Gliwice, 2011

Laboratorium Programowania Komputerów

Temat:
Aputapu – gra logiczna

Autor: **Michał Krupiński**
Informatyka sem. 4 gr. 2
Prowadzący: **Artur Migas**
Ścieżka: U:\Michał Krupiński G2 S1\

I. Pierwotne założenia i opis aplikacji

Niniejszy rozdział powstał przed rozpoczęciem pisania kodu – stanowi wstępną specyfikację pokazywaną **na początku semestru**.

1. Opis aplikacji

Realizowaną aplikacją w ramach projektu będzie gra logiczna, bazująca na idei gry „Bloppy". Wstępne założenia zakładają grę typu single-player z możliwością utworzenia trybu multi-player. Pojedyncze zagranie nie powinno zajmować wiele czasu, gra powinna być szybka oraz intuicyjna, dzięki czemu będzie wciągająca.

1.2 Mechanizm działania

Gra bazowała będzie na planszy, której rozmiary zwiększane będą z każdym kolejnym poziomem.

Każda plansza zawierać będzie określoną ilość kulek, których ilość będzie się zwiększać wraz z kolejnym poziomem.

Każda kulka ma określony kolor. Ilość kolorów na danej planszy zwiększa się wraz z kolejnym poziomem.

1.3 Zasady gry

- Gra polega na aktywowaniu wszystkich kulek znajdujących się na planszy.
- Aktywacja kulek następuje poprzez zaznaczenie części obszaru planszy, który w swych narożnikach zawiera kulki tego samego koloru.
- Po aktywacji kulki zmieniają losowo swoje kolory, z możliwościa zachowania koloru poprzedniego.
- Raz aktywowane kulki mogą być aktywowane ponownie, bez naliczania za ich aktywację punktów.
- Przydział ilości punktów za aktywację kulek zależny jest od ilości kulek nieaktywowanych w zaznaczonym poprawnie obszarze.
- Na ukończenie gry gracz ma wyznaczony czas.
- Za ukończenie każdego poziomu gracz otrzymuje punkty bonusowe.

1.4 Tryb multiplayer (opcjonalnie)

- Każdy gracz prowadzi niezależną od siebie turę.
- Pierwotne poziomy plansz są identyczne dla każdego gracza.
- Grę wygrywa gracz z większą ilością punktów.
- W przypadku, gdy którykolwiek z graczy przejdzie wszystkie poziomy czas gry zostaje ustawiony na 30 sekund, w przypadku gdy wynosił więcej.
- Po upływie wyznaczonego czasu gra dobiega końca.
- Zwycięski gracz otrzymuje dodatnie punkty do statystyk gry, natomiast gracz przegrany – ujemne.

2. Planowane do użycia biblioteki

Do realizacji projektu planowane jest głównie użycie biblioteki graficznej „OpenGL”. Ponadto w celu usprawnienia projektowania użyta zostanie biblioteka „Boost”. Podstawowymi bibliotekami będą biblioteki standardowe języka C++.

3. Wstępna faza projektu

W obecnej, wstępnej fazie projektu, głównym celem jest zapoznanie się z biblioteka graficzną „OpenGL”. Kolejnym celem jest poznanie protokołu sieciowego niezbędnego do rozwoju trybu multiplayer gry.
Końcowym etapem wstępnej fazy projektu będzie zaprojektowanie szczegółowej specyfikacji wewnętrznej w oparciu nabytą w tym etapie wiedzę.

4. Wnioski

W obecnej fazie projektu możliwe jest jedynie ustalenie podstawowych cech projektu. Niemożliwe jest utworzenie dokładniejszej specyfikacji bez wymaganej znajomości biblioteki graficznej, na której głownie opierać się będzie realizowana gra.

II. Analiza i projektowanie

Założenia z poprzedniego rozdziału zostały w pełni zrealizowane. Tryb multiplayer jako opcjonalny nie został utworzony, aczkolwiek bazując na pierwotnych założeniach aplikacja została przygotowana z myślą o dalszym rozwoju, w tym w szczególności do rozszerzenia o tryb multiplayer.

Planowana do użycia biblioteka OpenGL została użyta za pośrednictwem obiektowej biblioteki SFML, która bazuje na OpenGL. Biblioteka Boost została użyta, aczkolwiek w ramach refraktoryzacji kodu zdecydowałem się na jej wykluczenie, z uwagi na niewielką rolę pełniąca w projekcie. Ponadto oczywiście użyte zostały standardowe biblioteki języka C++.

Przed przystąpieniem do pisania programu przyjąłem konwencję stosowania angielskiego nazewnictwa oraz komentarzy w celu udostępnienia kodu programu jak najszerszej liczbie programistów.

Na podstawie wcześniej zdefiniowanej specyfikacji wstępnej sprecyzowałem wymagania dotyczące występujących w projekcie struktur danych:

1. Z uwagi na rodzaj aplikacji jakim jest gra, powinna ona zawierać klasy dotyczące reprezentacji graficznej – takie jak animacja czy przycisk.

2. Ponieważ elementy graficzne pojawiające się w grze, takie jak obrazy, są kosztowne pamięciowo i czasowo – należy utworzyć menadżery zasobów, za pomocą których w programie będzie występować ich właściwa organizacja.

3. Z pewnością występować będą klasy takie jak *Ball* (reprezentacja pojedynczej kulki), *Selection* (reprezentacja zaznaczenia na planszy), *Level* (reprezentacja poziomu gry) oraz kolekcja *BallsCollection* (reprezentacja wielu kulek na planszy).

Po sprecyzowaniu wyżej wymienionych wymagań zdecydowałem się na zastosowanie wzorca projektowego *Model-View-Controller*. W moim programie *kontroler* pełni typową rolę odbioru i przetworzenia danych wejściowych od użytkownika (głównie myszy), po czym zmienia stan *modelu* i odświeża *widok* oraz przełącza sterowanie na inny *kontroler* przy spełnieniu określonych wymagań. Warstwa *modelu* reprezentuje logikę biznesową aplikacji – i tak znajdują się tutaj m.in. klasy opisane w punkcjie trzecim poprzedniego zagadnienia. Z kolei *widok* reprezentuje konkretny sposób wyświetlania danych, pobieranych z *modelów*. Należy wspomnieć, że może występować wiele *widoków* operujących na tych samych danych, a jedynie inaczej je reprezentujących. W moim przypadku większość wyświetlanych modeli dziedziczy po klasie abstrakcyjnej *sf::Drawable* i posiada wewnętrznie opisany sposób rysowania. Rola *widoku* sprowadza się do odpowiedniego rozmieszczenia w renderowanym oknie obiektów oraz wywołania metod rysujących i odświeżających wyświetlanie danych obiektów – są to metody *Draw(const sf::RenderTarget&)* oraz *Update()*. Poszczególne warstwy zostały w projekcie odpowiednio pogrupowane.

W projekcie utworzona została rozbudowana warstwa graficzna aplikacji. Utworzenie spójnej warstwy graficznej zajęło bardzo dużo czasu. Występują klasy takie jak:

Animation – na podstawie wektora wskaźników do obrazów potrafi wyświetlić odpowiednią ich sekwencję w ustalonych przez programistę klatkach na sekundę. Posiada wiele metod pozwalających m.in. na określenie powtarzalności animacji, przerwy pomiędzy kolejnymi odtworzeniami lub też odwrotnego odtwarzania – więcej na ten temat w specyfikacji wewnętrznej.

Button – abstrakcyjna klasa zawierająca szkielet przycisku reagującego na konkretne zdarzenia. Konkretyzacją jest *AnimatedButton*, który dziedziczy po *Button* i *Animation* dzięki czemu w reakcji na określone zdarzenia odtwarza określone animacje.

W projekcie występują również menadżery zasobów. Są to:

ResourceManager – abstrakcyjna klasa bazowa dla konkretnych zasobów. Zawiera ona metody takie jak wyszukiwanie oraz zwalnianie zasobów.Zawiera czysto wirtualną metodę wczytywania zasobu.

ImageManager, ButtonManager, FontManager – konkretyzacje *ResourceManager,* występują jako zmienne globalne w programie będące magazynem potrzebnych zasobów.

W związku z powyższymi menadżerami utworzyłem współpracujące z nimi konfiguracje. Mają one na celu umieszczenie w menadżerach potrzebnych danych i zapamiętaniu, które dane zostały dodane. Następnie gdy dane przestały być potrzebne zwalniają z mendżerów określone zasoby. W ten sposób można by było potraktować menadżery jako *singleton'y*, aczkolwiek nadal nie jest to konieczne – gdyż można nie tworzyć konkretnych konfiguracji i wczytywać w konkretnym kontrolerze dane do lokalnego menadżera. Jest to jednak moim zdaniem niełądnie wyglądające w kodzie rozwiązanie i stąd też utworzyłem konfiguracje, ładujące „ładnie" dane wewnętrznie i zwalniane z globalnego menadżera przy destrukcji konfiguracji. W programie występują następujące klasy konfiguracji:

Configuration – abstrakcyjna klasa bazowa zawierająca metody zwalniania zasobów oraz odpowiednie pola. Zawiera czysto wirtualną metodę *Init()*, inicjalizującą konkretnych wczytywanie danych do menadżerów.

GameplayImages – wczytuje do menadżera obrazów potrzebne do gry obrazy.

BallsConfiguration – wczytuje do menadżera przycisków potrzebne do gry kulki, dziedziczące po przycisku. Ta konfiguracja wczytuje do menadżera przycisków odpowiednie kulki w sposób intuicyjny dla programisty – całą kulkę wystarczy później wczytać z menadżera przycisków za pomocą identyfikatorów „BlueBall", „RedBall" itp. Jest to moim zdaniem bardzo przejrzyste rozwiązanie.

GameplayConfiguration – łączy w sobie dwie powyższe klasy, stanowiąc w pełni wystarczalną konfigurację rozgrywki. Odpowiednio wywołuje wpierw *GameplayImages*, a następnie *BallsConfiguration*, wymagające obrazów wczytanych do menadżera przez *GameplayImages*. Przy destrukcji obiektu klasy *GameplayConfiguration* wywoływane są destruktory klas *GameplayImages* oraz *GameplayImages*, zwalniające wykorzystywane zasoby.

Najważniejsze klasy w warstwie modelu:

Ball – występująca w grze kulka, dziedziczy po *AnimatedButton*. Zawiera w sobie typ wyliczeniowy rozróżniający konkretne kolory. Ponadto zawiera *sf::String*, który widoczny jest po aktywowaniu kulki. Dokładniejszy opis w specyfikacji wewnętrznej.

BallsCollection – mapa - dwuwymiarowa tablica *Ball*, zrealizowana na *std::vector*. Zawiera metody operujące na wielu kulkach oraz m.in. odpowiednio je rozmieszcza na ekranie w stosunku do określonej ilości.

Selection – selekcja dziedzicząca po *sf::Shape*. Operuje na *BallsCollection* i w zależności od początkowo zaznaczonej kulki i aktualnym położeniu kursora rysuje prostokąt, będący obszarem zaznaczenia.

Level – zawiera m.in. wygenerowane cztery mapy, czyli *BallsCollection* oraz *Selection* na obecną mapę. Mapy generowane są na początku, dzięki czemu przy rozbudowywaniu na tryb multiplayer każdemu graczowi w łatwy sposób przypisze się identyczne mapy początkowe. Ponadto dzięki zawieraniu w sobie *Selection* można będzie w łatwy sposób monitorować przebieg rozgrywki u każdego z graczy, wyświetlając na ekranie każdego gracza na bieżąco cała rozgrywkę innych graczy – jest to jedna z moich wizji trybu multiplayer, stąd też w ten sposób zaprojektowana została klasa *Level*.

Countdown – klasa realizująca zadanie odliczania czasu pozostałego do końca gry.

Ponadto, jak już na wstępie wspomniano, występują *widoki* oraz *kontrolery*. Są to:
GameplayController z jednym widokiem *GameplayView* – obsługa przebiegu rozgrywki.
GameoverController z jednym widokiem *GameoverView* – obsługa zakończenia gry.

III. Specyfikacja zewnętrzna

Zgodnie z założeniami gra „Aputapu” powinna być szybka i intuicyjna. Niniejszy rozdział stanowi instrukcję użytkownika.

1. Zasady gry

- Gra polega na aktywowaniu wszystkich kulek znajdujących się na planszy.
- Aktywacja kulek następuje poprzez zaznaczenie części obszaru planszy, który w swych narożnikach zawiera kulki tego samego koloru.
- Po aktywacji kulki zmieniają losowo swoje kolory, z możliwościa zachowania koloru poprzedniego.
- Raz aktywowane kulki mogą być aktywowane ponownie, bez naliczania za ich aktywację punktów.
- Przydział ilości punktów za aktywację kulek zależny jest od ilości kulek nieaktywowanych w zaznaczonym poprawnie obszarze.
- Na ukończenie gry gracz ma wyznaczony czas – 180 sekund.
- Za ukończenie każdego poziomu gracz otrzymuje punkty bonusowe – 500*(wygrany poziom)

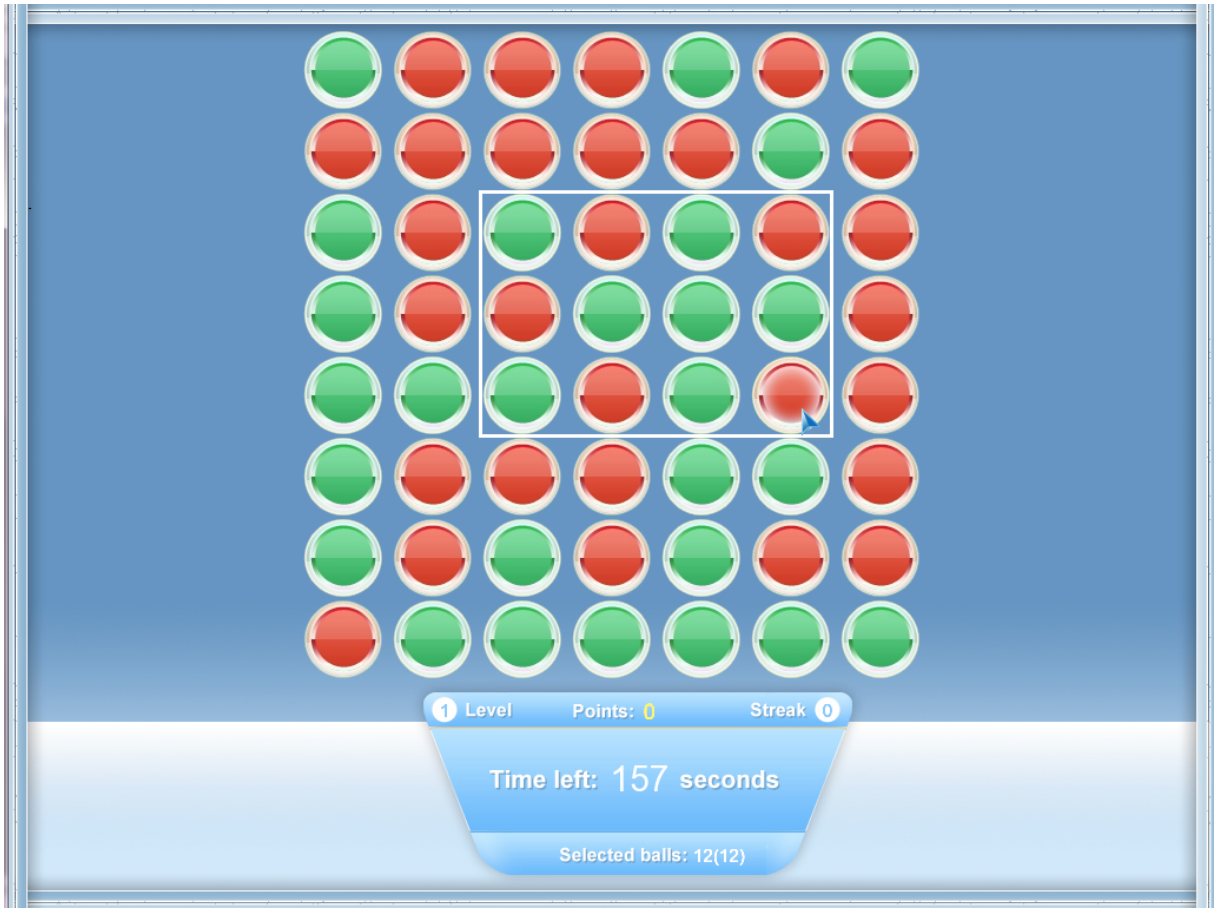
2. Interfejs użytkownika

1 – Aktualny poziom, na którym znajduje się gracz.
2 – Aktualna liczba punktów.

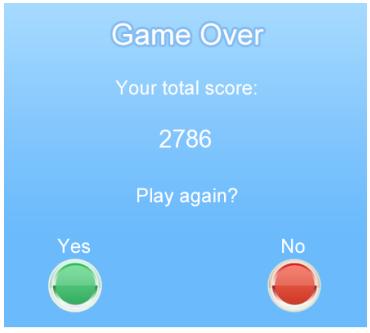
3 – Największa ilość aktywowanych za jednym zaznaczeniem kulek.
4 – Pozostały czas gry.
5 – Liczba zaznaczonych kulek. W nawiasie podana jest liczba nieaktywowanych kulek w obrębie zaznaczenia.

3. Obsługa programu

- W celu zaznaczenia kulek należy najechać na daną kulkę, wcisnąć prawy przycisk myszy i trzymając przeciągnąć nad odpowiednią kulkę. Aby zatwierdzić zaznaczenie należy puścić prawy przycisk myszy.



- W celu wyjścia z gry należy kliknąć przycisk „X” w prawym górnym rogu ekranu, lub kliknąć na czerwony przycisk z napisem „No” przy zapytaniu o ponowną grę.
- W celu chęci ponownego zagrania należy przy zapytaniu o ponowną grę nacisnąć zielony przycisk z napisem „Yes”.

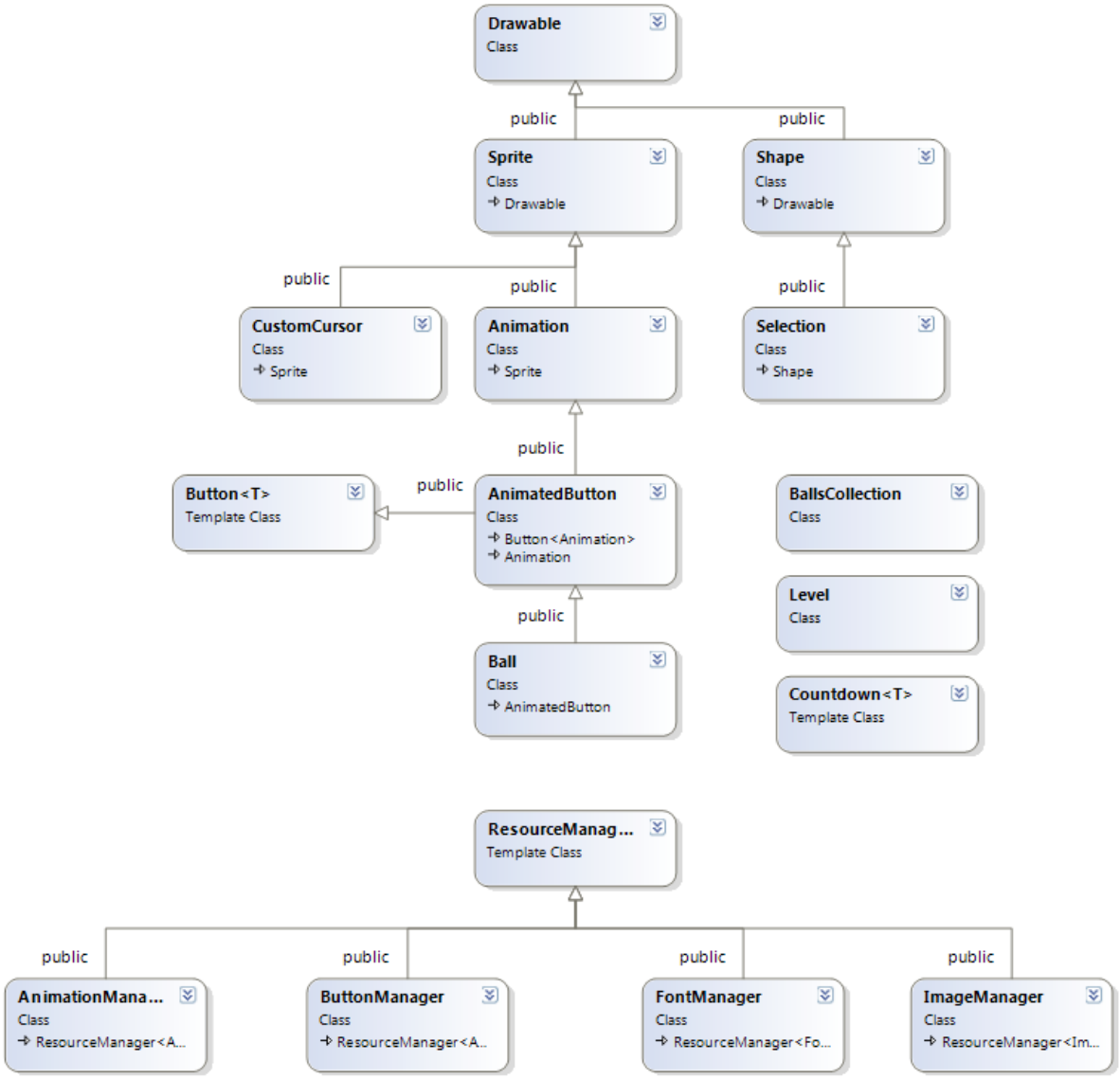


IV. Specyfikacja wewnętrzna

Program zorientowany jest obiektowo. Na wstępie przedstawiam diagram klas występujących w programie:

1. Diagramy klas

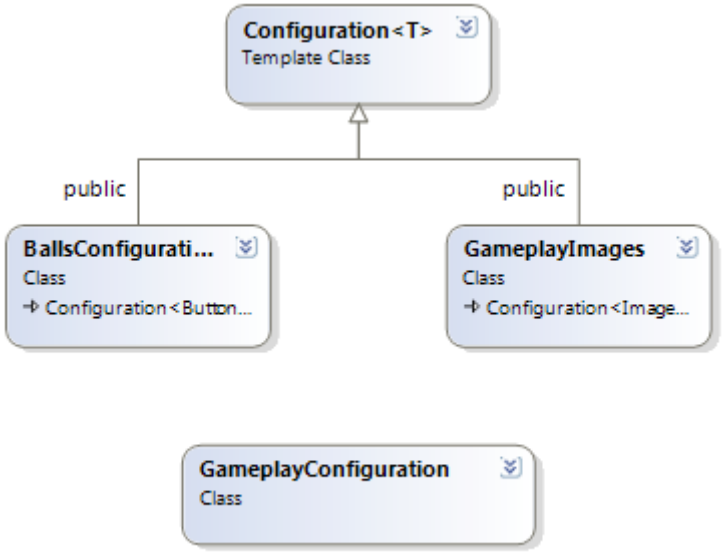
Warstwa *modelu*:



Warstwa *widoku*:

Warstwa *kontrolera*:

Klasy *konfiguracji*:



2. Zmienne globalne

Nazwa	`gImageManager`
Typ	`ImageManager`
Znaczenie	Globalny menadżer obrazów.

Nazwa	`gAnimationManager`
Typ	`AnimationManager`
Znaczenie	Globalny menadżer animacji.

Nazwa	`gButtonManager`
Typ	`ButtonManager`
Znaczenie	Globalny menadżer przycisków.

Nazwa	`gFontManager`
Typ	`FontManager`
Znaczenie	Globalny menadżer czcionek.

3. Klasy

Nazwa	Animation [namespace sf]
Rola	Klasa odpowiedzialna za prawidłowe wyświetlanie sekwencji obrazów tworzących animację.
Kod	`class Animation : public sf::Sprite` `{` `private:` `    std::vector<sf::Image*> Images;` `    std::vector<unsigned int> ImageLengths;` `    sf::Clock Clock;` `protected:` `    bool isCooldown;` `    bool isReverse;` `    bool isLooping;` `    double CooldownTime;` `    double CurrentFrame;` `    double CurrentImageFrame;` `    int CurrentImage;` `    unsigned int FramesPerSecond;` `public:` `    Animation();` `    Animation(unsigned int setFps, double setCooldown, bool setPlay);` `    virtual ~Animation();` `    void Play();` `    void Stop();` `    void Update();` `    void UpdateReverse();` `    void Draw(RenderTarget& Target);` `    void SetCooldown(double newCooldown);` `    void SetFPS(unsigned int fps);` `    void SetReverse(bool state);` `    void SetLoop(bool state);` `    void Reverse();` `    void Cooldown();` `    virtual void Reset();` `    bool AddFrame(sf::Image* NewFrame, unsigned int Length);` `    bool InsertFrame(sf::Image* NewFrame, unsigned int Length, unsigned int Position);` `    bool DeleteFrame(unsigned int Position);` `    bool isPlaying;` `    void Clear();` `    sf::Image* GetFrame(unsigned int Position);` `    Animation& operator=(const Animation& toCpy);` `};`
Pola	`std::vector<sf::Image*> Images; // Wektor wskaźników na obrazy` `std::vector<unsigned int> ImageLengths; // Wektor długości w klatkach poszczególnych obrazów.` `sf::Clock Clock; // Wewnętrzny zegar. Służy do obsługi fps animacji.` `bool isCooldown; // Określa, czy animacja jest w stanie spoczynku.` `bool isReverse;  // Określa, czy animacja jest odwrócona.` `bool isLooping;  // Określa, czy animacja się powtarza po ukończeniu.` `bool isPlaying;  // Określa, czy animacja jest w trakcie odtwarzania.` `double CooldownTime; // Określa czas spoczynku.` `double CurrentFrame; // Określa obecną klatkę.` `double CurrentImageFrame; // Określa ilość klatek na obraz.` `int CurrentImage; // Określa obecny obraz.` `unsigned int FramesPerSecond; // Określa FPS animacji.`

Metody	Animation(unsigned int setFps, double setCooldown, bool setPlay); Konstruktor. Argumenty: unsigned int setFps – fps animacji. double setCooldown – czas spoczynku po ukończonej animacji. bool setPlay – true, jeśli animacja ma od razu rozpocząć odtwarzanie. Wartości domyślne: isCooldown = false; isReverse = false; isLooping = true;
	void Play(); Ustawia wartość isPlaying na true.
	void Update(); Na podstawie wartości zmiennych klasy odpowiednio przechodzi do następnego obrazu bądź zostaje przy obecnym, zwiększając wartość CurrentFrame o wartość (Clock.GetElapsedTime() * FramesPerSecond).
	void UpdateReverse(); Tak samo jak wyżej, ale dla animacji odwróconej.
	void Reverse(); Ustawia wartość isReverse na przeciwną.
	void Cooldown(); Obsługa spoczynku. W przypadku isCooldown == true metoda Update() wywołuje metodę Cooldown() przez określony czas spoczynku.
	virtual void Reset(); Ustawienie wartości początkowych.
	bool AddFrame(sf::Image* NewFrame, unsigned int Length); Dodanie obrazu na ostatnią pozycję z określoną liczbą klatek w jakich ma być wyświetlany. Zwykle jedna klatka na obraz. Zwraca true w przypadku powodzenia.
	bool InsertFrame(sf::Image* NewFrame, unsigned int Length, unsigned int Position); Dodanie obrazu na konkretną pozycję z określoną liczbą klatek w jakich ma być wyświetlany i pozycją. Zwraca true w przypadku powodzenia
	bool DeleteFrame(unsigned int Position); Usunięcie klatki z konkretnej pozycji. Zwraca true w przypadku powodzenia.
	void Clear(); Usunięcie wszystkich klatek.
	sf::Image* GetFrame(unsigned int Position); Uzyskanie obrazu o określonej w animacji pozycji. Zwraca obraz z podanej pozycji.
	Animation& operator=(const Animation& toCpy); Operator przypisania. Zwraca wynik operacji.

Nazwa	Button [namespace sf]

Rola	Klasa abstrakcyjna bazowa dla obsługi przycisku.
Kod	

```
class Button
{
protected:
	enum State
	{
		isIdle,
		isButtonPressed,
		isButtonReleased,
		isMouseMovedOn,
		isMouseMovedOut
	};
	T my_IdleResource;
	T my_MouseButtonPressedResource;
	T my_MouseButtonReleasedResource;
	T my_MouseMovedOnResource;
	T my_MouseMovedOutResource;
	bool my_GotIdleResource;
	bool my_GotMouseButtonPressedResource;
	bool my_GotMouseButtonReleasedResource;
	bool my_GotMouseMovedOnResource;
	bool my_GotMouseMovedOutResource;
	State my_State;

public:
	Button() {}
	virtual ~Button() {}
	/*/ Setting up states /*/
	void SetIdleState(const T& newState)
	{
	my_IdleResource = newState;
	my_GotIdleResource = true;
	}
	void SetMouseButtonPressedState(const T& newState)
	{
	my_MouseButtonPressedResource = newState;
	my_GotMouseMovedOnResource = true;
	}
	void SetMouseButtonReleasedState(const T& newState)
	{
	my_MouseButtonReleasedResource = newState;
	my_GotMouseButtonReleasedResource = true;
	}
	void SetMouseMovedOnState(const T& newState)
	{
	my_MouseMovedOnResource = newState;
	my_GotMouseMovedOnResource = true;
	}
	void SetMouseMovedOutState(const T& newState)
	{
	my_MouseMovedOutResource = newState;
	my_GotMouseMovedOutResource = true;
	}
	/*/ Handling mouse events /*/
	virtual bool ButtonPressed(Event::MouseButtonEvent) = 0;
	virtual bool ButtonReleased(Event::MouseButtonEvent) = 0;
	virtual bool MouseMovedOn(Event::MouseMoveEvent) = 0;
	virtual bool MouseMovedOut(Event::MouseMoveEvent) = 0;
};
```

Pola	

```
enum State  // Typ wyliczeniowy stanu.
{
```

	isIdle, isButtonPressed, isButtonReleased, isMouseMovedOn, isMouseMovedOut }; T my_IdleResource; // Zasób w stanie aktywności pasywnej. T my_MouseButtonPressedResource; // Zasób w stanie przycisku myszy. T my_MouseButtonReleasedResource; // Zasób w stanie zwolnienia przycisku myszy. T my_MouseMovedOnResource; // Zasób w stanie wejścia myszy na przycisk. T my_MouseMovedOutResource; // Zasób w stanie zejścia myszy z przycisku bool my_GotIdleResource; // Określa, czy posiada zasób pasywny. bool my_GotMouseButtonPressedResource; // Określa, czy posiada zasób w stanie przycisku myszy. bool my_GotMouseButtonReleasedResource; // Określa, czy posiada zasób w stanie zwolnienia przycisku myszy. bool my_GotMouseMovedOnResource; // Określa, czy posiada zasób w stanie najechania myszy. bool my_GotMouseMovedOutResource; // Określa, czy posiada zasób w stanie zjechania przycisku myszy z przycisku. State my_State; // Określa obecny stan przycisku.
Metody	Poszczególne metody Set ustawiają wewętrzne wartości pól.
	Metody czysto wirtualne przyjmują za parametr konkretne informacje dotyczące zdarzenia potrzebne do weryfikacji zajścia zdarzenia na sprawdzanym przycisku. Są to `Event::MouseButtonEvent` do sprawdzenia koordynatów naciśnięcia przycisku myszy oraz `Event::MouseMoveEvent` do sprawdzenia koordynatów poruszenia kursora.

Nazwa	`AnimatedButton: public Button<Animation>, public Animation [namespace sf]`
Rola	Klasa reprezentująca animowany przycisk.
Kod	class AnimatedButton: public Button<Animation>, public Animation { bool my_GotReversedMouseMoveAnimation; // MouseOut = Reversed MouseOn public: AnimatedButton(); virtual ~AnimatedButton() {} virtual void Reset(); /*/ Handling mouse events /*/ virtual bool ButtonPressed(Event::MouseButtonEvent); virtual bool ButtonReleased(Event::MouseButtonEvent); virtual bool MouseMovedOn(Event::MouseMoveEvent); virtual bool MouseMovedOut(Event::MouseMoveEvent); /*/ Operators /*/ AnimatedButton& operator=(const Animation& toCpy); AnimatedButton& operator=(const AnimatedButton& toCpy); };
Pola	bool my_GotReversedMouseMoveAnimation; // Określa, czy po zjechaniu myszy z przycisku ma odtwarzać do tyłu animację po najechaniu myszy od momentu zejścia kursora z przycisku.

Metody	`AnimatedButton& operator=(const Animation& toCpy)` Operator przypisania, argumentem jest animacja, która będzie przypisana do wewnętrznego stanu przycisku – obecnej animacji. Zwraca referencję do obiektu wynikowego.
	`AnimatedButton& operator=(const AnimatedButton& toCpy);` Operator przypisania. Argumentem jest animowany przycisk. Zwraca referencję do obiektu wynikowego.
	Pozostałe metody stanowią rozwinięcie metod czysto wirtualnych z klasy bazowej *Button*.

Nazwa	`ResourceManager`
Rola	Klasa abstrakcyjna bazowa dla menadżerów zasobów.
Kod	

```
template< class T >
class ResourceManager {
public:
    typedef std::pair< std::string, T* >    Resource;
    typedef std::map< std::string, T* >     ResourceMap;

protected:
       ResourceMap m_resource;

       T* find( const std::string& strId ) {
         T* resource = NULL;
         typename ResourceMap::iterator it = m_resource.find( strId );
         if( it != m_resource.end() ) {
             resource = it->second;
         }
         return resource;
    }

    virtual T* load( const std::string& strId ) = 0;

public:
    ResourceManager() {
    }

    virtual ~ResourceManager() {
        releaseAllResources();
    }

    T* getResource( const std::string& strId ) {
        T* resource = find( strId );
        if( resource == NULL ) {
            resource = load( strId );
            // If the resource loaded successfully, add it do the
resource map
            if( resource != NULL )
                m_resource.insert( Resource( strId, resource ) );
        }
        return resource;
    }

    void releaseResource( const std::string& strId ) {
        T* resource = find( strId );
        if( resource != NULL ) {
            delete resource;
            m_resource.erase( m_resource.find( strId ) );
```

	`` } } void releaseAllResources() { while(m_resource.begin() != m_resource.end()) { delete m_resource.begin()->second; m_resource.erase(m_resource.begin()); } } }; ```
Pola	`typedef std::pair< std::string, T* >    Resource; // Reprezentacja zasobu, poprzez nazwę typu std::string i danego typu` `typedef std::map< std::string, T* >     ResourceMap; // Kontener na zasoby.`
Metody	`T* find( const std::string& strId )` `Szuka zasobu o określonym ID w postaci std::string.` `Zwraca wskaźnik do znalezionego zasobu, lub NULL w przypadku nie znalezienia.`
	`T* getResource( const std::string& strId )` `Szuka zasobu o określonym ID w postaci std::string i w przypadku nieodnalezienia zasobu próbuje go wczytać, wtedy dodaje go do kontenera.` `Zwraca wskaźnik do znalezionego zasobu, lub NULL w przypadku nie znalezienia i niepowodzenia wczytania.`
	`void releaseResource( const std::string& strId )` `Zwalnia zasób o podanym ID.`
	`void releaseAllResources()` `Zwalnia wszystkie zasoby.`
	`virtual T* load( const std::string& strId ) = 0;` `Jedyna metoda czysto wirtualna. Przeznaczona jest do wczytywania zasobu konkretnego typu o ścieżce ID.` `Zwraca wskaźnik do wczytanego zasobu.`

Nazwa	`ImageManager : public ResourceManager< sf::Image >`
Rola	Klasa reprezentująca menadżer obrazów.
Kod	``` class ImageManager : public ResourceManager< sf::Image > { private: protected: virtual sf::Image* load(const std::string& strId); public: }; ```
Pola	Patrz klasa bazowa `ResourceManager`.
Metody	`virtual sf::Image* load( const std::string& strId );` `Konkretyzacja wczytania obrazu z dysku o ścieżce ID.` `Zwraca wskaźnik do wczytanego obrazu lub NULL w przypadku niepowodzenia.`

Nazwa	`AnimationManager : public ResourceManager< sf::Animation >`

Rola	Klasa reprezentująca menadżer animacji.
Kod	`class AnimationManager : public ResourceManager< sf::Animation > {` `protected:` `    virtual sf::Animation* load( const std::string& Path, const std::string& FileName, const std::string& Extension, const int FramesAmmount );` `virtual sf::Animation* load( const std::string& strId );` `public:` `sf::Animation& AddResource( const std::string& Path, const std::string& FileName, const std::string& Extension, const int FramesAmmount, const std::string& AnimID );` `sf::Animation& getResource( const std::string& strId );` `};`
Pola	Patrz klasa bazowa `ResourceManager`.
Metody	`virtual sf::Animation* load( const std::string& Path, const std::string& FileName, const std::string& Extension, const int FramesAmmount );` `Konkretyzacja wczytania animacji z dysku o ścieżce Path, nazwie pliku FileName, rozszerzeniu Extension oraz ilości klatek FramesAmmount. Obecna implementacja wczytuje zgodnie z formatem animacji wyeksportowanej przez Adobe Flash CS5. Poszczególne klatki takiej animacji zapisane są w następujący sposób: (5 cyfrowy numer, pozycje nieznaczące oznaczają 0).(rozszerzenie). Przykładowo: 00001.jpg` `Wartość zwracana: wskaźnik do wczytanej animacji lub NULL w przypadku niepowodzenia.` `Obrazy wczytywane są do globalnego menadżera obrazów.`
	`sf::Animation& AddResource( const std::string& Path, const std::string& FileName, const std::string& Extension, const int FramesAmmount, const std::string& AnimID );` `Dodanie animacji do kontenera o ID określonym przez AnimID. Wywołuje metodę load z przekazanymi pierwszymi czterema parametrami.` `Zwraca wskaźnik do dodanej animacji lub NULL w przypadku niepowodzenia.`

Nazwa	`ButtonManager : public ResourceManager< sf::AnimatedButton >`
Rola	Klasa reprezentująca menadżer przycisków.
Kod	`class ButtonManager : public ResourceManager< sf::AnimatedButton > {` `private:` `protected:` `virtual sf::AnimatedButton* load( const std::string& strId );` `public:` `sf::AnimatedButton& AddResource( sf::AnimatedButton& newResource, const std::string& strId );` `sf::AnimatedButton& getResource( const std::string& strId );` `};`
Pola	Patrz klasa bazowa `ResourceManager`.
Metody	`sf::AnimatedButton& AddResource( sf::AnimatedButton& newResource, const std::string& strId );` `Dodaje do kontenera animowany przycisk NewResource i przypisuje mu ID strId.` `Zwraca wskaźnik do dodanego zasobu.`

Nazwa	`FontManager : public ResourceManager< sf::Font >`
Rola	Klasa reprezentująca menadżer czcionek.
Kod	`class FontManager : public ResourceManager< sf::Font > {` `private:` `protected:` `    virtual sf::Font* load( const std::string& strId );` `public:` `};`
Pola	Patrz klasa bazowa `ResourceManager`.
Metody	`virtual sf::Image* load( const std::string& strId );` `Konkretyzacja wczytania czcionki z dysku o ścieżce ID.` `Zwraca wskaźnik do wczytanej czcionki lub NULL w przypadku niepowodzenia.`

Nazwa	`Ball: public sf::AnimatedButton`
Rola	Klasa reprezentująca pojedyńczą kulkę.
Kod	`class Ball: public sf::AnimatedButton` `{` `public:` `enum Color{` `Red,` `Green,` `Blue,` `Yellow,` `Violet,` `Count` `};` `private:` `Color myColor;` `sf::String myActiveString;` `bool myActivated;` `public:` `Ball(): myActivated(false)` `{` `myActiveString.SetFont(*gFontManager.getResource("Data/Resources/arial.ttf"));` `myActiveString.SetText("+");` `RandomColor();` `}` `Ball(Color BallColor): myColor(BallColor), myActivated(false)` `{` `myActiveString.SetFont(*gFontManager.getResource("Data/Resources/arial.ttf"));` `myActiveString.SetText("+");` `SetColor(BallColor);` `}` `virtual ~Ball();` `bool isActivated() const;` `void Activate();` `void RandomColor();` `void RandomColor(int Ammount);` `void SetColor(Color);` `Color GetColor() const;` `void Draw(sf::RenderTarget& Target);`

	};
Pola	enum Color{ // Typ wyliczeniowy koloru kulki. Red, Green, Blue, Yellow, Violet, Count }; Color myColor; // Określa kolor kulki. sf::String myActiveString; // Wyświetlany string w przypadku, gdy kulka jest aktywowana. Domyślnie „+” bool myActivated; // Określa, czy kulka jest aktywowana – true jeśli tak, false jeśli nie.
Metody	Ball() Konstruktor tworzący kulkę o losowym kolorze.
	Ball(Color BallColor) Konstruktor tworzący kulkę o określonym kolorze.
	void Activate(); Aktywuje daną kulkę zmieniając wartość myActivated na true.
	void RandomColor(); Ustawia kolor losowy.
	void RandomColor(int Ammount); Ustawia kolor losowy. Ilość kolorów ograniczona przez Ammount.
	void SetColor(Color); Ustawia zadany kolor.
	Color GetColor() const; Zwraca kolor danej kulki.
	void Draw(sf::RenderTarget& Target); Rysuje kulkę na wyznaczonym celu.

Nazwa	BallsCollection
Rola	Klasa reprezentująca mapę – kolekcję kulek.
Kod	class BallsCollection { private: std::vector<std::vector<Ball>> myCollection; int myColors; protected: virtual void Render(RenderTarget& Target); public: BallsCollection() {} BallsCollection(int ColumnsX, int RowsY, int Colors); ~BallsCollection(); Vector2i MouseMovedEvent(Event::MouseMoveEvent&); Vector2i MouseButtonPressedEvent(Event::MouseButtonEvent&); int GetColors() const; int GetUnactivated(sf::Vector2i First, Vector2i Last) const; int Activate(sf::Vector2i First, Vector2i Last); void AnimatedBlink(); void Randomize(sf::Vector2i First, Vector2i Last); void Randomize(sf::Vector2i First, Vector2i Last, int Colors); bool CollectionActivated();

	`void Draw(RenderTarget& Target);` `void Update();` `std::vector<Ball>& operator[](int);` `};`
Pola	`std::vector<std::vector<Ball>> myCollection; // Reprezentacja dwuwymiarowej mapy kulek` `int myColors; // ilość kolorów na danej mapie`
Metody	`BallsCollection(int ColumnsX, int RowsY, int Colors);` `Utworzenie mapy o określonej ilości kolumn, wierszy i kolorów na mapie.`
	`Vector2i MouseMovedEvent(Event::MouseMoveEvent&);` `Informuje kolekcję o zajściu zdarzenia poruszenia myszą.` `Zwraca koordynaty kulki, której zdarzenie dotyczy w postaci Vectora2i lub Vector2i.x = -1 i Vector2i.y = -1, gdy nie dotyczy żadnej.`
	`Vector2i MouseButtonPressedEvent(Event::MouseButtonEvent&);` `Informuje kolekcję o zajściu zdarzenia wciśnięcia przycisku myszy.` `Zwraca koordynaty kulki, której zdarzenie dotyczy w postaci Vectora2i lub Vector2i.x = -1 i Vector2i.y = -1, gdy nie dotyczy żadnej.`
	`int GetColors() const;` `Zwraca ilość kolorów na mapie.`
	`int GetUnactivated(sf::Vector2i First, Vector2i Last) const;` `Zwraca ilość nieaktywowanych kulek na danym obszarze. Kulka początkowa ma koordynaty First, końcowa Last.` `Zwraca ilość nieaktywowanych kulek z tego przedziału.`
	`int Activate(sf::Vector2i First, Vector2i Last);` `Aktywuje kulki na danym obszarze. Kulka początkowa ma koordynaty First, końcowa Last.` `Zwraca ilość zaaktywowanych kulek.`
	`void AnimatedBlink();` `Podowuje animację mrugnięcia na całej mapie, poprzez odtworzenie animacji stanu idle.`
	`void Randomize(sf::Vector2i First, Vector2i Last);` `Przypisuje losowe kolory kulkom na podanym przedziale.`
	`void Randomize(sf::Vector2i First, Vector2i Last, int Colors);` `Przypisuje losowe kolory kulkom na podanym przedziale. Ilość kolorów podana w parametrze Colors.`
	`bool CollectionActivated();` `Zwraca true, jeśli cała kolekcja została zaaktywowana.`
	`void Update();` `Wywołuje Update() na swojej całej kolekcji.`
	`void Draw(RenderTarget& Target);` `Wywołuje Draw() do Target na całej swojej kolekcji.`
	`std::vector<Ball>& operator[](int);` `Operator tablicowy w celu dostania się do konkretnego elementu jak do elementu tablicy dwuwymiarowej.`

Nazwa	Countdown
Rola	Klasa reprezentująca zegar odliczający pozostały czas z określoną precyzją typu.
Kod	`template <typename T>` `class Countdown` `{` `clock_t myClock;` `T myCount;` `public:` `Countdown(): myCount(0), myClock(clock())` `{}` `Countdown(const T&): myCount(T), myClock(clock())` `{}` `~Countdown()` `{}` `T GetCurrentCount()` `{` `T CurrentCount = myCount - (clock()-myClock)/CLOCKS_PER_SEC;` `return CurrentCount > 0 ? CurrentCount : 0;` `}` `void SetCount(const T& newCount)` `{` `myCount = newCount;` `}` `void Start()` `{` `myClock = clock();` `}` `void IncreaseCount(const T& toAdd)` `{` `myCount += toAdd;` `}` `bool isFinished()` `{` `return GetCurrentCount() == 0 ? true : false;` `}` `};`
Pola	`clock_t myClock;  // Określa czas początku naliczania.` `T myCount; // Określa wartość od której rozpoczyna się naliczanie.`
Metody	`Countdown(const T&)` `Konstruktor. Ustawia odliczanie od podanej w parametrze wartości.`
	`T GetCurrentCount()` `Zwraca aktualny licznik.`
	`void SetCount(const T& newCount)` `Ustawia nową wartość odliczaniaod podanej w parametrze wartości.` `Ważne – należy wywołać metodę Start() w celu liczenia od podanej wartości od nowa.`
	`void Start()` `Ustawia czas początku naliczania na czas wywołania tej metody.`
	`void IncreaseCount(const T& toAdd)` `Zwiększa licznik o podaną wartość.`
	`bool isFinished()` `Zwraca true, jeśli GetCurrentCount zwraca wartość 0. W przeciwnym razie false.`

Nazwa	Selection: public sf::Shape
Rola	Klasa reprezentująca zaznaczenie.
Kod	class Selection: public sf::Shape { BallsCollection* myArea; Vector2i myFirstCoords; Vector2i myLastCoords; bool myMouseButtonPressed; bool mySelectionPassed; int myMaximumStreak; public: Selection(): myArea(NULL), myFirstCoords(-1,-1), myLastCoords(-1,-1), myMouseButtonPressed(false), mySelectionPassed(false), myMaximumStreak(0) { this->Shape::operator=(sf::Shape::Rectangle(-1, -1, -1, -1, sf::Color::White, 3, sf::Color::White)); this->EnableFill(false); this->EnableOutline(true); } Selection(BallsCollection* newArea): myArea(newArea), myFirstCoords(-1,-1), myLastCoords(-1,-1), myMouseButtonPressed(false), myMaximumStreak(0) { this->Shape::operator=(sf::Shape::Rectangle(-1, -1, -1, -1, sf::Color::White, 3, sf::Color::White)); this->EnableFill(false); this->EnableOutline(true); } ~Selection() {} int GetSelectedAmmount() const; int GetMaximumStreak() const; int GetUnactivatedAmmount() const; void SetArea(BallsCollection* newArea); bool Validate(); bool isSelectionPassed(); void MouseButtonPressedEvent(Vector2i newPosition); int MouseButtonReleasedEvent(); void MouseMovedEvent(Vector2i newPosition); void GroupCoords(Vector2i& First, Vector2i& Last) const; void Draw(RenderTarget& Target); void Update(); };
Pola	BallsCollection* myArea; // Wskaźnik na obszar zaznaczenia. Vector2i myFirstCoords; // Koordynaty pierwszego zaznaczonego elementu. Vector2i myLastCoords; // Koordynaty elementu ostatniego. bool myMouseButtonPressed; // True, jeśli przycisk myszy jest wciśnięty bool mySelectionPassed; // True, jeśli selekcja prawidłowa, tj. spełnia warunki naliczenia punktów. int myMaximumStreak; // Przechowuje maksymalny streak wykonany przez gracza.
Metody	Selection(BallsCollection* newArea) Konstruktor. Przypisuje selekcję na wyznaczony obszar. Tworzy również

	reprezentację graficzną zaznaczenia (prostokąt, przypisuje kolory).
	`int GetSelectedAmmount() const;` Zwraca ilość zaznaczonych kulek.
	`int GetMaximumStreak() const;` Zwraca wartość myMaximumStreak.
	`int GetUnactivatedAmmount() const;` Zwraca ilość kulek nieaktywowanych.
	`void SetArea(BallsCollection* newArea);` Zmienia myArea na newArea.
	`bool Validate();` Sprwadza, czy selekcja spełnia wymagania przyznania punktów. Ustawia mySelectionPassed na true lub false oraz wywołuje dla true animatedBlink na myArea.
	`bool isSelectionPassed();` Zwraca mySelectionPassed.
	`void MouseButtonPressedEvent(Vector2i newPosition);` Obsługa wciśnięcia przycisku myszy. Przypisuje koordynaty pierwszej pozycji wartością z parametru.
	`int MouseButtonReleasedEvent();` Obsługa zwolnienia przycisku myszy. Zwraca ilość aktywowanych kulek.
	`void MouseMovedEvent(Vector2i newPosition);` Obsługa poruszenia myszy. Przypisuje ostatnim koordynatom podaną wartość.
	`void GroupCoords(Vector2i& First, Vector2i& Last) const;` Grupuje koordynaty na obszarze jaki tworzą zgodnie z zasadą: Pierwszy – najbardziej na lewo, najbardziej u góry; Ostatni – najbardziej na prawo, najbardziej na dole.
	`void Update();` Uaktualnia zaznaczenie w stosunku do rysowania.
	`void Draw(RenderTarget& Target);` Rysuje dane zaznaczenie na danym obszarze Target.

Nazwa	`Level`
Rola	Klasa reprezentująca poziomy gry.
Kod	

```
class Level
{
	BallsCollection* myCurrentMap;
	Selection mySelection;
	std::vector<BallsCollection> myGeneratedMaps;
	int myCurrentLevel;
	unsigned int myPoints;
	bool myLevelsFinished;
public:

	Level();
	~Level() {}
	int GetUnactivatedItems();
	int GetSelectedItems() const;
	int GetMaximumStreak() const;
	int GetCurrentLevel() const;
	bool LevelsCompleted() const;
```

	unsigned int GetPoints() const; void AddPoints(int Value); int CalculatePoints(int ActivatedItems); void Update(); void Draw(RenderTarget& Target); void GenerateMaps(unsigned int Ammount); void NextLevel(); bool LevelCompleted(); void Restart(); void MouseMovedEvent(Event::MouseMoveEvent); void MouseButtonPressedEvent(Event::MouseButtonEvent); void MouseButtonReleasedEvent(Event::MouseButtonEvent); };
Pola	BallsCollection* myCurrentMap; // Wskaźnik do obecnej mapy z myGeneratedMaps. Selection mySelection; // Selekcja std::vector<BallsCollection> myGeneratedMaps; // Kontener wygenerowanych map. int myCurrentLevel; // Obecny poziom mapy, zaczyna się od 0 co oznacza poziom 1 unsigned int myPoints; // Aktualna ilość punktów gracza bool myLevelsFinished; // True, jeśli wszystkie poziomy ukończone.
Metody	int GetCurrentLevel() const; Zwraca myCurrentLevel.
	bool LevelsCompleted() const; Zwraca myLevelsFinished.
	unsigned int GetPoints() const; Zwraca myPoints.
	void AddPoints(int Value); Dodaje punkty w ilości Value.
	int CalculatePoints(int ActivatedItems); Oblicza ilość punktów w stosunku do ilości aktywowanych kulek. Algorytm to n^2.
	void GenerateMaps(unsigned int Ammount); Generuje mapy i wkłada je do kontenera myGeneratedMaps. Obecnie parametr Ammount nie ma znaczenia i generowane są zawsze 4 mapy zawierające kolejno: Level1(7,8,2), Level2(9,8,3), Level3(11,8, 4), Level4(13,8,5) Gdzie liczba 1 oznacza kolumny, druga wiersze a trzecia ilość kolorów na mapie. Parametr Ammount przewidywany był przy rozbudowie programu o dodatkowe opcje.
	void NextLevel(); Inkrementuje myCurrentLevel oraz przypisuje wskaźnik myCurrentLevel na kolejny element kontenera.
	bool LevelCompleted(); Zwraca true jeśli wszystkie kulki na obecnej mapie są aktywowane.
	void Restart(); Resetuje wszystkie wartości i generuje nowe mapy.
	void MouseMovedEvent(Event::MouseMoveEvent); Informuje swoją kolekcję i selekcję o zaszłym zdarzeniu poruszenia myszą.
	void MouseButtonPressedEvent(Event::MouseButtonEvent); Informuje swoją kolekcję i selekcję o zaszłym zdarzeniu naciśnięcia przycisku myszy.

	`void MouseButtonReleasedEvent(Event::MouseButtonEvent);` Informuje swoją kolekcję i selekcję o zaszłym zdarzeniu zwolnienia przycisku myszy.

Nazwa	`Configuration`
Rola	Klasa abstrakcyjna konfiguracji.
Kod	`template <typename T>` `class Configuration` `{` `protected:` `T* myResource;` `std::vector<std::string> myAdded;` `public:` `Configuration() {}` `virtual ~Configuration(){` `Release();` `}` `void Release(){` `for (unsigned int i = 0; i < myAdded.size(); ++i)` `myResource->releaseResource(myAdded[i]);` `}` `virtual void Init() = 0;` `};`
Pola	`T* myResource;` `// Wskaźnik na menadżer zasobów.` `std::vector<std::string> myAdded;` `// Przechowuje ID dodanych zasobów.`
Metody	`void Release()` Zwalnia wszystkie zasoby o ID z kontenera myAdded.
	`virtual ~Configuration()` Destruktor. Zwalnia wszystkie zasoby o ID z kontenera myAdded.
	`virtual void Init() = 0;` Czysto wirtualna metoda wczytująca konkretne zasoby do myResource i dodająca ich id do myAdded.

Nazwa	`BallsConfiguration: public Configuration <ButtonManager>`
Rola	Klasa konfiguracji kulek.
Kod	`class BallsConfiguration: public Configuration <ButtonManager>` `{` `public:` `BallsConfiguration() {}` `BallsConfiguration(ButtonManager* toLink);` `void Init();` `};`
Pola	Patrz klasa *Configuration*.
Metody	`BallsConfiguration(ButtonManager* toLink);` Przypisuje do myResource wskaźnik do danego ButtonManagera. Uwaga: W klasie bazowej nie została stworzona czysto wirtualny konstruktor z uwagi na zmienną liczbę parametrów – ten menadżer korzysta z jednego zasobu, ale jest wiele korzystających z różnej ilości.
	`void Init();`

	Metoda wczytująca konkretne zasoby do myResource i dodająca ich id do myAdded. Wywoływane w konstruktorze.

Nazwa	GameplayImages: public Configuration <ImageManager>
Rola	Klasa konfiguracji obrazów do rozgrywki.
Kod	class GameplayImages: public Configuration <ImageManager> { public: GameplayImages() {} GameplayImages(ImageManager* toLink); void Init(); };
Pola	Patrz klasa *Configuration*.
Metody	GameplayImages(ImageManager* toLink); Przypisuje do myResource wskaźnik do danego ImageManager'a.
	void Init(); Metoda wczytująca konkretne zasoby do myResource i dodająca ich id do myAdded. Wywoływane w konstruktorze.

Nazwa	GameplayConfiguration
Rola	Klasa konfiguracji całości rozgrywki.
Kod	class GameplayConfiguration { ButtonManager* myButtonManager; ImageManager* myImageManager; BallsConfiguration myGameplayBalls; GameplayImages myGameplayImages; public: GameplayConfiguration() {} GameplayConfiguration(ButtonManager* Buttons, ImageManager* Images): myButtonManager(Buttons), myImageManager(Images), myGameplayBalls(myButtonManager), myGameplayImages(myImageManager) { srand((unsigned int)time(NULL)); } ~GameplayConfiguration() { myGameplayBalls.Release(); myGameplayImages.Release(); } };
Pola	ButtonManager* myButtonManager; // Wskaźnik do ButtonManager'a ImageManager* myImageManager; // Wskaźnik do ImageManager'a BallsConfiguration myGameplayBalls; // Posiada konfigurację kulek GameplayImages myGameplayImages; // Oraz obrazów gameplay'a
Metody	GameplayConfiguration(ButtonManager* Buttons, ImageManager* Images) W konstruktorze tworzą się obiekty BallsConfiguration i GameplayImages, które inicjują wczytanie potrzebnych zasobów. Przypisane są odpowiednie menadżery oraz wywołana funkcja randomizacji.
	~GameplayConfiguration() Destruktor. Zwalnia zasoby poprzez wywołanie destruktorów

	BallsConfiguration i GameplayImages.

Nazwa	GameplayController
Rola	Kontroler rozgrywki
Kod	class GameplayController { sf::RenderWindow* App; GameplayView* View; sf::Sprite Background; CustomCursor Cursor; Level GameLevel; Countdown<int> myCountdown; bool myPlaying; public: GameplayController(sf::RenderWindow*); ~GameplayController() { delete View; } void StartGame(); void RestartGame(); void EndGame(); };
Pola	sf::RenderWindow* App; // Wskaźnik do okna. GameplayView* View; // Wskaźnik do widoku. sf::Sprite Background; // Zawiera poszczególne modele. CustomCursor Cursor; Level GameLevel; Countdown<int> myCountdown; bool myPlaying; // True jeśli gra się toczy.
Metody	GameplayController(sf::RenderWindow*); W konstruktorze tworzony jest widok i przypisane odpowiednie wartości.
	void StartGame(); Rozpoczyna grę.
	void RestartGame(); Restartuje grę.
	void EndGame(); Kończy grę.

Nazwa	GameoverController
Rola	Kontroler końca gry.
Kod	class GameoverController { sf::RenderWindow* App; GameoverView* View; sf::Sprite Background; Ball YesButton; Ball NoButton; CustomCursor& Cursor;

	Level& GameLevel; public: GameoverController(sf::RenderWindow* ptrWin, Level& refLvl, CustomCursor& refCursor) // Pozostała część patrz kod. ~GameoverController() { delete View; } bool PlayAgain(); };
Pola	sf::RenderWindow* App; // Wskaźnik do okna. GameoverView* View; // Wskaźnik do widoku. sf::Sprite Background; Ball YesButton; // Przycisk tak – kulka. Ball NoButton; // Przycisk nie – kulka. CustomCursor& Cursor; Level& GameLevel;
Metody	GameoverController(sf::RenderWindow* ptrWin, Level& refLvl, CustomCursor& refCursor) W konstruktorze przypisywane są odpowiednie pola oraz tworzony jes widok wraz z ustaleniem właściwości kulek – YesButton oraz NoButton.
	void Init(); Metoda wczytująca konkretne zasoby do myResource i dodająca ich id do myAdded. Wywoływane w konstruktorze.

Nazwa	GameplayView
Rola	Klasa widoku rozgrywki.
Kod	class GameplayView { /*/ References to models that view is using /*/ Level& GameLevel; CustomCursor& Cursor; sf::Sprite& Background; Countdown<int>& myCountdown; /*/ Strings that will display models data in current view's specific way /*/ sf::String mySelectedBallsDisplay; sf::String myLevelDisplay; sf::String myMaxStreakDisplay; sf::String myPointsDisplay; sf::String myCountdownDisplay; public: GameplayView(Level& refLevel, CustomCursor& refCursor, sf::Sprite& refBg, Countdown<int>& refCount): GameLevel(refLevel), Cursor(refCursor), Background(refBg), myCountdown(refCount) { /* Rozbudowane: patrz kod */} ~GameplayView() {} void Update(); void Draw(sf::RenderTarget& Target); };
Pola	Posiada referencje do modelów oraz stringi do wyświetlania danych.
Metody	void Update(); Aktualizuje wyświetlane dane.
	void Draw(sf::RenderTarget& Target);

	Rysuje wyświetlane dane (stringi) oraz powołuje modele do rysowania..

Nazwa	GameoverView
Rola	Klasa widoku zakończenia gry.
Kod	class GameoverView { /*/ References to models that view is using /*/ Level& GameLevel; CustomCursor& Cursor; Ball& YesButton; Ball& NoButton; sf::Sprite& Background; sf::String myPointsDisplay; public: GameoverView(Level& refLevel, CustomCursor& refCursor, sf::Sprite& refBg, Ball& refYes, Ball& refNo): GameLevel(refLevel), Cursor(refCursor), Background(refBg), YesButton(refYes), NoButton(refNo) { /* Rozbudowane: patrz kod */ } ~GameoverView() {} void Draw(sf::RenderTarget& Target); };
Pola	Posiada referencje do modelów oraz stringi do wyświetlania danych.
Metody	void Update(); - nie posiada z uwagi na brak konieczności odświeżania danych. Gra się zakończyła.
	void Draw(sf::RenderTarget& Target); Rysuje wyświetlane dane (stringi) oraz powołuje modele do rysowania.

V. Testowanie

Z uwagi na specyficzną formę programu jaklą jest gra – testowanie głównie polegało na prowadzeniu rozgrywki. Po nieokreślonej ilości zagrań oraz testowaniu na laboratorium znaleziony został bug, powodujący zwalnianie programu przy wyższych poziomach. Początkowo stwierdziłem, że SFML tworzy instancje sf::Font, co w rzeczywistości ma miejscę. Po utworzeniu menadżera czcionek problem nie ustąpił. Jak się ostatecznie okazało wina leży w kompilacji z opcją Debug, która najprawdopodobniej dodaje dużo „ciężkiego” kodu. Po zaznaczeniu opcji Release program działa bez zarzutów. Dlatego też sugeruję, aby program kompilowany był pod Release w celu poprawnego działania. Ponadto kompilator w VS2010 ostrzega przed konwersją, ale wynika to z faktu formy biblioteki SFML i typach tam występujących – mnie nie zawsze była potrzebna taka precyzja lub wartości signed/unsigned, stąd ostrzeżenia.

VI. Wnioski

Tworzenie tego typu programu przyniosło mi sporo satysfakcji. Jest to moja pierwsza gra, wcześniej nie miałem z tym styczności. Przede wszystkim zyskałem sporą wiedzę na temat biblioteki SFML oraz Boost, mimo że ostatecznie nie została ona zawarta. Ponadto tworzenie grafiki do gry było także przyjemne. Wnioski bardziej szczegółowe trudno tutaj zamiścić, z uwagi iż tak naprawdę wiele z nich to wiedza zdobyta w etapie projektowania, która przedstawiona jest w osobnym rozdziale.