Zabrze, 25.01.2014

Laboratorium Programowania Komputerów

Temat:

Prosta gra platformowa na nieskończonej planszy.

Autor: Michał Rosenbeiger

Informatyka, Semestr IV, Grupa 1

Prowadzący:dr inż Roman Starosolski

1. Temat

Prosta gra platformowa na nieskończonej planszy.

2. Analiza, projektowanie

Do wykonania programu została użyta dodatkowa zewnętrzna biblioteka graficzna SFML w wersji 2.2.

Gra zarządzana jest przez klasę `GameManager` i jej metody. Klasa ta przechowuje stany gry, które są obiektami typu `State`, w kolejce dwustronnej deque używa kolejki zamiast stosu, gdyż ułatwia to kasowanie starczych stanów gry (nowe stany są wywoływane przez inne stany). Ponadto jest też klasa GraphicsManager, która zarządza działaniem tekstur. Przechowuje je w mapie segregowanej nazwami tekstur (jako, ze testur jest mało nie powinno być to problemem).

Jeżeli chodzi o stany gry, są dwa: Menu oraz Level, jako osobne, polimorficzne klasy. Menu wykorzystuje Obiekty klasy Button, które pozwalając na wyjście z programu lub wejście do gry.

Gra zawiera obiekt postaci, klasy Player, obiekty elementów interface'u, oraz dwustronne kolejki (deque) wskaźników na obiekty klasy Enemy (przeciwnicy) oraz obiekty klasy Attack (ataki). Klasa Player obsługuje klawisze jej dotyczące (sterowanie). Klasy Player i Enemy potrafią tworzyć obiekty klasy Attack.

Zmiany w obiektach są uwzględniane odpowiednimi metodami w pętli programu.

Zderzenia obiektów Player i Enemy z obiektami Attack obniżają ich punkty życia, ewentualnie zabijając. Gdy gracz ginie, uruchamia się stan menu w zmienionej wersji.

Klasy Attack, Enemy i Player dziedziczą z klasy MobileObject, która zawiera podstawowe pola i metody dla obiektów przeznaczonych do poruszania się, jak również uniwersalną dla wszystkich klas metodę rysującą. Ponadto klasy Enemy i Player dziedziczą wielobazowo - również po klasie DamagableObject, która zawiera pola i metody dotyczące punktów życia, odnoszące się do nich, ale nie do klasy Attack. Teoretycznie DamagabelObject mógłby być dziedziczony przez klasę nie będącej MobileObjectem, więc nie ustawiałbym tu dziedziczenia w stylu MobileObject<-DamagableObject<-Player/Enemy.

3. Specyfikacja zewnętrzna

3.a Obsługa programu

Sterowanie:

Strzałki ← → - ruch w lewo/prawo

Strzałka ↑ - skok (gdy postać jest na ziemi)

X - Strzał

Z - Atak z bliska

Duchy giną od każdego ataku. Zielone stworki giną od jednego ataku z bliska lub dwóch z odległości.

Zabicie przeciwnika daje 1 punkt.

Wraz ze wzrostem ilości punktów zmniejsza się ilość platform i zwiększa ilość przeciwników.

Ataki przeciwników zabierają 1/0 życia postaci. Spadek z platformy zabija postać i kończy grę.

Menu obsługiwane jest poprzez pojedyncze kliknięcia myszką.

3.b Komunikaty

`Missing file <nazwa pliku>`	W plikach gry brakuje obrazka lub czcionki.

4. Specyfikacja Wewnetrzna

4.0 Biblioteki zewnetrzne

W programie użyto zewnętrznej biblioteki graficznej SFML w wersji 2.2.

4.a Obiekty

AnimationHandler

Klasa zajmuje się animacjami spritera postaci gracza. Przechowuje klatkę animacji i modyfikuje ją zgodnie w wytycznymi. Wykorzystywany wewnątrz klasy `Player`.

Pola prywatne w klasie `AnimationHandler`	
`defValue`	Zmienna typu `int`. Wartość domyślna klatki animacji (wraca do niej gdy zakończy się tymczasowa animacja).
`tempValue`	Zmienna typu `int`. Wartość chwilowa klatki animacji.
`timer`	Zmienna typu `float`. Licznik trwania klatki chwilowej

Metody w klasie `AnimationHandler`
`void setTemp(AnimationState value, double duration);`
Ustawia wartość klatki na `value` na czas `time`
`void setDefault(AnimationState value);`
Ustawia klatkę domyślną na wartość `value`
`AnimationState checkFrame(double dt);`
Update'uje animację bazując na różnicy czasu od ostatniego update'u `dt`, zwraca aktualną klatkę
`AnimationHandler();`
Konstruktor domyślny klasy

Attack

Klasa bazowa klas `Projectile` oraz `Hit`. Sama wywoływana jest jedynie w ramach list zawierających te elementy. Sama jest klasą potomną klasy `MobileObject`. Zawiera elementy sprawdzające czy atak jest przyjazny dla postaci gracza oraz sprawdzenia czy nie jest on poza mapą.

Pola prywatne w klasie `Attack`	
`isFriendly`	Zmienna typu `bool`. Informacja czy atak został wykonany przez gracza (wartość `true`) czy przeciwnika (wartość `false`).

Metody w klasie `Attack`
`virtual void update(std::vector<std::deque<BGElement*>> &map, const float dt) = 0`
Czysto wirtualna metoda wykorzystywana przez klasy potomne do update'owania. Zmienna `map` przechowuje plan planszy (patrz map w klasie Level), `dt` przechowuje czas od ostatniego update'u.
`bool isOutside()`
Sprawdza czy atak znajduje się poza planszą. Jeżeli tak, zwraca `true`.
`bool checkFriendly();`
Sprawdza czy atak został wykonany przez gracza (zwraca `true`) czy przeciwnika.(zwraca `false`)
`Attack(GameManager * game, int type, bool faceLeft, bool friendly, int x, int y)`
Konstruktor klasy. Zmienna `game` przechowuje wskaźnik na grę, `type` to typ ataku, `faceLeft` informuje w którą stronę względem postaci wykonany jest atak (`true` jeżeli w lewo), `friendly` informuje czy atak został wykonany przez gracza (`true`) czy przeciwnika (`false`), `x` i `y` to koordynaty położenia lewego górnego rogu hitboxa ataku.

BGElement

Klasa zawierająca elementy planszy. Dzięki użyciu zmiennej typu i odpowiednich stałych jest ona uniwersalna na wszelkie obiekty.

Pola prywatne w klasie `BGElement`	
`hitbox`	Obiekt typu `Hitbox`. Hitbox pola planszy.
`id`	Zmienna typu wyliczeniowego `BgId`. Rodzaj pola planszy.
`pos`	Obiekt klasy `sf::Vector2i`. Pozycja pola na planszy (w pikselach).
`texture`	Obiekt klasy `sf::Texture`. Przechowuje teksturę.

Metody w klasie `BGElement`
`Hitbox checkHB();`
Zwraca `Hitbox` pola planszy.
`void draw(GameManager* game)`
Rysuje pole planszy. Wskaźnik `game` wskazuje na grę.
`BgId checkID()`
Zwraca rodzaj pola planszy.
`BGElement()`
Domyslny konstruktor. Nadaje polu planszy Id = 0;
`BGElement(GameManager* game, BgId id, int cox, int coy)`
Konstruktor pola planszy. Wskaźnik `game` wskazuje na grę. `id` zawiera rodzaj pola, `cox`, `coy` to koordynaty na mapie planszy (w polach).

Button

Klasa przechowująca przyciski w menu. Dzięki użyciu zmiennej typu i odpowiednich stałych jest ona uniwersalna na wszelkie obiekty tego typu.

Pola prywatne w klasie `Button`	
`texpos`	Obiekt klasy `sf::Vector2i`. Pozycja tekstury w oknie.
`hitbox`	Obiekt klasy `Hitbox`. Hitbox przycisku.
`texture`	Obiekt klasy `sf::Texture`. Tekstura przycisku.
`action`	Zmienna typu wyliczeniowego `ButtonAction`. Akcja przycisku.

Metody w klasie `Button`
`bool CheckCollision(sf::Vector2f mpos)`
Zwraca `true` gdy punkt `mpos` znajduje się wewnątrz przycisku.
`void draw(GameManager * game)`
Rysuje przycisk. Wskaźnik `game` wskazuje na grę.
`Button(ButtonAction action, const std::string & texname, GameManager * game)`
Konstruktor przycisku. Prametrami są: `action` - akcja przycisku, `texname` - nazwa tekstury oraz `game` - wskaźnik na grę.
`const std::string ActionName()`
Zwraca akcję przycisku.

DamagableObject

Klasa po której dziedziczą wielobazowo klasy `Player` oraz `Enemy`. Jednak w przeciwieństwie do klasy MobileObject, nie dziedziczy po niej klasa `Attack`. Jej metody zajmują się zarządzaniem punktami życia postaci i przeciwników.

Pola prywatne w klasie `DamagableObject`	
`health`	Zmienna typu `int.` Aktualne punty życia obiektu.

Metody w klasie `DamagableObject`
`bool damage(int x)`
Zmniejsza ilość punktów życia obiektu o `x`.
`int checkHealth()`
Zwraca ilość punktów życia obiektu.

Enemy

Klasa przechowująca przeciwników. Dzięki użyciu zmiennej typu i odpowiednich stałych jest ona uniwersalna na wszelkie obiekty tego typu, więc nie było potrzeby tworzenia dodatkowych klas potomnych.

Pola prywatne w klasie `Enemy`	
`cooldown`	Zmienna typu `float`. Czas oczekiwania na kolejny atak przeciwnika.

Metody w klasie `Enemy`
`void update(GameManager * game, std::vector<std::deque<BGElement*>> &map, std::vector<Attack*> &attacks, const float dt)`
Update'uje obiekt. Wskaźnik `game` wskazuje na grę, map zawiera planszę, `attacks` zawiera wektor wskaźników na ataki, do którego ewentualne nowe ataki mają być przypisane, `dt` wskazuje różnicę czasu.
`bool checkHit(Attack * attack)`
Sprawdza kolizję obiektu z atakiem, jeżeli wystąpiła, zadaje mu obrażenia i zwraca `true`, `attacks` zawiera wektor wskaźników na ataki.
`bool isOutside()`
Zwraca `true` jeżeli obiekt jest poza planszą.
`void move(float x)`
Przemieszcza obiekt o `x` pikseli w prawo.
`Enemy(GameManager * game, float x, float y, int type)`
Konstruktor przeciwnika. Wskaźnik `game` wskazuje na grę, `x` i `y` to pozycja przeciwnika na planszy w pikselach, `type` to typ przeciwnika.

GameManager

Główna klasa gry zarządzająca stanami i zawierająca główną pętlę gry.

Pola prywatne w klasie `GameManager`	
`states`	Kontener `std::deque` obiektów klasy `State`. Przechowuje stany gry.

Pola publiczne w klasie `GameManager`	
`window`	Obiekt klasy `sf::Window`. Okno gry.
`textures`	Obiekt klasy `GraphicsManager`. Tekstury.

Metody w klasie `Game`
`void pushState(State* state)`
Umieszcza w deque'u `states` obiekt klasy `State` wskazywany przez wskaźnik `state`. jeżeli w states jest 3 lub więc stanów funkcja usuwa najstarszy.
`void popState()`
Usuwa obiekt na szczycie deque'a `states` jeżeli nie jest on pusty.
`State* checkState()`
Zwraca wskaźnik na obiekt na szczycie deque'a `states`. Jeżeli deque jest pusty zwraca `nullptr`.
`void loop()`
Petla główna gry.
`GameManager()`
Konstrutor klasy, tworzy okno gry.

GraphicManager

Klasa zarządzająca teksturami - ładuje je gdy jej obiekt jest utworzony i pozwala innym klasom i ich metodom na używanie jednokrotnie zapisanych tekstur przechowywanych w kontenerze mapy.

Pola prywatne w klasie `GraphicManager`	
`textures`	Kontener klasy `std::map`. Przechowuje tekstury sortowane nazwami.

Metody w klasie `GraphicManager`
`void load()`
Ładuje tekstury.
`void loadTexture(const std::string& texturename, const std::string &filename)`
Dodaje do mapy `textures` teksturę o nazwie `texturename` z pliku `filename`.
`sf::Texture& getRef(const std::string& name)`
Zwraca wskaźnik na teksturę o nazwie `name`.
`GraphicsManager()`
Konstruktor klasy. Ładuje tekstury z plików.

Hit

Polimorficzna klasa potomna klasy Attack. Podobnie jak polimorficzna klasa Projectile jest na tyle uniwersalna, by nie było potrzeby rozbijania podziały ataków na podklasy. Ta klasa różni się od niej faktem, że wykorzystuje mechanikę zanikania po określonym czasie.

Pola prywatne w klasie `Hit`	
`timer`	Zmienna typu `float`. Przechowuje pozostały czas trwania ataku.

Metody w klasie `Hit`
`void update(std::vector<std::deque<BGElement*>> &map, const float time)`
Update'uje atak. `map` przechowuje planszę, `dt` przechowuje czas od ostatniego update'u.
`bool expired()`
Sprawdza czy czas trwania ataku zakończył się, jeśli tak, zwraca `true`.
`Hit(GameManager * game, int type, bool faceLeft, bool friendly, int x, int y)`
Konstruktor klasy. Zmienna `game` przechowuje wskaźnik na grę, `type` to typ ataku, `faceLeft` informuje w którą stronę względem postaci wykonany jest atak (`true` jeżeli w lewo), `friendly` informuje czy atak został wykonany przez gracza (`true`) czy przeciwnika (`false`), `x` i `y` to koordynaty położenia lewego górnego rogu hitboxa ataku.

Hitbox

Klasa zarządzająca działaniem hitboxów. Jest wykorzystywana przez większość klas: Button, BGElement oraz wszystkie pochodne klasy MobileObject.

Pola prywatne w klasie `Hitbox`	
`hitbox`	Zmienna typu `sf::FloatRect`. Prostokąt będący hitboxem.

Metody w klasie `Hitbox`
`bool checkCollision(Hitbox thitbox)`
Sprawdza czy wystapiła kolizja hitboxu z hitboxem `thitbox`
`bool checkCollision(sf::Vector2f point)`
Sprawdza czy punkt `point` zawiera się wewnątrz hiboxa.
`bool checkOnTop(Hitbox thitbox)`
Sprawdza czy hitbox znajduje się dokładnie nad (i styka się) z hitboxem `thitbox`.
`void move(float x, float y)`
Przesuwa hitbox o koordynaty `x`, `y`.
`sf::Vector2f position()`
Zwraca `sf::Vector2f` z pozycją lewego górnego rogu hitboxa.
`sf::FloatRect checkBox()`
Funkcja zwraca prostokąt hitboxa.
`Hitbox(int x, int y, int sizex, int sizey)`
Konstruktor klasy. Parametry `x`, `y` to położenie lewego górnego rogu hitboxa, `sizex` i `sizey` to wymiary w pionie i poziomie hitboxa.

HPBar

Klasa zarządzająca paskiem życia postaci w lewym górnym rogu.

Pola prywatne w klasie `HPBar`	
`position`	Zmienna typu `sf::Vector2f`. Pozycja paska życia.
`texture`	Zmiena typu `sf::Texture`. Przechowuje tekstury.

Metody w klasie `Game`
`void draw(GameManager * game, sf::Vector2f pos, int size)`
Poprawia położenie paska życia na `pos` i rysuje pasek dla postać której zostało `size` punktów życia, wskaźnik `game` wskazuje na grę.
`HPBar(float x, float y, GameManager * game)`
Konstruktor klasy. Parametry `x`, `y` to położenie paska życia, wskaźnik `game` wskazuje na grę.

Level

Klasa obsługująca poziom (jedyny) na którym odbywa się gra. Dziedziczy po wirtualnej klasie `State`.

Pola prywatne w klasie `Level`	
`levelView`	Obiekt typu `sf::View`. Widok poziomu.
`defmap`	Kontener klasy `std::vector` kontenerów klasy `std::deque` zmiennych klasy int. Plan planszy.
`map`	Kontener klasy `std::vector` kontenerów klasy `std::deque` wskaźników na obiekty klasy `BGElement` . Plansza.
`playchar`	Obiekt klasy `Player`. Postać gracza.
`enemies`	Kontener klasy `std::vector` wskaźników na obiekty klasy Enemy. Przeciwnicy.
`attacks`	Kontener klasy `std::vector` wskaźników na obiekty klasy `Attack`. Aktywne ataki.
`hpbar`	Obiekt klasy `HPBar`. Interface paska życia.
`scorebar`	Obiekt klasy `ScoreBar`. Interface paska punktacji.
`points`	Zmienna typu `int`. Aktualna ilość punktów.

Metody w klasie `Level`
`virtual void draw()`
Rysuje planszę.
`virtual void update(const float dt)`
Update'uje plansze, `dt` to czas od ostatniego update'u.
`virtual void input()`
Sprawdza czy okno zostało zamknięte.
`Level(GameManager* game)`
Konstruktor klasy. Parametr `game` wskazuje na grę.
`void retry()`
Przełącza na stan Menu z wartością retry=true w razie przegranej.
`void generate()`
Generuje nową kolumnę plany planszy algorytmem i umieszcza ją na końcu defmap. Usuwa pierwszą kolumnę defmap.
`void genMap()`
Odświeża map generując ją jeszcze raz na podstawie defmap.
`void spawn(int x, int y, int type)`
Generuje przeciwnika typu `type` na polu planszy odpowiadającym koordynatom `x`, `y` (w polach) na planie planszy.

Menu

Klasa obsługująca menu gry oraz ekran śmierci (który działa na praktycznie tej samej zasadzie). Dziedziczy po wirtualnej klasie `State`.

Pola prywatne w klasie `Menu`	
`retry`	Zmienna typu `bool`. Zmienia menu z wersji normalnej na wersję wyświetlającą się po przegranej.
`menuView`	Obiekt klasy `sf::View`. Widok menu.
`buttons`	Kontener `std::vector` wskaźników na obiekty Button. Przechowuje przyciski.

Metody w klasie `Game`
`virtual void draw()`
Rysuje menu.
`virtual void input()`
Sprawdza akcje myszki.
`Menu(GameManager* game, bool retry)`
Konstruktor menu, `game` jest wskaźnikiem na grę, `retry==true` jeżeli menu ma ukazać się w formie retry po przegranej grze.
`void start()`
Rozpoczyna grę.

MobileObject

Klasa bazowa klas `Attack`, `Enemy`, `Player`. Zajmuje się ich teksturami, rysowaniem, dostrajaniem położenia do przemieszczającej się planszy, śledzenia zmian klatek.

Pola prywatne w klasie `MobileObject`	
`texture`	Obiekt typu `sf::Texture`. Przechowuje teksturę.
`hitbox`	Obiekt typu `Hitbox`. Hitbox obiektu.
`position`	Obiekt typu `sf::Vector2f`. Pozycja sprite'a obiektu (lewy górny róg)
`spriteSize`	Obiekt typu `sf::Vector2i`. Wymiary sprite'a obiektu.
`faceLeft`	Zmienna typu `bool`. Przyjmuje wartość true jeżeli obiekt zwrócony jest w lewo.
`frame`	Zmienna typu `int`. Aktualna klatka animacji.
`type`	Zmienna typu `int`. Typ obiektu. Pozwala na wykorzystywania tego samo kodu dla zróżnicowanych obiektów.

Metody w klasie `MobileObject`
`virtual void corrPos()`
Przemieszcza obiekt o długość jednego pola planszy.
`Hitbox getHB()`
Zwraca Hitobox obiektu.
`void draw(GameManager * game)`
Rysuje obiekt. Wskaźnik `game` wskazuje na grę.

Player

Klasa postaci gracza. Zarządza jej ruchem, również pod względem obsługi klawiatury, atakami, kolizjami. Dziedziczy wielobazowo po klasach `MobileObject` (tekstury, hitbox, położenie) i `DamagableObject` (obsługa punktów życia).

Pola prywatne w klasie `Player`	
`grounded`	Zmienna typu `bool`. Przyjmuje wartość `true` gdy postać stoi na podłożu.
`acc`	Zmienna typu `float`. Aktualne przyśpieszenie postaci w pionie.
`cooldown`	Zmienna typu `float`. Czas jaki musi minąć by postać mogła ponownie zaatakować.
`animation`	Obiekt klasy `AnimationHandler`. Obiekt zajmujący się animacją.

Metody w klasie `Player`
`void update(GameManager * game, std::vector<std::deque<BGElement*>> &map, std::vector<Attack*> &attacks, const float dt)`
Update'uje postać, sprawdza input z klawiatury. Parametrami są `game`, czyli wskaźnik na grę; `map`, czyli plansza; `attacks`, czyli vector ataków, do którego mają być dopisane ataki wykonane przez postać; `dt`, czyli czas od ostatniego update'u.
`void move(std::vector<std::deque<BGElement*>> &map, float x, float y)`
Przemieszcza hitbox postaci po planszy. Parametr `map` to plansza, a `x` i `y` to wartości o jakie przesuwa się postać.
`bool checkHit(Attack * attack)`
Sprawdza kolizję obiektu z atakiem, jeżeli wystąpiła, zadaje mu obrażenia i zwraca `true`.
`void fixPos()`
Dopasowuje pozycję sprite'a do pozycji hitboxa.
`sf::Vector2f GetPosition()`
Zwraca pozycję sprite'a postaci (lewy górny róg).
`Player(GameManager * game, float x, float y)`
Konstruktor klasy. Parametr `game` to wskaźnik na grę, a parametry `x` i `y` to pozycja sprite'a postaci (lewy górny róg).

Projectile

Polimorficzna klasa potomna klasy Attack. Podobnie jak polimorficzna klasa Hit jest na tyle uniwersalna, by nie było potrzeby rozbijania podziały ataków na podklasy. Klasa różni się od niej faktem, że wykorzystuje mechanikę przemieszczania się.

Pola prywatne w klasie `Projectile`	
`speed`	Zmienna typu `int`. Prędkość przemieszczania się pocisku.

Metody w klasie `Game`
`void update(std::vector<std::deque<BGElement*>> &map, const float dt)`
Update'uje atak. `map` przechowuje planszę, `dt` przechowuje czas od ostatniego update'u.
`void move(float x)`
Przemieszcza obiekt o `x` pikseli w lewo.
`Projectile(GameManager * game, int type, bool faceLeft, bool friendly, int x, int y, int speed)`
Konstruktor klasy. Zmienna `game` przechowuje wskaźnik na grę, `type` to typ ataku, `faceLeft` informuje w którą stronę względem postaci wykonany jest atak (`true` jeżeli w lewo), `friendly` informuje czy atak został wykonany przez gracza (`true`) czy przeciwnika (`false`), `x` i `y` to koordynaty położenia lewego górnego rogu hitboxa ataku, speed to prędkość poruszania się ataku.

ScoreBar

Klasa przechowująca pasek punktacji w prawym górnym rogu.

Pola prywatne w klasie `ScoreBar`	
`position`	Obiekt klasy `sf::Vector2f`. Pozycja paska punktacji (lewy górny róg).
`font`	Obiekt klasy `sf::Font`. Przechowuje czcionkę.

Metody w klasie `Game`
`void draw(GameManager * game, sf::Vector2f pos, int score)`
Rysuje pasek punktacji na pozycji `pos`, wyświetlający wartość `score`. Parametr `game` to wskaźnik na grę.
`ScoreBar(float x, float y, GameManager * game)`
Konstruktor klasy, tworzący pasek punktów na pozycji o koordynatach `x`, `y` (lewy górny róg). Parametr `game` to wskaźnik na grę.

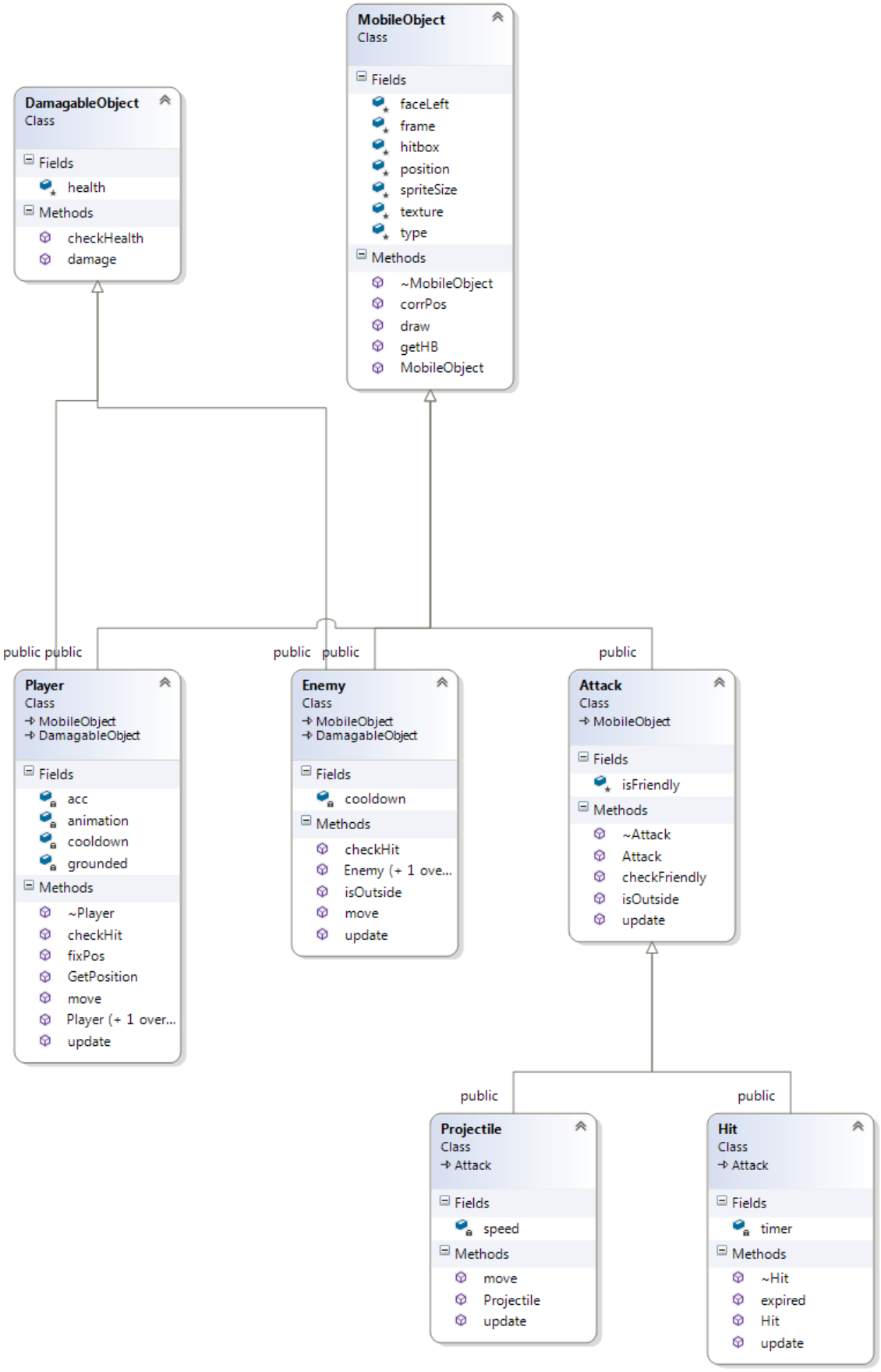
State

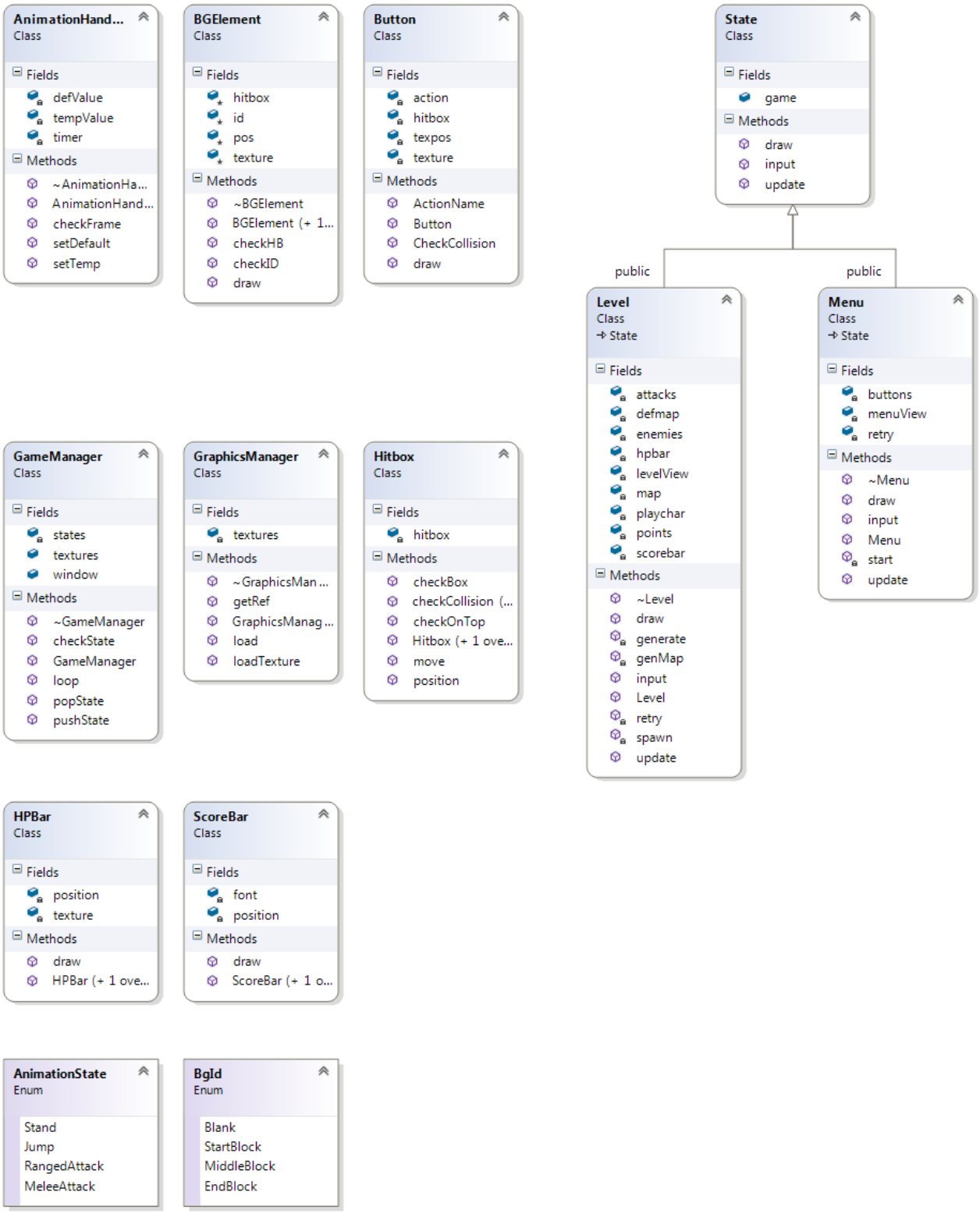
Wirtualna klasa bazowa, po której dziedziczą polimorficzne klasy `Menu` i `Level`. Zawiera elementy tych klas wykorzystywane przez klasę `GameManager`, w której znajduje się deque obiektów tej klasy.

Pola prywatne w klasie `State`	
`game`	Wskaźnik na typ GameManager. Wskaźnik na grę.

Metody w klasie `Game`
`virtual void draw()`
Czysto wirtualna funkcja. Rysuje stan w oknie.
`virtual void update(const float dt)`
Czysto wirtualna funkcja. Odświeża stan. Parametr `dt` to różnica czasu od ostatniego odświeżenia.
`virtual void input()`
Czysto wirtualna funkcja. Sprawdza eventy.

4.b Diagram hierarchii klas.




AnimationHand...
Class
Fields
defValue
tempValue
timer
Methods
~AnimationHa...
AnimationHand...
checkFrame
setDefault
setTemp
BGElement
Class
Fields
hitbox
id
pos
texture
Methods
~BGElement
BGElement (+ 1...
checkHB
checkID
draw
Button
Class
Fields
action
hitbox
texpos
texture
Methods
ActionName
Button
CheckCollision
draw
State
Class
Fields
game
Methods
draw
input
update
public
public
Level
Class
State
Fields
attacks
defmap
enemies
hpbar
levelView
map
playchar
points
scorebar
Methods
~Level
draw
generate
genMap
input
Level
retry
spawn
update
Menu
Class
State
Fields
buttons
menuView
retry
Methods
~Menu
draw
input
Menu
start
update
GameManager
Class
Fields
states
textures
window
Methods
~GameManager
checkState
GameManager
loop
popState
pushState
GraphicsManager
Class
Fields
textures
Methods
~GraphicsMan ...
getRef
GraphicsManag ...
load
loadTexture
Hitbox
Class
Fields
hitbox
Methods
checkBox
checkCollision (...
checkOnTop
Hitbox (+ 1 ove...
move
position
HPBar
Class
Fields
position
texture
Methods
draw
HPBar (+ 1 ove...
ScoreBar
Class
Fields
font
position
Methods
draw
ScoreBar (+ 1 o...
AnimationState
Enum
Stand
Jump
RangedAttack
MeleeAttack
BgId
Enum
Blank
StartBlock
MiddleBlock
EndBlock

4.c Zmienne globalne i typy wyliczeniowe.

Stałe	
TILE_W	Stała typu int. Szerokość pola planszy
TILE_H	Stała typu int. Wysokość pola planszy
MAP_W	Stała typu int. Szerokość planszy
MAP_H	Stała typu int. Wysokość planszy
GRAVITY	Stała typu int. Grawitacja
BASE_ACC	Stała typu int. Domyślne przyśpieszenie postaci
VIEW_W	Stała typu int. Szerokość widoku
VIEW_H	Stała typu int. Wysokość widoku
P_START_X	Stała typu int. Początkowe położenie lewej krawędzi spritera postaci.
P_START_Y	Stała typu int. Początkowe położenie górnej krawędzi spritera postaci.
P_SPRITE_W	Stała typu int. Wysokość spritera postaci.
P_SPRITE_H	Stała typu int. Szerokość spritera postaci.
P_HEALTH	Stała typu int. Początkowe HP postaci.
P_SPEED	Stała typu int. Prędkość poruszania się postaci
P_HB_DIFF_W	Stała typu int. Różnica między hitboxem postaci a lewym brzegiem spritera.
P_HB_DIFF_H	Stała typu int. Różnica między hitboxem postaci a lewym brzegiem spritera.
P_HB_SIZE_W	Stała typu int. Wysokość hitboxa postaci.
P_HB_SIZE_H	Stała typu int. Szerokość hitboxa postaci.
P_A_POS_X[]	Tablica stałych typu int. Pozycja ataków względem położenia postaci w poziomie.
P_A_POS_Y[]	Tablica stałych typu int. Pozycja ataków względem położenia postaci w pionie.
P_ATT_CD	Stała typu float. Cooldown ataku postaci.
P_PROJECTILE_SPEED	Stała typu int. Prędkość pocisków postaci
P_A_TYPE[]	Tablica stałych typu int. Typy ataków używane przez postać
E_SPRITE_W	Stała typu int. Wysokość spritera przeciwnika
E_SPRITE_H	Stała typu int. Szerokość spritera przeciwnika
E_HEALTH[]	Tablica stałych typu int. Początkowe HP przeciwników
E_SPEED[]	Tablica stałych typu int. Prędkość poruszania się przeciwników
E_HB_DIFF_W[]	Tablica stałych typu int. Różnice między hitboxami przeciwników a lewymi brzegami ich spriterów.
E_HB_DIFF_H[]	Tablica stałych typu int. Różnice między hitboxami przeciwników a górnymi brzegami ich spriterów.
E_HB_SIZE_W[]	Tablica stałych typu int. Wysokości hitboxów przeciwników.
E_HB_SIZE_H[]	Tablica stałych typu int. Szerokości hitboxów przeciwników.
E_HIT_DMG_TKN[]	Tablica stałych typu int. Odporności przeciwników na ataki klasy Hit
E_PROJECTILE_DMG_TKN[]	Tablica stałych typu int. Odporności przeciwników na ataki klasy

	Projectile
E_A_POS_X[]	Tablica stałych typu int. Pozycje ataków względem położenia przeciwników w poziomie.
E_A_POS_Y[]	Tablica stałych typu int. Pozycje ataków względem położenia przeciwników w pionie.
E_ATT_CD[]	Tablica stałych typu float. Cooldowny ataków przeciwników
E_PROJECTILE_SPEED[]	Tablica stałych typu int. Prędkość pocisków przeciwników
E_MELEE[]	Tablica stałych typu bool. true jeżeli przeciwnik używa ataku typu Hit
E_RANGED[]	Tablica stałych typu bool. true jeżeli przeciwnik używa ataku typu Projectile
E_A_TYPE[]	Tablica stałych typu int. Typy ataków używanych przez przeciwników.
A_SPRITE_W	Stała typu int. Wysokość spritera ataku
A_SPRITE_H	Stała typu int. Szerokość spritera ataku
A_HB_SIZE_W[]	Tablica stałych typu int. Wysokości hitboxów ataków.
A_HB_SIZE_H[]	Tablica stałych typu int. Szerokości hitboxów ataków.
E_HB_DIFF_W[]	Tablica stałych typu int. Różnice między hitboxami ataków a lewymi brzegami ich spriterów.
E_HB_DIFF_H[]	Tablica stałych typu int. Różnice między hitboxami ataków a górnymi brzegami ich spriterów.
A_TIMER[]	Tablica stałych typu float. Czasy trwania ataków.
BG_HB_DIFF_W[]	Tablica stałych typu int. Różnice między hitboxami elementów tła a lewymi brzegami ich spriterów.
BG_HB_DIFF_H[]	Tablica stałych typu int. Różnice między hitboxami elementów tła a górnymi brzegami ich spriterów.
BG_HB_SIZE_W[]	Tablica stałych typu int. Wysokości hitboxów elementów tła.
BG_HB_SIZE_H[]	Tablica stałych typu int. Szerokości hitboxów elementów tła.
B_POS_X[]	Tablica stałych typu int. Pozycja przycisku w poziomie.
B_POS_Y[]	Tablica stałych typu int. Pozycja przycisku w pionie.
B_SIZE_X	Stała typu int. Szerokość przycisku.
B_SIZE_Y	Stała typu int. Wysokość przycisku.
HPBAR_W	Stała typu int. Szerokość paska życia.
HPBAR_H	Stała typu int. Wysokość paska życia.
HPBAR_X	Stała typu int. Położenie paska życia w poziomie.
HPBAR_Y	Stała typu int. Położenie paska życia w pionie.
ScBAR_X	Stała typu int. Położenie paska punktacji w poziomie.
ScBAR_Y	Stała typu int. Położenie paska punktacji w pionie.

Typy wyliczeniowe	

enum ButtonAction
{
Start, Exit, Retry
};
	Reprezentuje możliwe funkcje przycisków menu

enum BgId
{
Blank, StartBlock, MiddleBlock, EndBlock
};
	Reprezentuje możliwe rodzaje obiektów BGElement (rodzaje elementów planszy).

4.d Użyte techniki

1. Klasy polimorficzne.

Attack <- Projectile
<- Hit

2. Dziedziczenie wielobazowe.

Klasy Enemy i Player dziedziczą zarówno po DamageableObject, jak i po MobileObject.

3. RTTI

Rozróżnianie rodzajów ataków, przy sprawdzaniu obrażeń dla przeciwników. jak również do uwzględniania nieco innych zachowań klas Hit i Projectile jak na przykład gdy Hit może uderzyć w kilku przeciwników.

4. Wyjątki

Wyrzucanie błędów przy źle załadowanych plikach.

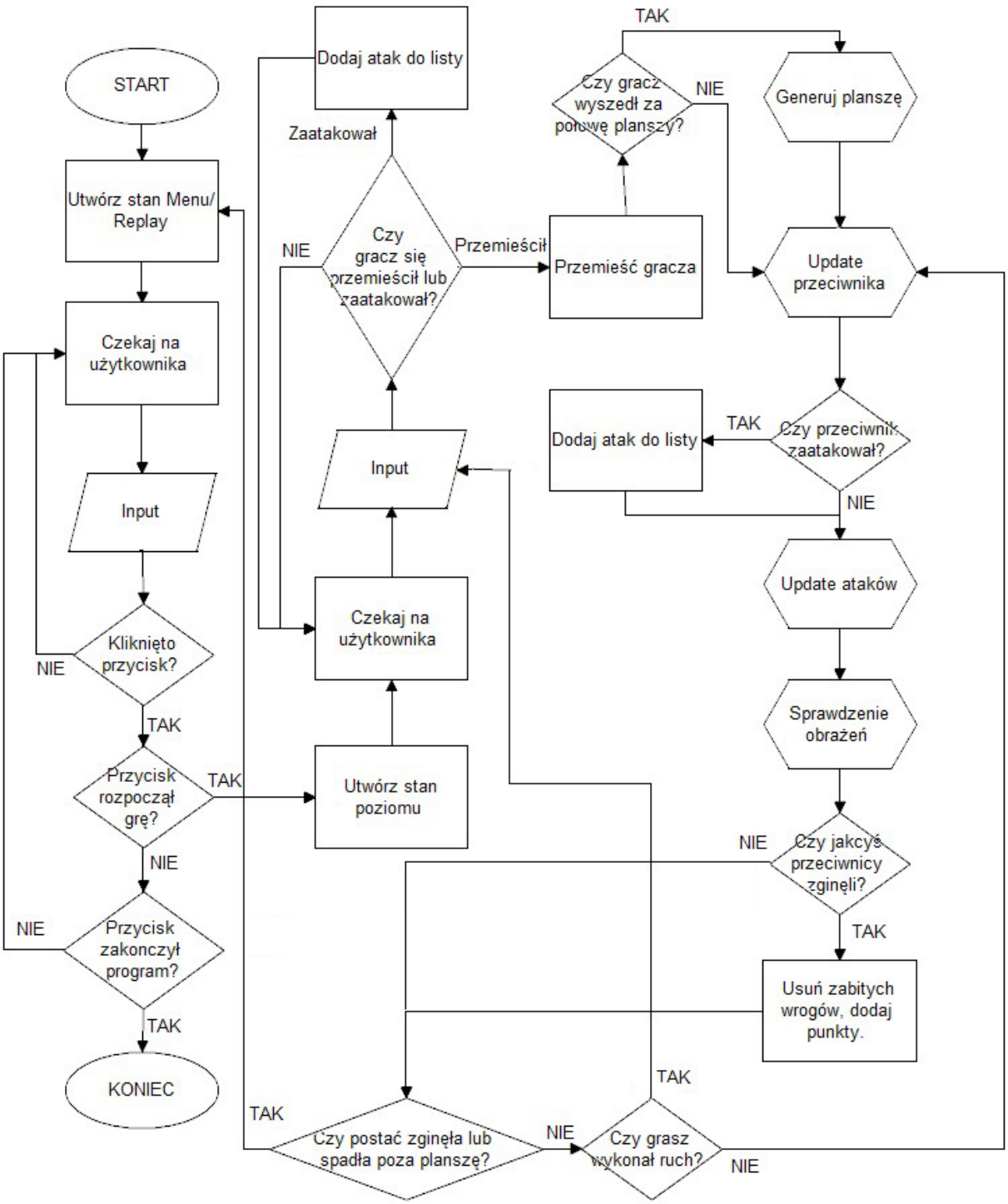
5. Kontenery STL

W wielu miejscach, głównie wektory i deque.

6. Iteratory i Algorytmy STL

Przeważnie w metodach klasy Level, która zawiera dużo kontenerów. Również w różnych destruktorach klas.

4.e Ogólny schemat działania programu.



5. Testowanie i Uruchamianie

Przykładowe znalezione błędy (w przypadkowej kolejności):

1. Czcionka tablicy wyników nie chciała przestać się wygładzić.

Rozwiązanie: okazało się, że wyłączałem wygładzanie tekstury dla złego rozmiaru czcionki.

2. Ataki klasy Hit które miały znikać nie znikały.

Rozwiązanie: Program sprawdzał typeid(*itattack), co zwracało *Attack, a nie typeid(**attack)

3. Platformy generowały się z jednopolowymi przerwami między sobą

Rozwiązanie: W niektórych case'ach w generatorze planszy brakowało breaków, przez co czasem przypadkiem generowały się dodatkowe puste pola.

4. Przeciwnicy i postać znika poza planszą.

Błąd pojawiał się w wyniku breakowania przy debugowaniu - gdy czas pomiędzy rozkazami był bardzo duży postaci wybiegały bo czas dt w instrukcjach ruchu był zbyt duży.

Błąd nie powinien sprawiać problemów przy normalnym użytkowaniu. Ponadto jest teoretycznie niegroźny, bo przeciwnicy i tak są niszczeni gdy znajdują się poza planszą, a postać ginie gdy spadnie za bardzo w dół (lub poleci w górę poza zakres.

5. "Wspinanie" się po platformie.

Fragment kodu sprawdzający upadki na platformę był za mało wybredny i można było na nię wejść doskakując tylko połowy.

Rozwiązanie: Ograniczenie hitboxów platform. Uniemożliwienie zatrzymywania się na platformie przy locie w górę. Ogólne zmiany działania interakcji postać-platfroma (np. dodanie metody `checkOnTop()`)

6. Dziwne zachowania spriterów przeciwników.

Rozwiązanie: centrum spritera było źle ustawione.

7. Niewidzialne/nietykalne/niewidzialno-nietykalne ataki

Rozwiązanie: W stałych były przecinki zamiast kropek.