Ubisoft - Séance 6, Tr1 & Boost
Exercices

7 mai 2009

URL Cours (Tous les slides et exercices):

`http://evasion.inrialpes.fr/Membres/Romain.Arcila/ubisoft/main.html`

1 TR1

1.1 Utilitaire - shared_ptr

Fichier: "01-utilitaire"

Remplacer les pointeurs par des shared_ptr, les fonctions doivent prendre des shared_ptr en arguments. La fonction *dynamic_pointer_cast* sera probablement nécessaire.

1.2 Functional

Fichier: "02-functional"

Le principe de ce programme est de faire un "mini-(mauvais)" mauvais moteur de jeux. Le corps du programme est écrit. Il faut:

- Finir d'écrire la classe Action (remplir le typedef) et les fonctions register_action et key: cette classe est responsable des entrées utilisateurs. Elle contient un unordered_map qui associe à une touche une action (équivalent à une fonction à appeler).
- Enregister les intéractions utilisateurs en utilisant la classe Action (par Aplayer1 et Aplayer2).
- Ecrire la boucle d'événement: déplacement des "sprites" (mouvement aléatoire avec bounded_random) et "affichage" des "sprites" et des "joueurs".

Les classes/fonctions mem_fn, bind et function seront nécessaires.

1.3 TypeTrait

Fichier: "03-metaprog"

La fonction fill peut etre implémenté comme dans la fonction fill_impl: copie élément par élément. Cette méthode n'est pas forcément la plus éfficace: la fonction memset peut être optimisée en tirant profit du matériel, cependant elle fonctionne uniquement avec des objets dont l'operateur= effectue une copie bit-a-bit et dont la taille est de 1 octet. Il faut donc utiliser cette fonction lorsque cela est possible: la fonction my_fill va donc dispatcher l'appel vers la fonction soit avec memset soit avec copie élément par élément.

memset(destination, valeur, taille)

2 Boost

2.1 Any

Fichier: "04-property"
Il s'agit d'implémenter ici un systeme de propriété dynamiques simple et minimaliste à l'aide de la classe Any. Un exemple d'utilisation est fournie dans la fonction main. Il utilise un systeme d'handle pour permettre l'acces aux propriété. Il s'agit d'écrire les méthodes:

- `request_property<Prop>(const string& prop)`: enregistre la propriété et renvoie une handle pour la propriété.
- `set_property(const Handle<Property>& h, const Prop\& p)`: met la valeur (avec erreur dans le cas d'un mauvais typage)
- `get_property(const Handle<Property>& h)`: récupère la valeur associée a la handle.
- `has_type_proprty<Prop>(const string& name, Prop p=Prop())`: renvoie un booleen si la propriété associée a name et du type p.

2.2 enable_if

Fichier: "05-enable"

La STL fournit plusieurs types d'iterateurs, notament les random_access_iterator (ceux des vectors) qui permettent les operations de type "iter+=5" contrairement aux autres iterateurs. L'idée est donc de fournir une fonction permettant d'avancer un iterateur avec la methode += (plus efficace) si l'iterateur est du type random_access_iterator et avec une boucle pour les autres.

La classe iterator_traits ainsi que le type random_access_iterator_tag peuvent être nécessaire.

2.3 concept

Fichier: "06-concept"

Comme vu précédement la fonction memset impose certains restrictions sur le type d'arguments utilisés. La fonction my_fill (version générique de memset) doit donc respecter ces criterès: écrire les contraintes nécessaires.

2.4 lambda

Fichier: "07-lambda"

- En utiisant la fonction remove_if et ls foncteur less, supprimer tous les elements inferieurs a 0.5 puis tous qui ne sont pas inferieur a 0.5 (bind est necessaire).
- Faire la memee chose en utilisant les lambda fonctions.
- Ecrire une fonction filter_if qui prend en parametre un conteneur et copie les elements de conteneur dans un nouveau conteneur suivant un predicteur (prenser a utiliser boost::function). Tester la en utilisant les fonctions lambda.
- Ecrire une fonction build qui prend en parametre deux conteneurs et construit un nouveau conteneur en appliquant une transformation a partir des deux autres. Tester la en utilisant les fonctions lambda.