

Découverte du moteur 3D Irrlicht



par [khayyam90](#)

Date de publication : 08/01/2007

Dernière mise à jour :

Ce tutoriel va présenter le moteur 3D Irrlicht ainsi que quelques unes de ses fonctionnalités de base.

- I - Introduction
 - I-1 - Qu'est-ce qu'un moteur 3D ?
 - I-2 - Fonctionnalités d'Irrlicht
 - I-3 - Installation d'Irrlicht
- II - Démarrage d'Irrlicht
- III - Utilisation d'Irrlicht
 - III-1 - Gestion de la caméra
 - III-1-A - Caméra qui suit le modèle 3D
 - III-1-B - Caméra personnalisée
 - III-2 - Gestion des archives
- IV - Pour aller plus loin
- V - Conclusion
 - V-1 - Remerciements

I - Introduction

I-1 - Qu'est-ce qu'un moteur 3D ?

Un moteur 3D est un composant logiciel permettant de manipuler, d'organiser, et d'afficher des objets 3D. Il prend en compte le support du contexte d'affichage, la gestion des différents types de fichiers et fournit des structures de données avancées pour manipuler une scène. Le moteur permet d'interfacer simplement les API de rendu 3D telles qu'OpenGL ou DirectX. Pour en savoir davantage sur les moteurs 3D, allez lire [l'excellent tutoriel](#) de Laurent Gomila à ce sujet.

La vitesse d'un jeu va directement dépendre des performances du moteur 3D, c'est la raison pour laquelle toutes les fonctions d'un moteur doivent être optimisées. De manière à avoir des performances correctes, un moteur doit implémenter des structures de données telles qu'un graphe de scène (scenegraph en anglais), un octree, etc.

Qu'est-ce qu'un graphe de scène ?

Le graphe de scène est une structure de données arborescente représentant les objets d'une scène de manière hiérarchique. Chaque noeud peut posséder des noeuds enfants et possède un noeud parent. Chaque noeud peut également posséder sa propre matrice de transformations. Chaque matrice représente les transformations du noeud par rapport à son noeud parent. Ce sont des transformations relatives.

Dans le contexte des images 3D, rappelons qu'une matrice de transformation est une matrice 4x4 pouvant représenter les rotations / translations / changements d'échelle dans un espace à 3 dimensions en utilisant des coordonnées homogènes.

La transformation absolue d'un noeud correspond à la multiplication de toutes les matrices de transformations le séparant de la racine de l'arbre. Le rendu de la scène se fait donc en parcourant le graphe de scène en profondeur d'abord de manière à dérouler toutes les multiplications de matrices de transformations. Ce processus correspond à une séquence d'empilages / dépilages de matrices.

Qu'est-ce qu'un octree ?

Un octree est une structure de données arborescente permettant de partitionner l'espace en 8 parties. Un espace 3D est divisé en deux selon chaque axe, d'où 8 partitions. La position spatiale de chaque objet dépend de sa position dans l'octree. La recherche d'objets contenus dans une zone est très rapide et s'effectue en temps logarithmique.

Le graphe de scène et l'octree sont les structures de données massivement utilisées dans les jeux vidéos pour leurs vitesses. L'octree est adapté à la représentation d'objets fixes, tels que des mondes, des cartes tandis que le graphe de scène est adapté aux objets mobiles, tels que des personnages. Il faut donc utiliser ces deux structures pour avoir des performances d'affichages optimales.

I-2 - Fonctionnalités d'Irrlicht

- Il est libre et multiplateformes
- Il gère les structures de données liées à la 3D : le graphe de scène et l'octree.
- Il gère également le BSP, qui est un arbre de partitionnement binaire, à la manière de l'octree.
- Il gère les formats de fichiers 3ds, x, md2, bsp ...
- les formats d'images classiques (jpg, bmp, png ...).
- Il peut s'interfacer entre autres avec DirectX 8, 9 et OpenGL 1.5.
- Il gère les formats d'archives zip et consors

- Il gère les pixel shaders et les vertex shaders des versions 1.1 à 3.0
- Irrlicht est un moteur C++ et C# entièrement orienté objet
- ...

Bref, il gère beaucoup de choses, pour notre plus grand bonheur.

I-3 - Installation d'Irrlicht

Irrlicht fonctionne sur les plateformes Windows, Linux et Mac. L'installation est très simple : il vous suffit de télécharger le SDK depuis le site officiel <http://irrlicht.sourceforge.net/>. Vous aurez alors les fichiers d'entête, la bibliothèque statique et la bibliothèque dynamique. Vous mettez tout ça dans votre compilateur favori et c'est bon. Le seul fichier d'entête que vous aurez besoin d'inclure est

```
#include <irrlicht.h>
```

II - Démarrage d'Irrlicht

Tout d'abord, il va falloir créer le contexte d'affichage dans lequel Irrlicht fera le rendu de la scène. Irrlicht est capable de s'interfacer aussi bien avec DirectX qu'avec OpenGL, il va donc falloir spécifier quelle API nous voulons utiliser ainsi que des informations sur le mode d'affichage. Nous aurons besoin du contexte d'affichage proprement dit mais aussi de l'interface avec l'API 3D.

```
irr::IrrlichtDevice *idevice = irr::createDevice( irr::video::EDT_OPENGL,
                                                irr::core::dimension2d<s32>(800, 600),
                                                32,
                                                false);
```

Cet appel renvoie un périphérique de sortie pour l'affichage. Le premier argument correspond à l'API choisie (irr::video::EDT_OPENGL pour OpenGL 1.5, irr::video::EDT_DIRECT3D8 pour DirectX 8, irr::video::EDT_DIRECT3D9 pour DirectX 9). Le second paramètre correspond à la résolution du périphérique à créer, ici 800 x 600 pixels. Ensuite le nombre de bits par pixels et enfin le mode plein écran ou pas.

Vous pouvez remarquer qu'Irrlicht possède beaucoup de namespaces. Tout est contenu dans irr. Pour simplifier l'écriture, il suffit de rajouter

```
using namespace irr;
```

Une fois le périphérique de sortie créé, il va falloir créer un driver pour l'interfacer ainsi que le gestionnaire de scène (qui va jouer le rôle du graphe de scène).

```
video::IVideoDriver* idriver = idevice->getVideoDriver();
scene::ISceneManager *gestionnaire = idevice->getSceneManager();
```

A ce stade là, Irrlicht est initialisé et prêt à tourner. Il lui manque juste une caméra :

```
gestionnaire->addCameraSceneNode(0, core::vector3df(0,0,0), core::vector3df(5,0,0));
```

Ceci va créer une caméra, fille de la racine du graphe de scène (dont l'adresse du père est 0), placée en (0,0,0) et pointée vers (5,0,0).

Voici, on peut faire tourner Irrlicht. Pour celà, rien de très original : une boucle qui affiche la scène. La vitesse de cette boucle correspond au framerate.

```
while(idevice->run()){
    idriver->beginScene(true, true, video::SColor(255,100,100,100));

    gestionnaire->drawAll();

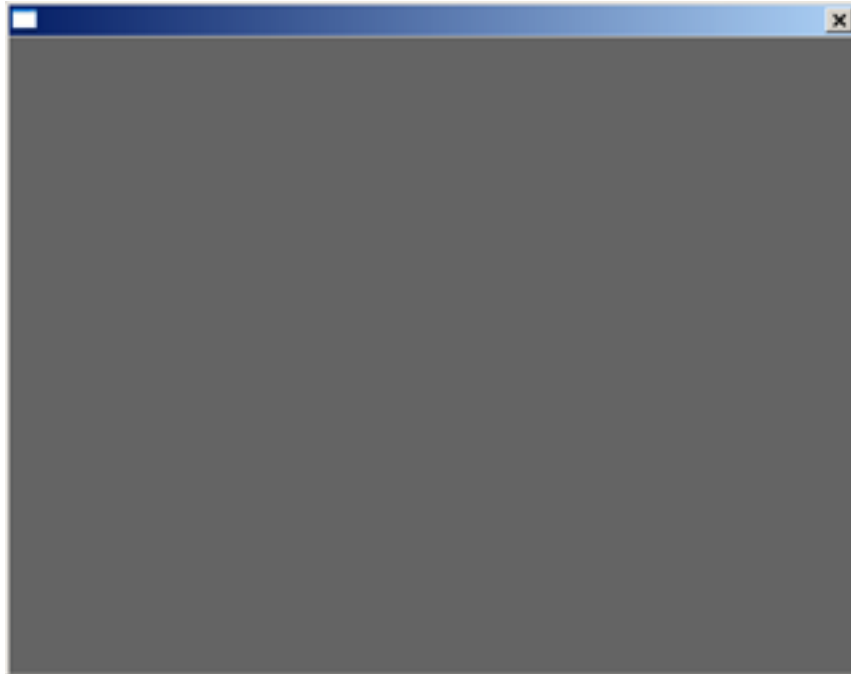
    idriver->endScene();
}
```

A chaque tour de boucle, nous commençons l'affichage en activant le backbuffer (true), le z-buffer (true) et avec une couleur de fond 255 [alpha], 100 [R], 100 [G], 100 [B]. Puis on affiche tout le graphe de scène, on termine l'affichage et on recommence.

Après la boucle, il faut penser à détruire tous les objets qui ont été créés. Irrlicht gère bien la mémoire : il nous suffit de détruire les objets qui ont été créés avec une méthode contenant 'Create'. Tous les autres objets seront automatiquement détruits. Il suffit donc d'appeler

```
idevice->drop();
```

Ce qui nous donne comme résultat une fenêtre toute vide, mais qui tourne correctement :



III - Utilisation d'Irrlicht

L'utilisation du moteur 3D concerne principalement la manipulation du graphe de scène : ajouts, modifications, suppressions. Pour ajouter un élément au graphe (et donc à la scène) il faut spécifier l'élément qui en sera le père. Cela dépend de votre hiérarchie d'objets ; un objet indépendant des autres sera fils de la racine de l'arbre, son père sera noté par 0 (NULL si vous préférez).

Pour ajouter un noeud dans l'arbre, il faut d'abord charger en mémoire l'objet graphique puis attacher une référence vers cet objet dans l'arbre. Ainsi il suffira de charger une seule instance de chaque objet même s'il est présent plusieurs fois dans une scène. Irrlicht gère les objets de plusieurs types : objets animés, lumières, caméras, terrain ... Chacun de ses types d'objets aura sa propre fonction d'ajout dans le graphe de scène.

Par exemple, pour ajouter un objet animé :

```
scene::IAnimatedMesh *imesh = gestionnaire->getMesh("le chemin vers le modèle 3D");
gestionnaire->addAnimatedMeshSceneNode( imesh );
```

Un objet animé peut être animé via une animation par frame ou bien via une animation squelettique. L'animation par frame possède une version du modèle 3D pour chaque frame (ou alors pour certaines frames clef) et le moteur va interpoler les mouvements entre ces frames. Dans le cas d'une animation squelettique, le moteur va simplement jouer l'animation déjà présente dans le modèle 3D. Ainsi pour avoir un objet animé il suffira de spécifier l'animation à reproduire :

```
scene::IAnimatedMesh *imesh = gestionnaire->getMesh("le chemin vers le modèle 3D");
scene::IAnimatedMeshSceneNode *node = gestionnaire->addAnimatedMeshSceneNode( imesh );
node->setLoopMode(true); // pour boucler sur l'animation
node->setFrameLoop(10, 20); // pour ne jouer que les frames 10 à 20, dans le cas d'une animation par
frame
// ou
node->setMD2Animation("nom de l'animation"); // pour jouer l'animation nommée, dans le cas d'une
anim squelettique
```

On remarquera que les animations sont gérées dans les noeuds du graphe de scène et non dans les instances des modèles chargés en mémoire. Ainsi on peut avoir des animations différentes pour plusieurs représentations du même modèle dans la scène.

L'ajout de terrain se fait via un octree, d'où l'intérêt de bien séparer les objets mobiles et les objets fixes.

```
scene::IAnimatedMesh *le_mesh_de_la_map = gestionnaire->getMesh("map.bsp");
gestionnaire->addOctTreeSceneNode(le_mesh_de_la_map->getMesh(0));
```

L'ajout de lumière dynamique se fait de la même manière :

```
gestionnaire->addLightSceneNode (0, // pointeur sur l'objet parent, ici la racine
                             core::vector3df(10, 0, 0), // position de la lumière, ici (10,0,0)
                             blanc, // couleur de la lumière, ici
                             );
```

Chacune des fonctions de création renvoie un pointeur sur l'objet créé. N'oublions pas que la création d'un objet peut échouer (par exemple si le modèle 3D spécifié est introuvable ou bien si on a atteint le nombre maximal de lumières autorisées par la carte graphique ...), il convient donc de tester le retour de chacune de ces fonctions. En cas d'échec, null est renvoyé.

III-1 - Gestion de la caméra

Dans l'exemple précédent, nous avons placé une caméra fixe à une position donnée et dans une orientation donnée. Irrlicht gère les caméras comme des objets pouvant écouter des évènements, ce qui signifie qu'on peut associer des actions à réaliser lors de la réception de tel ou tel évènement. Ainsi, la caméra pourra bouger si par exemple vous bougez la souris ou bien pressez une touche du clavier. Certains comportements de caméra sont déjà programmés, comme par exemple le mouvement à la première personne : la caméra tournera avec la souris et avancera au clavier. Les mouvements de caméra à la manière du logiciel Maya sont également programmés : vous pouvez orienter la caméra en cliquant sur le bouton gauche de la souris et vous pouvez le déplacer en cliquant sur le bouton droit.

```
gestionnaire->addCameraSceneNodeMaya();
// ou bien
gestionnaire->addCameraSceneNodeFPS();
```

Ainsi si on regroupe tous les objets, on peut obtenir une caméra qui se déplace dans un univers 3D :

```
int main(){
    // création du périphérique de sortie
    IrrlichtDevice *idevice = createDevice( video::EDT_OPENGL,
                                           core::dimension2d<s32>(640, 480),
                                           32,
                                           false);

    video::IVideoDriver* idriver = idevice->getVideoDriver();
    scene::ISceneManager* gestionnaire = idevice->getSceneManager();

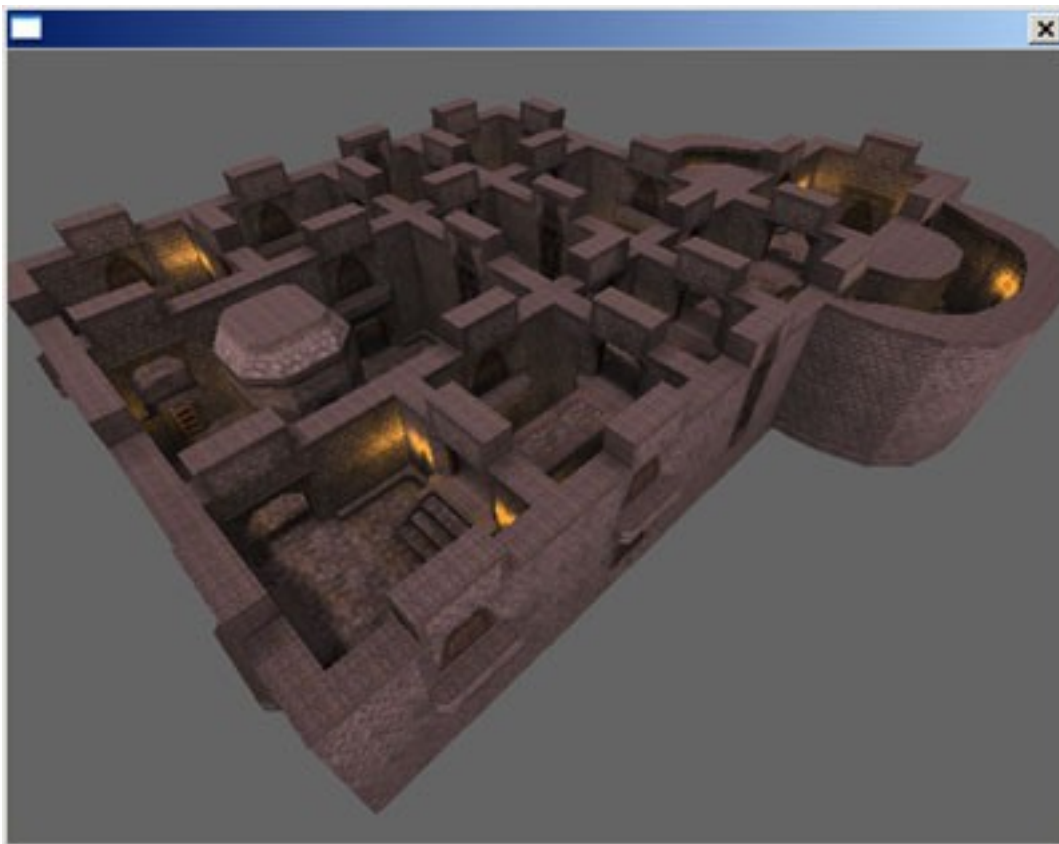
    // chargement du terrain, au format bsp
    scene::IAnimatedMesh* le_mesh_de_la_map = gestionnaire->getMesh("map.bsp");
    if (le_mesh_de_la_map)
        // utilisation du terrain par un octree
        gestionnaire->addOctTreeSceneNode(le_mesh_de_la_map->getMesh(0));

    // caméra à la première personne
    gestionnaire->addCameraSceneNodeFPS ();

    // la boucle de rendu
    while(idevice->run()){
        idriver->beginScene(true, true, video::SColor(255,100,100,100));
        gestionnaire->drawAll();
        idriver->endScene();
    }

    idevice->drop();

    return 0;
}
```



Le fichier BSP est celui fourni par défaut avec le SDK Irrlicht. Toutes les textures sont spécifiées dans le fichier BSP, les fichiers correspondants doivent être accessibles depuis l'exécutable. Irrlicht se charge du chargement des textures, c'est entièrement transparent pour l'utilisateur. On remarque qu'aucune gestion physique n'est faite, ainsi la caméra peut tout à fait passer à travers les murs ou bien s'envoler.

III-1-A - Caméra qui suit le modèle 3D

Irrlicht offre la possibilité de repérer les facettes d'un modèle 3D, pour contraindre la caméra à ne pas passer à travers les murs. De même, on peut gérer la gravité s'exerçant sur la caméra. Les options sont **beaucoup moins paramétrables** que si vous utilisiez un vrai moteur physique. Irrlicht est un moteur graphique qui offre quelques possibilités au niveau physique, ça n'est pas un moteur physique.

Pour contraindre la caméra dans le modèle 3D, il faut définir une animation à attacher à la caméra. Cette animation sera reliée à un sélecteur de triangle basé sur un octree, relié au modèle 3D. Concrètement :

```
// on charge le BSP
scene::IAnimatedMesh *le_mesh_de_la_map = smgr->getMesh("map.bsp");
scene::ISceneNode *map_node;
if (le_mesh_de_la_map)
    map_node = smgr->addOctTreeSceneNode(le_mesh_de_la_map->getMesh(0));

// on crée le sélecteur de triangles, attaché au modèle 3D
scene::ITriangleSelector* triangle_selector;
triangle_selector = gestionnaire->createOctTreeTriangleSelector(le_mesh_de_la_map->getMesh(0),
map_node);
map_node->setTriangleSelector(triangle_selector);
triangle_selector->drop();

// on crée la caméra
scene::ICameraSceneNode* camera = gestionnaire->addCameraSceneNodeFPS(
```

```

        0, 100.0f, 300.0f, -1, 0, 0,
        true); // on interdit le déplacement vertical, ainsi la caméra va rester collée au
terrain
// on crée l'animation à affecter à la caméra
scene::ISceneNodeAnimator *camera_anim = gestionnaire->createCollisionResponseAnimator(
    triangle_selector, camera,
    core::vector3df(30,50,30), // taille de l'ellipsoïde centré sur la caméra,
    qui va rechercher les collisions
    core::vector3df(0,-3,0)); // gravité, en unités par seconde

camera->addAnimator(camera_anim);
camera_anim->drop();
    
```

III-1-B - Caméra personnalisée

Si les caméras prédéfinies telles que FPS ou Maya ne vous conviennent pas, vous pouvez créer les vôtres. La création de caméra personnalisée se fait en dérivant la classe de réception de messages et en la liant au périphérique d'affichage. Le périphérique va capturer tous les messages, les envoyer à l'écouteur d'évènements qui va choisir les traitements à effectuer. Cette dérivation va même permettre d'intercepter les évènements clavier qui n'auront aucune incidence sur la caméra.

```

class MonEcouteur : public IEventReceiver {
private:
    scene::ISceneManager* smg;

public:
    MonEcouteur(scene::ISceneManager *s){
        smg = s;
    }

    virtual bool OnEvent(SEvent event){
        // étude de l'évènement intercepté

        return true;
    }
};

// [...]

scene::ISceneManager* gestionnaire = idevice->getSceneManager();
MonEcouteur ecouteur(gestionnaire);
idevice->setEventReceiver(&ecouteur);

gestionnaire->addCameraSceneNode();
    
```

Dans ce code, on crée notre propre écouteur d'évènements avec sa fonction de capture OnEvent et on crée une caméra sans rien de précis : addCameraSceneNode();. En passant le ISceneManager* en constructeur de notre écouteur, on a accès au graphe de scène dans la gestion des évènements. On peut donc modifier les propriétés de notre caméra et de tous les objets de la scène.

III-2 - Gestion des archives

Irrlicht gère nativement les archives au format zip, il va ainsi émuler un nouveau système de fichiers. Vous pouvez rajouter des chemins dans ce système de fichiers, y ajouter des archives. Pour tout chargement de ressource, Irrlicht ira chercher dans le système de fichiers émulé et dans le PATH de votre système s'il ne l'a pas trouvé dans le système de fichiers émulé.

Le fonctionnement est très intuitif : il suffit d'ajouter une archive au système de fichiers :

```

idevice->getFileSystem()->addZipFileArchive("le chemin de l'archive");
    
```

Et pour les prochains chargements de fichiers, Irrlicht ira également regarder dans ce système de fichiers émulé.

IV - Pour aller plus loin

Irrlicht essaie de gérer des fonctionnalités très diverses : il permet de gérer des fichiers xml (!), il a son système d'interfaces graphiques, etc. J'aurais préféré des fonctionnalités propres à un moteur de rendu. Si je dois utiliser des fichiers xml ou bien gérer une interface graphique à côté de mon moteur Irrlicht, je choisirais d'autres bibliothèques spécialement dédiées au traitement xml ou aux interfaces graphiques. Ces fonctionnalités *bonus* sont peu pratiques et font davantage office de gadgets, mais qui peuvent s'avérer pratiques s'il vous faut simplement et rapidement un résultat exploitable.

Il y a peu de ressources sur Irrlicht disponibles sur Internet. On y retrouve principalement la doc officielle, les tutos officiels et les exemples officiels, tout ça sur <http://irrlicht.sourceforge.net/>.

Mais quel est finalement l'avantage d'Irrlicht par rapport aux autres moteurs existants ?

- Il est extrêmement léger, avec une seule dll. C'est loin d'être une usine à gaz.
- Il est très intuitif.

V - Conclusion

Irrlicht est un moteur 3D qui mérite qu'on s'y attarde. Sa communauté de développement et d'utilisation est assez active et propose régulièrement des nouvelles fonctionnalités et/ou outils. Le moteur est devenu mature, c'est une valeur sûre pour le développement de jeux.

V-1 - Remerciements

Je tiens à remercier l'équipe de la rubrique 2D/3D/Jeux de [developpez.com](#) pour leur relecture et [Miles](#) pour la correction orthographique.