1. Domaća zadaća

Predati do: 23. Listopad 2016.

RAUPJC 2016./2017.

Za prvu domaću zadaću potrebno je kreirati novi GIT repozitorij[1] u kojem ćete držati rješenje prve domaće zadaće (sa svim pripadajućim projektima).

Zadatak 1.

U sklopu prvog zadatka kreirat ćemo strukturu podataka koja će nam efikasno omogućiti dinamičko spremanje cijelih brojeva. Za spremanje grupe elemenata istog tipa do sad smo koristili polja. Polje kao struktura podataka dobro je u scenarijima gdje imamo statičan broj elemenata jer nam omogućuje brzo pisanje i čitanje elemenata u polju (O(1)), ali jednom definirana veličina polja ne može se mijenjati.
Na ASP-u radili su se dinamički spremnici podataka – liste. Logička veličina spremnika mijenjala se ovisno kako smo dodavali ili micali nove čvorove unutar liste. Imali smo brz pristup prvom i zadnjem elementu (O(1)), brzo dodavanje novog elementa liste (O(1)), ali nešto složeniji pristup ostalim elementima liste (O(N)) i njihovo brisanje u listi (O(N)).

U ovoj zadaći napravit ćemo hibrid navedene dvije strukture podataka. Implementirat ćemo takvu listu koja će interno koristiti polje kao svoj spremnik podataka, ali će se logički ponašati kao dinamički spremnik. To znači da će interno (van briga vanjskog korisnika) morati rješavati probleme prekoračenja veličine internog spremnika, izbacivanja elementa s određene pozicije i sl.

Javno sučelje klase opisano je u nastavku:

```
public interface IIntegerList {
    /// <summary>
    /// Adds an item to the collection.
    /// </summary>
    void Add(int item);

    /// <summary>
    /// Removes the first occurrence of an item from the collection.
    /// If the item was not found, method does nothing.
    /// </summary>
    bool Remove(int item);

    /// <summary>
    /// Removes the item at the given index in the collection.
    /// </summary>
    bool RemoveAt(int index);

    /// <summary>
    /// Returns the item at the given index in the collection.
    /// </summary>
    int GetElement(int index);

    /// <summary>
    /// Returns the index of the item in the collection.
    /// If item is not found in the collection, method returns -1.
    /// </summary>
    int IndexOf(int item);

    /// <summary>
```

[1] Naziv prepozitorija neka bude prikladan svrsi - npr. RAUPJC-DZ1

```
        /// Readonly property. Gets the number of items contained in the
            collection.
        /// </summary>
        int Count {get; }

        /// <summary>
        /// Removes all items from the collection.
        /// </summary>
        void Clear();

        /// <summary>
        /// Determines whether the collection contains a specific value.
        /// </summary>
        bool Contains(int item);
    }
```

Vaša zadaća je napraviti klasu IntegerList koja će implementirati navedeno sučelje na način da podatke čuva u internom spremniku u obliku polja cijelih brojeva.

```
public class IntegerList : IIntegerList
{
    private int[] _internalStorage;

    // ... IIntegerList implementation ...
}
```

Klasa izlaže javno dva konstruktora:

1. Konstruktor bez ulaznih argumenata koji inicijalizira privatan spremnik na veličinu od 4 elementa.
2. Konstruktor koji prima cijeli broj **initialSize** i kreira privatan spremnik veličine **initialSize**. Možete pretpostaviti da će operacija dodavanja elementa biti skup proces u slučaju prekoračenja interne veličine spremnika. Iz tog razloga dajemo korisniku mogućnost definiranja inicijalne veličine spremnika kako bi izbjegao potencijalne rekonstrukcije. Što ćete napraviti ako korisnik unese negativnu vrijednost?

Pseudokod za neke stavke javnog sučelja naveden je u nastavku.

```
Add(X)
{
    Ako(X ne stane u trenutni spremnik)
    {
        Pripremi novi, dvostruko veci, spremnik;
    }
    Dodaj X u spremnik na iducu slobodnu poziciju u spremniku;
}

RemoveAt(Index)
{
    Ako(Index ne pada unutar spremnika)
    {
        Vrati false;
    }
    Makni element na poziciji Index;
    Shift desnih elemenata u lijevo za jedno mjesto;
```

```
        Vrati true;
    }

    Remove(X)
    {
        Pronadi poziciju elementa X;
        Vrati RemoveAt(pozicija elementa X);
    }

    GetElement(Index)
    {
        Ako(Index pada unutar spremnika)
        {
            Vrati element na poziciji Index;
        }
        Inace
        {
            Baci IndexOutOfRangeException;
        }
    }

    Get Count
    {
        Vrati poziciju zadnjeg elementa u spremniku + 1;
    }
```

Primjer korištenja implementirane liste:

```
public void ListExample(IIntegerList listOfIntegers)
{
    listOfIntegers.Add(1);  // [1]
    listOfIntegers.Add(2);  // [1,2]
    listOfIntegers.Add(3);  // [1,2,3]
    listOfIntegers.Add(4);  // [1,2,3,4]
    listOfIntegers.Add(5);  // [1,2,3,4,5]

    listOfIntegers.RemoveAt(0); // [2,3,4,5]
    listOfIntegers.Remove(5); [2,3,4]
    Console.WriteLine(listOfIntegers.Count); // 3
    Console.WriteLine(listOfIntegers.Remove(100)); // false
    Console.WriteLine(listOfIntegers.RemoveAt(5)); // false
    listOfIntegers.Clear(); // []
    Console.WriteLine(listOfIntegers.Count); // 0
}
```

Zadatak 2.

Lista koju je implementirana u prvom zadatku ograničena je na cijele brojeve. U slučaju da trebamo istu strukturu podataka koja će primjerice primati realne brojeve, morali bi pripremiti novu implementaciju. Budući da je u C# moguće definirati beskonačan broj vlastitih tipova, vidimo da kreiranje nove implementacije za svaki tip nije opcija.

Pri pokušaju implementacije liste realnih brojeva, primijetili bi da je logika čitanja, pisanja i upravljanja podacima identična onoj iz prvog zadatka i da je kao takva neovisna o tipu kojim se upravlja. U sklopu drugog zadatka kreirat ćemo generičku implementaciju liste iz prvog zadatka koja će moći raditi nad podacima proizvoljnog tipa.

Proučiti C# Generics na MSDN-u: https://msdn.microsoft.com/en-us/library/ms379564(v=vs.80).aspx

```
public interface IGenericList <X>
{
    /// <summary>
    /// Adds an item to the collection.
    /// </summary>
    void Add(X item);

    /// <summary>
    /// Removes the first occurrence of an item from the collection.
    /// If the item was not found, method does nothing.
    /// </summary>
    bool Remove(X item);

    /// <summary>
    /// Removes the item at the given index in the collection.
    /// </summary>
    bool RemoveAt(int index);

    /// <summary>
    /// Returns the item at the given index in the collection.
    /// </summary>
    X GetElement(int index);

    /// <summary>
    /// Returns the index of the item in the collection.
    /// If item is not found in the collection, method returns -1.
    /// </summary>
    int IndexOf(X item);

    /// <summary>
    /// Readonly property. Gets the number of items contained in the
       collection.
    /// </summary>
    int Count {get; }

    /// <summary>
    /// Removes all items from the collection.
    /// </summary>
    void Clear();

    /// <summary>
    /// Determines whether the collection contains a specific value.
    /// </summary>
    bool Contains(X item);
}
```

Potrebno je implementirati klasu GenericList<X> koja implementira dano generičko sučelje:

```
public class GenericList <X> :IGenericList <X>
{

}
```

Zadatak 3.

Ako želimo iterirati nad kolekcijama osim klasičnih while, do while i for petlji, na raspolaganju imamo i foreach petlju. Foreach petlja kompatibilna je s poljima i svim .NET tipovima koji implementiraju sučelje IEnumerable. Ako u ovom trenu pokušamo iterirati po našoj generičkoj kolekciji foreach petljom, dobivamo grešku prevoditelja budući da naša lista ne implementira sučelje IEnumerable.

```
IGenericList < string > stringList = new GenericList <string> ();
stringList.Add("Hello");
stringList.Add("World");
stringList.Add("!");

foreach (string value in stringList)
{
    Console.WriteLine(value);
}

"Error:Containing type does not implement interface
   'System.Collections.IEnumerable'"
```

U sklopu trećeg zadatka potrebno je napraviti našu generičku listu kompatibilnu sa foreach petljom. Proširit ćemo generičko sučelje sučeljem IEnumerable.

```
public interface IGenericList<X> : IEnumerable<X>
```

Time smo definirali da tko god želi implementirati IGenericList sučelje, morat će ponuditi i implementaciju IEnumerable sučelja (jer smo ovime definirali IGenericList sučelje kao proširenje IEnumerable sučelja). Budući da naša generička list implementira IGenericList, dužna je ponuditi implementaciju IEnumerable sučelja. Rješenje je navedeno ispod:

```
public class GenericList <X> : IGenericList <X>
{
    // ...
    // IEnumerable<X> implementation
    public IEnumerator <T> GetEnumerator()
    {
        return new GenericListEnumerator <T> (this);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
    // ...
}
```

Vaš zadatak je implementacija GenericListEnumerator klase koju vraćamo u rješenju kao enumeratora naše kolekcije.

```
public class GenericListEnumerator<T> : IEnumerator<T>
{

}
```

Posao enumeratora je da hrani svaku iteraciju petlje s novom vrijednosti kolekcije. Da bi bolje razumjeli implementaciju enumeratora, ispod je prikazan kod enumeratora u akciji.

```
// foreach
foreach (string value in stringList)
{
    Console.WriteLine(value);
}

// foreach without the syntax sugar
IEnumerator<string> enumerator = stringList.GetEnumerator();
while (enumerator.MoveNext())
{
    string value = (string)enumerator.Current;
    Console.WriteLine(value);
}
```

Zadatak 4.

U sklopu četvrtog zadatka napravit ćemo zajedno klasičnu igru Pong s konceptima objektnog programiranja obrađenima na predavanju. Za odradu prezentacijskog dijela priče, koristit ćemo jednostavan game engine *Monogame* koji omogućuje razvoj igara u C# jeziku za gotovo sve zamislive platforme. Da vam nije palo na pamet predložiti Monogame igru za projekt nakon ove zadaće! Nije u okviru predmeta i nećete sigurno odgurati to do kraja.

Za početak, skinite Monogame na stranici: http://www.monogame.net/. Instalacijom proširujemo Visual Studio novim predlošcima za projekte. Dodajte novi Monogame projekt u postojeći solution (Preporuča se Monogame Windows Project, ali kod opisan u nastavku bit će neovisan o platformi).

Monogame se prevodi u izvršni asemblij, što vjerojatno znači da sada u solutionu imate dva izvršna projekta. Budući da odjednom možemo pokrenuti samo jedan, potrebno je označiti Pong kao aktivni izvršni projekt unutar solutiona (desni klik na Pong projekt –> Set as Startup Project).

Skinite 1-DZ.zip datoteku s http://www.fer.unizg.hr/predmet/raupjc. Sadržaj zipa prebacite u Content folder unutar Monogame projekta (pomognite si s desnim klikom na Pong projekt -> Open Project in File Explorer). Ovo su već pripremljeni resursi (grafika i audio) Pong aplikacije koju ćemo pisati.

Monogame predložak sastoji se od samo dvije klase:

- **Game1.cs**
  - Centralna klasa koja predstavlja našu igru. Ovdje je definirana sva logika igre.

- **Program.cs**
  - Aplikacijska klasa koja sadrži main metodu. Main metoda kreira i pokreće jednu instancu Pong igre. Ne morate se zamarati sadržajem ove klase.

Game1 klasa dolazi sa četiri predefinirane metode - *Initialize*, *LoadContent*, *Draw* i *Update*. Za njih kažemo da predstavljaju metode životnog ciklusa igre. U ovom trenutku nećemo se zamarati tko i kako zove metode - bitno nam je samo da razumijemo kada će se one pozivati.

- **Initialize** - metoda će se pozvati samo jednom za vrijeme trajanja igre i to na samom početku pokretanja. Ova metoda prilika je da inicijaliziramo sve objekte koje koristimo unutar igre.
- **LoadContent** - metoda će se pozvati samo jednom za vrijeme trajanja igre i to neposredno nakon Initialize metode. Ova metoda je prilika da učitamo u memoriju sav sadržaj koji koristimo unutar igre (grafiku, glazbu i dr.).
- **Draw** - metoda koja se u idealnom slučaju zove 60 puta u sekundi. Njezina jedina zadaća je da crta elemente na ekranu (odnosno, delegira taj zadatak grafičkoj kartici).
- **Update** - metoda koja se u idealnom slučaju zove 60 puta u sekundi. Njezina zadaća je da osvježi stanje elemenata koji se koriste u igri i kroz tu izmjenu stanja objekata implicitno utječe na kako se elementi prikazuju na ekranu. Do promjene stanje može doći ili korisničkom akcijom putem kontrolera ili kroz fizikalni model koji vlada igrom. Sva ta logika evaluira se unutar update metode.

Uloga *Initialize* i *LoadContent* metoda trebala bi biti jasna. Metode *Draw* i *Update* pokušat ćemo bolje objasniti na primjeru.

Na ekranu želimo prikazati element koji se kreće brzinom 60 piksela u sekundi. Kao što smo rekli - Draw metoda ima ulogu crtanja elemenata na ekranu, Update metoda ima ulogu promjene unutarnjeg stanja tih elemenata.

- Draw metoda crta objekt Element na ekranu na poziciji Element.X, Element.Y frekvencijom 60 puta u sekundi.
- Update metoda mijenja unutarnje stanje tog elementa i to na način da povećava njegovu X komponentu za jedan. Budući da se Update zove 60 puta u sekundi, efektivno smo postigli da će se vrijednost X komponente unutar raspona jedne sekunde promijeniti za 60.

Budući da Draw metoda crta stanje objekta svakim svojim pozivom, na ekranu nam je jasno vidljivo kako se unutarnje stanje elementa (pozicija X) mijenja u vremenu.

Za početak implementacije igre, definirat ćemo apstraktnu klasu Sprite. Sprite će predstavljati bazu za sve objekte naše igre koji će se znati nacrtati na ekranu.

```
public abstract class Sprite
{
    public float X { get; set; }
    public float Y { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }

    /// <summary>
    /// Represents the texture of the Sprite on the screen.
    /// Texture2D is a type defined in Monogame framework.
    /// </summary>
```

```
        public Texture2D Texture { get; set; }

        protected Sprite(int width, int height, float x = 0 , float y = 0)
        {
            X = x;
            Y = y;
            Height = height;
            Width = width;
        }

        /// <summary>
        /// Base draw method
        /// </summary>
        public virtual void DrawSpriteOnScreen(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(Texture, new Vector2(X, Y), new Rectangle(0, 0,
                Width, Height), Color.White);
        }
    }
```

Svaki element ekrana, tzv Sprite, karakterizira veličina na ekranu (width, height), pozicija na ekranu (x, y) i slika u službi grafičke reprezentacije na ekranu (texture). Sprite uz to nudi i DrawSpriteOnScreen metodu koja zna nacrtati sprite na ekranu koristeći njegovu poziciju, veličinu i sliku. Za pomoć u tom zadatku koristi se klasa SpriteBatch koja se predaje kao ulazni argumen metodi. SpriteBatch je klasa dostupna unutar Monogamea i ona zna za nas predati slike grafičkoj kartici na crtanje. Budući da primamo objekt specifičan za Monogame u Draw metodi Sprite objekta, efektivno smo ovako vezali našu klasu za Monogame implementaciju.

Sprite je označen kao apstraktan bez obzira što nema apstraktne članove (koncept je apstraktan). Ako želimo instancirati Sprite objekt, morat ćemo to napraviti preko izvedenih tipova poput Ball, Background, Paddle koji nisu apstraktni unutar konteksta igre Pong. U slučaju da izvedena klasa želi ponuditi svoju implementaciju Draw metode, može to slobodno napraviti budući da je Draw označena kao virtualna.

Kreirat ćemo i ostale izvedene tipove.

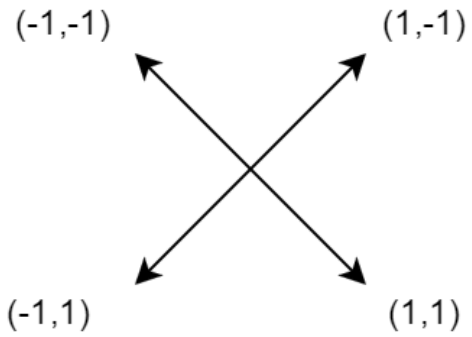
- Background - običan Sprite koji će reprezentirati pozadinu naše igre.
- Ball - Sprite koji će reprezentirati lopticu u Pong igri. Loptica za razliku od pozadine proširuje sprite implementaciju atributima poput "početna brzina", "faktor ubrzanja na udarac", "smjer kretanja" i sl. koji se vežu uz svaku lopticu u našoj igri.
- Paddle - Sprite koji reprezentira palicu igrača. Kao loptica, dolazi s dodatnim atributima poput "brzine kretanja", ali i svojom implementacijom Draw metode. Za razliku od ostalih Sprite objekata, palica će biti prikazana jednom bojom - "Ghost white".

```
/// <summary>
/// Game background representation
/// </summary>
public class Background : Sprite
{
    public Background(int width, int height) : base(width, height)
    {
```

```
        }
    }

    /// <summary>
    /// Game ball object representation
    /// </summary>
    public class Ball : Sprite
    {

        /// <summary>
        /// Defines current ball speed in time.
        /// </summary>
        public float Speed { get; set; }

        public float BumpSpeedIncreaseFactor { get; set; }

        /// <summary>
        /// Defines ball direction.
        /// Valid values (-1,-1), (1,1), (1,-1), (-1,1).
        /// Using Vector2 to simplify game calculation. Potentially
        /// dangerous because vector 2 can swallow other values as well.
        /// OPTIONAL TODO: create your own, more suitable type
        /// </summary>
        public Vector2 Direction { get; set; }

        public Ball(int size, float speed, float
            defaultBallBumpSpeedIncreaseFactor) : base(size, size)
        {
            Speed = speed;
            BumpSpeedIncreaseFactor = defaultBallBumpSpeedIncreaseFactor;
            // Initial direction
            Direction = new Vector2(1, 1);
        }
    }

    /// <summary>
    /// Represents player paddle.
    /// </summary>
    public class Paddle : Sprite
    {
        /// <summary>
        /// Current paddle speed in time
        /// </summary>
        public float Speed { get; set; }

        public Paddle(int width, int height, float initialSpeed) : base(width,
            height)
        {
            Speed = initialSpeed;
        }

        /// <summary>
        /// Overriding draw method. Masking paddle texture with black color.
        /// </summary>
```

```
        public override void DrawSpriteOnScreen(SpriteBatch spriteBatch)
        {
            spriteBatch.Draw(Texture, new Vector2(X, Y), new Rectangle(0, 0,
                Width, Height), Color.GhostWhite);
        }
    }
```

Smjer loptice opisujemo dvodimenzionalnim vektorom i smije poprimiti sljedeće vrijednosti za smjerove:



Primijetite da korisnik klase može postaviti smjer na neku nepodržanu vrijednost budući da smo ga deklarirali kao javnu varijablu. Opcionalno – kreirajte novi tip podataka koji će sigurnije opisati zadane 4 dimenzije smjera. Nadjačajte operatore množenja s Vector2 tipom (https://msdn.microsoft.com/en-us/library/6fbs5e2h.aspx). Rezultat množenja ili dijeljenja novog tipa s Vector2 tipom uvijek je Vector2 tip podatka. Npr.

$$(1,1) * (4,2) = (4,2)$$

$$(-1,1) * (7,6) = (-7,6)$$

$$(1,-1) * (2,3) = (2,-3)$$

$$(-1,-1) * (-9,9) = (9,-9)$$

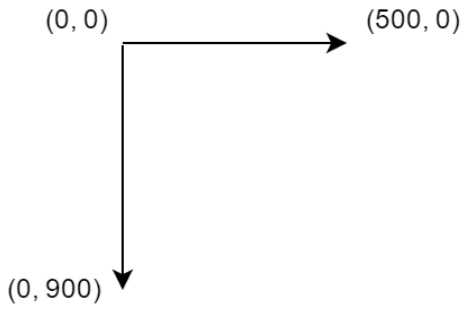
Imamo definirane sve tipove koji su nam potrebni za implementaciju. Prije nego što se prebacimo u Game klasu i napokon krenemo s implementacijom logike igre, kreirat ćemo klasu koja će držati tzv "magične brojeve" odnosno konstante koje koristimo unutar igre. Dobra je praksa definirati takve informacije na jednom mjestu.

```
public class GameConstants
{
    public const float PaddleDefaulSpeed = 0.9f;
    public const int PaddleDefaultWidth = 200;
    public const int PaddleDefaulHeight = 20;

    public const float DefaultInitialBallSpeed = 0.4f;
    public const float DefaultBallBumpSpeedIncreaseFactor = 1.05f;
    public const int DefaultBallSize = 40;
}
```

Dalje nastavljamo s Game1 klasom. Unutar konstruktora igre, postavit ćemo dimenzije unutar kojih se igra odvija.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this)
    {
        PreferredBackBufferHeight = 900,
        PreferredBackBufferWidth = 500
    };
    Content.RootDirectory = "Content";
}
```

Naš koordinatni sustav igre je sad postavljen je na sljedeći način:



Dodajte u Game1 klasu sljedeće članove:

- Referencu na donju palicu ekrana.
- Referencu na gornju palicu ekrana.
- Referencu na lopticu igre.
- Referencu na pozadinsku sliku igre.
- Generičku listu Spriteova koji će se crtati na ekranu.

```
/// <summary>
/// Bottom paddle object
/// </summary>
public Paddle PaddleBottom { get; private set; }

/// <summary>
/// Top paddle object
/// </summary>
public Paddle PaddleTop { get; private set; }

/// <summary>
/// Ball object
/// </summary>
public Ball Ball { get; private set; }

/// <summary>
/// Background image
/// </summary>
public Background Background { get; private set; }
```

```
/// <summary>
/// Sound when ball hits an obstacle.
/// SoundEffect is a type defined in Monogame framework
/// </summary>
public SoundEffect HitSound { get; private set; }

/// <summary>
/// Background music. Song is a type defined in Monogame framework
/// </summary>
public Song Music { get; private set; }

/// <summary>
/// Generic list that holds Sprites that should be drawn on screen
/// </summary>
private IGenericList<Sprite> SpritesForDrawList = new GenericList<Sprite>();
```

Kako bi mogli koristiti GenericList unutar Monogame projekta, potrebno mu je postaviti ovisnost (referencu) na projekt s GenericListom (References -> Add Reference). U slučaju da ne možete referencirati projekt, provjerite ciljaju li oba projekta istu verziju .NET-a (Desni klik na projekt -> Properties -> Target .NET Runtime).

U slučaju da ne možete referencirati jer ste GenericList definirali unutar nekog izvršnog projekta (npr. konzolne aplikacije), kreirajte novi projekt tipa ClassLibrary koji ćete moći referencirati te kopirajte implementaciju GenericList u njega.
Inicijalizaciju upravo kreiranih game objekata prikladno je raditi u initialize metodi.

```
protected override void Initialize()
{
    // Screen bounds details. Use this information to set up game objects
       positions.
    var screenBounds = GraphicsDevice.Viewport.Bounds;

    PaddleBottom = new Paddle(GameConstants.PaddleDefaultWidth,
       GameConstants.PaddleDefaulHeight, GameConstants.PaddleDefaulSpeed);
    PaddleBottom.X = <PaddleBottom X Position>;
    PaddleBottom.Y = <PaddleBottom Y Position>;

    PaddleTop = new Paddle(GameConstants.PaddleDefaultWidth,
       GameConstants.PaddleDefaulHeight, GameConstants.PaddleDefaulSpeed);
    PaddleTop.X = <PaddleTop X Position>;
    PaddleTop.Y = <PaddleTop Y Position>;

    Ball = new Ball(GameConstants.DefaultBallSize,
       GameConstants.DefaultInitialBallSpeed,
       GameConstants.DefaultBallBumpSpeedIncreaseFactor)
    {
        X = <Ball center X Position>
        Y = <Ball center Y Position>
    };

    Background = new Background(<width of the screen bounds>, <height of the
       screen bounds>);
```

```
        // Add our game objects to the sprites that should be drawn collection..
            you'll see why in a second
        SpritesForDrawList.Add(Background);
        SpritesForDrawList.Add(PaddleBottom);
        SpritesForDrawList.Add(PaddleTop);
        SpritesForDrawList.Add(Ball);

        base.Initialize();
    }
```

Posao inicializacije dovršavamo u LoadContent metodi gdje ćemo dohvatiti grafiku, audio i ostale resurse o kojima naša igra ovisi. U tu svrhu možemo koristiti Content.Load Monogame metodu koja zna dohvatiti resurse koje smo pripremili.

```
    protected override void LoadContent()
    {
        // Initialize new SpriteBatch object which will be used to draw textures.
        spriteBatch = new SpriteBatch(GraphicsDevice);


        // Set textures
        Texture2D paddleTexture = Content.Load<Texture2D>("paddle");
        PaddleBottom.Texture = paddleTexture;
        PaddleTop.Texture = paddleTexture;
        Ball.Texture = Content.Load<Texture2D>("ball");
        Background.Texture = Content.Load<Texture2D>("background");


        // Load sounds
        // Start background music
        HitSound = Content.Load<SoundEffect>("hit");
        Music = Content.Load<Song>("music");

        MediaPlayer.IsRepeating = true;
        MediaPlayer.Play(Music);
    }
```

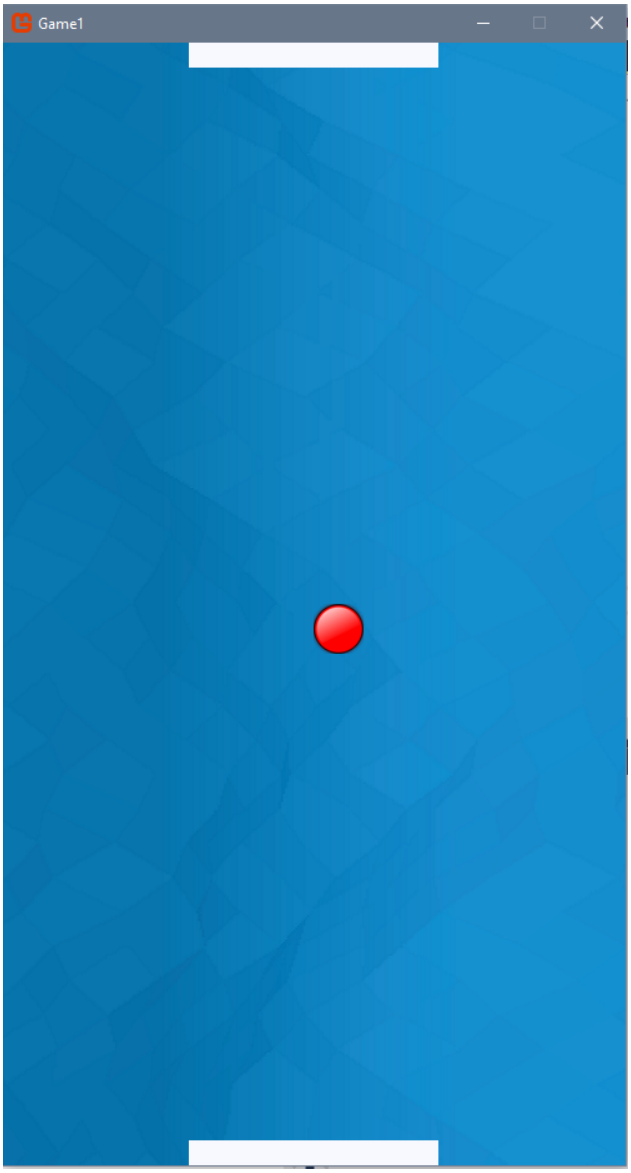
Inicijalizirali smo objekte koje koristimo u igri, učitali smo resurse (grafiku i audio). Sljedeća stvar je prikazati te elemente na ekranu. Vrijeme je da implementiramo Draw metodu igre. Implementacija metode je trivijalna. Iteriramo po listi Spriteova koje želimo vidjeti na ekranu (koju smo inicijalno napunili u initialize metodi) te zovemo njihovu DrawSpriteOnScreen metodu koja ih već zna nacrtati na ekranu.

```
    protected override void Draw(GameTime gameTime)
    {
        // Start drawing.
        spriteBatch.Begin();
        for (int i = 0; i < SpritesForDrawList.Count; i++)
        {
            SpritesForDrawList.GetElement(i).DrawSpriteOnScreen(spriteBatch);
        }

        // End drawing.
        // Send all gathered details to the graphic card in one batch.
```

```
        spriteBatch.End();
        base.Draw(gameTime);
    }
```

Ovako postavljena Draw metoda vrlo je fleksibilna. Npr. ako želimo dodati novi objekt na ekran, recimo nekakav power-up svakih 30 sekundi, dovoljno je samo gurnuti sprite u listu, ne moramo dirati implementaciju Draw metode. Zahvaljujući dinamičkom polimorfizmu, unatoč tome što zovemo DrawSpriteOnScreen nad pokazivačem na Sprite, poziva se DrawSpriteOnScreen metoda izvedene klase ako je ona definirana (npr. texture-ignore implementacija koju ima naša Paddle klasa).

Ako sada pokrenemo projekt možemo provjeriti da smo dobro konstruirali Spriteove u LoadContent metodi i da Draw metoda dobro crta postavljene objekte.



Objekti izgledaju u redu postavljeno. Objektima se trenutno stanje ne mijenja u vremenu zato imamo statičnu poziciju. Logika koja mijenja stanje objektima treba biti pozicionirana u Update metodi.
Prebacite se u update metodu: prva stvar koju ćemo implementirati jest upravljanje gornjim i donjim

reketom. Donjim reketom želimo upravljati strelicama lijevo i desno, dok gornjim reketom tipkama A i D. Na raspolaganju imamo Monogame klasu Keyboard koji zna vratiti trenutno stanje tipkovnice (koje su tipke aktivne, odnosno pritisnute).

```
var touchState = Keyboard.GetState();
(touchState.IsKeyDown(Keys.Left))
{
     ...
}
if (touchState.IsKeyDown(Keys.Right))
{
     ...
}
```

Ako unutar update metode evaluiramo da je pritisnuta tipku za lijevo, želimo pomaknuti poziciju donjeg reketa za neki iznos (određeno brzinom objekta). Budući da se update metoda zove neprestano za vrijeme trajanja igre, ako korisnik stisne i drži tipku za lijevo - 60 puta u sekundi poziva se update metoda, 60 puta u sekundi evaluiramo da je pritisnuta tipka za lijevo, 60 puta u sekundi pomaknemo reket za neki mali iznos u lijevo. Pozicija reketa linearna je funkcija vremena. Računamo je formulom:

$$x(t) = x_0 + v * t$$

odnosno

```
PaddleBottom.X = PaddleBottom.X - (float)(PaddleBottom.Speed *
    gameTime.ElapsedGameTime.TotalMilliseconds);
```

GameTime.ElapsedGameTime.TotalMilliseconds vratit će broj milisekundi koji je prošao od zadnjeg poziva update metode (16.6ms u idealnom slučaju, ali može varirati). Dakle, nova pozicija reketa bit će stara pozicija pomaknuta za brzina reketa * vrijeme od zadnje promjene pozicije.
Ako slučajno vrtite igru na nekom starijem računalu koje nije sposobno pozivati update metodu svakih 16.6 ms već sporije, množenjem s ElapsedGameTime si osiguravate zakoni fizike ostaju konzistentni u oba slučaja. Za 10 sekundi držanja tipke lijevo, reket će se pomaknuti za isti iznos na brzom računalu, ali i na sporom koje zove update metodu recimo samo 30 puta u sekundi.
Kada želimo ići desno:

```
PaddleBottom.X = PaddleBottom.X + (float)(PaddleBottom.Speed *
    gameTime.ElapsedGameTime.TotalMilliseconds);
```

Implementirajte upravljanje drugim igračem, a zatim pokrenite projekt. Možemo primijetiti da je s trenutnom implementacijom moguće reketima izaći van granica ekrana tako da je idući korak ograničavanje raspona vrijednosti X komponente reketa (neposredno nakon što smo postavili X komponentu u Update metodi).

U tu svrhu možemo koristiti Monogame tip MathHelper i njegovu metodu Clamp. Clamp metoda ograničava ulaznu vrijednost elementa na neki raspon. Prima 3 argumenta – ulaznu vrijednost, minimalnu vrijednost i maksimalnu vrijednost. Ako je ulazna vrijednost unutar zadanog raspona, metoda vraća ulaznu vrijednost. Inače vraća odgovarajuću granicu raspona.

```
PaddleBottom.X = MathHelper.Clamp(PaddleBottom.X, bounds.Left, bounds.Right -
    PaddleBottom.Width);
```

X komponenta ne smije ići lijevije od lijevog ruba ekrana (0), niti desnije od desnog ruba ekrana minus širina reketa. Implementirajte isto za gornji reket.

Sljedeće nas zanima kretanje loptice. Loptica se kreće konstantno dijagonalno, a već smo vidjeli kakve vrijednosti opisuju njezino kretanje. U update metodi želimo osvježavati njezinu poziciju sljedećim formulama:

$$x(t) = x_0 \pm D_x(v * t)$$

$$y(t) = y_0 \pm D_y(v * t)$$

$$D \in \{-1, 1\}$$

Budući da vektori u Monogameu imaju nadjačane (isto vrijedi za vaš tip, ako ste ga implementirali) operatore zbrajanja i množenja skalarima, kretanje loptice svodi se na jednostavan izraz:

```
var ballPositionChange = Ball.Direction *
    (float)(gameTime.ElapsedGameTime.TotalMilliseconds * Ball.Speed);
Ball.X += ballPositionChange.X;
Ball.Y += ballPositionChange.Y;
```

Pokrenite igru. Naša loptica trenutno se kreće, no kao što smo imali problem s reketima, nitko ne ograničava njezinu kretnju i izlazi van dimenzija igre. Ono što trebamo implementirati: ako loptica dotakne lijevi ili desni rub ekrana, kao da je udarila u zid, mora se odbiti u drugu stranu. Isto vrijedi i za rekete. U slučaju da je dotaknula u gornji ili donji zid, želimo resetirati poziciju loptice te uvećati eventualno rezultat nekom igraču. Kako opisati ovakav model?

- Ako ima koliziju sa rubom ekrana, želimo joj okrenuti X komponentu smjera. Loptica će poprimiti suprotan smjer kretanja i počet će se udaljavati od zida koji je udarila. Kut pod koji je loptica dotaknula zid je jednak kutu kojim će se udaljavati što savršeno odgovara fizikalnom modelu koji želimo opisati.
- Ako loptica ima koliziju s reketom, želimo joj okrenuti Y komponentu smjera. Loptica će se početi udaljavati od reketa kojeg je dotaknula. Sliča ideja kao s kolizijom s rubnim zidovima, samo što je druga dimenzija smjera u igri
- Ako loptica ima koliziju s gornjim ili donjim reketom, želimo resetirati njezinu poziciju na ekranu (vratiti je na centar ekrana).

Morat ćemo malo promjeniti model kako bi efikasno računali kolizije. Prije svega, potreban nam je jedan novi koncept - element igre koji ima dimenzije i poziciju unutar igre, a može kolidirati međusobno s drugim takvim elementima. Zvuči jako kao naš Sprite zbog postojanja pozicije i dimenzija u prostoru, ali ovim modelom trebali bi opisati i neke elemente koji se ne crtaju nužno na ekranu poput granica igre (zidovi). Napravit ćemo sljedeće sučelje:

```
public interface IPhysicalObject2D
{
    float X { get; set; }
    float Y { get; set; }
    int Width { get; set; }
    int Height { get; set; }
}
```

Ovo sučelje implementiraju svi objekti koji imaju poziciju i dimenzije unutar prostora naše igre. Zahvaljujući tim atributima, vrlo jednostavno možemo napraviti sljedeću klasu koja će nam omogućiti izračun kolizija (dodira / preklapanja) takvih objekata

```
public class CollisionDetector
{
    public static bool Overlaps(IPhysicalObject2D a, IPhysicalObject2D b)
    {
        // return true if overlaps, false otherwise...
    }
}
```

Budući da naši Spriteovi imaju sve atribute potrebne za implementaciju ovog sučelja i da sami sudjeluju u kolizijama (loptica može udariti reket i sl.), proširit ćemo naš Sprite na sljedeći način:

```
public abstract class Sprite : IPhysicalObject2D
```

Dalje trebamo modelirati elemente koji su nas i natjerali na promjenu modela - zidovi.

```
public class Wall : IPhysicalObject2D
{
    public float X { get; set; }
    public float Y { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }


    public Wall(float x, float y, int width, int height)
    {
        X = x;
        Y = y;
        Width = width;
        Height = height;
    }
}
```

Naša Pong aplikacija omeđena je sa četiri zida. Preostaje nam opis tih zidova unutar aplikacije. Dodat ćemo dvije liste u Game1.cs klasu, jedna će držati rubne zidove, a druga "pobjedničke" odnosno ciljne zidove iza pojedinih reketa. Inicijalizaciju zidova radimo u initialize metodi.

```
public List<Wall> Walls { get; set; }
public List<Wall> Goals { get; set; }


protected override void Initialize()
{

    // ...

    Walls = new List<Wall>()
    {
        // try with 100 for default wall size!
        new Wall(-GameConstants.WallDefaultSize,0,
            GameConstants.WallDefaultSize, screenBounds.Height),
        new Wall(screenBounds.Right,0, GameConstants.WallDefaultSize,
            screenBounds.Height),
```

```
        };

        Goals = new List<Wall>()
        {
            new Wall(0, screenBounds.Height, screenBounds.Width,
                GameConstants.WallDefaultSize),
            new Wall(screenBounds.Top,-GameConstants.WallDefaultSize,
                screenBounds.Width, GameConstants.WallDefaultSize),
        };
    }
```

Koliziju računamo trivijalno. Unutar update metode potrebno je vidjeti je li došlo do kolizije između rubnih zidova, pobjedničkih zidova ili reketa te odraditi odgovarajuću akciju koju smo definirali iznad.

```
    // Ball - side walls
    if (Walls.Any(w => CollisionDetector.Overlaps(Ball, w)))
    {
        Ball.Direction.X = -Ball.Direction.X;
        Ball.Speed = Ball.Speed * Ball.BumpSpeedIncreaseFactor;
    }
    // Ball - winning walls
    if (Goals.Any(w => CollisionDetector.Overlaps(Ball, w)))
    {
        Ball.X = bounds.Center.ToVector2().X;
        Ball.Y = bounds.Center.ToVector2().Y;
        Ball.Speed = GameConstants.DefaultInitialBallSpeed;
        HitSound.Play();
    }

    // Paddle - ball collision
    if (CollisionDetector.Overlaps(Ball, PaddleTop) && Ball.Direction.Y < 0
    || (CollisionDetector.Overlaps(Ball, PaddleBottom) && Ball.Direction.Y > 0))
    {
        Ball.Direction.Y = -Ball.Direction.Y;
        Ball.Speed *= Ball.BumpSpeedIncreaseFactor;
    }
```

Primjetite metodu *Any()* nad kolekcijom. Prima lambda funkciju (više na predavanju napredni C#) koja će vratiti bool ako se element kolekcije Walls preklapa s lopticom. Ako za barem jedan element lambda funkcija vrati true, metoda Any vraća true.

Pokrenite igru. Ostao je još jedan mali problem. Budući da loptica nakon nekog vremena postane prebrza za normalnu igru, trebamo ograničiti njezinu maksimalnu brzinu. Postavite maksimalnu moguću brzinu loptice na 1 (i pazite što s "magičnim brojevima"). Gdje je to najbolje napraviti? (maksimalna brzina je stvar loptice)

I s kolizijama i ograničavanjem brzine loptice uspješno ste finalizirali četvrti zadatak zadaće. Prostora za poboljšanje uvijek ima pa ambiciozni mogu pokušati proširiti igra s praćenjem rezultata pojedinog igrača ili čak dodavanjem različitih pojačanja unutar igre poput ubrzanje lopte, povećanje broja lopti na ekranu, smanjenje odnosno povećanje reketa jednog ili drugog igrača, obrambeni zid koji poništava pobjedničke udarce i sl.
:)