FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Tahová strategie pro Android
Student:	Dorian Řehák
Vedoucí:	Ing. Miroslav Balík, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Zanalyzujte, navrhněte, implementujte a otestujte hru žánru tahové strategie, pro implementaci si vytvořte vlastní 2D engine.

Požadavky
1. Implementace v jazyku Java pro OS Android,
2. podpora módu hry pro více hráčů přes síť,
3. podpora umělé inteligence pro počítačem řízené hráče,
4. podpora automatického generování herní mapy,
5. automatické testování komponent hry jednotkovými testy.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 29. ledna 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Tahová strategie pro Android

Dorian Řehák

Katedra softwarového inženýrství
Vedoucí práce: Ing. Miroslav Balík, Ph.D.

14. února 2019

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. února 2019 .....

Odkaz na tuto práci

Řehák, Dorian. *Tahová strategie pro Android*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Bakalářská práce popisuje návrh a implementaci 2D herního enginu a na něm založené hry žánru tahové strategie pro OS Android implementovaných v programovacím jazyce Java. Hra obsahuje umělou inteligenci v režimu hry jednoho hráče a podporu režimu více hráčů po síti. Herní úrovně jsou automaticky generovány.

Klíčová slova Android, Java, hra, tahová strategie, herní engine, OpenGL, režim více hráčů, umělá inteligence

Abstract

Bachelor's thesis describes design and implementation of a 2D game engine and a turn-based strategy game based on such engine for the Android OS implemented in the Java programming language. The game contains artificial intelligence in singleplayer mode and supports multiplayer mode over network. Game levels are automatically generated.

Keywords Android, Java, game, turn-based strategy, game engine, OpenGL, multiplayer, artificial intelligence

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Videohry se staly už v dobách rozšíření prvních osobních počítačů velmi populární záležistostí. A zájem neupadá, spíše naopak. S rostoucím výkonem mobilních telefonů se i trh s hrami pro mobilní telefony stává větší. Velká část uživatelů si dnes někdy hru na svém mobilním telefonu zahraje. Nicméně žádná hra nesplňovala mé požadavky, a tak jsem se rozhodl vytvořit svoji, podle svých představ. Mám tedy i osobní zájem na výsledku.

Tato práce popisuje tvorbu hry žánru tahové strategie. Jedná se o typ hry, kde, na rozdíl od real-time strategie, hráči nemusí reagovat okamžitě na měnící se situaci a mohou si plán do detailu promyslet na několik tahů dopředu.

Výsledek práce bude prospěšný pro všechny, kteří si rádi zahrají takovýto typ hry či kteří by se rádi takovýto typ hry sami naprogramovali, jelikož herní engine je znovupoužitelný.

V práci popisuji analýzu, návrh, implementaci a možnosti dalšího rozvoje aplikace.

Tahové strategie se vyznačují těmito typickými vlastnostmi:

- Hráči se střídají ve svých tazích.
- Herní pole je rozděleno do diskrétních prvků („políček“).
- Klíčem k úspěchu není rychlost reakce hráče, ačkoliv hra může obsahovat funkcionalitu časového limitu na tah proti cílenému zdržování.

Kapitola 1

Analýza

1.1 Analýza konkurence

1.1.1 Hoplite

Jedná se též o tahovou strategii pro Android. Zde hráč ovládá jen jedinou jednotku (hoplíta). Herní prostředí spočívá v šestiúhelníkové herní mapě, kde hráč musí využívat vlastností terénu, kalkulovat dostřely nepřátelských jednotek i své vlastní, jejich možné akce a tak podobně.

Hra nepodporuje hru více hráčů po síti, vždy jde jen o soutěž mezi uživatelem a umělou inteligencí řízenou protistranou. Hra neobsahuje seznam pravidel v přečtení, místo toho obsahuje interaktivní cvičnou herní úroveň. Hra implementuje sekvenci pevně daných herních úrovní, skrz které se hráč musí probít. Ačkoliv je hra zdarma, pro pokračování po 16 úrovních je třeba koupit si „prémiovou“ verzi, která dále odemyká některé další funkcionality. Uživatel si nemůže nastavit úroveň obtížnosti.

Použití šestiúhelníkových políček mi přislo praktické a tak jsem se jimi inspiroval.

1.1.2 Antiyoy

I u této hry je mapa složena ze šestiúhelníkových políček. Cílem hry je obsazovat neutrální a soupeřova políčka svými figurkami. Herní úrovně jsou pevně dané, ale hra obsahuje i editor úrovní, která umožňuje hráči vytvořit si vlastní. Hra též umožňuje uložit a načíst rozehranou hru.

Ani tato hra nepodporuje hru po síti, uživatel se musí spokojit s umělou inteligencí. Lze u ní nastavit obtížnost.

1.1.3 Militia

Jde o jednoduchou hru postavenou na herním enginu Unity. Herní pole se skládá ze čtverových políček, tvořící pole 8x8. Cílem je eliminace soupeřových

figurek skokem na jejich pozici či v některých situacích vedle nich. Hráč je omezen celkovým počtem tahů.

Hra neobsahuje popis pravidel hry, ale má tréninkovou úroveň. Režim hry po síti hra neobsahuje. Hra je zdarma, avšak některá funkcionalita hry je placená. Rozehranou hru nelze uložit.

1.2 Herní engine

Herní engine poskytuje všechny funkcionality nutné k vytvoření hry: vykreslování grafiky, načítání zdrojů, komunikaci po síti a tak podobně. Čelil jsem rozhodnutí, zda vytvořit engine čistě pro hry žánru tahové strategie či zda vytvořit obecnější engine, umožňující použití i v jiných žánrech her. Přiklonil jsem se k druhé možnosti a tudíž je funkcionalita specifická pro tahové strategie součástí hry a ne herního enginu. Celkové množství kódu je stejné, ale takto lze engine využít i pro jiné účely (i neherní).

Pro herní engine jsem zvolil jméno *Beryl*, čistě náhodně, pro lépe čitelný text práce.

Beryl vyžaduje Android API verze alespoň 24, důvodem je zpřístupnění některých novějších Java knihoven (například třída Optional[1]). Tato verze API dále garantuje OpenGL ES verze 3.2[2], avšak Beryl užívá k vykreslování grafiky OpenGL ES verze 2.0. Zvolil jsem tuto verzi, jelikož na rozdíl od verze 1.x nepoužívá *fixed pipeline*, ale *programmable shaders*[3], kterých užívám. Funkcionalitu poskytnutou vyššími verzemi OpenGL ES Beryl nepotřebuje.

Verze Androidu	Podíl na počtu Android zařízení
2.3.3-4.3	3,5%
4.4	7,6%
5.0	3,5%
5.1	14,4%
6.0	21,3%
7.0	18,1%
7.1	10,1%
8.0	14,0%
8.1	7,5%

Tabulka 1.1: Rozšíření verzí OS Android na trhu[4]

Android API 24 odpovídá verzi 7.0 (Nougat). Podle statistik společnosti Google z října 2018 podporuje API verze 24 49,7 % zařízení s OS Android na trhu [tabulka 1.1].

1.3 Event systém

Srdcem každého herního enginu je takzvaný *event system*[5] (též znám pod jinými názvy, například *messaging system*). Cílem této komponenty je informování o událostech a předávání zpráv mezi komponentami herního enginu či, především, hry.

Výhody takového systému oproti přímému volání funkcí/metod jsou následující:

- Lze za běhu měnit vazby mezi komponentami, které vysílají zprávy, a komponentami, které je poslouchají. Ačkoliv tato funkce se příliš nevyužije v herním enginu samotném, je téměř nezbytnou v programu komplexnější hry, kdy se v reakci na akce hráče mění herní stav a souvislosti mezi entitami na herním poli.

- Typicky umožňuje předávání zpráv mezi komponentami běžícími na různých vláknech, kde přímé volání metod nelze uplatnit.

Nevýhodou oproti přímému volání funkcí/metod je výrazně vyšší režie a komplexita, je tedy vhodné event system používat jen v nutnosti.

Existují dvě hlavní architektury takového systému:

- Broadcasting. U event systému tohoto typu existuje jeden globální objekt, který má frontu zpráv, do které každý odesílatel může přidat svoji. Tento globální objekt z fronty odebírá jednu zprávu po druhé a „vysílá“ je. Posluchači (v angličtině *listeners*) jsou registrováni spolu s filtrem určujícím, o které zprávy mají zájem[6]. Zprávy tedy musejí mít nějaký identifikátor svého typu.

- „Signály a sloty“. Objekt *slot* má svou callback metodu a je registrován u žádného, jednoho či více objektů typu *signal*. Objekt typu *signal* má metodu pro vyslání „signálu“, který způsobí zavolání callback metody ve všech objektech typu *slot*, které jsou u daného objektu *signal* registrovány. Ten je použit například v sadě knihoven Qt[7] či Boost[8].

1.4 Resource management

Pod tímto pojmem se skrývá funkcionalita herního enginu pro načítání a uvolňování různých zdrojů, jako například obsahů souborů, načtených modelů a textur či bufferů modelů a textur na grafické kartě. Tato komponenta musí být schopná načítat zdroje v nutném pořadí, například pro načtení textury do grafické karty, musí nejdřív načíst obsah souboru, zpracovat ho a nahrát ho do grafické karty. Dále musí být navržena tak, že zbytek herního enginu nemusí řešit zda nějaký zdroj je momentálně načtený či ne — resource management komponenta toto řeší sama za všechny ostatní.

Běžným návrhem resource managementu v herních enginech je globální objekt, *resource manager*[9], který vytváří objekty reprezentující žádaný zdroj. Ovšem tyto objekty nejsou přímo samotnými žádánými daty, slouží pouze jako takzvané *handlers*. Komponenta, která žádá data, k datům může přistupovat pouze pomocí metod handleru. Resource manager skrytě provádí jejich načítání či uvolňování podle potřeby. Pokud je handler požádán o předání dat, která reprezentuje, podívá se, zda jsou načtená. Pokud ano, vrátí je. Pokud ne, provede jejich načtení (vlákno typicky čeká než se tato operace provede), uloží si je pro budoucí použití a vrátí je jako výsledek. Může se stát, že systému dochází například paměť, v tom případě resource manager uvolní nejméně používané zdroje (zjištěno přes nějakou formu LRU algoritmů[10]). Zbytek herního enginu jen volá metodu *get* nad handlerem, která vrací žádaná data, a tyto operace načítání a uvolňování jsou mu zcela skryty.

1.5 Vykreslování grafiky

Vykreslování 2D grafiky v OS Android lze docílit několika různými způsoby, především použitím třídy ImageView a OpenGL.

První možnost je založena na systému *views*, tedy elementárních prvků UI v Android aplikaci. Každý vykreslovaný objekt by měl svůj vlastní *ImageView*, který by zobrazoval jeho texturu. Výhodou takového řešení by byla jednoduchost implementace vykreslování i získávání uživatelského vstupu, který by Android UI knihovna rovnou předávala příslušnému *ImageView*. Nevýhody, pro mé použití však převyšují výhody: nejistota existence HW akcelerace při vykreslování, celková neflexibilita vykreslování (neexistence *shaders*[11]), problematické chování alfa kanálu při vstupu (i prázdná část ImageView je chápána jako jeho plocha pro vstup).

Proto jsem se rozhodl vybrat si druhou možnost, OpenGL. Konkrétně se jedná o OpenGL ES 2.0, jenž je povinnou součástí Androidu od verze 2.2.

1.6 Umělá inteligence

Ve hrách se užívá široká škála algoritmů pro umělou inteligenci. Liší se ve svých schopnostech a především svojí výpočetní složitostí. U mobilní hry je velmi žádoucí možnost paralelizace úlohy, jelikož mobilní telefony zpravidla užívají procesorů s větším počtem jader, ale nižší výkonem na jádro. Dalším aspektem vhodným vzít v úvahu je dopad na spotřebu.

Pro potřeby své hry jsem zvážil následující algoritmy:

- Behaviour trees,
- Minimax,
- Monte Carlo Search Tree.

1.6.1 Behaviour trees

Tento algoritmus je jeden z nejběžněji užívaných ve videohrách. Jeho princip spočívá v programátorem definovaném stromu rozhodování, jenž je během hry procházen a na základě v něm definovaných podmínek se provádí v něm definované akce[12]. V prostředí tahové strategie by pro podmínky mohly být použity například poměr sil či vzdálenost nejbližší figurky soupeře, pro akce například přesun na určité políčko.

Implementace takového algoritmu je velice jednoduchá, ale jeho největší výhodou je minimální náročnost na výpočetní sílu počítače. Proto se často používá ve hrách odehrávájících se v realném čase, například v žánru závodních her či FPS[13], kde není prostor pro mnohasekundová rozhodování.

Hlavní nevýhodou, především v kontextu mé práce, je potřeba definování stromu rozhodování programátorem. To s sebou přináší problém „osobnosti“ takové umělé inteligence — její tendence preferovat určitý styl rozhodování. Zkušenější hráč je pak schopen předvídat její tahy.

Vzhledem k tomu, že u tahové strategie není tlak na rychlé reakce, tento algoritmus jsem zavrhl.

1.6.2 Minimax

Minimax je známý algoritmus, často užívaný právě v tahových strategiích. Vytváří strom všech možných herních stavů na několik tahů dopředu, vyhodnocuje „kvalitu“ každého dosaženého herního stavu a vybírá tah, který vede k nejžádanějšímu výsledku. Jelikož v praxi je nemožné projít všechny možné herní stavy, algoritmus je omezen na určitý počet tahů dopředu. Tím se také snadno reguluje jeho obtížnost pro lidského hráče.

Minimax ovšem selhává u her, které mají příliš vysoký tzv. *branching* factor (počet možných tahů z určitého herního stavu). Důvodem je potřeba Minimaxu projít všechny možné herní stavy do dané úrovně. Čím je tedy strom širší, tím musí být mělčí, pokud chceme zachovat dobu výpočtu. Varianty minimaxu, jako například alfa-beta prořezávání, mohou pomoci.

1.6.3 Monte Carlo Search Tree

MCST je vhodným kandidátem na použití v případech, kde je Minimax nevhodný z důvodu vysokého branching factoru. Stejně jako Minimax vytváří strom možných herních stavů, ovšem nezkouší se všechny do určité hloubky. Místo toho zkouší různé tahy, pro ně vytváří nové uzly ve stromu a pro každý z nich provede tzv. simulaci. Simulace spočívá v simulované hře, která se ukončí pouze prohrou či výhrou. Při tzv. light play se tohoto konečného stavu dosahuje náhodnou hrou, lze též využít nějaké heuristiky. Jakmile je dosaženo vítězství, je strom procházen od daného uzlu k vrcholu stromu a v každém prošlém uzlu je zvýšeno počítadlo vítězství. V případě prohry i výhry se zvyšuje počítadlo simulovaných her ve všech uzlech zmíněných v předchozí

větě. Algoritmus tak vidí, které tahy častěji vedou k vítězství, a které méně často. Je žádoucí dále prozkoumávat hru pokračující z tahů častěji vedoucím k vítězství — je však otázkou, jaký má být poměr mezi exploitací a explorací. Toto je řešeno mnoha způsoby, oblíbeným řešením je UCT (*Upper Confidence bound 1 applied to Trees*)[14].

Obtížnost je určena počtem simulací.

Oproti Minimaxu nepotřebuje evaluační funkci herního stavu, čímž odpadá možnost zkreslení schopností umělé inteligence prográmátorem. Nevýhodou je však vysoká výpočetní náročnost v případě light play — ta roste s větším počtem figurek a větším prostorem pro pohyb figurek, protože simulace her v algoritmu musí vždy dojít až do stavu prohry či výhry.

1.7 Generování herní mapy

Generovaná herní mapa musí být generována pseudonáhodně a ne náhodně, pokud ji nechceme celou posílat po síti protějšku v režimu dvou hráčů. Pokud je generována pseudonáhodně, stačí poslat pouze *seed* pseudonáhodného generátoru čísel a druhá strana si může totožnou mapu vygenerovat lokálně.

Hlavní výzvou při generování herních úrovní/map je fakt, že políčka nemohou být jednoduše náhodná. Mapa by pak vypadala jako šum. Generovaná políčka musejí být nějakým způsobem ovlivněna políčky ve svém okolí, aby dohromady tvořila větší požadované obrazce.

Častým zdrojem dat pro generování herních úrovní jsou procedurální textury, jejichž hodnoty jsou pak interpretovány a převedeny do reprezentace herní úrovně. Příkladem je Perlinův šum, jeho vylepšení Simplex noise či value noise. Tyto algoritmy mohou generovat textury i vyšší dimenze než 2, čehož je využito například ve hře Minecraft.[15]

Kapitola 2

Návrh a implementace

2.1 Struktura projektu

Projekt je rozdělen na dvě části:

- Herní engine, který jsem nazval *Beryl*.
- Samotná hra, zvaná *Battle for Hexland*, mající jako závislost herní engine.

Obě tyto dvě části jsou tzv. Android moduly. Znovupoužití modulu Beryl například pro jinou hru je velice jednoduché.

2.2 Užité nástroje

K vývoji bylo použito Android Studio[16], oficiální vývojové prostředí pro Android. Jako nástroj pro automatizované sestavování programu byl použit Gradle[17]. Správu verzí měl na starost nástroj git[18]. Pro tvorbu UML diagramů byl užit nástroj ArgoUML[19]. Textury byly vytvořeny v programu GNU Image Manipulation Program[20] a Inkscape[21]. Soubory obsahující geometrii byly vytvořeny mým vlastním programem k nalezení na přiloženém optickém disku.

2.3 Herní engine

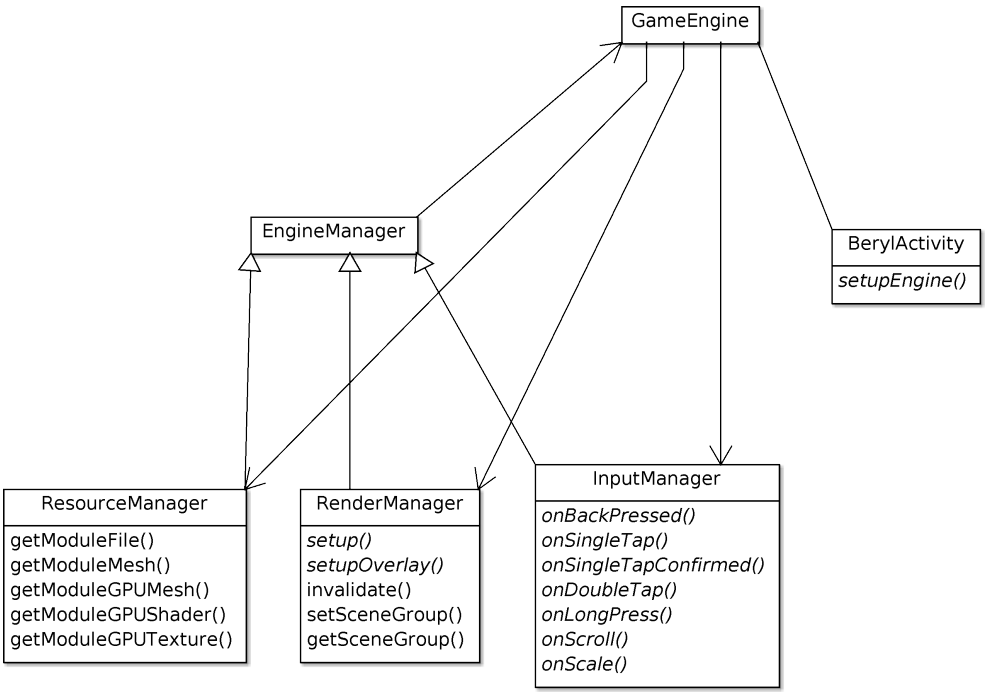
Vstupním bodem hry používající Beryl jako svůj herní engine je třída *BerylActivity*. Hlavní třídou reprezentující celý herní engine je třída *GameEngine*. V celé běžící aplikaci existuje právě jen jedna instance a ta je referencovaná z *BerylActivity*. Během běhu hry nelze používat jiné aktivity, jelikož pak by nebylo možno vykreslovat OpenGL grafiku — ta je vykreslována pomocí podtřídy *GLSurfaceView* jménem *SKSurfaceView*, ležící ve stromu Views

právě *BerylActivity*. Toto však není nijak omezující, Android UI prvky je stále možno jednoduše používat, vizte kapitola o vykreslování grafiky.

Třída *GameEngine* funguje jako můstek mezi několika podtřídami třídy *EngineManager*. Každý *EngineManager* poskytuje funkcionalitu určitého subsystému enginu, například reagování na uživatelský vstup či načítání zdrojů, a dále provázání zpět k instanci *GameEngine* a skrz ni možnost přístupu k jiným *EngineManager* podtřídám.

Beryl poskytuje tři podtřídy *GameManager*:

- *ResourceManager* — poskytuje přístup k instancím podtříd *ResourceModule*; slouží k vytváření handlerů na zdroje (vizte kapitola Resource Management).
- *RenderManager* — poskytuje rozhraní pro reprezentaci scén k vykreslení pomocí OpenGL a vytváření vrstvy Android UI widgetů.
- *InputManager* — dostává oznámení o uživatelském vstupu; tuto třídu je třeba uživatelem enginu rozšířit o smyslupný kód, který je bude zpracovávat.



Obrázek 2.1: Struktura herního enginu

Hra využívající Beryl jako svůj engine, musí vytvořit svého potomka třídy *BerylActivity* a implementovat metodu *setupEngine*. Ta má za úkol vytvořit instanci třídy *GameEngine*. Uživateli enginu je zde umožněno užít své podtřídy *GameEngine*, či podtřídy jednotlivých *GameManager* tříd, ze kterých se skládá. Může si i přidávat nové podtřídy *GameManager* do své instance GameEngine.

V ten moment už lze aplikaci zkompilovat a spustit, bude však zobrazovat pouze černou obrazovku. Dále je potřeba vytvořit instance *Scene*, naplnit je objekty *RenderEntity* a *InputEntity*, přidat je do *SceneGroup* a *SceneGroup* zaregistrovat u *RenderManager*. K tomu slouží metoda *setup* v *RenderManager*, kterou si uživatel musí implementovat. Bude zavolána jen jednou při startu a to tehdy, když už je zbytek enginu, především *ResourceManager*, připraven k použití.

Jednoduchým způsobem, jak použít Beryl, je zkopírovat jeho soubory do projektu a přidat:

```
implementation project(path: ':beryl')
```

do sekce závislostí v Gradle skriptu hry.

2.4 Event system

V mém herním enginu jsem se rozhodl použít systém signálů a slotů. Důvodem je moje zkušenost s takovou architekturou z frameworku *Qt* a celkově méně centralizovaná povaha. Ta umožňuje nižší režii kvůli absenci zamykání globální fronty událostí a zbavuje mě povinnosti udržovat systém identifikátorů typů zpráv a jejich filtrů.

Stejně jako u *Qt*[7], má každý signál a slot přiřazen své číslo vlákna. V případě, že signál i slot, který je jím aktivován, mají stejné číslo vlákna, je možné provést přímé volání metody a tedy zcela se zbavit výpočetních nákladů použití fronty zpráv. V případě, že se čísla vláken liší, signál vloží zprávu do fronty vlákna daného slotu (každé vlákno má svoji) — vlákno si později tuto zprávu přečte a aktivuje daný slot. Můj event systém umožňuje jak asynchronní (výchozí) tak i synchronní komunikaci mezi vlákny. Též umožňuje zpožděné zpracování zprávy po uplynutí zadané prodlevy - tuto funkci lze využít pro timeouty či animace.

Android už svůj systém pro předávání zpráv má, implementovaný formou tříd *Handler* a *Looper*. Nepoužil jsem ho přímo, jelikož jsem ho vyhodnotil jakožto příliš nízkoúrovňový — neumí synchronní posílání zpráv, neřeší pamatování si propojení ve stylu signal/slot a předávání instancí *Message* nemá compile-time typovou kontrolu pro parametry zpráv (parametry zprávy se předávají pomocí slabě typovaného asociativního pole implementovaného třídou *Bundle*[22]). Nicméně Android implementace má vše potřebné pro doprogramování všech těchto funkcí nad ní. Proto můj event systém tedy využívá implementace poskytnuté Androidem a dále ji rozšiřuje.

Můj event system lze použít formou signál-slot mechanismu popsaného výše, tak i přímou aktivací slotů bez signálů. To je výhodné pro jednorázové akce, kde by instance signálů jen překážely.

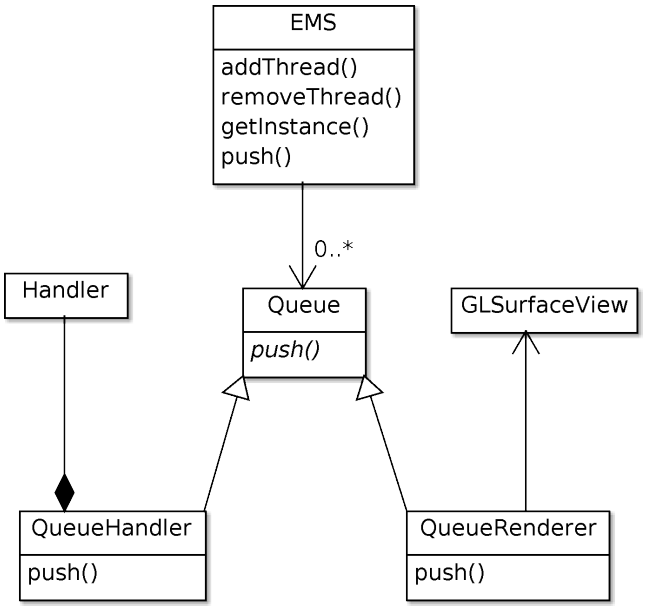
```
public class TestClass{
  private ObjectInfo oi = new ObjectInfo();
  public Signal1<String> signal = new Signal1<>(oi);
  public Slot1<String> slot = new Slot1<String>(oi){
    @Override
    void receive(String p1){
      System.out.println("Slot activated! " + p1);
    }
  };
  public TestClass(){
    signal.register(slot);
    // pouziti signalu
    // v tomto pripade bude metoda receive primo zavolana
    signal.emit("parameter");
    // prima aktivace slotu
    // asynchronni a zpozdena o 1000 milisekund
    // v tomto pripade bude uzito fronty zprav
    slot.process("parameter", false, 1000);
  }
}
```

Pro registraci vláken do event systému a pro přístup k jejich frontám zpráv slouží třída *EMS*. Využívá návrhového vzoru *singleton*, existuje tedy jen jedna instance v rámci aplikace. Třída *EMS* udržuje mapování unikátních čísel vláken (přiřazováno Javou) na jejich fronty. Existují dvě implementace front:

- *QueueHandler* — implementace fronty jako rozšíření *Looper*/*Handler* systému poskytnutého Androidem.
- *QueueRenderer* — implementace fronty pro OpenGL vlákno. Toto je nutné, jelikož OpenGL vlákno má svůj vlastní systém předávání zpráv skrz *GLSurfaceView*. Nikde jinde tato implementace fronty není užita.

Před ukončením vlákna je nutno vlákno z event systému odregistrovat, jinak dojde k úniku paměti — Androidem poskytnutý *Handler* by stále držel referenci na objekt vlákna a jeho *Looper*.

Pro zjednodušení vytváření nových vláken užívajících tohoto event systému, obsahuje Beryl též speciální podtřídu třídy *Thread*. Ta zařizuje vše potřebné automaticky, uživatel nemusí řešit registraci, deregistraci atd.



Obrázek 2.2: Event systém (třídy *Handler* a *GLSurfaceView* jsou součástí systémové knihovny OS Android)

2.5 Resource management

V mém herním enginu jsou následující typy zdrojů (třídy sloužící jako *handlers* podle schématu popsaného v kapitole *Analýza*):

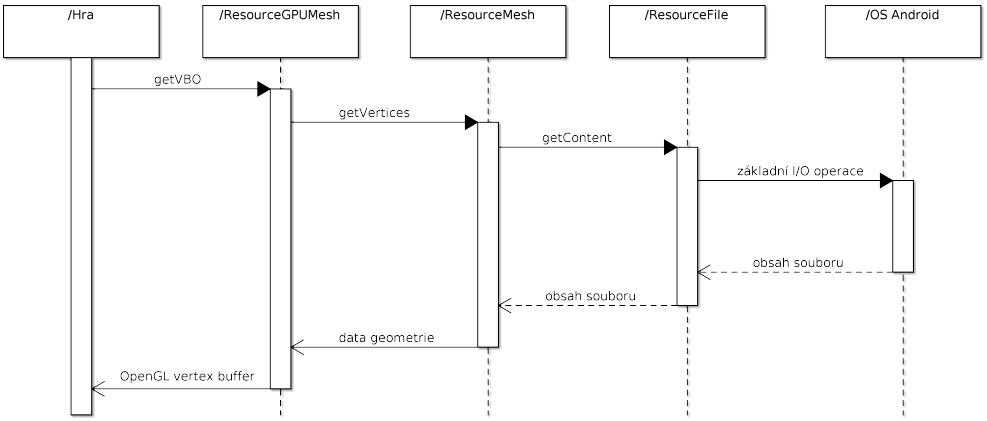
- *ResourceFile* — reprezentuje obsah souboru. Přistupuje k souborům relativně ke složce assets v souborovém stromu projektu.
- *ResourceMesh* — reprezentuje model složený z trojúhelníků a jejich parametrů (například texturové souřadnice). Jako závislost má *ResourceFile*, ze kterého jsou data čtena a zpracována.
- *ResourceGPUMesh* — reprezentuje OpenGL buffer obsahující *ResourceMesh*, který je tedy jeho závislostí.
- *ResourceGPUTexture* — reprezentuje OpenGL buffer obsahující texturu. Jeho závislostí je *ResourceFile* reprezentující soubor zpracovatelný Android knihovnami (například PNG soubor).
- *ResourceGPUShader* — reprezentuje OpenGL *vertex shader* a *fragment shader*[11].

Důvodem k rozdělení *ResourceMesh* a *ResourceGPUMesh* je fakt, že pro zpracování uživatelského vstupu potřebuji mít geometrická data i mimo GPU, pro zjištění zda byl dotek uvnitř vykresleného objektu či ne. U *ResourceGPUTexture* tato potřeba není.

Každý z výše uvedených zdrojů má i svůj *ResourceModule*. *Resource* třídy se chovají jako cache dat, které reprezentují. *ResourceModule* objekty fungují jako *factory* (v rámci *factory pattern*) pro příslušné *Resource* objekty.

ResourceManager objekt obsahuje instanci každé podtřídy *ResourceModule*. Uživatel herního enginu má možnost snadno si definovat své nové podtřídy *ResourceModule* a přidat si je do vlastní podtřídy *ResourceManager*.

Závislosti načítání dat jsou řešeny jednoduchým zřetězením volání metod *load* mezi *Resource* objekty. Mějme například instanci *ResourceMesh*, který při svém načítaní čte data z přiřazeného *ResourceFile* (který při načítání čte data pomocí Android funkcí pro čtení souboru) a mějme je oba nenačtené. Pokud u této instance *ResourceMesh* zavoláme metodu *getVertexCount*, zavolá se vnitřní metoda *load*, která načte data modelu. Tato metoda *load* provede volání metody *getContent* u přiřazené instance *ResourceFile*.



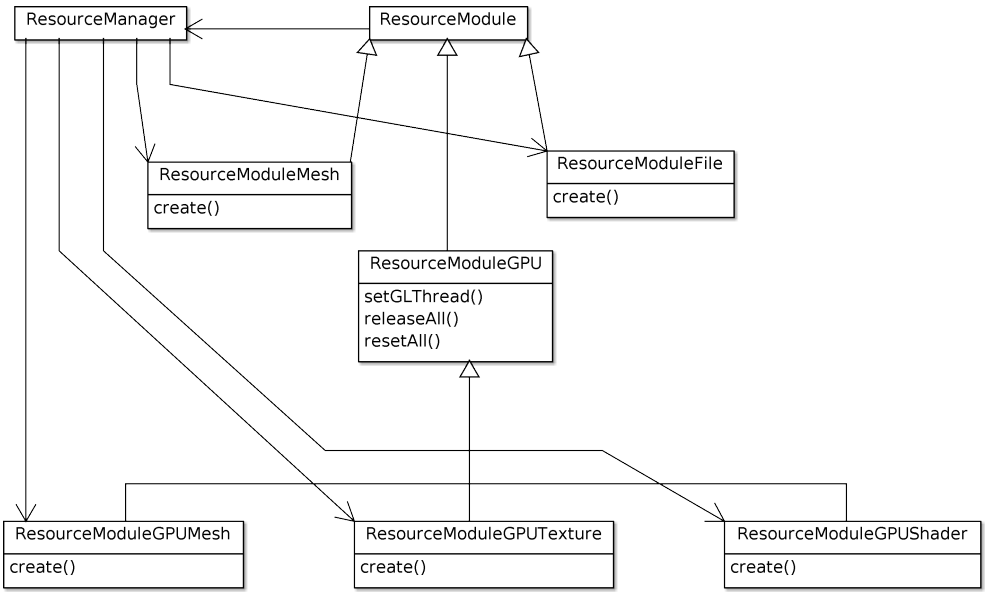
Obrázek 2.3: Načítání zdrojů (v tomto případě dat geometrie do GPU) za předpokladu, že žádný z nich zatím nebyl načten

Závislosti *Resource* instancí na jiných *Resource* instancích jsou určeny jako parametry konstruktorů a jsou neměnné. Důsledkem je, že *Resource* instance tvoří *directed acyclic graph*.

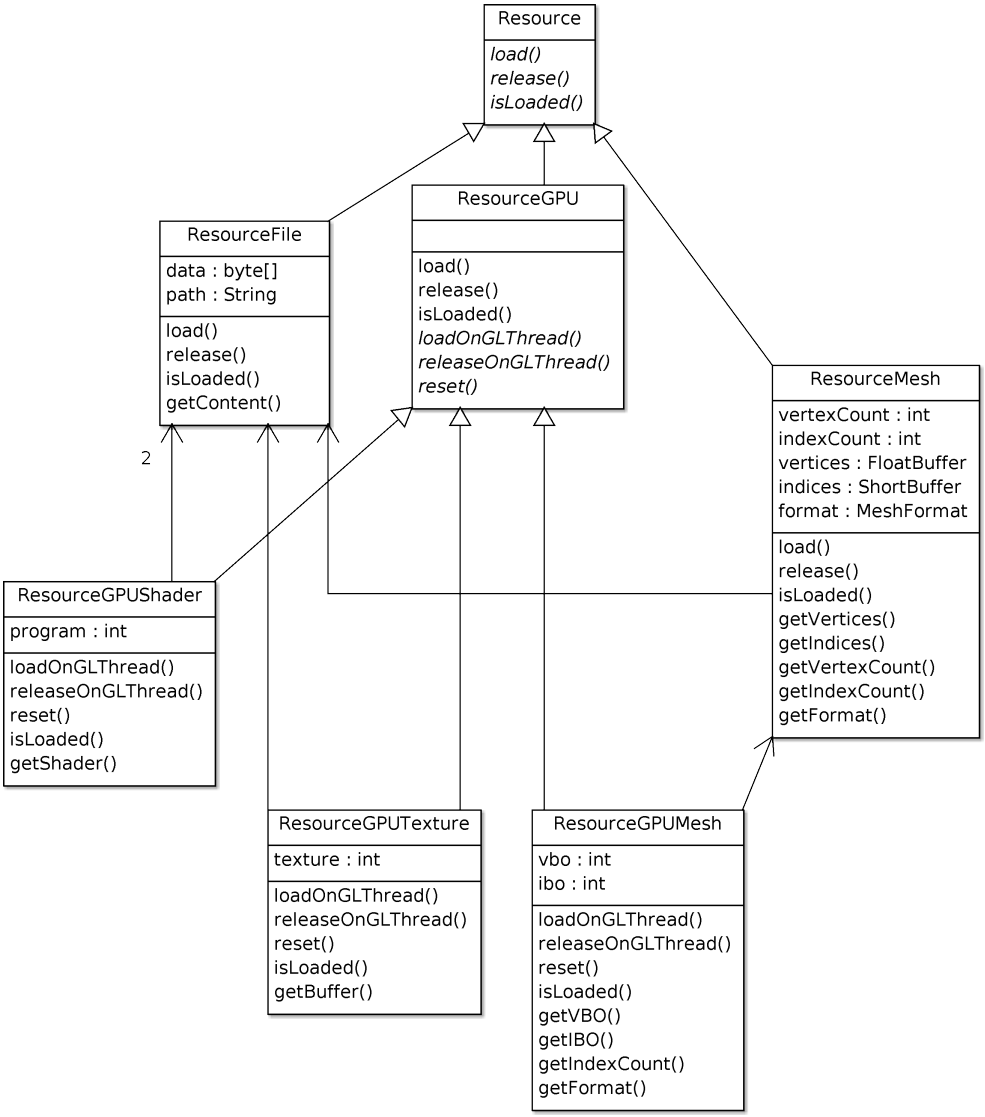
Všechny implementace podtříd *ResourceModule* musí být thread-safe, jelikož zdroje mohou být žádány z různých vláken, především z vykreslovacího vlákna.

Třídy *ResourceGPU* a *ResourceModuleGPU* neslouží k přímému použití, ale jako základ pro podtřídy operující s OpenGL zdroji. Poskytují sdílenou funkcionalitu pro komunikaci s OpenGL vláknem pomocí event systému. Platí zde tedy stejná pravidla jako ta popsaná v kapitole o event systému: pokud je žádost o načtení zdroje podána z OpenGL vlákna, je zpracována přímo zavolánímdané metody; pokud je žádost podána z jiného vlákna, je přidána zpráva do fronty OpenGL vlákna skrz event systém a synchronně se čeká na její splnění v OpenGL vlákně.

Existují též pomocné třídy *ResourceList* a *ResourceWeakList* zaobalující seznam instancí *Resource*. Jsou užitečné například pro hromadné uvolnění zdrojů při opuštění herní úrovně. Třída *ResourceGPUWeakList* je speciálním případem pro OpenGL zdroje a je užíván podtřídami *ResourceModuleGPU* pro držení slabé reference na všechny vytvořené OpenGL zdroje — Beryl toto potřebuje pro resetovaní zdrojů po resetu OpenGL kontextu, například v důsledku otočení obrazovky a tím vynuceného restartu aktivity.



Obrázek 2.4: Výchozí *ResourceManager* a jeho moduly



Obrázek 2.5: Třídy zdrojů

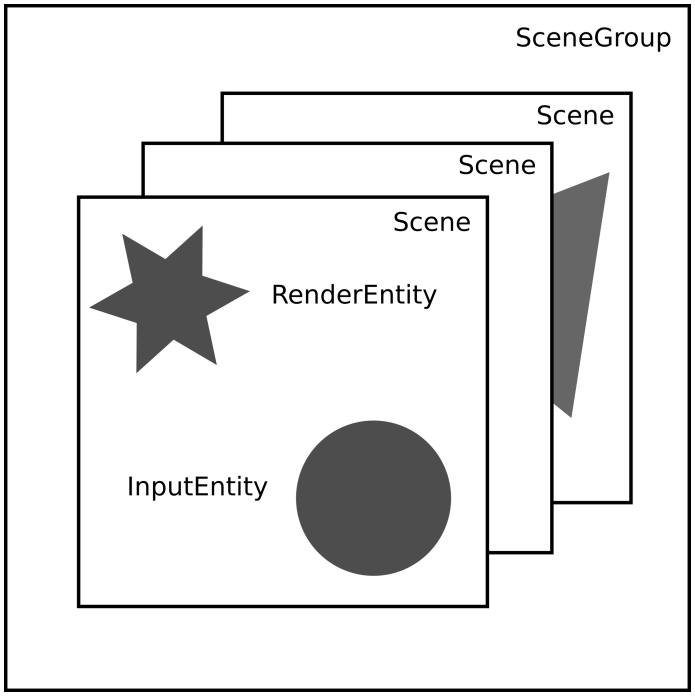
```
void accessResources(ResourceManager rm){
  ResourceModuleFile rmf = rm.getModuleFile();
  ResourceModuleGPUTexture rmt =
     rm.getModuleGPUTexture();
  ResourceGPUTexture tex =
     rmt.create(rmf.create("textures/test.png"));
  // ...
  /* vrati OpenGL buffer index;
     pokud textura neni nahrana v grafickem zarizeni,
        bude automaticky nahrana */
  int gl_buffer = tex.getBuffer();
  // ...
}
```

2.6 Vykreslování grafiky

Beryl k vykreslování grafiky používá OpenGL, které zpřístupňuje hardwarovou akceleraci na grafické kartě. To zvyšuje složitost implementace oproti prvnímu popsanému způsobu řešení použitím *ImageView*, Beryl totiž musí spravovat nahrávání dat do grafického zařízení a též jejich uvolňování. Android vyžaduje pro vykreslování grafiky pomocí OpenGL samostatné vlákno, volání OpenGL funkcí mimo toto vlákno nejsou přípustná. Toto vlákno ve smyčce jen vykresluje právě nastavenou scénu, přičemž se uspí, pokud se scéna nemění. Manipulace se scénou se provádí v jiných vláknech, konkrétně v mé hře herní logika běží na hlavním (Android UI) vlákně. Z toho vyplývá nutnost synchronizace při přístupu k objektům scény.

Hra tahové strategie, narozdíl od jiných žánrů her, nevyžaduje neustálé vykreslování. V běžných herních enginech běží nekonečná smyčka, která aktualizuje stav herních objektů a poté je vykresluje. U tahové strategie toto není potřeba, jelikož scéna se mění jen v reakci na akce hráčů. Není tedy nutné vykreslovat scénu 60krát za sekundu, ale jen když se scéna změní. To je na mobilních zařízeních o to více důležité, protože jsou silně omezeny spotřebou.

Scéna k vykreslení je reprezentována objekty tříd *Scene* a *SceneGroup*. SceneGroup reprezentuje celý „svět“ k vykreslení a skládá se z žádného až mnoha objektů třídy *Scene*. Právě jedna instance *SceneGroup* může být registrována k použití Berylem v jeden okamžik. *Scene* reprezentuje jednu „vrstvu“ k vykreslení a obsahuje seznam objektů k vykreslení (instance třídy *RenderEntity*) a seznam objektů přijímajících vstup (instance třídy *InputEntity*). Každá *Scene* má své nezávislé nastavení pohledu. Tím lze například docílit toho, že uživatel posouvá herní mapu (první vrstva), přes kterou je vykreslována nepohybující se grafika uživatelského rozhraní (druhá vrstva). Jednotlivé scény jsou vykreslovány v pořadí, v jakém jsou přidány — naopak při



Obrázek 2.6: Reprezentace scény k vykreslení a přijímání uživatelského vstupu

zpracovávání vstupu se scény prochází od poslední přidané k první přidané. Objekty pro vykreslování a pro přijímání vstupu (*RenderEntity*, *InputEntity*) jsou odděleny, jelikož někdy je žádoucí vykreslovat objekty, které nereagují na vstup a zároveň jsou někdy potřeba „neviditelné“ objekty, které na vstup reagují.

Instance *ModelMatrixGenerator*, *ViewMatrixGenerator* a *ProjectionMatrixGenerator* poskytují funkcionalitu nutnou ke generování transformačních matic[23]. *ProjectionMatrixGeneratorSC* generuje projekční matici, která bere v úvahu poměr stran displeje; *ProjectionMatrixGeneratorNDC* nebere v úvahu poměr stran displeje — obraz je roztažený, pokud obrazovka není čtvercová. Tyto třídy si ukládají vygenerované matice a parametry k tomu užité a pokud engine znovu požádá o vygenerování se stejnými parametry, bude vrácena již existující matice.

Třída *RenderEntity* má metodu *render*, která je volána z OpenGL vlákna. Je na uživateli enginu, aby si ji implementoval. Beryl též poskytuje podtřídu *GenericRenderEntity*, která implementuje běžné potřeby vykreslování, jako například multitexturing či předávání hodnot barev do shaderů. Má hra svoji vlastní speciální implementaci nepotřebovala a vystačila si s *GenericRenderEntity*.

Třída *InputEntity* má metodu *isWithin*, přijímající souřadnice relativní ke středu souřadnicového systému entity, odpovídající, zda se tento bod nachází

uvnitř entity. K převodu absolutních souřadnic na obrazovce na souřadnice relativní ke středům jednotlivých objektů se provede přesně opačný postup než při vykreslovaní grafiky. Bod dotyku je vynásoben inverzní maticí součinu modelové, pohledové a projekční matice. Beryl poskytuje dvě implementace *InputEntity*:

- *InputEntityCircle* — jednoduše definuje platnou oblast dotyku jako kruh o zadaném poloměru. Jedná se o výpočetně velmi efektivní řešení.
- *InputEntityMesh* — jako geometrii entity používá zadaný *ResourceMesh*. Pro každý trojúhelník meshe vypočítá barycentrické souřadnice bodu dotyku a podle jejich hodnoty rozhodne, zda se v trojúhelníku bod nachází. Pokud se v žádném nenachází, bod dotyku neleží v entitě. Jedná se o výrazně vypočetně náročnější řešení, jehož náročnost se zvyšuje se složitostí geometrie. Je proto rozumné nepoužívat tu samou geometrii jako při renderingu, pokud je složitá, a použít její zjednodušenou variantu.

Dotyky, posouvání a další vstupní operace od uživatele jsou detekovány a je zjištěno, kterého *InputEntity* objektu se týkají. Příslušné prázdné metody *InputManager* jsou zavolány (jako parametr mimo jiné předána relevantní instance InputEntity) a je na uživateli enginu, aby si je implementoval v podtřídě.

Je nutno podotknout, že při každém narušení popředí běžící aktivity Android ruší OpenGL kontext[24]. To zneplatní všechny OpenGL buffery (textury, pole s data vertexů a tak dále). Uživatel enginu toto nemusí řešit, Beryl tento problém řeší sám, ovšem při použití většího množství grafických dat by mohlo docházet k prodlevám při jejich opětovném nahrávání do grafického HW.

K zobrazení Android UI widgets slouží metoda *setupLayout* třídy *RenderManager*. Ta je zavolána kdykoliv *BerylActivity* potřebuje znovuvytvořit UI vrstvu. Je na uživateli enginu, jak si ji implementuje; k dispozici má několik metod *BerylActivity* pro vytvoření, nastavení či získání aktuální UI vrstvy.

Jak již bylo zmíněno v podkapitole *Struktura projektu*, soubory obsahující geometrii byly vygenerovány mým vlastním externím programem, který je přiložen na optickém disku. Beryl používá vlastní binární formát pro geometrii, skládající se z hlavičky popisující formát vertexu, počet vertexů a indexů a dále obsahující samotná pole vertexů a indexů. Popis formátu vertexu je nutný pro fungování *GenericRenderEntity*. Můj výše zmíněný externí program, napsaný v jazyce C, umí překládat ze vstupního textového formátu do výstupního binárního formátu — vhodné pro jednoduchou geometrii, jako například čtverec. Dále je ho užito ke generování binárního výstupu bez textového vstupu — využito pro tvorbu texturovaného šestiúhelníku reprezentujícího herní políčko.

2.7 Umělá inteligence

2.7.1 Monte Carlo search tree

Nejdříve jsem zvolil *Monte Carlo Search Tree*, užívající *light play* a UCT variantu. Implementoval jsem ho jako běžící v samostatném vlákně, aby uživatel mohl nadále ovládat UI. Komunikace s hlavním vláknem, ve kterém běží herní logika, je zprostředkována použitím vzoru signály a sloty (více v kapitole o event systému). Hlavní vlákno pošle zprávu žádající výpočet tahu AI a obsahující současný herní stav. Jakmile je výpočet dokončen, je tah odeslán ve zprávě hlavnímu vláknu, které daný tah aplikuje na současný herní stav a překreslí obsah obrazovky.

Ovšem po naprogramování této umělé inteligence jsem zjistil, že na přijatelně kvalitní hru vyžaduje nepřijatelně mnoho času při každém tahu. Rozhodl jsem se proto implementovat Monte Carlo Search Tree algoritmus znovu, tentokrát vícevláknový.

Výsledná AI používá 9 vláken:

- 1x Řídící vlákno — generuje požadavky na *light play* simulace
- 8x Výpočetní vlákno — zpracovává požadavky na simulace

Řídící vlákno prochází strom herních stavů, vytváří nové uzly a zadává požadavky na simulace do fronty. Následně jsou probuzena výpočetní vlákna, která požadavky odebírají, vypočítávají simulace (přičemž ukládají výsledky do stromu herních stavů) a pokud narazí na prázdnou frontu, uspí se.

Praktická zkušenost po implementaci ukazuje, že řídící vlákno téměř všechen čas běhu spí, jeho práce je nenáročná v porovnání s výpočetními vlákny. Řídící vlákno se uspí, pokud je fronta plná. Také se uspí, pokud mu absence výsledků zatím nevypočítaných simulací nedovoluje dále pokračovat — četnost tohoto jevu se snižuje s rozvětvujícím se stromem herních stavů v průběhu algoritmu. Jinými slovy, čím déle algoritmus běží, tím lépe škáluje.

Vlákna existují po celou dobu běhu singleplayer režimu, avšak uspaná, pokud AI hráč není na tahu.

Počet právě osmi výpočetních vláken byl zvolen z důvodu častého použití právě osmijádrových procesorů v mobilních telefonech. Ostatní vlákna aplikace typicky spí, když se čeká na tah AI.

Užití *light play* způsobuje růst doby výpočtu tahu s rostoucím počtem figurek a rostoucí velikostí herní mapy. I přes výrazné zkrácení vypočetní doby je tento algoritmus stále příliš pomalý na herních mapách větších než nejmenší z nabídky. Proto jsem se rozhodl dále implementovat Minimax. MCST ve hře zůstává a uživatel si ho může zvolit z nabídky herních obtížnosti.

2.7.2 Minimax

Druhá implementace AI ve hře je Minimax, konkrétně varianta s alfa-beta prořezáváním. Stejně jako výše zmíněná implementace AI, i tato běží ve svém vlastním vlákně a nezdržuje tak hlavní vlákno. Algoritmus je velmi rychlý i na větších mapách a proto jsem se rozhodl ho implementovat pouze jako jednovláknový. V nabídce obtížnosti si uživatel může vybrat ze tří úrovní, liší se hloubkou procházeného stromu herních stavů. Tuto implementaci jsem kvůli její výpočetní efektivitě vybral ja hlavní a výchozí.

2.8 Hra

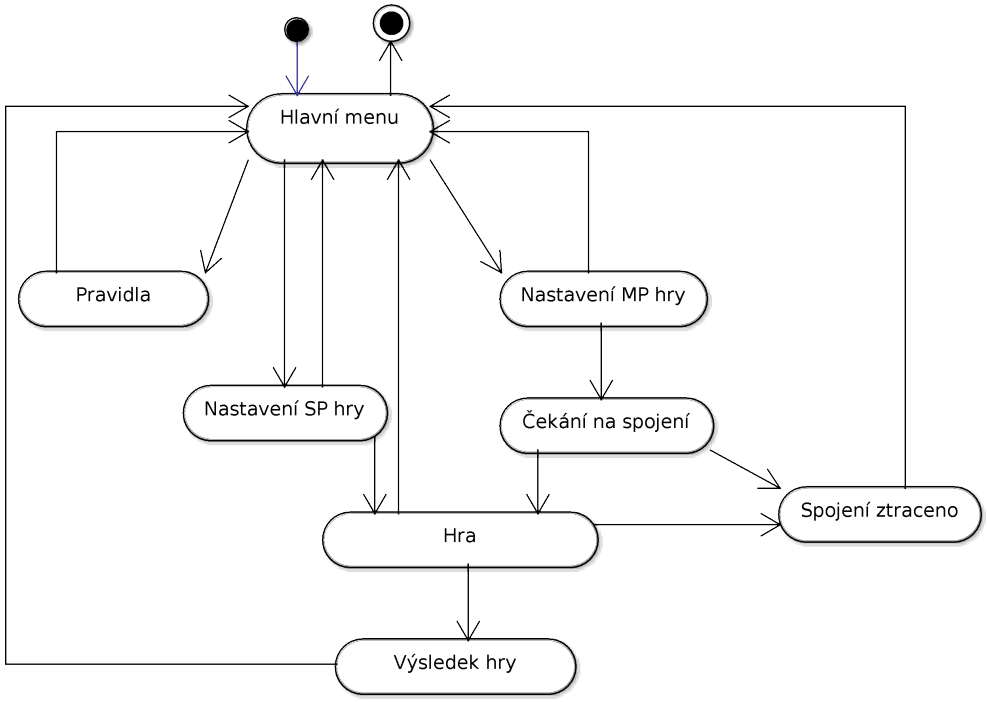
Hra je samozřejmě postavena na Berylu. Implementuje samotná pravidla hry, uživatelské rozhraní, protokol hry po síti, umělou inteligenci a další prvky neobsažené v Berylu. Její zdrojový kód zároveň slouží jako relativně přehledná ukázka použití tohoto enginu.

2.8.1 Pravidla hry

Pravidla hry jsou dostupná kliknutím na tlačítko *Rules* v hlavním menu. Ve stručnosti: jedná se o hru dvou týmu na šestiúhelníkovém poli, kde cílem je eliminovat všechny figurky protivníka.

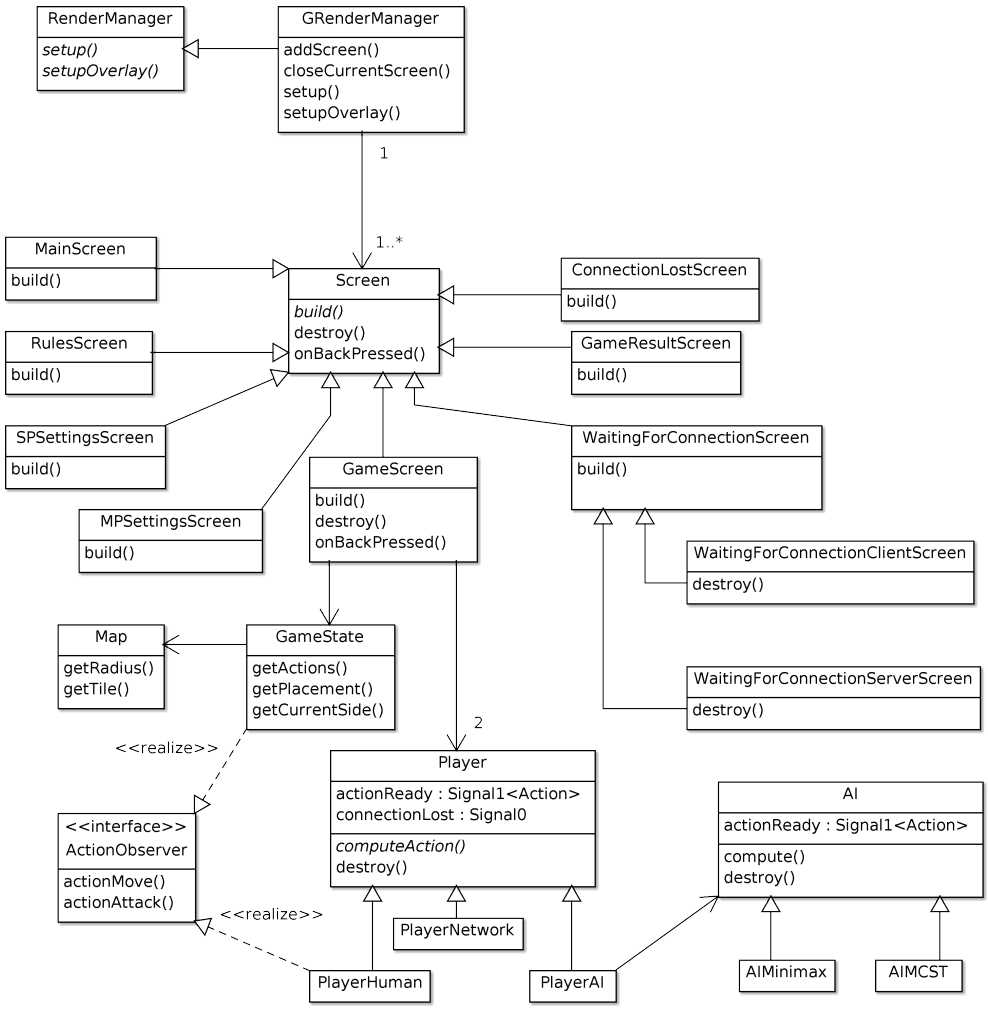
2.8.2 Uživatelské rozhraní

Po spuštění hry se otevře hlavní menu. Uživatel se naviguje do dalších menu pomocí tlačítek a vrací se stiskem tlačítka zpět.



Obrázek 2.7: Přechodový diagram mezi obrazovkami hry

Každé obrazovka se skládá z texturovaného pozadí (vykreslené pomocí OpenGL) a UI widget vrstvou na něm. Obrazovky jsou implementovány jako podtřídy třídy *Screen*. Ta má metodu *build*, zavolanou, kdykoliv je potřeba UI vrstvu vytvořit, a metodu *destroy*, zavolanou, když se daná obrazovka ukončuje. Třída *GRenderManager* hry rozšiřuje *RenderManager* poskytnutý Berylem o *LIFO* frontu obrazovek. Pomocí *RenderManager* tedy lze vložit (a tím zobrazit) novou obrazovku či odebrat obrazovku na vrcholu fronty (a tím zobrazit tu předchozí). Kdykoliv uživatel stiskne tlačítko zpět, je zavolána metoda *onBackPressed* u současné obrazovky, která pokud vrátí *false*, bude daná obrazovka odebrána. To umožňuje každé obrazovce si stisknutí tlačítka zpět implementovat po svém bez navrácení na předchozí obrazovku, pokud je to vhodné. Toho je užito u obrazovky s hracím polem, kde tlačítkem zpět může uživatel zrušit výber figurky.



Obrázek 2.8: Třídy obrazovek

Instance třídy *GameScreen* je obrazovkou samotné herní plochy.

Objekt třídy *GameState* drží herní stav, implementuje pravidla hry a poskytuje dotazovací rozhraní užitá zbytkem hry ke zjištění možných tahů a tak dále. Klíčová je metoda *clone*, která vytváří hlubokou kopii celého herního stavu (kromě přiřazené instance *Map*, která je neměnný objektem). Ta je velmi často volaná implementacemi umělé inteligence.

Třída *Player* reprezentuje jednoho hráče. Ve hře jsou vždy právě dva. Existují celkem tři podtřídy:

- *PlayerAI* — Hráč reprezentující umělou inteligenci. Konkrétní implementace AI lze snadno vyměnit za jinou — stačí, když je podtřídou třídy *AI*.
- *PlayerNetwork* — Hráč, jehož tahy jsou čteny ze sítě. Za použití *MPServerObserver* a *ServerNetworkManager*, respektive *MPClientObserver* a *ClientNetworkManager* je schopen komunikovat s instancí *PlayerNetwork* na druhé straně spojení.
- *PlayerHuman* — Reprezentuje místního hráče, vytváří grafickou reprezentaci herního stavu a zpracovává uživatelský vstup. Ze dvou instancí *Player* užívaných instancí *GameScreen* bude právě jedna *PlayerHuman*. Jinak by nebylo možné hru sledovat a ovládat.

Pomocné rozhraní *ActionObserver* zjednošuje aplikování tahů pomocí návrhového vzoru *visitor*.

2.9 Generovaní herní mapy

Herní mapa se skládá z šestiúhelníkových políček tří různých typů — poušť, les, skaliska. Typ políčka ovlivňuje možnosti figurek.

Beryl obsahuje implementaci dvourozměrného Perlinova šumu. Zvolil jsem tento algoritmu, jelikož je je velmi flexibilní a osvědčený. Generování herní mapy je provedeno převedením souřadnic každého políčka na souřadnice v procedurální textuře generované Perlinovým šumem s vhodně nastavenými konstantami. Výsledkem je skalární hodnota v rozsahu $\langle 0;1 \rangle$, která je mapována na typ políčka. Jedná se o novější verzi algoritmu Perlinova šumu od jeho původního autora[25], mnou upraven ze trojdimenzionální varianty na dvoudimenzionální. Jako prvek náhody při generování herní mapy slouží jeden celočíselný vstupní parametr (*seed*), který je použit pseudonáhodným generátorem čísel při vytváření permutace užité Perlinovým šumem. Permutace na základě daného *seed* je vytvořena algoritmem známým pod názvy „Fisher-Yates shuffle“ či „Knuth shuffle“.

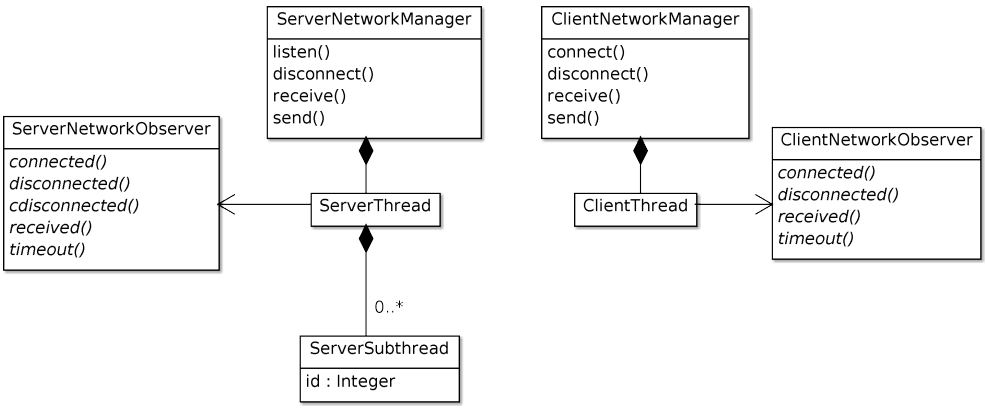
Ne všechna políčka mohou být proměnná. Hra zajišťuje, že počáteční pozice figurek budou vždy zcela průchodná a že mezi počátečními pozicemi obou

hráčů povedou alespoň tři zcela průchodné cesty — prostředkem mapy a po okrajích.

2.10 Multiplayer

Beryl poskytuje TCP implementaci jak serverové části (zvládající více klientů připojených najednou), tak i klientské části. Hra se sítí komunikuje pouze pomocí tříd *ClientNetworkManager* a *ServerNetworkManager*, které mají metody umožňující připojování, odpojování, posílání a přijímání dat. Tyto metody jsou asynchronní a přijímají argument určující timeouty. Různé události, jako například příchozí data, ztráta spojení či timeout dříve požadované operace, hra přijímá skrz své podtřídy tříd *ClientNetworkObserver* a *ServerNetworkObserver* implementující příslušné abstraktní metody.

Vnitřně jsou tyto asynchronní operace prováděny synchronně na samostatných vláknech; serverová implementace vytváří pro každé spojení nové vlákno. Tato vlákna komunikují s objekty popsanými výše pomocí event systému. Třída *SocketParser* obaluje standardní Java třídu Socket o metody *read* a *write* provádějící posílání/přijímání serializovaných objektů podtříd třídy *NetworkMessage* užitím objektu implementujícího rozhraní *NetworkMessageSerializer*. Jelikož TCP se chová jako proud bajtů a ne jako sekvence datagramů, je nutno přikládat informaci o délce každé zprávy. *SocketSerializer* nejdříve pomocí *NetworkMessageSerializer* serializuje objekt *NetworkMessage*, zjistí velikost výsledného pole bajtů a odešle tuto velikost jako 16-bitové číslo před samotným polem bajtů. *SocketSerializer* na druhé straně spojení počká až přijme daný počet bajtů a až pak se pokusí zprávu deserializovat do objektu *NetworkMessage* užitím *NetworkMessageSerializer*.



Obrázek 2.9: Třídy pro síťovou komunikaci v Berylu

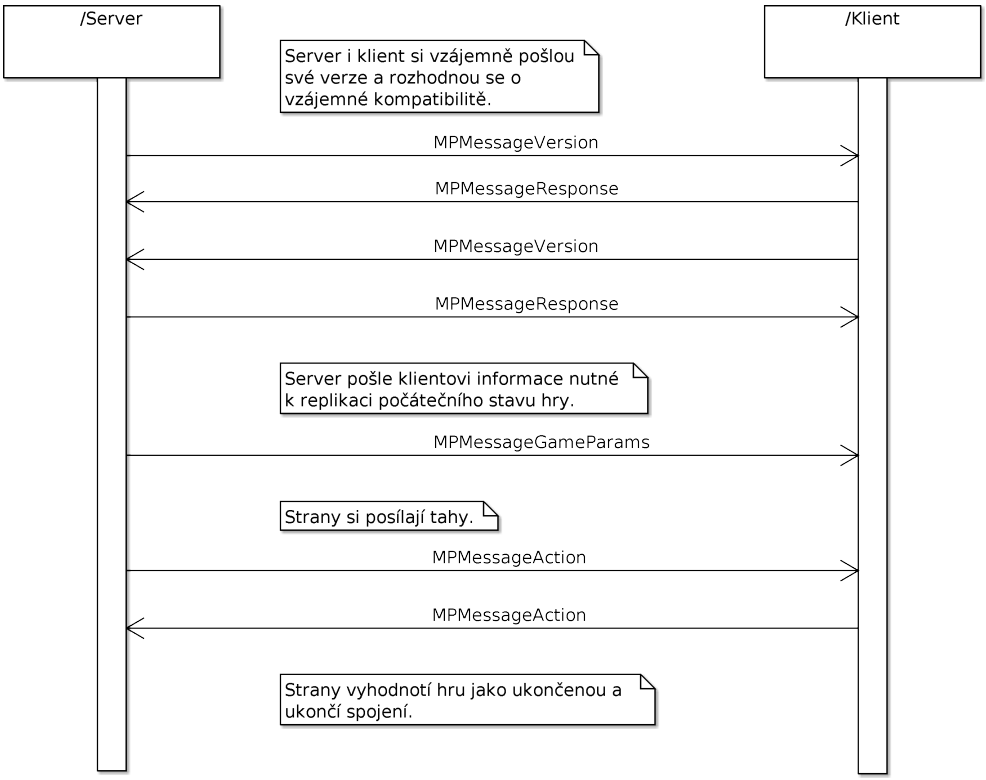
Hra využívá výše popsané rozhraní poskytnuté Berylem. Třída *MPMessage* implementuje rozhranní *NetworkMessage* a veškeré zprávy posílané hrou po síti jsou reprezentovány jako podtřídy *MPMessage*.

Jedná se o následující:

- *MPMessageResponse* — Popisuje odpověď během inicializace spojení. V současnosti jsou kódy odpovědi jen dva: úspěch a nekompatibilní verze.
- *MPMessageVersion* — Popisuje verzi hry odesílající strany.
- *MPMessageGameParams* — Obsahuje parametry hry odesílané serverem klientovi. Konkrétně se jedná o seed pro generaci herní mapy a velikost mapy.
- *MPMessageAction* — Popisuje tah.

Třída *MPMessageSerializer* implementuje rozhraní *NetworkMessageSerializer*, čímž poskytuje funkcionalitu serializace a deserializace zpráv hry ve formě podtříd *MPMessage* pro síťový kód Berylu.

Dále pak třída *MPServerObserver* rozšiřuje třídu Berylu *ServerNetworkObserver*, respektive *MPClientObserver* třídu *ClientNetworkObserver*. Implementují jejich abstraktní metody a tím umožňují hře reagovat na síťové události. Vnitřně obě fungují jako stavový automat. Při jakémkoliv narušení komunikace, ať už vypršením časového limitu či přijetím neočekávané zprávy, ukončují spojení.



Obrázek 2.10: Síťový protokol pro hru více hráčů

Časový limit pro síťovou komunikaci hra nastavuje na patnáct sekund, po jeho překročení se spojení považuje za ztracené.

Veškeré přijaté tahy každá strana kontroluje, není tedy možné posíláním nepovolených tahů podvádět. Strany mezi sebou nijak nesynchronizují herní stav — jen si pošlou počáteční situaci a následně si posílají tahy až do ukončení hry, které obě strany vyhodnotí nezávisle na sobě.

2.11 Testování

Testování kódu se skládá z jednotkových testů, instrumentation testů a detekce úniků paměti. Jednotkové a instrumentation testy užívají knihovnu *JUnit* verze 4.12, jak v Berylu tak ve hře samotné. Možnosti *JUnit* rozšiřuje knihovna Hamcrest[26] verze 1.3. Úniky paměti jsou detekovány knihovnou *LeakCanary*[27] verze 1.6.3 — ty lze zjistit jen ze spuštěné aplikace, proto je komunikace s touto knihovnou implementována v modulu hry. Uživatelské rozhraní a vykreslování OpenGL grafiky (které bohužel tvoří velkou část kódu) automaticky testované není.

Pro provedení automatických testu stačí otevřít projekt v programu Android Studio a vybrat jednu ze čtyř mnou definovaných množin testů: *AllUnitTestsBeryl* pro jednotkové testy enginu, *AllUnitTestsApp* pro jednotkové testy hry, *AllInstruTestsBeryl* pro instrumentation testy enginu a *AllInstruTestsApp* pro instrumentation testy hry. Jednotkové testy mohou běžet lokálně, protože závisí jen na standardním *Java* prostředí, *JUnit* a *Hamcrest*. Instrumentation testy vyžadují zařízení s OS Android, jelikož využívají funkcionalit knihoven OS Android. To lze vyřešit připojením zařízení s Androidem pomocí USB či emulátorem (zabudován v Android Studio).

2.12 Dokumentace

Zdrojový kód enginu i hry je označkován dokumentací ve formátu Javadoc[28]. To umožňuje vývojovým prostředím zobrazovat dokumentaci přímo při psaní kódu a zároveň vygenerovat dokumentaci pro zobrazení ve webovém prohlížeci — ta je k dispozici na přiloženém disku.

2.13 Instalace

Na přiloženém optickém disku je APK soubor. Stačí ho zkopírovat do zařízení s OS Android, otevřít ho a potvrdit instalaci. V závislosti na nastavení zařízení bude možná nutné v nastavení systému povolit instalaci z neznámých zdrojů.

Alternativně lze instalovat propojením zařízení s počítačem pomocí USB, otevřením projektu v programu Android Studio a použít jeho funkci pro instalaci na zařízení. To je též praktické při experimentování s úpravami kódu aplikace či enginu.

Kapitola 3

Nápady pro další rozvoj

3.1 Lepší grafika

V současnosti jsou možnosti grafických efektů v Berylu značně omezené. Pokud by uživatel enginu chtěl vytvořit hru pro komerční účely, bylo by na místě tyto možnosti rozvinout. Především funkcionality animací a částicových efektů. Užitečným rozšířením by bylo přidání nového typu zdroje pro text renderovaný do OpenGL textury — to by umožnilo vkládat text na úrovni objektů scény, oproti současnému vykreslování textu na úrovni vrsty Android UI prvků.

3.2 Jiné možnosti multiplayer spojení

Ne vždy je připojení přes jednu podsíť možné či žádoucí. Pro lepší místní spojení bez použití sítě by bylo vhodné přidat podporu Bluetooth. Pro lepší celosvětové spojení by bylo vhodné implementovat dedikovaný server na veřejném serveru spojující hráče i z jiných podsítí.

3.3 Statistiky

Někteří hráči by jistě ocenili možnost vidět přehled svých schopností proti AI i jiným hráčům ve formě statistického zpracování odehraných her.

3.4 Prohloubení herních mechanismů

Hra je v současném stavu dosti jednoduchá a jednoznačně by jí prospěla složitější pravidla.

3.5 Podpora jiných formátů modelů

Herní engine v současnosti podporuje pouze svůj vlastní formát. U 2D enginu není typicky potřeba mít modely se složitou geometrií, avšak podpora formátů používaných populárními programy pro modelování by bylo jistě užitečné.

3.6 Podpora zvuku

Herní engine Beryl nepodporuje zvuk. Pravděpodobnou implementací by bylo vytvoření nové podtřídy *GameManager* pro zvukový podsystém a nový typ zdroje pro reprezentaci jednotlivých zvuků.

Závěr

Cílem práce byl návrh a implementace hry žánru tahové strategie pro OS Android na bázi vlastního herního enginu s podporou: umělé inteligence, hry po síti, generování herních úrovní. Tyto cíle byly splněny.

Kromě samotné hry je užitečným výstupem práce i herní engine samotný, ve formě Android modulu. Byl navržen pro užití i mimo žánr tahových strategií. Umožňuje tvůrci hry soustředit se na implementaci herních principů samotných, nemusí se tak vůbec starat o nízkoúrovňové záležitosti jako například správa OpenGL bufferů. Oproti jiným herním enginům pro Android jsou jeho integrace do aplikace hry a jeho použití velice přímočaré.

Literatura

[1] Optional — Android Developers. leden 2019, [cit. 2019-01-25]. Dostupné z: https://developer.android.com/reference/java/util/Optional

[2] Android 7.0 for Developers — Android Developers. leden 2019, [cit. 2019-01-25]. Dostupné z: https://developer.android.com/about/versions/nougat/android-7.0

[3] OpenGL© ES Common Profile Specification 2.0.25 (Difference Specification). 2008, [cit. 2019-01-25]. Dostupné z: https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_cm_spec_2.0.pdf

[4] Distribution dashboard — Android Developers. leden 2019, [cit. 2019-01-25]. Dostupné z: https://developer.android.com/about/dashboards/

[5] Mike McShaffry, D. G.: *Game Coding Complete*. Cengage Learning PTR, čtvrté vydání, ISBN 978-1-133-77657-4, strana 307.

[6] Mike McShaffry, D. G.: *Game Coding Complete*. Cengage Learning PTR, čtvrté vydání, ISBN 978-1-133-77657-4, strana 314.

[7] Signals & Slots. [cit. 2019-01-25]. Dostupné z: http://doc.qt.io/qt-5/signalsandslots.html

[8] Gregor, D.; Hess, F. M.: Chapter 36. Boost.Signals2 - 1.69. červen 2007, [cit. 2019-01-25]. Dostupné z: https://www.boost.org/doc/libs/1_69_0/doc/html/signals2.html

[9] Gregory, J.: *Game Engine Architecture*. A K Peters/CRC Press, druhé vydání, ISBN 978-1466560017, strana 319.

[10] OGRE: Resource Management. 2015, [cit. 2019-01-25]. Dostupné z: https://ogrecave.github.io/ogre/api/1.11/_resource-_management.html

[11] Michael Bailey, S. C.: *Graphics Shaders: Theory and Practice*. A K Peters/CRC Press, druhé vydání, 2011, ISBN 978-1568814346, strany 42-49.

[12] Colledanchise, M.; Ögren, P.: Behavior Trees in Robotics and AI: An Introduction. *CoRR*, ročník abs/1709.00084, 2017, strana 13, 1709.00084. Dostupné z: http://arxiv.org/abs/1709.00084

[13] McQuillan, K.: A Survey of Behaviour Trees and their Applications for Game AI. [cit. 2019-01-25]. Dostupné z: http://www.academia.edu/33601149/A_Survey_of_Behaviour_Trees_and_their_Applications_for_Game_AI_A_Survey_of_Behaviour_Trees_and_their_Applications_for_Game_AI

[14] Kocsis, L.; Szepesvári, C.: Bandit based Monte-Carlo Planning. In *In: ECML-06. Number 4212 in LNCS*, Springer, 2006, s. 282–293. Dostupné z: https://www.lri.fr/~sebag/Examens_2008/UCT_ecml06.pdf

[15] Abraham, S.: Noise Functions. [cit. 2019-01-25]. Dostupné z: http://www.cs.utexas.edu/~theshark/courses/cs354/lectures/cs354-21.pdf

[16] Download Android Studio and SDK tools — Android Developers. [cit. 2019-01-25]. Dostupné z: https://developer.android.com/studio/

[17] Gradle Build Tool. [cit. 2019-01-25]. Dostupné z: https://gradle.org/

[18] Git. [cit. 2019-01-25]. Dostupné z: https://git-scm.com/

[19] ArgoUML. [cit. 2019-01-25]. Dostupné z: http://argouml.tigris.org/

[20] GIMP - GNU Image Manipulation Program. [cit. 2019-01-25]. Dostupné z: https://www.gimp.org/

[21] Draw Freely — Inkscape. [cit. 2019-01-25]. Dostupné z: https://inkscape.org/

[22] Message — Android Developers. leden 2019, [cit. 2019-01-25]. Dostupné z: https://developer.android.com/reference/android/os/Message.html

[23] Silisteanu, P.: OpenGL 101: Matrices - projection, view, model. květen 2013, [cit. 2019-01-25]. Dostupné z: https://solarianprogrammer.com/2013/05/22/opengl-101-matrices-projection-view-model/

[24] Brothaler, K.: *OpenGL ES 2 for Android: A Quick-Start Guide*. Pragmatic Bookshelf, první vydání, ISBN 978-1937785345, strana 11.

[25] Perlin, K.: Improved Noise reference implementation. [cit. 2019-01-25]. Dostupné z: https://mrl.nyu.edu/~perlin/noise/

[26] Java Hamcrest. [cit. 2019-01-25]. Dostupné z: http://hamcrest.org/JavaHamcrest/

[27] LeakCanary. [cit. 2019-01-25]. Dostupné z: https://github.com/square/leakcanary

[28] Javadoc. [cit. 2019-01-25]. Dostupné z: https://docs.oracle.com/javase/9/javadoc/javadoc.htm

Příloha A

Seznam použitých zkratek

OS Operační systém

API Application programming interface

LRU Least-recently used

UI User interface

HW Hardware

FPS First-person shooter

UCT Upper confidence bound 1 applied to trees

UML Unified modeling language

GPU Graphics processing unit

LIFO Last in, first out

MCST Monte Carlo search tree

TCP Transmission control protocol

AI Artificial intelligence

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `game.apk` ..... instalační balíček pro OS Android
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `ext_impl` ..... zdrojové kódy externích programů
    - `mesh_compiler` ..... generátor souborů geometrie
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `doc` ..... Javadoc dokumentace
  - `beryl` ..... Dokumentace herního enginu
  - `game` ..... Dokumentace hry
- `text` ..... text práce
  - `assignment.pdf` ..... text zadání práce ve formátu PDF
  - `thesis.pdf` ..... text práce ve formátu PDF