MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Martin Galajda

Platforma pro výzkum prostorové orientace ve virtuální realitě

Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Michal Bída, Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové systémy

Praha 2018

Na tomto mieste by som sa rád poďakoval vedúcemu diplomovej práce Mgr. Michalovi Bídovi, Ph.D. . Ďalej ďakujem Ph.D, Mgr. Kamilovi Vlčkovi, Ph.D a Mgr. Ivete Fajnerovej, Ph.D, z oddelenia neurofyziológie pamäti AV ČR za ich rady, pripomienky a konzultácie. V neposlednom rade aj rodičom, Mgr. Kamilovi Vasilíkovi, Michalovi Pekarčíkovi a Nelli Echtnerovej za podporu, pochopenie a priestor, ktorý mi poskytli pre napísanie tejto práce.

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

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

V …...... dne............ podpis

Název práce: Platforma pro výzkum prostorové orientace ve virtuální realitě

Autor: Martin Galajda

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Michal Bída, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Na oddelení neurofyziológie pamäti Akadémie Vied Českej Republiky (UAV), už niekoľko rokov prebieha výskum experimentov sledujúcich dopad rôznych štádií Alzheimerovej choroby na kognitívne schopnosti pacientov. K tomuto účelu sa používa systém SpaNav, vyvinutý pred takmer 9 rokmi v spolupráci UAV s MFF UK. V súčasnosti spomínaný systém naráža na technické možnosti dané dobou v ktorej vznikol. Cieľom tejto práce je priniesť nový systém, ktorý odstráni limity pôvodného systému a umožní pokračovať vo výskume.

Klíčová slova: Orientácia v priestore, Unreal Engine 4, virtuálna realita

Title: Platform for spatial navigation research in virtual reality

Author: Martin Galajda

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Michal Bída, Ph.D., Department of Software and Computer Science Education

Abstract: The Department of Neurophysiology of the Memory of the Academy of Sciences of the Czech Republic (UAV) has for several years been researching experiments to monitor the impact of various stages of Alzheimer's disease on patients' cognitive abilities. For this purpose, SpaNav was developed almost 9 years ago in cooperation of the UAV with the MFF UK. The system currently hits its technical limits given by possibilities of the time when it was created. The aim of this work is to introduce a new system that will remove the limits of the original system and allow research to continue.

Keywords: spatial navigation, Unreal Engine 4, virtual reality

Názov práce: Platforma pro výzkum prostorové orientace ve virtuální realitě

Autor: Martin Galajda

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedúci diplomovej práce: Mgr. Michal Bída Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Na oddelení neurofyziológie pamäti Akadémie Vied Českej Republiky (UAV), už niekoľko rokov prebieha výskum experimentov sledujúcich dopad rôznych štádií Alzheimerovej choroby na kognitívne schopnosti pacientov. K tomuto účelu sa používa systém SpaNav, vyvinutý pred takmer 9 rokmi v spolupráci UAV s MFF UK. V súčasnosti spomínaný systém naráža na technické možnosti dané dobou v ktorej vznikol. Cieľom tejto práce je priniesť nový systém, ktorý odstráni limity pôvodného systému a umožní pokračovať vo výskume.

Kľúčové slová: Orientácia v priestore, Unreal Engine 4, virtuálna realita

Obsah

Úvod ... 1
Diagnostika Alzheimerovej choroby ... 1
Kognitívne testovanie ... 2
Obsah textu ... 3
Analýza problému ... 4
Experiment AAPA ... 4
Výstup a vyhodnotenie ... 5
SpaNav 1.0 ... 7
Problém so zaznamenávaním času ... 8
Unreal engine ... 10
SpaNav 2.0 ... 12
Architektúra ... 12
Editor ... 14
SLAVE ... 15
CONTROLLER ... 16
Použité technológie ... 17
Unreal Engine 4 ... 17
Javascript a Google V8 ... 18
C# a Visual Studio IDE ... 19
Configuration management ... 19
Projektový management ... 20
Programátorská dokumentácia ... 21
CONTROLLER ... 21
Funkcie CONTROLLERu ... 21
Implementácia CONTROLLERu ... 22
SLAVE ... 24
ScriptEngineActor ... 24
Experiment ... 27
TCP/IP modul ... 28
Skript Engine ... 29
C++ do Skriptovacieho enginu ... 31
Logovanie ... 33
Časovanie ... 33
Zakázané a cieľové oblasti ... 33
Postava testovaného subjektu ... 35
Zvuky ... 35
HUD ... 37
Náhľadový systém ... 40
Úvodná obrazovka (multiplayer setup) ... 44
Užívateľská dokumentácia ... 48
CONTROLLER ... 48
EDITOR ... 50
MyScriptedActor ... 51
Skriptovanie ... 61
Direktíva Include ... 62
Registrácia udalostí ... 62
Registrácia načasovaných úloh ... 63
Logovanie ... 64

Získanie údajov o polohe a rotácii 65
Zvuky 65
Matematické funkcie 66
SpaNav 1.0 vs. SpaNav 2.0 67
Testovanie SpaNav 1.0 67
Testovanie SpaNav 2.0 69
Záver 71
Použitá literatúra 73
Príloha 75
Blueprints 75
Refenčná príručka Javascript + SpaNav 2.0 78

Úvod

Súčasná medicína má snahu vysporiadať sa s mnohými výzvami, ktoré môžu znižovať kvalitu života moderného človeka. Najväčší dopad typicky majú degeneratívne poruchy organizmu, ktoré sú často ťažko liečiteľné. Všeobecne panuje názor, že kľúčovým faktorom pre prístup k pacientovi a to nielen u degeneratívnych porúch býva štádium, v ktorom sa choroba nachádza, keď dôjde k jej odhaleniu. Obzvlášť ak nemáme možnosti poruchu liečiť, ale poznáme spôsoby, ako zamedziť jej zhoršovaniu. Medzi takéto poruchy patrí Alzheimerova choroba. Včasná diagnostika choroby, má nielen momentálny význam pre pacienta, ale umožňuje nám študovať vývoj choroby v jej zárodku a potenciálne tak pripraviť úspešnejšiu liečbu a diagnostiku pre ďalších pacientov v budúcnosti. Predmetom tejto práce je vytvoriť systém, ktorý by pomáhal pri diagnostike Alzheimerovej choroby známymi prostriedkami, a zároveň mal vlastnosti, ktoré by mu umožňovali rozširovať základnú sadu testov a možno tak jedného dňa prispieť ku včasnejšej a kvalitnejšej diagnostike.

Diagnostika Alzheimerovej choroby

Alzheimerova choroba je neurodegeneratívna porucha, ktorá vedie ku strate kognitívnych funkcií človeka, eventuálne k demencii. Charakteristikou poruchy je degenerácia časti mozgu, konkrétne sa jedná o hippocampus a cortex. Degradácia je zapríčinená chemickými procesmi v mozgu. Povaha týchto chemických procesov sa postupom času v ľudskom mozgu mení. Posun zmeny procesov v mozgu vedúcich k poruche popisuje štatistika, ktorá tvrdí [9], že po 65 roku života sa riziko vzniku Alzheimerovej choroby každých 5 rokov zdvojnásobí. Ďalšia štúdia tvrdí, že iba v Británii bude do roku 2050, 25% ľudí starších než 65 rokov a až 10 percent ľudí bude starších než 80 rokov [10]. To znamená, že až štvrtina populácie bude potenciálne ohrozená Alzheimerovou chorobou, z ktorých časť bude vo veľkom riziku. V súčasnosti sú známe spôsoby ako s chorobou bojovať. Liečbou známymi prostriedkami pozorujeme zlepšenie kognitívnych schopností u pacientov, avšak ani jeden spôsob liečby nedokáže zastaviť postup ochorenia. Máme teda pred sebou

problém, u ktorého očakávame prudký nárast počtu výskytov, a ktorý zatiaľ nedokážeme riešiť.

V dnešnej dobe neexistuje jediný komplexný test, ktorý by dokázal potvrdiť alebo vylúčiť skoré štádium Alzheimerovej poruchy u pacienta. Na diagnostiku sa používa súbor testov v nasledujúcich oblastiach:

- História pacienta — ďalšie ochorenia, lieky, výskyt poruchy v rodine, konzumácia alkoholu
- Fyzické vyšetrenie – tlak, tep, vzorky krvi a moču
- Neurologické vyšetrenie – reflexy, koordinácia, pohyby očí
- Vyšetrenie mentálneho stavu pacienta – pamäť, schopnosť riešiť jednoduché úlohy
- Skeny mozgu pacienta – Používa sa najmä, aby boli vylúčené ostatné poruchy, ktoré by mohli mať ovplyvniť výsledky testovania prítomnosti Alzheimerovej poruchy.

Kognitívne testovanie

Výskum v ústave Akadémie vied Českej Republiky sa zaoberá vybranými časťami diagnostického procesu, konkrétne kognitívnymi testami. Príprava kognitívnych testov ale býva typicky náročná na čas a zdroje. Taktiež je zložité pripraviť experiment tak, aby bol opakovateľný vždy rovnakým spôsobom. Niektoré navrhnuté experimenty navyše nie sú v prostredí reálneho sveta vhodné pre vyšetrovanie ľudí. Experiment vo vodnom bludisku napríklad predpokladá plávajúcu vodnú plošinu a bazén v miestnosti so zhasnutým svetlom. Cieľom testovaného subjektu je zapamätať si polohu plošiny a neskôr ju za pomoci niekoľkých navigačných bodov po tme nájsť. Čitateľovi je zrejme jasné, že podobný experiment na ľuďoch, obzvlášť v staršom veku, je nevhodný a nebezpečný. V dnešnej dobe aj v dobe nedávno minulej sa naskytuje možnosť previesť testy do prostredia virtuálnej reality, ktorá rieši takmer všetky spomínané problémy. Existujú navyše štúdie, ktoré sa zaoberajú účinnosťou testovania kognitívnych experimentov na ľuďoch v prostredí virtuálnej reality (VR), priamo spojených s diagnostikou Alzheimerovej choroby. Testujú sa výsledky experimentov na pacientoch s Alzheimerovou chorobou a sledujú sa korelácie výsledkov oboch typov testov. Štúdie [12] ukazujú, že kognitívne testy je možné skutočne bez straty informácie vykonávať aj v prostredí

virtuálnej reality. Na základe týchto poznatkov vznikol systém SpaNav, ktorému sa venuje naša práca.

Obsah textu

V úvode práce si popíšeme, ako funguje jeden z experimentov používaných v praxi. Ukážeme si spôsob, ktorým sa experimenty vyhodnocujú a z toho poznáme, ktoré aspekty aplikácie sú kritické pre úspešné testovanie pacientov. Pozrieme sa na súčasný stav projektu, a ako si stojí proti požiadavkám, ktoré sme pomenovali ako najkritickejšie. Ukážeme si, že súčasný systém bojuje s niekoľkými problémami, a tiež si povieme, prečo tieto problémy už nemožno ďalej riešiť v prostredí súčasného systému.

Ukážeme si návrh možného riešenia. Popíšeme si teoretický model architektúry, použitím ktorého očakávame zlepšenie momentálneho stavu. Predstavíme si jednotlivé technológie, ktoré sme pri implementácii použili. Pouvažujeme o ich výhodách aj nevýhodách a zamyslíme sa nad tým, ako nám pomôžu pri plnení našej úlohy.

V najrozsiahlejšej časti práce, programátorskej dokumentácii si detailne popíšeme programátorskú cestu, ktorá nás priviedla k prostrediu SpaNav novej generácie. Popíšeme všetky dôležité architektonické, dizajnové a implementačné postupy použité pri programovaní.

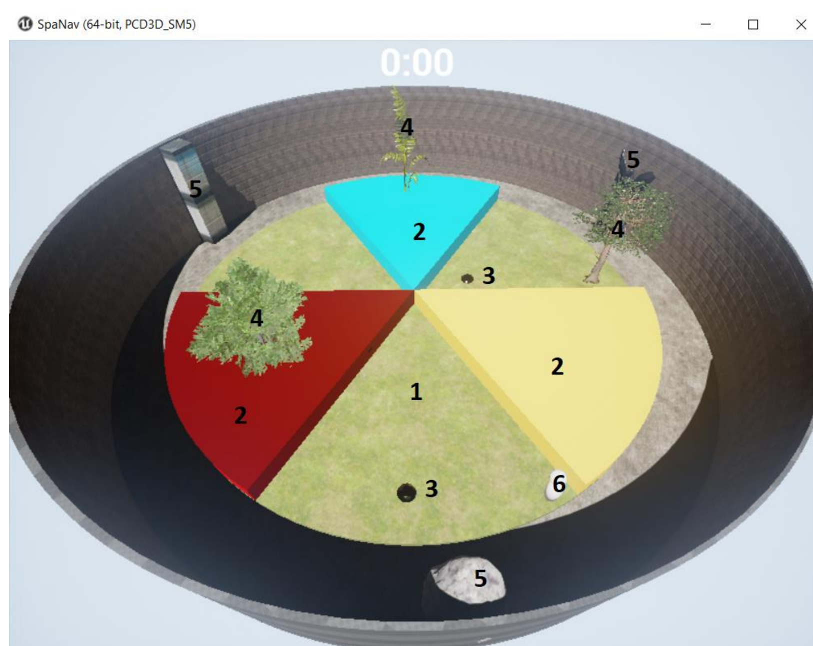
Následne sa prenesieme do roviny užívateľa a rozoberieme implementáciu experimentu dodaného s naším systémom. Touto cestou ukážeme konštrukciu experimentu v editore UE, ktorý sme obohatili o potrebné funkcie. Pozrieme sa na všetky dôležité komponenty, ktoré dodaný experiment používa. V časti venovanej skriptovanému kódu si prejdeme možnosti skriptovania, pripravené pre dodaný experiment. Vysvetlíme si, ako registrovať funkcie pre obsluhu udalostí a načasovaných úloh.

V závere práce sa pozrieme na výsledok nášho snaženia. Popíšeme si testy použité pre testovanie oboch verzií systému. Pomocou grafov vizualizovaných z dát získaných pri týchto testoch, porovnáme výkonnosť aplikácií s ohľadom na kľúčové aspekty popísané v úvode. Z tohto porovnania bude zrejmé, že nová verzia systému plní úvodné očakávania, vo všetkých navrhnutých oblastiach a v plnej miere.

Analýza problému

Experiment AAPA

Pre lepšie pochopenie tejto práce je dôležité, poznať ako popísané testovanie vyzerá v praxi. Jeden z experimentov v pôvodnom systéme a zároveň experiment dodaný s novou verziou práce sa nazýva *AAPA*, ktorý vychádza z iného experimentu, nazvaného *Blue Velvet Arena*. Popisu Blue Velvet Arena – testu sa venuje pôvodná práca, ktorá bola predlohou tej našej, od autorky Mgr. Šupalovej. Jedná sa o principiálne rovnaké experimenty s tým rozdielom, že AAPA existuje vo virtuálnej realite, ale Blue Velvet Arena je skutočne skonštruovaný experiment v reálnom svete.



Obrázok 1: AAPA experiment. 1-otáčajúca sa platforma, 2-zakázané oblasti, 3-cieľové oblasti, 4-pohyblivé orientačné body, 5-statické orientačné body, 6-postava testovaného subjektu

Experiment AAPA spočíva v pohybe testovaného subjektu vo vnútri kruhovej arény, pričom má plniť zadané úlohy. Jednou úlohou testovaného subjektu, je hľadať a zbierať náhodne rozložené diamanty (cieľové oblasti). Zároveň však musí subjekt plniť druhú úlohu, a tou je vyhýbanie sa zakázaným oblastiam. Zakázané oblasti sú vo viacerých fázach experimentu neviditeľné, no subjekt je o vstupe do oblasti upovedomený zvukovým znamením a dĺžka pobytu v zakázaných oblastiach je

meraná a zobrazená na terminále. Počas celého experimentu sa podstava arény otáča. Táto rotácia má za úlohu miasť testovaný subjekt.

Zároveň sú v aréne rozložené viditeľné orientačné body. Jedná sa o tri rastliny na otáčajúcej sa platforme a tri kamenné objekty, ktoré sú statické. Tieto objekty majú slúžiť subjektu pre orientáciu a pomoc pri pamätaní si, kde sa nachádza aktuálna zakázaná oblasť.

Experiment pozostáva z viacerých fáz, kde prvá je testovacia a v tejto fáze je zakázaná oblasť viditeľná. V ďalších fázach sú aktivované postupne tri neviditeľné zakázané oblasti na rôznych miestach. Tieto fáze sa neotáčajú s platformou, a preto je pri ich sledovaní kľúčové pozorovať statické kamenné objekty. Posledná fáza obsahuje zakázanú oblasť, ktorá sa otáča s platformou a preto je dôležité sledovať rastliny na platforme, ktoré sú voči tejto oblasti statické.

Do procesu experimentu môže vstupovať externý užívateľ a ovplyvňovať jeho beh. Typicky môže napríklad meniť viditeľnosť zakázaných oblastí.

Počas celého experimentu na pozadí ukladáme správy o priebehu experimentu, ktoré neskôr použijeme pre spätné vyhodnotenie experimentu. Správy sa ukladajú s presnou časovou známkou. Aké dáta sa ukladajú, a ako sa potom tieto dáta vyhodnocujú, ukazuje nasledujúca kapitola.

Výstup a vyhodnotenie

Účelom tejto kapitoly je ukázať spôsob akým sa vyhodnocujú údaje získané počas behu experimentov v prostredí SpaNav. SpaNav nie je jeden konkrétny experiment ale rámec pre beh rôznych experimentov. Vyhodnotenie experimentov má preto mnoho podôb. Pozrieme sa však na to, čo majú existujúce experimenty spoločné a ako do procesu vstupuje nová architektúra SpaNav 2.0 .

Viacero experimentov, pre ktoré sa SpaNav používa sú popísané v článkoch, ktoré vznikli pod záštitou Ivety Fajnerovej, Kamila Vlčka a spol. . V týchto článkoch sú popísané podmienky experimentov, získané údaje, ich vyhodnotenie aj ich interpretácia. Úlohou tejto kapitoly nie je prezentovať výsledky zistené v týchto experimentoch ale priniesť náhľad pre spôsob akým sa spracúvajú dáta. Z vykonaných experimentov je možné vysledovať nasledujúce metriky dôležité pre vyhodnotenie experimentu:

- počet chýb (vstupov do zakázaných oblastí)

- maximálna doba medzi dvoma chybami/mimo chybu
- doba po prvú chybu
- maximálna doba zotrvania v zakázanej oblasti
- efektivita cesty
- doba zotrvania v cieľovej oblasti
- percentuálne vyjadrenie doby zotrvania v zakázanej oblasti ku celkovej dobe experimentu
- celková trasa prejdená počas experimentu

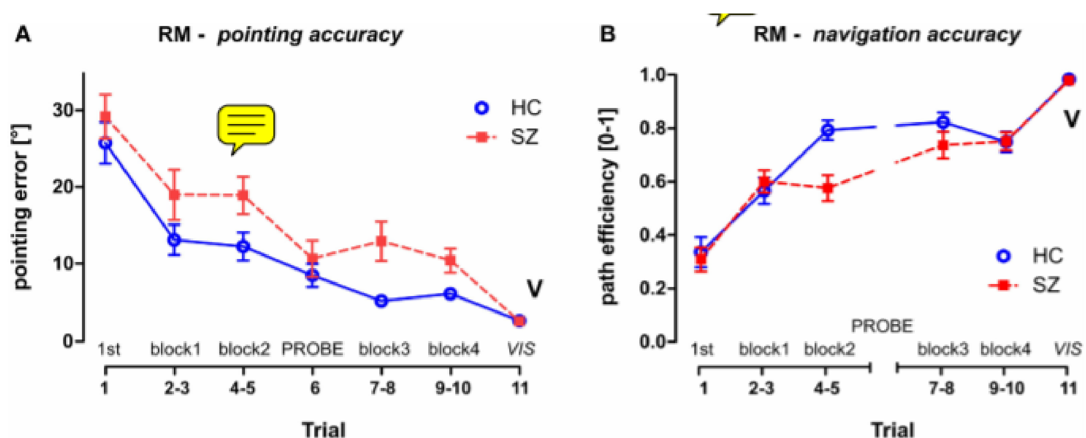
Podľa potreby experimentu sa potom nad nameranými hodnotami štatistickými metódami testujú hypotézy, v závislosti na rôznych premenných vstupujúcich do experimentu.

Príkladom vyhodnotenia experimentu je zistenie rozdielov v hodnotách podľa demografických premenných (vek, dosiahnuté vzdelanie) pomocou bez-parametrického Mann-Whitneyho U Testu.

Tabuľka 1:Výsledok experimentu riadeného demografickými premennými (prevzaté z [5])

	Group mean (*SD*)		Group differences	
Demographic variable	**Schizophrenia patients (SZ)**	**Healthy Controls (HC)**	**Mann-Whitney U**	***p*-value**
N	29	29		
Sex (M: F)	17: 12	17: 12		
Age	25.8 ± 6.2	25.7 ± 5.4	419.5	0.994
Education level (1–6)	3.1 ± 1.6	3.7 ± 1.2	323	0.131
Gaming experience (0–2)	1.1 ± 0.7	0.6 ± 0.5	258	0.012

Ďalším príkladom je GLM analýza s dvoma kategorickými predátormi (skupina x pohlavie)



Graf 1:Vizualizácia výsledku experimentu vyhodnoteného GLM analýzou (prevzaté z [5])

Podkladové dáta pre zobrazené výsledky pochádzajú z časových údajov zaznamenaných počas behu experimentu. Kvalita dát je priamo úmerná ich vernosti voči skutočným udalostiam, najmä teda času, kedy udalosť nastala. V surovej forme tieto dáta môžeme vidieť na obrázkoch nižšie.
Príklad výstupu zo SpaNav:

```
20  [2018.3.11-11:40:15.695][JavscriptEngine]Diamant entered, total: 6
21  [2018.3.11-11:40:15.695][JavscriptEngine]Diamant moved to new position: -259,-289
22  [2018.3.11-11:40:17.649][JavscriptEngine]Avoidance entered
23  [2018.3.11-11:40:18.554][JavscriptEngine]Avoidance left
24  [2018.3.11-11:40:19.359][JavscriptEngine]Avoidance entered
25  [2018.3.11-11:40:21.523][JavscriptEngine]Avoidance left
```

Výstup 1: Záznamy priebehu experimentu z hlavného log súboru

```
25  [2018.3.11-11:40:10.896][JavscriptEngine]Location: [-120.17959594726562;-264.0750732421875;132.19387817382812]
26  [2018.3.11-11:40:10.896][JavscriptEngine]Rotation: [-52.00553512573242;0;0]
27  [2018.3.11-11:40:11.896][JavscriptEngine]Location: [-456.35577392578125;-462.0761413574219;132.19387817382812]
28  [2018.3.11-11:40:11.896][JavscriptEngine]Rotation: [159.73968505859375;0;0]
29  [2018.3.11-11:40:12.896][JavscriptEngine]Location: [-776.2789916992188;-194.5626525878906 2;132.19387817382812]
30  [2018.3.11-11:40:12.896][JavscriptEngine]Rotation: [-168.2599639892578;0;0]
31  [2018.3.11-11:40:13.897][JavscriptEngine]Location: [-655.5169677734375;-560.552978515625;132.19387817382812]
32  [2018.3.11-11:40:13.897][JavscriptEngine]Rotation: [-19.344039916992188;0;0]
```

Výstup 2Záznamy priebehu experimentu z logu pre načasované úlohy

Je zrejmé, že zaznamenané časové údaje sú absolútne kľúčové pre lepšie vyhodnocovanie experimentov. SpaNav 2.0 si kladie za cieľ poskytnúť zlepšenie práve v tejto oblasti testovania.

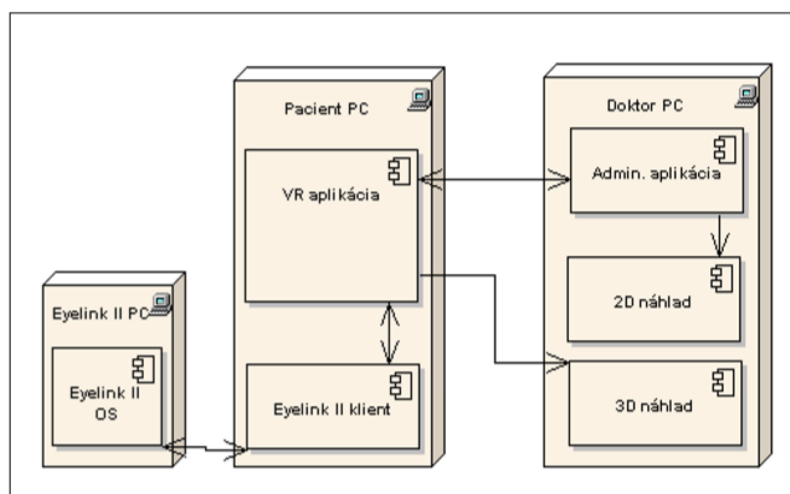
SpaNav 1.0

Obsah úvodnej kapitoly bol známy už pred niekoľkými rokmi kolegom v ústave akadémie vied, aj kolegom na Matematicko-fyzikálnej fakulte Univerzity Karlovej. V roku 2009 bola výsledkom ich spolupráce prvá verzia projektu SpaNav. Autorkou je Ivana Šupalová a práca bol vedená Mgr. Cyrilom Bromom, Ph.D. Na projekte spolupracovali, tak ako je tomu aj našom prípade ako konzultanti a hlavní stakeholderi Mgr. Kamil Vlček, Ph.D s Mgr. Ivetou Fajnerovou, Ph.D.
SpaNav 1.0 bol posledných 7 rokov aktívne používaný, a dokonca nad ním prebiehal aj vývoj, vďaka ktorému je možné používať ho do istej miery až do dnešnej doby. Doba, v ktorej aplikácia vznikala a možnosti, ktoré autorka projektu mala, sa podpísali na architektúre a dizajne aplikácie. V dnešnej dobe sa už niektoré tieto rozhodnutia ukazujú ako obmedzujúce a bránia naplno využiť potenciál testovania pacientov vo virtuálnej realite. Herné enginy prešli od roku 2009 búrlivým rozvojom, a za hlavný pokrok s ohľadom na možností vývoja možno považovať snahu o otvorenie enginu pre riešenie väčšieho okruhu úloh, než je iba herná zábava. Herné enginy už nie sú iba prostredím pre vývoj hier, a túto vlastnosť sa snaží naplno využiť nová verzia SpaNav 2.0.

Aby sme pochopili motiváciu pre vznik SpaNav 2.0, pozrieme sa podrobne na SpaNav 1.0, pomenujeme jeho slabiny a ukážeme si tak, kde má nová verzia najväčší priestor pre zlepšenie súčasného stavu.

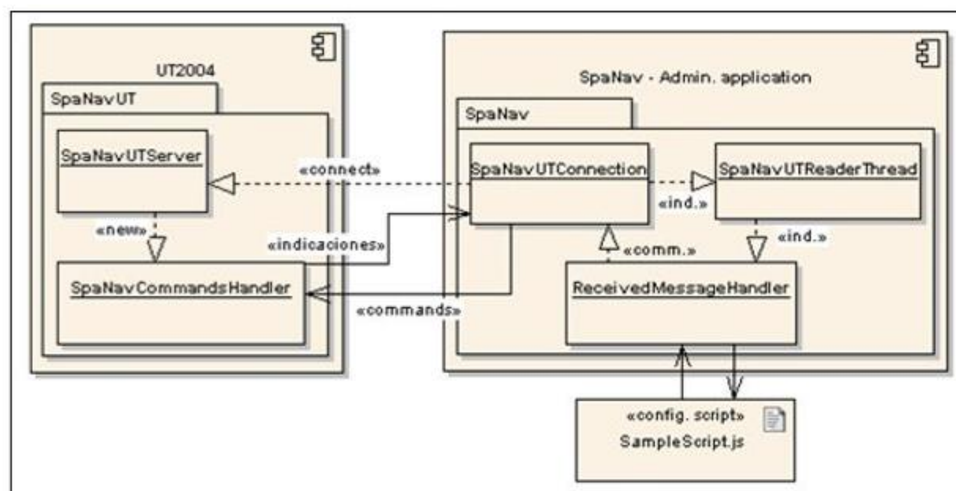
Problém so zaznamenávaním času

SpaNav 1.0 bol navrhnutý ako klient-server aplikácia, a ako uvidíme, túto architektúru rešpektuje aj SpaNav 2.0. Táto architektúra je vynútená problémom, ktorý má aplikácia riešiť, a na tejto úrovni detailu nespôsobuje žiadne problémy. High-level náhľad architektúry, ukazuje obrázok 1, ktorý bol uvedený v pôvodnej práci [8].



Obrázok 2: Architektúra SpaNav 1.0. Systém obsahuje tri samostatné komponenty, ktorých časti spolu komunikujú

Hlavný problém, súčasnej architektúry však vidieť na detailnejšom náhľade, popisujúcom sieťovú komunikáciu medzi komponentmi (Obr. 2).



Obrázok 3: Detailný náhľad na komunikáciu medzi komponentmi SpaNav 1.0

Všimnime si umiestnenie procesu, ktorý vykonáva skript experimentu. Z obrázku je zrejmé, že experiment je vykonávaný v prostredí klienta aplikácie (na obrázku vpravo). Pomocou nástrojov Unreal Enginu pre komunikáciu po sieti, je každý krok experimentu prenášaný po sieti. Správy sa prenášajú ako otvorený text, vo formáte: *TYP_PRÍKAZU {názov_atribútu1, hodnota_atribútu1} {názov_atribútu2, hodnota_atribútu2}* ... . Tieto správy menia chovanie serverovej aplikácie, na ktorú reaguje testovaný subjekt. Informácie o reakciách subjektu sú následne pomocou sieťovej komunikácie, spomínaným textovým protokolom prenášané späť do prostredia klienta.

Aplikácia vytvára záznam o udalostiach zápisom do súboru. Akcia sa zaznamená, ako dvojica [čas, udalosť]. Tieto súbory sú ďalej analyzované a zaznamenané časy udalostí sú kľúčové, pre spracovanie výsledku experimentu. V pôvodnom SpaNav-e tento záznam vzniká v prostredí klienta. Problém tohoto prístupu je možno vidieť na nasledujúcom príklade, v ktorom testovaný subjekt musí zareagovať na zmenu viditeľnosti objektu v hre. V súčasnej verzii SpaNav tento príklad povedie k vykonaniu nasledujúcich krokov:

1. [klient] Kód experimentu označí viditeľný objekt *A*, aby sa odstránil
2. [klient] Udalosť sa zapíše ako odstránenie klienta v čase *t1*
3. [klient] Vytvorí sa správa s príkazom na zmazanie objektu *A* a pošle sa po sieti
4. [server] Server obdrží správu, a dekóduje ju na volanie akcie na objekte *A*
5. [server] Objekt *A* je označený ako vymazaný
6. [server] Pri behu najbližšieho cyklu zobrazovania je odstránený z 3D scény
7. [server] Užívateľ reaguje na zmenu akciou
8. [server] Vytvorí sa správa s informáciou o akcii a pošle sa po sieti
9. [klient] Klient obdrží správu, a dekóduje ju
10. [klient] Informácia o akcii sa zapíše v čase *t2*

Algoritmus 1: Prenos správ v systéme SpaNav 1.0

Kľúčový je časový rozdiel, medzi momentom, keď objekt zmizol užívateľovi z obrazovky (bod 6) a momentom, keď užívateľ zareagoval na túto udalosť(bod 7) . Čas, ktorý sa ale zapíše, je čas v bode 2 a čas v bode 10. Medzitým okrem iného dochádza ku sieťovej komunikácii, ktorá je všeobecne vzaté pomalá, a to aj v prostredí lokálnej siete.

Táto vlastnosť súčasnej verzie SpaNav, je hlavným motivátorom pre nový vývoj. Spomenieme si však ešte niekoľko ďalších.

Unreal engine

SpaNav 1.0 bol navrhnutý pre technológiu Unreal Engine (UE) 2.5 . Táto historická technológia, síce bola napísaná v C/C++, ale ďalší vývoj enginu v základnej verzii nepodporovala. Miesto toho sa spoliehala na možnosti vlastného, objektovo orientovaného jazyka (UnrealScript). Jedná sa o vysoko-úrovňový jazyk, ktorý mal byť veľmi dobre prispôsobený potrebám enginu a vývoju hier v ňom. Tento jazyk slúži pre ovládanie UE viac než pre jeho vývoj a jeho domovský engine je zároveň jeho limitom.

UnrealScript bol dôvodom, prečo bola architektúra SpaNav postavená tak, ako sme si popísali v predošlej kapitole. UnrealScript totiž neumožňuje okrem práce s objektmi enginu v zásade nič navyše. UE2.5 bol napísaný pre vývoj hier, a pri hernej aplikácii sa nepočítalo s implementáciou vlastného skriptovacieho enginu, logovania a tak podobne. Z tohoto dôvodu je proces pre beh skriptu vyvedený do klientskej časti, tak ako aj logovanie, kde sú pomocou univerzálnejších technológií implementované natívnym spôsobom.

Jedno-účelové technológie ako spomínaný UnrealScript, so sebou prinášajú celý rad problémov.

Jednak je pomerne ťažké nájsť ľudí s potrebnými znalosťami, ktorí by boli schopní danú technológiu udržiavať. Vývoj nad SpaNav 1.0 v posledných 7 rokoch nasvedčuje tomu, že projekt nepatrí medzi aplikácie, ktoré sa vyvinú, odovzdajú a ich obsah ostane zakonzervovaný. Aplikácia SpaNav si žije vlastným životom, a neustále sa vyvíja.

Veľký problém UnrealScriptu, je navyše jeho obmedzenosť v doméne funkcií. Jazyk C++ je vďaka knižniciam ďaleko viac rozšíriteľný, než jeho vysoko úrovňový a jednostranne zameraný oponent. To je zásadná požiadavka na technológiu, ktorá má slúžiť vopred neohraničené časové obdobie, a v takej experimentálnej doméne akou je výskum.

Nakoniec to viedlo k faktu, že v poslednej fáze života SpaNav-u1.0 boli známe nároky na aplikáciu, ktoré už kombináciou UnrealScriptu a Java klienta nebolo možné splniť. Okrem iných, spomeňme obmedzenie počtu objektov, ktoré je možno

v experimente sledovať, z dôvodu zahltenia sieťovej komunikácie, pri použití textového protokolu, ktoré sa ukázalo v praxi.

SpaNav 2.0

Systém SpaNav 2.0 pozostáva z troch aplikácií, ktoré si v tejto kapitole stručne predstavíme. Zámer je, aby si čitateľ vytvoril obecné povedomie o aplikácii pred tým, než sa vrhneme do detailného popisu. Nie je nutné na konci tejto kapitoly systému rozumieť, a očakávame že čitateľ bude mať viac otázok ako odpovedí. Ich zodpovedaniu sa bude venovať zvyšok práce.

Tri aplikácie SpaNav 2.0 sú nasledujúce:

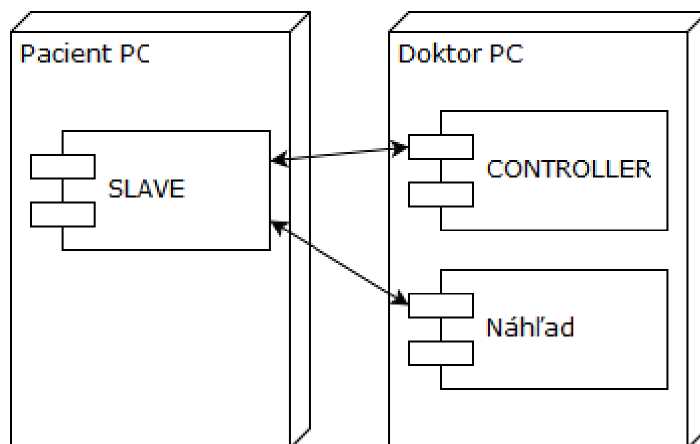
Editor – aplikácia z väčšej časti definuje vizuálnu podobu experimentu

Server – aplikácia realizuje beh experimentu (**SLAVE**)

Klient – aplikácia kontroluje beh experimentu (**CONTROLLER**)

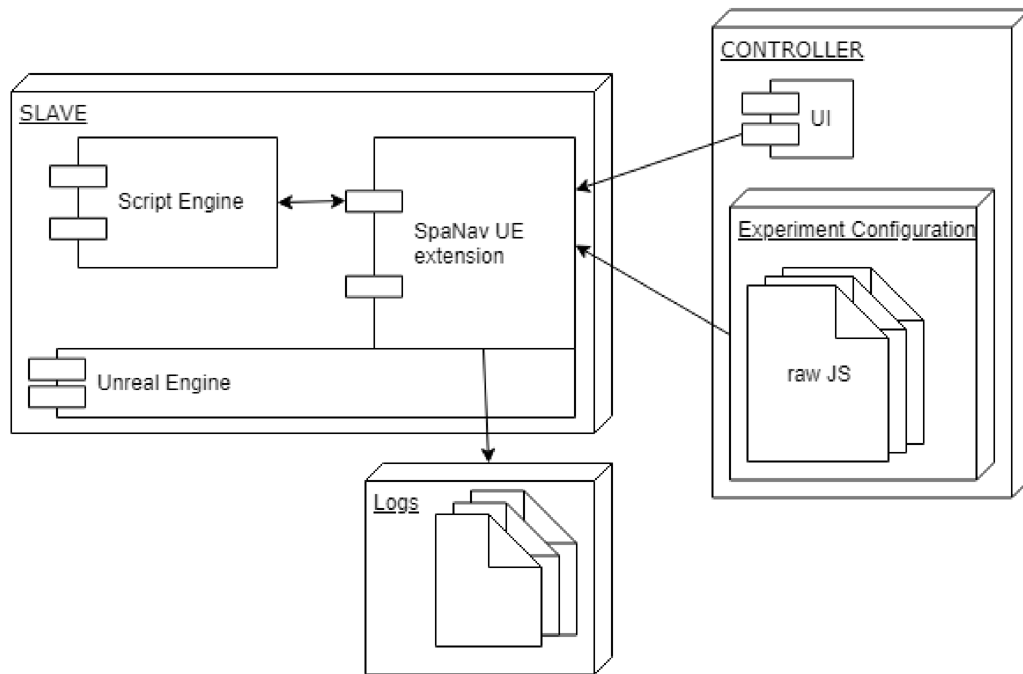
Architektúra

Nový systém sme principiálne meniť nechceli. Architektúra dvoch samostatne bežiacich aplikácií, ktoré spolu dokážu komunikovať po sieti bola identifikovaná ako najvhodnejší model už v pôvodnej verzii. Tento model sa empiricky osvedčil v čase a nemali sme žiaden dôvod s ním hýbať ani v novej implementácii. Výsledkom je teda aplikácia, ktorá na najvyššej úrovni detailu architektúry vyzerá rovnako ako pôvodná aplikácia. Tento efekt bol žiadaný a podarilo sa nám ho aj naplniť.



Obrázok 4: Architektúra SpaNav 2.0. Systém obsahuje tri samostatné komponenty, ktorých časti spolu komunikujú

Riešenie, ktoré nám umožnilo odstrániť problém s pomalou odozvou je možné vidieť až vo vyššej úrovni detailu. Tento detail je zobrazený na obrázku 8



Obrázok 5: Detailný náhľad architektúry SpaNav 2.0

Zásadný rozdiel proti pôvodnému systému, je spôsob akým je interpretovaný kód skriptu. Kým v pôvodnej verzii je program experimentu vykonávaný v CONTROLLER aplikácii a riadi beh UE v SLAVE aplikácii, v novej verzii je program experimentu vykonávaný v samostatnom vlákne priamo v SLAVE aplikácii. K tomuto riešeniu viedla úvaha, rešpektujúca nasledujúce kľúčové charakteristiky systému:

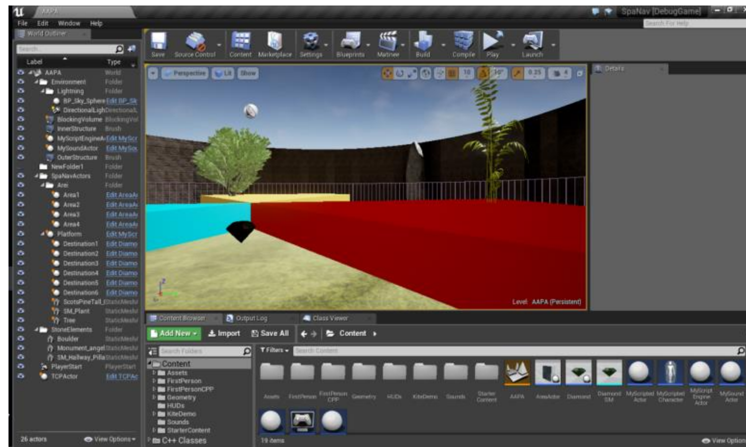
- Voľba experimentu musí byť na strane CONTROLLERu
- CONTROLLER musí mať možnosť zasiahnuť do behu experimentu, vopred definovaným a žiadnym iným spôsobom
- CONTROLLER ovláda konfiguráciu experimentu
- Konfigurácia experimentu sa vykoná pred spustením experimentu
- Nezáleží na tom, kde vznikajú logy experimentu

Prenesením vykonávania skriptu z CONTROLLER aplikácie na SLAVE nenarušíme žiadnu z týchto vlastnosti, rozdiel bude pre obe zúčastnené strany naoko transparentný. Naopak získame relatívne lacno výhodu, že obe komponenty pobežia v jednej aplikačnej doméne so zdieľanou pamäťou.

Pripomeňme, že toto rozhodnutie môžeme spraviť vďaka posunu v oblasti vývoja herných enginov, proti dobe, keď vznikala pôvodná verzia systému. Použitie UE2.5 rozhodlo o umiestnení komponenty pre vykonávanie experimentu do

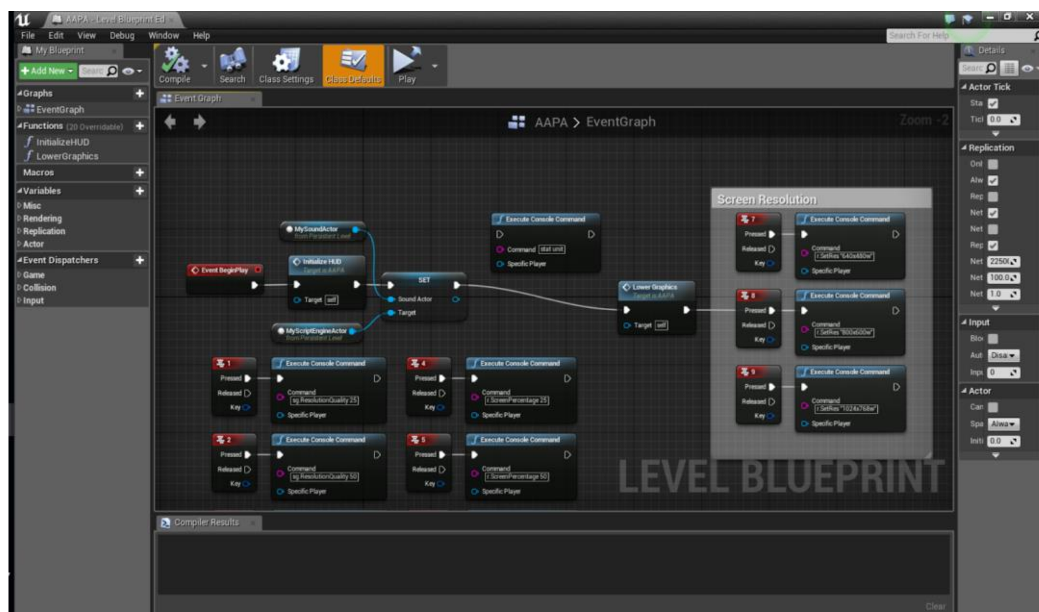
CONTROLLER aplikácie, a povodní autori s tým už nemohli veľa spraviť. Detailom implementácie skriptovacieho enginu do UE4 sa bude venovať samostatná kapitola.

Editor



Obrázok 6: Spustená aplikácia Editora s načítaným experimentom

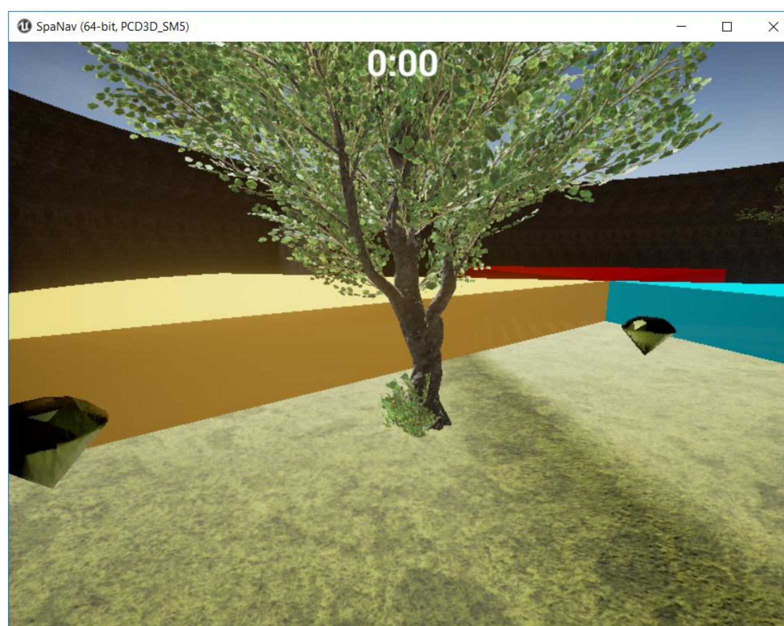
Základnú podobu editora dostaneme od UE4 “zadarmo”. Editor umožňuje vystavať celé prostredie experimentu. Umožňuje umiestniť želané objekty do scény, definovať ako sa budú užívateľovi zobrazovať a množstvo ďalších aspektov. Pomocou vlastného skriptovacieho jazyka (tzv. Blueprints, obrázok 4) umožňuje definovať interakcie medzi vloženými objektmi. Jedná sa o vizuálny skriptovací jazyk z dielne Epic štúdia.



Obrázok 7: Blueprints editor s čiastočnou implementáciou experimentu

Možnosti tohoto jazyka a editora je možné do veľkej miery rozširovať implementáciou v nižšom jazyku – C++, v ktorom je editor napísaný. Použitie Blueprints z veľkej časti nie je povinné. Narazili sme iba na pár prípadov, kde použitie Blueprints malo väčší zmysel, než C++ kód. Toto rozdelenie vyplynulo najmä zo špecifikácie požiadavkov na systém a rozdelenia znalostí jednotlivých členov tímu. Existuje predpoklad, že práca na inom type projektu môže viesť k výrazne odlišnému rozdeleniu účasti kódu na výslednom produkte než v našom prípade.

SLAVE

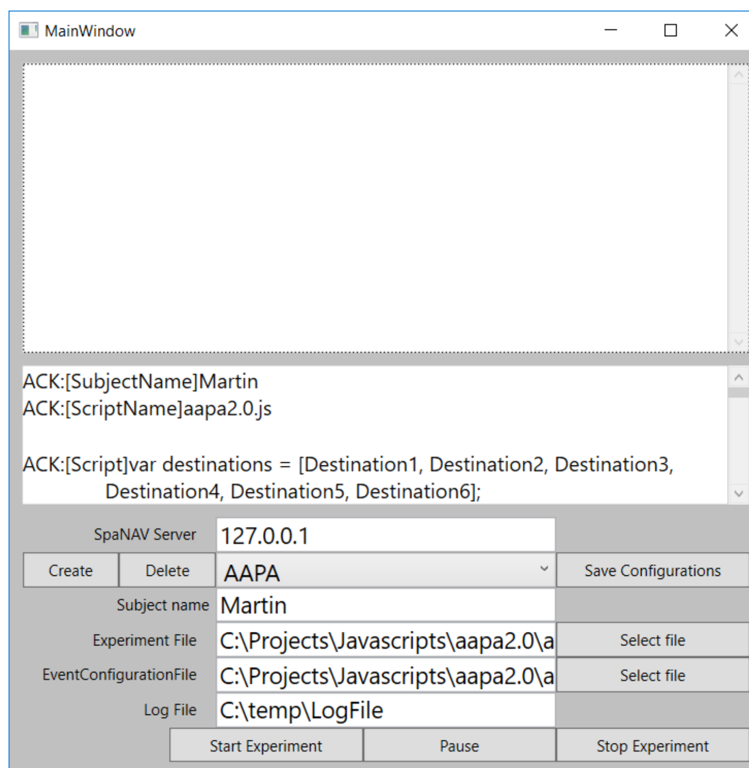


Obrázok 8: Aplikácia SLAVE so spusteným experimentom

"Publishnutím" obsahu editora dostaneme výslednú aplikáciu. Jedná sa o výsledok dizajnového a skriptovacieho procesu z popísaného editora. Aplikácia, ktorú sme vytvorili je samostatne bežiacim programom. Ihneď po spustení aplikácia čaká na pokyny z externého zdroja. Pomocou TCP komunikácie je možné nahrať do SLAVE aplikácie Javascriptový kód, a táto aplikácia je schopná ho začať vykonávať. Touto cestou do SLAVE aplikácie posielame skript navrhnutého experimentu. Rovnakou cestou je možno posielať do aplikácie rôzne príkazy, ktoré majú vplyv na beh experimentu. Tieto príkazy sú buď spracované UE4 enginom, alebo sú posunuté a spracované skriptovacím enginom. Aplikácia počas svojho behu zaznamenáva množstvo údajov do niekoľkých oddelených logovacích súborov. Niektoré logy sú

určené pre logovanie údajov posielaných z externej aplikácie, niektoré logujú z prostredia skriptu a niektoré z prostredia enginu.

CONTROLLER



Obrázok 9: Aplikácia CONTROLLER s pripraveným experimentom

CONTROLLER je jednoduchá aplikácia, ktorá plní niekoľko málo úloh. Prvou úlohou je správa konfigurácií experimentov. Konfigurácia experimentu je daná zoznamom ciest k súborom, v ktorých sú uložené údaje pre beh experimentu vrátane skriptovacích zdrojových súborov, mena konfigurácie a mena testovaného subjektu. Meno testovaného subjektu sa používa pre logovanie a posiela sa do SLAVE aplikácie. Tým sa dostávame k druhej úlohe CONTROLLER aplikácie. Tou je komunikácia so SLAVE aplikáciou. Tá spočíva v poslaní konfigurácie experimentu v zrozumiteľnej forme do aplikácie SLAVE, pred začiatkom experimentu. Počas experimentu ďalej CONTROLLER zabezpečuje komunikáciu kontrolných príkazov do SLAVE aplikácie. Počas experimentu CONTROLLER aplikácia na obrazovku vypisuje komunikáciu smerom od SLAVE aplikácie do CONTROLLER aplikácie.

Použité technológie

Unreal Engine 4

Pre základ novej verzie SpaNav sme použili Unreal Engine verzie 4. Medzi treťou a štvrtou verziou enginu, došlo k významnej zmene prístupu vo vývoji herného enginu od štúdia EPIC. Posledná verzia je písaná v C++ a je opensource. To malo dopad na priebeh vývoja aj výslednú podobu aplikácie. Narozdiel od staršej, veľmi reštriktívnej verzie, nová verzia sa snaží byť čo najviac otvoreným enginom. Programátorský model nevymedzuje, ale skôr sa snaží programátora viesť odporučeniami, komentármi v kóde, pomerne rozsiahlou bázou návodov, a demonštratívnymi implementáciami. Tie prichádzajú formou počiatočných šablón a množstvom ukážkových aplikácií so zdrojovými kódmi, ku ktorým je možné sa dostať priamo zo spúšťacej aplikácie UE4.

Celý UE4 engine je dostupný k stiahnutiu do offline adresára na lokálnom počítači, alebo je možné naklonovať ho z verejného git repozitára. Jedná sa o niekoľko gigabajtov C++ zdrojových súborov, assetov a konfiguračných súborov potrebných pre zostavenie aplikácie. Je dokonca možné automaticky vytvoriť súbory, ktoré z repozitára spravia Visual Studio kompatibilný projekt. Takýto projekt je potom možné pomocou *build* mechanizmov Visual Studia aj spúšťať. Samotný projekt je ale nezávislý na použitom IDE. Tak isto je v základe nezávislý na operačnom systéme, a až finálne zostavenie projektu do výslednej aplikácie určí potrebný operačný systém. Toto zostavenie je taktiež automatickým procesom, ktorému predchádza výber cieľovej platformy.

UE4 používa dva programovacie modely. Prvý, nízkoúrovňový, je C++ programovací model. Na začiatku je zdrojový kód enginu a po preložení dostaneme spustiteľný editor, v ktorom môžeme modelovať výsledný program. Druhým sú takzvané Blueprints, čo je vizuálny skriptovací jazyk vyvinutý pre UE4 a je k dispozícii v prostredí spusteného editora. Možnosti tohoto jazyka je do značnej miery možné ovplyvniť implementáciou v C++. Vývoj systému je tak rozdelený do dvoch fáz. Prvou fázou je príprava enginu a potrebnej funkcionality v C++. V tejto časti sme sa snažili odviesť väčšinu takzvanej “špinavej” práce. Blueprinty sme využili pre naskriptovanie funkcií, ktoré považujeme na dobrých adeptov pre budúce drobné zmeny, alebo je pre nich implementácia v tomto vyššom doménovom jazyku výrazne prirodzenejšia pre riešenie konkrétneho problému. Blueprinty sú na pozadí

implementované v C++ a pri zostavení do výslednej aplikácie sa preložia do strojového jazyka. Rýchlosť vykonávania kódu preto nebola kritériom pre umiestnenie jednotlivých častí implementácie.

Javascript a Google V8

Súčasťou špecifikácie systému, je možnosť skriptovať experimenty. Užívatelia sú oboznámení s jazykom Javascript, a preto rovnako ako starší systém aj nový musel podporovať možnosť skriptovania experimentu v tomto jazyku. Z navrhnutej architektúry nám vyplynulo zadanie, implementovať Javascriptový engine do vnútra UE4. Testovalo sa niekoľko možností. Najme sme sa zaoberali troma systémami a to:

1. Spidermonkey
2. Google V8
3. Flathead

Spidermonkey je Javascriptový engine používaný v produktoch Mozilly. Najmä teda prehliadača Firefox. Flathead je najjednoduchší z troch spomenutých systémov. Jedná sa o nezávislý projekt pod otvorenou licenciou, sľubuje najmenej práce s implementáciou do iného systému. Pôvodne sme teda začali s touto knižnicou, a pokúsil sa tak absolvovať cestu najmenšieho odporu.

Následnou analýzou špecifikácie potrieb prostredia, v ktorom sa bude vyvíjať experiment, sme vyvodili niekoľko bodov, o ktoré bude musieť byť štandard Javascriptu obohatený. Začal som nad *PoC* implementáciou skriptovacieho enginu skúmať možnosti ako pripraviť sadu knižníc použiteľných pre potreby experimentu. Tieto knižnice by boli vystavené do skriptu a na pozadí by volali C++ funkcie. Tento model by nám umožnil takmer neobmedzené možnosti skriptovacieho enginu. Čoskoro sme však narazili na limity vybraného Flathead enginu. Tento engine nepôsobil presvedčivo v niektorých prípadoch a nútil nás k obchádzaniu problémov, a vymýšľania rôznych work-aroundov. Vývoj v tomto systéme sa postupne stával náročnejším a začínalo byť zrejmé, že skôr či neskôr narazíme na jeho limity. V tejto fáze vývoja však bolo ešte relatívne bezpečné a málo prácne engine vymeniť. Začali sme preto skúmať iné možnosti.

Nakoniec sme vyberali medzi veľkými hráčmi a pre Google V8 engine sme sa nakoniec rozhodli na základe porovnania výkonnosti oboch enginov. Projekt Google

sľuboval navyše programátorský model, ktorý na základe skúseností potrieb z predošlých pokusov, umožňoval prirodzenejší prístup k implementácii.
Aj tento engine však so sebou priniesol niekoľko náročných výziev. Prvá sa objavila hneď na začiatku. Zakomponovanie enginu do existujúceho prostredia sa ukázalo byť veľmi časovo náročnou úlohou. Chýbajúca dokumentácia a veľmi nepriateľská, ak nejaká, komunikácia s tímom zodpovedným za implementáciu enginu, tento problém iba viac prehlbovala. Problém sme po výraznej časovej strate vyriešili z časti analýzou kódu inej opensource knižnice, ktorá používala staršiu verziu Google V8 a ich riešenie sme za pomoci analýzy kódu Google V8 knižnice prepísali do poslednej verzie Google V8. Táto časová investícia sa nám od tej doby odpláca bezchybným fungovaním najrýchlejšieho dostupného Javascript enginu vo vnútri našej UE4 aplikácie, so všetkým programátorským pohodlím, pekne navrhnutého API. Skriptovací engine sme sa pri implementácii snažili čo najviac od zvyšku aplikácie izolovať, pre prípad, že by sme opäť narazili na jeho limity.

C# a Visual Studio IDE

Aplikácia CONTROLLERu je na funkcionalitu nenáročný program, ktorý musí spĺňať hlavne požiadavky ako:

- Prácu so súbormi
- Základnú komunikáciu po lokálnej sieti
- Možnosť definovať príkazy pre aplikáciu SLAVE

Ďalej bolo vopred známe, že aplikácia CONTROLLERu bude používaná výhradne na staniciach s operačným systémom Windows. Tieto požiadavky umožňujú vybrať široké množstvo jazykov a technológií, no autor sa rozhodol pre jazyk C# a technológiu XAML pre podporu UI, s ktorými má najviac skúseností.

Configuration management

Z dôvodu zefektívnenia práce som aplikáciu vyvíjal na viacerých počítačoch. Na pracovnej stanici doma, ktorá slúžila ako úložisko pre hlavný repozitár a mobilne na notebooku. Z toho dôvodu som potreboval systém, ktorý by umožňoval správu konfigurácií aj v prípadoch, keď by som bol dlhší čas offline od hlavného repozitára. Táto požiadavka vedie k použitiu distribuovaného verzovacieho systému, akým je napríklad dobre známy GIT. Distribuovaný systém umožňuje commit akciu na

lokálnom repozitári, čím dáva možnosť udržiavať verzie aj v offline móde. Následne akcia publish premietne uložené zmeny do hlavného repozitára.

Verzovanie vývoja aplikácie sme riešili pomocou Mercurial HG systému od samého začiatku. Tento systém som použil, pretože som ho dôverne poznal z predošlých projektov, a umožňuje distribuovanú správu konfigurácií, ktorú som popísal v predošlom odstavci.

Celý systém sa nachádza v troch samostatných repozitároch:

1. SLAVE aplikácia, Unreal Engine 4 projekt (C++, súbory UE4)
2. CONTROLLER aplikácia (C#, Visual Studio súbory)
3. Skript experimentu (Javascript súbory)

Projekt je rozdelený do samostatných repozitárov, pretože toto rozdelenie lepšie zodpovedá konfiguračným jednotkám, podľa toho ako na nich prebieha vývoj. Navzájom sa ovplyvňujú iba málo, a prevažne k tomu dochádzalo v rannejších štádiách vývoja, keď sa formovala doména možností aplikácie.

Hlavné repozitáre v podobe archívov ZIP odlievam v niekoľkotýždňových intervaloch na samostatný disk, ktorý slúži ako záloha, a je aj po väčšinu času geograficky oddelený od ostatných repozitárov.

Projektový management

Pre projektový management sme použili Trello a e-mailovú komunikáciu.

K detailnejšiemu použitiu jednotlivých technológií sa dostaneme v kapitole patriacej softvérovému inžinierstvu.

Programátorská dokumentácia

Aplikácia pozostáva z niekoľkých častí. Programátorsky sa vyvíjalo na dvoch častiach systému. V tejto časti sa pozrieme detailne na obe časti. Prvou je CONTROLLER aplikácia napísaná v .Net C# pomocou technológie WPF. Je veľmi jednoduchá a budeme jej venovať len minimum priestoru. Hlavnou časťou bude popis programovacieho modelu SLAVE aplikácie. Na jej vývoj boli použité Javascript, C++ a UE4 Blueprints.

CONTROLLER

Aby sme pochopili úlohu CONTROLLER aplikácie v našom systéme, je potreba mať základné povedomie o experimentoch, ktoré spravuje.

Funkcie CONTROLLERu

Konkrétne má CONTROLLER v správe základné aspekty experimentu, akési metadáta, ktoré pomáhajú najme pri kategorizácii behu experimentu.
Pre jednotlivé behy experimentu sú dôležité nasledujúce údaje:

- Identifikácia subjektu testovania
- Cesta k zdrojovým súborom experimentu (definícia experimentu)
- Cesta k súboru mapovania kláves na udalosti posielané do experimentu
- Cesta pre ukladanie logovacích súborov

Definíciu týchto premenných nazývame konfiguráciou behu experimentu. Každú konfiguráciu je možno uložiť a priradiť jej identifikátor, ktorým je možno sa k tejto konfigurácii kedykoľvek vrátiť. Konfigurácie je možno vytvárať, upravovať a mazať. Experiment nezačne bežať automaticky po spustení aplikácie, aj napriek tomu že na začiatku sa automaticky vyberie existujúca konfigurácia, aplikácia čaká na zmenu výberu alebo spustenie vybraného experimentu. Naznačili sme, že aplikácia umožňuje ovládanie behu experimentu. Jedná sa o možnosti experiment spustiť, pozastaviť, a ukončiť. Čo presne sa stane keď niektorý z pokynov zvolíme sa dozvieme v časti venovanej SLAVE aplikácii. Momentálne potrebujeme vedieť iba toľko, že voľba akcie vyšle signál do SLAVE aplikácie, ktorá podnet prijme a spracuje ho.

Okrem základných-preddefinovaných akcií, užívatelia môžu definovať vlastné akcie a priradiť im voľnú klávesu. Užívateľ má možnosť definovať kód signálu, ktorý sa pošle z CONTROLLERu do skriptovacieho enginu, kde je možné kód odchytiť a priradiť mu vykonávanie požadovanej logiky.
Z uvedeného vyplýva, že aplikácia musí komunikovať s inou aplikáciou na sieti. Pre komunikáciu používame TCP/IP protokol na lokálnej sieti. Aby bolo možné prepojiť aplikácie v produkčnom prostredí bez nutnosti meniť kód, aplikácia dáva možnosť zvoliť IP adresu stroja, na ktorom očakáva bežiacu SLAVE aplikáciu.
Ďalej si je potreba uvedomiť, že napriek tomu, že zdrojový kód je umiestnený fyzicky na strane CONTROLLER aplikácie, po spustenú experimentu beží v prostredí UE. Aby bolo možné ladiť ako komunikáciu medzi aplikáciami, tak aj chyby vyplývajúce z behu experimentu, CONTROLLER aplikácia potrebuje zbierať a užívateľovi prezentovať informácie, ktoré by mu mohli pomôcť riešiť prípadné problémy. Najmä sa jedná o chyby v komunikácii, typicky ak CONTROLLER aplikácie nemôže dosiahnuť na SLAVE aplikáciu cez LAN, a typicky chyby, ktoré nastanú pri kompilácii alebo behu skriptovaného kódu.

Implementácia CONTROLLERu

Pre tvorbu vizuálnej podoby UI sme použili technológiu WPF. Aplikácia je jednoduchá a obsahuje iba jednu obrazovku, ktorej podoba je definovaná pomocou jedného .xaml súboru. Xaml obrazovku obsluhuje pridružená trieda napísaná v C#. Táto trieda priamo používa služby ďalších tried a to triedy na komunikáciu cez TCP/IP protokol, triedy pre JS preprocessing a triedy, ktorá má za úlohu prekladať vstup z klávesnice na inštrukcie posielané to SLAVE aplikácie podľa definície z konfiguračného súboru.
Po spustení aplikácie dôjde automaticky k načítaniu konfiguračného XML súboru. Tento súbor je rozparsovaný a namapovaný do C# štruktúry, ktorá následne slúži ako dátový zdroj konfiguračných okien CONTROLLERu. Komunikácia medzi ovládacími prvkami a dátovým zdrojom je na požiadanie obojsmerná, zavolaním akcie *Save Configuration*, sa zmeny trvale uložia do XML súboru.
Trieda pre TCP/IP komunikáciu neobsahuje žiadnu prevratnú logiku. Trieda umožňuje iniciovať spojenie so vzdialeným end-pointom definovaným IP adresou a číslom portu. V prípade úspechu pustí aplikácia nové vlákno, ktoré má za úlohu počúvať prichádzajúcu komunikáciu na end-pointe. Komunikácia má textový

charakter, a preto aplikácia obsahuje dve funkcie, jednu pre čítanie a jednu pre zápis z/do TCP/IP.

Funkcia, ktorá zo siete informácie číta, posiela tieto informácie do debugovacieho okna. Typicky sa jedná o odpovede SLAVE aplikácie na požiadavky posielané z CONTROLLERu. SLAVE aplikácia iným spôsobom beh CONTROLLER aplikácie neovplyvňuje.

Kódy vopred definovaných akcií (štart, pozastavenie, ukončenie experimentu) sú vopred definované a známe CONTROLLER aj SLAVE aplikácii. Sú definované natvrdo, neočakáva sa ich zmena. Po započatí experimentu v CONTROLLER aplikácii, CONTROLLER pošle niekoľko inicializačných správ do SLAVE aplikácie. Tieto inicializačné správy sú vždy uvedené ďalšou sadou kódov. Tieto kódy, spolu s úvodnými tromi kódmi patria medzi takzvané systémové kódy, a definovali sme konvenciu hranatých zátvoriek pre ich odlíšenie od zbytku kódov. Počas tejto úvodnej komunikácie CONTROLLER odošle na stranu SLAVEa meno testovaného subjektu, meno koreňového skriptu obsahujúceho definíciu experimentu, preprocesorom spracovaný skript experimentu, a cestu pre logovacie súbory. Každá správa je označená príslušným kódom, ktorý SLAVE použije pre identifikovanie obsahu.

Okrem systémových správ, ktoré spracúva aplikácia UE, umožňujeme definovať vlastné správy, ktoré sa posielajú do skriptovacieho enginu. Tieto správy môžu spúšťať skriptovanú logiku a upravovať tak chovanie experimentu. Kódy sa do siete posielajú ako reakcia na stlačenie konkrétnej klávesnice. Ktorá klávesa posiela ktorý kód, je definované dvojicou klávesa:kód, napríklad "V;ToggleDestinations". Uvedená konfigurácia po stlačení klávesy V, pošle po sieti kód "ToggleDestinations". Mapovanie sa nachádza v samostatnom súbore, cesta ku ktorému, je súčasťou konfigurácie experimentu. Komponent *TCPEventConverter* premapuje súbor dodaný na vstupe do .NET dictionary objektu, ktorý ako kľúč používa objekt reprezentujúci uvedenú klávesu, a vedie k hodnote kódu, ktorý sa bude po sieti odosielať. Tento mapovací objekt potom v reálnom čase prekladá udalosti na CONTROLLERi do signálov v aplikácii SLAVE.

Javasciptový engine, použitý pre tento projekt, prijíma JS zdrojový kód ako jeden reťazec. Neumožňuje tak out-of-the-box prácu s viacerými JS súbormi. Táto vlastnosť však pre jednoduchý vývoj experimentov bola vyžadovaná a museli sme ju do systému doimplementovať. K tomuto účelu slúži Javascriptový preprocesor.

Tento komponent na vstupe dostane cestu ku koreňovému JS zdrojovému súboru. Cesta k súboru je súčasťou konfigurácie experimentu. Preprocesor súbor otvorí a pre každý import, vyhľadá zdrojový súbor s uvedeným menom. Preprocesor podporuje aj relatívne a absolútne cesty. V pamäti si preprocesor vytvára štruktúru, ktorá postupne nahrádza všetky import ich obsahom, v poradí, v akom sú importy uvedené. Preprocesor podporuje aj kaskádové importy. Javascript nemá definované ako má importovací príkaz vyzerať preto sme sa dohodli na vlastnom importovacom príkaze vo formáte #Include(meno súboru). Umiestnenie preprocesoru na stranu CONTROLLERu vyplýva z faktu, že lokálne sa vyskytujú všetky zdrojové súbory, takže lokálny kontext je najprirodzenejší na riešenie tohoto problému. Preprocesor svoj výsledok odovzdá komponente pre posielanie komunikácie do siete, tá ho vybaví potrebným tagom pre identifikáciu obsahu a pošle ho ako jeden hotový skript na stranu SLAVE.

SLAVE

Aplikácia bola od začiatku vyvíjaná ako centrálny bod systému SpaNav. Je nositeľom všetkej logiky, vyjma konkrétnej implementácie chovania experimentu. Okrem výsledného chovania ale definuje všetky objekty v experimente. Definuje Vizuálnu podobu experimentu, a vystavuje do Javscriptu chovanie objektov experimentu. Javascriptový kód následne pomocou vystavených volaní dodá experimentu výslednú podobu. Ako je aplikácia SLAVE implementovaná popíšeme v nasledujúcich kapitolách. Každá kapitola sa bude venovať jednej konkrétnej funkcionalite, vysvetlíme si vždy najprv aká je úloha popísanej domény, a ako sme dosiahli chovanie, ktoré túto úlohu plní. Časť logiky aplikácia implementuje pomocou C++ kódu, a časť implementuje pomocou technológie Blueprints.

ScriptEngineActor

Centrálnym bodom aplikácie je trieda *ScriptEngineActor*. Táto trieda je ako aj väčšina komponent priamym potomkom triedy *AACtor*.

AActor je objekt dodaný Unreal Enginom a poskytuje základnú funkcionalitu pre všetky objekty umiestnené v hre. Dôvod pre použitie základnej triedy *AActor* je, že týmto spôsobom získame dobrý programovací model, ktorý nám v prostredí UE4 značne uľahčí vývoj. Okrem iného, poskytne jednoduchú cestu pre komunikáciu

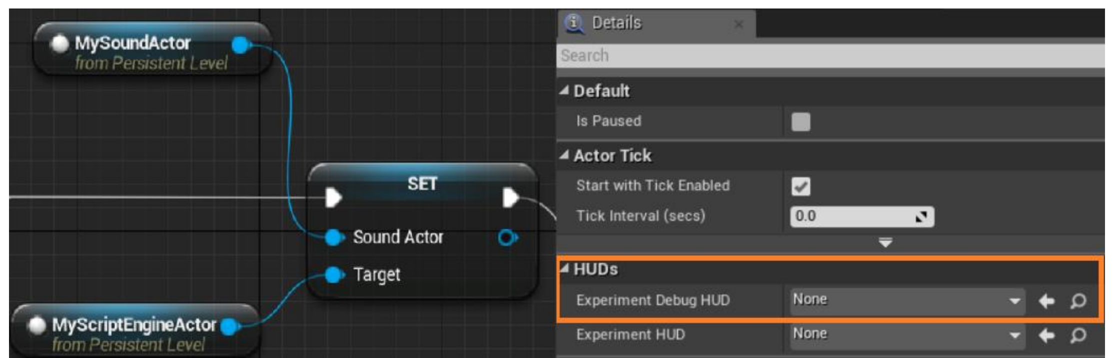
medzi jednotlivými objektami a umožní nám pohodlné rozšírenie možností objektov pre skriptovanie pomocou Blueprints. Táto úroveň je navyše pomerne dobre zdokumentovaná, čo nám dáva vysokú mieru istoty, že pre riešenie problému nevojdeme do slepej ulice.

Ako sme popísali, *ScriptEngineActor* slúži ako HUB, ktorý na začiatku prepojí hlavné komponenty aplikácie, aby neskôr spolu mohli komunikovať. K tomuto účelu definujeme zásuvky na rozhraní objektu, kam skriptovaním pripojíme po spustení mapy, hotové objekty umiestnené na mape. Pre definíciu zásuvky sa používa makro UPROPERTY.

Nasledujúci kód definuje zásuvku typu *ExperimentDebugHUD* (HUD – head up display), ktorú je možné nastavovať z Blueprints a objaví sa na objekte *ScriptEngineActor* v sekcii “*HUDs*”.

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "HUDs")
        UExperimentDebugHUD* ExperimentDebugHUD;
```

Pro spustení aplikácie sa použitie tohoto makra prejaví ako ukazuje nasledujúci obrázok



Obrázok 10: Nová vlastnosť a nová skupina vlastností v Blueprints

Takto sme schopní jednoducho prepojiť komponenty umiestnené v mape experimentu a komunikovať s nimi od C++ kódu cez Blueprints, skripty až po Javascript kód.

ScriptEngineActor ihneď po spustení (až boli skonštruované všetky objekty v mape) zaregistruje načasovanú úlohu, ktorá v krátkych intervaloch opakovane zavolá TCP/IP komponentu a vyžiada si od nej, nové správy prijaté zo siete. K tomuto účelu sa používa *FTimerManager*, ktorý ovláda všetky naplánované úlohy v aplikácii.

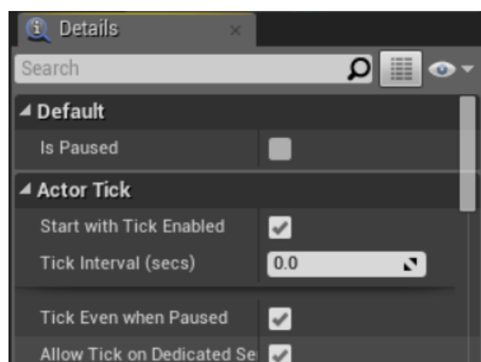
FTimerManager má tú výhodu, že je natívne podporovaný UE4 a jeho použitie nespôsobuje problémy s interným časovaním UE4 behu, tak ako je tomu ak použijeme vlákna. Hodí sa najmä na krátkodobé výpočetne nenáročné úlohy, mimo

štandardné tikanie aplikácie medzi jednotlivými frame-mami. Pre plnohodnotný paralelný beh sú vlákna nevyhnutné pretože časovač je pre beh aplikácie blokujúci. Vlákna preto používame pre beh skriptovacieho enginu a budeme sa im venovať v neskorších kapitolách.

My v čase prerušenia časovačom skontrolujeme sieťový endpoint a nad prípadnou prijatou správou vykonáme v tejto úrovni triediacu logiku. Podľa pravidiel konvencie obsahu správ určíme, či sa jedná o správu systémovú, alebo sa jedná o ostatný typ správy. Systémové správy prichádzajú prevažne pred začiatkom experimentu a okamžite sú spracovávané. Pre tento typ správ je príznačné, že sú vopred dobre definované a očakávané. Tieto správy typicky obsahujú informácie potrebné pre beh experimentu a pomocou nich inicializujeme objekt experimentu, ktorým sa budeme viac zaoberať čoskoro.

Špeciálnou skupinou sú správy pre ovládanie začiatku behu experimentu, pozastavenie a koniec experimentu. V tejto časti sa budeme venovať iba správam pre začiatok experimentu a koniec experimentu. Pozastavenie experimentu sa rieši na viacerých úrovniach aplikácie.

Samotný UE4 je ako herný engine na pozastavenie vykonávania prispôsobený. Na tejto úrovni stačí aktivovať *SetPause(boolean)* funkciu, ktorá vykonávanie enginu pozastaví. Problémom však je, že táto akcia štandardne pozastaví vykonávanie nad všetkými UE4 objektami, teda aj čítanie zo siete a zápis do nej, ktorý je iniciovaný práve z tejto vrstvy. Aby sme toto chovanie zmenili, potrebujeme zmeniť nastavenie objektu. Jeden zo spôsobov ako to dosiahnuť je v editore v paneli detailov nastaviť hodnotu *Tick Even when Paused* na hodnotu *True*.



Obrázok 11: Nastavenie vykonávania cyklu počas pauzy enginu

Ďalšie kroky pre pozastavenie sa dejú na nasledujúcich vrstvách aplikácie. V tomto momente komponent len zaistí čítanie zo siete a prevolá ďalšiu vrstvu.

Správa pre začiatok behu experimentu, nazhromaždí všetky objekty umiestnené v mape a ich referencie použije ako vstupné parametre pre spustenie začiatku experimentu. Získavajú sa dva typy objektov. V prvom rade sú to všetky objekty v mape, ktoré dedia od objektu *AScriptedActor*, čo je naša verzia základnej triedy pre objekty, s ktorými sa jednotne manipuluje z prostredia Javascriptu. Jedná sa objekty oblastí, ktorým sa treba vyhýbať a objekty oblastí, ktoré je treba nasledovať. Týmto objektom bude tak isto venovaná vlastná kapitola. Druhou skupinou sú objekty, ktoré priamo nesúvisia s vyhodnocovaním experimentu ale prispievajú k jeho realizácii. Sú to objekty typu HUD, objekty pre ovládanie zvukových efektov a objekt hráča v mape. Všetky tieto objekty vystavujú do Javascriptu svoje funkcie a musia sa preto podieľať na inicializácii enginu.

Trieda okrem objektov umiestnených na mape vlastní aj „plain“ C++ objekt *Experiment*, ktorý reprezentuje experiment po dobu jeho behu. Objekt *Experiment* okrem reprezentácie behu experimentu funguje ako obal pre celý Javascriptový modul. Beh Javascriptového prostredia je priamo viazaný na kontext experimentu a jeho beh má zmysel iba ak je experiment aktívny.

Keď sme v predošlých odstavcoch hovorili o spustení experimentu, mysleli sme zavolanie funkcie práve na tomto objekte.

Pre koniec experimentu platí podobné pravidlo, a to také, že po prijatí správy zo siete o konci experimentu, zavolá sa opäť funkcia objektu *Experiment*.

Správy prijaté zo sieťovej komunikácie, ak nie sú označené ako systémové, sú posielané do rozhrania objektu *Experiment* pre príjem príkazov, kadiaľ sú ďalej dopravené do Javascriptového enginu.

Experiment

Trieda experimentu obsahuje tri hlavné funkcie pre ovládanie experimentu, štart experimentu, pozastavenie a ukončenie experimentu. Tieto funkcie sú prevolané z nadradenej komponenty ako reakcia na pokyn zo sieťovej komunikácie.

Po zavolaní začiatku behu, komponent otvorí logovací súbor a zaloguje predom definovanú hlavičku. V druhom kroku sa vytvorí nové vlákno, v ktorom pobeží Javascriptový engine. V tomto prípade sa už skutočne jedná o plnohodnotné neblokujúce vlákno. Existencia spojenia s logovacím súborom a existencia vlákna pre beh skriptu je spojená s trvaním experimentu ako je definovaný pre triedu Experiment a preto sa zapisovanie do súboru aj vlákno zastavia na pokyn zastavenia

nad objektom Experiment. Počká sa na úspešný štart vlákna so skript enginom, a v prípade úspechu, sa odošle správa o úspechu do CONTROLLER aplikácie.

Pozastavenie aplikácie musí prebiehať na viacerých vrstvách. Dôvodom je použité vlákno. V našej architektúre potrebujeme aplikáciu pozastaviť aj v prostredí skriptovacieho enginu. Po zavolaní funkcie pozastavenia experimentu, pozastavíme UE4 engine a vlákno so skriptovacím enginom suspendujeme. Tým efektívne vlákno zastavíme a čas pre nami pozastavené vlákno prestane existovať.

Pre účely časovania nám však vo vlákne beží časovač, ktorý sa stará o všetky načasované funkcie v skripte a sleduje strojový čas na počítači. Tento časovač počíta časové intervaly, aby dokázal presne určiť, kedy má naplánovať beh funkcie podľa plánovacej tabuľky. Keďže čas plynie aj v čase pozastavenia aplikácie, je nutné sledovať dobu, počas ktorej je aplikácia pozastavená. Po opätovnom spustení vlákna, je potreba počítať s dobou, počas ktorej bolo vlákno pozastavené a túto dobu zohľadniť, keď počítame čas od posledného spustenia načasovanej úlohy.

Trieda Experiment pre komunikáciu s vláknom používa prenosový objekt (zdieľanú pamäť), do ktorého sa dívajú oboje vlákna, ako hlavné vlákno UE4 enginu tak aj naše nové vlákno venované skriptovaniu. Pre nekonfliktný prístup sa používa uzamykanie objektu.

TCP/IP modul

TCP/IP modul je riešený pomocou triedy *TCPActor*. Tento objekt dedí od predka *AActor*. Po spustení mapy experimentu, *TCPActor* zaregistruje dva UE-dodávané TCP listenery. Jeden listener počúva na *localhost* endpointe, a jeden počúva na IP adrese priradenej sieťovou kartou. Táto adresa sa zistí dynamicky pomocou UE4 rozhrania *ISocketSubsystem*. Localhost endpoint používame z dôvodu vývoja a testovania aplikácie na jednom počítači. Listenerom nastavíme call-back funkciu, ktorá sa pustí ak na endpointe požiada iná aplikácia o spojenie. Spojenie sa automaticky prijme. Tento komponent vystavuje dve verejné metódy. Je to metóda na zápis do spojenia a metóda na čítanie zo spojenia. Obe pracujú s textovými reťazcami. Dôvodom pre textový formát je, možnosť definície vlastných príkazov v CONTROLLER aplikácii a ich spracovanie v skript engine. Metóda pre čítanie zo spojenia si najprv zistí veľkosť správy a potom ju naraz načíta do jedného reťazca. Po spojení pošle potvrdzujúcu správu o prijatí, ktorá sa CONTROLLER aplikácii zobrazí v ladiacom výstupe. Pri čítaní zo siete zároveň komponent prekladá binárny

reťazec zo vstupu na reťazec v textovej podobe. Konverzia beží v UE aplikácii a na výkon systému nemá podstatný vplyv. Správy sú až na úvodný zdrojový kód skriptu krátke niekoľko málo znakov.

Funkcia pre zápis do spojenia je o poznanie jednoduchšia. Vstupný reťazec je akurát pretypovaný pre potreby UE4 a odoslaný do aplikácie CONTROLLER.

Skript Engine

Funkcionalita Javascriptového enginu sa sústredí okolo C++ triedy *JavascriptEngine*. Ten obsahuje jedinú nestatickú verejnú metódu *Run*, ktorá sa pustí pri vzniku skriptového vlákna, ako sme to popísali v predošlom texte. Táto metóda inicializuje Google V8 Engine. Engine sám obsahuje množstvo statických metód a vlastností, a tým si výrazne vynucuje spôsob práce s ním. Je navrhnutý tak, že prácu s ním je možno izolovať do kompaktného API. Treba však mať na pamäti, jeho výrazný "statický charakter". Komponent preto obsahuje triedu, ktorá má za úlohu inicializáciu enginu a následne aj jeho deštrukciu. Engine je potreba správne vyčistiť, aby bolo možné jeho prípadné opätovné použitie bez nutnosti opúšťať celú aplikáciu.

Po inicializácii sa spustí registrácia C++ vystavenej funkcionality do enginu. Týmto spôsobom, môžeme sprístupniť ľubovoľný C++ kód do skriptovania. Do Javascriptu môžeme sprístupniť ako voľné metódy, tak aj objekty. Vystavený objekt bude možné v skripte obsluhovať notáciou *Objekt.metóda*, tak ako sme zvyknutí z iných jazykov.

V prvej fáze registrujeme, samostatne volané obecné metódy ako napríklad *GetTime, RegisterEvent* alebo *Log*. Tieto metódy sú na strane C++ súčasťou triedy *ScriptGlobals*. Špeciálnou skupinou sú do prostredia skriptu vystavené globálne prístupné metódy matematických funkcií. Jedná sa napríklad o funkciu pre náhodné čísla, funkciu počítajúcu odmocninu a tak podobne.

V druhej fáze registrujeme inštancie objektov, existujúcich v C++, ktoré chceme využívať v skriptovacom engine. Do registrácie neposúvame celú triedu ale iba funkcionalitu vyžadovanú skriptovaním, podľa špecifikácie. Registrovanými triedami sú napríklad objekty typu Actor(zakázané oblasti, cieľové oblasti), objekt reprezentujúci hráča, objekty reprezentujúce rôzne druhy HUD. Z nich vyberáme typicky pomerne primitívne metódy, ktoré sú na strane skriptu volané a tým vytvárajú hotový experiment. Na konkrétny príklad implementácie sa pozrieme v kapitole venovanej skriptovaniu. Takýmito metódami sú napríklad metódy

nastavujúce polohu Actor objektu, spúšťanie zvuku vo zvukovej komponente alebo nastavenie textu pre HUD.

Po registrácii objektov sa pokúsime skompilovať kód dodaný z CONTROLLER aplikácie. Do vlákna sme zdrojový kód priniesli ako jeden z parametrov hlavnej funkcie vlákna. V prípade, že sa kompilácia nepodarí, dokážeme získať detailné informácie o dôvodoch chyby vrátane riadku v zdrojovom kóde, na ktorom ku chybe došlo.

V prípade úspechu kód spustíme. Kód sa vykonáva rovnako ako v bežných Javascript enginoch, ktoré poznáme. Automaticky beží kód napísaný mimo JS funkcie v poradí v ktorom sa nachádza v súbore, a prípadné funkcie volané z tohoto behu. Engine si počas behu ponecháva kontext – akýsi stav Javascriptového prostredia. Po skončení behu zdrojového kódu, tento engine s inicializovaným kontextom existuje ďalej, a my sa k nemu môžeme vracať, a pracovať s ním. Engine zaniká až tesne pred koncom existencie vlákna, kde kontext explicitne musíme zlikvidovať.

Pred tým než sa tak ale stane, chceme dať aplikácii možnosť tento kontext využívať na vykonávanie experimentu. Vstúpime preto do cyklu, z ktorého ďalej budeme kontrolovať beh skriptu.

Cyklus opakovane kontroluje či neprišiel podnet k ukončeniu experimentu, napríklad z aplikácie CONTROLLER. Jeho hlavnou úlohou je však obsluha funkcií v skripte. Telo cyklu priamo volá funkcie skriptu na základe dvoch podnetov. Jedným je časovač, druhým sú udalosti.

Časovač je komponent napísaný v C++. Do skriptu vystavuje API, pomocou ktorého je možno naplánovať beh ľubovoľného kódu zo skriptu. O práci tohoto časovača sme si už niečo povedali v kapitole venujúcej sa behu experimentu. Konkrétne sme si vysvetlili, ako tento časovač pracuje, aby bol agnostický ku pozastaveniu kódu, a počas doby suspendácie nehromadil načasované udalosti. Registrácie funkcií zo skriptu sú viditeľné v C++ kóde a táto mapa je využívaná pri spúšťaní týchto funkcií. Kód v cykle porovnáva naplánované časy behu funkcií proti ich poslednému behu(tzv. !"schedules") a v prípade uplynutia doby čakania, tieto funkcie pustí.

Druhou komponentou sú eventy. Komponent tak isto ako časovač vystavuje do skriptu API, pomocou ktorého je možno k udalosti priradiť ľubovoľnú funkciu skriptu. Eventy majú formu textových reťazcov a ich podoba nie je nijako vymedzená. Eventy prichádzajú najme zo sieťovej komunikácie s CONTROLLER

aplikáciou, alebo prostredia UE4. Aj ich použitie si ukážeme v časti venovanej skriptovaniu. Nateraz nám postačí vedieť, že niekde v aplikácii je cyklus, ktorý kontroluje či na objekte zdieľanom medzi UE4 aplikáciou a skriptovacím vláknom, nie je vyvolaný event. V prípade že je, vlákno prehľadá registrácie eventov, a zavolá registrovanú funkciu.

Žiadna z týchto dvoch komponent, nie je štandardom Javascriptu a museli byť doprogramované ako súčasť práce. Snažili sme sa o implementáciu, ktorá by sa spôsobom používania čo najviac priblížila našim skúsenostiam s pracou v technológiách, ktoré túto problematiku majú vyriešenú, konkrétne sme sa inšpirovali zdrojovými súbormi technológie Node.js.

V prípade prijatého signálu na ukončenie enginu, cyklus končí a s ním končí aj vykonávanie hlavnej funkcie vlákna. Pred jeho ukončením sa uprace skriptovací engine, aby pri ďalšom behu neobsahoval starý kontext a vlákno končí.

C++ do Skriptovacieho enginu

Ako sme spomenuli, budeme sa sústrediť na prepojenie UE4, C++ kódu a Blueprints, so skriptovacím enginom. Základným prvkom tejto štruktúry je registrácia C++ funkcie do enginu. Táto registrácia je podmienená API Google V8 a pre globálne vystavené funkcie, pre ktoré nevyžadujeme volanie objektu, vyzerá nasledovne:

```
1. Global->Set(V8Types::StringHandle("Name",isolate),
      FunctionTemplate::New(isolate, Func));
```

Kód 1: Registrácia globálnej funkcie do skript enginu

Týmto spôsobom zaregistrujeme C++ funkciu *Func* do skriptovacieho enginu pod menom *Name.*

Pre registráciu metód volaných na objekte je kód o málo zložitejší:

```
1. ot->SetInternalFieldCount(1);
2. Local<Object> oi = ot->NewInstance();
3. oi->SetInternalField(0, External::New(isolate, myClass));
4. Local<FunctionTemplate> setRotationTemplate =
5. FunctionTemplate::New(isolate, &AScriptedActor::SetRotation);
6. Local<Function> setRotationImplementation =
        setRotationTemplate->GetFunction();
7. setRotationImplementation->
        SetName(V8Types::StringHandle("SetRotation",isolate));
8. oi->Set(setRotationImplementation->GetName(),
   setRotationImplementation);
```

```
9. ...
10.global->Set(isolate, "ObjectName", objectInstance);
```

Kód 2: Registrácia objektu do skript enginu

Na riadkoch 1-3 musíme najskôr vytvoriť objekt v kontexte skript enginu, a ten spárujeme s C++ objektom *myClass*. Následne môžeme pridávať metódy z triedy tohoto objektu. V poslednom bode tento objekt pridáme do kontextu skript enginu, a zvolíme meno („ObjectName“), ktorým na inštanciu objektu budeme v skripte odkazovať.

Obmedzením je, že všetky metódy, ktoré sú do enginu pridávané, musia spĺňať jednotné statické rozhranie dané notáciou:

static void Method(const FunctionCallbackInfo<Value>& args){...}

Z tohoto dôvodu, sú všetky funkčné objekty odtienené od Javascriptu samostatnou vrstvou. Jedná sa o vrstvu *Scripted*-tried, ktoré dedia od funkčných tried a ich logiku prevolávajú. Sami však obsahujú iba metódy spĺňajúce rozhranie dané Google V8. Na úrovni týchto tried dochádza k premapovaniu rozhrania Google V8 na do rozhrania UE4, teda tradičného programovacieho modelu.

Príklad mapovania obalu *ScriptedActor* do funkcionality triedy *Actor,* od ktorej trieda *ScriptedActor* dedí, vyzerá nasledovne:

```
1. void AScriptedActor::SetVisibility(const
      v8::FunctionCallbackInfo<v8::Value>& args){
2. auto myClass = AScriptedActor::This(args);
3. myClass->SetVisible(args[0]->BooleanValue());
4. }
```

Kód 3: Vystavenie funkcie UE4 do Google V8 enginu

Táto metóda z parametru *args* získa informácie o registrovanom objekte, a parametroch volania. *SetVisible(bool)*, je vlastnosťou rodičovskej triedy Actor.

Každý objekt v UE4, ktorý je vystavený do skriptovacieho enginu má unikátne meno, pod ktorým v skriptovacom engine vystupuje. Dôvodom je, že v prostredí skriptu sa týmto menom odkazujeme ku konkrétnym inštanciám objektov vystavených z prostredia skriptu.

Prepojenie C++ kódu do skriptovacieho enginu je teda zrejmé. V nasledujúcich častiach sa budeme venovať jednotlivým typom objektov, vystavených do skriptovacieho enginu. Pokúsime sa na príkladoch ukázať, ako sme prepojili technológiu Blueprints s C++ kódom, aby sme mohli splniť všetky požiadavky na

systém, vyplývajúce zo špecifikácie. Účelom nasledujúcej časti nie je dokumentovať aplikáciu, ale vybrať reprezentatívnu množinu dizajnových a programátorských postupov, ktoré sme použili na dosiahnutie cieľa.

Logovanie

Logovanie prebieha na dvoch úrovniach.

Prvou úrovňou je logovanie špecifické pre experiment, čiastočne zo skriptovacieho enginu. Pre logovanie na tejto úrovni sa používajú 3 výstupy, všetky majú formu súboru. Logovacie výstupy združuje trieda *ExperimentLog*, a do skriptovacieho enginu sú vystavené pomocou globálne prístupných funkcií, rozdelených podľa zamerania. Sú nimi funkcie *LogT*, ktorá loguje načasované udalosti, a funkcia *LogE*, ktorá loguje eventuálne udalosti. Tretí výstup je logovanie takzvanej hlavičky. Jedná sa o záznam konfiguračných parametrov experimentu v samostatnom súbore. Tento log nie je pre zápis prístupný manuálne. Existencia logovacích prúdov použitých pre záznam je spätá s existenciou experimentu.

Druhou úrovňou logovania je logovanie ladiace. Infraštruktúra ladiaceho logovania sa používala pre záznam udalostí v UE4, pre prípad chyby. Momentálne je aplikácia v stabilnom stave, takže žiaden ladiaci výpis v tejto vetve vývoja nevzniká. Obe úrovne logovania používajú štandardný C++ *ofstream*.

Časovanie

Timing je komponent, ktorý obaľuje a obohacuje nižšie C++ knižnice, venujúce sa meraniu času a intervalov, *ctime* a *chrono*. Komponent vracia čas v dvoch rôznych presnostiach a zároveň sústredí logiku formátovania času pre lepšie čitateľný výstup. Komponentu *Timing* používame na určenie času vo všetkých komponentoch, o ktorých sme v predošlom texte v súvislosti s časom hovorili.

Zakázané a cieľové oblasti

Avoidance (zakázané) a target (cieľové) oblasti vychádzajú zo základného *SpaNavActor* objektu dodaného UE4. Pre začlenenie do experimentu sme potrebovali obohatiť *SpaNavActor* objekt o vlastnosti, ktoré by bolo možno vystaviť do skriptovacieho enginu a Blueprints editora. Takto umožníme meniť dynamické alebo statické chovanie experimentu aj bez zásahu do C++ kódu a následnej

opätovnej kompilácie. Popísať, ako boli vystavené funkcie nakoniec použité v Blueprints je záležitosťou užívateľskej dokumentácie.

Okrem iného sme implementovali funkcie pre:

- nastavenie polohy objektu
- nastavenie viditeľnosti objektu
- nastavenie rodiča objektu
- nastavenie zvuku objektu

Pre manipuláciu z Blueprints editora sme zverejnili API ktoré obsahuje aj funkcie pre:

- nastavenie rotácie objektu
- spracovanie udalostí o kolíziách
- získanie informácií o stave objektu

Pre účely skriptovania, avoidance a target arei používajú skriptovaciu proxy triedu, ktorú sme si popísali v časti o skriptovaní, s názvom *ScriptedActor*. Z nej sú prevolávané metódy triedy *SpaNavActor*. Nakoľko UE4 Engine neumožňuje priamo zasahovať do dodaných atribútov UE objektov z iných než z hlavného vlákna (napríklad z vlákna skriptovacieho enginu), navrhli sme systém, ktorý zmení nami definované medzi-hodnoty, a následne v ticku enginu si cieľový objekt zmenené hodnoty zmení sám. Metódy, ktoré môžu nastavovať hodnoty na objekte používajú mennú konvenciu s predponou "Tick".

Zvláštnou skupinou sú metódy pracujúce s kolíziami. Tieto metódy stav objektu nemenia, ale reagujú na fakt, že až na úrovni skriptovania chceme rozhodovať ako reagovať na kolízie. K úplnému prepojeniu týchto dvoch svetov, využívame ďalší konštrukt, ktorý sme si už popísali, udalosti v skriptovacom engine.

Kolízia je v engine UE4 udalosť vyvolaná na aktorovi a my ju teda preložíme do udalosti na aktorovi v skript engine. To zariadime tak, že vyvoláme k spracovaniu udalosť, ktorej meno pozostáva z mena objektu, pod ktorým je v skriptovacom engine zaregistrovaný, a pomocou mena udalosti. Záznam následne zaradíme do kontajnera udalostí. Pri spracovaní udalostí skriptovacím enginom sa rozhodne, ktorý handler, obsluhujúca funkcia, registrovaná v kóde skriptu bude udalosť spracovať, za predpokladu že k jej spracovaniu vôbec nejaký skriptovací kód zaregistrovaný je. Ako je popísaný systém implementovaný, vidíme na nasledujúcich riadkoch kódu:

```
1. string name = TCHAR_TO_UTF8(*this->ScriptName);
```

```
2. string eventName = ".OverlapBegin";
3. tto->EventName = name + eventName;
4. tto->EventPresent = true;
```

Kód 4: Vyvolanie udalosti pre začiatok kolízie

Postava testovaného subjektu

Komponent hráča pozostáva podľa dizajnového vzoru predošlých komponent z dvoch tried. Trieda *SpaNavCharacter* nás zaujímať nebude, pretože tá je štandardne dodaná enginom a túto pôvodnú triedu sme nijak neupravili. Pozrieme sa však na triedu, ktorá od nej dedí. *ScriptedCharacter* – trieda vystavujúca hráča do skriptovacieho enginu. Špecifikácia experimentu požadovala prácu s polohou a rotáciou hráča. Preto vystavujeme metódy na získanie polohy (Z,Y,Z) a rotácie (Pitch, Roll, Yaw). Vďaka registrácii objektu do skriptovacieho enginu pod menom “Player”, môžeme pristupovať v skripte k týmto údajom nasledujúcim spôsobom:

```
1. function LogPlayerStats(){
2. var loc = Player.GetLocation();
3. var rot = Player.GetRotation();
4. LogT("Location: [" + loc.X + ";" + loc.Y + ";" + loc.Z + "]");
5. LogT("Rotation: [" + rot.Yaw + ";" + rot.Roll + ";" + rot.Pitch + "]");
6. }
```

Kód 5: Záznam polohy a rotácie postavy hráča v prostredí skriptu

Kde *GetLocation* a *GetRotation* sú registračné mená funkcií pre získanie polohy a natočenia v skripte a *LogT* je funkcia pre zápis do logu. loc a *rot* sú potom objekty skriptu obsahujúce pomenované jednotlivé zložky vektorov. Podrobne sa na skriptový kód pozrieme v užívateľskej dokumentácii.

Zvuky

Programovo sa o obsluhu zvukových efektov stará komponent *SoundActor*. Engine ju natívne neposkytuje a preto sme ju museli vytvoriť sami. Špecifikácia vyžaduje v prostredí skriptu reagovať na rôzne udalosti aktivovaním zvukov. Zvuky sa delia na dva typy. Dokážeme pracovať s jednorazovými zvukmi a slučkami. Jednorazový zvuk je takzvaný “Fire and Forget”, čo znamená že je možno ho spustiť a aplikácia sa ďalej o prehrávanie nemusí starať. Slučku je potreba v správnom čase aktivovať, a bez ďalšieho zásahu sa slučka bude teoreticky do nekonečna opakovať. Je treba ju preto v správnom čase opäť vypnúť. Oba typy zvukov majú v experimente svoje

miesto. Na túto skutočnosť reaguje implementácia objektu. Púšťanie konkrétnych zvukov rieši Blueprints, teda popis bude uvedený v užívateľskej dokumentácii. Trieda *SoundActor* plní proxy funkciu medzi skriptovacím enginom a Blueprints. Pomocou tejto triedy potrebujeme rozlíšiť, či chceme pustiť jednorazový zvuk alebo slučku, a predať do editora informáciu o zvolenom zvuku. Riešením sú dve trojice funkcií ako je možné vidieť v nasledujúcich výpisoch:

```
1. static void PlaySound(const v8::FunctionCallbackInfo<v8::Value>& args);
2. static void PlayCue(const v8::FunctionCallbackInfo<v8::Value>& args);
3. static void StopCue(const v8::FunctionCallbackInfo<v8::Value>& args);
```

Kód 6: Rozhranie skriptu a kódu C++

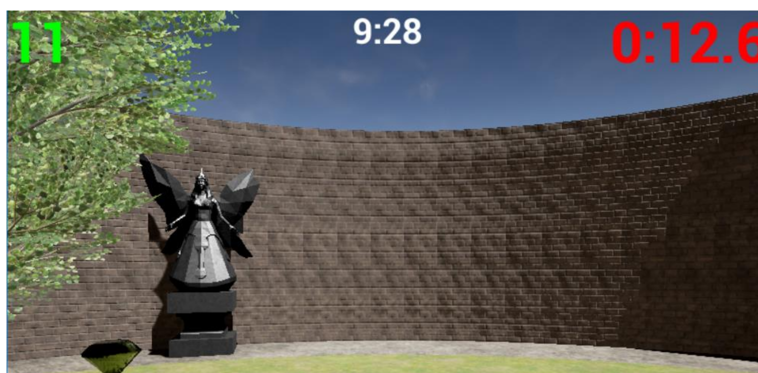
```
1. UFUNCTION(BlueprintImplementableEvent, Category = "Scripting")
       void PlayCue(int32 soundIndex);
2. UFUNCTION(BlueprintImplementableEvent, Category = "Scripting")
       void StopCue(int32 soundIndex);
3. UFUNCTION(BlueprintImplementableEvent, Category = "Scripting")
       void PlaySound(int32 soundIndex);
```

Kód 7: Analogicky pre rozhranie C++ a Blueprints

Kde *Play*/*StopCue* spúšťa alebo zastavuje slučku a *PlaySound* púšťa niektorý z jednorazových zvukov. Hodnota *soundIndex* určuje poradie zvoleného zvuku v poli zvukov. Hodnota *soundIndex* je do Blueprints prenesená z prostredia skriptu, takže voľba zvuku je súčasťou definície experimentu, ako bolo požadované v špecifikácii.

HUD

HUD (head up display), je komponent UE4 enginu, ktorý slúži ako podklad pre komunikáciu programu s užívateľom. Na plochu HUD je možno zobrazovať rôzne informácie o behu programu. Tieto displeje dokážu zobrazovať textové správy, údaje o skóre, časomieru alebo ako uvidíme neskôr, môžeme ich využiť aj premietanie náhľadu experimentu. Naša aplikácia momentálne obsahuje 5 head up displejov. Jeden displej je naviazaný na definíciu konkrétneho experimentu, vychádza z jeho špecifikácie. Podľa špecifikácie, pre experiment AAPA, ktorý sme implementovali, je potreba zobrazovať údaje o ostávajúcom čase do konca experimentu (bielou farbou), dobu výskytu v zakázaných oblastiach (červenou farbou) a počte vstupov do cieľových oblastí (zelenou farbou).



Obrázok 12: Head up display pre bežiaci experiment

Vizuálna podoba head up displeja je definovaná pomocou Blueprints. Informácie, ktoré zobrazujeme ale vznikajú v prostredí skriptu. Preto je opäť potrebné, tak ako sme si už ukázali v predošlej kapitole, prepojiť skriptovací engine s Blueprints.

Ako je to riešené pre HUD, ukazuje nasledujúci kód:

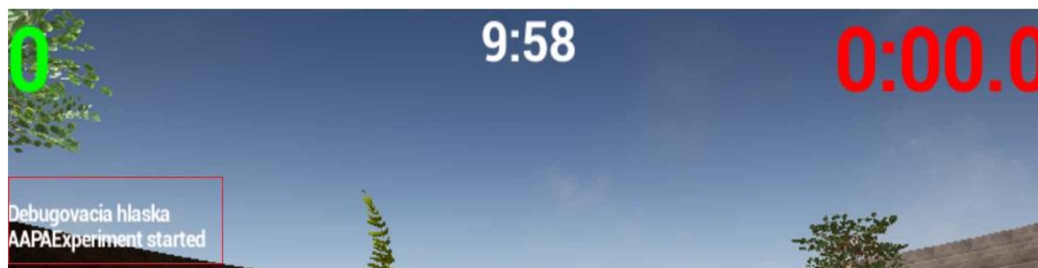
```
1. UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Binding")
2. FString AvoidanceText;
3. UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Binding")
4. FString TimerText = "0:00";
5. UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Binding")
6. FString DestinationText;

7. static void SetAvoidanceText(
      const v8::FunctionCallbackInfo<v8::Value>& args);
8. static void SetTime(
      const v8::FunctionCallbackInfo<v8::Value>& args);
9. static void SetDestinationText(
      const v8::FunctionCallbackInfo<v8::Value>& args);
```

Kód 8: Registrácia HUD vlastností do Blueprints z C++ a do skript enginu

Sada prvých troch funkcií popisuje vlastnosti, ktoré sú viditeľné z prostredia Blueprints. Zápis druhej sady funkcií by mal byť čitateľovi povedomý, sú to funkcie, ktoré je možno zavolať z prostredia skript enginu. Implementáciou Set-funkcií je naplnenie vlastností viditeľných z Blueprints. Týmto spôsobom dosiahneme prenos informácií medzi komponentmi. Popis finálneho spracovania a zobrazenia je záležitosťou užívateľskej dokumentácie, kde sa budeme venovať technológii Blueprints.

Ďalším head up displejom, ktorý sme pripravili je experiment ladiaci HUD. Tento displej slúži pre výpis ladiaceho výstupu z prostredia experimentu. Technicky je implementovaný rovnako ako predošlý experimentálny HUD, ale nie je viazaný na konkrétny experiment. Myšlienka je taká, že skriptovaný kód je možno ladiť priamo pri behu aplikácie výpisom správ, a v prípade že experiment bude hotový, ladiaci HUD sa jednoducho deaktivuje. V prípade že sa objavia problémy pri behu experimentu, je možno displej jednoducho opäť aktivovať.

Ako tieto výpisy vyzerajú je možno vidieť na obrázku nižšie v červenom štvorci.



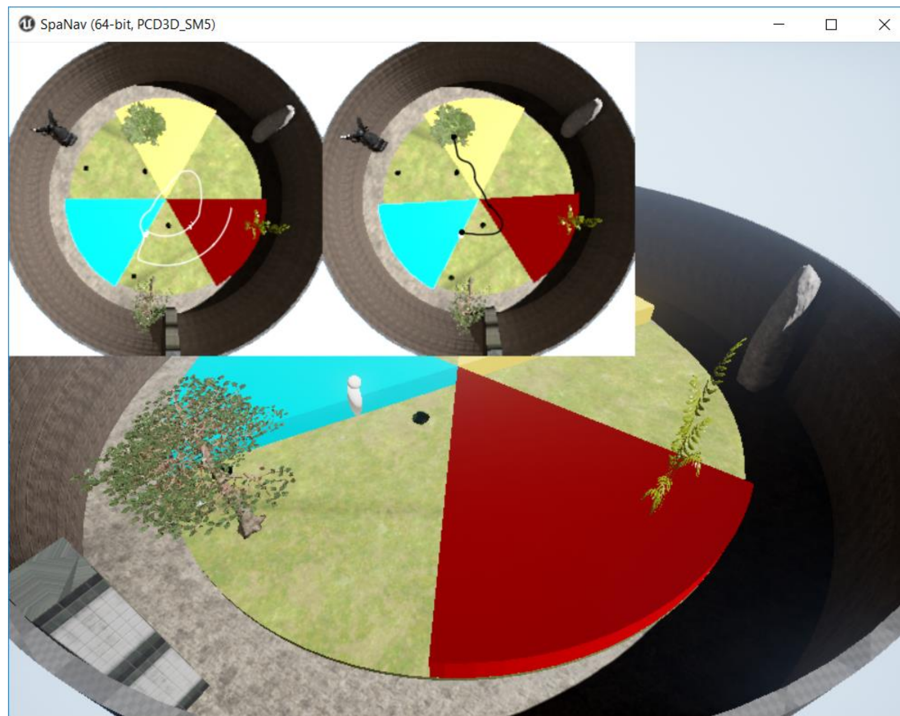
Obrázok 13: Ladiaci výstup v debug HUD

Posledným HUD ktorý má aj C++ triedu na pozadí, je východzí prázdny HUD, poskytovaný enginom. My pre tento HUD zatiaľ v projekte nemáme využitie, a preto o ňom viac písať nebudeme.

Aplikácia obsahuje ešte dva dôležité head up displeje. Napriek ich Blueprints povahe, ich popis uvedieme v časti programátorskej dokumentácie. Dôvodom pre implementáciu týchto komponent v Blueprints bola voľba čo najnatívnejšieho prostredia pre vývoj potrebných funkcií. Obecne vzaté ale stále do veľkej miery platí, že užívateľská časť aplikácie začína na jednej strane skriptovacím enginom a na strane druhej editorom UE4 a Blueprints. Nasledujúcu kapitolu teda venujeme popisu ostatných dvoch HUD.

Náhľadový systém

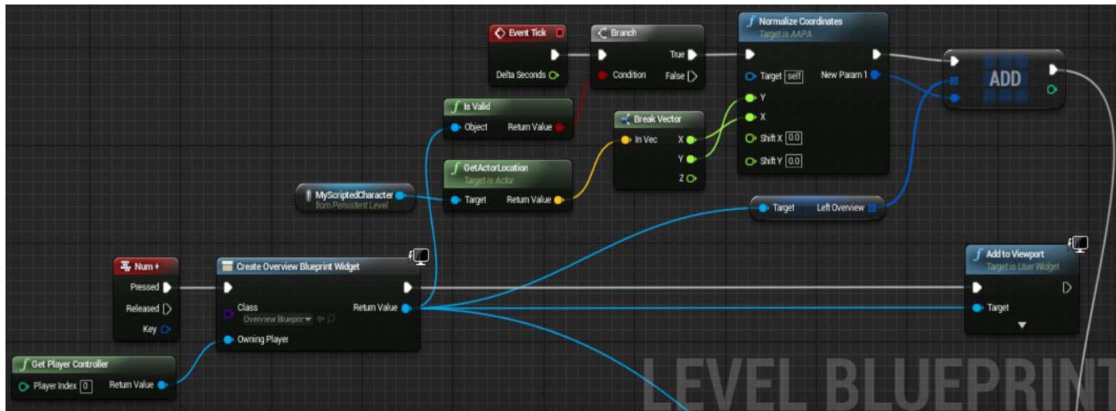
Aby sme mali predstavu o čom bude nasledujúca kapitola, na nasledujúcom obrázku je náhľad, ako ho vidí vedúci experimentu v okne na počítači, ktorý kontroluje beh experimentu (obecne sa jedná o ľubovoľný počítač na lokálnej sieti, ale predpoklad je, že pobeží spolu s CONTROLLER aplikáciou).



Obrázok 14: Náhľadový systém. SpaNav 2.0

Náhľadový systém, ako sme ho navrhli, má tri funkčné časti. Prvá časť získava údaje o polohe hráča. Špecifikácia požaduje 2 rôzne náhľady s rôznymi požiadavkami na typ zaznamenaných údajov o polohe. Záznam na obrázku naľavo, počíta absolútnu polohu hráča vo svete. Pre jednoduchšiu predstavu, čo to znamená, ak hráč ostáva na otáčajúcej sa platforme nehybne stáť, postupne získané údaje o polohe vykreslia po zaznačení bielou farbou kruhový výsek. Časti týchto výsekov je možno pozorovať aj na obrázku. Ak sa hráč hýbe po otáčajúcej sa plošine, výsledná trasa je zložením jeho pohybu na platforme s pohybom platformy. V skutočnosti je však tento prípad jednoduchším z uvedených, pretože sa jednoducho získava globálna poloha hráča v hre, a táto je potom prepočítaná do koordinátu 2D priestoru s počiatkom v ľavom hornom rohu.

Postup je znázornený na nasledujúcom obrázku:

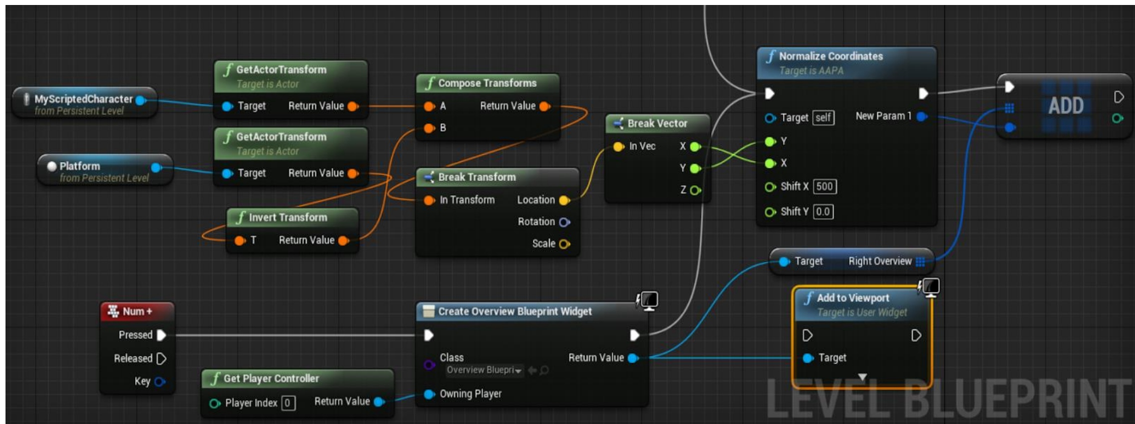


Obrázok 15: Implementácia zaznamenávania globálnej polohy hráča v Blueprints

Skript, ktorý začína v uzle *Num+* na obrázku naľavo dole, má za úlohu inicializovať head up displej na zobrazovanie náhľadu, ako reakciu na stlačenie klávesy “+”. Skript plynie na obrázku14 po bielej línii a končí v uzle “Add to Viewport”, ktorý zaistí že sa head up displej zobrazí. Skript pre získanie polohy postavy začína v uzle “Event Tick”, ktorý sa spustí po každom frame v hre. Po bielej línii nasleduje podmienka, ktorá kontroluje, existenciu head up displeja a následne funkcia *Normalize Coordinates*. Tá na vstupe očakáva X,Y koordináty hráča, ktoré získa tým že získa objekt postavy (*MyScriptCharacter*), z nej získa X,Y súradnice (*GetActorLocation* a *BreakVector*). 2D súradná sústava je proti 3D sústave natočená a má aj posunutý stred. Preto sa koordináty pred vstupom do funkcie prehodia, a funkcia akceptuje dva parametre pre posunutie stredu súradnicovej sústavy. Tieto parametre sú pevne dané, počas behu programu sa nemenia a sú špecifické pre jednotlivé okná náhľadu.

Prepočet do koordinátu 2D náhľadu má na starosti funkcia *Normalize Coordinates*. Táto funkcia prevedie koordináty do škály zodpovedajúcej zmenšenej verzii mapy a posunie ich v súradniciach 2D tak, aby stred 3D náhľadu zodpovedal stredu 2D náhľadu. Výsledkom je bod v 2D priestore, zodpovedajúci presnej polohe hráča, na ktorého sa dívame v 3D náhľade z kamery v strede mapy nasmerovanej kolmo k zemi. Posledným krokom je predanie získaných údajov o polohe do head up displeja. Ten za týmto účelom HUD vystavuje vlastnosť “LeftOverview”. Jedná sa o zoznam jednotlivých bodov, ktorými hráč prešiel. Nové koordináty predáme do zoznamu na koniec, pomocou uzlu “ADD”.

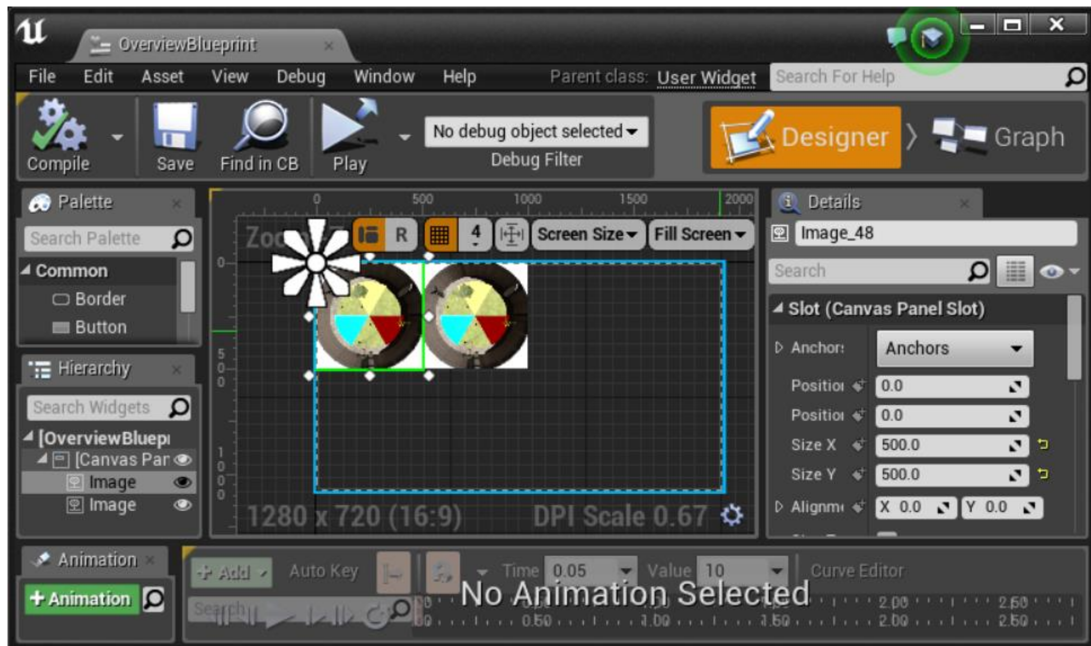
Na obrázku náhľadu vpravo, vidíme druhý záznam, kde je poloha hráča počítaná vzhľadom na otáčajúcu sa platformu. Pre ilustráciu, ak hráč stojí na platforme nehybne, výsledkom je jeden jediný bod. Údaje získavame analogicky ako v prvom prípade ale navyše korigujeme polohu hráča s globálnymi koordinátami o natočenie platformy. Implementáciu popísaného prípadu v Blueprints vidíme nižšie:



Obrázok 16: Implementácia náhľadu statického voči platforme v Blueprints

Získame objekt platformy (uzol *Platform*), spočítame inverznú transformáciu(*GetActorTransform* plus *InvertTransform*), tá obsahuje aj rotáciu, a túto opačnú transformáciu spojíme s transformáciou hráča (*ComposeTransforms*). Ďalej postupujeme ako v prvom prípade, s tým rozdielom, že pri volaní funkcie *Normalize Coordinates* použijeme iný posuv (*ShiftX* a *ShiftY*) a získané údaje pripojíme na koniec zoznamu *RightOverview* náhľadového head up displeja.

Samotný HUD je definovaný nasledujúcim spôsobom

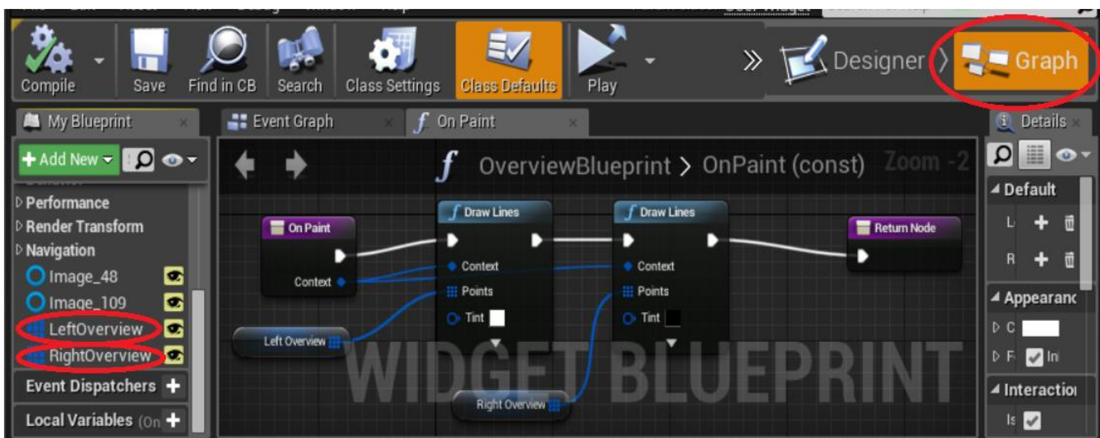


Obrázok 17: Definícia náhľadového HUD v editore

Vizuálna stránka obsahuje dva prvky typu Image so špeciálnym obsahom, zobrazujúcim aktuálny záznam z kamery umiestnenej nad plochou experimentu.

Obsahom prvkov je vlastný materiál (*RenderMaterial*) ktorému sme do *TextureSample* uzla vložili prvok *TextureRenderTarget2DTextureRenderTarget2D*. Prvok sme vytvorili z existujúcej kamery, ktorú sme pred tým umiestnili nad stred experimentu, so záberom smerom kolmo dolu. Detaily implementácie tohto materiálu je možné si pozrieť v priloženej prílohe, na konci práce.

Definícia náhľadového displeja obsahuje ešte jednu konfiguračnú obrazovku, ktorú si popíšeme. Je to obrazovka, ktorá definuje logiku na pozadí prvku náhľadu.

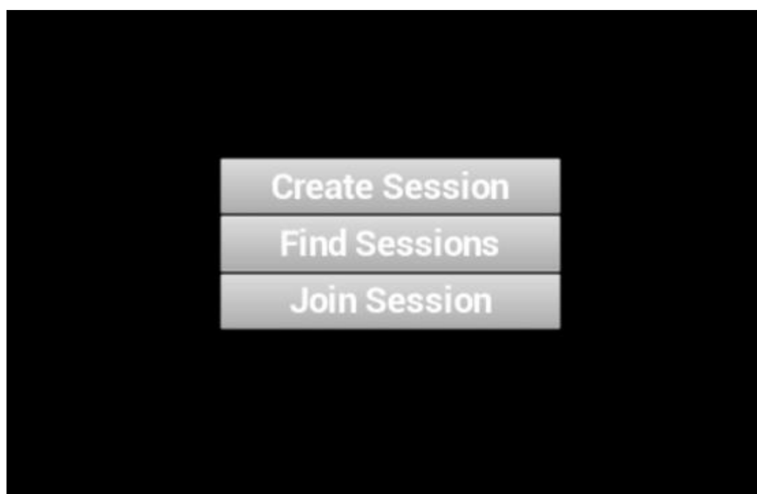


Obrázok 18: Implementácia vykreslenia trajektórií

Na tejto obrazovke definujeme vlastnosti Left/RightOverview, ktoré sme plnili v predošlej kapitole. Logika náhľadu v tomto prípade, reaguje na udalosť *OnPaint*, ktorá zavolá uzol *Draw Lines* a ten vykreslí najprv hrany definované bodmi v liste *LeftOverview*, a následne sa logika opakuje pre body v liste *RightOverview*. Uzol *Draw Lines* dostáva ešte argument farby, ktorou hrany vykreslí.

Popísaným spôsobom sme skonštruovali systém, ktorý umožňuje užívateľovi zobraziť náhľadovú obrazovku. Kľúčovým aspektom špecifikácie náhľadu však bola možnosť analyzovať tento náhľad z počítača na sieti. To zatiaľ popísaný systém nepodporuje, ale v ďalšej kapitole si ukážeme ako sme vyriešili aj toto zadanie.

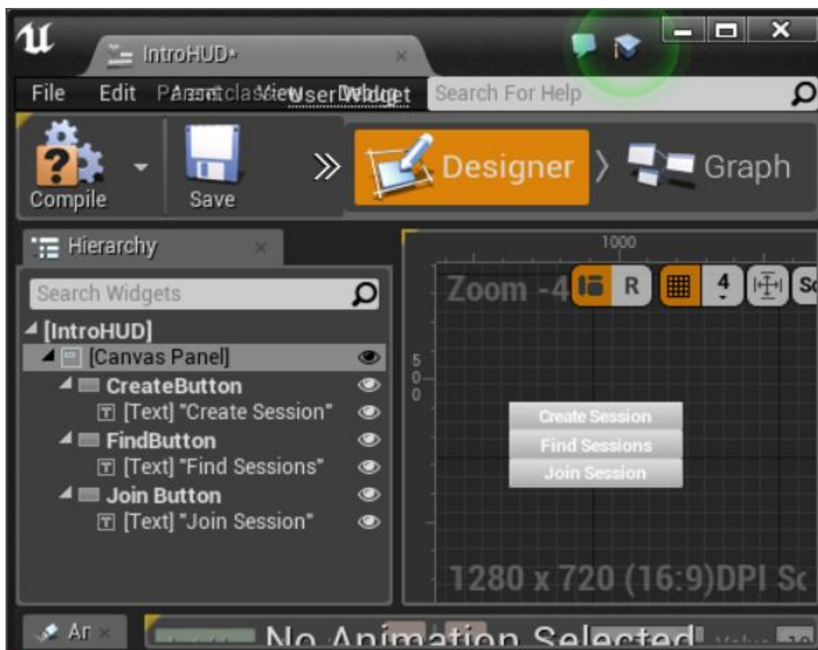
Úvodná obrazovka (multiplayer setup)



Obrázok 19: HUD úvodnej obrazovky

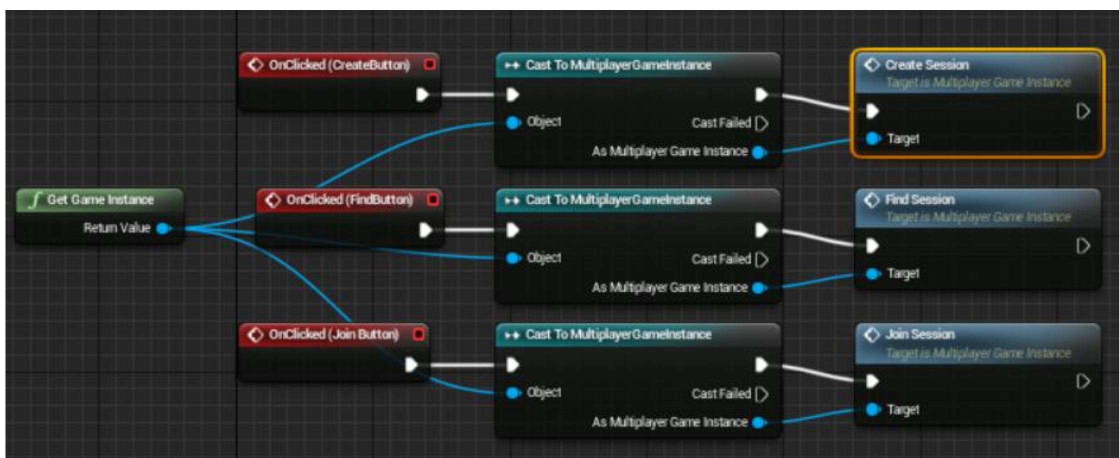
Kvôli možnosti náhľadu zo vzdialeného počítača, musel byť do aplikácie zavedený systém "multiplayeru". UE4 engine je na túto možnosť dobre pripravený, slabú stránku v tom prípade však ukazuje opäť raz dokumentácia. Metódou analýzy vzorových projektov, a pokusov a omylov, sa nám však podarilo systém pripraviť. Kľúčové pre pochopenie ako multiplayer funguje a jeho implementáciu, bolo pochopiť závislosti medzi vrstvami a komponentmi enginu. To nám umožňuje spraviť rozhodnutie, na ktorej vrstve skript volať, a ktoré komponenty budú cieľové pre naše volania.

Výsledkom implementácie je opäť HUD s troma tlačidlami, vizuálna stránka ktorého, je znázornená na nasledujúcom obrázku:



Obrázok 20: Definícia HUD pre úvodnú obrazovku

Prepnutie definície displeja do módu *Graph* nám odhalí logiku na pozadí



Obrázok 21: Obsluha tlačidiel úvodnej obrazovky

Definovali sme 3 funkcie, ktoré sú volané ako reakcia na stačenie jednotlivých tlačidiel. Detailnú implementáciu týchto funkcií možno vidieť v prílohe, na konci práce. Pre tieto funkcie však musela vzniknúť aj samostatná implementácia komponenty *GameInstance*. Jedná sa o objekt, ktorý pretrváva od spustenia programu po jeho koniec. Voľbu našej novej *GameInstance* komponenty sme si vynútili jej zvolením v paneli nastavení.

GameInstance je presne tá úroveň, z ktorej chceme obsluhovať na jednej strane vytvorenie sieťovej relácie(*session*) a na strane druhej pripojenie k existujúcej relácii. Pre zobrazenie vstupnej obrazovky máme vytvorený ďalší vlastný level objekt (scénu/mapu). Aby sme nemuseli načítať level experimentu pred tým než sa rozhodne, či plánujeme pokračovať, máme prichystaný vizuálne prázdny level, ktorého jedinou úlohou je zobraziť úvodnú obrazovku.

Takto naimplementovaný systém má veľmi blízko k tomu čo potrebujeme, ale po spustení si všimneme, že pripojením ďalšieho hráča, sa nám automaticky vytvorí tento hráč aj v experimente. My ale nechceme, aby bol testovaný subjekt rušený prítomnosťou ďalšej osoby a preto musíme postavu ďalších pripojených aplikácií odstrániť. Toto implicitné chovanie má na svedomí nastavenie objektu *GameMode->Default Pawn*. My potrebujeme, aby bola táto hodnota nastavená na „Spectator“ – teda pozorovateľ. Toto nastavenie sa dá zmeniť na dvoch miestach. Buď v nastaveniach *Project Settings*, kde sme nastavovali *GameInstanceClass*, alebo v C++ časti objektu *GameMode*. To, ktorú cestu zvolíme závisí najmä na tom, ako veľmi statické chovanie má nastavenie mať. Neočakávame, že si užívateľ dodaného editora bude meniť práve hodnotu Default Pawn, ale vzhľadom k tomu, že naše riešenie by jeho zmenou prestalo fungovať, poistíme si túto skutočnosť tým, že nastavíme hodnotu v kóde C++ v triede *SpaNavGameMode*:

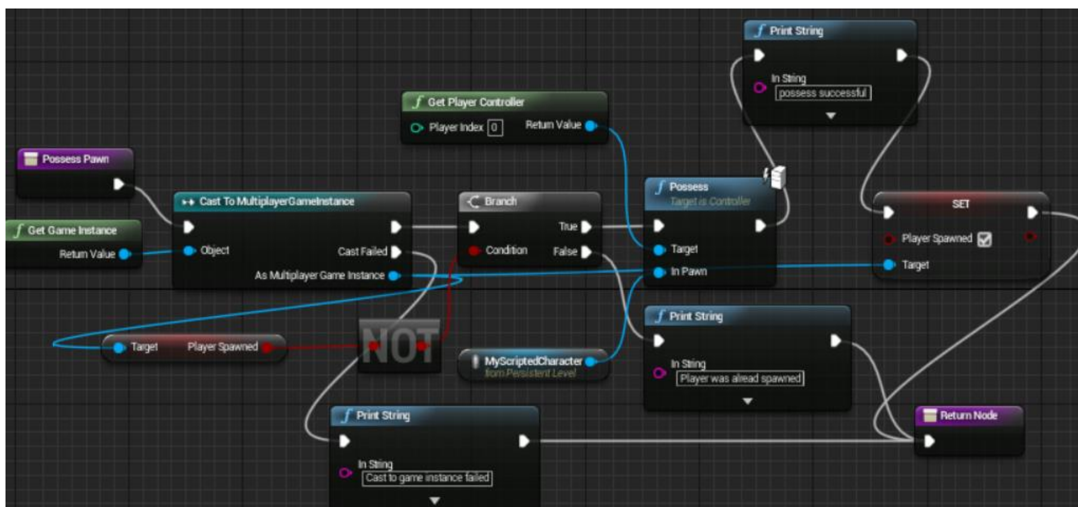
```
1. // set default pawn class to our Blueprinted character
2. DefaultPawnClass = SpectatorClass;
```

Kód 9: Nastavenie implicitnej hodnoty pre typ postavy

Implicitne dodaný *SpaNavGameMode* má tú výhodu, že jeho hodnoty nie je možno meniť v editore. Ak si bude užívateľ priať zmenu nastavení, bude si musieť vytvoriť vlastný *GameMode*. Výsledkom je, že vždy keď sa pripojí nový užívateľ, objaví sa automaticky v režime „Spectator".

Posledným krokom, je zaistiť, že aspoň testovaný subjekt dostane postavu. To je dôležité z toho dôvodu, že postava má iba obmedzené možnosti pohybu v experimente, na rozdiel od pozorovateľa. Jeho možnosti sú veľmi podobné možnostiam reálnych ľudí.

Skript, ktorý dodá testovanému subjektu telo môžeme vidieť nižšie.



Obrázok 22: Nastavenie postavy pre prvého hráča

Tento skript sa pustí pri každom spustení levelu experimentu. Jeho kľúčovou vlastnosťou však je, že testuje premennú objektu *PlayerSpawned,* ktorá je pri prvom behu nastavená na hodnotu *False*. Až sa spustí prvá aplikácia, skript zbehne, spojí aktuálneho užívateľa s postavou, ktorú sme prichystali na plochu, a nastaví PlayerSpawned na *True*. Od tohoto momentu každý ďalší beh skriptom vybočí v úvode skriptu mimo logiku pripojenia na postavu a všetci ďalší užívatelia ostávajú v režime pozorovateľa.

V tejto fáze máme prichystaný systém, ktorý umožňuje pripojiť sa k bežiacemu experimentu z počítača na sieti, a ktorémukoľvek účastníkovi umožňuje zapnúť si režim náhľadu. Prvý užívateľ, je v režime testovaného subjektu, a každý ďalší užívateľ je v režime pozorovateľ, pre prvého užívateľa neviditeľný s absolútne voľným pohybom v testovacom priestore. Tým sme splnili zadanie, ktoré sme popísali v úvode kapitoly.

Užívateľská dokumentácia

Užívateľom sa v tomto prípade rozumie osoba, ktorá je schopná vyvíjať experiment v prostredí SpaNav, na úrovni Javascriptu, Blueprints a obecne schopná používať UE4 editor. Užívateľská dokumentácia z tohoto dôvodu bude viac technická a bude obsahovať prvky, ktoré by boli inak mohli patriť do programátorskej dokumentácie. Rozdelenie padlo na hranicu medzi C++ kódom a skompilovaným editorom. Aplikácia je vyvíjaná spôsobom, aby zásahy do C++ kódu boli skôr výnimočné, aj keď úplne eliminovať to nejde, kým nedokážeme presne popísať všetky experimenty, ktoré kedy v systéme budú vyvinuté. Zámer však je, aby si do veľkej miery užívatelia vystačili práve s užívateľskými prvkami. Užívatelia majú k dispozícii:

- **CONTROLLER**– top level konfigurácia experimentu
- **EDITOR** – vizuálna stránka experiment a skriptovanie v Blueprints
- **EXPERIMENT** – Javascriptový kód pre popis logiky experimentu

Myšlienka je, že užívateľ si v editore pripraví komponenty pre potreby experimentu, a pomocou Blueprints zadefinuje primitívne schopnosti týchto komponent, a tieto schopnosti potom do zložitejšej logiky prepojí, a obohatí experiment o zložitejšie konštrukty v prostredí Javascriptu. Konfigurácia experimentu má potom skôr katalógový účel, umožňuje najmä odlíšiť jednotlivé behy experimentu pre rôzne testované subjekty.

V nasledujúcich kapitolách si postupne ukážeme ako sme postupovali pri vytváraní vzorového experimentu AAPA a to nám dá predstavu o tom, aké má systém možnosti a ako je zamýšľané jeho použitie.

CONTROLLER

O aplikácii CONTROLLER sme si už niečo povedali, zamerali sme sa však na náhľad a funkcie aplikácie na pozadí, na ktoré sme sa pozreli z programátorského hľadiska. Z užívateľského hľadiska nás bude zaujímať konfiguračný panel aplikácie

Create	Delete	AAPA	Save Configurations
	Subject name	Martin	
	Experiment File	C:\Projects\Javascripts\aapa2.0\a	Select file
	EventConfigurationFile	C:\Projects\Javascripts\aapa2.0\a	Select file
	Log File	C:\temp\LogFile	

Obrázok 23: Konfiguračný panel CONTROLLER aplikácie

Konfigurácia je unikátna pre každý jeden experiment a každý jeden testovaný subjekt. Aby sme jednotlivé konfigurácie odlíšili, každá konfigurácia má svoje meno. Na obrázku je Meno konfigurácie uvedené na prvom riadku. Meno konfigurácie sa zadáva pri jej vytvorení pomocou tlačidla *Create*, keď sa nám po jeho stlačení objaví jednoduchý formulár pre zadanie mena konfigurácie. Nová konfigurácia sa vytvorí automaticky s prázdnymi hodnotami. Po každej zmene hodnôt konfigurácie, je treba túto konfiguráciu uložiť pomocou tlačidla *Save Configuration*. Prvé pole, *Subject name*, ktoré potrebujeme definovať je pole s menom testovacieho subjektu. Toto meno sa neskôr objavuje v logovacích súboroch a jeho hodnota sa používa pri spracovaní výsledkov experimentu. Druhé pole volí zdrojový súbor so skript-kódom, ktorý sa nahraje do SLAVE aplikácie, kde sa bude vykonávať. Samotný skript môže byť definovaný vo viacerých súboroch skrze klauzulu pre pripojenie súboru, avšak vstupným bodom do skriptu musí byť práve jeden súbor. Meno tohoto súboru je treba uviesť do spomínaného poľa. Obsahom skriptovacích súborov sa budeme venovať v samostatnej kapitole. Pole *Event Configuration File* je súbor pre mapovanie kláves na udalosti, ktoré sa odosielajú po sieti do SLAVE aplikácie. Súbor momentálne pre potreby experimentu AAPA obsahuje jeden nasledujúci riadok:

```
1.  V;ToggleDestinations
```

Kód 10: Definícia vlastnej udalosti

Výsledkom je, že po stlačení klávesy V, za predpokladu že máme aktívne okno CONTROLLER, odošle sa po sieti udalosť s textom “ToggleDestinations”. Táto udalosť sa skrze C++ kód enginu vyvolá v skript engine, kde na ňu momentálne reagujeme tým, že zmeníme viditeľnosť zakázaných oblastí. Táto obsluha je definovaná skriptovaním experimentu. Pole *Log File* definuje šablónu cesty k výstupným logovacím súborom experimentu. Výstupom experimentu sú 3 logovacie súbory, a hodnota uvedená v tomto poli je koreňom absolútnej cesty k nim. Systém k tomuto koreňu pridá časovú pečiatku a typ výstupného súboru. Vo výsledku sa

môžeme hodnotou uvedenou v príklade na obrázku dopracovať k nasledujúcemu výstupu:

```
LogFile20180311114001.log
LogFile20180311114001_H.log
LogFile20180311114001_T.log
```

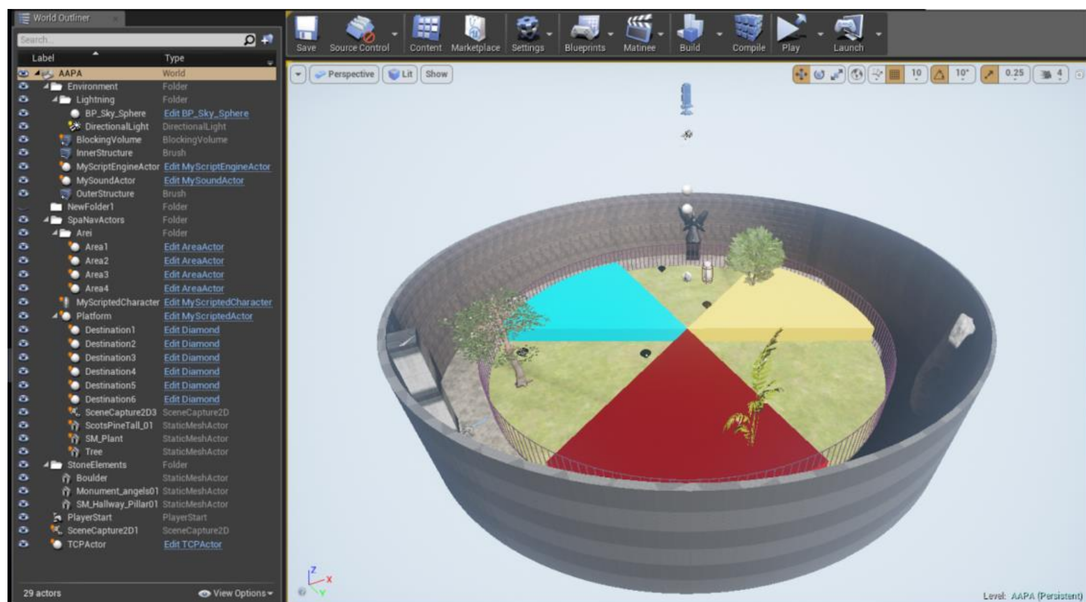
Výstup 3: Logovacie súbory vytvorené systémom SpaNav 2.0

Výstupom z experimentu a ich spracovaniu sa venuje samostatná kapitola. Týmto sme vyčerpali konfiguračné možnosti na strane klienta. V ďalšej kapitole sa budeme venovať experimentu AAPA, z pohľadu UE4 editora.

EDITOR

Mnohé implementačné detaily nám budú povedomé z časti programátorskej dokumentácie, pretože sa v mnohých prípadoch jedná o napojenie funkčných prvkov definovaných v C++ do skriptovacieho prostredia Blueprints a tým o jeho obohatenie.

Na obrázku nižšie vidíme v náhľade celý experiment AAPA, v okne UE editora.



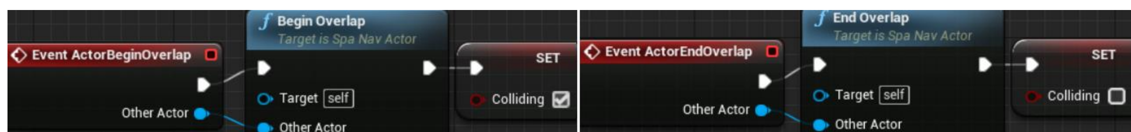
Obrázok 24: Náhľad na experiment v prostredí aplikácie editora

Napravo je vizuálna podoba levelu, kruhovej arény, v ktorej sa experiment odohráva. Vidíme tam tri farebne odlíšené oblasti položené na zelenej kruhovej platforme, ktorá sa počas behu experimentu otáča. Jedná sa o zakázané oblasti, ktorým sa má testovaný subjekt vyhýbať. Zelené objekty pripomínajúce diamanty sú cieľové oblasti na svojich počiatočných pozíciách. Biely objekt na platforme je postava,

ktorú bude pri behu experimentu ovládať testovaný subjekt. Na otáčajúcej sa platforme ďalej vidíme 3 rastliny. Tieto rastliny sa otáčajú s platformou. Ich účelom je pomáhať s orientáciou na platforme. Poslednými viditeľnými objektmi experimentu sú 3 kamenné štruktúry. Tie sú statické a taktiež majú pomáhať s orientáciou. Všetky objekty sú uzavreté v aréne, ktorá vymedzuje priestor pre beh experimentu. Všetky ďalšie objekty, ktoré tu vidíme sú viditeľné iba v editore a pridávajú do experimentu rôzne funkčné aspekty, ako napríklad zvuky, osvetlenie atď. . Kompletný zoznam všetkých objektov na scéne vidíme na zozname vľavo. V nasledujúcich riadkoch si o niektorých z týchto objektov povieme

MyScriptedActor

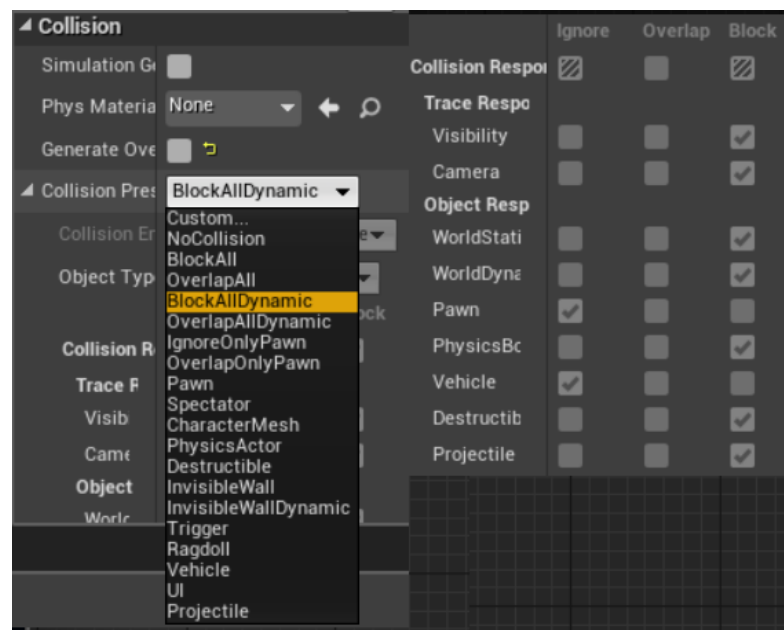
Základom mnohých objektov, ktoré môžeme vidieť v experimente je rodičovský objekt *MyScriptedActor*. Jedná sa o Bluescripts-based objekt, ktorý sám nemá žiadnu vizuálnu reprezentáciu. Len pre úplnosť si uvedieme, že v UE4 existujú dva typy objektov. Blueprints-only a Blueprints objekty, ktoré majú na pozadí C++ triedu. Trieda MyScriptedActor je druhým typom objektu, a na pozadí má C++ triedu s menom *ScriptedActor*. *MyScriptedActor* je kontajner pre vlastnosti spoločné všetkým objektom scény, s ktorými chceme interagovať v prostredí skriptu. Základným prvkom skriptovania experimentu je test kolízií. Aby bolo možné sa objektu dotazovať na kolízie s inými objektami, z prostredia skriptu, zaviedli sme vlastnú premennú spoločného objektu *MyScriptedActor*. V prípade počiatku kolízie sa binárna premenná *Colliding* nastaví na *True*, v prípade signálu znamenajúcemu koniec kolízie, sa premenná nastaví na *False*. Aktívne teda nečakáme na signál enginu, ale v prípade potreby sa dotážeme na aktuálny stav. Tento systém je prirodzenejší programovému modelu, ktorý používame v skriptovacom engine na tvorbu experimentu. Ako v našom prípade nastavovanie vlastnosti objektu ako reakcia na udalosť kolízie vyzerá, môžeme vidieť na obrázku nižšie.



Obrázok 25: Začiatok a koniec kolízie

Táto časť spracovania kolízií však upravuje iba poslednú fázu procesu, a teda konečnú reakciu. Aby sme mohli vôbec o kolíziách uvažovať musíme objektu vždy nastaviť kolíznu schému. Kolízna schéma však priamo súvisí s tvarom objektu, a

teda s jeho vizuálnou stránkou. Na tomto mieste si letmo predstavíme koncept kolíznej schémy, ale konkrétne nastavenia budeme riešiť pre jednotlivé objekty scény samostatne.



Obrázok 26: Možnosti kolíznej schémy v UE4

Každý objekt typu *Mesh*, teda objekt mriežky-tvaru má nastavenie *Collision*, kde z listu vyberáme kolíznu schému. Zoznam všetkých možností, vidíme na obrázku 25 vľavo. Výberom schémy sa nám potom nastaví matica na obrázku 25 vpravo. Vidíme, že je možné podľa typu objektu definovať kedy chceme kolíziu ignorovať, kedy chceme vyvolať udalosť kolízie ale umožniť prechod objektom, a kedy naopak chceme v prípade kolízie úplne blokovať hlbší prienik objektov. Na obrázku vpravo napríklad vidíme nastavenie pre IgnoreOnlyPawn schému. Efektívne toto nastavenie blokuje takmer všetky objekty, ale úplne ignoruje kolízie pre objekty, ktoré môžu reprezentovať hráča. Teda postavu alebo obecne typ objektu vozidlo.

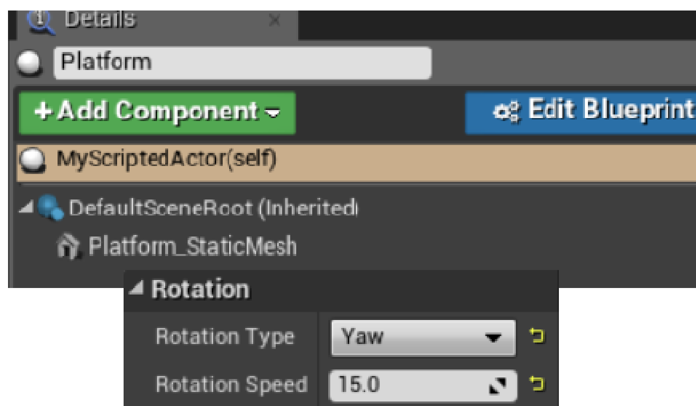
Pre triedu *MyScriptedActor* sme definovali celú samostatnú sekciu vlastností, nazvanú *Scripting*. Sekcia obsahuje napríklad aj vlastnosť *ScriptName*, ktorá definuje meno, pod ktorým objekt registrujeme do skript enginu. Týmto menom sa potom zo skriptu odkazujeme na objekt. Umožňuje nám to z prostredia skriptu komunikovať s konkrétnou inštanciou objektu na scéne a volať funkcie na nej definované.

Základná trieda ďalej definuje dve vlastnosti, ktoré upravujú spôsob, akým sa na začiatku experimentu chová. Jedná sa o vlastnosti popisujúce typ rotácie, ktorú objekt vykonáva a jej rýchlosť. Vlastnosti sme umiestnili do skupiny *Rotation* na objekte *MyScriptedActor*.

Obrázok 27: Rotácia objektov v eperimente

Objekty potom vďaka schéme na obrázku vpravo, ihneď po spustení aplikácie vykonávajú naprogramovaný pohyb.

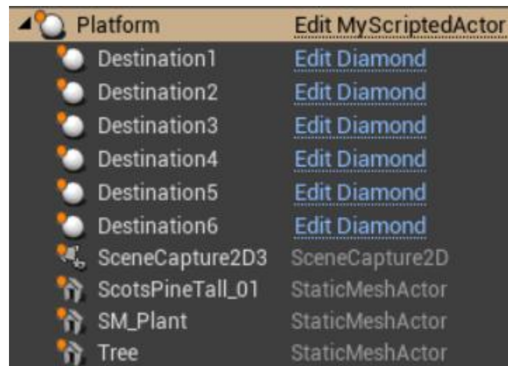
Tieto vlastnosti využíva napríklad objekt rotujúcej platformy, ktorá je ústredným objektom experimentu. Platforma je priamou inštanciou triedy *MyScriptedActor*, ktorej sme pridali vizuálny prvok, typu *Mesh*. Konkrétne sa jedná o prvok *Platform_StaticMesh*. Konečný importovaný prvok potom vyzerá nasledovne:



Obrázok 28: Nastavenie objektu otáčajúcej platformy

Z náhľadu časti detailu platformy vidíme, že základom objektu je *MyScriptedActor*. Ten definuje *DefaultSceeneRoot* vlastnosť, ktorá očakáva objekt, koreň, ktorý bude reprezentovať základný stavebný prvok celej inštancie. V našom prípade je ním spomínaná mriežka rotujúcej platformy. Na obrázku nižšie vidíme rotáciu platformy v smere vodorovnom s pomyselnou podlahou, na ktorej je vybudovaný experiment uhlovou rýchlosťou 15 stupňov za sekundu.

Objekty, ktoré sme potrebovali zviazať s rotáciou platformy sme v okne *WorldOutliner* zaradili pod objekt platformy, a tým pádom tieto objekty dedia rotačnú transformáciu platformy automaticky.



Obrázok 29: Hierarchické usporiadanie objektov v experimente

Všimnime si, že v zozname chýbajú objekty zakázaných oblastí. Je to zámer, pretože pohyb zakázaných oblastí nie je spojený s rotujúcou platformou počas celého experimentu. Pripojenie a odpojenie od pohybu platformy je riadené skriptom. Na začiatku experimentu oblasti začínajú v odpojenom režime a až neskôr počas experimentu sú ku platforme pripojené dynamicky.

Zakázané oblasti definujú vlastnú triedu *AreaActor*. Tá dedí od triedy *MyScritptedActor*. To nám napovedá, že sa jedná o objekt, na ktorý sa budeme chcieť odkazovať zo skriptovacieho enginu. *AreaActor* sa od *MyScriptedActor* triedy líši v tom, že navyše definuje vizuálnu podobu zakázanej oblasti. Konkrétne v okne vizuálnych úprav triedy, okno sa volá *Viewport*, si všimneme prítomnosť objektu typu *static mesh* nazvaného *60DegreesPlatform*. Je to jednoduchý výsek veľkosti 60 stupňov, z platformy, ktorú sme popisovali pred nejakým časom. V scéne sa bude nachádzať niekoľko týchto oblastí, a je treba ich od seba odlíšiť. Po vzore pôvodného SpaNav 1.0 používame farebné odlíšenie pomocou textúry. Z tohoto dôvodu v deklarácii obecnej triedy pre zakázanú oblasť textúru nepoužívame, a prenechávame toto rozhodnutie na jednotlivé inštancie v experimente. Dôležitým aspektom pre triedy zakázaných oblastí je kolízna schéma. Zakázané oblasti musia reagovať na kolízie s postavou testovaného subjektu, pretože potrebujeme počítať dobu, počas ktorej je subjekt s oblasťou v kolízii. Táto oblasť však musí byť pre postavu neblokujúca, pretože postava musí mať možnosť do oblasti vstúpiť, pohybovať sa v nej úplne voľne a po ľubovoľnom čase z oblasti opäť vystúpiť. Tomuto správaniu najlepšie zodpovedá predpripravená kolízna schéma *OverlapOnlyPawn* s nasledujúcou definíciou:

	Ignore	Overlap	Block
Collision Responses	▨	▨	▨
Trace Responses			
Visibility			✓
Camera	✓		
Object Responses			
WorldStatic			✓
WorldDynamic			✓
Pawn		✓	
PhysicsBody			✓
Vehicle		✓	
Destructible			✓
Projectile			✓

Obrázok 30: Kolízna matica pre OverlapOnlyPawn

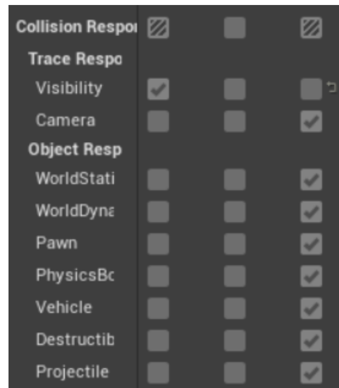
Vy výsledku teda oblasti blokujú pohyb všetkých objektov v scéne, okrem objektov reprezentujúcich hráča, pre kolízie ktorého však vyvolávajú udalosti. Ak by sme mali ísť do detailov, na chovaní oblastí k ostatným objektom scény nám vlastne nezáleží. Naša scéna je pomerne jednoduchá, a neobsahuje okrem hráča žiadne iné objekty.

Cieľové oblasti sú definované veľmi podobne ako zakázané oblasti. Líšia sa iba svojou vizuálnou reprezentáciou. Cieľové oblasti definujú vlastný objekt s názvom *Diamond*, podľa podobnosti s diamantom. Miesto kruhovej výseče, obsahujú importovaný static mesh. Pretože medzi jednotlivými inštanciami cieľových oblastí nepotrebujeme rozlišovať, definícia triedy obsahuje taktiež textúru objektu Diamond. Z definície experimentu vyplýva, že v každom momente musí byť na ploche 6 kópií objektu. Preto máme aj my v scéne umiestnených 6 inštancií triedy Diamond.

V duchu postoja, že Blueprints obsahujú iba primitívnu logiku s ohľadom na logiku experimentu, triedy zakázaných oblastí a cieľových oblastí považujeme predošlým textom z užívateľského pohľadu pre Blueprints popísané.

Trieda *MyScriptedCharacter* reprezentujúca postavu testovaného subjektu má predka v C++ triede *ScriptedCharacter*, o ktorej sme si už niečo povedali v programátorskej dokumentácii. Len stručne pripomenieme, že ScriptedCharacter umožňuje do skriptu prenášať údaje o polohe a rotácii postavy. Definícia a umiestnenie *MyScriptedCharacter* do scény má dva dôvody. Prvým je, že pomocou tejto triedy definujeme vizuálnu podobu hráča, ktorá vznikla spojením dvoch statických mriežok reprezentujúcich guľu, virtuálne šípky, aby sme poznali smer, ktorým testovaný subjekt pozerá a kolíznej siete. Tým že objekt je do projektu pridaný ako Blueprints typu *Charakter*, implicitne sa do definičného stromu pridal komponent reprezentujúci možnosti pohybu postavy. Obohacovanie možností Blueprints objektov pomocou komponent je bežná prax vo vývoji pomocou Blueprints. My sa s

ňou však v našom projekte stretáme minimálne. Je to dané charakterom projektu, kde sa snažíme podstatnú časť logiky, aspoň tú dôležitú z hľadiska definície experimentu, definovať v skriptovacom engine. Trieda *Character* je hierarchicky nástupca triedy *Pawn*. *Character* dedí kolíznu schému svojho predchodcu, definovanú nasledujúcim spôsobom:



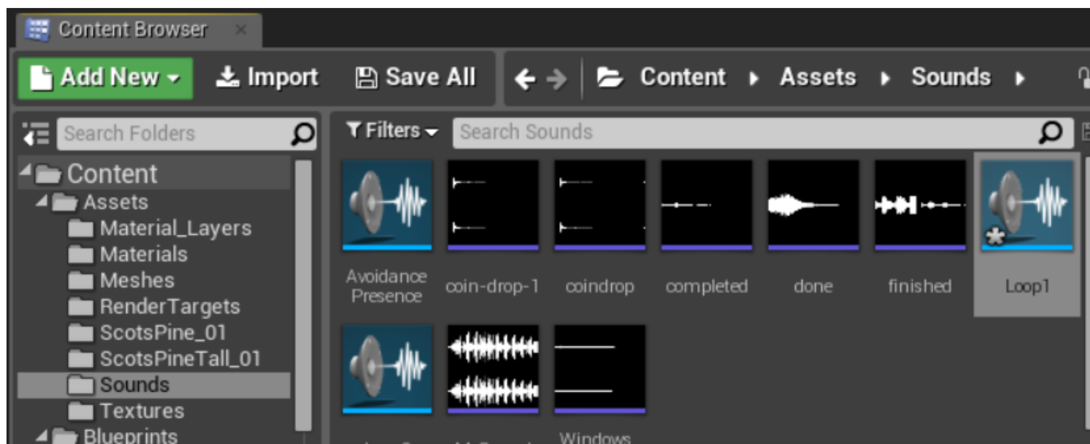
Obrázok 31: Kolízna matica objektu typu Pawn

V tomto momente, vyvstáva otázka, ako sa objekty zachovajú, v kolízii, ak majú rôzne definované kolízne schémy. UE4 engine to rieši tým, že použije kombináciu dvoch použitých kolíznych schém a pre každý aspekt vyberie menej obmedzujúce nastavenie. Obmedzenie rastie v smere *Ignore – Overlap – Block.*

Komponent ovládajúci zvuky umožňuje vyvolanie dvoch typov akcií. Jedna akcia ovláda spustenie jednorazovej zvukovej stopy. Druhým typom je pár funkcií, ktoré ovládajú spustenie a zastavanie zvukovej slučky.

Pustenie jednorazového zvuku je akcia typu “Fire and Forget”. Zvuk sa v prípade potreby spustí a je prehrávaný kým neskončí. Pre účely experimentu umožňujeme použitie niekoľkých jednorazových zvukov. Konkrétne sme ich pre možnosť výberu pripravili 7. Skript pre pustenie zvuku je naviazaný na udalosť *EventPlaySound*, nami definovanú v C++ komponente, ktorá navyše ako parameter dostáva index zvukovej stopy, ktorú chceme pustiť. Udalosť *EventPlaySound* je na strane zdrojového kódu volaná z funkcie kontrolovanej behom skriptu v skriptovacom engine aplikácie. Detaily procesu sme si ukázali v programátorskej dokumentácii. Dôležitý pre nás je momentálne fakt, že z prostredia skriptu rozhodujeme o tom, kedy a ktorú zvukovú stopu zavoláme. Jednotlivé zvuky sú uložené v poli zvukov, a pomocou vstupného parametru udalosti, špecifikujeme index v poli, na ktorom mieste sídli požadovaný zvuk. Pole zvukov je premennou patriacou objektu *SoundActor*. Jeho iniciálne hodnoty smerujú do knižnice zvukov a označujú

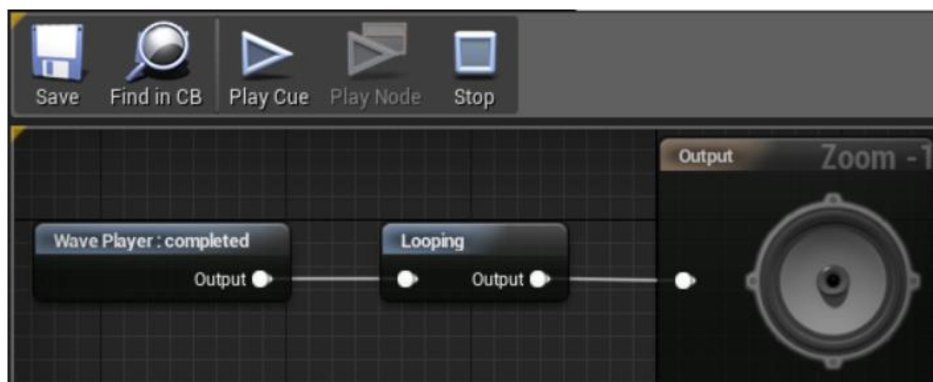
jednotlivé zvukové stopy, ktoré sme sa rozhodli použiť. Keď používame výraz knižnica zvukov, máme na mysli adresár *Sounds* v adresárovej štruktúre projektu.



Obrázok 32: Knižnica zvukov v adresárovej štruktúre

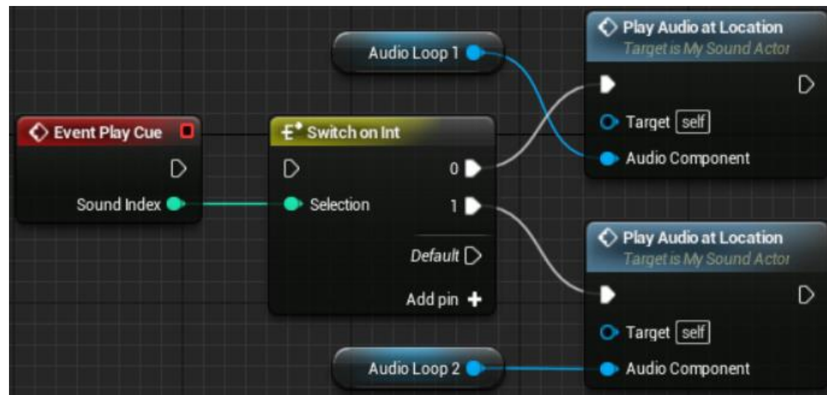
Tento systém usporiadania je arbitrárny, a slúži pre lepšiu orientáciu a ľahšiu správu zdrojov.

Stopa zvukovej slučky je samostatný typ objektu v knižnici zvukov (na obrázku Loop1 a Loop2). My sme si pre naše účely vytvorili viaceré slučky, z ktorých sme potom vybrali dve, ktoré sme sprístupnili zvukovej komponente. Zvuková slučka sa vytvára pomocou editora zvukov dostupného v EU4 editore. Základom slučky je jednorazový zvuk, ktorý v schéme reprezentuje uzol *Wave Player*. Výstup uzlu je možné nasmerovať do uzlov rôznych zvukových efektov. UE4 ponúka rôzne možnosti práce so zvukmi, okrem iného napríklad uzly pre miešanie viacerých zvukových stôp, oneskorenie prehrávania, dopplerov efekt na zvukovej stope atď. . Pre naše účely sme vybrali komponentu *Looping*, ktorá zariadi opakovanie zvukovej stopy.



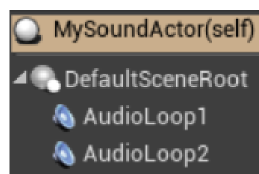
Obrázok 33: Definícia zvukovej slučky

Pre pustenie a zastavenie slučky na objekte *SoundActor* používame dvojicu skriptov. Prvý z nich je skript pre spustenie zvukovej slučky.



Obrázok 34: Spustenie zvukovej slučky

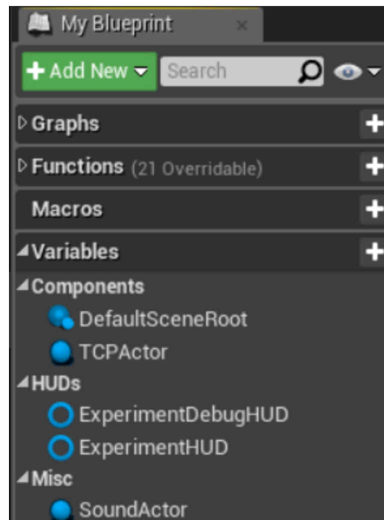
Skript je viazaný k udalosti *EventPlayCue*, tak isto ako v prípade jednorazových zvukov, nami definovanej v C++ komponente, ktorá tiež ako parameter dostáva index zvukovej stopy, ktorú chceme pustiť. Tiež je možné ju vyvolať z prostredia skriptovacieho enginu. Komponenty AudioLoop1 a AudiLoop2, boli pridané do grafu komponenty SoundActor z našej vlastnej knižnice zvukov.



Obrázok 35: Zvukové slučky ako súčasť definície objektu SoundActor

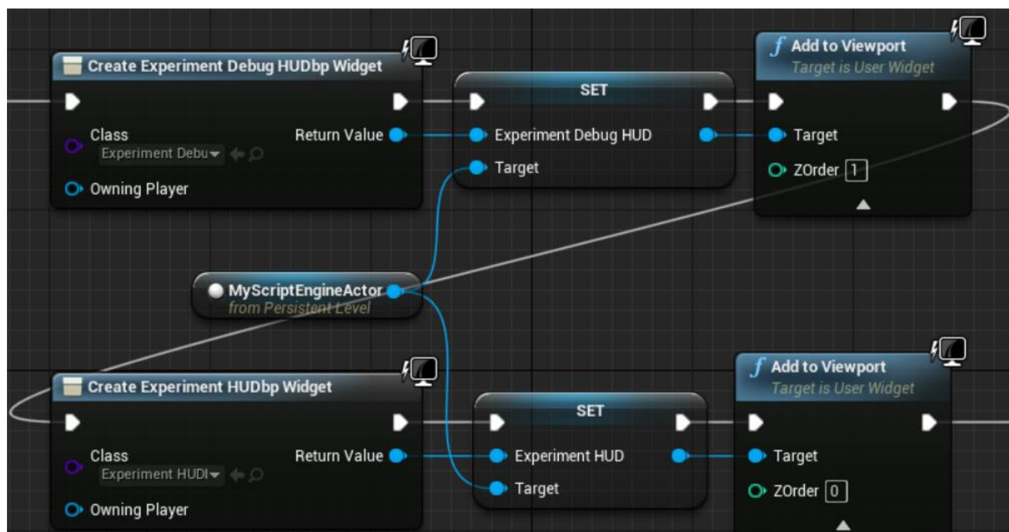
Komponent pre zastavenie prehrávania zvukov funguje analogicky k popísanej komponente. Rozdiel je v tom, že skript je naviazaný na inú udalosť, ktorá je však taktiež spojená so skriptovacím enginom, a vyžaduje tak isto argument pre upresnenie, ktorá zvuková stopa má byť zastavená. Keď už máme k dispozícii inštanciu zvukovej komponenty, ktorú chceme zastaviť, zavoláme na nej jednoducho funkciu *Stop*.

Možnosť skriptovania experimentu nám v mape reprezentuje komponent *MyScriptEngineActor*. Tento komponent je postavený na základe C++ triedy *ScriptEngineActor,* o ktorej sme si písali. Účelom Blueprints komponenty *MyScriptEngineActor*, je poskytnúť kódu na pozadí rámec pre prístup k objektom, vlastnostiam a životnému cyklu mapy, v ktorej sa nachádza. *MyScriptEngineActor* potom obohacuje pôvodnú triedu o niekoľko polí, reprezentujúcich rôzne objekty, s ktorými chce skriptovací engine pracovať. Kód na pozadí potrebuje poznať objekty vystavené do skript enginu, a popísaným spôsobom mu môžeme pripraviť dáta pre jeho prácu.



Obrázok 36: Objekt slúži ako kontajner pre objekty exportované do skript enginu

Na viacerých miestach v Blueprints potom možno vidieť podobne vyzerajúce schémy ako na obrázku nižšie.



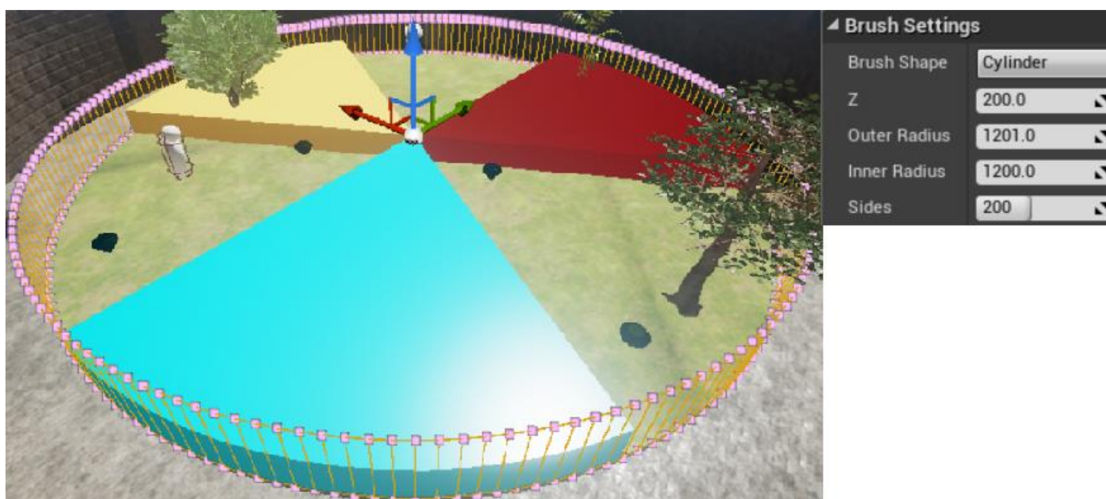
Obrázok 37: Priradenie inštancií HUD objektu reprezentujúcemu skript engine

Referencia na *MyScriptedActor* je typicky získaná pretiahnutím komponenty inštancie, buď zo zoznamu objektov z okna *World Outliner*, alebo priamo z okna editora mapy. Keďže sú pripravené polia verejné, Blueprints systém pre nich automaticky vygeneruje metódy get* a set*, so vstupným parametrom podľa typu poľa.

Komponent *TCPActor* je implementačne podobný prípad ako *MyScriptedActor* popísaný vyššie. Jeho existencia je podmienená výhradne potrebou prístupu k vlastnostiam a schopnostiam objektov, podliehajúcich životnému cyklu experimentu, ako je navrhnutý v editore. Tomuto typu Blueprints objektu sa hovorí Data Blueprints. Vyznačuje sa tým, že neobsahuje a štandardne ani nemôže definovať

vizuálnu reprezentáciu. Je možné to zmeniť, avšak pre naše potreby to nebude treba. Vďaka Blueprints komponente *TCPActor* potom môže jeho C++ reprezentácia pracovať, ako sme si popísali v programátorskej dokumentácii.

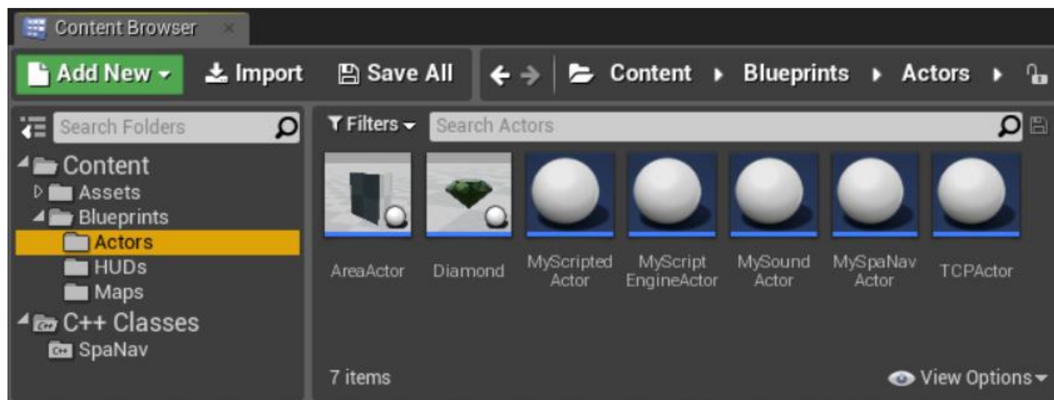
Na obrázku nižšie možno vidieť ako sme sa vysporiadali s požiadavkou na obmedzenie pohybu testovaného subjektu na plochu otáčajúcej sa platformy.



Obrázok 38: Neviditeľný plot okolo arény brániaci opusteniu platformy

Objekt na obrázku vyznačený ružovou farbou je takzvaný *blocking volume*. V konečnej aplikácii je neviditeľný ale má blokujúce vlastnosti na postavu, ktorá dedí z komponenty Pawn, ktorou ako už vieme je aj náš komponent reprezentujúci testovaný subjekt. Jedná sa o zdegenerovaný valec, ktorého vonkajší polomer je o jednu jednotku väčší, než priemer vnútorného valca, ktorý určuje akési vykrojenie z pôvodného objektu. Metódou viacerých pokusov a omylov sme definovali premennú Sides na hodnotu 200. Táto hodnota označuje z koľkých rovných plôch sa hotový valec skladá. Nízke hodnoty definovali hraničné uhly, prechod cez ktoré nebol vždy hladký a postava hráča sa medzi jednotlivými stenami zasekávala. Príliš vysoký počet stien je zas výpočetne náročný. Zvolili sme preto nižšiu hodnotu, ktorá už však nespôsobovala citeľné drhnutie pri prechode medzi blokmi.

Všetky objekty dostupné v experiment sú umiestené a organizované v adresárovej štruktúre. Tá je pre nový projekt automaticky dostupná a predpripravená. Dostupná je v okne *Content Browser* editora. My sme si našu adresárovú štruktúru pozmenili, aby vyhovovala našim potrebám a organizácii. Náš Blueprints projekt je relatívne jednoduchý, takže tomu zodpovedá aj štruktúra adresárov.



Obrázok 39: Adresárová štruktúra experimentu

Najvyššie delenie rozdelí objekty na tie, na ktoré pristupujeme priamo z Blueprints (Content) a na tie, ktoré s Bluperints súvisia, ale dominuje im C++ implementácia (C++ Classes). Zaoberať sa budeme prvou skupinou, keďže C++ implementácia je záležitosťou programátorskej dokumentácie. Blueprints ďalej delíme na Assets, čo sú v našom prípade statické, funkčne jednoduché objekty s dôrazom na vizuálnu stránku. Z veľkej časti sú to zdroje, ktoré sme nevyvíjali sami. Vizuálna podoba pre nás nie je myšlienkovo zaujímavá, a nemáme na ňu kapacitu. Skupina objektov, ktorým sme sa venovali primárne je v adresári Blueprints. Jedná sa o funkčne bohaté objekty vlastnej výroby troch typov. Sú nimi HUD, mapy a objekty vystupujúce v experimente ako aktéri. Popisu jednotlivých objektov z týchto troch skupín sa venovali predošlé kapitoly.

Skriptovanie

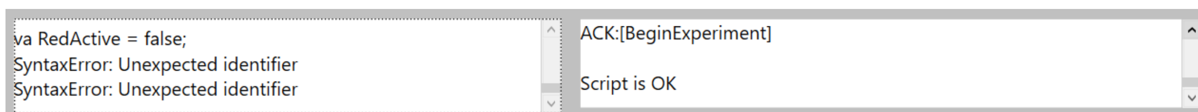
Pre účely skriptovania experimentu používame nami rozšírenú normu pre jazyk Javascript. Základ jazyka ECMAScript sme rozšírili o možnosti registrácie udalostí a ich handlerov a možností časovania. Pre implementáciu správy udalostí sme sa inšpirovali v spôsobe, akým daný problém rieši technológia Node.js. V nasledujúcej kapitole si na príklade implementovaného experimentu AAPA ukážeme, aké možnosti má užívateľ pri vývoji s použitím systému SpaNav 2.0.

Experiment AAPA je implementovaný rozšírenou verziou jazyka Javascript. Pri popise sa budeme venovať vlastnej nadstavbe jazyka. Dokumentácia pôvodného Javascriptu je jednoducho dohľadateľná z iných zdrojov. Implementácia experimentu sa nachádza v zdrojových súboroch v adresári aapa2.0, ktorý je súčasťou dodaného systému SpaNav 2.0. V adresári vidíme jeden konfiguračný súbor a niekoľko zdrojových súborov Javascriptu. O konfiguračnom súbore, sme si už písali v úvode

kapitoly. Spomenieme si ho však ešte raz v časti venovanej registrácii udalostí. Vstupným bodom do skriptu aplikácie je súbor aapa2.0.js a preto začneme práve v tomto súbore.

Direktíva Include

V úplnom úvode súboru môžeme vidieť použitie direktívy Include. Direktíva Include má notáciu ***#Include(absolútna/relatívna cesta k súboru)***. Direktíva funguje veľmi jednoducho a to tak, že reťazec s deklaráciou direktívy nahradí obsahom súboru, ku ktorému vedie cesta. Kód je nakoniec interpretovaný ako jeden súvislý blok. Pamätať na toto pravidlo sa môže ukázať veľmi užitočné, najmä v prípadoch keď sa kód nespráva podľa očakávania. Kód je už ako jeden súvislý blok kompilovaný. V prípade že kompilácia nájde v kóde chybu, táto chyba je zobrazená v okne klienta. Kompiláciu, tak ako aj obsah zobrazenej chyby kontroluje Google V8 skriptovací engine



Obrázok 40: zázam o chybe a úspešná kompilácia

Registrácia udalostí

Pre potreby registrácie udalostí používame funkciu *RegisterEvent*. Funkcia *RegisterEvent* dostane na vstup názov udalosti, a funkciu, ktorú má vykonať v reakcii na výskyt udalosti. V nami implementovanom experimente používame zápis s anonymnými funkciami. Notáciu anonymných funkcií je možné si dohľadať v externých zdrojoch. Naša implementácia sa nevyznačuje žiadnymi zvláštnosťami. Meno udalosti je citlivé na veľkosť písmen a užívateľ môže udalosť registrovať na strane CONTROLLER aplikácie ako reakciu na stlačenie klávesy.

```
1. RegisterEvent("ToggleDestinations", function(){
2. Log("ToggleDestinationEvent");
3. ToggleActiveAreaVisibility();
4. });
```

Kód 11: Obsluha udalosti "ToggleDestinations"

Druhý spôsob je registrácia udalosti, ktorá vzniká v prostredí SLAVE aplikácie. Momentálne máme 3 udalosti, ku ktorým je možno sa zaregistrovať. Sú to udalosti pre objekty typu *Actor*:

- OverlapBegin (začiatok kolízie)
- OverlapEnd (koniec kolízie – momentálne sa nepoužíva)
- Hit (kolízia typu “hit” – momentálne sa nepoužíva)

Každý Actor generuje vlastnú variantu udalostí a to tak, že kombinuje spoločný názov udalosti s menom objektu, na ktorom vznikli notáciou: ***Meno.NázovUdalosti***. Experiment používa *OverlapBegin* pre kontrolu kolízie s cieľovými oblasťami. Spôsobom, ktorý môžeme vidieť na nasledujúcich riadkoch:

```
1. RegisterEvent("Destination1.OverlapBegin", function(){
2. OnDestinationEnter(Destination1);
3. });
4. RegisterEvent("Destination2.OverlapBegin", function(){
5. OnDestinationEnter(Destination2);
6. });
```

Kód 12: Obsluha udalosti *OverlapBegin*

Registrácia načasovaných úloh

Naša rozšírená verzia Javascriptu umožňuje registrovať spúšťanie skriptovaného kódu aj v časových intervaloch. To buď jednorazovo, keď sa funkcia pustí po zadanom čase raz, alebo opakovane, vždy po uplynutí časového intervalu od doby posledného spustenia. Pre registráciu časových udalostí používame funkciu *Timer*. *Timer* funkcia má tri parametre. Prvým je časový interval v sekundách. Pre prípady keď požadujeme spúšťanie funkcie častejšie, než v intervaloch celých sekúnd, je možno použiť čísla s desatinnou časťou. V experimente efektívne využívame intervaly až na úrovni 0.005 sekundy, teda piatich tisícin sekundy. Celý kód beží skompilovaný do C++ v samostatnom vlákne, takže aj táto požiadavka je pre jednoduché úlohy reálna. Druhým parametrom je existujúca funkcia alebo anonymná funkcia. V existujúcom experimente používame oba prístupy, v závislosti na požadovanej čitateľnosti kódu.

Príklad registrácie anonymnej funkcie môžeme vidieť nižšie:

```
1. Timer(1, function(){
2. --TimeToGo;

3. LogT("Timer: " + TimeToGo);
4. ExperimentHUD.SetTime((TimeToGo/60 | 0) + ":" + TimeToGo%60);
   .
   .
   .
5. });
```

Kód 13: Načasovaná anonymná funkcia s intervalom 1 sekundy

Všimnime si, že v definícii opomíname tretí parameter. Tretím parametrom je boolean hodnota, ktorá v prípade hodnoty *True*, označí registráciu ako opakovateľnú. V prípade hodnoty *False*, sa registrovaná funkcia pustí iba raz. Tento parameter má implicitnú hodnotu nastavenú ako *True*, a je nepovinný.

Logovanie

Logovanie umožňujeme v troch variantoch:

- Logovanie do primárneho súboru (súbor bez prípony), pomocou funkcie *Log*. Vstup vložený do parametru funkcie *Log* je Google V8 enginom preložený do jeho reťazcovej verzie a spolu s aktuálnou časovou známkou zapísaný do súboru. Správa v súbore logu má navyše ešte pečiatku v tvare „[JavascriptEngine]“, ktorá informuje o zdroji zadanej správy. Táto pečiatka má historický význam, keď do logu zapisovala aj SLAVE aplikácia. Toto logovanie SLAVE aplikácie je možné v prípade potreby aktivovať a jej správy budú používať vlastnú pečiatku.
- Logovanie do súboru časových udalostí. Zo špecifikácie vyplynula potreba oddeliť typ logovaných správ do samostatného súboru. Jednalo sa o opakované logovacie správy o aktuálnom stave objektov v experimente naviazané na časové intervaly. Pre tieto potreby máme logovací súbor s príponou _T, a zápis do tohoto logu vykonávame funkciou *LogT*.
- Pre prípady ladenia skriptu máme pripravený systém logovania na obrazovku aplikácie SLAVE. Správa zapísaná metódou *AddDebug* na objekte *ExperimentDebugHUD* vypíše na 10 sekúnd správu priamo v okne aplikácie.

Získanie údajov o polohe a rotácii

Každý *Actor* a postava subjektu tiež, ponúka metódy pre zistenie polohy a natočenia. Tieto údaje sú kritické pre vyhodnotenie experimentu. Spracovaniu výstupu sa venuje samostatná kapitola v závere práce. Momentálne nám stačí vedieť, že v prostredí skriptu potrebujeme mať možnosť tieto informácie získať, a prípadne aj zmeniť (druhý spomenutý prípad sa používa pre rozmiestnenie cieľových oblastí v aréne). Metódy, ktoré čítajú spomínané dáta sú:

- *Objekt.GetLocation();*
- *Objekt.GetRotation();*

a analogicky:

- *Objekt.SetLocation(X,Y,Z)*
- *Objekt.SetRotation(Pitch, Yaw, Roll)* – momentálne sa nepoužíva

GetLocation funkcia vracia objekt s troma vlastnosťami, pomenovanými X, Y, Z a je možno k nim priamo pristupovať. Navrátený objekt je takzvaný *Immutable*, čo znamená že zmenou hodnôt, sa nezmenia hodnoty na originálnom objekte. Pre zmenu polohy originálneho objektu je treba použiť metódu *setLocation*.

GetRotation vracia objekt s troma vlastnosťami pomenovanými Pitch, Yaw, Roll, ktoré reprezentujú natočenie v jednotkách stupňov v troch na seba kolmých osách X, Y, Z. Inak pre tento objekt platia rovnaké pravidlá ako pre objekt *Location* z predošlého odstavca (*Immutable*)

Zvuky

Pre beh experimentu sú dôležité zvukové znamenia prezentované testovanému subjektu. Logika toho kedy sa tieto znamenia objavia ovláda skript kód pomocou metód objektu *Sounds*. Dva typy zvukových efektov sme už v našej práci popisovali v inom kontexte. Pripomenieme len, že sa jedná o jednorazový zvuk, ktorý po spustení prebehne raz a zvuková slučka (*cue*) ktorá musí byť po spustení explicitne zastavená. Pokiaľ sa tak nestane, slučka sa neustále opakuje. Zvuky obsluhujú 3 metódy a to

StartCue(index) a *StopCue(index),* pre spustenie a zastavenie zvukovej slučky identifikovanej indexom na vstupe, a funkcia *PlaySound(index),* ktorá púšťa jednorazový zvuk. Pole zvukov používaných pre slučky a pole zvukov používaných pre jednorazové zvuky sú dve rôzne entity a obsahujú iné zvuky.

Matematické funkcie

Zo špecifikácie experimentu vyplynula potreba niekoľkých základných matematických funkcií. Do skriptovacieho enginu sme previedli nasledujúce:

- ***Sqrt(Number)*** – spočíta základ druhej mocniny pre vstupný parameter
- ***Srand(Number)*** – inicializuje pseudogenerátor náhodných čísel
- ***Rand()*** – vráti pseudonáhodnú hodnotu v rozmedzí 0-1
- ***Abs(Number)*** – vráti absolútnu hodnotu spočítanú zo vstupného parametru

Tieto funkcie sa momentálne v experimente používajú na spočítanie novej pseudonáhodnej polohy cieľovej oblasti. Oblasť sa presúva na novú polohu vždy keď sa testovací subjekt dostane do kolízie s oblasťou na jej pôvodnej polohe.

Na predchádzajúcich riadkoch sme sa pokúsili popísať princípy základných konceptov použitých pri programovaní experimentu pomocou skriptovacieho jazyka. Vyčerpávajúci zoznam všetkých dostupných metód a funkcií je možné nájsť aj s ukážkou použitia na konci práce medzi prílohami.

SpaNav 1.0 vs. SpaNav 2.0

Úsilie vynaložené na prácu, ktorej sa venoval predošlý text, bolo v prvom rade zlepšiť rýchlosť komunikácie jednotlivých častí systému SpaNav, pre presnejšie zaznamenávanie dát. Teoreticky sme si popísali, ako sme zmenili architektúru a technológie a ako sme skrze mnoho dizajnérskych rozhodnutí a programátorských techník dospeli ku SpaNav 2.0. V tejto kapitole sa pozrieme výsledky testov, ktoré majú za účel zmerať a porovnať pôvodný SpaNav 1.0 s novou verziou.

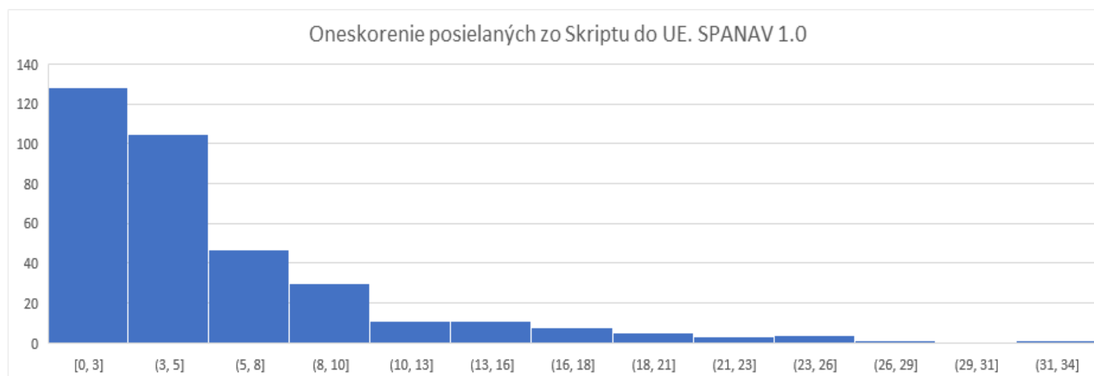
Pred tým, než popíšeme proces testovania s výsledkami, ktoré sme dosiahli, je dôležité uviesť nasledujúce: Testy boli spúšťané tak, že jednotlivé aplikácie, ktoré tvoria systém (CONTROLLER + SLAVE) boli spustené vždy na jednom počítači. Nevzdali sme sa zámeru používať SpaNav na viacerých počítačoch ale ako uvidíme pre testovanie pomernej rýchlosti odozvy medzi jednotlivými časťami systému je toto testovanie dostatočné. Dôvody pre takéto testovanie sú odlišné pre jednotlivé verzie a konkrétne ich uvedieme v kapitolách venujúcich sa jednotlivým verziám.

Testovanie SpaNav 1.0

Už SpaNav v pôvodnej verzii bol zamýšľaný ako systém pozostávajúci zo samostatných komponent, s potenciálom spúšťania jednotlivých častí na samostatných počítačoch. Z technických príčin, popísaných v úvodnej časti práce, súvisiacich s problémami komunikácie komponent cez sieťovú kartu, k realizácii tohoto zámeru však nikdy nedošlo. Dáta, ktoré máme k dispozícii, sú preto výhradne získané behom celého systému na jednom počítači. Je treba si však uvedomiť, že aj napriek tomu, že komponenty bežia na jednom počítači, komunikujú TCP protokolom skrze sieťovú kartu (localhost). Dáta vstupujúce do testu sme získali analýzou a spracovaní dvoch typov logovacích súborov, kde jeden vznikol logovaním UE (ďalej UE log) a druhý vznikol logovaním zo skriptu (ďalej skript log), čo v prostredí pôvodného SpaNav znamená že vnikol v JAVA aplikácii pôvodného CONTROLLERu. Tieto dva logy oddeľuje TCP komunikácia. Použitý experiment bol SleepForest. Sledovali sme dva typy komunikácie, jednak posielanie správ v smere zo skriptu do UE a potom v opačnom smere z UE do skriptu. Analýzou obsahu správ sme identifikovali dvojice záznamov, patriacich k jednej

správe (odoslanie a prijatie) a porovnávali sme časové pečiatky správ v logovacích súboroch, ktoré sú uvedené s presnosťou na milisekundy.

V smere zo skriptu do UE odchádza pomerne veľké množstvo správ. Konkrétne sme ich pre analýzu mali 354. Namerané údaje je možno vidieť v nasledujúcej tabuľke



Graf 2: Oneskorenie správ posielaných zo skriptu do UE

Osa x označuje intervaly v ms, v rámci ktorých došlo ku spracovaniu správ. Osa y počet správ, ktorým prislúcha daný interval spracovania.

Graf ukazuje, že väčšina správ v smere zo Skriptu do UE bola spracovaná v intervale kratšom než 5 milisekúnd. To je technicky vzaté, aj z pohľadu experimentu akceptovateľná doba. Stále však vidíme podstatnú časť správ prichádzať do komponenty UE s meškaním viac než 5 či 10 milisekúnd, a vidíme dokonca, že nemôžeme zaručiť ani to, že správa bude spracovaná v čase kratšom než 30 milisekúnd.

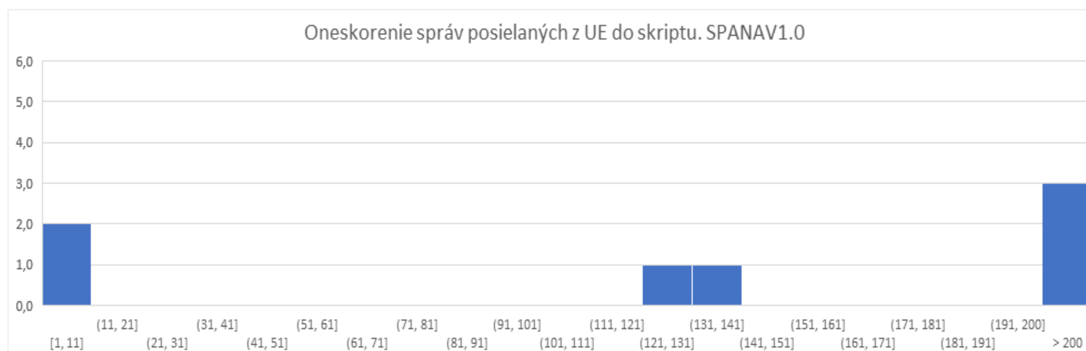
V smere z UE do skriptu máme údajov podstatne menej. V logoch sme identifikovali aktivitu na strane SLAVE aplikácie, pri ktorej došlo ku stlačeniu klávesy „K". Oznam tejto udalosti posielame do skriptu, kde sa udalosť zaloguje. V tomto prípade je treba dať pozor na interpretáciu dát pretože v logoch sa objavuje údaj o čase uplynulom od začiatku experimentu s presnosťou na milisekundy. Začiatok experimentu ale zaznamenávame iba s presnosťou na sekundy. Pomôže nám ale logovací súbor, ktorý obsahuje nasledujúce 4 riadky.

```
1. 190.523    636
2. JavaTime:18:22:47.514; Text.modify(); handle:4; text:;
3. JavaTime:18:22:47.515; Text.modify(); handle:9; text:F-F2;
4. 190.523    636
```

Výstup 4: Záznam správ prijatých z UE

Jedná sa o zapísaný presný čas uväznený v rámci dvoch záznamov jedného framu(s číslom 636). Z toho môžeme pomerne presne určiť že začiatok experimentu je buď

18:22:47.514 alebo 18:22:47.515 bez 190.523 sekundy. Týmto začiatok experimentu stanovíme s presnosťou na jednu milisekundu. S ohľadom na to, že za úspech považujeme zlepšenie v ráde aspoň desiatok tisícin sekundy, s chybou jednej milisekundy sme schopní ďalej pracovať. Výsledky merania sú potom nasledujúce.

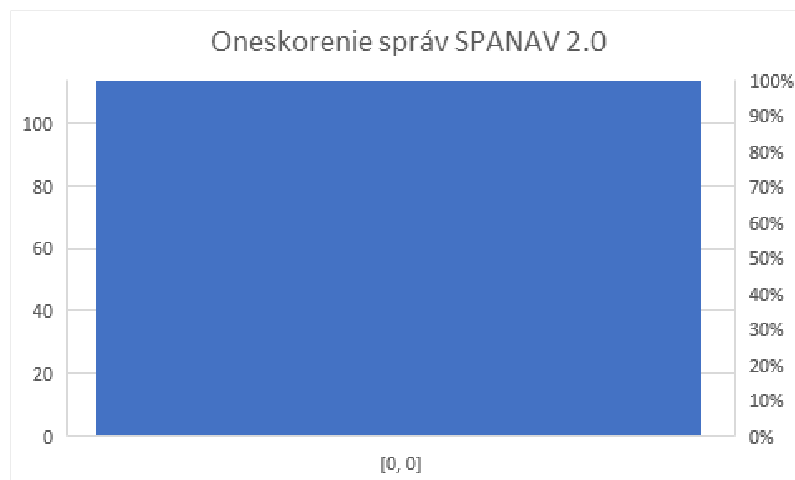


Graf 3: Oneskorenie správ v smere z UE do skriptu

Na začiatku vidíme pár správ spracovaných v intervale 11 milisekúnd. Ďalej na ose sú správy, ktoré boli spracované s oneskorením viac než 120 milisekúnd, a niekoľko správ bolo dokonca spracovaných s oneskorením väčším než 200 milisekúnd.

Testovanie SpaNav 2.0

Pre testovanie SpaNav sme použili funkciu *GetReadableTime* vystavenú do skriptovacieho enginu. *GetReadableTime* má implementáciu v C++, ktorá do Skriptovacieho enginu vráti formátovaný aktuálny čas vygenerovaný v aplikácii UE s presnosťou na milisekundy. Tento čas potom z prostredia Skriptu pomocou funkcie Log pošleme späť do C++(UE) prostredia, kde sa pre zápis vygeneruje nová časová známka s aktuálnym časom. Tá sa generuje pre každý logovaný záznam a mali sme možnosť ju vidieť v predošlých kapitolách práce. Na generovanie časovej známky sa v oboch prípadoch používa rovnaká funkcia UE. Výsledný formát je totožný. Túto akciu vykonáme každú sekundu. Následne v logu vyhľadáme tieto správy a porovnáme čas z obsahu správy(vygenerovaný na začiatku procesu) s časom zalogovania správy (vygenerovaný po ceste do Skript enginu a späť). Očakávame, že sa tieto dva časové údaje budú vo významnom počte odchyľovať čo najmenej. Výstup tohto testu potom porovnáme s predošlými výsledkami a vyhodnotíme v poslednej časti práce, v jej závere. Výsledky testu SpaNav 2.0:



Graf 4: Oneskorenie správ v smere z UE do skriptu a späť

Pre všetkých testovaných 114 prípadov, sú časové známky začiatku a konca komunikácie rovnaké. Vo všetkých prípadoch došlo ku spracovaniu správy po absolvovaní cesty z UE enginu do skript enginu a späť do UE enginu v čase, ktorý nepresiahol jednu milisekundu.

Záver

Hlavný zámer pre vznik tejto práce bolo, preniesť úspech pôvodného systému SpaNav do ďalšej životnej etapy. Pôvodný systém vznikol pred 9 rokmi a v mierne pozmenenej forme funguje a používa sa až doteraz. Postupne sa však stále častejšie naráža na mantinely aplikácie, ktorá bohužiaľ už s ohľadom na technologický pokrok a nároky užívateľov, nedokáže viac držať krok. Ďalšie investície do úprav systému sa prestávajú oplácať, pretože problematické miesta aplikácie sú zakorenené v samotnom jadre použitých technológií.

Miesto ďalších úprav sa tým užívateľov zhodol na potrebe nového projektu SpaNav 2.0, ktorý sa pokúsi vyriešiť problematické body pôvodnej aplikácie. SpaNav 2.0 sa stal hlavným predmetom tejto práce. Cenné poznatky a požiadavky na systém získané používaním jeho predchodcu boli zohľadnené od samého začiatku procesu výroby software. Zmenila sa architektúra, dizajn aplikácie, a nová verzia využíva najnovšie dostupné technológie a postupy k tomu, aby do čo najväčšej miery naplnila očakávania kladené na nový systém.

Súčasne so systémom SpaNav 2.0, vznikol vzorový experiment, založený na existujúcom experimente AAPA pre pôvodný systém. Hlavný dôvod pre presun experimentu AAPA pod krídla nového testovacieho frameworku, bolo získať programové vybavenie pre tvorbu ďalších experimentov za použitia reálneho scenára. Boli pomenované konkrétne problémy a aplikované konkrétne riešenia. Najdôležitejšou kontrolovanou metrikou nového systému bola jeho rýchlosť. Použitie UE4 postavenom nad C++ už od začiatku sľubovalo isté zlepšenie v tejto oblasti. Zámer priniesť rýchlejšie reakčné doby bol teoreticky naplánovaný zmenami architektúry, konkrétne prenosom vykonávania skriptu do srdca UE4. Následne bol potvrdený aj experimentálne, kde v porovnaní so staršou verziou ale aj s očakávaniami autora suverénne a úplne spoľahlivo posunul reakčné doby v niektorých prípadoch až o niekoľko rádov nižšie. Dôležitým aspektom nového systému je jeho odolnosť voči rozprestreniu systému na viacero počítačov po sieti. Ani tento model vďaka novému usporiadaniu nebude mať na kvalitu výstupných dát žiaden vplyv.

Vývoj systému SpaNav 2.0 nie je na konci. Očakáva sa, že k experimentu AAPA sa pridajú ďalšie projekty, a jeho programové vybavenie sa bude postupne rozširovať, a to zasa naopak prispeje k tvorbe ďalších a efektívnejších testovacích experimentov.

Použitá literatúra

[1] J. M. Cimadevilla, M. Wesierska, A. A. Fenton, and J. Bures (2000): Inactivating one hippocampus impairs avoidance of a stable room-defined place during dissociation of arena cues from room cues by rotation of the arena. Proc. Natl. Acad. Sci. USA 98 (6):3531-6

[2] Jose M. Comadevilla, Rosa Cánovas, Luis Iribarne, Armando Soria, Laudino López (2011): A virtual-based task to assess place avoidance in humans. Journal of Neuroscience Methods 196 (2011) 45–50

[3] I. Fajnerová, M. Rodriguez, L. Konrádová, P. Mikoláš, K. Dvorská, M. Ungrmanová, J. Horáček, K. Vlček, D. Levčík, A. Stuchlík, C. Brom (2013): Satial memory in a virtual arena. 978-1-4799-0774-8/13/$31.00 ©2013 IEEE

[4] Iveta Fajnerova, Jana Kenney, Veronika Lobellova, Sarka Okrouhlicova, Ales Stuchlik, Daniel Klement (2014): Can rats solve the active place avoidance task without the room-bound cues? Behavioural. doi: 10.1016/j.bbr.2014.03.028 0166-4328

[5] Iveta Fajnerová, Mabel Rodriguez, David Levčík, Lucie Konrádová, Pavol Mikoláš, Cyril Brom, Aleš Stuchlík, Kamil Vlček and Jiří Horáček (2014): A virtual reality task based on animal research – spatial learning and memory in patients after the first episode of schizophrenia. doi: 10.3389/fnbeh.2014.00157

[6] I Fajnerová, K Vlček, C Brom, K. Dvorská, D Levčík, L Konrádová, P Mikoláš, M Ungrmanová, M Bída, K Blahna, F Španiel, A Stuchlík, J Horáček, M Rodriguez (2014): Virtual spatial navigation tests based on animal research – spatial cognition deficit in first episodes of schizophrenia. ISBN 978-0-7049-1546-6

[7] A. Stuchlík , T. Petrásek, I. Prokopová, K. Holubová, H. Hatalová, K. Valeš, Š. Kubík , C. Dockery , M. Wesierska (2013): Place Avoidance Tasks as Tools in the Behavioral Neuroscience of Learning and Memory. ISSN 0862-8408

[8] Ivana Šupalová, MFF UK(2009): Orientace v prostoru zkoumaná ve virtuální realitě

[9] Guy M. McKhann, David S. Knopman, Howard Chertkow, Bradley T. Hyman, Clifford R. Jack Jr., Claudia H. Kawas, William E. Klunk, Walter J. Koroshetz, Jennifer J. Manly, Richard Mayeux, Richard C. Mohs, John C. Morris, Martin N. Rossor, Philip Scheltens, Maria C. Carrillo, Bill Thies, Sandra Weintraub, Creighton H.May (2011Volume 7, Issue 3, Pages 263–269): The diagnosis of dementia due to Alzheimer's disease: Recommendations from the National Institute on Aging-Alzheimer's Association workgroups on diagnostic guidelines for Alzheimer's disease Phelpshttps://doi.org/10.1016/j.jalz.2011.03.005

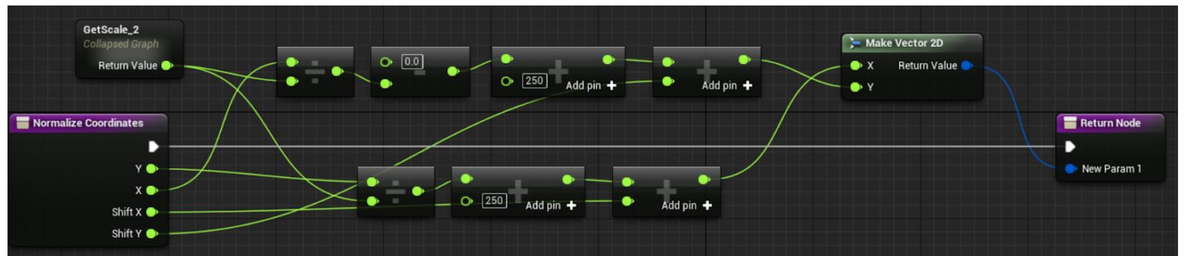
[10]Economic and Social Research Council. Britain in 2009. Swindon: ESRC Polaris House; 2008. pp60–8

[11] Adam Todd, Adrian Moore, Mark Ashton, Son Van (2010): Current research and development of treatments for Alzheimer's disease, The Pharmaceutical Journal (Vol 284) [https://www.pharmaceutical-journal.com/download?ac=1065649]

[12] Detecting navigational deficits in cognitive aging and Alzheimer disease using virtual reality Laura A. Cushman, Karen Stein, Charles J. Duffy: Neurology Sep 2008, 71 (12) 888-895; DOI: 10.1212/01.wnl.0000326262.67613.fe

Príloha

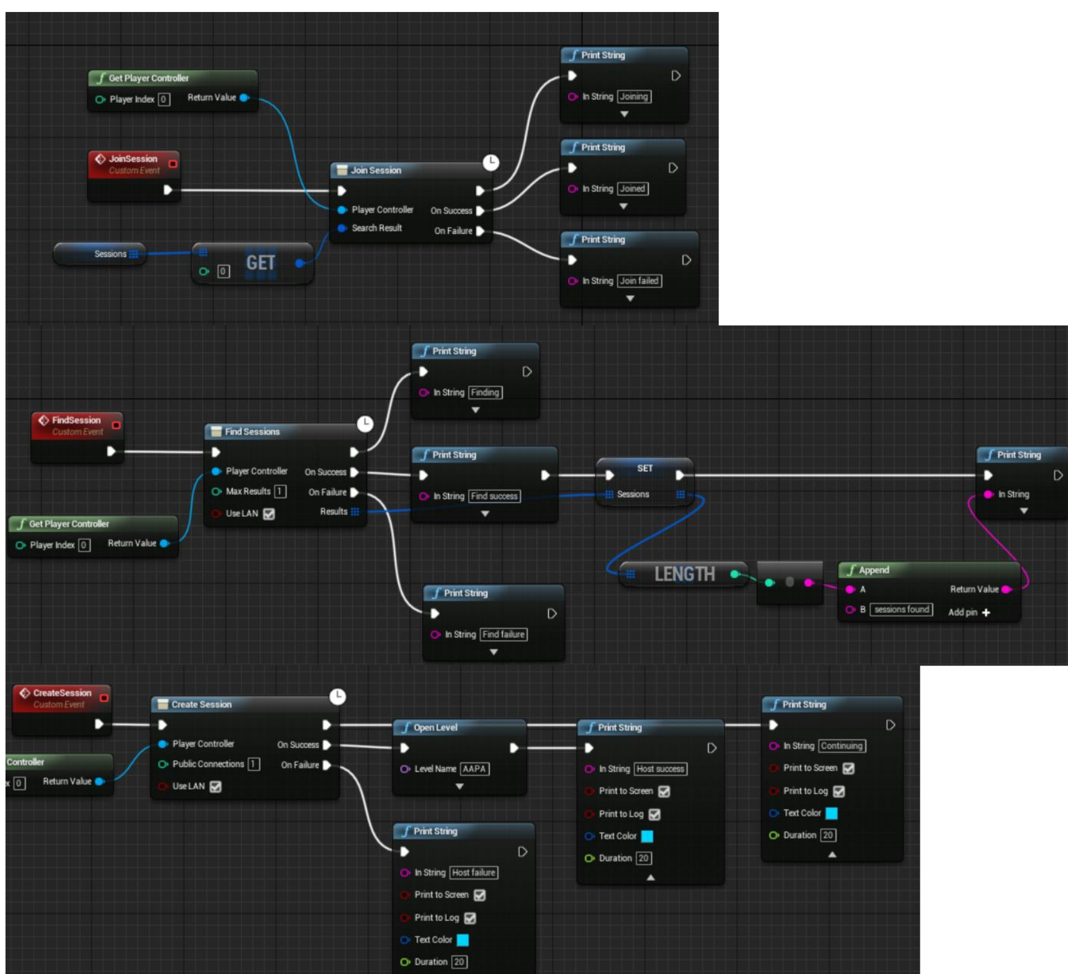
Blueprints



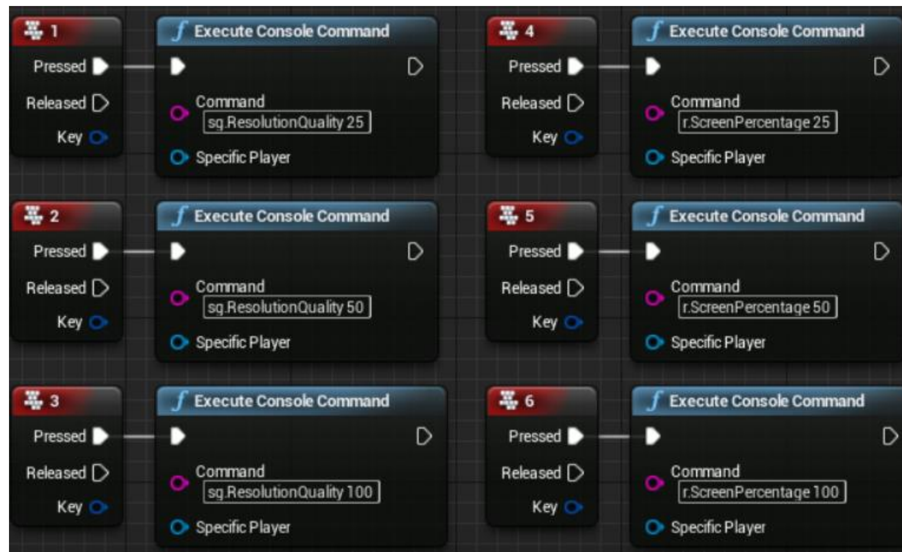
Príloha 1: Normalizácia koordinátov pre náhľad

Príloha 2: Materiál náhľadu

Príloha 3: Zmena rozlíšenia



Príloha 4: Funkcie mulitplayeru



Príloha 5: Ladenie výkonu

Refenčná príručka Javascript + SpaNav 2.0

Object	Method	Description	Example
ScriptedActor	*void SetRotation(Number, Number, Number)*	*SetRotation(Pitch, Yaw, Roll)* *No return value* *Sets rotation of actor in all three axeses.*	Platform.SetRotation(0,0,15)
	void SetVisibility(boolean)	SetVisibility(ToggleOn) No return value Sets visibility of actor	YellowArea.SetVisibility(!YellowArea.GetVisibility());
	boolean GetVisibility()	GetVisibility() Returns actual visiblity status GetVisibility() == true/false //actor is visible/not visible accordingly	YellowArea.SetVisibility(!YellowArea.GetVisibility());
	void AttachToActor(string)	AttachToActor(actorName) No return value Set actor with name == actorName as parent to object on which method was invoked. Actor will inherit transformation and rotation of the parent.	AttachedArea.AttachToActor("Platform");
	void DetachFromActor()	DetachFromActor() No return value Remove parent of object on whih method was invoked	AttachedArea.DetachFromActor();
	boolean IsColliding()	IsColliding() Returns collision status	function IsAvoidanceColliding(){ return ((RedArea.IsColliding() && RedActive)

		IsColliding() == true when there is active collision with another actor, according to rules of collisions defined in UE editor	|| (YellowArea.IsColliding() && YellowActive) || (TestArea.IsColliding() && TestActive) || (AttachedArea.IsColliding() && AttachedActive)); }

	void SetActive(boolean)	SetActive(toggleActiveStatusOn)
		
		No return value
		
		Sets actor's Active value. Active value is used in blueprints, to manage logic for altering visibility of actor using keyboard V-key.
			function ActivateArea(area){
			
			area.SetVisibility(false);
			area.SetActive(true);
			area.ActivateSound();
			
			return true;
			}

	void SetLocation(Number, Number, Number)	SetLocation(X,Y,Z) No result value Moves actor to coordinates [X,Y, Z] according to provided X, Y, Z values	//Log("New location resolved in " + iteration + " iteration"); **destination.SetLocation(x,y,86)** Log('Diamant moved to new position: '+x+','+y);

	Location GetLocation()	result GetLocation()
		
		Returns current location in *result* of a type *Location. Location* type has accessors for X,Y,Z values
			for(i = 0; i < destinations.length; i++){
			
			var loc = destinations[i].GetLocation();
			
			var sizeY = Abs(loc.Y – y);
			
			var sizeX = Abs(loc.X – x);
			}

	Rotation GetRotation()	result GetRotation()	function LogPlatformStats(){

		Returns current rotation in *result* of a type *Rotatio. Rotation* type has accessors for *Pitch, Roll, Yaw.*	var rot = Platform.GetRotation();
			
			LogT("Platform Rotation: [" + rot.Yaw + ";" + rot.Roll + ";" + rot.Pitch + "]");
			
			}

	void ActivateSound()	ActivateSound()
		No return value
		
		Activates sound alarm for collision with actor upon which method is invoked
			function ActivateArea(area){
			area.SetVisibility(false);
			area.SetActive(true);
			area.ActivateSound();
			
			return true;
			}

	void DeactivateSound()	DeactivateSound()
		
		No return value
		
		Deactivates sound alarm for collision with actor upon which method is invoked
			function DeactivateArea(area){
			
			area.SetVisibility(false);
			area.SetActive(false);
			area.DeactivateSound();
			
			return false;
			}

Player	Location GetLocation()	result GetLocation()
		
		Returns current location in *result* of a type *Location. Location* type has accessors for X,Y,Z values
			function LogPlayerStats(){
			var loc = Player.GetLocation();
			
			var rot = Player.GetRotation();
			
			LogT("Player Location: [" + loc.X + ";" + loc.Y + ";" + loc.Z + "]");
			
			LogT("Player Rotation: [" + rot.Yaw + ";" + rot.Roll + ";" + rot.Pitch + "]");
			
			}

	Rotation GetRotation()	result GetRotation()	function LogPlayerStats(){ var loc = Player.GetLocation();

		Returns current rotation in *result* of a type *Rotatio. Rotation* type has accessors for *Pitch, Roll, Yaw.*	var rot = Player.GetRotation();
			
			LogT("Player Location: [" + loc.X + ";" + loc.Y + ";" + loc.Z + "]");
			
			LogT("Player Rotation: [" + rot.Yaw + ";" + rot.Roll + ";" + rot.Pitch + "]");
			
			}

Experiment Debug HUD	void AddDebug(any)	AddDebug(input) No return value Adds temporary debug string representation of *input* to SLAVE application screen	Log("AAPA Experiment started"); ExperimentDebugHUD.AddDebug("AAPAExperiment started");
Experiment HUD	void SetAvoidanceText(any)	SetAvoidanceText(input) No return value Fills Experiment head up display text for avoidance areas (red) with string representation of *input*	if(seconds<10){secondsFormatted = "0" + seconds;} else{secondsFormatted = seconds;} ExperimentHUD.SetAvoidanceText(minutes + ":" + secondsFormatted + "." + centiseconds);

	void SetTime(any)	SetTime(input)
		No return value
		
		Fills Experiment head up display text for showing time (white) with sring representation of *input*
			var TimeToGo = 600;
			Timer(1, function(){
			--TimeToGo;
			LogT("Timer: " + TimeToGo);
			
			ExperimentHUD.SetTime((TimeToGo/60 | 0) + ":" + TimeToGo%60);
			
			}

	void SetDestinationText(any)	SetDestinationText(input)
		
		No return value
			// Initialize counter
			
			var destinationVisits = 0;

		Fills Experiment head up display text for showing destinations entrance count (green) with sring representation of *input*	ExperimentHUD.SetDestinationText(destinationVisits);
Sounds	void PlayCue(Number)	PlayCue(cueIdentifier) No return value Starts playing cue with identifier number == *cueIdentifier*	Sounds.PlayCue(1);
	void StopCue (Number)	StopCue(cueIdentifier) No return value Stops playing cue with identifier number == *cueIdentifier*	Sounds.StopCue(1);

	void PlaySound(Number)	PlaySound(soundIdentifier)
		
		No return value
		
		Plays (fire and forget) sound with index == *soundIdentifier*
			function PlayDestinationEnterSound(){
			
			Sounds.PlaySound(4);
			}

Math	Number Sqrt(Number)	result Sqrt(input)
		
		Returns square root of *input value* in *result*
			var powerY = sizeY*sizeY;
			var powerX = sizeX*sizeX;
			
			var distance = Sqrt(powerX + powerY);

	void Srand(Number)	Srand(seed)
		No return value
		
		Use *seed* value to set randomizer
			// Initialize randomizer
			Srand(GetTime());
			x = (Rand()%2000)-1000;
			y = (Rand()%2000)-1000;

	Number Rand()	result Rand()
		
		Returns random Number value in *result*
			// Initialize randomizer
			Srand(GetTime());
			x = (Rand()%2000)-1000;
			
			y = (Rand()%2000)-1000;

	Number Abs(Number)	result Abs(input)	var loc = destinations[i].GetLocation(); var sizeY = Abs(loc.Y – y);

		Returns *input* without decimals in *result*	var sizeX = Abs(loc.X – x);
Script globals*	void Log(any)	Log(input) No return value Logs *input* string representation into base log file(one without suffix)	Log("AAPA Experiment started");
	void LogT(any)	LogT(input) No return value Logs *input* string representation into log file for timed log messages(one with suffix 'T')	LogT("Platform Rotation: [" + rot.Yaw + ";" + rot.Roll + ";" + rot.Pitch + "]");

	Number GetTime()	result GetTime()
		
		Returns current time representation in number of miliseconds since start of epoch in *result*
			// Initialize randomizer
			Srand(GetTime());

	void Timer (Number, function)
	void Timer (Number, anonymousFunction)
		Timer(interval, existingFunction)
		
		Timer(interval, anonymousFunction)
		No return value
		
		Register function to run (either existing or anonymous) to a time interval specified in seconds. Values smaller than 1 second can be expressed as 0.NNN notation e.g.(0.005). Function will be run every 5 miliseconds.
			var TimeToGo = 600;
			
			Timer(1, function(){
			
			--TimeToGo;
			LogT("Timer: " + TimeToGo);
			
			ExperimentHUD.SetTime((TimeToGo/60 | 0) + ":" + TimeToGo%60);
			
			switch(TimeToGo){
			case 9*60:
			EndPhase(0);
			SetPhase(1); // v aapa2.0 js
			break;
			case 6*60:
			EndPhase(1);
			SetPhase(2);

			break; case 3*60: EndPhase(2); SetPhase(3); break; case 0: EndPhase(3); ExperimentHUD.SetTime("End"); StopScript(); break; } });
	void RegisterEvent(string, function) void RegisterEvent(string, anonymousFunction)	RegisterEvent(eventName, existingFunction) RegisterEvent(eventName, anonymousFunction) No return value Register event name to a function to run when speciefied event comes	RegisterEvent("Destination1.OverlapBegin", function(){ OnDestinationEnter(Destination1); });

	void StopScript()	StopScript()
		No return value
		
		Ends skript engine execution immediately
			switch(TimeToGo){
			EndPhase(3);
			ExperimentHUD.SetTime("End");
			
			StopScript();
			
			break;
			}