Compte rendu TP 3

Objectif : Introduction sur le concept de particule dans les moteurs de jeux.

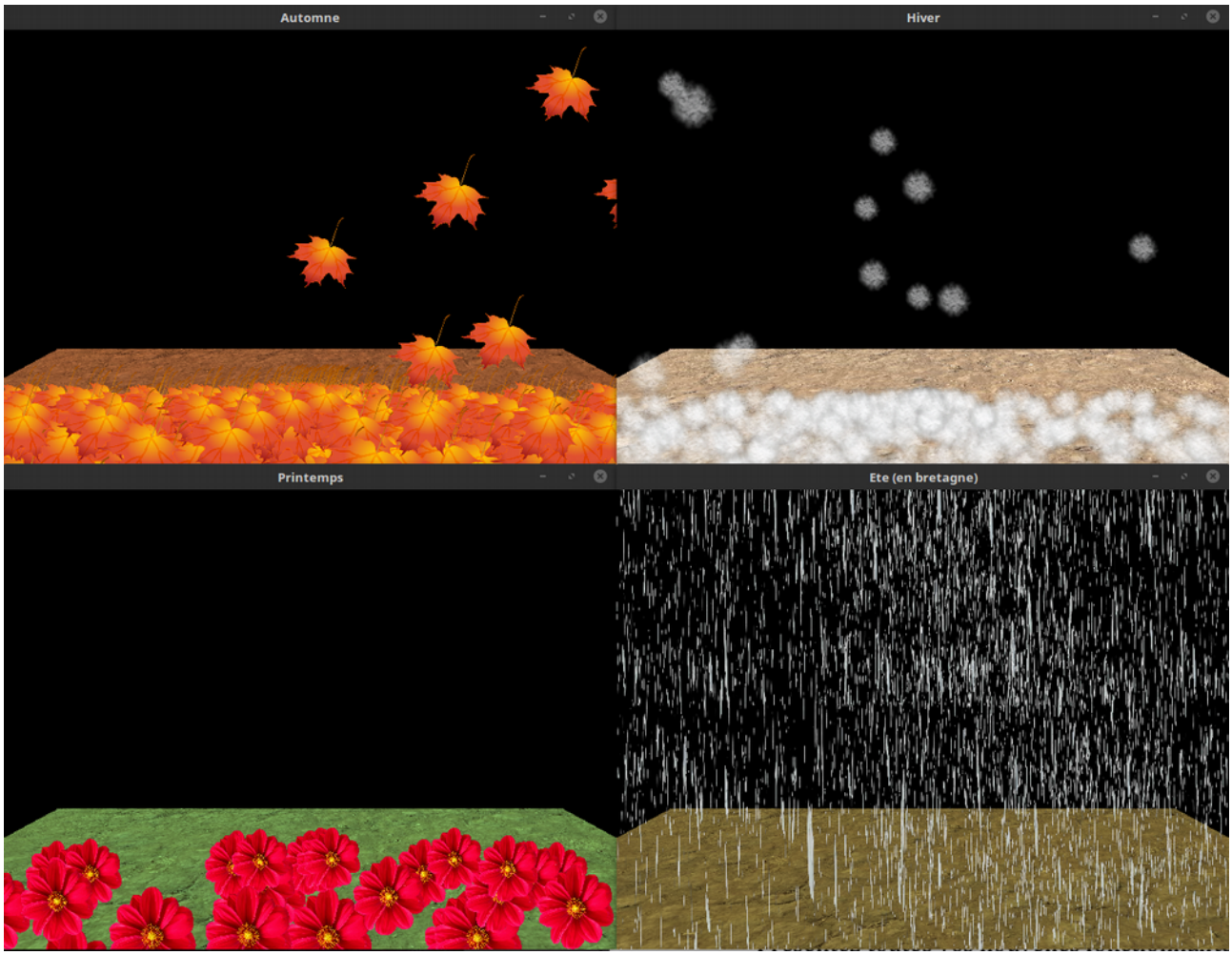


Table des matières

Nouvelles fonctionnalités : Les particules et couleur de terrain

Les particules sont des entités pouvant se déplacer dans un environnement (ici, 3D) et sont souvent liées à un moteur physique. Ce sont des éléments graphiques très intéressant pour simuler un effet de pluie, nuée d'insectes ou encore de la poussière.

Nous allons voir comment introduire ces particules dans notre moteur.

1 – La classe ModelParticle

La classe ModelParticle permet de décrire tous les éléments d'une particule. Nous avons donc des informations sur la taille, la masse, la texture, la durée de vie ou encore savoir si cette particule doit être détruite au contact du sol.

Cette classe sera donc attachée aux particules pour décrire leur type, et le fait de décomposer de la sorte les valeurs descriptives (comme ci-dessus) avec des valeurs dynamiques (comme la position) va permettre d'alléger la mémoire et de pouvoir instancier un nombre plus important de particules.

```
class ModelParticle
{
public:
    explicit ModelParticle(QOpenGLTexture *texture, float life = 10.f, float
mass = 0.05f, float ground = -0.f,
                           QVector3D size = QVector3D(0.5f, 0.5f, 0), bool
dieOnGround = false);
    virtual ~ModelParticle();
    void setTexture(QOpenGLTexture *texture);
    QOpenGLTexture* getTexture() const;
    void setMass(float mass);
    float getMass() const;
    void setGround(float ground);
    float getGround() const;
    void setLife(float life);
    float getLife() const;
    void setSize(const QVector3D &size);
    const QVector3D &getSize() const;
    void setDieOnGround(bool dieOnGround);
    bool isDieOnGround() const;
    static const float MASS;
private:
    QOpenGLTexture *m_texture;
    float m_mass, m_ground, m_life; // Masse en gramme
    QVector3D m_size;
    bool m_dieOnGround;
};
```

2 – La classe Particle

C’est la classe qui représentera une particule visible à l’écran.

Celle-ci prendra donc en paramètre le type de particule (ModelParticle) et la position sa position de départ.

La particule dispose aussi des informations sur son état de santé, le temps qui lui reste à vivre (*m_actualLife*) et si celle-ci est toujours en vie (*m_alive*).

La vie de départ des particules n’est pas forcement égale à la vie indiquée dans son type, mais varie entre 0 et 2 x la vie indiquée. Cela permet d’avoir des changements et de voir les particules détruites en même temps.

La méthode *update* permet de mettre à jour la position et vie restante de la particule suivant un delta de temps en millisecond.

Enfin, la méthode *draw* va dessiner la particule. Il faut savoir que la particule utilise un plan (GeometryEngine) pour être dessinée.

```
class Particle
{
public:
    Particle(const ModelParticle& model, QVector3D position = QVector3D());
    Particle(const Particle& p);
    ~Particle();
    Particle& operator=(const Particle& p);
    bool isAlive() const;
    void setActualLife(float actualLife);
    float getActualLife() const;
    void setActualZigZag(const QVector3D &actualZigZag);
    const QVector3D &getActualZigZag() const;
    void setPosition(const QVector3D &pos);
    const QVector3D &getPosition() const;
    void setModel(const ModelParticle &model);
    const ModelParticle &getModel() const;
    void update(const ParticleEmitter& emit, float delta);
    void draw(QMatrix4x4 &proj, QOpenGLShaderProgram *program);
    void resetLife();
private:
    bool m_alive;
    float m_actualLife;
    QVector3D m_actualZigZag, m_pos;
    ModelParticle m_model;
    GeometryEngine m_plan;
};
```

3 – La classe ParticleEmitter

C'est la classe génératrice des particules. Celle-ci prendra donc en paramètre le nombre de particules à générer, le type des particules, la position de la génératrice, l'altitude où seront générées les particules et enfin le rayon où les particules apparaîtront.

Il est possible de pouvoir changer dynamiquement le nombre de particules à générer, la hauteur d'apparition, le rayon mais aussi le type des particules.

On peut noter la présence des méthodes *update*(delta) et *draw* qui permettent d'actualiser chaque particule pour l'une, et les dessiner pour l'autre. À noter que la méthode *update* utilise une parallélisation d'une boucle ***for*** via **OpenMP**.

```
#pragma omp parallel for private(m_particles)
```

Finallement, la méthode *resetAllParticles* va remettre les particules dans un état de « départ ».

```
class ParticleEmitter
{
public:
    ParticleEmitter(unsigned int nbParticles, const ModelParticle &model,
QVector3D position = QVector3D(),
                    float height = 0.f, float radius = 1.f);
    ~ParticleEmitter();
    void setNbParticles(unsigned int nbParticles);
    unsigned int getNbParticles() const;
    void setHeight(float height);
    float getHeight() const;
    void setRadius(float radius);
    float getRadius() const;
    void setModel(const ModelParticle &model);
    const ModelParticle &getModel() const;
    void setPostion(const QVector3D &postion);
    const QVector3D &getPostion() const;
    void update(float delta);
    void draw(QMatrix4x4 &proj, QOpenGLShaderProgram *program);
    void resetAllParticles();
private:
    unsigned int m_nbParticles;
    float m_height, m_radius;
    ModelParticle m_model;
    QVector3D m_postion;
    std::vector<Particle> m_particles;
    void resetParticlePosition(Particle &particule);
};
```

4 – Couleur de terrain

Pour chaque saison correspond une couleur de terrain. Je voulais garder la possibilité d'afficher une texture sur le terrain, mais de pouvoir changer son thème (couleur).

J'ai dû pour cela modifier les shaders pour y ajouter une variable de couleurs (vec4) qui prendre des valeurs de 0 à 1,0. Cette valeur sera multipliée avec la texture pour obtenir le thème voulu.

Vshader.glsl

```
uniform mat4 mvp_matrix;
uniform vec4 a_color;
attribute vec4 a_position;
attribute vec2 a_texcoord;
varying vec2 v_texcoord;
varying vec4 v_color;
void main()
{
    // Calculate vertex position in screen space
    gl_Position = mvp_matrix * a_position;
    // Pass texture coordinate to fragment shader
    // Value will be automatically interpolated to fragments inside polygon
faces
    v_texcoord = a_texcoord;
    v_color = a_color;
}
```

Fshader.glsl

```
uniform sampler2D texture;
varying vec2 v_texcoord;
varying vec4 v_color;
void main()
{
    // Set fragment color from texture
    gl_FragColor = texture2D(texture, v_texcoord) * v_color;
}
```

Passer d’une saison à l’autre

Pour réaliser cet effet, je vais utiliser un Qtimer qui à chaque *timeout* de 10s, va envoyer un signal aux fenêtres, qui sera récupéré dans le slot *nextSaison*.

Main.cpp

```
MainWidget* win_hiver     = new MainWidget(nullptr, 30, WINTER);
win_hiver->show();


QTimer* saisonTimer = new QTimer;
QObject::connect(saisonTimer, SIGNAL(timeout()), win_hiver, SLOT(nextSaison()));


saisonTimer->start(10*1000);
```

MainWidget (.h et .cpp)

```
public slots:
    void nextSaison();

void MainWidget::nextSaison()
{
    m_actualSaison++;
    m_actualSaison %= 4;
    setSaison(m_actualSaison);
}
```

Le slot *nextSaison* va juste changer la valeur de la saison actuelle *m_actualSaison*, et s’assurer que la valeur reste entre 0 et 3, qui représente chaque saison.

La méthode *setSaison* va modifier la génératrice de particule ainsi que la couleur du terrain pour correspondre à la saison choisie :

```
void MainWidget::setSaison(int saison)
{
    switch(saison)
    {
        case WINTER:
            m_ep->setNbParticles(1000);
            m_ep->setModel(m_mpWinter);
            m_ep->setHeight(2.5f);
            m_ep->setRadius(4.f);
            groundColor = QVector4D(1.3f,1.3f,1.3f,1.f);
            setWindowTitle("Hiver");
            break;
        case SPRING:
            m_ep->setNbParticles(50);
            m_ep->setModel(m_mpSpring);
            m_ep->setHeight(0);
            m_ep->setRadius(4.f);
            m_ep->resetAllParticles();
            groundColor = QVector4D(0.7f,1.1f,0.7f,1.f);
            setWindowTitle("Printemps");
            break;
        case SUMMER:
            m_ep->setNbParticles(100);
            m_ep->setModel(m_mpSummer);
            m_ep->setHeight(2.5f);
            m_ep->setRadius(3.f);
            m_ep->resetAllParticles();
            groundColor = QVector4D(0.8f,0.8f,0.5f,1.f);
            setWindowTitle("Ete (en bretagne)");
            break;
        case AUTOMN:
            m_ep->setNbParticles(500);
            m_ep->setModel(m_mpAutomn);
            m_ep->setHeight(2.5f);
            m_ep->setRadius(3.f);
            m_ep->resetAllParticles();
            groundColor = QVector4D(1.0f,0.7f,0.5f,1.f);
            setWindowTitle("Automne");
            break;
    }
}
```

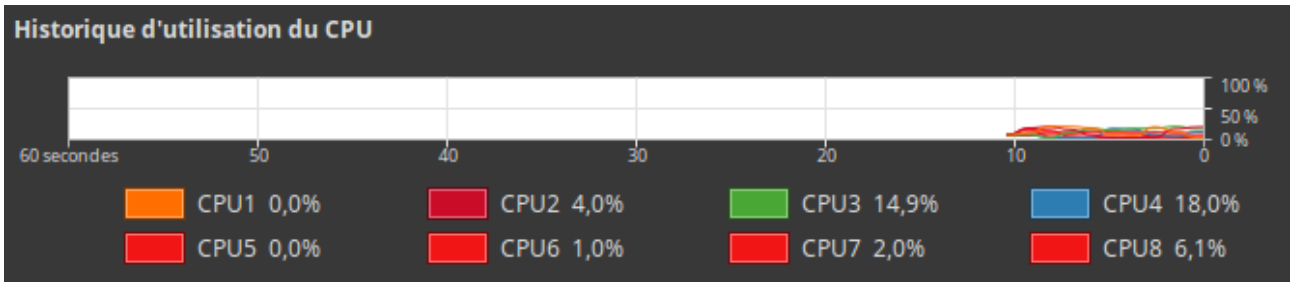
Ainsi, a chaque *timeout*, on obtient une nouvelle saison grâce aux nouveaux paramètres.

Bonus

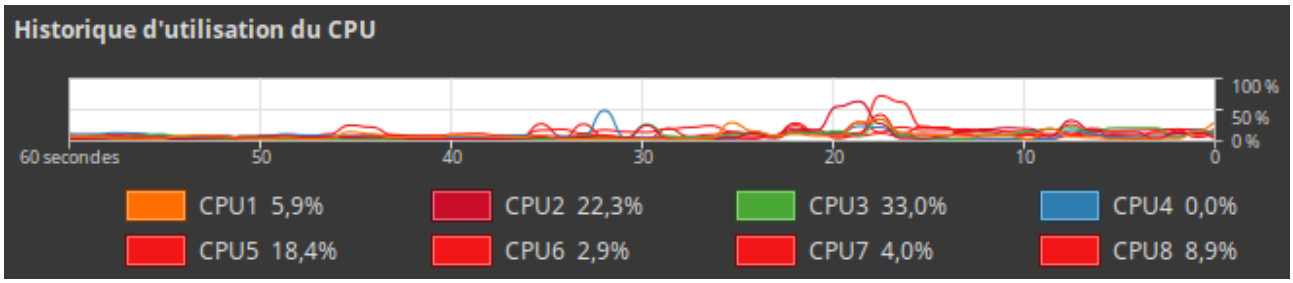
Pour accumuler les particules, j'ai tout simplement indiqué aux particules une hauteur à ne pas dépasser. Lorsque cette hauteur est atteinte, la particule ne bougera plus. Ainsi, on observe une accumulation des particules, visible en particulier pour l'automne et l'hiver.

Un autre approche serait d'ajouter un véritable moteur physique pour avoir un mouvement de particule plus réaliste, et pouvoir faire pour la pluie, un début de rivière.

Pour ce qui est de la parallélisation, voici les résultats obtenus :



Sans OpenMP



Avec OpenMP

On peut observer qu'il y a une meilleure répartissions sur les différents processeurs.

Difficultés

J'ai rencontré quelques difficultés pour insérer les particules sur le terrain correctement, tout faisant face à la caméra (billboard). J'ai donc indiqué des valeurs en dures pour simuler cet effet, mais lorsque on voudra changer la rotation, translation, …, la scène ne sera plus valide.