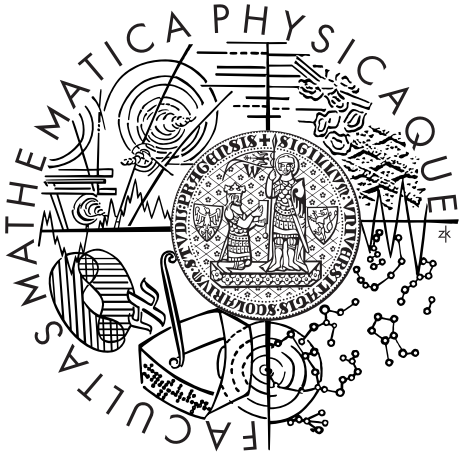
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Milan Rybář

2D Platform Game Creator

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Tomáš Balyo
Studijní program: Informatika
Studijní obor: Obecná informatika

Praha 2012

Chtěl bych poděkovat vedoucímu bakalářské práce Mgr. Tomášovi Balyovi za pomoc a čas, který mi s ochotou věnoval, a svému bratrovi Jakubovi Rybářovi za poskytnutí grafického podkladu pro hru v ukázkovém projektu.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 2. srpna 2012

Název práce: 2D Platform Game Creator

Autor: Milan Rybář

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Tomáš Balyo

Abstrakt: Práce se zabývá implementací aplikace pro vytvoření 2D her založených na simulaci reálné fyziky. Hru lze vytvořit bez znalosti programování. Je navrženo vizuální skriptování pro definici chování objektů ve hře, které je jednoduše rozšířitelné. Herní objekt je popsán konečnými automaty. Ve stavech se na virtuální ploše spojují uzly s předdefinovaným chováním. Je provedeno porovnání s podobnými aplikacemi. Práce obsahuje ukázkový projekt, který demonstruje schopnosti aplikace a vizuálního skriptování.

Klíčová slova: editor her, vizuální skriptování

Title: 2D Platform Game Creator

Author: Milan Rybář

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Tomáš Balyo

Abstract: The aim of this thesis is to implement an application for creating 2D games with physics simulation. The game can be created without any programming knowledge. We propose an extendable visual scripting to define the behaviour of game objects. The game objects are described by finite state machines. Nodes with pre-defined behaviour are connected on a virtual space at the object states. The work contains a comparison with similar applications. The capabilities of the application and the visual scripting method are demonstrated on a sample project.

Keywords: game creator, visual scripting

Obsah

Úvod **3**

1 Analýza problému **4**
1.1 Analýza her . . . 4
1.2 Vizuální skriptování . . . 4
1.3 Typ aktéra . . . 7
1.4 Požadované vlastnosti editoru . . . 8
1.5 Podobné programy . . . 8

2 Uživatelská dokumentace **10**
2.1 Fyzika . . . 10
2.2 Projekt . . . 10
2.3 Herní obsah . . . 11
2.4 Scéna (Scene) . . . 13
2.5 Vrstva (Layer) . . . 15
2.6 Cesta (Path) . . . 15
2.7 Aktér (Actor) . . . 16
2.8 Vizuální skriptování (Visual Scripting) . . . 17
2.9 Prototyp aktéra (Prototype) . . . 21
2.10 Nastavení hry (Game Settings) . . . 21
2.11 Testování hry (Run Game) . . . 22
2.12 Publikování hry (Publish Game) . . . 22
2.13 Klonování aktérů . . . 22
2.14 Bezpečné smazání . . . 22

3 Programátorská dokumentace **24**
3.1 Použité knihovny . . . 24
3.2 Game Engine . . . 24
3.2.1 Správa obrazovek . . . 25
3.2.2 Scéna . . . 26
3.2.3 Aktér . . . 26
3.2.4 Vizuální skriptování . . . 27
3.2.5 Herní obsah . . . 29
3.2.6 Inicializace scény . . . 29
3.2.7 Cyklus scény . . . 32
3.3 Editor . . . 32
3.3.1 Projekt . . . 34
3.3.2 Herní obsah . . . 34
3.3.3 Scéna . . . 36
3.3.4 Objekty ve hře . . . 37
3.3.5 Vizuální skriptování . . . 38
3.3.6 Přeložení hry . . . 41
3.3.7 Důležité formuláře . . . 43

Závěr **45**

Seznam použité literatury **47**

A Obsah přiloženého CD **48**

B Ukázkový projekt **49**

Úvod

Počítačové hry jsou v dnešní době velmi výraznou složkou softwarového průmyslu. Populární jsou především 3D hry, ale 2D hry ještě nejsou mrtvé. Výkon počítačů se zvýšil na takovou míru, že je možné ve hře používat simulaci reálné fyziky. Hra se chová nejen reálně, ale na fyzice může být postavena samotná logika hry. Řešení hádanek pomocí fyziky je založené na posunování a přemisťování věcí.

Cíle práce

- Cílem této práce je implementovat aplikaci pro vytváření 2D her založených na simulaci reálné fyziky. Fyzika nejvíce vyhovuje hrám s pohledem ze strany stylu skákačky (plošinovky) nebo střílečky.[1]
- Hlavním cílem je možnost vytvoření hry bez znalosti programování. Analyzovat a navrhnout řešení vizuálního skriptování, které musí být programátorem jednoduše rozšířitelné pro vytvoření pokročilejšího chování.

Struktura práce

Práce je rozdělena do tří kapitol. V první kapitole jsou popsány základní vlastnosti, které by editor a vytvořená hra měli obsahovat. Analyzuje a navrhuje řešení vizuálního skriptování hry pro uživatele bez programátorských znalostí. Je zde porovnání s podobnými aplikacemi. Druhá kapitola vysvětluje použití programu a tvorbu hry. Třetí kapitola se zabývá implementací programu, je zde popsána jeho architektura a uvedena struktura tříd. Popisuje rozšíření předdefinovaného chování. Jako příloha je přiložen ukázkový projekt, který představuje možnosti aplikace a navrženého vizuálního skriptování.

[1] V angličtině platform game (platformer) nebo action game (shooter)

1. Analýza problému

Tato kapitola popisuje vlastnosti, které by editor a vytvořená hra měli obsahovat. Analyzuje a navrhuje řešení vizuálního skriptování hry pro uživatele bez programátorských znalostí. Obsahuje porovnání s existujícími aplikacemi.

1.1 Analýza her

Každá komplexnější hra se skládá z několika oddělených částí - logických obrazovek, např. jednotlivé obrazovky menu nebo každé kolo hry. Kolo hry představuje herní plán, na kterém se hra odehrává. Budeme se zabývat především vývojem samotné hry, a ne menu, proto dále obrazovku budeme nazývat scéna. Na scéně se vyskytují jednotlivé herní objekty. Herní objekt budeme nazývat aktér.

Scéna je dána rovinou, takže všichni aktéři jsou na stejné úrovni, nicméně hra často obsahuje objekty představující pozadí nebo popředí. Například postavička hráče nesmí být překrytá postavičkou nepřítele. Pro ovládání vykreslování aktérů na scéně poslouží vrstvy. Scéna se skládá z libovolného počtu vrstev. Do vrstvy se pak přidávají jednotliví aktéři. Vykreslování probíhá po jednotlivých vrstvách.

Vytvořená hra bude postavená na simulaci reálné fyziky. Toto velice ovlivňuje žánr hry. Fyzika nejvíce vyhovuje hrám s pohledem ze strany. Lze ale vytvořit i hru s pohledem seshora, např. závodní hru.

1.2 Vizuální skriptování

Z herního hlediska je velice důležité chování aktérů ve hře. V klasickém průběhu vývoje hry by se tato část naprogramovala v nějakém programovacím nebo skriptovacím jazyce, např. Lua [9]. Program je ale určen pro běžného uživatele, který musí být schopen vytvořit základní hru bez programátorských znalostí.

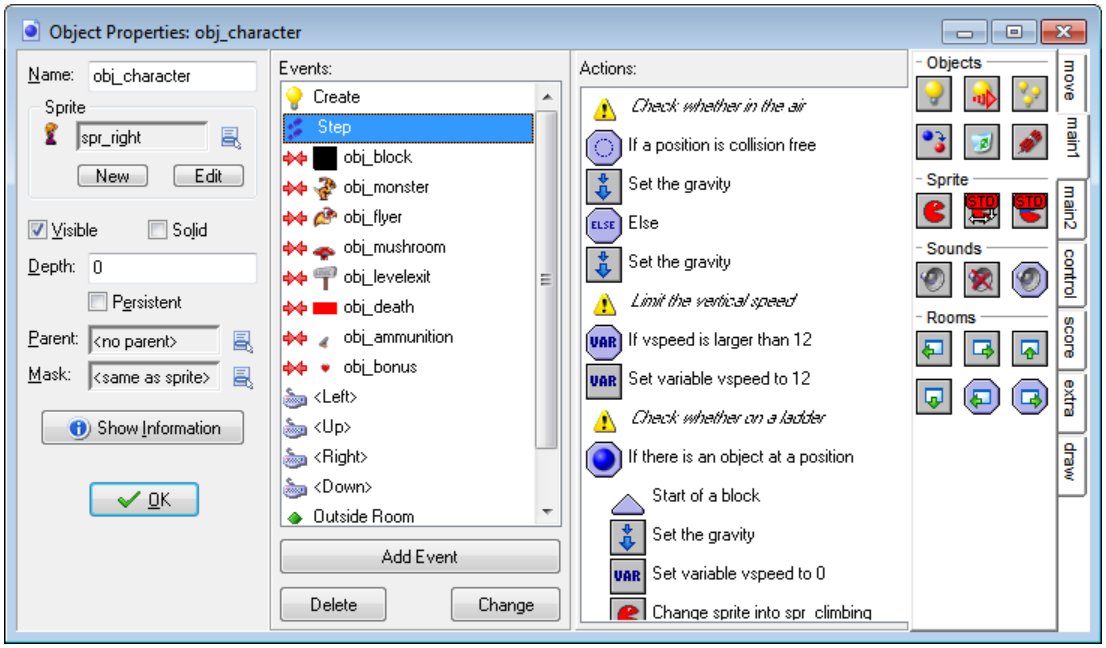
Předdefinované chování

Program musí nějakým způsobem obsahovat předdefinované chování, pomocí kterého lze vytvořit logiku hry. Zároveň musí být jednoduše rozšiřitelné. Nejpoužívanějším řešením je, že máme nějaké události ve hře, na které reagujeme posloupností akcí. Tento jednoduchý model lze zobrazit mnoha způsoby.

Obrázek 1.1 zobrazuje způsob, kdy posloupnost akcí je zobrazena v řadě za sebou. Tento způsob připomíná znázornění zdrojového kódu. Každá akce má ještě schované nastavení, které nelze okamžitě vidět. Problém nastává při pokusu o vytvoření složitějšího chování, kdy se tento způsob stává velice omezený a nepřehledný [8].

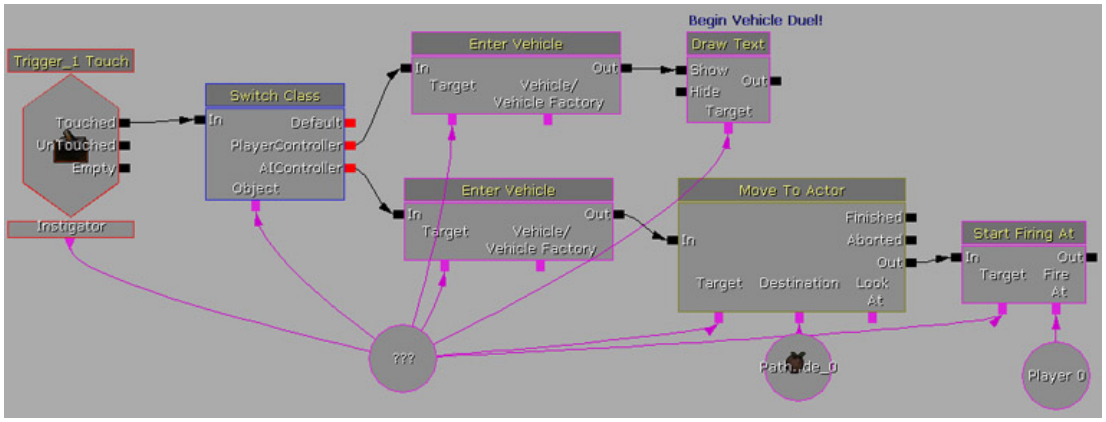
Jiný způsob zobrazuje obrázek 1.2. Zde se pracuje s uzly, které mají vstupy a výstupy. Každý uzel představuje krabičku, která vykonává předdefinovanou akci. Spojením těchto krabiček na virtuální ploše se vytvoří chování aktéra. Existují tři typy uzlů:

- Událost - Reaguje na jistou událost ve hře. Např. kolize aktéra.



Obrázek 1.1: Nastavení chování objektu v programu GameMaker [14]

- Akce - Provádí jistou akci, když je aktivována výstupem jiné akce nebo událostí. Např. pohyb aktéra.
- Proměnná - Umožňuje uložit hodnotu daného typu. Toto je jediná věc, která trochu připomíná klasické programování.



Obrázek 1.2: Ukázka vizuálního skriptování v programu UDK [3]

Zvolen byl druhý způsob pomocí spojování uzlů. Na virtuální ploše uzlů lze lépe vidět činnost aktéra.

Akce a události definují různé druhy vstupů a výstupů (viz obrázek 2.5). Nejdůležitější jsou vstupy a výstupy pro signál, který představuje průběh vykonávání uzlů. Dále je budeme nazývat pouze vstupy a výstupy. Spojuje se výstup se vstupem. Výsledné propojení uzlů definuje nějaké chování. Na vstup i výstup lze připojit libovolné množství uzlů. Událost obsahuje pouze výstupy, vstupy nemají smysl.

Akce a události potřebují pracovat s proměnnými, např. pro součet dvou čísel a uložení výsledku. Pomocí proměnných také definují či ovlivňují své chování, např. nastavení rychlosti pohybu aktéra. Tudíž definují vstupy a výstupy pro proměnné daného typu. Dále budeme vstup pro proměnnou nazývat vstupní proměnná a výstup pro proměnnou výstupní proměnná. Lze připojit pouze proměnné stejného typu. Na vstupní proměnnou lze připojit buď jednu, nebo libovolné množství proměnných, podle definice uzlu. Např. akce pro pohyb aktéra potřebuje rychlost pohybu, ale nemá smysl jí dávat více hodnot. Naopak uzel pro smazání aktéra ze scény může smazat všechny aktéry, které dostane. Do výstupní proměnné lze připojit libovolné množství proměnných.

Pro zjednodušení lze nastavit hodnotu vstupní proměnné také přímo v nastavení uzlu. Jinak by se kvůli nastavení pouhé konstanty musela vytvořit nová proměnná pro připojení, která by obsahovala potřebnou hodnotu. Označení vstupní a výstupní proměnné je spíše informativní, protože vstupní proměnná může měnit hodnoty připojených proměnných. Přesto se toto označení hodí pro označení logického významu proměnné pro uzel.

Pokud by existoval pouze jeden globální prostor pro scénu, kde lze vytvářet chování aktérů, tak při větším množství aktérů a uzlů se stává prostor nepřehledný. Aktéři se stejným chováním by měli duplikované chování v globálním prostoru. Proto by měl každý aktér obsahovat vlastní skriptování. Zároveň je ale vhodné ponechat skriptování pro scénu, pomocí kterého lze nadefinovat globální chování aktérů nebo herní logiku scény.

Vybraná varianta vyhovuje jednoduché rozšiřitelnosti. Programátor rozšíří program o specifické akce a události, které konkrétní hra vyžaduje. Designér vytvoří kolo hry a pomocí těchto uzlů je schopen vytvořit herní logiku.

Konečný automat

Představené spojování uzlů by stačilo na vytvoření chování aktéra. Ale pro komplexnější chování se bude hodit sofistikovanější nástroj.

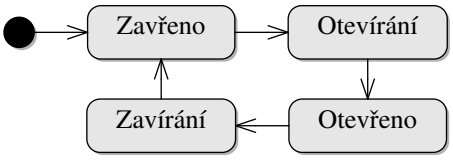
Aktéra lze typicky popsat pomocí konečného automatu, kde každý stav představuje jiné chování aktéra. Např. dveře jsou otevřené nebo zavřené, postavička hráče stojí, běží nebo skáče, apod. Proto je logická volba začlenit konečný automat do vizuálního skriptování. Tzn. lze definovat stavy, přechody mezi nimi a počáteční stav. Je vhodné přechody pojmenovávat a jejich názvy znovu využívat. Pro změnu stavu konečného automatu použijeme název přechodu. Pokud v aktuálním stavu existuje přechod s daným názvem, tak přejdeme do nového stavu.

V každém stavu se teprve nachází uzly představené dříve. Zavedením konečného automatu musíme omezit platnost uzlů, jinak by uzly byli aktivní v každém stavu, což v podstatě ruší význam konečného automatu. Uzel ve vizuálním skriptování je aktivní a bude prováděn právě tehdy, když je aktivní stav automatu, ve kterém se nachází (až na pár výjimek jako je přehrávání zvuku). Např. v každém stavu může být libovolné množství událostí, ale výstupy budou generovány pouze těm událostem, které jsou v aktivním stavu automatu. Při změně stavu se zapomenou aktuálně prováděné akce. Tzn. když je prováděná nějaká sekvence akcí a najednou se změní stav, tak po návratu do původního stavu se nebude pokračovat tam, kde se skončilo, ale vše znovu začíná od událostí.

Bez použití konečného automatu by plocha uzlů obsahovala hodně proměnných, které pouze rozlišují stav. Použitím automatu se zbavíme těchto proměnných a umožníme lepší přemýšlení nad aktérem.

Pro jednoduší práci je vhodné mít více konečných automatů. Např. jeden automat se může starat o pohyb a animaci aktéra a jiný automat může sloužit jako globální chování aktéra, které platí ve všech stavech prvního automatu. Každý automat funguje samostatně. V každém automatu je aktivní právě jeden stav. Ale změna stavu po hraně se bude týkat všech automatů, kde hrana daného jména v aktuálním stavu existuje.

Konečný automat zjednodušuje ovládání animace aktéra (viz obrázek 1.3).



Obrázek 1.3: Konečný automat aktéra představujícího dveře

Například mějme aktéra dveří, který obsahuje konečný automat znázorněný na obrázku 1.3. Při pohledu na aktéra dveří vidíme jeho stav. Pro otevření resp. zavření dveří stačí změnit stav aktéra.

Definovaná událost

Pomocí navrženého vizuálního skriptování se špatně řeší následující situace. Po aktivaci tlačítka ve hře se má provést nějaká akce, např. otevřít dveře, vytvořit nepřátele, apod. Jediná možnost je definovat toto chování přímo v aktérovi tlačítka. Tento přístup není efektivní a neumožňuje znovu využívání aktérů.

Obecně by se hodila možnost, jak aktér jednoduše oznámí aktérům, kteří o to mají zájem, že se stalo něco podstatného. Proto aktérovi lze vytvořit pojmenovanou událost, kterou může vyvolat. Ostatní aktéři na ní mohou reagovat.

Například než nepřítel zemře, tak vyvolá událost *Smrt* a jiný aktér nebo scéna na to může reagovat tak, že otevře zatím zavřené dveře.

Proměnné aktéra

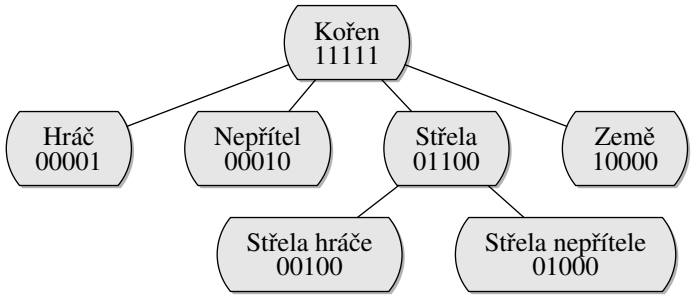
Na virtuální ploše uzlů můžeme používat proměnné, ale ty jsou přístupné pouze v daném stavu a ostatní stavy ani aktéři k nim nemají přístup. Přesto by se hodila možnost mít přístup k určitým hodnotám, jako je např. hodnota zdraví nepřítele.

Řešením je možnost definování pojmenovaných proměnných aktéra. To jsou globální proměnné aktéra, které jsou přístupné pro všechny stavy ve všech jeho automatech ale i pro ostatní aktéry.

1.3 Typ aktéra

Ve skriptování je často potřeba zjistit, o jaký typ aktéra se jedná. Např. jestli střela hráče narazila do nepřítele, apod. Tudíž je vhodné si aktéry zařadit do logických skupin, např. hráč, nepřítel, střela, hráčova střela, apod. Logickou

skupinu, kam aktér patří, budeme nazývat typ aktéra. Z typu aktéra se vyplatí udělat hierarchického uspořádání pomocí stromu (viz obrázek 1.4).



Obrázek 1.4: Strom typu aktérů se svými hodnotami

Mějme strom typu aktérů. Každý vrchol představuje jednu logickou skupinu. Aktér by měl vždy být označen listem stromu. Chceme jednoduše zjistit, jestli jeden typ aktéra je potomkem druhého, nebo naopak. Každý vrchol obsahuje binární číslo. Listy očíslujeme od nuly, pak list má hodnotu 1 << číslo listu. Vrchol stromu kromě listu má hodnotu jako binární logický součet ($\vee$) svých potomků. Potom když pro typ aktéra A a B platí, že $A \wedge B \neq 0$, tak patří do společné skupiny, jinak ne. Příklad je zobrazen na obrázku 1.4.

1.4 Požadované vlastnosti editoru

Editor bude pracovat s projekty. Projekt definuje jednu hru.

Do editoru musí být možnost vložení textury a zvuku a vytvoření animace. Aktéři pak budou využívat tento obsah. Po vytvoření aktéra na scéně budeme chtít upravovat jeho zobrazení (polohu, rotaci nebo velikost).

Hra je postavena na reálné fyzice, takže musíme mít možnost vytvořit kolizní tvary pro aktéry. Kolizní tvary je vhodné mít přiřazené přímo texturám a animacím, aby se nemuseli vytvářet znovu pro každého aktéra. Aktér pak pouze převezme kolizní tvar své grafiky.

Editor obsahuje množství dat, které si budeme moci pojmenovávat. Je nutné zajistit zobrazování aktuálních dat. Např. při změně jména nebo hodnoty proměnné se změna projeví okamžitě všude.

Především díky vizuálnímu skriptování jsou data velice propojena, což komplikuje smazání objektů z projektu. Např. textura může být použita jako grafika aktéra nebo ve vizuálním skriptování jako hodnota proměnné. Nelze pouze smazat texturu, protože bychom dostali nekonzistentní stav. Předtím než bude daný objekt smazán, musí být zkontrolován celý projekt, jestli na mazaném objektu něco nezávisí (především z vizuálního skriptování).

1.5 Podobné programy

Srovnání s podobnými programy je zobrazené v tabulce 1.1.

GameMaker

Editor pro vytvoření 2D i 3D hry. Chování objektů lze vytvořit pomocí

grafického rozhraní (viz obrázek 1.1). Na pokročilejší věci se používá skriptovací jazyk GameMaker Language (GML). Verze GameMaker Lite, která je zdarma, poskytuje pouze základní možnosti pro vývoj 2D hry. Standardní verze pro operační systém Windows a Mac OS X stojí $39.99. Nejvyšší verze GameMaker Studio za $99 obsahuje fyziku. Lze dokoupit moduly pro export hry pro iOS ($199), Android ($199) a technologii HTML5 ($99). [14]

Torque Game Builder 2D
Vývoj 2D her pro operační systém Windows a Mac OS X. Editor obsahuje grafické rozhraní pro vytvoření kola hry, vytvoření objektů a jejich základní úpravu (grafické zobrazení, nastavení fyziky, vrstva pro vykreslení a kolize, apod.). Obsahuje fyziku pro hru. Chování objektů se programuje pomocí C++ nebo skriptovacího jazyka Torquescript. Program stojí $128. [6].

Game Editor
Open source editor pro vývoj multiplatformních 2D her zahrnující operační systém Windows, Mac OS X, Linux, iOS, apod. Hra obsahuje předdefinované události, na které je možno reagovat pomocí základních předdefinovaných akcí nebo především skriptem v programovácím jazyce C. [12]

Unreal Development Kit (UDK)
Vývoj 3D her postavených na herním engine Unreal Engine 3 od společnosti Epic Games. Obsahuje vizuální skriptování (viz. obrázek 1.2) pro spojování uzlů nazvané Kismet. Editor je pouze pro operační systém Windows, ale vytvořená hra lze exportovat na platformy podporující Unreal Engine, např. Mac OS X, iOS. Program je zdarma pro nekomerční účely. [3]

	GameMaker	Torque Game Builder 2D	Game Editor	Unreal Development Kit	2D Platform Game Creator
2D hra	✓	✓	✓	×	✓
3D hra	✓	×	×	✓	×
Fyzika	✓	✓	×	✓	✓
Skriptování	✓	✓	✓	✓	×
Vizuální skriptování	✓	×	×	✓	✓
Multiplatformní editor	✓	✓	✓	×	×
Multiplatformní hra	✓	✓	✓	✓	×
Zdarma	×	×	✓	×	✓
Zdarma pro nekomerční účely	×	×	✓	✓	✓

Tabulka 1.1: Srovnání s vlastnostmi podobných programů

2. Uživatelská dokumentace

Tato kapitola postupně vysvětluje použití programu.

Program pro spuštění vyžaduje .NET Framework 4 (plná verze) a XNA Framework 4 [10]. Pro přidání obsahu do editoru a následného použití ve hře musí být nainstalované XNA Game Studio 4 [11]. Vytvořená hra potřebuje pouze XNA Framework 4. Editor i vytvořená hra vyžadují grafickou kartu podporující minimálně Shader Model 2.0.

2.1 Fyzika

Ve hře probíhá fyzikální simulace pevných těles v rovině. Každé těleso je složeno z kolizních tvarů, což jsou geometrické tvary jako např. kruh nebo polygon. Každé těleso má jisté vlastnosti, především materiální vlastnosti jako je hustota, tření a pružívost (restituce).

Ve fyzikální simulaci se mohou nacházet následující typy těles:

- **Statické (Static)** - Statické těleso se v simulaci nepohybuje, má nulovou rychlost a chová se jako by mělo nekonečnou hmotnost. Může být manuálně posunuto. Statická tělesa nekolidují s dalšími statickými nebo kinematickými tělesy.
- **Kinematické (Kinematic)** - Kinematické těleso se pohybuje podle své rychlosti a chová se jako by mělo nekonečnou hmotnost. Nereaguje na žádné síly. Kinematická tělesa nekolidují s dalšími statickými nebo kinematickými tělesy.
- **Dynamické (Dynamic)** - Dynamické těleso je plně simulované. Koliduje se všemi typy těles.

Fyzikální simulace probíhá v metrech, kilogramech a sekundách a používá radiány pro úhly. Nejlépe pracuje s pohybujícími se tělesy velikosti mezi 0.1 a 10 metrů. Statická tělesa mohou být velké až 50 metrů. Toto je velice důležité vzít na vědomí při vytváření aktérů na scéně.

Tělesa jsou ovládány pomocí sil a impulsů. Není doporučeno tělesa manuálně přesouvat nebo nastavovat dynamickým tělesům rychlost, protože tyto akce mohou vyvolat nefyzikální chování.

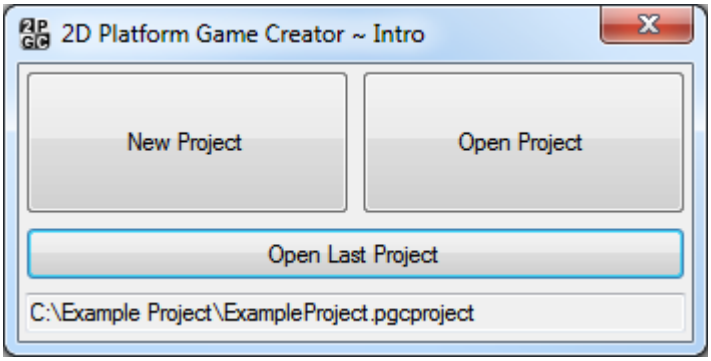
2.2 Projekt

Hra je definovaná pomocí projektu.

Po spuštění programu (viz obrázek 2.1) můžeme vytvořit nový projekt, otevřít projekt nebo otevřít naposledy otevřený projekt. Soubor s projektem má koncovku *.pgcproject*.

Hierarchie projektu na disku:

- **Content** - Složka obsahuje herní obsah a důležité soubory pro spuštění hry z editoru.



Obrázek 2.1: Úvodní okno programu pro výběr projektu

- **Publish** - Složka obsahuje spustitelnou hru po publikaci.
- **Jméno projektu.cs** - Zdrojový kód hry, který je vygenerován při každém spuštění nebo publikování hry.
- **Jméno projektu.pgcproject** - Soubor představující projekt.

2.3 Herní obsah

Textura

Lze vložit obrázek ve formátu PNG, JPEG nebo BMP. Po správném zpracování obrázku se vytvoří nová textura. Pro lepší přehlednost je dobré název textury vhodně pojmenovat, abychom později poznali podle jména, o jakou texturu se jedná. Veškeré nastavení textury najdeme v jejím editoru (viz obrázek 2.2).

Ovládání: Pravé tlačítko myši - pohyb po ploše
Kolečko myši - přiblížení nebo oddálení zobrazení textury
Klávesa Delete - smazání kolizního tvaru po jeho výběrů na seznamu kolizních tvarů

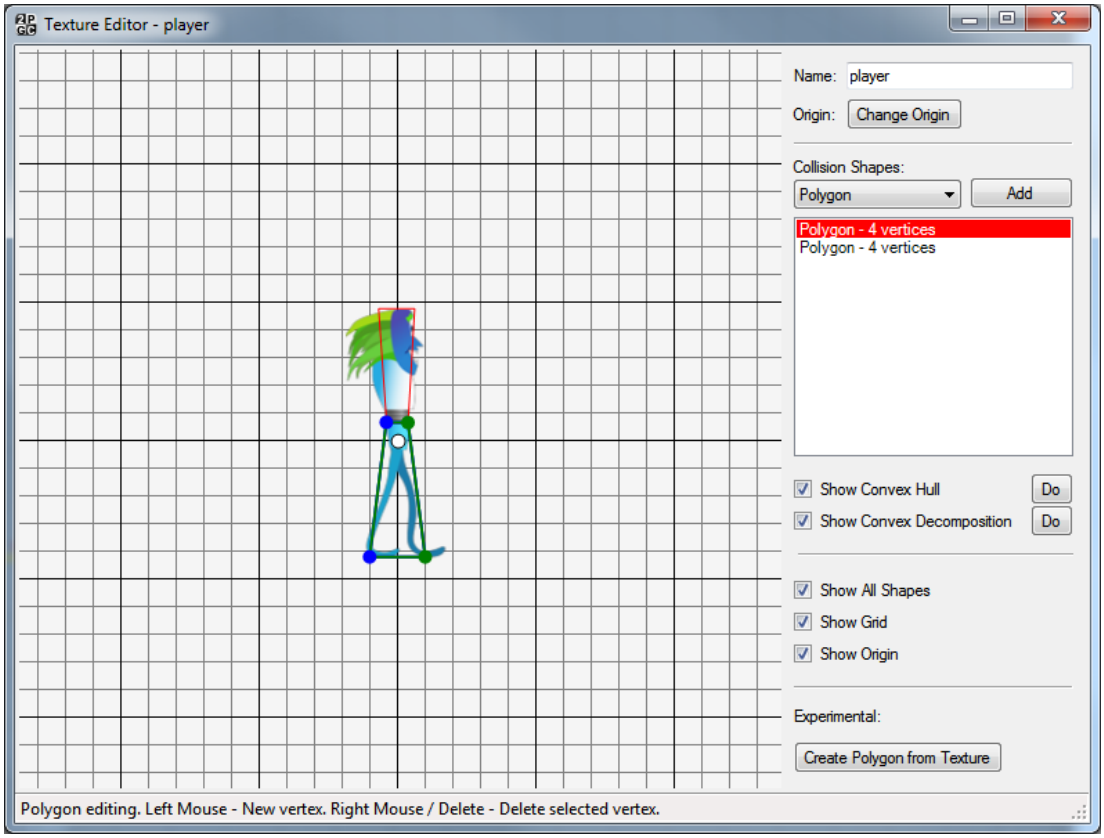
Bod Origin

Poloha bodu Origin, který definuje střed textury, má velký význam při vytváření animace nebo při použití v aktérovi jako jeho grafika. Tento bod se umístí do bodu Origin animace nebo na pozici aktéra, poté představuje pozici aktéra. Aktér se otáčí kolem tohoto bodu.

Kolizní tvary

Textura může obsahovat libovolné množství kolizních tvarů, které budou použity ve hře pro fyzikální simulaci tělesa. Není doporučeno používat velké množství tvarů. Počet tvarů přímo ovlivňuje rychlost fyzikální simulace. Kolizní tvary jsou v editoru rozlišeny pomocí barev. Podporované jsou následující tvary:

- **Mnohoúhelník (Polygon)**



Obrázek 2.2: Nastavení textury

Podporován je pouze konvexní polygon. Je možné vložit konkávní polygon, ale před použitím ve hře bude konkávní polygon rozložen na konvexní polygony stejným způsobem, který je možný v editoru zobrazit (Convex Decomposition). Není možné vytvořit polygon, kterému se protínají hrany. Není doporučeno používat složité polygony (více jak 8 vrcholů).

Ovládání: Levé tlačítko myši - přidání nového vrcholu
Pravé tlačítko myši / klávesa Delete - smazání myší označeného vrcholu

- **Hrana (Edge)**

  Hrana resp. množina hran jsou jednotlivé úsečky, které se mohou protínat. Lze si je představit jako neuzavřený polygon, kterému se mohou protínat hrany.

  Použití ve hře má jistá omezení. Pro správné fungovaní tohoto kolizního tvaru je nutné, aby aktér byl statické nebo kinematické těleso. Při použití v dynamickém tělese nebude tento kolizní tvar fungovat, protože tento tvar není uzavřen, takže v rovině nemá žádnou plochu ani objem.

  Ovládání: Levé tlačítko myši - přidání nového vrcholu
  Pravé tlačítko myši / klávesa Delete - smazání myší označeného vrcholu

- **Kruh (Circle)**

Ovládání: Levé tlačítko myši - vybrání středu kružnice
Pravé tlačítko myši - nastavení poloměru kružnice podle vzdálenosti od středu kružnice ke kurzoru myši

Animace

Vytvoření jednoduché animace po sobě jdoucích textur.

U animace se nenastavuje bod Origin. Ten je pevně určený. Do něj se umisťují body Origin jednotlivých textur. Proto je potřeba vhodně zvolit body Origin jednotlivých textur.

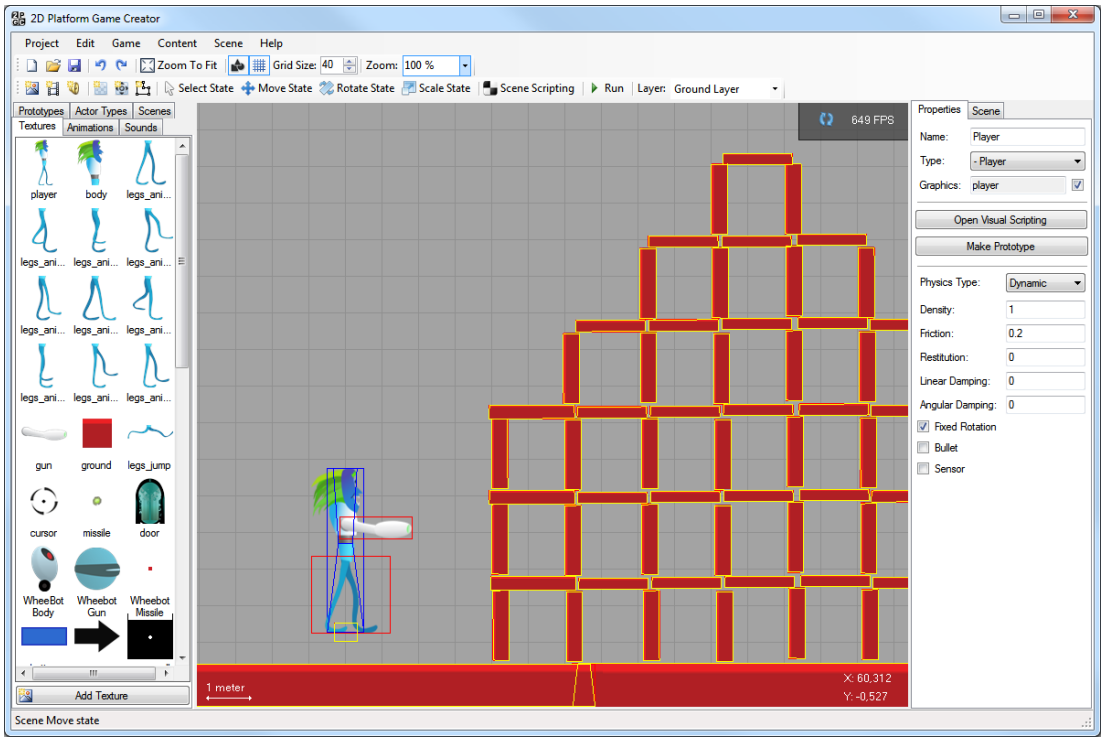
Animace může obsahovat libovolné množství kolizních tvarů, které budou použity ve hře pro fyzikální simulaci tělesa. Stejné vlastnosti a ovládání jako u editace textury.

Zvuk

Lze vložit zvuk ve formátu WMA, MP3, nebo WAV. Po vložení do editoru je možné si ho přehrát.

2.4 Scéna (Scene)

Hra se skládá z minimálně jedné scény. Scéna představuje jednu logickou obrazovku ve hře. Např. každé kolo ve hře je samostatná scéna, ale je také možno udělat více kol na jedné scéně. To už záleží na návrhu hry.



Obrázek 2.3: Hlavní okno editoru

Kromě samotné scény se na ploše scény zobrazují tři důležité informace (viz obrázek 2.3). V pravém horním rohu je aktuální FPS, ve kterém se scéna vy-

kresluje, a především ikonka dvou šipek. Když se scéna vykresluje, tak se ikonka otáčí. Tedy když se ikonka neotáčí, tak se scéna nevykresluje, takže aktuální stav scény může být jiný, než je zobrazeno. Scéna se vykresluje pouze tehdy, když je okno se scénou aktivní. V levém dolním rohu je měřítko jednoho metru. Hodí se kvůli dodržování rozumných velikostí aktérů. V pravém dolním rohu je aktuální pozice kurzoru myši na scéně.

Každý objekt na scéně je ohraničen obdélníkem. U aktéra se ohraničují kolizní tvary jeho grafiky a u cesty samotná cesta. Tento obdélník představuje aktivní oblast objektu. Pokud chceme objekt vybrat nebo na něm provádět nějakou akci, tak musíme pracovat s jeho aktivní oblastí.

Při vybrání jednoho objektu na scéně se zobrazí jeho vlastnosti v pravém sloupci editoru v tabu Vlastnosti (Properties).

Ovládání: Levé tlačítko myši - vybírání objektů na scéně
Levé tlačítko myši + Ctrl - přidání / odebrání objektu z vybraných objektů na scéně
Levé tlačítko myši + Alt - vybírání objektů na scéně pomocí obdélníku
Pravé tlačítko myši / Kurzorové šipky - pohyb po scéně
Kolečko myši - přiblížení nebo oddálení scény
Ctrl + C - kopírování vybraných objektů do schránky scény (kopírovat)
Ctrl + V - vložení objektů na scénu ze schránky scény (vložit)
Ctrl + Z - vrácení poslední akce (zpět)
Ctrl + Y - znovu provedení poslední akce (znovu)
Delete - smazání vybraných objektů na scéně

Stavy pro editaci objektů na scéně:

- **Vybírání (Select State)**

  Nepřidává žádnou další funkčnost. Velice se hodí, když procházíme scénu nebo pouze upravujeme nastavení objektů, ale nechceme omylem nic na scéně změnit.

- **Přesun (Move State)**

  Posun vybraných objektů na scéně.

  Ovládání: Stiskem levého tlačítka myši nad některým z vybraných objektů a následným posunem myši lze přesouvat dané objekty na scéně.

- **Otáčení (Rotate State)**

  Otáčení vybraných objektů na scéně. Aktér se otáčí kolem svého bodu Origin. Cesta se otáčí kolem prvního vrcholu cesty.

  Ovládání: Stiskem levého tlačítka myši nad některým z vybraných objektů a následným posunem myši kolem středu vybraného objektu se budou otáčet všechny objekty o stejný úhel.

- **Změna velikosti (Scale state)**

Změna velikosti vybraných objektů na scéně. Upozornění k aktérům: Změna velikosti se aplikuje na grafiku aktéra (textura nebo animace) a ne na jeho ohraničující obdélník. Toto chování se může zdát malinko divné, když se upravuje otáčený aktér. Není podporována změna velikosti, která mění originální poměr stran aktéra, kolizního tvaru kruhu (z kruhu by vznikla elipsa). Aktérovi lze libovolně změnit velikost, ale na kruh se bude aplikovat pouze X-ová část koeficientu pro změnu velikosti.

Ovládání: Stiskem levého tlačítka myši nad některým z vybraných objektů a následným posunem myši se bude měnit velikost všech objektů o stejný koeficient. Při stisku levého tlačítka Shift se bude zachovávat původní poměr stran objektu (uniform scale).

2.5 Vrstva (Layer)

Scéna se skládá z libovolného počtu vrstev. Do vrstvy se přidávají jednotliví aktéři. Vždy je vybraná jedna vrstva, do které se aktéři budou přidávat. Hra se vykresluje po jednotlivých vrstvách odspoda nahoru a ve vrstvě se taktéž vykreslují aktéři odspoda nahoru.

Parallax vrstva

Speciální vrstva simuluje 3D hloubku pohyblivého obrazu ve 2D. Aktéři nemají pevnou pozici na scéně, ale jejich pozice záleží na aktuální pozici kamery. Lze vložit pouze aktéry, kteří se nevyskytují ve fyzikální simulaci. Nastavení vrstvy (zobrazení pomocí pravého tlačítka myši):

- **Parallax Coefficient** - Koeficient pro ovlivňování pozice aktérů na scéně. K pozici aktéra se přičte součin pozice kamery a tohoto koeficientu.
- **Graphics Effect** - Efekt, který se bude aplikovat na grafiku aktérů.
  - **Repeat Horizontally** - Grafika aktérů se bude horizontálně opakovat vedle sebe.
  - **Repeat Vertically** - Grafika aktérů se bude vertikálně opakovat vedle sebe.
  - **Fill** - Grafika aktérů se bude opakovat do všech směrů tak, že vyplní celou viditelnou část scény.

2.6 Cesta (Path)

Scéna obsahuje libovolné množství cest. Například nepřítel ve hře se bude moci pohybovat po předem vytvořené cestě.

Vybráním cesty na scéně se zobrazí její vlastnosti a nastavení. Opět je velice doporučeno cestu vhodně pojmenovat, abychom ji později poznali pouze podle názvu. Cestu lze uzavřít pomocí možnosti *Loop*. Při samotné editaci cesty ji není možné uzavřít tím, že bychom dali vrcholy začátku a konce přes sebe.

Ovládání: Levé tlačítko myši - přidání nového vrcholu na cestu
Delete - smazání myší označeného vrcholu na cestě
Esc / Enter - ukončení editace cesty

2.7 Aktér (Actor)

Aktér představuje herní objekt ve hře. Může to být např. hráč, nepřítel, střela hráče nebo nepřítele a dokonce země i pozadí. Jednoduše řečeno to je cokoliv, co chceme zobrazit, použít nebo detekovat ve hře.

Přetažením textury nebo animace na scénu se vytvoří nový aktér do aktuálně vybrané vrstvy. Aktér se vytvoří na pozici kurzoru myši. Pozici aktéra udává bod Origin jeho grafiky. Tedy na pozici kurzoru myši bude bod Origin přenášené textury nebo animace.

Vybráním aktéra na scéně se zobrazí jeho vlastnosti a nastavení (viz obrázek 2.3). Opět je velice důležité vhodně aktéra pojmenovat, aby byla scéna přehledná a abychom ho poznali podle názvu. Textura nebo animace, pomocí které jsme aktéra vytvořili lze změnit. Stačí přetáhnout novou texturu nebo animaci do políčka, kde je uveden název aktuální grafiky aktéra. Je možné grafiku aktéra nezobrazovat. Vhodné pro aktéry, kteří budou představovat roli senzorů. (Aktuálně není možné vytvořit aktéra přímo na scéně tak, že bychom ho definovali pomocí kolizních tvarů. Ale pouze přes texturu nebo animaci. Tedy pro senzor musíme použít texturu s vhodným kolizním tvarem a nezobrazovat jeho grafiku.) Nastavení fyzikálního typu aktéra představuje jeho typ tělesa ve fyzice popsané v 2.1. Aktéra můžeme vyloučit z fyzikální simulace možností *None*. Po vybrání konkrétního fyzikálního typu máme možnost nastavit vlastnosti, které se použijí ve fyzikální simulaci. Možné vlastnosti:

- **Hustota (Density)** - Hustota slouží ke spočítání hmotnosti tělesa z jeho kolizních tvarů. Implicitní hodnota je 1.
- **Tření (Friction)** - Tření se používá, aby po sobě mohli tělesa realisticky klouzat. Jeho hodnota je obvykle mezi 0 a 1, ale může být i záporná. Hodnota 0 tření vypne a naopak hodnota 1 způsobí silné tření. Implicitní hodnota je 0,2.
- **Pruživost (Restitution)** - Pruživost (restituce) se používá, aby mohli tělesa skákat. Jeho hodnota je obvykle mezi 0 a 1. Uvážíme-li, že upustíme míč na stůl, tak hodnota 0 znamená, že se míč neodrazí, naopak hodnota 1 způsobí, že míč se odrazí stejně jako je jeho rychlost dopadu. Implicitní hodnota je 0.
- **Linear Damping** - Redukuje pohybovou rychlost tělesa. Implicitní hodnota je 0, tedy nic se neredukuje.
- **Angular Damping** - Redukuje rotační rychlost tělesa. Implicitní hodnota je 0, tedy nic se neredukuje.
- **One Way Platform** - Z tělesa udělá plošinku, která se velmi používá ve 2D hrách, na které je možné stát, ale lze zespoda proskočit. Taková plošinka

rozhodně porušuje fyzikální simulaci, protože se jedná o nereálné těleso. Díky tomu je tato možnost velice omezená. Funguje pouze pro plošinky, které mají jako kolizní tvar jeden polygon, a gravitace je v implicitním směru (směrem dolů). Další tělesa, která tuto plošinku budou ve hře používat, by měli obsahovat pouze jeden kolizní tvar. I při dodržení těchto pravidel se může stát, že plošinka nebude vždy fungovat správně. To vše je důsledkem nereálností tohoto tělesa ve fyzikální simulaci a obecnosti hry. Nastavení lze aplikovat pouze na statické nebo kinematické těleso.

- **Fixed Rotation** - Zakáže rotaci tělesa. Těleso se nebude otáčet ani pod velkým zatížením. Velice se hodí se pro hráče a nepřátele. Nastavení lze aplikovat pouze na dynamické těleso, protože u ostatních nemá smysl.

- **Bullet** - Je velice doporučeno, aby rychle pohybující se tělesa měli toto nastavení povolené. Fyzikální simulace, pak bude takovým tělesům věnovat více pozornosti, aby se náhodou nestalo, že těleso proletí jiným tělesem. Vhodné např. pro střely hráče nebo nepřátel. Nastavení lze aplikovat pouze na dynamické těleso.

- **Senzor (Sensor)** - Z tělesa se stane senzor. Těleso normálně zaregistruje kolizi, ale nebude se na ní odpovídat, jak by se normálně stalo. Logika hry někdy potřebuje zjistit, jestli jsou dvě tělesa v kolizi, ale nechce na kolizi nijak fyzikálně reagovat.

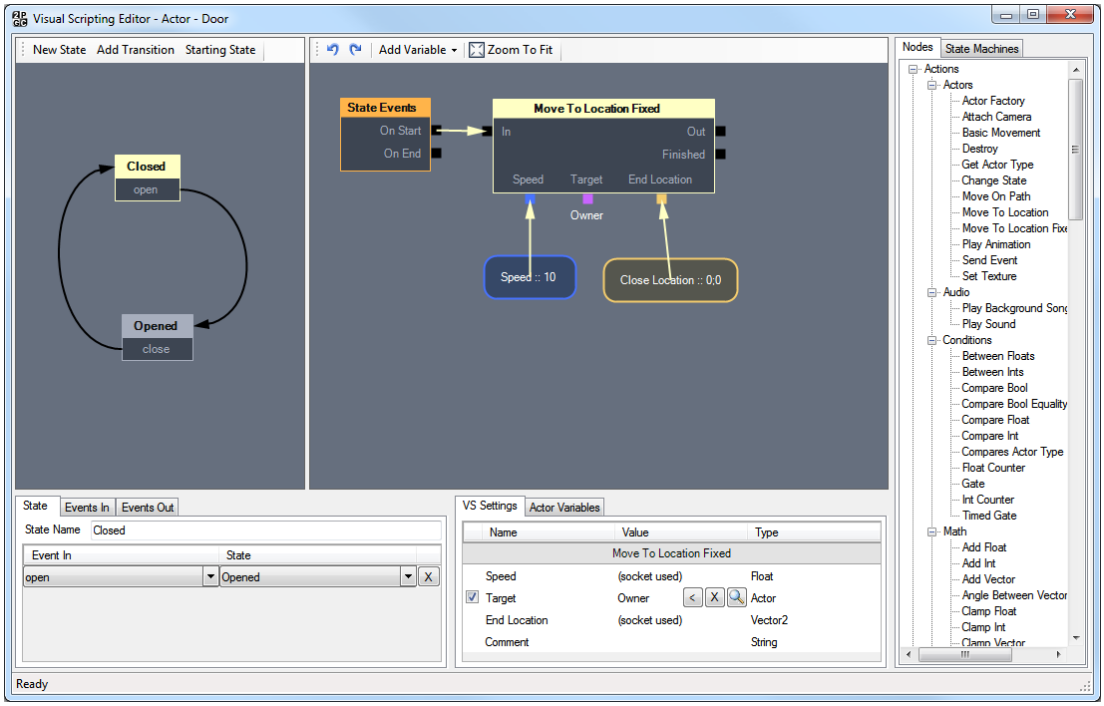
Aktér může obsahovat děti resp. další aktéry. Tzn. že aktér může být složen z více samostatných aktérů, kteří dohromady vytvářejí jeden logický celek. Např. hráč může obsahovat jako dítě pohyblivou ruku. Aktéři mohou být libovolně zanoření. Výsledný aktér (celek) bude ve fyzikální simulaci pouze jako jedno těleso, resp. všichni aktéři se sloučí do jednoho tělesa. U dětí je omezen výběr fyzikálního typu. Děti mohou mít pouze stejný fyzikální typ jako jejich rodič nebo nemusí být ve fyzikální simulaci (typ *None*). Tím se zamezí vzniku nelogických situací. Při vykreslování je nejdříve vykreslen rodič a pak jeho děti opět odspoda nahoru.

Aktér převezme kolizní tvar ze své grafiky (textury nebo animace). Tento tvar již není možné více upravovat. Aktér nemůže změnit svůj kolizní tvar ani pomocí skriptování. Pomocí skriptování může aktér změnit svojí grafiku, ale tím se nezmění jeho kolizní tvar. Tzn. aktér bude mít po celou dobu svojí existence kolizní tvar, který můžeme vidět v editoru na scéně. Jediná akce, která má za následek změnu kolizního tvaru aktéra, je pouze tehdy, když aktér obsahuje nějaké dítě. Když se jeho dítě ve hře zničí, tak se zničí i jeho kolizní tvar. Tedy z kolizních tvarů aktéra se odebere kolizní tvar dítěte.

2.8 Vizuální skriptování (Visual Scripting)

Bez možnosti skriptování by bylo možné vytvořit pouze statickou scénu. Tzn. scénu, na které pouze uvidíme fyziku v praxi. Například jak aktéři padají na zem nebo si vyzkoušet různá fyzikální nastavení.

Vizuální skriptování obsahuje každý aktér ale i každá scéna. V dalším textu bude popsáno skriptování především z pohledu aktéra. Vizuální skriptování aktéra (viz obrázek 2.4) otevřeme v jeho vlastnostech po vybrání aktéra na scéně.

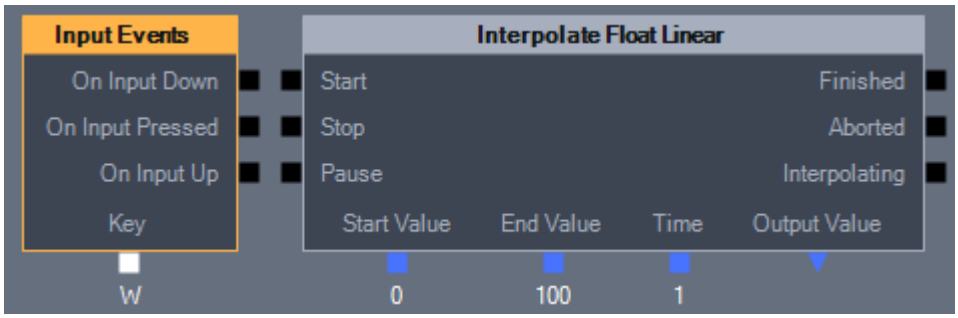


Obrázek 2.4: Okno vizuálního skriptování

Spojování uzlů (Nodes)

Na plochu přetahujeme z pravého sloupce jednotlivé uzly a pak je logicky propojujeme. Propojení uzlů musí být vytvořeno tak, aby nezáviselo na pořadí vykonávání jednotlivých sekvencí uzlů. Tzn. nezáleží na pořadí vyvolání jednotlivých událostí a při připojení více uzlů na jeden výstup nezáleží v jakém pořadí budou uzly vykonány. Skriptování obsahuje následující typy uzlů:

Událost (Event Node) a Akce (Action Node)



Obrázek 2.5: Událost a akce ve vizuálním skriptování

Typy uzlů jsou rozlišeny pomocí barev. Levá strana uzlu obsahuje vstupy (signal in sockets), pravá výstupy (signal out sockets) a dolní strana proměnné (variable sockets). Výstupní proměnná je zobrazena jako trojúhelník. Ukázka uzlů je zobrazena na obrázku 2.5.

Proměnná (Variable Node)

Proměnné umožňují uložit hodnotu daného typu. Nejdůležitější typy jsou rozlišeny pomocí barev. Barvy pomáhají propojování uzlů. Typy proměnné:

- **Bool** - Hodnota ano (true) / ne (false)
- **Int** - Celé číslo. Např. 5; 42; −897; 0
- **Float** - Číslo s plovoucí desetinnou čárkou. Např. 5; −42; 789,78; −42.987; 8E+5
- **Vector2** - Dvourozměrný vektor. Souřadnice jsou oddělené středníkem a každá část je typu *Float*. Např. 0;0 nebo −42;789,78
- **String** - Řetězec resp. text omezený pouze na ASCII znaky
- **Actor** - Aktér na scéně. Nastavení obsahuje tři tlačítka. První se symbolem < nastaví aktuálně vybraného aktéra na scéně, pokud je na scéně vybrán jeden aktér. Prostřední se symbolem *X* vymaže aktuální hodnotu. Poslední tlačítko zobrazí aktuální hodnotu aktéra na scéně. Některé akce a události mají proměnnou implicitně nastaveny jako *Owner*. Tzn. jako jeho hodnota bude nastaven aktér, ve kterém se příslušný uzel nachází. Velice zjednodušuje práci.
- **Actor Type** - Typ aktéra
- **Path** - Cesta
- **Texture** - Textura
- **Animation** - Animace
- **Sound** a **Song** - Oba typy označují zvuk z editoru, ale každý má jiné použití. *Sound* (zvuk) se používá pro krátký zvuk, který bude použit ve hře. Např. zvuk výstřelu. Z interních záležitostí je důležité vědět, že tento zvuk bude při hře uložen v nekomprimovaném formátu a ve hře bude celý načten do paměti. Naopak *Song* (píseň) se používá pro delší zvuky resp. písně. Tento typ bude především použit pro přehrávání hudby na pozadí hry. Písně budou při hře uloženy ve formátu WMA a ve hře se budou pouze streamovat, takže nebudou zabírat velké množství paměti.
- **Scene** - Scéna
- **Key** - Klávesa

Po vybrání uzlů na ploše se zobrazí jejich vlastnosti v pravé dolní části editoru v tabu Nastavení (VS Settings).

Ovládání: Levé tlačítko myši - vybírání uzlů na ploše
Levé tlačítko myši + Ctrl - přidání / odebrání uzlu z vybraných uzlů na ploše
Levé tlačítko myši + Alt - vybírání uzlů na ploše pomocí obdélníku
Pravé tlačítko myši - pohyb po ploše

Kolečko myši - přiblížení nebo oddálení zobrazení
Ctrl + C - kopírování vybraných uzlů do schránky (kopírovat)
Ctrl + V - vložení uzlů na plochu ze schránky (vložit)
Ctrl + X - kopírování do schránky a následné smazání vybraných uzlů na ploše (vyjmout)
Ctrl + Z - vrácení poslední akce (zpět)
Ctrl + Y - znovu provedení poslední akce (znovu)
Delete - smazání vybraných uzlů na ploše

Konečný automat (State Machine)

Konečný automat obsahuje alespoň jeden stav. Hrana automatu se skládá ze dvou částí. Stavu, do kterého se přejde, a jména hrany (event in). Jména hran se definují v levé dolní části editoru v tabu Názvy hran (Events In). Zde definujeme názvy hran, které budeme používat. Samotné změny stavů se vykonávají pomocí uzlů, kde v akci pouze určíme název hrany. Potom se zjistí, jestli v aktuálním stavu existuje hrana s daným názvem a pokud ano, tak se změní stav automatu. Z tohoto důvodu je velice důležité vhodně a výstižně zvolit název hrany.

Ovládání: Levé tlačítko myši - vybrání stavu na ploše nebo definování hrany mezi stavy
Dvojklik na stav - zobrazení uzlů vybraného stavu
Pravé tlačítko myši - pohyb po ploše
Delete - smazání vybraného stavu na ploše

Definovaná událost (Event Out)

V levé dolní části editoru v tabu Definované události (Events Out) si můžeme definovat vlastní události. Je velice vhodné zvolit výstižné jméno, aby bylo zřejmé, co událost znamená. Pomocí uzlů můžeme tyto události vyvolat. Jiný aktér pokud má zájem, tak může reagovat na tuto událost. A to tak, že ve svém vizuálním skriptování použije událost, která čeká na konkrétní definovanou událost jiného aktéra.

Proměnná aktéra (Actor Variable)

Proměnnou aktéra definujeme v pravé dolní části editoru v tabu Proměnné aktéra (Actor Variables). Zde je opět vhodné zvolit výstižné jméno a typ proměnné. Po definování proměnné ji můžeme použít na aktuální ploše uzlů pomocí tlačítka se symbolem > vedle proměnné.

Proměnné ostatních aktérů nebo globální proměnné vložíme na plochu uzlů pomocí nabídky Vložit proměnnou (Add variable), kde zvolíme globální proměnnou nebo proměnnou aktuálně vybraného aktéra na scéně.

Vizuální skriptování scény (Scene Scripting)

Skriptování scény se otevře z hlavního okna editoru (viz obrázek 2.3), kde se edituje scéna. Skriptování scény je úplně stejné jako skriptování v aktérech. Jediný

přídavek je, že proměnným ve skriptování scény se říká globální proměnné a aktéři mají k těmto proměnným jednoduší přístup.

Velice se hodí pro nadefinování většiny logiky hry a pomocí automatu se dají jednoduše vytvořit různé části kola ve hře.

2.9 Prototyp aktéra (Prototype)

Pokud aktéra dobře vytvoříme, především jeho vizuální skriptování, můžeme z něj vytvořit prototyp. Prototyp aktéra slouží už podle názvu jako prototyp a můžeme ho používat mezi různými scénami. Normálního aktéra na scéně nemůžeme nijak přenést ani zkopírovat na jinou scénu, protože může mít závislosti na scénu, ve které se nachází.

Prototyp vytvoříme ve vlastnostech aktéra po jeho vybrání na scéně. Pokud aktér obsahuje nějakou závislost na scénu, především hodnoty aktérů ze scény v proměnné typu *Actor*, tak na to budeme upozorněni a budeme mít možnost automaticky zrušit takové závislosti. Tedy aktér se zkopíruje a pokud obsahuje zmíněné závislosti, tak proměnné, jejichž hodnoty mají závislost na scénu, budou nastaveny na implicitní hodnotu. Jediné povolené odkazy na jiné aktéry jsou odkazy na svoje děti nebo rodiče. Tedy aktér jako prototyp může v proměnných typu *Actor* mít hodnoty svých dětí nebo rodičů a taktéž může používat jejich proměnné.

2.10 Nastavení hry (Game Settings)

- **Velikost okna hry (Game Window)** - Velikost (rozlišení) okna hry.
- **Simulation Units** - Přepočet na metry ve fyzikální simulaci. Číslo udává kolik pixelů odpovídá jednomu metru. Implicitní hodnota je 100, tedy 100 pixelů je 1 metr. Toto číslo je mít vhodně nastavené tak, aby byli dodrženy doporučené velikosti aktérů. Změna nastavení vizuálně neovlivní pozice objektů na scéně ani kolizní tvary aktérů, ale pouze veškeré hodnoty ve skriptování, které se týkají fyziky a pozice na scéně, a také nastavení gravitace. Resp. jejich hodnoty se nezmění, ale jejich vizuální význam ano. Pokud například na aktéra působíme nějakým impulsem a změníme hodnotu z 50 na 100, tak impuls bude mít ve hře vizuálně dvojnásobnou velikost (účinnost) oproti původnímu nastavení.
- **Gravitace (Default Gravity)** - Gravitace ve hře. Pomocí skriptování lze změnit.
- **Barva pozadí (Background Color)** - Barva pozadí ve hře. Hodnota je v RGB. Barva se zobrazuje pouze ve hře ale ne v samotném editoru.
- **Continuous Collision Detection** - Přesnější kontrola kolizí, aby rychle se pohybující těleso neproletělo jiným tělesem. Při vypnutém nastavení je hra výrazně rychlejší, ale může se stát, že rychlé těleso proletí jiným tělesem.

2.11 Testování hry (Run Game)

Hra lze přímo spustit z editoru a otestovat. Hra bude začínat aktuálně otevřenou scénou. Tímto způsobem spuštěná hra obsahuje pár vlastností navíc, které slouží k testování. Vykreslují se kolizní tvary aktérů a cesty. Kolizní tvar má odlišnou barvu, pokud je aktér v kolizi. Scénu lze oddálit nebo přiblížit, abychom měli celkový přehled o scéně. Tyto možnosti nebudou obsaženy v publikované hře.

Ovládání: F1 - vypnutí zobrazování kolizních tvarů aktérů a cest
Kolečko myši - přiblížení nebo oddálení scény

2.12 Publikování hry (Publish Game)

Pro získání hry se v adresáři projektu vytvoří složka *Publish*, kde se bude nacházet spustitelná hra se vším potřebným pro svůj běh. Stačí publikovat obsah této složky. Hra začíná aktuálně otevřenou scénou.

Při každém publikování hry se složka *Publish* smaže, takže není doporučeno do ní nic ukládat. Nelze vygenerovat hru, pokud se nepodaří složku smazat.

2.13 Klonování aktérů

Při kopírování skupiny objektů na scéně přes schránku se zkontroluje jejich propojení a u kopií se vytvoří stejná propojení. Skupina objektů se zkopíruje. Poté se zkontroluje, jestli neobsahují objekty, které se kopírovali. Pokud ano, tak se jejich hodnoty nastaví na nové hodnoty.

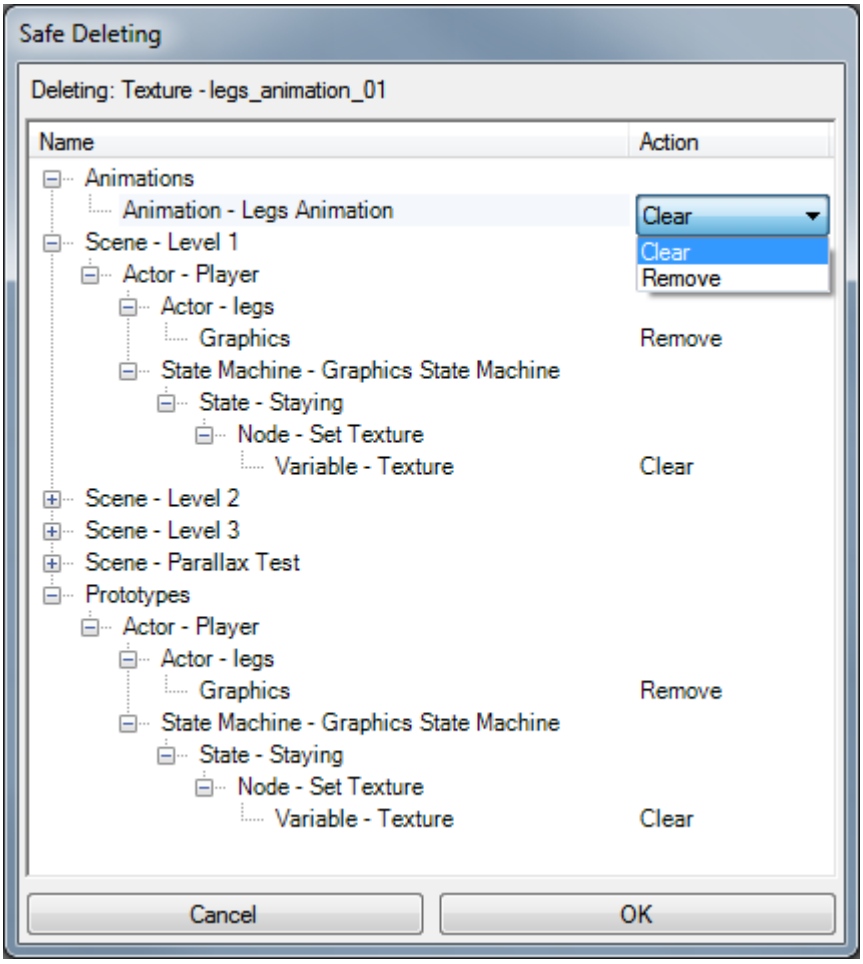
Mějme aktéry A a B. B ve svém vizuálním skriptování obsahuje hodnotu A. Při klonování samotného B bude kopie B obsahovat ve svém vizuálním skriptování hodnotu A. Pokud klonujeme celou skupinu (A i B), pak vytvořené kopie jsou propojené mezi sebou, resp. kopie B obsahuje hodnotu kopie A. Nechť C má odkaz na svoje dítě. Po klonování C má kopie C odkaz na své nové dítě. Nechť D obsahuje proměnnou dítěte E. Po klonování D i E, bude kopie D obsahovat proměnnou dítěte kopie E.

Stejné chování platí i při vytváření aktéra za běhu hry.

2.14 Bezpečné smazání

Před smazáním objektu z projektu se musí zkontrolovat jeho závislosti, jestli je někde použit.

Obrázek 2.6 ukazuje okno při smazání textury z projektu. Zobrazují se místa, kde je objekt použit včetně celé cesty. U každé závislosti můžeme rozhodnout, jak s ní bude naloženo. Jsou na výběr dvě možnosti *Vyčistit* (*Clear*) a *Smazat* (*Remove*). Každá možnost představuje logickou možnost pro daný typ závislosti. Například na obrázku 2.6 první závislost představuje použití textury v animaci, tedy textura se může z animace smazat (možnost *Clear*) nebo se odstraní animace (možnost *Delete*). Textura je použita jako grafika aktéra, tedy aktérovi můžeme grafiku smazat nebo odstranit aktéra. Dále je textura použita v proměnné uzlu. Zde lze proměnnou vyčistit (nastavit implicitní hodnotu) nebo uzel smazat.



Obrázek 2.6: Smazání textury z projektu

Smazání může mít více fází. Při odstranění animace na obrázku 2.6 se opět zkontrolují závislosti animace. Pokud je animace někde použita, tak dostaneme další závislosti pro animaci.

Bezpečné smazání je použito všude, kde mohou vznikat závislosti. Smazaní textury, typu aktéra, aktéra ze scény, vrstvy ze scény, proměnné aktéra, apod.

3. Programátorská dokumentace

V této kapitole je popsána konkrétní implementace programu, její architektura a uvedena struktura tříd.

Program je rozdělen na dva projekty. Editor pro vytváření hry a herní engine, který používá vytvořená hra.

Pro implementaci byl zvolen jazyk C# 4.0 (.NET Framework 4.0).

3.1 Použité knihovny

XNA Framework
: Microsoft XNA je framework určený pro vývoj her pro osobní počítače, herní konzoli Xbox 360 a Windows Phone. Je postavený nad technologií DirectX a snaží se odstínit odlišnosti jednotlivých platforem. Obsahuje pouze základní stavební kameny pro hru. K dispozici je herní smyčka a základní možnost vykreslování, která je pro 2D hru dostačují. Data se do hry načítají pomocí mechanismu Content Pipeline [2]. [10]

Farseer Physics Engine
: Farseer Physics Engine je fyzikální engine pro simulaci pevných těles v rovině. Jedná se o přepsanou knihovnu Box2D [1] do jazyka C# a optimalizovanou pro XNA Framework. Knihovna je open source a distribuována pod Microsoft Permissive License (Ms-PL) licencí. [4]

Windows Forms
: Windows Forms je knihovna součástí .NET Framework pro vývoj grafických aplikací. [13]

TreeViewAdv for .Net
: Pokročilejší TreeView než obsahuje knihovna Windows Forms. Obsahuje sloupce, více ovládacích prvků pro jeden uzel stromu a graficky znázorněné Drag&Drop. Knihovna je open source a distribuována pod BSD licencí. [7]

3.2 Game Engine

Hra je reprezentovaná třídou PhysicsGame, která dědí od třídy Game z XNA Framework. Game zajišťuje herní smyčku a automaticky volá metody *Draw* a *Update*. Pokud FPS hry klesne pod 60, tak se metoda *Draw* vynechává. Inicializace hry je provedena dvojicí metod *Initialize*, inicializace ne-grafického obsahu, a poté *LoadContent* pro načtení grafického obsahu. Naopak *UnloadContent* odstraní grafický obsah. Většina tříd používá tento přístup.

Jmenné prostory

Jmenný prostor *PlatformGameCreator.GameEngine* zastřešuje vše týkající se herního engine. Zde se nachází PhysicsGame, správce vstupu a pomocné struktury, které používá i editor. Ostatní třídy jsou rozděleny do jmenných podprostorů, podle svého určení:

PlatformGameCreator.GameEngine.Assets
: Nachází se zde herní obsah (textura a animace), který obsahuje data z editoru.

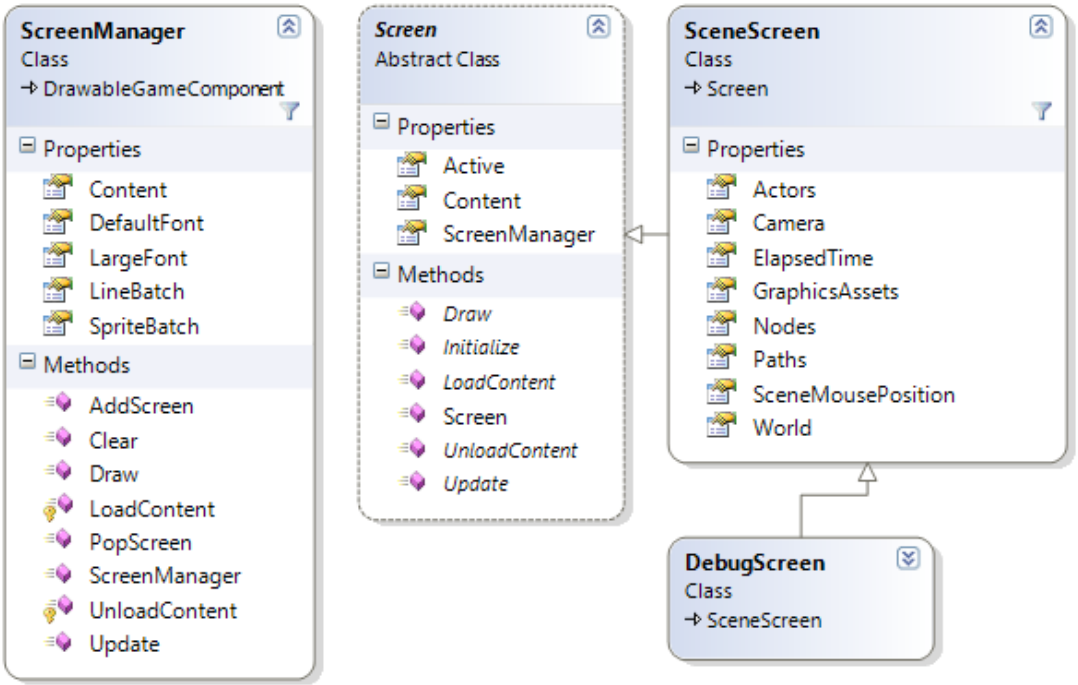
PlatformGameCreator.GameEngine.Scenes
: Třídy týkající se scény. Scéna, bázová třída pro objekt na scéně, aktér, cesta a kamera scény.

PlatformGameCreator.GameEngine.Screens
: Správce obrazovek a bázová třída pro obrazovku.

PlatformGameCreator.GameEngine.Scripting
: Zastřešuje skriptování. Bázový uzel pro skriptování, atributy pro uzel, stav aktéra a obálka proměnné a události. Akce se nacházejí ve jmenném podprostoru *Scripting.Actions* a události v *Scripting.Events*.

3.2.1 Správa obrazovek

PhysicsGame obsahuje jedinou komponentu ScreenManager (viz obrázek 3.1) pro správu obrazovek, která dědí od DrawableGameComponent z XNA. Objekty typu DrawableGameComponent obsahují metody *Initialize*, *Update*, *Draw* apod., stejně jako třída Game. Tyto metody jsou automaticky volány vždy po provedení metod stejného názvu v Game.



Obrázek 3.1: Správa obrazovek a obrazovky

ScreenManager obsahuje objekty, které mohou používat všechny obrazovky. ContentManager pro načítání obsahu pomocí mechanismu Content Pipeline, SpriteBatch pro kreslení textur a dva druhy základních fontů. Pokud při fázi *Update* nebo *Draw* objekt neobsahuje žádnou obrazovku, tak ukončí hru.

Obrazovky dané třídou Screen (viz obrázek 3.1) jsou uloženy v zásobníku. Vrchol zásobníku je jediná aktivní obrazovka. Ostatní obrazovky nejsou aktualizovány ani kresleny.

3.2.2 Scéna

Obrazovka SceneScreen (viz obrázek 3.1) představuje scénu hry. Z této obrazovky dále dědí DebugScreen, která navíc umožňuje zobrazení kolizních tvarů a oddálení kamery, pro testování hry při spuštění z editoru.

Scéna obsahuje stejný obsah jako v editoru. *Paths* obsahuje cesty, *Nodes* objekty na scéně a *Actors* obsahuje aktéry, kteří jsou dostupní ve skriptování. Do *Actors* jsou přidáni aktéři pouze během inicializace, ne aktéři vytvoření za běhu. Slouží pro hledání aktérů ve skriptování.

Objekt na scéně je reprezentován třídou SceneNode (viz obrázek 3.2). Třída obsahuje typické metody *Initialize*, *Update* a *Draw*.

Objekty na scéně nejsou přímo uloženy ve vrstvách, ale každý objekt obsahuje číslo značené *Layer*, podle kterého jsou objekty setříděny. Třída SpriteBatch umí kreslit textury podle z-bufferu, ale interně objekty před vykreslením setřídí. Proto raději máme objekty uložené už setříděné.

Objekty používají jednotky, které jsou použity ve fyzikální simulaci. Při vykreslování se převádí na pixely.

3.2.3 Aktér

Abstraktní třída Actor (viz obrázek 3.2) představuje aktéra. Konkrétní implementace aktéra se sama nastaví.

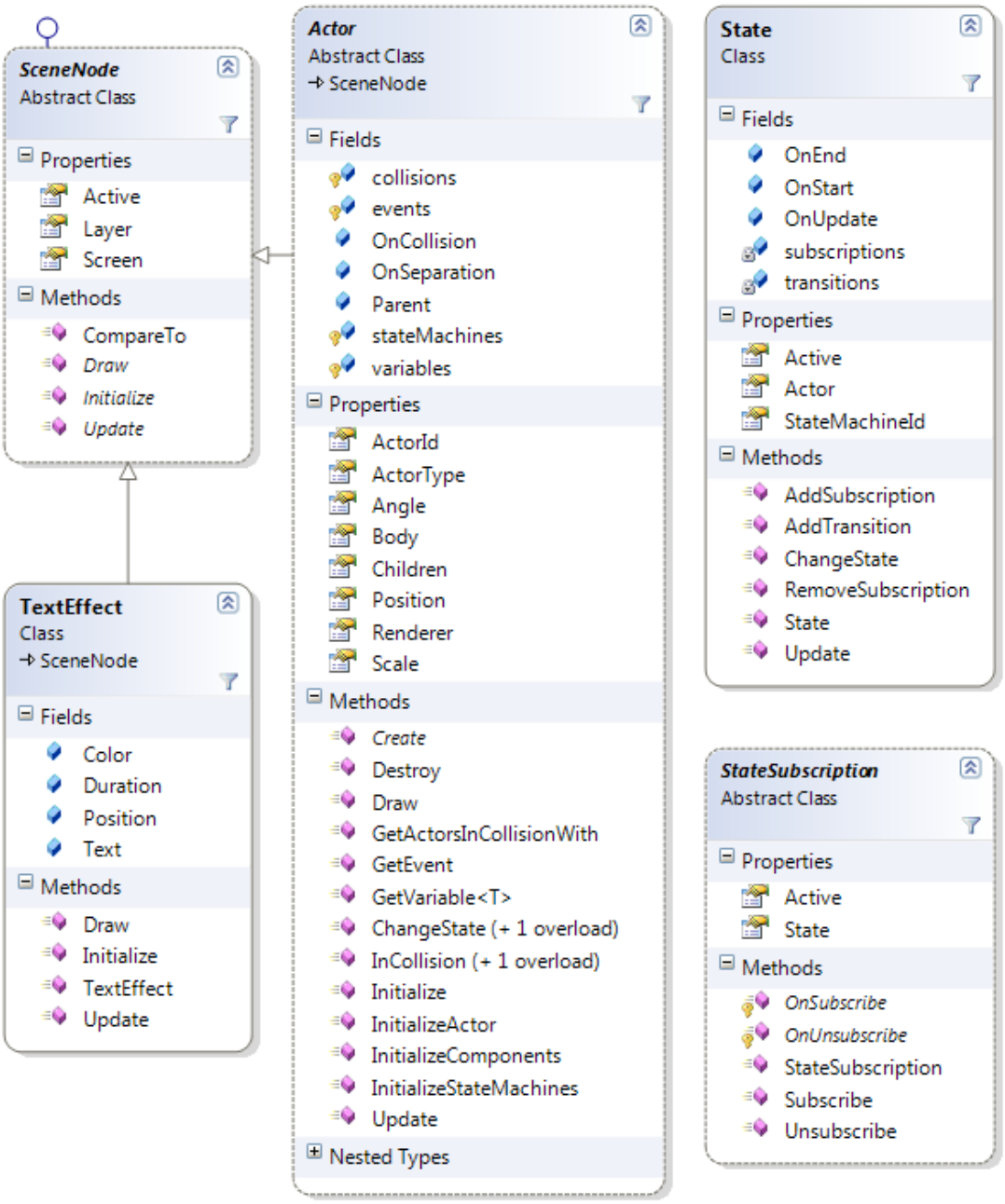
Aktér je vykreslen pomocí implementace bázové třídy ActorRenderer (viz obrázek 3.4), která představuje grafiku aktéra.

Každá třída aktéra (potomci) je identifikována pomocí čísla *ActorId*, které je totožné jako *Id* aktéra v editoru. *ActorId* slouží pro hledání konkrétního aktéra na scéně.

Actor si v proměnné *collisions* pamatuje aktéry, s kterými je v kolizi. Pamatuje si nejen aktéry, ale i počet kolizí s aktérem. Jakmile je tělo složeno z více tvarů, tak se kolize řeší pro každý tvar zvlášť. Aktér dostane informaci z fyzikální simulace, pokud se dostane do kolize nebo separace těles, z nichž jedno patří jemu. Pokud v *collisions* neexistuje záznam o kolizi s daným aktérem, tak ho vytvoříme, oznámíme rodičům vzniklou kolizi a uložíme si informaci o kolizi, abychom během *Update* mohli vyvolat *OnCollision*. Pokud existuje, tak pouze zvýšíme číslo značící počet tvarů v kolizi s daným aktérem. Separace funguje obdobně. Snížíme číslo značící počet kolizí s daným aktérem. Pokud už není v kolizi s žádným tvarem, tak záznam smažeme, informuje rodiče o separaci a uložíme si informaci o separaci, abychom během *Update* mohli vyvolat *OnSeparation*.

Pojmenované proměnné jsou uložené v *variables* a definované událost v *events*. Jejich hodnoty se získají pomocí metod *GetVariable* a *GetEvent*.

State (viz obrázek 3.2) představuje stav konečného automatu aktéra. V proměnné *transitions* jsou uloženy hrany z tohoto stavu. Z editoru víme o automatu veškeré informace, takže si nemusíme uchovávat celý automat, ale pouze jeho aktivní stav, za předpokladu že jsou stavy správně propojené. Proto má aktér

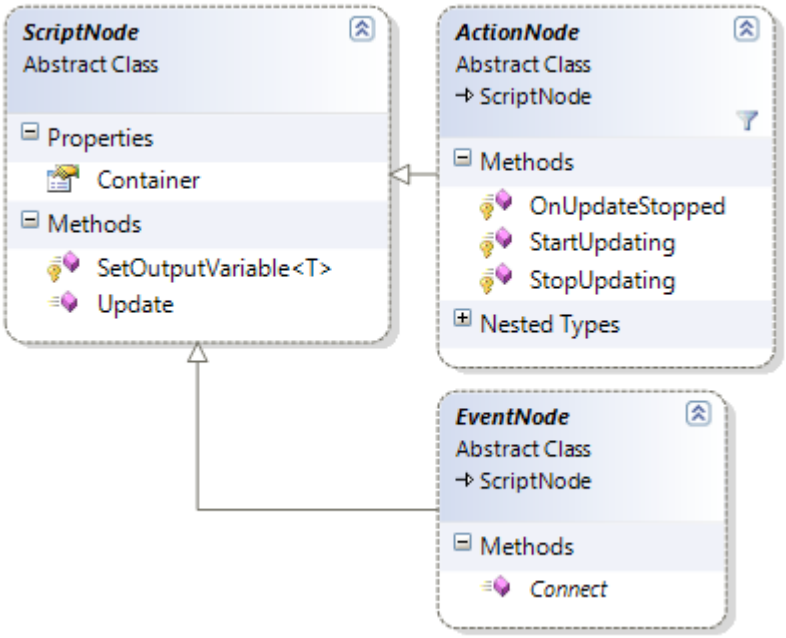


Obrázek 3.2: Objekty na scéně, aktér a jeho stav

v proměnné *stateMachines* uloženy aktivní stavy svých konečných automatů. Metodou *ChangeState* přejdeme do nového stavu po dané hraně. *OnStart* a *OnEnd* jsou vyvolány při aktivaci nebo deaktivaci stavu. *OnUpdate* je vyvolán z metody *Update* každý aktualizační cyklus. Ve stavu jsou samotné uzly pro skriptování. Opět vše známe z editoru, takže není potřeba si ukládat seznam všech použitých uzlů.

3.2.4 Vizuální skriptování

Uzel pro skriptování je definován třídou ScriptNode (viz obrázek 3.3). *Container* obsahuje stav aktéra, ve kterém je uzel použit.



Obrázek 3.3: Uzly ve skriptování

Uzel se připojí přímo na události, které ho zajímají. Tzn. pokud bude uzel chtít být prováděn, když v aktérovi nastane kolize, tak se připojí na *OnCollision* objektu Actor. Zároveň musí platit, že bude uzel prováděn pouze tehdy, když je aktivní stav, ve kterém se nachází. Jedno možné řešení by bylo, před vykonáním akce zkontrolovat, zda-li může být akce vykonána. Tzn. v každé akci uzlu by musela být kontrola nebo by se uzly nepřipojovali přímo, ale někdo by se staral o jejich správné provedení. Tedy uzly by byli od sebe oddělené, protože by se nepřipojovali mezi sebou. Tato varianta umožňuje zapouzdření jednotlivých uzlů, protože uzel nemůže ovlivnit ostatní uzly. Pro rychlejší vykonávání byl zvolen jiný způsob. Uzel se může připojit přímo na událost, pokud to nevadí, jinak použije prostředníka, který ho automaticky připojí a odpojí podle potřeby. Třída StateSubscription (viz obrázek 3.2) představuje tohoto prostředníka. Třída obsahuje 2 metody pro implementaci *OnSubscribe* a *OnUnsubscribe*. StateSubscription se přidá do stavu pomocí metody *AddSubscription*. State se sám stará o tyto objekty. Při změně stavu se na všech objektech typu StateSubscription uložených v *subscriptions* zavolá *Unsubscribe* a naopak při aktivaci stavu *Subscribe*.

ActionNode (viz obrázek 3.3) představuje akci. Definuje metody pro jednoduchou aktualizaci uzlu. Většina uzlů vykoná akci a skončí, ale některé uzly potřebují pracovat delší časový úsek. Například akce *Delay* aktivuje své výstupy poté, co uběhne daná doba. Pro aktualizaci uzel použije metodu *StartUpdating*. Poté bude automaticky volána metoda *Update*. *StopUpdating* slouží pro zastavení aktualizace. Po zastavení se zavolá metoda *OnUpdateStopped*. Metoda bude zavolána i pří zastavení aktualizace kvůli deaktivaci stavu.

EventNode (viz obrázek 3.3) představuje událost. Definuje jednu metodu *Connect* pro implementaci. Při inicializaci je metoda zavolána, aby se událost připojila, kam potřebuje.

Obálkou pro uložení proměnné a události jsou třídy Variable<T> pro ucho-

vání hodnoty daného typu a EventWrapper.

Proměnné ve skriptování mohou být pole. Vstupní proměnná může tedy být typu Variable<T> nebo Variable<T>[]. Výstupní proměnná je vždy pole.

Pro definici uzlu jsou použity následující atributy (viz příklad 3.1):

- FriendlyName - Jméno uzlu, proměnné nebo vstupu/výstupu. Bez jeho specifikace se použije název ze zdrojového kódu.
- Description - Popis uzlu, proměnné nebo vstupu/výstupu.
- Category - Definuje kategorii pro uzel.
- VariableSocket - Popisuje proměnnou použitelnou ve skriptování.
  - *Type* - Vstupní nebo výstupní proměnná.
  - *Visible* - Implicitní viditelnost v editoru pro vizuální skriptování.
  - *CanBeEmpty* - Proměnná může být prázdná (mít hodnotu *null*). Má smysl pro referenční typy a pole. Pokud proměnná může být prázdná, tak v editoru pro vizuální skriptování se mohou proměnné pouze napojovat. Není možnost nastavit proměnou přímo v uzlu.
- DefaultValue - Implicitní hodnota proměnné. Má 2 potomky DefaultValueVector2 pro nastavení hodnoty vektoru a DefaultValueActorOwnerInstance pro nastavení proměnné typu *Actor* hodnotou *Owner*.

Vstup uzlu pro použití ve skriptování a viditelný ve vizuálním editoru je veřejná metoda bez parametrů s návratovou hodnotou *void* (odpovídá delegátu ScriptSocketHandler). Výjimkou jsou 2 metody *Update* a *Connect* v EventNode, které se automaticky nezobrazují. Výstup uzlu představuje veřejný delegát ScriptSocketHandler. Použitelnou proměnnou se berou veřejné proměnné popsané dříve. Použitelný uzel je veřejný potomek ActionNode nebo EventNode. Příklad definice uzlu je zobrazen na 3.1.

3.2.5 Herní obsah

Textury a zvuky jsou načítány pomocí XNA mechanismu Content Pipeline.

TextureData a AnimationData (viz obrázek 3.4) obsahují informace vytvořené v editoru. Obě třídy implementují rozhraní IGraphicsAsset, které definuje metodu *CreateActorRenderer* pro vytvoření objektu ActorRender pro zobrazení aktéra pomocí této grafiky.

3.2.6 Inicializace scény

Inicializace scény, jak ji používají scény vygenerované pomocí editoru (potomci SceneScreen).

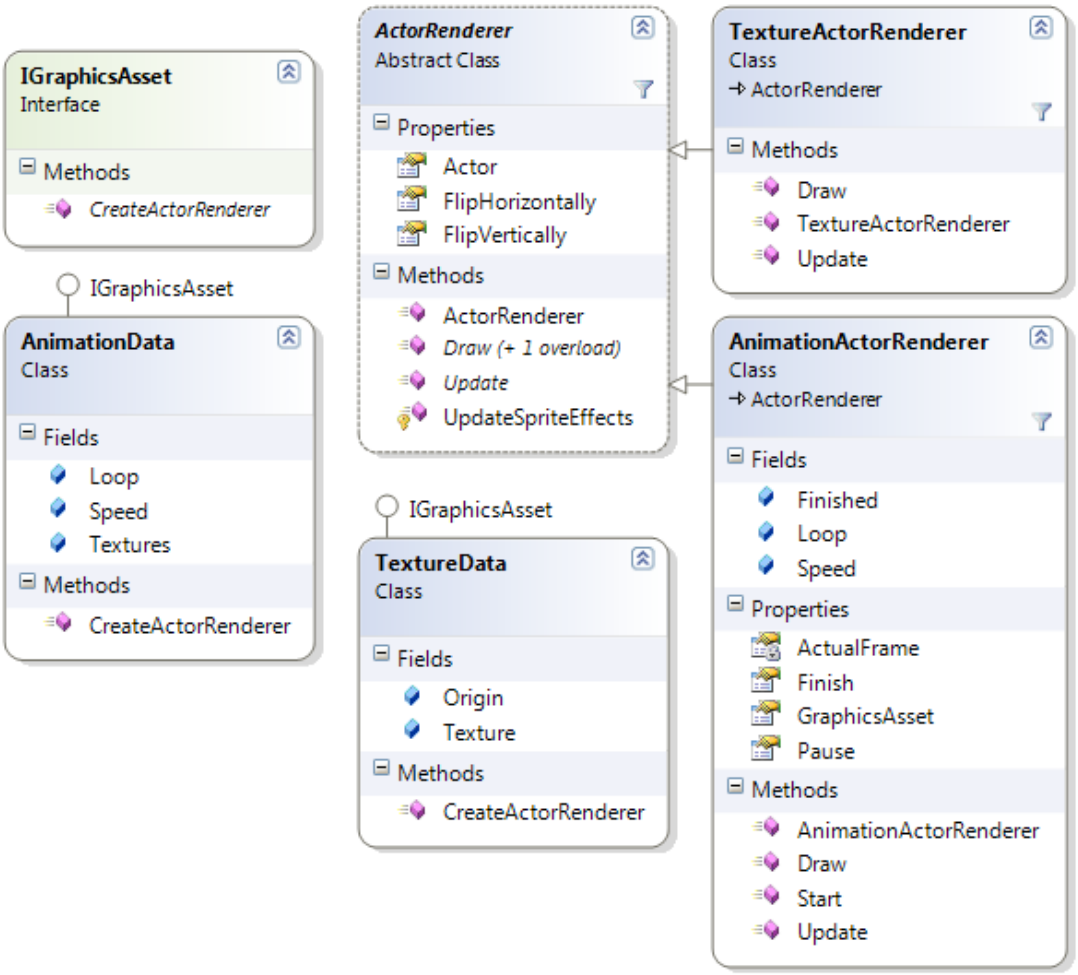
Metoda *Initialize* vytvoří fyzikální simulaci, kterou představuje třída World z Farseer Physics Engine, a připraví kameru pro scénu.

Metoda *LoadContent*:

```
[FriendlyName("My Super Action")]
[Description("Description of my action node.")]
[Category("Actions/My Category")]
public class MyAction : ActionNode
{
  [Description("Fires when the action is completed.")]
  public ScriptSocketHandler Out;

  [Description("Values to ...")]
  [VariableSocket(VariableSocketType.In)]
  public Variable<int>[] Values;

  [VariableSocket(VariableSocketType.In, Visible = false)]
  [DefaultValue(0.9f)]
  public Variable<float> Coefficient;

  [VariableSocket(VariableSocketType.In, CanBeEmpty = true)]
  [DefaultValueVector2(1f, 1f)]
  public Variable<Vector2>[] Vectors;

  [VariableSocket(VariableSocketType.In)]
  [DefaultValueActorOwnerInstance]
  public Variable<Actor>[] Instance;

  [VariableSocket(VariableSocketType.Out)]
  public Variable<int>[] Result;

  [Description("Activates the action.")]
  public void In()
  {
    ...
    if (Out != null) Out();
  }
}
```

Příklad 3.1: Použití atributů pro uzel ve skriptování

1. Nastavení proměnných z editoru: ContinuousPhysics, simulační jednotky, gravitace a barva pozadí.
2. Vytvoření cest a jejich uložení v *Paths*.
3. Vytvoření aktérů a jejich uložení v *Nodes* a *Actors*.
4. Zavolána metoda *InitializeActors* objektu SceneScreen (předek):
   (a) Na každém aktérovi zavolána metoda *InitializeActor*:
      i. Nastavení základních proměnných aktéra jako je vrstva, pozice, grafika, apod.
      ii. Vytvoření kolizních tvarů a následné vytvoření těla pro fyzikální simulaci. Nastavení fyzikálních proměnných těla.



Obrázek 3.4: Textura, Animace a ActorRenderer.

iii. Vytvoření pojmenovaných proměnných do *variables*.
iv. Vytvoření definovaných událostí do *events*.
v. Vytvoření dětí do *Children*.
vi. Zavolána metoda *InitializeActor* na objektu Actor (předek):
   A. Aktéra nastavíme pro příjem kolizí z fyzikální simulace.
   B. Inicializujeme děti metodou *InitializeActor*.

(b) Do *Actors* přidáme i všechny děti aktérů, abychom je mohli jednoduše hledat.

(c) Na každém aktérovi zavolána metoda *InitializeComponents*:
   i. Vytvoření všech uzlů pro skriptování.
   ii. Vytvoření konečných automatů. Nastavení a propojení jejich stavů.
   iii. Nastavení a propojení jednotlivých uzlů pro skriptování.
   iv. Zavolána metoda *InitializeComponents* na Actor (předek):
      A. Přípojení těl dětí do předka. Aktuálně má dítě vlastní tělo. Z jeho těla vezmeme kolizní tvary, upravíme je a přidáme do aktuálního těla. Dítěti nastavíme tělo stejné jako tělo předka.

B. Na dětech zavoláme metodu *InitializeComponents*.

(d) Na každém aktérovi zavolána metoda *InitializeStateMachines*:

i. Aktivujeme stavy, které mají být aktivní.

ii. Na dětech zavoláme metodu *InitializeStateMachines*.

3.2.7 Cyklus scény

Aktualizace scény (metoda *Update*) :

1. Krok fyzikální simulace.
   Vzniklé kolize a separace jsou oznámeny aktérům. Aktér si informace o kolizích a separacích uloží pro pozdější zpracování. Kolize a separace nemohou být okamžitě zpracovány, protože akce ve skriptování reagující na ně mohou ovlivnit aktuální krok fyzikální simulace.

2. Aktualizace kamery.

3. Aktualizace aktivních objektů na scéně. Neaktivní jsou vymazány.
   Aktualizace aktéra (metoda *Update*):

   (a) Aktualizace pozice aktéra z fyzikální simulace.

   (b) Zpracování kolizí a separací z fyzikální simulace. Vyvolány *OnCollision* a *OnSeparation*.

   (c) Aktualizace aktivních stavů konečných automatů. Stav vyvolá *OnUpdate*.

   (d) Aktualizace *ActorRenderer* představující grafiku aktéra.

   (e) Aktualizace dětí. Na dětech zavoláme metodu *Update*.

Kreslení scény (metoda *Draw*):

1. Vyplnění obrazovky barvou pozadí.

2. Kreslení aktivních objektů na scéně.
   Kreslení aktéra (metoda *Draw*):

   (a) Kreslení aktéra pomocí *ActorRenderer*.

   (b) Kreslení dětí. Na dětech zavoláme metodu *Draw*.

3.3 Editor

Editor je implementován pomocí WinForms.

Jmenné prostory

Jmenný prostor *PlatformGameCreator.Editor* zastřešuje vše týkající se editoru. Zde se nachází projekt, formulář představující editor a třídy pro bezpečné vymazání objektu z projektu. Ostatní třídy jsou rozděleny do jmenných podprostorů, podle svého určení:

PlatformGameCreator.Editor.Assets
: Nachází se zde bázová třída pro herní obsah. Konkretní implementace jsou v samostatném jmenném podprostoru. Animace je v *Assets.Animations*, zvuk v *Assets.Sounds* a textura v *Assets.Textures*. Každý obsah má správce, editor a ovládací prvek pro zobrazení obsahu správce.

PlatformGameCreator.Editor.Building
: Generování zdrojového kódu hry definovaném projektem. Překlad zdrojového kódu a spuštění hry z editoru.

PlatformGameCreator.Editor.Common
: Obecné třídy a rozhraní pro použití v celém editoru. Bázová třída příkazů, historie příkazů, kolekce informující o své změně a kolizní tvary.

PlatformGameCreator.Editor.GameObjects
: Nachází se zde bázová třída pro objekt použitelný ve hře. Konkrétní implementace jsou v samostatném jmenném podprostoru. Aktér je v *GameObjects.Actors* a cesta v *GameObjects.Paths*. Každý objekt definuje zobrazení na scéně a ovládací prvek pro zobrazení svého nastavení.

PlatformGameCreator.Editor.Scenes
: Třídy týkající se scény. Správce scény, scéna a vrstva. Ovládací prvek pro editaci scény a pro zobrazení stromu scény.

PlatformGameCreator.Editor.Scripting
: Zastřešuje vizuální skriptování. Uzly pro skriptování, konečný automat, apod. Formulář pro editaci vizuálního skriptování.

PlatformGameCreator.Editor.WinForms
: WinForms ovládací prvky pro použití v celém editoru.

PlatformGameCreator.Editor.Xna
: Třídy související s XNA Framework. Konverze obsahu do XNA formátu a XNA ovládací prvek ve WinForms.

Obecné GUI koncepty

Třídy obsahující data, která se zobrazují v editoru, definují události pro upozornění na změnu dat. Ovládací prvky použité ve WinForms nebo zobrazitelné objekty se připojí na jejich události. Kvůli tomu tyto třídy implementují rozhraní IDisposable, aby se mohli v metodě *Dispose* odpojit od událostí, které sledují. Samotná implementace rozhraní nestačí, ale musíme se sami postarat o správné volání metody *Dispose*, jakmile už objekt nepotřebujeme. Pokud bychom vše nechali na garbage collectoru, tak se nám nikdy žádný objekt z paměti neodstraní,

protože objekty jsou připojené na události dat, které jsme nesmazali, tedy stále na ně existuje odkaz a tudíž nebudou odstraněni. Pro upozornění na změnu kolekce je použita třída ObservableList. Obsahuje událost *ListChanged*, která je vyvolána při změně kolekce, když je prvek přidán, smazán, změněn nebo je celá kolekce smazána. Událost nese informace, o jakou akci se jedná, indexy a prvek definován konkrétní akcí. Potomek ObservableIndexedList navíc každému prvku nastavuje číslo jeho indexu. Prvek okamžitě zná svoje pořadí v kolekci.

Třídy končící názvem *View* představují zobrazení. Např. TexturesView zobrazuje textury, SceneTreeView strom scény, VariableView proměnnou ve skriptování, apod. Data a jejich zobrazení se nacházejí ve stejném jmenném prostoru.

Pro zobrazení zprávy uživateli z různých částí kódu je použita statická třída Messages. Lze zobrazit informaci, varování nebo chybu. Objekt implementující rozhraní IMessagesManager, který zajišťuje konkrétní zobrazení zprávy, nastavíme do *MessagesManager*. Informující objekt neví nic o tom, jak bude zpráva zobrazena, to závisí na konkrétní implementaci. Formuláře nejčastěji používají DefaultMessagesManager. Informaci a varování zobrazí ve stavové lište, akorát varování bude mít zvýrazněné pozadí. Chyba je zobrazena pomocí MessagesBox.

Bázová třída Command představuje příkaz nebo akci (návrhový vzor Command [5]). Obsahuje dvě metody pro implementaci - *Do* pro vykonání akce a *Undo* pro vrácení (zrušení) akce. Příkazy lze ukládat ve třídě History představující historii. Pomocí metod *Undo* a *Redo* se lze pohybovat v historii.

3.3.1 Projekt

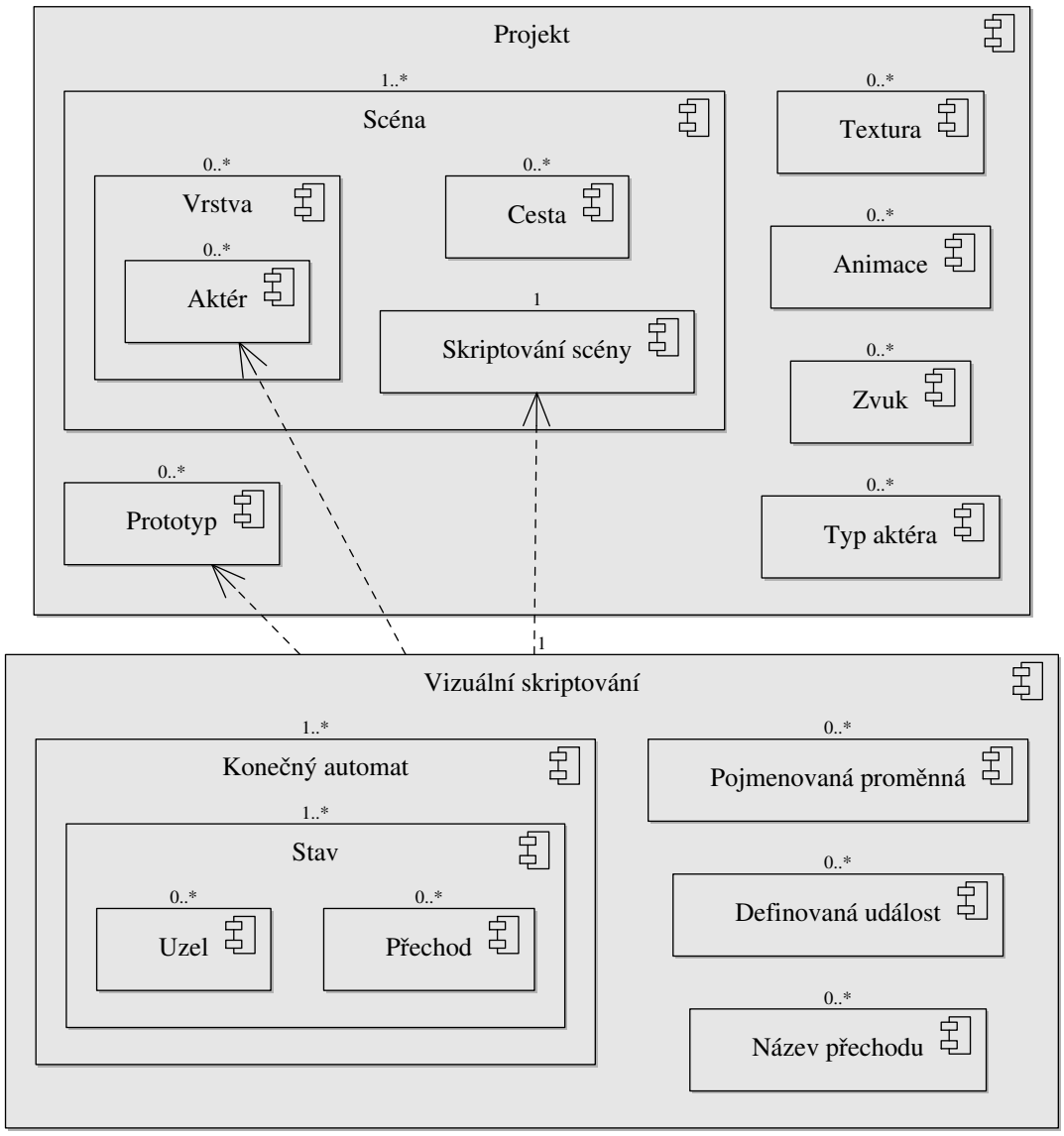
Třída Project představuje projekt obsahující veškeré informace (viz obrázek 3.5). V editoru je vždy otevřen právě jeden projekt, proto je Project singleton.

Uložení a načtení projektu lze vyřešit různými způsoby. První varianta je pomocí serializace, která podporuje cykly, kdy každá třída, která se potřebuje uložit, musí podporovat serializaci. Nebo vlastní řešení, kdy budeme generovat data do XML nebo vlastního formátu. Musíme vyřešit cykly. Tato varianta v podstatě implementuje existující řešení. Proto byla zvolena binární serializace.

3.3.2 Herní obsah

Obsah pro hru je před prvním použitím nutné zkonvertovat do formátu, který používá XNA mechanismus Content Pipeline. Knihovny pro konverzi nejsou distribuovány spolu s XNA Framework Redistributable, ale pouze s XNA Game Studio určeným pro vývojáře. Třída ContentBuilder konvertuje obsah za běhu. Pokud nenajde potřebné knihovny, tak není konverze možná.

Asset je bázová třída pro herní obsah. Každá instance má unikátní číslo *Id* v celém projektu, které slouží k vyhledávání. Má potomka DrawableAsset představujícího grafický obsah. Obsahuje metodu *CreateView* k implementaci, která vrací objekt typu ISceneDrawable, který slouží k vykreslení grafiky na scénu. Dále obsahuje seznam kolizních tvarů. Kolizní tvar je reprezentován bázovou třídou Shape. Obsahuje ohraničující obdélník a metody pro posun, rotaci a změnu velikosti (scale) kolizního tvaru. Potomci jsou konkrétní kolizní tvary - polygon,



Obrázek 3.5: Hierarchie projektu

kruh a hrana.

Textura je před přidáním do projektu zkonvertována do formátu pro XNA. Zvuk není ihned zkonvertován, protože může reprezentovat dva různé typy proměnných (*Sound* a *Song*). Zvuk se zkonvertuje, jakmile bude vyžadován ve hře. Proto třída Sound obsahuje *IsSoundEffect*, *IsSong*, *CompiledAsSoundEffect* a *CompiledAsSong*.

Ovládací prvek ShapesEditingScreen představuje plochu pro editaci kolizních tvarů. *State* představuje stav plochy reprezentován třídou ShapesEditingState. Obsahuje metody reagující na práci s myší a klávesnicí, které jsou volány po provedení metod stejného názvu v ShapesEditingScreen. Její potomek GlobalScreenState definuje chování, které by většina stavů měla mít. Pohyb po ploše a přiblížení/oddálení plochy. Konečné stavy ho dědí. Jeho potomek ShapeState je bázová třída pro editaci kolizního tvaru. Kolizní tvary jsou v ShapesEditingScreen uloženy v *Shapes* už jako stavy pro jejich přímou editaci. Z něj dědí konkrétní kolizní tvary CircleEditState, EdgeEditState a

PolygonEditState. GlobalScreenState má ještě potomka ChangeOriginState pro nastavení bodu Origin u textury. Ovládací prvek pro nastavení plochy pro editaci kolizních tvarů představuje BasicControllerForShapesEditing. Zobrazuje seznam kolizních tvarů na ploše a umožnuje jejich vytváření, mazání a přepínání. Obsahuje nastavení pro plochu jako je zobrazení mřížky, konvexního obalu, apod. Jednotlivé typy obsahu (textura a animace) rozšiřují tyto třídy a používají ve svém editoru.

Každý obsah má správce, ovládací prvek pro zobrazení obsahu správce a formulář představující editor. Tedy u textury dané Texture to je TexturesManager, TexturesView a TexturesForm. Třídy dodržují stejné pojmenování, takže ostatní obsah není třeba vyjmenovávat.

3.3.3 Scéna

Třídy týkající se scény jsou sdruženy ve jmenném prostoru *PlatformGameCreator.Editor.Scenes*.

Třída Scene představuje scénu hry. Scéna obsahuje vrstvy, kde jsou uloženy aktéři, aktuálně vybranou vrstvu, cesty a skriptování scény. Scény jsou uloženy v ScenesManager, kde se nachází vybraná scéna, která je zobrazená v editoru. Třída Layer představuje vrstvu pro scénu, kde jsou uloženi jednotliví aktéři.

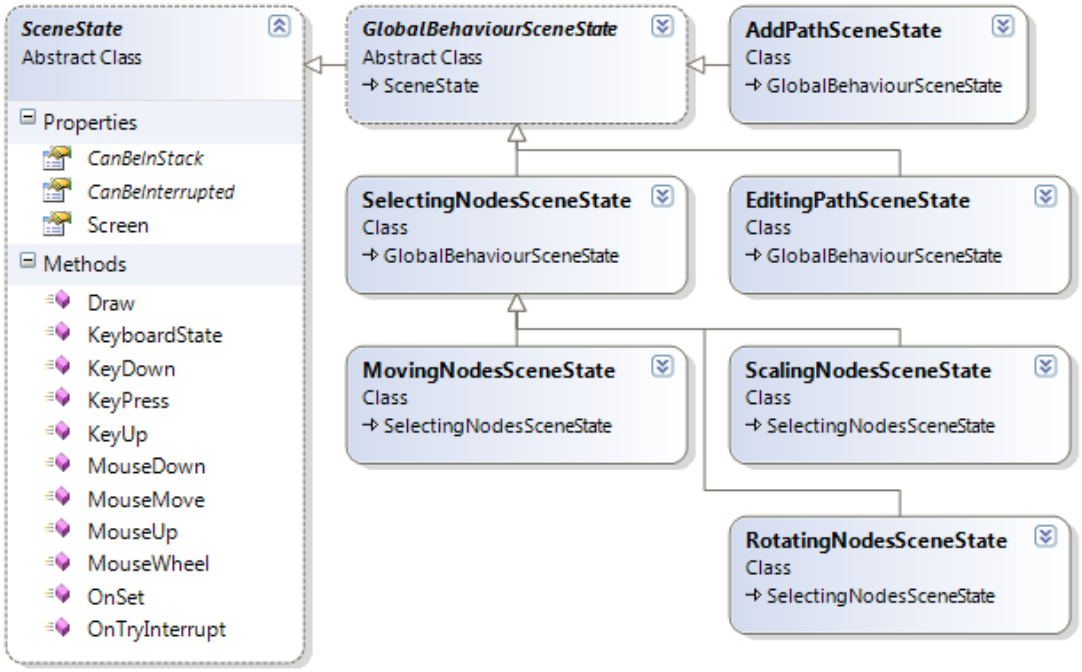
Musíme se rozhodnout, jakou technologií se bude scéna zobrazovat. První varianta je pomocí WinForms a vykreslování přes GDI+. Scéna se vykreslí pouze tehdy, když je potřeba. Druhá varianta je pomocí XNA Framework, které je použito pro hru. Scéna se automaticky překresluje, proto není nutné scénu při změně zneplatnit (invalidovat). Tudíž je snadné kreslení animací, ale naopak tato varianta více zatěžuje procesor. Má větší význam do budoucna. Možnost zobrazení efektu nebo testování a editace hry přímo v editoru. Zvolena byla druhá možnost z důvodu možnosti rozšíření do budoucna.

Ovládací prvek pro zobrazení scény reprezentuje třída SceneScreen, která dědí od GraphicsDeviceControl. Ta umožňuje použití XNA ve WinForms. Scéna je aktualizována a vykreslena, když je formulář se scénou aktivní a nemá nic jiného na práci. Obsahuje všechny objekty na scéně, vybrané a označené objekty. Má historii a schránku ke kopírování objektů.

SceneScreen a SceneNode automaticky spravují objekty na scéně pomocí jejich událostí. Tzn. při přidání aktéra do vrstvy se v reakci na změnu kolekce vrstvy vytvoří SceneNode reprezentující aktéra, apod.

State představuje stav scény reprezentován třídou SceneState (viz obrázek 3.6). Obsahuje metody reagující na práci s myší a klávesnicí, které jsou volány po provedení metod stejného názvu v SceneScreen. *CanBeInterrupted* značí, jestli může být stav přerušen a změněn. Pokud se pokusíme změnit stav, když to nejde, tak se na stavu zavolá metoda *OnTryInterrupt*, která typicky poskytuje informaci uživateli. *CanBeInStack* značí, jestli může být stav uložen v jiném stavu jako předchozí stav, na který je možné se vrátit. Její potomek GlobalBehaviourSceneState (viz obrázek 3.6) definuje chování, které by většina stavů měla mít. Pohyb po scéně, přiblížení/oddálení scény a kopírovaní/vložení objektů přes schránku. Konečné stavy ho dědí.

Ovládací prvek ScenesView zobrazuje seznam scén projektu a SceneTreeView zobrazuje upravitelný strom scény, kde jsou viditelné všechny cesty, vrstvy



Obrázek 3.6: Stavy scény pro SceneScreen

a aktéři.

3.3.4 Objekty ve hře

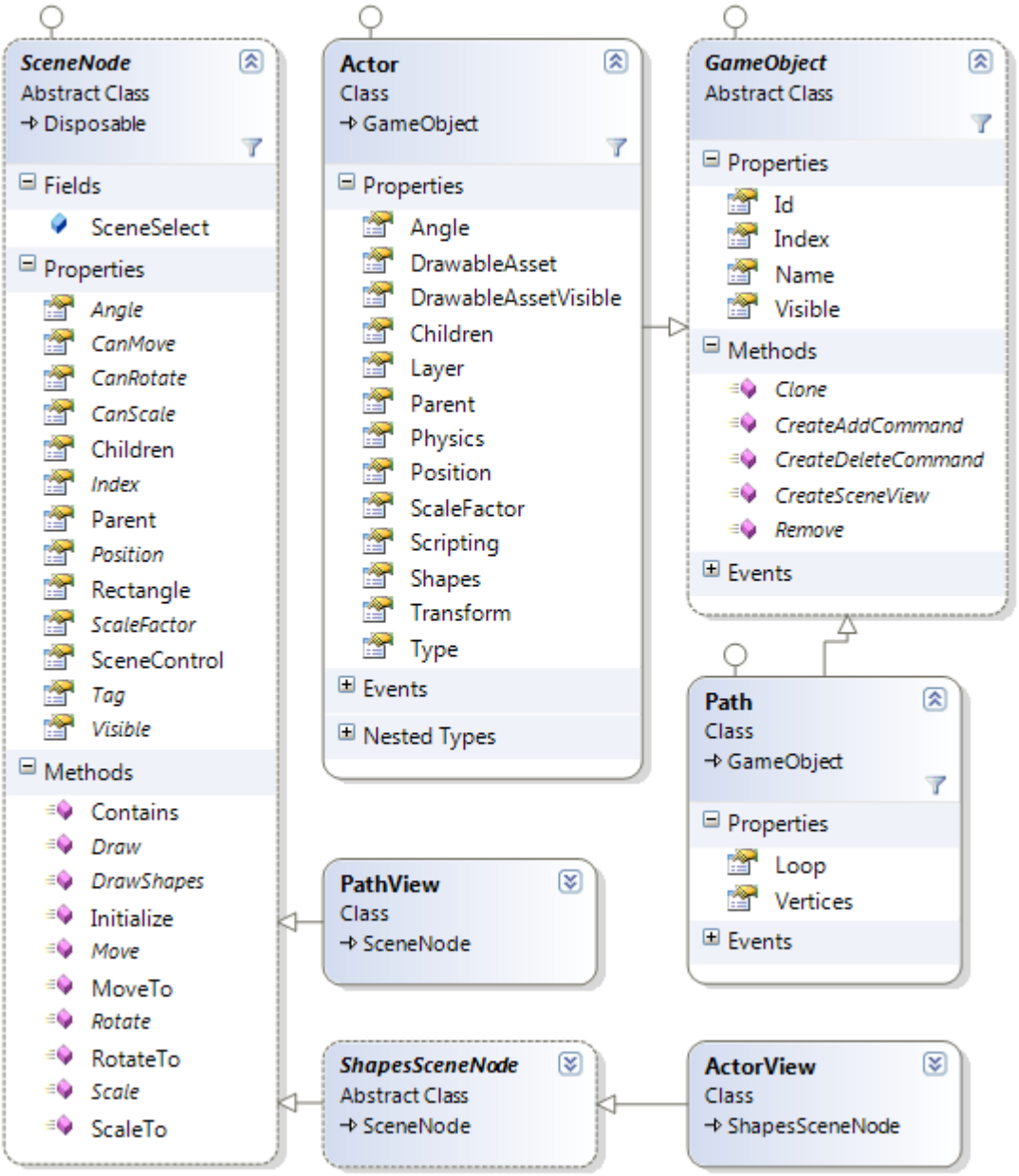
Objekty pro použití ve hře reprezentuje bázová třída GameObject (viz obrázek 3.7). Každá instance má unikátní číslo *Id* v celém projektu, které slouží k vyhledávání. Metoda *CreateSceneView* vytvoří zobrazení pro scénu (objekt typu SceneNode). Má dva potomky Actor a Path.

Každý typ objektu pro použití ve hře definuje zobrazení na scéně a ovládací prvek pro zobrazení svého nastavení. Tedy u aktéra daného Actor to je ActorView a ActorInfo. Třídy dodržují stejné pojmenování, takže u ostatních není potřeba vyjmenovávat.

Objekty pro zobrazení na scéně jsou reprezentovány pomocí SceneNode (viz obrázek 3.7). Jsou uloženy v *Nodes* ve SceneScreen a jsou setříděny podle svého *Index* pro okamžité vykreslování podobně jako v herním engine. *Index* je definován podle čísla vrstvy a objektu. Obsahují data a metody pro jejich základní editaci na scéně. Tzn. posun, otočení a změnu velikosti. *SceneSelect* značí stav objektu na scéně, jestli je objekt vybrán nebo aktuálně označen.

Třída CloningHelper umožňuje klonování objektů popsané v 2.13. Skupina objektů se naklonuje. Poté se zkontroluje, jestli neobsahují objekty, které se kopírovali. Pokud ano, tak se jejich hodnoty nastaví na nové hodnoty.

Třída ActorType představuje typ aktéra. Obsahuje hodnotu podle pravidla v 1.3. Jsou uloženy ve stromu a jejich kořen je *Root* v ActorTypesManager.



Obrázek 3.7: Objekty na scéně

3.3.5 Vizuální skriptování

Třídy týkající se vizuálního skriptování jsou sdruženy ve jmenném prostoru *PlatformGameCreator.Editor.Scripting*.

ScriptingComponent reprezentuje kompletní vizuální skriptování. Obsahuje konečné automaty, pojmenované proměnné, apod. Také obsahuje aktéra nebo scénu, kde je skriptování použito. ScriptingForm je formulář pro editaci objektu ScriptingComponent.

Proměnná a událost

Rozhraní IVariable představuje obálku pro proměnnou. Jediná implementace je Var<T>, od které dědí jednotlivé typy proměnných jako BoolVar, Actor-

Var, TextureVar, apod. Ve třídách se používá pouze obálka, jinak by objekty používající proměnnou museli být generické a to by zkomplikovalo práci při určování typu uzlů a jejich zpracování.

Pojmenovaná proměnná je reprezentovaná třídou NamedVariable. Obsahuje svůj název a proměnnou daného typu. Ovládací prvek NamedVariablesView zobrazuje seznam pojmenovaných proměnných a umožňuje jejich editaci, přidávání a smazání.

Třída Event představuje název přechodu (event in) nebo definovanou událost (event out). Obsahuje pouze svůj název. Ovládací prvek EventsView zobrazuje seznam událostí a umožňuje jejich editaci, přidávání a smazání.

Editace nastavení

Ovládací prvek BaseSettingsView představuje editaci nastavení ve formě tabulky. Využívá DataGridView z WinForms, do kterého se uloží jednotlivé řádky nastavení. Chceme, aby buňka byla informována, když se změní její hodnota. DataGridViewCell neobsahuje metodu, kterou by automaticky vyvolala při změně své hodnoty, proto naše buňky implementují rozhraní ICellValueChanged, které takovou metodu definuje. BaseSettingsView při změně hodnoty buňky automaticky zavolá její metodu *CellValueChanged*, pokud implementuje toto rozhraní.

Proměnná typu IVariable definuje metodu *GetGridCell*, která vrací buňku pro editaci hodnoty dané proměnné. Buňky proměnných jsou přizpůsobeny na změnu ze všech možných stran. Tzn. při změně hodnoty buňky se nová hodnota nastaví do proměnné, což může mít za následek změnu zobrazení hodnoty na virtuální ploše uzlů. Naopak při změně hodnoty proměnné buňka automaticky převezme novou hodnotu proměnné. Pokud buňka umožňuje výběr z daného seznamu (např. textur, typu aktérů, apod.), tak při změně toho seznamu se automaticky upraví i seznam v buňce, apod.

Spojování uzlů

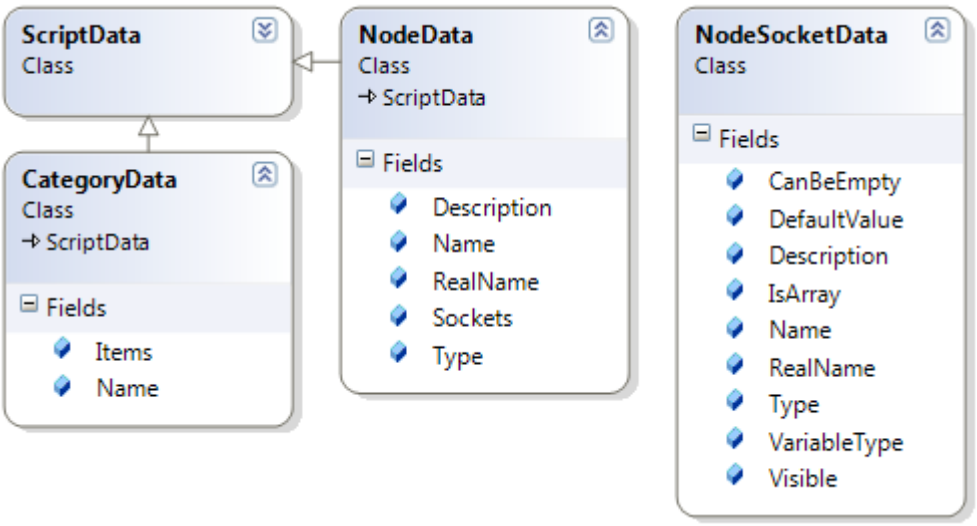
Uzly pro skriptování jsou rozděleny na tři vrstvy. První vrstva představuje data uzlů získaných z herního engine. Následující vrstva používá tato data a představuje jejich instance. Poslední vrstva je zobrazení uzlů na virtuální ploše pro editaci.

Třída ScriptingNodes získá pomocí reflexe validní uzly pro skriptování z assembly patřící hernímu engine. Typy proměnných jsou získány podle tabulky 3.1. Získané informace jsou uloženy v odpovídajících objektech NodeData a NodeSocketData (viz obrázek 3.8). Automaticky je vytvořena hierarchie kategorií. Uzly a kategorie jsou setříděny podle abecedy. Reflexe je použita pouze jednou, poté jsou uzlu uloženy v *Root*, který představuje kořen všech uzlů. Data nejsou ukládána a jsou znovu získány při každém běhu programu.

BaseNode (viz obrázek 3.9) je bázová třída pro instance uzlů. Obsahuje pozici na virtuální ploše a metodu *CreateView* pro vytvoření zobrazení konkrétního uzlu (objekt typu SceneNode). Node představuje akci nebo událost. Obsahuje odkaz na NodeData, který reprezentuje. Vstupy a výstupy uzlu jsou reprezentovány bázovou třídou NodeSocket, která obsahuje odkaz na NodeSocketData, který reprezentuje. Má dva potomky - SignalNodeSocket pro signál a

Typ proměnné	Typ v herním engine	Typ v editoru
Bool	bool	bool
Int	int	int
Float	float	float
Vector2	Vector2 z XNA	Vector2 z XNA
String	string	string
Actor	Scenes.Actor	GameObjects.Actors.Actor
Actor Type	uint	GameObjects.Actors.ActorType
Path	Scenes.Path	GameObjects.Paths.Path
Texture	Assets.TextureData	Assets.Textures.Texture
Animation	Assets.AnimationData	Assets.Animations.Animation
Sound	SoundEffect z XNA	Assets.Sounds.Sound
Song	Song z XNA	Assets.Sounds.Sound
Scene	Screens.Screen	Scenes.Scene
Key	Keys z XNA	Keys z XNA

Tabulka 3.1: Tabulka typu proměnných v herním engine a editoru

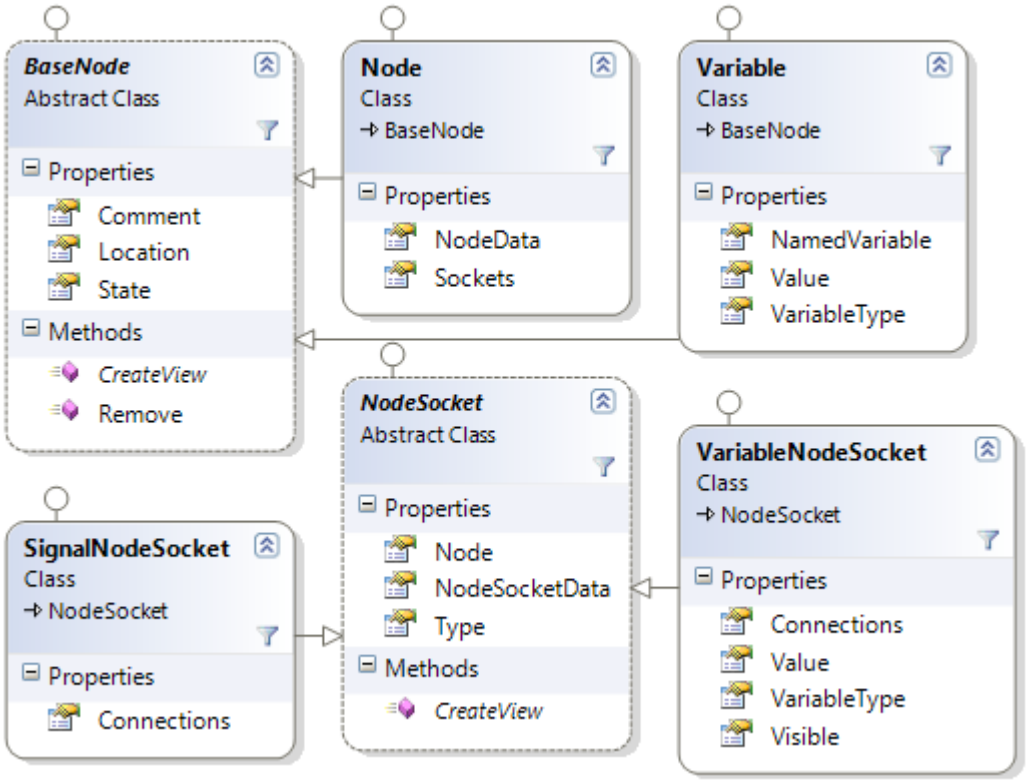


Obrázek 3.8: Reprezentace dat uzlů pro skriptování

VariableNodeSocket pro proměnou. Variable představuje proměnnou daného typu. Může pouze odkazovat na pojmenovanou proměnnou, pokud je *NamedVariable* nastaveno.

Ovládací prvek pro virtuální plochu pro editaci uzlů reprezentuje třída ScriptingScreen. Obsahuje všechny objekty na virtuální ploše, vybrané a označené objekty. Má historii a schránku ke kopírování objektů.

Zobrazení uzlu na scéně je dané bázovou třídou SceneNode (viz obrázek 3.10). Uzly jsou uloženy v *Nodes* v ScriptingScreen. *CanConnect* označuje, jestli může být objekt spojen s jiným uzlem. Pokud ano, tak *IConnecting* obsahuje implementaci rozhraní IConnection, které umožňuje spojení dvou uzlů. *CanEditSettings* označuje, jestli má uzel nějaké editovatelné nastavení. Pokud ano, tak *IEditSettings* obsahuje implementaci rozhraní IEditSettings pro zobrazení nastavení do ovládacího prvku typy NodeSettingsView, který dě-



Obrázek 3.9: Uzly pro skriptování

dí od BaseSettingsView. Jeho potomci jsou NodeView, VariableView, ConnectionView a NodeSocketView z něj dále SignalInNodeSocket, SignalOutNodeSocket a VariableNodeSocket.

Konečný automat

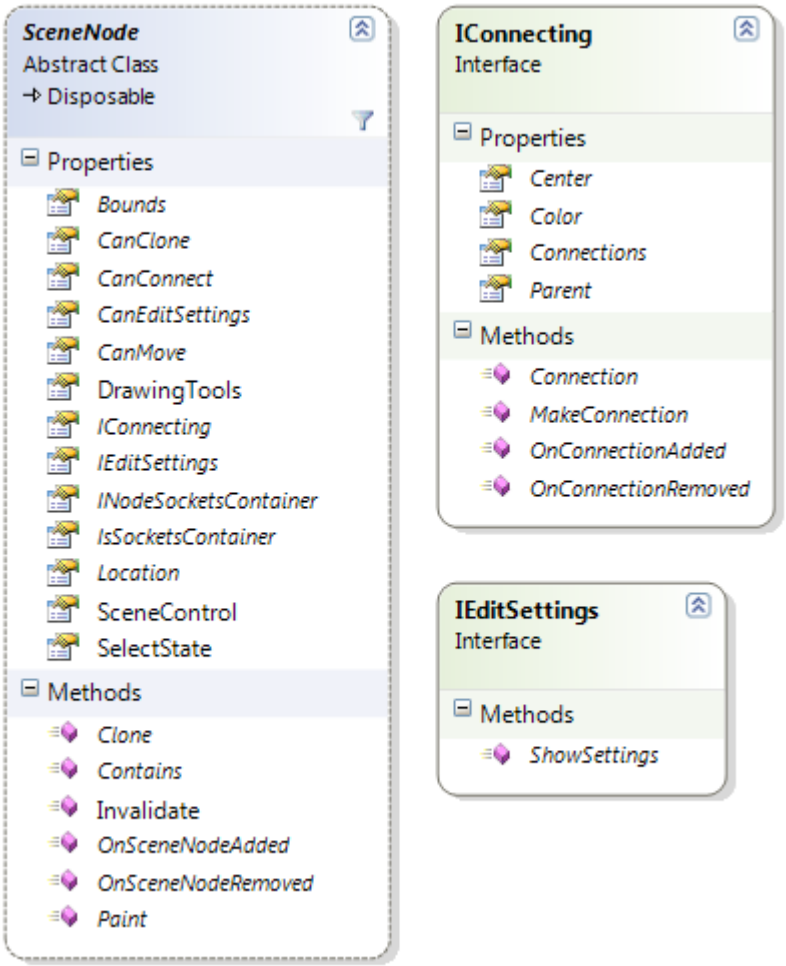
Konečný automat je reprezentován třídou StateMachine. Obsahuje stavy a počáteční stav. Stav reprezentován třídou State obsahuje pozici na virtuální ploše, kde se konečný automat edituje, přechody pomocí třídy Transition a uzly pro skriptování.

Ovládací prvek StateMachineView zobrazuje a edituje konečný automat na virtuální ploše. Zobrazení stavů reprezentované třídou StateView jsou uložené v *States*. Stav obsahuje zobrazení svých přechodů třídou TransitionView v *transitions*. Přechod je vykreslen pomocí Bezierovy křivky.

Ovládací prvek StateMachinesView zobrazuje konečné automaty ve skriptování a StateSettingsView zobrazuje nastavení stavu. Stavu je možné nastavit jméno a přechody - jméno přechodu a cílový stav.

3.3.6 Přeložení hry

Jakým způsobem vytvořit výslednou hru? První možnost je pouze vygenerovat data z editoru do předem daného formátu. Hra prezentovaná herním engine sama data načte a zpracuje. Formát dat může být XML, rozšíření pro XNA Content Pipeline nebo vlastní formát. Všechny formáty spojuje nutnost použití reflexe pro inicializaci správných uzlů pro skriptování. Toto také znamená menší zpomalení



Obrázek 3.10: Zobrazení uzlů na virtuální ploše ve skriptování

při načítání dat. Druhá možnost je přímo vytvořit výslednou hru. Tzn. vygenerovat zdrojový kód představující hru a ten přeložit nebo v C# můžeme také přímo generovat IL kód. Z možnosti rozšíření do budoucna byla vybrána druhá možnost. Kdybychom chtěli hru přepsat do jiného jazyka a technologie (např. C++ s OpenGL), tak by se nehodilo kdyby herní engine vyžadoval technologii, kterou jiné jazyky nemají. Pak stačí pouze přepsat herní engine a generování hry v editoru. Tedy vygenerujeme zdrojový kód představující hru a ten přeložíme.

Třída GameBuild se stará o překlad a následné spuštění hry definované projektem. Hru lze přeložit pro testování v editoru nebo publikaci. Metoda *GenerateSourceCode* generuje zdrojový kód hry do adresáře projektu. Pro generaci kódu jsou použity třídy GameGenerator, SceneGenerator a ActorGenerator. Metoda *BuildGame* přeloží zdrojový kód. Hra se překládá v jiné aplikační doméně. Při překladu pro testování hry se generuje assembly do paměti a bez jiné aplikační domény bychom jí nemohli odstranit z paměti. Metoda *RunGame* spustí hru v aplikační doméně, kde jsme hru přeložili, protože se zde nachází assembly hry.

Generovaný zdrojový kód obsahuje: Třídu GameUsingPlatformGameCreator zděděnou od PhysicsGame, která nastaví rozlišení okna hry, adresář s herním obsahem a do správce obrazovek přidá obrazovku představující aktu-

álně otevřenou scénu. Třídu Program obsahující main metodu, která vytvoří instanci GameUsingPlatformGameCreator a spustí hru. Obrazovku GlobalContentForScene zděděnou od SceneScreen, která inicializuje textury a animace. Každá scéna se nachází ve vlastním jmenném prostoru, kde je obrazovka SceneLevel představující scénu a všichni aktéři použití na scéně. Každý aktér je reprezentován třídou. Třída aktéra neobsahuje svoje děti. Každý potomek je dán vlastní třídou. Instance děti se vytvoří až během inicializace. Díky tomu můžeme bez problémů vytvářet za běhu instance dětí. Aktér a scéna jsou vygenerovány tak, že splňují popsanou inicializaci v 3.2.6. Skriptování scény je vygenerováno jako aktér, který není ve fyzikální simulaci.

Aktér při inicializaci hledá svoje příbuzné pomocí předdefinované cesty (vlastnost z 2.13). Např. pokud potřebuje aktéra, který je potomek jeho rodiče, apod. Cesta je reprezentovaná posloupností čísel. Záporné číslo znamená rodiče aktéra, jinak index dítěte aktéra. Tzn. posloupnost $-1, -1, 2, 3$ znamená cestu přes rodiče, jeho rodiče, dítě na indexu 2 a nakonec dítě na indexu 3. Pokud cesta neexistuje, tak se použije původní aktér ze skriptování. Např. aktér obsahuje dítě, které používá jeho proměnnou. Za běhu vytvoříme nové instance tohoto aktéra. Pak každé dítě bude používat proměnnou svého rodiče. Pokud vytvoříme pouze instanci aktéra, který reprezentuje představené dítě, tak dítě nemůže použít proměnnou svého rodiče, protože žádného nemá. Místo toho použije proměnnou aktéra, který představuje jeho rodiče v editoru (má *ActorId* jeho rodiče v *Actors* na scéně).

3.3.7 Důležité formuláře

Formulář EditorApplicationForm představuje hlavní okno editoru (viz obrázek 2.3). Před jeho samotným zobrazením je ještě zobrazen IntroForm (viz obrázek 2.1) pro výběr projektu. Třída EditorApplication představuje vstupní bod programu pomocí metody *Main*, která vytvoří a spustí aplikaci danou pomocí formuláře EditorApplicationForm.

Bázová třída ItemForDeletion představuje prvek, který má být vymazán. Obsahuje metody pro vyčištění nebo smazání prvku a metodu pro zjištění závislostí na tomto prvku, která vrací objekty typu ItemForDeletion představující závislosti. Z ní dědí konkrétní prvky pro smazání ActorForDeletion, TextureForDeletion, ScriptVariableNodeSocketForDeletion, apod. Pro zjišťování závislostí je použita třída ConsistentDeletionHelper, která obsahuje metody pro nalezení závislostí všech potřebných typů prvků.

Formulář ConsistentDeletionForm představuje grafické zobrazení bezpečného mazání (viz obrázek 2.6). Pomocí konstruktoru předáme formuláři prvky, které chceme smazat, a on se o vše postará sám. Nejsou zobrazeny prvky, které jsou obsaženy v prvcích pro smazání. Např. aktér má dítě, které používá jeho proměnnou. Při smazání aktéra nebude zobrazena závislost, že proměnná aktéra je použita v dítěti, protože dítě bude automaticky smazáno spolu se svým rodičem. Nebo při smazání vrstvy ze scény se nezobrazí závislosti mezí aktéry ve vrstvě, protože budou stejně všichni smazáni, apod. Původní prvky a jejich závislosti jsou smazány. Zkontroluje se, jestli smazání závislostí neobsahuje další závislosti. Pokud ano, tak se nové závislosti zobrazí. Tyto závislosti jsou smazány a opět se zkontroluje, pokud nejsou vyvolány další závislosti. Pokračuje se do té do do-

by, dokud už nejsou žádné další závislosti. Po provedení bezpečného smazání se vyčistí historie a schránka pro kopírování všech formulářů, aby nemohli nastat problémy, kdyby zde byli uloženy prvky, které jsme už z projekty vymazali.

Formulář ProcessingForm umožňuje provádění operace na pozadí a zobrazení informací o průběhu uživateli. Interně používá BackgroundWorker z WinForms, proto třída funguje na stejném principu. Metoda *Process* začne vykonávat operaci danou pomocí dvou předaných delegátů. První delegát představuje operaci, která bude vykonána v jiném vlákně. Druhý delegát bude zavolán po skončení operace ve stejném vlákně jako formulář. Oba delegáti obsahují odkaz na formulář pomocí rozhraní IProcessingContainer, pomocí kterého lze změnit informace zobrazované uživateli. Formulář obsahuje implementaci rozhraní IMessagesManager, pomocí kterého jsou informace zobrazeny v ovládacím prvku, kde je zobrazen výpis operace. Zprávy jsou typicky posílány z jiného vlákna, protože operace probíhá v jiném vlákně. Formulář je použit pro všechny operace jako je přidávání textury do projektu, spuštění a publikace hry, apod.

Závěr

Cílem této práce bylo implementovat aplikaci pro vytváření 2D her založených na simulaci reálné fyziky. Hru lze vytvořit bez znalosti programování. Podařilo se implementovat všechny potřebné vlastnosti pro editor.

Navržené vizuální skriptování používá princip konečných automatů a spojování předdefinovaných uzlů. Objekt ve hře je popsán pomocí konečných automatů. V každém stavu je na virtuální ploše vytvořeno chování pomocí spojení uzlů představujících krabičky s předdefinovaným chováním se vstupy a výstupy.

Editor a vizuální skriptování bylo v praxi ověřeno na ukázkovém projektu.

Zhodnocení vizuálního skriptování

Vizuální skriptování nemůže nahradit klasické programování. Některé věci se dělají složitěji a hlavně není tak mocné. Rozhodně má smysl pro jednoduché hry nebo návrh hry, kdy velice jednoduše lze vidět a otestovat návrh nebo nápad pro hru.

Výhody

- Jednoduché rozšíření programu o nové uzly. Programátor rozšíří program o specifické akce a události, které konkrétní hra vyžaduje. Designér vytvoří kolo hry a pomocí těchto uzlů je schopen vytvořit herní logiku.
- Přemýšlení nad aktérem jako nad konečným automatem. Stačí vytvořit jeden prototyp daného objektu a ten později pouze znovu využívat. Například mějme aktéra dveří a tlačítka. Jejich samotná funkčnost se vytvoří až později a je jedno, kde se bude nacházet.

Nevýhody

- Spojování uzlů se může stát velice nepřehledné, jakmile máme hodně provázaných akcí nebo proměnnou používá hodně uzlů.
- Problém umístění konkrétní funkčnosti aktérů resp. jejich provázání. Mějme fungující dveře a tlačítko vytvořené pomocí konečného automatu a definovaných událostí. Chceme otevřít dveře, když je tlačítko zmáčknuté. Toto chování lze vložit do tlačítka, dveří, skriptování scény nebo jiného logického aktéra, který s nimi souvisí. Potom okamžitě nevíme, kde je dané chování vytvořeno.
- Jednoduché matematické výpočty se vytvářejí složitě, protože na každou operaci je použita jedna akce a je potřeba používat hodně pomocných proměnných pro uchování mezivýsledku.

Zlepšení a rozšíření

- Vytvoření uzlu, který představuje sekvenci uzlů. Ve hře se často opakuje nějaká sekvence uzlů nebo chování. Tuto sekvenci bychom mohli zapouzdřit a představovala by jeden uzel s vstupu a výstupy, který bychom mohli kdekoliv používat.

- Možnost vytváření dynamických uzlů. Tzn. konkrétní chování uzlu se vytvoří až během překladu. Máme horší práci s jednoduchými matematickými výpočty. Pro jednoduchý výpočet musíme často použít velké množství akcí a pomocných proměnných. Hodil by se uzel, kam bychom pouze mohli napsat vzorec a připojit správné vstupy. S tím také souvisí složité přetypování proměnných hlavně mezi *Int* a *Float*. Mohlo by fungovat automatické přetypování, jaké je v programovacím jazyce. *Int* by šel použít jako *Float* nebo jako *String* (textová hodnota proměnné), apod.

- Předávání proměnných mezi scénami. Postavička hráče bude chtít mít stejnou hodnotu zdraví jako na konci předchozího kola.

- Možnost pluginů pro přidávání vlastních uzlů. Nyní se uzly hledají pouze v assembly herního engine. Pomocí pluginů by šlo také přenášet prototypy aktérů a prototypy uzlů (sekvence představené dříve).

- Editace kolizních tvarů aktéra na ploše scény. Aktérovi budeme moci změnit kolizní tvar, který převezme ze své grafiky. Také možnost vytváření aktérů přímo na ploše scény pomocí kolizních tvarů. Samotná grafika se jim může přiřadit později nebo vůbec. Tím se budou moci vytvářet jednoduše senzory a kolizní tvary pro zemi. Na scéně bychom vytvořili kolo hry a teprve potom bychom mohli označit zemi pomocí kolizních tvarů.

- Využít plný potenciál fyzikální knihovny. Nyní se fyzikální knihovna používá pouze na základní věci, ale knihovna toho obsahuje mnohem více. Možnost vytváření kloubů[1] přímo na scéně. Kloub je objekt, který má za úkol udržovat vzájemnou polohu dvou těles podle daného vztahu. Existují různé druhy kloubů, např. pro udržování konstantní vzdálenosti nebo kloub představující kladku. Poté bychom mohli jednoduše vytvořit fyzikálně simulované auto nebo motorku a hru ve stylu terénních 2D závodů z pohledu ze strany.

- Vytváření složitějších animací a skeleton animací pro aktéry.

- Výsledná hra použitelná pro více platforem. V dnešní době především na mobilní telefony.

- Optimalizace generovaného kódu. Vytvořit hierarchii tříd pro aktéry a zbavit se duplicitních tříd aktérů.

- Přímá podpora vytváření menu obrazovky pro hru. Nyní lze vytvořit scénu, která představuje jednoduché menu, ale je to spíše obcházení a složitější.

[1] V angličtině joint

Seznam použité literatury

[1] CATTO, Erin . Box2D [online]. Verze 2.1.2 [cit. 2012-07-16]. URL: `http://box2d.org`.

[2] CAWOOD, Stephen a MCGEE Pat. *Microsoft XNA Game Studio Creator's Guide*. New York: McGraw-Hill, 2nd edition, 2009. ISBN 00-716-1406-0.

[3] Epic Games. Unreal Development Kit [online]. Verze May 2012 Beta [cit. 2012-07-16]. URL: `http://www.unrealengine.com/udk/`.

[4] Farseer Physics Engine [online]. Verze 3.3.1 [cit. 2012-07-16]. URL: `http://farseerphysics.codeplex.com/`.

[5] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph a VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston (Massachusetts): Addison-Wesley, 1995, s. 233-242. ISBN 0-201-63361-2.

[6] GarageGames. Torque 2D [online]. Verze 1.7.6 [cit. 2012-07-16]. URL: `http://www.garagegames.com/products/torque-2d`.

[7] GLIZNETSOV, Andrey . TreeViewAdv for .Net [online]. Verze 1.7.0.0 [cit. 2012-07-16]. URL: `http://sourceforge.net/projects/treeviewadv/`.

[8] HABGOOD, Jacob, NIELSEN, Nana a RIJKS, Martin. *The Game Maker's Companion*. Berkeley (California): Apress, 2010. ISBN 14-302-2826-1.

[9] IERUSALIMSCHY, Roberto . *Programming in Lua*. Rio de Janeiro: Lua.Org, 2nd edition, 2006. ISBN 85-903798-2-5.

[10] Microsoft. XNA Framework [online]. [cit. 2012-07-16]. URL: `http://create.msdn.com/en-US/`.

[11] Microsoft. XNA Game Studio [online]. Verze 4.0 [cit. 2012-07-16]. URL: `http://www.microsoft.com/en-us/download/details.aspx?id=23714`.

[12] RODRIGUES, Makslane . Game Editor [online]. Verze 1.4.0 [cit. 2012-07-16]. URL: `http://game-editor.com`.

[13] SELLS, Chris a WEINHARDT, Michael. *Windows Forms 2.0 Programming (Microsoft .NET Development Series)*. Upper Saddle River (New Jersey): Addison-Wesley, 2nd edition, 2010. ISBN 0-321-26796-6.

[14] YoYo Games. GameMaker [online]. Verze 8.1 [cit. 2012-07-16]. URL: `http://www.yoyogames.com`.

A. Obsah přiloženého CD

Přiložené CD obsahuje následující soubory a složky:

- **Bin** - Obsahuje program.
- **Example Project** - Složka obsahuje ukázkový projekt zobrazující možnosti editoru a vizuálního skriptování.
- **Install** - Obsahuje instalaci programu. Před samotnou instalací programu ještě zkontroluje, jestli systém obsahuje potřebné knihovny (.NET Framework, XNA Framework a DirectX) pro běh programu. Pokud ne, tak pomůže k jejich instalaci. Nekontroluje XNA Game Studio, musí se nainstalovat ručně. Také zkontroluje, jestli grafická karta vyhovuje požadavkům pro běh programu, především minimální podporou Shader Model 2.0.
- **Source** - Složka se zdrojovými kódy.
- **Video Tutorial** - Obsahuje videa vysvětlující principy pro vytvoření jednoduché hry.
- **2D Platform Game Creator Documentation.chm** - Vygenerovaná dokumentace ze zdrojového kódu (v angličtině).
- **2D Platform Game Creator Nodes Reference.pdf** - Přehled uzlů ve vizuálním skriptování (v angličtině).
- **2D Platform Game Creator Thesis.pdf** - Tato bakalářská práce.

B. Ukázkový projekt

K práci je přiložen projekt hry zobrazující možnosti editoru a vizuálního skriptování. Spíše než o funkční hru se o jedna o ukázku všech možností. Sestává z pěti scén. Jedna scéna ukazuje možnosti parallax vrstvy a ostatní představují primitivní hru. Každé kolo postupně ukazuje možnosti. Jsou ukázány výhody i nevýhody.

Ovládání: Klávesy W, D - pohyb
Klávesa W - skok
Levé tlačítko myši - střelba
Pravé tlačítko myši - naklonování ruky na dobu pěti vteřin
Klávesa mezerník - aktivace štítu na dobu třiceti vteřin (pouze ve třetím kole)

Level 1

Postavička hráče - Jednoduchá implementace pohybu (díky uzlu *Basic Movement* má negativní vlastnosti na pohybující se plošince) a skákání. Ovládání animace pomocí konečného automatu. Zaměřování a ovládání ruky pomocí myši. Střílení, které nezpůsobuje zranění. Vytvoření aktéra svojí ruky za běhu hry. Postavička je složena ze dvou polygonů, tedy jeden tvar nemusí projít plošinkou s vlastností one-way platform.

Postavička nepřítele - Konstantní pohyb doprava nebo doleva. Při nárazu do stěny změní směr. Pravidelný pohyb ruky. Ruka střílí, bez účinku, pokud je postavička hráče na úsečce definované rukou a jejím zaměřováním. Pro použití aktéra jako prototyp musí také obsahovat jako dítě střelu, kterou střílí.

Tlačítko při kolizi s hráčem vyvolá událost, na kterou reaguje skriptování scény. Plošinka s vlastností one-way platform, která se pohybuje po předdefinované cestě.

Level 2

Skriptování scény pomocí konečného automatu definuje a ovládá jednotlivé části kola. Hráč a nepřítel mají zdraví a jejich střely už zraňují. Představeny senzory bez grafiky. Vytváření nepřátel za běhu.

Level 3

Nepřítel produkuje další nepřítele. Boss při menším počtu životů začne být zuřivější. Hráč může použít štít. Větší množství střel na scéně.

End

Představuje scénu po dohrání hry. Klávesou Enter se hra vypne.

Parallax Test

Simulace 3D hloubky pomocí parallax vrstev.