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

Svemirska pucačina "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 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].

1.2 Unity Editor

Editor 2 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 podešavnja 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.

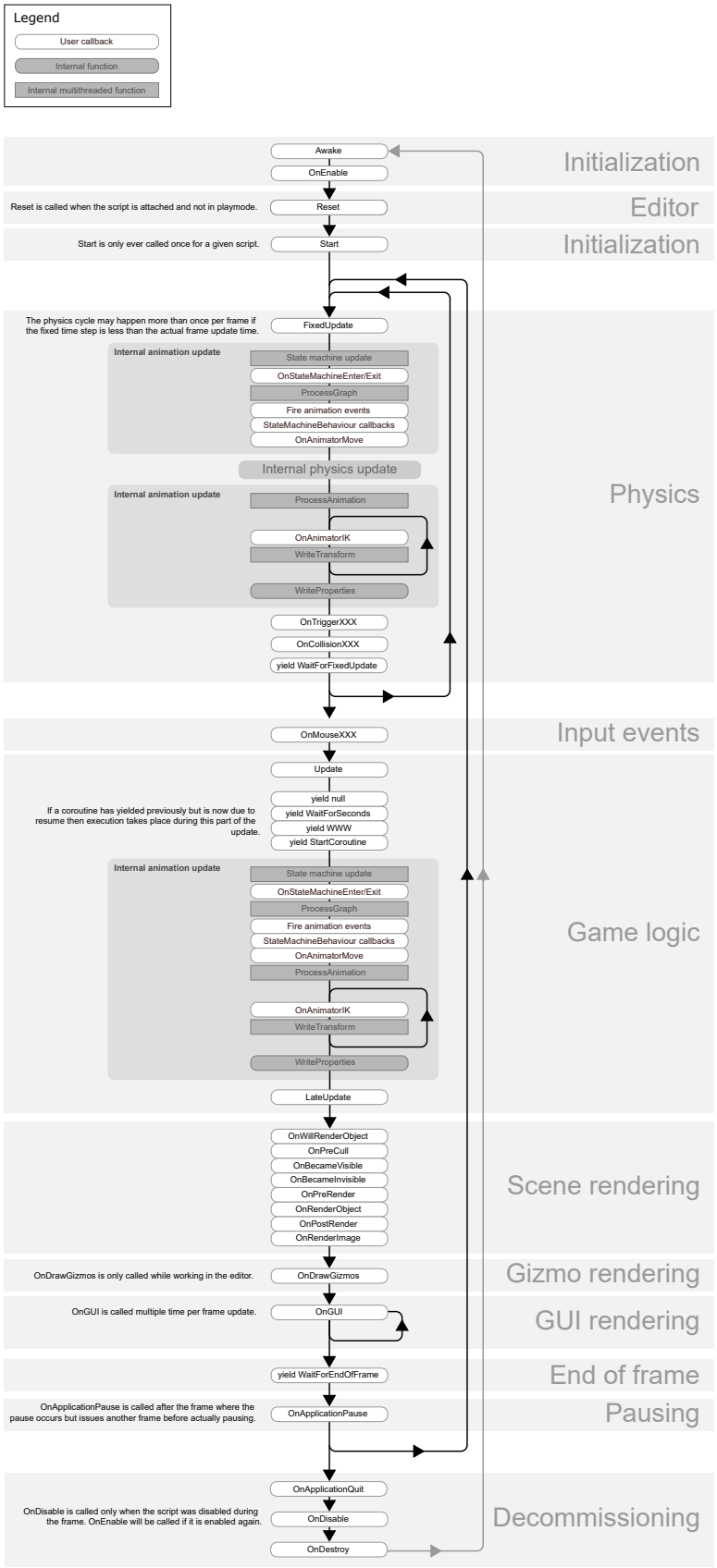

Figure 1: MonoBehaviour lifecycle

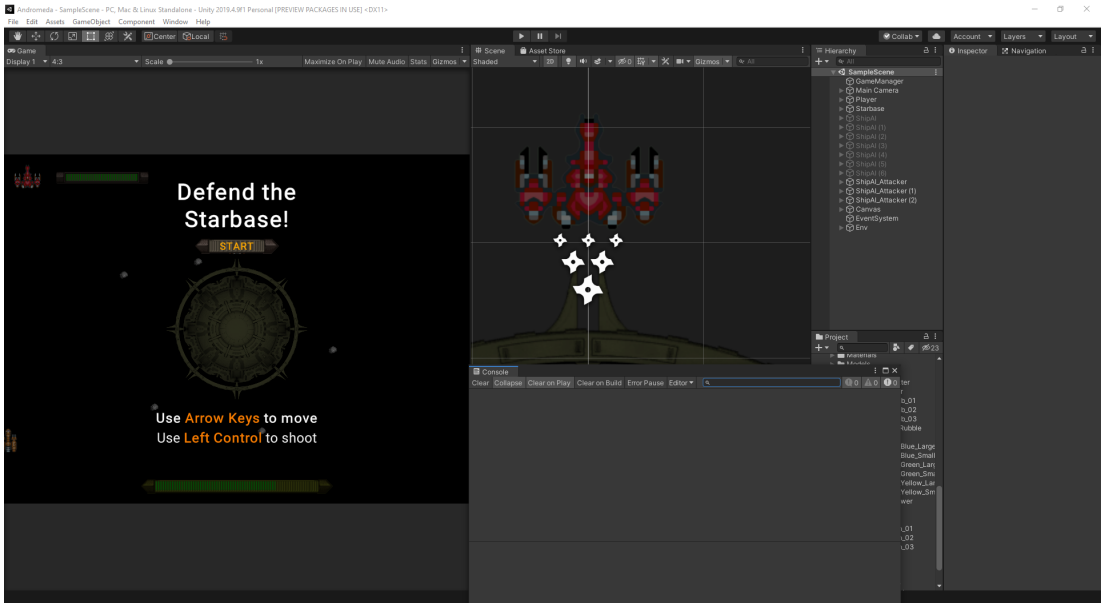

Figure 2: Unity Editor

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.

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 definsana 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 dokumentaicje, 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 pucnja.

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 znavči 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 3 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.

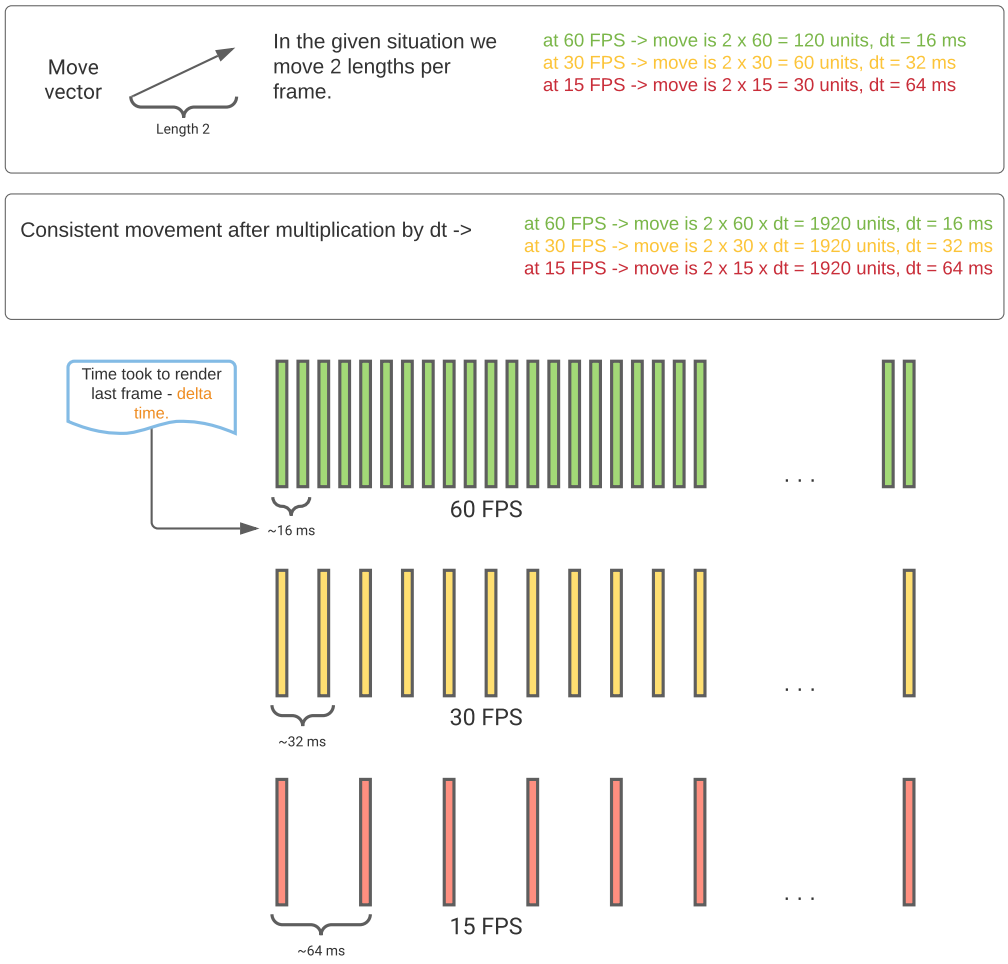

Figure 3: DeltaTime objašnjenje

- Inicijalizacija se dešava tokom izvšavanja programa (izbegavamo korišćenje čisto statičkih klasa).
- 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 pomoćsu statičkih klasa, stoga moramo koristiti Singleton kako bi smo osigurali globalni pristup ali pri tome sačuvati osobine MonoBehavior-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 comp
/// Example:
/// <code>
/// public class Foo : MonoSingleton<Foo>
/// </code>. To get the instance of Foo class, use <code>Foo.insta
/// Override <code>Init()</code> method instead of using <code>Awa
/// from this class.
/// </summary>
public abstract class MonoSingleton<T> : MonoBehaviour where T : M
{
    private static T m_Instance = null;
    public static T Instance
    {
```

```
    get
    {
        // Instance requiered for the first time, we look for
        if (m_Instance == null)
        {
            m_Instance = GameObject.FindObjectOfType(typeof(T)

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

                isTemporaryInstance = true;
                m_Instance = new GameObject("Temp Instance of

                // Problem during the creation, this should no
                if (m_Instance == null)
                {
                    Debug.LogError("Problem during the creatio
                }
            }
            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
```

```
    // 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() + "
            DestroyImmediate(this);
            return;
        }
        if (!_isInitialized)
        {
            DontDestroyOnLoad(gameObject);
            _isInitialized = true;
            m_Instance.Init();
        }
    }


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

    /// Make sure the instance isn't referenced anymore when the u
    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 ac
}
```

Dalje prema potebama 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 ac
        {
            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
InputManager.Instance.InputMap[KeyCode.RightArrow] = new Rotat
InputManager.Instance.InputMap[KeyCode.LeftControl] = new Shoo
    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

3.2 GameManager klasa

3.3 Kontrole

3.4 Ponašanje svemirskog broda i ShipController klasa

3.5 Neprijatelji i osnovno ponašanje

3.6 Poboljšanja i optimizacije

References

[1] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN: 0990582906.

[2] Unity. *Unity Documentation*.