

# Compte Rendu des TP 3 & 4 de Moteurs de jeux

## M2 IMAGINA

Jérémy DAFFIX  
Arnault LEMMEL  
Fatma MILADI

<https://github.com/jeremydaffix/hmin317-tp3>

### Synchronisation des fenêtres

Nous avons repris le code du TP précédent qui créait 4 instances de MainWidget avec des fréquences de rafraîchissement différentes.

Ces instances sont ensuite passées au constructeur d'un objet de type Calendar (héritant de QObject, la classe des base des objets QT, qui implémente notamment le système de Slots / Signaux) :

```
MainWidget widget1(1, 0), widget10(10, 1), widget100(100, 2), widget1000(1000, 3); // 4 fenetres

Calendar calendar(&widget1, &widget10, &widget100, &widget1000); // timer pour les saisons

widget1.show();
widget10.show();
widget100.show();
widget1000.show();|
```

A noter que nous n'avons pas créé nos propres Slots / Signaux, car cela ne nous semblait pas adapté à ce qui était demandé dans le sujet, c'est à dire passer à chaque scène un message contenant directement le numéro de saison à afficher.

En effet, le système de Slots de QT permet, pour simplifier, d'abonner des méthodes à un événement donné (-> Pattern Observer). Ensuite, à chaque fois que cet événement survient, toutes les méthodes abonnées sont automatiquement appelées, avec éventuellement un paramètre propre au type d'événement (touches de clavier pressées, position de la souris, etc).

En revanche, nous avons bien utilisé l'événement timerEvent exposé par un QTimer contenu dans l'objet Calendar :

```
timerSeason.start(100, this); // 1j = 100ms
```

Cet événement s'occupe d'incrémenter le compteur de jours, et, lorsqu'il est temps de changer de saison, demande à chacun des MainWidget d'afficher la saison suivante, de manière "roulante" :

```
// un jour est passé
void Calendar::timerEvent(QTimerEvent *e)
{
    // changement de saison :
    // envoi d'un message à chaque fenêtre

    if(currentDay == 92) {

        widget1->setSeason(0);
        widget2->setSeason(1);
        widget3->setSeason(2);
        widget4->setSeason(3);
    }

    else if (currentDay == 183) {

        widget1->setSeason(1);
        widget2->setSeason(2);
        widget3->setSeason(3);
        widget4->setSeason(0);
    }

    else if (currentDay == 274) {

        widget1->setSeason(2);
        widget2->setSeason(3);
        widget3->setSeason(0);
        widget4->setSeason(1);
    }

    else if (currentDay == 0) {

        widget1->setSeason(3);
        widget2->setSeason(0);
        widget3->setSeason(1);
        widget4->setSeason(2);
    }

    currentDay = (currentDay + 1) % 365; // jour + 1
}
```

De cette manière, chaque fenêtre fait dérouler les saisons une à une, et à un instant T, les 4 saisons sont affichées.

### Simulation des changements de saison

Nous avons réutilisé le shader de coloration de sommets du TP précédent, associant une couleur à la "hauteur" de la position du pixel.

Chaque saison a un shader qui lui est dédié, et qui reprend le même principe, en faisant varier simplement les couleurs (du orange pour l'automne, beaucoup de vert pour l'été, de la neige pour l'hiver, etc).

Par exemple, voici le shader utilisé par le terrain hiver :

```
// Calculate vertex position in screen space
gl_Position = mvp_matrix * a_position;

if (a_position.z > 1.1) // neige
{
    terrain_color = vec3(1.0, 1.0, 1.0);
}

else if (a_position.z > 0.8) // neige moins blanche
{
    terrain_color = vec3(0.9, 0.9, 0.9);
}

else if (a_position.z < 0.60) // eau gelee
{
    terrain_color = vec3(0.7, 0.7, 0.85);
}

else // végétation enneigee
{
    terrain_color = vec3(0.6, 0.9, 0.6);
}
```

La méthode `setSeason()` de `MainWidget` se contente de changer le shader associé au terrain. Cela est rendu très simple par l'organisation de notre moteur et par son gestionnaire de scène : le terrain est un objet, et on appelle sa méthode `setShader()` pour définir le shader avec lequel le rendre :

```
void MainWidget::setSeason(int season)
{
    qDebug() << "Scene " << idScene << " : SEASON " << season;

    if(season == 0) terrain->setShader(&shaderTerrainWinter);
    else if(season == 1) terrain->setShader(&shaderTerrainSpring);
    else if(season == 2) terrain->setShader(&shaderTerrainSummer);
    else terrain->setShader(&shaderTerrainAutumn);

    update();
}
```

Un shader est chargé en une ligne, grâce à la méthode `loadShader()` :

```

// méthode pour simplifier le chargement d'un shader
void MainWidget::loadShader(QOpenGLShaderProgram &shader, QString vpath, QString fpath)
{
    // Compile vertex shader
    if (!shader.addShaderFromSourceFile(QOpenGLShader::Vertex, vpath))
        close();

    // Compile fragment shader
    if (!shader.addShaderFromSourceFile(QOpenGLShader::Fragment, fpath))
        close();
}

// chargement de tous les shaders
void MainWidget::initShaders()
{
    GameScene::setCurrentNumInstance(idScene);

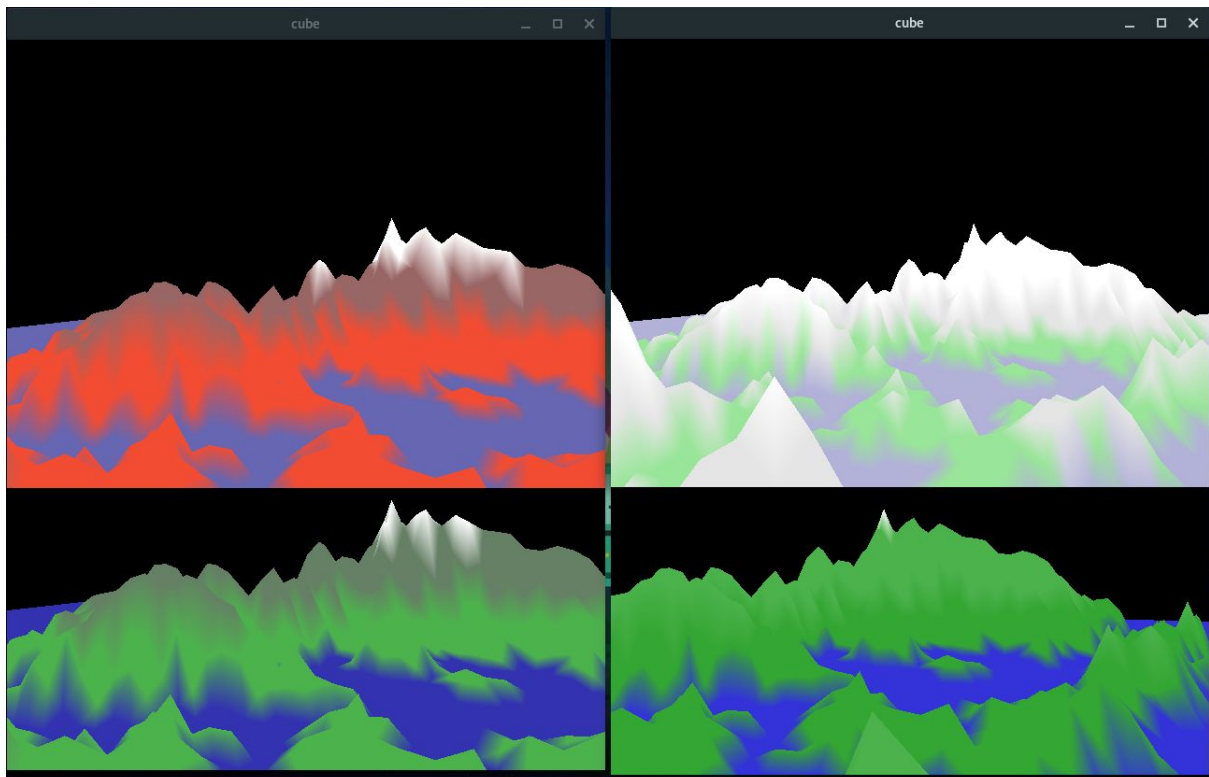
    loadShader(shaderDice, ":/vshader.glsl", ":/fshader.glsl");
    loadShader(shaderTerrain, ":/vshader_color.glsl", ":/fshader_color.glsl");

    loadShader(shaderTerrainWinter, ":/vshader_winter.glsl", ":/fshader_color.glsl");
    loadShader(shaderTerrainSpring, ":/vshader_spring.glsl", ":/fshader_color.glsl");
    loadShader(shaderTerrainSummer, ":/vshader_summer.glsl", ":/fshader_color.glsl");
    loadShader(shaderTerrainAutumn, ":/vshader_autumn.glsl", ":/fshader_color.glsl");

    GameScene::getInstance()->setDefaultShader(&shaderDice);
}

```

Cela nous donne le rendu suivant (dans l'ordre : automne / hiver / printemps / été) :



## Intégration d'un QuadTree

-> Non réalisé, au profit du Graph de scène.

## Gestionnaire de scène

Concernant le graph de scène, nous avons structuré notre moteur afin de rendre la création, modification et manipulation des objets 3D la plus simple possible. Nous nous sommes notamment inspiré de ce que fait Unity3D.

Le principe est simple. Tous les éléments de la scène sont des instances de GameObject, ou de classes en descendant. Cette classe contient tout ce qui est commun à ces différents objets du graph de scène : changements de repères, calcul des positions / rotations / tailles, modification de l'arbre du graphe de scène (ajout / suppression d'enfants,...), déplacements dans le bon repère,...

L'arbre du graph de scène est implémenté par l'intermédiaire d'un Pattern Composite : chaque GameObject a un parent, et une liste d'enfants, de type GameObject eux aussi.

La classe GameScene représente la scène / caméra du jeu, et est la racine du graph. Elle hérite de GameObject, mais est un peu particulière : puisque nous n'avons besoin que d'une seule scène dans un jeu, et que nous avons besoin d'y accéder à divers endroits du code, nous lui avons joint un Pattern Singleton. (à noter que nous avons un peu "bidouillé" celui-ci pour les besoins très précis de ce TP, qui nécessite l'affichage de 4 scènes différentes en même temps -> il est possible d'accéder à 4 instances de la classe avec la méthode de classe `GameScene::getInstance(idScene)`).

Une autre spécificité de cette classe GameScene est que nous avons redéfini ses méthodes de calcul de la matrice de transformation, afin de simuler l'utilisation d'une caméra. En effet, en OpenGL, c'est la scène qui bouge, et non la caméra, mais il est plus intuitif de faire "comme si" on bougeait une caméra -> on inverse juste la translation et la rotation par rapport au `getProjection()` du GameObject "normal".

La classe Model3D, héritant elle-aussi de GameObject, n'a pas vocation à être instanciée directement, mais à être dérivée. Elle représente un GameObject affichable, et ajoute notamment des méthodes pour gérer le shader et des variables utilisées par OpenGL (par exemple les "buffers").

Les différentes classes d'objets 3D (Cube, Plane, Terrain,...) héritent de cette classe Model3D. Elles ont à redéfinir 2 méthodes :

- \* `createGeometry()`, qui crée les sommets nécessaires et spécifiques à la forme.
- \* `draw()`, qui s'occupe d'afficher l'objet, à la bonne position / rotation / scale, dans le bon repère en prenant en compte la hiérarchie / l'arbre du graphe de scène.

Autrement dit, pour gérer un nouveau type d'objet 3D, il suffit de dériver Model3D, et d'adapter le contenu de ces 2 méthodes.

Grâce à cette architecture du code et à l'arbre du graph de scène, un appel à la méthode `draw()` de l'instance de `GameScene` appelle les méthodes `draw()` de tous les objets de la scène (pareil pour `createGeometry()`) :

```
void MainWidget::paintGL()
{
    GameScene::setCurrentNumInstance(idScene);

    // Clear color and depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    texture->bind();

    // à chaque rafraîchissement :
    // on parse l'arbre du graphe de scène pour appeler les draw() de tous les objets (GameObject) à afficher

    GameScene::getInstance()->update(); // pour les composants, EN COURS DE DEV
    GameScene::getInstance()->draw();
}
```

Il est également très facile d'ajouter de nouveaux objets 3D à la scène. Il suffit d'instancier la classe voulue (`Cube`, `Plane`, etc), de configurer cette instance, et de l'ajouter au gestionnaire de scène, soit à la racine, soit en tant qu'enfant d'un autre objet 3D :

```
// CREATION DU GRAPHE DE SCENE ET DES OBJETS 3D

Cube *cube = new Cube(QVector3D(-0.5, 5, -5.), QQuaternion::fromEulerAngles(0, 20, 0), QVector3D(1.5, 2.0, 1.0));
Cube *cube2 = new Cube(QVector3D(0.5, 5, -5));
Cube *cube3 = new Cube(QVector3D(2, 2, 0), QQuaternion(), QVector3D(1, 1, 1));

GameScene::getInstance()->addChild(cube);
GameScene::getInstance()->addChild(cube2);
cube2->addChild(cube3);

cube2->addComponent(new MovingCubeComponent());

terrain = new Terrain(":/island_heightmap.png",
    128, QVector3D(0, -3, 0), QQuaternion::fromEulerAngles(-90, 0, 0), QVector3D(2, 2, 2));
terrain->setShader(&shaderTerrain);
GameScene::getInstance()->addChild(terrain);

GameScene::getInstance()->createGeometry();

GameScene::getInstance()->setLocalRotation(QQuaternion::fromEulerAngles(0, 0, 0));
```

*Nous avons commenté les fichiers .h de chaque classe pour que tout cela soit clair en lisant le code.*

## Gestionnaire de niveaux de détails (LOD)

Nous avons choisi de charger des modèles 3D au format `.obj`, car un des formats les plus simples.



Là encore, le but est d'utiliser notre architecture pour simplifier les choses : un tel objet 3D dérive de Model3D, et redéfinit les méthodes createGeometry() et draw().

Dans l'instance, nous chargeons plusieurs .obj d'un même objet, avec des niveaux de détails différents. Un paramètre de la classe permet de définir le niveau de détails actuel (soit : le .obj à afficher à l'instant T dans la scène).

Celui-ci est choisi en fonction de la distance par rapport à la caméra. Le tout est de faire en sorte que la transition soit invisible pour l'utilisateur : passer d'un modèle à un autre plus détaillé ni trop loin (pour optimiser les performances), ni trop près (pour que cette transition se fasse discrètement).

Suite à un souci au niveau du chargement du fichier .obj, et par manque de temps, nous n'avons pas pu implémenter tout cela correctement, et l'affichage bug (mais l'implémentation prévue est celle expliquée ci-dessus) :



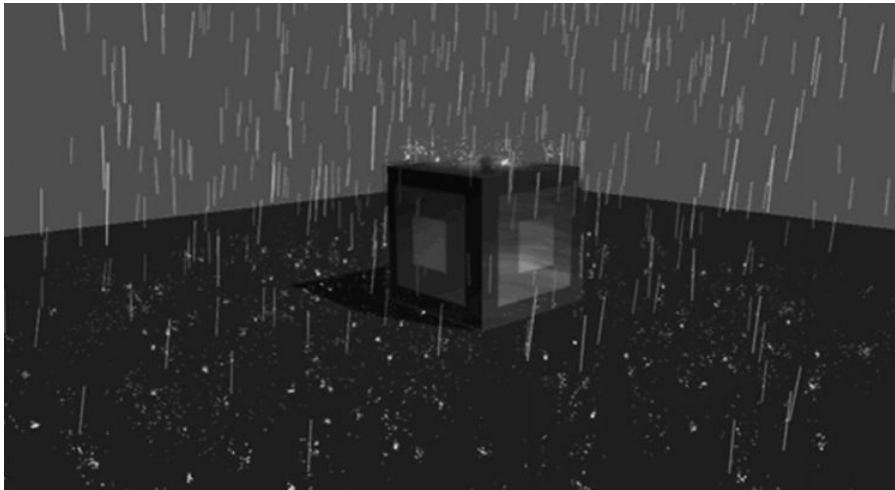
## Questions Bonus

### *Saisons*

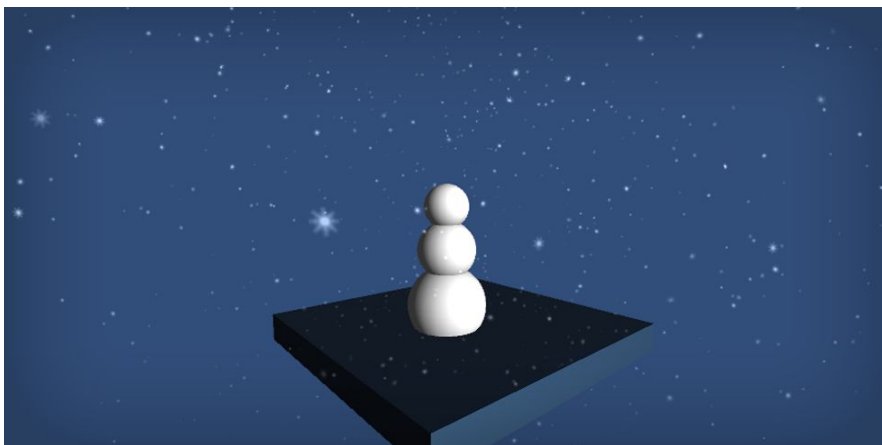
Les moteurs de particules sont souvent utilisés pour simuler simplement des effets spéciaux comme des liquides, du feu, un laser, de la neige, une explosion, etc.

Le principe est simple : on génère aléatoirement des particules associées à un ou plusieurs sprites, avec des tailles / colorations / transparences / orientations / vitesses / vitesses de rotation / etc qui varient. Le sprite choisi, la distribution (par exemple en cercle, en ligne,...), et d'une manière générale les paramètres du moteur de particules caractérisent l'effet que l'on souhaite donner.

Exemple d'effet pluie :



Exemple d'effet neige :



Il serait donc possible d'améliorer nos saisons en implémentant un moteur de particules simple : des flocons pour l'hiver, de la pluie en automne,...

En adjoignant aux particules des colliders, il serait également envisageable de créer un effet d'entassement : par exemple, lorsqu'un certain nombre de particules de neige ont touché un triangle du terrain en un temps limité, il devient blanc et sa position en hauteur augmente légèrement.

### *Niveaux de détails*

On peut structurer l'objet en quadrees ou octrees. Selon la distance de la caméra au modèle, on ne parcourt qu'un certain nombre d'étages de ces \*trees, pour adapter le niveau de détails. Ainsi, plus la caméra est loin, moins il y a de triangles à afficher, et plus le parcours du modèle est rapide.



### *Afficher une scène infinie*

On pourrait utiliser par exemples des Tiles chargées dynamiquement (ou calculées procéduralement) pour afficher une scène infinie, lorsque le joueur est suffisamment proche de la zone adéquate.

Sinon, pour faire illusion “de loin”, il serait possible de mixer des structures de données comme des quadtree ou octree (cf question précédente), avec des imposteurs 2D pour simuler des objets 3D lointains.

### *Garder un rendu temps réel*

Différentes méthodes peuvent être utilisées et combinées pour optimiser les performances du rendu de la scène.

Notamment :

- L'utilisation de structures adaptées aux fonctionnalités de calcul intégrées aux cartes graphiques (par exemple SOA pour calculs vectoriels, etc).
- Le partitionnement de la scène et des objets afin d'en optimiser le parcours, selon les besoins du projet : utilisation de BSP, quadtree,...
- Paralléliser le code : partager les calculs entre différents coeurs / threads, par exemple avec l'utilisation d'une bibliothèque comme OpenMP.
- Pratiquer des optimisations spatiales et géométriques, pour n'afficher que ce qui est nécessaire à un instant T : “technique de l'imposteur”, backface culling, frustrum culling, occlusion culling, portals,...
- Calculer (ou paramétrer) le maximum de choses en “hors ligne”, c'est à dire à la compilation et non à l'exécution : lumières, navigation,...