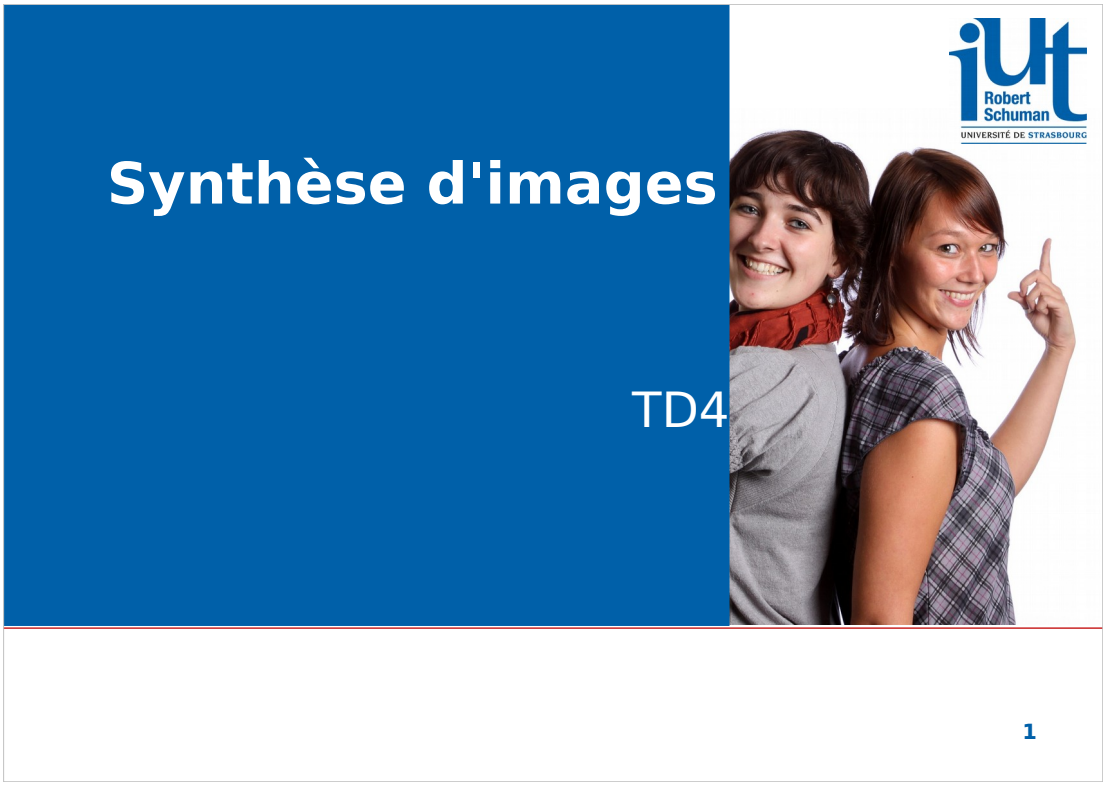


Je rappelle que le format du module ne permet que de survoler le domaine de l'image de synthèse. P4x n'est qu'une introduction au sujet. J'encourage fortement les personnes motivées et intéressées à aller plus loin par elles-mêmes.

Après l'aspect des surfaces (TD3) et la forme des objets (TD2), ce dernier TD sert de fourre-tout pour passer en revue quelques concepts classiques que vous pourriez rencontrer dans la littérature de l'image de synthèse. C'est pourquoi il n'est pas essentiel à la réalisation du TP4, bien que certains concepts, comme le chaînage des transformation dans un graphe de scène pourrait servir fortement dans la réalisation, par exemple, d'un système solaire.

On y évoquera quelques concepts classiques comme les graphes de scène ou les structures d'accélérations. Comme pour les précédents TDs, le contenu est surtout un éventail de mots-clés et de jargon pour vous faciliter les recherches par vous-mêmes.

Moteur 3D

La notion de **moteur 3D** couvre un spectre large, sa définition variant selon les développeurs/concepteurs. Vous avez découvert une bibliothèque JavaScript, three.js, qu'on pourrait qualifier de moteur 3D, alors que si l'on regarde ce que propose des produits comme *Unity3D* ou l'*Unreal Engine*, ça n'a clairement rien à voir.

D'un côté vous avez une simple bibliothèque/collection de classes-méthodes-fonctions de programmation, avec des fonctions permettant de créer des sphères, cubes, ou des matériaux basiques, et qui vous permet rapidement d'avoir des vues 3D en OpenGL ES dans votre page web. De l'autre vous avez des produits complets avec toute une chaîne d'outils permettant de fabriquer des jeux vidéos/logiciels de simulation en entier allant de l'interface utilisateur (GUI) jusqu'à l'import de ressources ou objets 3D en passant par la construction de scénarios via un langage de scripts, etc. On pourrait faire l'analogie entre un éditeur de texte basique versus un IDE (*Integrated*

Development Environment) qui fournit en sus de l'éditeur de texte, le compilateur, le débogueur, le gestionnaire de projets, le versionnage du code source, etc.

Or bien que ces produits ont l'air totalement différents, on peut les qualifier abusivement de moteur 3D. Typiquement, ce qu'on retrouvera sous la terminologie moteur 3D, ce seront tous les produits proposant une surcouche aux *drivers*. Cette surcouche propose généralement :

- Une abstraction de l'API 3D de l'OS (DirectX, OpenGL, Vulkan, Metal…) avec des concepts 3D transverses : *buffers* de données pour les *vertices*, *indices*, textures, …
- Des concepts de niveau plus haut que ce que propose les API 3D : *meshs*, primitives basiques (sphères, cubes, …), modèles d'éclairages (lambert, phong, PBR, …), mais aussi tout bêtement : des fonctionnalités mathématiques (vecteurs, matrices, courbes, …)
- La gestion des données depuis le disque : chargement des textures, des *meshs*, des animations, …
- Un accès simplifié à la programmation des GPUs (*shaders*), via du *glue code* (https://en.wikipedia.org/wiki/Glue_code) tout prêt, comme les *attributes* position, normal, etc. dans three.js. Voire même un système de programmation graphique à base de briques à paramétrer et à relier entre elles
- Et évidemment une gestion correcte des performances, ou du moins la fourniture des outils le permettant (comme l'*instancing hardware*), pour satisfaire les contraintes temps réel

Note : pour rappel, le *driver* correspond peu ou prou à la couche logicielle située dans le système d'exploitation entre le matériel (*hardware* avec une interface de programmation propriétaire) et l'interface logicielle générique proposée par l'OS aux développeurs. Dans le cas de la 3D temps réel, l'interface logicielle générique sera par exemple OpenGL, DirectX, Metal ou Vulkan. Grâce à cela, on programmera une puce NVidia comme on programmerait une puce Intel ou AMD. Le *driver* est généralement fourni par le fabricant de matériel.

Parfois, on essaiera de faire le distinguo avec les suites plus complètes comme *Unity*, en parlant de ***Game Engine*** (moteur de jeu) plutôt que d'utiliser le terme « moteur 3D ».

Des fonctionnalités qu'on pourrait trouver en supplément dans un G*ame Engine* :

- Un *pipeline* complet d'éclairage (PBR), avec gestion des ombres
- Un moteur de physique : collision, destruction, déformation, …
- Des systèmes de particules performants
- La reproduction de l'environnement réel : océan, météo, cycle jour/nuit, nuages, ciel…
- Un éditeur de scène/niveau
- Du scripting
- ...

Quelques exemples :

- https://unity.com/fr
- https://www.unrealengine.com/en-US/

- https://unigine.com/
- https://www.ogre3d.org/

Pour les plus curieux, vous trouverez une liste sur wikipedia :
https://fr.wikipedia.org/wiki/Liste_de_moteurs_de_jeu

Pourquoi un moteur 3D ?

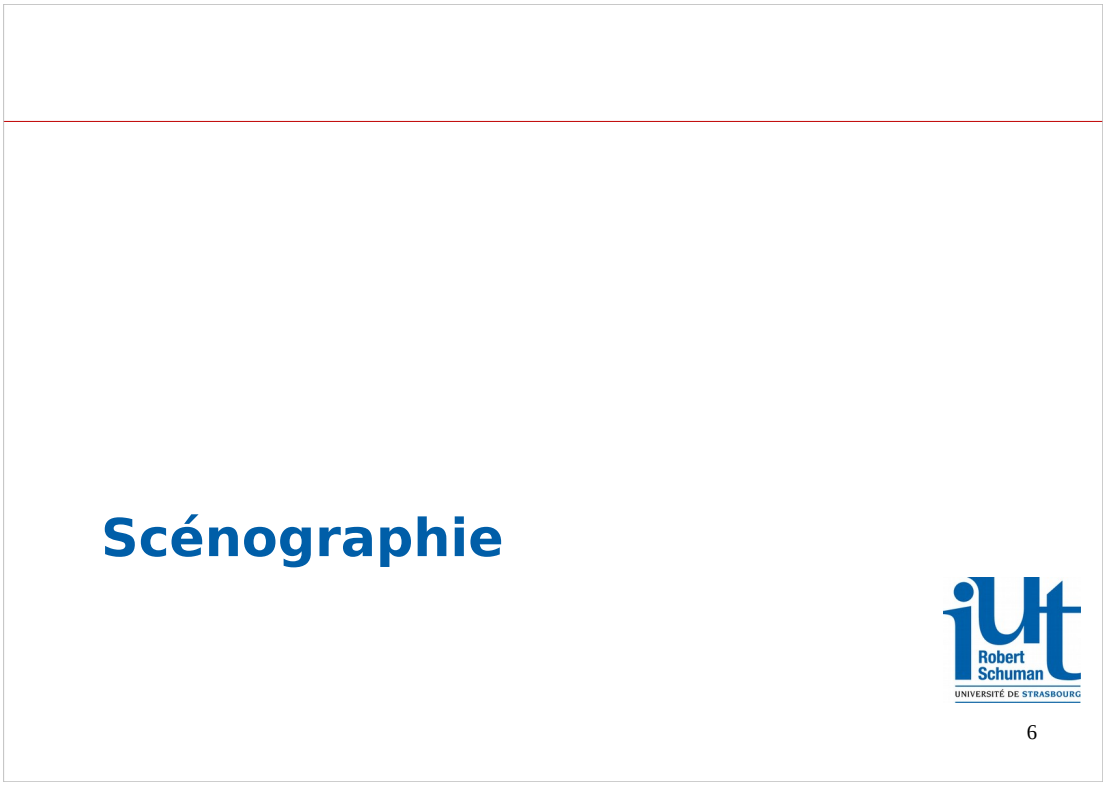
Le premier intérêt, c'est évidemment de faciliter le travail des développeurs. Comme toute bibliothèque logicielle, un moteur 3D facilite la programmation de certains algorithmes.

La complexification des technologies (GPUs) mais aussi des algorithmes, du fait de plus de puissance de calculs disponibles, impliquent un besoin d'expertise extrêmement pointue. Cela devient difficile aujourd'hui de réunir toutes ces compétences dans chaque entreprise.

Par exemple, il y a 15-20 ans, chaque studio de jeu vidéo avait son propre moteur 3D fait maison, qui consistait simplement à afficher quelques triangles texturés avec un semblant d'éclairage « classouille ». Aujourd'hui, les modèles d'éclairage physiquement réaliste demandent une compréhension du niveau d'un physicien chevronné, et ne sont plus accessibles au bidouilleur dans son garage. Les fonctions d'animations, de déformation, de traitement d'images, demandent des connaissances en mathématiques de niveau post-bac.

Avoir ce niveau de compétences dans une petite équipe n'est plus possible, il est donc nécessaire de sous-traiter cela pour des questions de coûts et de mutualisation de compétences. C'est dans cette logique que le marché des moteurs 3D/moteur de jeu vidéo, qui sont des ***middlewares***, s'est naturellement développé depuis une quinzaine d'années pour se concentrer sur quelques acteurs majeurs (*Unity*, *Epic Games*) aujourd'hui.

La suite du support consistera donc en une énumération et un survol de quelques concepts généraux qu'on rencontre dans les moteurs 3D en général, d'une part pour savoir un peu « comment c'est fait à l'intérieur », et d'autre part pour votre culture sur le sujet.



Lorsqu'on organise une scène 3D, on se retrouve rapidement confronté à vouloir matérialiser les relations entre objets. Le tronc et le feuillage forment un « arbre », les roues sont attachées à un essieu lui-même attaché à un châssis dans une voiture, la chaise repose sur le sol, …

Pourquoi structurer ses données ?

L'être humain aime bien faire un peu de **tri**, de **rangement**. Bien que certains spécimens soient très tolérants au désordre, il arrive toujours un moment où l'on ressent le besoin d'organiser les choses, pour au moins s'y retrouver. Les cours d'architecture logicielle, de gestion de projet, les concepts introduits avec UML ou ce genre de méthode, tout cela contribue à ajouter de l'organisation pour éviter le chaos.

Aujourd'hui, on en arrive à un stade où le concept d'organisation des données est crucial pour gagner en **performance**. En effet, depuis les années 2000, la loi de Moore est régulièrement contredite, et l'époque ou il suffisait simplement d'avoir plus de transistors et de gigahertz pour avoir plus de performance est révolue. On constate depuis la moitié des années 2000 une stagnation des fréquences des CPUs, et une multiplication des cœurs dans les puces. On se retrouve alors dans des architectures de traitement parallèles et non plus impératives.

Plus particulièrement dans le domaine du jeu vidéo, les ***design patterns*** comme **ECS** (https://en.wikipedia.org/wiki/Entity_component_system) sont furieusement à la mode en ce moment. Cela suppose de repenser complètement le modèle de la programmation orientée objet (POO), qui devient progressivement désuète, au détriment de la *componentisation* du code.

Naturellement, vous avez déjà eu des cours sur des structures de données en programmation. Celles-ci s'appliquent aussi dans notre cas, on retrouvera donc naturellement des structures d'arbres, de listes, de tableaux, etc.

Une des structure classique que l'on rencontre dans la littérature des moteur 3D, c'est le **graphe de scène**, ***scenegraph*** en anglais, qui, comme son nom l'indique, est un graphe.

Rappel : un graphe est une structure à base de nœuds (*nodes*) reliés entre eux par des liens (*links*). un arbre est un type de graphe particulier. https://fr.wikipedia.org/wiki/Graphe_(type_abstrait)

Alors je vais couper court tout de suite au débat : il n'y a pas de définition officiel de ce qu'est un graphe de scène.

Si l'on regarde par exemple son implémentation par http://www.openscenegraph.org/ , on se retrouve avec un structure qui implique l'usage du *design pattern* ***Traversal*** (qui est une variante du *Visitor,* qui sont donc tous les 2 de la grosse m*rde).

J'ai personnellement eu l'occasion de travailler avec 4 ou 5 modèles différents au cours de ma carrière professionnelle, et aucune implémentation n'était identique. Partez du principe qu'on peut appeler « graphe de scène » un bête tableau (*array*, *vector*) ou une simple liste avec tous vos éléments constituant votre scène.

Ne prenez donc pas pour argent comptant ce que vous pourrez lire sur le sujet dans la littérature. Vous aurez cependant quelques points communs qu'on retrouvera dans quasiment toutes les implémentations :

- c'est un graphe (heureusement…), pour bien insister, liste, tableaux peuvent être vus comme des graphes…
- généralement acyclique (j'expliquerai plus bas pourquoi)
- souvent arborescent : la relation entre 2 nœuds connectés a un sens, une sémantique
- le contenu de la scène est représenté par les nœuds du graphe

On désire généralement un graphe acyclique, pour des raisons de traitement de données. Les développeurs (sauf au niveau scolaire, pour la culture générale) n'aiment pas, mais pas du tout, le code récursif. Avoir des cycles dans un graphe tend à provoquer du code récursif, voire même des boucles, qui du coup peuvent facilement tomber dans le cas pathologique de la boucle infinie.

Note : le code récursif implique pas mal d'emmerdes. C'est élégant, et certains langages fonctionnels reposent fortement dessus, mais ça reste marginal. Il est souvent possible de réimplémenter un algorithme récursif avec une boucle non récursive. La *stack* mémoire vous remerciera...

Généralement, le graphe de scène ne contient que le contenu de la scène, c'est-à-dire les objets et l'organisation hiérarchique spatiale de ceux-ci. Dans ce cas habituel, les éléments externes comme le point de vue (*camera*), ou les éléments d'algorithme (*shadow map*, système de particules, *environment map*, …) n'apparaissent pas dans le graphe de scène et sont des systèmes découplés ailleurs dans le code.

Dans d'autres implémentations, vous pourrez rencontrer des nœuds dans le graphe qui sont en fait des nœuds correspondants à des fonctionnalités et pas uniquement reliés au contenu de la scène. Dans ce cas, le graphe de scène se confond avec le graphe d'exécution du rendu. On croisera par exemple plus loin le cas ou le graphe de scène se confond avec la structure d'accélération spatiale.

Arbre, transformations, vive l'algèbre linéaire !

L'arbre, on parle souvent de **hiérarchie** dans ce contexte, permet de traduire naturellement en code une dépendance spatiale : les roues de la voiture suivent la position de la voiture, le stylo dans la main suit la main, etc. La position des nœuds fils est généralement décrite **relativement** au parent : la lune tourne autour de la terre qui tourne autour du soleil. Il est plus simple de dire que la Lune décrit une orbite autour de la Terre que de dire que la Lune dessine une épicycloïde autour du Soleil...

Il existe alors un outil mathématique tout prêt, la matrice 4x4 en coordonnées homogènes (https://fr.wikipedia.org/wiki/Coordonn%C3%A9es_homog%C3%A8nes) qui modélise simplement les transformations relatives d'un objet par rapport à un autre. Sans rentrer dans les détails mathématiques du calcul matriciel et de la représentation de ces matrices (voir la page wikipédia), une des élégances de l'outil est l'associativité de la multiplication.

Plus concrètement pour vous, il faut simplement comprendre qu'on va généralement associé à un nœud de l'arbre, une matrice de transformation, qui correspond à la transformation géométrique (rotation, translation, homothétie) de l'objet (la Lune, la roue), par rapport a son parent (la Terre, le châssis).

Note : Dans three.js, les nœuds du graphe de scène sont modélisés par la classe ***THREE.Object3D***, que vous avez déjà manipulé, via la classe ***THREE.Mesh*** qui en dérive. On a donc un arbre d'*Object3D*. Le nœud racine de l'arbre est l'instance de scène de type ***THREE.Scene***, qui dérive aussi d'*Object3D* si vous ne l'aviez par remarqué. Lorsque vous modifiez *position* et *rotation* des cubes, vaches ou nounours, vous étiez en train de manipuler le contenu de cette matrice de transformation associée au nœud.

Ces multiplications de matrices, vous en avez déjà aussi rencontré dans l'implémentation du *vertex shader*. Mathématiquement parlant, c'est le même principe. Transformations, changement de repère, multiplication de matrices, c'est kifkif.

Limitations du graphe de scène

- Architecture lourde
  - Les objets *node* sont des « gros » objets informatiquement parlant
  - Pas adapté pour les performances brutes
- Le graphe acyclique ne gère pas tous les cas
  - Le stylo sur la table qu'on pousse avec un doigt : est-il le nœud fils de la table (association par la gravité) ou du doigt (actionneur du mouvement) ?
  - Le projectile est-il associé au tireur, ou à la cible, ou indépendant ?
- Le *design pattern Traversal* est pénible
  - Un peu comme le *pattern Visitor*

C’est une architecture lourde

Le fait d’avoir un concept unique (les nœuds) pour décrire tout objet de la scène fait que ces nœuds représentent aussi bien une voiture qu’un personnage, ou une maison, voire une plume de canard. Typiquement ce qui arrive dans ce cas, c’est le phénomène de la « classe de base qui fait tout même le café ». Les nœuds de graphe de scène sont souvent des gros objets : ils occupent un espace mémoire conséquents, et offre une interface « foisonnante » (qui la façon politiquement correcte pour dire « bordélique »).

C’est tellement *old school* !

Le fait d’avoir une structure associant des objets hétérogènes (voiture, personnages, maison, arbres), n’est pas du tout adapté à un traitement **parallèle**. Or comme évoqué précédemment, aujourd’hui, si on veut de la performance, il faut faire du traitement parallèle : faire la même chose sur N objets identiques. Parcourir ce graphe, c’est croiser des nœuds de type différents à chaque étape, et donc mettre sous pression la mémoire et son cache avec des accès mémoire éparpillés partout, et faire des traitements conditionnels dans le code (*if, switch,* polymorphisme,…), ce qui met sous pression le CPU qui ne peut plus exploiter la prédiction de branche de manière optimale (voir *branch misprediction* dans la littérature comme https://en.wikipedia.org/wiki/Branch_predictor).

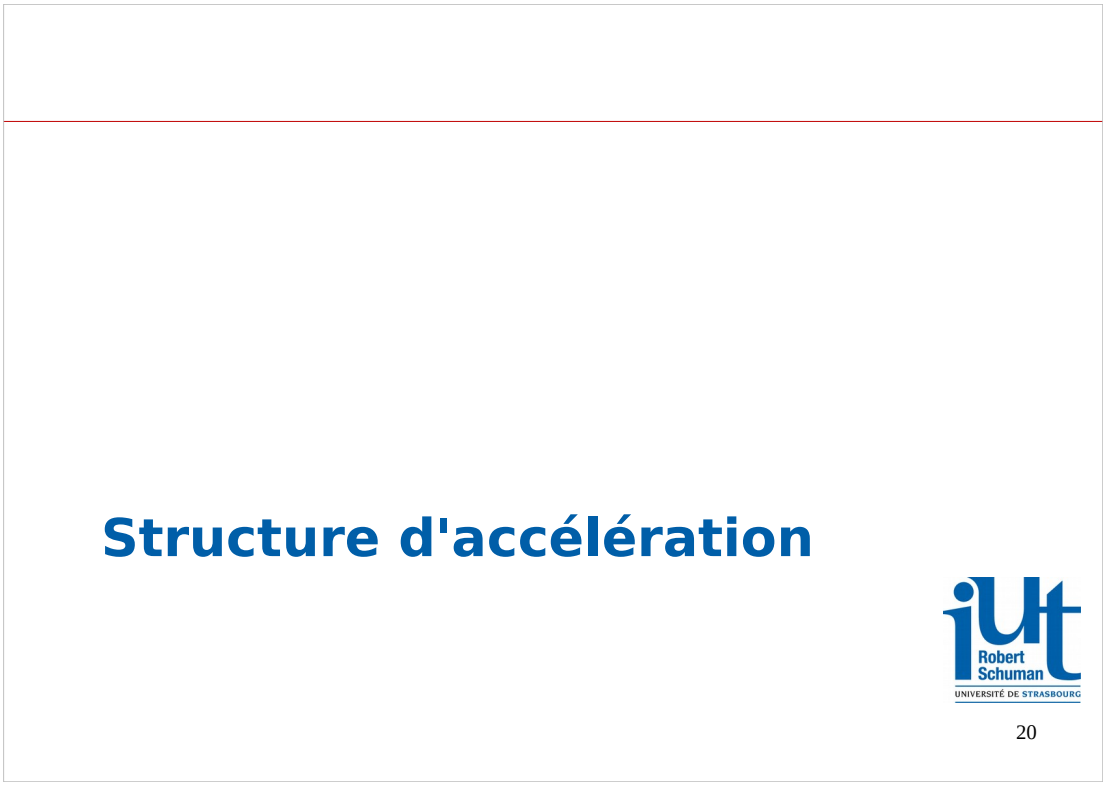
ECS crash course : On en revient à la mode de l’ECS évoqué plus haut, ou le *pattern* est *data-driven,* on organise les données en mémoire pour maximiser les accès en mémoire : mettre toutes les voitures (*Entity*) ensemble pour les mettre à jour (*System*) de la même manière pour éviter les branchements de code, le tout en parallèle. A première vue on voudrait mettre côte-à-côte en mémoire RAM toutes les voitures, mais comme certains attributs (*Component)* sont communs aux voitures et aux camions et vont subir des traitements identiques, plutôt que d’organiser la mémoire au niveau des *Entities*, on le fait au niveau des *Components*.

Cependant, la programmation orientée objet n’est pas morte, elle reste utile car beaucoup plus simple à maintenir et accessible aux développeurs. Il faut cependant avoir conscience de ses limites dans un contexte temps réel.

Un histoire de compromis

L’arbre ne permet pas de transcrire toutes les situations de la vie réelle. L’état d’une ficelle dépend des 2 objets ou sont attachées ses extrémités. La pointe du stylo est contrainte par la feuille sur la table sur laquelle on écrit, mais aussi par la main qui tient le stylo. Ces chaînes de « conditions » ne peuvent pas être décrites dans une structure d’arbre avec un lien de parentalité spatial sans introduire des cycles, et comme je l’ai déjà dit, les cycles, on aime pas !

En général, dans ces situations, on utilisera des systèmes spécifiques pour résoudre des problèmes ponctuels comme par exemple faire coller les pieds du personnages au sol avec de la cinématique inverse.



Une scène peut contenir énormément de données. La volonté d'être exhaustif dans la reproduction de la réalité, comme la tendance actuelle des *openworlds* dans les jeux vidéos, contribuent à la complexification des scènes à afficher par les moteurs 3D.

Dans ce contexte de « masses de données », bien qu'on ne soit pas encore à l'échelle de la *Big Data*, il est nécessaire de mettre en place des stratégies pour conserver un maximum de performance. C'est dans ce contexte que s'inscrivent les techniques algorithmiques dites de « **structures d'accélération** ».

Principe

L'idée sous-jacente des structures d'accélération est de traiter le **minimum** nécessaire pour réaliser la représentation des informations. Or celle-ci est partielle car limitée par l'écran d'affichage. Tout ce qui n'est pas visible à l'écran n'a pas besoin d'être traité. Pour cela, il faut efficacement pouvoir sélectionner ce qui sera utile, et éliminer du traitement ce qui ne servira pas.

Le principe algorithmique commun à toutes les structures d'accélération est le suivant :

1. la structure va **partitionner** en case l'espace de la scène de manière optimale. Cette partition peut-être dynamique (modification en cours d'utilisation) ou statique (pré-calculée). Chaque case contiendra les éléments de la scène contenus dans celle-ci. **Note** : j'élude le cas où les éléments chevaucheraient plusieurs cases pour simplifier les explications, sachez juste que « ça se fait bien ».
2. On parcours la structure selon le besoin, l'idée étant d' :
    - évincer rapidement du parcours des parties qu'il serait superflu/inutile de considérer
    - aller en même temps vers la « bonne » réponse au plus vite.

Cas d'usage des structures d'accélération

Il y a typiquement 2 cas d'usage :

- collecter un ensemble utile d'objets, par exemple :
  - l'ensemble des objets visible a l'écran
  - l'ensemble des objets projetant une ombre visible à l'écran
  - l'ensemble des objets en contact avec un objet bien défini (détection de collision)
- ou sélectionner un objet bien précis : *picking* à la souris, *raytracing* de réflexions, …

Pour la détermination des objets visibles à l'écran, on parle dans le jargon d'*occlusion culling* (https://fr.wikipedia.org/wiki/D%C3%A9termination_des_surfaces_cach%C3%A9es). Une vidéo étant bien plus parlante qu'une tartine de texte vous pouvez consulter ces démonstrations du principe : https://www.youtube.com/watch?v=OmuQmydipGg (anglophone), ou https://www.youtube.com/watch?v=Qg6r7vUfR_k (en français).

Note : Alors on pourra m'objecter que « si j'ai un miroir dans la scène je vais voir des zones qui ne sont pas visibles ni illustrées dans tes vidéos ! Donc c'est du bullshit ! ». Complètement. La problématique des « cas à la con comme les (inter)réflexions » est un problème totalement ouvert encore aujourd'hui dans la 3D temps réel. L'arrivée en force du *raytracing* par NVidia l'an dernier apporte une lueur d'espoir aux développeurs, mais c'est pas pour tout de suite. Pour l'instant on se contente de faire des bidouilles comme https://en.wikipedia.org/wiki/Reflection_mapping , https://www.ea.com/frostbite/news/stochastic-screen-space-reflections , http://www.cse.chalmers.se/edu/year/2018/course/TDA361/Advanced%20Computer%20Graphics/Screen-space%20reflections.pdf . Bidouilles qui consistent à contourner la difficulté du problème par des grosses approximations à la louche.

Confusion courante…

Dans beaucoup de moteurs, les notions de graphe de scène et de structure d'accélération se superposent bien que les 2 concepts répondent à 2 problématiques séparées :

- le graphe de scène répond à un besoin **d'organiser les données**, les ranger proprement pour s'y retrouver plus tard.
- La structure d'accélération répond à une problématique de **performance**

Historiquement, le graphe de scène était un effet de bord de la structure d'accélération. Les besoins peu gourmands des applications de l'époque, il y a 10 ans et plus, avec peu de choses à afficher, ne réclamaient pas un graphe de scène très complexe. Les fameux *BSP-trees* utilisés par John Carmack (https://fr.wikipedia.org/wiki/Partition_binaire_de_l%27espace) dans le moteur de *Doom* et *Quake* en sont un bon exemple.

Implémentation

Généralement, on veut pouvoir concilier performance et praticité des algorithmes. On va donc opérer des simplifications et des approximations.

La première consiste à associer aux éléments de la scène une représentation simplifiée, plus facilement manipulable dans la structure d'accélération qu'un lapin avec un maillage de 30000 triangles. Typiquement on utilisera un volume englobant comme une sphère ou une boîte (*bounding sphere*, *bounding box*). Il est important que ce volume soit « englobant » pour éviter les faux-positifs lors des tests. En effet si l'objet réel « déborde » de son approximation, la partie débordante pourrait être visible alors que le test sur son approximation serait négatif (hors champ). Il y a une notion de conservatisme nécessaire pour éviter les ***glitchs*** graphiques (erreurs visuelles) souvent mis en avant par les utilisateurs peu conciliants avec ce genre de bugs.

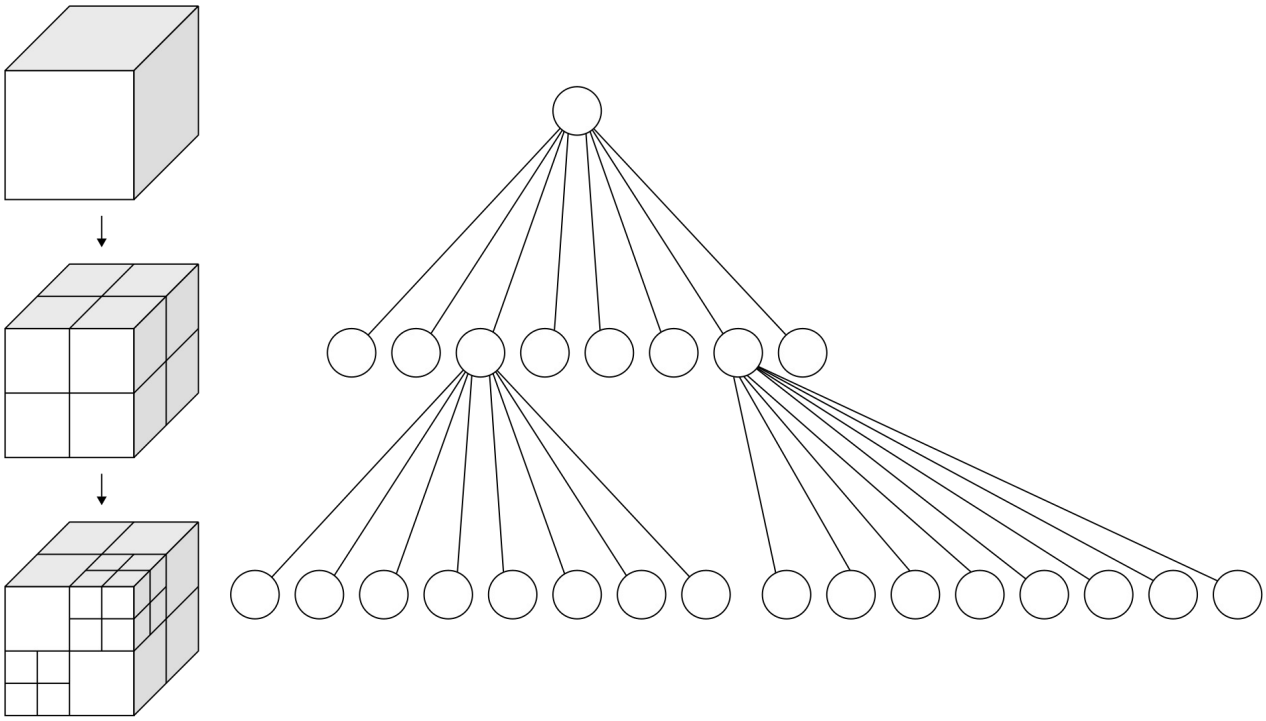
Ces éléments simplifiés serviront ensuite de briques élémentaires de construction pour fabriquer la structure d'accélération. Le principe de la structure sera de regrouper ces éléments selon certains critères comme la proximité spatiale, pour pouvoir facilement éliminer des groupes entiers d'un seul coup. Ce sont généralement des structures arborescentes où le critère de test permettra d'élaguer tout une partie de l'arbre et en éviter le parcours coûteux.

Note : Tout le défi est le juste équilibre entre précision, coûteuse en calcul, et performance, faire le moins de calcul possible. Imaginons que notre simplification soit trop grossière, certes on va gagner du temps sur les tests, mais on risque d'avoir trop de faux positifs, et au final avoir un algorithme trop conservatif retournant trop d'éléments superflus. Si on reprend le cas de l'*occlusion culling*, un tel écueil serait d'avoir sélectionnés beaucoup d'objets alors qu'ils sont complètement hors champ. Mais cela peut valoir le coût d'avoir *a priori*, des calculs plus coûteux, car le coût sera largement compensé par la précision du résultat. C'est très dépendant des conditions d'usage, et donc il n'y a pas de solution universelle.

Note 2 : Prenons par exemple, un jeu à la première personne urbain type *GTA*. Celui-ci n'a pas du tout les mêmes contraintes de visibilité qu'un *SimCity*. Dans le premier, on aura des grands immeubles masquant beaucoup d'objets, et il sera intéressant d'utiliser les immeubles comme critères de masquage (*occluders*) pour éliminer du traitement la moitié de la ville. Dans le second, les immeubles sont essentiellement vu du dessus, et les utiliser comme *occluders* serait absurde.

Ci-après je vous présente quelques grands classiques des structures d'accélération, dont certains d'entre vous auront probablement déjà entendu parler.

L’*Octree*



(source https://en.wikipedia.org/wiki/Octree)

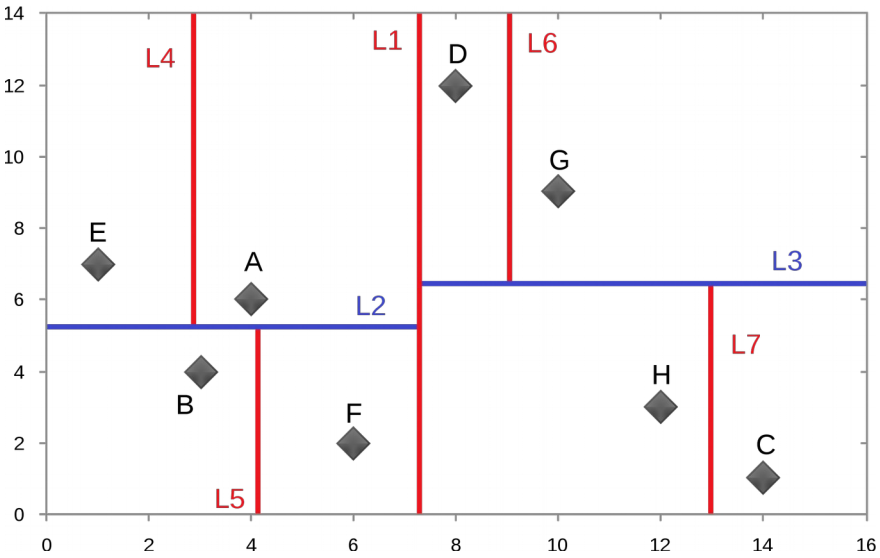
L’***octree*** est un arbre avec des nœuds de degré 8 (8 enfants). Le principe est de prendre le volume englobant alignés sur les axes (*AABB*, pour *Axis-Aligned Bounding Box*) de toute la scène, et ensuite de le couper selon les 3 axes cartésiens X, Y, Z simultanément. On se retrouve alors avec 8 nœuds fils correspondant chacun à un **octant**. Récursivement on subdivisera ensuite les fils encore en 8, et ainsi de suite. En général on coupe au milieu, mais ce n’est pas obligatoire.

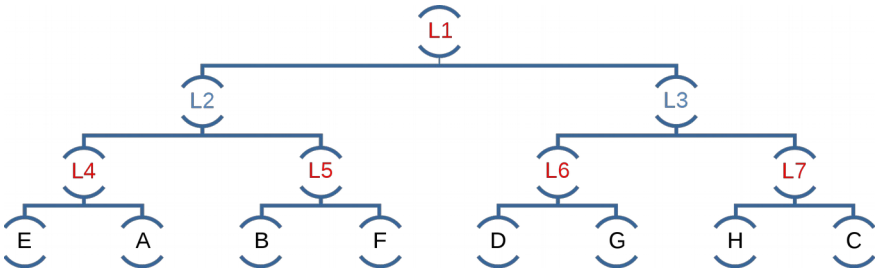
Évidemment, on ne va pas avoir une subdivision infinie, puisque l’informatique n’aime pas l’infini, il faut donc avoir un critère de **subdivision**, on parlera souvent d’**heuristique** (https://fr.wikipedia.org/wiki/Heuristique_(math%C3%A9matiques)#Heuristique_au_sens_%C3%A9troit).

Pour saisir la problématique du critère d’arrêt de subdivision, il faut comprendre que chaque case de l’*octree* stockera *in fine* une référence aux éléments de la scénographie (les arbres, la maison, la voiture, …) . Supposons que le coût de tester une case de l’*octree* coûte aussi chère que tester la boîte englobante de l’élément, ce qui est généralement le cas. Si une case de l’*octree* ne contient qu’un élément, tester la case pour savoir si l’élément est visible, puis tester l’élément ensuite pour être sûr qu’il est bien visible (car la case pourrait très bien être 100 fois plus grande que l’objet...) revient à faire 2 tests de boîte, alors que tester immédiatement l’élément aurait été plus judicieux.

« Instinctivement, tester les 8 fils ne vaut donc pas le coup, s’il y a moins de 8 éléments référencés dans le nœud ». Cette phrase est par exemple une heuristique. On cherchera donc a trouver le bon curseur pour avoir la meilleure performance, dans un cas précis, en moyenne, etc.

Le *Kd-tree*





(source https://fr.wikipedia.org/wiki/Arbre_kd)

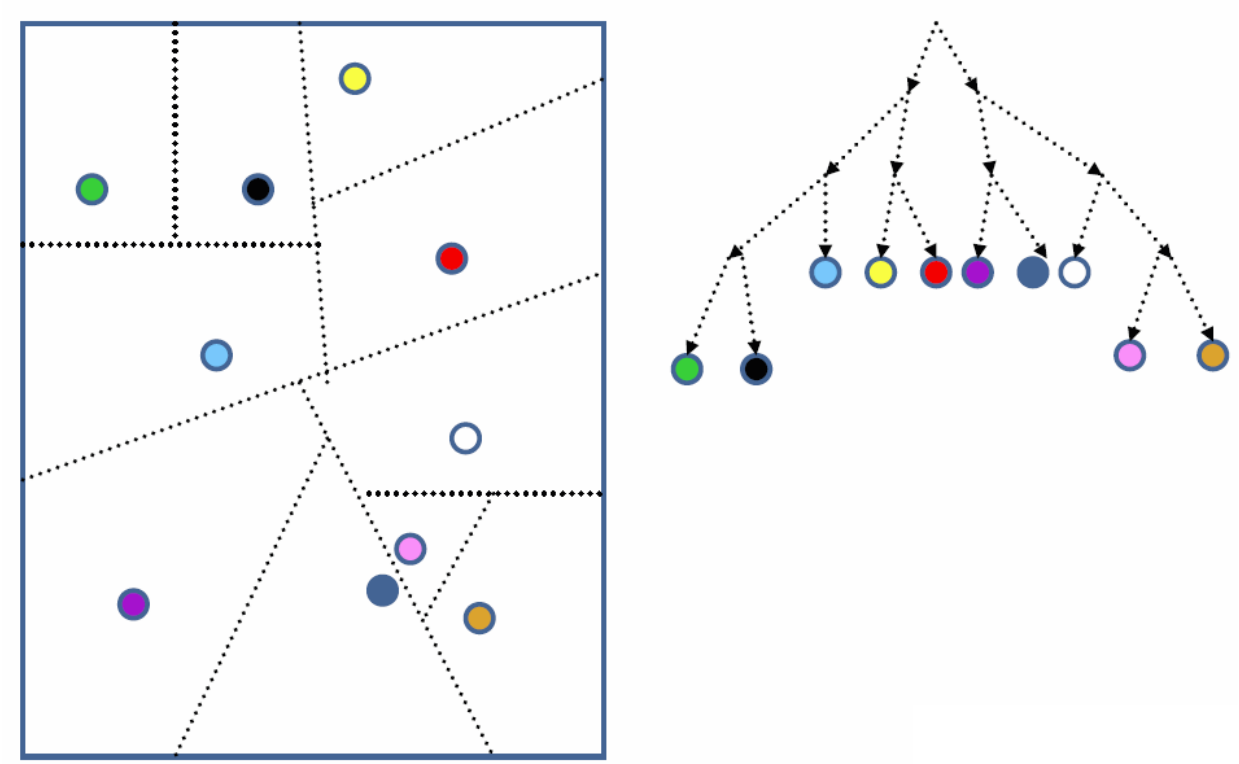
Ici, le principe est similaire à l'*octree*, on part sur un système alignés sur les axes, mais les subdivisions ne se feront que sur un seul axe à la fois.

L'avantage par rapport à l'*octree* c'est que dans ce cas on a un arbre binaire. Les arbres binaires sont généralement plus simples à organiser en mémoire. Les points de séparation sont aussi plus facile à coder (une coordonnée et un axe) et donc occupe moins de place en mémoire également. L'inconvénient c'est que l'arbre aura plus de profondeur pour réaliser une subdivision spatiale équivalente à celle d'un *octree*.

Dans exemple ci-dessus tiré de wikipédia, le *kd-tree* est en 2 dimensions. La première subdivision est verticale et faite sur l'axe des X, en coordonnées ~7,5. Cela permet de diviser l'espace en 2 zones contenant chacune 4 points. Ensuite pour chaque zone à gauche et à droite de la ligne on opère de nouveau une subdivision pour répartir les points en quantité égale de chaque coté de la ligne de séparation. Cette deuxième subdivision se fait sur l'axe Y (L2 et L3 sur le schéma). On construit ainsi l'arbre binaire.

Le choix de l'axe de subdivision et la coordonnée à laquelle subdiviser dépend de l'**heuristique** choisie. On cherchera généralement à avoir un arbre à peu près équilibré pour avoir des temps moyen, et donc des performances, en log(n) sur le parcours.

Le *BSP-tree*

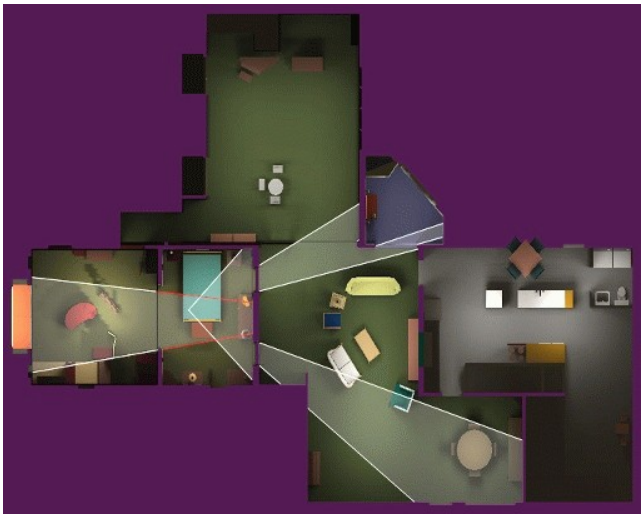


Les *BSP*-tree, pour *Binary Search Partitioning tree*, ont le même principe que le *kd-tree* mais avec des axes de coupe arbitraires, i.e. qui ne sont plus parallèles aux axes des coordonnées. Du coup il est plus difficile à implémenter, et les coordonnées de séparation sont plus complexes à coder (équation de droite en 2D, équation de plan en 3D).

L'intérêt est limité par rapport au *kd-tree* du fait de la complexité d'implémentation. Dans le cas de jeux comme *Doom* ou *Quake*, l'avantage, c'est que les plans de subdivision peuvent se confondre avec les murs des niveaux des jeux. On retrouve l'amalgame graphe de scène versus structure d'accélération.

Je ne m'étendrai pas dessus, vous trouverez plus d'infos sur la page wikipédia du sujet : https://fr.wikipedia.org/wiki/Partition_binaire_de_l%27espace .

Les *Portals* ou PVS (pour *Potentially Visible Set*)



Voir aussi https://fr.wikipedia.org/wiki/Potentially_visible_set

Ici on profite de la connaissance de l'organisation de la scène, typiquement une ville divisée en quartier, un appartement divisé en pièces, pour pré-calculer une structure, typiquement un graphe, qui organise chaque quartier/pièce comme un nœud de la structure d'accélération.

Sur l'illustration exemple ci-dessus, à gauche est la vue finale, avec un miroir au milieu et 2 chambranles de chaque coté de la commode à travers lesquels on voit partiellement la pièce suivante. À droite une vue du dessus de la situation.

Les zones trapézoïdales blanches dans l'image de droite correspondent aux volumes visibles depuis le point situé dans la 2e pièce en partant de la gauche (pointe du triangle, c'est quand même beaucoup plus simple à expliquer sur le vidéoprojecteur…).

Une zone est visible à cause du miroir sur le mur (trapèze en direction de la gauche) , et 2 autres à travers les portes de la pièce (vers la droite).

Les portes, miroirs et fenêtres définissent donc des portails (*portals*), seules zones à travers lesquelles ont peut observer la pièce suivante. Ces portails ont 2 fonctions :

- dire quelle pièce est connexe à une autre (graphe)
- contraindre le volume visible en un trapézoïde plus étroit par le phénomène d'occlusion des murs encadrant les portes.

Les volumes visibles étant plus petits, on aura moins de faux positifs, et donc des objets cachés par un mur seront bien éliminés.

Le gros défaut cette méthode est la nécessité d'une grosse intervention humaine : quelqu'un doit définir les limites, généralement à la main, et ça peut vite être fastidieux.

Conclusion sur les structures d’accélération

Évidemment, on pourrait envisager des structures hybrides, comme un *portal* qui contiendrait un *kd-tree* pour une pièce, un *octree* pour une autre. Tout est possible, l’objectif étant les performances, il faut tester et mesurer en conditions réelles.

On a fait le tour des structures d’accélération, il en existe d’autres, parfois un *brute force* linéaire sur tous les éléments est possiblement plus performants, comme on peut le voir dans cette présentation de DICE, un studio de jeu vidéo suédois d’Electronic Arts qui font le jeu *Battlefield* et le moteur 3D utilisé dans la plupart des autres studios d’EA, donc ils sont pas mauvais... https://fr.slideshare.net/DICEStudio/culling-the-battlefield-data-oriented-design-in-practice.

Note : Je le remets ici, mais j’ai déjà dû le dire lors de TDs : quand on parle de **performance**, ou d’**optimisation**, la seule chose qui fait foi, ce sont les **mesures**. La théorie, l’intuition, et le reste c’est du bullshit.

Shadow mapping

Autre sujet intéressant que certains d'entre vous ont déjà rencontré dès le TP1. Comment sont calculés les ombres dans un moteur 3D ?

Une ombre, c'est une zone qui n'est pas éclairée par la source de la lumière à cause un élément bloquant le chemin des photons, entre la zone dans l'ombre et la source lumineuse. L'élément bloquant projette donc une ombre sur cette zone de la scène. On parle de ***shadow caster***.

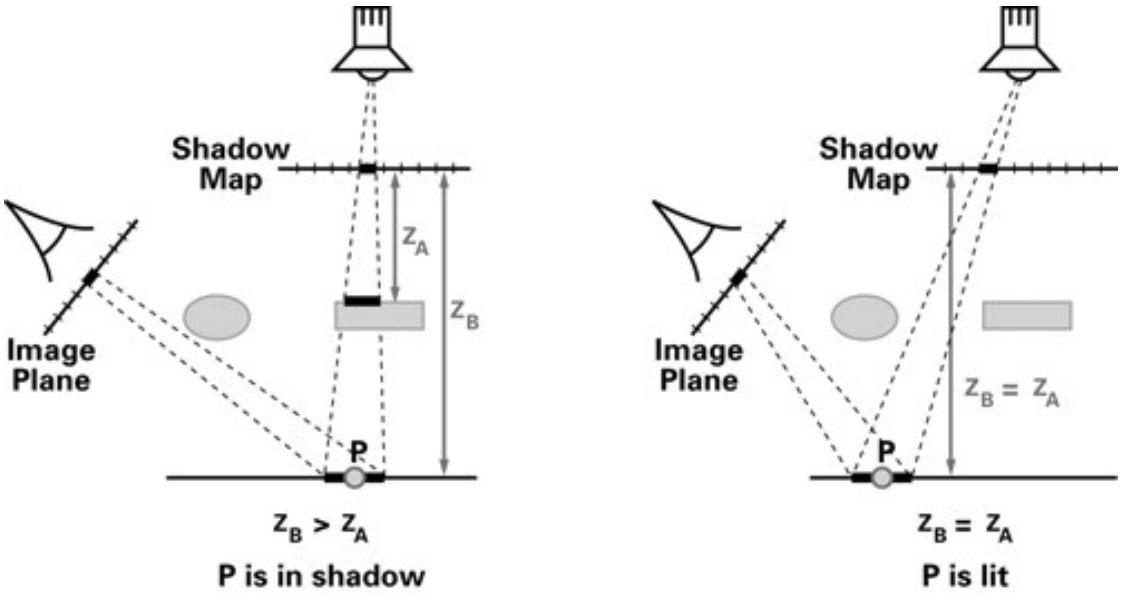
Soit un point **x** dans la scène, généralement ce point est à la surface d'un objet et correspond à un pixel visible à l'écran. Le plus simple pour savoir si ce point est à l'ombre est de voir si entre ce point et la source lumineuse, il y a un élément bloquant ou non. Les contributions de chaque source lumineuse non masquée permettront d'estimer la couleur visible de ce point.

Cette algorithme « simple » à comprendre, c'est celui utilisé dans le ***raytracing*** : on trace un rayon (segment de droite, imaginez que je suis en train de dessiner au tableau…) depuis **x** vers la source de lumière, ce rayon correspond au chemin qu'aurait pris le photon pour aller de la lumière sur l'objet et ainsi l'éclairer. Si un obstacle se trouve sur ce chemin, c'est que **x** est dans l'ombre de la source lumineuse.

Sauf que cet algorithme coûte un bras. Impossible de le reproduire aujourd'hui raisonnablement en temps réel. En effet pour savoir si un objet opère une occlusion de la lumière, le seul moyen est plus ou moins de considérer tous les objets de la scène pour vérifier s'ils se trouvent sur le chemin entre la lumière et **x**. Ça reste encore trop cher pour du temps réel.

Il a donc fallu trouver des astuces pour reproduire le phénomène de projection d'ombre, de manière détournée. Historiquement 2 algorithmes se sont battus pendant un temps, le ***shadow mapping*** et les ***shadow volumes***. Le vainqueur est le premier, le second a été rapidement abandonné. Pour votre culture je vous mets quand même un lien : https://en.wikipedia.org/wiki/Shadow_volume .

Explication sommaire de l’algorithme



Le principe est de calculer une carte des profondeurs des objets masquant la lumière, c’est la ***shadow map***. Cette carte est en fait une grille de pixel. On définit pour chaque pixel de la *shadow map* une direction, qui correspond à la droite imaginaire passant par la source lumineuse et le-dit pixel. En réalité, cette direction correspond a un petit volume pyramidal, mais on est plus a une approximation près.

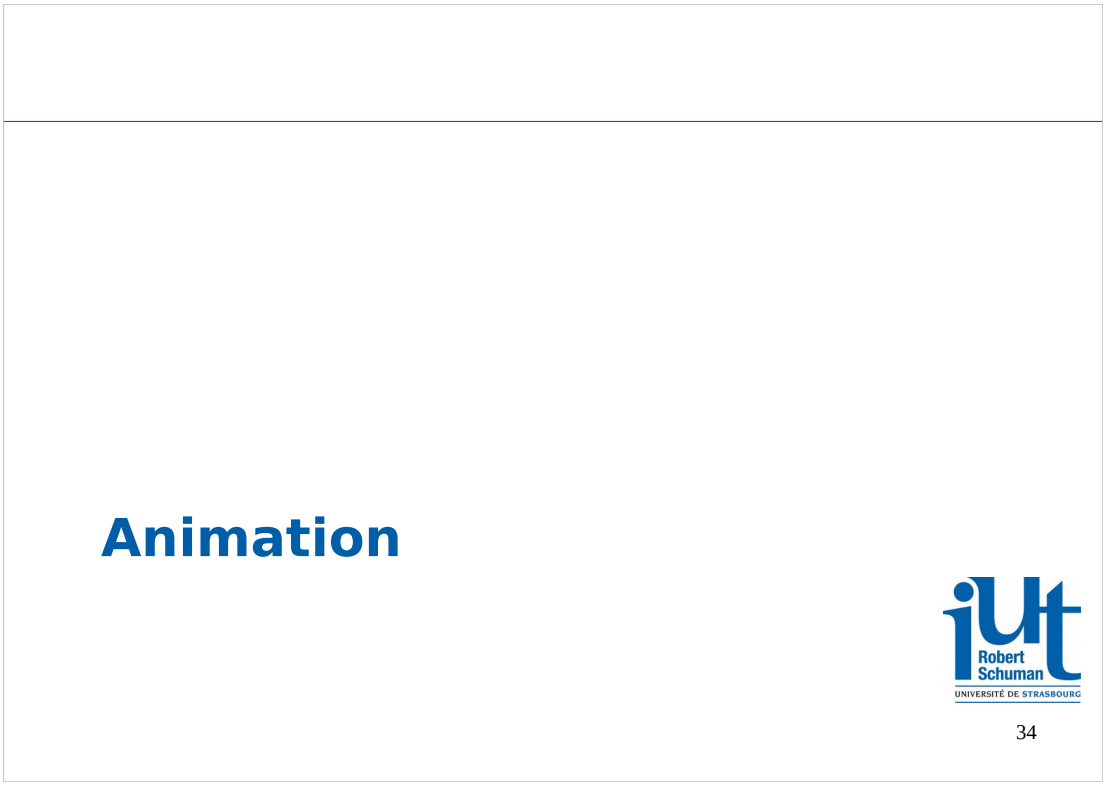
Pour chaque direction correspondant au pixel, on va stocker la distance entre la lumière et le premier objet masquant (ici les valeurs Z_A dans le schéma). Il suffit ensuite au moment de calculer la couleur en **P** de voir si la distance entre la lumière et **P** (Z_B dans le schéma) est plus grande ou égale à la valeur stockée dans la *shadow map* pour cette même direction.

L’intérêt premier de la *shadow map*, c’est la simplicité du principe. On retrouve un algorithme très proche de la rastérisation à l’écran. Ici on rastérise, mais vu depuis la lumière et dans une grille qui n’est pas un écran, mais qui est quand même une grille de pixel. On ne calcule pas des couleurs, mais des distances. L’analogie fait que l’implémentation est triviale.

Inconvénients

Le principal problème de cet algorithme, c’est qu’on utilise une grille, qui par définition à une résolution, et on se retrouve donc avec un phénomène de **crénelage** (***aliasing*** en anglais), généralement très visible. D’autres problèmes se posent également, je vous laisse voir le sujet en détail sur https://en.wikipedia.org/wiki/Shadow_mapping.

De plus, la fabrication d’une *shadow map, c’*est « calculer une image sur un écran virtuel ». Du coup c’est comme si on demandait à l’ordinateur de calculer 2 fois plus d’images que ce qu’on voit. L’image à l’écran plus celle de la *shadow map*. C’est pourquoi dans les moteurs 3D, certaines sources pas très importantes (loupiottes, diodes, décorations) ne projettent pas d’ombre car ce serait trop coûteux.



L'**animation** en image de synthèse, c'est ce qui va apporter la dimension temporelle aux scènes. Sans animation, on calcule l'équivalent d'une photographie.

Bouger la caméra avec la souris, c'est une animation. Déplacer un objet dans le temps, c'est une animation. Déformer un maillage, changer la couleur de sa surface, etc. ce sont des animations.

En fait animer, c'est faire varier des grandeurs dans le temps : les coordonnées x,y,z d'un objet, de ses sommets, les valeurs RGB de sa surface, etc.

Comment animer ?

Pour faire varier ces grandeurs, plusieurs moyens :

- en utilisant des formules en fonction du temps : x=f(t). Par exemple x = sin(t) va faire décrire à x une sinusoïde dans le temps. On parle ici d'animation **procédurale** car la valeur de la grandeur est calculée à partir d'une formule mathématique.
- En utilisant des données existantes :
  - des *keyframes* : plutôt que d'utiliser une formule pour décrire une courbe, on récupère une description discrétisée de celle-ci. Ça veut dire qu'on à la valeur de la courbe que pour certains temps clés t_0, t_1, t_2,… et entre ces temps on doit estimer la valeur que prend notre grandeur. Les valeurs aux temps clés sont les *keyframes*. Les valeurs entre ces instants clés sont alors interpolées.

La ***motion capture*** est une façon de produire des *keyframes* en capturant les mouvements réels d'un humain à l'aide de capteurs.

La **cinématique inverse** est une technique procédurale qui calcul l'état d'une chaîne cinématique à partir de contrainte mécanique (« le pied est collé au sol »).

Des illustrations sont visibles dans les slides.

Récapitulatif du module

Les slides proposent 2 ou 3 autres sujets à découvrir (les systèmes de particules, les moteurs de physique), ils feront peut-être l'objet de votre sujet de TP4, à vous de choisir. On est déjà à la 20e page, ça commence à être dense (voire rébarbatif). Je vous laisse découvrir par vous-même ces thématiques, si vous êtes curieux n'hésitez pas à me poser des questions à moi ou Paul.

P4x a été l'occasion pour vous de découvrir l'image de synthèse 3D en temps réel, et d'effleurer ses arcanes. Le contenu du module était essentiellement du dégueulage de jargon à n'en plus finir, que ce soit OpenGL, *shaders*, graphe de scène, *meshs*, *vertices*, équation de rendu, *BRDF*, *raytracing* … Ce sont des mots clés qui vous permettront d'approfondir le sujet si vous êtes curieux.

Pour aller plus loin, n'hésitez pas à consulter des ressources comme :

- http://www.pbr-book.org/ la référence sur le rendu (non-temps réel) à l'heure actuelle, les auteurs ont même reçu un oscar honorifique, tellement l'ouvrage a permis la diffusion de connaissance
- https://www.realtimerendering.com/ le site compagnon du livre éponyme, avec beaucoup de liens et de ressources, en particulier son index sur les publications de recherche actuelles sur le domaine http://kesen.realtimerendering.com/

Conclusion

Malgré le format magistral de la présentation du contenu, j'essaie d'impulser une manière différente d'apprendre et d'enseigner via les TPs. Ça ne m'intéresse pas vraiment que vous sachiez répondre à des questions formatées. La réalité des métiers de l'informatique n'a rien à voir avec ça. Certes je sais que l'objectif de « bonne note pour avoir le diplôme » reste ancré fortement et attendre autre chose de votre part est probablement utopique.

Ce qui m'importe c'est que vous ayez appris et retenu quelque chose. J'ai essayé d'être sur un mode différent pour susciter votre intérêt, et que vous appreniez par plaisir. Pour moi, c'est en s'amusant, en s'intéressant, en faisant preuve d'une sincère curiosité, qu'on apprend et qu'on retient des choses. J'espère que j'aurai réussi un peu à faire cela sur P4x.

La plupart d'entre vous ne feront pas de 3D par la suite, peut-être un peu d'algèbre linéaire, au mieux. L'image de synthèse ne vous servira à rien. Tous les sujets connexes, comme l'usage d'un débogueur, de git, l'organisation du code source, savoir blablater, vous serviront probablement bien plus. C'est pourquoi j'ai insisté sur ces points.

J'ai également cherché à développer votre démarche d'ingénierie logiciel par la réflexion sur les problèmes, les solutions, les angles d'attaques, ainsi que comment organiser votre travail pour obtenir des résultats. Cela vous sera plus utile par la suite que savoir ce qu'est une *BRDF*.

Merci à tous et amusez-vous bien sur le dernier TP !