Gra „Ogień i Woda” w wxWidgets

Adam Jakubcza, Jakub Janusz, Szymon Gazdowicz, Kacper Gałan

22 lutego 2023

Spis treści

1 O projekcie

Gra „Ogień i Woda”, w oryginale „Fireboy and Watergirl” to gra platformowa, w które dwójka graczy kieruje, za pomocą klawiatury, tytułowymi postaciami. Ich zadaniem jest dotarcie do drzwi przez mapę pełną przeszkód.

Ten projekt to próba implementacji tego tytułu w języku C++, przy pomocy biblioteki wxWidgets.

Projekt przedstawia „menu” z ekranem głównym gry - to tutaj można odpalić konkretną mapę, czy wyświetlić informacje na temat twórców. Wybranie gry powoduje pojawienie się nowego okna, w którym znajduje się mapa z 2 postaciami.

2 Technikalia

Żeby ułatwić pracę w grupie, projekt podzieliliśmy na 3 części - każda z nich to oddzielny *projekt* Code::Blocks. Dzięki temu, można było pracować na każdym z nich oddzielnie, w tym samym czasie bez problemów z kompatybilnością.

Gdyby każdy pracował na kopii tego samego, połączenie wszystkiego w całość wymagałoby dodatkowej pracy w postaci łączenia różnych skryptów w jeden*. Mogliśmy użyć systemu kontroli wersji *git*, ale wymagałoby to umiejętności obsługi go przez wszystkich. Dodatkowo Code::Blocks domyślnie nie wspiera tej funkcjonalności.

.* W optymistycznym scenariuszu, gdzie wszystko nam się udawało, map było kilka, wtedy taki podział byłby jeszcze bardziej pomocny – więcej kodu nie wymagałoby większej ilości pracy nad łączeniem projektu

Podział umożliwia też korzystanie z wxSmith, tj. rozszerzenia Code::Blocksa, w 3 oddzielnych przypadkach. Ekran główny korzysta z zupełnie innych elementów graficznych niż sama gra, więc było to jak najbardziej pomocne, bo nie trzeba było umieszczać wszystkiego w jednej ramce.

Podział jest następujący:

1. `menu` – wyświetlanie informacji i otwieranie ramki `mapka`;
2. `mapka` – ramka z przeszkodami i rozgrywką, dziedziczy wszystkie funkcje z ramki `fizyka`;
3. `fizyka` – poligon doświadczalny do testowania, zawiera system poruszania się.

Poniższe opisy będą w odwrotnej kolejności w stosunku do powyższej listy, żeby lepiej pokazać zależności między tymi modułami.

2.1 Fizyka, czyli system poruszania postaci

Poniższe mechaniki implementuje klasa `wxFizykaFrame`.

2.1.1 Klawiatura dla 2 graczy

Gra pozwala na ruch postaci w 4 kierunkach: lewo, prawo, góra, dół, które wykonuje się za pomocą strzałek i klawiszy *WSAD*:

1. W – skok postacią Woda
2. A – ruch postacią Woda w lewo
3. D – ruch postacią Woda w prawo
4. strzałka w górę – skok postacią Ogień
5. strzałka w lewo – ruch postacią Ogień w lewo
6. strzałka w prawo – ruch postacią Ogień w prawo

Aby to zrobić trzeba wykryć zdarzenie wciśnięcia przycisku, ale też jego puszczenia (zatrzymanie ruchu):

```
void fizykaFrame::OnKeyboardKeyDown(wxKeyEvent& event) {

    switch((int)event.GetKeyCode()) {

        // GRACZ 1
        case 87:
            kierunki[0].gora = true;
            break; // W

        ... // pozostałe klawisze
    }
    event.Skip();
}
```

Analogicznie jest dla puszczenia klawisza, wtedy mamy do czynienia z funkcją `OnKeyboardKeyUp`, a wartości ustalamy na `false`, a nie `true`. Obie funkcje można stworzyć w wxSmith standardową praktyką dodawania tzw. *eventu* (wydarzenia), ewentualne różnice można poprawić ręcznie.

2.1.2 Grawitacja i kolizja

Kod, który zajmuje się kolizjami jest identyczny dla obu graczy, więc zastosowana została pętla po indeksach g graczy

```
for (int g = 0; g < n_graczy; g++)
```

Domyślnie postać spada, jeśli nie ma nic pod nią. Nie może też poruszać się przez przeszkody: komputer sprawdza czy nie występuje kolizja z jakimś elementem, jeżeli taka jest, to ruch oczywiście jest uniemożliwiony a gracz „przykleja się do ściany". Kolizja występuje gdy jedna z bitmap stoi na drodze gracza (blokuje możliwość ruchu w danym kierunku).

Poniższy kod prezentuje schemat sprawdzania kolizji i ewentualne przyklejenie, ze względu na czytelność część kodu jest zastąpiona komentarzem:

```
for (int p = 0; p < n_przeszkod && !kolizja; p++) {
    // Kolizja jest prawdziwa, kiedy gracz nachodzi
    // na przeszkodę zarówno w poziomie jak i pionie.
    // Warunek kolizji (pion/poziom) to logiczne sprawdzenie,
    // czy odcinki (pion/poz.) się pokrywają:
    kolizja =
        kolizja_poziom(gracze[g], prze..y[p], przesuniecie_x) &&
        kolizja_pion(gracze[g], przeszkody[p], 0);
    if(kolizja == true && prze...y[p]->IsShown() == false){
        kolizja = false;
    }

    if(kolizja == true)
        z_jaka_kolizja = p;
}

if(kolizja && .../* gracz nie wychodzi po za mapę */) {

    if(przesuniecie_x > 0) {
        gracze[g]->Move(.../* pozycja przytulająca */);
    }

    if(przesuniecie_x < 0) {
        gracze[g]->Move(.../* pozycja przytulająca */);
    }
}
```

2.1.3 Polimorfizm: mapka-fizyka

Ramka z mapą, jak już wcześniej wspomnianno, jest pochodną klasą ramki *fizyki*. Podczas *includowania* pliku z fizyką w pliku z mapą, celem dziedziczenia, dochodzi do konfliktu związanego z *build info*. Żeby go uniknąć trzeba usunąć część kodu, który tworzy CodeBlocks przy nowym projekcie. Moduł działa bez tego:

```
wxString wxbuildinfo(wxbuildinfoformat format)
{
    wxString wxbuild(wxVERSION_STRING);

    if (format == long_f )
    {
#if defined(__WXMSW__)
        wxbuild << _T("-Windows");
#elif defined(__UNIX__)
        wxbuild << _T("-Linux");
#endif

#if wxUSE_UNICODE
        wxbuild << _T("-Unicode build");
#else
        wxbuild << _T("-ANSI build");
#endif // wxUSE_UNICODE
    }

    return wxbuild;
}
```

2.2 Mapka

Ustalanie które bitmapy to gracze, a które przeszkody:

```
// Konstruktor mapy
n_graczy = 2;
gracze = new wxStaticBitmap*[n_graczy] {
    Jaszczur, Smok // bitmapy o zmiennej
    // Jaszczur i Smok będą graczami
};

n_przeszkod = 3;
przeszkody = new wxStaticBitmap*[n_przeszkod] {
    StaticBitmap1, StaticBitmap2, StaticBitmap3
    // dane bitmapy będą kolidować na mapie
    // z graczami
};
```

Działanie interaktywnych elementów:

Na mapie którą stworzyliśmy instnieją dwa ineraktywne elemnety, są to guziki na które należy wskoczyć aby uruchomić ich działanie. Sam fakt czy gracz stoi na przycisku sprawdzają funkcje *kolizja_poziom* i *kolizja_pion*, gdy dochodzi do takiej sytacji, w zależności od guzika znika ściana blokująca resztę mapy, lub pojawia się schodek umożliwiający dojście do mety.

Poza wymienionymi elemntami które reagują na zachowanie gracza, istnieją również przeszkody przez które można przegrać, są to: lawa, woda i kwas, gdzie lawa zabija ludzika niebieskiego, woda czerwonego, natosmiast kwas należy unikać obiema postaciami.

Kod na następnej stronie opodwiada za odblokowanie przejścia do drugiego przycisku (odblokowuje niedostnępną część mapy). Gdy drugi stanie na przed chwilą wspomnianym guziku, pojawia się schodek umożwliający przejście gry.

Obsługa schodka:

```
if(kolizja_pion(gracze[0], StaticBitmap20) &&
kolizja_poziom(gracze[0], StaticBitmap20)
    || kolizja_pion(gracze[1], StaticBitmap20) &&
    kolizja_poziom(gracze[1], StaticBitmap20)){

    if(kolizja_pion(gracze[0], StaticBitmap20) &&
    kolizja_poziom(gracze[0], StaticBitmap20)){
        StaticBitmap23->Hide();
        StaticBitmap22->Hide();

        if(kierunki[0].prawo){
            gracze[0]->Move(StaticBitmap20->GetPosition().x +
            StaticBitmap20->GetSize().x, gracze[0]->GetPosition().y);
            kierunki[0].prawo = false;
        }
        if(kierunki[0].lewo){
            gracze[0]->Move(StaticBitmap20->GetPosition().x -
            gracze[0]->GetSize().x, gracze[0]->GetPosition().y);
            kierunki[0].lewo = false;
        }
    }
}

else{
    StaticBitmap23->Show();
    StaticBitmap22->Show();
}
```

Pojawianie się ukrytego schodka

```
if(kolizja_pion(gracze[0], StaticBitmap15) &&
kolizja_poziom(gracze[0], StaticBitmap15)
   || kolizja_pion(gracze[1], StaticBitmap15) &&
   kolizja_poziom(gracze[1], StaticBitmap15)){

    if(kolizja_pion(gracze[0], StaticBitmap15) &&
    kolizja_poziom(gracze[0], StaticBitmap15)){
        StaticBitmap30->Show();

        if(kierunki[0].prawo){
            gracze[0]->Move(StaticBitmap15->GetPosition().x +
                StaticBitmap15->GetSize().x, gracze[0]->GetPosition().y);
            kierunki[0].prawo = false;
        }
        if(kierunki[0].lewo){
            gracze[0]->Move(StaticBitmap15->GetPosition().x -
                gracze[0]->GetSize().x, gracze[0]->GetPosition().y);
            kierunki[0].lewo = false;
        }

    }
} else{
    StaticBitmap30->Hide();
}
```

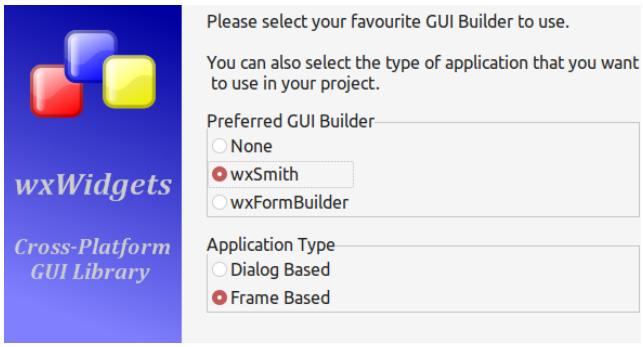
Sprawdzanie czy gracz nie wpadł do niebezpiecznego płynu:

```
    if(kolizja_pion(gracze[0], lawa, +5) && kolizja_poziom(gracze[0], lawa)){
        czy_przegrana = true;
    }

    ... // analogicznie jest dla pozostałych takich elementów
```

2.2.1 Tworzenie nowej mapy w CodeBlocks [instrukcja]

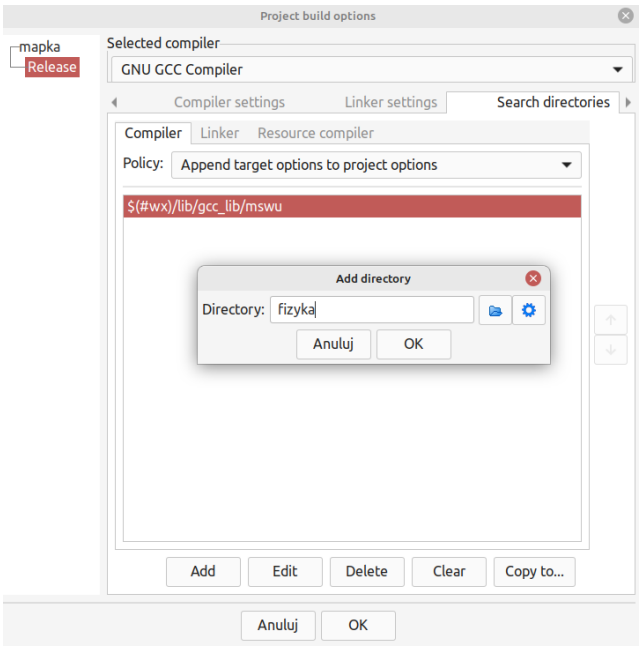
1. Stwórz nowy projekt wxWidgets w CodeBlocks, przykładowo *mapa*



2. Dodaj pliki modułu z fizyką

   - w opcjach projektu (w lewym pasku, prawy przycisk na ikonę projektu) za pomocą `Add files...` trzeba znaleźć (albo wrzucić do projektu) i dodać pliki `fizykaMain.cpp` i `fizykaMain.h`

   - w opcjach projektu w `build options` trzeba dodać lokalizację projektu z fizyką, najlepiej jak jest w katalogu podrzędnym



3. Ustal polimorfizm (tzn.: dodaj trochę kodu w klasach)

   - zmień konstruktor w `mapaMain.cpp` i użyj nowych funkcji:

```
MapaFrame::MapaFrame(wxWindow* parent,wxWindowID id)
    : fizykaFrame(parent, id) // to trzeba dopisać
{

  ... // domyślny kod - nie trzeba nic zmieniać

  // nowe funkcje:
  oczysc();

  // ustalenie graczy
  n_graczy = 2;
  gracze = new wxStaticBitmap*[n_graczy] {
    ... // elementy graczy, np: StaticImage1, StaticImage2
  };

  n_przeszkod = ...; // przeszkody
  przeszkody = new wxStaticBitmap*[n_przeszkod] {
    ... // obrazki przeszkód, np: StaticImage3, StaticImage4
  }
}
```

   - ustal polimorfizm i nadpisz funkcję, którą tworzy wxSmith w pliku `mapaMain.h`

```
#include "fizykaMain.h"
class mapkaFrame: public fizykaFrame {

  ... // właściwości klasy

  private: void Create(wxWindow*, wxWindowID, const wxChar*,
    wxPoint, wxSize, long style, const wxString&) {}
}
```

   Funkcja `Create` którą tworzy wxSmith CodeBlocksie jest używana już w fizykaFrame: dodanie tego zapisu sprawia, że nie jest uruchamiana dwa razy.

2.2.2 Dodawanie mapy do menu

Dodawanie mapy do menu wymaga dodania jakiegoś przycisku który tę mapę uruchomi, dodania głównych plików tej mapy (informacje o buildzie też będzie trzeba usunąć tak jak w fizyce - to menu je zapewni).

Implementacja porażki i zwycięstwa:

```
void MenuFrame::Przycisk_Start(wxCommandEvent& event) {

    if(mapka != NULL && (mapka->czy_wygrana || mapka->czy_przegrana)) {
        delete mapka;
    }

    mapka = new mapkaFrame(0);
    mapka->Show();
    mapka->SetFocus();
    Timer1.Start();
}
```

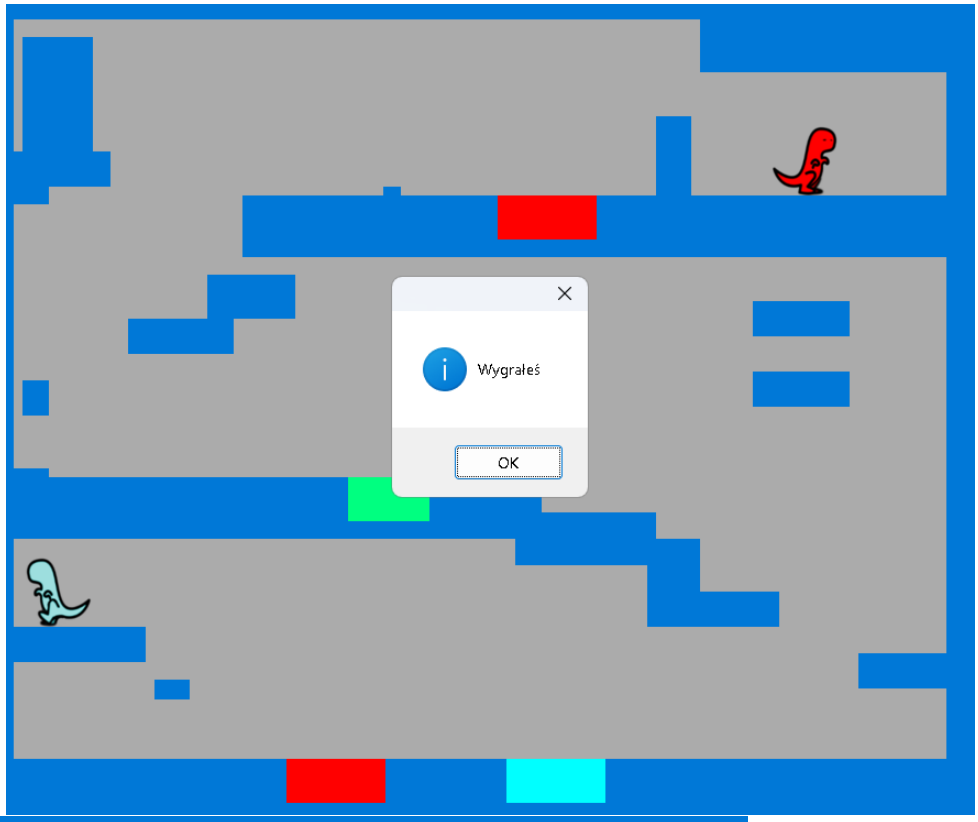
To co się po ewentualnej przegranej można rozwiązać za pomocą timera

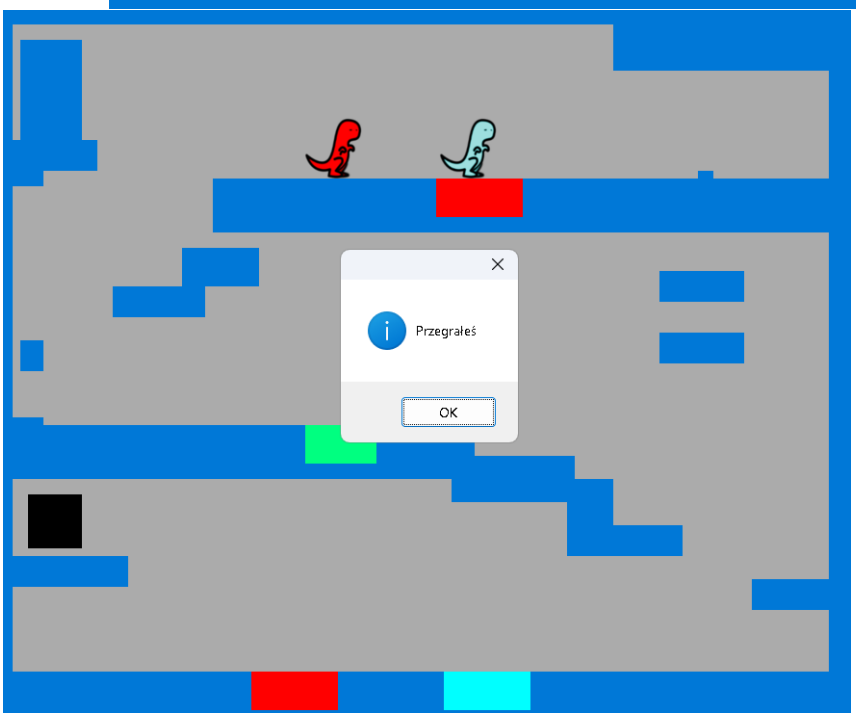
```
void MenuFrame::OnTimer1Trigger(wxTimerEvent& event) {

    if(mapka != NULL && (mapka->czy_wygrana || mapka->czy_przegrana)){
        Timer1.Stop();
        if(mapka->czy_wygrana)
            wxMessageBox(wxString::FromUTF8("Wygrałeś"), "");
        else
            wxMessageBox(wxString::FromUTF8("Przegrałeś"), "");

        mapka -> Hide();

        Timer1.Stop();
    }
}
```


Wygrałeś
OK


Przegrałeś
OK

3 Aproksymacja udziału w projekcie

Poniższe diagramy prezentują szacowany udział poszczególnych udziałowców projektu: