Projekt programistyczny						
Rok akademicki	Termin	Rodzaj studiów	Kierunek	Prowadzący	Grupa	Sekcja
2014/2015	Wtorek 12:45 - 15:00	SSI	INF	dr inż. Arkadiusz Biernacki	GKiO3	1

Realizacja wybranego przypadku użycia

Danmaku Shooter

Skład sekcji:

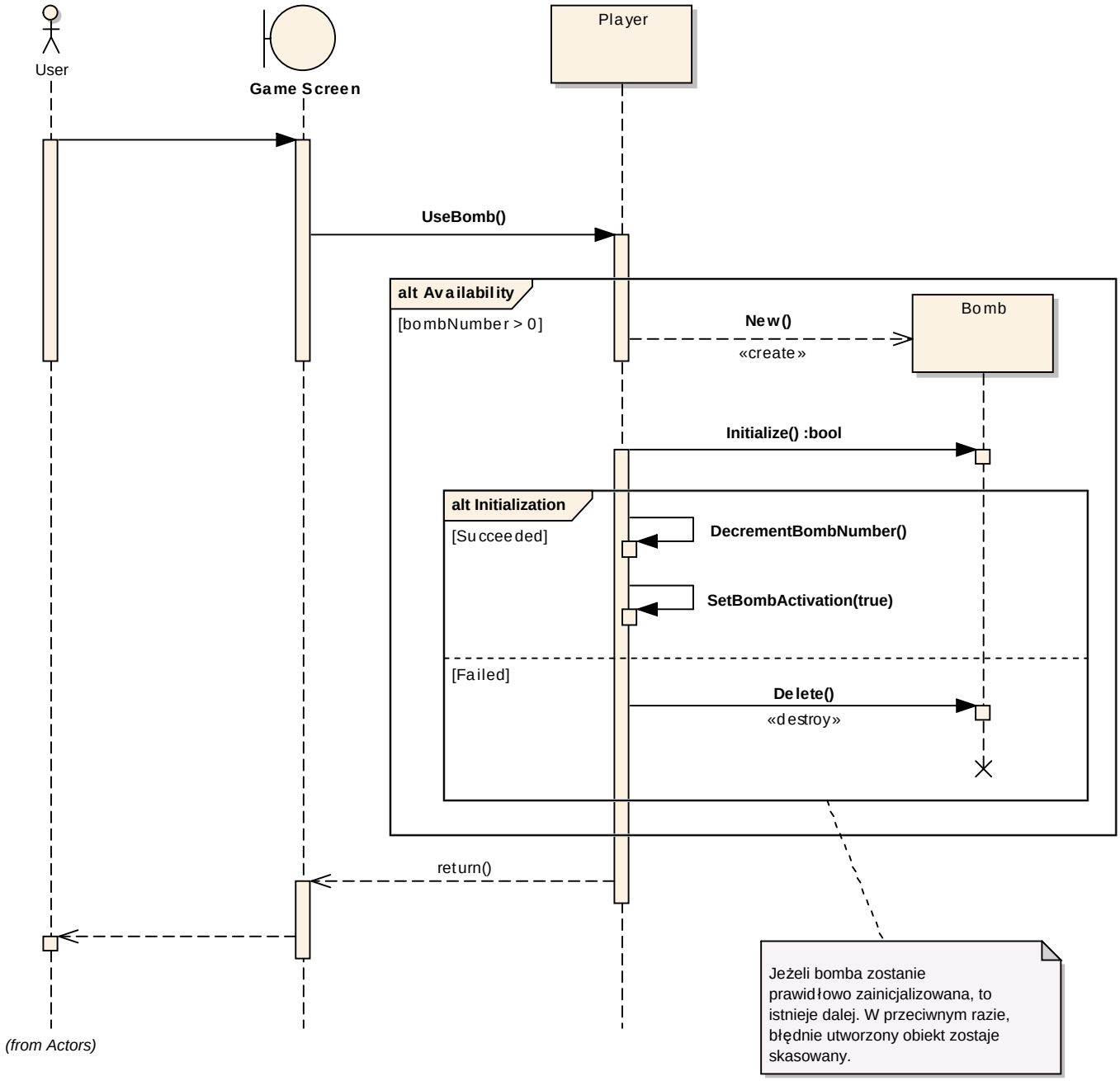
Buchała Bartłomiej

Forczmański Mateusz

Motyka Marek

Wudecki Wojciech

Wybrany diagram przypadku użycia



Rysunek 1: Diagram przepływu przypadku użycia bomby

Specyfikacja wewnętrzna

1. Implementacja uruchomienia bomby

1.1. Wywołanie metody

Za interfejs *Game Screen* odpowiada klasa Game. Gdy użytkownik zażyczy sobie włączyć bombę, poniższy fragment kodu zostaje wywołany.

```
void Game::Update(float const time)
{
    if (input->GameKey(GameControl::BOMB))
    {
        if(this->player->UseBomb())
        {
            this->bombBar->DecrementCount();
            this->player->SetIsInvulnerable();
        }
    }
}
```

W momencie gdy wykorzystanie bomby się powiedzie, w interfejsie gry zostaje zmniejszony licznik bomb, a sam gracz staje nie się nietykalny i nie można go zranić.

1.2. Użycie bomby (funkcja UseBomb())

```
bool Player::UseBomb()
{
    if (!IsUsingBomb() && _bombCount > 0)
    {
        _bomb->Launch();
        DecrementBombNumber();
        return true;
    }
    return false;
}
```

W funkcji sprawdzana jest dostępność bomby - licznik musi mieć wartość większą od zera oraz bomba nie może być w użyciu (jednocześnie może być uruchomiona tylko jedna). Jeżeli warunek jest spełniony, bomba zostaje uruchomiona. W przeciwnym wypadku nic się nie dzieje.

1.2.1. Wystrzelenie bomby (funkcja Launch())

W funkcji *Launch*() klasy Bomb następuje zaznaczenie, że bomba jest już aktywowana.

```
void Bomb::Launch()
{
    _elapsedTime = 0;
    SetBombActivation(true);
};
```

```
void Bomb::SetBombActivation(bool activated)
{
    _activated = activated;
};
```

Ideą jest enkapsulacja informacji - gracz wie, ile bomb ma do wykorzystania, z kolei to bomba wie, czy jest w użyciu lub nie.

1.2.2. Dekrementacja licznika bomb

Ciało funkcji gracza, która zmniejsza liczbę bomb, wygląda następująco:

```
void Player::DecrementBombNumber()
{
    if (_bombCount > 0)
        _bombCount--;
}
```

1.3. Utworzenie bomby

Podczas tworzenia gry uległa zmianie konstrukcja obiektu bomby. Ponieważ forma i realizacja bomby, zawsze jest taka sama, aby zmniejszyć liczbę obliczeń i nie ładować tych samych sprajtów, jest ona tworzona tylko raz, podczas uruchamiania gry. Natomiast w trakcie rozgrywki, użytkownik jedynie wczytuje obiekt bomby oraz każe jej realizować tę samą operację wystrzału.

Jeżeli podczas uruchamiania gry, bomba (albo inny istotny obiekt) nie zostanie prawidłowo wczytany, gra się nie uruchomi - nie było sensu w tym, żeby gracz uruchamiał niekompletną grę lub w czasie *gameplay*'u istniało niepotrzebnie większe prawdopodobieństwo niepowodzenia uruchomienia któregoś z elementów.

Aktualny proces tworzenia obiektu bomby wygląda następująco:

1. W czasie uruchamiania gry, instancja bomby zostaje utworzona poprzez klasę gracza:

   ```
   bool Player::InitializeBomb(LPDIRECT3DDEVICE9 device)
   {
       _bomb = BombPtr(new Bomb(&centerPoint));
       return _bomb->Initialize(device);
   };
   ```

2. Jeżeli utworzenie bomby się nie powiodło, w klasie gry zostaje wyrzucony wyjątek i inicjalizacja gry kończy się niepowodzeniem:

   ```
   bool Game::Initialize()
   {
       try
       {
           if (!this->CreatePlayer())
               throw PlayerInitializationFailedException();
           return true;
       } catch (GameInitializationFailedException & ex)
       {
           ex.ToMessageBox();
           return false;
       }
   }
   ```

   ```
   bool Game::CreatePlayer()
   {
       bool success = true;
       success &= this->player->InitializeBomb(gDevice->device);
       return success;
   }
   ```

3. W przypadku niepowodzenia, gra zwraca informację, że nie została poprawnie utworzona, a jej instancja - wraz z jej wszystkimi elementami, w tym bombą - zostaje usunięta.

2. Odwołania do interfejsów

2.1. IDrawable

2.1.1. Opis

Klasy implementujące ten interfejs mogą zostać wyświetlone na ekranie gry.

2.1.2. Klasy wykorzystujące ten interfejs

- Sprite

2.1.3. Metody interfejsu

Wyświetlenie elementu na ekranie. Funkcja przyjmuje pozycję, gdzie sprajt ma zostać wyświetlony, oraz jego skalę i obrót. Implementacja funkcji musi zapewnić, że sprajt zostanie prawidłowo wyświetlony zgodnie z przekazanymi parametrami.

```
virtual void Draw(D3DXVECTOR2 const & position, float scale, float rotation) = 0;
```

2.1.4. Fragmenty implementacji

```
class Sprite : public IDrawable
{
public:
    void Draw(D3DXVECTOR2 const & position, float scale = 1.0f, float rotation = 0.0f)
         override;
};

void Sprite::Draw(D3DXVECTOR2 const & position, float scale, float rotation)
{
    if (this->sprite && tex)
    {
        this->sprite->Begin(D3DXSPRITE_ALPHABLEND);
        D3DXMATRIX mat;
        D3DXVECTOR2 center2D( position.x + center.x, position.y + center.y );
        D3DXVECTOR3 position3D( position.x, position.y, 0.0f );
        // skalowanie i obrót
        D3DXMatrixTransformation2D( &mat, &center2D, NULL, new D3DXVECTOR2( scale,
            scale ), &center2D, rotation, NULL );
        this->sprite->SetTransform( &mat );
        this->sprite->Draw(tex[this->currentTex], NULL, NULL, &position3D, this->color)
            ;
        this->sprite->End();
    }
};
```

2.2. ITransformable

2.2.1. Opis

Klasy implementujące ten interfejs mogą zmieniać swój kształt i pozycję.

2.2.2. Klasy wykorzystujące ten interfejs

- GameObject
- Hitbox
- Trajectory

2.2.3. Metody interfejsu

1. Przesunięcie całego obiektu zgodnie z przekazanym wektorem.

```
virtual void Translate( D3DXVECTOR2 const & translate ) = 0;
```

2. Proporcjonalne skalowanie obiektu zgodnie z przekazaną skalą.

```
virtual void Scale( float const scale ) = 0;
```

3. Obrót całego obiektu wokół własnego środka o zadany kąt.

```
virtual void Rotate( float const theta ) = 0;
```

2.2.4. Fragmenty implementacji

```
class GameObject: public ITransformable
{
public:
    void Translate( D3DXVECTOR2 const & dv ) override;
    void Rotate( float const angle ) override;
    void Scale( float const scale ) override;
};
void GameObject::Translate( D3DXVECTOR2 const & dv )
{
    this->position += dv;
};
void GameObject::Scale( float const scale )
{
    this->scale *= scale;
    this->hitbox->Scale( scale );
};
void GameObject::Rotate( float const angle )
{
    this->rotation += angle;
};
```

2.3. IException

2.3.1. Opis

Klasy implementujące ten interfejs służą do generowania wyjątków i muszą zostać prawidłowo obsłużone.

2.3.2. Klasy wykorzystujące ten interfejs

- Direct3DInitializationFailedException
- FileException
- GameInitializationFailedException
- GameWindowInitializationFailedException
- StageCreationFailedException

2.3.3. Metody interfejsu

1. Zwrócenie komunikatu w postaci łańcucha znakowego.

```
virtual std::string ToString() const = 0;
```

2. Wyświetlenie wiadomości o wyjątku w postaci Message Boxa.

```
virtual void ToMessageBox() = 0;
```

2.3.4. Fragmenty implementacji

```
class FileException: public IException
{
protected:
    std::string _fileName;
public:
    virtual std::string ToString() const override;
    void ToMessageBox() override;
};
std::string FileException::ToString() const
{
    return "Unable to open " + _fileName + " file";
};
void FileException::ToMessageBox()
{
    MessageBox(NULL, this->ToString().c_str(), "Error!", MB_OK | MB_ICONERROR);
};
```

2.4. IPattern

2.4.1. Opis

Klasy implementujące ten interfejs należą do grupy "wzorów" - są formą ataku, która może zostać wykorzystana przez jedne elementy do atakowania drugich. Wzory składają się z wielu pocisków, które układają się w pewną formę.

2.4.2. Klasy wykorzystujące ten interfejs

- EnemyPattern
- PlayerPattern

2.4.3. Metody interfejsu

1. Ustawienie wskaźnika na pozycję obiektu, który jest źródłem wzoru.

```
virtual void SetPositionPtr( D3DXVECTOR2 * const position ) = 0;
```

2. Aktualizacja stanu wzoru.

```
virtual void Update(float const time) = 0;
```

3. Narysowanie wszystkich pocisków wzoru w zadanych granicach.

```
virtual void Draw(RECT const & rect) = 0;
```

4. Dodanie nowego pocisku do zbioru.

```
virtual void AddBullet() = 0;
```

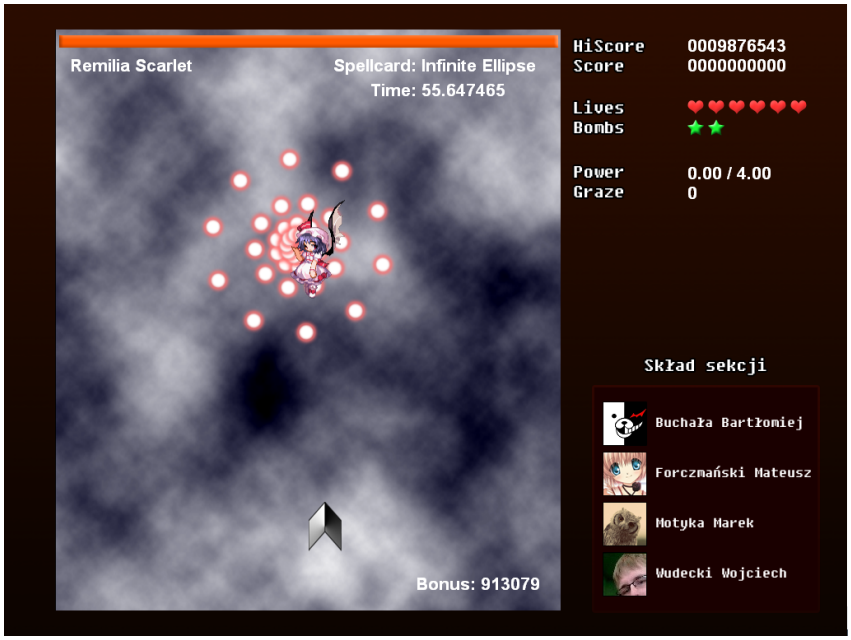
2.4.4. Fragmenty implementacji

```
class EnemyPattern: public IPattern
{
    void Draw(RECT const & rect) override;
    void SetPositionPtr(D3DXVECTOR2 * const position) override;
};
class EnemyPatternLine: public EnemyPattern
{
    void Update(float const time) override;
    void AddBullet() override;
};
void EnemyPattern::SetPositionPtr(D3DXVECTOR2 * const position)
{
    _position = position;
};
void EnemyPattern::Draw( RECT const & rect )
{
    for (EBulletQue::const_iterator it = _bullet.begin(); it != _bullet.end(); it++)
    {
        (*it)->Draw(rect);
    }
};
```

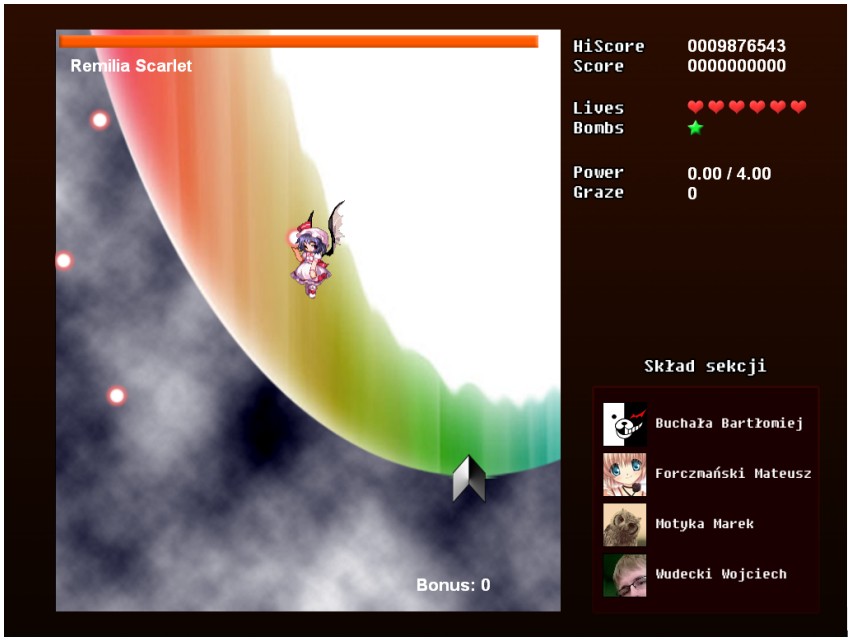
```
void EnemyPatternLine::Update(float const time)
{
    if (_activated)
    {
        EnemyPattern::Update(time);
    }
};
void EnemyPatternLine::AddBullet()
{
    EnemyBullet * newBullet = new EnemyBullet(_bulletSpeed, _bulletAcc);
    newBullet->InitializeSprite( _bulletSprite );
    newBullet->InitializeHitbox( _hitboxShape, _hitboxSize );
    newBullet->SetTrajectory( _traj );
    _bullet.push_back(newBullet);
};
```

Specyfikacja zewnętrzna

1. Przedstawienie interakcji z przypadku użycia



Powyższy rysunek prezentuje interfejs gry po jej uruchomieniu. Bomba zostaje uruchomiona, gdy gracz naciśnie przycisk wystrzelenia bomby, domyślnie jest to klawisz X.



Widoczne efekty uruchomienia bomby:

- Narysowanie sprajtu bomby
- Dekrementacja licznika, którą można zaobserwować poprzez odjęcie jednej gwiazdki w polu Bombs.
- Zmniejszenie paska życia Bossa.
- Utrata bonusu za walkę z Bossem.