Razvoj video igara u Unreal Engine alatima

Vučković, Nikola

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Nikola Vučković

RAZVOJ VIDEO IGARA U UNREAL ENGINE ALATIMA

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, 2021.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Posvećujem ovaj diplomski rad mojoj majci Renati i djevojci Luciji koje su mi bile bezuvjetna podrška. Hvala svim prijateljima koji su mi olakšali put do diplome.

Sadržaj

Poglavlje 1

RAII princip i pametni pokazivači

1.1 RAII princip

Uvod

Opisat ćemo u kratkom i jednostavnom primjeru grešku koja se često potkrade početnicima, a nerijetko i iskusnijim programerima koji su dobro upoznati s bilo kojim jezikom koji ima sakupljač smeća (eng. *garbage collector*) i sam se brine za memory management, poput Java-e ili C#-a. Pretpostavimo da smo dobili zadatak u kojem često moramo izvoditi neke operacije nad nekim danim nizom, kao npr. suma svih članova, umnožak svih članova, prosječna vrijednost niza... Kako bi si olakšali posao odlučili smo implementirati pomoćnu klasu `Niz` koja sadrži pokazivač na dani niz i pomoćne funkcije koje računaju opisane zadatke.

```
class Niz
{
public:
    Niz(int duljina, std::initializer_list<int> clanovi)
        : _duljina(duljina)
    {
        _niz = new int[_duljina];

        int i = 0;
        for(int broj : clanovi)
            _niz[i++] = broj;
    }

    int Suma();
    int Produkt();
```

```
        int ProsjecnaVrijednost();

    private:
        int* _niz;
        int _duljina;
    };
```

Konstruktor klase prima duljinu niza i niz brojeva za koje, prilikom konstrukcije objekta, dinamički alocira potrebnu količinu memorije te ih kopira na pripadne memorijske lokacije. Ukoliko prije završetka izvršavanja programa dođe do destrukcije objekta klase `Niz` pozvat će se dodijeljeni destruktor klase te će uništiti pokazivač koji sadrži adresu mjesta u memoriji koje smo prijašnje alocirali. U tom trenutku izgubit ćemo pristup te sva saznanja gdje je u memoriji pohranjen naš niz. Taj blok memorije ostat će alociran do kraja izvršavanja programa te nam neće biti na raspolaganju kod budućih alociranja memorije. Dogodio nam se gubitak memorije (eng. *memory leak*), greška koja nije nečesta kod programskog jezika *C++*. Na sreću, u našem primjeru lako je izbjeći opisanu grešku. U destruktoru naše klase potrebno je pozvati dealokaciju memorije na koju pokazuje pokazivač `_niz`. Dodijeljivanjem tog zadatka destruktoru više ne trebamo brinuti jesmo li memoriju alociranu u konstruktoru oslobodili.

Opisani način, gdje memoriju alociramo u konstruktoru, a dealociramo u destruktoru, upravo je primjer programskog principa *Resource Acquisition Is Initialization* (**RAII**). Zbog problema gubitka memorije s verzijom *C++11* uvedeni su pametni pokazivači koji se drže programskog principa **RAII** pri upravljanju memorijom i uveliko olakšavaju tu obavezu. Implementacije klasa pametnih pokazivača moguće je pronaći u biblioteci *memory*. Pod princip **RAII**-a uključuje se i osiguravanje otključavanja zaključanih lokota kod višedretvenog programiranja te zatvaranje otvorenih *stream*-ova prilikom čitanja podataka s diska.

1.2 Pametni pokazivači

Prošlo rješenje je očito konstruirano za svrhu primjera jer u praksi u *C++*-u najčešće nećemo čuvati pokazivač na niz. Za takve slučajeve poželjnije je i preporuča se koristiti već gotove kontenjere (eng. *container*) koji dolaze s programskim jezikom, kao npr. `std::vector`-a, `std::array`-a ili `std::string`, ukoliko je niz sadržavao vrijednosti tipa *char*. Svi oni slijede **RAII** princip i nude metode i funkcije koje su već optimizirane i daju najefikasnija rješenja za određeni problem, npr. `insert`, `resize`, ... U objektno orijentiranom programiranju (**OOP**) vrlo često nam je potreban pokazivač na objekt klase u svrhu dijeljenja tog objekta da se izbjegne nepotrebno kopiranje ili radi iskorištavanja polimorfizma klase. U tim situacijama koristit ćemo pametne pokazivače koji će se umjesto nas brinuti da se zauzeta memorija kasnije i oslobodi. Ukoliko koristimo i obične pokazivače

i pametne pokazivače tada treba pripaziti da objekt pametnog pokazivača ne izađe iz svog dosega i pozove svoj destruktor u kojem će dealocirati objekt koji se nalazi na adresi na koju pokazivač referira jer nakon toga lako može doći do greške prilikom dereferenciranja pokazivača koji sada referira na adresu dealociranog memorijskog bloka.

`std::unique_ptr<T>`

Unique pointer implementira koncept ekskluzivnog vlasništva nad objektom. On sadrži pokazivač na memoriju koju smo alocirali i koju će on kasnije dealocirati ukoliko dođe do poziva destruktora pametnog pokazivača ili preuzimanja vlasništva nekog drugog pametnog pokazivača. Koncept ekskluzivnog vlasništva ostvaruje se tako što su izbrisani konstruktor kopije kopiranjem i operator pridruživanja kopiranjem. Time je izbjegnuta potencijalna greška duplog oslobađanja memorije (eng. *double free errors*). `std::unique_ptr<T>` zamijenio je dotadašnji `std::auto_ptr<T>` koji je kasnije s *C++17* i uklonjen iz jezika.

```
// prije C++11
template <typename T>
class UniquePTR
{
public:
    UniquePTR(T value) : _ptr(new T(value)) {};
    ~UniquePTR() { delete _ptr; };

    T* get() { return _ptr; }

private:
    // Onemogucimo konstruktor kopije kopiranjem
    // i operator pridruzivanja kopiranjem
    UniquePTR(const UniquePTR&);
    UniquePTR operator=(const UniquePTR&);

private:
    T* _ptr;
};
```

`std::shared_ptr<T>`

Zbog toga što u praksi najčešće trebamo više pokazivača koji referiraju na isti objekt nam `std::unique_ptr<T>` neće biti od tolike koristi. Problem *double free errors*-a tada bismo riješili brojanjem referenci, tj. jednostavnim brojačem koji broji koliko pametnih pokazivača postoji koji sadrže pokazivač na istu adresu. Kod kreiranja novog objekta s

istim pokazivačem treba brojač povećati za jedan, a prilikom destrukcije objekt pametnog pokazivača mora provjeriti postoji li još neki pametni pokazivač s istom adresom te ukoliko ne postoji, odnosno ako je brojač jednak 1 (postoji samo objekt čiji destruktor je pozvan), osloboditi memoriju na koju pokazuje. Kako bi svi objekti klase imali ažuriranu vrijednost brojača, njegova vrijednost sprema se u dinamički alociranu memoriju te se čuva pokazivač na nju. Da bi se izbjegao problem utrke za resursima, kod višedretvenog programiranja, za rad s brojačem koriste se *atomic* operacije koje osiguravaju ispravnost rada programa. `std::shared_ptr<T>` je klasa pametnog pokazivača koja implementira brojanje referenci. Za razliku od prijašnje opisanog `std::unique_ptr<T>`, kod `std::shared_ptr<T>` je dozvoljeno kopiranje objekta.

```
// prije C++11
template <typename T>
class SharedPTR
{
public:
    SharedPTR(T value) : _ptr(new T(value)),
        _cnt(new int(1)) {}

    // CCtor
    SharedPTR(const SharedPTR& sPtr) : _ptr(sPtr._ptr),
    _cnt(sPtr._cnt) { ++*_cnt; };

    SharedPTR& operator=(const SharedPTR& sPtr);

    ~SharedPTR()
    {
        if(--*_cnt == 0)
        {
            delete _ptr;
            delete _cnt;
        }
    }

private:
    T* _ptr;
    int* _cnt;
};
```

Pomoćne funkcije

Kako bi se u potpunosti iskoristio potencijal pametnih pokazivača preporuča se korištenje pomoćnih funkcija koje osiguravaju ispravnu inicijalizaciju pametnih pokazivača. To su funkcije `std::make_unique` i `std::make_shared` koje kreiraju objekte pametnih pokazivača te kreiraju objekte na koje će pokazivati i pozivaju njihove odgovarajuće konstruktore ovisno o proslijeđenim argumentima u funkcijama.

```
#include <memory>

class Vector3i
{
public:
  Point3D(int x, int y) : _x(x), _y(y), _z(0) {};
  Point3D(int x, int y, int z) : _x(x), _y(y), _z(z) {};

  // ...

private:
  int _x, _y, _z;
};

std::unique_ptr<Vector3i> p1 =
    std::make_unique<Vector3i>(2, 3);
std::shared_ptr<Vector3i> p2 =
    std::make_shared<Vector3i>(2, 3, 4);
```

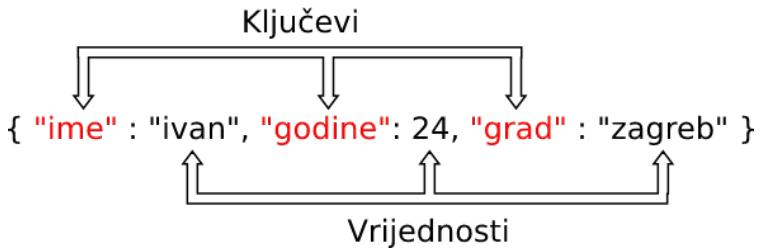
Poglavlje 2

JSON parser

2.1 JSON

Video igra, opisana na kraju ovog diplomskog rada, sastoji se od servera i klijenata koji stupaju u komunikaciju i razmjenjuju poruke, ali poruke koje razmjenjuju mogu biti poslane jedino u obliku niza byte-ova, a to nam u *C++*-u može predstavljati problem jer nam ne pruža mogućnost da neki složeni objekt klase pretvorimo u niz byte-ova. *Serijalizacija* je proces konverzije neke strukture podataka u format koji može biti spremljen na disk ili poslan preko mreže, a koji omogućava kasniju rekonstrukciju iz dobivenog formata u originalni objekt procesom *deserijalizacije*. Nažalost, s *C++* jezikom ne dolazi podrška za opisanim procesima, međutim postoje mnoge biblioteke koje pružaju klase za serijalizaciju, poput već spomenute *boost* biblioteke ili *Qt* biblioteke, no u ovom projektu korišten je **JSON** format te su objekti serijalizirani koristeći vlastite klase.

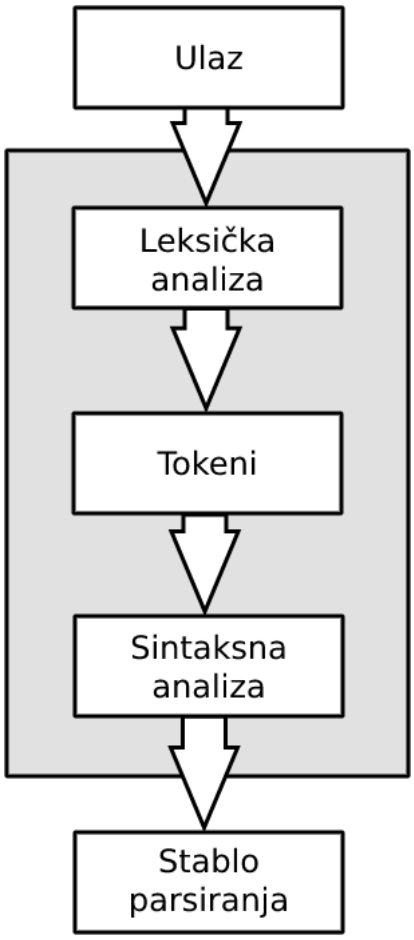
Kod **JSON** (*JavaScript Object Notation*) formata, objekt se transformira u `std::string`. Podaci su reprezentirani uređenim parom ključ-vrijednost, gdje je ključ tipa `std::string`, dok vrijednost može biti broj, `std::string`, niz, objekt, `bool`... Osim jednostavnosti i čitljivosti JSON *string*-a, jedna od velikih prednosti formata je malo zauzimanje prostora što je iznimno bitno kod slanja podataka preko mreže.



Slika 2.1: Primjer JSON *string*-a

2.2 Parser

Parser je program koji za dani ulazni podatak, najčešće proizvoljni *string*, provede analizu te na temelju nje kao rezultat vrati određenu strukturu podataka koju korisnik kasnije može jednostavno koristiti. Analiza se sastoji od *leksičke* analize koja nad danim ulazom provodi proces *tokenizacije* i *sintaksne* analize koja pomoću dobivenih tokena provjerava poštuje li niz tokena gramatiku jezika te vraća strukturirani oblik ulaza.



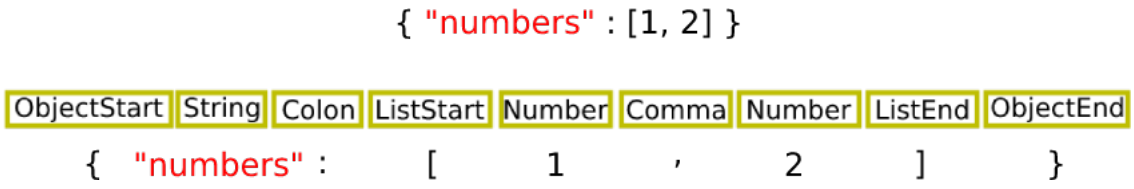
Slika 2.2: Dijagram parsera

Leksička analiza

Leksička analiza je proces rascjepljivanja ulaza u tokene. *Tokeni* su skup ”imena” koji imaju neko značenje u kontekstu jezika koji parsiramo. Program koji provodi analizu zo-

vemo *lekser*. Leksička analiza utvrđuje sastoji li se ulazni podatak od ispravnih tokena koje smo unaprijed definirali i koji čine jezik, ali ne može utvrditi nalaze li se dobiveni tokeni u ispravnom poretku. U našem primjeru neki od tokena za JSON format bi bili navodnici, dvotočka, brojevi, riječi unutar navodnika... Tokeni koje koristimo u našem parseru su navedeni u kodu ispod.

```
// Tokeni
enum class TokenType
{
  ObjectStart, // 0
  ObjectEnd, // 1
  String, // 2
  Number, // 3
  Key, // 4
  Colon, // 5
  Comma, // 6
  Bool, // 7
  Invalid, // 8
  ListStart, // 9
  ListEnd // 10
};
```



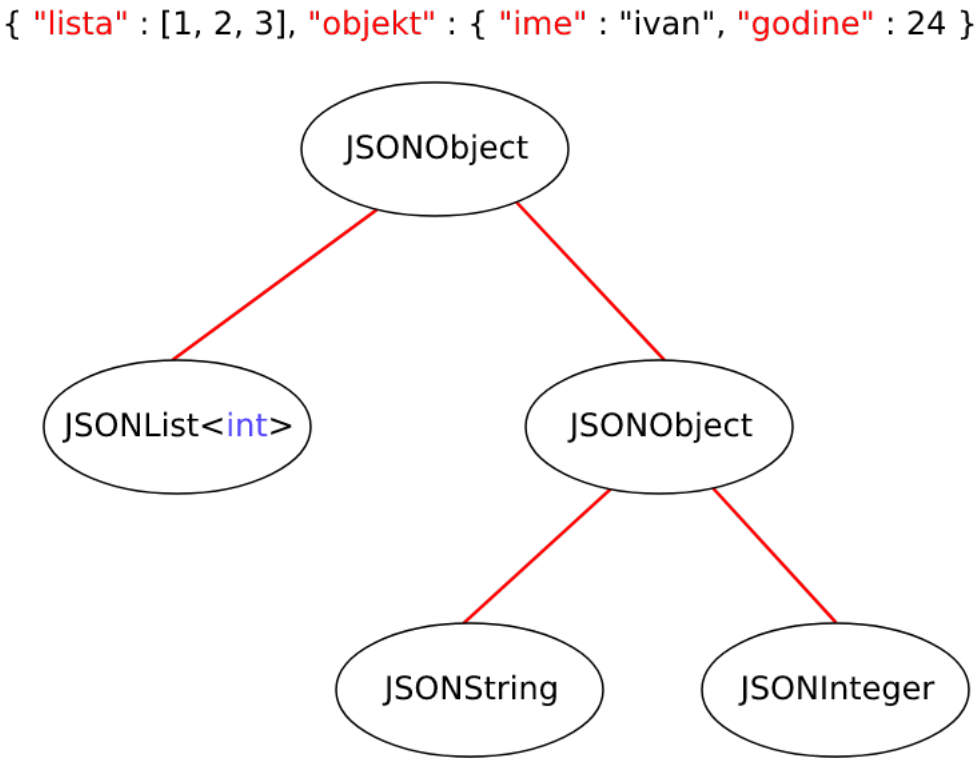
Slika 2.3: Primjer tokenizacije

Sintaksna analiza

Nakon što je završila leksička analiza i proces tokenizacije ulaznog podatka započinje faza *sintaksne* analize. U sintaksnoj analizi za dobiveni niz tokena pokušava se utvrditi zadovoljava li poredak tokena u nizu gramatiku jezika. Ukoliko utvrdimo da niz dobiven tokenizacijom zadovoljava uvjete onda generiramo stablo parsiranja iz dobivenih tokena. Tipove objekata koje će naš parser podržavati su `int`, `float`, `bool`, `std::string`, `std::vector<int>` te ”object” koji predstavlja sami JSON podatak iz čega slijedi da je dozvoljeno imati JSON unutar JSON-a.

Za svaki od navedenih tipova imat ćemo klasu koja će ih reprezentirati u našem stablu. Svaka klasa će sadržavati vrijednost koja je pročitana iz JSON stringa pod tim ključem

te će svaka klasa, nasljeđivanjem iz bazne klase, imati osnovne funkcije koje će nam biti potrebne. `JSONNode` je apstraktna bazna klasa iz koje ćemo izvesti sve ostale klase. Nadjačavanjem (eng. *override*) čiste virtualne funkcije `ToString` u svakoj podklasi `JSONNode`-a u mogućnosti smo opet njenim pozivom nad objektom koji čini korijen stabla dobiti originali JSON string. Svaki korijen stabla ili podstabla bit će objekt tipa `JSONObjectNode`. Djelomičnom specijalizacijom predloška, funkcija `GetValue` vratit će ispravan tip ovisno o tome koji je tip zadan u predlošku te su mogući tipovi predloška ograničeni na tipove koje smo dopustili za naš parser. Istim načinom metoda `SetValue` dodat će `JSONObjectNode`-u novi čvor kao dijete pri izgradnji stabla. Sva njegova djeca spremljena su u rječniku `Table` koji za vrijednosni tip ima pokazivač na objekt tipa `JSONNode`. Polimorfizam klasa omogućava da iteriranjem po njegovim vrijednostima i pozivima `ToString` funkcije za rezultat dobijemo originalni JSON string.



Slika 2.4: Primjer stabla parsiranja

```
#include "JSONNode.h"

// JSONObjectNode.h
class JSONObjectNode : public JSONNode
```

```
{
public:
  typedef std::shared_ptr<JSONObjectNode> Ptr;

  template<typename T>
  T GetValue(std::string key);

  template<typename T>
  void AddValue(std::string key, T value);

    // ..

  virtual std::string ToString() override;
private:
  // ..

  std::map<std::string, JSONNode::Ptr> Table;
};

// JSONObjectNode.cpp
std::string JSONObjectNode::ToString()
{
  std::string result = "{";

  for (auto &pair : Table)
  {
    std::string value = pair.second == nullptr ? "{}" :
        pair.second->ToString();

    result += "\"" + pair.first + "\" : " + value + ", ";
  }

  return result + "}";
}
```

Serijalizacija objekta pomoću JSON formata

JSON parser uveliko će nam pomoći pri serijalizaciji objekata, međutim i dalje nemamo način da neki objekt pretvorimo u JSON string. Mogućnosti koje tek imamo s parserom su pretvorba JSON stringa u stablo parsiranja i obrnuto. Želimo da sve klase, koje želimo serijalizirati, sadrže metode u kojima bismo specifirali koje varijable će pri serijalizaciji biti zapisane u JSON string i pod kojim ključem i obratno kod isčitavanja pri

deserijalizaciji. Iako *C++* ne nudi opciju sučelja, taj nedostatak možemo nadomjestiti višestrukim nasljeđivanjem i abstraktnim klasama. "Sučelje" `ISerializable` dozvolit će nam, nadjačavanjem funkcija `Encode` i `Decode`, serijalizaciju objekata podklase. Odsad nadalje varijable članice neke klase imenovat ćemo u *UpperCamelCase* formatu dok ćemo ključeve u JSON stringu pisat u *lowerCamelCase* formatu. Tako ćemo se zaštititi od grešaka i iznimiki koje bi se javile zatraživanjem vrijednosti s krivim ključem.

```
#include "JSONObjectNode.h"

// Primjer ISerializable podklase
class Car : public ISerializable
{
public:
    virtual std::string Encode() override;
    {
        JSONObjectNode::Ptr node
            = std::make_shared<JSONObjectNode>();

        node->AddValue<int>("price", Price);
        // ...

        return node->ToString();
    }

    virtual void Decode(JSONObjectNode::Ptr node) override;
    {
        Price = node->GetValue<int>("price");
        // ...
    }

private:
    int Price;
    // ...
};
```

Podklasa `ISerializable` mora nadjačati metodu `Decode` i funkciju `Encode` te u njihovim definicijama definirati koje vrijednosti će se dohvatiti iz JSON stringa i spremiti u JSON. S implementiranom klasom imamo sve potrebno za serijalizaciju objekata i u mogućnosti smo kao poruku poslati složenije strukture podataka preko mreže.

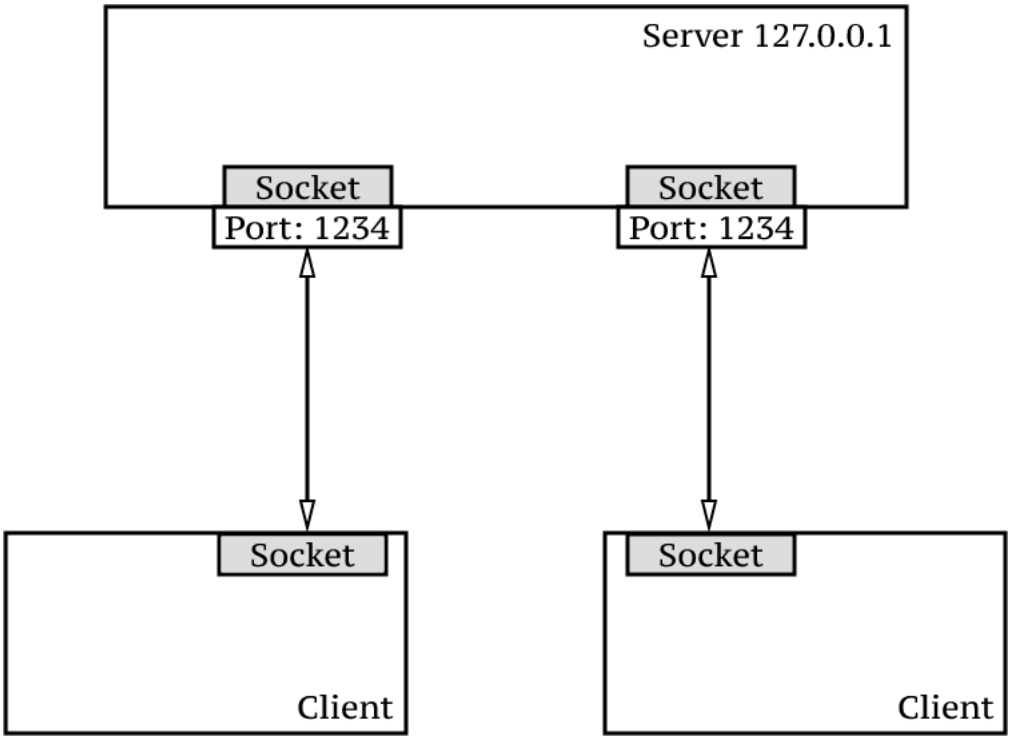
Poglavlje 3

TCPServer i TCPConnection klase

3.1 *Asio* biblioteka

U ovom poglavlju cilj nam je implementirati server, centralnu jedinicu u našem sustavu koja će upravljati novim i postojećim konekcijama, primati i razdjeljivati pristigle poruke te biti zadužena za uparivanje igrača i započinjanje nove igre. Klijent će ulaskom u igru otvoriti konekciju sa serverom i poslati zahtjev za novom igrom. Server, kada obradi zahtjev, dodat će klijenta u red čekanja za novu igru gdje će klijent ostati do trenutka dok isto ne napravi neki drugi klijent. U tom trenutku server će generirati matricu dimenzija $m \times n$ koja predstavlja glavnu ploču u igri i broj figurica koje igrač mora skupiti za pobjedu te će ih proslijediti kao poruku klijentima što će označiti početak igre.

Asio (*asynchronous input/output*) je samo jedna od mnogo *C++* biblioteka koje dolaze u sklopu *Boost* paketa koji se može preuzeti s njihove službene stranice i slobodno koristiti u svojim projektima. *Asio* sadrži klasu `tcp::socket` potrebnu za otvaranje konekcije, osluškivanje dolaznih podataka i slanje podataka na mreži neoviosno o platformi na kojoj se program izvršava. Klijent mora definirati krajnju točku (eng. *endpoint*) koja je određena IP adresom i port brojem. Pozivom metode `connect` mrežna utičnica (eng. *socket*) pokušat će uspostaviti mrežnu konekciju s krajnjom točkom koja je prosljeđena kao argument u pozivu metode. Ukoliko je uspostava konekcije uspješna, server na čiju krajnju točku se klijent spojio, otvorit će svoj `socket` s klijentovom krajnjom točkom.



Slika 3.1: Dijagram server klijent

Asinkroni način rada u programu omogućuje nam da se više stvari događa odjednom, odnosno da se izvršavanje programa ne izvršava liniju po liniju nego rascjepkano od trenutka poziva asinkrone funkcije. Najčešće nam je to korisno kada neko određeno vrijeme moramo nešto čekati, a ne želimo zaustaviti izvršavanje glavnog programa s tim čekanjem. Asinkronost nam omogućuje da dretva glavnog programa nastavi dalje s radom do trenutka kada će asinkrona funkcija javiti da je gotova s radom ili počne s radom ukoliko smo zakazali njeno izvršavanje u nekom kasnijem trenutku.

Konstruktor klase `tcp::socket` kao argument prima referencu objekta klase `io_context` kojeg kasnije upošljava s trenutnim zadacima ili onima koji su zakazani za kasnije izvršavanje. Pozivom metode `run` klase `io_context` izvršavanje glavne dretve blokira se dokle god postoje zadaci koji su zakazani za kasnije izvršavanje ili se trenutno izvršavaju. Bitno je pripaziti da se objekt, nad kojim je zakazano buduće izvršavanje funkcije, ili argumenti, koji su prosljeđeni funkciji, ne unište do trenutka izvršavanja zakazane funkcije jer inače će doći do greške za vrijeme izvršavanja programa.

Klasa `acceptor` (u našem slučaju vezana uz TCP protokol) u konstruktoru traži referencu `io_context` objekta i specifikacije za koje vrste krajnjih točaka će prihvatiti buduću konekciju, a s metodom `async_accept` započet će posao osluškivanja, prihvaćanja konek-

cije i otvaranja socketa prema klijentu. Kada dođe do prihvaćanja nove konekcije, pozvat će se tzv. *Accept Handler*, metoda koju smo proslijedili kao argument pri pozivu, a koja prima argument tipa `boost::system::error_code`.

U ovom slučaju, uvjet da Accept Handler prihvaća samo jedan argument može nas poprilično ograničiti. Što ako bismo za *callback* htjeli pozvati metodu koja prima više od jednog argumenta? U teoriji izračunljivosti *specijalizacijom* smo algoritmu fiksirali ulazne podatke na neke konkretne vrijednosti. Time smo dobili ekvivalentan algoritam sa smanjenim brojem ulaznih podataka. Unutar biblioteke *functional* nalazi se funkcija `std::bind` koja kao argumente prima funkciju kojoj želimo fiksirati argumente, adresu objekta nad kojim će biti pozvana prosljeđena funkcija te argumente koje želimo fiksirati, dok ćemo za slobodne argumente proslijediti tzv. `placeholder`-e. Za povratnu vrijednost dobit ćemo novu ekvivalentnu funkciju s fiksiranim argumentima koje smo odlučili fiksirati pri pozivu. Problem smo mogli također riješiti tako da funkciju pozovemo u tijelu *lambda* funkcije, a argumente koje smo htjeli fiksirati dohvatimo u klauzuli hvatanja lambda funkcije.

```
#include <boost/asio.hpp>
#include <functional>

// TCPServer.h
class TCPServer
{
public:
    typedef std::shared_ptr<tcp::socket>> SocketPTR;

    TCPServer(io_service& ioService) : IoService(ioService),
        Acceptor(ioService, tcp::endpoint(tcp::v4(), 1234)) {};

    void StartAcceptingConnections();
    // ...

private:
    void OnConnectionAccepted(SocketPTR, const error_code&);

private:

    io_service& IoService;
    tcp::acceptor Acceptor;

    std::vector<SocketPTR>> Connections;
};


// TCPServer.cpp
```

```
void TCPServer::StartAcceptingConnections()
{
    SocketPTR newSocket
        = std::make_shared<tcp::socket>(IoService);

    Acceptor.async_accept(newSocket,
        std::bind(&TCPServer::OnConnectionAccepted, this,
        newSocket, boost::asio::placeholders::error);
    // ili
    // Acceptor.async_accept(newSocket, [this, newSocket]
    // (const error_code& error)
    // {
    //      this->OnConnectionAccepted(newSocket, error);
    // });
}

void TCPServer::OnConnectionAccepted(SocketPTR socket,
    const error_code& error)
{
    if(!error)
    {
        std::cout << "Prihvacena nova konekcija!\n";
        Connections.push_back(socket);
    }

    StartAcceptingConnections();
}
```

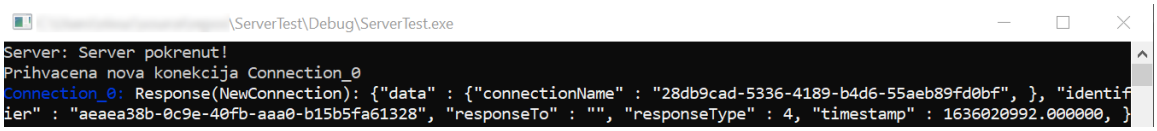
Sada kada imamo server koji prihvaća konekciju i za danu konekciju kreira novi socket, pozivom funkcije `async_recieve` nad tim socketom započet će osluškivanje nadolazećih poruka. U trenutku kada socket javi da postoje podaci koji čekaju da se obrade pozvat će se *Read Handler* u kojem smo definirali što želimo s podacima u trenutku kada pristignu. Pristigli podaci bit će spremljeni u *buffer* iz kojeg kasnije lagano konstruiramo `std::string` i dobijemo poruku koja je poslana. Analogno, pozivom `async_write` možemo poslati poruku te će se u trenutku kada se poruka pošalje pozvati *WriteHandler* u kojem možemo provjeriti je li se poruka uspješno poslala. Funkcije `async_receive` i `async_write` zadat će `io_context`-u posao na kojem će raditi do trenutka primanja i slanja poruke. U trentuku kada su završili posao uklonit će ga s liste zakazanih poslova `io_context`-a. Kako bismo održali kontinuirani rad servera bitno je u *handler*-ima ponovnim pozivima funkcija zadati novi posao primanja ili slanja podataka.

```
// TCPConnection.h
class TCPConnection
{
```

```
public:
    // ..

    void StartReceivingData();
    void OnDataReceived(const boost::system::error_code&,
        std::size_t);

private:
    tcp::socket Socket;
    boost::array<char, 1024> Buffer;

    // ..
};

// TCPConnection.cpp
void TCPConnection::StartReceivingData()
{
    Socket.async_receive(boost::asio::buffer(Buffer),
        0, std::bind(&TCPConnection::OnDataReceived,
        this, placeholders::error,
        placeholders::bytes_transferred));
}

void OnDataReceived(const boost::system::error_code& error,
        std::size_t bytesReceived)
{
    if(!error)
    {
        // ...
    }

    StartReceivingData();
}
```

3.2 Request i Response klase

U nastavku ćemo uspostaviti neka pravila u komunikaciji, tj. kojeg tipa poruka treba biti i što svaka poslana poruka treba sadržavati od informacija. Razlikovat ćemo dvije vrste poruka: *request* i *response*. Klijent će, kada želi obavijestiti server da se neki događaj ili promjena dogodila, poslati poruku tipa *request* prema serveru. Server će poslanu poruku primiti te, ovisno o tipu zahtjeva, obraditi zahtjev i odgovoriti porukom tipa *response* koju će klijent kasnije primiti.



Slika 3.2: Server i primjer poruke

Svaka poruka, neovisno o tipu, sadržavat će jedinsteveni *Id* broj pomoću kojeg će server ili klijent raspoznati o kojem podtipu se radi. Podtipove ćemo spremiti u *enum* klase *RequestType* i *ResponseType* čije će definicije imati i serverski kod i klijentski kod. Zbog toga je bitno svaki put kada napravimo promjenu ili dodamo novi *Id* ažurirati kod na serveru i klijentu jer u suprotnom jedan od sudionika neće moći prepoznati o kakvoj poruci se radi i doći će do neispravnog rada aplikacije. Poruke koje šaljemo prema serveru uvijek će biti tipa *request* dok će klijent primati poruke tipa *response*. Tako ćemo uvijek znati o kojem tipu se radi, ovisno o serveru ili klijentu, te nemamo potrebu za utvrđivanjem tipa poruke.

```
// RequestType.h
enum class RequestType
{
  StartNewGame = 1,
  CancelStartNewGame = 2,
  TileMove = 3,
  GenerateNewTiles = 4,
  EndTurn = 5
};
```

Tijelo poruke može se sastojati samo od *Id*-a te bi u tom slučaju poruka predstavljala neki signal upućen drugoj strani da se nešto dogodilo ili da je zahtjev obrađen. Međutim, u većini slučajeva htjet ćemo poslati neke podatke u poruci. U tom slučaju iskoristit ćemo `JSONObjectNode` u koji ćemo spremiti sve podatke koje želimo da poruka sadrži. Kasnije će pozivom funkcije `ToString`, objekt biti serijaliziran, tj. naši podaci bit će sadržani u JSON stringu koji lako možemo uključiti u poruku i poslati. Prilikom obrade poruke, program će deserijalizirati JSON string te će time dobiti željene podatke.

Željeli bismo da nam server na zahtjev klijenta odgovori porukom, ali što ako je klijent poslao dva zahtjeva s istim *Id* brojem? Kako će klijent prepoznati na koju poruku je točno server odgovorio? Kako bismo izbjegli taj problem u poruku ćemo uključiti *globally unique identifier* (**GUID**), 128-bitnu strukturu koja se sastoji od 32 heksadecimalne vrijednosti koja će nam poslužiti kao ključ za poslanu poruku. Iako u teoriji taj ključ nije jedinstven, u praksi je vjerojatnost da se izgeneriraju dva ista ključa gotovo jednaka nuli.

```
// Primjer GUID-a
28db9cad-5336-4189-b4d6-55aeb89fd0bf
```

Klase `Request` i `Response` obje implementiraju opisana svojstva te se razlikuju ovisno je li klasa implementirana na serveru ili klijentu. Klijentski `Request` sadrži statičku funkciju `GenerateRequest` koja za dani *Id* zahtjeva vraća objekt te klase. Također, klijentski `Request` podklasa je `IEncodable`-a što nam omogućava serijalizaciju klase koju će serverski `Request`, podklasa klase `IDecodeable`, deserijalizirati. Tvrdnja analogno vrijedi i za klasu `Response`, samo u obrnutim ulogama klijenta i servera uz malu iznimku klijentskog `Request`-a koji sadrži još `Callback` varijablu koja se pozove kada klijent za odgovor dobije poruku s njegovim **GUID**-om.

U trenutku kada server prihvati poruku on će instancirati objekt klase `Request` sa sadržajem dobivene poruke te ga proslijediti klasi `CallbackController` koja je zadužena za pozivanje određene funkcije ovisno o podtipu zahtjeva. Konstruktor klase pozvat će metodu `RegisterOnRequestCallbacks` u kojoj smo uparili `RequestType` i funkciju koja kao povratnu vrijednost vraća `Response`. Time smo osigurali da u trenutku primanja poruke svaki podtip bude uparen s *callback* funkcijom. Generirani `Response` onda se šalje natrag prema klijentu koji će obraditi poruku i pozvati `Callback` koji je `Request`, s odgovarajućim **GUID**-om, zadao u trenutku slanja poruke prema serveru.

```
// TCPConnection.cpp
void OnDataReceived(const boost::system::error_code& error,
        std::size_t bytesReceived)
{
    if(!error)
    {
        std::string receivedMessage(Buffer.begin(),
            Buffer.begin() + bytesTransfered);

        JSONObjectNode::Ptr requestNode
            = JSONParser::GetInstance().Decode(receivedMessage);

        Request request;
    request.Decode(requestNode);

    HandleRequest(request);
    }

    StartReceivingData();
}

void TCPConnection::HandleRequest(Request request)
{
```

```
    std::string responseMessage = _gameController->
        Callbacks->InvokeCallback(request).Encode()->ToString();

    SendResponse(responseMessage);
}

// CallbackController.h
class CallbackController
{
public:
    // ...

  void RegisterOnRequestCallbacks();

  Response InvokeCallback(Request& request);

private:
  std::map<RequestType, std::function<Response(Request)>>
      _onRequestCallbacks;
};

// CallbackController.cpp
Response CallbackController::InvokeCallback(Request& request)
{
  return _onRequestCallbacks[request.Type](request);
}

void CallbackController::RegisterOnRequestCallbacks()
{
    _onRequestCallbacks[RequestType::StartNewGame]
        = [](Request request) {
            return Response::CreateResponse
            (ResponseType::NewGameStarted, request.Identifier,
            std::make_shared<JSONObjectNode>());
        });
}
```

Poglavlje 4

Unreal Engine 4

4.1 Općenito

Unreal Engine (**UE**) jedan je od najpoznatijih i najkorištenijih tzv. *pokretača igara* (eng. *game engine*). Pokretač igara je skup alata koji se koriste za razvoj i pokretanje video igara. Njegovim korištenjem uveliko si olakšavamo sami proces razvoja igara jer mnoge stvari dolaze već implementirane koje bismo inače, u svakom novom projektu, morali sami isprogramirati i postaviti. Pri završetku razvoja UE nam omogućava izgradnju projekta za specifičnu platformu (*Windows*, *Android*, *iOS*...). U prijašnjim verzijama *UE* je koristio *UnrealScript*, vlastiti jezik inspiriran *C++* i *Java* programskim jezicima. Izlaskom Unreal Engine 4 verzije prelazi na *C++*, stoga ćemo i sav kod za igru pisati u njemu. Osim programiranja koristeći tekstualni kod, *Unreal Engine* podržava tzv. *visual scripting* u obliku njegovih *Blueprint*-a koje ćemo kasnije detaljnije opisati. Za potrebe ovog diplomskog rada korištena je verzija 4.26.2 Unreal Engine-a.

4.2 Kreiranje novog projekta

Ulaskom u Unreal Engine bit će nam ponuđene dvije opcije: nastavak rada na prijašnjim projektima ili kreacija novog projekta. Za novi projekt ponuđene su 4 kategorije: *Games*, *Film, Television and Live Events*, *Architecture, Engineering and Construction* i *Automotive, Product Designand Manufacturing*. Svaka od njih nudi opcije i postavke projekta koje će olakšati razvoj u odabranom smjeru. Tema ovog diplomskog rada je razvoj igara i zbog toga ćemo odabrati *Games* kategoriju. U sljedećem koraku ponuđeno je mnoštvo već gotovih malih projekata koji dolaze s Unreal Engine-om. Njihovim pokretanjem moguće je brzo isprobati mogućnosti Unreal Engine-a, ali i nastaviti razvoj nadogradnjom na pripremljenim klasama. U ovom projektu odabrana je *Blank* opcija. U zadnjem koraku odabiremo

glavne postavke projekta te naziv projekta. Za naš projekt odabrali smo *C++* projekt i *Mobile* kao ciljanu platformu za razvoj.

Slika 4.1: Opcije kod novog projekta

4.3 Sučelje editora

Nakon što smo odabrali postavke novog projekta započet će generiranje projekta. Završetkom generiranja projekta otvorit će se *Unreal Editor*. Sučelje Unreal Editor-a sastoji se od sljedećih djelova:

1. *Tab* i *Menu* trake - Menu traka sadrži neke opće opcije i one vezane uz editor, dok su u Tab traci prikazane kartice svih otvorenih editora i levela.

2. Alatna traka (eng. *toolbar*) - Sadrži skup naredbi, pružajući brz pristup najčešće korištenim alatima i naredbama (npr. *Compile*, *Save*, *Play*...).

3. *Place Actor* panel - Omogućava brzo dodavanje već predefiniranih objekata u scenu.

4. *Viewport* - Prozor u kojem vidimo scenu koju stvaramo. Njime se možemo kretati baš kao što bismo se kretali kad bismo igrali igru s odabranim levelom. Sadrži niz alata koji će nam pomoći da pogledamo detalje točno onih podataka koje želimo (npr. klikom na objekt označit ćemo ga te će se u *Details* prozoru prikazati informacije odabranog objekta).

5. Preglednik sadržaja (eng. *Content Browser*) - Prozor u kojem možemo stvarati, dodavati, pregledavati i modificirati podatke unutar Unreal Editor-a. Također, pruža

mogućnost organizacije podataka unutar mape i operacije nad podacima (*move*, *copy*, *rename*...).

6. *World Outliner* - Panel koji prikazuje sve *Actor*-e unutar scene u hijerarhijskom prikazu stabla. Sadrži i traku za pretraživanje *Actor*-a unutar scene koja će nam olakšati pretragu kasnije u projektu kada ćemo imati više *Actor*-a.

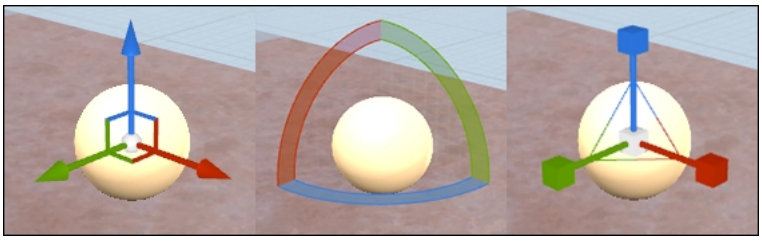
7. *Details* panel - Područje koje pruža opciju za uređivanje svojstava odabranog objekta. Sastoji se od trake za pretraživanje svojstava i kategorija koje sadrže svojstva pridružena objektu koja možemo pogledati i modificirati.

Slika 4.2: Sučelje UE alata (slika preuzeta s [1] stranice)

Primjer dodavanja novog objekta u scenu

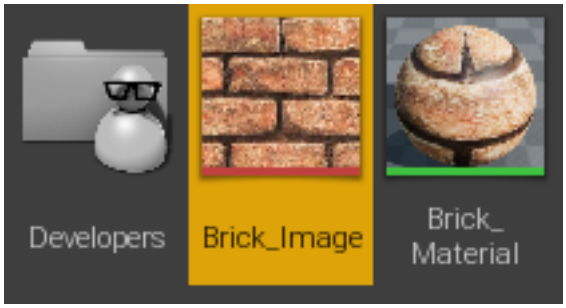
Ako želimo dodati novi objekt u scenu, unutar *Place actor* panela možemo odabrati *Basic* → *Cube* opciju te željeni objekt povući u naš *Viewport*. Unutar *World Outliner* panela možemo vidjeti da je *Actor* naziva *Cube* uspješno dodan u scenu. Klikom na objekt unutar *Viewport* označit ćemo ga te ga možemo dalje transformirati. Pritiskom **W**, **E** ili **R** odabrat ćemo opciju s kojom, pritiskom i držanjem lijevog klika na objektu i pomicanjem miša, možemo **pomicati**, **rotirati** ili **mijenjati veličinu** odabranog objekta. U gornjem desnom kutu *Viewport* prozora možemo odabrati restrikcije za navedene transformacije (npr. minimalna

rotacija pomakom miša ili *snap to grid* opcija). Ukoliko želimo unijeti konkretne vrijednosti za veličinu objekta to možemo učiniti unutar *Details* panela pod *Transform* kategorijom gdje su prikazana svojstva pozicije, rotacije i veličine. Nakon što smo učinili željenu transformaciju, promjene možemo pogledati u *Viewport* prozoru. Pritiskom desnog klika možemo rotirati pogled unutar scene, dok se pritiskom i desnog i lijevog klika možemo pomicati lijevo, desno, gore i dolje relativno obzirom na trenutni pogled unutar scene.



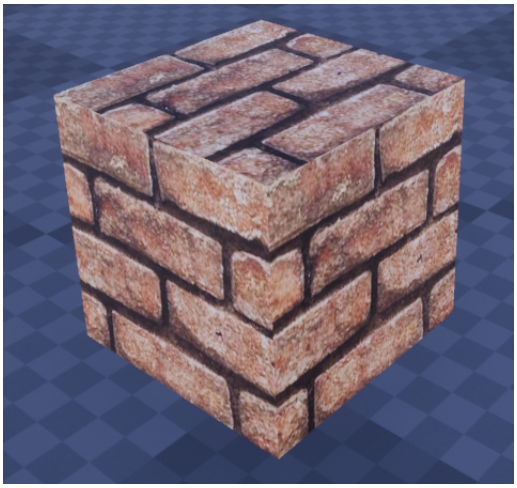
Slika 4.3: Opcije *translatiranja*, *rotacije* i *skaliranja* (slika preuzeta iz knjige [6])

Desnim klikom unutar *Content browser*-a i odabirom *Import Asset* opcije možemo dodati nove resurse u projekt. Želimo dodati sliku koju ćemo pretvoriti u `material` koji se onda može postaviti kao `material` kreiranog *Cube* objekt. Nakon što smo odabrali proizvoljnu sliku, u ovom slučaju slika ciglenog zida, desnim klikom na nju i odabirom *Create Material* UE će za nas automatski generirati novi materijal koji za teksturu ima zadanu odabranu sliku.



Slika 4.4: *Content browser* nakon dodane slike i generiranog *Material*-a

Unutar *Details* panela i kategorije *Materials* klikom na padajući izbornik možemo odabrati naš materijal. Preporuča se iz `Material`-a kreirati instancu `Material`-a u kojoj je dopušteno mijenjati samo vrijednosti varijabla koje definiramo u `Material` klasi. Tako je postignuto brže vrijeme kompiliranja materijala. U našem primjeru preskočili smo taj korak zbog jednostavnosti našeg materijala.

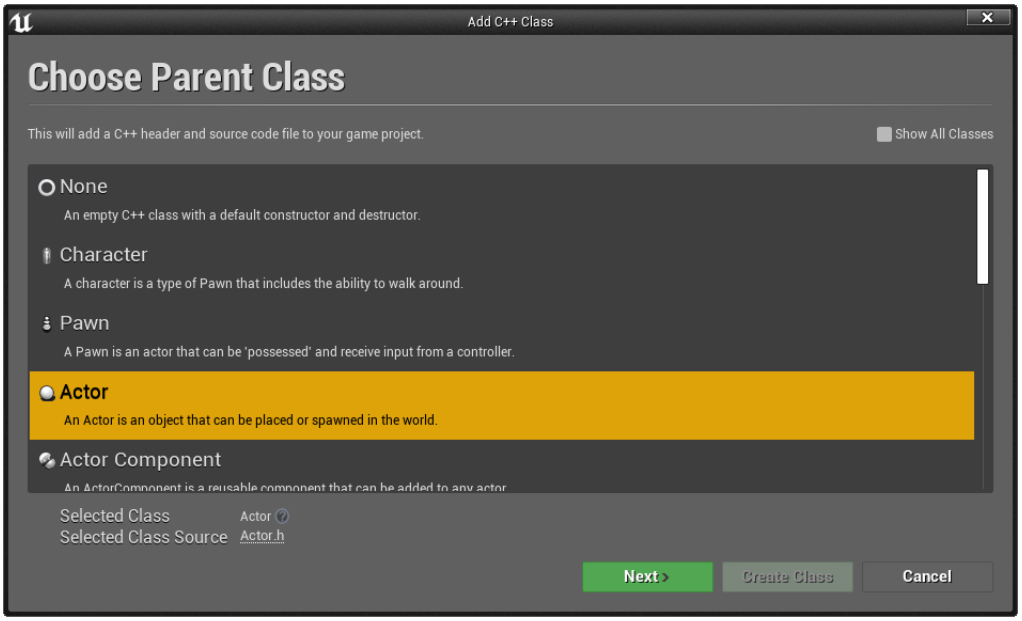


Slika 4.5: *Cube* s novim *Brick_Material* materijalom

4.4 UCLASS, UFUNCTION, UPROPERTY

U ovom poglavlju opisat ćemo detaljnije klase i strukture Unreal Engine-a. Novu klasu možemo dodati tako da u *Content browser*-u otvorimo mapu *C++ Classes* te unutar mape odaberemo *Desni klik* → *New C++ Class* opciju.

U novootvorenom prozoru možemo odabrati baznu klasu za našu novu klasu. Već smo se susreli s klasom `Actor`. Ona predstavlja sve objekte koje možemo staviti u scenu. Iz nje su izvedene klase `Pawn` i `Character`. `ActorComponent` je klasa koju možemo dodati objektu klase `Actor` i ona ne sadrži *transform* komponentu pa tako ne sadrži ni fizičku lokaciju ni rotaciju u našoj sceni. Ukoliko označimo *Show All Classes* opciju prikazat će nam se traka za pretraživanje gdje možemo potražiti i odabrati naše prijašnje kreirane klase. Odabirom opcije *None* generirat će se prazna klasa koju kasnije možemo koristiti u drugim UE klasama, ali ju ne možemo postaviti u scenu niti će se pojaviti u *Content Browser*-u što može biti zbunjujuće.



Slika 4.6: Prozor s opcijama za novu klasu

Sve navedene klase izvedene su iz klase `Object` koja predstavlja glavnu klasu u UE hijerarhiji. Proučavanjem *header* dokumenata izvedenih podklasa `Object`-a lako je uočiti da se nazivi klasa razlikuju od imena koje smo unijeli prilikom kreacije klase. Radi se o nomenklaturi koju UE slijedi za podklase *Object* klase. Tako je klasama izvedenim iz *Object* klase dodan prefixs **U** (npr. `UObject`), dok je klasama izvedenim iz *Actor* klase dodan prefiks **A** (npr. `AActor`). Od sada nadalje mi ćemo se također pridržavat tog pravila.

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "TestActor.generated.h"

UCLASS()
class TESTNIPROJEKT_API ATestActor : public AActor
{
  GENERATED_BODY()

public:
  // Sets default values for this actor's properties
  ATestActor();

protected:
  // Called when the game starts or when spawned
  virtual void BeginPlay() override;
```

```
public:
  // Called every frame
  virtual void Tick(float DeltaTime) override;

};
```

Generirane klase razlikuju se od klasičnih klasa u *C++*-u. UE prilikom kreiranja klasa umjesto nas ubaci razne potrebne *macro*-e koji generiraju kod potreban za ispravno funkcioniranje Unreal Engine-a. Na vrh definicije dodan je makro *UCLASS* koji generira kod unutar **.generated.h* zaglavlja koji se može pronaći kod svake podklase `UObject`-a. Bitno je napomenuti da za ispravni rad programa makro *#include "*.generated.h"* mora biti zadnji u nizu naredbi inače će doći do javljanja greške.

`UCLASS` makrou moguće je proslijediti argumente koji određuju način na koji će se klasa ponašati. *Blueprintable* omogućit će nam kreiranje *Blueprint* klase desnim klikom na klasu u *Unreal Editor*-u i odabirom opcije. *Blueprint* klase moguće je modificirati, proširivati i dodavati nove komponente i funkcionalnosti preko *Blueprint Visual Scripting*-a (**BVS**) unutar *Unreal Engine*-a bez korištenja "tekstualnog" koda. Klase izvedene iz klasa sa svojstvom *Blueprintable* naslijedit će dano svojstvo. *NotBlueprintable* onemogućit će kreiranje *Blueprint* klase. Klase sa svojstvom *BlueprintType* moguće je koristiti kao varijable unutar nekog drugog *Blueprint*-a. Obratno vrijedi za *NotBlueprintType*.

Pomoću *UPROPERTY* makroa odredit ćemo svojstva i ponašanja varijabli članica klase ili strukture prosljeđivanjem argumenata u poziv. *EditAnywhere* dozvolit će nam uređivanje vrijednosti varijable i iz *Unreal Editor*-a direktno na klasi i na instancama klase unutar scene. Ukoliko želimo ograničiti opisano svojstvo samo na editor koristit ćemo *EditDefaultsOnly* dok će *EditInstanceOnly* dozvoliti samo promjene na instancama klase. Varijable sa *BlueprintReadWrite* svojstvom moguće je koristiti unutar *Blueprint* skripte. Bitno je naglasiti da varijabla mora imati *public* pristup inače će doći do greške. *BlueprintReadOnly* dopustit će jedino pristup varijabli. Mijenjanje vrijednosti varijable iz *Blueprint*-a neće biti dozvoljeno. Privatne varijable moguće je izložiti editoru kombiniranjem *meta* i *AllowPrivateAccess* svojstava.

UFUNCTION makro koristi se kako bi funkcije klase mogli koristiti u *Blueprint*-ima. Funkcije s *BlueprintCallable* svojstvom moći ćemo pozvati iz *Blueprint* skripte. *BlueprintImplementableEvent* svojstvo dozvoljava nam da tijelo funkcije definiramo unutar *Blueprint*-a te ju kasnije možemo pozivat u *C++*-u kodu.

Primjer kreiranja *Blueprint* klase

Klasu `ATestActor` proširit ćemo s varijablama i funkcijama s gore opisanim svojstvima.

```
// TestActor.h
```

```
UCLASS(Blueprintable)
class TESTNIPROJEKT_API ATestActor : public AActor
{
  GENERATED_BODY()

public:
  // Sets default values for this actor's properties
  ATestActor();

protected:
  // Called when the game starts or when spawned
  virtual void BeginPlay() override;

public:
  // Called every frame
  virtual void Tick(float DeltaTime) override;

  UFUNCTION(BlueprintImplementableEvent)
  void ChangeNumberToValue(int value);


  UFUNCTION(BlueprintCallable)
  void PrintNumber();

public:
  UPROPERTY(BlueprintReadWrite)
  int Number;
};

// TestActor.cpp
void ATestActor::PrintNumber()
{
  UE_LOG(LogTemp, Display, TEXT("Broj: %d"), Number);
}
```

UE_LOG makro ispisat će poruku u *Output Log* koji se može otvoriti odabirom *Window* → *DeveloperTools* → *OutputLog* na *Menu* traci.

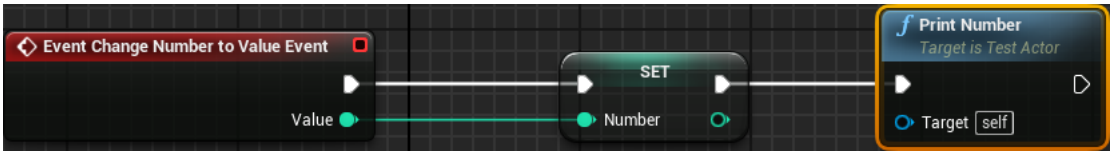
Odabirom *Window* → *DeveloperTools* → *ClassViewer* otvorit će se prozor sa svim klasama koje koristimo unutar editora. Koristeći traku pretraživanja lako je pronaći našu `ATestActor` klasu. *Desni klik* → *CreateBlueprintClass* kreirat će `BP_TestActor` *Blueprint* klasu baziranu na `ATestActor`-u te otvoriti *Blueprint Editor*

Slika 4.7: *Class Viewer* prozor

Desnim klikom na *Graph Editor* otvorit će se izbornik u kojem možemo pronaći našu funkciju `ChangeNumberToValueEvent` klase `ATestActor`. Odabirom će se kreirati čvor u našoj *Blueprint* skripti. Čvor se sastoji od ulaznih i izlaznih utora. Trokutasti utor označava početak niti koja označava redoslijed izvođenja naredbi u grafu. Ukoliko kliknemo na njega i nastavimo držati klik, stvorit će se nit koju možemo spojiti u ulaz nekog drugog čvora ili na proizvoljnom mjestu pustiti klik što će otvoriti izbornik za dodavanje nove naredbe. Čvorovi će biti spojeni te će nakon izvođenja prvog čvora biti izvedena naredba drugog čvora. Utor `Value` označava naš ulazni argument koji smo definirali u *header*-u klase. Klikom na njega također možemo kreirati nit koju kasnije možemo spojiti s ulaznim utorom nekog drugog čvora. U ovom slučaju to će predstavljati pridruživanje vrijednosti varijable.

Slika 4.8: Primjer čvora unutar *Blueprint* skripte

U funkciji želimo promijeniti vrijednost varijable `Number` te na kraju pozvati funkciju `PrintNumber`. *Blueprint*-i nam nude *getter*-e i *setter*-e za varijable koje smo izložili editoru pomoću makroa *UPROPERTY*. Sada nam je jedino još preostalo pozvati funkciju iz našeg *C++* koda te kompilirati kod pritiskom na *Compile* gumb koji se nalazi na *alatnoj traci*. Korištenje tekstualnog i vizualnog programiranja ubrzava i olakšava proces razvoja igara i stoga je dobra praksa koristiti ih oboje.



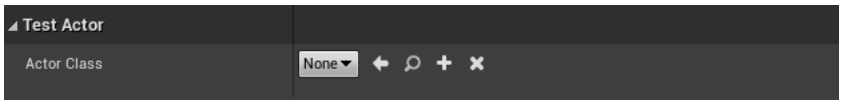
Slika 4.9: Primjer grafa funkcije

4.5 TSubclassOf, SpawnActor i CreateDefaultSubobject

Znamo kako dodati objekt u scenu preko *Place Actor* panela, međutim željeli bismo to napraviti dinamički kroz kod i inicijalizirati kreirani objekt s konkretnim vrijednostima. Ukoliko želimo instancirati objekt *Blueprint* klase naići ćemo na problem jer ne možemo dobiti tip te klase, tj. ne možemo s `#include` uključiti zaglavlje *Blueprint* klase te zbog toga *compiler* neće znati za tu klasu.

`TSubclassOf` dozvoljava nam da spremimo tip klase u varijablu. Specificiranjem predloška funkcije će restringirati odabir na samo one klase koju su podklase klase koju smo postavili za predložak. Dodavanjem *EditDefaultsOnly* svojstva *UPROPERTY*-u izložit ćemo našu `TSubclassOf` u *Details* panelu *Blueprint* klase koja će biti izvedena iz *C++* klase kojoj varijabla pripada. U *Details* panelu, unutar *Blueprint Editor*-a, pojavit će se opcija za padajući izbornik uz ime varijable. Klikom na izbornik ponudit će nam se sve opcije koje možemo postaviti kao vrijednost varijable, odnosno sve podklase klase koju smo predodredili u tipu varijable.

```
// TestActor.h
UPROPERTY(EditDefaultsOnly)
TSubclassOf<AActor> ActorClass;
```



Slika 4.10: Primjer `TSubclassOf` varijable

Funkcija `SpawnActor` stvorit će novi objekt klase koju smo mu proslijedili kao argument, a čiji tip smo spremili u varijablu tipa `TSubclassOf`. `FVector` predstavlja 3-dimenzionalni vektor u koji možemo spremiti lokaciju našeg `Actor`-a, dok ćemo `FRotator` koristiti za reprezentaciju rotacije objekta. Osi oko kojih rotiramo objekt imaju drugačije nazive u *Unreal Editor*-u. **Roll**, **Pitch** i **Yaw** nazivi su za X,Y i Z osi. Kao povratnu vrijednost funkcija

`SpawnActor` vratit će `AActor` pokazivač na instancirani objekt. Pozivom `Cast` funkcije *Unreal Engine*-a možemo dobiti pokazivač na tip naše klase. Funkcija `GetWorld` vratit će nam scenu kojoj naš `Actor` pripada.

```
// TestActor.h
UPROPERTY(EditDefaultsOnly)
TSubclassOf<AActor> ActorClass;

UPROPERTY()
AActor* NewActor;

// TestActor.cpp
FVector position(0, 0, 0);
FRotator rotation(0, 0, 0);

NewActor
    = GetWorld()->SpawnActor(ActorClass, &position, &rotation);
```

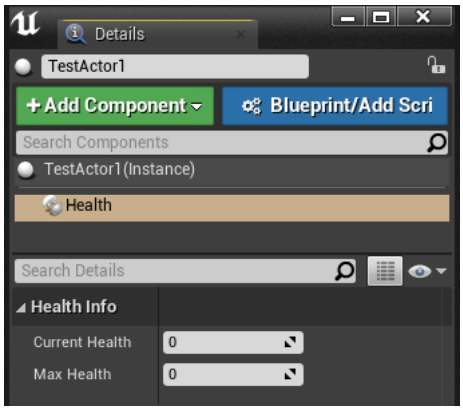
Primjetimo da u primjeru nismo koristili pametne pokazivače što može dovesti do kršenja **RAII** principa o kojem smo pisali u prošlom poglavlju. Dodavanjem makroa *UPROPERTY* istaknut ćemo *Unreal Editor*-u da *Garbage Collector Unreal Engine*-a "počisti" i oslobodi memoriju koju je zauzeo objekt na koga je postavljen pokazivač. To nam osigurava da nećemo prekršiti **RAII** princip.

Kreirajmo novu komponentu `UHealthComponent` unutar *Content Browser*-a. Za baznu klasu odabrat ćemo `UActorComponent`. Klasa `UHealthComponent` sadržavat će dvije `int` varijable `MaxHealth` i `CurrentHealth`. Željeli bismo da opisana komponenta reprezentira život (eng. *health*) našeg `ATestActor`-a. Svaki `AActor` može sadržavati više komponenti tipa `UActorComponent`. Jedan od načina na koji možemo dodati komponentu `AActor`-u je preko *Details* panela. Prvo trebamo označiti objekt kojem želimo pridružiti komponentu klikom na njega u *Viewport*-u ili i u *World Outliner*-u. U *Details* panelu pojavit će nam se opcija *Add Component* gdje ćemo odabrati naše novo napravljenu komponentu.

Svojstvo *Category UPROPERTY* makroa grupirat će nam sve varijable s jednakim ključem pod istom karticom u *Details* panelu.

```
// HealthComponent.h
class TESTNIPROJEKT_API UHealthComponent : public UActorComponent
{
    // ...
public:
    UPROPERTY(EditInstanceOnly, Category = "Health␣Info")
  int CurrentHealth;
  UPROPERTY(EditInstanceOnly, Category = "Health␣Info")
  int MaxHealth;
```

```
};
```



Slika 4.11: Detalji *HealthComponent*-e

Drugi način za dodati novu komponentu je kroz kod. Funkcija `CreateDefaultSubobject` kreirat će nam objekt željene komponente i vratiti ga kao povratnu vrijednost koju onda možemo pridružiti željenom `AActor` objektu. U pozivu funkcije proslijedit ćemo ime komponente preko varijable tipa `FString`.

`FString` je *Unreal Engine* klasa koja zamjenjuje `std::string` i nudi još mnoge funkcije koje olakšavaju rad sa stringovima. `FString` smo koristili kod ispisivanja poruke u `Output` prozor sa `UE_LOG` makro-om. Pozivom nadjačanog operatora dereferenciranja (*) dobit ćemo tekst koji smo spremili u varijablu tipa `FString`. Lako je napraviti grešku jer smo navikli pri ispisu stringa proslijediti kopiju ili referencu objekta, međutim u UE potrebno je svugdje gdje se za argument traži neka vrsta niza `char`-ova proslijediti dereferencirani `FString`. U protivnom će doći do greške prilikom kompiliranja programa.

Novu komponentu koji smo dobili od `CreateDefaultSubobject` možemo pridružiti `AActor`-u pozivom `AttachToComponent` metode nad kreiranom komponentom. U našem primjeru komponentu ćemo pridružiti *RootComponent*-u, od `ATestActor`-a, koji predstavlja korijen hijerarhijskog stabla komponenti.

```
// TestActor.h
#include "HealthComponent.h"

UCLASS(Blueprintable)
class TESTNIPROJEKT_API ATestActor : public AActor
{
    // ...
    void AddHealthComponent();
};
```

```
// TestActor.cpp
void ATestActor::AddHealthComponent()
{
  UHealthComponent* healthComponent =
      CreateDefaultSubobject<UHealthComponent>(TEXT("Health"));

    healthComponent->AttachToComponent(RootComponent,
        FAttachmentTransformRules::KeepWorldTransform);
}
```

4.6 Životni ciklus (eng. *lifecycle*) `AActor`-a

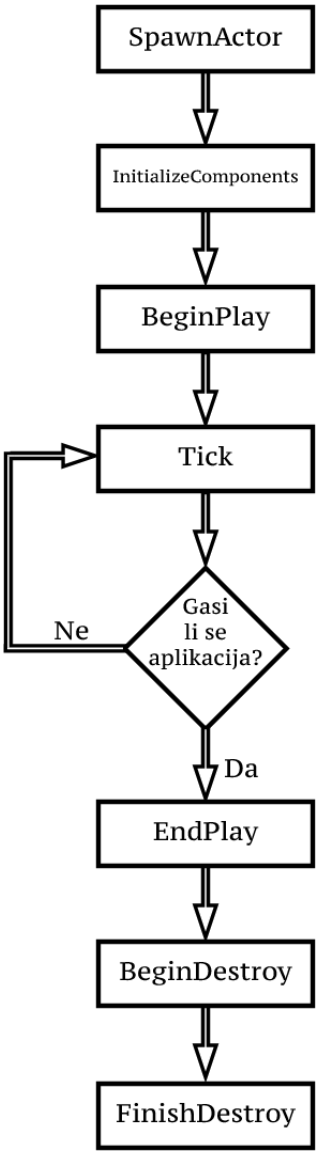
Svaka automatski izgenerirana klasa od strane *Unreal Engine*-a sadržavat će nadjačane *virtual* metode `BeginPlay` i `Tick`. To su samo jedne od mnogo metoda i funkcija koje čine životni ciklus (eng. *lifecycle*) nekog objekta `AActor` klase. `Lifecycle` je redoslijed poziva metoda koji započinje kreiranjem objekta, nastavlja njegovim postojanjem u sceni i završava destrukcijom objekta.

Slika 4.12 prikazuje pojednostavljeni primjer *lifecycle*-a objekta kreiranog koristeći `SpawnActor` funkciju.

`BeginPlay` pozvat će se u trenutku kada se scena učita i započne igra u toj sceni. Bitno je dobro proučiti redoslijed poziva prije dizajniranja sustava jer mnoge stvari mogu poći po krivu iz razloga što je možda neka funkcija trebala biti pozvana ili prije ili poslije u *lifecycle*-u.

Svaki *game engine* zapravo radi tako da se u pozadini izvršava beskonačna petlja koja u svakoj iteraciji poziva razne metode i funkcije kao što su ažuriranje objekata ili iscrtavanje grafike. U trenutku kada aplikacija dobije signal za gašenje, izlazi se iz petlje te rad aplikacije završava. Glavna petlja *Unreal Engine*-a poziva metodu *EngineTick* koja za svaki *UObject*, za koji smo postavili zastavicu za `bCanEverTick` na istinitu vrijednost, u *Unreal Editor*-u pozove njihovu `Tick` metodu. Ukoliko ne želimo da se pozove `Tick` metoda za naš objekt u konstruktoru trebamo postaviti vrijednost zastavice `bCanEverTick` na `false`.

```
while( !IsEngineExitRequested() )
{
  EngineTick();
}

void EngineTick( void )
{
  GEngineLoop.Tick();
}
```



Slika 4.12: *Lifecycle* objekta

Pri kraju *lifecycle*-a objekta pozvat će se `EndPlay` te će nakon toga objekt biti označen za destrukciju. Tako će *garbage collector* znati da objekt treba uništiti te će to učiniti u svom sljedećem ciklusu.

Ovdje smo opisali, dosta pojednostavljen, životni ciklus koji započinje pozivom `SpawnActor` metode. Ipak, u stvarnosti se on sastoji od mnogo više metoda i funkcija. Navedene metode odabrane su jer se najčešće koriste. Detaljniji opis svih *lifecycle*-ova nekog `AActor`-a

može se pogledati u službenoj dokumentaciji.

4.7 UMG i `UUserWidget`

Unreal Motion Graphics UI Designer (**UMG**) je vizualni alat s kojim se mogu kreirati *user interface* (**UI**) elementi. Zahvaljujući **UMG**-u kreacija se svodi na izgradnju pomoću *drag and drop*-a i modificiranja vrijednosti unutar editora što uveliko ubrzava i olakšava kreaciju u usporedbi s prijašnjim načinom koji je zahtjevao da se cjelokupna izgradnja napravi u *C++* kodu. *Grafičke elemente* (eng. *widgete*), koje ćemo dodavati na ekran, izvodit ćemo iz bazne klase `UUserWidget`.

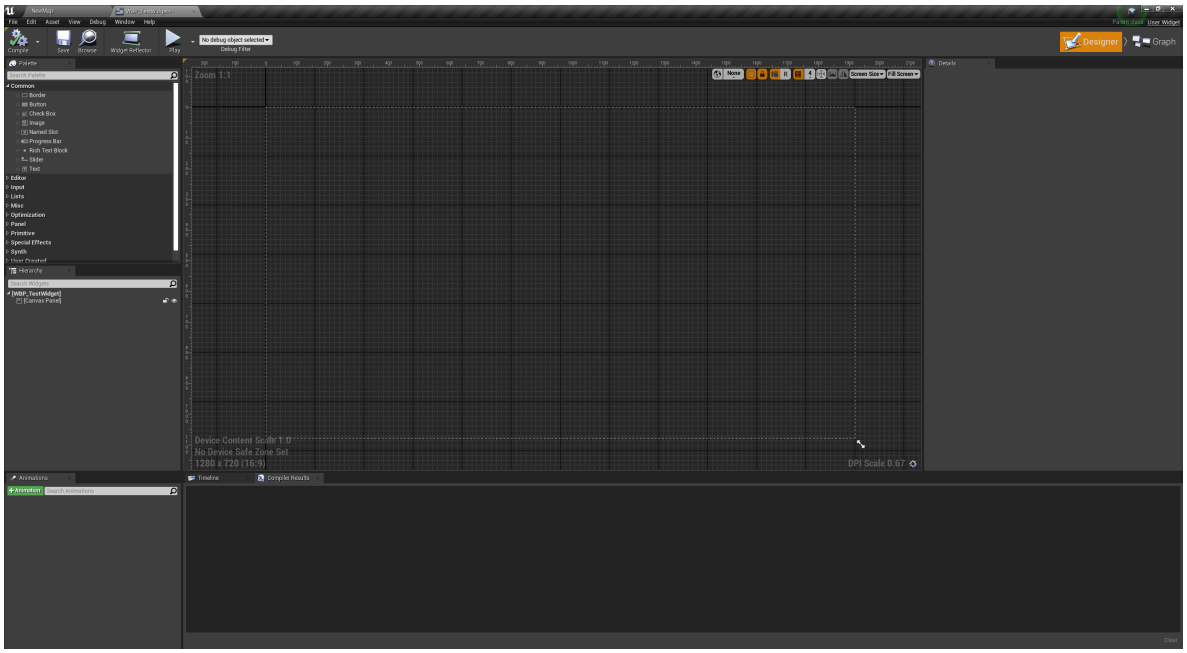
U sljedećem primjeru napravit ćemo *UI* koji će nam pokazivati kolika je vrijednost `CurrentHealth`-a našeg `AActor`-a. Kreirajmo prvo novu *C++* klasu `UHealthWidget`. Za baznu klasu odabrat ćemo `UUserWidget`. Naš *health bar* bit će dvije slike postavljene jedna preko druge. Prva slika služit će kao pozadina, dok će druga slika predstavljati *health*. Ovisno o količini *health*-a koja je preostala našem `AActor`-u mijenjat će se širina druge slike.

Pomoću klase `UImage` možemo kreirati sliku na našem *widget*-u, ali nažalost sučelje te klase nam ne nudi funkciju za promjenu veličine. Zbog toga ćemo našu `UImage` sliku postaviti kao dijete `USizeBox` objekta koji će kontrolirati veličinu svog djeteta, a nama nudi funkcionalnost za promjenu veličine.

Varijabli `FillSizeBox` dodijelit ćemo *meta = (BindWidget)* svojstvo preko *UPROPERTY* makroa. Editor će od nas zahtijevati da za sve varijable s navedenim svojstvom kreiramo objekt u našem *widget*-u istog tipa i istog imena. Tako će *Unreal Engine* moći sa svojim sustavom refleksijama povezati te dvije varijable i dozvolit nam modificiranje objekta koji se nalazi na *widget*-u unutar našeg *C++* koda. Klasu `UTextBlock` koristit ćemo za tekst ispod naših slika.

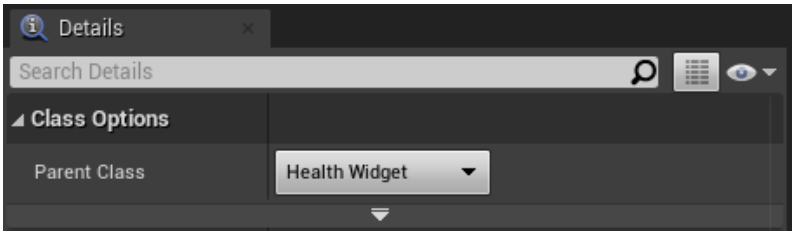
```
// HealthWidget.h
UCLASS()
class TESTNIPROJEKT_API UHealthWidget : public UUserWidget
{
  GENERATED_BODY()

public:
  UPROPERTY(meta = (BindWidget))
  class USizeBox* FillSizeBox;

  UPROPERTY(meta = (BindWidget))
  class UTextBlock* HealthTextBlock;
};
```

Kreirajmo sada novu *Widget blueprint* klasu izvedenu iz `UHealthWidget`-a unutar *Content Browser*-a koristeći *desni klik* → *User Interface* → *Widget Blueprint*. Taj odabir neće nam otvoriti prozor za odabir bazne klase pa ćemo to morati sami podesiti unutar editora.



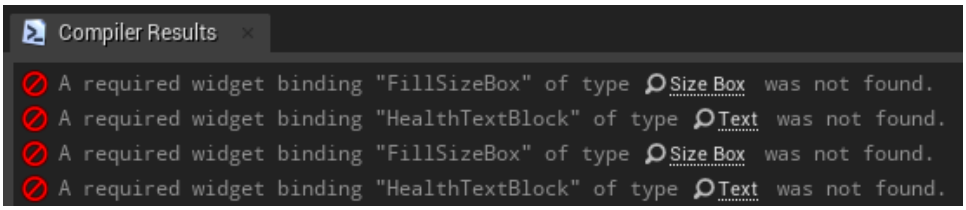
Slika 4.13: *UMG* editor

Baznu klasu možemo promijeniti klikom na *Graph* gumb u gornjem kutu. Otvorit će nam se editor s kojim smo se već upoznali kod *blueprints*-a. Klikom na *Class Settings* na alatnoj traci pojavit će se kategorija *Class options* unutar *Details* panela gdje možemo promijeniti baznu klasu. Ponuđene opcije bit će samo podklase `UUserWidget` klase.



Slika 4.14: *Details* panel

Promjenom bazne klase pojavit će nam se greške u *Compiler Results*. Kompilacija neće uspjeti dok god ne dodamo `FillSizeBox` i `HealthTextBlock` varijable u naš *widget*.



Slika 4.15: *Compiler Results* kartica

U *Designer* kartici s lijeve strane možemo pronaći *Palette* panel. U njemu možemo pronaći sve klase koje možemo staviti u naš *widget*, a koje dolaze s *Unreal Editorom*. Kada smo odabrali klasu, treba je još samo povući u *Hierarchy* karticu i time je dodati u naš *widget*. S desne strane pojavit će se *Details* panel u kojem možemo promijeniti ime varijable. Također, pored polja za unos teksta pronaći ćemo *Is Variable* opciju. Bitno je označiti je samo za one varijable kojima smo dodjelili svojstvo *BindWidget*, u protivnom može doći do greške u izvođenju programa.

Za `USizeBox` u *Details* panelu označit ćemo *Size To Content* opciju, a s *Width Override* mijenjat ćemo veličinu njegovog dijeteta. *Anchor* ćemo postaviti na vrijednost lijevo-sredina. Sada će se slika promjenom veličine proširiti s lijeva na desno.

U našu `ATestActor` klasu dodat ćemo novu `TSubclassOf` varijablu u koju ćemo spremiti klasu našeg `Widget Blueprint`-a. Ukoliko bismo kreirali objekt naše `UHealthWidget` klase na ekranu se ne bi ništa pojavilo. Varijable koje smo dodali u naš *Widget* pripadaju *blueprint* klasi. Stoga treba pripaziti i razlikovati *C++* klasu i *blueprint* klasu koja je izvedena iz nje.

U `BeginPlay` pozvat ćemo parametriziranu funkciju `CreateWidget`. Ona za prvi argument prima scenu kojoj ćemo dodat *widget* te vrstu klase. Kreirani *widget* spremit ćemo u varijablu `HealthWidget`. Na kraju, pozivom `AddToViewport` nad varijablom dodat ćemo ju na ekran. Ukoliko želimo imati više *widgeta* na ekranu, ali želimo određeni poredak jednog iza drugog, možemo proslijediti argument koji će postaviti dubinu *widget*-a.

```
// TestActor.h
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
TSubclassOf<UUserWidget> HealthWidgetClass;

class UHealthWidget* HealthWidget;

// TestActor.cpp
```

```
void ATestActor::BeginPlay()
{
  Super::BeginPlay();

  HealthWidget = CreateWidget<UHealthWidget>(GetWorld(),
      HealthWidgetClass);
  HealthWidget->AddToViewport();
}
```

Slika 4.16: Izgled našeg *widget*-a

Naš *health bar* ne bi imao smisla kada se ne bi ažurirao prilikom promjene vrijednosti `CurrentHealth`. U `HealthWidget` dodat ćemo metodu `OnHealthChanged` koja prima trenutnu vrijednost i maksimalnu vrijednost *health*-a našeg `AActor`-a. Ovisno o vrijednostima `UHealthComponent`-e, promijenit će se širina `FillSizeBox` varijable.

```
// HealthWidget.h
UCLASS()
class TESTNIPROJEKT_API UHealthWidget : public UUserWidget
{
  GENERATED_BODY()

public:

public:
  UPROPERTY(meta = (BindWidget))
  class USizeBox* FillSizeBox;

  UPROPERTY(meta = (BindWidget))
  class UTextBlock* HealthTextBlock;

  void Setup();
  void OnHealthChanged(float currentHealth, float maxHealth);

public:
  float MaxImageWidth;
};
```

```
// HealthWidget.cpp
void UHealthWidget::Setup()
{
  MaxImageWidth = FillSizeBox->WidthOverride;
}

void UHealthWidget::OnHealthChanged(float currentHealth,
    float maxHealth)
{
  FillSizeBox->SetWidthOverride((currentHealth / maxHealth)
      * MaxImageWidth);

  FString text(("Health: " + std::to_string(currentHealth)
      + "/" + std::to_string(maxHealth)).c_str());
  HealthTextBlock->SetText(FText::FromString(text));
}
```

Poglavlje 5

Match3

U prethodnom poglavlju, kroz jednostavne primjere, opisali smo neke osnovne koncepte *Unreal Engine*-a s kojima možemo započeti rad na većem i kompleksnijem projektu. U ovom poglavlju bit će opisani još neki alati, karakteristični za žarn igre, koji su korišteni u video igri izrađenoj za ovaj diplomski rad.

5.1 Općenito

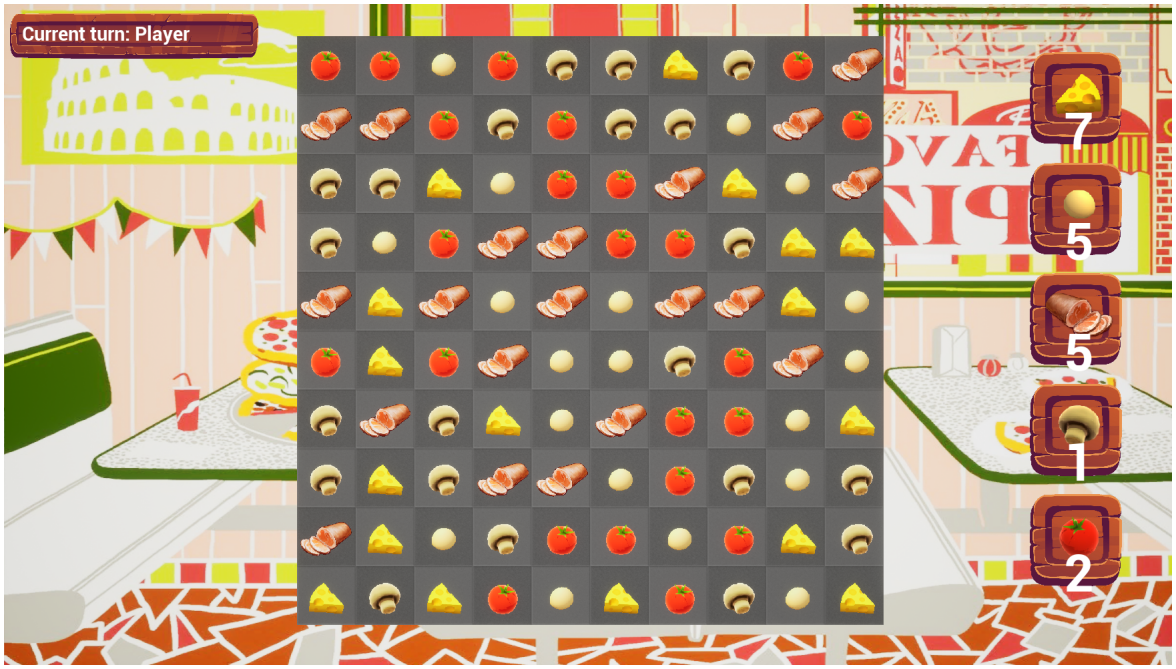
Match3 je oduvijek bio jedan od najigranijih žanrova na tržištu mobilnih igara. Njegove jednostavne mehanike i pravila privlače veliki broj igrača. Cilj u ovom tipu igara je upariti figure istog tipa u niz ili lanac duljine 3 ili više, ovisno o pravilima. Uparene figure čine *combo* koji igraču donosi bodove. Nakon toga figure se uništavaju, a figure iznad uništenih i nove figure, koje su se stvorile umjesto uništenih, spuštaju se i popunjavaju prazninu na ploči.

Uparivanjem 4 ili više figure stvorit će se posebne figure koje imaju veću razornu moć. Igraču je najčešće dodijeljen zadatak sakupljanja dovoljnog broja bodova u određenom vremenu ili razbijanja svih pločica koje se nalaze na ploči ispod figura.

Ispunjavanjem zadatka igrač otključava nove nivoe u kojima su prepreke s kojima se igrač mora suočiti sve zahtjevnije, a samim time je i nivo teži. Izgled igrače ploče na početku nivoa uvijek je isti, ali položaj figurica na ploči je drugačiji čime se dobiva varijabilnost (eng. *replayability*) i umanjuje se osjećaj monotonosti. Igrač najčešće ima 5 života koji se na kasnijim nivoima lagano isprazne, ali se mogu napuniti kupnjom koristeći pravi novac. Jednostavnost izrade igre ovakvog žanra te ogroman prostor za monetizaciju razlozi su zašto je tržište mobilnih igara preplavljeno ovakvim žanrom igara.

Tema naše video igre bit će *pizza*. Figure koje će igra sadržavati su sljedeće: sir, šunka, tijesto, rajčica i gljive. Svaki igrač će na početku igre dobiti određeni broj figura koje mora skupiti njihovim razbijenjem na ploči. Pobjednik je onaj igrač koji prvi skupi sve figurice.

Ulaskom u aplikaciju igrač otvara konekciju sa serverom. U glavnom izborniku igrač može odabrati opciju nove igre gdje će biti uparen s novim igračem.



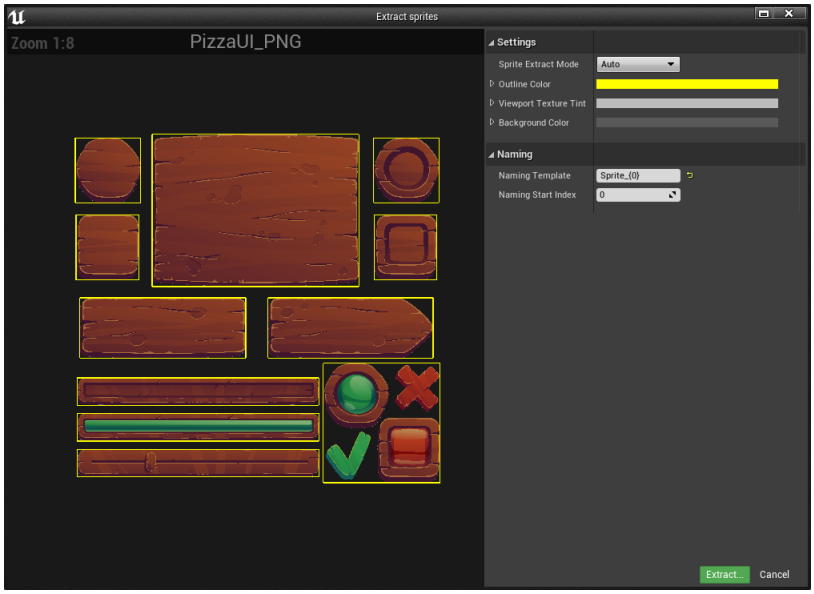
Slika 5.1: Match3 Pizza video igra

Paper2D

Paper2D je sistem koji dolazi s *Unreal Engine*-om, a temelji se na *sprite*-ovima. Koristeći njega možemo kreirat 2D i hibride 2D i 3D igara. *Unreal Editor* dolazi sa *Sprite Editor*-om u kojem možemo uređivati naše *sprite*-ove ili raditi *Flipbook* animacije. *Flipbook* su animacije temeljene na *sprite*-ovima. *Sprite*-ovima dodjelimo indeks u nizu i vremenski okvir koliko će vremena u animaciji *sprite* trajati.

Kada dodamo sliku (ovdje su korištene slike formata **PNG** radi transparentnosti) u naš projekt, desnim klikom na nju možemo odabrati opciju *Sprite Actions* → *Apply Paper2D Texture Settings* koja će za nas obraditi sliku i pripremiti je za rad u *Paper2D* okruženju. Opcija *Extract Sprites* automatskim generiranjem rastavit će jednu sliku na više *sprite*-ova ovisno o kriteriju (u našem slučaju transparentnost). Zahvaljujući toj opciji ne moramo imati više tekstura nego možemo sve zapakirati u jednu veliku čime štedimo na memoriji.

Figurice ćemo reprezentirati `ATile` klasom koju ćemo izvesti iz `APaperSpriteActor` klase koja osim svojstava `AActor`-a implementira i funkcionalost potrebnu za korištenje



Slika 5.2: *Sprite Extract* prozor

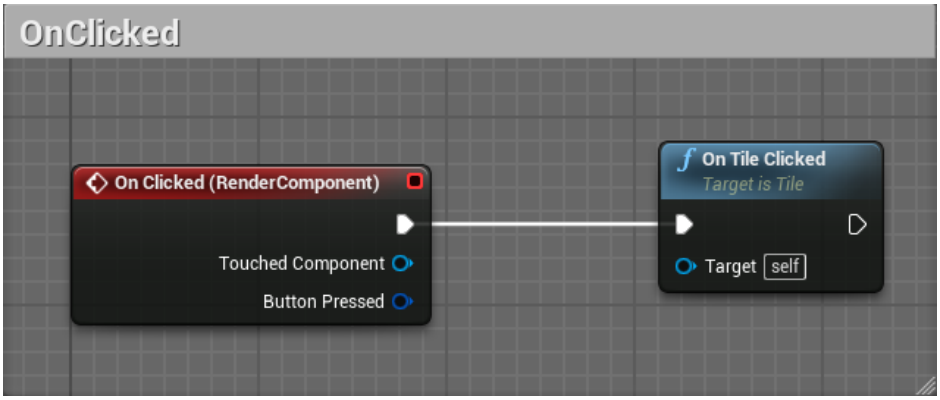
Paper2D. Možemo zamisliti ovu klasu kao 2D `AActor`-a. Svaki tile sadrži `FVector` u kojem je spremljena jedinstvena pozicija na ploči i `int` broj kojim identificiramo tip figure. Za svaku sliku, koju ćemo koristiti za figuru i koju smo dodali u projekt, napravit ćemo `UPaperSprite` klasu u *Content Browser*-u. Napravljene objekte referencirat ćemo u `ATile` klasi te ćemo ih, ovisno o tipu figure, postaviti kao teksturu. Pozivom metode `GetRenderComponent` dohvatit ćemo komponentu koja je zadužena za iscrtavanje grafike i koja sadrži metodu `SetSprite` s kojom ćemo mijenjati *sprite* koji će se iscrtati.

```
// Tile.h
enum class ETileType
{
    Cheese,
    Dough,
    Ham,
    Mushroom,
    Tomato
}

UCLASS()
class MATCH3_API ATile : public APaperSpriteActor
{
  GENERATED_BODY()
public:
```

```
    // ...

  void ChangeTileSprite();

    UFUNCTION(BlueprintCallable)
  void OnTileClicked();

  void MoveTile(FVector newLocation);

public:
  UPROPERTY(EditDefaultsOnly)
  std::map<ETileType, UPaperSprite*> SpritesMap;

private:
    AGrid* Grid;

  FVector BoardPosition;
    ETileType TileType;
};

// Tile.cpp
void ATile::ChangeTileSprite()
{
    GetRenderComponent()->SetSprite(SpritesMap[TileType]);
}

void ATile::OnTileClicked()
{
  // Poziv funkcije iz AGrid klase
  // koju cemo definirat u nastavku
  Grid->OnTileClicked(this);
}

void ATile::MoveTile(FVector newLocation)
{
  SetActorLocation(newLocation);
}
```

Unreal Engine za nas obrađuje ulazne događaje od strane kontrolera. Kako bismo pretplatili neku funkciju na neki događaj trebamo napraviti *blueprint* klasu iz naše klase. Kada se događaj na koju je funkcija pretplaćena dogodi, pozvat ćemo tu funkciju i sve funkcije koje su se pretplatile, Unutar *blueprint* klase možemo pronaći `OnClicked` čvor *RenderComponent*-e iz kojeg onda možemo pozvati našu `OnTileClicked` metodu.

Klasa `AGrid` brine se za stvaranje i uništavanje figura na ploči te njihovo pomicanje po

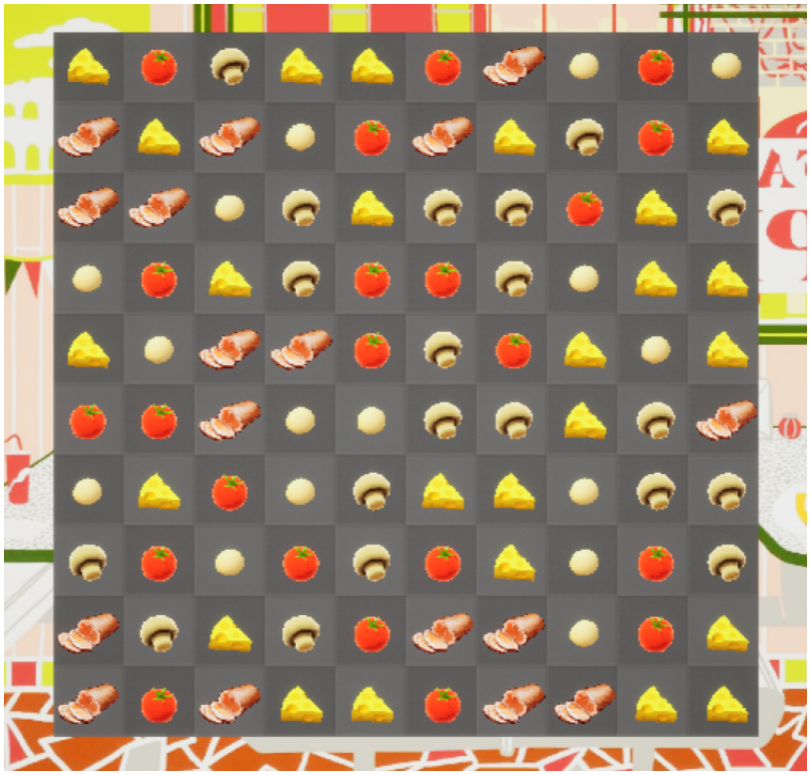


Slika 5.3: OnClicked event

ploči. Pozivom `CreateBoard` metode stvorit ćemo ploču za igranje čiji je raspored figura generirao server te poslao uparenim igračima. Pritiskom lijevog klika na figuru označit ćemo je, a zatim pritiskom na neku drugu susjednu figuru provjerit će se je li potez valjan te ukoliko je, figure će se zamijeniti. Uparivanjem 3 ili više figurica istog tipa napravili smo *Combo* i dolazi do uništavanja figura.

```
// Grid.h
UCLASS()
class MATCH3_API AGrid : public AActor
{
  GENERATED_BODY()
public:
    // ...
  void CreateBoard(std::array<std::array<int, 10>, 10> board);

public:
  UPROPERTY(EditDefaultsOnly)
  TSubclassOf<ATile> TileClass;

    UPROPERTY(EditAnywhere)
  FVector StartingPosition;

private:
  std::vector<ATile*> Tiles;

  FVector TileSize;
};

// Grid.cpp
```

```
void AGrid::CreateBoard(std::array<std::array<int, 10>, 10> board)
{
  FRotator rotation(0, 0, 0);
  for (int i = 0; i < 10; ++i)
  {
    for (int j = 0; j < 10; ++j)
    {
      FVector position(StartingPosition.x - i * TileSize.x,
          StartingPosition.y + j * TileSize.y, 50);

      ATile* tile = Cast<ATile>(GetWorld()
          ->SpawnActor(TileClass, &position, &Rotation));
      tile->AttachToActor(this,
          FAttachmentTransformRules::KeepWorldTransform);

        // ...

      Tiles.push_back(tile);
    }
  }
}
```



Slika 5.4: Ploča i figure

5.2 Socket

Već smo spomenuli da je ovo igra za 2 igrača koji komuniciraju preko servera. Ukoliko jedan igrač napravi ispravan potez, drugi igrač mora biti obaviješten i njihove ploče moraju biti identične nakon svakog poteza. U trenutku kada igrač napravi potez, napravit ćemo `Request` koji će sadržavati koordinate figura koje želimo zamijeniti te ga poslati prema serveru. Server će obraditi `Request` te odgovoriti klijentu koji je poslao poruku i poslati novu poruku drugom klijentu da ga obavijesti da je prvi igrač napravio potez.

```
Connection_1: Requst(TileMove): {"data" : {"firstX" : 5, "firstY" : 3, "secondX" : 5, "secondY" : 4, },
 "identifier" : "61B0AC8A44853C9AE5B8E88138297586", "requestType" : 3, "timestamp" : 35, }
```

Slika 5.5: Primjer *TileMove request*-a

Da bismo mogli otvoriti konekciju sa serverom i razmijenjivati poruke trebamo stvoriti *socket*. *Unreal Engine* nudi klasu `FTcpSocketBuilder` koja će za nas stvoriti socket ovisno o platformi na kojoj se aplikacija izvršava i otvoriti konekciju s proslijeđenom adresom pozivom `connect` metode. Naša `Socket` klasa temelji se na *Singleton* oblikovnom obrascu te je dostupna bilo gdje u kodu pozivom njegove statičke funkcije `GetInstance`.

```
// Socket.h
class MATCH3_API Socket
{
public:
  static Socket& GetInstance()
  {
    static Socket instance;

    return instance;
  }

  bool Connect(FString ipAddress, int port);
  void Close();

  void SendMessage(std::string message);

    // ...

private:
  Socket() {};

private:
  std::unique_ptr<FSocket> MySocket;
};

// Socket.cpp
bool Socket::Connect(FString ipAddress, int port)
{
    bool success = true;

  FIPv4Address ipv4Address;
  FIPv4Address::Parse(FString("127.0.0.1"), ipv4Address);
  FIPv4Endpoint endpoint(ipv4Address, (uint16)1234);

  MySocket = std::unique_ptr<FSocket>
      (FTcpSocketBuilder(TEXT("TcpSocket")).AsNonBlocking()
          .AsReusable());
```

```
  ISocketSubsystem* socketSubsystem
      = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM);

  TSharedRef<FInternetAddr> internetAddress
      = socketSubsystem->CreateInternetAddr();
  internetAddress->SetIp(endpoint.Address.Value);
  internetAddress->SetPort(endpoint.Port);

  MySocket->Connect(*internetAddress);

    return success;
}
```

Poruku možemo poslati koristeći `Send` funkciju koja prima niz *byte*-ova i veličinu niza u *byte*-ovima kao arguemnte. Serijalizacijom `Request` objekta možemo poslati željenu poruku prema serveru.

```
void Socket::SendMessage(std::string message)
{

  FString logMsg(message.c_str());
  UE_LOG(LogTemp, Warning, TEXT("SALJEM: %s"), *logMsg);

  const uint8* byteMessage
      = reinterpret_cast<const uint8_t*>(message.c_str());

  int bytesSent = 0;
  _socket->Send(byteMessage,
      message.size() * sizeof(uint8_t), bytesSent);
}
```

5.3 `ResponseEventSubscription` klasa

Pretpostavimo da smo pronašli na ploči potencijalni niz od 3 ili više figurica. Imamo funkcionalnost klikanja na figure, ali sada im želimo zamijeniti pozicije na ploči. Nakon što smo odabrali dvije figurice koje želimo zamijeniti izvršit će se provjera ispravnosti poteza i u uspješnom scenariju klijent će poslati serveru *MoveTiles* `Request`. Oblikovni obrazac poput *Observer*-a ovdje bi se odlično uklopio u primjeru pozivanja pretplaćenih funkcija u trenutku kada klijent primi poruku od servera. U projektu je implementiran sličan sustav u obliku `ResponseEventSubscription` klase. Objekt neke klase koji želi osluškivati željeni `Response` mora se na njega pretplatiti. Pretplate čuvamo u varijabli tipa `std::map<ResponseType, std::vector<ResponseEventSubscription::Ptr>>` u

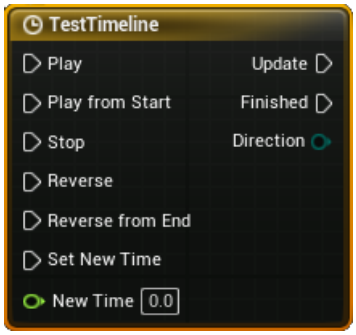
klasi za koju smo odlučili da će rukovoditi pretplatama. U trenutku kada dođe `Response` s tom vrijednosti `ResponseType`-a pozvat će se `Callback` funkcije od svih pretplaćenih objekata.

```
// Primjer
// ResponseEventSubscription.h
class MATCH3_API ResponseEventSubscription
{
public:
  ResponseEventSubscription(ResponseType type,
      std::function<void(JSONObjectNode::Ptr)> callback)
          : Callback(callback), Type(type) {};

  typedef std::shared_ptr<ResponseEventSubscription> Ptr;

public:
  std::function<void(JSONObjectNode::Ptr)> Callback;
  ResponseType Type;
};

// SubscriptionHandler.h
class SubscriptionHandler
{
public:
    static ResponseEventSubscription::Ptr
        SubscribeToResponseEvent (ResponseType type,
            std::function<void(JSONObjectNode::Ptr)> callback);

public:
    static std::map<ResponseType,
        std::vector<ResponseEventSubscription::Ptr>>
            ResponseEventSubscribers;
}

// SubscriptionHandler.cpp
ResponseEventSubscription::Ptr
    EventQueue::SubscribeToResponseEvent(ResponseType type,
        std::function<void(JSONObjectNode::Ptr)> callback)
{
  ResponseEventSubscription::Ptr subscription
      = std::make_shared<ResponseEventSubscription>
          (type, callback);

  if (!ResponseEventSubscribers.count(type))
```

```
  {
    ResponseEventSubscribers[type]
        = std::vector<ResponseEventSubscription::Ptr>();
  }
  ResponseEventSubscribers[type].push_back(subscription);

  return subscription;
}
// Test.h
#include "ResponseEventSubscription.h"

class TestClass
{
public:
    TestClass()
    {
        TestSubscription
            = SubscriptionHandler::SubscribeToResponseEvent(
                ResponseType::TestResponse,
                    std::bind(&TestClass::OnResponse, this,
                        std::placeholders::_1));
    }

    void OnResponse(JSONObjectNode::Ptr response)
    {
        UE_LOG(LogTemp, Warning, TEXT("Dobio sam response!");
    }

public:
    ResponseEventSubscription::Ptr TestSubscription;
};
```

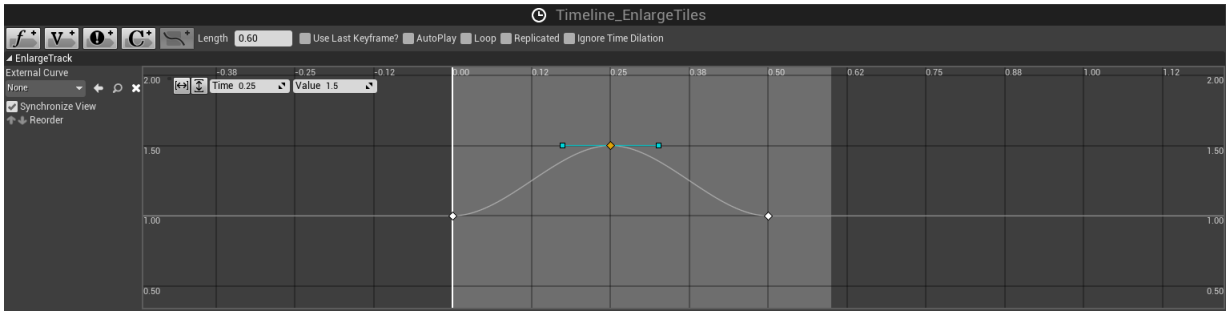
5.4 `Timeline` i primjer animacije

`Timeline` čvorovi su posebni čvorovi unutar *blueprint* sistema koji pružaju podršku za brzo i jednostavno kreiranje animacija. Sustav se temelji na grafu čija $X - os$ predstavlja vrijeme, a $Y - os$ vrijednost koja ovisi o vremenu. Neke od često korištenih opcija za graf su *float*, *vector* i *color*. Ovisno o opciji za povratnu vrijednost dobit ćemo vrijednost, 1−dimenzionalnu ili više dimenzionalnu, u tom trenutku u vremenu.



Slika 5.6: `Timeline` čvor

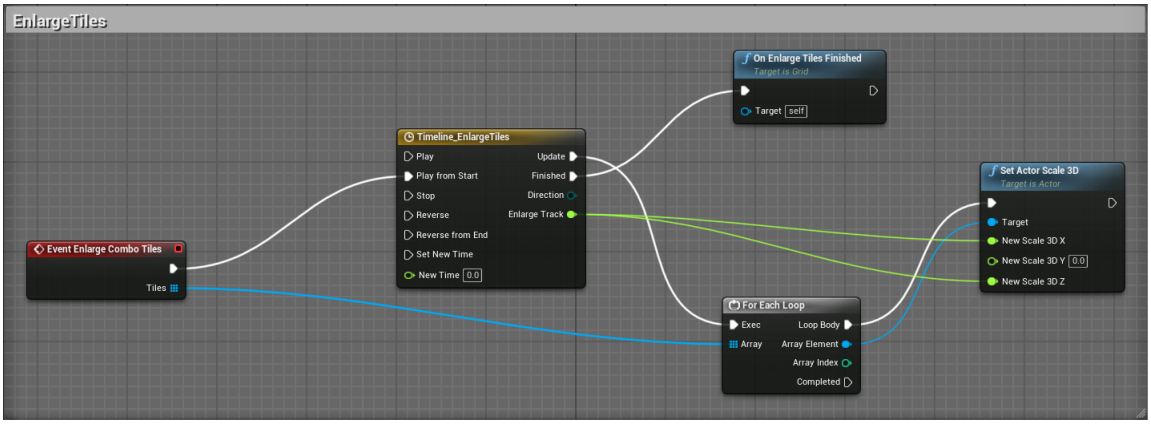
Duplim klikom na čvor otvorit će se *Timeline Editor*. Unutar njega možemo dodati graf i opisati njegovo ponašanje. *Length* vrijednost određuje sveukupnu duljinu animacije. Klikom na *Add Float Track* dodat ćemo novi graf našem `Timeline`-u. Desnim klikom na koordinatni sustav otvorit će nam se opcija *Add Key To...*. Njenim odabirom dodat ćemo točku u naš koordinatni sustav na mjesto gdje smo kliknuli mišem. Preciznije vrijednosti za svaku točku možemo unijeti lijevim klikom na njih i upisivanjem vrijednosti u *Time* i *Value* polja za unos. *Timeline* će za nas, ovisno o točkama, linearnom interpolacijom po segmentima (od točke do točke) izmodelirati funkciju. Osim linearne, možemo odabrati i kubnu interpolaciju.



Slika 5.7: `Timeline editor` i *Float Track*

Play from Start ulaz na čvoru postavit će vrijednost vremena uvijek na 0 kada se pozove pokretanje animacije, dok *Play* ulaz nastavlja od trenutka kada je zaustavljena animacija. Analogno, *Reverse* ulazi imaju istu funkcionalnost, ali tok animacije ide od kraja prema početku. *Update* izlaz nastavit će izvršavanje ukoliko je izvođenje animacije još u tijeku, a ukoliko je animacija završila izvršavanje rada funkcije nastavit će se u smjeru *Finished* izlaza. Za svaki graf koji smo dodali dobit ćemo po jedan izlaz iz kojeg možemo dobiti vrijednost funkcije u tom trenutku.

BlueprintImplementableEvent metoda `EnlargeComboTiles` pozvat će se ako je korisnik napravio *combo* potez te će se figurice prvo povećati, a zatim smanjiti. Kada animacija završi figurice će se uništiti. Metoda prima niz pokazivača na klasu `ATile` nad kojima će se transformacija dogoditi. Njih iteriramo pomoću *For Each Loop* čvora te za svaki od njih mijenjamo *scale* vrijednost ovisno o vrijednosti koju smo dobili iz `Timeline` čvora. Na kraju animacije pozvat će se `OnEnlargeTileFinished` metoda koja će nastaviti izvršavanje *C++* koda te pozvati metodu za uništenje figurica koje se nalaze u nizu od 3 ili više.

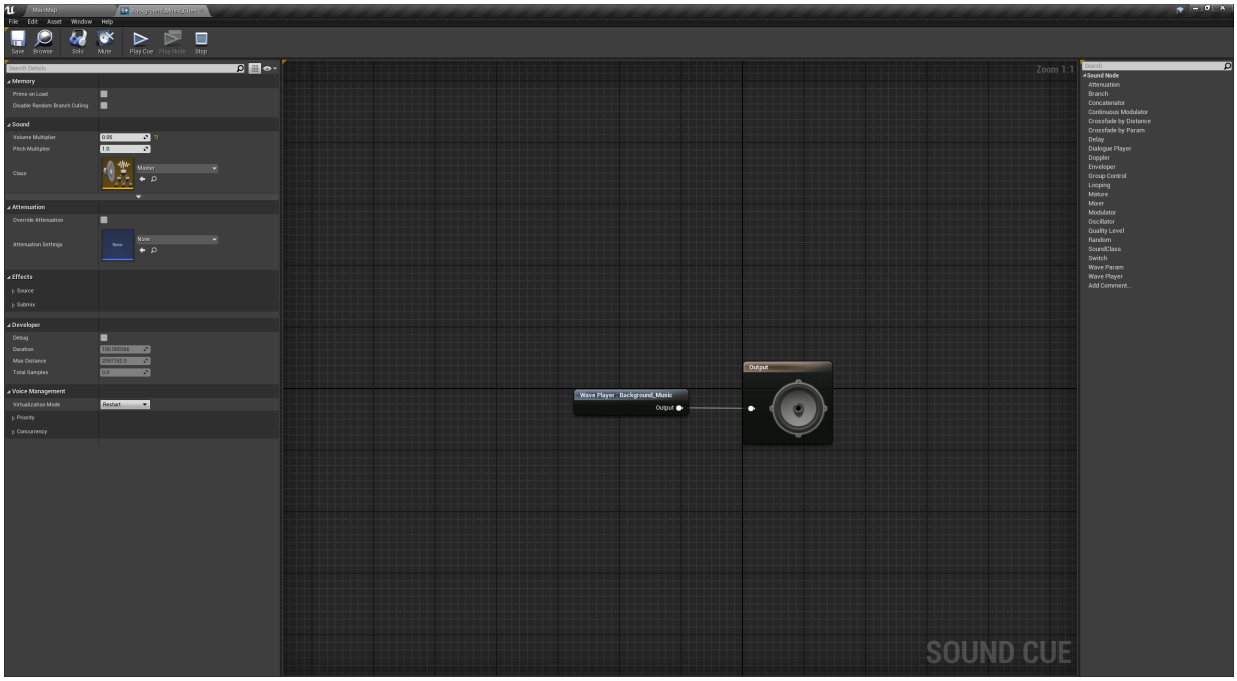


Slika 5.8: `EnlargeComboTiles` metoda

5.5 Zvuk

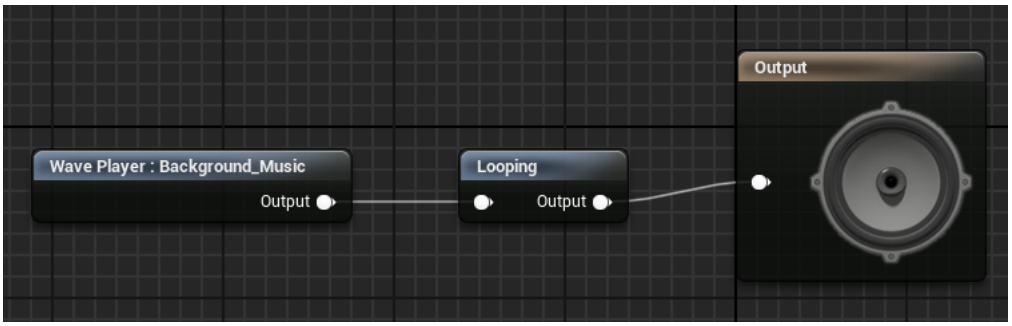
Za kraj nam je preostalo dodati zvuk u našu igru. Bez zvuka naša igra se može činiti statičkom i beživotnom. Kako bismo to izbjegli, dodat ćemo pozadinsku glazbu koja će svirati tokom cijele igre i zvučne efekte pri klikanju, pomicanju i uparivanju figurica.

Odabrane zvukove možemo lagano dodati u projekt tako da njihove datoteke povučemo u *Content browser* panel. UE podržava *WAV* format tako da sve ostale formate trebamo prvo konvertirati u ovaj ako ih želimo koristiti. Nakon što smo ih dodali u projekt, desnim klikom i odabirom opcije *Create Cue* kreirat ćemo objekt klase koji možemo koristiti u našoj igri. Duplim klikom otvorit će se editor u kojem možemo editirati naš zvuk ili podesiti njegove postavke.



Slika 5.9: *Sound editor*

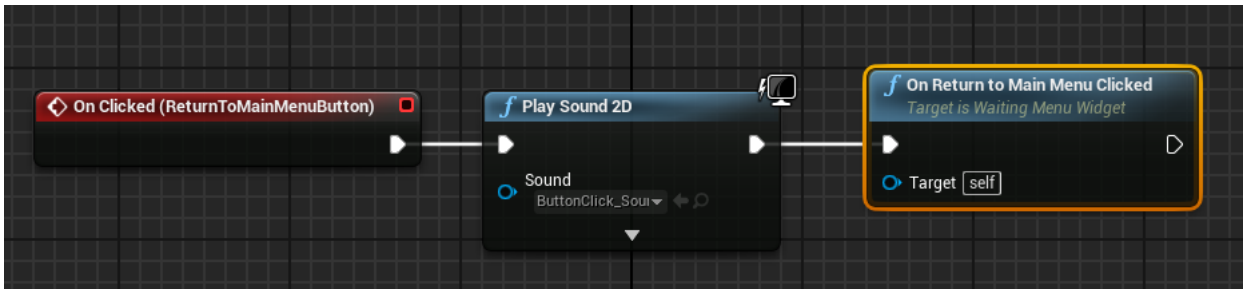
Za našu pozadinsku glazbu bismo željeli da u trenutku kada završi krene od početka. Kako bismo to postigli dovoljno je u našu skriptu dodati *Looping* čvor. Unutar editora ćemo također smanjiti njenu glasnoću jer glazba odabrana za našu igru je trenutno preglasna i nadjačava ostale efekte. U *Volume Multiplier* polju, koji se nalazi u *Sound* kategoriji, postavit ćemo vrijednost za koju je omjer zvučnih efekata i pozadinske glazbe odgovarajući.



Slika 5.10: *Loop* efekt

Sada preostaje dodati *Cue* objekt pozadinske glazbe u scenu. Ulaskom u igru on će se automatski pokrenuti.

Za zvučne efekte postupak dodavanja u projekt i kreiranja *Cue* objekta je isti. Sada jedino moramo odrediti gdje ćemo pozvati određene zvukove u igri. Čvor `Play Sound 2D` nudi nam opciju reproduciranja zvuka te modificiranja postavki zvuka. Reproducirani zvuk ne ovisi o lokaciji puštanja stoga je glasnoća zvuka uvijek jednaka neovisno o poziciji.



Slika 5.11: Primjer reprodukcije zvuka pritiskom na gumb u meniju

5.6 Zaključak

S dodanim zvukom razvoj naše igre priveden je kraju. Koristeći materijale iz ponuđene literature i primjenjivanjem tog znanja na našoj igri savladali smo neke osnovne, a i složenije koncepte *Unreal Engine*-a i *C++*-a. Upoznali smo se s *visual scripting*-om, njegovim *C++* klasama i odnosima među klasama te smo dobili puno bolje razumijevanje samog alata. *Unreal Engine* nudi mnoge opcije s kojima možemo napraviti izrazito kompleksne i vizualno atraktivne igre. Upravo zbog mnoštva mogućnosti koje pruža, *Unreal Engine* nije lagan za korištenje.

Za upoznavanje s alatom preporučam korištenje *blueprint*-a, a kasnije, savladavanjem osnovnih ideja alata i *game development*-a, kombinaciju *C++* koda i *blueprint*-a. Za početnike u području razvoja video igara alat je poprilično zahtjevan za savladati te je potrebno dobro razumijevanje *C++* programskog jezika. Prije korištenja *Unreal Engine*-a također bi bilo preporučljivo proučiti *SFML* biblioteku za razvoj video igara kako bi se savladali neki osnovni koncepti objektno orijentiranog programiranja te *game development*-a.

Bibliografija

[1] Epic Games, *Unreal Engine službena dokumentacija*, https://docs.unrealengine.com/4.26/en-US/

[2] boost, *boost::asio službena dokumentacija* https://www.boost.org/doc/libs/1_77_0/doc/html/boost_asio.html

[3] Ben Tristem, Sam Pattuzzi, GameDev.tv Team, *Unreal Engine C++ Developer: Learn C++ and Make Video Games*, https://www.udemy.com/course/unrealcourse/

[4] William Sherif , Stephen Whittle, *Unreal Engine 4 Scripting with C++ Cookbook*, Pact Publishing, 2016, www.packtpub.com/product/unreal-engine-4-scripting-with-c-cookbook/

[5] Rachel Cordone, *Unreal Engine 4 Game Development Quick Start Guide*, Pact Publishing, 2019, www.packtpub.com/product/unreal-engine-4-game-development-quick-start-guide

[6] Benjamin Carnall, *Unreal Engine 4.X By Example*, Pact Publishing, 2016, https://subscription.packtpub.com/book/game_development/9781785885532/1

[7] Mladen Jurak, *Objektno Programiranje (C++) predavanja*

Sažetak

U ovom radu opisali smo osnovne koncepte *Unreal Engine*-a uz koje možemo započeti rad na novom projektu. Uz ove koncepte i daljnjim proširivanjem znanja pročavanjem alata kroz službenu dokumentaciju moguće je napraviti tehnološki kompleksne projekte koje ne bismo bili u mogućnosti napraviti bez *Unreal Engine* alata ili bilo kojeg drugog pokretača igara. Za potrebe ovog diplomskog rada napravljena je video igra u UE alatima čiji smo žanr, pravila, strukturu programa i klase opisali u radu.

Također smo se upoznali s dizajnom server-klijent aplikacije i detaljnije opisali način na koji funkcioniraju i komuniciraju međusobno sudionici. Koristeći *boost* biblioteku i moderni *C++17* standard implementirali smo asinkroni server i definirali poruke za komunikaciju. Za potrebe razmjenjivanja složenijih struktura podataka preko mreže koristeći poruke implementirali smo i detaljno opisali naš JSON parser.

Summary

In this master thesis we elaborated the basic concepts of Unreal Engine with which we can start working on a new project. With those concepts and further expanding knowledge by reading official documentation, it is possible to create technologically complex projects that we would not be able to do so without Unreal Engine tool or any other tool for game launching. For the purposes of this thesis, the video game was made in UE tools. Its genre, rules and program structure are described in the paper.

Also, the design of the server-client application is introduced. The way participants function and communicate with each other is described in detail. Using the *boost* library and the modern *C ++ 17* standard, we have implemented an asynchronous server and defined messages for communication. For the purpose of exchanging more complex data structures over the network using messages, we have implemented and described in detail our JSON parser.

Životopis

Nikola Vučković rođen je 11. kolovoza 1995. u Zagrebu. Završio je Osnovnu školu "Medvedgrad", a kasnije Prirodoslovno-matematičku gimnaziju u Zagrebu. Od djetinjstva gaji ljubav prema matematici i programiranju te zbog toga odlučuje upisati preddiplomski sveučilišni studij Matematike na PMF-u u Zagrebu te nakon toga diplomski studij Računarstva i matematike na istom fakultetu. Zbog izrazite zainteresiranosti za rad na razvoju video igara, zapošljava se kao *junor game developer* u Nanobit-u u kolovozu 2020. godine gdje i dalje razvija svoju karijeru i znanje.