Strzelce Opolskie, 30.06.2018

Laboratorium Programowania komputerów

Temat:

Gra kółko-krzyżyk

Autor: Jakub Klimek
Informatyka, semestr 2, gr. 1
Prowadzący: mgr inż. Grzegorz Wojciech Kwiatkowski
Adres programu: Odnośnik do serwisu Dropbox

Spis treści

Temat ... 4

Analiza i Projektowanie ... 4

Analiza problemu ... 4

Algorytm ... 4

Specyfikacja zewnętrzna ... 6

Uruchamianie programu ... 6

Obsługa programu ... 6

Singleplayer ... 7

Multiplayer ... 8

Rozwiazywanie problemów ... 9

Specyfikacja wewnętrzna ... 10

Zmienne globalne ... 10

Typy wyliczeniowe ... 11

Ważne zmienne ... 12

Funkcje ... 13

buton.h ... 13

game.h i game.c ... 13

inputBox.h ... 14

mediaLoader.h ... 15

minmax.h ... 15

minmax.c ... 16

scene.h ... 16

texture.h ... 16

utils.h ... 17

textureList.h ... 17

Funkcje scen ... 18

sceneMenu.c ... 19

sceneSingleplayer.c ... 19

sceneScoreboard.c ... 20

sceneMultiplayerEngaging.h ... 20

sceneMultiplayerEngaging.c ... 21

sceneMultiplayer.c ... 21

Wydruk programu ... 22

Testowanie 28
Wycieki pamięci 29
Valgrind 29
Visual Studio 30
Wnioski 31

Temat

Napisać grę kółko-krzyżyk działającą w trybie graficznym, zawierającą możliwość gry samemu oraz przeciwko innemu graczowi.

Analiza i Projektowanie

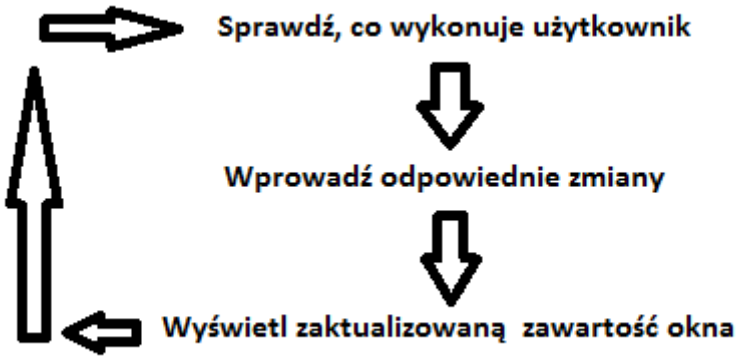
Analiza problemu

Ponieważ aplikacja ma działać w trybie graficznym, na samym początku pojawił się pierwszy problem do rozważenia: biblioteka użyta do wyświetlania obrazu. Zdecydowałem się na bibliotekę SDL, ponieważ jest ona używana przez największe studia developerskie takie jak Valve oraz posiada rozbudowaną dokumentację. Ponadto jest multiplatformowa, co pozwoli na łatwe przeniesienie aplikacji na systemy Windows (proces tworzenia aplikacji planuję wykonywać na Linuxie).

Pracując nad tym projektem chciałbym utworzyć szkielet, który pozwoli mi nie tylko na zrobienie gry kółko-krzyżyk, ale także będzie solidną podstawą podczas tworzenia przyszłych aplikacji okienkowych opierających się na grafice 2D. Ponieważ SDL dostarcza narzędzia do budowy aplikacji, a nie gotowe klocki, z których składamy aplikację(takie podejście oferuje np. biblioteka Qt), uważam, że ta biblioteka będzie idealna do tego projektu.

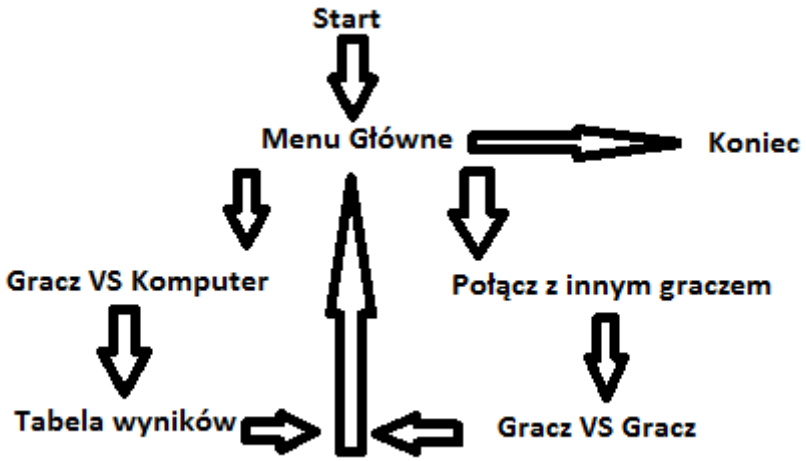
Algorytm

Sam algorytm działania planuję oprzeć o nieskończoną pętlę:



Wyjście z pętli będzie warunkowane odpowiednią akcją użytkownika, np.. kliknięciem krzyżyka w rogu okna lub wybraniem odpowiedniej pozycji w menu.

Uważam, że logiczne będzie podzielenie zawartości okna na sceny. Każda scena jest swoistym stanem aplikacji, tj. stanowi o aktualnej zawartości okna, metodach interakcji z użytkownikiem oraz o metodach przejścia do następnej sceny. Wyobrażam sobie następujący graf przejść między scenami:



Innymi słowy, program zaczynałby się w menu głównym, a każda próba wyjścia z programu następowałaby po wybraniu odpowiedniej pozycji w głównym menu.

Sama pętla programu to nie wszystko. Konieczne będą takie elementy jak tekstury, czcionki i odpowiednie metody ich przechowywania.

Jeżeli chodzi o czcionki, nie zajmują one dużo pamięci, a ich il;pść nie będzie duża, więc uważam, że można je przechowywać w tablicy. Natomiast przy teksturach pojawiają się dwie istotne kwestie:

- Rozmiar tekstur
- Ilość tekstur
- Moment ich ładowania

Dla tego programu nie przewiduję tekstur dużej rozdzielczości, a ich ilość nie powinna przekroczyć kilkunastu. Takie okoliczności pozwoliłyby na zastosowanie tablicy jako metody ich przechowywania, jednak ponieważ planuję wykorzystywać główny kod w przyszłych projektach, muszę się przygotować na sytuację, w której program będzie musiał sobie poradzić z brakiem ciągłego miejsca w pamięci na tekstury. Dlatego decyduję się na użycie dynamicznej struktury – listy jednokierunkowej do przechowywania tekstur. Inne, mniej istotne z perspektywy pamięci dane planuję przechowywać w tablicach.

W trybie pojedynczego gracza planuję użycie algorytmu minmax do określania pozycji gracza sterowanego przez komputer. Algorytm zawsze zwróci najoptymalniejszy ruch, a jeżeli dla komputera nie będzie perspektywy wygrania, będzie on przeciągał grę jak najdłużej. Pozwoli to na utworzenie kilku poziomów trudności, między którymi użytkownik będzie mógł się przełączać.

Do komunikacji w trybie multiplayer chcę wykorzystać protokół TCP. Liczne mechanizmy potwierdzania poprawności danych, czy komunikacja polegajaca na potwierdzaniu odebranych danych znacznie upraszczają proces projektowania. Z perspektywy ilości przesłanych danych, użycie lżejszego protokołu takiego jak UDP wydaje się pozbawione sensu majac na uwadze wyżej przedstawione zalety TCP.

Specyfikacja zewnętrzna

Uruchamianie programu

Program nie wymaga parametrów podawanych przy uruchamianiu, ale mogą one okazać się pomocne. Są to:

- -d [liczba z przedziału <1;4>] – określa poziom trudności gry z komputerem
- -r – usuwa obecny zapis stanu gry i czyści tablice najlepszych wyników

Przykładowe metody użycia:

- ./ttt
- ./ttt –r
- ./ttt –d 4
- ./ttt –d 3 –r

Użycie programu z nieprawidłowymi parametrami spowoduje wyświetlenie krótkiej informacji:

```
Wrong parameters. Use with:
-r - to restore default settings
-d [difficulty from 1 to 4] - to set single player difficulty
```

Obsługa programu

Program uruchamia się w oknie, po którym poruszamy się używając myszki.
Pierwszym oknem, ma które trafi użytkownik będzie menu główne:

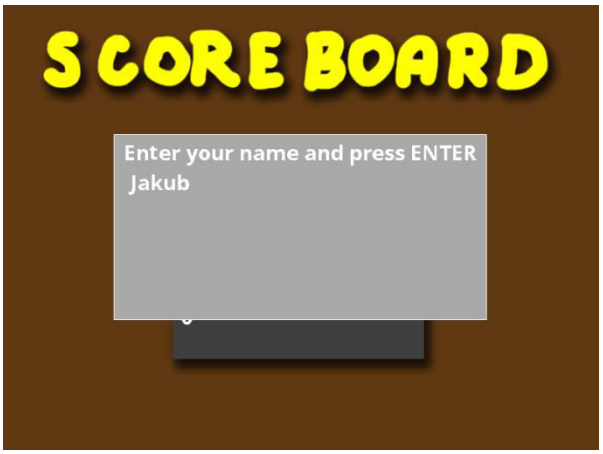
Użytkownik ma do dyspozycji 3 przyciski:

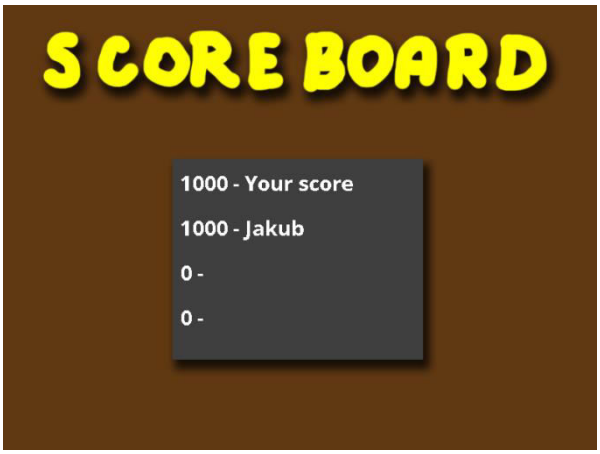
- Player vs AI – zaczyna grę przeciwko komputerowi
- Multiplayer – przejście do ekranu łączenia z innym graczem
- Quit Game – wychodzi z gry

Singleplayer

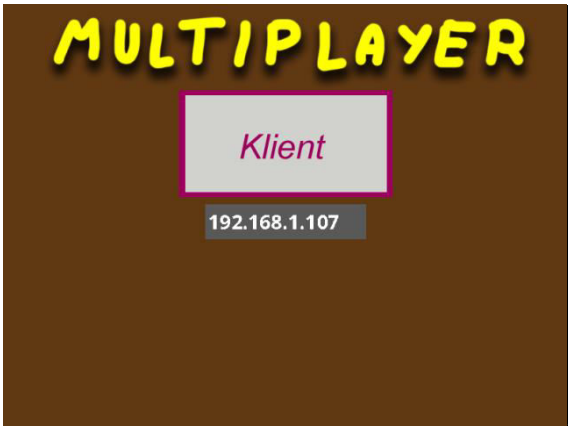
Kliknięcie na wolnym polu skutkuje postawieniem tam symbolu gracza (X). Pola zajęte przez komputer oznaczane są przy pomocy kółka (O). Dana strona zwycięży, jeżeli uda jej się zająć 3 pola w rzedzie, kolumnie lub po przekątnej. Każde cofnięcie ruchu skutkuje odjęciem punktów.

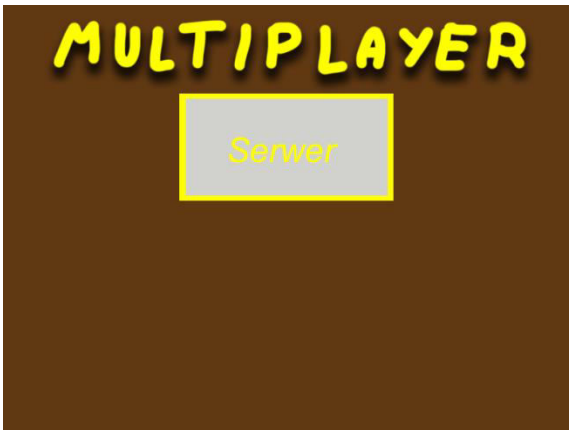
Po zakończonej grze, użytkownikowi zostanie przedstawiona tablica najlepszych wyników, a jeżeli jego wynik okaże się dostatecznie dobry, zostanie poproszony o wpisanie swojej nazwy przy pomocy klawiatury i zatwierdzenie jej klawiszem ENTER:

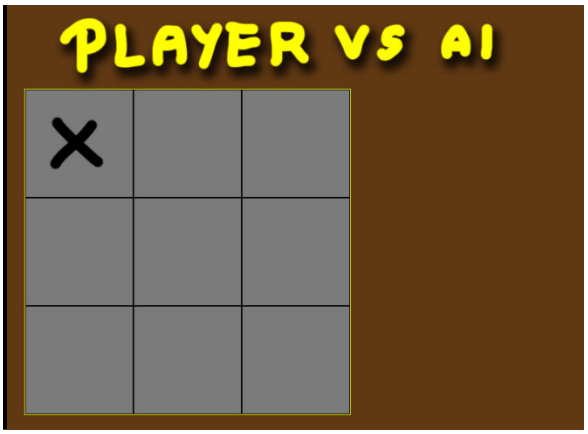




Multiplayer







Aby zmienić tryb działania, należy nacisnąć na przycisk Client lub Server. Przycisk zmienia się w zależności od wybranej roli. W trybie klienta użytkownik musi wprowadzić adres ip serwera przy pomocy klawiatury. Próba połączenia następuje po naciśnięciu klawisza ENTER. Jest ona symbolizowana napisem „Connecting” na środku ekranu. W przypadku niepowodzenia użytkownik wróci do ekranu wpisywania adresu IP. Natomiast w przypadku powodzenia użytkownikowi zostanie zaprezentowana plansza do gry podobnej do tej z gry przeciwko komputerowi.

Symbole graczy zależą od roli, jaką przyjęli podczas łączenia:

- Serwer – X
- Klient – O

Rozwiazywanie problemów

"…. nie mogl zostac zinicjalizowany” – W systemie brakuje wskazanej biblioteki. Można ja ściągnąć bezpośrednio ze strony SDL lub z oficjalnego repozytorium dystrybucji (Linux).

"SDL nie mogl utworzyc okna” – Błąd wewnętrzny biblioteki SDL. Może wynikać z braku uprawnień, zbyt małej przestrzeni roboczej (przynajmniej 800px na 600px) lub braku zainstalowanych sterowników graficznych.

„Nie mozna zaladowac obrazu: …„ – Upewnij się, że folder **media,** dostarczony razem z kodem źródłowym, znajduje się w tym samym miejscu co plik wykonywalny

Przycisk cofnięcia ruchu w grze przeciw komputerowi jest nieaktywny - gracz nie wykonał ruchu lub cofnął już ruch.

Podczas łączenia w trybie dla wielu graczy napis *„Connecting”* znika, a gra nie przechodzi do następnej sceny – połączenie się nie powiodło. Możliwe, że zapora sieciowa blokuje połączenie lub wpisany został błędny adres IP.

Specyfikacja wewnętrzna

Sama biblioteka gry zdefiniowana jest w następujących plikach

Pliki nagłówkowe **.h**	Pliki źródłowe **.c**
button.h game.h inputBox.h mediaList.h mediaLoader.h minmax.h scene.h sceneList.h texture.h textureList.h utils.h bool.h defines.h sceneGlobals.h	button.c game.c inputBox.c mediaList.c mediaLoader.hc minmax.c scene.c sceneList.c texture.c textureList.c utils.c

Ponadto, w skład tej konkretnie aplikacji wchodzą pliki :

Pliki nagłówkowe **.h**	Pliki źródłowe **.c**
sceneMenu.h sceneMultiplayer.h sceneMultiplayerEngaging.h sceneScoreboard.h sceneSingleplayer.h	sceneMenu.c sceneMultiplayer.c sceneMultiplayerEngaging.c sceneScoreboard.c sceneSingleplayer.c main.c

Pliki scene___.h i scene___.c są to pliki scen, traktowane przeze mnie jako pliki-obiekty scen. Każdy z tych zestawów plików definiuje jedną niepowtarzalna scenę w programie.

Zmienne globalne

Ze względu na budowę biblioteki SDL, w każdym pliku w którym chcę wyświetlić coś na ekranie, muszę dostarczyć wskaźnik na element **SDL_Window** (okno programu) oraz **SDL_Renderer**(płótno, po którym rysujemy). To czyni z nich zmienne o charakterze globalnym, do których dostęp realizowany jest poprzez dwie funkcje z pliku game.h:

```
SDL_Window* getWindow()
SDL_Renderer* getRenderer()
```

Funkcje zostaną omówione w dalszej części programu. Ponadto, w pliku defines.h znajdują się 2 instrukcje preprocesora, definiujące szerokość i wysokość okna:

```
#define SCREEN_WIDTH 800
#define SCREEN_HEIGHT 600
```

Typy wyliczeniowe

Buton.h

```
enum buttonState {
        BUTTON_DEFAULT,
        BUTTON_MOUSEOVER,
        BUTTON_PRESSED
};
```

Typ opisuje stan przycisku, w jakim się znajduje

mediaList.h

```
enum fontList{
                FONT_OPENSANS_BOLD,
                FONT_COUNT
};
```

Do tego typu wyliczeniowego należy dodać wszystkie czcionki, które chcemy załadować podczas startu aplikacji

mediaList.h

```
enum mediaList{
                IMG_HELLOWORD,
                IMG_UP,
                IMG_DOWN,
                IMG_LEFT,
                IMG_RIGHT,
                IMG_MENU_BG,
                …
                MEDIA_COUNT
};
```

Do tego typu wyliczeniowego należy dodać wszystkie tekstury, które chcemy załadować podczas startu aplikacji

minmax.h

```
enum whoWon{
        OPPONENT = -10,
        NONE = 0,
        PLAYER = 10
};
```

Typ wyliczeniowy określający, kto wygrał w grze singleplayer

minmax.h

```
enum signs {
        X = 'x',
        O = 'o'
};
```

Typ wyliczeniowy zawierający symbole graczy

sceneList.h

```
enum sceneList{
        MAIN_MENU,
        SINGLEPLAYER,
```

```
        MULTIPLAYER_ENGAGING,
        SCOREBOARD,
        MULTIPLAYER,
        QUIT
};
```

Do tego typu wyliczeniowego należy dodać wszystkie sceny występujące w programie

sceneMultiplayerEngaging.h

```
enum messageType {
        /*0-8 - recently assigned tile */
        ABORT_CONNECTION = 9,
        CHECK_CONNECTION = 10
};
```

Typ wyliczeniowy określający zawartość wiadomości sieciowej

sceneMultiplayerEngaging.h

```
enum multis {
        SERVER,
        CLIENT,
} multiplayerState;
```

Typ wyliczeniowy określający, czy aplikacja jest klientem czy serwerem (potrzebne do ustalenia, czyja teraz tura)

Ważne zmienne

mediaList.c

```
char* mediaLocations[]={
                        /*TEXTURES*/
                        "media/img/helloworld.bmp",
                        "media/img/up.png",
                        "media/img/down.png",
                        "media/img/left.png",
                        "media/img/right.png",
                        "media/img/menuBg.png",
                        …
                        /*FONTS*/
                        "media/fonts/OpenSans-Bold.ttf"
};
```

Tablica zawierająca ścieżki do plików określonych w typie wyliczeniowym mediaList i fontList

sceneList.c

```
struct scene* sceneList[] = {
                &menuScene,
                &singleplayerScene,
                &multiplayerEngagingScene,
                &scoreboardScene,
                &multiplayerScene
};
```

Tablica zawierająca adresy wszystkich scen

Funkcje

buton.h

struct button* createButton(**int** *x*, **int** *y*, **enum mediaList** *defaultTexture*, **enum mediaList** *mouseoverTexture*, **enum mediaList** *pressedTexture*, **void** (***pFunction*)**()**)
Funkcja tworzy nowy przycisk (**struct button**) i zwraca jego **adres**. Parametry: *x,y* - pozycja przycisku (lewo górny róg) *defaultTexture* - nazwa domyślnej tekstury przycisku *mouseoverTexture* - nazwa tekstury, która ma się wyświetlić po najechaniu na przycisk myszka *pressedTexture* - nazwa tekstury, która ma się wyświetlić po kliknięciu przycisku *pFunction* - wskaznik na funkcje, która ma się wykonać po naciśnięciu przycisku **Wynik zwracany:** Jeżeli operacja się powiedzie, zwraca adres struktury, w przeciwnym wypadku NULL

void renderButton(**struct button*** *pButton*)
Funkcja renderuje przycisk na domyślnym ekranie. Parametry: *pButton* - adres przycisku, który ma zostać wygenerowany

void handeEvent(**struct button*** *btn*, **SDL_Event*** *e*)
Funkcja sprawdza, czy event który wystąpił dotyczy przycisku. Jeżeli tak, wykonuje funkcje, której adres został przekazany podczas tworzenia przycisku. Parametry: *btn* - adres przycisku *e* - wskaźnik na zmienną przechowującą informacje o evencie

void destroyButton(**struct button*** *pointer*)
Funkcja usuwa przycisk z pamięci. Parametry: *pointer* - adres przycisku

void changeButtonTexture(**struct button*** *pButton*, **enum buttonState** *state*, **enum mediaList** *texture*)
Funkcja zmienia teksturę przycisku na inna teksturę (tekstura musi zostać załadowana podczas uruchamiania). Parametry: *pButton* - adres przycisku *state* - stan przycisku, dla którego teksturę chcemy zmienić *texture* - nazwa tekstury, którą chcemy ustawić dla danego stanu

void toggleButtonVisibility(**struct button*** *pButton*)
Funkcja wyłącza/włącza widoczność przycisku (domyślnie przycisk jest widoczny). Parametry: *pButton* - adres przycisku

game.h i game.c

int run()
Funkcja uruchamia GUI i zawiera cala logikę gry. Zwraca 0, nawet w przypadku wystąpienia problemów.

void restoreDefaults()
Funkcja usuwa plik z wynikami i poprzedni zapis gry (o ile istnieją)

int difficulty(**int** *difficulty*)
Funkcja pobiera/ustawia trudność gry singlePlayer. Parametry: *difficulty* - jeżeli w przedziale <1;4> - ustawia nowy poziom trudności i go zwraca, w innym przypadku - zwraca aktualny poziom trudności

SDL_Window* getWindow()
Funkcja zwraca adres zmiennej globalnej typu **SDL_Window,** potrzebnej do rysowania po oknie

SDL_Renderer* getRenderer()
Funkcja zwraca adres zmiennej globalnej typu **SDL_Renderer,** potrzebnej do rysowania po oknie

SDL_Window** setWindow(**SDL_Window*** *pGWindow*)
Funkcja używana przez **`SDL_Window*`** `getWindow()` do utworzenia obiektu typu **SDL_Window**

SDL_Renderer** setRenderer(**SDL_Renderer*** *pRenderer*)
Funkcja używana przez **`SDL_Renderer*`** `getRenderer()` do utworzenia obiektu typu **SDL_Renderer**

bool inicjalizacja()
Funkcja inicjalizuje bibliotekę SDL. **Wartość zwracana:** *true* jeżeli inicjalizacja się powiodła, w przeciwnym wypadku *false*

void zamknij()
Funkcja usuwa z pamięci wszystkie załadowane podczas uruchamiania elementy i wyłącza bibliotekę SDL

inputBox.h

struct inputBox* createInputBox(**size_t** *textFieldSize*, **int** *x*, **int** *y*, **bool** *isVisible*, **enum fontList** *font*,**uint8_t** *r*, **uint8_t** *g*, **uint8_t** *b*, **bool** *addBackground*, **enum mediaList** *defaultTexture*)
Funkcja tworzy nową strukturę **inputBox** i zwraca jej **adres**. Parametry: * *textFieldSize* - ile (maksymalnie) liter na przechowywać pole * *x,y* - pozycja pola na ekranie (lewy, górny róg) * *isVisible* - czy pole ma być widoczne * *font* - nazwa czcionki, której pole ma używać do wyświetlania tekstu * *r,g,b* - kolor czcionki * *addBackground* - czy pole ma tło * *defaultTexture* - nazwa tekstury, która ma być tłem dla pola (załadowana podczas ładowania programu). * Jezeli wczesniejszy parametr wynosił false, wartość tego pola nie ma znaczenia

void destroyInputBox(**struct inputBox*** *pInputBox*)
Funkcja zwalnia z pamięci strukturę **inputBox**. Parametry: * *pInputBox* – adres **struktury**

void inputBoxAppendChar(**struct inputBox*** *pInputBox*, **char** *character*)
Funkcja dodaje do struktury **InputBox** jeden znak. Parametry: * *pInputBox* - adres struktury **InputBox** * *character* - znak, który ma zostać dodany do pola

void inputBoxRender(**struct inputBox*** *pInputBox*)
Funkcja renderuje pole na domyślnym ekranie. Parametry: * *pInputBox* - adres struktury **inputBox**

void inputBoxToggleVisibility(**struct inputBox*** *pInputBox*)
Funkcja włącza/wyłącza widoczność pola. Parametry: * *pInputBox* - adres struktury **inputBox**

void inputBoxClearTextField(**struct inputBox*** *pInputBox*)
Funkcja czyści zawartość pola (Jako zawartość rozumiemy ciąg znaków, który pole wyświetla). Parametry: * *pInputBox* - adres struktury **inputBox**

void inputBoxPopCharacter**(struct inputBox*** *pInputBox*)
Funkcja usuwa z **struktury** jeden znak (o ile taki istnieje). Parametry: * *pinputBox* - adres struktury **inputBox**

mediaLoader.h

bool loadMedia()
Funkcja ląduje pliki określone w plikach mediaList.* do pamięci. Wynikiem działania funkcji jest informacja, czy udało się wszystko załadować(*true* jeżeli tak, *false* jeżeli nie)

SDL_Texture* getTexture(**int** *mediaNumber*)
Funkcja zwraca **wskaźnik** teksturę (**SDL_Texture**) o podanej nazwie. Parametry: * *mediaNumber* - nazwa tekstury (z typu wyliczeniowego **mediaList**)

struct texture* getTextureStr**(int** *mediaNumber*)
Funkcja zwraca **wskaźnik** na strukturę **texture** o podanej nazwie. Parametry: * *mediaNumber* - nazwa tekstury(z typu wyliczeniowego **mediaList**)

TTF_Font* getFont(**int** *fontNumber*)
Funkcja zwraca **wskaźnik** na typ **TTF_Font** o podanej nazwie. Parametry: * *fontNumber* - nazwa czcionki(z typu wyliczeniowego **fontList**)

void unloadMedia()
Funkcja usuwa z pamięci wszystkie elementy, które zostały załadowane przez funkcję **loadMedia()**

minmax.h

bool isEmptyPlace**(char** *board[3][3]*)

Funkcja sprawdza, czy na planszy istnieje wolne pole w tablicy 3x3 i zwraca wynik tej operacji. Parametry:
* *board* – tablica 3x3, której dotyczy sprawdzenie
Zwracana wartość: *true* jeżeli jest wolne miejsce, w przeciwnym wypadku *false*

enum whoWon lookForWinner**(char** *board[3][3]*)

Funkcja sprawdza, czy ktoś wygrał grę. Parametry:
* *board* - tablica 3x3, plansza, której dotyczy sprawdzenie
Zwracana wartość: zwycięzca zgodnie z typem wyliczeniowym **whoWon**

int minMax**(char** *board[3][3]*, **int** *depth*, **bool** *isMaximalising*)

Funkcja sprawdza, jak dobra dla danego gracza jest plansza. Zwraca wynik w postaci punktowej. Parametry:
* *board* – tablica 3x3, plansza, której dotyczy sprawdzenie
* *depth* - "głębokość" pomiaru
* *isMaximalising* - czy oczekujemy największego czy najmniejszego wyniku

struct minMax_Move findBestMove(**char** *board[3][3]*)

Funkcja ma za zadanie wyszukać najlepszy ruch dla komputera. Parametry
* *board* - tablica 3x3, plansza, na której rozgrywa komputer
Zwracana wartość: struktura **minMax_Move** zawierająca najoptymalniejszy ruch

void setupBot(**enum signs** *sign*)

Funkcja ustawia komputerowi znak. Parametry
* *sign* - znak komputerowego gracza (typ wyliczeniowy **signs**)

minmax.c

int max(**int** *a*, **int** *b*)

Funkcja sprawdza, która liczba jest większa, **a** czy **b** i **zwraca** większą z nich

int min(**int** *a*, **int** *b*)

Funkcja sprawdza, która liczba jest mniejsza, **a** czy **b** i **zwraca** mniejszą z nich

scene.h

void selectScene(**enum sceneList** *scene*)

Funkcja ustawiająca daną scenę na aktualną.
Podczas zmiany sceny następuje usuniecie z pamięci wcześniejszej sceny i załadowanie następnej.
Parametry:
* *scene* - nazwa sceny (typ wyliczeniowy **sceneList**)

texture.h

void renderTexture(**struct texture*** *pTexture*, **int** *x*, **int** *y*)

Funkcja wyświetla **teksturę** w domyślnym oknie. Parametry:
* *pTexture* – adres struktury **texture**
* *x,y* - współrzędne tekstury (lewy górny róg)

bool createFromText(**struct texture*** *pTexture*, **int** *fontName*, **const char*** *text*, **SDL_Color** *textColor*)
Funkcja tworzy nową teksturę zawierającą podany **tekst** i zapisuje ją. Wynikiem funkcji jest informacja, czy udało się utworzyć teksturę (*true* jeżeli tak, *false* jeżeli nie). Parametry: * *pTexture* - adres, pod którym ma być dostępna **struktura** (jeżeli w strukturze znajduje się inna tekstura, usuwa ją) * *fontName* - nazwa czcionki, która ma zostać użyta (typ wyliczeniowy **fontList**) * *text* - **ciąg znakowy**, który ma wystąpić na teksturze * *textColor* - kolor czcionki

void destroyTexture(**struct texture*** *pTexture*)
Funkcja usuwa z pamięci strukturę **texture** i przechowywaną przez nią strukturę **SDL_Texture**. Parametry: * *pTexture* - adres **struktury**

utils.h

int checkParameters(**int** *argc*, **char**** *argv*)
Funkcja sprawdza parametry i reaguje na nie. Parametry: * *argc* – liczba parametrów * *argv* - tablica *łańcuchów znakowych* (tablica parametrów) W przypadku wystąpienia nieznanego parametru funkcja **zwraca** wartość -1, w przeciwnym wypadku 0.

textureList.h

struct textureList* listAddTexture(**struct textureList*** *root*, **unsigned int** *id*, **struct texture*** *pTexture*)
Funkcja dodaje teksturę do listy. Parametry * *root* - głowa listy * *id* - nazwa(**id**) tekstury * *pTexture* - wskaźnik na strukturę **texture** * Funkcja zwraca **adres** nowo utworzonego elementu**.**

struct textureList* listAddElement(**struct textureList*** *root*, **unsigned int** *id*)
Funkcja dodaje do **listy** element o danym **id**, ale z pustą teksturą. Parametry: * *root* - głowa listy * *id* - nazwa(id) elementu(tekstury). * Funkcja zwraca adres nowo utworzonego elementu struct **textureList**

struct textureList* listGetElement(**struct textureList*** *root*, **unsigned int** *id*)
Funkcja szuka w **liście** elementu (struktury **textureList**) o danym **id** i zwraca jego **adres** (lub NULL, jeżeli taki nie istnieje). Parametry: * *root* - głowa listy * *id* - numer(**id**) elementu(struktury **textureList**)

struct texture* listGetTexture(**struct textureList*** *root*, **unsigned int** *id*)
Funkcja działa analogicznie do **listGetElement**(...), z tą różnicą, ze zwraca **adres** struktury **texture**, którą przechowywał element

struct textureList* listRemoveItem(**struct textureList**** *root*, **unsigned int** *id*)

Funkcja usuwa z listy element o danym **id** i zwraca **adres wcześniejszego elementu**. Parametry: * *root* - głowa **listy**(może ulec zmianie jeżeli usuwamy pierwszy element) * *id* - nazwa(**id**) elementu

void freeList(**struct textureList*** *root*)
Funkcja zwalnia pamięć zajmowana przez listę. UWAGA: Zwolnieniu podlegają także tekstury, które są przechowywane w liście. Parametry: * *root* - głowa **listy**

Funkcje scen

W każdym pliku źródłowym sceny (np. sceneMenu.c) powinny się znaleźć cztery statyczne funkcje:

static void init()
Zadaniem funkcji jest załadowanie do pamięci sceny

static void unInit()
Zadaniem funkcji jest usunięcie z pamięci wszystkiego, co dana funkcja używała podczas swojego istnienia

static void handleEvents(**SDL_Event ***e*)
Zadaniem funkcji jest obsługa wydarzeń (interakcji ze strony użytkownika z programem). Parametry: * *e* – wskaźnik na typ **SDL_Event** zawierający informacje o akcjach użytkownika

static void renderScene()
Funkcja wyświetla na domyślnym ekranie (przekazywanym przez **getWindow()**) zawartość sceny

Należy zwrócić uwagę, że każda scena musi posiadać te cztery funkcje oraz dołączony plik nagłówkowy *game.h* i *sceneGlobals.h*. Ten pierwszy zawiera funkcja dające dostęp do zmiennych globalnych, natomiast drugi deklaruje zmienne statyczne potrzebne do poprawnego działania w/w funkcji. Zwracam uwagę, że każdą scenę, tj. plik **.h** i **.c** traktuję jako niepowtarzalny obiekt opisujący scenę i implementujący ją. W każdej scenie implementacja powyższych funkcji może się różnić, jednak powinna ona być ona spójna (np. przyciski są dodawane w funkcji **init()** tylko i wyłącznie).

UWAGA: Ponieważ każda scena z założenia jest indywidualnym obiektem, używam statycznych zmiennych jak zmiennych prywatnych w językach obiektowych. Są one dostępne tylko z poziomu danego pliku źródłowego, niewidoczne podczas procesu linkowania, dlatego traktuję je jako zmienne lokalne.

Dwie z tych zmiennych muszą być ustawione wcześniej (przed rozpoczęciem rezerwowania pamięci):

```
static int btnCount = 9;
static int inputBoxCount = 0;
```

Pierwsza z nich służy do określenia ile przycisków znajduje się na scenie (w celu zarezerwowania pamięci), a druga do określenia ilości pól służących do wprowadzania tekstu.

Każdy z niżej przedstawionych plików źródłowych (**.c**) scen zawiera cztery wyżej wymienione funkcje, dlatego przedstawię tylko funkcje charakterystyczne.

sceneMenu.c

Funkcje przycisków – zostają wywołane gdy przycisk zostanie wciśnięty	
static void quitBtnClicked()	Funkcja wywoływana po kliknięciu przycisku „Quit Game”. Zmienia scenę na **QUIT**
static void singleBtnClicked()	Funkcja wywoływana po kliknięciu przycisku „Player vs AI”. Zmienia scenę na **SINGLEPLAYER**
static void multiBtnClicked()	Funkcja wywoływana po kliknięciu przycisku „Multiplayer”. Zmienia scenę na **MULTIPLAYER_ENGAGING** (tutaj nastąpi połączenie klienta z serwerem)

sceneSingleplayer.c

Funkcje przycisków – zostają wywołane gdy przycisk zostanie wciśnięty	
static void boardClicked()	Funkcja wywoływana po kliknięciu planszy do gry. Zmienia wartość zmiennej informującej o tym, że pole planszy zostało kliknięte
static void loadBtnClicked()	Funkcja wywoływana po kliknięciu przycisku „Wczytaj”. Odczytuje z dysku plik zapisu, i o ile jest to możliwe, wczytuje stan gry
static void undoBtnClicked()	Funkcja wywoływana po kliknięciu przycisku „Cofnij”. Jeżeli jest to możliwe, cofa stan planszy o jedną pełną turę (tj. jeżeli został wykonany ruch)
static void saveBtnClicked()	Funkcja wywoływana po kliknięciu przycisku „Zapisz”. Zapisuje stan gry na dysku w pliku „save.dat”

Inne funkcje	
static void refreshBoardTextures()	Funkcja aktualizuje tekstury planszy, aby zgadzały się z tablicą przechowującą stan planszy
static void printMessage(**struct texture *** *text*, **const char*** *msg*)	Funkcja wypisuje wiadomość na ekranie na czas 2s. Parametry: * *text* – wskaźnik na **strukturę texture**, na której zostanie „napisana” wiadomość * *msg* – **ciąg znaków** będący wiadomością dla użytkownika
static void makeAIMove()	Funkcja wykonuje ruch na planszy (ruch gracza komputerowego)
static void handleQuad**(int** *processedButton*)	Zadaniem funkcji jest reakcja na kliknięcie przez gracza pola planszy (numerowanego od 0 do 8). Parametry: * *pressedButton* – nr pola planszy, które zostało kliknięte
static void disableAllFunctionButtons()	Funkcja wyłącza przyciski „Zapisz”, „Cofnij”, „Wczytaj”
static void enableUndoButton()	Funkcja włącza przycisk „Cofnij”
static void disableUndoButton()	Funkcja wyłącza przycisk „Cofnij”
static void disableLoadButton()	Funkcja wyłącza przycisk „Wczytaj”
static void disableSaveButton()	Funkcja wyłącza przycisk „Zapisz”,

sceneScoreboard.c

Inne funkcje	
void deleteScores ()	Funkcja usuwająca wszystkie wyniki z pamięci
void loadScores()	Funkcja ładująca do pamięci wyniki z pliku. Jeżeli jest to niemożliwe, ustawia 0 jako wyniki.
int tryScore(**int** *score*)	Funkcja sprawdza, czy wynik(**score**) trafiłby na tablice wyników. Parametry: * *score* - sprawdzany wynik Funkcja **zwraca** -1 w przypadku gdy nie trafi, w przeciwnym wypadku pozycję, na którą trafiłby
int placeScore(**int** *score*)	Funkcja umieszcza wynik(**score**) na tablicy wyników. Parametry: * *score* - wynik Funkcja **zwraca** -1 w przypadku gdy wynik nie trafił na tablicę, w przeciwnym wypadku pozycję, na którą trafił
void saveScores()	Funkcja zapisuje wyniki z tablicy wyników na dysku i usuwa je z pamięci
void addScore(**int** *score*)	Funkcja informująca scenę o wyniku(**score**), jaki osiągnął gracz.Parametr: * *score* – wynik, który gracz otrzymał w grze przeciw komputerowi
void prepareScoresTexture()	Funkcja przygotowywuje tekstury z wynikami do wyswietlenia

sceneMultiplayerEngaging.h

TCPsocket* getSocket()
Funkcja **zwraca** adres struktury **TCP_Socket**. Jeżeli taka struktura istnieje zwraca jej adres, w przeciwnym wypadku NULL

void closeSocket()
Funkcja zamyka socket(struktura **TCP_Socket**) służący do komunikacji z innym graczem i zwalnia przyznaną mu pamięć.

enum multis getClientState()
Funkcja **zwraca** status aplikacji (klient/serwer) jako typ wyliczeniowy **multis**.

bool sendMessage(**enum messageType** *msg*)
Funkcja wysyła przez **socket** wiadomość do drugiego gracza. Parametry: * *msg* – wiadomość do innego gracza zgodna z typem wyliczeniowym **messageType.** Funkcja **zwraca** *true*, jeżeli wiadomość udało się wysłać, w przeciwnym wypadku *false*.

SDLNet_SocketSet* getSocketSet()
Funkcja **zwraca** adres struktury **SDLNet_SocketSet** potrzebnej do sprawdzenia aktywności **socketu**. Jeżeli taka struktura nie istnieje, zwraca NULL.

bool isActiveSocket()
Funkcja sprawdza, czy w ostatnim czasie pojawiła się jakaś aktywność na **sockecie** (komunikacja z innym graczem). **Zwracana wartość**: *true* jeżeli aktywność się pojawiła, w przeciwnym wypadku *false*.

sceneMultiplayerEngaging.c

Funkcje przycisków – zostają wywołane gdy przycisk zostanie wciśnięty	
static void clientServerBtnClicked()	Wywoływana po kliknięciu przycisku Client / Server. Powoduje zmianę układu sceny z klienta na serwer i odwrotnie.

Inne funkcje	
static void handleClientEvents(**SDL_Event** ***e**)	Funkcja obsługuje wejście, gdy scena jest w stanie „klient”. Parametry: * *e* – adres struktury **SDL_Event** zawierającej informacje o wejściu.
static int startServer()	Funkcja uruchamia serwer czekający na połączenie. **Zwraca 0** w przypadku powodzenia i **-1** w przypadku błędów.
static void stopServer()	Funkcja zatrzymuje działanie serwera i zwalnia pamięć używaną przez serwer.
static bool clientConnected()	Funkcja sprawdza, czy ktoś próbował połączyć się z serwerem. Jeżeli tak, nawiązuje połączenie z klientem i wyłącza serwer. **Zwracana wartość**: *true* jeżeli połączenie zostało nawiązane, *false* w przeciwnym wypadku.
static int connectToServer(**char*** *ipAddr*)	Funkcja łączy się z serwerem o danym **adresie ip**. Parametry: * *ipAddr* –łańcuch znaków zawierający adres IP serwera **Wartość zwracana**: **0** jeżeli połączenie udało się nawiązać, w przeciwnym wypadku **-1**.

sceneMultiplayer.c

Funkcje przycisków – zostają wywołane gdy przycisk zostanie wciśnięty	
static void boardClicked()	Funkcja wywoływana po kliknięciu planszy do gry. Zmienia wartość zmiennej informującej o tym, że pole planszy zostało kliknięte

Inne funkcje	
static void printEndMessage(**const char*** message)	Funkcja wypisuje wiadomość na ekranie końcową wiadomość. Parametry: * *msg* – **ciąg znaków** będący wiadomością dla użytkownika
static void checkForEnd()	Funkcja sprawdza, czy nastąpił koniec potyczki i reaguje w odpowiedni sposób.
static void handleQuad(**int** *processedButton*)	Zadaniem funkcji jest reakcja na kliknięcie przez gracza pola planszy (numerowanego od 0 do 8). Parametry: * *pressedButton* – nr pola planszy, które zostało kliknięte
static void handleIncomingData(**char** *data*)	Funkcja obsługuje odebraną paczkę danych i reaguje na jej zawartość. Parametry: * *data* – treść odebranej wiadomości.
static void handleRemotePlayer()	Funkcja obsługuje połączenie ze zdalnym graczem, m. in. Przetwarza informacje odebrane od niego

Wydruk programu

Z uwagi na ogromny rozmiar programu i dużą ilość plików, postanowiłem wydrukować najważniejsze z mojej perspektywy funkcje:

Funkcje odpowiedzialne za strukturę dynamiczną (textureList.c)

```
struct textureList* listAddElement(struct textureList* root, unsigned int id){
	if(root == NULL){ /*jezeli glowa nie istnieje*/
		struct textureList* pointer;
		pointer = malloc(sizeof(struct textureList));
		pointer->id = id;
		pointer->pElement = NULL;
		pointer->pNext = NULL;
		return pointer;/*zwracany wskaznik to jednoczesnie glowa nowej listy*/
	}
	struct textureList* pointer;
	for(pointer = root; pointer->pNext != NULL; pointer = pointer->pNext);/*Przejdz na koniec
listy*/
	/*Wstaw nowy element*/
	pointer->pNext = malloc(sizeof(struct textureList));
	pointer->pNext->id = id;
	pointer->pNext->pElement = NULL;
	pointer->pNext->pNext = NULL;
	return pointer->pNext;/*Zwroc jego adres*/
}
```

```
struct textureList* listAddTexture(struct textureList* root, unsigned int id, struct texture* pTexture){
	struct textureList* pointer = listAddElement(root,id);/*Dodaj element bez tekstury*/
	pointer->pElement = pTexture;/*Ustaw mu teksture*/
	return pointer;/*Zwroc jego adres*/
}
```

```
struct texture* listGetTexture(struct textureList* root, unsigned int id){
	struct textureList* pointer = listGetElement(root,id);/*Wyszukaj element*/
	return pointer == NULL ? NULL : pointer->pElement;/*Jezeli element nie istnieje zwroc NULL,
w przeciwnym wypadku przechowywana teksture*/
}
```

```
struct textureList* listRemoveItem(struct textureList** root, unsigned int id){
	if(root == NULL) return NULL;/*Jezeli lista nie istnieje zakoncz*/
	struct textureList* pointer = *root;
	if(pointer != NULL && pointer->id == id){/*Jezeli usuwanym elementem jest glowa*/
		if(pointer->pElement)destroyTexture(pointer->pElement);/*Jezeli tekstura istnieje
nalezy ja usunac*/
		*root = pointer->pNext;/*Mamy nowa glowe*/
		free(pointer);
		return *root;
	}
	/*Przeszukaj liste w poszukiwaniu elementu*/
	for(;pointer->pNext != NULL && pointer->pNext->id != id; pointer = pointer->pNext);
	if(pointer->pNext == NULL){/*Jezeli nie znaleziono*/
		return NULL;
	}
	else{/*W przeciwnym wypadku*/
		if(pointer->pNext->pElement)destroyTexture(pointer->pNext->pElement);/*Jezeli
```

```
tekstura istnieje nalezy ja usunac*/
                struct textureList* pNextElement =  pointer->pNext->pNext;/*Element 1 za
usuwanym*/
                free(pointer->pNext);
                pointer->pNext = pNextElement ? pNextElement : NULL;/*Jezeli za usunietym
elementem cos bylo, "wstaw" to w miejsce usunietego elementu*/

        }
        return pointer;/*Wskaznik na poprzedzajacy element*/
}
```

```
void freeList(struct textureList* root){
        struct textureList* pNext;
        while(root){/*Tak dlugo jak wskaznik na cos pokazuje*/
                pNext = root->pNext;/*Nastepny element*/
                if(root->pElement)destroyTexture(root->pElement);/*Jezeli tekstura istnieje, nalezy
ja usunac*/
                free(root);
                root = pNext;/*Ustaw wskaznik na nastepny element (jezeli lista sie skonczyla,
zostanie ustawiony NULL)*/
        }

}
```

Funkcje odpowiedzialne za przyciski (button.c)

```
struct button* createButton(int x, int y, enum mediaList defaultTexture,  enum mediaList
mouseoverTexture, enum mediaList pressedTexture, void (*pFunction)()){
        struct button* pButton = malloc(sizeof(struct button));
        pButton->position.x = x;
        pButton->position.y = y;
        pButton->bTexture[BUTTON_DEFAULT] = getTextureStr(defaultTexture);
        pButton->bTexture[BUTTON_MOUSEOVER] = getTextureStr(mouseoverTexture);
        pButton->bTexture[BUTTON_PRESSED] = getTextureStr(pressedTexture);
        pButton->width = pButton->bTexture[BUTTON_DEFAULT]->width;
        pButton->height = pButton->bTexture[BUTTON_DEFAULT]->height;
        pButton->state = BUTTON_DEFAULT;
        pButton->pFunction = (void(*)())pFunction;
        pButton->isVisible = true;
        return pButton;
}
```

```
void destroyButton(struct button* pointer){
        if(pointer != NULL){
                free(pointer);
        }
}
```

```
void renderButton(struct button* pButton){
        if(pButton && pButton->isVisible)/*Renderuj tylko wtedy, gdy przycisk jest widoczny*/
                renderTexture(pButton->bTexture[pButton->state], pButton->position.x, pButton-
>position.y);
}
```

```
void handeEvent(struct button* btn, SDL_Event* e){
```

```
        if(e->type == SDL_MOUSEMOTION || e->type == SDL_MOUSEBUTTONUP || e->type ==
SDL_MOUSEBUTTONDOWN){ /*Jezeli event dotyczy myszki*/
                int x,y;
                SDL_GetMouseState(&x, &y); /*pobierz pozycje kursora*/
                bool inside = true;/*Na poczatek przyjmujemy, ze kursor znajduje sie w przycisku*/
                if(x < btn->position.x){/*Jezeli jest na lewo od przycisku*/
                        inside = false;
                }
                else if(x > btn->position.x + btn->width){/*Jezeli jest na prawo od przycisku*/
                        inside = false;
                }
                else if(y < btn->position.y){/*Jezeli jest nad przyciskiem*/
                        inside = false;
                }
                else if(y > btn->position.y + btn->height){/*Jezeli jest pod przyciskiem*/
                        inside = false;
                }
                if(inside == false){/*Mysz poza przyciskiem*/
                        btn->state = BUTTON_DEFAULT;/*Przycisk w stanie domyslnym*/
                }
                else{/*Jezeli mysz jest w przycisku, sprawdz rodzaj eventu*/
                        switch(e->type){
                        case SDL_MOUSEMOTION: btn->state =
BUTTON_MOUSEOVER;break;/*mysz pojawila sie nad przyciskiem, zmiana statusu na mouseover*/
                        case SDL_MOUSEBUTTONDOWN: btn->state = BUTTON_MOUSEOVER;
if(btn->pFunction != NULL) btn->pFunction(); break;/*Mysz kliknela, zmiana statusu na pressed i
wykonanie funkcji przypisanej do przycisku*/
                        case SDL_MOUSEBUTTONUP: btn->state = BUTTON_MOUSEOVER;break;
/*Przycisk zostal zwolniony, wystarczy zmiana statusu*/
                        }
                }
        }
}
```

```
void changeButtonTexture(struct button* pButton, enum buttonState state, enum mediaList
texture){
        pButton->bTexture[state] = getTextureStr(texture);
}
```

```
void toggleButtonVisibility(struct button* pButton){
        pButton->isVisible = !(pButton->isVisible);
}
```

Funkcja ładująca tekstury podczas startu (mediaLoader.c)

```
bool loadMedia(){
        pTextureList = NULL;
        bool success = true;
        if(MEDIA_COUNT > 0){
                SDL_Surface* pTmpSur;/*Tymczasowa powierzchnia*/
                int currMedia = 0;
                while(currMedia < MEDIA_COUNT){
                        /*zaladuj tymczasowa powierzchnie*/
```

```
                        pTmpSur = IMG_Load(mediaLocations[currMedia]);
                        if(pTmpSur == NULL){
                                printf("Nie mozna zaladowac obrazu: %s %s\n",
mediaLocations[currMedia],IMG_GetError());
                                success = false;
                        }
                        else {
                                /*Utworz element na teksture*/
                                struct textureList* ptr =
listAddTexture(pTextureList,currMedia,malloc(sizeof(struct texture)));
                                ptr->pElement->height =pTmpSur->h;
                                ptr->pElement->width = pTmpSur->w;
                                ptr->pElement->lTexture =
SDL_CreateTextureFromSurface(getRenderer(),pTmpSur);/*utworz teksture z powierzchni */
                                if(ptr->pElement->lTexture == NULL){/*jezeli nie udalo sie*/
                                        printf("Nie mozna zaladowac tekstury: %s Blad: %s\n",
mediaLocations[currMedia],SDL_GetError());
                                        success = false;
                                }
                                if(currMedia == 0) pTextureList = ptr;/*pierwsza tekstura bedzie
glowa listy*/
                        }
                        SDL_FreeSurface(pTmpSur);/*to juz nie jest potrzebne*/
                        ++currMedia;
                }
                pFonts = malloc(FONT_COUNT*sizeof(TTF_Font*));
                while(currMedia-MEDIA_COUNT < FONT_COUNT){
                        pFonts[currMedia-MEDIA_COUNT] =
TTF_OpenFont(mediaLocations[currMedia],FONT_SIZE);/*Odczyt czcionki*/
                        if(pFonts[currMedia-MEDIA_COUNT] == NULL){/*Jezeli nie udalo sie
przypisac czcionki*/
                                printf("Nie mozna zaladowac czcionki %s %s\n",
mediaLocations[currMedia],TTF_GetError());
                                success = false;
                        }
                        ++currMedia;
                }
        }
        return success;/*Nie bylo bledow, wiec sukces*/
}
```

Przykładowe funkcje scen (sceneMenu.c)

```
static void init(){
        quit = malloc(sizeof(bool));
        *quit = false;
        /*alokacja pamieci na przyciski i pola tekstowe*/
        if(btnCount){
                pBtns = malloc(btnCount*sizeof(struct button*));
        }
        if(inputBoxCount){
                pInputBoxes = malloc(inputBoxCount*sizeof(struct inputBox*));
```

```
	}
	/*tworzenie przyciskow */
	pBtns[0] = createButton(250,125,IMG_SCENE_MENU_BTN_SINGLEGAME_DEFAULT,
IMG_SCENE_MENU_BTN_SINGLEGAME_MOUSEOVER,IMG_SCENE_MENU_BTN_SINGLEGAME_MOU
SEOVER, &singleBtnClicked);
	pBtns[1] = createButton(250,280,IMG_SCENE_MENU_BTN_MULTIPLAYER_DEFAULT,
IMG_SCENE_MENU_BTN_MULTIPLAYER_MOUSEOVER,IMG_SCENE_MENU_BTN_MULTIPLAYER_MO
USEOVER, &multiBtnClicked);
	pBtns[2] = createButton(250,435,IMG_SCENE_MENU_BTN_QUITGAME_DEFAULT,
IMG_SCENE_MENU_BTN_QUITGAME_MOUSEOVER,IMG_SCENE_MENU_BTN_QUITGAME_MOUSEO
VER, &quitBtnClicked);
	/*to juz wszystko */
	wasInitiated = true;
}
```

```
static void unInit(){
	/*trzeba inicjalizowac na nowo */
	wasInitiated = false;
	free(quit); /*zwolnienie pamieci*/
	int i;
	for(i=0; i < btnCount; ++i){
		destroyButton(pBtns[i]); /*usuwanie przyciskow */
	}
	for(i=0; i < inputBoxCount; ++i){
			destroyInputBox(pInputBoxes[i]); /*usuwanie pol tekstowych */
		}
	/*usuwanie tablic */
	free(pBtns);
	free(pInputBoxes);
}
```

```
static void handleEvents(SDL_Event *e){
	/*jezeli scena nie zostala zainicjalizowana, inicjalizuj */
	if(!wasInitiated){
		init();
	}
	/*jezeli okno ma zostac zamkniete, ustaw odpowiednia zmienna i nie przetwarzaj eventow */
	if(e->type == SDL_QUIT){
		*quit = true;
		return;
	}
	/*sprawdz obecny event na kazdym z przyciskow, o ile scena nie ulegnie zmianie
	 *  (moze nastapic zmiana sceny przez nacisniecie przycisku) */
	int processedButton;
	for(processedButton = 0;selectedScene == CURRENT_SCENE && processedButton <
btnCount; ++processedButton){
		handeEvent(pBtns[processedButton], e);
	}
}
```

```
static void renderScene()
{
	/*jezeli uzytkownik chce wyjsc, zmien scene na finalna i nie renderuj */
	if(*quit == true){
```

```
                selectScene(QUIT);
                return;
        }
        /*jezeli tekstura tla nie jest obecna, pobierz ja */
        if(currTex == NULL){
                currTex = getTexture(IMG_BACKGROUND);
        }
                SDL_RenderClear(getRenderer()); /*wyczysc okno*/
                SDL_RenderCopy(getRenderer(),currTex,NULL,NULL); /*wyrenderuj tlo*/
                /*wyrenderuj przyciski*/
                int processed;
                for(processed = 0; processed < btnCount; ++processed){
                        renderButton(pBtns[processed]);
                }
                /*wyrenderuj pola tekstowe*/
                for(processed = 0; processed < inputBoxCount; ++processed){
                                        inputBoxRender(pInputBoxes[processed]);
                }
                /*pokaz na ekranie*/
                SDL_RenderPresent(getRenderer());
}
```

Główna pętla programu (game.c)

```
int run(){
        if(inicjalizacja() == false){
                perror("Nie udalo sie zinicjalizowac.\n");
        }
        else{
                if(!loadMedia()){
                        perror("Nie udalo sie zaladowac mediow.\n");
                }
                else{
                        /*zaczynamy na scenie z menu glownym*/
                        selectScene(MAIN_MENU);
                        SDL_Event e;
                        /*tak dlugo, jak wybrana scena jest rozna od QUIT, obsluguj eventy i
renderuj scene*/
                        while(selectedScene != QUIT){
                                while(SDL_PollEvent(&e) != 0){
                                        if(selectedScene == QUIT) break;
                                        currScene->handleEvents(&e);
                                }
                                if(selectedScene== QUIT) break;
                                currScene->renderScene();
                        }
                        /*Petla gry zakonczona, zwalniamy zasoby*/
                        zamknij();
                }
        }
        return 0;
}
```

Funkcja zmieniająca sceny (scene.c)

```
void selectScene(enum sceneList scene){
	if(currScene != NULL && currScene->destroyScene != NULL)
	{
		currScene->destroyScene();/*usun z pamieci poprzednia scene*/
	}
	currScene = sceneList[scene];
	selectedScene = scene;
	if(scene != QUIT){
		if(currScene->initScene != NULL){
			currScene->initScene();/*jezeli jest taka mozliwosc, inicjalizuj
scene*/
		}
	}
}
```

Wszystkie typy wyliczeniowe, zmienne globalne i deklaracje funkcji obecnych w projekcie zostały zaprezentowane w specyfikacji wewnętrznej.

Testowanie

Program testowany był pod względem poprawności działania w następujących sytuacjach:

1. W trybie pojedynczego gracza
   - 1.1. Poprawnie rozegrana gra
   - 1.2. Próba wyboru już zajętego pola
   - 1.3. Próba cofnięcia ruchu
   - 1.4. Zapis stanu gry
     - 1.4.1.Zakończony powodzeniem
     - 1.4.2.Brak miejsca na dysku
     - 1.4.3.Brak uprawnień do zapisu
   - 1.5. Odczyt stanu gry
     - 1.5.1.Zakończony powodzeniem
     - 1.5.2.Brak uprawnień do odczytu
     - 1.5.3.Brak pliku ze stanem gry
     - 1.5.4.Uszkodzony plik ze stanem gry
   - 1.6. Tablica wyników
     - 1.6.1.Brak wpisu do tablicy
     - 1.6.2.Poprawnym wpis do tablicy
     - 1.6.3.Wpis zawierający znaki specjalne
2. W trybie dla wielu graczy
   - 2.1. Podczas nawiązywania połączenia
     - 2.1.1.Jako klient
       - 2.1.1.1. Poprawne połączenie
       - 2.1.1.2. Poprawne dane, nieudane połączenie

2.1.1.3. Niepoprawne dane(w tym zawierające znaki specjalne)
2.1.1.4. Wpisanie danych za pomocą klawiatury ekranowej
2.1.2.Jako serwer
2.1.2.1. Poprawne otworzenie portu
2.1.2.2. Brak uprawnień do otwarcia portu
2.1.2.3. Port aktualnie zajęty
2.1.3.Wielokrotna zmiana roli Klient <-> Serwer
2.2. Podczas właściwej gry
2.2.1.Poprawnie rozegrana gra
2.2.2.Próba wyboru już zajętego pola
2.2.3.Próba wyboru pola mimo braku zgody na ruch
2.2.4.Zerwanie połączenia przez jedną ze stron (przez protokół)
2.2.5.Zerwanie połączenia (z powodu problemów z siecią)
3. Wielokrotna zmiana między trybami gry

Dla każdej z powyższych sytuacji program zachował się stabilnie, informując użytkownika o zaistniałej niedogodności, kontynuując dalszą pracę.

Wycieki pamięci

Program był również testowany pod kątem wycieków pamięci programem Valgrind. Na sam koniec gra została przeniesiona na system Windows i sprawdzona w debuggerze Visual Studio (korzystając z dostępnych narzędzi Heap Profiler oraz Visual Leak Detector).

Valgrind

Program nie znalazł wycieków w kodzie napisanym przeze mnie, jednak wskazał na wycieki pamięci w funkcjach biblioteki xlib (służącej do połączenia z X serwerem). Krótko przeanalizowałem zawartość tych funkcji w oparciu o kod źródłowy biblioteki SDL i xlib dostępny w internecie. Znalazłem prawdopodobną przyczynę wycieków, jednak z uwagi na ogrom biblioteki xlib nie jestem w stanie stwierdzić, czy i gdzie ta pamięć jest zwalniana.
W środku jednej z funkcji znajduje się zapytanie o wskaźnik, a w przypadku jego nieotrzymania tworzony jest nowy obiekt, na którym następnie rysowany jest obraz. Mając na uwadze, że Xserwer obsługuje wszystkie okna na ekranie, tak naprawdę nie ma dowodu na to, że alokacja została dokonana podczas działania mojego programu. Faktem jest, że ilość pamięci, którą obejmuje „wyciek” nie zwiększa się podczas wykonywania programu mimo dokonywania setek operacji renderingu na sekundę. Mając to na uwadze, postanowiłem potraktować ten wyciek jako „false positive” wygenerowany przez Valgrinda.

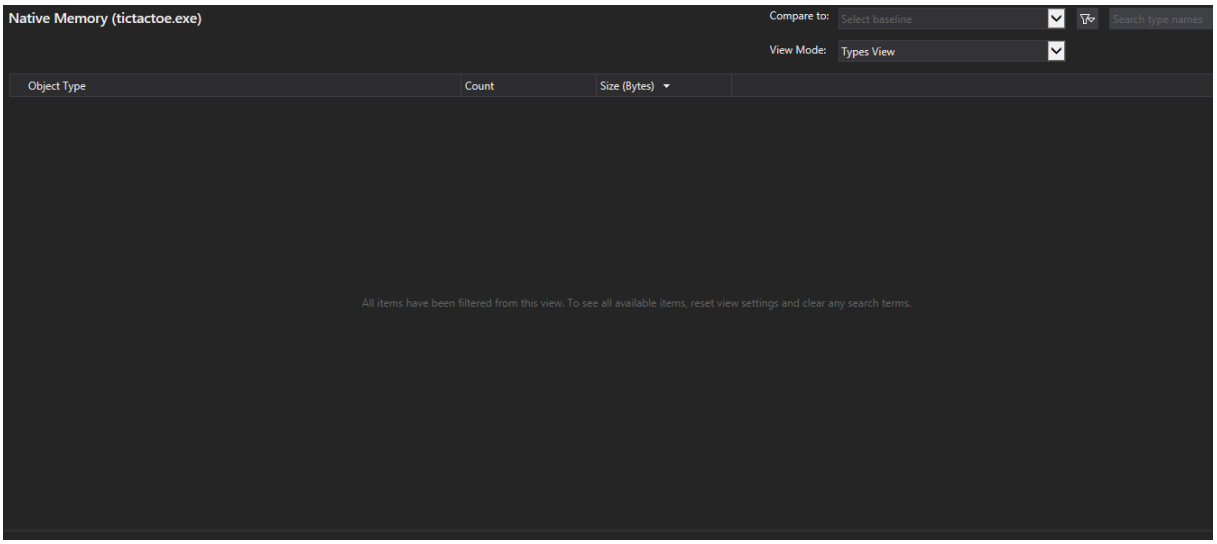
```
112 (8 direct, 104 indirect) bytes in 1 blocks are definitely lost in loss record 1,902 of 1,951
   at 0x48323BB: realloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
   by 0x4E84EFE: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E853F6: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E86B39: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8734D: _XlcCreateLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4EA5032: _XlcDefaultLoader (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8EF5D: _XOpenLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8F0B6: _XlcCurrentLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8F0F9: XSetLocaleModifiers (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x49077C4: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
   by 0x490E6AB: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
   by 0x48F4E81: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)

980 (68 direct, 912 indirect) bytes in 1 blocks are definitely lost in loss record 1,931 of 1,951
   at 0x48323BB: realloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
   by 0x4E84EFE: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E853F6: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E86B39: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8734D: _XlcCreateLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4EA9000: _XlcUtf8Loader (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8EF5D: _XOpenLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8F0B6: _XlcCurrentLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4E8F0F9: XSetLocaleModifiers (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
   by 0x4907879: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
   by 0x490E6AB: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
   by 0x48F4E81: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
```

Visual Studio

Test wtyczką Visual Leak Detector nie wykazał wycieków. Podobny efekt uzyskałem Heap Profilerem – podglądając zawartość pamięci:

```
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\tictactoe.exe'. Symbols loaded.
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\SDL2.dll'. Module was built without symbols.
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\SDL2_image.dll'. Module was built without symbols.
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\SDL2_ttf.dll'. Module was built without symbols.
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\SDL2_net.dll'. Module was built without symbols.
'tictactoe.exe' (Win32): Loaded 'C:\Program Files (x86)\Visual Leak Detector\bin
\Win32\vld_x86.dll'. Symbols loaded.
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\libfreetype-6.dll'. Module was built without symbols.
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\zlib1.dll'. Module was built without symbols.
Visual Leak Detector read settings from: C:\Program Files (x86)\Visual Leak
Detector\vld.ini
Visual Leak Detector Version 2.5.1 installed.
'tictactoe.exe' (Win32): Loaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\libpng16-16.dll'. Module was built without symbols.
The thread 0x4ff0 has exited with code 0 (0x0).
The thread 0x4f64 has exited with code 0 (0x0).
The thread 0x5008 has exited with code 0 (0x0).
The thread 0x3b88 has exited with code 0 (0x0).
The thread 0x49b8 has exited with code 0 (0x0).
'tictactoe.exe' (Win32): Unloaded 'C:\Windows\SysWOW64\d3d9.dll'
'tictactoe.exe' (Win32): Unloaded 'C:\Users\Jakub Klimek\Documents\Visual Studio
2015\Projects\tictactoe\Debug\libpng16-16.dll'
No memory leaks detected.
Visual Leak Detector is now exiting.
The program '[12536] tictactoe.exe' has exited with code 0 (0x0).
```



Wnioski

Praca nad programem była dla mnie przyjemnym doświadczeniem. Zawsze interesował mnie sposób, w jaki działają gry komputerowe, a dzięki temu projektowi miałem okazję spróbować swoich sił w pisaniu gier. Nauczyłem się dużo o zarządzaniu pamięcią oraz odkryłem, że pisząc program nie możemy być całkowicie pewni, jak nasz kod będzie się zachowywał na różnych platformach, pomimo użycia tej samej biblioteki. Program uruchomiony na Windowsie nie raportował żadnych wycieków pamięci. Valgrind z drugiej strony, wskazywał na wycieki w bibliotece Xlib, do których nie jestem przekonany, czy aby na pewno są wyciekami, czy może raczej pewną sztuczką zastosowaną przez programistę tworzącego tą bibliotekę.

Zrozumiałem również, że bez względu na to, jaki system operacyjny wybierzemy i jaką bibliotekę zastosujemy, możemy się spodziewać wycieków pamięci nie do końca powiązanych z naszym kodem. Z drugiej strony, sam system operacyjny pozwala na takie „kontrolowane niedbalstwa”, ponieważ po zamknięciu programu usuwa wszystkie jego pozostałości z pamięci. Przeglądając forum SDL można dojść do wniosku, że programiści tam wypowiadający się nie traktują wycieków pamięci poważnie tak długo, jak wyciek się nie powiększa, ponieważ „system operacyjny za nas posprząta”.