



UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET
DEPARTMAN ZA MATEMATIKU I
INFORMATIKU



Svemirska puččina "Andromeda" **u Unity platformi**

Informatički Projekat

Tojagić Rastko

Novi Sad, 2020.

Predgovor

Razvoj računarskih igara je tokom protekle decenije veoma uznapredovao zahvaljujući pojavi platformi koje taj razvoj znatno olakšavaju i ubrzavaju. Jedna od takvih platformi jeste Unity koji podržava razvoj kako jednostavnijih 2d tako i zahtevnijih 3d igara. U ovom informatičkom projektu pokazaćemo upravo koliko je jednostavno napraviti 2d svemirsku pucačinu u svega nekoliko stotina linija koda, kao i neke često korišćene obrasce (eng. design patterns) u razvoju igara.

Tojagić Rastko

Novi Sad, Septembar 2020.

Sadržaj

| | | |
|----------|---|-----------|
| 1 | Platforma za razvoj igara Unity | 1 |
| 1.1 | MonoBehaviour i pisanje skripti u jeziku C# | 1 |
| 1.1.1 | Scriptable Objects | 1 |
| 1.1.2 | Coroutines | 1 |
| 1.2 | Unity Editor | 4 |
| 2 | Osnovni pojmovi i sistemi u igrima | 6 |
| 2.1 | Game Loop | 6 |
| 2.1.1 | FPS | 6 |
| 2.1.2 | Update | 6 |
| 2.1.3 | FixedUpdate | 6 |
| 2.1.4 | Vremena, klasa Time | 7 |
| 2.2 | Obrasci (eng. design patterns) | 7 |
| 2.2.1 | "Singleton" obrazac | 7 |
| 2.2.2 | "Command" obrazac | 13 |
| 3 | Andromeda - Svemirska puččina | 16 |
| 3.1 | Pregled i ideja igre | 16 |
| 3.2 | GameManager klasa | 16 |
| 3.3 | Kontrole i InputManager | 19 |
| 3.4 | Ponašanje svemirskog broda i ShipController klasa | 21 |
| 3.5 | Neprijatelji i osnovno ponašanje | 24 |
| 3.6 | Zaključak | 27 |

1 Platforma za razvoj igara Unity

1.1 MonoBehaviour i pisanje skripti u jeziku C#

MonoBehaviour je osnovna klasa koju svaka Unity skripta nasleđuje. Kada koristimo C# za skriptni jezik ovu klasu moramo eksplicitno da nasledimo [2]. U svakom trenutku moguće je isključiti ili ponovo uključiti skriptu. Isključivanje izuzima skriptu iz aktivnih skripti. Ukoliko je skripta uključena i ima definisane neke od specifičnih funkcija koje se pozivaju u određenom trenutku tokom životnog ciklusa objekta smatra se aktivnom i reaguje na pozive tih funkcija onako kako je korisnik to definisao. Neke od ključnih funkcija su: *Start()*, *Update()*, *FixedUpdate()*, *OnEnable()*, *OnDisable()*. Životni ciklus skripte prikazan je grafikonom 1. Kasnije u tekstu biće data preciznija objašnjenja za najbitnije funkcije.

Drugim rečima *MonoBehaviour* predstavlja definisano ponašanje nekog živog objekta u svetu igre. Svaki objekat u svetu igre naziva se *GameObject* i tipa je istoimene klase. Svaki *MonoBehaviour* jeste *GameObject*. Na *GameObject* je moguće dodati komponente. Treba imati na umu da ovakav sistem koji Unity pruža nije pravi ECS (Entity Component System) te da zbog ovakvog dizajna cela platforma radi sporije. Unity od skoro uvodi pravi ECS uz *Burst Compiler*, ali ovo još uvek nije deo standardne platforme [2].

1.1.1 Scriptable Objects

ScriptableObject je kontejner za skladištenje podataka, nezavisan od instance klase. Oni omogućavaju uštedu memorije za podatke koji su konfigurabilni i ne menjaju se tokom izvršavanja igre. Na primer, konfiguracije poput jačine potiska, brzine okretanja i slično kod neprijateljskih brodova (kojih će biti puno tokom igre). Takođe ono što je zgodno jeste da te podatke možemo da sačuvamo kao fajl u projektu, i onda jednostavnim prevlačenjem u slot koristimo 2.

1.1.2 Coroutines

Korutine su specijalne funkcije povratnog tipa *IEnumerator* čije izvršavanje se može pauzirati naredbom

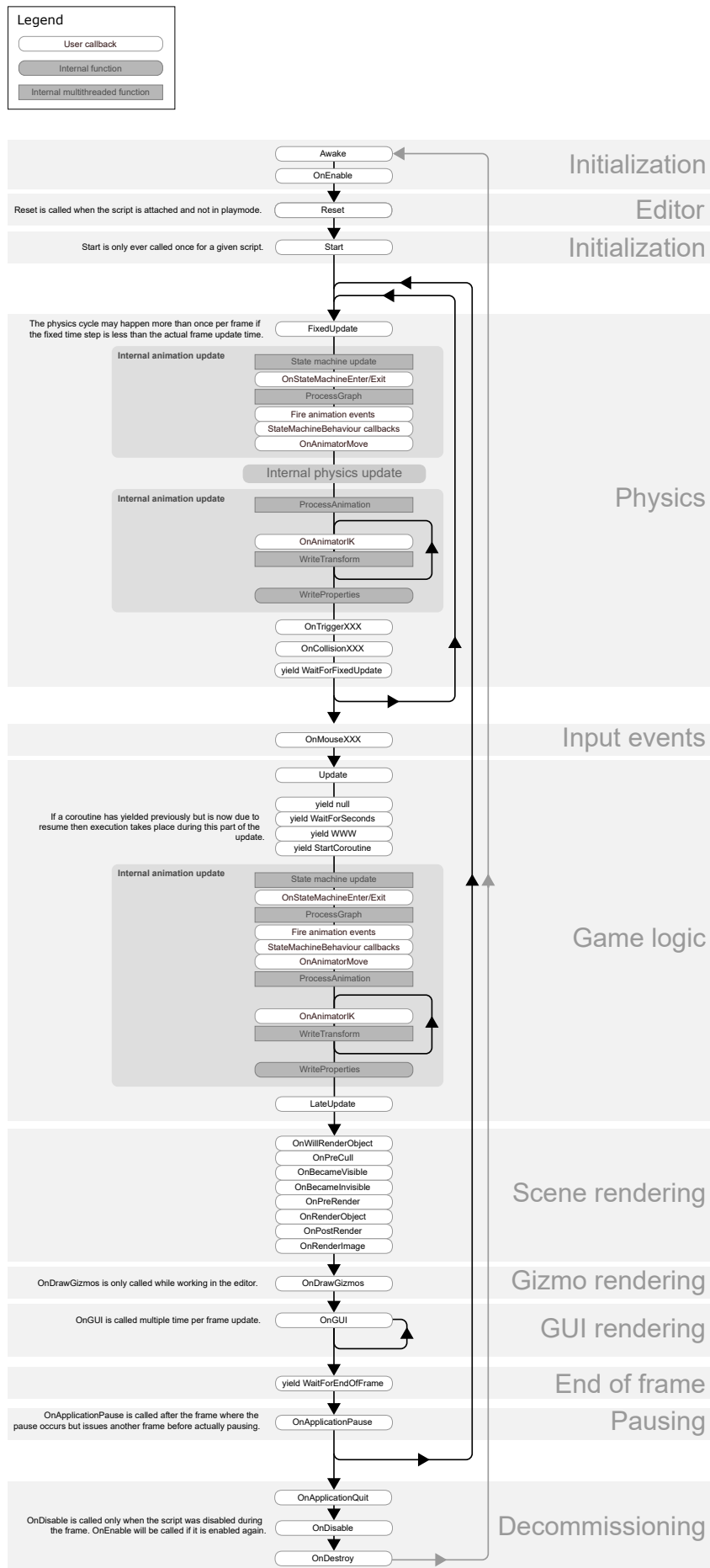


Figure 1: MonoBehaviour lifecycle

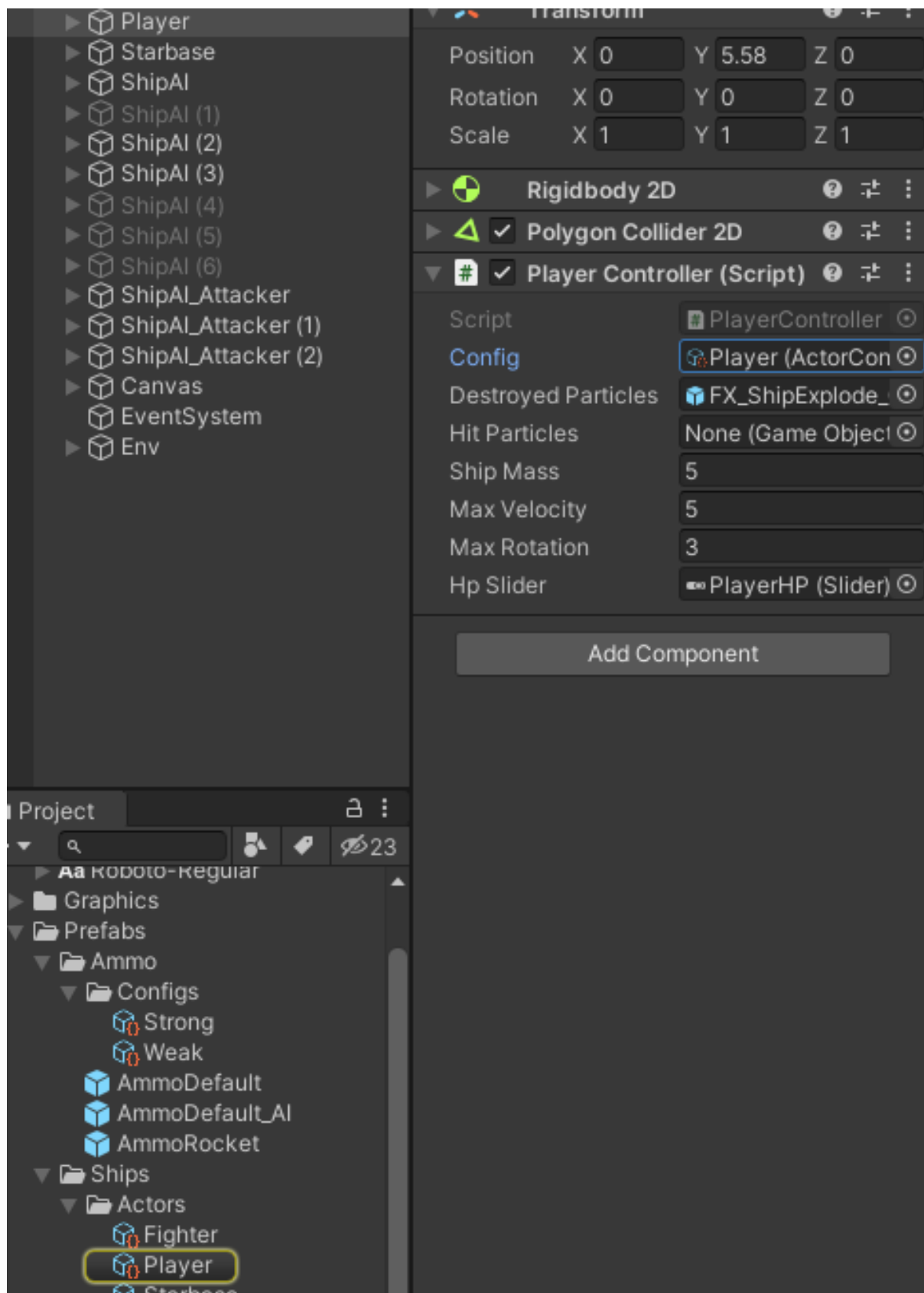


Figure 2: Slot for scriptable object

```
yield return new WaitForSeconds(secs);  
  
// ili  
  
yield return null;
```

gde ova druga pauzira izvršavanje do sledećeg frejma. Korutine koristimo kada želimo da se neka stvar dešava tokom nekoliko frejmova, recimo želimo da napravimo odlaganje kraja igre nakon uništenja svemirske baze (videti 3.2).

1.2 Unity Editor

Editor 3 je grafičko okruženje namenjeno za postavljanje sveta igre. Editor pruža vizuelni pregled svih objekata na sceni, kamere i efekata. Prozor je podeljen na nekoliko potprozora čije uloge su da olakšaju podešavanja i pregled igre.

SceneView predstavlja pregled trenutne, aktivne scene. Scena predstavlja skup objekata i kameru koji su u datom trenutku prikazani korisniku. Poželjno je zasebne logičke celine igre odvajati u zasebne scene, tako na primer, možemo imati scenu za *glavni meni* i *scenu igre*, gde bi kroz u meniju bila omogućena neka podešavanja, pregled najboljih rezultata i slično, dok je scena igre sama igra.

Hierarchy View omogućava pregled svih objekata na trenutnoj sceni. Objekti mogu biti u roditelj-dete odnosu, te stoga i ime, jer tako čine hijerarhiju.

Project View daje pregled svih fajlova trenutno otvorenog projekta na disku.

Game View obično složen pored pregleda scene prikazuje svet igre onakav kakav će biti prikazan igraču, dakle bez pomoćnih elemenata poput kvadratne mreže ili mogućnosti pomeranja objekata.

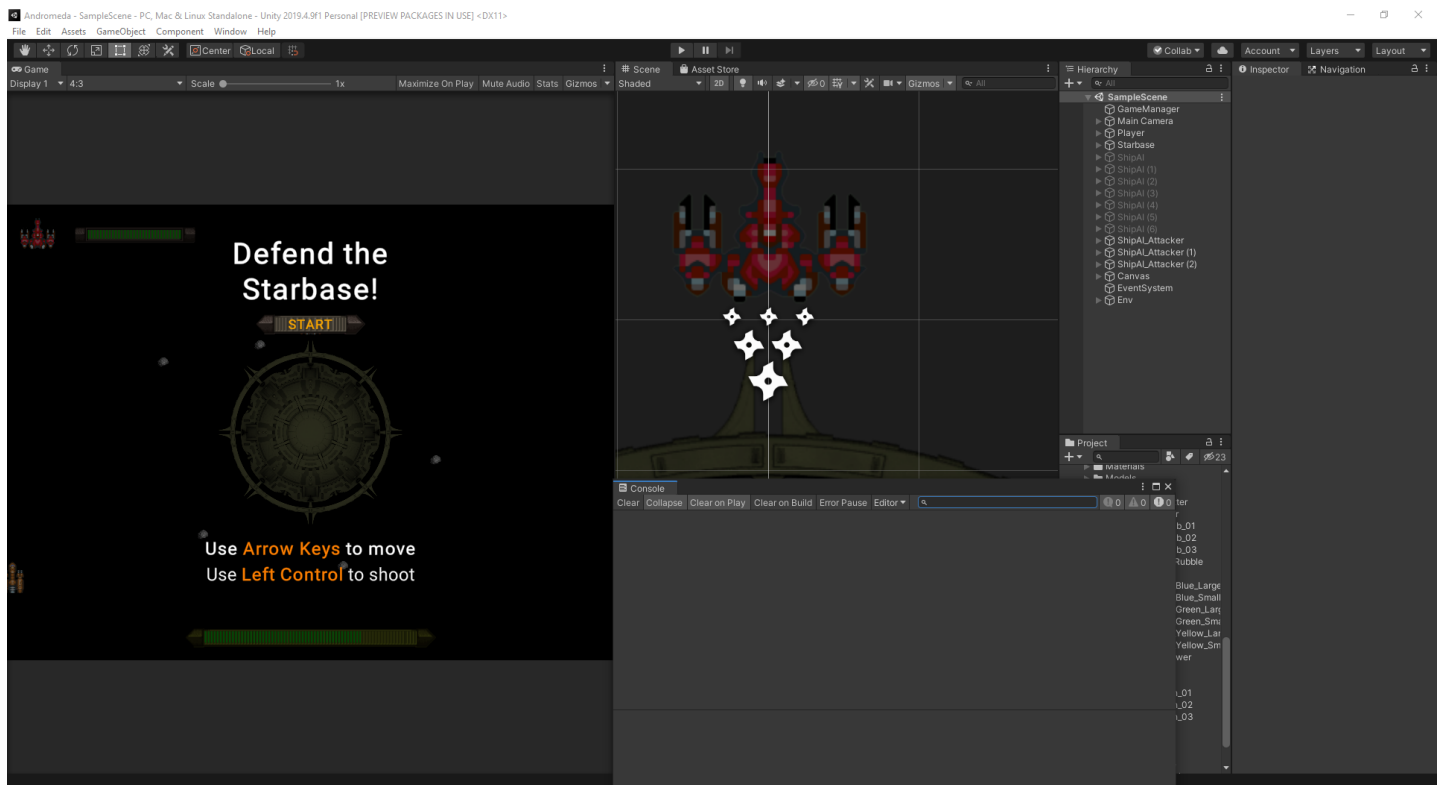


Figure 3: Unity Editor

2 Osnovni pojmovi i sistemi u igrama

2.1 Game Loop

Ključna tačka izvršavanja igre jeste glavna petlja igre (eng. *game loop*). Jednom kada je igra pokrenuta sve ono što je vidljivo korisniku, grafika, zvuk, interakcija procesira se kroz glavnu petlju. U tom jednom prolazu petlje koji nazivamo *tick* ili *frame* dešava se iscrtavanje grafike, reagovanje na korisnički unos (unos sa tastature ili putem miša ili nekih drugih perifernih uređaja) te reakcija sveta na zadate promene... odnosno, ovde se dešava sve, zato je glavna petlja srce same igre. Modernije arhitekture dozvoljavaju određeni stepen paralelizacije nekih procesa, pa bi na primer iscrtavanje grafike moglo da se izdvoji u paralelan proces i slično.

2.1.1 FPS

FPS skraćeno od *frames per second* je osnovna mera performansi igre. Ova mera kazuje nam koliko je puta po sekundi moguće izvršiti glavnu petlju. Veći broj je i bolji, pa tako igra koja se izvršava na manje od 30 frejmova po sekundi nije prijatna za igranje i događa se takozvano *seckanje*, odnosno objekti se ne pomeraju glatko kako bi trebalo već iscepkano što narušava celokupan doživljaj igranja. Nizak FPS može da bude rezultat loše napisanog koda ili prosto zastarelog hardvera. Moderne igre se trude da se održe na 60 FPS.

2.1.2 Update

Update je funkcija koja se poziva upravo iz glavne petlje igre. To je funkcija koja je može biti definisana u *MonoBehavior* klasi. Update funkcija se poziva nad svakim objektom tipa *MonoBehavior* (ukoliko je definiše) tačno jednom svakog frejma [2]. Sva promenljiva ponašanja objekta mogu se definisati ovde.

2.1.3 FixedUpdate

Slično funkciji *Update*, i ova funkcija se poziva periodično ali ne svakog frejma, već na fiksno vreme svake 0.02 sekunde [2]. Prema uputstvima iz dokumentacije, u ovoj funkciji se mora raditi sve što ima veze sa bilo kakvim izračunavanjem fizike. U igri se oslanjamo na Unity-jev sistem za fiziku za kretanje, pa ćemo se dalje u tekstu detaljnije baviti time.

2.1.4 Vremena, klasa *Time*

Jedna od bitnijih stavki u igrama jeste vreme. Merimo dve vrste vremena, koja očitavamo iz klase *Time*.

Time.time je vreme koje je proteklo od pokretanja igre i ono je korisno obično kada želimo da ograničimo izvršavanje nekog dela koda vremenski. Na primer, ne želimo da neprijatelji prebrzo ispaljuju metke na igrača, te možemo meriti vreme proteklo od prethodnog pucanja i proveriti da li je to vreme veće od neke konstante koja nam predstavlja minimalan vremenski interval između dva pucanja.

Time.deltaTime je vreme za koje se izvršio prethodni frejm. Ovo je jako bitna stavka, i potrebno nam je kada imamo bilo kakvu promenu stanja u igri koja je vidljiva (na primer, kretanje). Kako se u glavnoj petlji izvršava svaka takva promena, to znači da se promena dešava jednom po frejmu. Već je pomenuto da moderne igre teže ka konstantnom izvršavanju na 60 FPS ali to nije uvek moguće, odnosno dešava se da zbog lošijeg hardvera ili lošeg koda FPS opada.

Uzmimo kao primer kretanje nekog objekta. Tada imam vektor v koji je vektor pomeraja u svakom frejmu. Zbog nekonzistentnosti FPSa na različitim računarima postoji opasnost da dobijemo različite brzine kretanja, ovo je problem koji direktno narušava mogućnost igranja na zamišljen način, posebno u multiplayer igrama. Pogledati 4 za ilustraciju.

2.2 Obrasci (eng. design patterns)

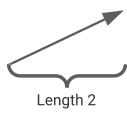
U razvoju softvera a posebno kompleksnijih sistema i većih projekata često se sreće upotreba raznih obrazaca koji olakšavaju organizaciju koda i pomažu da se razreše nepotrebna uplitanja (eng. coupling) između klasa. Obrasci su tu da nadomeste nedostatke nekih programskih jezika.

2.2.1 "Singleton" obrazac

Ovaj obrazac osigurava da postoji tačno jedna instanca singleton klase, kao i mogućnost globalnog pristupa toj instanci [1]. Ostale prednosti korišćenja ovog obrasca su:

- Instanca klase se ne pravi ukoliko se ne koristi.
- Inicijalizacija se dešava tokom izvršavanja programa (izbegavamo korišćenje čisto statičkih klasa).

Move vector



In the given situation we move 2 lengths per frame.

at 60 FPS -> move is $2 \times 60 = 120$ units, $dt = 16$ ms

at 30 FPS -> move is $2 \times 30 = 60$ units, $dt = 32$ ms

at 15 FPS -> move is $2 \times 15 = 30$ units, $dt = 64$ ms

Consistent movement after multiplication by $dt \rightarrow$

at 60 FPS -> move is $2 \times 60 \times dt = 1920$ units, $dt = 16$ ms

at 30 FPS -> move is $2 \times 30 \times dt = 1920$ units, $dt = 32$ ms

at 15 FPS -> move is $2 \times 15 \times dt = 1920$ units, $dt = 64$ ms

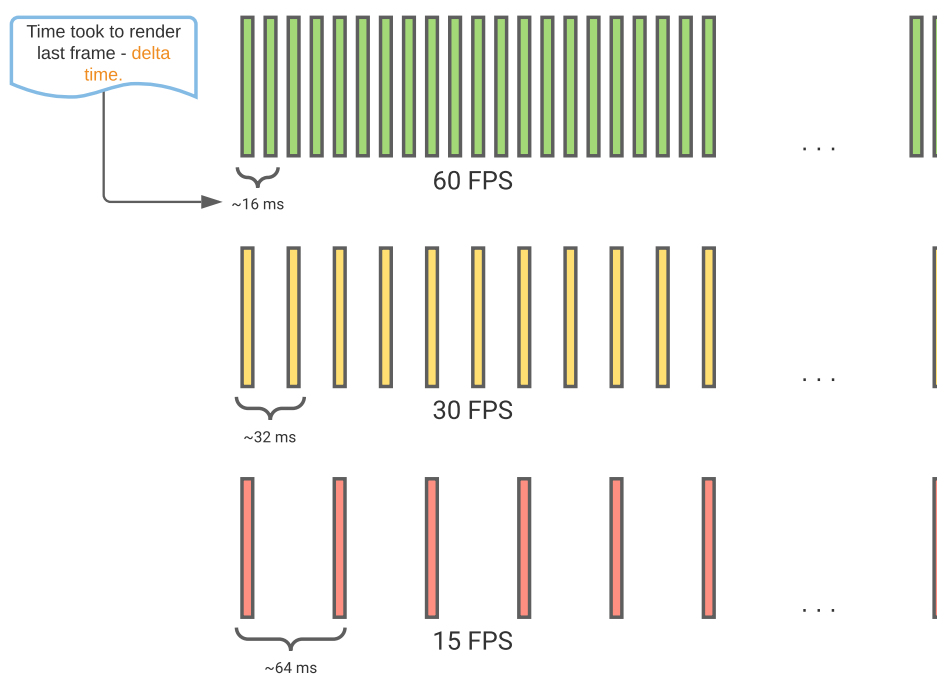


Figure 4: DeltaTime objašnjenje

- Klasa može biti nasleđena.

Primer implementacije Singleton obrasca koji nije thread-safe (loše)!

```
public class Singleton
{
    private static Singleton _instance = null;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            return _instance;
        }
    }
}
```

Jednostavna thread-safe implementacija:

```
public class Singleton
{
    private static Singleton _instance = null;
    private static readonly object _root = new object();

    Singleton() { }

    public static Singleton Instance
    {
        get
        {
            lock (_root)
            {

```

```
        if (_instance == null)
        {
            _instance = new Singleton();
        }

        return instance;
    }
}
}
```

U Unity platformi postoje situacije kada se neki problemi ne mogu rešiti pomoću statičkih klasa, stoga moramo koristiti Singleton kako bi smo osigurali globalni pristup ali pri tome sačuvati osobine MonoBehaviour-a. Primer takve klase je klasa *GameManager* o kojoj će biti više reči dalje u tekstu. Implementacija Singleton obrasca se u Unity platformi dodatno razlikuje, jer komponente koje su izvučene na scenu vizuelno, već jesu instance tih klase, ali to se lako rešava, što ćemo videti na priloženom kodu (izvor github.com):

```
/// <summary>
/// Mono singleton Class. Extend this class to make singleton component.
/// Example:
/// <code>
/// public class Foo : MonoSingleton<Foo>
/// </code>. To get the instance of Foo class, use <code>Foo.instance</code>
/// Override <code>Init()</code> method instead of using <code>Awake()</code>
/// from this class.
/// </summary>
public abstract class MonoSingleton<T> : MonoBehaviour where T : MonoBehaviour
{
    private static T m_Instance = null;
    public static T Instance
    {
        get
        {
            // Instance required for the first time, we look for it
            if (m_Instance == null)
            {
```

```
m_Instance = GameObject
    .FindObjectOfType(typeof(T)) as T;

// Object not found, we create a temporary one
if (m_Instance == null)
{
    Debug.LogWarning("No instance of " +
        typeof(T).ToString() +
        ", a temporary one is created.");

    isTemporaryInstance = true;
    m_Instance = new GameObject("Temp Instance of " +
        typeof(T).ToString(), typeof(T))
        .GetComponent<T>();

    // Problem during the creation,
    // this should not happen
    if (m_Instance == null)
    {
        Debug
            .LogError("Problem during the creation of " +
                typeof(T).ToString());
    }
}
if (!_isInitialized)
{
    _isInitialized = true;
    m_Instance.Init();
}
return m_Instance;
}

public static bool isTemporaryInstance { private set; get; }
```

```
private static bool _isInitialized;

// If no other monobehaviour request
// the instance in an awake function
// executing before this one,
// no need to search the object.
private void Awake()
{
    if (m_Instance == null)
    {
        m_Instance = this as T;
    }
    else if (m_Instance != this)
    {
        Debug.LogError("Another instance of " + GetType() +
            " is already exist! Destroying self...");
        DestroyImmediate(this);
        return;
    }
    if (!_isInitialized)
    {
        DontDestroyOnLoad(gameObject);
        _isInitialized = true;
        m_Instance.Init();
    }
}

/// <summary>
/// This function is called when
/// the instance is used the first time
/// Put all the initializations you need here,
/// as you would do in Awake
/// </summary>
public virtual void Init() { }
```



```
/// Make sure the instance
/// isn't referenced anymore when the
/// user quit, just in case.
private void OnApplicationQuit()
{
    m_Instance = null;
}
}
```

2.2.2 "Command" obrazac

Ovaj obrazac služi da delegira izvršavanje neke funkcije te tako omogući da se izbegne pojam *hardwire* odnosno čvrsto vezivanje između nekih entiteta, na primer, čitanja korisnikovog unosa sa tastature te reagovanje na isti. Uzmimo baš taj primer:

```
void HandleInput()
{
    if (isPressed(BUTTON_X)) Jump();
    else if (isPressed(BUTTON_Y)) FireGun();
    else if (isPressed(BUTTON_A)) SwapWeapon();
    // ...
}
```

Ukoliko želimo da drugačije mapiramo komande, što je vrlo čest slučaj u igrama, kroz konfiguracioni meni. Za naše potrebe napravićemo apstraktnu klasu *ACommand*.

```
public abstract class ACommand
{
    public KeyState State { get; set; }
    public bool IsPhysics { get; private set; }

    public ACommand(bool isPhysics)
    {
        IsPhysics = IsPhysics;
    }

    public abstract void Execute(ShipController ship, Actor actor);
}
```

Dalje prema potrebama nasleđujemo ovu klasu i definišemo konkretna ponašanja. Na primer, dodavanje potiska:

```
public sealed class Thrust : ACommand
{
    public Thrust() : base(true) { }

    public override void Execute(ShipController ship, Actor actor)
    {
        ship.AddThrust(ship.transform.up, actor.ThrustPower);
    }
}
```

Ove komande se onda pakuju u neku strukturu, u našem slučaju koristimo *Queue*, i procesiramo ih u glavnoj petlji. Ovo radimo ovako, jer hoćemo da registrujemo i po nekoliko unosa istovremeno, odnosno, na primer, brod može da se kreće napred i da puca istovremeno.

```
while (actions.Count > 0)
{
    var action = actions.Dequeue();
    action.Execute(this, _actor);
}
```

Pošto smo napravili ovakvu apstrakciju, sada je moguće da i ponašanje neprijatelja zadajemo preko komandi, odnosno skripta koja kontrliše neprijatelja će samo dodavati potrebne akcije u niz akcija. Na primer, kretanje prema igraču:

```
// Move towards player target
Vector2 targetDir = _target.position - transform.position;
if (targetDir.magnitude >= 5f)
{
    actions.Enqueue(new Thrust());
}
```

Na drugoj strani, kod igrača, definišemo akcije kao mapu:

```
InputManager.Instance.InputMap[KeyCode.UpArrow] =
    new Thrust();
```

```
InputManager.Instance.InputMap[KeyCode.LeftArrow] =  
    new Rotate(1);  
InputManager.Instance.InputMap[KeyCode.RightArrow] =  
    new Rotate(-1);  
InputManager.Instance.InputMap[KeyCode.LeftControl] =  
    new Shoot(AmmoType.DEFAULT,  
        new Vector2[]  
        {  
            Vector3.up * 1.5f  
        }  
    ));
```

a onda u klasi koja se bavi učitavanjem unosa sa tastature imamo sledeće:

```
foreach (var input in InputMap)  
{  
    if (Input.GetKeyDown(input.Key))  
    {  
        input.Value.State = KeyState.PRESSED;  
        inputQueue.Enqueue(input.Value);  
    }  
  
    else if (Input.GetKey(input.Key))  
    {  
        input.Value.State = KeyState.HELD;  
        inputQueue.Enqueue(input.Value);  
    }  
}
```

Postavlja se pitanje da li bi onda i akcije Thrust, Rotate (moža dodatno RotateLeft i RotateRight) mogle da budu Singleton objekti, i odgovor je pozitivan ali sa nekim određenim izmenama, ali time se nećemo baviti dublje.

3 Andromeda - Svemirska pucačina

3.1 Pregled i ideja igre

Originalna ideja igre je nešto šira od onoga što imamo ovde, tako da ćemo ovo nazvati demo verzijom. Sa tim na umu obraćena je posebna pažnja na to da kod bude lako proširiv, te da nemamo tesne veze između klasa, a posebna je vođeno računa o tome da se sami podaci o aktorima u igri (brodovi, svemirska baza) izdvoje kao posebni objekti (videti 1.1.1) radi lakše konfiguracije i proširivosti. Ukratko ćemo dati pregled najbitnijih stavki igre.

Igrač dobija kontrolu nad jednim svemirskim brodom koji može da se kreće isključivo napred, potiskom, ali isto tako može da se okreće oko svoje ose, te je tako omogućeno kretanje u svim pravcima u ravni. Pored kretanja, igrač ima mogućnost da puca na neprijatelje. Tasteri koji se koriste prilikom ovih akcija su strelice, i to *UpArrow* za potisak, *LeftArrow* i *RightArrow* za rotaciju, i *LeftControl* za pucanje.

Neprijatelji su brodovi kontrolisani od strane računara koji za cilj imaju da unište glavnu bazu u igri nazvana *Starbase*. U prvoj verziji postoje dve vrste neprijatelja čiji opisi se nalaze dalje u tekstu.

Fighter ili lovac je mali okretan brod koji napada isključivo igrača. Neprijatelj šalje gomilu lovaca kako bi zaštitio napadače i otežao odbranu svemirske baze za igrača.

Attacker ili napadač je sporiji brod lošijih manevarskih sposobnosti poslat sa ciljem da napada isključivo svemirsku bazu. Iako sporiji i manje okretljivi od lovaca ovi brodovi opremljeni su jakim topovima koji su u mogućnosti da raznesu svemirsku bazu i nanesu ogromnu štetu igraču ukoliko se nađe na njihovoj putanji.

3.2 GameManager klasa

Već je bilo reči o tome da je ova klasa Singleton odnosno, preciznije, MonoSingleton. Ova klasa zadužena je da vodi računa o stanju igre. U našem slučaju, uzevši u obzir da je ovo tek demo, ona ne radi puno stvari u ovom trenutku.



Figure 5: Igra

```
public sealed class GameManager : MonoSingleton<GameManager>
{
    [SerializeField]
    private GameObject _pnGameOver;
    [SerializeField]
    private GameObject _pnStartGame;

    public int gameWorldRadius = 300;

    private void Start()
    {
        _pnGameOver.SetActive(false);
        _pnStartGame.SetActive(true);
        Time.timeScale = 0;
    }

    public void GameOver(float secs = 0f)
    {
        StartCoroutine(Delay(secs));
    }

    public void StartGame()
    {
        _pnGameOver.SetActive(false);
        _pnStartGame.SetActive(false);
        Time.timeScale = 1;
    }

    private IEnumerator Delay(float secs)
    {
        yield return new WaitForSeconds(secs);
        _pnGameOver.SetActive(true);
        Time.timeScale = 0;
    }

    public void Retry()
```

```
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex)
    Destroy(this.gameObject);
}
}
```

Dakle, za sada vodi računa o aktivnim ekranima, početku igre i restartovanju nivoa nakon izgubljene igre. Funkcija *GameOver* zapada za oko, i vidimo da poziva funkciju *StartCoroutine* kojoj kao argument prosleđuje funkciju koja zapravo radi ono što treba da se desi kada se igra završi. Ovo je i razlog zašto ova klasa ne može da bude prosto statička. Da bismo mogli da pozovemo *korutinu* 1.1.2 klasa mora da nasledi *MonoBehaviour*, ali nam je istovremeno potreban jednostavan globalan pristup za ovu klasu, jer hoćemo da možemo da završimo igru sa više mesta, recimo iz klase koja kontroliše igrača ili iz klase koja predstavlja svemirsku bazu. Odlaganje poziva ovde radimo kako bismo dali trenutak igraču da ukapira šta se desilo, i zašto je igra završena. Ovo odlaganje omogućava igraču da vidi eksploziju baze ukoliko je to uzrokovalo kraj igre.

Uloga ove klase je dosta šira jednom kada igra počne da izlazi iz demo faze i počne da dobija neku celinu. Recimo, ova klasa vodila bi računa i nivoima u igri, poenima igrača, ostalim ekranima koji se smenjuju iz menija pa do glavne scene igre i slično.

GameOver završava igru tako što postavlja *timeScale* na 0 čime prestaje proticanje vremena u igri što zaustavlja sve elemente igre. Pri ponovnom startovanju igre ovo vreme se vraća na prvobitnu vrednost 1.

3.3 Kontrole i InputManager

Ova klasa predstavlja blagu apstrakciju za procesiranje korisničkog unosa. Jedina stvar koja nam treba zapravo jeste da unos pretvorimo u niz komandi, koja takođe treba da zna stanje tipke. Stanje definišemo pomoću enuma:

```
public enum KeyState
{
    NONE,
    PRESSED,
    HELD
}

public sealed class InputManager : MonoSingleton<InputManager>
{
```

```
public Dictionary<KeyCode, ACommand> InputMap
{
    get;
    private set; } = new Dictionary<KeyCode, ACommand>();

public Queue<ACommand> GetInput()
{
    Queue<ACommand> inputQueue = new Queue<ACommand>();

    foreach (var input in InputMap)
    {
        // Down
        if (Input.GetKeyDown(input.Key))
        {
            input.Value.State = KeyState.PRESSED;
            inputQueue.Enqueue(input.Value);
        }

        // Pressed
        else if (Input.GetKey(input.Key))
        {
            input.Value.State = KeyState.HELD;
            inputQueue.Enqueue(input.Value);
        }
    }

    return inputQueue;
}
```

InputMap smo videli već kada smo pričali o *Command* obrascu, a ovde sada vidimo celu funkciju *GetInput* koja vraća niz komandi kroz strukturu *Queue* radi lakše obrade i očuvanja redosleda. Imamo dve vrste registrovanja pritiska tipke sa tastature: tipka je pritisnuta i tipka je zadržana. Ovo nam je bitno, jer recimo hoćemo da igrač mora konstantno da pritiska tipku ako hoće da puca, a to isto bi bilo dosta naporno raditi za kretanje, tako da je za kretanje dovoljno da je tipka zadržana.

3.4 Ponašanje svemirskog broda i ShipController klasa

ShipController klasa je klasa koja opisuje ponašanje broda u opštem slučaju, dakle ovu klasu u osnovi koriste i neprijatelji i igrač. Ova klasa obrađuje komande (akcije) kroz glavnu petlju igre, odnosno u funkciji *FixedUpdate* u kojoj se izvršavaju sve akcije koje utiču na fiziku. Akcije dodajemo u red.

```
protected Queue<ACommand> actions = new Queue<ACommand>();  
public void AddAction(ACommand action) => actions.Enqueue(action);
```

Ove akcije dalje procesiramo:

```
protected void FixedUpdate()  
{  
    while (actions.Count > 0)  
    {  
        var action = actions.Dequeue();  
        action.Execute(this, _actor);  
    }  
}
```

Ostatak klase bavi se samim pomeranjem kao i detekcijom kolizije. Ono što je još bitno napomenuti jeste da se klasa oslanja na *ScriptableObjects* za konfiguraciju, tako vidimo definiciju polja tipa *Actor* i *ActorConfig* na osnovu kojeg se zapravo pravi sam *Actor*. *ActorConfig* je apstrakcija podataka i definisan je kao *ScriptableObject*. Priložene su obe klase:

```
[CreateAssetMenu(fileName = "Actor",  
    menuName = "Andromeda/Ships/ActorConfig", order = 1)]  
public class ActorConfig : ScriptableObject  
{  
    public int HitPoints;  
    public float ThrustPower;  
    public float RotationCoef;  
}  
  
public class Actor  
{  
    public int HitPoints;
```

```
    public float ThrustPower;
    public float RotationCoef;

    public Actor(ActorConfig config)
    {
        HitPoints = config.HitPoints;
        ThrustPower = config.ThrustPower;
        RotationCoef = config.RotationCoef;
    }
}

public class ShipController : MonoBehaviour
{

    public ActorConfig config;
    protected Actor _actor;

    public void AddThrust(Vector2 direction, float force)
    {
        if (GetComponent<Rigidbody2D>().velocity.magnitude >
            maxVelocity)
        {
            Debug.Log("Already at max speed");
            return;
        }

        GetComponent<Rigidbody2D>().AddForce(direction * force);
    }

    public void AddRotation(float rotation)
    {
        transform.Rotate(Vector3.forward,
            rotation * Time.deltaTime);
    }

    protected void OnTriggerEnter2D(Collider2D collider)
    {
```

```
if (collider.gameObject.tag.Equals("Projectile"))
{
    var ammo =
        collider.gameObject
        .GetComponent<Projectile>()
        .GetAmmoConfig;
    _actor.HitPoints -= ammo.Damage;

    // Check death
    if (_actor.HitPoints <= 0)
    {
        // Died
        if (OnDeath != null)
        {
            // Custom death event
            OnDeath();
        }
        else
        {
            // Default death event
            // TODO: avoid dynamic allocation
            Instantiate(destroyedParticles,
                transform.position, Quaternion.identity);
            Destroy(this.gameObject);
        }
    }
    else
    {
        // TODO: avoid dynamic allocation
        Instantiate(hitParticles,
            transform.position, Quaternion.identity);
    }
}
}
```

Ovu klasu nasleđuje i *PlayerController* klasa koja dodatno još procesira akcije:

```
public sealed class PlayerController : ShipController
{
    ...

    private void Update()
    {
        var inputs = InputManager.Instance.GetInput();
        while (inputs.Count > 0)
        {
            var action = inputs.Dequeue();
            if (!action.IsPhysics)
            {
                action.Execute(this, _actor);
            }
            else
            {
                this.AddAction(action);
            }
        }

        if (transform.position.sqrMagnitude >= 300f)
        {
            Debug.LogWarning("Game Over out of range!");
        }
    }

    ...
}
```

U zavisnosti od toga da li akcije utiču na fiziku, procesiraju se direktno, ili se delegiraju u red akcija koje procesira ShipController klasa u FixedUpdate.

3.5 Neprijatelji i osnovno ponašanje

Neprijatelji su skriptovani krajnje jednostavno, sa svega dve akcije koje izvršavaju: pomeraj se ka meti, pucaj na metu, stoga nije bilo potrebe za kompleksnijim pristupom

skriptovanja AI ponašanja. Jedan od takvih pristupa jeste tzv. *stablo ponašanja* (eng. behavior tree) ali se ovime nećemo baviti. Ponašanja neprijatelja skriptovana su u Update funkciji. Date su Update funkcije koje opisuju ponašanje obe vrste neprijatelja. Uočavamo da se ne razlikuju suštinski.

```
// Attacker
private new void Update()
{
    base.Update();

    // Target does not exist
    if (!_starbase) return;

    // Move towards
    Vector2 targetDir =
        _starbase.transform.position - transform.position;

    // Rotate towards
    float dot =
        Vector2.Dot(transform.right, targetDir) + 0.5f;

    if (Mathf.Abs(dot) > float.Epsilon)
    {
        actions.Enqueue(new Rotate(-dot));
    }

    if (targetDir.magnitude >= 10f)
    {
        actions.Enqueue(new Thrust());
    }
    else
    {
        // Close enough to shoot
        if (Time.time - _lastFire > fireRate)
        {
            _lastFire = Time.time;
            actions.Enqueue(
```

```
        new AIShoot(Ship.Ammo.AmmoType.ROCKET, _guns));
    }
}

// Fighter
private new void Update()
{
    base.Update();

    // Move towards
    Vector2 targetDir = _target.position - transform.position;
    if (targetDir.magnitude >= 5f)
    {
        actions.Enqueue(new Thrust());
    }

    // Face target
    float dot = Vector2.Dot(transform.right, targetDir) + 0.5f;
    if (Mathf.Abs(dot) > float.Epsilon)
    {
        actions.Enqueue(new Rotate(-dot));
    }

    // Shoot
    if (targetDir.magnitude <= 15f &&
        Time.time - _lastFire >= fireRate)
    {
        _lastFire = Time.time;
        actions.Enqueue(
            new AIShoot(Ship.Ammo.AmmoType.DEFAULT_AI, _guns));
    }
}
```

3.6 Zaključak

Pojedini detalji oko korišćenja funkcija koje daje Unity platforma nisu objašnjeni jer se iz konteksta može zaključiti kada se koriste. Detalje je moguće lako pronaći na sajtu dokumentacije pretragom po imenu klase ili funkcije.

Za kompletnu implementaciju pogledati repozitorijum na github.com. Verzija korišćena za izradu ovog projekta jeste *Unity 2019.4.9f1 Personal*.

References

- [1] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN: 0990582906.
- [2] Unity Documentation. <https://docs.unity3d.com/Manual/index.html>.