Podstawy tworzenia gier w unity

Ten rozdział poświęcony jest podstawowym elementom tworzenia gier w Unity. Trzeba zaznaczyć, że tworzenie pełnoprawnej gry wymaga bardzo szerokiego zakresu informacji, zaś niniejszy rozdział skupia się tylko na aspektach niezbędnych do stworzenia grywalnego tytułu.

Planowanie

Jednym z decydujących czynników przy rozpoczynaniu pracy nad grą jest odpowiednie zaplanowanie procesu jej tworzenia. Podstawą jest opisany w trzecim rozdziale dokument GDD, który zawiera w sobie wszelkie niezbędne informacje, poczynając od opisu gier, kończąc na narzędziach potrzebnych do pracy nad nią. Jest to aspekt niezwykle ważny, gdyż może się okazać, że np. silnik który wybraliśmy nie jest w stanie sprostać zaawansowanym obliczeniom fizycznym, portowanie gry na inną platformę jest zbyt czasochłonne, bądź jakość renderowanej przezeń grafki jest za niska.
W niektórych silnikach optymalizacja gry jest znacznie trudniejszym zadaniem niż w innych, co ma duże znaczenie jeśli tworzymy grę mobilną. Ważny jest także format danych z których będziemy korzystać, np. modeli 3D, oraz czy rozgrywka skupiać będzie się na grze dla pojedynczego gracza, czy na grze wieloosobowej, ze względu na różne implementacje obsługi sieci w silnikach.
Przykładowy typ gry odpowiedni dla silnika Unity: mała gra, tworzona przez mały niedoświadczony zespół.
Gra ma być platformówką, działać dobrze na telefonach, ale zostać też wypuszczona na konsole i pecety i nie zawierać rozgrywki multiplayer. Grafka 3D mocno stylizowana/uproszczona, bez grafcznych fajerwerków.

Powyższe wymagania idealnie łączą się z tym co ma do zaoferowania Unity: prostota obsługi, wsparcie aż 21 platform, możliwość importowania/eksportowania paczek, obsługa wielu formatów modeli 3D i narzędzia do ich obsługi. Dzięki temu, że gra ma posiadać prostą grafkę nie musimy przejmować się faktem, że Unity ustępuje innym silnikom w kwestii fizyki i zaawansowanego renderingu 3D. W fazie planowania nie powinno się także zapominać, że tworzenie gier jest bardzo czasochłonne, dlatego warto wyznaczyć sobie ramy czasowe dla danego projektu. Pozwoli to określić czy proces produkcji przebiega prawidłowo i uniknąć częstego powodu śmierci gry jakim jest za długi czas produkcji i brak środków na kontynuowanie prac.

Początek pracy z Unity

Silnik pobieramy z oficjalnej strony. Od wersji 5 Unity, po odpaleniu instalatora i zaakceptowaniu regulaminu, sami decydujemy o tym, które komponenty zostaną zainstalowane. Pozwala to pominąć nieistotne dla nas opcje. Warto zaznaczyć, że o ile sam silnik jest darmowy, to możliwość wydawania gier na niektóre platformy wymaga posiadania odpowiedniej licencji. Przy tworzeniu nowego projektu, możemy zaznaczyć, czy nasza gra będzie w 2D, czy w 3D. Zaznaczenie odpowiedniej opcji zaimportuje nam odpowiednie podstawowe assety (materiały używane przy tworzeniu gier), oraz ustawi odpowiednią konfgurację.
Wybraną opcję można później zmienić.

Podstawowe terminy

Assets/assety - to reprezentacja każdego pliku, który może być użyty w grze. Asset może być dowolnym plikiem zewnętrznym np. dźwiękiem lub modelem 3D wspieranym przez Unity, a także wewnętrznym, tworzonym w Unity takim jak scena lub renderer.

Game Objects/obiekty gry – to podstawowe obiekty w Unity, które reprezentują wszystkie elementy

widoczne w grze. Same z siebie nie robią nic, za to ich rolą jest bycie kontenerem na tzw. Komponenty, która implementują ich prawdziwą funkcjonalność.
Przykładowo poruszająca się dwuwymiarowa piłka, powstaje poprzez dodanie do Game Objectu komponentu do renderowania grafki 2D, odpowiedniego skryptu i animacji.
Component/komponenty – są podstawą każdego obiektu i zachowania w grze. Są częścią funkcjonalną każdego Game Objectu i są bezpośrednio doń przypisywane.
Każdy obiekt domyślnie posiada komponent Transform, który opisuje jego pozycję, skalę i rotację. Bez tych informacji obiekt byłby nie do zlokalizowania na scenie.

Używając wyrażenia obiekt i komponent w tej pracy, będziemy każdorazowo odnosić się odpowiednio do wyżej zdefniowanych Game Objectów i Componentów.

Ekran główny

Po stworzeniu projektu naszym oczom ukaże się główny ekran. Jeśli żaden projekt nie był wcześniej otwierany na tym komputerze to ekran będzie wyglądał jak na zdjęciu poniżej (dla wersji 5.x).

<ekran_glowny.jpg>

Cyfry na zdjęciu oznaczają poszczególne okna:

1. Pasek Menu – daje dostęp do bardziej zaawansowanych opcji, min. pozwala na dodawanie nowych okien, komponentów czy konfgurowanie całej aplikacji.
2. Project – tu umieszczone są wszystkie pliki jakie znajdują się w grze. Pliki możemy dodawać do projektu, przenosząc je bezpośrednio do tego okna, lub dodając je do folderu „Assets” w naszym projekcie.
3. Scene – jest to przestrzeń 3D na której umieszczane są wszystkie obiekty znajdujące się w grze.
4. Game – okno pozwalające na zobaczenie naszej gry w akcji po naciśnięciu przysiku „Play”. Domyślnie widać tylko niebieskie tło – jest to aktualny obraz z głównej kamery umieszczonej na scenie.
5. Hierarchy – w tym oknie znajduje się lista wszystkich obiektów znajdujących się na scenie. Pozwala nam na manipulowanie obiektem bez konieczności szukania go bezpośrednio na scenie.
6. Inspector – wyświetla właściwości aktualnie zaznaczonego obiektu.
7. Pasek opcji – pozwala na zmianę opcji poruszania się po scenie, zmianę widoku przy zaznaczaniu obiektu, odpalenie naszej gry, zarządzanie warstwami i zmianę layoutu.

Ułożeniem poszczególnych okien można swobodnie manipulować poprzez przenoszenie zakładek lub zmianę layoutu z paska opcji. Możliwe jest też dodawanie nowych zakładek. Przy tworzeniu nowego projektu Unity wczyta ustawienia okien z poprzednio otwartego projektu, zaś przy otwieraniu innego projektu, okna ustawione będą tak jak ustawiła je osoba przy nim pracująca. Wszystkimi oknami można zarządzać klikając na nie, bądź korzystając z zakładki „Window” z paska menu (1).

Praca z obiektami i sceną

Scena jest miejscem na którym umieszczać będziemy każdy element gry. Aby zapewnić sobie swobodną pracę, niezbędne jest opanowanie podstawowych zasad poruszania się po niej. Jak w każdej przestrzeni 3D podstawowym sposobem przemiszczania się po scenie jest

oddalanie/przybliżanie widoku oraz obracanie go w dowolnym kierunku. Po kliknięciu na scenę, aby przybliżyć bądź oddalić kamerę korzystamy z pomocy rolki myszki, lub wciskamy klawisz alt i przytrzymujemy prawy przycisk, poruszając jednocześnie myszką w tył lub przód. Aby rotować widok, przytrzymujemy alt i lewy przycisk poruszając myszką w dowolnym kierunku. Aby mieć na czym operować niezbędne jest stworzenie Game Objectu. Aby to zrobić klikamy lewym przyciskiem myszki w oknie Hierarchy (5), lub klikając Game Object z paska menu (1) i i wybieramy „Create empty".

Stworzy nam to na scenie pusty obiekt, a w hierarchii pojawi się jego nazwa i możliwość zaznaczenia go. Po zaznaczeniu zobaczymy jego pozycję na scenie reprezentowaną przez mały sześcian i 3 strzałki różnego koloru. Kolory strzałek oznaczają różne osie - zielona oś Y, symbolizująca ruch góra/dół, czerwona oś X, symbolizująca ruch lewo/prawo, oraz niebieska oś Z symbolizująca głębię.

Jako, że obiekt posiada już domyślny komponent Transform, możliwe są operacje na jego prezentacji w przestrzeni. Na pasku opcji (7), poczynając od lewej znajduje się 5 przycisków do wykonywania tych operacji i odpowiedniego przełączania się między nimi. Wszystkie tryby można uruchamiać także przy pomocy skrótów klawiszowych.

Pierwszy przycisk od lewej pozwala nam na poruszanie samym widokiem w przestrzeni przy pomocy myszki, nawet gdy obiekt jest zaznaczony.

<drag_option.jpg>

Drugi przycisk umożliwia nam na zmienianie pozycji obiektu w przestrzeni. Możemy swobodnie poruszać obiektem klikając na kwadrat, bądź jeśli chcemy poruszać obiekt po konkretnej osi – na odpowiednie strzałki. Można uruchomić ten tryb korzystając z przycisku „W" na klawiaturze.

<move_option.jpg>

Trzeci przycisk służy to zmiany rotacji obiektu w przestrzeni. Po uruchomieniu tego trybu zobaczymy, że wygląd osi zmienił się na kształt sfery. Rotację wykonujemy ta samo jak zmianę pozycji poprzez klikanie na odpowiednie osie.

<rotate_option.jpg>

Czwarty przycisk służy do zmiany skali obiekty. Zmianę wykonujemy poprzez naciśnięcie na małe szcześciany widoczne na osiach.

<scale_option.jpg>

Ostatni przycisk służy do obsługi tzw. Rect Transform i wychodzi poza zakres podstawow i nie będzie omawiany w tym rozdziale.

<recttrans_option.jpg>

Mimo, że puste obiekty same w sobie nic nie robią to są używane równie często, co obiekty z komponentami. Wynika to z faktu, że można je zagnieżdżać tak samo jak foldery – przenosząc jeden obiekt na drugi w oknie hierarchii. Ma to kilka zastosowań. Po pierwsze pozwala na zorganizowanie obiektów na scenie, która może składać się z tysięcy obiektów i nawigacja między nimi mogłaby być bardzo kłopotliwa. Przykładowo posiadając obiekt „samochód", obiekty takie jak „koło", „karoseria" , „lampy" itd. powinny być umieszczone w tym obiekcie. Dzięki temu,

poruszając „samochodem”, jednocześnie przemieścimy o tę samą wartość „koła”, „karoserię” i „lampy”. Te obiekty nazywane są obiektami potomnymi, ang. Child Objects. Samochód jest w tym momencie rodzicem, ang Parent Object. Ma to szczególne znaczenie przy skomplikowanych obiektach, gdzie przemieszczanie każdego z nich oddzielnie byłoby bardzo niepraktyczne. Ta sama zasada dotyczy się rotacji i skali. Modyfikacja obiektu potomnego nie wpływa za to na jego rodzica. Drugim ważnym zastosowaniem obiektów potomnych jest możliwość odwoływania się do nich poprzez rodzica. Oznacza to, że skrypty umieszczone w obiekcie, posiadającym jedno lub wiele dzieci, mogą za jednym zamachem kontrolować wszystkie z nich.

<przyklad_relacji_rodzic_dziecko.jpg>

Każdy obiekt który umieszczamy na scenie automatycznie pojawia się w oknie hierarchii. Zaznaczenie obiektu w hierarchii zaznaczy nam obiekt na scenie i vice versa, a także wyświetli wszystkie komponenty podpięte do tego obiektu w oknie inspektora (4).

Okno inspektora i komponenty

Unity daje nam dostęp do kilku niepustych obiektów pozwalających na szybkie rozpoczęcie pracy. Aby stworzyć taki obiekt klikamy na zakładkę „GameObject” z paska menu (1). Do wyboru mamy kilka obiektów. Wybieramy 3D Object → Cube, co stworzy nam na scenie obiekt w kształcie szarego sześcianu i automatycznie go zaznaczy. W oknie inspektora (5) poza komponentem Transform dostrzeżemy też trzy inne. Mesh Filter pobiera informacje na temat siatki modelu i przekazuje je do ostatniego komponentu, Mesh Renderera, który tę siatkę renderuje. Box Collider tworzy kolider wokół siatki który pozwala na interakcje z innymi obiektami – zostanie on omówiony w dalszej części tego rozdziału. Większość komponentów zawiera w sobie dodatkowe opcje umożliwiające modyfikację ich działania. Przykładowo zmiana wartości x, y lub z w komponencie Box Collider zmieni jego rozmiary podobnie jak właściwość Scale w komponencie Transform. Komponenty można dodawać, usuwać, zmieniać ich kolejność oraz kopiować, klikającym prawym przyciskiem na jego nazwie, klikając przycisk Add component na dole inspektora. Przykładowo, usunięcie komponentu Mesh Renderer z naszego obiektu, sprawi, że jego ściany przestaną się wyświetlać jednak nadal będą możliwe kolizje z innymi obiektami.
Istnieje też możliwość wyłączenia danego komponentu (nie będzie on działał, ale nie będzie też usunięty) poprzez kliknięcie checkboxa oraz ukrycie właściwości przy pomocy małej strzałeczki, oba usytuowane tuż obok jego nazwy.

Za każdym razem gdy zaznaczymy nowy obiekt, w inspektorze wyświetlą się jego komponenty. Klikając na małą ikonkę w górnym prawym rogu inspektora, możemy zablokować widok na komponentach aktualnie wybranego obiektu.

Skrypty

Unity umożliwia nam rozszerzanie swojej funkcjonalności oraz kontrolę zachowań poprzez skrypty. Do wyboru mamy trzy języki – C#, Unity Script (zmodyfikowana wersja Java Script) oraz Boo, który jednak przestał być wspierany od wersji 5.0 . Zaleca się aby w ramach danego projektu używać tylko jednego języka aby uniknąć problemów z kompatybilnością.
Największą popularnością cieszy się C#. Może też pochwalić się najlepszą dokumentacją, dlatego jest zalecany osobom zaczynającym pracę z Unity. Także wszystkie skrypty wykorzystane w tej pracy napisane zostały w tym języku.

Skrypty tworzymy klikając prawym przyciskiem myszy w oknie projektu i wybierając opcję Create → C# Script z menu kontekstowego. Tak samo tworzymy foldery oraz wewnętrzne assety. Po wpisaniu nazwy (zaczynającej się dużą literą, gdyż mamy do czynienia z nazwą klasy) możemy

kliknąć dwukrotnie lewym przyciskiem na skrypt, aby go otworzyć. Domyślnie uruchomi to środowisko programistyczne Mono Develop (możemy korzystać z dowolnego edytora, wymaga to jednak wcześniejszej konfiguracji, lub bezpośredniego otwierania pliku z folderu projektu). Każdy nowy skrypt posiada taki szablon (można go zmienić w pliku ScriptTemplates w folderze Unity):

```
<code>
using UnityEngine;
using System.Collection;

public class NazwaSkryptu: MonoBehaviour {

	//use this for initialization
	void Start () {

	}

	//Update is called once per frame
	void Update() {

	}
}
</code>
```

Każdy skrypt domyślnie dziedziczy po klasie MonoBehaviour, która udostępnia wszystkie funkcje jakie oferuje nam Unity. Ta klasa jest niezbędna aby móc dodawać skrypty do Game Objectów.

Funkcja Start() wywołuje się zawsze, jednorazowo przy załadowywaniu nowej sceny. Jest wykorzystywana do inicjalizacji wszelkich ustawień i zachowań na początku gry. Jeśli na scenie znajduje się wiele obiektów z podpiętymi różnymi skryptami posiadającymi tę funkcję, dla każdego z nich ta funkcja wykona się raz, jednak kolejność ich wykonywania zależna będzie od rodzaju i specyfikacji urządzenia. Jeśli w jednym Starcie użyty będzie kod, który wymaga wpierw inizjalizacji Startu z innego skryptu może wystąpić tzw. efekt wyścigów. Może to spowodować wyświetlanie błędów, lub prowadzić do niechcianych i trudnych do przewidzenia zachowań.

Aby pomóc w uniknięciu tego typu problemów Unity udostępnia, trzy dodatkowe funkcje tego typu, które różnią się przede wszystkim kolejnością ich wykonywania dla wszystkich skryptów:
Awake() - wykonuje się jako pierwsze
OnEnable – wykonuje się gdy obiekt staje się aktywny, jeśli jest on aktywny od samego początku, to wykonuje się on tuż po Awake()
OnLevelWasLoaded() - wykonuje się tuż przed Startem, jednak nie bezpośrednio po starcie gry, a dopiero po zmianie sceny
Start() - wykonuje się jako ostatnia, tuż przed startem pierwszej klatki funkcji Update() jeśli obiekt jest aktywny

Istnieje specjalna funkcja DontDestroyOnLoad(), która powoduje, że obiekt nie jest niszczony pomiędzy scenami, oznacza to, że niektóre z powyższych funkcji nie wykonają się po zmianie sceny.

Funkcja Update() jest wywoływana co klatkę dla każdego obiektu z podpiętym skryptem. Wszystkie akcje w grze wymagające ciągłego wykonywania tego samego kawałka kodu, pisane są wewnątrz tej funkcji. Podobnie jak Start(), Update() posiada dwie funkcje tego samego typu:

FixedUpdate() - wykonuje się jako pierwsze. Wszystkie akcje związane z fizyką powinny być pisane wewnątrz tej funkcji. Jest to powodowane tym, że częstotliwość odświeżania klatek może się różnić w zależności od sprzętu i faktu, że w niektórych miejscach nasza gra może bardziej obciążać zasoby sprzętowe, a w innych mniej. W tej funkcji czas pomiędzy poszczególnymi klatkami jest zawsze taki sam, a obliczenia fizyczne opierają się na stałych wartościach. Losowy czas pomiędzy klatkami może powodować dziwne zachowania wynikające z problemów w obliczeniach.

Update() - wykonuje się drugie w kolejności. Wszystkie pozostałe akcje niezwiązane z fizyką powinny być pisane w tej funkcji.

LateUpdate() - jw. z tym, że wykonuje się dopiero gdy dana klatka została wykonana wcześniej we wszystkich Updateach

Powyższe funkcje są jednymi z najbardziej podstawowych i najczęściej wykorzystywanych funkcji w Unity, dlatego ich znajomość jest taka ważna.

Wszystkie komponenty są tak naprawdę skryptami. Oznacza to, że nie tylko możemy odwoływać się w napisanym przez nas skrypcie do dowolnego komponentu, ale możemy podpiąć dowolny skrypt do każdego obiektu w grze. Operując modyfikatorami dostępu możemy decydować do której właściwości możemy uzyskać dostęp i umożliwić jej zmianę z poziomu inspektora. Przeanalizujmy poniższy kod:

<code>

```
using UnityEngine;
using System.Collections;

public class Move : MonoBehaviour {

	int speedX = 3;
	public int speedY = 3;
	[SerializeField]
	int randomNumber = 8;
	void Update () {
	gameObject.transform.Translate (Time.deltaTime * speedX, Time.deltaTime * speedY, 0);

	}
}
```

</code>

Na początku zdefiniowaliśmy trzy zmienne. Pierwsza zmienna jest prywatna (jeśli nie zadeklarowaliśmy jawnego modyfikatora, to domyślnie zmienna jest prywatna). Prywatne zmienne nie są widziane przez inne skrypty. Nie są także widoczne z poziomu inspektora. Druga zmienna jest zmienna publiczną i można się do niej odwołać z poziomu innych skryptów. Dodatkowo będzie ona widoczna w inspektorze, i możliwa będzie jej modyfikacja bez konieczności jej zmiany w skrypcie. Raz tak zmieniona wartość jest na stałe zapamiętywana, do momentu jej ponownej zmiany w inspektorze lub modyfikacji kodu. Jeśli ta wartość zmienia się podczas samej gry, to ta zmiana zapamiętywana jest tylko do momentu zmiany sceny, bądź wyłączenia gry.

Mimo, że ostatnia zmienna także jest zmienną prywatną, to pole [SerializeField] sprawia, że będzie ona widoczna w inspektorze. Dalej jednak możliwość jej edycji będzie niedostępna.

Aby zapamiętać wartość zmiennej pomiędzy scenami należy użyć modyfikatora static, bądź skorzystać ze specjalnej klasy PlayerPrefs.

W funkcji Update(), która jak wiemy jest wykonywana co klatkę dodaliśmy możliwość poruszania

się dla obiektu. Oto jak działą ta linia kodu: gameObject wkazuje na obiekt do którego aktualnie jest podpięty nasz skrypt, a transform odwołuje nas do komponentu Transform tego obiektu. Translate() jest nazwą funkcji w klasie Translate z której korzystamy. Translate przyjmuje parametr typu Vector3 (wkazuje on na punkt o danych x,y,z w przestrzeni trójwymiarowej). W tym wypadku podaliśmy wartości naszych zmiennych pomnożone przez wartość Time.deltaTime. Zrobiliśmy to, ponieważ bez tego nasz obiekt zmieniałby pozycję o 3 punkty w górę i 3 punkty w prawo co wyświetlaną klatkę. Ilość klatek w jednej sekundzie może sięgać nawet kilkuset, dlatego nasz obiekt poruszałby się z tak dużą prędkością, że byłoby to niewidoczne dla oka. Time.deltaTime dzieli naszą wartość odpowiednio do ilości wyświetlanych klatek, dzięki czemu nasza pozycja będzie zmieniać się co sekundę zamiast co klatkę.

Skrypt podpinamy zaznaczając nasz obiekt (np. cube), i przenosząc skrypt z okna Projects do inspektora, lub klikając Add component i wyszukując nasz skrypt. Aby zobaczyć efekt działania skryptu nasz obiekt musi znaleźć się w polu widzenia kamery. Domyślnie jeśli pozycja kamery nie była zmieniana, to umieszczenie obiektu w pozycji x=0, y=0,z=0 ustawi go na środku pola widzenia. Po naciśnięciu przycisku Play ujrzymy efekt działania naszego skryptu.

<moving_cube.jpg>

Kamera, światło i Collidery

Nawet bezpośrednio po stworzeniu nowego projektu nasza scena nie jest całkowicie pusta. W oknie hierarchii znajdziemy dwa obiekty: Main Camera oraz Directional Light. Pierwsze służy do przechwytywania i wyświetlania elementów sceny które ma widzieć gracz. Tak naprawdę poruszając się po scenie również korzystamy z kamery aby widzieć określoną część sceny. Ilość kamer na scenie jest nieograniczona. Przy pomocy skryptów możemy przemieszczać kamery i dodawać do nich różne efekty. Odpowiednie ustawienie widoków pozwala nam na podgląd kilku, nawet zupełnie oddalonych od siebie fragmentów sceny na jednym ekranie. Dobrym przykładem wykorzystania dodatkowej kamery jest minimapa, używana często w strategiach czasu rzeczywistego lub w grach fabularnych. Main Camera jak sama nazwa sugeruje pełni główną rolę w wyświetlaniu akcji gry i tylko ona wykorzystywana była w tej pracy.

Directional light służy do generowania odpowiedniego oświetlenia na scenie. Istnieje kilka rodzajów świateł i ich odpowiednie ustawienie jest kluczowe przy budowaniu świata gry. Directional Light cechuje się głównie tym, że jego położenie na scenie nie ma najmniejszego znaczenia, cała scena oświetlana jest równomiernie,za to rotacja ma wpływ na to z której strony to światło pada. Ten rodzaj światła praktycznie zawsze reprezentuje światło słoneczne na otwartych przestrzeniach. W zależności od pory dnia można modyfikować jego intensywność oraz inne parametry. Wbrew pozorom usunięcie światła ze sceny nie spowoduje, że obiekty będą kompletnie niewidoczne, a powiązane jest to z tzw. Ambient Light, opis którego nie wchodzi w zakres tego rozdziału.

Collider jest to komponent, który definiuje fizyczny kształt danego obiektu, niezbędny do wykrywania kolizji z innymi obiektami. Często przybiera on kształt siatki modelu który został podpięty do danego obiektu. Ze względu na fakt, że skomplikowane kolidery są bardzo procesożerne, bardzo często korzysta się z tzw. prymitywnych koliderów – w kształcie sfer, sześcianów i kul. Obiekt może posiadać dowolną liczbę ich liczbę, dlatego skomplikowane modele z reguły składają się z wielu prymitywnych koliderów. Efekt tego bardzo często widać w grach, gdy część postaci przenika przez ścianę, lub jesteśmy blokowani przez przeszkodę nie dotykając jej. Odpowiednie ustawienie koliderów ma więc kluczowe znaczenie i pozwala uniknąć tzw. Glitchy, gdy w pewnych miejscach obiekty przechodzą przez siebie mimo że nie powinny. Collidery opierają się mocno na fizyce, co także bywa przyczyną błędów, np. gdy obiekty zderzają się ze sobą

przy zbyt dużej prędkości. Bardziej zaawansowane kolidery takie jak Mesh Collider, zapewniają lepszą dokładność niż kolidery prymitywne, jest to jednak okupione większym zużyciem zasobów sprzętowych.

Collidery dzielą się jeszcze na dwa oddzielne typy niezależnie od ich kształtu. Są to tzw. Trigger Collidery i Collision Collidery. Pomiędzy tymi dwoma typami możemy przełączać się przy pomocy checkboxa „isTrigger" we właściwościach komponentu. Różnica między nimi polega na tym, że w przypadku tego pierwszego, gdy dwa obiekty zetkną się to zostanie to odnotowane i przy pomocy odpowiedniej funkcji (OnTriggerEnter()) będziemy mogli je obsłużyć, lecz obiekty przenikną się i nie będą miały wpływu na swoją pozycję w przestrzeni. W wypadku gdy opcja „isTrigger" jest odznaczona, oba obiekty zderzą się i w zależności od ich parametrów fizycznych nastąpi odpowiednia reakcja. Ten typ zderzenia obsługujemy funkcją OnCollissionEnter().

Jako, że Collidery opierają się na fizyce, obiekty z nich korzystające muszą posiadać odpowiednie cechy fizyczne. Aby nadać te cechy obiektowi musimy dodać do niego komponent Rigidbody. Pozwala on min. na ustalenie masy obiektu i czy działa na niego grawitacja.

Co ciekawe Unity korzysta aż z dwóch systemów do obsługi fizyki, jeden dla obiektów 2D i drugi dla obiektów 3D. Każdy komponent korzystający z fizyki 3D ma swój odpowiednik 2D. Aby uzyskać do niego dostęp przy nazwie komponentu/funkcji dopisujemy na końcu 2D, np. Rigidbody2D lub OnCollisionEnter2D. Obiekty 2D nie mogą kolidować z obiektami 3D, a korzystanie z funkcji dedykowanej 2D, przy trójwymiarowym modelu korzystającym z kolidera 3D zwróci błąd.

Korzystając z koliderów trzeba bardzo uważać by nie zagnieżdżać wielu koliderów w jednym miejscu i unikać ich nadużywania, gdyż może to prowadzić do poważnych problem wydajnościowych.

Poniższy screen przedstawia przykładowy model korzystający z kilku podstawowych koliderów:

<model_prymitywne_kolidery.jpg>

Temat Colliderów zamyka podstawowe zagadnienia związane z Unity.