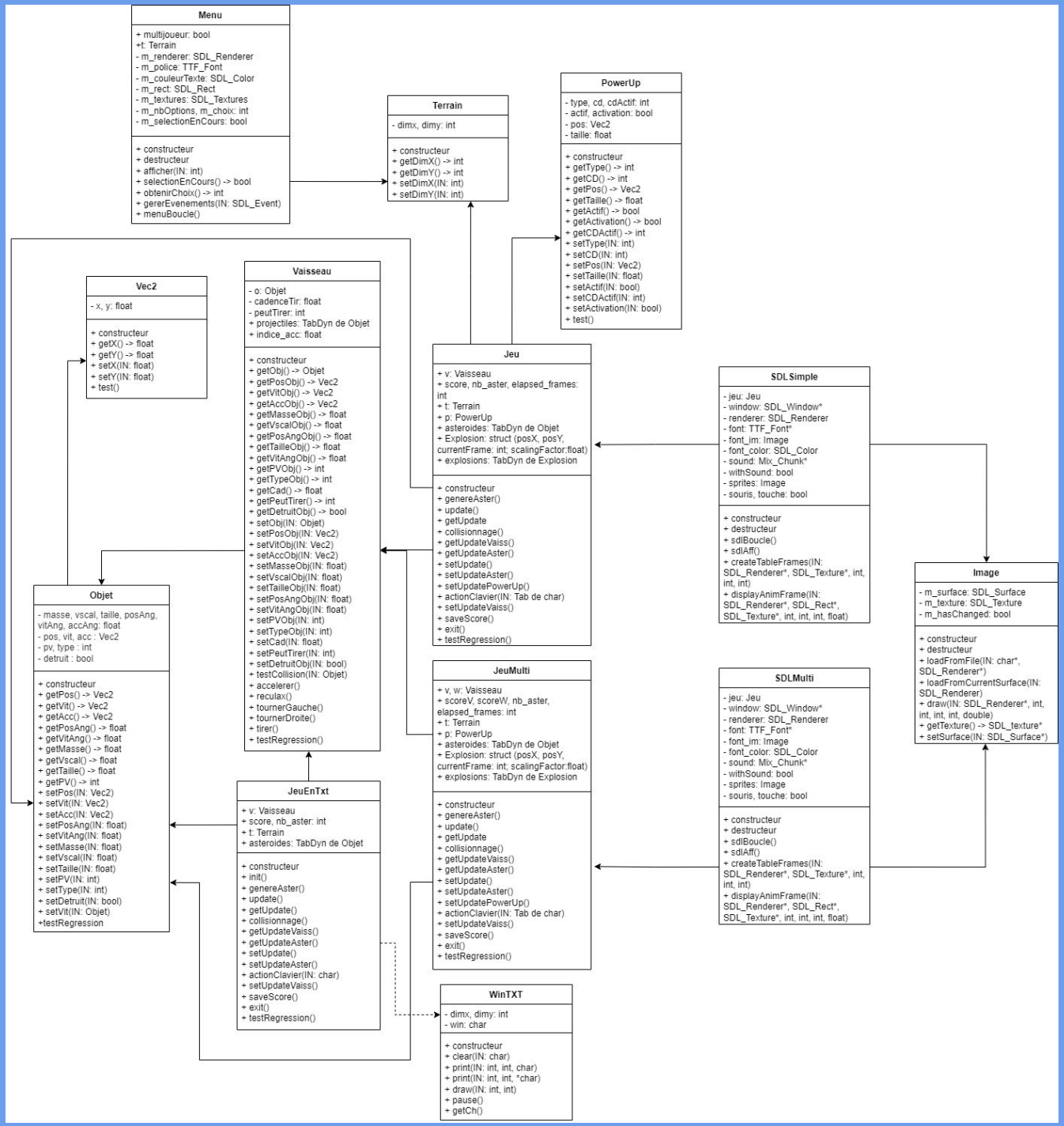
LIFAPCD - Staresiod

Ugo POUPON 12020982
Joey DAVID 12115771

C’est quoi ?

Vue d'ensemble du code - diagramme des classes UML



Les bases : Vec2

```
class Vec2 {
    private:
        float x,y;
    public:
```

→ **Constructeurs**

→ **Accesseurs**

→ **Mutateurs**

→ **Opérateurs**

→ **test de régression…**

nécessaire pour un jeu en deux dimensions: position, vitesse, accélération…

Objet

```
class Objet {
    private:
        Vec2 pos, vit, acc;
        float masse, vscal, taille, posAng, vitAng, accAng;
        int pv, type;
        bool detruit;
    public:
```

→ **Constructeur**

→ **Accesseurs, mutateurs pour chaque élément de Objet.**

→ **testCollision(Objet &ObjEtr) modifie les PV des deux Objets concernés s'il y a collision.**

→ **test de régression…**

Notre jeu est composé d'un vaisseau (qui hérite d'Objet), d'objets et d'un terrain : et c'est tout.

Vaisseau

```
class Vaisseau{
        private:
                Objet o;
                float cadenceTir;
                int peutTirer;
        public:
                vector<Objet> projectiles;
                float indice_acc;
```

Vaisseau est un Objet avec 4 attributs supplémentaires :

- une cadence de tir
- une capacité à tirer
- un tableau dynamique d'Objets (projectiles)
- un indice d'accélération

Pour les accesseurs et mutateurs, on a donc ceux de Objet (écrits *Set/GetAttObj*), et ceux propres au vaisseau (écrits *Set/GetAtt*).

Vaisseau est aussi muni de fonctions de déplacement et d'une fonction de tir (ainsi que du classique test de régression).

```
void Vaisseau::testCollision(Objet& ObjEtr) {···

void Vaisseau::accelerer() {···

void Vaisseau::reculax() {···

void Vaisseau::tournerGauche() {···

void Vaisseau::tournerDroite() {···

void Vaisseau::tirer() {···
```

On ne fait appel aux fonctions de déplacement que lors d'un input du joueur - pour les modifications faites "automatiquement", on utilise les mutateurs et accesseurs de pos, vit et acc.

Jeu

```
class Jeu {
    public:
        Vaisseau v;
        int score;
        Terrain t;
        int nb_aster;
        PowerUp p;
        vector<Objet> asteroides;
        Jeu();
        float vmax;
        int elapsed_frames;
        struct Explosion {
            int posX;
            int posY;
            int currentFrame = 0;
            float scalingFactor = 1.0;
        };
        vector<Explosion> explosions;};
```

Principe de Jeu :
Jeu est le “haut de la pyramide”, où toutes les classes codées précédemment sont mises en application.
La classe **Jeu** est donc munie de fonctions de génération, de mise à jour et d’action (via les input clavier par les futures classes SDL).

Quelques exemples :

→ **genereAster()**

Génere un astéroïde sur les bords de l’écran, en lui donnant une vitesse non nulle et une direction de manière à ce qu’il se dirige vers l’intérieur de l’écran.

```
do {
    if(rand()%2==0) {
        //spawn aléatoirement en haut ou en bas
        x = rand()%2 * (t.getDimX()+s2) - s;
        y = rand()%(t.getDimY()+s2) - s;
    }
    else {
        //spawn aléatoirement à gauche ou à droite
        x = rand()%(t.getDimX()+s2) - s;
        y = rand()%2 * (t.getDimY()+s2) - s;
```

```
        if (distance < aster.getTaille() + asteroides[i].getTaille()) {
            superposition = true;
        }
    }
} while(superposition);
```

Par la suite, genereAster() est appelé dans les fonctions de mise à jour jusqu’à ce que le nombre d’astéroïdes soit satisfaisant.

```
if ((int)asteroides.size()<nb_aster) {
    while((int)asteroides.size()<nb_aster) {
        genereAster();
```

→ **collisionnage()**

collisionage() a un fonctionnement assez simple grâce à la fonction membre Objet testCollision(). On parcourt simplement les Objets actuellement dans le jeu à l’aide de double boucles, et on leur applique testCollision(). **(i > j pour éviter les doubles tests).**

Jeu

```
class Jeu {
    public:
        Vaisseau v;
        int score;
        Terrain t;
        int nb_aster;
        PowerUp p;
        vector<Objet> asteroides;
        Jeu();
        float vmax;
        int elapsed_frames;
        struct Explosion {
            int posX;
            int posY;
            int currentFrame = 0;
            float scalingFactor = 1.0;
        };
        vector<Explosion> explosions;};
```

Principe de Jeu :
Jeu est le “haut de la pyramide”, où toutes les classes codées précédemment sont mises en application.
La classe **Jeu** est donc munie de fonctions de génération, de mise à jour et d’action (via les input clavier par les futures classes SDL).

Quelques exemples :

→ **Power-Up :**

La classe comporte seulement les accesseurs, mutateurs et la fonction de test en public:

```
class PowerUp {
    private:
        int type; /**< Le type de power-up. Les types possibles sont :
        - 1 : destruction des astéroides
        - 2 : invulnérabilité de la vaisseau
        - 3 : augmentation de la vitesse du vaisseau
        - 4 : invulnérabilité des projectiles */
        int cd; /**< Le temps de recharge du power-up en secondes. */
        Vec2 pos; /**< La position du power-up dans l'espace de jeu. */
        float taille; /**< La taille du power-up. */
        bool actif; /**< Un booléen indiquant si le power-up est actif ou non. */
        int cdActif; /**< Le temps de durée d'activation du power-up en secondes. */
        bool activation; /**< Un booléen indiquant si le power-up est en cours d'activation. */

    public:
```

Tout se fait dans Jeu:

```
if (p.getCD()>0) {
    p.setCD(p.getCD()-1);
}
else {
    p.setPos(Vec2(rand()%(t.getDimX()-1),rand()%(t.getDimY()-1)));
    p.setType(rand()%4 + 1);
    p.setCD(600);
    p.setActif(false);
    p.setActivation(true);
}
if(p.getCDActif()<0) {
    if (v.getPVObj()>0)
        v.setPVObj(1);
    for(int i = 0; i < (int)v.projectiles.size(); i++){
        if (v.projectiles[i].getPV()>0)
            v.projectiles[i].setPV(1);
    }
    v.setVscalObj(1.0);
}
```

```
p.setCDActif(p.getCDActif()-1);
```

Jeu - fonctions de mise à jour.

Deux types d'update : **getUpdate()** et **setUpdate()**.

Les **getUpdate()** sont simplement chargés de détruire les objets concernés si nécessaire, et de créer une explosion, qui sera gérée par les classes SDL.

```
void Jeu::getUpdateAster() {
    for(int i = 0; i<(int)asteroides.size(); i++){
        if (asteroides[i].getPV() <= 0){
            Explosion newExplosion = {...};
            explosions.push_back(newExplosion);
            asteroides.erase(asteroides.begin()+i);
        }
        if (asteroide hors du terrain...) {
            asteroides.erase(asteroides.begin()+i);
        }
    }
```

```
void Jeu::setUpdateVaiss() {
    //on vérifie que le vaisseau ne dépasse pas une certain vitesse
    ...
    //on réduit la vitesse à la vitesse max
    ...
    v.setPosAngObj(v.getPosAngObj()+v.getVitAngObj()*0.1);
    if (abs(v.getVitAngObj()) <= 0.001) v.setVitAngObj(0);
    else v.setVitAngObj(v.getVitAngObj()*0.9);
    //update pos et vit vaisseau
    v.setPosObj(v.getPosObj()+v.getVitObj()*scaleVitPos*v.getVscalObj());
    v.setVitObj((v.getVitObj()*0.9)+v.getAccObj()*scaleAccVit);
    v.setAccObj(v.getAccObj()*0.9);

    //si le vaisseau sort de l'écran, on le ramène de l'autre côté
    if (v.getPosObj().getX()<0) {
        v.setPosObj(Vec2(t.getDimX()+v.getPosObj().getX(),v.getPosObj().getY()));
    ... //autres tests if pour chaque côté de l'écran
    }

    //update position projectiles
    for(int i = 0; i < (int)v.projectiles.size(); i++){
        v.projectiles[i].setPos(v.projectiles[i].getPos()+(v.projectiles[i].getVit())*8);
    }
    v.setPeutTirer(v.getPeutTirer()-1);

    //update comparaison avec position powerup
    float distance = sqrt(pow(v.getPosObj().getX()-p.getPos().getX(),2) +
    pow(v.getPosObj().getY()-p.getPos().getY(),2));

    if(distance <= (v.getTailleObj()+p.getTaille()) / 2 && p.getActivation()) {
        //activation du power-up
        p.setActif(true);
        p.setCDActif(240);
        p.setActivation(false);
    }
```

Une fois que les **getUpdate()** ont vérifié que l'objet concerné avait toujours le droit d'être en vie, les **setUpdate()** appliquent toutes les modifications nécessaires à l'objet.

→ Position, Vitesse, accélération.

→ Gestion des objets dépendant de l'objet (ici, projectiles du vaisseau)

→ Dans le cas du vaisseau, gestion de l'activation des power-ups.

*On aurait bien sûr pu rassembler **getUpdate()** et **setUpdate()** en une seule fonction, à chaque fois. On a préféré cette implémentation fragmentée par souci de modularité.*

les classes SDL → Trois classes : **Image**, **SDLSimple** et **SDLMulti**

→ La classe Image est à peu de choses près la même que celle du TD image. La classe **SDLMulti** est une simple adaptation de **SDLSimple** pour le mode multijoueur, on se concentrera donc sur **SDLSimple**.

```
SDL_Rect* SDLSimple::createTableFrames(SDL_Renderer* renderer, SDL_Texture* spriteSheet,
    int nbCols, int nbLines, int displayScale) {···

void SDLSimple::displayAnimFrame(SDL_Renderer* renderer, SDL_Rect* tab,
    SDL_Texture* spriteSheet, int counter, int posX, int posY, float displayScale) {···
```

→ Une des difficultés : implémenter un système d'affichage d'animations à partir de spritesheets, pour les explosions lors des destructions d'astéroïdes.

```
//affichage des explosions
for (unsigned int i = 0; i<jeu.explosions.size(); i++){
    if (jeu.explosions[i].currentFrame > 49){
        jeu.explosions.erase(jeu.explosions.begin()+i);
    }
    else {
        displayAnimFrame(renderer, table_exp, spriteSheetExp,
        jeu.explosions[i].currentFrame, int(jeu.explosions[i].posX),
        int(jeu.explosions[i].posY), int(jeu.explosions[i].scalingFactor));
        jeu.explosions[i].currentFrame += 1;
    }
}
```

En bref…

→ Ce qui a marché

Satisfaction de tout le cahier des charges, sauf pour l'IA → pas assez de temps.

→ Ce qu'on est parvenu, tant bien que mal, à faire marcher

L'affichage des animations s'annonçait compliqué, mais au final fonctionnement assez efficace et clair.

L'implémentation du score s'est avérée légèrement plus corsée qu'anticipé.

→ Ce qui n'a pas marché

Implémentation de l'IA, du coup.

→ Ce qui aurait peut-être marché avec plus de temps

L'implémentation de l'IA, conceptuellement simple

Ajout de sprites d'état pour le vaisseau (indicateur de power-up actif, etc). Ajout d'un mode croissant de difficulté.

Séparation du multijoueur en 2 modes, versus et coop.