



MATEMATICKO-FYZIKÁLNÍ FAKULTA Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Pavel Halbich

Tau Ceti f 2 – budovatelská počítačová hra se strategickými prvky

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2017

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) 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 má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V dne

Podpis autora

Děkuji mému vedoucímu Pavlu Ježkovi za pomoc s touto prací, mým rodičům za podporu a pevné nervy, mé přítelkyni Veronice takéž za podporu a pomoc s 2D grafikou a Jiřímu Kurčíkovi za laskavé poskytnutí práv na použití jeho hudební tvorby v mé hře.

Název práce: Tau Ceti f 2 – budovatelská počítačová hra se strategickými prvky

Autor: Pavel Halbich

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: Tau Ceti f 2 – A Creative Computer Game with Strategic Elements

Author: Pavel Halbich

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Abstract.

Keywords: key words

Obsah

1 Úvod	4
1.1 Charakteristika her	4
1.1.1 Stylizované hry	4
1.1.2 Hry s prvky realismu	6
1.1.3 Hry s maximálním důrazem na simulaci reality	9
1.1.4 Možné další inspirace	10
1.2 Čemu se budeme věnovat	11
1.3 Herní bloky	11
1.4 Inventář	12
1.5 Cíle práce	12
2 Analýza zadání	13
2.1 Stávající implementace mechanismů	13
2.1.1 Bloky	13
2.1.2 Komunikace bloků	15
2.1.3 Skládání bloků do struktur	15
2.1.4 Herní svět	16
2.1.5 Inventář	18
2.1.6 Herní postava	19
2.2 Co bychom chtěli implementovat	19
2.2.1 Bloky a herní svět	19
2.2.2 Podrobný popis bloků	21
2.2.3 Komunikace bloků	23
2.2.4 Zdraví bloků	24
2.2.5 Skládání bloků do struktur	24
2.2.6 Herní svět	24
2.2.7 Inventář	25
2.2.8 Herní postava	25
2.3 Shrnutí cílů	26
3 Detailní analýza	27
3.1 Herní engine	27
3.1.1 Vlastní engine	28
3.1.2 Vlastní engine s použitím již existujících grafických knihoven	28
3.1.3 Existující herní engine	28
3.1.4 Volba engine -verdikt	29
3.2 Struktura projektu	30
3.3 Bloky	30
3.3.1 Celková struktura	30
3.3.2 Textové soubory	31
3.4 Vlastnosti bloků	32
3.4.1 Energie	32
3.4.2 Energetická síť	32
3.4.3 Kyslík	32
3.4.4 Označovatelnost	32

3.4.5	Možnost vzít do inventáře	32
3.4.6	Interakce	32
3.4.7	Zapojení do rozpoznávání tvarů	33
3.5	Komponenty bloků	33
3.6	Blok v herním světě	33
3.7	Konstruktor objektů	33
3.8	Počasí	33
3.9	Hráčova postava	33
3.10	Inventář	34
3.11	Ukládání hry	34
3.12	Doplňující vlastnosti	34
3.12.1	Lokalizace	34
3.12.2	Hudba	34
3.13	Backlog	34
4	Programátorská dokumentace	35
4.1	Počáteční inicializace projektu	35
4.2	Struktura projektu	37
4.3	Struktura projektu v Unreal Enginu	37
4.3.1	Mapy	37
4.4	Struktura kódu	37
4.4.1	Struktura modulu	38
4.5	Modul Commons (C++)	38
4.5.1	Herní definice a konstanty	39
4.5.2	Herní instance	39
4.5.3	Enumerátory	40
4.5.4	Helpery	40
4.6	Modul Game Save (C++)	40
4.6.1	GameSaveInterface	40
4.6.2	FFileVisitor	41
4.6.3	Helpers	41
4.6.4	Kontejner s uloženou hrou	41
4.6.5	NewGameSaveHolder	43
4.7	Modul Blocks (C++)	43
4.7.1	Definice bloků	44
4.7.2	Třídy s popisem bloků	44
4.7.3	Ukládání a načítání bloků	44
4.7.4	Interfaces	44
4.7.5	Komponenty bloků	44
4.7.6	Implementace bloků	45
4.7.7	Stromové struktury	45
4.8	Modul Inventory (C++)	45
4.8.1	Tag group	45
4.8.2	Inventory tag group	45
4.8.3	Inventory tags	46
4.8.4	Inventory component	46
4.9	Modul TauCetiF2 (C++)	46
4.10	Backlog	46

5	Uživatelská dokumentace	47
5.1	Požadavky pro spuštění hry	47
6	Závěr	48
6.1	Zhodnocení práce	48
6.2	Zhodnocení dotazníku	48
6.3	Budoucí práce	48
	Seznam použité literatury	49
	Přílohy	52

1. Úvod

V době vzniku této práce jsou velice populární hry s otevřeným světem. Lákají hráče na obsáhlou světa a možnost nelineárního řešení problémů a herních úkolů. Her s otevřeným světem najdeme nepřeberné množství v různých herních žánrech. My se zaměříme na podmnožinu her, které kromě otevřeného světa nabízí také možnosti budování struktur a vyžadují od hráče netriviální styl hraní, který mu umožňuje ve hře přežít. V herním průmyslu se tyto hry často označují jako *sandboxové*, *s budováním*, *s průzkumem prostředí*, *o přežití*. Autor této práce má tento typ her v oblibě a rád by touto prací představil svoji vizi dalšího možného rozvoje her tohoto žánru. Cílem práce by měla být implementace nového herního principu stavění, které současné herní tituly nenabízí.

1.1 Charakteristika her

V práci se budeme zabývat několika různými hrami, které však mají několik společných vlastností. Jedním ze základních konceptů je využívání herních bloků. Dalším význačným prvkem je způsob integrace herních bloků do herního prostředí. Některé hry jsou celé tvořeny bloky, jiné se snaží dosáhnout vyššího stupně realismu ve hře a bloky využívají pouze pro konstrukci různých herních objektů. Důležitým tématem této práce tedy bude rozbor systému bloků a práce s nimi a popis hráčských problémů způsobených danými koncepty. V další části práce pak navrhнемe a implementujeme vlastní řešení.

1.1.1 Stylizované hry

Začneme hrami, které využívají bloků jako základního elementu celé hry. Bloky zde tvoří doslova celý svět. Mezi nejpopulárnější a širokou veřejností nejznámější bychom měli zařadit hru *Minecraft*. Na obrázku 1.1 z této hry si můžeme všimnout několika zásadních faktů. Vidíme zde kostičkované listí stromů (1) či hrad na skále (2), který byl postaven z kostiček. Taktéž slunce, měsíc a mraky (3) jsou stylizovány do kostiček. Výrazně je kostičkovaný styl vidět na nehratelných postavách (*non-playable character – NPC*) – na obrázku ovce (4), krávy a prasata. Stejným způsobem je pak zpracována i hráčova postava (5), kterou hráč přímo ovládá.



Obrázek 1.1: Hra Minecraft – hrad na skále

Ve hře *Minecraft* může hráč bloky umisťovat do herního světa. Bloky získává těžbou (například kutáním krumpáčem do kamenného bloku), nebo je může vyrobit tzv. *craftingem*. *Crafting* probíhá skrze uživatelské rozhraní, kdy hráč z nějaké kombinace herních bloků či obecně elementů vytváří nové bloky či elementy. *Minecraft* používá takový systém *craftingu*, kdy hráč musí umístit bloky do konkrétního tvaru, aby získal nějaký nový objekt.

Na obrázku 1.2 můžeme vidět dvě varianty tvorby krumpáče. V jednom případě je hráčovým cílem vytvořit *dřevěný krumpáč* (na obrázku vlevo), v druhém případě pak *kamenný krumpáč* (vpravo). Z toho důvodu v prvním případě do vstupního tvaru umístí do horní části 3 bloky dřevěných prken, v druhém případě použije 3 bloky kamení. Tímto způsobem je možné vytvářet nejen nástroje, ale i zbraně, nové bloky a dokonce i jídlo, které pak hráč ve hře konzumuje.



Obrázek 1.2: Hra Minecraft – Crafting

Stavění pak ve hře probíhá tak, že si hráč zvolí blok nebo objekt, který chce umístit do světa. Tento objekt musí mít ve svém inventáři. Namíří kurzor na už existující blok ve světě a klikne levým tlačítkem myši. Ke straně bloku, na který hráč mířil, se pak připojí vybraný blok. Postavený blok se odebere z inventáře a dále nevyžaduje žádnou další akci (na rozdíl od jiných her, což si ukážeme v následující sekci).

Mezi dalšími hrami bychom mohli zmínit například hru *Terraria*. Ta je o něco

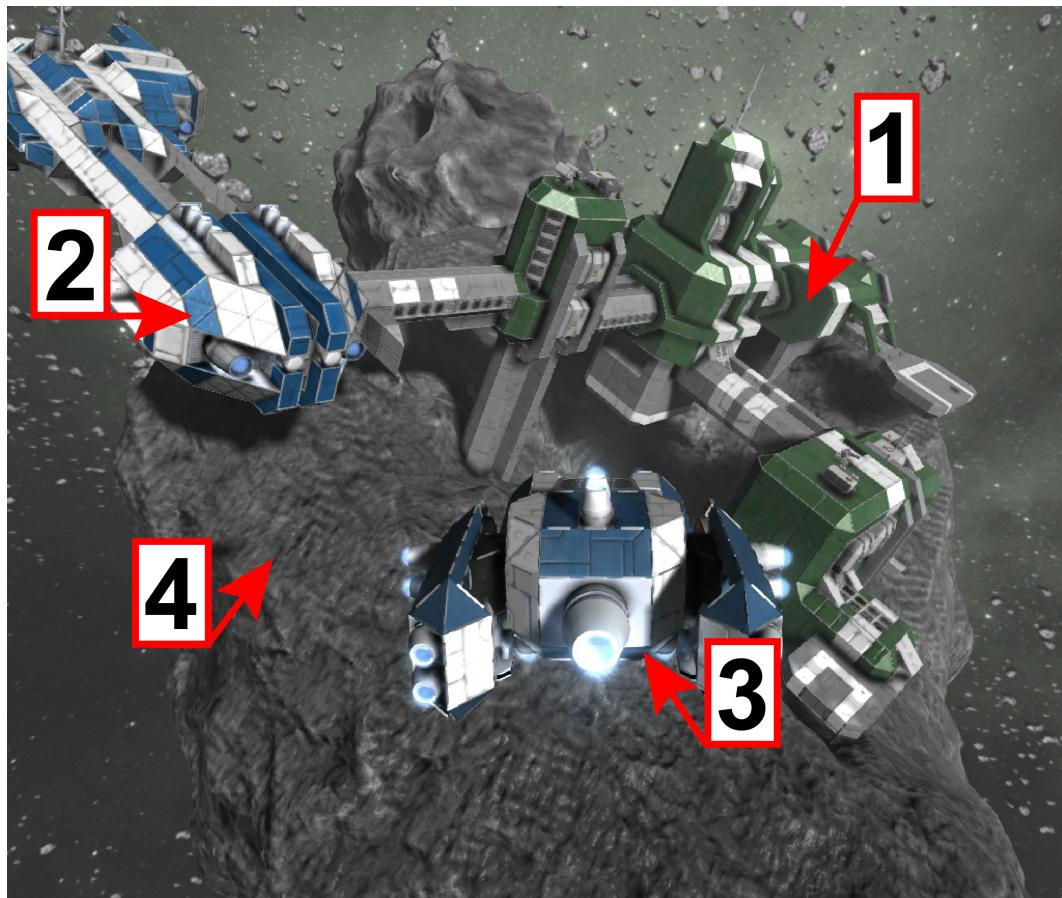
mladší než *Minecraft*, ale je častým zdrojem diskusí, zda je lepší než *Minecraft*, nebo ne. Pravdou je, že obě hry mají svůj svět kompletně složený z kostek (*Terraria* je však 2D hra), ale každá si klade trochu jiné cíle. *Terraria* je více orientovaná na příběh, obsahuje více *NPC* i *bossů*¹. *Minecraft* je pak orientován spíše na stavění (porovnání *Minecraft vs Terraria (facts)* [2] na Minecraftovém fóru), ačkoliv od *Combat update*, tedy verze 1.9, jsou možnosti boje oproti dřívějším verzím větší (stránka s popisem aktualizace hry [3]).

1.1.2 Hry s prvky realismu

Mezi hry s prvky realismu bychom mohli zařadit třeba hry *Space Engineers* či *Medieval Engineers*, využívající kombinaci herních bloků s *voxelovou* reprezentací světa (voxely si můžeme představit jako bloky stejné velikosti). Obě hry jsou implementovány v proprietárním enginu společnosti Keen Software House nazvaném *VRAGE™*. Z voxelové reprezentace terénu je pak v enginu za běhu hry procedurálně generovaná polygonální reprezentace terénu, kterou pak grafická karta standardním způsobem vykreslí na obrazovce (oficiální popis vlastností enginu [4]). Obdobným způsobem jako terén se pak ve hře *Space Engineers* chovají různá vesmírná tělesa či asteroidy. Během procedurálního vytváření asteroidu je na třídimenzionální strukturu voxelů aplikován šum a tím je možné ve hře vygenerovat prakticky neomezené množství různých asteroidů vycházejících z jedné voxelové struktury. „*The “procedural asteroids” feature adds a practically infinite number of asteroids to the game world*“ [5]. Tímto způsobem pak hry dosahují vyššího stupně realismu – nespoléhají se pouze na předpřipravené 3D modely, ale generují vizuální reprezentaci herních objektů za běhu hry. Z toho vyplývá, že každý hráč může stejnou hru prožít jinak.

Podívejme se na obrázek 1.3 ze hry *Space Engineers*. Na něm můžeme vidět převážně kamenný asteroid a na něm je postavená vesmírná základna (1). K základně je přistavena větší vesmírná loď (2), hráč pak k základně letí v další, malé lodi (3). Můžeme si všimnout, že povrch asteroidu (4) není pravidelný a obsahuje spoustu nerovností (na rozdíl od tvaru základny a lodí). To je způsobeno právě algoritmickou approximací voxelové reprezentace asteroidu.

¹*Boss* je v herní terminologii významný nepřítel, obvykle je silnější než ostatní protivníci a velmi často bývá v závěrečných částech hry. Duel s *bossem* pak obvykle od hráče vyžaduje zjištění jeho silných a slabých stránek a schémat jeho útoků [1].

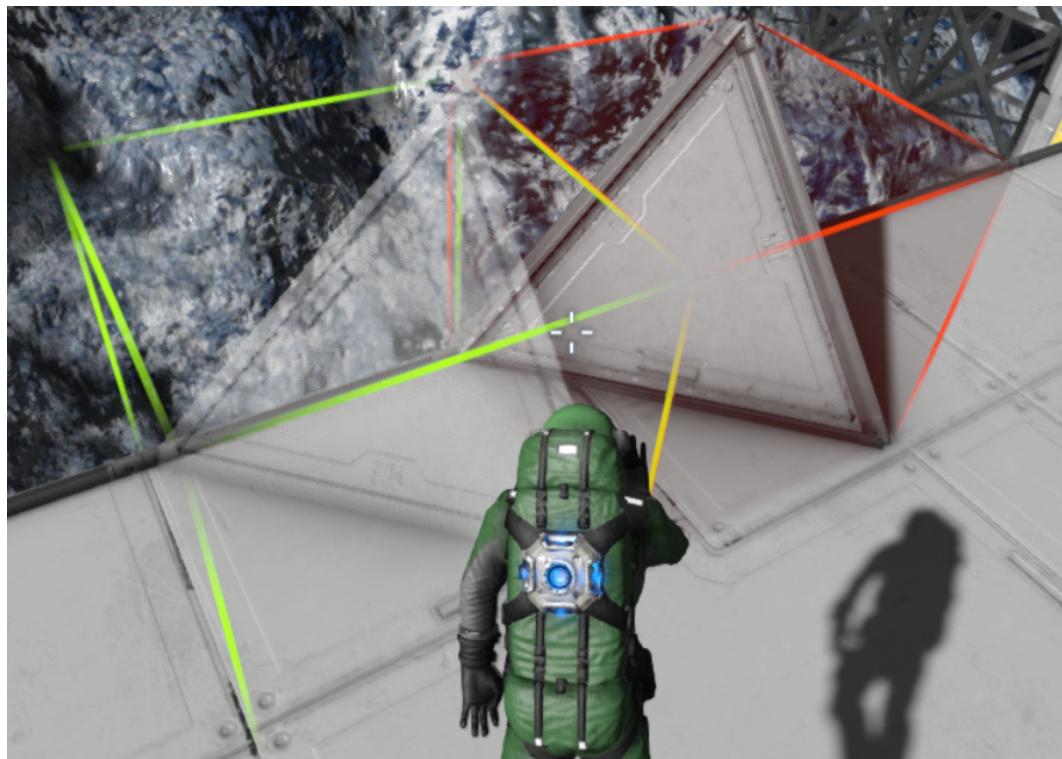


Obrázek 1.3: Hra Space Engineers – základna. Zdroj: Gamespot.com [6]

Samotná základna i vesmírná plavidla (detailní pohled na jiné plavidlo je na obrázku 1.4) jsou tvořeny bloky. Vizuální reprezentace bloku může být i jiného tvaru než jen krychle – to je možné vidět na obrázku 1.5. Na stejném obrázku můžeme vidět barevně zvýrazněné hranice bloků. Jak základny, tak vesmírné lodě (které jsou navíc oproti základnám pohyblivé) využívají tento systém bloků.



Obrázek 1.4: Hra Space Engineers – dron. Zdroj: space-engineer.net [7]



Obrázek 1.5: Hra Space Engineers – bloky

Space Engineers umožňuje stavět pohyblivé stroje, které si hráč postaví z herních bloků, jež se pak dohromady chovají jako jedna entita. Stále je na ně však aplikována fyzika, takže je možné plavidlo poškodit, nebo dokonce zničit. Obdobně lze ve hře *Space Engineers* stavět různá mechanická obléhací zařízení, například pojízdné katapulty.

Crafting v podobě, ve které je použit ve hře *Minecraft*, se ve hře *Space Engineers* neobjevuje, ale můžeme zde nalézt obdobný systém. Hráč musí těžit rudy, které pak dává do specializovaného stroje (bloku) na zpracování. V ovládacím rozhraní tohoto bloku si pak vybere, kterou součástku chce vytvořit. Tyto součástky jsou pak použity ke stavbě dalších bloků (zde je zásadní rozdíl se hrou *Minecraft*).

Stavění je ve hře *Space Engineers* vyřešeno tak, že si hráč vybere z nabídky všech dostupných bloků nějaký blok, který chce postavit. Tento stavěný blok můžeme chápát spíše jako návod, jak jej postavit, než jako fyzické umisťování bloku do herního světa (případ *Minecraftu*). Po umístění vybraného bloku do světa hráč uvidí pouze základní konstrukci vycházející z tvaru bloku. Za použití specializovaného nástroje a součástek ze svého inventáře pak po nějakou dobu „staví“ daný blok, přičemž v průběhu tohoto stavění se spotřebují potřebné součástky z hráčova inventáře a zároveň se po nějakých krocích mění vizuální podoba stavěného bloku. Můžeme tedy říct, že *crafting* se ve hře *Space Engineers* neobjevuje v podobě tvorby specifických tvarů v uživatelském rozhraní a následném získání předmětu, nýbrž v podobě získávání prostředků pro tvorbu součástek, které jsou vyžadovány pro další stavbu.

Hra *Medieval Engineers* implementuje systém stavění na pomezí her *Minecraft* a *Space Engineers*. Hráč nemá na začátku hry k dispozici všechny bloky, ale může

si postavit blok *Výzkumného stolu*, ve kterém pak za nějakou cenu může znalost konstrukce nového bloku vyzkoumat. Z výzkumu vzejde svitek, který si hráč může přečíst a tím se naučí stavět nový blok. Zajímavou vlastností hry je, že na multiplayerovém serveru je možné tyto svitky předávat dalším hráčům a ti se pak mohou stavbu nového bloku naučit také, aniž by museli sami zkoumat stavbu daných bloků. Stavba pak dále funguje obdobným způsobem jako u *Space Engineers* s tím rozdílem, že místo technických součástek se používají různé kamenné, dřevěné a jiné další herní objekty.

1.1.3 Hry s maximálním důrazem na simulaci reality

Mezi hry s maximálním důrazem na simulaci reality bychom mohli zařadit například vesmírný simulátor *Take on Mars*. Tato hra si klade za cíl se maximálně přiblížit zážitku, který by mohli astronauti zažít při cestách na Měsíc a na Mars. Bloky se zde objevují v podobě standardizovaných částí budov, ale třeba vozidla jsou kompletně vymodelována a hráč je nemůže stavět ani nijak modifikovat. Představu o hře si můžeme udělat z obrázku 1.6.



Obrázek 1.6: Hry Take On Mars – vozítka před budovou. Zdroj: Hry.cz [8]

Stavění ve hře *Take on Mars* probíhá tak, že hráč přidává bloky k už postaveným blokům na konkrétní, vývojáři definované přípojné body. To má výhodu v tom, že hra snadno pozná, že je stavba někde „děravá“. Pokud je někde ve struktuře stavby netěsnost, pak například neprobíhá okysličení a natlakování vnitřních prostor takovéto nedokončené stavby.

Hra dále nabízí možnost vytváření nových bloků a objektů dvěma způsoby. V rámci *kreativní* hry (tedy hry bez omezení) má hráč k dispozici speciální *3D tiskárnu*, která umí „vytisknout“ libovolný herní objekt (a to dokonce bez jakýchkoliv nákladů). Později do hry přibyl druhý způsob, použitelný v *běžné* hře – *konstruktör objektu*. Ten je složen z několika bloků a je možné díky němu „vytisknout“ třeba celé vozidlo. Velikost výsledných objektů je omezena velikostí

konstruktoru, tudíž není možné vytvářet objekty větší, než je velikost konstruktoru (bráno vnitřními rozměry).

Take on Mars taktéž používá systém těžby a zpracování nerostů a minerálů, které slouží jako zdroj surovin pro konstruktory objektů. Suroviny jsou plněny do speciálních kontejnerů, ze kterých mohou být opětovně spotřebovány na stavbu.

1.1.4 Možné další inspirace

Námi zmíněné hry rozhodně nejsou všechny dostupné hry. Stále vznikají nové hry a ty stávající se postupně upravují a vylepšují. Navíc mnohé z nich jsou v různých fázích vývoje (Alfa či Beta verze), a tak se situace velice rychle mění. Rádi bychom však zmínili ještě některé další hry, které nás sice zaujaly, ale kromě různých variant prostředí obvykle nepřináší žádnou novou a zásadní funkcionality.

Hra *Novus Inceptio* patří do žánru *MMORPG (massively multiplayer online role-playing game* – onlinová hra na hrdiny s velkým množstvím hráčů). Na této hře bychom rádi zmínili koncept stavění, který probíhá poměrně netradičně. Hráč se přepne do staviteckého módu a celou stavbu si naplánuje. V tomto plánovači vidí výsledek rovnou tak, jak bude budova po dokončení vypadat. Bloky se během stavění přichytávají do mřížky a k sobě. Po ukončení toho módu pak hráč vidí obrysů naplánovaných bloků. Pak je může začít konstruovat, přičemž v nabídce (během konstrukce) má možnost si chybějící součásti vytvořit pomocí *craftingu*. V tomto ohledu je to tedy podobné *Space Engineers*. Plánovač se nám sice líbí, ale zastáváme názor, že je to pouze pomůcka, bez které je možné se obejít. Ostatně bloky lze kupříkladu ve *Space Engineers* stavět i na nedokončených blocích a později, až je výsledná hrubá stavba hotová ke hráčově spokojenosti, je všechny dokončit specializovaným nástrojem.

Hra *Planet Nomads* funguje podobně jako *Space Engineers* – získávání surovin, jejich crafting na komponenty a používání komponent na konstrukci objektů. Nově postavený (nedokončený) objekt ukazuje základní konstrukci a komponenty nutné k dokončení. Stejně jako u *Space Engineers* je zde možné vytvářet pohyblivá vozidla, měnit terén apod. Z pohledu stavění tedy nepřináší nic nového. Hra navíc kromě velice hezké grafiky řeší vlastnosti herní postavy – pokud hráč nějakou činnost vykonává opakováně, tak se v ní zlepšuje a naopak – a tím se výrazně odlišuje od jiných her.

Hra *ARK Survival Evolved* je pak opět podobnou kombinací předchozích her a liší se jiným prostředí. Taktéž využívá při stavění systém přípojných bodů.

Hra *No man's sky* je opět podobná předchozím, stavění využívá systém přípojných bodů. Hra má hezký řešený systém multibloků, kdy je možné upravit některé části staveb. Takže například místo zdi je možné vytvořit okno. Svojí funkcionality je tedy podobná hře *Space Engineers*.

Abychom výše uvedeným hrám v této kapitole nekřivdili, musíme zmínit fakt, že hry nejsou úplně stejné. Mnohé využívají třeba systém procedurálně generovaných planet a každá hra se snaží v něčem odlišit. Nicméně my se tímto aspektem (procedurálním generováním) zabývat nebudem a tedy pro účely naší práce můžeme výše zmíněné hry považovat za podobné.

1.2 Čemu se budeme věnovat

V této práci bychom rádi navrhli a posléze implementovali nové přístupy a herní mechaniky, které by měly usnadnit stavění ve staviteckých hrách. Rádi bychom zachovali koncept použití herních bloků, který shledáváme jednoduchý na pochopení i použití. Zaměříme se na rozšíření možnosti práce s bloky tak, aby bychom hráči nabídli, pokud možno, ještě lepší herní zážitek ze stavění vlastních výtvarů. V této práci se nebudeme nijak důkladně věnovat vizuální reprezentaci prostředí, protože ta pro nás v tuto chvíli není podstatná.

Změna v přístupu k herním blokům bude vyžadovat i úpravy herního mechanismu s tím souvisejícího – hráčova inventáře. Všechny výše zmíněné hry nějakým způsobem nabízí hráči výběr bloků, které může do herního světa umístit. Naše změna by bohužel znamenala, že by se takový inventář postavitelných bloků velmi rychle stal nepřehledným a proto musíme systém nabídky postavitelných bloků upravit pro naše potřeby.

1.3 Herní bloky

Obvykle je ve hře definován jeden základní rozměr bloku, který je neměnný². To však může být problémem, pokud se hráč rozhodne postavit v herním světě nějakou větší a komplexnější strukturu podle reálné či fiktivní předlohy. Pro příklad uvedeme některé výtvarny ze hry *Minecraft* – město Královo přístaviště z knih Píseň ledu a ohně od George R. R. Martina, nebo hlavní město Gondoru Minas Tirith z knih Pána prstenů od J. R. R. Tolkiena. Autoři těchto výtvarů museli volit takové měřítko, aby byly výtvarny dostatečně detailní, ale zároveň aby bylo možné výtvarny postavit v nějakém rozumném čase. Obecně můžeme říct, že čím větších detailů chtějí autoři ve hře *Minecraft* dosáhnout, tím větší musí celý výtvarny být. To pak ale znamená, že celá stavba trvá déle, nebo je zapotřebí více spolupracujících hráčů.

Hra *Space Engineers* a jí podobné díky svému přístupu a více bloků, které nejsou tvaru krychle, nabízí lepší možnosti staveb rozsáhlých objektů (představme si třeba Hvězdu smrti z Hvězných válek), ale stále je potřeba volit nějakou rozumnou výslednou velikost. Možnosti detailů se sice zvyšují s rostoucím počtem různých vizuálních variant bloků (což je třeba případ *Space Engineers* nebo *Novus Inceptio*), ale i tak bychom chtěli navrhnout nový systém, ve kterém lze používat bloky různých velikostí.

Náš návrh úpravy

Chtěli bychom se v této práci zabývat myšlenkou proměnlivé velikosti stavitelných bloků. Tím by hráči mohli rychleji stavět rozsáhlější struktury a přitom se věnovat i drobným či estetickým detailům. Tento návrh však s sebou nese několik problémů, které se v této práci budeme snažit vyřešit.

Ačkoliv se nám idea postupné stavby bloku ze součástek velice líbí, od naší hry tuto funkcionality nebudeme požadovat. Znamenalo by to pro nás, že bychom museli řešit vizuální podobu rozpracovaných bloků, navíc v kombinaci s různými

²*Space Engineers* definuje více velikostí – ty však nelze vzájemně kombinovat

rozměry. Pak bychom nejspíše museli generovat tyto objekty procedurálně, přičemž realizace této vlastnosti by nám zabrala velké množství času. Implementaci této funkcionality tedy vidíme (v současné době) jako zbytečnou, protože nevíme, zda vůbec bude myšlenka proměnlivé velikosti bloků přijata hráči.

1.4 Inventář

Dalším společným prvkem stavitelských her je inventář bloků, které může hráč umístit do herního světa. Hráč vidí přes celé herní okno *HUD* (Head-Up Display [9]), ve kterém má zobrazenou kromě jiného nabídku bloků, které má na rychlé volbě, může je snadno zvolit a daný blok umístit do herního světa. Navíc hry mohou definovat i inventární skupiny bloků (*Space Engineers*, *Medieval Engineers*), mezi kterými hráč může přepínat a tím rychle kompletně změnit sadu rychlé nabídky. Vidíme však limitaci v tom, že hráč musí ručně spravovat tyto seznamy a jednotlivé bloky (či nástroje) ručně umisťovat do příslušných pozic, například pomocí systému Drag and Drop (tedy přetáhnutí bloku myší z nabídky inventáře do rychlé volby stavitelných bloků).

Náš návrh úpravy

Rádi bychom navrhli jiný způsob správy těchto inventárních skupin tak, aby hráč jednou definoval, jaké prvky chce mít v příslušných skupinách. Při vytvoření nového bloku či vytvoření jiné velikosti bloku by pak nemusel ručně přiřazovat nový blok do skupiny, ale tento blok by měl být automaticky zařazen a nabídnut hráči.

1.5 Cíle práce

Tato práce bude mít dvě fáze. V první fázi provedeme analýzu současného stavu a možných řešení, ze které nám vzejdou podrobnější cíle práce. Druhá fáze práce pak bude implementace hry jako takové. Naše cíle tedy jsou:

1. Analytická část
 - Navrhnout způsob řešení proměnlivé velikosti bloků
 - Navrhnout automatizovanou správu inventáře
 - Navrhnout celkovou koncepci hry vzhledem k předchozím bodům
2. Implementační část
 - Implementovat navržený způsob řešení proměnlivé velikosti bloků
 - Implementovat automatizovanou správu inventáře
 - Implementovat zbylé koncepty a cíle hry, vzeště z 2. kapitoly
 - Kvůli očekávaným nárokům na pochopení nových konceptů do hry implementovat základní výukový tutoriál
 - Získat a zhodnotit zpětnou vazbu (dotazník) na výslednou hru

2. Analýza zadání

V této části provedeme rozbor toho, jak různé hry v současné době přistupují k řešení jednotlivých součástí hry. Tím si připravíme prostor pro podrobnější specifikaci toho, jak by naše hra mohla vypadat a co všechno by měla umět.

2.1 Stávající implementace mechanismů

Rozebereme si, jakým způsobem je v současné době přistupováno k blokům a jak jsou tyto bloky umístovány do herního světa. Dále nás budou zajímat ostatní herní mechaniky, jako například denní cyklus, nebo jakým způsobem je řešena herní postava, její inventář a možnosti hráče interakce se světem. Poté se rozhodneme, jakým způsobem budeme uvedené mechaniky řešit my.

2.1.1 Bloky

Pod pojmem „blok“ budeme chápat objekt, který je umístěn v nějaké ortogonální mřížce 3D prostoru a bez zbytku tento prostor vyplňuje. Bloky spolu sdílí společnou množinu vlastností a dále pak samy definují své vlastní vlastnosti, které vychází z povahy bloku, tedy toho, co daný blok reprezentuje. V této kapitole rozebereme tyto vlastnosti a následně definujeme, jaké problémy budeme muset v této práci řešit.

Základní vlastnosti

Mezi základní vlastnosti bloku bychom měli zařadit *vizuální reprezentaci, pozici ve světě, rotaci a velikost*. Tento výčet není kompletní, ale můžeme říct, že jsou pro hráče nejvíce viditelné. Podle typu hry, jejího stylu a celkové koncepce bychom pak mohli do této množiny přidat třeba *zdraví bloku*. V této části se zaměříme právě na základní vlastnosti a rozšiřujícím vlastnostem se budeme věnovat v následujících částech této kapitoly.

Vizuální reprezentace zřejmě vychází z povahy bloku. Vždy ale platí, že se vždy musí vejít do nějakých hranic bloku, obvykle daných v rámci pravidelné mřížky, ve které se blok nachází. Může v sobě zahrnovat i nějaké informace, které jsou pak hráčem vnímány pohledem. Kupříkladu mohou různými způsoby indikovat vnitřní stav objektu, podobně jako třeba svítivé diody indikují různé stavy elektronických zařízení v reálném světě. Podrobněji se budeme vzhledem zabývat až budeme řešit konkrétní bloky v naší hře.

Pozice bloku v herním světě je u her z kapitoly 1 může být buď na jednoznačně definovaném místě v herním světě, přičemž všechny bloky mají vůči sobě stejnou rotaci (*Minecraft*), nebo je blok součástí nějaké skupiny¹ bloků s jednoznačnou pozicí v herním světě. Tato skupina pak může mít libovolnou rotaci vůči nějakému globálnímu souřadnicovému systému a můžeme říct, že tato skupina tvoří lokální ortogonální systém. Různé skupiny mohou být vůči sobě různě natočeny. Toto chování je možné pozorovat třeba ve hře *Medieval Engineers*, kde postavením

¹Za skupinu budeme považovat shluk alespoň jednoho bloku a všechny bloky ve skupině musí být ve stejně mřížce.

prvního bloku určíme pozici a rotaci této skupiny. Pokud budeme chtít přidat do světa další blok, do nějaké vzdálenosti od prvního bloku je tento nově stavěný blok stále přichytáván do mřížky definované prvním blokem a rotace bloku jsou vždy o 90 stupňů. Obvykle platí, že první blok je umístěn vodorovně (tedy krychle bude i na šikmém terénu umístěna tak, že její horní a dolní strana je ve vodorovné poloze) a je možné ho libovolně rotovat kolem vertikální osy.

Rotace bloků jsou u naprosté většiny her řešeny v zásadě podobně. Výjimkou je pouze *Minecraft*, který rotaci bloků neumožňuje. U většiny bloků to nevadí, protože jsou bloky umístěny po směru, ze kterého je blok umisťován. Můžeme však nalézt některé speciality, třeba blok *kolejí* (oficiální popis bloku [10]). Koleje v *Minecraftu* mohou být rovné (ve směru sever – jih nebo východ – západ), mohou být nakloněné a překonávat výškový rozdíl mezi dvěma bloky, nebo mohou tvorit zatačku. Díky absenci rotací tak hra používá několik pravidel, které určují výslednou podobu bloku. Podrobněji se jimi však zabývat nemusíme. U zbývajících her je situace, kterou jsme popsali v úvodu této části – první blok je natočen vodorovně, má libovolnou rotaci dle vertikální osy a tímto určuje přichytávací mřížku pro další stavěné bloky.

Hry mají velikost bloků vždy stejně velkou. *Minecraft* má hranu bloku o délce 1 m (popis bloku na oficiální Wiki stránce *Minecraftu* [11], oficiální popis jednotek použitých v *Minecraftu* [12])). *Space Engineers* sice bloky hranově omezuje dle kategorií od 0,5 m do 2,5 m (oficiální popis bloků ve hře [13]), ale tyto kategorie mezi sebou nelze kombinovat a je možné k sobě vázat pouze bloky z jedné a té samé kategorie. U ostatních her je situace podobná, byť některé jsou v raných fázích vývoje a tudíž pro ně neexistuje žádný oficiální zdroj informací, takže velikosti bloků bychom mohli pouze odhadovat.

Jako další základní vlastnosti bloku bychom mohli brát třeba *zdraví, cenu za postavení, čas doby stavby, délku trvání destrukce* (pokud vychází z hráčovy akce), co hráč získá za zničení bloku apod. U doby konstrukce a destrukce bloku můžeme říct, že doba trvání dané akce závisí na parametrech bloku a použitím nástroji. Kupříkladu v *Minecraftu* trvá kutání bloku kamene *diamantovým krumpáčem* podstatně rychleji, než při použití krumpáče *dřevěného*. Taktéž rychlosť opotřebení nástrojů je různá. Nicméně když se ve hrách přepneme do tzv. *kreativního* módu, pak máme stavbu bloků „zadarmo“ a ihned. Tedy nemusíme mít žádné komponenty ke stavbě, ani specializovaný nástroj (případ *Space Engineers*, *Medieval Engineers*), nebo nám při postavení bloku tyto bloky neubývají z inventáře (*Minecraft*).

Součásti bloků

Jako součásti bloků můžeme brát cokoliv, co rozšiřuje základní vlastnosti. Zde bychom mohli zmínit třeba *Redstone* v *Minecraftu*. Redstone je speciální typ bloku, chová se jako elektrický vodič a v kombinaci s jinými bloky je možné vytvářet logická hradla. Je zřejmé, že pokud vhodným způsobem zkombinujeme určitá hradla, je možné v *Minecraftu* vytvořit třeba bitovou sčítačku. Ačkoliv vytvoření jednoho hradla je snadné, složitější logické obvody (například kódový zámek) jsou pak velmi náročné na prostor. Nejen z tohoto důvodu pro *Minecraft* vnikly různé módy, rozšiřující základní funkcionalitu hry o nové elektrické komponenty. Pro zajímavost, mód *RedPower* (blog autora [14]) má jednotlivá hradla jako samostatné bloky (což šetří místo) a navíc má možnost skládat různě barevné vodiče

(což jsou pouze ekvivalenty Redstonových bloků) do sebe (až 16 linek signálu). Bez tohoto módu by hráč potřeboval takový prostor, aby položil 16 vedení Redstone tak, aby se nedotýkaly a vzájemně neovlivňovaly (na rovině by tyto vodiče zabíraly šířku 31 bloků).

Další možné součásti bloků, kromě vedení elektřiny, může být například práce s kyslíkem či práce s inventárem. Některé bloky hráči nabízí nějaký úložný prostor, kam může přesunovat objekty ze svého inventáře. Později pak hráč může k bloku přijít a objekty si opět navrátit do svého inventáře. Zajímavou specialitou je náhled inventáře. Kupříkladu u hry *Medieval Engineers* se jedná stůl a jídlo na stole. Stůl má svůj inventář, do kterého je možné umisťovat jídlo. Ve hře je pak obsah inventáře stolu zobrazen tak, že jídlo má svůj náhledový model na talířích na stole, takže to vypadá, jako by bylo jídlo připraveno ke konzumaci.

Dále můžeme zmínit interakci s uživatelem, ať už přímou, nebo nepřímou. Jako přímou interakci budeme chápát takové použití bloku, kdy rovnou vidíme nějakou změnu. To může být například stisknutí tlačítka, nebo změna polohy nějaké páky. Výsledek této přímé interakce pak hráč vidí okamžitě a vizuálně se blok nějakým způsobem změní. Jako nepřímou interakci bychom mohli uvést například otevření nějakého ovládacího rozhraní bloku, což je obvykle nějaká UI obrazovka. Obě interakce se mohou prolínat, takže výsledkem nepřímé interakce může být třeba změna barvy bloku.

2.1.2 Komunikace bloků

Bloky spolu mnohdy umí komunikovat. Dříve zmíněný Redstone z *Minecraftu* by se dal také považovat za jistou metodu komunikace mezi bloky. Například stiskem tlačítka lze změnit výstupní Redstonový signál z tohoto tlačítka (třeba z neaktivního na aktivní), který způsobí změnu polohy nějakého pístu. Ovšem komunikace může být i méně viditelná – například z terminálu v *Space Engineers* je možné ovládat písty, otevírat a zavírat dveře hangáru apod. bez explicitních vodičů signálu.

Ve hrách *Minecraft* a *Space Engineers* můžeme nalézt příklad dopravníkových systémů. Ty umožňují „poslat“ bloky či objekty na jiné místo, kupříkladu z jednoho křídla budovy do druhého. V *Minecraftu* bez módů se této funkcionality dá vcelku snadno dosáhnout, nicméně je to zdlouhavé a rychlosť přesunu bloků je poměrně pomalá. Ale opět se můžeme obrátit na módy, třeba na *BuildCraft* (oficiální stránky projektu [15]), který umožňuje rychle doprovávat bloky i na velké vzdálenosti. Hra *Space Engineers* má kvalitní dopravníkový systém už ve svém základu a slouží například pro dopravu natěženého materiálu od bloku *Vrtáku* do bloku *Skladiště* (který má svůj inventář). Svým způsobem je pak přeprava v rámci těchto systémů také druhem meziblokové komunikace - bloky si mezi sebou předávají konkrétní instance objektů.

2.1.3 Skládání bloků do struktur

Asi jediné příklady, který můžeme zmínit, jsou ve hře *Minecraft* – postavení portálu do Netheru (obrázek 2.1), sněhuláka či golema. V momentě, kdy nějaká skupina bloků splňuje přesně definovaný tvar, tak se tyto bloky chápou jako jedna struktura a tak se s nimi i zachází. Portál je možné pomocí *křesadla* aktivovat, ale

pokud je portál aktivní a hráč odebere z portálu některý blok, který jej tvorí, portál se uzavře. Bloky ve tvaru sněhuláka a golema se po dokončení tvaru odeberou a namísto nich je do světa umístěno *NPC* sněhuláka či golema.



Obrázek 2.1: Portál do Netheru. Zdroj: minecraftpocketedition.wikia.com [16]

Speciality

Ve hrách *Medieval Engineers* a *No man's sky* můžeme nalézt zajímavou funkcionality *multibloků*. Tato funkcionalita nabízí například nahrazení nějaké stěny nějakého bloku oknem či nějakým dalším vizuálním či funkčním elementem.

Také bychom měli zmínit propagaci kyslíku v rámci postavených budov, kterou můžeme nalézt ve hrách *Space Engineers* nebo *Take on Mars*. Ačkoliv tento koncept je spíše vlastností herního prostředí, velice úzce souvisí s bloky. Pokud bloky tvoří uzavřenou strukturu, tak je možné vnitřní prostory natlakovat, zaplnit kyslíkem a sundat si helmu či dokonce celý skafandr. *Space Engineers* využívá systém tzv. „místnosti“ (oficiální dokumentace o kyslíku [17]). Místnosti lze zaplnit kyslíkem díky *ventilaci*, což je blok, který umí do prostoru místnosti vhánět kyslík a také ho z něj odebírat. *Take on Mars* dělá prakticky to samé, liší se pouze způsobem implementace rozpoznání místnosti, což souvisí se způsobem stavby bloků (jak jsme zmínili v kapitole 1).

2.1.4 Herní svět

V této části se zaměříme na zpracování herního světa a s tím souvisejících mechanik.

Reprezentace

Pokud shrneme poznatky z kapitoly 1, tak můžeme říct, že máme v zásadě dvě možnosti. Bud můžeme po vzoru *Minecraft* mít celý svět tvořen bloky. Pak bychom kvůli optimalizaci výkonu také museli tyto bloky shlukovat do skupin (v *Minecraftu* se jim říká *chunks* a jsou velké $16 * 16 * 256$ bloků). Nebo můžeme mít po vzoru *Space Engineers* svět více realistický – a třeba po vzoru této hry hráči připravit třeba celé planety (oficiální představení přidání planet do hry [18]). Koncept planet se objevuje i v dalších hrách (*Medieval Engineers*, *No man's sky*, atd.).

Bloky v herním světě

Jak jsme se zmínili v části 2.1.1, bloky jsou v herním světě umístěny vždy do nějaké mřížky. Různé hry však nabízí různé možnosti práce s těmito mřížkami a ta navíc vychází z možností reprezentace herního světa.

Denní / noční cyklus

Všimněme si, že všechny hry implementují denní cyklus. To dodává hře na dynamičnosti – ne všechny hráče by bavilo desítky či stovky hodin trávit v identickém prostředí. Navíc, pro vývojáře je to možnost, jak hráči připravit nějaké nebezpečí spojené s příchodem noci. Jeden příklad za všechny – v *Minecraftu* se mnozí nepřátelé začínají objevovat, až když je pro ně dostatečná tma. Což nemusí být nutně způsobeno začátkem noci, stačí třeba bouřkové počasí, málo osvětlená jeskyně ve skále či podzemní prostory. Ovšem denním cyklem je hráč nucen se nepřátelům bránit a tím je pro něj zajištěn nějaký netriviální herní prvek, po jehož překonání má hráč obvykle radost a pocit úspěchu. Délka denního cyklu se různí podle hry. *Minecraft* má délku dne 20 minut, u dalších her to je podobné.

Počasí

Zážitek ze hry umocňuje například *proměnlivé počasí*. Stejně jako u denního cyklu, změna počasí do hry přináší prvek náhody a neopakovatelnosti hraní. Můžeme říci, že všechny námi zmíněné hry z kapitoly 1 počasí řeší (v závislosti na jejich herním stylu).

(Ne)fyzikální chování

Za zmínku stojí také přítomnost fyziky ve hře. Hry s vyšším či úplným stupněm realismu ji implementují a u každé hry se více či méně přibližuje realitě. Vymyká se hra *Minecraft*, jejíž fyzikální model se může zdát na první pohled zvláštní, ale je zapotřebí si uvědomit, že tento model zapadá do celkové koncepce hry. Proto je v této hře poměrně zajímavá mechanika kapalin, která je zcela nefyzikální a umožňuje vytvoření nekonečných zdrojů vody. Ostatně sám blok zdroje vody je možné umístit z boku k nějakému sloupu a voda bude neustále vytékat. Pokud navíc odebereme bloky ze sloupu, voda bude vytékat z ničeho a bude z daného místa vytékat dokud dané místo nenahradíme nějakým pevným blokem, třeba hlínou či pískem. Stejně jako voda se ve hře chová i láva, její chování se liší v drobných detailech (například se šíří pomaleji než voda).

Další zajímavou vlastností je „levitace“ bloků. Trochu jsme na toto chování narazili v předcházejícím odstavci, tak to rozvedeme. Každý blok v *Minecraftu* obývá právě jednu pozici v herním světě. A u většiny bloků platí, že mohou zůstat viset volně ve vzduchu. Ale například u písku se po přidání nebo odebrání některého z okolních bloků provede aktualizace bloku. A když blok po aktualizaci zjistí, že by neměl být jen tak ve vzduchu (pod ním nic není), tak začne padat dolů. A jak nastane situace, že hráč potká ve vzduchu visící písek? To souvisí s algoritmem generování herního světa, který má více cyklů a ve výsledku mohou být v herním světě umístěny obdobné kuriozity. Nicméně stačí jeden přidaný či odebraný blok v těsné blízkosti bloku podléhajícímu aktualizaci a dle situace se bloky začnou kaskádovitě aktualizovat, takže v případě písku se začnou sypat dolů. Tam se bloky opět úhledně seskládají do sloupců (což u písku není standardní fyzikální chování – v reálném životě by se písek sesypal na hromadu).

2.1.5 Inventář

Obdobně jako u fyziky ve hrách, chování inventáře je závislé na stupni realismu. *Minecraft* nabízí 27 slotů inventáře, 9 slotů rychlé nabídky a 4 sloty pro brnění. Sloty inventáře a rychlé nabídky jsou plněny *kupitelnými předměty*. To znamená, že podle typu bloku či předmětu může být v daném slotu 1 až 64 prvků daného objektu. Takže například je možné mít v jednom slotu 64 kusů *hlíny*, ale jen 16 kusů *vajec* a pouze 1 *meče*. Plnění rychlé nabídky je možné z UI nabídky inventáře pomocí systému Drag and Drop. Navíc za použití kláves Ctrl a Shift lze řídit počet přenášených kusů daného objektu. Při stavění se blok z daného rychlého slotu odebere, zbraň či nástroj se opotřebí.

U *Medieval Engineers* a *Space Engineers* jsou také pevné sloty, ale existuje zde koncept *skupin slotů*. Přepínáním těchto skupin se mění všechny sloty rychlé nabídky, takže je možné si do různých skupin nastavit různé stavební prvky. Můžeme říct, že *Minecraft* má pouze jednu skupinu slotů. V momentě, kdy hráč staví z více prvků než je velikost rychlé nabídky, se tato funkcionalita stává potřebnou, aby hráč nemusel neustále přehazovat stavěné bloky. Výhodou je, že u těchto her jsou bloky spíše konstrukčního charakteru a neodebírají se z inventáře, takže prvky ve skupině slotů zůstávají zachovány. V *Minecraftu* by i při této funkcionalitě bylo potřeba udržovat skupiny slotů, protože jak jsme již zmínili, stavba v této hře odebírá položky z inventáře. Ve hře jsou však objekty (třeba *jídlo*), které jsou *kupitelné* a jejich chování je stejné jako u *Minecraftu*, včetně řízení počtu přenášených položek za pomocí stisknutých kláves během přenosu do inventárního slotu.

Co se týče stavění, zbývající hry nepřináší nic nového či převratného. Pozorný čtenář si mohl všimnout, že v *Minecraftu* je možné, aby hráč nesl $36 * 64$ kostek kamene, každou o objemu 1 m^3 . Je zřejmé, že nosnost opět nevychází z realistických předpokladů. Nicméně u her s vyšším stupněm realismu se u nosnosti setkáváme, takže kupříkladu je omezena váha, kterou může hráčova postava nést. Navíc se u některých her s rostoucí zátěží zpomaluje pohyb. I toto je součástí herního designu, který větší či menší měrou odráží fyzikální realitu.

2.1.6 Herní postava

Mezi vlastnosti herní postavy bychom mohli zmínit *zdraví*, *výdrž*, *hlad*, zbyvající *zásoba kyslíku* či *energie* a další podobné charakteristiky. Tyto vlastnosti jsou vlastně *survival* prvky a hráč se musí o svoji herní postavu „starat“, aby mu neumřela a aby mohl hru hrát dál. Zde opět můžeme říct, že tyto herní mechaniky hry z kapitoly 1 implementují podobným způsobem a výsledný dojem z nich není u žádné z her natolik zásadní a inovativní, abychom se tím hlouběji zabývali.. Další vlastnosti, kterou jsme již zmínili, je nosnost v rámci inventáře postavy. Podrobněji jsme tuto mechaniku rozebrali v části 2.1.5 a proto se tím zde nebudeme podrobněji zabývat. Dále bychom mohli zmínit možnost přepínání pohledů (pohled z 1. osoby nebo z 3. osoby), což většina her také má.

2.2 Co bychom chtěli implementovat

V následujících podkapitolách si rozebereme naše požadavky na hru. Vzhledem ke komplexnosti celé práce zde shrneme základní koncepci hry. Hráč má omezené zdroje elektrické energie a kyslíku a snaží se přežít na nehostinné planetě. V přežití mu může pomoci třeba postavení nové budovy, přičemž s prostředky musí šetřit. Hráčovým úhlavním nepřítelem je kyselý déšť, který ničí postavené budovy, ale zároveň je z něj možné vyrábět elektrickou energii.

2.2.1 Bloky a herní svět

Pro naše potřeby bude stačit, když budeme mít bloky zarovnané do jednotné mřížky v rámci celého světa (tedy tak, jak to má *Minecraft*). Z toho vyplývá, že nebudeme požadovat implementaci volné počáteční rotace bloku (ve smyslu rotace kolem vertikální osy). Opět zdůrazňujeme, že budeme implementovat prototyp hry se stavěním různě velkých bloků, jednotná mřížka tudíž bude v této fázi dostačovat. S tím souvisí další zjednodušení – nechceme řešit terén ani jeho modifikace a s tím související řešení umístování bloků. V této fázi si postačíme s pouhou rovinou. Protože hry z kapitoly 1 mají rozsáhlé světy a techniky implementace rozsáhlých světů nejsou úplně triviální (použití systému chunků, streamování levelů apod.), nevidíme jako přínosné se jimi zabývat, aniž bychom věděli, že samotná koncepce stavby dynamicky škálovatelných bloků je správný směr dalšího vývoje. Proto budeme pro účely této práce považovat celou herní mapu se všemi postavenými a umístěnými objekty za dostatečně malou, která (ač celá načtená v paměti) zásadním způsobem neovlivňuje výkon ani vykreslování hry. Samozřejmě budeme požadovat, aby i v této variantě byla implementace herního světa dostatečně efektivní jak z pohledu výkonu, tak i z pohledu paměťové náročnosti. Podrobnější rozbor herního světa bude v detailní analýze (TODO link).

Budeme chtít mít bloky, které nebudou pouze statické objekty ve hře. Bloky by měly mít svůj význam ve světě a mělo by jít s nimi interagovat. Navíc budeme chtít, aby bloky byly vizuálně přitažlivé, což bude u některých bloků znamenat, že je budeme muset vymodelovat v nějakém 3D modelovacím programu. Autor této práce však nějaké minimální znalosti v tomto oboru má, takže by to neměl být problém. *Multiblokový* systém není potřeba implementovat. Vzhledem k dynamickému škálování bloků bude potřeba vyřešit škálování vzhledem k případným

3D modelům a také vzhledem k UV mapování textur – chceme, aby se textury chovaly „inteligentně“ a při škálování bloku se textury (při zachování mapování na geometrii modelu) neroztahovaly do nějakých nepěkných výtvorů.

Dále budeme chtít, aby bylo možné definovat jednotlivé vlastnosti bloků jednoduchým a přímočarým způsobem, nejlépe mimo samotný zdrojový kód hry a v nějakém editoru. Tento požadavek je zde z toho důvodu, že je zcela běžné, že herní designéři ladí různé konstanty a nastavení během vývoje tak, aby hra co nejvíce odpovídala jejich představám a byla pro hráče zábavná. Mít tedy tyto konstanty pevně zakomplikované v kódu by znamenalo opětovnou komplikaci celého projektu, což v pozdějších fázích může znamenat výrazné zpomalení vývoje. Protože očekáváme další vývoj této hry, měli bychom se držet nějakých rozumných postupů na udržovatelnost kódu a celého vývoje hry.

Vlastnosti bloků

Chceme navrhnut systém, ve kterém bude nejmenší blok o hraně 20 cm, tedy objemu odpovídající $0,008 \text{ m}^3$. Tento blok nazveme jako *jednotkový*, identicky s tímto budeme používat i pojem *jednotková krychle*. Největší blok pak omezíme na 20-ti násobek jednotkové krychle ve všech 3 rozměrech. Největší blok tedy bude mít objem 64 m^3 . Může se stát, že dolní limit bude příliš malý, ale v tuto chvíli považujeme tuto konstantu za dostatečnou. Naopak horní limit bude nejspíše dostatečný – práce s příliš velkými bloky by mohla být neefektivní a stavba nepřehledná.

V následující tabulce (tabulka 2.1) si popíšeme, jaké bloky budeme ve hře chtít mít, jak by měly vypadat a co by měly umět. Bloky jsme rozdělili do dvou kategorií – první (**A**) jsou *konstrukční* prvky, druhá kategorie (**B**) obsahuje *funkční* prvky. *Název* bloku je zřejmý. Sloupce *Min* a *Max* popisují velikost bloku jako násobky jednotkové krychle. Dále jsou v tabulce sloupce *P* a *R*². Ty popisují to, zda je blok možné rotovat podle daných os. Správně bychom tu měli mít i sloupec *Y* (z anglického *Yaw*, ale protože je pro všechny bloky povolen, tak jsme tento sloupec vynechali).

Jako poslední je sloupec *T*, tedy typ bloku. Typ bloku může být: *Kostka*, *Zkosený*, *Rohový*, *Vlastní*. Typ pak ovlivňuje další vlastnosti bloku, jako například cenu za jeho postavení, ale i jeho zdraví. (TODO podrobný popis algoritmu; , K = 1, Z = 0.5, R = 1/6, V = 1 (není v potaz objem))

²Písmenka vychází z anglického *Pitch* a *Roll*.

Název		Min	Max	P	R	T
A Základní bloky						
A1.	Blok základny	1–1–4	20–20–4		K	
A2.	Blok stavby	1–1–1	20–20–20	✓	✓	K
A3.	Blok polykarbonátu	1–1–1	20–20–20	✓	✓	K
A4.	Zkosený blok základny	1–1–4	20–20–4		Z	
A5.	Zkosený blok stavby	1–1–1	20–20–20	✓	✓	Z
A6.	Roh bloku stavby	1–1–1	20–20–20	✓	✓	R
B Speciální bloky						
B1.	Terminál	1–8–5	1–8–5		V	
B2.	Napájené okno	2–1–2	20–1–20	✓	✓	K
B3.	Dveře	7–7–11	7–7–11		V	
B4.	Světlo	1–1–1	1–1–1	✓	✓	V
B5.	Přepínač	1–1–1	1–1–1	✓	✓	V
B6.	Generátor energie	3–3–2	20–20–2		K	
B7.	Generátor objektů	3–3–2	20–20–2		K	
B8.	Akumulátor	3–3–3	3–3–3		V	
B9.	Plnička kyslíkových bomb	4–3–4	4–3–4		V	
B10.	Kyslíková bomba	2–2–2	2–2–2		V	

Tabulka 2.1: Požadované bloky a jejich základní vlastnosti

2.2.2 Podrobný popis bloků

V následujících odstavcích si podrobně popíšeme všechny bloky a nastíníme jejich význam. Některé bloky mohou obsahovat elektrické a kyslíkové *komponenty*, jejichž význam bude zmíněn o pár odstavců dálé (TODO link!).

A1 – Blok základny

Tento blok je ve hře z toho důvodu, protože má sloužit jako základový blok staveb. Pokud bychom měli nerovný terén, tento blok by mohl zahrnovat podstavce pro vyrovnání terénu. Velikost v ose Z omezena na 4 základní bloky. Obsahuje *elektrickou komponentu*.

A2 – Blok stavby

Tento blok je základním stavebním blokem ve hře a proto může existovat ve všech možných velikostech. Obsahuje *elektrickou komponentu*.

A3 – Blok polykarbonátu

Tento blok je nejlevnější, neobsahuje žádné komponenty. Ideou bloku je podpora průhledných stěn a také možné pomocné stavební konstrukce pro výstavbu do výšky. Inspiraci můžeme vidět v používání třeba bloku hlíny ve hře *Minecraft*, kdy hráč vyskočí a pod sebe umístí nový blok hlíny a tím se ve světě posune o 1 metr výš.

A4 – Zkosený blok základny

Tento blok je zde ze stejného důvodu jako blok A1 – Blok základny. Má stejné vlastnosti, pouze má zkosený tvar. Může sloužit jako přístupová rampa.

A5 – Zkosený blok stavby

Obdobně jako u základny, tento blok má stejné vlastnosti jako A2 – Blok stavby, pouze má jiný tvar.

A6 – Roh bloku stavby

Tento blok má stejné vlastnosti jako A2 – Blok stavby a opět definuje pouze jiný tvar.

B1 – Terminál

Terminál má pevnou velikost (1 x 8 x 5 bloků) a měl by vypadat jako nějaká obrazovka budoucnosti. Obsahuje *elektrickou komponentu*. Terminál by pro hráče měl být použitelný přímo i nepřímo. Přímé použití by mělo dobrít herní postavě energii, nepřímé by mělo otevřít uživatelské rozhraní. Co přesně by mělo být obsahem tohoto rozhraní bude řešeno v podrobné analýze (TODO link).

B2 – Napájené okno

Napájené okno je omezeno pouze v jednom rozměru, jinak povolíme velikost od 2 do 20 násobku jednotkové krychle. Dolní omezení je zde z toho důvodu, že okno o velikosti 20 x 20 cm je zbytečně malé. Obsahuje *elektrickou komponentu*.

B3 – Dveře

Dveře jsou další ze speciálních bloků s pevnou velikostí (7 x 7 x 11 bloků). Jejich účel je jasné – možnost uzavření budovy a pokud se rozhodneme implementovat systém okysličování vnitřků budov, tak i hermetické uzavření jednotlivých místností.

B4 – Světlo

Blok světla má také jasný účel – osvětlení míst, kde není dostatek přirozeného světla či osvětlení v budově, pokud nastane noc. Světlo by mělo být možné použít (přinejmenším ho chceme umět vypnout a zapnout). Obsahuje *elektrickou komponentu*.

B5 – Přepínač

Účel přepínače je zřejmý – ovládání ostatních bloků. Proto bude muset být použitelný. Skvělé by bylo, kdybychom na základě vizualizace modelu přepínače ihned poznali stav, ve kterém se přepínač nachází. Obsahuje *elektrickou komponentu*.

B6 – Generátor energie

Tento blok byl přidán proto, aby hráč získával novou elektrickou energii z prostředí. Energie je vyráběna během bouře kyselého deště (TODO forward link), tato mechanika bude podrobněji popsána později. Na výšku byl blok omezen na 2 jednotkové krychle. Obsahuje *elektrickou komponentu*.

B7 – Generátor objektů

Smyslem tohoto bloku je generování nových bloků. Vzhledem k dalším herním mechanikám můžeme říct, že to je spíše nutná komponenta *Konstruktoru objektů* (sekce 3.7). Na výšku byl blok omezen na 2 jednotkové krychle. Obsahuje *elektrickou komponentu*.

B8 – Akumulátor

Tento blok byl do hry přidán z toho důvodu, že je potřeba pokrýt elektrické nároky stavby v období, kdy neprší kyselý déšť. Funkce akumulátoru je tedy uchování elektrické energie. Z toho vyplývá, že musí obsahovat *elektrickou komponentu*. Bylo by také hezké, kdybychom měli ve hře rychlý náhled na úroveň jeho nabité.

B9 – Plnička kyslíkových bomb

Plnička je příklad bloku, která má *elektrickou* a zároveň *kyslíkovou komponentu*. Jejím cílem je plnit Kyslíkové bomby kyslíkem. Zároveň by bylo hezké, kdyby blok uměl doplnit kyslík herní postavě napřímo. Pokud to bude možné, rádi bychom viděli (podobně jako u stolu v *Medieval Engineers*) náhled plněné kyslíkové bomby. Tedy aby hráč na první pohled věděl, zda je nějaká bomba plněna.

B10 – Kyslíková bomba

Tento blok slouží k uchování kyslíku, použitím tohoto bloku je možné doplnit kyslík herní postavě. Zároveň je možné tento blok sebrat do inventáře a později z inventáře umístit do herního světa. Má *kyslíkovou komponentu*. Pokud to bude možné, chtěli bychom hráči zobrazit stav naplnění, stejně jako u B8 – Akumulátor.

2.2.3 Komunikace bloků

Chceme, aby bloky v elektrické síti spolu uměly komunikovat a bylo třeba možné vzdáleně tyto bloky ovládat. Obdobný systém je možné nalézt i ve hře *Space Engineers*, kde jsou tlačítka pro ovládání různých dveří, pístů a dalších interaktivních bloků. My pro základní ovládání použijeme blok B5 – Přepínač.

Celkově vidíme koncept elektrického vedení a elektrické sítě jako zajímavý i pro naši hru, takže obdobnou funkcionalitu bychom také chtěli mít.

2.2.4 Zdraví bloků

Chceme, aby bloky měly zdraví a aby bylo možné je zničit. Bloky v elektrické síti ale necháme se uzdravovat, což bude spotřebovávat energii. Protože očekáváme, že pouze bloky exponované na vnější straně budov budou předmětem uzdravování, dává nám smysl požadovat nějaký způsob přednostního uzdravování bloků, které budou s největší pravděpodobností nejdříve zničeny. Cílem je větší podpora exponovaných a tedy kriticky důležitých bloků. Oproti tomu pokud bude blok z větší části zastíněn nějakými jinými bloky, nebude jeho expozice vůči celkovému zdraví tak velká, že by hrozilo okamžité zničení.

2.2.5 Skládání bloků do struktur

Chceme hráči umožnit postavení komplexní struktury bloků, která bude do hromady dávat nějaký speciální význam. V našem případě to bude *Konstruktor objektů* (3.7), díky kterému za pomoci bloku B1 – Terminál může hráč vytvářet nové bloky, které pak bude moci umístit do světa. V našem pojetí to bude spíše objekt, který bude imaginárně vymýšlet optimální rozvržení bloku (de facto takový automatizovaný návrhář nových bloků). Bloky jednou vymyšlené pak hráč bude moci stavět libovolně mnohokrát, jen musí mít dostatečnou zásobu energie pro jejich postavení.

2.2.6 Herní svět

Jaký bychom chtěli mít herní svět z pohledu bloků a jejich umisťování jsme již řešili v části 2.2.1. V následujících odstavcích se zaměříme na další charakteristiky světa naší hry.

Denní / noční cyklus

Aby naše hra nebyla úplně nudná z nedostatku stavebních bloků, tak budeme chtít mít také denní a noční cyklus. *Minecraft* má délku denního cyklu cca 20 minut. My zkusíme 30 minut³.

Počasí

V naší hře budeme chtít mít pouze dva druhy počasí – jasno a kyselý dešť. Jak jsme již zmínili, kyselý dešť se nám bude hodit i jako zdroj elektrické energie, jasné počasí chceme z toho důvodu, že mít jenom jeden typ počasí by hráče brzy přestalo bavit. Budeme tedy chtít nějakým způsobem vizualizované počasí. Pokud to bude možné, budeme rádi, pokud ve hře bude i kombinace těchto dvou druhů, minimálně z toho důvodu, aby změna počasí byla dynamická a nebyla příliš náhlá. Kyselé deště budou přicházet v náhodných intervalech a budou sloužit jako překážka v rozvoji hry. Zároveň to ale bude pro hráče nástroj, jak získávat prostředky pro ochranu před dalšími dešti a rozvoj svých staveb.

³Původně jsme zamýšleli 60 minut, ale z interního testování během vývoje vyplynulo, že je to až příliš dlouhá doba, hlavně kvůli malému množství bloků.

(Ne)fyzikální chování

Fyzikou se nebudeme nijak podrobně zabývat. Pokud budeme mít možnost snadné implementace nějaké fyziky, budeme za to rádi. V tuto chvíli však fyziku nevidíme jako klíčovou vlastnost naší hry.

2.2.7 Inventář

Rádi bychom zkusili neomezeně mnoho slotů a tím, že jednotlivé inventární skupiny bude možné pokud možno automatizovaně spravovat tak, aby hráč nemusel neustále řešit správu inventáře. Počet inventárních skupin by měl být koncipován jako potenciálně nekonečný, ale pokud se bude ve výsledné hře počet skupin limitovat, nebude to příliš vadit.

Položky v inventáři budou *stavitelné* (ty, které vzejdou z *konstruktoru objektů*) a *umístitelné* (bloky, které je možné „vzít“ do inventáře a později je opět umístit do herního světa).

2.2.8 Herní postava

Budeme chtít, aby naše herní postava měla *zdraví*, zásobu *kyslíku* a *energie*. Všechny charakteristiky by mělo být možné obnovovat. Pokud hráčovo zdraví nebo kyslík bude na nule, hra by mělo skončit. Dále bychom chtěli mít možnost přepínání pohledů (pohled z 1. osoby nebo z 3. osoby). Hráč by také měl vidět *HUD*, kde uvidí aktuální stav vlastností postavy a případné další informace (například během stavění).

2.3 Shrnutí cílů

V dalším textu práce se budeme věnovat analýze a následné implementaci následujících cílů:

1. Bloky a herní svět
 - Způsob řešení proměnlivé velikosti bloků
 - Umístění bloků v herním světě
 - Skládání bloků do struktur
 - Komunikace bloků
 - Interakce s bloky
 - Denní cyklus
 - Proměnlivé počasí
 - Použitelné datové struktury vzhledem k ostatním částem hry
2. Inventář
 - Automatizovaná správa inventáře
 - Stavitelné a umístitelné bloky
3. Herní postava
 - Inventář herní postavy
 - Zbylé vlastnosti postavy dle 2.2.8
4. Ostatní herní prvky
 - Systém ukládání a načítání hry
 - Kreativní herní mód
 - Herní tutorial
5. Případné prvky navíc
 - Lokalizace hry do různých jazyků
 - Hudba ve hře
6. Zhodnocení práce
 - Získat a zhodnotit zpětnou vazbu (dotazník) na výslednou hru

3. Detailní analýza

V této kapitole podrobně rozebereme cíle práce. Už víme, čeho bychom chtěli dosáhnout a nyní potřebujeme vyřešit *jak* toho dosáhnout.

3.1 Herní engine

V první řadě bychom se měli zamyslet nad tím, jaký nástroj pro vývoj hry použijeme. Díky tomu budeme moct počítat s možnostmi a omezeními danými touto volbou. Shrňme si, co budeme ve hře potřebovat:

- Renderování 3D objektů, pokročilé možnosti texturování
- Podpora I/O pro práci se savy
- Podpora UI
- Správa assetů
- Správa scény

Plusem pak bude:

- Podpora zvuků
- Snadná implementace lokalizace

Pro další případný rozvoj bychom potřebovali:

- Podpora pathfindingu
- Podpora síťové hry
- Podpora AI

Cílová platformy pro nás bude PC s OS Windows. Pokud se rozhodneme pro již existující herní engine, který bude navíc podporovat multiplatformní vývoj, bude to pro nás, i s ohledem na další vývoj, plus. Dále bychom chtěli, abychom minimalizovali finanční náklady na vývoj. Z toho vyplývá, že placené řešení bychom byli ochotni použít pouze pokud by to pro nás znamenalo naprostou výhodu oproti jiným, neplaceným řešením.

Dalším kritériem je volba programovacího jazyka. Ta vychází z autorových znalostí. Budeme tedy preferovat primárně jazyk C#, který známe nejlépe. Pokud to bude nezbytně nutné, nebudeme se bránit ani jazyku C++, který je v herní branži dlouho zavedený a je stále hojně využívaný. Ačkoliv zkušenost s tímto programovacím jazykem máme minimální, můžeme se tímto způsobem naučit novým dovednostem.

Jak můžeme naši hru implementovat?

- Implementace kompletního vlastního enginu
- Použit existující grafické knihovny a nad tím implementovat vlastní engine
- Použit existující herní engine

Možných použitých enginů a frameworků je opravdu mnoho. Pro jejich seznam se můžeme podívat do databáze herních enginů na stránce Devmaster. Jen zde je možné nalézt 236 možných řešení našeho problému volby herního enginu [19]. Všechny záznamy jsme omezili na *vývojově aktivní*, v jazycích C#, C++ a vybrali jsme námi požadované vlastnosti. Je zřejmé, že možností na výběr máme opravdu hodně. V následujících podkapitolách si jednotlivé možnosti podrobně rozebereme.

3.1.1 Vlastní engine

Tuto možnost rovnou zavrhneme. Vzhledem k tomu, kolik funkcionality budeme implementovat, nevidíme přínos v další práci s implementací vlastního enginu. Naším cílem je prototyp hry a tudíž nechceme ztráct drahocenný čas vývojem nutných nástrojů a systému pro naši hru.

3.1.2 Vlastní engine s použitím již existujících grafických knihoven

Máme na výběr z více druhů grafických frameworků postavených na různých platformách. Mezi známějšími bychom mohli uvést například *XNA* (C#) či jeho klon *Monogame* (C#). Oba frameworky jsou k dispozici zdarma, podpora *XNA* je v současné době už ukončena, vývoj *Monogame* je stále aktivní. Implementace hry s použitím některého z těchto frameworků by byla rychlejší než v předchozím případě, ale stále bychom museli spoustu funkcionality implementovat sami.

3.1.3 Existující herní engine

Jak jsme již předeslali výše, v této kategorii máme nejvíce možností. Budu můžeme využít enginy jako třeba *Ogre* (C++), nebo použít více robustnější řešení v podobě enginů typu *Unity* (C#) či *Unreal Engine* (C++)¹. Zde opět použijeme předchozí argument – budeme hledat engine, který nám nabídne pokud možno co nejvíce uživatelské a vývojářské přívětivosti a bude poskytovat dostatek nástrojů pro vývoj naší hry v uvažovaném rozsahu.

Výhodou zmíněných robustních enginů je to, že jsou k dispozici zdarma. Taktéž zde, díky práci komunity, existuje pro oba enginy kvalitní vývojová dokumentace. Dalším kladem je fakt, že jsou oba multiplatformní a tedy zde existuje relativně snadný postup v případě distribuce na různé typy herních zařízení. Pojdme si je tedy rozebrat podrobněji.

¹V době volby herního engine byl další známý zástupce *CryEngine* placený a proto jsme tento herní engine nebrali v potaz.

Unity

Výhodu *Unity* vidíme v tom, že i programátor bez rozsáhlých zkušeností s herním vývojem může začít velmi brzy prototypovat a vyvíjet hry v tomto enginu. Dalším pozitivem je programování v C# a možnost editovatelného terénu.

Použití *Unity* s sebou přináší i několik problémů, které bychom museli během vývoje řešit. Během rešerše jsme zaznamenali problémy s aktualizací dynamického navigačního meshe, kdy aktualizace tohoto meshe způsobovala krátkodobé zaseknutí hry (tzv. lagy). Můžeme očekávat, že tato funkcionality bude v budoucnu vylepšena a zrychlена, nicméně na konkrétní datum se nemůžeme spoléhat. Vzhledem k povaze naší hry ale můžeme očekávat časté modifikace herního světa a tudíž toto chování pro nás představuje významný problém. Další nevýhodu vidíme v materiálovém editoru, který nabízí oproti *Unreal Engine* limitované možnosti a pro implementaci náročnějších materiálových funkcí bychom museli přistoupit k implementaci vlastních shaderů.

Co se lokalizace hry týče, museli bychom si napsat vlastní správu lokalizace (oficiální tutorial na implementaci lokalizace [20]). *Unreal Engine* má tuto funkcionality implementovanou ve svém editoru (oficiální dokumentace k lokalizaci [21]).

Unreal Engine

Oproti *Unity* je *Unreal Engine* podstatně komplexnější a pochopení všech vztahů a závislostí může být pro začínajícího herního programátora obtížné. Přes tuto zjevnou nevýhodu jsme však během rešerše zjistili, že *Unreal Engine* nám poskytuje podstatně příjemnější prostředí pro vývoj s komplexnějšími nástroji. Grafické možnosti máme díky materiálovému editoru k dispozici od začátku a nemusíme k tomu umět psát shadery třeba v jazyce HLSL. Je nám jasné, že výsledný grafický výkon nemusí být nutně optimální, nicméně vzhledem k povaze této práce stejně nebudeme cílit na grafickou a výkonovou optimalizaci.

Testy s navigačním meshem a jeho dynamickou aktualizací byly uspokojivé – nenarazili jsme na žádný zádrhel nebo pokles výkonu během aktualizace meshe.

Co musíme zmínit jako nevýhodu je absence editovatelného terénu. V editoru je možné vytvořit krásný terén se rozličnými možnostmi detailů, nicméně tento terén není možné jednoduchým způsobem editovat. Chápeme to spíše jako nepříjemnost, než zásadní nevýhodu.

Další komplikaci vidíme v použití C++, se kterým jsme v době rešerše měli malé zkušenosti.

3.1.4 Volba engine -verdikt

Nakonec jsme zvolili poslední možnost – *Unreal Engine*. Autorovy znalosti především z oblasti C# sice hovořily pro použití *Unity*, nicméně výhody použití *Unreal Engine* byly dostatečnými argumenty pro to, abychom navzdory malé znalosti C++ a obtížnosti pro začínajícího vývojáře zvolili právě toto řešení.

3.2 Struktura projektu

Celý projekt v *Unreal Engine* je rozdělen do dvou částí – C++ část a Blueprintová část. *Unreal Engine* umožňuje herním vývojářům implementovat celou hru kompletně za pomocí Blueprintů, tedy vizuálního rozhraní. Toho se však obvykle nevyužívá, protože vykonávání programu v Blueprintu je přibližně 10 krát pomalejší, než vykonávání nativního C++ kódu (odpověď jednoho z vývojářů *Unreal Engine* na dotaz ohledně rychlosti Blueprintů: „*Blueprints are approximately the speed of UnrealScript in UE3, and our rule of thumb there was about 10x slower than C++.*“, [22]). V praxi tak dochází k tomu, že vývojář (i neprogramátor) může za pomocí Blueprintů rychle prototypovat funkcionalitu, kterou pak kodér přepíše do metod v C++, správně tyto metody označí makry tak, aby *Unreal Engine* tyto metody v kódu našel (což se provádí za pomocí reflexe v *UBT* a použitím příslušných C++ maker) a upraví Blueprint tak, aby původní kód tyto metody volal.

V Blueprintu je kód vizualizován jako graf uzlů a jsou zde zaznamenány vztahy mezi těmito uzly. Uzly tedy odpovídají funkcím v C++ a těm je pak možné předávat parametry ať už v podobě proměnných definovaných v C++ kódu nějaké třídy, nebo proměnných definovaných přímo v Blueprintu. Vztahy mezi uzly pak označují následující kód určený k vykonávání. Můžeme také říct, že vykonávání kódu v Blueprintu je *interpretované*, z čehož vyplývá absence případných kompilačních optimalizací. Ergo slabý výkon.

Některé části programu jsou implementovány na úrovni C++, jiné bylo nutné implementovat v Blueprintech. Už víme, že komunikace z Blueprintu do C++ je možná prostým voláním metod, ale určitě budeme potřebovat i možnost opačného směru. Toho je možné dosáhnout více způsoby – kupříkladu použitím delegátů a událostí (Blueprint se pak na tuto událost naváže a v případě vyvolání dané události vyvolá v Blueprintu definovanou obsluhu), nebo BlueprintImplementable či BlueprintNative metod. (TODO formátování!, link na Specifikaci?) Poslední dvě nám nabízí možnost, jak volat virtuální metody, které je možné přepisovat jak na straně C++, tak na straně Blueprintu, což se nám bude hodit.

3.3 Bloky

V této části rozebereme, jak můžeme definovat a následně implementovat bloky a popíšeme, jaké jsou výhody a nevýhody jednotlivých implementací. V prvé řadě se změříme na celkovou strukturu bloků a následně budeme řešit, jak budeme spravovat konstanty ovlivňující chování bloků.

3.3.1 Celková struktura

Aby se nám s bloky dobře pracovalo, jistě bude vhodné využít jednoho ze základních principů *OOP* (objektově orientovaného programování) – dědičnosti. Takže v naší hře bude existovat základní třída, která bude vycházet ze třídy **UActor**² a bude předkem všech našich herních bloků.

²UActor je základní třída *Unreal Engine*, ze které dědí všechny herní objekty, které chceme v hlavní herní smyčce aktualizovat a renderovat.

Tento prapředek bude obsahovat dvě podstatné informace – referenci na *definici* daného bloku a referenci na třídu s vlastnostmi dané *instance* bloku. Díky tomu, že oddělíme definiční třídu a instanční třídu, tak získáme možnost získat definiční třídu pro daný typ bloku za běhu hry pouze jednou a posléze tuto referenci předávat všem instancím bloku daného typu. Instanční třída bude mít pro každý blok jiné hodnoty, takže je zřejmé, že by měla být samostatná. Smyslem definičního souboru je popis omezujících podmínek kladených na daný typ bloku (například povolené minimální a maximální rozměry), přičemž hodnoty v instanční třídě by měly být v mezích dané definicí. Vlastnosti, které nejsou omezující (například cena za postavení bloku), nebudeme uchovávat v *instanční* třídě.

Dále budeme držet komponenty.

Určitě budeme schopni najít množinu vlastností, které mají všechny bloky společnou. Jak již bylo zmíněno v analýze (TODO ref), mezi tyto vlastnosti určitě bude patřit *pozice* ve světě, *rotace* bloku a jeho *velikost*. Tyto vlastnosti pak pro danou instanci můžeme nějakým způsobem serializovat do souboru a tím budeme mít validní soubor pro uložení rozehrané hry.

Protože jednotlivé bloky rozšiřují tyto základní informace o další vlastnosti (například *elektrická* či *kyslíková* komponenta má také své vlastnosti), můžeme se zamyslet nad tím, že bychom pro jednotlivé části bázovou třídu základních vlastností rozšířili. Zde ale narážíme na problém, protože jednotlivé bloky mohou komponenty libovolně kombinovat. Ačkoliv C++ standardně povoluje vícenásobnou dědičnost tříd, *Unreal Engine* toto nepovoluje (TODO link!) a tudíž tento přístup není vhodný. Řešením bude rozdělit vlastnosti jednotlivých komponent do samostatných tříd a bloky, které budou tyto komponenty obsahovat, si pak budou držet referenci na instanci dané třídy s vlastnostmi pro danou komponentu. Jako bonus tím získáme typovou bezpečnost a odpadá nám nutnost přetytovávání na vyšší typ (protože ve společném předkovi všech bloků

Z předchozí části víme, že budeme chtít takovou strukturu, abychom mohli v *Unreal Engine* snadno modifikovat hodnoty jejich vlastností. Takže nepřipadá v úvahu, abychom měli konstanty uložené ve zdrojovém kódu. Pak ale tyto konstanty budeme muset bud načítat z nějakého souboru, nebo budou muset být uložené v nějakém objektu z *Unreal Engine*.

Při načítání ze souboru máme více možností – můžeme popisovat bloky a jejich chování v třeba XML souborech. Obdobné řešení používá hra *Medieval Engineers*. Tyto soubory jsou pak zpracovány herním enginem během načítání hry.

3.3.2 Textové soubory

V této části se zamyslíme nad použitím definic v textovém souboru. Může se jednat o množinu samostatných souborů, přičemž budeme uvažovat pouze XML soubory, nebo můžeme použít třeba popis bloků v nějakém tabulkovém formátu, třeba csv.

Popis tabulkou

Pokud bychom měli velice málo vlastností bloků, tento přístup by mohl být použitelný. Nicméně s každým dalším nově přidaným blokem se do množiny všech vlastností mohou zanášet nové vlastnosti. To by znamenalo, že popis ostatních

bloků, které danou vlastnost nemají, by musel nutně v tomto tabulkovém zápisu uvažovat nějakou (byť prázdnou) hodnotu. Zbytečně by nám tak rostl definiční soubor. Další nevýhodou je absence typové bezpečnosti.

Popis samostatným souborem – XML

Proč neuvažujeme běžné soubory textového formátu? Výsledek je stejný jako při použití XML, ale nemůžeme zde použít definiční soubory pro automatickou kontrolu platnosti hodnot. Navíc bychom museli psát vlastní parser takového textového souboru, přičemž již hotové parsery XML jsou volně k dispozici.

Výhodou tohoto přístupu je také budoucí rozšiřovatelnost o nové bloky

Soubor obsahuje popis vlastností

- externě editovatelné formáty (+ – modding, – těžší implementace, parsing, validace)
- binární formát
 - xml
 - interní formát - specifické subclassy pro bloky včetně specifických vlastností
- přímo na - definiční struktura

3.4 Vlastnosti bloků

Popis toho, co blok umí

3.4.1 Energie

- popis energie, co to umí (např. výkon)

3.4.2 Energetická síť

-způsob zapojení do sítě

3.4.3 Kyslík

- to je podobný jako energie
- mít možnost uchování kyslíku, v případě použití elektirky pak i generování

3.4.4 Označovatelnost

- hráč může avatarem zamířit na blok a ten se označí červeně, žlutě zeleně

3.4.5 Možnost vzít do inventáře

- bloky mohou být sebratelné, tedy hráč si je může dát do svého inventáře. vlastnosti jako třeba uchovaná hodnota kyslíku, pak zůstávají zachované

3.4.6 Interakce

- vypínač, světla – vlastní UI
- bloky mohou být použitelné, tj. hráč s nimi může nějakým zásobem interagovat

3.4.7 Zapojení do rozpoznávání tvarů

- generátor bloků
- jaké byly možnosti – abecné rozpoznávání (původní implementace, rozvést nutnost rozpadu tvarů na menší (slope) + doplnění kvádry)

3.5 Komponenty bloků

popis jednotlivých komponent dle předchozího, co všechno umí (např. přidání / odebrání hodnoty energie za použité zámku (není transakce))

3.6 Bloky v herním světě

- je více množností. Uchování pole 50000 x 50000 x 25000 // todo ověřit je nesmysl.
- nepotřebujeme otevřený svět bez mřížky (pozdější aktualizace ME, jinak SE), takže budeme hledat nějakou variantu stromové struktury
- nabízí se možnost clustrování budov a shlukování do skupin, s následnou optimalizací počtu hladin
 - my jsme zvolili K-D strom kombinovatný s AABB. (proč?)
 - náš strom má optimalizaci jedinného potomka, v případě potřeby se dogeneruje do úrovně níže, případně rozpadne na podčásti a rekursivně se přidá.
 - díky této variantě se můžeme snadno dotazovat na sousedy, což je hlavní cíl (proto)

3.7 Konstruktor objektů

popsat mechaniku konstruktoru

3.8 Počasí

- počasí chceme proměnlivé ale s tím, že gamedesignéři mohou snadno ovlňovat výsledné počasí, případně aby šlo snadno rozšířit varianty pro různé herní módy
- budeme mít ve světě umístěnou entitu (Pawn) ovládaný AI Controllerem – to z toho důvodu, že pro AI Controller můžeme použít BehaviorTree
 - popsat ideu BT
 - další možnosti by byly, že bychom prostě použili update smyčku nějakého Actora – není potřeba, tohle se vyřeší updatem na komponentě

3.9 Hráčova postava

- pohled 1st person, 3rd person
- má komponenty kyslíku, energie
- může stavět, interagovat s bloky
- může zařvat

3.10 Inventář

- je to vlastnost hráče
- v inventáři má několik přepínatelných banků
- bank může být se stavitelnými bloky nebo s intentárními předměty
- bank je možné filtrovat
- důvod pro použití banku – rychlé přepnutí při stavění (minecraft složitá organizace při stavění a použití 10+ druhů bloků)

3.11 Ukládání hry

- vše se musí korektně uložit
- ?? specifikace binárního formátu zde, nebo v programátorský?

3.12 Doplňující vlastnosti

3.12.1 Lokalizace

- použití lokalizace

3.12.2 Hudba

- atmosférický hudební doprovod

3.13 Backlog

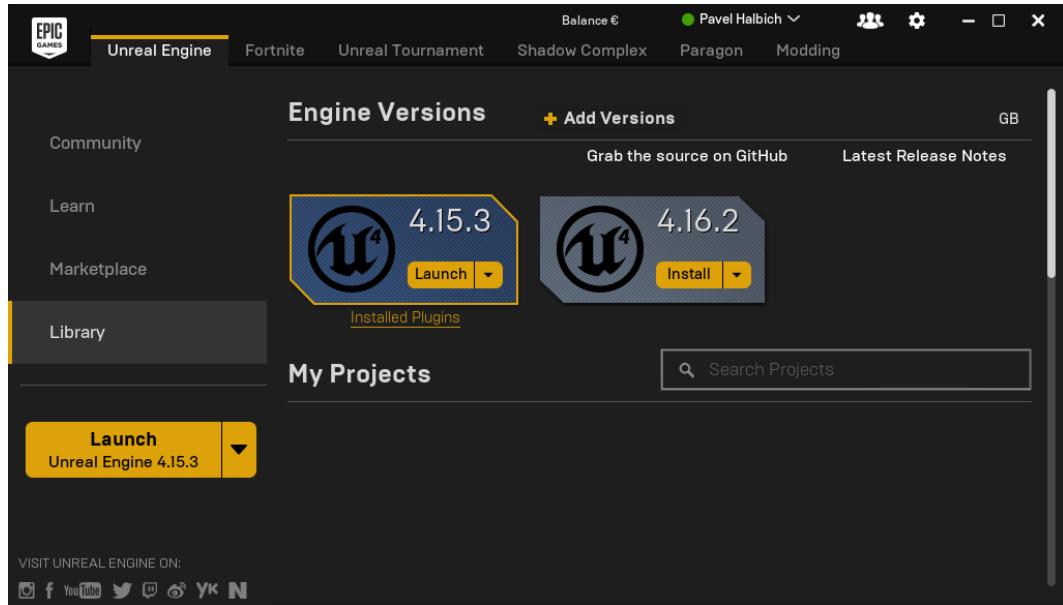
- Popsat, že bychom chtěli nějaké UI + nabídky menu

Popis některých vlastností – má energetickou komponentu – > implikuje definici bindovacích bodů má kyslíkovou komponentu – implikuje TotalObjectOxygen Producer nebo Consumer implikuje Total object energy Controllable implikuje IsController nebo IsControllable

4. Programátorská dokumentace

4.1 Počáteční inicializace projektu

Pokud je cílem spustit projekt hry ze zdrojových kódů, je potřeba si stáhnout *Unreal Engine* ve verzi 4.15.3. Na hlavní stránce *Unreal Engine* je potřeba stáhnout *Epic Games Launcher* (ke stažení zde [23]). Po vytvoření uživatelského účtu a následnému přihlášení do aplikace bude vidět obrazovka podobná obrázku 4.1:

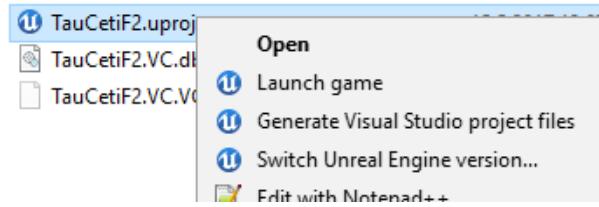


Obrázek 4.1: Epic Games Launcher

Pokud není vidět *Unreal Engine* dané verze v seznamu verzí, je potřeba jej přidat kliknutím na tlačítko *Add Versions* a případném následném výběru správné verze. Na obrázku 4.1 je vidět *Unreal Engine* verze 4.16.2, který je připravený ke stažení. Posledním krokem je samotná instalace stisknutím tlačítka *Install*. Použití novější verze *Unreal Engine* je možné, ale běžnému uživateli to nedoporučujeme. Mezi verzemi se mohou projevit nekompatibility v kódu, které je mnohdy nutné řešit přímo do zdrojových kódů hry.

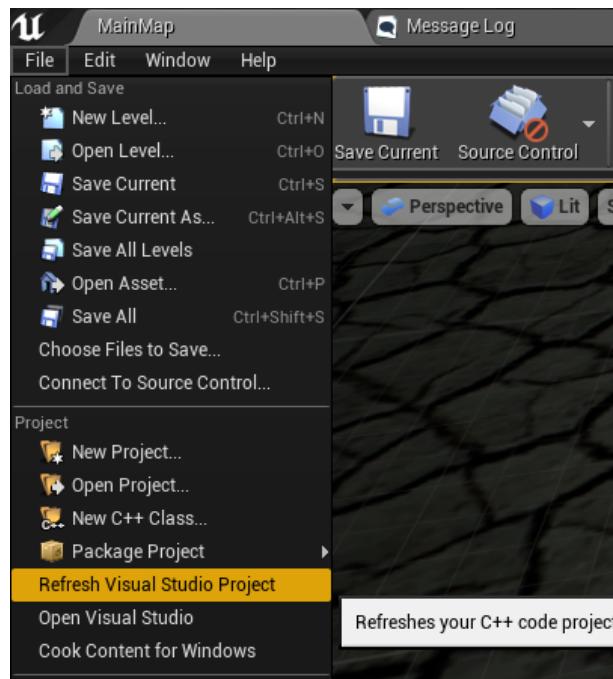
Dále je potřeba mít k dispozici zdrojové kódy ať už z DVD, nebo z tohoto release na GitHubu (TODO link na public repo, release commit).

Následujícím krokem je vytvoření projektu hry. Toho se docílí kliknutím pravým tlačítkem myši na soubor *TauCetiF2.uproject* a následnou volbou *Generate Visual Studio project files* (viz obrázek 4.2).



Obrázek 4.2: Vytvoření projektu hry

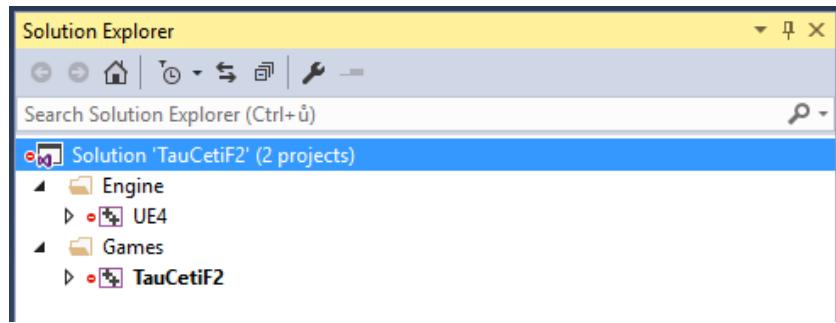
Pokud v kontextové nabídce chybí možnosti pro *Unreal Engine*, je potřeba provést asociaci s uproject soubory. Tato asociace by měla být provedena automaticky, nejpozději po vytvoření libovolného C++ projektu z kterékoliv šablony. Ze zkušenosti autora – toto se mnohdy nemusí podařit. Pokud se nepodaří vygenerovat projekt hry, může pomoci otevřít editor přes *Epic Games Launcher*, najít a otevřít projekt a po načtení vybrat možnost obnovy projektu, jak je naznačeno na obrázku 4.3. K tomu je zapotřebí mít Visual Studio 2015 alespoň ve verzi *Community*.



Obrázek 4.3: Vygenerovaný projekt hry

Pokud i toto selže, ověřte si, prosím, že je možné založit nějaký projekt zařazený na C++, zkompilovat jej a taktéž vygenerovat projekt pro Visual Studio. Pokud se to povede s projektem z nějaké připravené šablony, mělo by to fungovat i s naší hrou.

Předposledním krokem je v případě úspěšného vytvoření projektu hry jeho otevření ve *Visual Studio*, nastavení *TauCetiF2* jako hlavní spustitelnou část projektu (viz obrázek 4.4) a následné spuštění editoru hry.



Obrázek 4.4: Vygenerování projektu hry

4.2 Struktura projektu

V dalším textu postupně popíšeme celkovou strukturu hry z pohledu *Unreal Engine* a poté se budeme zabývat kódovou částí napsanou v C++. Ještě bychom zde měli zmínit, že ačkoliv se v textu budeme odkazovat převážně na hlavičkové soubory, stále budeme brát v potaz i implementační, tedy .cpp soubory a obsah textu se může na tuto implementaci odkazovat.

4.3 Struktura projektu v Unreal Enginu

- ukázat jak se to dělí v UE editoru

Struktura složek

- Složka XY

Popsat lokalizaci

4.3.1 Mapy

popsat jednotlivé mapy, jaký je jejich cíl

a

4.4 Struktura kódu

Protože jsme zvolili implementaci práce v *Unreal Engine*, můžeme využít toho, že engine umožňuje rozdělit celý herní projekt do jednotlivých herních modulů [24]. Tím docílíme modularity, nebudeme mít celý projekt v jednom kuse a zároveň tím urychlíme překlad projektu při komplikaci.

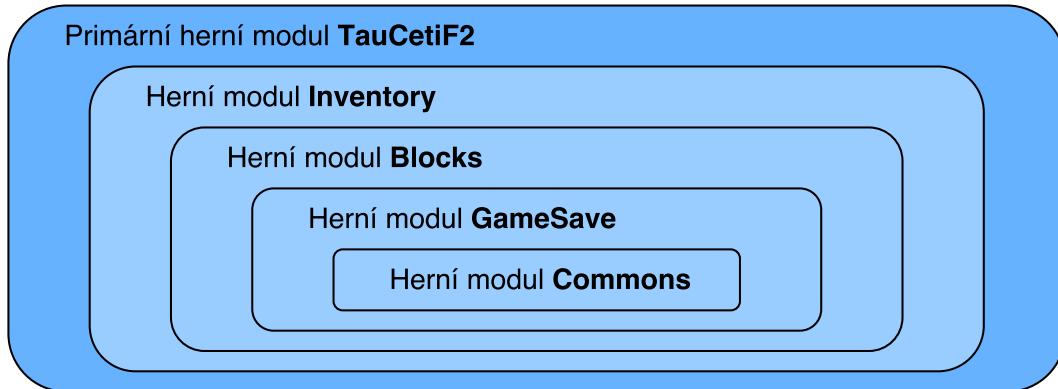
Každý *Unreal Engine* projekt musí definovat právě jeden primární herní modul. Pokud využijeme možnosti vytvoření nového projektu založeného na C++, editor tento modul automaticky vytvoří za nás. My jsme pojmenovali náš projekt *TauCetiF2* a tak se jmenuje i náš primární modul.

Jednotlivé části projektu jsme rozdělili do několika herních modulů:

1. TauCetiF2 (primární modul)
2. Inventory

3. Blocks
4. Game Save
5. Commons

Herní moduly jsme seřadili dle jejich závislosti tak, že každý modul závisí na všech modulech s vyšším očíslováním. Tedy primární modul využívá všech ostatních modulů a poslední modul (Commons) není závislý na žádném dalším modulu. Pro lepší představu můžeme tyto souvislosti vyjádřit obrázkem 4.5:



Obrázek 4.5: Diagram závislostí modulů projektu.

Obecnou koncepci programu jsme popsali v kapitole 4.3. Abychom se vyhnuli velkému množství dopředných referencí, strukturu programu z pohledu C++ budeme popisovat od nejnižších vrstev.

4.4.1 Struktura modulu

Všechny moduly dodržují společnou strukturu. Obsahují:

- složku `Public`
- složku `Private`
- soubor `[název_modulu].Build.cs`
- soubory `[název_modulu].h`, `[název_modulu].cpp`

Každý modul pak má hlavičkové soubory ve složce `Public`, implementaci tříd pak ve složce `Private`. Poslední tři soubory jsou kvůli *UBT* a použitím herních modulů v rámci *Unreal Engine*

4.5 Modul Commons (C++)

Tento modul je základním modulem, který na jednom místě definuje všechny potřebné informace, které využívají ostatní moduly. Jedná se zejména o definici herních konstant, či definice všech sdílených enumerátorů. Najdeme zde také prapředka použité herní instance. Tuto vlastní implementaci herní instanci využijeme pro ukládání nalezených bloků.

4.5.1 Herní definice a konstanty

V souboru `GameDefinitions.h` jsou definovány všechny herní konstanty. Například je zde definována velikost jednotkové krychle, velikost použitého světa, vztah mezi délkou dne herního světa a počtem uplynulých reálných sekund, převody mezi energií, kyslíkem, zdraví a jednotkou zásahu kyselého deště. Taktéž jsou zde definovány konstanty, které využívá technika obrůs objektu (todo link na outline). Dále jsou zde definovány konstanty ID implementovaných bloků, abychom s nimi mohli pracovat i v kódu.

4.5.2 Herní instance

Herní instance `TCF2GameInstance.h` se chová jako návrhový vzor *Singleton* a jako jediná zůstává vždy stejná po celou dobu běhu hry (jak uchovávat globální data [25]). Proto se využívá například pro uchovávání dat při přechodu mezi jednotlivými Levely a my toho také využijeme. Zároveň se tato třída dá využít pro implementaci delegátů, kterými je možné vyvolat nějakou událost a libovolný prvek z herního světa může tuto událost obsloužit. My toho využijeme u reakce na denní cyklus u bloku *Přepínače*.

Dalším důležitým bodem pro nás bude, že tato třída si bude držet referenci na všechny nalezené bloky. Z předchozího textu již víme, že bloky je potenciálně možné rozšířit o DLC (TODO budem to tu řešit?), takže je nutné, abychom si nalezené bloky a jejich definice udrželi v paměti i při přechodu mezi levely. K tomu slouží proměnná `BlockHolder`, která sice drží referenci na objekt definovaný v modulu `Blocks`, ale kvůli zpětným referencím mezi moduly (které nejsou povolené) musíme zde použít dostupného předka.

Kód tedy bude vypadat následovně:

```
UPROPERTY(Transient)
UObject* BlockHolder;

UFUNCTION(BlueprintCallable, Category = "TCF2 | GameInstance")
void SetHolderInstance(UObject* holder);
```

Parametr `Transient` u makra `UPROPERTY` znamená, že daná proměnná bude vždy nastavena na svoji výchozí hodnotu. V tomto případě je to použito spíše z důvodu zachování konzistence napříč projektu, ale zjednodušeně bychom důsledky mohli popsat následovně – pokud bude nějaký Blueprint dědit z nějaké C++ třídy, tak vývojář může nastavit výchozí hodnoty properties. Tyto hodnoty jsou pak serializovány do *CDO*, což je *Class Default Object* (otázka na Answers Unreal Engine [26]). Během procesu vytváření nové instance objektu, který vychází z daného Blueprintu pak budou tyto hodnoty naplněny během fáze inicializace properties (popis životního cyklu actorů [27]). V konečném důsledku by pak byla tato hodnota nějakým způsobem naplněna. Pokud chceme vynutit, aby tato property nebyla serializována do CDO, tak ji označíme jako `Transient`.

V průběhu hry pak jednou naplníme tuto property pomocí metody `SetHolderInstance`, do které předáme referenci na korektně inicializovanou instanci třídy `BlockHolder` z modulu `Blocks`. Pak si můžeme odkudkoliv získat

aktuální herní instanci, přetypovat na `TCF2GameInstance` a získat si (přetypovanou) referenci na `BlockHolder`. Z něho pak již můžeme získávat informace o všech dostupných blocích.

4.5.3 Enumerátory

`Enums.h` Tento soubor slouží jako jednotné umístění pro všechny výčtové typy (enumerátory), které se používají napříč celým projektem. Neznamená to, že nutně obsahuje všechny – některé třídy mohou využívat své specifické enumerátory, které ale nemusí být umístěny v tomto globálně dostupném modulu.

4.5.4 Helpery

`CommonHelpers.h` Tato třída poskytuje metody pro práci s konfigurací. Statické metody umožňují načítat a ukládat konfigurační položky typu `float`, `bool` a `string`. Aby byla práce co nejjednodušší, metody přijímají enumerátor `EGameUserSettingsVariable`. Třída pak sama na základě hodnoty tohoto enumerátoru použije správný klíč (který je textový) a tak může ukládat či vracet hodnotu daného typu z konfiguračního souboru.

4.6 Modul Game Save (C++)

Modul GameSave slouží k ukládání a načítání informací o probíhající hře do binárního formátu. K tomu používáme streamové operátory `<<`, které jsou v tomto případě implementovány tak, že je možné je použít jak pro ukládání, tak pro načítání. // TODO link na tutorial

Díky tomuto přístupu tak můžeme definovat celou strukturu výsledného binárního souboru na jednom místě a tedy rozšířování uložené hry je triviální. Co si ovšem musíme pohlídat je to, abychom si drželi informaci o verzi uloženého souboru. V našem případě, pokud se bude lišit verze načteného souboru a uložená konstanta v programu, save prostě odmítneme (a dokonce smažeme). V produkčním prostředí bychom si mazání nemohli dovolit, ale museli bychom save ignorovat a uživateli zobrazit nějakou hlášku o tom, že verze souboru není podporovaná. My jsme se však v tomto případě rozhodli save mazat, protože jsme očekávali, že během vývoje hry se bude binární struktura savu často rozšiřovat. Po každé iteraci jsme si savy prostě vytvořili nové.

Co by se stalo, kdybychom se snažili načíst save jiné verze? Celá hra by nejspíše byla ukončena s chybou, protože by se pokoušela číst neplatná data a/nebo by očekávala nějaká data tam, kde žádná nejsou. Tím bychom četli z neplatné lokace.

4.6.1 GameSaveInterface

`UGameSaveInterface.h` Tento soubor definuje rozhraní, které je možné implementovat a používat v BlueprintTech (struktura vychází z tutoriálu na Unreal Engine Wiki [28]). Rozhraní definuje dvě veřejné metody, které musí actoři v UE při implementaci tohoto rozhraní implementovat:

```

UFUNCTION(BlueprintNativeEvent, BlueprintCallable, ... )
    bool SaveGame();

UFUNCTION(BlueprintNativeEvent, BlueprintCallable, ... )
    bool LoadGame();

```

Hlavní Blueprint levelu pak během svého běhu určité actory přetypuje na tento interface a bude s nimi dále pracovat. Podrobněji tato funkciionalita bude popsána v Blueprintové části.

4.6.2 FFileVisitor

`FFileVisitor.h`

Tento soubor obsahuje implementaci návrhového vzoru Visitor pro získání všech herních savů z nějaké zadáné složky. Implementace vychází z internetové diskuse na téma procházení adresářů [29].

4.6.3 Helpers

`SaveHelpers.h` Tento soubor řeší získání seznamu všech uložených her a využívá k tomu implementaci FileVisitoru z předchozí části.

4.6.4 Kontejner s uloženou hrou

`SaveGameCarrier.h` Tato třída je v zásadě přepravkou pro data s možností ukládání a načítání dat do binárního formátu. Navíc umožňuje během ukládání v době vývoje vypsat vlastnosti právě ukládané hry do konzole tak, abychom mohli během vývoje snadno tento výstup zkopirovat a vytvořit pevně implementované hry. Tyto pevně implementované hry pak nebudou data vracet po přečtení nějakého binárního souboru, ale budou vracet pevně nastavená data. Vytváření těchto pevně vytvořených her pak bylo o dost jednodušší.

Přepravka obsahuje pouze holá data, tedy nevytvářejí se žádné nové instance herních objektů. To je z toho důvodu, že nemůžeme použít následující kód:

```

// vlastnost kontejneru
TArray<UBlockInfo> UsedBlocks;

// během ukládání
void USaveGameCarrier::SaveLoadData(
    FArchive& Ar,
    USaveGameCarrier& carrier,
    TArray<FText>& errorList,
    bool bFullObject)
{
    Ar << carrier.UsedBlocks;
}

```

Museli bychom mít referenci na datový typ `UBlockInfo`, který je ale definovaný v modulu `Blocks`, který musí nutně modul `GameSave` referencovat. Tudíž zde použijeme následující konstrukci:

```
// vlastnost kontejneru
TArray<FBlockInfo> usedBlocks;

// během ukládání
void USaveGameCarrier::SaveLoadData(
    FArchive& Ar,
    USaveGameCarrier& carrier,
    TArray<FText>& errorList,
    bool bFullObject)
{
    Ar << carrier.usedBlocks;
}
```

Třídu dědící z `UObject` jsme nahradili strukturou `FBlockInfo`, která sama slouží pouze jako přepravka na data. Vyšší vrstva, tedy modul `Blocks` si těmito daty naplní své vlastní objekty, které se pak dále využívají ve hře. A naopak, před samotným uložením se postará o to, aby bylo toto pole korektně naplněno všemi daty určenými k uložení. O samotnou serializaci a deserializaci dat do a z přepravky se stará pouze modul `GameSave`.

Celý systém vychází z tutoriálu [30] je postaven na tom, že v C++ je možné přetěžovat operátory, mimo jiné i `<<`. Této vlastnosti je využito tak šikovně, že v závislosti na volání funkce buď zapisuje do archivu, nebo z něj čte, ale pořád se jedná o jeden zápis jedné funkce. To je výhodné, protože to předchází chybám, které by mohly vzniknout při použití 2 metod – jedné čtecí, jedné zapisovací. Chybám typu přehození dvou datových typů (což by v případě typů různých velikostí znamenalo následné špatné pochopení binárních dat), nebo kupříkladu prohození dvou vlastností stejného typu, což by vytvářelo těžko odhalitelné situace změn hodnot ve hře.

Tento systém je použit pro všechny rozšířené části herního savu – Bloky, Inventář, Počasí – a všechny používají podobný způsob práce. Jedná se o definici struktur, tedy samotných kontejnerů a dále pak jeden soubor pojmenovaný `*ArchiveHelpers.h`, kde je popsána vlastní struktura daných kontejnerů v Archivu. Hvězdičku v tomto případě bereme opravdu jako zástupný symbol. Navíc, některé objekty mohou řídit archivaci dle nějakých podmínek nastavených shora. Příkladem budiž definice serializace elektrické komponenty:

```
// přetížení, které se nevolá vždy
FORCEINLINE FArchive& operator<<(
    FArchive &Ar,
    FPoweredBlockInfo& componentInfo)
{
    Ar << componentInfo.IsOn;
    Ar << componentInfo.AutoregulatePower;
    Ar << componentInfo.PowerConsumptionPercent;
    return Ar;
}
```

```

FORCEINLINE FArchive& operator<<( 
    FArchive &Ar, 
    FElectricityComponentInfo& componentInfo)
{
    Ar << componentInfo.CurrentObjectEnergy;
    Ar << componentInfo.HasPoweredBlockInfo;

    // pokud máme navíc rozšiřující data, přidáme další data
    // tím efektivně zavoláme metodu uvedenou výše
    if (componentInfo.HasPoweredBlockInfo)
        Ar << componentInfo.PoweredBlockInfo;

    return Ar;
}

```

Jak vidíme z kódu, celý kód se chová korektně jak při serializaci, tak i při deserializaci. Je pouze potřeba vzít v úvahu, že vyšší struktury, které pak budou tyto data používat pro vlastní inicializaci, musí také brát v potaz podmínku. Pokud není splněna, tak svázaná proměnná (v tomto případě `PoweredBlockInfo`) nebude obsahovat platná data.

4.6.5 NewGameSaveHolder

`NewGameSaveHolder.h` Tato třída je hlavní třídou, se kterou hra před načítáním pracuje. Definuje seznam napevno zabudovaných levelů a obsahuje jejich implementaci.

4.7 Modul Blocks (C++)

Modul bloků obsahuje podstatné informace o tom, jak hra pracuje s bloky, jak se tyto bloky skládají do herního světa, jaké jsou jejich komponenty apod. Také je v tomto modulu možné nalézt specifické implementace jednotlivých bloků.

V dalším textu se budeme odkazovat na složky. Odkazujeme se tím do složek `/Source/Blocks/Public` a jejich `Private` implementací. Strukturu bychom mohli shrnout následovně:

1. Definice bloků (složka `Definitions`)
2. Třídy s popisem bloků (složka `Info`)
3. Systém ukládání a načítání bloků (složka `Helpers`)
4. Rozhraní, které mohou bloky implementovat (složka `Interfaces`)
5. Komponenty, kterými bloky rozšiřují svoji základní funkcionalitu (složka `Components`)
6. Implementace jednotlivých bloků (složky `BaseShapes`, `Special`)
7. Stromové struktury herního světa (složka `Tree`)

4.7.1 Definice bloků

V této složce se nachází všechny definiční soubory bloků. Definiční soubor obsahuje pouze popis datové struktury a nějakou minimální funkcionalitu (kupříkladu získání korektního vektoru velikosti v závislosti na tom, zda má definice daného bloku nastavenou vlastní velikost). Jednotlivé konkrétní instance s daty jsou pak definovány na straně editoru. Konstanty (například minimální a maximální škálování) je pak možné měnit v editoru a není vyžadována rekompilace projektu hry.

Definiční soubor se skládá následujícím způsobem:

- UBlockDefinition (`BlockDefinition.h`)
 - FUsableBlockDefinition (`UsableBlockDefinition.h`)
 - FBlockMeshStructureDefinition (`BlockMeshStructureDefinition.h`)
 - FBlockMaterialDefinition (`BlockMaterialDefinition.h`)
 - FBlockAdditionalFlags (`BlockAdditionalFlags.h`)
 - FBlockFlagValue (`BlockFlagValue.h`)
 - FOxygenComponentDefinition (`OxygenComponentDefinition.h`)
 - FElectricityComponentDefinition (`ElectricityComponentDefinition.h`)
 - FElectricityBindableAreas (`ElectricityBindableAreas.h`)

4.7.2 Třídy s popisem bloků

Tyto třídy popisují už konkrétní instance bloků v rámci hry. Jejich hodnoty jsou pak v mezích definovaných v definičních třídách. Tyto třídy jsou pak předmětem ukládání a načítání. Dalším důležitým prvkem je BlockHolder, který slouží pro nalezení bloků.

4.7.3 Ukládání a načítání bloků

- ukládání – máme něco jako block saving helpers

4.7.4 Interfaces

poskytují nástroje pro volání metod na instančních interfacích
popsat ideu za Implementation, Execute (BlueprintNativeEvent, BlueprintImplementableEvent)

4.7.5 Komponenty bloků

- pak máme komponenty bloků a nějaké interfaces

Elektrická komponenta

Elektrická síť

Kyslíková komponenta

Select target

World object

4.7.6 Implementace bloků

– základ Block.h, zbytek v jendotlivých podkategoriích (BaseShapes / Special

TODO jak moc podrobné? vypsat všechny bloky a co všechno implementují, nebo to stačí stručně zmínit? – co implementují by si čtenář mohl uvědomit z předchozího textu a navíc je to jen nudný popis, jehož významové hodnota je ve zdrojácích a není asi nutné to tu duplikovat

- popsat speciální bloky + nějaké speciality co umějí (showableWidget)

4.7.7 Stromové struktury

popsat stromové struktury, které tam mám

MinMaxBox

prapředek všeho

KDTree

dědí z MMB, základ ve světě

WeatherTargetsKDTree

dědí z MMB, slouží pro potřeby počasí

4.8 Modul Inventory (C++)

Modul inventáře byl vyčleněn do samostatné části. Je to hlavně jako ukázka možného členění do modulů. Navíc časem by se mohl tento modul rozšířovat jak by rostla komplexita správy inventáře.

Nejdůležitější inventory component

4.8.1 Tag group

nejnižší úroveň, odpovídá 'nebo'

4.8.2 Inventory tag group

celá skupina, odpovídá 'A zároveň'

4.8.3 Inventory tags

sdružuje všechny banky

4.8.4 Inventory component

celá komponenta, která je pak navázaná na hráčův charakter
definuje delegáty notifikující o změnách v aktívní skupině, po filtrování apod.
na této úrovni se řeší aktualizace cache buildable i inventorybuildable při
změnách, zároveň poskytuje možnost clear cache pro volání shora (BP)

4.9 Modul TauCetiF2 (C++)

- primární modul
 - popsat co všechno obsahuje (widgety, gamemodes, weather apod)
 - popsat synchronize widget (// TODO link na důvod, proč to tam mám),
popsat object widget, napsat důvody
 - popsat to stackování
 - popsat komponenty (weather, game electricity)

4.10 Backlog

Zde popsat jak jsem to celé implementoval a proč

Popsat jednotlivé C++ třídy a jejich odvozené Blueprintové deriváty + přidat
případné obrázky z BL kódu (např. BlueprintImplementable event, který se zavolá
jak na C++ tak i na BP)

Udělat rozbor BT počasí + mechaniku počasí + denního cyklu popsat řízení
osvětlení dle počasí

Udělat rozbor bloků, škálování, konfigurace, datovou strukturu, implementaci
dynamických textur, zvýraznění

Popsat mechaniku Selector – SelectTarget + napojení na Builder

Popsat mechaniku používání objektů + zvýraznění

-> Mám svět, ten má v sobě bloky, ty jsou v nějaké stromové struktuře, bloky
mají komponenty, které přes tuto strukturu mohou na sebe vázat Svět má také
počasí se svojí vlastní strukturou, využívající podobnosti s bloky (2D KD strom
s Heapem na listech)

-> hráč může to a tamto, díky inventáři se dostane na bloky, a díky selectoru je
pak můževložit do světa skrz World controller (zmíněno v předchozím) ->zároveň
jsou všechny entity savovatelné

-> Popsat struktury Widgetů, zmínit použití Synchronize Widgetu, imple-
mentaci mechaniky stackovatelných widgetů

-> popsat implementaci hudby

// TODO obrázky s konfiguračními ukázkami do příloh (např. jak se definuje
Blok z UE

5. Uživatelská dokumentace

Tato část obsahuje informace o tom, jak hru spustit a jaké jsou požadavky ke spuštění. Dále jsou zde uvedeny obrázky ze hry a popis toho, co znamenají.

5.1 Požadavky pro spuštění hry

Hardwareové požadavky

Doporučená minimální sestava (na ní byla hra vyvíjena):

Procesor:	Intel i7-2630QM @ 2.00GHz
RAM:	12 GB (8 GB by mělo také stačit)
Grafika:	ATI Radeon HD 6700M
OS:	Win 10 x64 (7 a vyšší by měly být v pohodě)

Výše uvedenou konfiguraci je potřeba brát jako orientační. Hru jsme úspěšně spustili i na notebooku s procesorem Intel i5, integrovanou grafickou kartou a 8 GB operační paměti. Bylo však nutné nastavit grafické vlastnosti na minimální možnou konfiguraci.

Softwareové požadavky

Pro spuštění zkompilované hry není potřeba nic speciálního. Je zapotřebí mít stroj s minimální uvedenou konfigurací. Dále je dobré mít nainstalované poslední verze ovladačů HW komponent (hlavně grafiky). Taktéž je zapotřebí mít nainstalovanou poslední verzi DirectX.

6. Závěr

6.1 Zhodnocení práce

6.2 Zhodnocení dotazníku

6.3 Budoucí práce

- dynamičtějí mřížka? 20cm je nejspíše dost málo a vyžaduje to dost preciznosti // TODO zkusit pro test 25 či 30 cm a patřičným způsobem upravit velikosti modelů? (nejspíše to musí zůstat hardcoded, ale zkusím se nad tím zamyslet, pokud bude čas)
- vlastní sortování v seznamech

Seznam použité literatury

- [1] Brian Johnson. Urbandictionary.com: Boss fight, 2014. URL <http://www.urbandictionary.com/define.php?term=boss%20fight>.
- [2] dillonsup. Minecraft vs terraria (facts), 2013. URL <http://www.minecraftforum.net/forums/minecraft-discussion/discussion/178129-minecraft-vs-terraria-facts>. internetové fórum.
- [3] Minecraft Wiki. Combat update, 2017. URL http://minecraft.gamepedia.com/Combat_Update.
- [4] Keen Software House. Vrage, 2017. URL <http://www.keenswh.com/vrage.html>.
- [5] Marek Rosa. Space engineers: Super-large worlds, procedural asteroids and exploration, 2014. URL http://blog.marekrosa.org/2014/12/space-engineers-super-large-worlds_17.html.
- [6] GameSpot. Space engineers, 2014. URL <https://static.gamespot.com/uploads/original/1365/13658182/2626082-space-engineers1.jpg>.
- [7] gFleka. Automated relay drone tutorial, 2014. URL <http://space-engineer.net/space-engineers/automated-relay-drone-tutorial/>.
- [8] Hry.cz. Recenze hry take on mars, 2017. URL <https://www.hry.cz/hra/take-on-mars>.
- [9] Erik Fagerholt; Magnus Lorentzon. Beyond the hud - user interfaces for increased player immersion in fps games. Master's thesis, 2009. URL <http://publications.lib.chalmers.se/records/fulltext/111921.pdf>.
- [10] Minecraft Wiki. Rail, 2017. URL <http://minecraft.gamepedia.com/Rail>.
- [11] Minecraft Wiki. Block, 2017. URL <http://minecraft.gamepedia.com/Block>.
- [12] Minecraft Wiki. Tutorials/units of measure, 2017. URL http://minecraft.gamepedia.com/Tutorials/Units_of_measure.
- [13] Space Engineers WIKI. Blocks, 2015. URL <http://spaceengineers.wikia.com/wiki/Category:Blocks>.
- [14] Eloraam. Eloraams blog, 2015. URL <http://www.eloraam.com/>.
- [15] Buildcraft Team. Official page, 2011. URL <https://www.mod-buildcraft.com/>. Pozn.: Uvedené datum se váže k první publikované verzi 1.4.1 dle dostupných zdrojových kódů na GitHubu <https://github.com/BuildCraft/BuildCraft/releases/tag/1.4.1>.

- [16] Darthwikia25. Minecraft nether portal, 2015. URL http://minecraftpocketedition.wikia.com/wiki/File:Minecraft_Nether_Portal_02.jpg.
- [17] Space Engineers Wiki. Oxygen, 2017. URL <http://spaceengineerswiki.com/Oxygen>.
- [18] Space Engineers. Space engineers now with planets, 2015. URL <http://www.spaceengineersgame.com/space-engineers-now-with-planets.html>.
- [19] Devmaster.net. Game engines (filtered), 2017. URL http://devmaster.net/devdb/engines?query=&name=&developer_name=&status=active&languages_supported_ids%5B%5D=1&languages_supported_ids%5B%5D=3&features_ids%5B%5D=1&features_ids%5B%5D=2&features_ids%5B%5D=3&features_ids%5B%5D=4&features_ids%5B%5D=5&features_ids%5B%5D=6&features_ids%5B%5D=7&features_ids%5B%5D=8&features_ids%5B%5D=12&features_ids%5B%5D=13&features_ids%5B%5D=16&features_ids%5B%5D=18. Citováno 16. 5. 2017.
- [20] Unity. Localization manager tutorial, 2017. URL <https://unity3d.com/learn/tutorials/topics/scripting/localization-manager>.
- [21] Epic Games. Localization, 2017. URL <https://docs.unrealengine.com/latest/INT/Gameplay/Localization/>.
- [22] JamesG. Blueprint performance benchmark?, 2014. URL <https://forums.unrealengine.com/showthread.php?1105-Blueprint-Performance-Benchmark>.
- [23] Epic Games. Download unreal engine, 2017. URL <https://www.unrealengine.com/download>.
- [24] Epic Games. Gameplay modules, . URL <https://docs.unrealengine.com/latest/INT/Programming/Modules/Gameplay/>.
- [25] Rama. Global data access, data storage class accessible from any cpp or bp class during runtime, 2016. URL https://wiki.unrealengine.com/Global_Data_Access,_Data_Storage_Class_Accessible_From_Any_CPP_or_BP_Class_During_Runtime.
- [26] Epic Games. What is cdo?, 2015. URL <https://answers.unrealengine.com/questions/191353/what-is-cdo.html>.
- [27] Epic Games. Actor lifecycle, . URL <https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/ActorLifecycle/>.
- [28] Epic Games. How to make c++ interfaces implementable by blueprints (tutorial), 2016. URL [https://wiki.unrealengine.com/How_To_Make_C%2B%2B_Interfaces_Implmentable_By_Blueprints\(Tutorial\)](https://wiki.unrealengine.com/How_To_Make_C%2B%2B_Interfaces_Implmentable_By_Blueprints(Tutorial)).

- [29] Lions Den. Iteratedirectory questions, 2014. URL
<https://forums.unrealengine.com/showthread.php?48881-IterateDirectory-Questions>. Citováno 2. 4. 2015.
- [30] Rama. Save system, read & write any data to compressed binary files, 2016. URL https://wiki.unrealengine.com/Save_System,_Read_%26_Write_Any_Data_to_Compressed_Binary_Files.
TODO citováno všecko dne... Odebrat neznámá data, pokud možno vyřešit tu čárku, co přebývá

Přílohy