Introduction à Irrlicht
seconde partie

CPE — ETI5–IMI

18–23 novembre 2016

1 Introduction

Voici la deuxième partie du TP à propos du moteur d'affichage Irrlicht.

Les différentes sections sont assez indépendantes, vous pouvez donc vous focaliser sur les parties qui vous intéressent.

Certains exemples viennent s'ajouter à ceux qui étaient proposés afin de faire un tout plus cohérent.

2 Du débug

Le moteur Irrlicht permet de visualiser des informations supplémentaires pour chaque nœud graphique. On peut par exemple ajouter l'affichage des normales d'un personnage désigné par le nœud *node* ainsi :

```
1 node->setDebugDataVisible(is::EDS_NORMALS);
```

Le paramètre de la fonction `setDebugDataVisible()` est un champ de bits où chaque bit ajoute une information. Par exemple on pourra ajouter :

- `irr::scene::EDS_BBOX` pour afficher la boite englobante du modèle. Cette boite est utilisé pour accélérer les affichages et les calculs de collisions.
- `irr::scene::EDS_MESH_WIRE_OVERLAY` pour afficher les bords des triangles en surimpression du modèle.
- `irr::scene::EDS_HALF_TRANSPARENCY` pour afficher le modèle en mode semi-transparent.

L'utilisation de ces quatre valeurs en même temps (séparées par des « | ») permet d'avoir un affichage comme sur la figure 1.

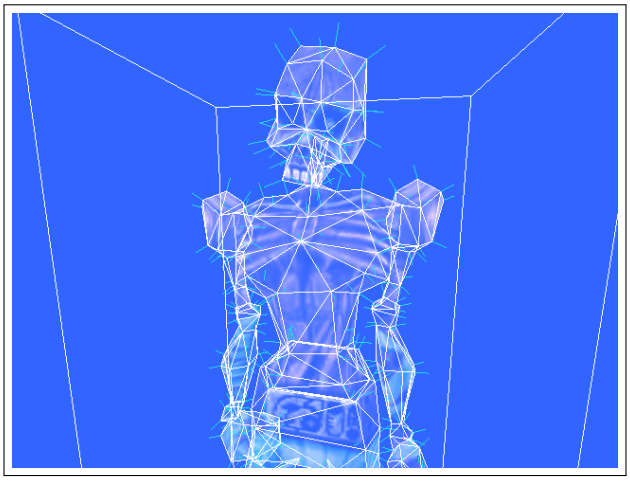

FIGURE 1 – Une version "debug" de notre personnage

Question 1 : *Consultez la documentation de la fonction `setDebugDataVisible()` afin de voir les possibilités offertes par cette fonction. Notez que toutes les informations ne sont pas forcément disponibles pour tous les modèles.*

Le code correspondant à ceci est dans `debug.tar.gz`

3 GUI

3.1 Souris

Jusqu'à maintenant, notre gestionnaire d'événements ne gérait que le clavier (`EET_KEY_INPUT_EVENT`). Le moteur Irrlicht permet de gérer bien d'autres types d'événements. En particulier, il est possible de gérer la souris (sa position, l'état des boutons, la molette, etc.)

Il suffit pour cela de tester si le type d'événement est `EET_MOUSE_INPUT_EVENT`. On peut ensuite tester la valeur contenue dans `event.MouseInput.Event`.

Cette valeur peut entre autres prendre les valeurs suivantes :
— `EMIE_LMOUSE_PRESSED_DOWN` indique que le bouton gauche de la souris vient d'être pressé
— `EMIE_LMOUSE_LEFT_UP` indique que le bouton gauche de la souris vient d'être relâché
— `EMIE_MOUSE_MOVED` indique que la souris a bougé
— `EMIE_MOUSE_WHEEL` indique que la molette de la souris a été utilisée
La position de la souris est toujours consultable dans `event.MouseInput.X` et `event.MouseInput.Y`.

Question 2 : *Consultez la documentation de `SMouseInput` afin de voir ce qu'il est possible de savoir de la souris.*

Question 3 : *Maintenant que la fonction `OnEvent()` de la classe `EventReceiver` devient imposante, séparez le fichier `main.cpp` en trois : créez donc `events.h` et `events.cpp`*

Question 4 : *Ajoutez la possibilité de faire pivoter notre personnage à l'aide de la souris, et de lui faire changer de tenue à l'aide de la molette.*

Le code correspondant à ceci est dans `souris.tar.gz`

Note : deux caméra particulières sont intégrées à Irrlicht : FPS et Maya (voir TP précédent). Ces caméras utilisent la souris pour modifier la vue, cette gestion peut entrer en conflit avec votre utilisation de la souris.

3.2 Interface complète

Un *widget* (abréviation de *window gadget*) est un élément graphique 2D permettant d'interagir avec l'utilisateur. Parmi ces éléments, on trouve :
— des labels (un bout de texte),
— des boutons à cliquer,
— des cases à cocher,
— des menus,
— des boites de dialogue,
— etc.
Notez que pour tous les affichages graphiques de texte, Irrlicht utilise des `wchar_t` au lieu de `char`. Il faudra donc utiliser des chaînes de caractères étendues : `L"bonjour"` au lieu de `"bonjour"`.

Les classes et fonctions concernant les *widgets* sont définies dans l'espace de noms `irr::gui`.

Tout ce qui concerne l'interface utilisateur est créé et géré via un objet `IGUIEnvironment` que l'on peut obtenir ainsi :

```
1  ig::IGUIEnvironment *gui = device->getGUIEnvironment();
```

Il est ensuite possible d'ajouter des `widgets` qui seront gérés automatiquement lors de l'appel

```
1  gui->drawAll();
```

de l'application.

Quelque soit son type, un *widget* possède au moins trois attributs :
— Un rectangle englobant.
— Un *widget* parent, qui le contient. Les coordonnées du rectangles sont données par rapport à ce parent. La valeur par défaut du parent est `nullptr`, ce qui signifie que le parent est la fenêtre.
— Un éventuel identifiant qui permettra de reconnaitre un *widget* ou un groupe de *widgets* lorsqu'il sera utilisé.

Comme cet identifiant sera utilisé d'une part lors de la création (dans `main.cpp`) et d'autre part pendant la fonction `OnEvent` (dans `events.cpp`), il est habituel de créer une liste d'identifiants dans un `enum` dans un fichier d'entête (`gui_ids.h`) qui sera inclu partout où l'on en a besoin.

Fenêtre

Nous allons maintenant construire une interface telle que celle présentée figure 2.

La première chose à faire est de créer la fenêtre nommée "Settings", en l'ajoutant à notre GUI :

```
1  ig::IGUIWindow *window = gui->addWindow(ic::rect<s32>(420,25, 620,460), false, L"Settings");
```

Le rectangle définit donc la taille et la position initiale de cette fenêtre (en donnant les positions des deux points extrêmes). Le `false` indique que la fenêtre ne sera pas modale, et qu'il sera donc possible d'utiliser d'autres parties de l'interface pendant que cette fenêtre sera affichée. Le troisième paramètre est le titre de la fenêtre. Il est possible d'ajouter d'autres paramètres comme un parent ou un identifiant, mais nous n'en avons pas besoin pour l'instant. Vous trouverez plus d'information dans la documentation de la fonction `addWindow()`

Question 5 : *En utilisant tout ce qui vient d'être dit, ajoutez une fenêtre à votre programme.*

On peut maintenant ajouter différents widgets dans cette fenêtre. Le plus simple est certainement le label, que Irrlicht appelle « Texte statique ».

Il est possible d'en ajouter un aussi facilement que ceci :

```
1  gui->addStaticText(L"Value", ic::rect<s32>(22,48, 65,66), false, false, window);
```

Le premier paramètre est évidemment le texte que l'on désire affiche, vient ensuite le rectangle qui représente la place que le label prendra dans notre fenêtre (dernier paramètre). Les deux "false" indiquent que l'on ne veut pas de cadre dessiné autour du texte, et qu'il ne présente pas de retour à la ligne.

Ajouter d'autres *widgets* se fait de la même manière. Il suffit de consulter les documentations des différentes fonctions "add*" de la documentation de la classe `IGUIEnvironment`.

Par exemple, on pourra ajouter un bouton ou une boite d'édition de texte avec les lignes suivantes :

```
1  gui->addButton(ic::rect<s32>(40,74, 140,92), window, WINDOW_BUTTON, L"Click me!");
2  gui->addEditBox(L"1.0", ic::rect<s32>(65,46, 160,66), true, window, WINDOW_VALUE);
```

On retrouve pour chaque ajout d'un *widget* : la définition d'un rectangle, d'un parent (`window` dans notre cas) et un identifiant (`WINDOW_BUTTON` et `WINDOW_VALUE`).

Question 6 : *Ajoutez différents widgets à votre programme.*

La barre de menu est un *widget* un peu particulier, qui n'a pas de parent et est unique. On crée une barre de menu simplement avec la ligne suivante :

```
1  ig::IGUIContextMenu *menu = gui->addMenu();
```

Mais elle est pour l'instant vide. On peut ajouter des entrées de menu de la manière suivante :

```
1    menu->addItem(L"File", -1, true, true);
```

Le premier "true" indique que l'entrée de menu est active (ce qui est la valeur par défaut), le second indique que cette entrée aura un sous-menu. Il devient ensuite possible d'ajouter un tel sous-menu ainsi :

```
1  submenu = menu->getSubMenu(0);
2  submenu->addItem(L"New game...", MENU_NEW_GAME);
3  submenu->addItem(L"Quit", MENU_QUIT);
```

Le paramètre de `getSubMenu` est un numéro d'ordre, il vaut 0 pour le sous-menu de la première entrée, 1 pour le second, etc. Les paramètres `MENU_GAME` et `MENU_QUIT` sont des identifiants qui nous permettront de savoir quelle entrée a été cliquée.

Question 7 : *Ajoutez une barre de menu avec des sous-menus dans votre programme.*

Justement, il est temps de voir comment gérer tous les événements qui proviennent de la GUI. Cela se fait dans la fonction `OnEvent`, tout comme pour les événements clavier ou souris, en traitant le type `EET_GUI_EVENT`.

Chaque type de *widget* peut produire des événements différents (que vous trouverez dans leur documentation respective). Pour savoir quel type de *widget* a déclenché un événement, il faut regarder le contenu de `event.GUIEvent.EventType`. Par exemple, on saura qu'une entrée de menu a été sélectionnée si ceci vaut `irr::gui::EGET_MENU_ITEM_SELECTED`. Un pointeur sur le *widget* ayant provoqué l'événement peut-être récupéré dans `event.GUIEvent.Caller`, qu'il faudra *caster* en un type particulier de *widget*. Cela permettra d'en savoir plus sur les conditions de l'événement. Par exemple, pour récupérer l'identifiant (`id`) de l'entrée de menu qui vient d'être cliquée, on peut utiliser le bout de code suivant :

```
1 irr::gui::IGUIContextMenu *menu = (irr::gui::IGUIContextMenu*)event.GUIEvent.Caller;
2 s32 item = menu->getSelectedItem();
3 s32 id = menu->getItemCommandId(item);
```

Ensuite, on peut utiliser cet `id` de la sorte :

```
1 switch(id)
2 {
3   case MENU_NEW_GAME:
4     // Faire quelque chose ici !
5     break;
6   case MENU_QUIT:
7     exit(0);
8   ...
9 }
```

Chaque type de `widget` peut ainsi être traité, mais le traitement dépend fortement du type de *widget*, il faut alors se référer à la documentation pour récupérer les informations voulues. Par exemple, voici le traitement d'un bouton à cliquer :

```
1 case irr::gui::EGET_BUTTON_CLICKED:
2 {
3   s32 id = event.GUIEvent.Caller->getID();
4   if (id == WINDOW_BUTTON)
5     std::cout << "Button clicked\n";
6 }
7 break;
```

Vous trouverez d'autres traitements dans le code que je vous fournis.

Question 8 : *Ajoutez autant d'éléments d'interface (dans `main.cpp`) que possible avec leur traitement (dans `events.cpp`. N'oubliez pas de mettre à jour le fichier des identifiants `gui_ids.h`.*

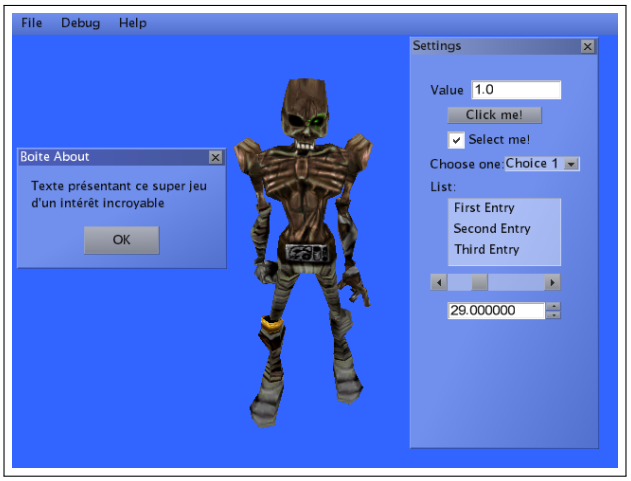

FIGURE 2 – Notre personnage entouré d'éléments d'interface utilisateur

Le code correspondant à ceci est dans `full-gui.tar.gz`

3.3 Des images 2D

Un *widget* particulier pouvant être utilisé dans un jeu sans avoir l'aspect formel des autres *widgets* est l'image 2D.

Elle ne génère pas d'événement particulier (comme un label) mais permet un affichage assez joli d'informations en 2D.

Pour ajouter un tel widget, il faut disposer d'une texture que l'on aura préalablement chargée ainsi (comme pour les textures habillant notre personnage :

```
1 irr::video::ITexture *tex = driver->getTexture("data/texture.png");
```

On peut ensuite créer le *widget* image en donnant juste le rectangle qu'il devra occuper à l'écran :

```
1   irr::gui::IGUIImage *image = gui->addImage(irr::core::rect<s32>(10,10, 50,50));
2   image->setScaleImage(true);
3   image->setImage(tex);
```

La deuxième ligne indique que la texture devra être étirée ou rétrécie pour s'adapter à la taille du rectangle et la troisième ligne associe la texture à notre widget image.

En utilisant ceci astucieusement, on peut créer une interface complète comportant un score (comme sur la figure 3), des niveaux de vie, des poings ou des armes d'un héro en vue à la première personne, etc.

FIGURE 3 – Notre personnage avec un score

Question 9 : *Ajouter des images 2D en surimpression pour donner des informations au joueur, comme un score par exemple.*

Le code correspondant à ceci est dans `2D-images.tar.gz`

3.4 Billboard

Les *billboards* ne sont pas vraiment des éléments d'interface 2D, mais ils ne sont pas non plus de véritables objets 3D.

Il s'agit d'objets présents dans la scène 3D formés d'un seul quadrilatère faisant toujours face à la caméra. Cela permet de donner l'illusion que l'on a un objet complexe alors qu'il ne s'agit que d'une image. L'imposture apparaît cependant si l'on fait le tour de l'objet.

Les billboards restent très utiles pour représenter de la végétation (comme dans notre exemple), des explosions, de la fumée, des nuages, etc.

La création d'un billboard dans Irrlicht demande un parent éventuel (pour attacher la position du billboard à un nœud), la dimension horizontale et verticale (un billboard n'a pas de profondeur) et sa position dans la scène 3D. Par exemple, on peut avoir :

```
1 irr::scene::IBillboardSceneNode *billboard;
2 billboard = smgr->addBillboardSceneNode(nullptr,
3                                         irr::core::dimension2d<f32>(50, 80),
4                                         irr::core::vector3df(0, 0, 50));
```

On peut ensuite paramétrer cet objet. Dans notre cas, on veut désactiver l'éclairage (pour éviter de révéler l'absence des nombreuses normales qui devraient être là). On veut également activer l'utilisation du canal alpha pour la transparence afin de ne pas voir les contours de l'image et il nous faut également charger une texture et l'associer au billboard :

```
1 billboard->setMaterialFlag(irr::video::EMF_LIGHTING, false);
2 billboard->setMaterialType(irr::video::EMT_TRANSPARENT_ALPHA_CHANNEL);
3 billboard->setMaterialTexture(0, driver->getTexture("data/tree.png"));
```

Mais bien évidemment, beaucoup d'autres réglages sont possibles.

FIGURE 4 – Notre personnage devant un arbre

FIGURE 5 – En bougeant la caméra, l'arbre ressemble toujours à un arbre

Question 10 : *Ajoutez un ou plusieurs billboard à votre scène.*

Le code correspondant à ceci est dans `billboard.tar.gz`

4 Collisions

4.1 Un peu de physique

Irrlicht ne présente pas un moteur physique aussi complet que Havoc ou Bullet mais permet tout de même de tester des collisions avec le décor afin de ne pas traverser les murs ni le sol, ou de pouvoir monter des escaliers ou sauter en contrebas.

Pour cela, il suffit de charger le décor comme on l'a déjà fait lors du TP précédent :

```
irr::scene::IAnimatedMesh  *mesh = smgr->getMesh("20kdm2.bsp");
irr::scene::IMeshSceneNode *node = smgr->addOctreeSceneNode(mesh->getMesh(0));
node->setPosition(ic::vector3df(-1350, -130, -1400));
```

Notez que le maillage a été ajouté à la scène sous la forme d'un octree, de sorte que retrouver les triangles autour d'un point donné de l'espace soit assez efficace.

La notion importante pour jouer avec la physique (ou les collisions comme nous allons bientôt le voir) est celle de *triangle selector*. Il s'agit d'un objet capable de choisir des triangles d'un maillage suivant certains critères, et de manière aussi efficace que possible. Dans notre cas, on utilisera ce *selector* comme paramètre d'un détecteur de collision (la position de notre caméra sera testée par rapport à chaque triangle afin d'éviter de le traverser).

Pour créer et utiliser un tel *selector*, on peut utiliser les lignes suivantes :

```
is::ITriangleSelector *selector;
selector = smgr->createOctreeTriangleSelector(node->getMesh(), node);
node->setTriangleSelector(selector);
```

On crée donc un *selector* correspondant à un octree, en donnant comme paramètre le maillage de notre décor et le nœud correspondant, puis on indique que ce *selector* est le sélecteur par défaut du nœud du décor. Cela peut sembler très redondant, mais les *triangles selector* ont beaucoup d'application et l'on pourrait imaginer des cas où l'on sélectionne les triangles d'un maillages, transformés par une matrice d'un autre nœud, le tout pouvant être géré par un troisième.

On peut alors ajouter une caméra de type "FPS". Je vous invite à consulter la documentation de la fonction `addCameraSceneNodeFPS()` si ce n'est pas déjà fait. Ici, je règle simplement les vitesses de rotation, de déplacement et de saut (touche « J »), avant de positionner la caméra et l'endroit où elle regarde :

```
is::ICameraSceneNode *camera;
camera = smgr->addCameraSceneNodeFPS(nullptr, 100, 0.3, -1, nullptr, 0, true, 3);
camera->setPosition(ic::vector3df(50, 50, -60));
camera->setTarget(ic::vector3df(-70, 30, -60));
```

Il ne nous reste plus qu'à demander à Irrlicht de gérer les collisions. Pour cela, nous allons créer un animateur, un peu comme ceux du TP précédent qui nous permettaient d'avoir un personnage qui se déplaçait en ligne droite et un autre qui tournait en rond. Un animateur est simplement un objet qui permet à Irrlicht de déplacer un nœud en particulier, soit de manière automatique, répétitive soit en respectant certains critère. Ici, nous allons utiliser un animateur qui nous replace dans les limites de chaque pièce si on essaie de traverser un triangle. En fait notre caméra (qui nous représente) est vue comme un ellipsoïde dont on donne les rayons en troisième paramètre et la position du centre en dernier paramètre :

```
is::ISceneNodeAnimator *anim;
anim = smgr->createCollisionResponseAnimator(selector, camera,
                                             ic::vector3df(30, 50, 30),
                                             ic::vector3df(0, -10, 0),
                                             ic::vector3df(0, 30, 0));
camera->addAnimator(anim);
```

Nous ne voulons pas une caméra au raz du sol, c'est pourquoi le dernier paramètre n'est pas nul. Le paramètre (0, -10, 0) est la gravité que cet animateur doit utilisé pour nous faire tomber, après un saut par exemple.

Encore une fois, je vous invite à consulter la documentation de la fonction `createCollisionResponseAnimator()` afin d'en comprendre les paramètres.

Ces quelques lignes sont suffisantes pour avoir un comportement plus physiquement réaliste.

Le code correspondant à ceci est dans `physique.tar.gz`

4.2 Sélection

Un autre type d'intersection à considérer est celle de la trajectoire d'un projectile avec un ennemi (ou d'un projectile ennemi avec le personnage que l'on représente).

Là encore, Irrlicht nous facilite grandement la vie. Nous allons à nouveau utiliser des sélecteurs de triangles et un objet prêt à l'emploi qui réalisera des intersections rayon/triangles pour nous.

L'exemple que je propose consiste à avoir plusieurs personnages (par exemple créés dans une boucle) et de permettre de les sélectionner à la souris. La figure 6 montre le résultat de cet exemple : les personnages en rouge ont été cliqués.

Pour chaque nœud de la scène que l'on veut pouvoir sélectionner, il faut ajouter un *triangle selector* ce qui peut être fait de la façon suivante au moment de la création de ces nœuds :

```
selector = smgr->createTriangleSelector(node);
node->setTriangleSelector(selector);
selector->drop();
```

Il faut également donner un identifiant (non nul !) aux nœuds sélectionnables, ce qui permettra d'avoir un traitement différent si on a un décor, des personnages différents, etc. :

```
node->setID(ENEMY_ID);
```

Il nous faut ensuite un gestionnaire de collisions qui fera le travail pour nous :

```
is::ISceneCollisionManager *collision_manager = smgr->getSceneCollisionManager();
```

Puis, dans la boucle principale de rendu, nous allons créer un rayon (une `line3d<f32>`). Cela peut-être fait de différentes manières : en utilisant les paramètres de la caméra, en calculant une trajectoire (rectiligne) d'une arme, etc. Dans le cas de notre exemple, je demande simplement au gestionnaire de collisions de créer un rayon en fonction des coordonnées de la souris (qui proviennent du gestionnaire d'événements) :

```
irr::core::line3d<f32> ray;
ray = collision_manager->getRayFromScreenCoordinates(irr::core::position2d<s32>(mouse_x, mouse_y));
```

Il ne nous reste alors qu'à appeler la fonction magique `getSceneNodeAndCollisionPointFromRay()` qui renvoie directement le nœud le plus proche intersecté par le rayon (ou `nullptr` si aucun nœud n'est atteint) :

```
irr::core::vector3df intersection;
irr::core::triangle3df hit_triangle;

irr::scene::ISceneNode *selected_scene_node =
            collision_manager->getSceneNodeAndCollisionPointFromRay(
                    ray,
                    intersection, // On récupère ici les coordonnées 3D de l'intersection
                    hit_triangle, // et le triangle intersecté
                    ENEMY_ID); // On ne veut que des nœuds avec cet identifiant

if (selected_scene_node)
  selected_scene_node->setMaterialTexture(0, wounded_texture);
```

Vous remarquerez qu'en plus du nœud lui-même, cette fonction renvoie aussi les coordonnées 3D de l'intersection ainsi que le triangle précis qui a été intersecté par le rayon. Dans notre cas, si on a un nœud, on change simplement sa texture, mais on pourrait évidemment imaginer plein d'autres choses. Notez que pour des raisons de simplicité, les personnages ne bougent pas dans cet exemple, évidemment, rien n'empêche de les faire se déplacer de manière automatique avec des animateurs comme vus lors du TP précédent.

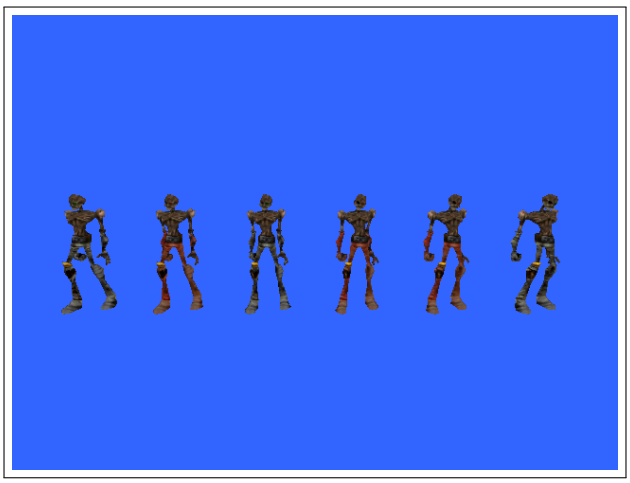

FIGURE 6 – Plusieurs personnages, ceux en rouge ont été cliqués (blessés, tués, encouragés, promus, etc.)

Le code correspondant à ceci est dans `selection.tar.gz`