Projet OpenGL/C++

Synthèse d'image II / Programmation Objet

—IMAC deuxième année—

PortalStein

Objectifs

Le projet vise à mettre en pratique, au sein d'un projet complet, les notions vues en synthèse d'image et programmation objet. L'objectif est un jeu Portal simplifié, appellé par tradition, PortalStein.

Seront notamment mis en pratique : le chargement et le positionnement d'une géométrie de décors immersifs, leurs rendu temps réel et le déplacement intéractif d'une caméra associé à un personnage.

Plus spécifiquement, le concept de déplacement original de Portal permettra, par le positionnement des portails et les collisions, d'expérimenter le concept pilier du lancer de rayon ; par la visualisation des portails et la téléportation, d'aller plus loin à la fois dans la technologie OpenGL et dans la maîtrise de l'espace 3D.

Enfin, la gestion d'un projet en groupes, avec les compromis, la planification, le partage des tâches et la mise en commun qu'ils requièrent, constitue l'un des challenges principaux.

Plan du sujet

▸ Consignes générales

Groupes

Les groupes sont de 4 ou 5 étudiants. Ils doivent être transmis aux enseignants par mail dès le début du projet. Une adresse mail réferente doit être fournie par groupe, celle du chef de projet.

Echéances

Le rapport doit être rendu, au format papier et au format numérique au plus tard quatre jours avant la soutenance. L'archive de code sera rendu le jour même de la soutenance. La soutenance aura lieu juste avant le début des trois semaines dédiées au projet tutoré, en salle 1B077, selon un ordre de passage. La date exacte de ces deux échéances ainsi que les horaires de passage seront données ultérieurement.

Programmation

Le dossier du programme doit être complet et permettre directement la création d'un exécutable sur les machines de la 1B077 sous linux. Il est accompagné d'un fichier "readme" contenant la marche à suivre pour la compilation et l'exécution. Il est primordial que le rendu soit unique, c'est à dire que toutes les parties du travail aient pu être rassemblées avec succès. Cette phase étant délicate, elle doit être réalisée progressivement pendant toute la durée du projet. Le code doit être écrit et organisé selon les normes vues en TD (notamment en C++) et suffisamment commenté. Il doit également répondre à la norme OpenGL3.

▸ Gestion de projet

Le déroulement du projet a lieu selon les étapes suivantes :

1 - Répartition des roles au sein du groupe,

2 - Détermination des objectifs (esthétique et ambiance, priorisation des éléments retenus du gameplay)

3 - Planification (à faire évoluer au cours du projet),

4 - Détermination des taches de programmation prioritaires et répartition,

5 - Programmation

6 - Mise en commun, tests

7 - Réalisation du rapport, planification de la soutenance-démo, préparation du code pour le rendu final.

Les étapes 4, 5 et 6 forment en réalité un ensemble qui doit être répété au maximum pendant le projet. Les tâches à répartir doivent donc être les plus petites possibles, de sorte que chacune des itérations du cycle dure au plus, quelques jours. Lorsque la tâche s'avère plus difficile que prévue, remettre en cause l'attribution des ressources est inévitable. L'utilisation de cycles courts permet de détecter les problèmes au plus tôt.

L'étape 7, doit absolument être débutée avant la fin de la programmation. Le rapport notamment devra donc être achevé avant le rendu final, ce qui implique qu'il contienne une description prévisionnelle du rendu final.

Répartition des rôles

Différentes responsabilités doivent être réparties parmi les membres. Le "responsable" a pour rôle de diriger la tâche, et notamment de trancher en cas de conflit. Toutefois il ne décide pas tout seul, ni ne traville tout seul sur le domaine. Il doit savoir répartir et déléguer en justifiant ses choix avec intelligence et tact. Les rôles (notamment les deux derniers) ne sont pas obligatoires, et sont non exaustifs.

- **Chef de projet**
  Le chef de projet est le reponsable du bon déroulement du projet, principalement du point de vue humain. Il connaît les points forts de chacun et dispose de bonnes dispositions en matière de négociation, d'empathie, de communication, d'organisation et de gestion du stress. Il supervise la planification des tâches en tenant compte des emplois du temps, priorités et facultés de tous les membres. Il s'engage à tenir les enseignants informés en cas de grosses difficultés, notamment de conflits importants entre des membres du groupe.
- **Directeur technique**
  Le directeur technique est responsable des décisions en terme d'architecture du projet, et de la qualité du rendu final en terme de fonctionalités opérationnelles. Il est a l'aise avec la programmation, connait les capacités de son groupe et sait faire les choix techniques qui s'imposent en prennant en considération le délai imparti et l'investissement de tous. Il est responsable de l'architecure globale, des normes de code et de commentaire, du système de gestion de version, de la validation du code de chacun et de la mise en commun. Il sait consulter pour la prise de décision et déléguer.
- **Directeur artistique**
  Le directeur artistique est responsable du projet final en terme d'apparence et de justification scénaristique et esthétique. En début de projet, il gère le brainstorming sur l'atmosphère, l'environnemnt matériel et spatial, l'époque et le scénario. Il rédige les objectifs choisis et enclenche rapidement la phase de création ou de rassemblement des éléments du décors et de l'ambiance (notamment son si prévu). Il valide les créations et est garant de leur utilisation dans le jeu. Il s'assure au cours du projet que le jeu converge bien vers l'idée de départ, ou adapte les objectifs au mieux.
- **Responsable communication**
  Il est reponsable de la qualité de la communication finale sur le projet. Il gère notamment la rédaction du rapport (choix du plan, rédaction des parties générales et choix du style graphique, collection des descriptifs techniques et screenshots des différentes parties auprès de leur créateurs). Il organise le bon fonctionnement matériel et temporel de la soutenance, gère les répetitions, organise le rendu final (archive de code fonctionnel, rapport au format numérique et papier) dans les délais impartis. En plus des choix artistiques, techniques et des fonctionalités finales, la communication doit décrire le déroulement temporel, la répartition, et les difficultés rencontrées, qu'elles soient surmontées ou non. Le rapport est aussi l'occasion de valoriser tous les temps de reflection, création et programmation, notamment sur les parties inachevées ou abandonnées. Il est globalement reponsable du fait que le rendu illustre tous les aspects du travail accompli et leur fasse honneur.
- **Responsable logistique**
  Le directeur de la logistique gère la qualité des conditions de travail lors du projet. Il organise les réunions à la demande des différents partis, est responsable de la gestion des locaux, repas et matériels. Il s'assure notamment que l'oganisation du fonctionnement à distance est opérationnel (gestion de version, dropbox, mailing et contacts, disponibilités, chat et réunions skype).

Détermination des objectifs esthétiques
Le style de votre Portal est clairement le choix qui aura le plus d'impact sur votre projet. Choisir un environnement original, qui plaît a toute l'équipe est d'abord l'élément de motivation principal. L'image et l'ambiance finale de votre jeu est aussi la raison pour laquelle on se souviendra de votre projet. N'hésitez donc pas à être imaginatif et à vous éloigner de l'aspect et de l'atmosphère du Portal initial. Vous pouvez également adapter le gameplay à un scénario différent, tant que le déplacement par la téléportation reste au coeur du jeu. En cas de doute sur votre thème et le respect du sujet, n'hésitez pas à présenter vos idées à vos enseignants.

La phase de direction artistique, aussi importante soit-elle, doit être courte. Votre thème, scénario et principaux choix esthétiques devraient pour bien être finaux une semaine après le début du projet. N'hésitez pas à rédiger cette partie du rapport (2-3 pages max) aussitôt cette phase terminée, pour avoir fixé vos idées de manière définitive et non équivoque, et aller de l'avant.

Planification
La création d'un calendrier permet de se préparer aux temps de développement disponibles pour chaque tâche et d'être capable de se lancer vite dans les plus importantes. Il peut également permettre de s'apercevoir des difficultés en amont et de revoir en conséquence les objectifs à la baisse. N'hésitez pas à comparer vos échéances avec les autre groupes pour établir un planning plus juste. Le planning sera nécessairement revu et corrigé lorsque vous prendrez la mesure des difficultés réelles de chaque tâche, et de vos capacités mutuelles. Pour éviter les surprises, il paraît pertinent d'avoir bien entamé la programmation avant les vacances ou au pire avant Noël. Le planning devrait se stabiliser à partir de là.

Travail groupé et réunions
Travailler en groupe peut être motivant. N'hésitez pas à prevoir un maximum de journées de travail à plusieurs (éventuellement avec skype ou chat si à distance), avec plusieurs points dans la journée. En début de projet, programmer en binôme peut aussi être une bonne idée, pour se motiver et se lancer. Tâchez de travailler efficacement pendant vos les phases de réunion, pour éviter les sources de stress. Dans cette optique, il est tout aussi important de se mettre dans des conditions confortables et de prendre des pauses.

Faites des réunions une à deux fois par semaines au minimum, selon l'avancement attendu. Pour profiter au mieux des réunions, mieux vaut avoir préparé une démonstration de son avancement et des questions pour la suite.

Prototypage
Il est primordial de comprendre que vu le temps imparti, vous devez aller au plus simple concernant l'architecture et les choix techniques de votre projet.

Puique tout ce que vous devez programmer est nouveau, vous ne pouvez connaître à l'avance, ni le temps nécessaire, et les meilleurs choix d'architecture. Il faut donc ABSOLUMENT apprendre à faire des prototypes. L'idée d'un protoype est de choisir une fonctionalité, et de la faire marcher le plus vite possible dans un programme indépendant du reste, sans prendre garde à l'organisation ou à la beauté du code. Une fois que celà fonctionne, vous avez parfaitement compris les étapes de l'algorithme, les besoins en informations, la complexité des différentes étapes. Et seulement à ce moment là êtes vous capable de faire les choix appropriés d'architecture, et d'intégrer la fonctionalité au reste du projet.

Programmer des prototypes est le seul moyen de terminer avec un code bien écrit et surtout qui marche. Si vous cherchez à organiser tout parfaitement avant de débuter, vous prennez le risque de ne rien finir des fonctionalités prévues car vous devrez à chaque instant vous adapter à une architecture trop complexe et mal pensée. Faire des prototypes permet également tout simplement de pouvoir facilement tester et valider chaque partie du code avant l'intégration et donc de localiser bien plus vite les bugs, que dans le programme complet.

Le projet Portal est intéressant pour vous car permet de travailler sur plusieurs parties très indépendantes. Elles sont (dans un ordre indépendant de la chronologie, difficulté ou importance) :

- la modélisation,
- l'intégration du décors,
- le positionnment des portails,
- l'affichage des portails,
- le déplacement,
- le système de collision,
- le déplacement avec physique,
- la téleportation.

Toutes ces parties sont des examples de fonctionalités qui doivent d'abord être testées dans des prototypes indépendants avant d'être progressivement rassemblées dans l'application complète.

Un exemple de prototype pour le positionnement des portails est la construction d'un environnement cubique puis sphérique, dans lequel on verifie que les portails peuvent bien être disposés sur toutes les faces, orientés correctement.

Pour tester le déplacement ou la téléportation, un cube est une fois encore une bien meilleure idée qu'un décors complet. Vous pouvez tout à fait tester l'affichage des textures issues de FBOs (explications plus loin) des portails dans une application avec juste deux rectangles positionnés de manière fixe.

Savoir découper une application en une multitude de fonctionalités atomiques testables indépendamment est une réelle qualité chez un développeur.

Gestion de version et dossiers partagés

Si vous souhaitez travailler en sécurité sur un code mis en commun à tous les membres du groupe, il est préférable d'utiliser un système de gestion de version (ex : SVN, Git, Mercurial). L'idée est de conserver une historique de tous les ajouts au projet et de pouvoir revenir en arrière si besoin. Le système de gestion de version facilite également la mise en commun des parties de chacun.

Si vous vous sentez dépassés par le concept, elle peut être gérée uniquement par le directeur technique, qui réalise alors, seul, la mise en commun.

Si vous ne connaissez pas la gestion de version, contentez-vous peut-être de travailler sur des archives que vous sauvegardez à chaque ajout. Partager les dossiers de travail peut être facilement réalisé avec Dropbox.

Pour faire le rassemblement des codes des chacun au fur et à mesure en l'absence d'un système de gestion de version, vous pouvez utiliser l'utilitaire en ligne commande "diff" ou avec interface graphique "meld" ou "kdiff" par exemple. Ces logiciels permettent de colorer les différences entre deux fichiers pour mettre en commun les ajouts de chacun à un fichier donné.

Mises en commun régulières

A chaque nouvelle petite fonctionnalité ajoutée, faites la mise en commun. Attendre le dernier moment pour réunir les parties est complètement impraticable et conduira inévitablement à des parties abandonnées. Il est aussi plus motivant pour vous de voir le jeu progresser avec les ajouts de chacun, tout au long du projet.

Esprit critique et réévaluation des objectifs

Sachez vous adapter au temps qu'il vous reste pour revoir vos objectifs. Sachez aussi remettre en cause vos choix artistiques ou de gameplay si vous pensez qu'ils ne fonctionnent pas. Avoir changé d'avis au vu des difficultés montre esprit critique et capacités d'adaptation. Ces qualités mettent en avant la maturité de votre groupe, et seront toujours à l'avantage de votre projet, même si les objectifs sont beaucoup revus à la baisse.

▸ Rendu final

Rapport

Le rapport doit respecter le style graphique du jeu. Il doit être clair, concis, d'un ton et orthographe suffisamment sérieux. Il ne doit en aucun cas contenir du code (même en annexes). Si vous ressentez le besoin d'en mettre, vous pouvez très vraissemblablement le remplacer par des schémas, algorithmes en pseudocode, diagramme de classes, use-cases... Il fait entre 10 et 15 pages. Il doit être rendu en version numérique `.pdf` aux deux enseignants et en version papier au bureau de Sylvie Donard au plus tard quatre jours avant la soutenance.

Il doit contenir (liste désordonnée non exaustive) :

- Description et justification (rapides) des choix esthétiques, scénaristiques (au moin pitch) et de l'ambiance,
- Description des fonctionnalités finales du jeu,
- Screenshots du jeu (différents moments),
- Plans des salles et description rapide avec screenshots de la phase de modélisation et de l'assemblage (processus de création des modèles et du puzzle),
- Description rapide de l'architecture (avec diagramme de classes si nécessaire),
- Description des choix techniques et du fonctionnement de chaque fonctionalité (avec schéma et screenshots si nécessaire),
- Planification prévisionnelle et effective, répartition (diagramme de Gant par exemple),
- Description des fonctionnalités abandonnées ou inachevées,
- Description des difficultés recontrées (humaines, techniques, artistiques, logistiques...),
- Bilan personnel de chaque membre,
- Bilan global et apprentissages.

Démonstration

La démonstration vise à faire admirer le résultat final de votre travail. Profitez-en pour le montrer sous sont meilleur jour en préparant bien le matériel et le démonstrateur.

Soutenance

La soutenance contient une seléction des élements du rapports, présentés de la même manière ou différemment, à l'aide de slides. Elle peut également ajouter des développements ultérieurs au rendu du rapport (le travail final est celui qui sera pris en compte).

Les soutenances de tous les groupes auront lieu en salle 1B077 sur une courte journée. La soutenance dure en tout 25 minutes (5 minutes de démonstration, 15 minutes de présentation, 5 minutes de questions et remarques). Les 5 minutes suivantes seront pour la délibération. A l'issue de la journée, chaque projet aura été évalué mais la note finale pourra également dépendre de la matière (OpenGL ou C++) et de la qualité du code rendu. Vos enseignants se reservent le droit de ne pas mettre la même note à tous les membres du groupe, si la répartission des tâches a très manifestement posé problème. Votre note ne vous sera communiquée qu'ultérieurement.

Archive de code

L'archive de code permet de vérifier l'intégrité de votre travail, et la qualité du code (respect d'une norme de codage, commentaires, qualité de l'architecure et des algorithmes, éventuellment de l'optimisation). Le code doit absolument fonctionner sur les ordinateurs de la 1B077, sous linux.

▸ Objectifs du jeu

Concept de jeu

L'objectif est de concevoir et programmer un jeu de type puzzle basé sur Portal : PortalStein. Les fonctionalités de PortalStein sont inspirées du gameplay de déplacement du jeu original, créé par Valve en 2007. Le paragraphe suivant résume les concepts retenus pour PortalStein :

Le jeu consiste à atteindre une zone a priori inaccessible en créant des portails par paires, permettant de se téléporter de l'un à l'autre. On est immergé dans un personnage, en vue à la première personne. On dispose d'un "portal device", qui permet de créer les portails sur les surfaces qui en sont la cible. Du point de vue du joueur, le déplacement est celui d'une marche naturelle, en mode FPS. Le clic gauche et le clic droit positionnent les portails dans l'alignement du centre de l'écran, si la surface correspondante permet une telle création. Pour utiliser le couple des portails créés, il suffit de passer à travers l'un des deux, pour émerger au delà du deuxième. On conserve en théorie sa vitesse, la direction de déplacement étant orientée de manière à subir une trajectoire rectiligne au cours de la téléportation. Le vecteur de trajectoire relatif au portal de sortie est donc exactement le même que celui relatif au portail d'entrée. Visuellement, la transistion est continue puisque chaque portal affiche la vue depuis l'autre portail. Plus précisément chaque portal affiche le point de vue de l'avatar dans l'espace accessible par le portail. Le gameplay est fondé sur les stratégies plus ou moins difficiles de positionnements successifs de portails pour traverser une zone.

Votre projet s'inspire au mieux de ce principe de jeu, mais peut et doit se libérer de l'atmosphère de Portal, pour créer un scénario et une ambiance qui vous soit propre.

Fonctionalités

Les fonctionnalités obligatoires de votre votre jeu sont :

- Atmosphère et scénario originals,
- Déplacement 2D en vue à la première personne,
- Décors immersif éclairé, texturé et légèrement animé (un objet animé visible au minimum),
- Déplacement du modèle de personnage avec la caméra,
- Cible centrée à l'écran,
- Possibilité de placer des portails par clic sur certaines surfaces,
- Visualisation de "l'autre côté" par le portail,
- Téléportation lors du passage du portail,
- Construction de deux salles minimum, qui présente si possible un challenge par rapport au gameplay.

Les fonctionalités techniques mentionnées ci-dessus peuvent être réalisées par la méthode de votre choix, même si elle paraît simpliste et limitante pour le gameplay (visualisation approximative par les portails, portails uniquement verticaux, salles au plancher horizontal unique, téléportation brutale...). Aller au plus simple est souvent la clé d'un projet réussi. Une implémentation de base de toutes les fonctionalités mentionnées garantie à peu près 13/20 (en synthèse d'image).

Les fonctionnalités supplémentaires conseillées sont :

- Son,
- Immersion initiale dans le scénario,
- Amélioration des méthodes employées pour les fonctionalités obligatoires,
- Gravité, inertie.

Les fonctionalités bonus plus difficiles sont :

- Conservation de la trajectoire de vitesse lors de la téléportation,
- Retournement progressif du personnage lors d'un changement du champ de gravité (couple vertical/horizontal de portails),
- Visualisation et transition visuelle parfaite entre les portails,
- Déplacement et physique plausibles,
- Environnement richement décoré ou animé,
- Séquence d'actions déclenchées en cours de parcours,
- Innovation par rapport au gameplay original,

- Menu, aide et messages.

Ces fonctionnalités sont bien sûr non exhaustives.

Votre imagination, le niveau de finition ou encore la cohérence de vos choix de fonctionalités par rapport au scénario seront récompensés.

Difficultés et bénéfices

Ces tableaux récapitulent les difficultés et intérêts pédagogiques de diverses fonctionalités :

Fonctionnalités obligatoires	Difficultés	Intérêt pédagogique
Construction d'un décors immersif	○ Choix d'une thématique et ambiance ○ Construction et/ou utilisation de modèles ○ Import et assemblage de l'environnement visible (texturé et éclairé) ○ Import et assemblage d'objets sémantiques (surfaces pour portails, plans de collision invisibles, surface de déclenchement d'actions...)	○ Intégration de modèles 3D texturés et éclairés dans un décors cohérent ○ Réflexion sur la création d'une ambiance
Architecture et déplacement dans l'optique du gameplay	○ Déplacement FPS 2D, ○ Collisions avec les murs et les limites de zones inaccessibles ○ Construction élaborée de chaque pièce pour générer le gameplay des portails	○ Recherche de la méthode la plus simple pour résoudre un problème (déplacement 2D et collisions) ○ Réflexion algorithmique et géométrique sur le mode puzzle spécifique à Portal
Animation (du personnage et d'au moins un autre objet par pièce)	○ Positionnement et orientation du personnage et de la cible avec la caméra ○ Animation d'éléments de la scène (cycle d'animation prévu avec des matrices ou interpolation entre des positions clés)	○ Création d'un environnement dynamique, plus immersif et riche ○ Réflexion sur l'animation d'environnements virtuels ○ Expérimentation sur l'immersion par un avatar
Placement des portails	○ Mise en place d'un picking par lancer d'un rayon dans la direction de vue ○ Positionnement et orientation corrects de portails (indifféremment du point de clic, de la forme et de l'orientation de la surface cliquée)	○ Aperçu d'une problématique récurrente et fondatrice en synthèse d'images : le lancer de rayons
Vue dans les portails	○ Manipulation de plusieurs caméras ○ Utilisation de plusieurs rendu par frame : rendu dans une texture avec un FBO ou multiples rendus dans le framebuffer	○ Travail sur les matrices et la perceptions des différents points de vue ○ Dépassement des bases d'OpenGL et meilleure compréhension des mécanismes sous-jacents (notamment par les multiples rendus)
Téléportation	○ Système de détection du passage d'un portail ○ Mise en place d'un système pour simuler une gravité simple et éventuellement un retournement à la sortie des portails	○ Faculté à simplifier un problème (construction des pièces selon les fonctionnalités physiques opérationnelles) ○ Travail sur la gestion du temps et les transformations pour une première approche de physique

Fonctionnalités bonus	Difficultés	Intérêt pédagogique
Son (voix, musique et/ou sons d'ambiance)	◦ Apprentissage autonome de SDL sound ◦ Construction d'une ambiance sonore et/ou musicale	◦ Travail en autonomie face à une bibliothèque ◦ Réflexion sur l'ambiance sonore des jeux
Immersion dans le scénario et l'ambiance	◦ Integration du scénario par une voix, des actions, des évenements avant le jeu ◦ Intégration d'une vidéo cinématique (réalisée par l'intermédiaire du cours de modélisation par exemple)	◦ Construction d'un produit fini
Physique plus élaborée	◦ Gravité (sur le personnage et éventuellement certains objets) ◦ Mouvement avec inertie ◦ Collisions fines et sur plusieurs éléments du décors	◦ Réflexion sur le fonctionnement d'un moteur physique
Multiplication des pièces ou décors plus riche	◦ Pipeline de production manuel ou automatique des salles au sein de l'équipe ◦ Construction de plus de décors (ou combinaison des éléments existants)	◦ Découverte de la masse de travail et de l'automatisation de la création de décors en jeu
Animation avancée (cycle de marche pour le personnage, objets transformés par keyframes)	◦ Import de plusieurs clés et utilisation cyclique dans le temps, ◦ Eventuellement interpolation pour mouvement continus	◦ Gestion fine du temps et des objets en mémoire ◦ Eventuellement interpolation des positions entre les keyframes
Evenements séquencés (par l'avancée du personnage ou le temps)	◦ Interprétation de collisions avec des surfaces pour le déclenchement d'évenements ◦ Prévision d'évenements pour enrichir le gameplay (animation d'objets, fenêtres temporelles d'action réduites, sons, changement d'ambiance...)	◦ Complexification et amélioration du gameplay ◦ Dynamisation de l'environnement
Innovation par rapport au gameplay original	◦ Idées et mise en place	◦ Connaissance d'autres remakes de Portal ◦ Réflexion sur le gameplay ◦ Jeu original
Menus, aide ou messages (accueil, fin, transitions...)	◦ Integration d'un menu 2D	◦ Construction d'un produit fini

▸ Aide technique

Cette partie donne les principaux algorithmes du jeu, testés. Les algorithmes donnés sont ceux d'une application complète. Vous devez comprendre qu'il n'est pas forcément nécessaire ni raisonnable de chercher à programmer les méthodes avancées qui sont présentées ici.

Tachez de toujours commencer par une version simplifiée de chaque fonctionnalité, de la tester puis éventuellement de la complexifier si vous le jugez nécessaire. Par exemple, le déplacement FPS avec gravité et collisions est difficile à mettre en place. Il vaut donc mieux dans un premier temps ne permettre le positionnement de portails que sur les murs verticaux, et uniquement collés au sol. De cette manière, la gravité n'est pas nécessaire, et le dépacement FPS peut donc être tout bonnement en 2D.

Les directions proposés ici le sont juste à titre indicatif. Chaque méthode auquelle vous pourriez penser dispose probablement d'avantages qui la rendrait plus simple, plus complète ou tout simplement mieux adaptée à votre jeu. Voyez l'aide comme des pistes pour comprendre les mécanismes globaux et vous lancer.

Les points abordés sont les suivants :

- Modélisation de l'environnement,
- Intégration des modèles
- Inversion de matrices,
- Modèle du héros, volume englobant et cible,
- Déplacement FPS simple,
- Positionnement des portails,
- Affichage des portails,
- Système de collision,
- Téléportation,
- Déplacement avec physique plus élaborée,
- Conditions de victoire.

Les notations employées sont les suivantes :

M : matrice M $m_{i,j}$: valeur dans M à la ligne i et colonne j $m_{i,0}$: vecteur première colonne de M I : matrice identité $translate(\vec{t})$: matrice translation du vecteur $\vec{t}$ $scale(s)$: matrice scale uniforme par le coefficent s $rotate(\alpha, \vec{v})$: matrice rotation d'angle α autour de $\vec{v}$ ε : petite valeur, à règler selon le cas d'utilisation (par exemple produit scalaire considéré nul : 10^{-6}, décalage surface à portail : 10^{-2}...)

Des erreurs peuvent avoir été commises au cours de la rédaction des algorithmes. Le seul moyen de les éviter est de tâcher de bien comprendre les méthodes et de travailler itérativement. N'hésitez pas à signaler les erreurs importantes de sorte que le sujet soit corrigé pour les autres.

Modélisation de l'environnement

Quelle que soit l'utilisation initiale que vous faites de vos modèles (par exemple dans une cinématique pour le cours de 3DS max), il faudra probablement envisager de n'utiliser dans votre application OpenGL que des modèles low-poly. Dans ce cas, les fonctionalités de simplification du modeleur seront requises avant l'export en .obj.

De même vous devrez vous assurer que vos modèles sont bien dans l'échelle qui vous convient, orientés convenablement, disposant de normales, coordonnées de textures et textures adéquates.

La richesse, l'esthétique et la complexité de votre environnement participent grandement à l'image de votre projet. Ces caractéristiques mettent en valeur votre créativité et la maîtrise de la phase d'intégration. Elles seront donc évaluées. Toutefois la maîtrise de la qualité technique des modèles et des textures sort totalement du cadre des cours OpenGL/C++, ce pourquoi vous êtes tout à fait autorisés à utiliser toutes formes de sources pour vos modèles.

- **Personnage**

  Qualité Le personnage n'est vu qu'à quelque mètres : la haute résolution est donc superflue.

  Posture Il est principalement debout, "portal device" à la main.

Animation Il est possible d'envisager une animation de marche sous la forme d'un cycle de keyframes.

- **Portal Device**

  Qualité La vue du "Portal Device" est proche et permanente (mode FPS) : il faut une bonne résolution.

  Versions Trois versions peuvent éventuellement être créées (pointé inactif, tir portail bleu, tir portail orange).

- **Portails**

  Modèle Quelle que soit la forme que vous souhaitez donner à votre portail, il est plus simple de le modéliser sous forme de deux triangles formant un rectangle (économie de vertices et collision plus simple). Texturer le plan permettra de lui donner sa forme finale (ne pas oublier donc les coordonnées uvs).

  Texture Quatre versions de texture peuvent être réalisées. Il faut deux couleurs pour différencier les portails créés par le clic gauche et ceux créés par le clic droit (par exemple bleu et orange). Deux versions par couleur peuvent ensuite être dérivées, selon que le portail est fermé ou ouvert. Pour modéliser la forme et rendre l'intérieur par l'intermédiaire d'un autre moyen, on doit pouvoir détecter les fragments dedans et en dehors de la forme depuis le fragment shader. Un moyen est de colorer ces zones d'une couleur improbable (rose, vert fluo...) et d'appeller, à la détection de cette couleur dans le shader, le mot-clé `discard` (ou bien de remplacer par l'autre vue).

- **Environnement**

  Quantité Deux salles à quatre salles par groupes peuvent être envisagées, selon leur complexité.

  Style Un choix thématique précis doit être fait (si possible différent de l'atmosphère du jeu d'origine). L'originalité sera valorisée.

  Architecture Les salles doivent être construites de sorte que l'on ne puisse atteindre l'arrivée sans l'usage plus ou moins avancé de portails. Il est recommandé de faire des plans au préalable, qui pourront éventuellement être joints au rapport.

  Types De manière à gérer l'apparence de la scène indépendamment de son fonctionnement en terme de gameplay, il est recommandé de prévoir plusieurs types d'objets, importés et stockés séparément. Voici une proposition de cinq types possibles :

  1 - Murs visibles et gros objets : sols/plafonds/parois, escaliers, vitres, meubles et objets imposants destinés à être collisionnés,

  2 - Objets du décors : objets de relativement petite taille, non collisionnable, d'assez bonne qualité, notamment porteurs de l'ambiance et du thème),

  3 - Murs de téléportation : plans des murs visibles au sein desquels la téléportation est autorisée (sur lesquels peuvent être accolés les éventuels portails),

  4 - Murs invisibles de collision : parois destinées aux collisions aux limites des zones infranchissables (ex: rivières, précipices, amas d'objets, zones dangereuses...),

  5 - Murs invisibles d'action : parois invisibles dont le passage déclenche des actions (ex: déclenchement d'une animation, d'un message oral ou tout simplement délimitation de la zone d'arrivée pour la détermination les conditions de victoire).

Type	Polygones initiaux	Triangulation	Complexité	Disposition	Groupes	Normales	Coord. de textures	Sous-ensemble
1	quads/triangles	√	×	absolue	√	√	√	×
2	quelconque	√	√	absolue/relative	×	√	√	×
3	quads/triangles	×	×	absolue	√	√	×	de 1
4	quads/triangles	×	×	absolue	√	√	×	×
5	quads/triangles	×	×	absolue	√	√	×	×

Si l'on souhaite utiliser des meshs faits de triangles, rien n'oblige leur créateur à modeliser directement en triangles. En effet, la triangularisation des facettes peut simplement être réalisée au moment de l'export du obj.

Pour les objets non animés, il vous est également conseillé de charger vos objets à l'échelle et à la position qu'ils ont dans votre scene du modeleur (disposition "absolue"). Dans ce cas, il faut désactiver la normalisation et le centrement des objets, actuellement présents dans le chargeur de .obj.

Les objets non animés peuvent être regroupés dans un .obj global pour simplifier le chargement.

Les facettes destinées aux tests de collisions doivent être peu nombreuses et les plus grandes possibles. Pour celles du type 3, il s'agit d'un sous-ensemble des objets appartenant au type 1.

L'utilisation des quads est uniquement proposées dans l'idée que le positionnement des portails et les collisions peuvent potentiellement être simplifiées dans des quads. Notamment, si vous souhaitez placer les portails de manière à

ce qu'ils ne dépassent pas des bords des surfaces rectangulaires des murs, analyser la surface pour décaler le portail en fonction des bords sera plus simple si le mur est en une seule pièce dans un quad. Dans tous les autres cas, les triangles fontionnement parfaitement. Si vous souhaitez charger certains objs en quads, il faudra réaliser une nouvelle version du chargeur de .obj, spécifique aux quads.

Si vous réalisez vous-même votre modélisation dans le cadre du cours de 3DSmax, créer à cette occasion une cinématique de mise dans l'ambiance peut être un bon exercice. Votre film pourrait ainsi être tout à fait créatif, tout en permettant qu'une partie des objets soit réutilisable pour le décors du jeu. Jouée en introduction au jeu la cinématique introduirait de manière obtimale votre scénario.

Intégration des modèles

Pour intégrer vos modèles, vous devrez probablement programmer des variations sur le chargeur de obj, comme expliqué plus haut.

Il faudra également envisager l'implantation d'une nouvelle classe, ressemblant à `Object`, pour stocker les objets "virtuels" (types 3, 4, 5). La différence principale est que ces objets ne sont jamais envoyés sur GPU pour affichage, mais servent juste au programme pour réaliser différents tests, par exemple d'intersection. Il faut donc stocker leur géométrie sur CPU de manière permanente, ainsi que les matrices de transformation si necessaire. Le principe de bibliothèque d'objets et d'instances actuellement utilisé dans `Scene` peut fonctionner. D'ailleurs, ces "objets virtuels" peuvent être placés dans `Scene` aussi bien.

Inversion de matrice

Nous avons vu que regarder dans l'oeil d'une camera consiste à appliquer sur le monde la transformation inverse du positionnement de cette caméra. Il arrive souvent que l'on doive inverser une matrice pour prendre en compte différemment son impact dans un calcul.

Les méthodes pour inverser une matrice de transformation classique, ou une matrice de type view sont rappellées dans cette partie.

On appelle ici une matrice de transformation classique, la transformation résultant d'une translation pour positionner l'objet, suivi d'une rotation pour l'orienter, suivi d'un scale uniforme sur les trois axes pour en changer l'échelle. Les objets sont positionnés dans l'espace à l'aide de telles matrices. Obtenir la transformation inverse s'effectue par la reconstitution des trois matrices inverses et leur multiplication dans l'ordre inverse.

Algorithme d'inversion d'une matrice classique M :

$position \leftarrow M \times \begin{matrix}0\\0\\0\\1\end{matrix}$
$T^{-1} \leftarrow translate(-position)$

$scaleFactor \leftarrow \|m_{i,0}\|$(norme de la première colonne de matrix $\vec{x}$)
$S^{-1} \leftarrow scale(\frac{1}{scaleFactor})$

$R^{-1} \leftarrow \frac{M^t}{scaleFactor}$
${R^{-1}}_{i,3} \leftarrow \begin{matrix}0\\0\\0\\1\end{matrix}$
$M^{-1} \leftarrow T^{-1}(R^{-1}(S^{-1}I))$

Algorithme d'inversion de matrice view V (avec c, $\vec{x}$, $\vec{y}$, $\vec{z}$ connus):

${V^{-1}}_{i,0} \leftarrow \vec{x}$
${V^{-1}}_{i,1} \leftarrow \vec{y}$
${V^{-1}}_{i,2} \leftarrow \vec{z}$

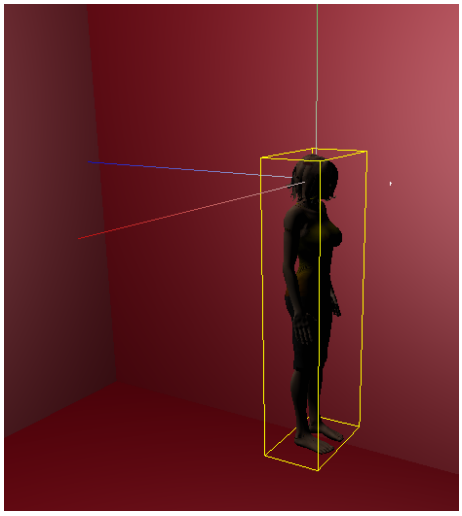
Lorsque votre algortithme d'inversion est programmé, vous pouvez vérifier qu'il fonctionne en multipliant une matrice par sa matrice inverse. Si le résultat est proche de l'identité, c'est bon.

Modèle du héros, volume englobant et cible

Le modèle du héros doit être positionné correctement par rapport au point de vue, et donc mis à jour à chaque déplacement FPS. Sa matrice de transformation est évaluée d'après la position et les axes de la camera. Le modèle du corps doit être baissé et reculé, de sorte que les yeux soient au niveau de `camera->c`. Le modèle doit rester vertical malgré les mouvements verticaux de la caméra, comme s'il ne s'agissait que de mouvements de la tête et des yeux. Ainsi, en baissant le point de vue, on doit pouvoir voir ses pieds. Il est raisonable de conserver un pointeur sur cette matrice pour pouvoir facilement la mettre à jour.

La matrice est identique à celle du **volume englobant** du héros. On appelle volume englobant ou **bounding**

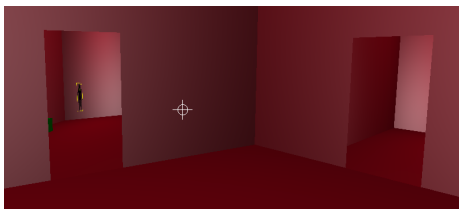
volume (BV), une enveloppe externe simple construite autour et au plus proche d'un mesh. Les BV sont notamment employés pour simplifier les tests d'intersection et de collision avec les objets complexes. Une solution simple de BV est la sphère, avec laquelle les intersections sont souvent triviales. Ce BV est toutefois très approximatif, notamment pour des objets plutôt longilignes comme les corps humains. Une solution plus couramment employés est celle des **boîtes englobantes** ou **Bounding Box** (BB), orientées (OBB) ou non (AABB). Il s'agit de pavés (parallélépipèdes rectangle), dont les parois sont alignées ou non avec les axes, et qui encadrent au mieux la forme de l'objet. Les calculs associés sont modérément compliqués et permettent d'obtenir des résultats bien plus fins qu'avec une sphère, pour une complexité qui reste raisonnable. Lorqu'une solution plus précise est requise, une combinaison articulée de OBB ou une enveloppe convexe peuvent être utilisées, mais cette précision nous sera inutile. Le modèle retenu ici est la OBB. Nous n'utiliserons de bounding box que sur le héros.



Voici un algorithme pour la transformation du héros :

$M_{hero} \leftarrow V^{-1}$
$m_{heroi,1} \leftarrow \begin{matrix} 0 \\ 1 \\ 0 \end{matrix}$
$m_{heroi,2} \leftarrow m_{heroi,0} \times m_{heroi,1}$
$m_{heroi,0} \leftarrow m_{heroi,1} \times m_{heroi,2}$

Sur l'image ci-dessus, on aperçoit aussi une tache blanche devant les yeux. Il s'agit de la cible, qui est une marque toujours présente au centre de l'écran et représentant la direction de visée. Cette autre capture montre une portion du point de vue, avec la cible.



Dans cet exemple, la cible est un objet 3D positionné dans le champ de la caméra, et déplacé pour rester au centre et à distance constante à chaque instant. Il est également possible d'"écrire" la cible ainsi que d'autres informations concernant la situation du jeu directement dans l'image finale, par dessus le rendu, à l'aide de textures par exemple.

Pour la méthode de l'objet 3D, l'algorithme de positionnement est le suivant (positionnement au niveau de la caméra puis déclage devant le plan near et scale) :

$T_{offset} \leftarrow translate(\begin{matrix} 0 \\ 0 \\ -near-\varepsilon \end{matrix})$ avec ε un petit chiffre
$S \leftarrow scale(scaleRatio)$
$M_{target} \leftarrow S(T_{offset}V^{-1})$

La cible permet notamment de positionner les portails, sur la surface derrière la cible, à chaque clic souris.

Déplacement FPS simple

La caméra FPS dont vous disposez permet de voler. Ce comportement est assez contradictoire avec le concept de base de portal. On doit doit donc simuler l'effet de la gravité en contraigant le déplacement au sol. Une méthode simple pour faire ça est de se restreindre à des environnements au plancher horizontal d'altitude constante. Ainsi, la hauteur de la caméra peut être fixée au départ, et le déplacement FPS peut être réduit aux directions dans le plan $\vec{x}$ $\vec{z}$. Dans ce cas, les portails ne peuvent pas, a priori, être placés en hauteur et doivent être contraints aux murs verticaux et à l'altitude du plancher.

Une autre méthode, employée notamment dans les environnements ouverts dont le terrain est construit à partir d'une carte d'altitude (**heightmap**), est d'interpoler la hauteur à la position $[x, z]$ courante. Vous pouvez tout à fait réaliser de telles cartes (texture ou tableaux), pour faire varier simplement les altitudes. Cette solution ne permet tout de même pas directement de pouvoir placer les portails en hauteur.

La troisième méthode est de simuler l'effet de la gravité et de faire intervenir un système de collision entre la BB du personnage et le sol pour l'empêcher de s'"enterrer". Cette méthode est plus délicate à mettre en place, mais permet en échange des effets intéressants, comme les sauts (sur le sol ou les murs), l'inertie, les trajectoires paraboliques... Comme son temps de développement et de test est plus long, il vous est toutefois déconseillé de chercher à la mettre en place avant l'une des alternatives simplifiées.

En attendant plus de physique, on peut toutefois imaginer une gravité de difficulté intermédiaire, qui favoriserait la chute des objets sans nécessiter pour autant d'un sytème de collisions. Il s'agirait tout simplement de faire chuter le personnage jusqu'à la hauteur minimale qu'il peut atteindre (valeur fixe ou interpolée sur une carte).

Positionnement des portails

Le positionnment des portails permet d'aborder une problématique récurrente en synthèse d'image : le lancer de rayons. L'idée est qu'il va falloir positionner chaque portail à l'intersection d'un rayon (vecteur ou demi-droite) avec la surface susceptible d'accueillir le portail la plus proche dans le prolongement de ce rayon. Pour trouver le point d'intersection et donc positionner le portail, il existe plusieurs méthodes numériques plus ou moins précises ou rapides. Une fonction calculant s'il existe une intersection entre un rayon et un triangle (et qui renvoie aussi le point d'intersection) est fournie dans les fonctions annexes. La difficulté consiste donc plutôt à selectionner la facette victorieuse... Pour ce faire, il n'y a pas vraiment d'autre solution que de parcourir toutes les facettes susceptibles d'être intersectées, et à chaque intersection, de retenir la facette et le point d'intersection. Dans l'éventualité où une seule facette est touchée par le rayon, elle est celle de positionnement du portail. Si plusieurs sont obtenues, on ne conserve que celle dont le point d'interection est le plus proche du centre de la caméra.

Avant de vous lancer dans cet algorithme vous devez avoir construit le modèle d'un portail (rectangle avec normales et coordonnées uv), et être capable de le positionner à un endroit choisis à l'avance lors d'un clic souris. Disposer d'un objet C++ Portal dont vous créez deux instances (orange et bleue) peut être une bonne idée.

Vous devriez aussi avoir construit un environnement simplicime constitué de quelques grandes facettes (un ou deux cubes par exemple) et être confortablement positionné à l'intérieur. Attention à bien avoir retourné l'ordre des vertices pour toutes les faces et à avoir inversé toutes les normales si vous comptez utiliser l'intérieur d'un objet.

Dans un premier temps, contentez vous d'utiliser un mesh de décors qui ne nécessite pas de matrice de transformation. Ainsi, vous pouvez utiliser directement les vertices et les normales dans la fonction d'intersection. Lorsque cette étape fonctionne, il vous est possible de multiplier avant le calcul chaque vertex et normale par la matrice de transformation du décors. Cette multiplication peut être permanente (réalisée une fois au départ) ou temporaire (le temps du test d'intersection). Dans le cas d'objets animés, la multiplication est nécéssairement temporaire. Toutefois, si vous réalisez que faire cette multiplication autant de fois que de points n'est pas obtimal, il vous viendra surement à l'esprit de non pas multiplier chaque point par la matrice, mais plutôt de multiplier le rayon par la matrice inverse, pour chaque objet testé... C'est cette méthode qui est employée dans l'agorithme qui suit.

Les fonctions d'intersection d'un rayon avec un triangle (ou un quadrillatère) présentes en annexe ont les prototypes suivants :

```
// Returns true if intersection between ray (from position pos with direction dir)
// and triangle ABC, if true : intersection point in result.
bool intersectRayTriangle(GLfloat * pos, GLfloat * dir, GLfloat * normal, GLfloat * A, GLfloat * B, GLfloat * C,
GLfloat * result);

// Returns true if intersection between ray (from position pos with direction dir)
// and quad ABCD, if true : intersection point in result.
bool intersectRayQuad(GLfloat * pos, GLfloat * dir, GLfloat * normal, GLfloat * A, GLfloat * B, GLfloat * C, GLfloat * D,
GLfloat * result);
```

Attention, la fonction utilisant un triangle nécessite peut-être des corrections (dérivée de celle du quadrilatère mais non testée).

`pos` est le point de départ du rayon (`camera->c`, trois coordonnées).

`dir` est un vecteur directionnel normalisé dans la direction du rayon (trois coordonnées).

`normal` est la normale à la facette.

`A`, `B`, `C` (et `D`) sont les vertices de la facette, dans l'ordre trigonométrique.

`result` est l'adresse d'un tableau pour que la fonction renvoie le point d'intersection (si intersection il y a, trois coordonnées).

$minDistance \leftarrow \infty$

POUR chaque objet $iObjects$ de type 3 :

$M_{iObjects}{}^{-1} \leftarrow$ matrice inverse de $Model_{iObject}$

$pos \leftarrow M_{iObjects}{}^{-1} \times$ `camera->c`

$\overrightarrow{dir} \leftarrow M_{iObjects}{}^{-1} \times (-$`camera->z`$)$

POUR chaque triangle $iTriangles$:

$\vec{n} \leftarrow$ normale de $iFaces$

$A, B, C \leftarrow$ vertices de $iFaces$

$result \leftarrow \begin{matrix} 0 \\ 0 \\ 0 \\ 1 \end{matrix}$

$intersects \leftarrow false$

$intersects \leftarrow intersectRayTriangle(pos, \overrightarrow{dir}, \vec{n}, A, B, C, \&result)$

SI $(intersects)$

$distance \leftarrow \|\overrightarrow{result - pos}\|$

SI $(distance < minDistance)$:

$result \leftarrow Model_{iObject} \times result$

$minDistance \leftarrow distance$

$\overrightarrow{up} \leftarrow \begin{matrix} 0 \\ 1 \\ 0 \\ 0 \end{matrix}$

$\vec{z} \leftarrow \vec{n}$

SI $(up \cdot z < \varepsilon)$ *// Si $\vec{n}$ et $\overrightarrow{up}$ sont alignés (surface horizontale)*

$\overrightarrow{up} \leftarrow$ `camera->y` *// utiliser autre $\overrightarrow{up}$*

$\vec{x} \leftarrow normalize(\overrightarrow{up} \times \vec{z})$

$\vec{y} \leftarrow normalize(\vec{z} \times \vec{x})$

$Model_{portal} \leftarrow translate(result)$

$Model_{portal} \leftarrow \begin{matrix} x_0 & y_0 & z_0 & 0 \\ x_1 & y_1 & z_1 & 0 \\ x_2 & y_2 & z_2 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \times Model_{portal}$

$Model_{portal} \leftarrow translate(\begin{matrix} 0 \\ 0 \\ \varepsilon \\ 0 \end{matrix}) \times Model_{portal}$ *// Décalage du portail devant la surface*

Le décalage du portail par rapport à la surface est nécessaire pour éviter le clignotement du à la superposition de deux surfaces (**z-fighting**). Toutefois, il peut tout aussi bien être réalisé dans le vertex ou le fragment shader.

Une fois la matrice à appliquer au portail calculée, il faut l'utiliser, pour le dessin et pour le calcul de la matrice caméra associée à l'autre portail, si il a été créé. Pour connaître l'état de l'autre portail, possèder une référence sur l'autre dans chaque objet `Portal` peut être une bonne idée. Ainsi, à chaque modification d'un portail, les informations concernées dans l'autre peuvent directement être mises à jour.

Affichage des portails

Deux méthodes sont possibles, pour l'affichage du contenu des portails. La première est décrite par le tutoriel suivant : http://en.wikibooks.org/wiki/OpenGL_Programming/Mini-Portal.

La seconde, quoique légèrement plus dure à règler (transition plus difficilement douces, déformation si mauvais uvs), est plus itérative, plus courte, permet la récursion et est plus générique et utile à connaître. Elle consiste à utiliser les FBOs (Frame Buffer Objects) pour faire des rendus intermédiaires non pas sur l'écran mais dans des textures. Une fois ces rendus terminés, les textures peuvent être affichées sur les surfaces des portails pour donner l'impression d'une vue en profondeur. Les instructions OpenGL pour créer et utiliser un FBO vous sont ici données sans explications. Si vous êtes interessés ou ne parvenez pas à le faire fonctionner tel quel, vous êtes encouragés à d'abord chercher par vous-même la documentation explicative des FBOs et de chaque fonction utilisée ici.

Ces instructions consiste en la création du FBO, réalisée une seule fois par portail au cours de l'application :

```
// in Portal::Portal(...) for example

// Create the FBO, renderbuffer and final texture
    glGenFramebuffers(1, &(this->fboID));
    glGenRenderbuffers(1, &(this->depthBufferID));
    glGenTextures(1, &(this->portalTextureID));
    this->fboBuffs=GL_COLOR_ATTACHMENT0;

// Attributes in class Portal
        GLuint fboID;
        GLuint depthBufferID;
        GLuint portalTextureID;
        GLenum fboBuffs;
```

La fonction suivante paramètre le FBO et spécifie la taille de la texture destinée à recevoir le rendu (et celle du "`render buffer`" dédié à la profondeur). Bien que la profondeur n'est pas rendue dans une texture pour votre utilisation ultérieure, elle est nécessaire pour que le `depth-test` ait lieu dans le rendu de `this->portalTextureID`).

La fonction indique également les paramètres de la texture résultat.

Elle prend en entrée la taille courante de la fenêtre car il est nécessaire que les rendus soit identiques en taille au rendu principal. La fonction doit donc être appellée à la fin de `Application::resize(...)`. N'oubliez pas non plus dans `resize(...)` de mettre à jour les projections des différentes instances de `Camera` que vous aurez créées (par exemple celles des portails).

```
void Portal::prepareFBO(int width, int height)
{
    // Binds the FBO
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, this->fboID);

    // Creates depth renderbuffer
    glBindRenderbuffer(GL_RENDERBUFFER, depthBufferID);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT32, width, height);

    // Creates the color final texture
    glBindTexture(GL_TEXTURE_2D, this->portalTextureID);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_FLOAT, NULL);

    // How to handle not normalised uvs
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    // How to handle interpolation from texels to fragments
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    // Attaches texture to the shader color output and the RBO to the depth
    glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, this->portalTextureID, 0);
    glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, this->depthBufferID);

    //getFBOErrors();
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
}
```

A chaque déplacement de portail et déplacement du point de vue, il faut mettre à jour la caméra associée à chaque portail, puis la texture. Mettre à jour la texture consiste à utiliser le FBO associé au portail pour faire un rendu intérmédiaire dans la texture.

La suite des opérations à effectuer peut être la suivante :

- Copie de `scene->camera`
- Remplacement des valeurs de `scene->camera` par celles du point de vue d'où l'on souhaite faire le rendu (camera spécifique stockée dans le portail par exemple)
- Rendu
- Remise des valeurs initiales de `scene->camera`

On réalise l'étape de rendu avec les instructions suivantes :

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, this->fboID);
glDrawBuffers(1, &(this->fboBuffs));

//scene->setDrawnObjectShaderID(this->drawnObjectID, discardShaderID);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    scene->drawObjectsOfScene();
//scene->setDrawnObjectShaderID(this->drawnObjectID, shaderID);

glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
```

Il est possible que vous ne souhaitiez pas voir les portails lors d'un tel rendu. Dans ce cas, la méthode la plus simple par rapport au fonctionnement de `Scene` (dans laquelle il est difficile de retirer des objets) est d'annuler l'affichage de tous les fragments du portail.

Le shader suivant (dont l'utilsation est commentée ci-dessus) réalise l'annulation des fragments du portail :

```
#ifdef _VERTEX_
// Attributes : per vertex data
in vec4 vertexPosition;
in vec3 vertexNormal;
in vec2 vertexUvs;
in vec4 vertexColor;

void main()
{
    gl_Position=vec4(1.0);
}
#endif

#ifdef _FRAGMENT_
// Final output
out vec4 fragColor;

void main()
{
    discard();
}
#endif
```

Si vous souhaitez disposer de la récursivité des portails, il faudra surement jongler avec l'affichage des textures et des portails, voire faire plusieurs rendus successifs en multipliant les textures pour ne pas afficher une texture dans laquelle vous êtes en train d'écrire... mais il s'agit là d'une fonctionnalité bien avancée...

La caméra utilisée dans le rendu du portail A représente le point de vue depuis l'autre portail (B), éventuellement légèrement décalé et réorienté selon le décalage personnage-portailA. La mise à jour de la caméra du portail est une fonction qui sera utilisée après chaque placement de portail, et à chaque appel à `moveFPS()`. L'algorithme permettant d'obtenir la camera "ultime" est donné à la fin de la partie téléportation.

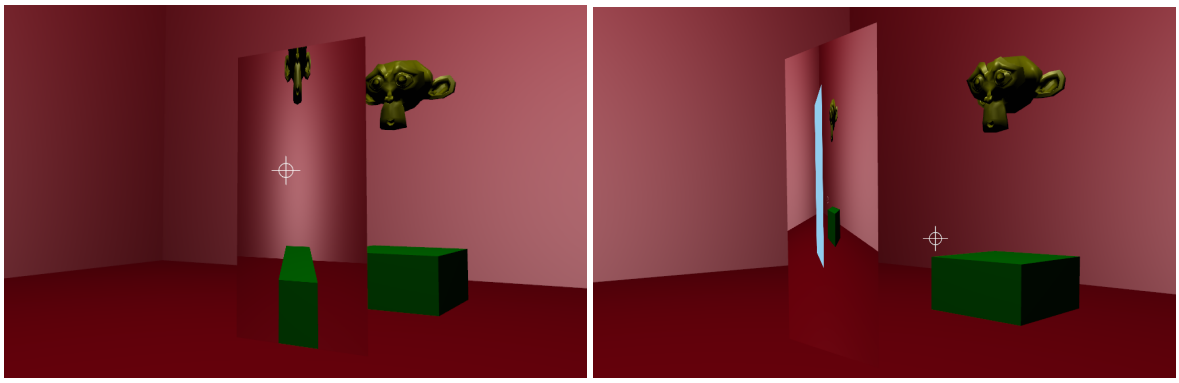
Disposer d'une matrice `view` satisfaisante n'est pourtant pas suffisant pour que le portail soit crédible. Il faut en plus adapter le positionnement de la texture sur le rectangle du portail. En effet, pour l'instant, l'image de la scene entière est condensée dans le cadre du portail, ce qui conduit inévitablement à des déformations. En plus d'éviter les déformations, on souhaite que la zone de l'espace visible dans le portail soit bien celle qui serait visible à travers le cadre de la porte.

Pour selectionner la portion de la texture à conserver dans le portail, il suffit de changer les coordonnées uvs du portail, dans le shader.

Avant de voir le moyen de règler ces coordonnées uvs, voici une idée de prototype pour vérifier la caméra et le plaquage de la texture. On peut réaliser le rendu de l'intérieur d'une porte normale qu'on dispose de manière fixe au

centre de la scene. L'idée est que le rendu sur la porte soit cohérent avec ce que l'on pourrait voir à travers (vérifiable aisément en passant en wireframe). Il est inutile de modéliser les murs autour de la porte. On place donc pour tester un faux portail au centre de la scene, qu'on remplit d'une texture rendue au préalable.

Dans un premier temps, on utilise comme coordonnées uvs des valeurs allant de 0.0 à 1.0 sur le portail. On réalise bien le rendu dans une texture qui fait la même taille en pixels que l'écran ($width \times height$). La première image montre un résultat lorsque la camera est positionnée depuis la porte. La seconde montre une vue utilisant `scene->camera`.

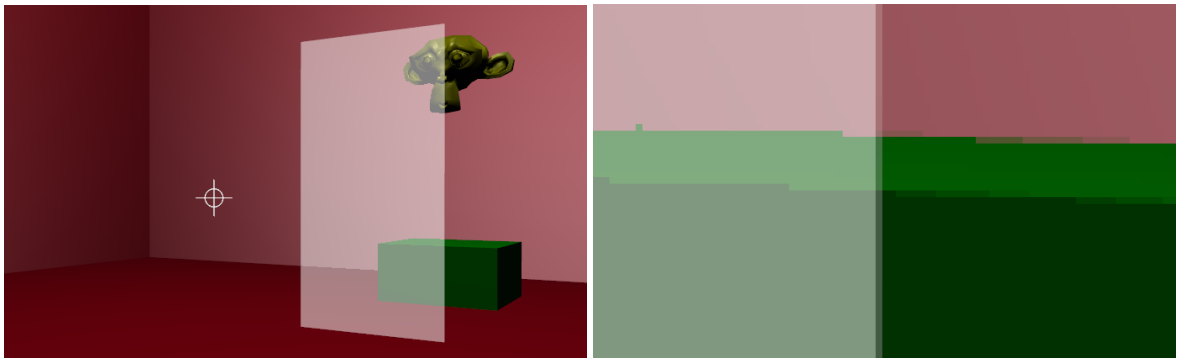


Pour règler la proportion de l'image de sorte qu'on ne voit que le contenu réel normalement vu à travers, il suffit d'utiliser des coordonnées uvs différentes, permettant de selectionner la bonne portion de l'image. Les coordonnées que l'on souhaite utiliser sont en rélité exactement la projection des coins du portail sur le plan **near**, rapportées entre 0.0 et 1.0. Plutôt que de modifier le tableau des coordonnées uvs sur CPU et de le tranférer au GPU à chaque frame, on peut réaliser un shader spécifique qui calcule les coordonnées uvs par fragment à partir des positions des vertices une fois projetés sur l'écran ($projection \times view \times model \times vertexPosition$). L'idée est de prendre dans la texture, pour chaque fragment de la porte, le texel qui correspond exactement à la position 2D du fragment. Faites un schéma pour bien comprendre. Voici le shader correspondant à cette méthode :

```
#ifdef _VERTEX_
// Attributes : per vertex data
in vec4 vertexPosition;
in vec3 vertexNormal;
in vec2 vertexUvs;
in vec4 vertexColor;

// Varyings : data to transmit to fragments
smooth out vec4 posNorm;

void main()
{
    posNorm=projection * view * model * vertexPosition;
    gl_Position = posNorm;
}
#endif


#ifdef _FRAGMENT_
// Varyings : data receved and interpolated from the vertex shaders
smooth in vec4 posNorm;

// Final output
out vec4 fragColor;

void main()
{
    vec2 uvs=posNorm.xy/posNorm.w;
    uvs=(uvs+1.0)/2.0;
    fragColor=vec4(texture2D(textureUnitDiffuse, uvs).rgb, 1.0);
    //fragColor=mix(vec4(texture2D(textureUnitDiffuse, uvs).rgb, 1.0), vec4(1.0), 0.5);
}
#endif
```

Cette méthode fonctionne si bien qu'on ne perçoit plus la porte en FBO du rendu normal. Pour la rendre visible, du blanc a été mélangé à la couleur de la texture pour qu'on perçoive la porte dans l'image suivante (à gauche). Si

l'anti-aliasing est activé (avec le logiciel NVIDIA X Server Settings par exemple), on peut aussi détecter la porte en remarquant que l'anti-aliasing n'a lieu qu'en dehors de la texture (à droite).



Système de collision

Un système de collision basique peut utiliser une configuration spécifique de salle (cylindrique, parallélépidèdique...) et effectuer des tests simples de sortie du volume. On teste alors uniquement la position du centre de l'avatar. Les collisions avec des objets internes ne sont pas gérées.

Une version nettement plus avancée consiste à utiliser pour le personnage une boîte englobante orientée, qu'on teste contre un sous-ensemble de toutes les facettes de la scene. La version proposée ici est coûteuse et nécessite donc que le nombre des surfaces testées soit le plus petit possible. On ne teste donc que les grandes surfaces des murs et autres objets imposants et/ou appartenant au gameplay. Des versions plus optimales que la fonction proposées existent si toutefois vous rencontrez trop de problèmes. Sinon, vous pouvez vous contenter de boîtes englobantes non orientées (l'algorithme gère des OBB mais la fonction proposée n'est conçue que pour des AABB).

L'idée de base est de parcourir tous les triangles, jusqu'à intersection avec la BB. Si plusieurs triangles sont intersectés, dans un cas simple, on ne prend en compte que le premier (par exemple en empèchant le déplacement). Dans le cas plus complexe, impliquant une physique plus élaborée, on prend en compte chaque triangle intersecté pour réagir à la collision. Par exemple, on peut blocker le mouvement uniquement dans les directions normales aux triangles intersectés.

Voici un algorithme non optimal (mais fonctionnel) pour la gestion des OBBs :

$iCollObjects \leftarrow 0$
$iTriangles \leftarrow 0$

TANT QUE $iCollObjects < nbCollObjects$:

- $Model_{iCollObjects} \leftarrow$ matrice de l'object iCollObject, de type 1 ou 4
- ${Model_{BB}}^{-1} \leftarrow$ matrice inverse de la bounding box de l'avatar

- TANT QUE $iTriangles < nbTriangles$:
  - $\vec{n}_{walls} \leftarrow normalize(Model_{iCollObjects} \times$ normale de $iTriangles)$
  - $\vec{n} \leftarrow {Model_{BB}}^{-1} \times \vec{n_{walls}}$
  - $A, B, C \leftarrow {Model_{BB}}^{-1}(Model_{iCollObjects} \times$ vertices de $iFaces)$
  - $collision \leftarrow intersectAABBTriangle(boxHalfSize, \vec{n}, A, B, C)$

  - SI $(collision)$:
    - **Traitement de la collision (utilisation possible de $\vec{n}_{walls}$)**
  - $iTriangles \leftarrow iTriangles + 1$
- $iTriangles \leftarrow 0$
- $iCollObjects \leftarrow iCollObjects + 1$

Voici le prototype de la fonction, jointe en annexe, d'intersection entre une boîte englobante de type AABB et un triangle :

```
// Returns true if intersection of Axis Aligned (centred) Bounding Box with ABC triangle
// (uses separating axis theorem)
bool intersectAABBTriangle(GLfloat * boxHalfSize, GLfloat * normal, GLfloat * A, GLfloat * B, GLfloat * C);
```

`boxHalfSize` est un vecteur des trois demi-largeurs de la boîte dans les directions $\vec{x}$, $\vec{y}$ et $\vec{z}$.

`normal` est la normale au triangle.

`A`, `B` et `C` sont les vertices du triangle, dans l'ordre trigonométrique.

La méthode qui consiste à simplement blocker le mouvement en cas de collision est largement suffisante. Dans ce cas, on peut simplement réaliser la séquence d'actions suivante, à chaque appel de `moveFPS()` :

- Copie des informations initiales de la caméra
- Calcul des nouvelles positions et orientations de la caméra et de la boîte englobante
- Test de collision
- SI intersection : retour à la caméra initiale

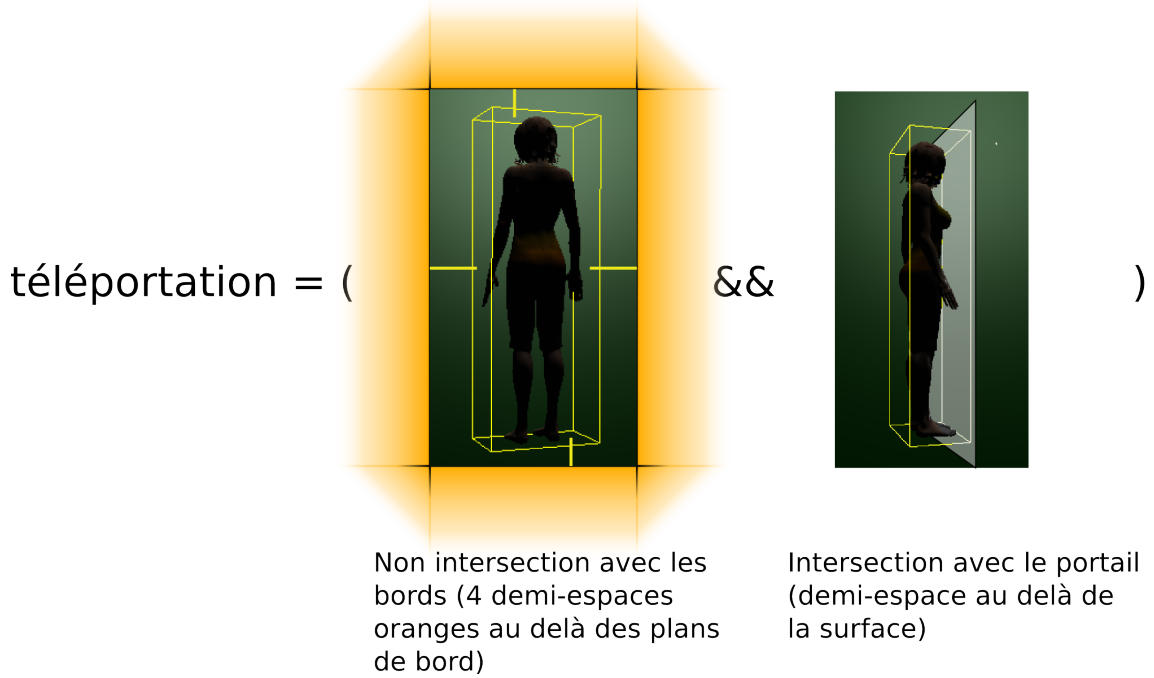
Téléportation

La téléportation est le fait, en entrant dans le portail A, de passer de la position du portal A à celle du portail B instantanément. Indépendamment de ce qui est visualisé dans le portail, il s'agit tout simplement d'un déplacement arbitraire, déclenché lors du passage du personnage dans le portail.

Détecter le passage dans le portail peut être fait par l'utilisation de la distance au centre du rectangle du portail.

Une méthode plus avancée consiste à déclencher la téléportation lors de l'intersection du volume englobant du héros avec le rectangle du portail.

Une méthode encore plus avancée (celle décrite ici) consiste à déclencher la téléportation lorsque le volume englobant intersecte le plan du rectangle sans dépasser des bords de la porte.



Dans l'algorithme suivant, la téléportation à travers le portail A (dont les vertices sont p_{BL}, p_{BR}, p_{TR} et p_{TL}) doit avoir lieu si *crossing* est a `true` :

$c \leftarrow Model_{BB} \times \begin{matrix}0\\0\\0\\1\end{matrix}$
$\vec{n} \leftarrow M_{portal} \times \begin{matrix}0\\0\\1\\0\end{matrix}$
$ptOnPlane \leftarrow M_{portal} \times \begin{matrix}0\\0\\0\\1\end{matrix}$
$crossingValue \leftarrow intersectOBBHalfPlane(center, Model_{BB}, boxHalfSize, \vec{n}, ptOnPlane)$
$insideWall_{bottom}, insideWall_{right}, insideWall_{top}, insideWall_{left} \leftarrow$ `false`

SI ($crossingValue \neq 0$)

- $\vec{n}_{bottom} \leftarrow M_{portal} \times \begin{matrix}0\\1\\0\\0\end{matrix}$
- $ptOnPlane_{bottom} \leftarrow M_{portal} \times p_{BL}$
- $insideWall_{bottom} \leftarrow (intersectOBBHalfPlane(c, Model_{BB}, boxHalfSize, \vec{n}_{bottom}, ptOnPlane_{bottom}) = 0)$
- $\vec{n}_{right} \leftarrow M_{portal} \times \begin{matrix}-1\\0\\0\\0\end{matrix}$
- $ptOnPlane_{right} \leftarrow M_{portal} \times p_{BR}$
- $insideWall_{right} \leftarrow (intersectOBBHalfPlane(c, Model_{BB}, boxHalfSize, \vec{n}_{right}, ptOnPlane_{right}) = 0)$
- $\vec{n}_{top} \leftarrow M_{portal} \times \begin{matrix}0\\-1\\0\\0\end{matrix}$
- $ptOnPlane_{top} \leftarrow M_{portal} \times p_{TL}$
- $insideWall_{top} \leftarrow (intersectOBBHalfPlane(c, Model_{BB}, boxHalfSize, \vec{n}_{top}, ptOnPlane_{top}) = 0)$
- $\vec{n}_{left} \leftarrow M_{portal} \times \begin{matrix}1\\0\\0\\0\end{matrix}$
- $ptOnPlane_{left}) \leftarrow M_{portal} \times p_{TR}$
- $insideWall_{left} \leftarrow (intersectOBBHalfPlane(c, Model_{BB}, boxHalfSize, \vec{n}_{left}, ptOnPlane_{left}) = 0)$

$crossing \leftarrow (insideWall_{bottom}$ ET $insideWall_{right}$ ET $insideWall_{top}$ ET $insideWall_{left})$

Pour implémenter l'algorithme, on a besoin d'une fonction indiquant si la boîte englobante intersecte le demi-espace au delà d'un plan. Le demi-espace dit "positif" est l'avant du plan (côté vers lequel pointe la normale).

Le prototype de la fonction fournie en Annexe à cet effet est le suivant :

```
// Returns code for intersection of Oriented Bounding Box with a plane and the part of space behind it
// returns 0 : fully in positive half−space
// returns 1 : intersecting plane
// returns 2 : fully in negative half−space
int intersectOBBHalfPlane(GLfloat * center, GLfloat * OBBModel, GLfloat * boxHalfSize, GLfloat * normal,
GLfloat * pointOnPlane);
```

`center` est la position du centre de la boîte englobante.

`OBBModel` est la matrice de transformation de la boîte englobante orientée.

`boxHalfSize` est un vecteur des trois demi-largeurs de la boîte dans les directions $\vec{x}$, $\vec{y}$ et $\vec{z}$.

`normal` est la normale au plan.

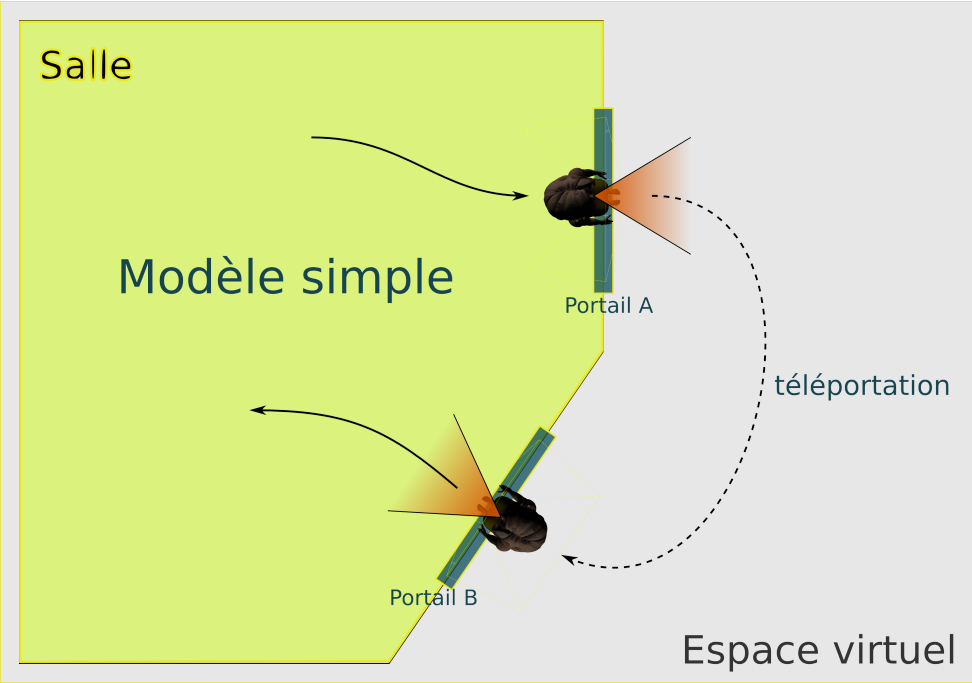
`pointOnPlane` est un point quelconque du plan (peut être par exemple un coin du portail).

La téléportation dure tant que l'intersection entre la boîte englobante et le plan du portail B ne renvoie pas 0.

Lors du passage du portail A, la téléportation consiste à tranformer `scene->camera` en `PortalB->camera`.

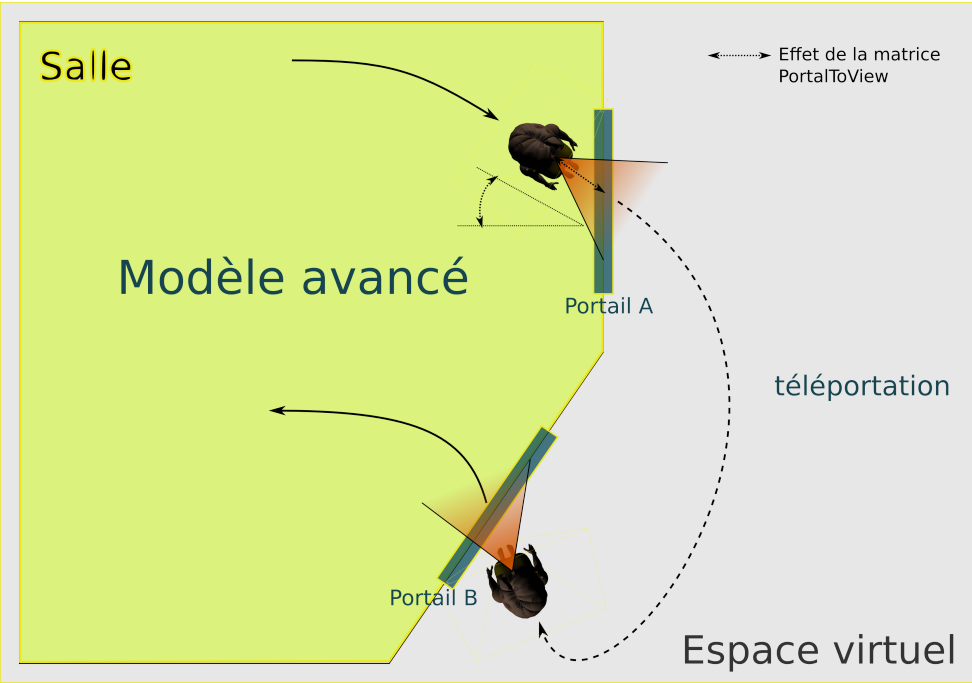
Dans un premier temps, `portalB->camera` n'est rien d'autre que l'inverse (car c'est une caméra) de la tranformation appliquée au portail : ${M_B}^{-1}$. Pour que l'orientation soit juste, il faut de plus appliquer un demi-tour sur l'axe $\vec{y}$ du portail.

Le schéma suivant illustre cette méthode :



Dans le jeu initial, la téléportation n'est pas ressentie comme telle car la visualisation du décors est continue entre les deux portails. Pour obtenir cet effet, la camera est calculée de sorte que la position et l'orientation du héros par rapport au portail A est transferée sur la position derrière le portail B. Ainsi, le mouvement comme l'image vue dans le portail sont parfaitement continus au cours de la téléportation.

Le schéma suivant illustre ce cas :



Réussir cette continuité constitue une fonctionalité tout à fait facultative. Elle est en effet confortable mais non primordiale au jeu.

Voici toutefois l'algorithme permettant d'obtenir la caméra dans le cas d'une transition douce. Cette caméra est bien sûr la même que celle utilisée pour la vue dans un portail. Tâchez donc de ne la calculer qu'une fois pour les deux opérations (à chaque placement de portail et `moveFPS()`).

Est obtenue par cet algorithme la caméra pour le portail A, disposant de la matrice de transformation est M_A :

$V^{-1} \leftarrow$ inverse de `camera->view`
${M_A}^{-1} \leftarrow$ inverse de M_A
$PortalToView \leftarrow {M_A}^{-1} \times V^{-1}$
$R \leftarrow rotate(\pi, m_{B\,i,1})$
$M \leftarrow PortalToView(R \times M_B)$
`camera->c` $\leftarrow M \times \begin{matrix}0\\0\\0\\1\end{matrix}$
`camera->x` $\leftarrow m_{i,0}$
`camera->y` $\leftarrow m_{i,1}$
`camera->z` $\leftarrow m_{i,2}$
`camera->updateView()`

Une difficulté inattendue à la transition douce est l'amélioration de la caméra FPS. En effet, la version corrigée en TD calcule l'orientation de la caméra de manière absolue à partir de la position de la souris. Or, on souhaite, dans le cas de la téléportation, changer l'orientation sans que la souris soit déplacée. La méthode paraissant la plus appropriée pour résoudre ce problème consiste en l'utilisation d'un déplacment non pas absolu mais relatif de la souris à chaque frame. Autrement dit, il faut modifier légèrement la caméra précédente, et non la remplacer par une nouvelle.

La transformation de téléportation doit s'appliquer, sur le modèle du personnage, sur son volume englobant si existant, sur sa cible et sur la caméra principale.

La téléportation du portail A au portail B doit potentiellement momentanément désactiver les collisions et la physique. En effet, on ne veut pas être bloqué par les murs ni tomber en dehors de la salle. La durée de la téléportation va du début de l'intersection avec le portail A, jusqu'à la fin de l'intersection avec le portail B. Activer un "flag" `teleportation` pendant cette periode semble donc être une bonne idée, de sorte que les actions à éviter (collisions, gravité, téléportation, placement de portail...) puissent l'être aisément. Pour bien, le personnage peut tout de même avancer pendant cette pèriode (pour pouvoir sortir du portail B).

A l'issue de la téléportation, le regard se trouve positionné derrière la surface sur laquelle est appliqué le portail B. Cette surface vient a priori bloquer le champ de vision. Un moyen simple de pallier à ça est d'activer le **culling**, c'est à dire de n'afficher que les facettes faisant face à la caméra. Une facette face à la caméra a, par convention, ses sommets dans le sens trigonométrique lorsque vue depuis la caméra. La commande pour activer le culling est la suivante : `glEnable(GL_CULL_FACE)`. Activer le culling en début de projet est une bonne idée. Cela permet nottament de vérifier que les facettes de tous les meshs sont correctes. En effet, l'export depuis le modeleur déteriore souvent l'ordre des vertices (ce qui se manifeste par des trous si le culling est activé).

Déplacement avec physique plus élaborée
Si vous souhaitez utiliser un système externe pour gérer une physique plus avancée dans votre projet, c'est possible. Une bibliothèque appropriée serait par exemple Bullet.

Sinon, voici une méthode à peu près fonctionnelle pour obtenir déplacement avec inertie, gravité, collisions multiples, jumps... $\vec{F}$ représente les forces, $\vec{g}$ la gravité, m la masse, $\vec{a}$ l'accélération, $\vec{v}$ la vitesse, $\vec{d}$ le déplacement et $cameraNewPos$ la position finale.

$\vec{g} \leftarrow \begin{matrix}0\\-9.81\\0\\0\end{matrix}$
$dt \leftarrow 30/1000$
$m \leftarrow 60$
$\vec{d} \leftarrow \begin{matrix}0\\0\\0\\0\end{matrix}$
$viscousDrag \leftarrow 100$
$k \leftarrow 300$
$\vec{F} \leftarrow k(m_{heroi,0} \times \texttt{moveFlags[0]} + m_{heroi,1} \times \texttt{moveFlags[1]} - m_{heroi,2} \times \texttt{moveFlags[2]})$
SI $(!teleportation)$ $\vec{F} \leftarrow \vec{F} + m \times \vec{g}$
$\vec{F} \leftarrow \vec{F} - viscousDrag \times \vec{v}$
$\vec{a} \leftarrow \frac{\vec{F}}{m}$
$\vec{v} \leftarrow dt \times \vec{a}$
$\vec{d} \leftarrow dt \times \vec{v}$
$cameraNewPos \leftarrow \texttt{scene->camera->c} + \vec{d}$

Pour que l'inertie soit possible, il faut sauvegarder le vecteur vitesse, d'une frame sur l'autre. Pour ce faire, à la toute fin de `moveFPS()`, on enregistre la vitesse ainsi par : $\vec{v} \leftarrow \frac{\overrightarrow{\texttt{scene->camera->c}} - \overrightarrow{\texttt{scene->camera->c}}_{backup}}{dt}$

Pour chaque collision, on veut appliquer une force de réaction le long de la normale à la surface collisionnée, sans blocker le mouvement, et se sorte qu'il soit possible de prendre appui sur les surfaces pour sauter. La méthode suivante est expérimentale et nécessiterait améliorations et règlages. On annule en fait la composante de la vitesse normale à la surface, pour ne garder que la composante tangentielle. Pour ce faire on fabrique un déplacement artificiel "de réaction" qui vient annuler la partie de $\vec{d}$ ayant provoqué la collision.

$\vec{d}_{reaction} \leftarrow |\vec{n}_{walls} \cdot \vec{d}| \times \vec{n}_{walls}$
$\vec{d} \leftarrow \vec{d} + \vec{d}_{reaction}$
$\texttt{scene->camera->c} \leftarrow \texttt{scene->camera->c} + \vec{d}_{reaction}$

(Equilibre des forces et sauts)
$\texttt{scene->camera->c} \leftarrow \texttt{scene->camera->c} + \vec{n}_{walls} \times \frac{\texttt{this->moveFlags[1]}}{15}$

`scene->camera->updateView()`
Update du héros, de sa BB et de la cible

Ces opérations sont à effectuer pour chaque surface collisionnée. Les limites de cette méthode sont rapidement atteintes, par exemple dans le cas où le personnage intersecte deux plans non orthogonaux (faire un schéma pour comprendre). Dans ce cas, il faut soit "tricher", soit trouver une meilleur méthode...

Conditions de victoire
Si les portails et les collisions fonctionnent, un moyen simple de tester les conditions de victoire consiste à tester le dépassement d'un plan invisible proche de la sortie de la salle. Sinon, tester à chaque frame la distance à ce point de sortie peut également fonctionner.