La **programmation orientée objet** et le langage **C++**

Pablo Rauzy
rauzy@enst·fr
pablo.rauzy.name/teaching.html#epu-cpp

EISE4 @ Polytech'UPMC

22 octobre 2014 — Cours 5

Cours précédents

- Nouveautés du C++ par rapport au C.
- Programmation orientée objet, encapsulation.
- Les classes en C++.
- Surcharge des opérateurs.
- Héritage.
- Modèles.
- Exceptions.
- La STL.

SFML

- La SFML (Simple and Fast Multimedia Library) est une bibliothèque graphique conçue pour la programmation de jeux vidéo.
- Tutoriels : `http://sfml-dev.org/tutorials/1.6/`.
- Documentation : `http://sfml-dev.org/documentation/1.6/`.

- *Logiciel libre.*
- Fait pour les jeux : fenêtres, graphismes 2D, son, réseau, système.
- Multi-plateformes : GNU/Linux, Windows, Mac OS X
  (et bientôt Android et iOS).
- Multi-langages : C++ mais aussi C, Go, .NET, Java, Python, Ruby...

- Lorsqu'on programme un jeu il faut gérer :
  - l'affichage,
  - l'interaction avec l'utilisateur,
  - le temps,
  - (parfois) le réseau.
- L'organisation du code d'un jeu fait appel à beaucoup des concepts de programmation orientée objet que l'on a vu en cours.

- Dans un jeu, l'affichage est critique.
- Beaucoup de composants à gérer tout en restant fluide.
- Utilisation de la technique du *double buffer*.
- Automatique dans les bibliothèques haut niveau comme SFML.

- Si on essaye d'afficher plein d'éléments à l'écran, un scintillement peut se produire et casser la fluidité de l'affichage.
- La technique du double buffer consiste à avoir deux zones d'affichage, et d'alterner entre les deux.
- De cette manière, on dessine toujours sur la zone "cachée", et aucun artefact gênant ne peut arriver à l'écran.
- Chaque fois qu'on a fini de dessiner, on inverse les deux zones et celle qu'on vient de dessiner est affichée d'un seul coup.

Le double buffer dans la SFML

```
sf::RenderWindow game(sf::Video(300, 400), "Mon Jeu");

//...

game.Clear(sf::Color(255, 255, 255)); // efface la zone de dessin (ici en blanc)
                                      // mais pas celle qui est à l'écran

game.Draw(stuff);                     // dessine sur la zone de dessin
game.Draw(more_stuff);
//...

game.Display(); // échange des zones de dessin, affiche tout ce qu'on a
                // dessiné depuis le dernier appel à la méthode Display
```


Composants

- Il est évidemment mieux de faire le moins d'opérations d'affichage possible.
- Par exemple dans un jeu de plateau, il est possible de construire le plateau au début du niveau puis de le garder en mémoire déjà construit pour l'afficher d'un seul coup.
- Pour les autres composants du jeu, par exemple une balle en mouvement qui va systématiquement bouger, on peut les mettre dans un conteneur qui va permettre d'itérer dessus pour les afficher.
- De la même manière, il faut mieux éviter de charger plein de petites images.
- On préfère donc charger une seule grosse image qui sert pour plusieurs *sprites*.

Exemple : composants

```
std::vector<sf::Shape> shapes;

//...

shapes.push_back(sf::Shape::Rectangle(0, 0, 42, 42, sf::Color(0, 0, 0)));
shapes.push_back(sf::Shape::Line(50, 60, 100, 200, 1.0, sf::Color(0, 0, 0)));

//...

for (auto s : shapes) {
  game.Draw(s);
}
```


Exemple : sprites

```
typedef enum { STAND, LOOK_RIGHT, LOOK_LEFT } CharacterPosition;

sf::Image character_sprites;
if (!character_sprites.LoadFromFile("character.png")) {
  std::cerr << "Can't load character.png" << std::endl;
  exit(EXIT_FAILURE);
}
character_sprites.Bind();
std::map<CharacterPosition, sf::Sprite *> character;
sf::Sprite *c;
c = new sf::Sprite(character_sprites);
c->SetSubRect(sf::IntRect(0, 0, 16, 20));
character[STAND] = c;
c = new sf::Sprite(character_sprites);
c->SetSubRect(sf::IntRect(0, 16, 32, 20));
character[LOOK_RIGHT] = c;
c = new sf::Sprite(character_sprites);
c->SetSubRect(sf::IntRect(0, 32, 48, 20));
character[LOOK_LEFT] = c;

game.Draw(character[character_position]);
```

- Les entrées utilisateurs arrivent par la souris, le clavier, et/ou un joystick, dans le cas des jeux.
- La gestion de ces périphériques repose sur le principe d'évènements.
- Chaque fois que l'utilisateur est actif, un évènement est généré :
  - mouvement de la souris,
  - clic du bouton gauche / droit / milieu,
  - appui sur une touche,
  - ...
- Quand c'est applicable, l'évènement contient des informations utiles :
  - position du curseur de la souris,
  - quelle touche du clavier a été enfoncée,
  - ...

- La gestion des évènements doit se faire en continue.
- Elle est donc placée au début de la boucle principale du programme.
- La structure de la boucle principale d'un jeu va être :

```
sf::RenderWindow game(sf::Video(300, 400), "Mon Jeu");
sf::Event event;
while (game.IsOpened()) { // boucle principale
  while (game.GetEvent(event)) { // boucle d'évènement
    // gestion des évènements
  }

  // dessin des composants

  game.Display();
}
```


- La SFML fourni une gestion complète et simple des évènements.
- La méthode `sf::Window::GetEvent` renvoit `false` si il n'y a pas d'évènement en attente, sinon `true` et remplie une instance de la classe `sf::Event` qui lui en passé par référence.
- La classe `sf::Event` a un attribut `Type` qui permet de savoir à quel type d'évènement on a à faire.
- Ensuite, elle contient une union de structures contenant les informations sur l'évènement en question.

Comment peut-on faire si on veut gérer un évènement pour ctrl+Z ?


```
//...
while (game.GetEvent(event)) { // boucle d'évènement
  switch (event.Type) {
  case sf::Event::Closed: // fenêtre du jeu fermée
    game.Close();
    break;
  case sf::Event::KeyPressed: // appui sur une touche
    switch (event.Key.Code) {
    case sf::Key::Up: // touche haut
      player2.moveUp();
      break;
    case sf::Key::Z: // touche Z
      player1.moveUp();
      break;
    //...
    }
    break;
  case sf::Event::MouseButtonPressed: // appui bouton de la souris
    //...
    break;
  default: break;
  }
}
```

Exemple : gestion entrées temps-réel

```
//...
bool ctrl_key = false;
while (game.GetEvent(event)) { // boucle d'évènement
  switch (event.Type) {
  //...
  case sf::Event::KeyPressed: // appui sur une touche
    switch (event.Key.Code) {
    case sf::Key::LControl:
      ctrl_key = true;
      break;
    case sf::Key::Z: // touche Z
      if (ctrl_key) { // ctrl+Z
        // annulation de la dernière action
      }
      break;
    default: break;
    }
    break;
  case sf::Event::KeyReleased:
    switch (event.Key.Code) {
    case sf::Key::LControl
      ctrl_key = false;
      break;
    default: break;
    }
    break;
  default: break;
  }
}
```


Entrées temps-réel

- On a souvent besoin de maintenir un état des entrées, par exemple pour savoir si la touche ctrl est enfoncée au moment où l'on reçoit un évènement d'appui sur une autre touche du clavier.
- Plutôt que de devoir le faire à la main, la SFML fourni la classe `sf::Input` qui permet de faire cela.

Exemple : gestion entrées temps-réel avec `sf::Input`

```
//...
while (game.GetEvent(event)) { // boucle d'évènement
  const sf::Input &input = win_->GetInput();
  switch (event.Type) {
  //...
  case sf::Event::KeyPressed: // appui sur une touche
    switch (event.Key.Code) {
    case sf::Key::Z: // touche Z
      if (input.IsKeyDown(sf::Key::LControl)) { // ctrl+Z
        // annulation de la dernière action
      }
      break;
    default: break;
    }
    break;
  default: break;
  }
}
```


Documentation SFML

- Tutoriels : `http://sfml-dev.org/tutorials/1.6/`.
- Documentation : `http://sfml-dev.org/documentation/1.6/`.
- Regardons un peu :
  - Texte.
  - Nombres aléatoires.
  - Temps.
  - Audio.
  - Caméra.
  - Réseau.
- Puis voyons un exemple et parcourons son code.

Projets

- Seul ou en groupe de 2, ou 3 exceptionnellement (gros projets).
- Programmation d'un jeu graphique 2D.
- Rendre un premier rapport par email pour le 5 décembre.
- Rendre le code du jeu pour le 19 décembre.

Contraintes

- C++, STL, SFML (pour d'autres bibliothèques vérifier avec moi avant).
- Code propre et bien organisé.
- Utilisation du polymorphisme (fonctions virtuelles, classe abstraites...) à bon escient.
- Pas de fuites de mémoires (vérification avec `valgrind`).
- Fournir un `Makefile` avec des règles `all` et `clean`.

Rapport intermédiaire

- Rendre le rapport (txt, odt, ou pdf) par email.
- Expliquer et justifier l'architecture (prévue) de votre logiciel.
- Répartition (prévue) des tâches dans le groupe.

Rapport final

- Rendre le rapport (txt, odt, ou pdf) avec le code.
- Expliquer et justifier l'architecture de votre logiciel.
- Expliquer les difficultés rencontrées et comment vous les avez surmontées ou contournées.
- Répartition des tâches dans le groupe, et temps consacré à chaque tâches.

Idée de projet

- Quelques familles de jeu :
  - Tetris
  - Casse briques
  - Bubble shooter
  - Pacman
  - Jewel
  - Copter
  - Tanks
  - 2048
  - Snake
  - Flipper
  - Tron
  - Sokoban
  - Plateforme (Mario-like)
  - RPG (Pokemon-Like)
  - ...

C'est tout pour aujourd'hui :)