ChatGPT

Návod: Tvorba jednoduché hry v Unreal Engine 5.5.4

Tento podrobný návod vás provede vytvořením jednoduché hry v Unreal Engine 5.5.4 na základě šablony (např. Third Person nebo Top Down template). Ukážeme si úpravu existujících objektů, tvorbu nových Blueprintů a herních objektů, implementaci sbírání předmětů s bodovým skóre, ukončení hry při dosažení cíle, detekci převrácení vozidla, práci s proměnnými, slučování objektů do jednoho modelu, vytváření materiálů, používání Blueprint událostí a nastavení kolizí. V návodu jsou uvedeny praktické kroky, důležité pojmy jsou zvýrazněny a nechybí odkazy na oficiální dokumentaci Epic Games pro další informace.

Úprava existujících objektů ve scéně

Po vytvoření projektu ze zvolené šablony (např. Third Person Template) se ve výchozí úrovni nachází několik objektů (Actorů), jako je podlaha, stěny, světla, Player Start apod. **Pro úpravu existujících objektů** ve scéně je nejprve vyberte ve Viewportu nebo v seznamu **World Outliner**. Poté můžete v **Details** panelu měnit jejich vlastnosti, například polohu, rotaci a měřítko: - **Přemístění, natočení či změna velikosti objektu:** Použijte transformační nástroje (gizmos) – šipky pro posun, kroužky pro rotaci a kostku pro změnu měřítka. Tyto nástroje umožňují intuitivně manipulovat s objekty přímo ve Viewportu [1]. Můžete také přímo zadat číselné hodnoty transformace v **Details** panelu pro přesné nastavení. - **Změna materiálu objektu:** Pokud chcete existujícímu objektu přiřadit jiný materiál, vyberte objekt a v jeho **Details** panelu najděte sekci **Materials** (Materiály). Zde klikněte na aktuální materiál u daného mesh komponentu a z nabídky vyberte jiný materiál (případně zvolte nový materiál asset z Content Browseru). Tím objektu změníte povrchovou úpravu.

Např. můžete vybrat podlahu a v **Details** panelu změnit její *Scale* (měřítko) na větší rozměr nebo jí přiřadit jiný materiál, čímž změníte vzhled úrovně. Podobně lze přesouvat startovní pozici hráče nebo otáčet světla pro změnu nasvícení scény. Krátké úrovně s jednoduchými úpravami prostředí vám pomohou seznámit se s ovládáním editoru.

Vytvoření nového sběratelského objektu (mince)

Nyní vytvoříme **nový herní objekt** – například sběratelskou **minci**, kterou bude hráč ve hře sbírat. V Unreal Engine je nejlepší takovýto interaktivní objekt vytvořit jako **Blueprint Class** typu Actor. Postup je následující:

1. **Vytvoření Blueprintu:** V Content Browseru klikněte na tlačítko **Add** (**+**) a zvolte **Blueprint Class**. V dialogu vyberte jako základní třídu *Actor* (aktér). Pojmenujte nový Blueprint například **BP_Coin** (Mincička). Otevře se editor Blueprintu.
2. **Přidání statického mesh komponentu:** V novém Blueprintu **BP_Coin** v levém panelu **Components** klikněte na **Add Component** a přidejte komponentu **Static Mesh**. Tato komponenta bude reprezentovat vizuální model mince ve hře [2]. V **Details** panelu pro Static Mesh component vyberte

konkrétní mesh – můžete zvolit například základní tvar (např. válec nebo kouli) a upravit jeho měřítko, aby připomínal minci (tenký disk). Přejmenujte komponentu třeba na *CoinMesh*.

3. **Přidání kolizní komponenty:** Abychom mohli detekovat, že hráč minci sebral, přidejte další komponentu přes **Add Component**, například **Sphere Collision** (kulová kolize) [3]. Tato neviditelná kolizní koule bude obalovat minci a vyvolá událost, když do ní hráč vstoupí. U komponenty **SphereCollision** v **Details** panelu nastavte její poloměr tak, aby pokrýval celý model mince.
4. **Nastavení kolizí pro minci:** Vyberte kolizní komponentu (SphereCollision) a v **Details** najděte kategorii **Collision**. Zde zajistěte, že kolize bude fungovat jako trigger (spouštěč překryvu):
5. Nastavte **Collision Preset** na hodnotu **OverlapAllDynamic** (nebo ekvivalentní vlastní nastavení). Tím říkáme, že mince (jako dynamický objekt) nebude nic blokovat, ale bude generovat překryvové události se všemi dynamickými objekty včetně hráče.
6. Ověřte, že je zaškrtnuto **Generate Overlap Events** (generovat události překryvu). *Poznámka:* Pro fungování překryvových událostí musí mít možnost generování překryvu povolenou jak objekt mince, tak i objekt hráče [4]. Pokud by *Generate Overlap Events* bylo vypnuté, objekty by se vzájemně ignorovaly podobně jako při nastavení Ignore [5].
7. **Přidání komponenty pro rotaci (volitelně):** Pokud chcete, aby mince ve hře rotovala (což je častý efekt u sběratelských předmětů), můžete do Blueprintu přidat komponentu **Rotating Movement**. Tato komponenta automaticky rotuje aktér kolem zvolené osy. Stačí ji přidat a nastavit např. rychlost rotace kolem osy Z.

Tímto máme připravený Blueprint mince s viditelným modelem a kolizní zónou. V další části nastavíme, co se stane, když dojde ke kolizi (překryvu) mince s hráčem.

Umožnění sbírání objektů hráčem (událost OnOverlap)

Aby hráč mohl minci sebrat a hra to rozpoznala, musíme do Blueprintu mince přidat **událost překryvu** a obsloužit ji v Blueprint grafu. Konkrétně použijeme událost `OnComponentBeginOverlap` pro naši kolizní komponentu (SphereCollision):

1. **Přidání události překryvu:** Otevřete Blueprint **BP_Coin** (pokud není otevřen) a přepněte se do záložky **Event Graph**. Ujistěte se, že v seznamu Components máte vybranou komponentu **SphereCollision**. Poté v Event Graphu **klikněte pravým tlačítkem** do prázdného místa. V kontextovém menu zvolte **Add Event > Collision > OnComponentBeginOverlap (<komponenta>)**. Tím přidáte uzel události **OnComponentBeginOverlap** spojený s naší kolizní koulí [6]. (Alternativně můžete v kontextovém menu použít vyhledávání – napište "Overlap" a vyberte příslušnou událost pro SphereCollision. [7])
   Poznámka: Můžete také využít událost na úrovni aktéra `Event ActorBeginOverlap`, která se vyvolá při překryvu celého aktéra [4]. My však používáme událost komponenty, což je vhodné, když aktér má více kolizních zón.
2. **Identifikace objektu, který způsobil překryv:** Událost OnComponentBeginOverlap má mimo jiné výstup **Other Actor** (jiný aktér), což je odkaz na objekt, který do kolize vstoupil. V našem případě chceme ověřit, že tímto objektem je hráč (tj. jeho postava). Existuje několik způsobů, jak to udělat. Jednoduchý přístup je použít **Casting** (přetypování). Z výstupu *Other Actor* přetáhněte spojku a pusťte ji v prázdném místě grafu – objeví se kontextové menu s filtrací podle typu **Actor**. Napište například jméno třídy hráče (ve Third Person šabloně je hráčova BP třída defaultně **ThirdPersonCharacter**). Zvolte uzel **Cast To ThirdPersonCharacter**. Tímto uzlem se pokusíme převést odkaz *Other Actor* na typ hráčovy postavy. **Cast** se povede, pokud se skutečně jedná o postavu hráče – jinak (např. pokud

by do kolize vstoupilo něco jiného) se Cast nezdaří a následující logika se neprovede. *(Obecně řečeno, casting zkusí převést vstupní referenci na zadaný datový typ; uspěje, pokud vstup je potomkem onoho typu [8].)*

3. **Reakce na sebrání předmětu:** Z výstupu **As ThirdPersonCharacter** našeho Cast uzlu nyní máme odkaz na hráčovu postavu, takže můžeme provést akce spojené se sebráním mince. Přetáhněte z výstupu *As ThirdPersonCharacter* a vyhledejte uzel **Destroy Actor** – ale pozor, *Destroy Actor* chceme volat na minci, ne na hráče! Proto tento uzel zavolejte na **Self** (tj. na sebe sama – BP_Coin). Nejdříve však vyřešíme přičtení bodů do skóre hráče (viz další sekce). V tuto chvíli můžeme pro ověření funkčnosti přidat ladicí zprávu: např. přetáhněte z *As ThirdPersonCharacter* a najděte **Print String**; propojte jej před Destroy Actor a do zprávy napište třeba "Coin collected". Při hraní hry by se pak po sebrání mince zobrazila tato zpráva, což indikuje, že událost funguje.

Blueprint graf mince by měl vypadat tak, že z **OnComponentBeginOverlap** vedou dvě větve: - Z výstupu **Other Actor** -> Cast To ThirdPersonCharacter -> (dále použijeme pro zvýšení skóre a případně další akce) - Z výstupu **Cast succeeded (True)** -> naváže se další logika (pouze pokud je Other Actor hráč). Sem přijde přičítání bodů a zničení mince.

Tímto způsobem jsme zajistili, že **pouze hráčova postava může spustit událost sebrání**. Kdyby do kolizní oblasti vstoupil jiný aktér (což se v této hře asi nestane, ale obecně třeba jiný NPC nebo fyzikální objekt), Cast na ThirdPersonCharacter selže a sbírací logika se nepustí [8].

(Poznámka: Místo castování na konkrétní třídu lze také použít porovnání tagů nebo rozhraní (Blueprint Interface) pro obecnější řešení sbírání, ale pro jednoduchost volíme přímočarý Cast.)

Přičítání bodů a zničení objektu při sebrání

Nyní implementujeme **herní logiku sbírání**: při překryvu mince s hráčem se přičte bod ke skóre a mince zmizí. K tomu potřebujeme **proměnnou pro skóre** a poté upravit Blueprint mince i hráče:

1. **Vytvoření proměnné skóre:** Otevřete Blueprint hráčovy postavy (např. ThirdPersonCharacter). V levém panelu **My Blueprint** (Můj Blueprint) klikněte v sekci **Variables** (Proměnné) na ikonu **+** a přidejte novou proměnnou typu **Integer** (celé číslo). Pojmenujte ji **Score** (skóre). V **Details** této proměnné můžete nastavit výchozí hodnotu **Default Value** na 0. Tím jsme do Blueprintu postavy přidali vlastní proměnnou – **My Blueprint panel umožňuje přidávat vlastní proměnné do Blueprintu** a zobrazuje všechny existující proměnné [9]. (Volitelně můžete tuto proměnnou zaškrtnout jako **Instance Editable**/Public, aby byla viditelná i mimo blueprint – není to však nutné.)
2. **Zvýšení skóre v rámci sbírání:** Vraťte se do Event Graphu Blueprintu **BP_Coin**. Na výstupu z **Cast to ThirdPersonCharacter** (z uzlu As ThirdPersonCharacter) nyní můžeme manipulovat s proměnnou *Score* hráče. Přetáhněte z výstupu *As ThirdPersonCharacter* a do vyhledávání napište jméno proměnné “Score”. Měli byste vidět uzel **Set Score** (případně **Score (set)**). Vyberte jej – vloží se uzel pro nastavení proměnné Score v hráčově blueprintu. Protože chceme hodnotu zvýšit o 1, potřebujeme aktuální hodnotu přečíst, přičíst k ní 1 a výsledek zapsat zpět. Proto klikněte pravým tlačítkem do grafu a vyhledejte uzel **IncrementInt** nebo použijte výrazy: z *As ThirdPersonCharacter* přetáhněte proměnnou Score jako **Get Score** (čtení hodnoty), pak přetáhněte výstup a vyhledejte **+** (integer + integer) a zadejte konstantu 1. Výstup sumy připojte do vstupu nového *Score* v uzlu **Set Score**. Nakonec nezapomeňte propojit exekuci: z výstupu **Cast to ThirdPersonCharacter (True)** do vstupu **Set Score**, a z **Set Score** dále do **Destroy Actor** (který ničí minci). Tím je zajištěno, že:

3. Při sebrání mince se hráčovo **Score** zvýší o 1.
4. Následně dojde ke zničení aktéra mince pomocí `Destroy Actor` (volané na self, tj. minci), takže mince zmizí ze scény [10].
5. **Dokončení sběru:** Ujistěte se, že uzel **Destroy Actor** je připojen až po nastavení skóre, aby se mince nezničila dříve, než stihneme změnit hodnotu Score. Také je vhodné přidat např. zvuk či partikl efekt při sebrání (to lze udělat před zničením objektu). Nicméně v našem jednoduchém příkladu stačí, že mince zmizí a hodnota skóre se uloží.

Tímto máme funkční mechaniku sbírání: Hráč se dotkne mince -> vyvolá se OnOverlap -> pokud je to hráč, zvýšíme skóre a minci zničíme. Můžete otestovat hru (Play) a v **Debug** režimu sledovat proměnnou Score u hráče, zda se zvětšuje.

(Pro ladění můžete do hráčova Blueprintu přidat do Event Tick výpis Score pomocí Print String, nebo vytvořit jednoduché UI zobrazující skóre, ale to přesahuje rámec tohoto návodu.)

Ukončení hry při dosažení určitého skóre (Game Over)

Do hry přidáme podmínku, že jakmile hráč nasbírá určitý počet bodů (mincí), hra skončí – **Game Over**. To bude zahrnovat: znehybnění hráče (zákaz dalšího pohybu) a třeba efekt exploze zbývajících mincí pro efekt. Postup:

1. **Stanovení cílového skóre:** Rozhodněte, při kolika bodech hra končí (např. 5 bodů). Můžete tuto hodnotu uložit buď jako konstantu v Blueprintu (např. proměnná *TargetScore* v GameMode či v hráči), nebo ji prostě vložit napevno do podmínky. Pro jednoduchost použijeme pevnou hodnotu v podmínce.
2. **Detekce dosažení skóre:** Kde provést kontrolu? Nabízí se udělat to přímo ve chvíli sebrání mince – tedy v Blueprintu **BP_Coin** po zvýšení skóre. Tam máme aktuální hodnotu Score hráče. Po uzlu **Set Score** tedy můžeme vložit větev s podmínkou. Přetáhněte z **Set Score** výstupní exekuce a přidejte uzel **Branch** (podmínka). Jako **Condition** použijte porovnání, zda aktuální Score hráče >= cílové skóre. Aktuální Score získáte tak, že z *As ThirdPersonCharacter* (stále dostupné) přetáhnete **Get Score** a to napojíte do porovnávacího uzlu **>=** s konstantou (např. 5).
3. **Reakce na dosažení cíle – Game Over:** Výstup **True** z Branch (tj. skóre dosaženo nebo překročeno) použijeme k ukončení hry. Můžeme to udělat několika způsoby, zde je jednoduchá varianta:
4. **Znemožnění pohybu hráče:** Zavoláme na hráčově postavě funkci **Disable Input**, která odpojí daného aktéra od ovládání hráčem [11]. K tomu potřebujeme referenci na Player Controller. V Blueprintu mince můžeme z *As ThirdPersonCharacter* vytáhnout uzel **Get Player Controller** (s indexem 0) a poté zavolat na *ThirdPersonCharacter* uzel **Disable Input** s tímto Controllerem jako vstupem. Výsledkem je, že hráčova postava přestane reagovat na vstupy (pohyb, skok apod.). *Poznámka:* Uzel `Disable Input` odpojí aktéra od fronty zpracování vstupu konkrétního PlayerControlleru [11]. Alternativně by šlo nastavit proměnnou (např. boolean *GameOver*) a v Player Blueprintu u vstupů či v Event Tick kontrolovat tuto proměnnou a zastavit pohyb – použití `Disable Input` je však přímočařejší.
5. **Exploze objektů:** Pro dramatický efekt můžeme nechat zbývající objekty explodovat. Pokud například ve scéně zůstaly další mince (hráč nemusel sebrat všechny, stačilo dosáhnout cíle skóre), můžeme je všechny zničit najednou s efektem exploze. V Blueprintu mince (po větvi True) můžeme iterovat přes všechny mince: použijte uzel **Get All Actors of Class** (třída BP_Coin) a z výsledného pole pomocí **For Each Loop** projděte každou minci. Pro každou minci zavolejte **Spawn Emitter at**

Location s nastaveným šablonovým efektem exploze (pokud máte např. particle systém P_Explosion) a jako pozici použijte **Get Actor Location** d dané mince. Hned poté zavolejte **Destroy Actor** na danou minci. Tím každá zbývající mince vybuchne a zmizí. *(Jednodušší alternativa:* spustit jeden **víceúčelový výbuch** uprostřed scény nebo u hráče – záleží na požadovaném efektu. Pro ukázku techniky však popisujeme výbuch každé mince.)

6. **Označení stavu hry:** Je užitečné nastavit si proměnnou, která označí, že hra skončila, pro případ dalších logik. Vytvořte v Blueprintu hráče proměnnou **GameOver** (Boolean), která je defaultně *false*. Když dojde k ukončení hry (ve větvi True, jak řešíme výše), tak přes *As ThirdPersonCharacter* zavolejte **Set GameOver = true**. Tuto proměnnou by šlo využít například v HUD pro zobrazení hlášky "Game Over" nebo v jiných částech logiky.

Nyní při testu hry (Play) by se mělo stát, že jakmile nasbíráte stanovený počet mincí, další pohyb se zastaví a mince (pokud zbyly) explodují a zmizí, čímž je jasně indikováno, že hra skončila.

(Tip: Pro přehrání explozí můžete použít buď zabudovaný systém Cascade/Niagara s existujícími šablonami efektů, nebo si stáhnout jednoduchý efekt z Marketplace či Starter Content. Ujistěte se, že volané efekty existují v projektu.)

Detekce převrácení vozidla na střechu a spuštění exploze

Pokud vaše hra využívá vozidlo (např. auto v šabloně Advanced Vehicle nebo vlastní vozidlo v Top Down pohledu), může být požadováno detekovat, kdy se vozidlo převrátí na střechu, a následně vyvolat explozi (např. simulace zničení vozidla). **Detekci převrácení** lze provést více způsoby:

- **Kontrola úhlu rotace:** Jedna možnost je sledovat rotaci aktéra (vozidla) v ose X nebo Y. Pokud rotace (náklon) přesáhne určitý práh (např. ±90° od svislé polohy), znamená to, že vozidlo leží na boku nebo na střeše. Zkrátka lze ověřit, zda *pitch* nebo *roll* rotace vozidla překročila 90° [12].
- **Kontrola vektoru „nahoru“:** Elegantnější způsob je využít **Up Vector** aktéra – to je vektor, který ukazuje směr „nahoru“ lokálně pro daný objekt. Pokud je vozidlo převrácené, jeho Up Vector míří dolů k zemi místo vzhůru. Z komponent tohoto vektoru můžeme zkontrolovat Z-ovou složku (výšku). **Je-li Z složka Up Vectoru záporná, aktér je vzhůru nohama** [13]. Tuto podmínku můžeme testovat průběžně.
- *Poznámka:* U podvozků aut by šlo i vyhodnocovat, zda se žádné kolo nedotýká země apod., ale to je složitější.

Implementace v Blueprintu: Otevřete Blueprint vozidla (pokud je to například potomkem `WheeledVehicle`, či `Pawn`). Ujistěte se, že má povolen **Event Tick** (událost každého snímku). V Event Graphu vozidla přidejte uzel `Event Tick` (volá se každý frame [14]) a za něj vložte podmínku (Branch) kontrolující převrácení. Podmínku nastavte např. takto:

```
IsFlipped = ( GetActorUpVector.Z < 0 )
```

kde `GetActorUpVector` vrátí právě ten vektor „nahoru“ a porovnáváme jeho Z [13]. Pokud je podmínka True (vozidlo je převrácené), můžeme provést následující akce: - **Exploze vozidla:** Můžete podobně jako u mincí spawnout emitter: použijte **Spawn Emitter at Location** nebo **Spawn Emitter Attached** k vozidlu, s efektem exploze. Případně přehrajte zvuk exploze. - **Zničení nebo reset vozidla:** Poté můžete vozidlo zničit

(`Destroy Actor` samo sebe) nebo ho respawnovat či převrátit zpět na kola. Záleží, co je cílem – jestli „hra končí výbuchem vozidla“ (pak destrukce) nebo jen penalizace (můžete třeba odpočítat životy a obnovit vozidlo). V našem případě předpokládejme, že jde o konec hry nebo konec vozidla – použijeme Destroy. - **Další logika:** Případně opět můžete využít proměnnou GameOver nebo jiný mechanismus k ukončení hry či restartu úrovně po explozi.

Touto kontrolou v Ticku tedy neustále sledujeme, zda je vozidlo invertované. Event Tick se volá každý frame, což je v pořádku pro takovouto lehkou podmínku – je ale vhodné uvažovat, zda kontrolu nepotřebujeme dělat méně často (např. pomocí Timera), kvůli optimalizaci. Pro jednoduchost však Tick vyhovuje.

(U vozidel založených na `WheeledVehicle` *může existovat i interní pomoc s vracením na kola, ale zde demonstrujeme obecný způsob detekce rotace aktéra.)*

Práce s proměnnými v Blueprintu

Proměnné jsou základní prvek logiky v Blueprintech – umožňují ukládat stav (čísla, texty, reference na objekty atd.). Již jsme využili dvě vlastní proměnné: **Score** (Integer) a **GameOver** (Boolean). Shrňme si práci s proměnnými:

- **Vytvoření proměnné:** V editoru Blueprintu je panel **My Blueprint**, v němž je sekce **Variables**. Kliknutím na **+** zde přidáte novou proměnnou. Můžete jí nastavit jméno a zvolit datový typ (v rolovacím menu vedle jména). Unreal podporuje různé datové typy – např. **Boolean** (true/false), **Integer** (celé číslo), **Float** (desetinné číslo), **Vector** (3D vektor), **Object Reference** (odkaz na objekt určité třídy) atd. [15] [16] . Každý typ má v Blueprintech odlišnou barvu pro snadnou orientaci (bool – červená, int – zelená, float – světle modrá, apod.).
- **Nastavení a čtení proměnných (Get/Set):** Když máte proměnnou definovanou, můžete ji v Event Graphu použít. Přetáhnutím proměnné z panelu do grafu se zeptá, zda ji chcete **Get** (číst hodnotu) nebo **Set** (nastavit novou hodnotu). Podle toho se vytvoří uzel. **Set uzel** má vstupní pin pro hodnotu a prováděcí piny (bílé) pro tok programu; **Get uzel** má pouze výstupní pin s aktuální hodnotou. V naší logice jsme např. dělali Get Score, pak +1, pak Set Score.
- **Veřejné vs. soukromé proměnné:** Každá proměnná může být **private/public** (soukromá/veřejná). Veřejná (označená otevřeným okem vedle názvu proměnné) umožňuje vidět a měnit hodnotu proměnné zvenčí, například v Detail panelu při vybrání instancí Blueprintu v úrovni [17] . To je užitečné pro ladění nebo nastavení instance. V našem případě proměnná Score byla v hráči a nepotřebovali jsme ji měnit ručně, takže mohla klidně zůstat private.
- **Proměnné v různých Blueprintech:** Pokud potřebujete přistupovat k proměnné z jednoho Blueprintu v jiném (např. z mince k proměnné hráče), musíte mít odkaz na instanci onoho Blueprintu a pak použít buď **Cast** na správný typ (jak jsme udělali s ThirdPersonCharacter) nebo využít jiný mechanismus (Game Mode, Level Blueprint, atd.). V našem návodu jsme z mince castovali na hráče a pak přímo nastavovali jeho proměnnou. Alternativně by šlo udržovat skóre např. v Blueprintu GameMode a přistupovat k němu přes `GetGameMode` – to ale vyžaduje vlastní GameMode třídu.
- **Globální proměnné:** Blueprinty neumožňují přímo globální proměnné přístupné odkudkoliv, ale lze použít např. **Game Instance** nebo zmíněný **Game Mode** k uchovávání hodnot platných po celou hru. To se hodí např. pro skóre přenášené mezi úrovněmi.

Stručně řečeno, práce s proměnnými v Blueprintu je intuitivní: *vytvořit* v My Blueprint, *nastavit* či *číst* v grafu pomocí uzlů Set/Get a případně přenášet hodnoty mezi Blueprinty skrze reference a casting. Doporučujeme

udržovat pořádek – pojmenovávat proměnné srozumitelně a využívat kategorie v My Blueprint (lze si vytvořit složky pro proměnné).

Sloučení aktérů a ukládání statických meshů (Merge Actors)

Unreal Engine nabízí nástroj **Merge Actors** pro optimalizaci a práci s objekty. **Sloučení aktérů** umožňuje spojit více statických meshů do jednoho, čímž snížíte počet draw calls a zjednodušíte scénu [18]. V praxi to znamená, že pokud máte např. složitou scénu sestavenou z mnoha malých meshů (domy z kostek apod.), můžete je sjednotit do jednoho většího mesh. Postup použití Merge Actors:

1. **Výběr objektů:** V úrovni označte v **Level Viewport** nebo **World Outlineru** všechny **Static Mesh Actor** objekty, které chcete sloučit do jednoho [19]. (Lze slučovat pouze statické meshe, nikoli pohyblivé objekty s animacemi apod.)
2. **Spuštění Merge:** V horním menu editoru klikněte na **Actor > Merge Actors**. Otevře se menu s variantami sloučení. Zvolte základní možnost **Merge** (případně **Simplify** pro vytvoření zjednodušené proxy, atd. – pro naše účely stačí Merge) [19]. Engine provede sloučení automaticky s výchozími nastaveními, nebo můžete předem vybrat **Merge Actor Settings** pro detailnější konfiguraci.
3. **Uložení nového meshe:** Po zvolení merge typu se zobrazí dialog pro uložení. Zadejte jméno nového sloučeného objektu a cílovou složku v Content Browseru, kam se výsledný **Static Mesh asset** uloží, poté klikněte na **Save** (Uložit) [20]. Unreal nyní vytvoří nový Static Mesh, který reprezentuje sloučení všech vybraných objektů. Zároveň může vygenerovat nový aktér v úrovni, který tento mesh používá.
4. **Výsledek:** Výsledný asset najdete v Content Browseru – je to standardní Static Mesh, který můžete používat jako kterýkoli jiný model (umístit ho do úrovně, instance apod.). Původní objekty lze případně skrýt nebo odebrat ze scény, pokud je nahrazujete tímto novým. Pamatujte, že **po sloučení přijdete o jednotlivou editaci původních částí** – pokud byste potřebovali něco změnit, museli byste upravit původní objekty a znovu provést merge.

Merge Actors zlepšuje výkon zejména snížením počtu objektů (draw calls). Má několik režimů: - **Merge:** prosté sloučení všech meshů (s možností sloučit i materiály do jednoho materiálu – to ale zruší původní UV mapy) [21]. - **Simplify:** sloučení s redukcí detailů (LOD) pro vytvoření jednodušší proxy meshe [22]. - **Batch:** sloučení více instancí téhož meshe do jednoho s instancovanými komponentami [23]. - **Approximate:** novinka UE5 – sloučení schopné pracovat i s Nanite a velmi komplexními meshi [24].

Pro běžné účely stačí režim **Merge**. V kontextu naší jednoduché hry byste třeba mohli **sloučit statické prostředí** (např. několik dekorativních meshů) do jednoho, abyste snížili zátěž. *Například*: pokud jste si scénu upravili tak, že jste postavili překážkovou dráhu z mnoha krychlí, můžete tyto krychle sloučit do jednoho modelu.

(Upozornění: Sloučení aktérů by se nemělo používat na objekty, které se budou hýbat nezávisle – je to hlavně optimalizační krok pro statické dekorace. Také mějte na paměti, že sloučením různých meshů s odlišnými materiály může výsledný mesh mít víc materiálových elementů, nebo můžete zvolit sloučení materiálů do jednoho, což ale může ovlivnit vzhled.)

Vytváření vlastních materiálů

Vlastní **materiály** umožňují definovat vzhled povrchu objektů (barvu, odrazivost, hrubost, průhlednost aj.). V našem případě si můžeme vytvořit například zlatý materiál pro minci. **Postup vytvoření materiálu:**

1. **Vytvoření nového Material assetu:** V Content Browseru klikněte na **Add New** a z nabídky **Materials & Textures** vyberte **Material**. (Případně můžete pravým kliknutím v prázdném místě Content Browseru otevřít kontextové menu a zvolit *Material* [25].) Nový materiál pojmenujte např. *M_Gold*.
2. **Otevření Material Editoru:** Dvojklikem otevřete vytvořený materiál. Otevře se **Material Editor** – plátno s grafem uzlů. Uvidíte výchozí uzel **Main Material Node** (označený názvem materiálu), který má vstupy jako Base Color, Metallic, Roughness, Emissive apod.
3. **Vytvoření základního vzhledu materiálu:** V grafu klikněte pravým a přidejte potřebné uzly pro definování vlastností. Pro zlatý materiál stačí jednoduché nastavení:
4. **Base Color:** Přidejte uzel **Constant3Vector** (nebo **Vector Parameter**) a nastavte barvu na zlatou (např. RGB kolem 1.0, 0.85, 0.0 pro sytě žlutou). Výstup tohoto uzlu připojte do vstupu **Base Color** hlavního uzlu.
5. **Metallic:** Přidejte uzel **Constant** (jednoduchá konstanta) s hodnotou 1 a připojte do vstupu Metallic – tím řekneme, že materiál je kovový (1 = kov).
6. **Roughness:** Přidejte **Constant** s hodnotou kolem 0.2 a připojte do Roughness – nižší hodnota znamená povrch hladší/lesklejší. Hodnota 0.2 dá materiálu lesk.
7. (Případně **Specular:** můžete ponechat implicitně, nebo nastavit drobně pro doladění, ale není nutné, u kovu se specular většinou neřeší.)
8. **Kompilace a uložení:** Klikněte na **Apply** nebo **Save** v horní liště Material Editoru. Tím se materiál zkompiluje. Ve Viewportu náhledu byste měli vidět zlatě vypadající kouli. **Shrnutí kroků:** vytvořit materiál, otevřít editor, umístit uzly, propojit je s Main Material node, zkompilovat a uložit [26].
9. **Aplikace materiálu na objekt:** Zavřete Material Editor. Nyní můžete nový materiál přiřadit například našemu blueprintu mince. Otevřete **BP_Coin**, vyberte komponentu *CoinMesh* (Static Mesh) a v Details panelu najděte sekci **Materials**. Zde nastavte **Element 0** na váš materiál *M_Gold* (vyberte z nabídky). Uložte Blueprint. Nyní každá instance mince ve hře bude používat tento materiál a bude vypadat jako zlatá mince.

Gratulujeme, vytvořili jste vlastní materiál a použili ho ve hře! Materiál Editor Unreal Enginu je velmi mocný nástroj – umožňuje pomocí uzlů vytvářet i složité efekty povrchu. V tomto návodu jsme ukázali jen jednoduchý příklad. Pro hlubší pochopení doporučujeme projít si oficiální dokumentaci k materiálům (např. *Essential Material Concepts* a tutoriály v *Materials* sekci dokumentace [27] [28]).

(Tip: Můžete zkusit experimentovat – např. do materiálu mince přidat lehkou Emissive složku, aby mince jemně zářila. Stačí přidat Constant3Vector s žlutou barvou a malou hodnotou (např. 0.2) a připojit do Emissive Color, pak Apply. Mince pak bude viditelně zářit i ve stínu.)

Důležité události v Blueprintu (BeginPlay, Tick, Hit, Overlap)

V Blueprint skriptech se hojně využívají **události** (Events), které slouží jako začátek provádění logiky v reakci na určité situace. Pojďme si stručně představit nejběžnější události, se kterými jsme se setkali či které byste měli znát:

- **Event BeginPlay:** Tato událost se vyvolá **na začátku hry** pro daný aktér – přesněji, když aktér vstoupí do hry nebo je spawnutý [29]. Typicky se používá pro inicializace. Příklady: nastavení výchozích hodnot proměnných, zobrazení úvodní obrazovky, start hudby, atd. V našem projektu bychom `Event BeginPlay` mohli použít třeba v Blueprintu hráče k vynulování skóre na začátku (i když proměnná už má default 0, ale pro jistotu) nebo k vypsání instrukcí. *Pozor:* Každý aktér má svou vlastní BeginPlay, která se volá při jeho zrození ve hře.
- **Event Tick:** Událost volaná **každý snímek (frame)** hry, pokud je u aktéra povoleno tikání [14]. Poskytuje i delta čas `Delta Seconds` mezi snímky. Používá se pro kontinuální kontrolu nebo pohyb – např. aktualizace kamery, kontrola podmínek (viz detekce převrácení vozidla v našem případě), počítání času, plynulé animace posuvem atd. Je však třeba opatrnosti – kód v Tick běží velmi často, takže náročné operace by mohly hru zpomalovat. Vždy zvažte, zda danou logiku nelze volat méně často nebo na konkrétní událost místo neustále. V našem návodu jsme Tick použili v kontextu vozidla pro zjištění, jestli se převrátilo.
- **Event Hit:** Tato kolizní událost se volá, **když aktér narazí do jiného** (při blokující kolizi) nebo když do něj něco narazí [30]. Aby se volala, musí mít objekty nastaveno generování Hit událostí (zejm. u simulační fyziky *Simulation Generates Hit Events* [31]). Příklad využití: projektil (střela) má Event Hit, kde zjistí, do čeho narazil a spustí např. efekt exploze nebo ubere zdraví zasaženému objektu. Naše mince měla jen overlap (průchozí), takže Hit jsme nepoužili. Ale pokud by mince byla pevná a hráč do ní vrážel fyzicky, mohli bychom místo Overlap použít Event Hit. Obecně platí, že *Hit* se používá u fyzických kolizí (např. dopad ragdoll panáčka na zem, srážka auta se zdí), kdežto *Overlap* u triggerů.
- **Event ActorBeginOverlap / OnComponentBeginOverlap:** Událost, která se vyvolá, když se jeden aktér nebo komponenta **překryje** (overlap) s jiným [4]. Překryv znamená neblokující průnik kolizních těl – typicky vstup do trigger zóny. My jsme hojně využili OnComponentBeginOverlap u mince, ale existuje i Actor varianta (ActorBeginOverlap) reagující na jakýkoli začátek překryvu aktéra. K této události je důležité mít povolené **Generate Overlap Events** (na obou objektech, jak uvádí dokumentace [4]). Kromě BeginOverlap existuje také **EndOverlap** (pro chvíli, kdy objekt z triggeru vystoupí). Overlap události jsou ideální pro sbírání předmětů, detekci vstupu do oblasti (např. spínač), atd.
- **Další události:** Blueprinty nabízejí mnoho dalších eventů, např. **Event Destroyed** (když je aktér zničen), **Event EndPlay** (když aktér opouští hru, třeba při úrovně konci), **Event OnClicked** (při kliknutí myší na objekt, pokud povoleno), **Event OnBeginCursorOver** (hover myši nad objektem), události vstupu (kláves – ty se typicky řeší v Player Controlleru nebo Pawn), a také události časovačů a delegátů. Pro komplexní hry je dobré tyto eventy znát, ale pro naše účely byly klíčové výše zmíněné: BeginPlay, Tick, Hit, Overlap.

Události se v Blueprintu **nezdvojují** – každou konkrétní událost (např. BeginPlay) můžete v daném Event Graphu mít jen jednou. Proto když potřebujete na jednu událost navěsit víc logiky, řeší se to z jednoho uzlu dál větvením (sekvence, volání funkcí, custom eventy apod.), nikoli přidáním druhého BeginPlay. Pokud se vám nedaří přidat událost, zkontrolujte, jestli tam už není (často jsou předpřipravené šedé "ghost" uzly na začátku).

Nastavení a správa kolizí

Správné **nastavení kolizí** je zásadní pro to, aby se objekty ve hře chovaly, jak očekáváme – aby do sebe buď narážely, nebo procházely, a vyvolávaly správné události. Unreal Engine používá systém **Collision Channelů a Response** (reakcí). Každý objekt má přiřazen **Object Type** (typ objektu, např. WorldStatic, WorldDynamic, Pawn, PhysicsBody, Vehicle apod.) a ke každému jinému typu může mít nastaveno, zda **Block** (blokuje), **Overlap** (překryje), nebo **Ignore** (ignoruje) [32].

Klíčové pojmy a tipy: - **Collision Preset:** V **Details** panelu komponenty (např. StaticMeshComponent, CapsuleComponent atd.) najdete přednastavení kolizí. Presety jako *BlockAll*, *OverlapAll*, *Pawn*, *PhysicsActor*, *Trigger* atd. jsou sady typických nastavení. Např. **Pawn** preset pro kapsli hráče blokuje WorldStatic (aby hráč stál na podlaze), ale overlapuje Pawn (aby se hráči navzájem neblokovali v multiplayeru) atd. - **Custom nastavení:** Přepnutím Preset na **Custom…** získáte detailní matici. Zde vidíte položky **Collision Enabled** (např. CollisionEnabled (Query and Physics) – zda se řeší dotazy a fyzika), dále **Object Type** (např. Pawn u hráče, WorldDynamic u mince) a tabulku **Collision Responses**. V ní pro každý Channel (WorldStatic, WorldDynamic, Pawn, Camera, Visibility, atd.) zvolíte reakci: Block, Overlap, Ignore. - **Block vs Overlap vs Ignore:** - **Block** znamená, že objekty se fyzicky střetnou – nemohou se procházet. Pokud dojde k block kolizi, mohou se vyvolat události Hit (pokud jsou povolené) [30]. - **Overlap** znamená, že objekty se navzájem neblokují (projdou skrz), ale vyvolá se událost overlap (Begin/End) pokud je povoleno generování událostí [4]. Pro hráče a mince jsme použili overlap, protože jsme chtěli, aby skrz sebe prošli a spustila se logika sběru. - **Ignore** znamená úplné ignorování – žádná fyzická kolize ani událost se nevyvolá. - Důležité: Overlap může nastat i mezi objekty, které by jinak blokovaly, pokud se pohybují rychle (může "proletět" a vyvolat overlap). Obecně se ale **nedoporučuje kombinovat současně blokování i overlap** pro stejný pár objektů, může to vést ke složitostem [33]. - **Generate Overlap Events:** Aby *Overlap* reakce skutečně vyvolala Blueprint události, musí být zaškrtnutá tato volba. Jak už bylo zmíněno, obě strany kolize musí mít generování zapnuto [4]. Např. standardní **Trigger** preset má vše nastaveno na Overlap a zapnuté generování událostí. - **Simulation Generates Hit Events:** Obdobně, pro fyzikální simulované objekty je potřeba zapnout tuto volbu, pokud chcete, aby při kolizi vznikaly Hit eventy [31]. Například u fyzických objektů (PhysicsBody) vypnutím ušetříte výkon, ale nebudete mít eventy o nárazu. - **Kolizní matrice:** Při ladění se můžete podívat do **Project Settings > Engine > Collision**, kde je tabulka definic channelů a presetů. Můžete si dokonce vytvořit vlastní Object Channel nebo Trace Channel s vlastní logikou, pokud hra něco speciálního potřebuje. - **Debugging kolizí:** Unreal má užitečný nástroj pro vizualizaci kolizí – pokud zapnete **Player Collision** zobrazení nebo použijete příkaz `pxvis collision`, uvidíte obrysy kolizních těles. Také **Collision Analyzer** může pomoci analyzovat, proč se něco sráží nebo ne.

V naší minihře jsme konkrétně nastavili: - Mince (BP_Coin, resp. její kolizní komponenta) má **Object Type** = WorldDynamic (výchozí pro aktéry) a pro **Pawn** nastavena reakce **Overlap** (přes preset OverlapAllDynamic se to stalo automaticky) – tím pádem když **Pawn (hráč)** vstoupí, řeší se overlap. - Hráč (ThirdPersonCharacter) má kapsli s **Object Type** = Pawn a ta pro WorldDynamic (mince) podle **Pawn** presetu obvykle dělá **Overlap** také, takže to je oboustranné. (Kapsle Pawn preset typicky blockuje WorldStatic a Overlapuje vše dynamické.) - GenerateOverlapEvents: je defaultně zapnuté na kapsli hráče i na naší kolizní sféře mince, takže overlap event proběhne.

Shrnutí: Při nastavování kolizí si vždy promyslete, co s čím má kolidovat. Pokud chcete sbírat předmět bez fyzického střetu – dejte oběma Overlap vztah. Pokud chcete, aby se objekty odrážely – nastavte Block. A pokud chcete něco zcela vyjmout (např. projektil nemá kolidovat s vlastní střelcem) – nastavte Ignore pro tu

kombinaci. Dokumentace Unreal Engine nabízí přehled interakcí kolizí a doporučení [34]. S praxí se v tom rychle zorientujete.

Závěrem: Prošli jsme si základní postup vytvoření jednoduché hry v UE 5.5.4 s využitím šablony. Během toho jsme upravili existující objekty scény, vytvořili vlastní Blueprint sběratelského předmětu, nastavili kolize a události pro sbírání, udržujeme skóre v proměnné, reagujeme na dosažení cíle ukončením hry a naučili jsme se detekovat speciální případ (převrácení vozidla). Také jsme se seznámili s nástrojem Merge Actors pro slučování modelů, vytvořili jsme jednoduchý vlastní materiál a zopakovali si hlavní Blueprint události a zásady nastavování kolizí. V odkaze na oficiální dokumentaci Epic Games můžete najít další detaily ke každému tématu, doporučujeme prostudovat uvedené zdroje pro hlubší pochopení. Hodně štěstí při vytváření vaší hry a úspěch u zkoušky!

Použité zdroje: Dokumentace Unreal Engine a komunitní fóra (viz odkazy v textu). [1] [35] [36] [4] [5] [6] [8] [11] [10] [12] [13] [9] [18] [19] [20] [26] [27] [30]

---

[1] Transforming Actors in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/transforming-actors-in-unreal-engine

[2] [3] [35] [36] How do I make a collectable that will we added to a collectable menu? - Asset Creation - Epic Developer Community Forums
https://forums.unrealengine.com/t/how-do-i-make-a-collectable-that-will-we-added-to-a-collectable-menu/761068

[4] Event ActorBeginOverlap | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/AddEvent/Collision/EventActorBeginOverlap

[5] [31] [33] [34] Collision in Unreal Engine - Overview | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/collision-in-unreal-engine---overview

[6] [7] [8] Quick Start Guide for Blueprints Visual Scripting in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/quick-start-guide-for-blueprints-visual-scripting-in-unreal-engine

[9] [15] [16] [17] Blueprint Variables in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprint-variables-in-unreal-engine

[10] Unreal Engine 4 Particle Systems Tutorial - Kodeco
https://www.kodeco.com/270-unreal-engine-4-particle-systems-tutorial/page/3

[11] Disable Input | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/Input/DisableInput

[12] [13] How to see if actor is upside down - Blueprint - Epic Developer Community Forums
https://forums.unrealengine.com/t/how-to-see-if-actor-is-upside-down/452763

[14] Event Tick | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/AddEvent/EventTick

18 19 20 21 22 23 24 Merging Actors in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/merging-actors-in-unreal-engine

25 26 28 Unreal Engine Material Editor User Guide | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-material-editor-user-guide

27 Unreal Engine Materials | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-materials

29 Event BeginPlay | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/AddEvent/EventBeginPlay

30 Event Hit | Unreal Engine 5.5 Documentation | Epic Developer Community
https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/AddEvent/Collision/EventHit

32 I don't understand “Collision Presets” : r/UnrealEngine5 - Reddit
https://www.reddit.com/r/UnrealEngine5/comments/17yfz5e/i_dont_understand_collision_presets/