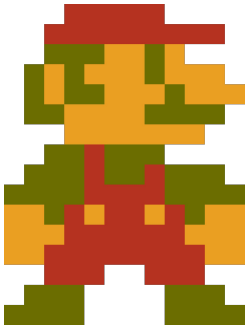
Pràctica PRO2



PART 2

Quadrimestre Primavera 2024/25

RogeR Bitlloch | *roger.bitlloch@estudiantat.upc.edu*

Part 2: Eficiència

La classe que volem hauria de tenir la següent funcionalitat:

1. Serà un **contenidor**, és a dir, contindrà objectes (del joc). Però només contindrà els punters als objectes, no els objectes en sí. Això és important perquè la classe `Game` ja té els seus contenidors (ara mateix un `vector<Platform>` per guardar les plataformes), i ja ens va bé. Només volem un altre contenidor que ens resol el problema dels rectangles.

```
#include "platform.hh"
template class Finder<Platform>;

#include "strawberry.hh"
template class Finder<Strawberry>;


template <typename T>
void Finder<T>::add(const T *t) {
    // Afegir al mapa de rectangles
```

Com estem creant un **contenidor** hem de posar lo del “template <typename T>” abans de cada funció i incloure les classes per les quals és un template en el .cc

2. Farà la suposició que els objectes que es posin al contenidor **tenen un mètode** `get_rect()`, que retorna un `Rect`. Això és perquè, independentment del tipus d'objecte que tinguem, si aquest objecte té el mètode `get_rect`, el nostre contenidor ja es podrà espabilar i classificar els rectangles d'alguna manera, sense haver de conèixer els altres detalls.

```
pro2::Rect get_rect() const {
    return {left_, top_, right_, bottom_};
}

Rect Strawberry::get_rect() const {
    return {pos_.x-Width/2, pos_.y-Height, pos_.x+Width/2, pos_.y};
}
```

Ja ho hauríeu de tenir de la part 1 però per si de cas ho heu fet d'alguna altre manera, cal implementar get_rect pel vostre col·leccionable

*Suposant que la posició es guarda a l'aresta d'abaix al centre del vostre personatge

3. Tindrà 4 mètodes: afegir, esborrar, actualitzar i consultar. Aquests mètodes permetran:

- Afegir un objecte al contenidor.
- Esborrar un objecte del contenidor.
- Actualitzar un objecte que ja estava al contenidor, que vol dir actualitzar el seu rectangle, realment.
- I finalment, el mètode més important: **consultar els objectes de dins el contenidor que estiguin dins d'un cert rectangle**. Aquest mètode el cridarem amb les coordenades de la càmera per saber quins objectes són visibles.

```
    /**
     * @brief Retorna el conjunt d'objectes amb rectangles
     *        total o parcialment dins de `rect`.
     *
     * Si el nombre de rectangles del contenidor és `n`, el
     * cost de l'algorisme ha de ser O(log n).
     *
     * @param rect El rectangle de cerca
     *
     * @returns Un conjunt de punters a objectes que tenen un
     *          rectangle parcial o totalment dins de `rect`
     */
    std::set<const T *> query(pro2::Rect rect) const;
};
```

Anem a fer-ho d'ineficient O(n) a eficient

Utilitzarem una llista i recorrarem la llista en busca dels rectangles que ens interessen per retornar únicament aquells

(així comprovem que tot vagi bé i després ja ens barallem per millorar l'eficiència)

```
template <typename T>
void Finder<T>::add(const T *t) {
    objects_.insert(t);
}

template <typename T>
void Finder<T>::remove(const T *t) {
    objects_.erase(t);
}

template <typename T>
void Finder<T>::update(const T *t) {
    remove(t);
    add(t);
}
```

```
template <typename T>
set<const T*> Finder<T>::query(Rect rect) const {
    set<const T*> result;
    typename set<const T*>::const_iterator it;
    for (it = objects_.begin(); it != objects_.end(); ++it) {
        if (rects_overlap((*it)->get_rect(), rect)) {
            result.insert(*it);
        }
    }
    return result;
}
```

A game.hh i game.cc:

Declarem a Game.hh

```
Finder<Platform> platform_finder_;
Finder<Strawberry> strawberry_finder_;
```

A game.cc omplim

```
// Añadir todas las plataformas al finder
for (const Platform& platform : platforms_) {
    platform_finder_.add(&platform);
}

// Añadir todas las fresas al finder
for (const Strawberry& strawberry : strawberrys_) {
    strawberry_finder_.add(&strawberry);
}
```

Canviem paint per usar només allò imprescindible

```cpp
void Game::paint(pro2::Window& window) {
    window.clear(sky_blue);

    // Crear un rectángulo que representa el área visible (la cámara)
    Pt camera_center = window.camera_center();
    Rect visible_area = {
        camera_center.x - window.width() / 2,  // left
        camera_center.y - window.height() / 2, // top
        camera_center.x + window.width() / 2,  // right
        camera_center.y + window.height() / 2  // bottom
    };

    // Consultar solo las plataformas visibles en el área de la cámara
    std::set<const Platform*> visible_platforms = platform_finder_.query(visible_area);
    for (const Platform* p : visible_platforms) {
        p->paint(window);
    }

    // Consultar solo las fresas visibles en el área de la cámara
    std::set<const Strawberry*> visible_strawberrys = strawberry_finder_.query(visible_area);
    for (const Strawberry* s : visible_strawberrys) {
        s->paint(window);
    }

    mario_.paint(window);
}
```

Només pintem aquelles que és troben enmig de la pantalla

Amb això us hauria d'anar ja per moltes plataformes i col·leccionables que tingueu (paint, operació més costosa)

Prova de posar 10.000 plataformes i **que funcioni correctament i sense lag abans de continuar**

```
for (int i = 1; i < 100000; i++) {
    platforms_.push_back(Platform(250 + i * 200, 400 + i * 200, 150, 161));
    strawberrys_.push_back(Strawberry({325 + i * 200, 150}));
}
```

Si tens dubtes d'alguna cosa o et falta alguna part de codi aquí tens penjada la meva implementació ineficient (la que has vist fins ara):

Codi: [PART 2.0 O(n)]

Ara anem a millorar l'eficiència de l'estructura de dades finder

- Volem evitar recorrer tots els elements que tenim
- Podem fer una **cerca (temps logN)** per començar a iterar en els elements que la seva part dreta >= esq_pantalla i acabar d'iterar en aquells que la seva part esquerra >= dreta_pantalla?
- Quines estructura de dades podem utilitzar per fer aquesta cerca?

Podem utilitzar un map, on tinguem els punts on tinguem l'esquerra dels quadrats ordenats

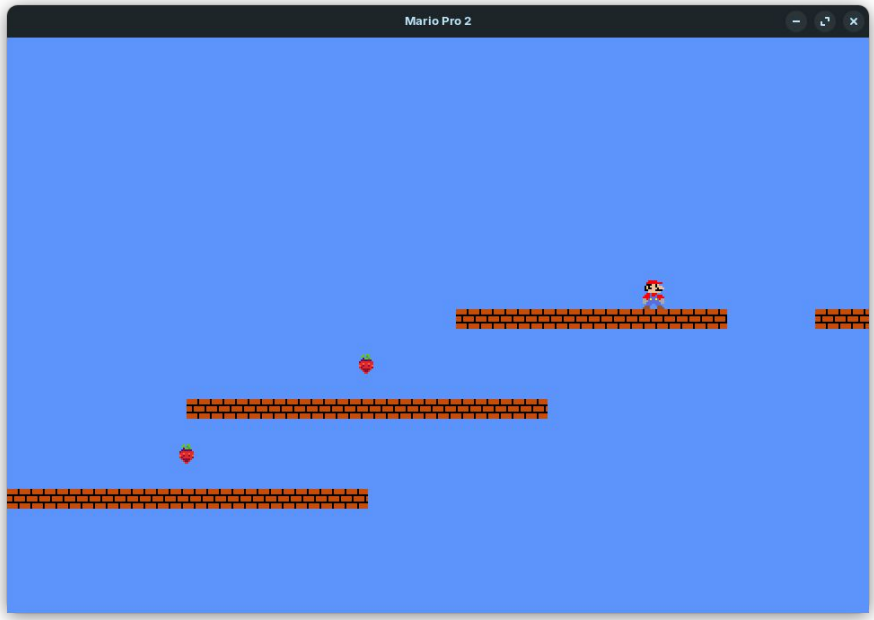
```
private:
    // Mapa que donat un objecte retorna el seu rect
    map<const T*, pro2::Rect> obj_rect_map_;

    // Mapa que donat un left retorna els objectes que comencen allà
    map<int, set<const T*>> x_start_map_;
```

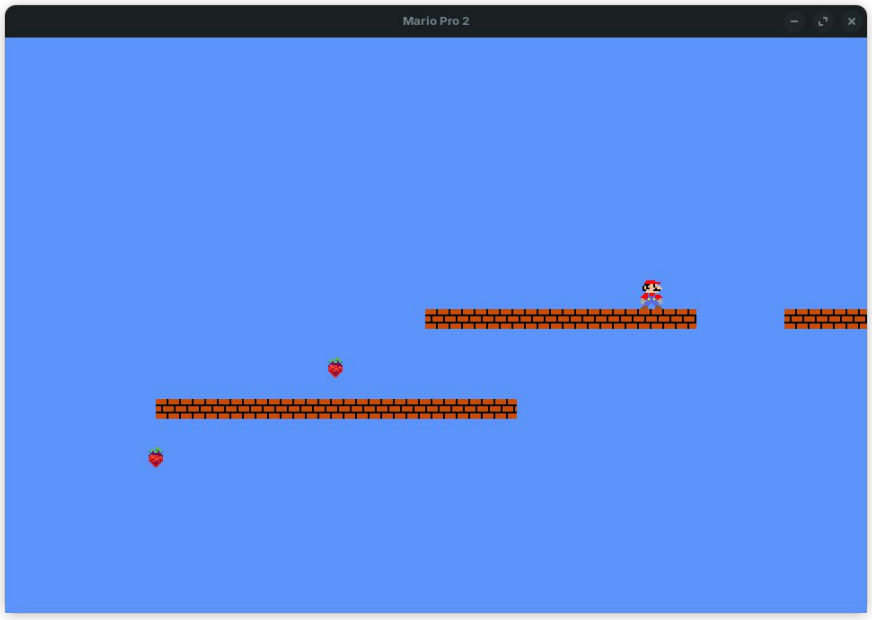
D'aquesta manera, al estar ordenat, podem fer una cerca binària per saber a on hem de començar a buscar

```
template <typename T>
set<const T*> Finder<T>::query(Rect rect) const {
    set<const T*> result;

    // Busco el primer objecte que left <= rect.right
    // Esta operación es O(log n)
    auto it = x_start_map_.lower_bound(rect.left);

    // Itero només amb aquells objectes que left > rect.right
    while (it != x_start_map_.end() && it->first <= rect.right) {
        // Per cada objecte
        for (const T* obj : it->second) {
            // Agafo el rectàngle
            const Rect& obj_rect = obj_rect_map_.at(obj);

            // Comprovo si esta dins (podria estar abaix o dalt)
            if (obj_rect.left <= rect.right &&
                obj_rect.right >= rect.left &&
                obj_rect.top <= rect.bottom &&
                obj_rect.bottom >= rect.top) {

                result.insert(obj);
            }
        }
        ++it;
    }

    return result;
```

Però tenim un problema…

Com que només comprovo la paret del rectangle de l'esquerra a la que aquesta surt de la pantalla és deixa de pintar, tot i que part de l'objecte encara hi sigui





Codi: [PART 2.1 O(log n) SENSE FUNCIONAR]

Podem concloure que…

Llavors necessitem fer una **cerca** al primer objecte que compleixi que té, almenys una aresta del rectangle dins la pantalla

- Per tant necessitem una estructura de dades on poder fer una cerca “personalitzada” (si, fer 4 maps amb les quatre direccions i recorrer els 4 és molt cutre)

- **Deures:** Buscar i provar d’implementar una estructura de dades on pugueu fer la cerca de forma eficient