outils de qualité de vie, de débogage ou même d'un IDE à part entière.

Console développeur

La plupart des utilisateurs d'ordinateurs connaîtront cette fonctionnalité dans les jeux hors ligne. Bien qu'il soit principalement utilisé comme moyen de déclencher des actions, des scripts ou des sons lors d'une phase de test, il est souvent laissé dans le jeu pour permettre la "triche" ou pour aider les moddeurs.

Métriques en jeu

Ces fonctionnalités sont le plus souvent utilisées pour l'analyse comparative ou pour déboguer une zone spécifique, et peuvent contenir une gamme de mesures utiles telles que la position mondiale, les images par seconde, le lagomètre (latence du réseau, images perdues), etc.

Éditeur de monde/scène/assets

Cela peut être autonome ou activé par un indicateur spécial au moment de la compilation ou de l'exécution, et est utilisé pour placer des actifs sur le monde, en tirant parti à la fois de la physique et du moteur de rendu pour être aussi proche que possible de ce qui serait possible pendant le jeu. C'est aussi un moyen rapide de créer de nouveaux niveaux ou de configurer une cinématique dans le jeu.

PORTABILITÉ

La plupart des moteurs professionnels visent à être portables, donc un jeu développé pour une plate-forme fonctionnera avec des changements minimes sur une autre, pris en charge par le même moteur.
Pour y parvenir, la plupart des moteurs cachent leurs détails d'implémentation derrière des abstractions personnalisées avec lesquelles le développeur du jeu travaillera ensuite.

À ce stade, votre prototype**devrait**travailler à la fois sur Windows et Linux. Cependant, certaines améliorations supplémentaires pourraient être apportées vers plus de portabilité.

Abstractions tierces

Si vous utilisez des bibliothèques tierces, celles-ci peuvent ne pas être compatibles avec d'autres systèmes (par exemple, Boost peut ne pas être compatible avec la dernière console de jeu, SFML n'est pas compatible avec Android ou iOS au moment de la rédaction, etc.).

Abstractions des fonctionnalités du système

À présent, vous devriez avoir remarqué que chaque système n'expose pas les mêmes fonctions dans son ensemble de bibliothèques (par exemple, les fonctions liées au réseau, . . .).

Abstraction de la bibliothèque standard

Malheureusement, la bibliothèque standard c++ n'est pas toujours entièrement implémentée, selon votre système cible et votre compilateur (comparez la liste d'en-têtes dansSur piedethébergébibliothèques par exemple). En tant que tel, vous souhaiterez peut-être extraire certaines fonctionnalités clés de la bibliothèque standard en ce qui concerne l'interaction avec le système cible (les E/S, la mise en réseau et le système de fichiers sont les abstractions les plus courantes, car elles s'intègrent parfaitement avec d'autres fonctionnalités de qualité de vie).

SERVEUR AVANCÉ / RÉSEAU

Le but de cette piste est de pousser les sous-systèmes serveur et réseau de votre moteur de jeu à un niveau plus mature, correspondant à ce qui est réellement implémenté dans vos jeux favoris.

SERVEUR MULTI-INSTANCE

De nos jours, la plupart des serveurs de jeux dédiés sont capables de gérer plusieurs instances de jeu, et pas une seule comme vous l'avez déjà développé jusqu'à présent.

Avec la possibilité d'avoir plusieurs instances de jeu en cours d'exécution, il est nécessaire d'apporter diverses fonctionnalités de gestion (pour gérer les instances de jeu, les utilisateurs, etc. . .), de gérer correctement la concurrence, ainsi que de fournir des fonctionnalités permettant aux utilisateurs de se connecter au serveur, voir les instances du jeu, les laisser discuter, etc.

L'idée est d'apporter de telles fonctionnalités dans votre projet. Voici quelques sujets sur lesquels vous pourriez enquêter :

- Le serveur est-il capable de faire tourner plusieurs instances de jeu en parallèle ?
- Comment la concurrence est-elle gérée entre ces instances : threads ? fourches ?
- Aviez-vous un système de lobby ou de salles, généralement utilisé pour le matchmaking ?
- Comment gérez-vous les utilisateurs et leurs identités/authentifications ?
- Comment peuvent-ils communiquer entre eux et lancer une partie ?
- Peuvent-ils changer certaines règles de base ?
- Existe-t-il un tableau de bord mondial ?
- Avez-vous un tableau de bord d'administration, ou au moins une interface console en mode texte ? Un admin peut-il kick/bannir un utilisateur spécifique ?

Etc. . . soyez intelligent !

MISE EN RÉSEAU AVANCÉE

Votre pile réseau doit utiliser UDP à ce stade, pour pouvoir étudier ce sujet.

En raison de problèmes de réseau inévitables, tels qu'une faible bande passante, une latence/décalage ou un manque de fiabilité (pertes de paquets, paquets en panne, duplication), le jeu peut être confronté à de sérieux problèmes entraînant une expérience de jeu sous-optimale, ou pire, des désynchronisations ou des plantages complets.

La question générale à laquelle il faut répondre sur ce sujet est de savoir quels mécanismes vous fournissez pour aider à atténuer ces problèmes de réseau, sans sacrifier les performances.

Voici quelques outils utiles pour évaluer le comportement du programme dans des conditions réseau dégradées (simulation de lag, pertes de paquets, duplication, réorganisation, faible bande passante, throttling, etc.) :

- sous Windows :maladroit
- sous Linux :réseau

Assurez-vous de tester votre jeu dans des conditions de réseau dégradées. En utilisant l'un de ces outils, quel est le comportement de votre jeu avec :

- Une perte de paquets de 2 %, une duplication de paquets de 2 % et une réorganisation des paquets de 5 % ?
- Une latence > 150ms ?
- Une bande passante très faible ?

Vous serez probablement surpris par les résultats.

Notez que ces outils peuvent être utilisés pendant la défense, pour vérifier vos affirmations sur la gestion réussie des problèmes de réseau.

Voici quelques aspects techniques sur lesquels vous pouvez enquêter :

- **Emballage de données**

Si des données simples en mémoire sont envoyées telles quelles sur le réseau, cela peut être vraiment inefficace. La disposition habituelle des données en mémoire n'est pas nécessairement optimisée pour une telle transmission, en raison des types de données primitifs qui peuvent être trop volumineux, du remplissage interne des champs de structure optimisé par le compilateur pour l'alignement du processeur, etc.

Conseils : tailles des types de données, conditionnement des données au niveau du bit, alignement des structures et optimisation du remplissage, quantification des données, etc.

- **Compression de données à usage général**

Dans la transmission de données, il y a généralement beaucoup de redondance. L'utilisation d'algorithmes de codage et de compression à usage général est indispensable pour réduire l'utilisation de la bande passante.

Conseils : RLE (Run Length Encoding), encodage Huffman, algorithmes et bibliothèques de type zlib

Question annexe :
- Savez-vous quelle est la bande passante moyenne utilisée par votre jeu ?

- **Techniques d'atténuation des erreurs réseau (abandons de paquets, réorganisation, duplication)**

Le protocole UDP n'est pas fiable et, en cas de forte congestion du réseau, les paquets peuvent être perdus, réorganisés ou même dupliqués. Le jeu pourrait souffrir de ces problèmes, donnant des résultats étranges, ou pire, des désynchronisations complètes.

Vous devez prévoir des moyens d'éviter tout type de problème causé par le manque de fiabilité d'UDP : pertes de paquets, réorganisation et duplication.

Conseils : numéro de séquence du datagramme, tolérance aux pannes

Questions annexes :
- Savez-vous quelle est la taille moyenne de votre datagramme UDP ?
- Avez-vous déjà entendu parler du MTU et de la fragmentation des paquets ?

- **Fiabilité des messages**

Suite au point précédent, même si certains datagrammes UDP peuvent être perdus, divers messages DOIVENT être envoyés/reçus de manière fiable (exemple : connexion à une instance de jeu, un joueur est mort, etc.).

Apportez-vous une solution technique à ce besoin ?

Conseils : canal TCP dédié pour une livraison fiable des messages, modèles "ACK" pour UDP, duplication des messages

COMPENSATION DE LAG

Le problème le plus redouté dans la programmation de réseau de jeu est**latence**, ou plus familièrement,*décalage*.

La latence est mesurée par la**ping**, ou temps d'aller-retour pour qu'un message soit envoyé et répondu entre un client et un serveur. Sur les réseaux locaux, avec une faible latence typique (< 25 ms), ce n'est généralement pas vraiment un problème. Sur internet, la latence peut devenir très élevée (>100ms), et pire, elle peut être extrêmement variable.

Les conséquences de la latence sont ce que vous observez régulièrement dans vos sessions de jeu en réseau préférées : les joueurs et les entités semblent "sauter" des positions de manière erratique, un décalage de délai d'entrée étrange, ou être tué alors qu'il est caché derrière un mur, etc.

Le problème général est qu'en raison de la latence du réseau, tout le monde n'est pas conscient de ce qui se passe en même temps. En tant que tel, les choses peuvent être désynchronisées à un moment donné (le client A, le serveur et le client B ont chacun des informations/états de jeu différents), ou de longs délais peuvent être observés entre vos actions et leurs résultats immédiats.

Cependant, les développeurs de jeux doivent donner l'illusion que tout est**lisse**et**cohérent**. Sujet délicat !

Exemples techniques :*prédiction côté client*,*rapprochement serveur*,*retard d'entrée*,*retour en arrière/rejouer*,*interpolation d'état d'entité*,*extrapolation d'entités distantes*,*simulation déterministe*, etc.

Il existe de nombreuses possibilités pour résoudre ce problème, qui ne sont pas nécessairement mentionnées dans ce document. Pour plus de commodité, un document annexe est fourni avec ce document, donnant des matériaux supplémentaires, des pointeurs et des liens externes pour plonger dans ce sujet.

JEU AVANCÉ / JEU AVANCÉ

Le but de cette piste est d'améliorer les aspects Gameplay et Game Design de votre jeu.

Jusqu'à présent, votre R-Type devrait être un prototype fonctionnel avec un ensemble limité de fonctionnalités et de contenu en termes de gameplay. Changeons cela, et rendons votre jeu plus agréable pour les joueurs finaux ET pour les game designers !

JOUEURS : ÉLÉMENTS DE GAMEPLAY

Bien que R-Type soit un ancien jeu, il s'agit néanmoins d'un jeu complet : avec de nombreux monstres, niveaux, armes et autres éléments de gameplay. Pouvez-vous faire passer votre prototype au niveau supérieur ? Ça devrait être**amusement**jouer.

Éléments que vous pourriez considérer :

- Des monstres, avec des mouvements et des attaques variés. Exemple : monstres de type serpent comme au niveau 2 de R-Type, ou tourelles au sol.
- Des niveaux, avec des thèmes différents, et des rebondissements de gameplay intéressants. Regardez le niveau 3 du R-Type original : vous vous battez avec un énorme vaisseau pendant tout le niveau, ou le niveau 4 avec des monstres laissant derrière eux des traînées solides.
- Patrons. Ils sont légendaires dans la tradition R-Type et ont joué un rôle important dans le succès du jeu. En particulier le premier boss idiomatique, fréquemment représenté sur les boîtes de jeu : le*Dobkératops*.
- Armes. Par exemple, le**Force**fait partie intégrante du jeu original : il peut être fixé à l'avant ou à l'arrière du joueur (et lorsqu'il est fixé à l'arrière, il tire vers l'arrière), peut être « détaché » et agit indépendamment, peut être rappelé vers l'arrière, il protège le joueur des missiles ennemis, permet de tirer des missiles surchargés, etc.
- Règles de jeu : est-il possible de changer ou de peaufiner les règles du jeu ? Pensez à des choses comme le tir amical, la disponibilité des bonus, la difficulté, les modes de jeu ("coop", "versus", "pvp", etc.)
- Conception sonore : elle joue un rôle essentiel dans toute expérience de jeu.

Inspirez-vous du jeu existant :
- Aperçu du gameplay
- Répartition de l'étape individuelle

La notation n'évaluera pas seulement quantitativement. Par exemple, avoir un type d'élément N ou M n'est pas la seule chose qui compte. Avoir des sous-systèmes réutilisables pour ajouter facilement du contenu est tout aussi important. Voir le point suivant.

GAME DESIGNERS : SOUS-SYSTÈMES DE CRÉATION DE CONTENU

Avoir beaucoup de contenu c'est bien, mais avoir de bons sous-systèmes permettant de créer facilement du nouveau contenu c'est encore mieux.

Fournissez-vous de tels systèmes pour permettre aux concepteurs de jeux -**qui ne sont pas nécessairement des développeurs de jeux chevronnés**-pour ajouter facilement de nouveaux contenus dans le jeu ? Doivent-ils connaître le C++ et modifier de nombreux fichiers de votre projet (éventuellement devoir tout recompiler lorsqu'ils ajoutent un nouveau niveau ou boss par exemple) ?

Votre moteur de jeu**DEVRAIT**fournir des API bien définies permettant une extensibilité d'exécution (par exemple, un système de plug-in, des Dll) pour ajouter de nouveaux contenus, des formats standard ou même un langage de script dédié aux comportements des entités de programme (lire : plus simple que C++ !).

Exemple : Le langage de programmation Lua est fréquemment utilisé dans les jeux vidéo.

La question générale à laquelle répondre est la suivante : est-il facile d'ajouter de nouveaux contenus et comportements dans le jeu ? Bien sûr, connaître C++ peut être une exigence, mais votre système est-il suffisamment simple pour être utilisé au quotidien par des personnes ayant des compétences limitées en développement (mais potentiellement avec de grandes compétences en Game Design) ?

Encore mieux, avez-vous des outils d'édition de contenu pour divers éléments du jeu (éditeur de niveaux, éditeur de monstres, etc.) ? Ceux-ci peuvent être des programmes séparés ou intégrés directement dans le jeu principal.

La documentation est un élément essentiel de tout outil de création de contenu, API ou système. Les tutoriels et les tutoriels en particulier sont définitivement ce que les créateurs de contenu recherchent.

Cette partie est tout aussi importante que la précédente : cela ne sert à rien d'avoir 5 niveaux si chaque niveau nécessite de lourdes modifications des internes du jeu de base. Mieux vaut n'avoir que 2 niveaux et démontrer à l'évaluateur que votre sous-système peut être utilisé pour ajouter facilement n'importe quel niveau.

JEU SOLO

Ce projet s'est concentré sur le jeu multijoueur. Mais vous voudrez peut-être aussi jouer à une version solo du jeu.

Votre architecture le permet-elle ? Peut-être voudriez-vous générer automatiquement un serveur local pour ce faire, ou réintroduire la logique du jeu dans le client.

Jouer à la version solo ne signifie pas nécessairement que vous êtes seul : avoir des Bots pour d'autres joueurs pourrait être une très bonne idée. Faire une IA pour un jeu R-Type peut aussi être quelque chose de très intéressant à faire !