



**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 taktéž 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	Hry kompletně blokové	4
1.1.2	Hry s prvky realismu	5
1.1.3	Hry s maximálním důrazem na simulaci reality	7
1.1.4	Ostatní - zařadit TODO	7
1.2	Čemu se budeme věnovat	7
1.3	Herní bloky	7
1.3.1	Náš návrh úpravy	8
1.4	Inventář	8
1.4.1	Náš návrh úpravy	8
1.5	Cíle práce	8
2	Analýza zadání	9
2.1	Stávající implementace mechanismů	9
2.1.1	Bloky	9
2.1.2	Komunikace bloků	9
2.1.3	Skládání bloků do struktur	10
2.1.4	Zdraví bloků	10
2.1.5	Herní svět	10
2.1.6	Inventář	11
2.1.7	Avatar hráče	11
2.2	Co bychom chtěli implementovat	11
2.2.1	Bloky	11
2.2.2	Podrobný popis bloků	12
2.2.3	Herní svět	14
2.2.4	Inventář	14
2.2.5	Avatar hráče	14
2.3	Herní nepřítel	15
2.4	Backlog	15
3	Detailní analýza	16
3.1	Herní engine	16
3.1.1	Vlastní engine	17
3.1.2	Vlastní engine s použitím již existujících grafických knihoven	17
3.1.3	Existující herní engine	17
3.1.4	Volba engine -verdikt	18
3.2	Bloky	18
3.3	Vlastnosti bloků	18
3.3.1	Energie	19
3.3.2	Energetická síť	19
3.3.3	Kyslík	19
3.3.4	Označovatelnost	19
3.3.5	Možnost vzít do inventáře	19
3.3.6	Interakce	19

3.3.7	Zapojení do rozpoznávání tvarů	19
3.4	Komponenty bloků	19
3.5	Bloky v herním světě	19
3.6	Počasí	20
3.7	Hráčova postava	20
3.8	Inventář	20
3.9	Ukládání hry	20
3.10	Doplňující vlastnosti	20
3.10.1	Lokalizace	20
3.10.2	Hudba	20
3.11	Backlog	21
4	Programátorská dokumentace	22
4.1	Počáteční inicializace projektu	22
4.2	Struktura projektu	22
4.3	Struktura kódu	23
4.3.1	Struktura modulu	24
4.4	Modul Commons (C++)	24
4.4.1	Herní definice a konstanty	24
4.4.2	Herní instance	24
4.4.3	Enumerátory	25
4.4.4	Helpery	25
4.5	Modul Game Save (C++)	25
4.5.1	GameSaveInterface	26
4.5.2	FFileVisitor	26
4.5.3	Helpers	26
4.5.4	Kontejner s uloženou hrou	26
4.5.5	NewGameSaveHolder	28
4.6	Modul Blocks (C++)	29
4.6.1	Definice bloků	29
4.6.2	Třídy s popisem bloků	30
4.6.3	Ukládání a načítání bloků	30
4.6.4	Interfaces	30
4.6.5	Komponenty bloků	30
4.6.6	Implementace bloků	30
4.6.7	Stromové struktury	30
4.7	Modul Inventory (C++)	31
4.7.1	Tag group	31
4.7.2	Inventory tag group	31
4.7.3	Inventory tags	31
4.7.4	Inventory component	31
4.8	Modul TauCetiF2 (C++)	31
4.9	Struktura projektu v Unreal Enginu	31
4.10	Backlog	32
5	Uživatelská dokumentace	33
5.1	Požadavky pro spuštění hry	33

6	Závěr	34
6.1	Zhodnocení práce	34
6.2	Zhodnocení dotazníku	34
6.3	Budoucí práce	34
	Seznam použité literatury	35
	Přílohy	37

1. Úvod

V době vzniku této práce jsou velice populární hry s otevřeným světem. Lákaají hráče na obsáhlou svět 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 nabízejí 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 *sanboxové*, *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 navrhneme a implementujeme vlastní řešení.

1.1.1 Hry kompletně blokové

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 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 zpracován i hráčův charakter (5), tedy postava, kterou hráč přímo ovládá.



Obrázek 1.1: Hra Minecraft - hrad na skále

Mezi dalšími hrami bychom mohli zmínit například *Terraria*. Ta je o něco mladší než *Minecraft*, ale je častým zdrojem diskusí, zda je lepší new *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. 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 útoku [1]. *Minecraft* je pak orientován spíše na stavění. (Porovnání *Minecraft* vs *Terraria* (facts) [2] na *Minecraft*ovém fóru.)

1.1.2 Hry s prvky realismu

Mezi hry s prvky realismu bychom mohli zařadit třeba hry *Space Engineers* či *Medieval Engineers*, využívají kombinaci herních bloků s *voxelovou* reprezentací světa. Obě hry jsou implementovány v proprietárním enginu společnosti Keen Software House nazvaném *VRAGE™*. Voxelový terén je pak v enginu za běhu hry procedurálně generován do polygonální reprezentace, kterou pak grafická karta standardním způsobem vykreslí na obrazovce (Oficiální popis vlastností enginu [3]). Během tohoto procedurálního vytváření je na třídimenzionální strukturu voxelů (které si pak můžeme představit jako bloky stejné velikosti) aplikován nějaký šum tím je možné ve hře vygenerovat prakticky neomezené množství různých objektů vycházejících z jedné voxelové struktury. „The “procedural asteroids” feature adds a practically infinite number of asteroids to the game world“ [4].

bn b je jednotka grafické informace, která definuje bod v třídimenzionálním prostoru. (TODO zdroj?) Hra pak algoritmicky renderuje polygonální reprezentaci tohoto objemu, přičemž může a nemusí přesně dodržovat hranice voxelu. Tímto způsobem je možné dosáhnout vyššího stupně realismu ve hře.

Podívejme se na obrázek 1.2 převzatý ze serveru Gamespot [5]. Na něm můžeme vidět asteroid (správněji spíše planetku). Na planetce se nachází zkonstruovaná vesmírná základna (obarvená zelenou barvou). K základně je přistavena

větší vesmírná loď (modrobílá), hráč pak k základně letí v další, malé lodi (modrobílá uprostřed). Bližší pohled na planetku ukazuje, že její povrch ani zdaleka není kostičkový. To je způsobeno právě algoritmickou aproximací voxelové reprezentace planetky.

Samotná základna i vesmírná plavidla (detailní pohled na jiné plavidlo je na obrázku 1.3) jsou však tvořeny bloky. Ve hře však vizuální reprezentace nemusí být vždy tvaru krychle. *Space Engineers* umožňuje stavět pohyblivé stroje, které si hráč postaví z herních bloků a ty se pak chovají jako jedna entita. Stále je na ně však aplikována fyzika, takže je možné plavidlo poškodit, nebo dokonce zničit. Tento stupeň realismu od naší hry vyžadovat nebudeme.



Obrázek 1.2: Hra Space Engineers - základna na asteroidu



Obrázek 1.3: Hra Space Engineers - vesmírná loď

1.1.3 Hry s maximálním důrazem na simulaci reality

Do této sekce bychom měli zařadit například vesmírný simulátor *Take on Mars*. // TODO popis, obrázek

bloky na stavění, ale třeba vozidla kompletní

1.1.4 Ostatní - zařadit TODO

Můžeme však nalézt i další příklady her (// TODO *Take on Mars*, *Novus Inceptio*, *Planet Nomads*, *ARK Survival Evolved*, *No man's sky*).

1.2 Čemu se budeme věnovat

Rádi bychom zachovali koncept použití herních bloků, který shledáváme jednoduchý na pochopení i použití. Zaměříme se na rozšíření možnosti práce s bloky tak, abychom uživateli nabídli, pokud možno, ještě lepší herní zážitek ze stavění vlastních výtvorů. 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 neměnný. (*Space Engineers* definuje více velikostí – ty však nelze vzájemně kombinovat). 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 uveďme některé výtvořky ze hry *Minecraft* – město Královo přístaviště z knih

Píseň ledu a ohně od Geoge R. R. Martina, nebo hlavní město Gondoru Minas Tirith z knih Pána prstenů od J. R. R. Tolkiena.

Autoři těchto výtvorů museli volit takové měřítko, aby byly výtvoř dostatečně detailní, ale zároveň aby bylo možné výtvoř postavit v nějakém rozumném čase. Obecně můžeme říct, že čím větších detailů chtějí autoři ve hře *Minecraft* dosáhnout, tím větší musí celý výtvoř být. To pak ale znamená, že celá stavba trvá déle, nebo je zapotřebí více spolupracujících hráčů. Hra *Space Engineers* díky svému přístupu a více bloků, které nejsou tvaru krychle, nabízí lepší možnosti staveb rozsáhlých objektů (představme si třeba Hvězdu smrti z Hvězdných válek), ale stále je potřeba volit nějakou rozumnou výslednou velikost.

1.3.1 Náš návrh úpravy

Chtěli bychom se v této práci zabývat myšlenkou proměnlivé velikosti stavitelných bloků. Tím by hráči mohli rychleji stavět rozsáhlejší 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.

1.4 Inventář

Dalším společným prvkem tohoto druhu her je inventář bloků, které může hráč umístit do herního světa. Hráč přes celé herní okno vidí *HUD* (Head-Up Display [6]), 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) umisťovat do příslušných pozic.

1.4.1 Náš návrh úpravy

Rádi bychom navrhli jiný způsob správy těchto inventárních skupin, 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 by měla naplnit následující cíle:

- Navrhnout a implementovat způsob řešení proměnlivé velikosti bloků
- Navrhnout a implementovat automatizovanou správu inventáře
- Kvůli očekávaným nárokům na pochopení nových konceptů do hry implementovat výukový tutoriál (TODO má to být tady?)
- Získat a zhodnotit zpětnou vazbu 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 specifikaci toho, jak by naše hra mohla vypadat a co všechno by měla umět.

// TODO remove me: Úkol zněl jasně: Cílem bakalářské práce je implementace budovatelské hry se strategickými prvky, hranou z pohledu třetí osoby. Hra se odehrává na nehostinné planetě, kde je hráčův úhlavní nepřítel nedostatek zdrojů a superkyselé deště. Hráč začíná v menší budově – zbytek přistávacího modulu kosmické lodi. Dochází mu elektrická energie i kyslík a je na hráči, aby takticky využíval dostupné zdroje, hledal nové možnosti výroby energie a přežil kyselé bouře. Cílem práce není vytvořit dohratelnou hru, spíše proof-of-concept, zda je tento typ hry s uvedenými mechanikami zábavný a má smysl v jejím vývoji pokračovat i nadále.

2.1 Stávající implementace mechanismů

V následujících podkapitolách si rozebereme jednotlivé části her a jak je implementují ostatní.

2.1.1 Bloky

Bloky, respektive jejich vizuální reprezentace, nemusí být vždy ve tvaru krychle. (TODO reference na MC, SE) Stále však budeme blok chápat jako objekt, který je umístěn v ortogonální mřížce 3D prostoru a beze zbytku tento prostor vyplňuje. Nebudeme se tedy zabývat multibloky (reference na ME)

různé druhy, velikosti, jejich vizuální reprezentace, rozšiřovatelnost, obecně co všechno by měly umět

Základní vlastnosti

Chceme navrhnout systém, ve kterém bude nejmenší blok o hraně 20 cm, tedy objemu odpovídající $0,008 \text{ m}^3$. Tento blok nazveme jako jednotkový. 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.

Součásti bloků

- TODO zmínit MC má readstone, popsat základy chování.
- TODO komponenty elektřiny, inventáře

2.1.2 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.

2.1.3 Skládání bloků do struktur

Chceme hráči umožnit postavení komplexní struktury bloků, která bude dohromady dávat nějaký speciální význam. V našem případě to bude konstruktor objektů, díky kterému za pomoci bloku *B1* – terminálu – 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 Blueprintů). Bloky jednou vymyšlené pak hráč bude moci stavět libovolně mnohokrát, jen musí mít dostatečnou zásobu energie pro jejich postavení.

2.1.4 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.

Speciality

Multiblocks, náhled inventáře (*Medieval Engineers* - stůl a jídlo), conveyor system SE, MC složité, ale dá se vyřešit módy, propagace kyslíku ME

2.1.5 Herní svět

jaký je herní svět

Reprezentace

MC - bloky, chunks, SE + ME planety

Bloky v herním světě

do gridu, start-free grid

Denní / noční cyklus

obvykle tam je, MC 20minut. My zkusíme 30 minut (zkusili jsme 60 minut, ale ukázalo se to jako příliš dlouhá doba - brzy nebylo co dělat kvůli malé nabídce bloků).

Herní překážky

počasí, *NPC*, atributy avataru

(Ne)fyzikální chování

MC - bloky stojí ve vzduchu, ale třeba písek při updatu začne padat

2.1.6 Inventář

mc pevné sloty, SE skupiny slotů.
neomezíme váhově ani jinak

2.1.7 Avatar hráče

avatar má nějaké vlastnosti, *HUD*, 1st / 3rd person view, zdraví, stamina, hlad, O₂

2.2 Co bychom chtěli implementovat

V následujících podkapitolách si rozebereme naše požadavky na hru

2.2.1 Bloky

TODO Většina bloků je stejně velká a má hranu o délce 1 metru [7], [8].
(TODO přesunout do detailní analýzy. TODO popisek odkazu)

V současné době jsou velikosti bloků omezeny na konstantné velikost. Ve hře Minecraft je blok hranově omezen na 1m, hra Space Engineers bloky omezuje dle kategorií od 0.5 m do 2.5 m [9].

různé druhy, velikosti, jejich vizuální reprezentace, rozšiřovatelnost, obecně co všechno by měly umět.

Název - Min - Max - Pitch - Roll - Type (kostka, zkosený, roh, vlastní)

Tam kde Min == Max -> Vlastní škálování

Typ ovlivňuje další chování

Třeba u Světla by typ mohl být i K a hra by se chovala stejně, $K = 1$, $Z = 0.5$, $R = 1/6$, $V = 1$ (není v potaz objem)

komponenty bloků a nějaké další ptákoviny

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

2.2.2 Podrobný popis bloků

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

A1 - Blok základny

- velikost v ose Z omezena na 4 základní bloky
- má elektriku

Pokud bychom měli nerovný terén, tento blok by mohl zahrnovat podstavce pro vyrovnání terénu.

A2 - Blok stavby

- všechny velikosti
- má elektriku

Tento blok je základním stavebním blokem ve hře.

A3 - Blok polykarbonátu

- všechny velikosti Tento blok je nejlevnější, není připojen do elektrické sítě. 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 a tím se ve světě posune o 1 metr výš.

A4 - Zkosený blok základny

- velikost v ose Z omezena na 4 základní bloky
- má elektriku

Stejně jako blok A1, jen je zkosený. Může sloužit jako přístupová rampa.

A5 - Zkosený blok stavby

- všechny velikosti
- má elektriku

A6 - Roh bloku stavby

- všechny velikosti
- má elektriku

B1 - Terminál

- speciální, pevná velikost 1 x 8 x 5 bloků
- má elektriku, konzument, rychlé doplnění energie, ovládání rozhraní, komplexní přehled připojené elektrické sítě.

B2 - Napájené okno

- minimální velikost 2 x 1 x 2, maximální velikost 20 x 1 x 20 základních bloků
- má elektriku, konzument

B3 - Dveře

- speciální, pevná velikost 7 x 7 x 11 bloků
- má elektriku, otevírání

B4 - Světlo

- velikost omezena na 1 x 1 x 1 blok
- má elektriku, konzument, ovládání bez přepínače

B5 - Přepínač

- velikost omezena na 1 x 1 x 1 blok
- má elektriku, náhled stavu

B6 - Generátor energie

- omezená velikost v ose Z na 2 bloky, jinak 3 x 3 až 20 x 20 v ostatních osách
- má elektriku, producent

B7 - Generátor objektů

- omezená velikost v ose Z na 2 bloky, jinak 3 x 3 až 20 x 20 v ostatních osách
- má elektriku, konzument

B8 - Akumulátor

- speciální, pevná velikost 3 x 3 x 3 bloků
- má elektriku, producent, konzument, rychlý náhled naplnění

B9 - Plnička kyslíkových bomb

- speciální, pevná velikost 4 x 3 x 4 bloků
- má elektriku, kyslíkovou komponentu, konzument, UI, rychlé doplnění kyslíku
- využijeme ideu náhledu inventáře a plnička bude zobrazovat blok B10, pokud bude nějaký takový blok plnit.

B10 - Kyslíková bomba

- speciální, pevná velikost 2 x 2 x 2 bloků
- má kyslíkovou komponentu, možnost sebrat, rychlý náhled naplnění, rychlé doplnění kyslíku

2.2.3 Herní svět

jaký chceme herní svět

Reprezentace

bude nám stačit nějaký tree, definovat rozměry, na chunky kašlem

Bloky v herním světě

do gridu

Denní / noční cyklus

dáme ho

Herní překážky

počasí, atributy avataru

(Ne)fyzikální chování

nebudeme hrotit

2.2.4 Inventář

chceme volné sloty, rozšiřitelnost

2.2.5 Avatar hráče

avatar má nějaké vlastnosti, HUD, 1st / 3rd person view, zdraví, O2, energie

2.3 Herní nepřítel

Protože samotné stavění bez nějakého cíle či překážky není úplně zábavné, musíme hráči připravit nějakou překážku, komplikaci, kterou musí překonávat. Zde neexistuje jednoznačné řešení — to je závislé na celkovém prostředí hry, zamýšlené cílové skupině a mnoha dalších faktorech. Cílem našeho hráče bude přežít kyselé deště. Ty 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.

2.4 Backlog

???

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 moci 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
- Podpora zvuků
- Snadná implementace lokalizace
- Správa assetů
- Správa scény

Pro další případný rozvoj bychom potřebovali:

- Podpora pathfindingu
- Podpora síťové hry
- Podpora AI

Cílové 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.

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 využíváný. Ačkoliv zkušenost s tímto programovacím jazykem máme minimální, můžeme se tímto způsobem naučit novým dovednostem.

Možných použitých engineů a frameworků je opravdu mnoho. Podívat do databáze herních engineů na stránce Devmaster. Jen zde je možné nalézt 236 možných řešení našeho problému volby herního engineu [10]. Všechny záznamy jsme omezili na *vývojově aktivní*, v jazycích **C#**, **C++** a vybrali jsme námi požadované vlastnosti.

Mezi čím tedy můžeme volit?

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

- Použit existující herní engine

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 engine. Naším cílem je prototyp hry a tudíž nechceme ztrácet 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. Buď můžeme využít engine jako třeba *Ogre* (C++), nebo použít více robustnější řešení v podobě engineů typu *Unity* (C#) či *Unreal Engine* (C++). 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. Tudíž engine jako třeba *Ogre* nebudou naší volbou.

Výhodou zmíněných robustních engineů je to, že jsou k dispozici zdarma (oproti třeba *CryEngine*). Taktéž zde, díky práci komunity, existuje pro oba enginey 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.

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 engineu. 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 zrychlena, 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[11]. *Unreal Engine* má tuto funkcionalitu implementovanou ve svém editoru[12].

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 běhe, řešerš 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. (TODO link). 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 řešerše 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* převážily nad nevýhodami i všemi výhodami *Unity*. // TODO tohle chce vyladit

3.2 Bloky

Zde by měl být popis možností jak definovat a následně implementovat bloky. jaké jsou výhody a nevýhody jednotlivých implementací

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

3.3 Vlastnosti bloků

Popis toho, co blok umí

3.3.1 Energie

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

3.3.2 Energetická síť

- způsob zapojení do sítě

3.3.3 Kyslík

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

3.3.4 Označovatelnost

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

3.3.5 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

3.3.6 Interakce

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

3.3.7 Zapojení do rozpoznávání tvarů

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

3.4 Komponenty bloků

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

3.5 Bloky v herním světě

- je více možností. 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 clustorování 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 degeneruje do úrovně níže, případně rozpadne na podčásti a rekurzivně se přidá.
- díky této variantě se můžeme snadno dotazovat na sousedy, což je hlavní cíl (proto)

3.6 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
- popsát ideu BT
- další možnosti by byly, že bychom prostě použili update smyčku nějakého Actora - není potřeba, tohle se vyřeší updatem na komponentě

3.7 Hráčova postava

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

3.8 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)

3.9 Ukládání hry

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

3.10 Doplnující vlastnosti

3.10.1 Lokalizace

- použití lokalizace

3.10.2 Hudba

- atmosférický hudební doprovod

3.11 Backlog

- Popsat, že bychom chtěli nějaké UI + nabídky menu

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 (TODO link!). Použití novější verze je možné, ale běžnému uživateli to nedoporučujeme. Mezi verzemi se mohou projevit nekompatibility v kódu, které je případně nutné řešit zásahy 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).

Dále je zapotřebí vygenerovat solution pravým klikem na uproject file (TODO img!) Pokud tato možnost v kontextové nabídce není, je potřeba provést FIX . Ze zkušenosti autora - toto se mnohdy nemusí podařit. Pokud se nepodaří vygenerovat solution, může stačit otevřít projekt a dát zkompilevat chybějící binárky (TODO img!). Je zapotřebí mít VS 2015 alespoň ve verzi Community.

Pokud i toto selže, ověřte si, prosím, že je možné založit nějaký projekt založený na C++ (todo font), zkompilevat jej a taktéž vygenerovat solution. Pokud se to povede s template, mělo by to fungovat i s tímto projektem.

Dalším krokem je v případě úspěšného otevření ve VS (todo abbreviation) nastavení TCF2 jako výchozího projektu a následné spuštění. Dále by měl následovat krok spuštění Play in Editor v UE.

Pokud vše selže, je možné nalézt příčinu chyby v logu (TODO) v Saved/Logs

4.2 Struktura projektu

Celý projekt je rozdělen do dvou částí - C++ část a Blueprintová část. *Unreal Engine* umožňuje herním vývojářům implementovat celou hru kompletně za pomoci Blueprintů, tedy vizuálního rozhraní. Toho se však obvykle nevyužívá, protože vykonávání programu v Blueprintu je přibližně 10 krát pomalejší, než vykonávání nativního C++ kódu (TODO link). V praxi tak dochází k tomu, že vývojář (i neprogramátor) může za pomoci 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 pomoci 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.

V dalším textu tedy postupně probereme prvně kódovou část napsanou v C++ a poté strukturu projektu s samotným *Unreal Engine*. 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 kódu

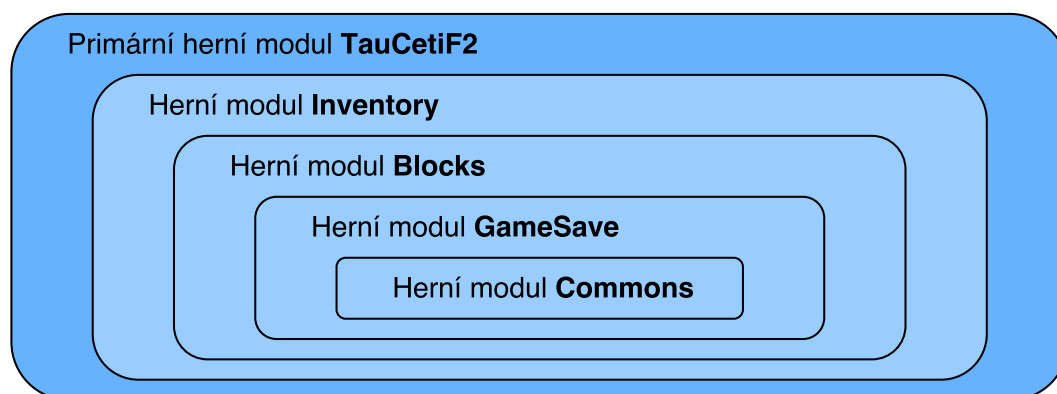
Protože jsme zvolili implementaci práce v *Unreal Engine*, můžeme využít toho, že engine umožňuje rozdělit celý herní projekt do jednotlivých herních modulů[13]. Tím docílíme modularity, nebudeme mít celý projekt v jednom kuse a zároveň tím urychlíme překlad projektu při kompilaci.

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 4.1:



Obrázek 4.1: Diagram závislostí modulů projektu.

4.3.1 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 použití herních modulů v rámci *Unreal Engine*

4.4 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.

4.4.1 Herní definice a konstanty

`GameDefinitions.h`

Tady 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, zdravím a jednotkou zásahu kyselého deště. Taktéž jsou zde definovány konstanty, které využívá technika obrysů objektu (todo link na outline). Dále jsou zde definovány konstanty ID implementovaných bloků, abychom s nimi mohli pracovat i v kódu.

4.4.2 Herní instance

`TCF2GameInstance.h`

Tato třída je Singleton ([TODO link UE docs](#)) a jako jediná zůstává vždy stejná po celou dobu běhu hry. 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 budu 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 projektem, 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 [14]). Během procesu vytváření nové instance objektu, který vychází z daného Blueprintu pak budou tyto hodnoty naplněny během fáze inicializace properties (popis životního cyklu actorů [15]). 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.

4.4.3 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.

4.4.4 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 **EGameUserSettingsVar**. 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.

4.5 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 rozšiřování 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 savy 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.

4.5.1 GameSaveInterface

`UGameSaveInterface.h` Tento soubor definuje rozhraní, které je možné implementovat a používat v Blueprintech (struktura vychází z tutoriálu na Unreal Engine Wiki[16]). 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 funkcionálna bude popsána v Blueprintové části.

4.5.2 FFileVisitor

`FFileVisitor.h`

Tento soubor obsahuje implementaci návrhového vzoru Visitor pro získání všech herních savů z nějaké zadané složky. Implementace vychází z internetové diskuse na téma procházení adresářů [17].

4.5.3 Helpers

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

4.5.4 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 zkopírovat 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.

Přepравka 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;
}
```

Třídou dědicí z `UObjektu` jsme nahradili strukturou `FBlockInfo`, která sama slouží pouze jako přepравka 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 přepравky se stará pouze modul `GameSave`.

Celý systém vychází z tutoriálu [18] 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 jediný zápis jedné funkce. To je výhodné, protože to předchází chybám,

kteře 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.

4.5.5 NewGameSaveHolder

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

4.6 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`)

4.6.1 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 rekompile 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)`

4.6.2 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.

4.6.3 Ukládání a načítání bloků

- ukládání - máme něco jako block saving helpers

4.6.4 Interfaces

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

4.6.5 Komponenty bloků

- pak máme komponenty bloků a nějaké interfaces

Elektrická komponenta

Elektrická síť

Kyslíková komponenta

Select target

World object

4.6.6 Implementace bloků

- základ Block.h, zbytek v jednotlivých 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 výžpovědní hodnota je ve zdrojácích a není asi nutné to tu duplikovat

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

4.6.7 Stromové struktury

popsat stromové struktury, které tam mám

MinMaxBox

prapředek všeho

KDTree

dědí z MMB, základ ve světě

WeatherTargetsKDTTree

dědí z MMB, slouží pro potřeby počasí

4.7 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 rozšiřovat jak by rostla komplexita správy inventáře.

Nejdůležitější inventory component

4.7.1 Tag group

nejnižší úroveň, odpovídá 'nebo'

4.7.2 Inventory tag group

celá skupina, odpovídá 'A zároveň'

4.7.3 Inventory tags

sdružuje všechny banky

4.7.4 Inventory component

celá komponenta, která je pak navázaná na hráčův charakter

definuje delegáty notifikující o změnách v aktivní 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)

4.8 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)

4.9 Struktura projektu v Unreal Engine

- ukázat jak se to dělí v UE editoru

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, implementaci 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

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

Softwarové 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

- dynamičtější 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. URL <http://www.urbandictionary.com/define.php?term=boss%20fight>.
- [2] dillonsup. Minecraft vs terraria (facts). URL <http://www.minecraftforum.net/forums/minecraft-discussion/discussion/178129-minecraft-vs-terraria-facts>. internetové fórum.
- [3] Keen Software House. Vrage. URL <http://www.keenswh.com/vrage.html>.
- [4] Marek Rosa. Space engineers: Super-large worlds, procedural asteroids and exploration. URL http://blog.marekrosa.org/2014/12/space-engineers-super-large-worlds_17.html.
- [5] GameSpot. Space engineers. URL <https://static.gamespot.com/uploads/original/1365/13658182/2626082-space-engineers1.jpg>.
- [6] 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>.
- [7] Minecraft Wiki. Block, . URL <http://minecraft.gamepedia.com/Block>.
- [8] Minecraft Wiki. Tutorials/units of measure, . URL http://minecraft.gamepedia.com/Tutorials/Units_of_measure.
- [9] Space Engineers WIKI. Blocks. URL <http://spaceengineers.wikia.com/wiki/Category:Blocks>.
- [10] Devmaster.net. Game engines (filtered). 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.
- [11] Unity. Localization manager tutorial, 2017. URL <https://unity3d.com/learn/tutorials/topics/scripting/localization-manager>.
- [12] Epic Games. Localization, 2017. URL <https://docs.unrealengine.com/latest/INT/Gameplay/Localization/>.
- [13] Epic Games. Gameplay modules, . URL <https://docs.unrealengine.com/latest/INT/Programming/Modules/Gameplay/>.
- [14] Epic Games. What is cdo?, . URL <https://answers.unrealengine.com/questions/191353/what-is-cdo.html>.

- [15] Epic Games. Actor lifecycle, . URL <https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/ActorLifecycle/>.
- [16] Epic Games. How to make c++ interfaces implementable by blueprints (tutorial), 2017. URL [https://wiki.unrealengine.com/How_To_Make_C%2B%2B_Interfaces_Implementable_By_Blueprints\(Tutorial\)](https://wiki.unrealengine.com/How_To_Make_C%2B%2B_Interfaces_Implementable_By_Blueprints(Tutorial)).
- [17] Lions Den. Iteratedirectory questions. URL <https://forums.unrealengine.com/showthread.php?48881-IterateDirectory-Questions>.
- [18] Rama. Save system, read & write any data to compressed binary files. URL https://wiki.unrealengine.com/Save_System,_Read_%26_Write_Any_Data_to_Compressed_Binary_Files.

Přílohy