Réalité Virtuelle Irrlicht

Anthony Chomienne

CPE Lyon 2021

Irrlicht

Ce TP est une reprise de certains tutoriaux officiels Irrlicht en version simplifiée afin de se concenter sur un élément précis du moteur à la fois.

Objectifs

- Prendre en main Irrlicht
- Gestion des nodes
- Gestion de la caméra
- Chargement de modèle (environnement, personnage)
- Gestion des évènements
- Gestion des collisions

Note:

Irrlicht est un moteur de jeu extrèmement simple. Il n'offre clairement pas toutes les possibilités d'un Unity mais à l'avantage d'être léger, simple et open source, ce qui permet de faire des modifications du moteur pour nos besoins. Dans le projet que vous avez récupéré, il y a plusieurs dossiers qui vont servir a différent moment.

- include
- lib
- projet
- source

Source, lib et include seront utiles pour le déploiement sur Android. Projet va être utilisé à la fois pour Android et pour Linux.

Linux

Dans le dossier projet, il y a encore pas mal de fichier et dossier qui ne seront utiles que pour Android. Seul le Makefile et le fichier main.cpp se trouvant dans src/main/cpp seront utiles pour linux

Depuis le dossier projet:

```
make
./build/pc/irrlicht-linux
```

Actuellement le code ne fait strictement rien, rien ne sert vraiment de le lancer.

Première fenêtre

Si vous ouvrez le code, vous retrouvez trois fonctions:

```
void main_function(IrrlichtDevice *device, MyEventReceiver* receiver, void *app_param);
void android_main(android_app* app);
int main();
```

La fonction **android_main** est la fonction d'entrée d'une application Android C++, elle ne nous intéressera pas avant la partie Android.

La fonction **main** est la fonction habituelle quand on fait du C ou C++.

La fonction **main_function** sera utilisé par les deux fonctions précédentes pour nous permettre d'avoir un seul et unique code (ou presque) indépendant de la plateforme (PC ou Android).

Dans un premier temps, seule main et main_function nous intéresseront donc.

La première fonction a appelé pour pouvoir ensuite utiliser le moteur irrlicht est **createDevice**. Elle renvoi un pointeur sur un IrrlichtDevice qui sera nécessaire pour pouvoir ouvrir une fenêtre, charger et utiliser des modèles 3D...

Il y a 7 paramètres:

- deviceType qui correspond a la façon dont on va utiliser la carte graphique ou le rendu logiciel (DirectX, OpenGL...)
- windowSize qui correspond à la taille de la fenêtre. (ne pas choisir Fullscreen tant que vous n'avez pas de moyen de quitter)
- bits qui correspond au nombre de bit de couleur par pixel
- fullscreen si on veut la fenêtre en fullscreen ou non
- stencilbuffer pour avoir ou non des ombres
- vsync si on veut activer vsync
- eventReceiver qui correspond au receiver d'evennement. C'est lui qui nous permettra de traiter les évennements clavier, souris...

Dans notre fonction main nous aurons donc ce qui suit pour une fenêtre de rendu opengl en 640x480, 16bit de couleur par pixel.

```
IrrlichtDevice *device = createDevice( iv::EDT_OPENGL, dimension2d<u32>(640, 480), 16,
                                       false, false, false, 0);

if (!device) //Si le device n'a pu être créé on retourne 1
    return 1;
```

Nous pouvons remplacer le premier nullptr de l'appel à main_function par le device que nous venons de créer. Nous ne modifierons pas la fonction **main** avant la gestion des évènements, tout le reste se fera dans **main_function**

Le device contient tout ce qui nous sera nécessaire pour gérer notre jeu:

- le VideoDriver pour le chargement des textures par exemple
- Le SceneManager pour le chargement des modèles mais également la gestion de la scène
- le GuiEnvironment pour tout ce qui est interface utilisateur

La boucle principale de notre programme sera la suivante:

```
iv::IVideoDriver* driver = device->getVideoDriver();
is::ISceneManager* smgr = device->getSceneManager();
ig::IGUIEnvironment* guienv = device->getGUIEnvironment();

while(device->run())
{
    driver->beginScene(true,true,iv::SColor(255,100,200,100));
```

```
    smgr->drawAll();
    guienv->drawAll();

    driver->endScene();
}
```

La couleur fournie est la couleur d'effacement de la fenêtre. Ensuite on demande au **SceneManager** de faire son rendu. On demande la même chose au **GuiEnvironment**. Dans notre cas, ils ne font strictement rien puisque nous n'avons rien ajouté ni à l'un ni à l'autre.

Figure 1: Première fenêtre vide

Charger un modèle 3D et une caméra

Une fenêtre vide c'est bien mais ce serait mieux si nous avions un modèle 3D dedans ainsi qu'une caméra pour notre scène. Le **SceneManager** fonctionne à l'aide d'un arbre de noeud. Chaque noeud est lié soit à la racine soit à un autre noeud. S'il est lié à un autre noeud, la position de l'élément est exprimée dans le repère du noeud auquel il est lié et si l'on bouge le noeud auquel il est lié alors il bouge avec lui.

Le chargement d'un modèle se fait en plusieurs étapes, la première consiste à charger le mesh. Ce dernier est au format md2, un format contenant un certain nombre d'animation pré-existante (debout, accroupi, marche, saut...)

```
ic::stringc dataPath = "src/main/assets/data/";
is::IAnimatedMesh* mesh = smgr->getMesh(dataPath+"tris.md2");
if (!mesh)
{
    device->drop();
    exit(1);
}
```

Le mesh est chargé mais n'est pas encore utilisé dans la scène pour cela, il faut utiliser une des fonctions add du **SceneManager**, ici nous avons un mesh animé donc *addAnimatedMeshSceneNode*. L'animation choisie sera "debout", vous pouvez retrouver les animations disponibles dans la documentation de Irrlicht. Nous n'avons pas activé la gestion de l'éclérage ni ajouter de lumière dans la scène donc nous désactivons la lumière sur le modèle et nous lui choisissons une texture qui a été conçu pour ce modèle.

```
is::IAnimatedMeshSceneNode* node = smgr->addAnimatedMeshSceneNode( mesh );
node->setMaterialFlag(iv::EMF_LIGHTING, false);
```

```
node->setMD2Animation(is::EMAT_STAND);
node->setMaterialTexture(0, driver->getTexture(dataPath + "blue_texture.pcx"));
```

Ensuite nous ajoutons une caméra à la scène en position (40,30,0) et regardant la position (0,5,0).

```
is::ICameraSceneNode* camera = smgr->addCameraSceneNode(nullptr, ic::vector3df(40,30,0),
                                                        ic::vector3df(0,5,0));
```

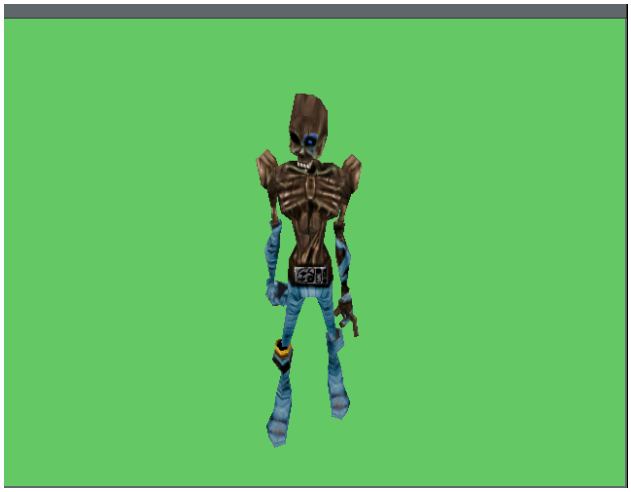

Figure 2: Modèle avec Camera

Chargement de la carte

Base

Irrlicht gère les cartes de Quake 3 au format pk3. Le format pk3 est en réalité une archive zip contenant un certain nombre de dossier utilisé pour les textures, les objets de la carte, les shaders...

La première chose à faire pour charger une carte est donc d'ajouter à irrlicht le pk3.

```
device->getFileSystem()->addFileArchive(dataPath+"map-20kdm2.pk3");
```

Ensuite nous chargerons le *bsp* à savoir le mesh de la carte en elle-même

```
is::IQ3LevelMesh* map = (is::IQ3LevelMesh*) smgr->getMesh("20kdm2.bsp");

if (!map)
{
    device->drop();
    exit(1);
}
```

Une fois celui chargé nous l'ajouterons à la scène mais non comme un animatedMeshSceneNode mais comme un Octree. L'octree a pour but de ne rendre dans la scene que ce qui est visible et donc optimise le rendu.

```
is::IMeshSceneNode* mapNode = smgr->addOctreeSceneNode(map->getMesh(
                                    is::quake3::E_Q3_MESH_GEOMETRY), 0, -1, 1024);
```

Le décor, le personnage et la caméra n'étant pas tout à fait au même endroit, nous allons déplacer ces deux derniers. Il s'agit des mêmes coordonées que précédemment mais déplacer de (1300,104,1250) dans le repère monde.

```
camera->setPosition(ic::vector3df(1340,134,1249));
camera->setTarget(ic::vector3df(1300,109,1249));
node->setPosition(ic::vector3df(1300,104,1249));
```

Figure 3: Un Décor

Shader

Certaines cartes Quake 3 embarque un ou plusieurs shader. Pour les charger c'est relativement simple. Le bsp contient des mesh aditionnels contenant certaines infos sur les shaders. La première étape consiste donc à extraire les mesh additionnels.

```
const is::IMesh * const additional_mesh = map->getMesh(is::quake3::E_Q3_MESH_ITEMS);
```

Au sein de ce mesh se trouvent un certain nombre de buffers que nous devront parcourir et dont nous extrairons les données afin d'ajouter les shaders à notre scène. L'index du shader est contenu dans le matériel du meshBuffer considéré. Le code qui suit montre les différentes étapes pour charger un shader s'il existe

```
for ( u32 i = 0; i!= additional_mesh->getMeshBufferCount(); ++i )
{
    const is::IMeshBuffer* meshBuffer = additional_mesh->getMeshBuffer(i);
    const iv::SMaterial& material = meshBuffer->getMaterial();

    const s32 shaderIndex = (s32) material.MaterialTypeParam2;

    const is::quake3::IShader *shader = map->getShader(shaderIndex);
    if (0 == shader)
    {
        //pas de shader, donc on continue
        continue;
    }
    smgr->addQuake3SceneNode(meshBuffer, shader);
}
```

Figure 4: Une torche allumée

Si vous voulez tester d'autres carte, vous pouvez regarder du côté de https://lvlworld.com/ attention cependant toutes ne sont peut-être pas compatible avec Irrlicht.

Gestion d'évènement

EventReceiver

Le device que nous avons créé au début a comme dernier paramètre un **IEventReceiver**. Ce dernier permet de traiter les évènements (clavier, souris, joystick, gui...) se produisant pendant le fonctionnement de votre jeu. La fonction principale de cet EventReceiver est OnEvent qui reçoit un SEvent. Ce dernier contient toutes les informations concernant l'évenèment produit:

- le type d'évennement
- les informations associées a ce type d'évennement (touche pressé, position de la souris lors d'un clic, mouvement...

Ainsi vous pourrez traiter ledit évènement en fonction de vos besoins.

```
#ifndef EVENTRECEIVER_H
#define EVENTRECEIVER_H

#include <irrlicht.h>

class EventReceiver : public irr::IEventReceiver
{
public:
    EventReceiver();
    virtual bool OnEvent(const irr::SEvent& event);
};

#endif
```

En dessous, vous trouvez un exemple de traitement, pour chaque touche clavier pressé ou relaché on affiche la touche concernée et l'etat pressé (true ou false) de celle-ci.

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

```cpp
#include <iostream>

using namespace irr;


EventReceiver::EventReceiver()
{
}

bool EventReceiver::OnEvent(const SEvent& event)
{
    if (event.EventType == irr::EET_KEY_INPUT_EVENT)
        std::cout<<event.KeyInput.Key<<" pressed:"<<event.KeyInput.PressedDown<<std::endl;

    return false;
}
```

Il faut fournir une instance de notre EventReceiver au device lors de la création de celui-ci, pour cela il faut modifier notre fonction main légèrement.

```cpp
#include "eventreceiver.h"

//...
int main()
{
    EventReceiver receiver;
    IrrlichtDevice *device = createDevice(iv::EDT_OPENGL, ic::dimension2d<u32>(640, 480),
                                          16, false, false, false, &receiver);
    if (!device)
        return 1;
    main_function(device,&receiver,nullptr);
    return 0;
}
```

Pensez à modifier votre makefile pour ajouter les nouveaux fichiers à la compilation

Mouvement

OnEvent est appelé une fois par touche pressée et potentiellement plusieurs fois si la touche est maintenue mais cette fonction ne traite qu'un évènement à la fois. Si nous voulons en traiter plus pour gérer par exemple un déplacement, la solution n'est pas de traiter l'évènement dans l'EventReceiver mais dans la boucle principale. L'EventReceiver va stocker l'état de l'ensemble des touches afin que l'on puisse savoir dans la boucle principale l'ensemble des touches pressées et fournir un moyen d'accéder à cette information depuis la boucle principale.

Le framerate peut varier en fonction de la quantité de donnée à afficher, de l'utilisation cpu/gpu, le mouvement doit donc être indépendant de ce fait. Pour cela nous pouvons utiliser le timer interne de Irrlicht afin d'obtenir sa valeur entre deux frame et donc calculer le temps écoulé et du coup la distance à parcourir pour notre mouvement

```cpp
u32 then = device->getTimer()->getTime();

//vitesse en unité par seconde
const f32 MOVEMENT_SPEED = 5.f;

while(device->run())
{
    // Work out a frame delta time.
```

```
    const u32 now = device->getTimer()->getTime();
    const f32 frameDeltaTime = (f32)(now - then) / 1000.f; // Temps en secondes
    then = now;

    //pseudo code:
    //Si touche Z pressé alors la camera va de l'avant
    //Sinon si touche S pressé alors la camera va de l'arrière
    //Si touche Q pressé alors la camera tourne vers la gauche
    //Sinon si touche D pressé alors la camera tourner vers la droite
    //mettre à jour position de la caméra, sa rotation et sa target

    //... le reste de la boucle principale
    // smgr->drawAll();
    //...
}
```

Collision

Les collisions sont gérée avec un ou plusieurs TriangleSelector. Ceux-ci sont obtenus à partir d'un Mesh, ici c'est un OctreeTriangleSelector puisque nous voulons gérer la collision avec la carte.

Une fois obtenu le triangle selector nous allons créer une réponse animée a notre collision (le mur/sol nous repousse). Le volume de notre objet (ici la caméra) est construit sous là forme d'une ellipsoïde (30,40,30), la gravité est représentée par le vecteur (0,-10,0), le dernier paramètre est pour déplacer le centre de l'ellipsoïde, dans le cas de notre caméra pour qu'elle soit un peu au-dessus du sol.

```
is::ITriangleSelector *selector = smgr->createOctreeTriangleSelector(mapNode->getMesh(),
                                                                     mapNode, 128);
mapNode->setTriangleSelector(selector);

is::ISceneNodeAnimator* anim = smgr->createCollisionResponseAnimator(selector, camera,
                                                                     ic::vector3df(30,40,30),
                                                                     ic::vector3df(0,-10,0),
                                                                     ic::vector3df(0,30,0));
camera->addAnimator(anim);
```

Références

- Tutoriaux Irrlicht https://irrlicht.sourceforge.io/?page_id=291
- Documentation Irrlicht https://irrlicht.sourceforge.io/docu/
- Documentation Android https://d.android.com