{EPITECH}

TYPE R

< UN MOTEUR DE JEU QUI RUGIT ! />

TYPE R

Préliminaires

binaire : r-type_server, r-type_client
langage : C++
build : CMake, gestionnaire de paquets

Ce projet de l' unité de connaissances C++ avancé vous présentera le développement de jeux vidéo en réseau et vous donnera l'occasion d'explorer des techniques de développement avancées ainsi que d'apprendre de bonnes pratiques d'ingénierie logicielle.

L'objectif est d'implémenter un serveur multithread et un client graphique pour un jeu vidéo hérité bien connu appelé R-Type, en utilisant un moteur de jeu 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 étendrez plusieurs aspects du prototype au niveau supérieur, en explorant des domaines spécialisés de votre choix à partir d'une liste d'options proposées.

R-Type, le jeu

Pour ceux d'entre vous qui ne connaissent peut-être pas ce jeu vidéo à succès, voici c'est une petite introduction.

Ce jeu est appelé officieusement un Shmup Horizontal (ou simplement, un Shoot'em'up), et bien que R-Type ne soit pas le premier de sa catégorie, il a connu un énorme succès auprès des joueurs dans les années 90 et a connu plusieurs portages, spin-offs et remakes 3D sur les systèmes modernes.

D'autres jeux similaires et bien connus sont la série Gradius et Blazing Star sur Neo Geo.

Dans ce projet, vous devez créer votre propre version de R-Type, avec des exigences supplémentaires non présentes dans le jeu original :

- Il DOIT s'agir d'un jeu en réseau, où plusieurs joueurs pourront combattre le maléfique Bydos ;
- Sa conception interne DOIT démontrer les caractéristiques architecturales d'un véritable moteur de jeu.

Organisation du projet

Ce projet est divisé en deux parties, chaque partie menant à une Livraison et évaluée dans une Soutenance dédiée.
De plus, il existe également une partie commune qui sera évaluée lors des deux soutenances.

Ce document est structuré selon le plan suivant :

Partie commune : Exigences en matière d'ingénierie logicielle, de documentation et d'accessibilité

Cette partie définit les attentes de votre projet en matière d'ingénierie logicielle, de documentation et d'accessibilité. Des sujets tels que la documentation technique, les outils de build, la gestion des dépendances tierces, le workflow de développement et le packaging seront abordés.

Cette partie doit être un effort continu, et non une étape réalisée en toute fin de projet. Ainsi, chaque soutenance de projet tiendra compte du travail effectué sur ce sujet.

Partie 1 : Architecture logicielle et premier prototype de jeu

L'objectif de la première partie est de développer les fondations principales et l'architecture logicielle de votre moteur de jeu en réseau, vous permettant de créer et de livrer un premier prototype de jeu fonctionnel.

La date limite pour cette première livraison et soutenance est de 4 semaines après le début du projet.

Partie 2 : Sujets avancés : du prototype de jeu à l'infini et au-delà !

L'objectif de cette deuxième partie est d'améliorer différents aspects de votre prototype et d'amener votre version finale à un niveau de maturité supérieur. Trois axes techniques sont proposés : architecture logicielle avancée, réseaux avancés et/ou gameplay et conception de jeux avancés.

Vous aurez l'opportunité de choisir les sujets sur lesquels vous souhaitez travailler, vous conduisant finalement à la livraison finale du projet.

La date limite pour cette livraison finale et soutenance est de 3 semaines après la première livraison, pour un total de 7 semaines complètes pour l'ensemble du projet

Exigences en matière d'ingénierie logicielle

Le projet DOIT utiliser CMake comme système de construction.

Le projet DOIT utiliser un gestionnaire de packages pour gérer les dépendances tierces.

Le gestionnaire de paquets peut être l'un des suivants :

- Conan
- Vcpkg
- CMake CPM

L'objectif est que le projet soit entièrement autonome : il doit être compilé et exécuté sans aucune modification du système. Autrement dit, le projet ne doit pas s'appuyer sur des bibliothèques installées à l'échelle du système ni sur des en-têtes de développement, à l'exception des compilateurs et SDK C++ standard, ainsi que de certaines bibliothèques de bas niveau et spécifiques à la plateforme (comme les bibliothèques OpenGL ou X11 par exemple).

Copier l'intégralité du code source des dépendances directement dans votre référentiel n'est PAS considéré comme une méthode appropriée de gestion des dépendances !

Le projet DOIT fonctionner sous Linux, à la fois pour le client et le serveur.

Le projet DOIT être multiplateforme.

En plus de Linux, il doit fonctionner sur Windows à l'aide du compilateur Microsoft Visual C++ .

Ni MacOS ni WSL ne sont considérés comme multiplateformes car ils sont tous deux des systèmes ou environnements de type UNIX. ments.

Un véritable jeu multiplateforme permet d'exécuter le serveur et le client à la fois sur Windows et Linux, et de « jouer en cross-play » entre les clients de différents systèmes d'exploitation.

Il convient de noter que la décision de mettre en œuvre une approche multiplateforme a des impacts majeurs sur le développement, et c'est un élément à aborder dès le début pour réussir.

Un flux de travail de développement logiciel bien défini DOIT être utilisé

Vous êtes censé utiliser de bonnes pratiques de développement, et en particulier, vous devez adopter de bonnes pratiques et usages de Git : branches de fonctionnalités, demandes de fusion, problèmes, balises pour les étapes importantes, contenu et description des commits, etc.

C'est encore mieux si un workflow CI/CD est utilisé pour créer, tester et même déployer le serveur.

NE mettez PAS un membre dédié à temps plein sur le CI/CD, car c'est une tâche très chronophage pour ce projet.

Ne récupérez pas l'intégralité des dépendances après chaque commit, sinon vous dépasserez rapidement le quota CI/CD. Il existe des méthodes plus intelligentes pour gérer ce problème (caches, images de build préconfigurées, etc.).

Un autre aspect à prendre en compte est l'utilisation d'outils spécifiques tels que les linters ou les formateurs C++ (par exemple, clang-tidy et clang-format), car ils aident à repérer les erreurs de programmation courantes et à appliquer un style commun.

Exigences en matière de documentation

La documentation n'est pas votre tâche préférée, nous le savons tous ! Pourtant, c'est aussi la première chose à laquelle vous devrez prêter attention si vous souhaitez vous lancer dans un nouveau projet.

L'idée est de fournir les éléments de documentation essentiels que vous seriez heureux de voir si vous souhaitiez contribuer vous-même à un projet pour une nouvelle équipe.

Vous DEVEZ rédiger la documentation en anglais.

Cela comprend :

Tout d'abord, le fichier README !

C'est la première chose que tout développeur verra : la façade publique de votre projet. Il est donc préférable de la réaliser correctement.

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

La documentation du développeur

C'est la partie que vous n'aimez pas. Mais réfléchissez-y : son objectif principal est d'aider les nouveaux développeurs à se plonger dans le projet et à comprendre son fonctionnement dans ses grandes lignes (et non dans les moindres détails ; le code est là pour ça).

Pas besoin d'être exhaustif ou verbeux, il faut avant tout être pratique.

Les informations suivantes sont généralement celles dont vous aurez besoin :

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

Les directives de contribution et les conventions de codage sont également très utiles. Elles 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.

Notez que générer de la documentation à partir des commentaires du code source, comme avec l' outil Doxygen , bien que ce soit une bonne pratique, ne peut pas être considéré à lui seul comme une véritable documentation de projet. Il est impératif de produire une documentation plus complète que de simples descriptions de classes et de fonctions !

L' étude technique et comparative

Vous êtes amené à expliquer la pertinence d'utiliser les différentes technologies que vous employez (langage de programmation, bibliothèque graphique, algorithmes, techniques de mise en réseau, etc...).

Une bonne approche consiste à réaliser une étude comparative des technologies pour justifier chacun de vos choix, notamment en fonction des différents axes ci-dessous :

Algorithmes et structures de données : vous devez être en mesure d'expliquer votre sélection d' algorithmes, de modèles de conception et de structures de données existants pour le projet, ainsi que votre sélection d' algorithmes nouveaux et personnalisés, et pourquoi vous avez dû les développer.

Stockage : une étude des différentes techniques de stockage doit être incluse dans votre étude comparative, concernant la persistance, la fiabilité et les contraintes de stockage.

Sécurité : la sécurité et l'intégrité des données doivent être gérées et sécurisées efficacement. Dans votre étude comparative, il pourrait être pertinent d'examiner les principales vulnérabilités de chaque technologie. Expliquez également comment vous avez mis en œuvre la surveillance de la sécurité de ces technologies à long terme.

La documentation doit être disponible et accessible de manière moderne.

Les documents tels que PDF ou .docx ne sont plus vraiment le format de documentation utilisé aujourd'hui. Il est plus pratique de consulter une documentation en naviguant en ligne à travers un ensemble de pages structurées et bien reliées, avec un plan d'accès rapide, une barre de recherche pratique et un contenu indexé par les moteurs de recherche.

Les outils de génération de documentation sont conçus à cet effet et permettent de générer un site web statique à partir de fichiers de documentation source. Les wikis en ligne constituent également une alternative intéressante au travail collaboratif.

Exemples : markdown, reStructuredText, Sphinx, Gitbook, Doxygen, Wikis, etc.
Il existe aujourd'hui de nombreuses possibilités rendant les documents hérités définitivement obsolètes.

Documentation du protocole

Ce projet est un jeu en réseau : le protocole de communication est donc un élément essentiel du système. La documentation du protocole réseau doit décrire les différentes commandes et paquets échangés entre le serveur et le client. Il est absolument nécessaire de pouvoir écrire un nouveau client pour votre serveur, simplement en consultant 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 cet effet. Rédaction d'une RFC c'est une bonne idée.

Exigences d'accessibilité

Le projet, y compris sa documentation, doit être accessible aux personnes en situation de handicap. Voici les catégories de handicap habituelles à prendre en compte (liste non exhaustive) :

- Handicaps physiques et moteurs
- Handicaps auditifs et visuels
- Handicaps mentaux et cognitifs

Pour chacune de ces catégories, quelle solution proposez-vous ? Inspirez-vous des fonctionnalités et des paramètres existants dans les jeux réels ; il existe de nombreuses façons de les gérer. Vous pouvez également intégrer ces aspects à votre étude comparative.

Partie 1 : Architecture logicielle et premier jeu Prototype

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.

Les objectifs généraux sont les suivants :

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

- Le jeu DOIT démontrer les fondements d'un moteur de jeu, avec au moins des sous-systèmes/couches visibles et découplés pour le rendu, la mise en réseau et la logique du jeu.

Serveur

Le serveur implémente toute la logique du jeu. Il agit comme source faisant autorité des événements de logique de jeu dans le jeu : quelle que soit la décision du serveur, les clients doivent s'y conformer.

Dans une architecture de jeu vidéo client/serveur classique et simplifiée, les clients envoient les entrées utilisateur locales au serveur, qui les traite et met à jour l'univers du jeu. Ce dernier renvoie ensuite des mises à jour régulières à tous les clients. À leur tour, les clients affichent l'univers du jeu mis à jour à l'écran.
Il existe cependant de nombreuses variantes autour de ce principe de base, et vous devez concevoir votre solution.

Le serveur DOIT notifier chaque client lorsqu'un monstre apparaît, se déplace, est détruit, tire, tue un joueur, et ainsi de suite..., ainsi que notifie les actions des autres clients (un joueur bouge, tire, etc.).

Le serveur DOIT être multithread, afin de ne pas bloquer ou attendre les messages des clients, car Le jeu doit s'exécuter image après image sur le serveur, quelles que soient les actions des clients.

Si un client plante pour une raison quelconque, le serveur DOIT continuer à fonctionner et DOIT avertir les autres clients du même jeu qu'un client a planté. Plus généralement, le serveur doit être robuste et capable de fonctionner quel que soit le problème.

Vous POUVEZ utiliser la bibliothèque Asio pour la mise en réseau ou vous appuyer sur une API réseau spécifique au système d'exploitation (par exemple, Unix BSD Sockets) sur Linux par exemple, ou Windows Sockets sur Windows).

Si vous décidez d'utiliser des sockets de bas niveau fournis par le système, vous DEVEZ les encapsuler avec des abstractions appropriées !

Client

Le client est l' affichage graphique du jeu.

Il DOIT contenir tout ce qui est nécessaire pour afficher le jeu et gérer les entrées du joueur, tandis que tout ce qui est lié à la logique du jeu doit se produire sur le serveur.

Le client PEUT néanmoins exécuter certaines parties du code logique du jeu, comme les déplacements locaux des joueurs ou les mouvements des missiles, mais dans tous les cas, le serveur DOIT avoir autorité sur ce qui se passe au final. Cela est particulièrement vrai pour tout effet ayant un impact important sur le gameplay (mort d'un ennemi/joueur, récupération d'un objet, etc.) : tous les joueurs doivent jouer au même jeu, dont les règles sont régies par le serveur.

Vous pouvez utiliser la SFML pour le rendu, l'audio, l'entrée et le réseau, mais d'autres bibliothèques (comme SDL ou Raylib par exemple) peuvent être utilisées. Cependant, les bibliothèques trop vastes ou les moteurs de jeu existants (UE, Unity, Godot, etc.) sont strictement interdits.

Voici une description de l'écran officiel du R-Type :

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

Protocole

Vous DEVEZ 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 telles quelles depuis la mémoire (données brutes), soit avec un codage spécifique optimisé pour la transmission de données.

Le protocole doit être binaire. Veuillez consulter Internet pour connaître les différences entre les protocoles binaires et texte.

Vous DEVEZ utiliser UDP pour les communications entre le serveur et les clients. Une seconde connexion via TCP peut être tolérée dans certains cas, mais une justification solide est requise. Dans tous les cas, TOUTES les communications en jeu (entités, mouvements, événements) DOIVENT utiliser UDP.

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

Pensez à l'exhaustivité de votre protocole, et en particulier à la gestion des messages erronés ou des dépassements de tampon. De tels messages ou paquets malformés NE DOIVENT PAS provoquer un plantage du client ou du serveur, consommer une quantité excessive de mémoire ou compromettre la sécurité du serveur.

Vous DEVEZ documenter votre protocole. Consultez la section relative à la documentation pour plus d'informations sur les exigences relatives à la documentation du protocole.

Moteur de jeu

Cela fait maintenant un an que vous expérimentez le C++ et la conception orientée objet. Cette expérience vous permettra de créer des abstractions et d'écrire du code réutilisable.

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

Le moteur de jeu est le fondement 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, physique, réseau...) communiquent.

Lors de la conception de votre moteur de jeu, le découplage est l'élément le plus important à prendre en compte. 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 connaître les dégâts qu'elle peut infliger ni sa vitesse de déplacement ! Réfléchissez aux meilleures façons de découpler les différents systèmes de votre moteur.

Nous vous recommandons de vous pencher sur le modèle d'architecture Entité-Composant-Système, ainsi que sur le modèle de conception Médiateur. Il existe cependant de nombreuses autres façons d'implémenter un moteur de jeu, des paradigmes orientés objet classiques aux paradigmes entièrement pilotés par les données. N'hésitez pas à consulter des articles sur Internet.

Gameplay

Le client DOIT 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 d'étoiles, le comportement des entités et le flux temporel global ne doivent PAS être liés à la vitesse du processeur. Il est impératif d'utiliser des minuteurs.

Les joueurs DOIVENT pouvoir se déplacer à l'aide des touches fléchées.

Il DOIT y avoir des esclaves Bydos dans votre jeu, ainsi que des missiles.

Les monstres DOIVENT pouvoir apparaître aléatoirement à droite de l'écran.

Les quatre joueurs d'un jeu DOIVENT être clairement identifiables (par la couleur, le sprite, etc.)

Les sprites de type R sont disponibles gratuitement sur Internet, mais un ensemble de sprites est disponible avec ce sujet.

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

C'est le minimum, vous pouvez ajouter tout ce qui, selon vous, rapprochera votre jeu de l'original.

Partie 2 : Sujets avancés

Maintenant que vous disposez d'un prototype de jeu fonctionnel, il est temps d'explorer de nouveaux domaines et de profiter de l'occasion pour approfondir vos connaissances en matière de développement logiciel avancé.

Ce document présente trois axes, chacun divisé en thèmes différents. Vous pouvez choisir les axes et sous-thèmes sur lesquels travailler.

Vous travaillez en équipe et de nombreux sujets abordés sont interdépendants. Ainsi, chaque membre de l'équipe a toute latitude pour travailler sur la deuxième partie, éventuellement en parallèle sur des fonctionnalités totalement indépendantes.

Alors, respirez profondément, lisez tout ce qui se trouve dans cette deuxième partie, discutez avec votre équipe, discutez avec votre équipe pédagogique, choisissez vos sujets préférés... et résolvez des problèmes du monde réel !

En raison de la portée de cette partie, gardez à l'esprit que tout ne doit pas être fait pour valider le projet.
Cependant, il est prévu que des travaux importants soient réalisés sur un ou plusieurs sujets et fonctionnalités.

Piste n°1 : Architecture avancée – Construire un véritable moteur de jeu

De nos jours, la plupart des jeux sont basés sur un « moteur de jeu ». Pour faire simple, un moteur de jeu est ce qui reste de votre base de code, une fois les règles, l'univers et les ressources du jeu supprimés. Un moteur de jeu peut être spécialisé pour un genre spécifique (par exemple, le Creation Engine de Bethesda a été conçu pour créer des RPG 3D avec des scénarios à embranchements) ou polyvalent (par exemple, Unity).

Suivre les objectifs

Dans cette piste, l'objectif est de poursuivre la conception de votre moteur de jeu, afin de livrer un moteur bien défini, avec des abstractions, un découplage, des fonctionnalités et des outils appropriés.

La section suivante du document présente les fonctionnalités dont disposent la plupart des moteurs.

Nous ne vous demandons pas forcément de développer toutes les fonctionnalités vous-même. N'hésitez pas à demander à votre responsable d'unité locale d'utiliser des bibliothèques tierces pour certaines fonctionnalités avancées (rendu d'interface utilisateur, physique avancée, prise en charge matérielle, etc.).

Dans une deuxième étape, vous souhaiterez faire du moteur de jeu un bloc de construction générique et réutilisable, livré en tant que projet distinct et autonome, complètement indépendant de votre jeu R-Type original.

Autrement dit, votre jeu R-Type utilisera le moteur comme une dépendance de bibliothèque, tandis que le moteur lui-même ne sait rien de tout ce qui concerne la logique/les ressources/les fonctionnalités du jeu R-Type.

Pour démontrer que votre moteur de jeu est entièrement générique et autonome, l'objectif ultime est de créer un deuxième exemple de jeu (différent de R-Type !) utilisant votre moteur de jeu autonome. Vous pourrez ainsi prouver qu'il s'agit d'un système réellement réutilisable et générique, sur lequel vous pourrez développer différents jeux.

Il n'est pas nécessaire d'implémenter un deuxième jeu complet pour valider cet objectif. Considérez-le comme une démonstration des fonctionnalités de votre moteur. Cependant, la qualité du deuxième jeu sera évaluée : d'un jeu de pong basique à un jeu complet et complètement différent, comme un clone de Mario, en passant par toutes les variantes intermédiaires.

Fonctionnalités du moteur de jeu

Voici une liste des fonctionnalités essentielles d'un moteur de jeu performant. La quantité et la qualité des sous-systèmes et fonctionnalités de votre moteur seront évaluées, ainsi que tout élément supplémentaire pertinent non mentionné dans la liste suivante.

Modularité

Un bon moteur ne doit pas occuper plus d'espace mémoire que nécessaire, que ce soit sur le disque dur du consommateur ou en mémoire vive pendant l'exécution. La modularisation est un bon moyen d'y parvenir. Voici quelques méthodes courantes pour créer un moteur de jeu modulaire :

temps de compilation : le développeur choisit le module à compiler à partir de votre moteur, en utilisant des indicateurs dans son système de construction ou son gestionnaire de paquets ;
temps de liaison : le moteur est construit sous forme de plusieurs bibliothèques, le développeur peut ensuite choisir de les lier ;
API de plugin d'exécution : les modules sont construits sous forme de bibliothèques d'objets partagés, qui sont soit chargées à partir d'un chemin donné, soit configurées au début du jeu, soit chargées/déchargées selon les besoins pendant l'exécution.

Sous-systèmes du moteur

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 prédéfinis, ...).

Moteur physique :
Le moteur physique a pour objectif principal de gérer les collisions et la gravité. Un moteur plus avancé permet également de déformer, briser, faire rebondir des entités, etc.

Moteur audio : un
moteur audio de base est chargé de diffuser le son en arrière-plan pendant le jeu. Les moteurs audio plus avancés incluent des effets sonores, allant des clics dans l'interface utilisateur aux bruits de jeu. Ils peuvent également gérer les sons géolocalisés dans certains jeux.

Interface homme-machine : dans la
plupart des cas, le moteur est responsable de la gestion des périphériques de contrôle ; le développeur n'a donc qu'à spécifier ceux qu'il souhaite utiliser. Cela peut aller d'une fonctionnalité simple, comme la configuration du clavier et de la manette, à des fonctionnalités plus avancées ou intégrées, comme le déclenchement d'un événement lors d'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 plutôt que par sa lettre, afin de mieux prendre en charge différentes configurations.

Interface de transmission de
messages. À mesure que les jeux deviennent de plus en plus sophistiqués, le nombre de systèmes interagissant rend la synchronisation et la communication plus difficiles à gérer de manière découplée. Une interface de transmission de messages permet de résoudre certains de ces problèmes, car la plupart des interactions sont alors gérées via un système d'événements. Les interfaces de base permettent de gérer des événements asynchrones simples, tandis que les plus avancées peuvent prendre la priorité, répondre et même envoyer des messages synchrones si nécessaire.

Gestion des ressources et des ressources :
la gestion des ressources est une tâche fastidieuse, souvent confiée au moteur, le jeu ne les référençant que par leur identifiant ou leur nom. Les schémas de gestion des ressources les plus courants incluent le préchargement (écran de chargement).

ou chargement à la volée, laissant le moteur gérer un cache de ressources.

Dans la
plupart des jeux , les comportements des entités sont confiés à des scripts externes, ce qui permet des cycles de test/développement plus rapides (sans recompilation). Ainsi, la plupart des moteurs prennent en charge l'intégration de scripts (via un langage personnalisé ou un interpréteur).

LUA ou Python sont généralement utilisés comme systèmes de script

Outillage

Outre les fonctionnalités mentionnées ci-dessus, la plupart des moteurs de jeu fournissent également certains 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 connaissent cette fonctionnalité dans les jeux hors ligne. Principalement utilisée pour déclencher des actions, des scripts ou des sons pendant les phases de test, elle est souvent conservée dans le jeu pour permettre la triche ou aider les moddeurs. Elle introduit également le concept de variables de console personnalisables (aussi appelées « CVars »).

Métriques et profilage 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, l'utilisation des ressources (CPU, mémoire), les images par seconde (FPS) ou le lagomètre. (latence du réseau, images perdues), etc.

Éditeur de monde/scène/ressources :
autonome ou activé par un indicateur spécial à la compilation ou à l'exécution, il permet de placer des ressources dans le monde, en exploitant à la fois la physique et le moteur de rendu pour obtenir un résultat aussi proche que possible de ce qui serait possible en jeu. C'est également un moyen rapide de créer de nouveaux niveaux ou de configurer une cinématique en jeu.

La qualité du ou des éditeurs sera évaluée : d'une solution très simpliste utilisant uniquement des fichiers de configuration, à une application graphique interactive à part entière par glisser-déposer.

Piste n°2 : Réseautage avancé - Créer un jeu en réseau fiable

L'objectif de ce track est d'améliorer les sous-systèmes serveur et réseau de votre moteur de jeu, en les faisant correspondre aux fonctionnalités réellement implémentées dans vos jeux préférés.

Cette piste propose 3 sujets sur lesquels vous pourriez vouloir travailler : serveur multi-instance, efficacité et fiabilité de la transmission de données et architecture de moteur de réseau de haut niveau.

Serveur multi-instances

La plupart des serveurs de jeu dédiés sont capables de gérer plusieurs instances de jeu en parallèle, et pas seulement une 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 pour les gérer, gérer la concurrence, fournir des méthodes permettant aux utilisateurs de se connecter au serveur, de voir les instances, de rejoindre les jeux, de discuter, etc.

L'idée est d'intégrer ces fonctionnalités à votre projet. Voici quelques pistes de réflexion :

La possibilité pour le serveur d'exécuter plusieurs instances de jeu différentes en parallèle ;
Un système de lobby/salle, généralement utilisé pour le matchmaking, ou simplement la découverte d'instances ;
Gestion des utilisateurs et des identités : stockage, sessions, authentification, etc. ;
Communication entre utilisateurs : chat textuel, voire vocal ; Gestion
des règles du jeu : modification de certaines règles de base du jeu, par instance de jeu ; Tableau
de bord global et/ou système de classement ; Tableau
de bord d'administration : interface console en mode texte (exemple : un administrateur peut expulser/bannir un joueur spécifique)
utilisateur), voire une interface Web.

Efficacité et fiabilité de la transmission des données

Dans votre prototype actuel, le serveur envoie probablement des mises à jour à chaque client 60 fois par seconde pour toutes les entités du monde, sans considération technique approfondie. Sur un réseau local, l'expérience devrait être satisfaisante, mais sur Internet, la situation pourrait être bien différente.

En raison de problèmes de réseau inévitables tels qu'une faible bande passante, une latence du réseau, une congestion ou un manque de fiabilité (pertes de paquets, paquets désordonnés, duplication), le jeu peut être confronté à de graves problèmes de synchronisation conduisant à une expérience de jeu sous-optimale, ou pire, à des déconnexions complètes.

La question générale à se poser est donc : quels mécanismes pouvez-vous mettre en place pour atténuer ces problèmes de réseau ? Voici quelques sujets courants que vous pourriez envisager d'explorer :

Emballage des données

Les données en mémoire simples envoyées « telles quelles » sur le réseau peuvent être vraiment inefficaces, même si elles utilisent un protocole binaire. La disposition habituelle des données en mémoire n'est pas nécessairement optimisée pour la transmission de données, en raison de types de données primitifs qui peuvent être trop volumineux, du remplissage interne des champs de structure optimisés par le compilateur pour l'alignement du processeur et non pour la transmission de données, etc.

Conseils : tailles de types de données adéquates, sérialisation efficace en termes d'espace, compactage au niveau des bits, alignement des structures et optimisation du remplissage, quantification des données, etc.

Compression de données à usage général

La transmission de données présente généralement une forte redondance. L'utilisation d'encodages et d'algorithmes de compression polyvalents constitue un moyen intéressant de réduire la consommation de bande passante. De plus, certaines techniques de compression sont spécifiquement adaptées aux jeux.

Conseils : pour les algorithmes généraux, RLE (Run Length Encoding), codage Huffman, LZ4, zlib. Pour les techniques spécifiques aux jeux : compression delta snapshot.

Atténuation des erreurs réseau (pertes de paquets, réorganisation, duplication)

Le protocole UDP n'est pas fiable et, en cas de forte congestion du réseau, des paquets peuvent être perdus, ralentis, réorganisés, voire dupliqués. Le jeu pourrait en pâtir. Il est donc important de prévoir des moyens pour prévenir tout problème lié à la fiabilité du protocole UDP.

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.).

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

Les jeux vidéo en réseau rapides tels que R-Type ou tout autre FPS sont fondamentalement des simulations distribuées en temps réel sur un réseau.

Le problème central de conception de tels programmes est que leur état (c'est-à-dire le « monde du jeu » dans notre cas) doit être cohérent pour chaque participant individuel de la simulation (c'est-à-dire les clients et le serveur), en temps réel, MAIS il existe une latence de communication incompressible entre eux, familièrement appelée « décalage ».

Ainsi, toute information envoyée, comme un déplacement de joueur ou une mise à jour du jeu, sera reçue par d'autres personnes dans un futur proche, tandis que toute information reçue d'autres personnes proviendra du passé. Autrement dit, à tout moment, chaque participant possède une version du monde légèrement différente des autres.

Pourtant, la mission des développeurs de jeux est de donner l'illusion que tout se passe de manière cohérente et efficace ... Sujet délicat !

Ce paradoxe explique ce que vous observez régulièrement dans vos sessions de jeu en réseau préférées : les positions des joueurs et des entités sont parfois instables ou se « téléportent » de manière incohérente, il peut y avoir un décalage d'entrée gênant ou un manque de réactivité, ou plus frustrant, vous êtes parfois tué alors que vous êtes caché derrière un mur.

Solutions possibles

Notre architecture étant basée sur une source de vérité centrale et faisant autorité, le serveur, plusieurs solutions sont possibles pour résoudre le problème. Comme toujours, une solution pragmatique se situe entre deux approches opposées, celle de la cohérence de l'état du jeu et celle des performances réseau. L'encadré suivant propose quelques mots-clés à rechercher concernant diverses techniques généralement mises en œuvre dans les jeux vidéo en réseau.

Conseils : prédiction côté client avec réconciliation du serveur, mises à jour du serveur à basse fréquence avec interpolation de l'état de l'entité, compensation du décalage côté serveur, retard d'entrée, restauration du netcode.

Vous êtes censé être capable de donner et de démontrer des mesures réelles en termes d'utilisation de la bande passante, de réactivité face au lag)

Piste n°3 : Gameplay avancé - Créer un jeu vidéo amusant et complet

L'objectif de ce parcours 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 de jeu. Changeons cela et rendons votre jeu plus agréable pour les joueurs finaux ET les concepteurs !

Joueurs : éléments de gameplay

Bien que R-Type soit un jeu ancien, il n'en demeure pas moins riche en fonctionnalités : il propose de nombreux monstres, niveaux, armes et autres éléments de gameplay. Vous devriez faire évoluer votre prototype vers un niveau de gameplay supérieur. Il devrait être amusant à jouer !

Éléments que vous pourriez prendre en compte :

Des monstres, avec des mouvements et des attaques variés. Exemple : les monstres serpentins, comme dans niveau 2 de type R, ou tourelles terrestres.

Des niveaux aux thèmes variés et aux rebondissements intéressants. Prenons l'exemple du niveau 3 du R-Type original : vous combattez un immense vaisseau tout au long du niveau, ou du niveau 4, où des monstres laissent des traces solides derrière eux.

Boss. Ils sont légendaires dans l'univers de R-Type et ont joué un rôle important dans le succès du jeu. En particulier le premier boss, souvent représenté sur les boîtes du jeu : le Dobkeratops.

Armes. Par exemple, la Force joue un rôle essentiel dans le jeu original : elle peut être attachée devant ou derrière le joueur (et une fois attachée, elle tire vers l'arrière), peut être « détachée » et agir indépendamment, peut être rappelée, protège le joueur des missiles ennemis, permet de tirer des missiles surpuissants, etc.

Règles de jeu : est-il intéressant de laisser les utilisateurs modifier ou peaufiner les règles du jeu. Pensez à des éléments tels que les tirs amis, la disponibilité des bonus, la difficulté, les modes de jeu (« coopératif », « versus », « pvp », etc.)

Conception sonore. Elle joue un rôle essentiel dans toute expérience de jeu : la musique (ou mieux encore, les processus) musique dure), effets environnementaux, effets sonores, etc.

Inspirez-vous du jeu existant : - Aperçu du gameplay - Décomposition des étapes individuelles en particulier.

La notation ne se fera pas uniquement sur le plan quantitatif, mais aussi qualitatif : disposer d'éléments de type N ou M n'est pas le seul critère important. Disposer de sous-systèmes réutilisables pour ajouter facilement du contenu est tout aussi important. Voir la section suivante.

Concepteurs de jeux : outils de création de contenu

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

Les concepteurs de jeux ne sont pas forcément des développeurs expérimentés, et il est très utile pour eux de pouvoir ajouter facilement du nouveau contenu au jeu. La nécessité de connaître le C++ et de modifier de nombreux fichiers de votre projet (éventuellement devoir tout recompiler lorsqu'ils ajoutent un nouveau niveau ou un boss par exemple) est généralement un obstacle.

Votre moteur de jeu DOIT fournir des API bien définies permettant une extensibilité d'exécution (par exemple, un système de plugins, des Dll) pour ajouter du nouveau contenu, des formats standard ou même un langage de script dédié aux comportements des entités de programmation.

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

La question générale est : est-il facile d'ajouter du nouveau contenu et des comportements au jeu ? Bien sûr, la connaissance du C++ peut être requise, mais le système doit être suffisamment simple pour être utilisé au quotidien par des personnes ayant des compétences en développement limitées.

Il est encore plus judicieux de disposer d'outils d'édition de contenu pour différents éléments du jeu (éditeur de niveaux, éditeur de monstres, etc.). Ceux-ci peuvent être des programmes distincts ou intégrés directement au jeu principal. Consultez la section correspondante du module « Architecture avancée » pour plus d'informations à ce sujet.

Dans tous les cas, disposer d'outils ou de sous-systèmes de création de contenu solides et utiles est tout aussi important que de disposer d'un contenu abondant : inutile d'avoir cinq niveaux si chacun nécessite une modification importante du fonctionnement interne du jeu de base. Mieux vaut n'en avoir que deux et démontrer à l'évaluateur que votre système permet d'ajouter facilement tous les niveaux souhaités.

La documentation est un élément essentiel de tout outil, API ou sous-système de création de contenu. Les tutoriels et les guides pratiques sont particulièrement recherchés par les créateurs de contenu.

v 4.2-dev