Spis treści

Przedmowa

Unreal Engine 4 (UE4) jest niebywale efektywnym zestawem technologii, obecnie dostępnym dla każdego za darmo. Zapewnia środki do tworzenia wszelakiego rodzaju i wielkości gier i aplikacji każdemu, od studentów po wielkie zespoły projektowe. Książka ta ma na celu zbudowanie poczucia pewności siebie każdemu, kto go używa i podniesienia poziomu pracy deweloperów na poziom mistrzowski, gdzie każdy problem i każde wyzwanie pojawiające się w dowolnym projekcie da się rozwiązać.

Dla kogo jest ta książka

Deweloperzy z doświadczeniem w korzystaniu z UE4 mogą postrzegać tę książkę jako zbiór najlepszych praktyk, przykładów i przedstawienie głównych systemów silnika. Ci, którzy dopiero zaczynają swoją przygodę z tym środowiskiem, uzyskają solidne podstawy pozwalające na prowadzenie projektów z pewnością siebie i kompetencją.

Co znajdziemy w tej książce

Rozdział 1, *Tworzenie projektu C++ dla strzelanki*, jest punktem wyjścia dla całej książki i odpowiadającego jej projektu UE4 na GitHubie. Rozpoczniemy od instalacji silnika i jego skompilowania w C++. Następnie dodamy dostarczony przez Unreal szablon prostej strzelanki, przejdziemy przez budowanie go również w C++, po czym dodamy nową klasę gracza i elementy sterujące.

Rozdział 2, *Wyposażenie i broń gracza*, dodaje kilka podstawowych elementów rozgrywki, jednocześnie zapewniając swobodę w dodawaniu klas C++ i łączeniu ich z blueprintami w edytorze. Na koniec rozdziału gracz będzie miał w pełni funkcjonalny system ekwipunku, obejmujący zbieranie elementów i przełączanie ich kolejno przy użyciu nowych kontrolek.

Rozdział 3, *Przegląd blueprintów i skryptów BP*, jest tą częścią, w której zajmiemy się systemem skryptowania blueprintów UE4, jego zaletami i ograniczeniami, włącznie z praktyczną implementacją w mapie naszej gry.

Rozdział 4, *Wymagania UI: menu, HUD oraz ładowanie/zapis*, szybko przechodzi do kilku bardziej zaawansowanych tematów: najpierw skonfigurujemy interfejs użytkownika i HUD i powiążemy je z naszym ekwipunkiem, a później zagłębimy się w zagadnienia systemu plików w UE4 i innymi klasami potrzebnymi do implementacji mechanizmu zapisu stanu gry w dowolnym momencie i jego załadowania, po czym go również powiążemy z interfejsem użytkownika.

Rozdział 5, *Dodawanie wrogów!*, pokazuje, jak zaimportować do gry nową postać oraz zbudować i powiązać z nim cały system sztucznej inteligencji, aby mogła ona zauważyć i zaatakować naszego gracza.

Rozdział 6, *Zmienianie poziomów, strumieniowanie i zachowywanie danych*, wyjaśnia, że zmienianie map (poziomów) w UE4 lub wykorzystanie jednej z oferowanych przez silnik opcji strumieniowania jest niezbędne w każdej grze. Poświęcimy trochę czasu na poznanie dostępnych tu opcji Unreal, po czym przejdziemy do zachowywania danych przy zmienianiu map. Zaadaptujemy nasz system zapisu/ładowania stanu gry, aby zapewnić trwałość ekwipunku gracza pomiędzy poziomami, jednocześnie zachowując stan każdej mapy z chwili jej opuszczenia.

Rozdział 7, *Wprowadzanie dźwięku do gry*, omawia dźwięk jako często lekceważony i pomijany aspekt gry, który może zbudować lub zniszczyć wrażenia! Po przeglądzie głównych systemów audio dostępnych w UE4 zagłębimy się w takie zagadnienia, jak dźwięki uderzeń zależne od materiałów i efekty otoczenia.

Rozdział 8, *Edytowanie shaderów i wskazówki optymalizacyjne*, pokazuje, że materiały i tworzone dla nich shadery są zdecydowanie najważniejszym systemem w Unreal. Efekty wizualne możliwe do osiągnięcia w UE4 są niemal nieograniczone, ale wymagają dobrego zrozumienia granic i kosztów ich stosowania. Przejdziemy przez kilka praktycznych przykładów tworzenia nowych materiałów i ich profilowania, aby zapoznać się z technikami optymalizowania shaderów i dostosowywania się do możliwości różnych platform.

Rozdział 9, *Dodawanie przerywników w grze przy użyciu sekwencera*, przedstawia Sequencer, narzędzie do tworzenia przerywników w grze, które jest podstawowym rozwiązaniem tego typu obsługiwanym w UE4. Po wykonaniu sceny przerywnika z udziałem naszego gracza i wrogiej AI omówimy niektóre alternatywne rozwiązania.

Rozdział 10, *Pakowanie gry (PC, platformy mobilne)*, wyjaśnia ostatni, ale niezbędny etap tworzenia każdej gry: umieszczenie jej i uruchomienie na docelowej platformie! Przejdziemy przez kilka przykładów pakowania i instalowania gry i wyjaśnimy

niektóre różnice pomiędzy uruchamianiem gry na urządzeniu z poziomu edytora UE4 a tworzeniem autonomicznej kompilacji.

Rozdział 11, *Wolumetryczne oświetlenie, mgła i wstępne obliczanie*, pokazuje, że wprawdze UE4 oferuje mnóstwo zadziwiających systemów graficznych, ale wśród nich to oświetlenie jest najbardziej znane i podziwiane w branży jako najwyższe osiągnięcie. W tym rozdziale przeanalizujemy niektóre z zaawansowanych sposobów oświetlenia i pokażemy, jak można je dodać do gry. Przedstawimy w nim również dodawanie i modyfikowanie mgły, zarówno atmosferycznej, jak i wolumetrycznej.

Rozdział 12, *Wideo w scenie i efekty wizualne*, przedstawia komponent Media Framework i niektóre z oferowanych przez niego atrakcyjnych efektów. W szczególności zarejestrujemy najpierw krótkie wideo z gry i odtworzymy go w grze jako plik wideo (MP4). Poznamy też systemy cząsteczek Unreal, dodając jeden z nich do efektu uderzenia naszego pocisku.

Rozdział 13, *Rzeczywistość wirtualna i rzeczywistość rozszerzona w UE4*, zawiera omówienie dwóch najnowszych dodatków do rosnącej listy platform obsługiwanych przez Unreal: VR u AR. W tym finalnym rozdziale utworzymy dwa samodzielne projekty (po jednym dla każdego z tych środowisk), po czym zmodyfikujemy i zaimplementujemy funkcje unikatowe dla nich, w tym przeniesienie naszych pocisków z głównego projektu do AR.

Jak najbardziej skorzystać z tej książki

Podstawowa znajomość posługiwania się Unreal Engine 4 jest ważnym punktem startowym, ale nie jest konieczna. Celem tej książki jest zabranie osób, które posługują się tą technologią, na poziom, na którym będą się czuć swobodnie przy używaniu wszystkich aspektów silnika, tak by mogły stać się liderami w stosowaniu jej w różnych projektach. Choć UE4 jest wieloplatformowym zestawem technologii, podstawową platformą deweloperską jest komputer systemu Windows z Visual Studio. Często również wykorzystywane są komputery Mac z XCode, a ponadto oddzielnie zostały zaprezentowane telefony Android (włącznie z rozwiązaniem GearVR) i urządzenia iOS.

Pobieranie przykładowych plików kodu

Całość kodu powiązanego z książką jest hostowana na GitHubie pod adresem *https://github.com/PacktPublishing/Mastering-Game-Development-with-Unreal-Engine-4-Second-Edition*. Jeśli nastąpi potrzeba uaktualnienia tego kodu, zostanie on zaktualizowany w tym właśnie, już istniejącym repozytorium.

Pobieranie kolorowych ilustracji

Ponieważ książka jest drukowana jako czarno-biała, zdecydowaliśmy się udostępnić plik PDF zawierający kolorowe wersje tych zrzutów ekranu. Można go pobrać z następującego adresu: *http://www.ksiazki.promise.pl/aspx/produkt.aspx?pid=112144*.

Stosowane konwencje

W książce używanych jest kilka konwencji mających na celu poprawę czytelności tekstu.

- `Kod_w_tekście`: Wskazuje słowa kodu (nazwy funkcji, klas, zmiennych), nazwy folderów i plików, rozszerzenia nazw plików i ścieżki. Oto przykład: „Zamontuj pobrany plik obrazu dyskowego `WebStorm-10*.dmg` jako kolejny dysk w swoim systemie”.
- Bloki kodu (listingi) zostały złożone jak poniżej:

```
/** Przesunięcie lufy */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Projectile)
class USceneComponent* MuzzleLocation;
```

- **Wytłuszczenie**: Sygnalizuje nowy termin lub definicję.
- *Kursywa*: Wskazuje adresy URL oraz angielskie odpowiedniki używanych terminów.
- **Czcionka pogrubiona bezszeryfowa**: Wskazuje wyrazy widoczne na ekranie, na przykład nazwy menu, okien dialogowych lub komponentów blueprintów. Na przykład „Przejdź do menu **Project Settings** | **Platforms** | **Android** w edytorze UE4”.

W ten sposób oznaczane są ważne uwagi i ostrzeżenia.

W ten sposób oznaczane są wskazówki i triki.

1

Tworzenie projektu C++ dla strzelanki

Wprowadzenie

Zapraszam do lektury *Tajników Unreal Engine 4*! Celem tej książki jest przeprowadzenie w sposób holistyczny osób, które znają już projektowanie w UE4 i C++, na kolejny poziom rozwoju. Podczas gdy niektóre rozdziały skupiają się na konkretnych implementacjach systemowych i najlepszych praktykach, inne powinny zapewnić szerszą perspektywę rozległych systemów UE4, które są często używane przez zespoły tworzenia zawartości. Po zakończeniu lektury Czytelnik powinien dysponować solidnym fundamentem do podejmowania najlepszych decyzji związanych z wykorzystaniem technologii UE4 dla dowolnego zakresu lub platformy i będzie w stanie poprowadzić cały zespół przez pracę projektową aż do końca. Pojawi się tu wiele miejsc, gdzie będzie można bezpośrednio zaimplementować te systemy w projektach gier, ale ogólnym celem książki jest osiągnięcie zdolności do realizowania dowolnych potrzeb z technicznego punktu widzenia, zapewniając podstawową wiedzę na wyższym poziomie, niż niezbędna dla osób po prostu piszących kod.

W tym rozdziale rozpoczniemy projekt dla podstawowej rozgrywki bitewnej, dzięki czemu będziemy mieli podstawę do późniejszego dodawania i opracowywania bardziej zaawansowanych funkcjonalności. Choć niektóre części takiego projektu są automatycznie zarządzane z poziomu szablonów UE4, przejdziemy przez wszystkie

wymagane kroki, aby mieć pewność, że podstawowa konfiguracja, do której będę odwoływać się w dalszej części tej książki, została właściwie przygotowana, skompilowana i przetestowana i że zostały zaimplementowane w niej niektóre nowe systemy.

Główne zagadnienia omówione w tym rozdziale to:

- Skonfigurowanie i utworzenie nowego projektu gry FPS (*first-person shooter*)
- Zastąpienie istniejących klas UE4
- Dodanie i zaimplementowanie dla nich prostych funkcji C++
- Szybki przegląd opcji kompilacji i uruchamiania

Wymagania techniczne

W tym rozdziale potrzebne będą następujące komponenty:

- Visual Studio 2015 lub 2017 (dowolne wydanie)
- Unreal Engine 4.18.3 lub wersja wyższa, skompilowany z kodu źródłowego

Na początek kilka szybkich uwag dotyczących platform i instalacji: wspomniane wyżej komponenty zakładają, że używamy komputera systemu Windows 10, ale nie ma powodu, aby nie można było wykonać całej pracy przedstawionej w tej książce na komputerze Mac z uruchomioną bieżącą wersją Xcode. Pracę będę przedstawiał z punktu widzenia Visual Studio i odwoływał się do niektórych funkcjonalności tego środowiska, zatem jest to zalecane środowisko robocze, ale nie jest to koniecznie wymagane. Niektóre przykłady będą od czasu do czasu testowane i kompilowane na Macu, głównie ze względu na wymagania iOS, jednak książka nie skupia się na specyficznych funkcjach IDE inaczej niż przedstawianie kolejnych kroków (typowo przy użyciu terminologii Visual Studio) i wskazówek. Przykłady kodu odzwierciedlają formatowanie, które pochodzi z narzędzia Visual Assist firmy wholeTomato.com, które zdecydowanie polecam użytkownikom Visual Studio, ale narzędzie to nie wpływa na przebieg kompilacji ani na jej wyniki.

Cały kod źródłowy, do którego odwołuję się w tej książce, można znaleźć w repozytorium GitHub pod adresem *https://github.com/PacktPublishing/Mastering-Game-Development-with-Unreal-Engine-4-Second-Edition*, z historią zmian dla pracy przedstawianej w każdym rozdziale.

W przypadku tego rozdziału trzeba dopilnować, aby wybrać gałąź o nazwie *Chapter1*, używając listy rozwijanej w lewym górnym rogu interfejsu Web GitHuba albo użyć bezpośredniego łącza dla tej gałęzi: *https://github.com/PacktPublishing/Mastering-Game-Development-with-Unreal-Engine-4-Second-Edition/tree/Chapter-1*.

Jeśli ktoś zdecyduje się używać kodu projektu bezpośrednio z GitHuba, trzeba wziąć pod uwagę dwa wymagania: konieczne jest lokalne zainstalowanie silnika ze źródła, aby możliwe było kliknięcie prawym klawiszem myszy naszego projektu (`Mastering/Mastering.uproject`) i wybranie polecenia **Select Unreal Engine Version** (wybierz wersję silnika Unreal), a także wybranie własnego pliku instalacyjnego, który również zbuduje właściwe pliki projektu. Po wywołaniu pliku `Mastering.sln` w Visual Studio należy wykonać następujące kroki:

1. Prawym klawiszem myszy kliknij projekt gry `Mastering` w eksploratorze rozwiązań i wybierz **Set as Startup Project** (ustaw jako projekt startowy), tak by przy uruchamianiu z VS przejść bezpośrednio do edytora dla tego projektu.
2. Ustaw konfigurację na **Development Editor** (edytor deweloperski), z platformą ustawioną jako **Win64**.
3. Konieczne może być bezpośrednie kliknięcie prawym klawiszem myszy **UE4 Project** w eksploratorze i wybranie **Build** (kompiluj). Niekiedy przy kompilowaniu projektu gry nie są samoczynnie wybierane wszystkie elementy wymagane do skompilowania silnika, jeśli nie zostało to wcześniej wykonane oddzielnie.

Wersja silnika, której będziemy używać, to 4.19.0.

Budowanie projektu FPS w C++

W tym podrozdziale przejdziemy kroki tworzenia nowego projektu od zera przy użyciu Unreal Project Browser (przeglądarka projektów Unreal). Dla kogoś, kto już zna te działania, proces ten powinien być względnie szybki i oczywisty. Dla tych, którzy jeszcze tego nie robili, co może być typowe w przypadku członków zespołu dołączających do projektów, które są już w biegu, koniecznych jest kilka wstępnych kroków. Ponieważ będziemy pracować w C++ i używać Visual Studio, muszę poczynić tu uwagę dla użytkowników silnika, którzy nie korzystają z kodu źródłowego: wprawdzie jednym z zaleceń tej książki jest nie modyfikowanie źródeł silnika, jednak nadal konieczne będzie kompilowanie projektów C++, jako że na tym właśnie skupia się ta książka. Projekt wykorzystujący tylko same blueprinty* można wykonać bez kompilowania źródeł, zaś pewne zalety i wady takiego podejścia zostaną przedstawione

* Angielskie słowo „blueprint” oznacza plan, rysunek konstrukcyjny, a w znaczeniu ogólniejszym – szczegółowy sposób działania (postępowania) – można spotkać się z określeniem „recepta” albo „przepis”. Jednak zarówno w środowisku graczy, jak i projektantów gier najczęściej używany jest angielski termin i stąd decyzja o użyciu go w tej książce (wszystkie przypisy pochodzą od tłumacza).

w rozdziale 3, „Przegląd blueprintów i skryptów BP”, ale ponownie nie jest to sposób prezentowania informacji w tej książce dla większości przypadków implementacji. Warto też zauważyć, że Epic Games chętnie przyjmuje od użytkowników uwagi dotyczące wykrytych problemów lub pożądanych ulepszeń i jeśli otrzymają właściwe informacje debugowania, zazwyczaj reagują na takie zgłoszenia znacznie sprawniej, wykorzystując forum odpowiedzi, takie jak *http://answers.unrealengine.com*. Dodatkowo, jeśli czyjś projekt natychmiast potrzebuje poprawki lub zmian w kodzie silnika, to zanim jeszcze Epic Games będą w stanie pomóc trzeba umieć debugować i skompilować silnik. Na koniec, jeśli ktoś chciałby wykonać *pull requests* rodzajów zmian lub poprawek, które mają zostać zintegrowane przez Epic Games, powinien utworzyć powiązanie z ich projektem GitHub. Tak więc będziemy postępować tak, jakbyśmy instalowali silnik i projekt na „czystym” komputerze, z zapewnieniem pełnych możliwości odbudowania z kodu źródłowego.

W kolejnym podrozdziale dokonamy pewnych modyfikacji trybu gry i gracza, skompilujemy je ponownie, po czym obejrzymy wyniki w działaniu. Jeśli ktoś woli to pominąć i przejść dalej, cała praca zaprezentowana w tym rozdziale jest dostępna w repozytorium GitHub w gałęzi *Chapter-1*.

Stworzenie i uruchomienie nowego projektu wymaga zrealizowania trzech głównych kroków:

1. Pobrania i zainstalowania kodu źródłowego UE4 i skompilowania go.
2. Uruchomienia edytora w przeglądarce projektów po raz pierwszy i wybrania szablonu.
3. Skompilowania i uruchomienia tego projektu.

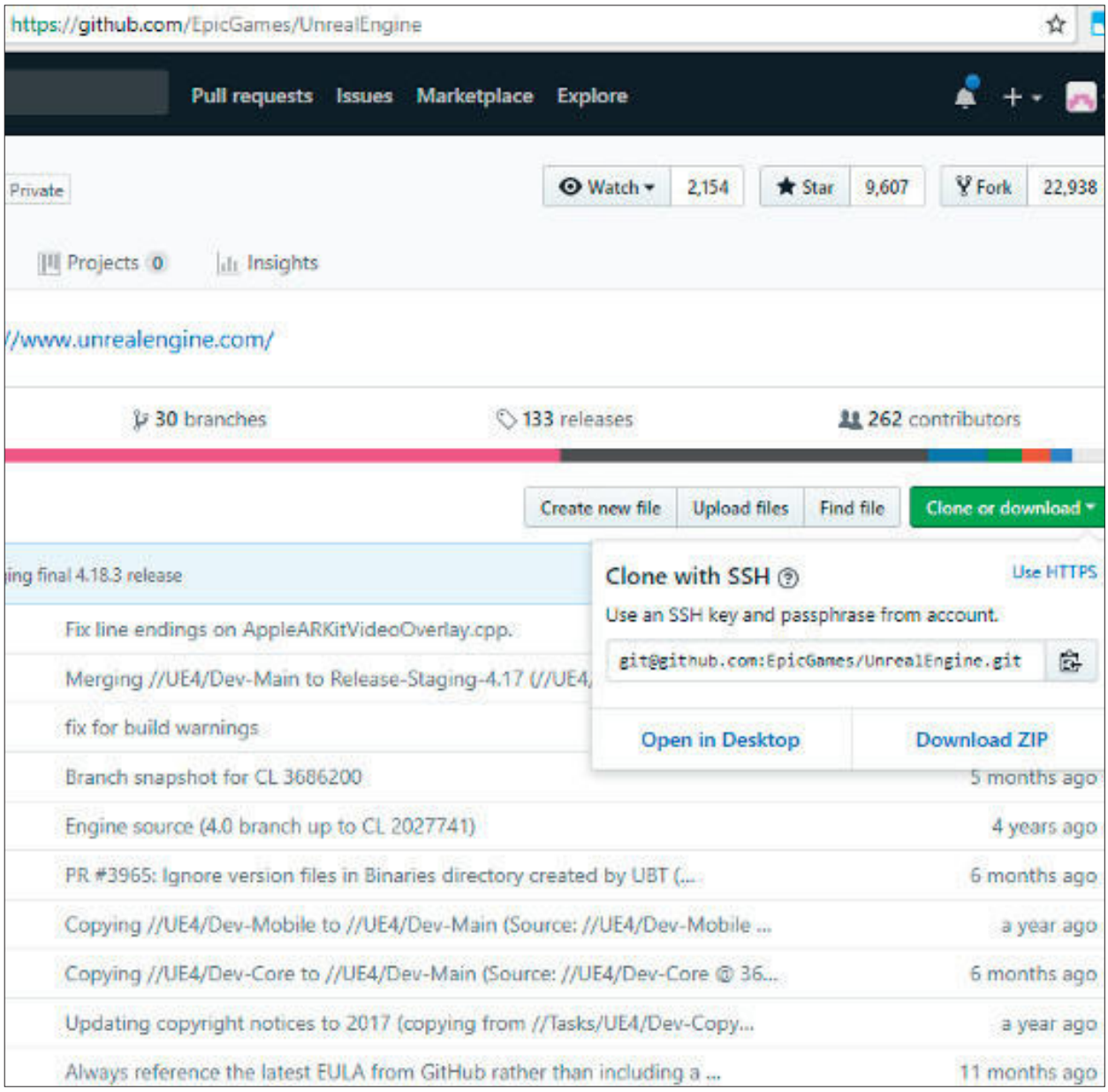
Instalowanie i kompilacja UE4

Naszym pierwszym krokiem będzie pobranie kodu źródłowego UE4 i jego skompilowanie. Można to wykonać kilkoma sposobami. Jeśli ktoś wykonuje to po raz pierwszy, najłatwiejszą metodą jest przejście do witryny GitHub w celu pobrania silnika: *https://github.com/EpicGames/UnrealEngine*.

Aby powyższe łącze działało, użytkownik musi wcześniej zarejestrować się na GitHubie i zgłosić się jako Unreal Developer. Szczegóły można znaleźć pod adresem *https://wiki.unrealengine.com/GitHub_Setup*.

Kliknięcie przycisku **Clone or download** (klonuj lub pobierz) ujawnia dostępne opcje. Spośród nich najprostszym wyjściem jest wybranie opcji pobrania projektu jako pliku

ZIP, a następnie rozpakowanie go w dowolnie wybranym miejscu na swoim dysku, jak na poniższym zrzucie ekranu:



Wykorzystanie witryny Web zawsze jest dostępną i wygodną opcją. Choć osobiście nie jestem fanem apki GitHub Desktop, również ona stwarza możliwość eksploracji projektu na tym etapie procesu. I choć muszę zauważyć, że korzystanie z SourceTree wiąże się z pewnymi utrudnieniami z punktu widzenia wrażeń użytkownika, jest to bezpłatna aplikacja do zarządzania projektami GitHub, którą mogę spokojnie polecić. Zaś ci, którzy czują się swobodnie przy pracy z wierszem polecenia, mają do dyspozycji liczne opcje oraz terminal, który można otworzyć w celu posłużenia się

tymi poleceniami w SourceTree. To, co jest istotne w tym momencie, jest zainstalowanie drzewa UE4, abyśmy mogli przejść do budowania projektu!

Pierwszą rzeczą do zrobienia jest pobranie najnowszej wersji silnika, albo jako aktualizacji, albo jako świeżej instalacji. Ważne jest, aby zawsze uruchomić `Setup.bat` (albo użyć polecenia `setup` na Macu) w głównym folderze instalacji przed wykonaniem dowolnych innych kroków. Trzeba się upewnić, że w wyskakującym oknie zostaną wyliczone wszystkie używane przez nas platformy, co zagwarantuje, że zostaną pobrane wszystkie wymagane pliki dla danej platformy, zgodnie z opisem zawartym w pliku `README.md` znajdującym się w tym samym folderze.

Po zakończeniu działania polecenia `Setup.bat/setup` należy uruchomić plik `GenerateProjectFiles.bat` i w tym samym folderze pojawi się plik `UE4.sln`. Szybka uwaga na temat statusu rozwiązań generowanych przez UE4 w kontekście VS 2015 lub VS 2017: UE4 generuje domyślnie pliki projektu zgodne z VS 2015. Możliwe jest wyspecyfikowanie opcji `-2017` jako argumentu pliku wsadowego. Jak dotąd, nie jest konieczne tworzenie projektu dla wersji 2017 i pliki projektu w wersji 2015 otwierają się, kompilują i działają doskonale zarówno w VS 2015, jak i VS 2017. Jeśli jednak mamy zainstalowane obie wersje Visual Studio, pliki projektu domyślnie będą się otwierać w VS 2015, co może być dość irytujące. W chwili pisania tej książki użycie dowolnej spośród tych wersji Visual Studio powinno dawać te same rezultaty i zostało to przetestowane, ale z czasem zapewne okaże się, że to, co znajduje się na GitHubie, wymusi użycie VS 2017 do budowania rozwiązań w edytorze. Dlaczego tak się dzieje i jak to skonfigurować, omówię w dalszej części tego rozdziału w podpunkcie *Nadpisywanie klasy Character* podrozdziału *Modyfikowanie naszej gry w C++*.

Kroki potrzebne do wykonania w celu skompilowania silnika są teraz już całkiem proste:

1. Podwójnie kliknij plik `.sln`, aby otworzyć go w Visual Studio.
2. Prawym klawiszem myszy kliknij projekt UE4 w eksploratorze rozwiązań i wybierz **Set as Startup Project** (ustaw jako projekt startowy).
3. Wybierz **Development Editor** jako konfigurację (alternatywnie możesz wybrać **DebugGame**; więcej na ten temat w dalszej części tego rozdziału) oraz **Win64** jako platformę.
4. Skompiluj projekt. To może potrwać nawet godzinę, zależnie od konfiguracji sprzętowej. Niektóre zalecenia kompilacji zostały wyliczone pod koniec tego podrozdziału.

Uruchamianie edytora i wybór szablonu

Naszym kolejnym krokiem jest uruchomienie edytora. W tym celu trzeba uruchomić silnik w Visual Studio, naciskając *F5*. Ze względu na brak projektu gry lub aplikacji w panelu rozwiązań przekieruje to nas bezpośrednio do przeglądarki projektów Unreal. Można ją również wywołać w dowolnej chwili w późniejszym czasie, klikając prawym klawiszem projekt UE4 w eksploratorze rozwiązań i wybierając jego debugowanie lub bezpośrednie uruchomienie. Zalecam również po prostu utworzenie skrótu do pliku `/Engine/Binaries/Win64/UE4Editor.exe` w folderze instalacji UE4, jako że niekiedy przydatna będzie możliwość jego szybkiego uruchomienia poza naszym programistycznym IDE. W oknie **Project Browser** trzeba teraz wykonać następujące kroki:

1. Kliknij kartę **New Project** (nowy projekt), a wewnątrz niej kartę **C++**.
2. Zaznacz ikonę **First Person** jako nasz typ projektu, aby utworzyć szkielet naszej strzelanki (*first-person shooter* – FPS).
3. Wybierz folder docelowy i nazwę projektu, po czym kliknij **Create Project** (Utwórz projekt).
4. Jeśli wybrałeś nazwę projektu inną niż **Mastering**, przeczytaj umieszczoną niżej notkę informacyjną.

Możliwe wybory dla opcji **Desktop/Console**, **Quality** oraz **Starter Content** możemy na razie pozostawić w wartościach domyślnych, ale warto poświęcić nieco czasu, by im się przyjrzeć, klikając strzałki rozwijania, aby poznać dostępne opcje każdej z tych list wraz z krótkimi opisami, co one robią. **Starter Content** to w rzeczywistości paczka zawartości Unreal i będziemy ją dodawać ręcznie w dalszej części książki.

Jako że projekt prezentowany w tej książce został skonfigurowany na GitHubie jako **Mastering**, ta nazwa będzie używana w całej tej książce jako nazwa projektu. Szablony Unreal używają jej również przy tworzeniu wielu podstawowych plików dołączanych do projektu. Jeśli wybierzesz inną nazwę projektu, to gdy odwołam się (na przykład) do pliku `MasteringCharacter.h`, będziesz musiał odnieść się do nazwy (*`TwojaNazwaProjektu`*)`Character.h` utworzonej przez szablon. Zatem dla uproszczenia polecam po prostu używanie tej samej nazwy, co w książce.

W tym momencie UE4 zamknie przeglądarkę projektów, wygeneruje pliki projektu gry i podejmie próbę otwarcia ich VS. Naturalnie w takiej chwili dobrym pomysłem jest zamknięcie sesji IDE powiązanej z samym silnikiem, gdyż projekt silnika również

został otwarty jako część rozwiązania projektu. Jak można zauważyć, nazwany przez nas projekt powinien być teraz projektem startowym i powinien zawierać kilka plików źródłowych pochodzących z szablonu C++.

Kompilowanie i uruchamianie projektu gry

Teraz możemy wreszcie skompilować i uruchomić naszą grę. Kompilowanie przykładowego projektu FPS powinno przebiec bardzo szybko i jeśli nie zostanie zmieniona konfiguracja lub platforma, nie będzie wymagane ponowne kompilowanie żadnego fragmentu kodu silnika. Jako regułę ogólną można przyjąć, że na potrzeby testów należy kompilować wersje DebugGame. Przy użyciu tej opcji otrzymujemy trochę dodatkowych informacji w czasie wykonania oraz kilka testów zabezpieczeń naszego kodu, ale typowo nie powoduje to znaczącej zmiany wydajności wskutek testowania. Tak więc w naszym przypadku zalecam użycie DebugGame Editor, mimo tego, że skompilowaliśmy silnik w trybie deweloperskim. DebugGame Editor jako samodzielna konfiguracja (DebugGame) uruchomiona na naszym komputerze będzie kompilować jedynie kod projektu gry w trybie debugowania, ale nadal będzie używać silnika w jego szybciej działającej konfiguracji projektowej. Zmiana konfiguracji na przykład na Debug Editor wymusi pełne przekompilowanie także silnika w trybie debugowania. Dodatkowo silnik skompilowany w tym trybie działa zauważalnie wolniej, zaś utrzymywanie zarówno wersji debugowania, jak i projektowej jest czasochłonne, a zasadniczo zbędne, chyba że faktycznie chcemy wykonywać bezpośrednie debugowanie kodu silnika. Po zakończeniu kompilowania projektu możemy go uruchomić przy użyciu *F5*, podobnie jak w sesji rozwiązania z samym tylko silnikiem. Spowoduje to wywołanie edytora z naszą grą jako aktywnym projektem. W środowisku UE4 edytor jest tym miejscem, w którym wszyscy projektanci, w tym programiści, wykonują większą część pracy i testów podczas tworzenia gry lub aplikacji. Jedną z największych zalet Unreal jest funkcjonalność **Play In Editor** (**PIE**), jako że umożliwia przeładowanie na gorąco bibliotek gry podczas pracy w edytorze. Gdy złożoność projektu wzrośnie, a przebieg gry się zmieni i nie będzie polegać już na prostym rozpoczęciu poziomu, ładowanie na gorąco i samo PIE nie będą już rozsądnymi opcjami testowania, ale w tej chwili możemy pokazać kilka ich zalet. W ogólności przy pracy nad systemami rozgrywki lub debugowaniu nowego kodu PIE będzie naszym najlepszym przyjacielem.

„Brzmi świetnie!", można powiedzieć. Warto to wypróbować, używając przycisku **Play** w prawej części górnego paska domyślnego układu edytora. Od razu będzie można się przekonać, że możemy się poruszać przy użyciu tradycyjnych dla strzelanek klawiszy *WASD*, strzelać z broni (z pewnymi fizycznymi efektami dla pocisków, jeśli trafią w kostki naszego poziomu), a nawet skakać po przyciśnięciu spacji.

W tej wstępnej fazie dobrym pomysłem jest rozważenie wykorzystywanego sterowania. W każdej sytuacji zalecane jest utrzymywanie sensownej koncepcji sterowania z komputera PC dla dowolnego typu gry, gdy zespół pracuje nad nią w edytorze i PIE. Nawet jeśli gra czy apka, którą tworzymy, nie będzie używać PC jako natywnej platformy, tak jak w przypadku tych przeznaczonych dla telefonów/tabletów lub VR, szybkość i łatwość testowania w PIE sprawia, że równoległe utrzymywanie sterowania dla PC jest bardzo przydatne. W istocie, jeśli przejdziemy do projektu gry i otworzymy plik `MasteringCharacter.cpp`, po czym przejrzymy początkowy kod wejściowy, zauważymy, że obsługiwane są dwie metody obrotu, odpowiadające joystikowi (lub wirtualnemu joystikowi w przypadku telefonu), a także wejście bezpośrednio wskazujące oś, takie jak mysz. Można tu zauważyć również wykomentowany kod obsługujący ruch bazujący na dotykowym ekranie tabletu/telefonu. W kolejnym podrozdziale dodamy jeszcze jeden mechanizm wejściowy. Zdecydowanie zachęcam do przejrzenia istniejących technik wejścia, aby się przekonać, co zostało już skonfigurowane i przypisane do różnych platform. Trzeba po prostu pamiętać, że w naszych czasach typowo jest znacznie łatwiej umieścić w kodzie obsługę sterowania dla wielu różnych platform, niż dodawać kontrolki dla jakiejś platformy w późniejszym czasie.

Modyfikowanie naszej gry w C++

W tym podrozdziale przyjrzymy się kilku szybkim sposobom dodawania nowych funkcji i rozgrywek do projektu poprzez dołączenie do tej gry FPS nowej mechaniki: skradania się (*stealth*). Osiągniemy to, nadpisując niektóre spośród istniejących klas dostarczonych w szablonie i dodając nowy mechanizm wejściowy oraz nieco nowego kodu. Na koniec tego podrozdziału wykonamy testy sprawdzające, że nasz kod rzeczywiście robi to, co chcemy i zobaczymy efekty w grze – nasza postać będzie przykucać, gdy wskażemy w dół. W tym celu wykonamy następujące kroki:

1. Dodamy nową klasę C++ w edytorze.
2. Zmodyfikujemy tę klasę i wymusimy jej gorące przeładowanie z powrotem do działającego edytora.
3. Dodamy nowe wejście i mechanikę rozgrywki, po czym obejrzymy ją w działaniu.

Nadpisywanie klasy Character

Aby ułatwić sobie przyszłą pracę i zacząć od stosowania dobrych praktyk, dodamy nieco specjalizowanego kodu gry, tworząc klasę potomną dla istniejącej natywnej klasy `MasteringCharacter` (zaimplementowanej w C++), dostarczonej przez szablon. Można to wykonać bezpośrednio w Visual Studio, ale EpicGames po raz kolejny udostępnia kilka skrótów, których można użyć z poziomu edytora. Zatem zacznijmy od edytora z otwartym naszym projektem, tak jak go pozostawiliśmy na koniec poprzedniego podrozdziału.

Zaczniemy od okna przeglądarki zawartości, które typowo jest zadokowane w dolnej części edytora. Jeśli nie jest już otwarte lub zostało zamknięte z jakiegoś powodu, po prostu trzeba je otworzyć ponownie, klikając polecenie **Window** w górnej części i przewijając do pozycji **Content Browser** (przeglądarka zawartości), a następnie wybierając **Content Browser 1**; w efekcie przeglądarka otworzy się jako oddzielne, autonomiczne okno. Osobiście wolę dokować to okno, gdyż taki sposób pracy z edytorem jest dla mnie wygodniejszy, ale oczywiście można pozostawić je swobodnie pływające. To jednak, co istotne, znajduje się po lewej stronie. Bezpośrednio pod listą rozwijaną **Add New** (dodaj nowy) znajdziemy małą ikonę z trzema liniami i małą strzałką. Kliknięcie jej otwiera panel **Sources** (źródła), moim zdaniem niezwykle pomocny dla nawigowania po zawartości w edytorze. W panelu tym, poniżej węzła **Content** (zawartość), znajdziemy folder o nazwie `FirstPersonCPP`, a w nim folder `Blueprints`. Po kliknięciu tego folderu zauważymy element `FirstPersonCharacter` w prawym panelu. Jest to reprezentacja jako blueprintu tej postaci, którą aktualnie gramy po uruchomieniu gry. Wystąpienie tego blueprintu w mapie jest wymagane, aby gra mogła działać poprawnie, co wynika ze sposobu przygotowania szablonu C++ FPS. Jest to tylko jeden z kilku blueprintów używanych w szablonie FPS. Warto go otworzyć i przyjrzeć się temu, co udostępnia natywny kod C++ zawarty w plikach `MasteringCharacter.h/.cpp`, pokazywany w edytorze jako blueprint. Na pierwszy rzut oka wygląda to po prostu jak zbiór zmiennych, z pierwszym wierszem zaczynającym się od **NOTE:**, a kończący się niebieskim łączem do **Open Full Blueprint Editor** (otwórz pełny edytor blueprintów). Pogłębione omówienie planów i odpowiadających im klas oraz interakcji z C++ znajduje się w rozdziale 3, *Przegląd blueprintów i skryptów BP*. Na razie po prostu klikniemy niebieskie łącze, aby zobaczyć, jak te zmienne definiują naszą postać w grze. Rygorystyczny styl C++ szablonu w bardzo niewielkim stopniu wykorzystuje ten blueprint; naprawdę jest to tylko zbiór pewnych zmiennych używanych w szablonie. Jeśli jednak klikniemy teraz kartę **Viewport** (pogląd) w górnej części edytora, będziemy mogli zobaczyć, co robią niektóre z tych zmiennych. Na przykład w prawej części karty **Details** powinno być otwarte wyskakujące okno z etykietą **Camera**. W tym oknie

pierwsza zmienna to **Base Turn Rate** (bazowy krok obrotu) z wyszarzoną wartością 45.0. W kodzie zmienna ta służy do określenia, jak szybko może obracać się nasza postać, ale wartości tej nie można edytować. Przyjrzyjmy się, dlaczego.

Cofnijmy się odrobinę i przyjrzyjmy plikowi `MasteringCharacter.h` w Visual Studio. Przewijając w dół, w jednej z sekcji klas publicznych powinniśmy zobaczyć poniższe wiersze:

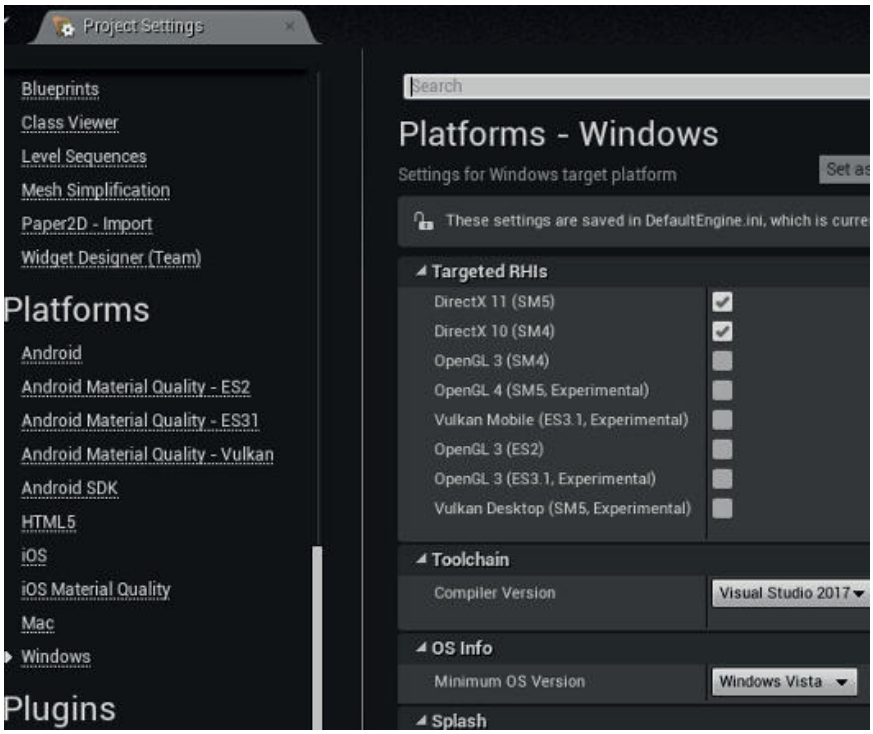
```
/** Podstawowe tempo obrotu w stopniach na sekundę. Inne skalowanie może
wpływać na finalne tempo obrotu. */

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Camera)
float BaseTurnRate;
```

Makro `UPROPERTY` jest metodą wiązania zmiennej C++ z klasą blueprintu, w której można ją zobaczyć i/lub edytować. W tym przypadku wyspecyfikowane w makrze flagi `VisibleAnywhere` oraz `BlueprintReadOnly` są powodem tego, że możemy zmienną zobaczyć, ale nie edytować, zaś `Category=Camera` sprawia, że znalazła się ona w oknie opcji **Camera**. Zmienne w blueprintach są wyświetlane automatycznie na podstawie ich „wielbłądziej pisowni” w kodzie C++: każda wielka litera oznacza początek wyrazu w reprezentacji w blueprincie, jak w przypadku `BaseTurnRate` w naszym przykładzie. W pliku `MasteringCharacter.cpp` możemy zauważyć konstruktor, w którym wyspecyfikowane jest `BaseTurnRate = 45.f`, przez co ta właśnie wartość jest pokazana w edytorze. Będziemy wracać do tych klas i metod ciągle i ciągle, ale teraz potrzebujemy krótkiego przeglądu, aby zrozumieć zmiany, których dokonamy za chwilę.

Nasze kolejne kroki będą obejmować skompilowanie gry w edytorze, zatem trzeba wspomnieć o problemach, które mogą się pojawić, gdy mamy zainstalowane jednocześnie Visual Studio 2015 i VS 2017. Jeśli ktoś nie ma zainstalowanego VS 2015 i używa jedynie VS 2017, może pominąć kolejne akapity aż do procedury tworzenia nowej klasy (na następnej stronie).

Aby wymusić otwieranie każdego pliku w projekcie w Visual Studio 2017, należy wykonać dwa kroki. Po pierwsze, w pliku `readme.md` pobranym z GitHuba można znaleźć instrukcje tworzenia pliku wsadowego lub użycia tego samego polecenia poprzez jego skopiowanie i wklejenie w konsoli wiersza polecenia, aby wygenerować pliki projektu w VS 2017 z poziomu Explorera. Jednak jak wspomniałem, gdy edytor wygeneruje te pliki, domyślnie powiąże je z VS 2015, co może być prawdziwie irytujące (oczywiście sytuacja ta ma miejsce tylko wtedy, gdy mamy zainstalowane obie wersje). Wyobraźmy sobie, że pracujemy w VS 2017 i wywołujemy skompilowanie projektu w edytorze, po czym nasz projekt otwiera się (a raczej próbuje otworzyć) w VS 2015! Aby sobie z tym poradzić, trzeba przejść do zakładki **Settings** (ustawienia) w górnej

części edytora, otworzyć okno **Project Settings**, po czym przewinąć listę poniżej wpisu **Platforms** do węzła **Windows**, jak na kolejnym zrzucie ekranu:



Na tej stronie z listy **Compiler version** wybieramy **Visual Studio 2017** jako używany kompilator, jak na pokazanym ekranie. Powinno to raz na zawsze rozwiązać problem tworzenia niepożądanych plików projektu w wersji VS 2015. Jest to sprawdzane przy zatwierdzaniu kodu na GitHubie i jeśli ktoś używa tylko wersji VS 2015 albo po prostu nie życzy sobie takiego zachowania, może po prostu wrócić do tego samego wpisu **Compiler Version** i ustawić go z powrotem na **Default** albo jawnie na **VS 2015**.

Teraz, gdy znów jesteśmy w głównym oknie edytora, dodamy naszą nową klasę dziedziczącą z `MasteringCharacter`, wykonując poniższą procedurę:

1. W pasku menu kliknij **File** (Plik), po czym wybierz **New C++ Class** (Nowa klasa C++).
2. W wyświetlonym oknie dialogowym **Choose Parent Class** (Wybierz klasę nadrzędną) kliknij pole wyboru **Show All Classes** (Pokaż wszystkie klasy) w prawym górnym rogu.
3. W polu wyszukiwania zacznij wpisywać **MasteringCh**, aż w podpowiedziach pokazywane będzie tylko `MasteringCharacter`, po czym kliknij ten wybór.
4. Upewnij się, że pole **Selected Class** zawiera wpis **Mastering Character**, po czym kliknij **Next** (Dalej).

5. Zmień nazwę nowo tworzonej klasy z **MyMasteringCharacter** na **StealthCharacter**, po czym kliknij **Create Class** (Utwórz klasę).

W kroku 1 można zauważyć, że dostęp do opcji **New C++ Class** umożliwia również menu podręczne, dostępne po kliknięciu prawym klawiszem myszy w normalnym oknie przeglądarki zawartości. Również w kroku 2 warto z czasem nabrać nawyku natychmiastowego zaznaczania pola wyboru **Show All Classes**. Przyjemne jest to, że Epic Games samoczynnie filtruje widok i pokazuje domyślnie tylko kilka klas użytecznych dla nowych użytkowników, jednak w miarę rozrastania się projektu typowo będziemy znacznie częściej używać własnych niestandardowych klas, a nie tych domyślnych.

Edytor powinien poinformować, że kompiluje nowy kod C++, po czym wyświetlić powiadomienie o udanym ukończeniu w prawym dolnym rogu okna. Ważną rzeczą, którą trzeba mieć na uwadze przy tego typu działaniu z przeładowywaniem na gorąco, jest to, że jeśli powrócimy teraz do Visual Studio, będzie on chciał ponownie załadować rozwiązanie projektu gry. Spowoduje to wyświetlenie monitu o zatrzymanie debugowania, a jeśli to potwierdzimy, edytor gry zostanie zamknięty! Techniką, której osobiście często używam, jest dołączanie i odłączanie debugera do edytora w miarę potrzeb. W VS w menu **Debug** znajdziemy polecenie **Detach All** (odłącz wszystko) – lubię powiązać je ze skrótem *Ctrl* + *D*. Pozwala to edytorowi na kontynuowanie działania i możemy wówczas bezpiecznie ponownie ładować rozwiązanie, ilekroć jest to potrzebne. Gdy pojawi się potrzeba zdebugowania jakiegoś fragmentu kodu, możemy ponownie przyłączyć debuger do działającego edytora. W tym celu trzeba wrócić do menu **Debug** i wybrać polecenie **Attach to Process** (podłącz do procesu) – lubię przypisać mu skrót *Alt* + *D*. Następnie klikamy w dużym polu **Processes**, po czym naciskamy klawisz *U* i szukamy wpisu `UE4Editor.exe`. Po podwójnym kliknięciu tego wpisu powracamy do trybu debugowania.

Edytowanie klasy w VS i przeładowywanie edytora na gorąco

Teraz zatem, gdy dodaliśmy już nową klasę dziedziczącą z `MasteringCharacter`, możemy przeedytować jej kod C++ i obejrzeć dokonane zmiany w edytorze. Sugeruję odłączenie debugera zgodnie z opisem w poprzednim akapicie, ale zatrzymanie debugowania i ponowne uruchomienie edytora również nie stanowi problemu, jeśli ktoś woli taki tryb pracy. W przypadku odłączenia trzeba zauważyć, że konieczne jest kliknięcie prawym klawiszem myszy projektu **Mastering** w eksploratorze rozwiązań, wybranie polecenia **Unload Project**, po czym ponowne kliknięcie go prawym

klawiszem i wybranie polecenia **Reload Project**, aby mieć pewność, że cała zawartość będzie zgodna z bieżącym stanem (to znacznie szybsze, niż zamknięcie i ponowne uruchomienie Visual Studio). W eksploratorze rozwiązań możemy teraz zauważyć plik `StealthCharacter.h` oraz kilka plików `.cpp` w węźle **Source/Mastering**. Otwieramy te pliki. Na razie nie ma w nich zbyt wiele, ale za chwilę dodamy nową zmienną, którą będziemy mogli później wyszukać w edytorze. Na początku pliku `StealthCharacter.h` dodajemy poniższe wiersze poniżej wiersza `GENERATED_BODY()`:

```
public:
    // Modyfikator stopnia obracania i podnoszenia w trybie stealth

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Gameplay)
    float StealthPitchYawScale = 0.5f;
```

Ponownie w Visual Studio dodamy teraz poniższy fragment do pliku `StealthCharacter.h` po zmiennej `StealthPitchYawScale` dodanej przed chwilą:

```
public:
    virtual void SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent) override;

    virtual void AddControllerPitchInput(float Val) override;
    virtual void AddControllerYawInput(float Val) override;

    void Stealth();
    void UnStealth();

protected:
    bool bIsStealthed = false;
```

Dalej będziemy postępować zgodnie z wzorcami zawartymi w klasie `MasteringCharacter`, które można dalej przeanalizować, ale mówiąc w skrócie, zamierzamy powiązać nowy mechanizm wejściowy z dwiema funkcjami (`Stealth` oraz `UnStealth`), a następnie nadpisać funkcje klasy bazowej, aby używały danych wejściowych odchylenia i wysokości, aby spowolnić ruch zgodnie z naszą skalą. Zrealizujemy to, dodając poniższy kod do pliku `StealthCharacter.cpp`:

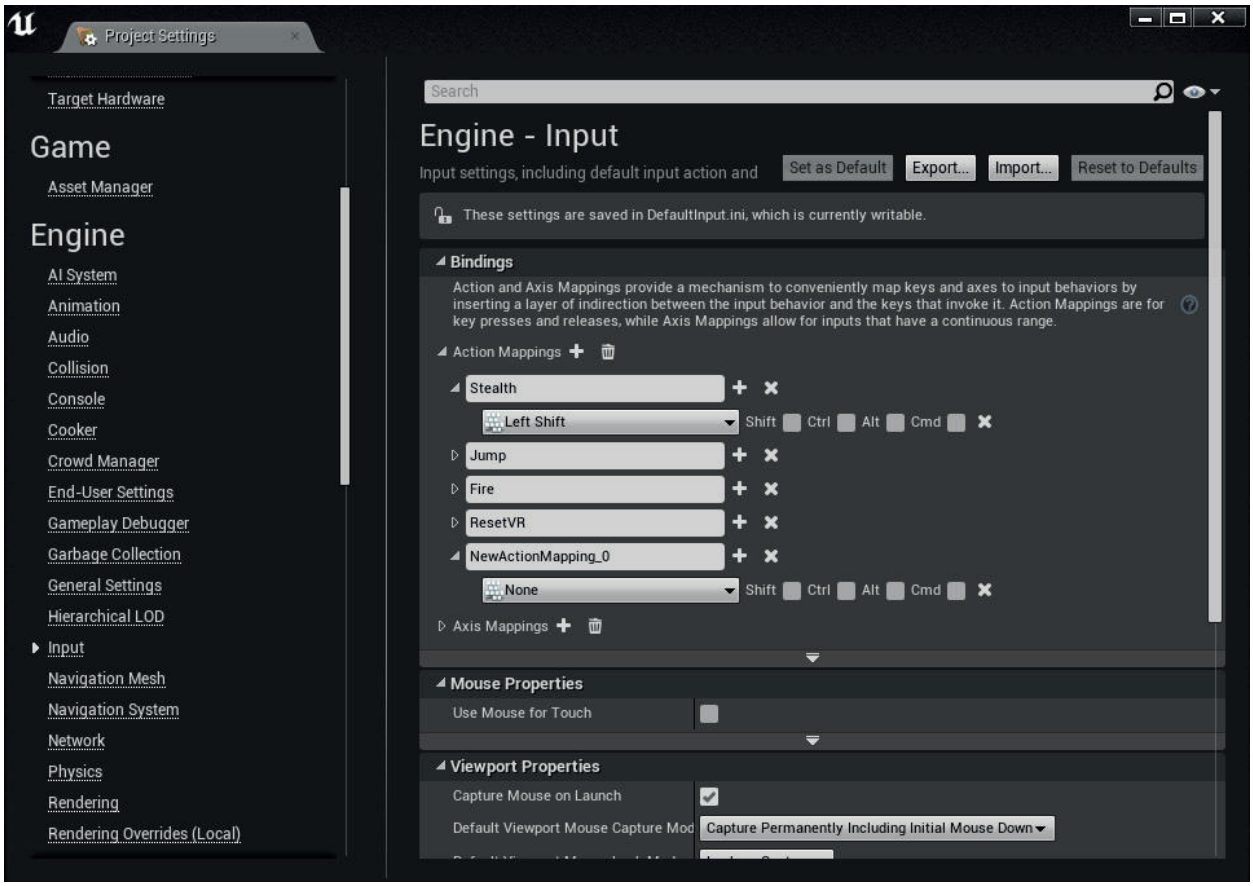
```
void AStealthCharacter::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
{
    // Wiązanie zdarzeń skoku
    PlayerInputComponent->BindAction("Stealth", IE_Pressed, this,
        &AStealthCharacter::Stealth);
```

```
    PlayerInputComponent->BindAction("Stealth", IE_Released, this,
        &AStealthCharacter::UnStealth);
    Super::SetupPlayerInputComponent(PlayerInputComponent);
}

void AStealthCharacter::AddControllerPitchInput(float Val)
{
    const float fScale = bIsStealthed ? StealthPitchYawScale : 1.0f;
    Super::AddControllerPitchInput(Val * fScale);
}

void AStealthCharacter::AddControllerYawInput(float Val)
{
    const float fScale = bIsStealthed ? StealthPitchYawScale : 1.0f;
    Super::AddControllerYawInput(Val * fScale);
}

void AStealthCharacter::Stealth()
{
    bIsStealthed = true;
    Super::Crouch();
}

void AStealthCharacter::UnStealth()
{
    bIsStealthed = false;
    Super::UnCrouch();
}
```

Większość tego kodu powinna być jasna dla doświadczonych programistów C++ wykorzystujących UE4, ale trzeba zauważyć, że nasze nadpisujące funkcje dla `Stealth` i `UnStealth` wywołują istniejące funkcje klasy `ACharacter`, z której ostatecznie dziedziczy nasza klasa. Pozwala to wykorzystać istniejącą już mechanikę przykucnięcia i wyprostowania postaci, oszczędzając nam trudu tworzenia jej samodzielnie. Możemy teraz ponownie skompilować projekt albo dopiero po dodaniu mechanizmu wejściowego, co tak czy inaczej spowoduje restart edytora, zatem trzeba pamiętać, aby zapisać wykonane w nim zmiany!

Istnieją dwa sposoby dodania powiązania z naszym nowym mechanizmem wejściowym. Najlepszą techników jest ponowne otwarcie panelu **Project Settings** na karcie **Settings** w głównym oknie edytora. W panelu tym trzeba przewinąć zawartość do wpisu **Engine** i **Input** w bocznym menu, po czym w prawej części zobaczymy sekcję

Bindings (powiązania), a w niej **Action Mappings** (mapowania działań). Po kliknięciu małego symbolu plus na prawo od **Action Mappings** poniżej niego pojawi się nowy wiersz. Przemianowujemy nowo dodaną **NewActionMapping** na **Stealth**, po czym klikamy mały znak plus na prawo od nowego wiersza **Stealth**. Następnie klikamy menu rozwijane zawierające domyślnie opcję **None** (brak) i przewijamy aż do **Left Shift**, jak na poniższym zrzucie ekranu:



Powiązaliśmy teraz akcję o nazwie **Stealth** z lewym klawiszem Shift na klawiaturze. Spowoduje to dodanie wiersza do pliku `/Config/DefaultInput.ini` gry i jednoczesne uaktualnienie wersji działającej w edytorze. Gdybyśmy zamiast tego ręcznie dodali ten wiersz na samym początku sekcji `[/Script/Engine.InputSettings]` (wiersz ten ma postać `+ActionMappings=(ActionName="Stealth", Key= Left Shift)`) i zapisali ten plik, silnik nie załadowałby automatycznie plików `.ini`, co wymagałoby ponownego uruchomienia edytora w celu pobrania zmian! Warto więc zadbać o to, aby wykonywać edycję ustawień za pośrednictwem okna **Settings**, a jeśli z jakiegoś powodu musimy bezpośrednio modyfikować wpisy w którymś z plików `.ini`, trzeba pamiętać o ponownym uruchomieniu każdego działającego edytora lub samodzielnej wersji gry na naszym komputerze, aby uwzględnić te zmiany.

Musimy jeszcze wykonać jedną, ostatnią zmianę w naszym blueprincie `FirstPersonCharacter`, aby zebrać wszystkie elementy razem, zatem otwieramy go ponownie w oknie edytora blueprintów. W lewym dolnym rogu, poniżej zakładki **Components**, znajdziemy wpis `CharacterMovement (Inherited)`. Po jego kliknięciu możemy teraz zauważyć wiele właściwości w panelu po prawej, ale na razie przewiniemy je, aż znajdziemy opcję **Nav Movement**. Po jej otwarciu w górnej części znajdziemy opcję **Movement Capabilities**, a w niej musimy zaznaczyć opcję **Can Crouch** (może przykucnąć). Zauważmy, że po tej zmianie przycisk **Compile** w lewym górnym rogu zmieni się z zielonego „ptaszka" na pomarańczowy znak zapytania. Klikamy przycisk **Compile**, aby uaktualnić blueprint o wykonaną zmianę, po czym naciskamy *Ctrl* + *S*, aby go zapisać.

Teraz podczas gry, jeśli naciśniemy i przytrzymamy lewy klawisz Shift, punkt widzenia gracza przesunie się w dół o niewielki dystans, co jest realizowane przez kilka istniejących zmiennych klasy nadrzędnej blueprintu, zaś tempo obracania i pochylania zostanie zmniejszone zgodnie z wybraną przez nas wartością `StealthPitchYawScale`. Sukces! Proponuję teraz, aby Czytelnik poeksperymentował z modyfikowaniem wartości tej zmiennej, nawet podczas działania gry i zobaczył, jak bardzo zmienia się prędkość w trybie skradania. Jest to również doskonały moment na ustawienie kilku punktów przerwań w naszych funkcjach i wykonywanie ich krokowo, aby zorientować się, jak to wszystko działa po stronie C++, ale na tym etapie nasza mechanika jest już włączona i sprawdzona.

Podsumowanie

W tym rozdziale przeszliśmy od braku silnika, kodu źródłowego i projektu do posiadania lokalnej wersji silnika UE4 i działającego już wstępnego projektu FPS, a także dodaliśmy kod i nadpisaliśmy funkcje, aby dodać do niej nowe elementy rozgrywki. To doskonały początek, który przeprowadził nas przez początkowe trudności tworzenia gier i zapewnia solidny fundament, na którym będziemy budować w kolejnych rozdziałach.

Teraz przyjrzymy się dokładniej sterowaniu i ulepszeniom, które możemy wykonać na podstawach zbudowanych w tym rozdziale. Nauczymy się też, jak dodawać więcej funkcjonalności gry, w tym stan posiadania i wybór broni. Później przedstawię pogłębione omówienie blueprintów i co one robią dla nas, dlaczego są tak wartościowe, a kiedy mogą stanowić problem. Powracając do naszych początkowych działań w tym rozdziale zajmiemy się interfejsem użytkownika, ładowaniem i zapisywaniem oraz dodawaniem istot AI, zanim szybko przejdziemy do wielu bardziej zaawansowanych i zróżnicowanych tematów w kolejnych rozdziałach!

Pytania

Spróbuj odpowiedzieć na poniższe pytania, aby przetestować zdobytą wiedzę:

1. Jakie korzyści wynikają ze skompilowania silnika z kodu źródłowego?
2. Gdzie można znaleźć edytor kodu źródłowego dla UE4?
3. Jakie kroki zawsze trzeba wykonać po uzyskaniu dowolnej zaktualizowanej wersji UE4, a przed skompilowaniem czegokolwiek?
4. Jak zmienne zadeklarowane w C++ są eksponowane w blueprintach?
5. Jak można szybko dodać i przetestować nowe funkcjonalności bez konieczności tworzenia nowego blueprintu w edytorze?
6. Dlaczego tryb DebugGame jest właściwym wyborem konfiguracji używanej podczas projektowania gry?
7. Dlaczego wymaganie dokonywania zmian w plikach `.ini` w celu dodania nowych funkcjonalności jest kiepskim wyborem?
8. Jaki krok trzeba wykonać przed zapisaniem blueprintu, gdy zmieniamy jakąś właściwość planu?

Dodatkowa lektura

https://docs.unrealengine.com/en-us/Programming/Introduction

2

Wyposażenie i broń gracza

Wprowadzenie

Witam w kolejnym rozdziale. Zajmiemy się w nim rozbudową naszego projektu Mastering, tak jak wygląda do tej pory, o zupełnie nowy system obsługi wyposażenia i zmieniania broni. Ilekroć będzie to możliwe, zostaną również usunięte dowolne zakodowane na stałe typy systemów, takie jak odwoływanie się w kodzie C++ do zasobów przez ich nazwy. Przy wykonywaniu takich zmian typowo zamieścimy wyjaśnienia, dlaczego jest to ważne w naszym projekcie. Przed ukończeniem tego rozdziału jednak powinniśmy móc wskazać wszystkie nasze klasy i zasoby w łatwy do modyfikowania sposób i uzyskamy bardzo krótki czas kolejnych iteracji, zmieniając takie rzeczy, jak dodawanie nowych broni i sposobów ich wybierania. Omówimy zatem następujące zagadnienia:

- Dodawanie klasy `Weapon`
- Tworzenie magazynu ekwipunek (broni) w C++
- Tworzenie i wykorzystywanie klasy `WeaponPickup`
- Kolejne przełączanie broni przy użyciu nowych powiązań sterowania

Wymagania techniczne

Pozostają w mocy wszystkie wymagania przedstawione w rozdziale 1, *Tworzenie projektu C++ dla strzelanki*, włącznie z utworzonym w nim projektem Mastering albo jakimś innym w podobnym punkcie rozwoju. Cała praca wykonana w tym rozdziale dostępna jest w gałęzi projektu na GitHubie pod adresem:

> *https://github.com/PacktPublishing/Mastering-Game-Development-with-Unreal-Engine-4-Second-Edition/tree/Chapter-2*

Używana wersja silnika: 4.19.0.

Dodawnie klas Weapon i Inventory

Naszym celem w tym podrozdziale będzie dodanie do naszej gry dwóch nowych klas i przekonwertowanie pochodzących z szablonu zakodowanych na sztywno broni na nową klasę i dodanie ich do nowej klasy ekwipunku dla naszego gracza. Zaczniemy od uruchomienia edytora gry i poświęcimy nieco czasu na zbadanie istniejącej broni, aby zobaczyć, jak zostało to wykonane w szablonie. W ten sposób zbierzemy informacje potrzebne do zaprojektowania i zaimplementowania tej nowej klasy broni.

Tworzenie klasy Weapon

Choć szablon FPS zapewnia doskonały punkt wyjściowy dla takiego projektu, jak ten, który realizujemy w tej książce, jest bardzo ograniczony pod wieloma względami. Ma to na celu zapewnienie możliwie minimalnej, agnostycznej implementacji pozwalającej nam, czyli deweloperom, rozbudowywanie projektu w dowolnym wymaganym kierunku. Motywem przewodnim tego rozdziału jest ulepszanie i rozbudowa tego projektu w miarę pojawiania się zapotrzebowania na nowe systemy i funkcjonalności, taka konfiguracja wstępna powinna być mocną motywacją do pracy. Bardzo prosty system broni zaimplementowany w szablonie pozwala zademonstrować wszystkie kluczowe elementy uzbrojenia w grze FPS, jednak nie jest on łatwy do modyfikowania (słuszniejsze byłoby powiedzenie, że jest to niewykonalne). Tym samym potrzebujemy nowej klasy. W typowej strzelance nasza postać będzie często wymieniać wiele unikatowych rodzajów uzbrojenia, zatem teraz zajmiemy się realizacją tego zadania krok po kroku.

Aby zobaczyć, jak wykonana została istniejąca broń, trzeba otworzyć gałąź **Content** | **FirstPersonCPP** | **Blueprints** | **FirstPersonCharacter** w panelu Content Browser, a jeśli