Izrada razvojnog okruženja za računalne igre u programskom jeziku C#

Sršen, Mirko

Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

Mirko Sršen

IZRADA RAZVOJNOG OKRUŽENJA ZA RAČUNALNE IGRE U PROGRAMSKOM JEZIKU C#

Diplomski rad

Pula, rujan 2020.

Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

Mirko Sršen

IZRADA RAZVOJNOG OKRUŽENJA ZA RAČUNALNE IGRE U PROGRAMSKOM JEZIKU C#

Diplomski rad

JMBAG: 0246059975, izvanredni student

Studijski smjer: Informatika

Predmet: Dizajn i programiranje računalnih igara

Znanstveno područje: društvene znanosti

Znanstveno polje: informacijske i komunikacijske znanosti

Znanstvena grana: informacijski sustavi i informatologija

Mentor: izv. prof. dr. sc. Tihomir Orehovački

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani **Mirko Sršen**, kandidat za **magistra informatike**, ovime izjavljujem da je ovaj diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio diplomskog rada nije napisan na nedozvoljen način, odnosno prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem također da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, 16. 9. 2020. godine

IZJAVA

o korištenju autorskog djela

Ja, **Mirko Sršen**, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom **Izrada razvojnog okruženja za računalne igre u programskom jeziku C#** koristi na način da gore navedeno autorsko djelo kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu sa Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 16. 9. 2020.

Potpis

_____

IZRADA RAZVOJNOG OKRUŽENJA ZA RAČUNALNE IGRE U PROGRAMSKOM JEZIKU C#

Mirko Sršen

Sažetak: U svrhu diplomskog rada implementiran je projekt koristeći se programskim jezikom C# u kojem je prikazan način na koji se može kreirati jednostavno razvojno okruženje za proizvodnju računalnih igara. Pri izradi projekta korištena je biblioteka MonoGame koja sadrži skup funkcija i naredbi za olakšano kreiranje virtualnog prostora, te podržava kreaciju računalnih igara na svim modernim konzolama (XBOX, PC, Play Station, Nintendo i sl.). Na samom početku rada nalaze se objašnjenja, teorije, ideje i strukture koje će u kasnijim poglavljima biti implementirane uz objašnjenja. Većinski dio rada su praktični dijelovi koda s objašnjenjima i idejama iza njih. Razumijevanje praktičnog dijela projekta zahtijeva visoku razinu znanja objektnog programiranja te prosječnu razinu matematičke teorije. Prvi dio praktičnog dijela bavit će se dvodimenzionalnim igrama dok će se drugi dio baviti trodimenzionalnim igrama. Naglasak u radu je na razvojno okruženje, a ne računalne igre tako da su primjeri igara korišteni samo za prikaz određene funkcionalnosti/sposobnosti razvojnog okruženja.

Ključne riječi: *game development*, C#, .NET, XNA, MonoGame

DEVELOPING GAME ENGINE IN C# PROGRAMMING LANGUAGE.

Mirko Sršen

Abstract: For the purpose of the thesis, a project was implemented using programming language C#, in which its displayed step by step guides how to build development environment for production of videogames. For creation of project library MonoGame was used and it contains collection of functions and commands for easy creation of virtual space and creation of videogames on all modern consoles (XBOX, PC, Play Station, Nintendo etc.). On the very beginning of the thesis there are theories, ideas and structures that will be explained and implemented in the later chapters of the thesis. Huge part of the thesis displays practical parts of code with explanation and ides behind them. Understanding of practical part of the code requires high level of knowledge about object-oriented programming and average understanding of mathematical theory. First part of practical part will be focused on the two-dimensional videogame development while the 2nd part will be focused on three-dimensional videogame development. Focus on thesis is on creating the development environment for videogames and it's not focused on making videogames, so all examples of videogames are used only for display of certain functionality/capabilities of development environment.

Keywords: Game development, C#, .NET, XNA, MonoGame

Sadržaj

1. Uvod

Industrija računalnih igara je u konstantnom porastu iz dana u dan. Prema zadnjim podatcima iz 2019. godine, računalne igre su zaradile više nego filmska i muzička industrija skupa, što ih postavlja na vrh piramide kao najprofitabilniju industriju za zabavu (engl. *entertainment*). Proporcionalno profitu koji igre ostvaruju raste i konkurencija na tržištu te sve kompanije pokušavaju dobiti bilo kakvu prednost nad konkurencijom. Sva snaga i konkurentnost kompanije nalazi se u njihovom razvojnom okruženju (engl. *game engine*), koji je često zatvoren za javnost te je specijaliziran za određeni žanr računalnih igara.

U diplomskom radu objašnjavaju se osnovna načela i tehnike koje se koriste prilikom kreiranja razvojnog okruženja za računalne igre. Glavni problemi s kojima se današnji programeri susreću su često vezani za performanse ili ponašanje određenog razvojnog okruženja, gdje se neznanje uzima kao glavni uzrok problema. Gotovo sva konkurentna razvojna okruženja na tržištu su zatvorenog koda te pristup implementaciji određenih funkcionalnosti unutar razvojnog okruženja nije dostupan krajnjem programeru. Jedini način da programer shvati način na koji određene komponente funkcioniraju i komuniciraju jest da sam pokuša izraditi svoje vlastito razvojno okruženje za izradu računalnih igara.

Stoga je glavna motivacija za pisanje ovog diplomskog rada edukacija i dokumentacija istraživanja o razvoju okruženja za izradu računalnih igara. Budući da rad ima svrhu istraživanja načina na koji razvojna okruženja funkcioniraju, a ne stvoriti komercijalno uporabljivo razvojno okruženje, korišten je programski jezik C# koji je idealan za programere svih razina i znanja. U prvim poglavljima govori se generalno o tehnikama, jeziku, idejama i bibliotekama koje će se koristiti pri izradi razvojnog okruženja. Prvi dio rada fokusira se na dvodimenzionalni razvoj računalnih igara dok se drugi dio odnosi na trodimenzionalni razvoj računalnih igara. Za potrebe dvodimenzionalnog razvoja bilo je potrebno definirati komponente: zvuk, koliziju, animaciju, teksturu i korisničko sučelje, dok smo za potrebe trodimenzionalnog razvoja definirali komponente: kameru, trodimenzionalni model i sustav čestica. Svaka komponenta posjeduje odgovarajući

primjer koda koji je prikazan u poglavljima: Implementacija prototipa 3D igre i Implementacija prototipa 2D igre. Valja naglasiti da je projektni kod pisan po uzoru na Unity razvojno okruženje, odnosno strukturalno su oba razvojna okruženja nalik jedan drugom.

2. C# / .NET programski jezik

2.1. Programski jezik C#

C# je programski jezik razvijen od Microsofta 2000. godine. Glavni cilj iza C# je bio kreiranje .NET okvira koji se danas uglavnom koristi za razvoj *web*-aplikacija. Glavne odlike C# su jednostavnost, kontroliran kod sa GC-om te snažna refleksija. S obzirom na to da je .NET jezik zatvorenog koda te je Microsoftov tim zadužen za pružanje svih novijih verzija i unaprjeđenja koda, 2016. godine se pojavila prva verzija .NET Corea koja je otvorenog koda te je proteklih godina počela zamjenjivati originalni .NET, ali i određeni dio tržišta drugih okvira/jezika.

C# je omogućio jako kvalitetnu kontrolu koda, lakoću korištenja i pisanja biblioteka, izbacio je nepotrebne stvari koje su u C++ developerima zadavale glavobolju te samim time ubrzao produkcijski ciklus aplikacija. C# ne bi postojao bez druge komponente pod nazivom CLI (engl. *Common Language Runtime*). CLI je izvršiteljska komponenta (engl. *Execution engine*) koja kontrolira životni ciklus aplikacije. CLI za glavnu zadaću ima:

- korištenje funkcionalnosti napisanih u drugim jezicima unutar C# (DLL import naredbe)
- strukturirano upravljanje greškama (stabla iznimki, vraćanje korak unazad, promjena vrijednosti varijabli na mjestima prekida)
- sigurnost pristupa kodu
- upravljanje nitima unutar programa, praćenje izvršavanja pojedinih niti
- ispravljanje grešaka (engl. *Debugging*)
- provjera grešaka tijekom prevođenja koda
- automatsko upravljanje memorijom pomoću GC-a.

2.2 Game development i C#

Većina razvojnih okruženja za kreaciju računalnih igara napisana je u programskom jeziku C++. Do dolaska Unitija i XNA biblioteke na scenu računalnih igara, C# je bio gledan kao jezik slabijih performansi te se kao takav nije mogao mjeriti sa C++, također C# je kasnio 15 godina za C++ te niti jedna biblioteka do dolaska DirectX-a nije podržavala direktno pisanje grafičkog koda u C#-u. Razvojem DirectX-a Microsoft 2006. godine kreira XNA s kojim želi stupiti na tržište računalnih igara te omogućiti developerima brzo i jednostavno kreiranje igara za Microsoftove platforme (Xbox, Windows OS, Windows mobiteli). Ideja XNA je bila omogućiti developerima maksimalan stupanj iskoristivosti koda na različitim platformama. XNA je sadržavao posebnu verziju koja je bila optimizirana za razvoj računalnih igara, te je i dalje omogućila kontrolirani razvoj koda. XNA je enkapsulacijom omogućio krajnjem korisniku pristup samo funkcijama na visokoj razini koje su bile potrebne za razvoj računalne igre, dok se za sve ostalo XNA brinuo u pozadini. XNA je bio namijenjen 2D i 3D razvoju te je omogućio korištenje tipkovnice i kontrolera za igranje.

Drugo razvojno okruženje je Unity koji je nastao 2004. godine te je u suštini trebao biti isključivo namijenjen razvoju računalnih igara za Mac i OS X operacijske sustave. Danas Unity podržava kreiranje računalnih igara za sve platforme te koristi C# kao glavni skriptni jezik pri tome. Budući da su performanse kod C# osjetno slabije nego kod C++ Unity razvija svoj kompilator (engl. *Compiler*) koji je po prvim testiranjima i do par puta brži od C++-a, razlog tome je što se svi objekti zamjenjuju strukturama koje su dostupne na *stacku* umjesto na *heapu* računala. Glavna prednost Burst kompilatora će biti to što će omogućiti deterministično obavljanje logike, fizike i sl. što će donijeti revoluciju unutar mrežnih računalnih igara iz razloga što serveri neće morati više obavljati svu logiku za svakog pojedinog igrača, već će se dio logike moći obavljati i na klijentskoj strani, dok će server uglavnom provjeravati ispravnost sesije s vremena na vrijeme.

2.3 MonoGame

MonoGame je jednostavna i jaka .NET biblioteka koja služi za kreiranje računalnih igara. Biblioteka podržava razvoj računalnih igara za osobna računala, platforme kao što su XBOX, PlayStation, Nintenddo Switch te mobilne uređaje.

Glavne stavke MonoGame biblioteke su:

- 2D i 3D razvoj računalnih igara
- zvučni efekti
- mogućnost korištenja tipkovnice, miša, kontrolera (engl. *joysticks*) ili dodirnog ekrana (engl. *touch screen*)
- kreiranje sadržaja i optimizacija koda unutar razvojnog okruženja za računalne igre
- posjedovanje matematičke biblioteke s optimiziranim funkcijama za potrebe razvoja računalnih igara.

MonoGame je baziran na XNA biblioteci te je implementacija funkcionalnosti uglavnom ista za obje biblioteke. Pri kreiranju ovog rada korištena je isključivo XNA dokumentacija, ali sve funkcionalnosti su implementirane pomoću MonoGamea. Budući da je MonoGame biblioteka otvorenog koda (engl. *Open source*), moguće je direktno pridonijeti razvoju pojedinih svojstava (engl. *feature*) na github repozitoriju „https://github.com/MonoGame/MonoGame".

2.4 Zašto MonoGame?

Iako je industrija računalnih igara toliko uznapredovala da je teško biti konkurentan na tržištu bez nekomercijalnih razvojnih okruženja poput Unityja i Unreala, potreba za inovacijom i osvježenjem u industriji je svakako potrebna. Dobar primjer inovacije u industriji bi bio Godot razvojno okruženje koje je također otvorenog koda te eksponencijalno raste u popularnosti od 2018. godine. Iako su sva komercijalna okruženja generalno daleko bolja i naprednija od bilo čega što mali tim ljudi može uraditi, developeri ne ciljaju na razvoj okruženja za komercijalnu uporabu već razmišljaju o

načinima kako napraviti okruženje koje će njima pomoći pri razvoju svojih ideja na puno brži i lakši način. MonoGame biblioteka omogućuje kreiranje vlastitog okruženja koristeći se funkcijama na visokoj razini (eng. *high level API*), što uvelike olakšava pisanje i razumijevanje koda koji se piše. Za razliku od komercijalnih okruženja, korištenje MonoGamea je besplatno za osobne i produkcijske potrebe. Također, MonoGame ne posjeduje nikakve restrikcije pisanja koda što uvelike olakšava odlikovanje arhitekture okruženja. U ovom projektu kod je pisan po uzoru na Unity razvojno okruženje što se može vidjeti iz primjera koda 1 (projektno razvojno okruženje) i 2 (Unity razvojno okruženje).

```
1.  protected override void InitComponents()
2.  {
3.      gameObject.AddComponent(new BoxCollider2D(0, 10, 10));
4.      //gameObject.AddComponent(new AudioSource(@"Content/sound/bgmusic.wav"));
5.      gameObject.AddComponent(new SpriteAnimation(50, @"Content/images/threerings.png",ne
    w Point(0,0), new Point(75, 75), 1, true));
6.      gameObject.transform.Position2D = new Vector2(100, 100);
7.
8.      var audioSource = gameObject.GetComponent<AudioSource>();
9.      audioSource.SoundInstance.IsLooped = true;
10.     audioSource.Play();
11.
12.     gameObject.GetComponent<BoxCollider2D>().OnCollisionEnter += a => method;
13. }
```

Primjer koda 1- Primjer pisanja koda u projektnom razvojnom okruženju

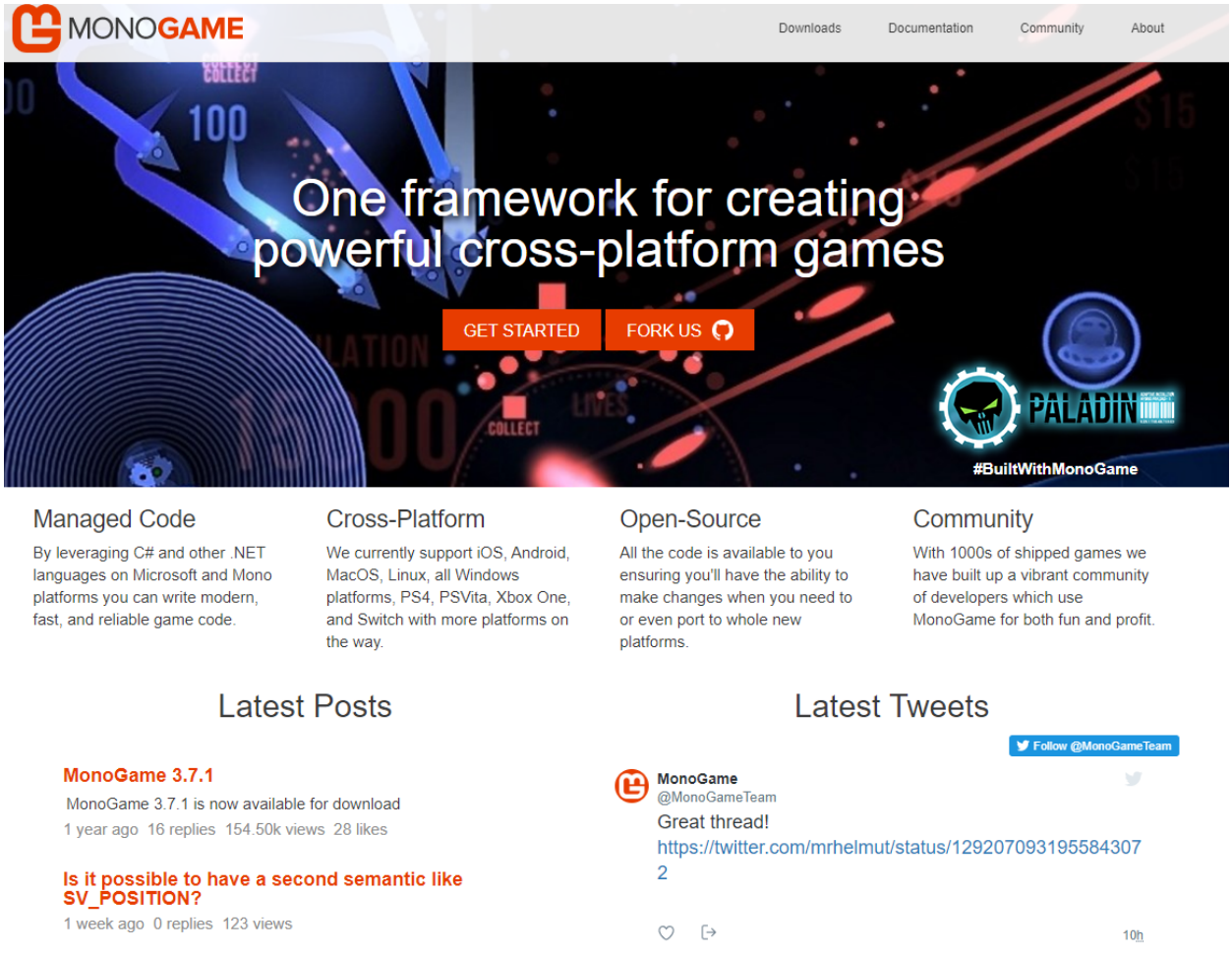
```
1.  public void InitComponents()
2.  {
3.      this.gameObject.AddComponent<BoxCollider2D>();
4.      this.gameObject.transform.position = new Vector2(100, 100);
5.
6.      var audioSource = gameObject.GetComponent<AudioSource>();
7.      gameObject.GetComponent<BoxCollider2D>().enabled = false;
8.  }
```

Primjer koda 2 – Primjer pisanja koda u Unity razvojnom okruženju

3. Instalacija MonoGame biblioteke

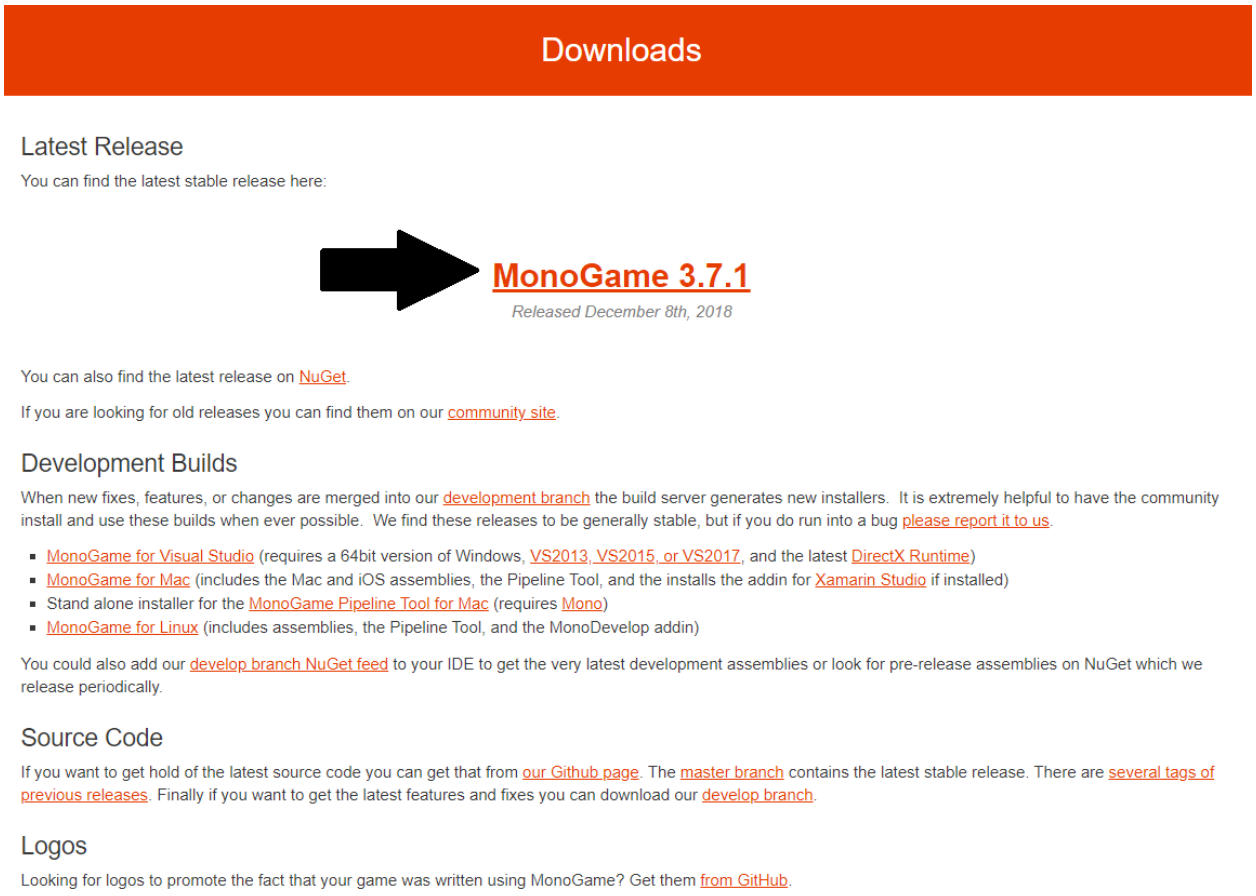
Prije početka samog rada s MonoGame bibliotekom potrebno ju je prvo instalirati na sljedeći način:

Posjetiti stanicu „https://www.monogame.net/” prikazanu na slici 1.



Slika 1. Izgled MonoGame *web*-stranice

Pritisnuti na dugme „Downloads“ koje preusmjerava na stranicu gdje se može preuzeti instalacija MonoGamea. Pritisnuti na zadnje izdanje (engl. *Latest Release*) označeno na slici 2. crnom strelicom.

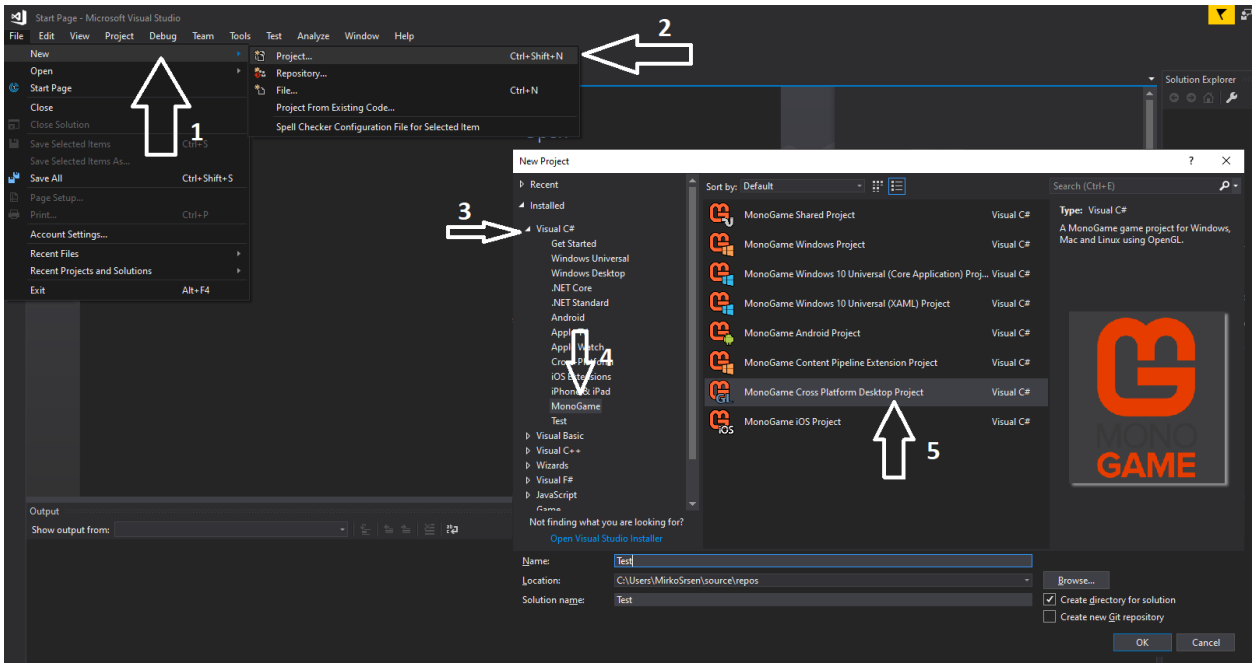


Slika 2. Link za preuzimanje biblioteke

Klikom na link otvara se novi prozor s više vrsta instalacije kao na slici 3. Odabrati instalacijski paket ovisno o vrsti operacijskog sustava i alata koji se koristi.

Slika 3. Ponuđena skidanja ovisna o platformi koja se koristi

Nakon preuzimanja instalacijskog paketa potrebno ga je pokrenuti te pratiti instalacijske upute. Za potrebe ovoga projekta pruzeli smo MonoGame 3.7.1 verziju za Visual studio. Nakon instalacije potrebno je otvoriti Visual studio te pritisnuti na karticu Datoteka>Novo>Projekt>Visual C#>MonoGame>MonoGame Cross Platform Desktop Project te dajemo naziv projektu te definiramo put na kojem ćemo spremiti projekt. Vizualni prikaz koraka nalazi se na slici 4.



Slika 4. Kreacija novog MonoGame projekta

4. Dizajn razvojnog okruženja

Aaron Reed u svojoj knjizi Learning XNA 4.0 govori o tome da postoji puno vrsta i načina dizajniranja razvojnog okruženja te su sve odluke pisanja uglavnom ostavljene na volju developeru koji ih razvija. Postoji zlatno pravilo kojeg bi se svaki developer trebao pridržavati, a to je pravilo konzistentnosti. Konzistentnost se odnosi na održavanje pravila koje developer definira sam sebi te primjenu tih istih pravila kroz cijeli razvoj okruženja. Ovime se olakšava donošenje odluka po pitanju smjera razvoja te se budućim korisnicima omogućuje lakše snalaženje budući da svaka komponenta unutar okruženja prati isti dizajn. Razvojno okruženje u ovom projektu bazira se na neovisnom komponentnom programiranju odnosno svaka skripta se promatra kao zasebna funkcionalnost unutar okruženja te niti jedna komponenta ne zavisi potpuno o drugoj. Sve komponente nasljeđuju baznu klasu koja definira osnovne funkcionalnosti koje svaka komponenta mora sadržavati. U primjeru koda 3 prikazana je implementacija bazne klase Component.

```
1.  using Kernel.Components;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.
5.  namespace Kernel
6.  {
7.      public abstract class Component
8.      {
9.          public Component()
10.         {
11.             Enabled = true;
12.             Visiable = true;
13.         }
14.
15.         public GameObject GameObject { get; set; }
16.
17.         public bool Enabled { get; set; }
18.
19.         public bool Visible { get; set; }
20.
21.         public abstract void LoadContent(GraphicsDevice graphicsDevice);
22.
23.         public abstract void Update(GameTime gameTime);
24.
25.         public abstract void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
    ;
26.
```

```
27.         public abstract void UpdateReferences();
28.
29.         public abstract void Destroy();
30.     }
31. }
```

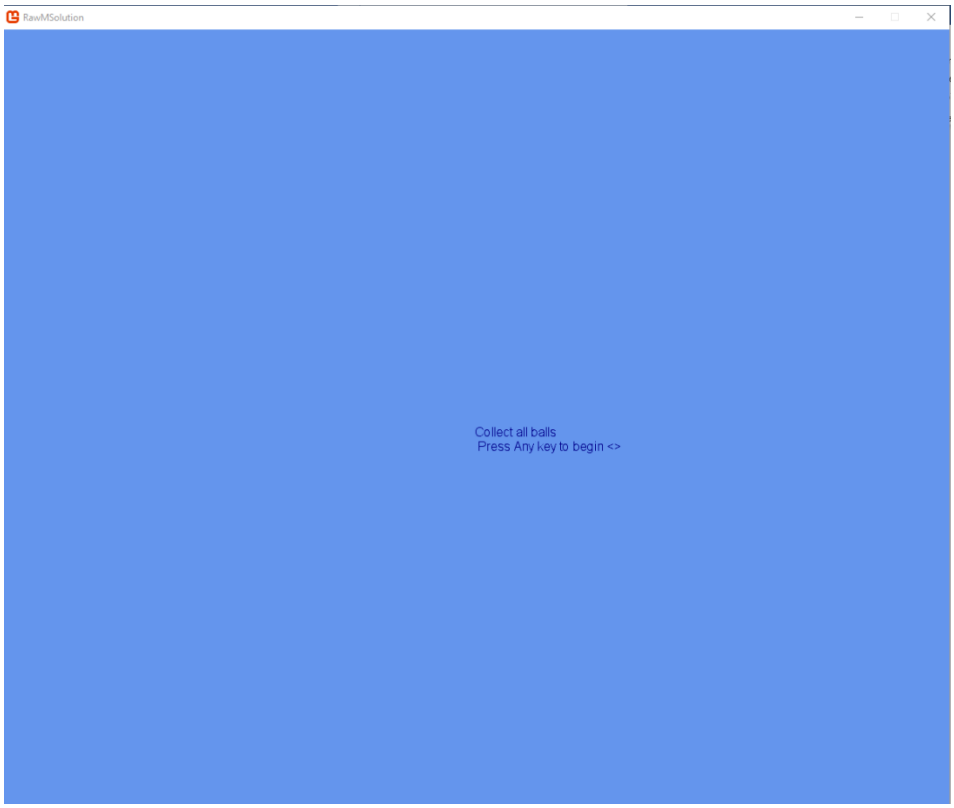
Primjer koda 3 - Implementacija klase Component

Klasa Component je apstraktna klasa što znači da se nikad ne može kreirati instanca objekta direktno pomoću nje, već se ona koristi kao kalup za svoju djecu koja ju nasljeđuju. U konstruktoru klase postavljamo dvije vrijednosti Enable i Visible na istinu (engl. True). Svrha GameObject, Enable i Visible varijabli će biti opisana u idućim poglavljima ovog rada. LoadContent funkcija služi za učitavanje sadržaja u komponentu, koristi se isključivo kada komponenta koristi neku datoteku koju treba učitati iz memorije računala i pripremiti za korištenje. Datoteke koje okruženje koristi su isključivo: font, slika, 3D modeli i zvukovi. *Update* funkcija ili funkcija ažuriranja ima ulogu izvršavanja logike koju je potrebno izvršavati više puta po sekundi. Broj izvršavanja ove funkcije uvelike ovisi o broju slika po sekundi (engl. *frames per second*). U slučaju da imamo 30 slika po sekundi, funkcija Update će se izvršavati maksimalno 30 puta u sekundi. Postoje slučajevi kada se u igri izvršava previše funkcija pa dolazi do pada broja slika po sekundi što se može primijetiti po „zapinjanju“ ili usporenosti računalne igre ili vizualnim problemima kao što je trganje ekrana (engl. *Screen tearing*). Draw funkcija kao i Update se izvršava u ovisnosti o broju slika u sekundi samo se izvršava isključivo nakon Updatea te služi isključivo za crtanje objekata po prostoru unutar igre. Za 2D development koristimo SpriteBatch koji služi za crtanje tekstura unutar prostora, dok za 3D koristimo Camera objekt koji služi za određivanje relativne pozicije trodimenzionalnog objekta unutar prostora igre u odnosu na promatrajući objekt (kameru). Update reference funkcija se koristi kako bismo ažurirali sve reference na komponenti koju kreiramo, odnosno prilikom mijenjanja vlasnika određene komponente potrebno je ažurirati sve njegove reference u odnosu na reference koje novi vlasnik posjeduje. Dobar primjer korištenja ove funkcije može se izvući iz stvarnog života, a odnosi se na kupnju auta. Prilikom kupnje auta svi dokumenti se prepisuju s jednog vlasnika na drugog odnosno polica osiguranja će i dalje vrijediti i imati iste funkcionalnosti samo će se referencirati na drugog vlasnika. Destroy funkcija služi za uništavanje/dekonstruiranje određene

komponente. U Destroyu uglavnom prvo pokušavamo zaustaviti rad funkcija Update i Draw, a nakon toga pokušavamo dealocirati sve reference kako bi GC što lakše očistio objekt iz memorije. U sljedećim poglavljima proći ćemo kroz sve komponente te ćemo na primjerima računalne igre prikazati određene funkcionalnosti.

5. Prototip 2D igre

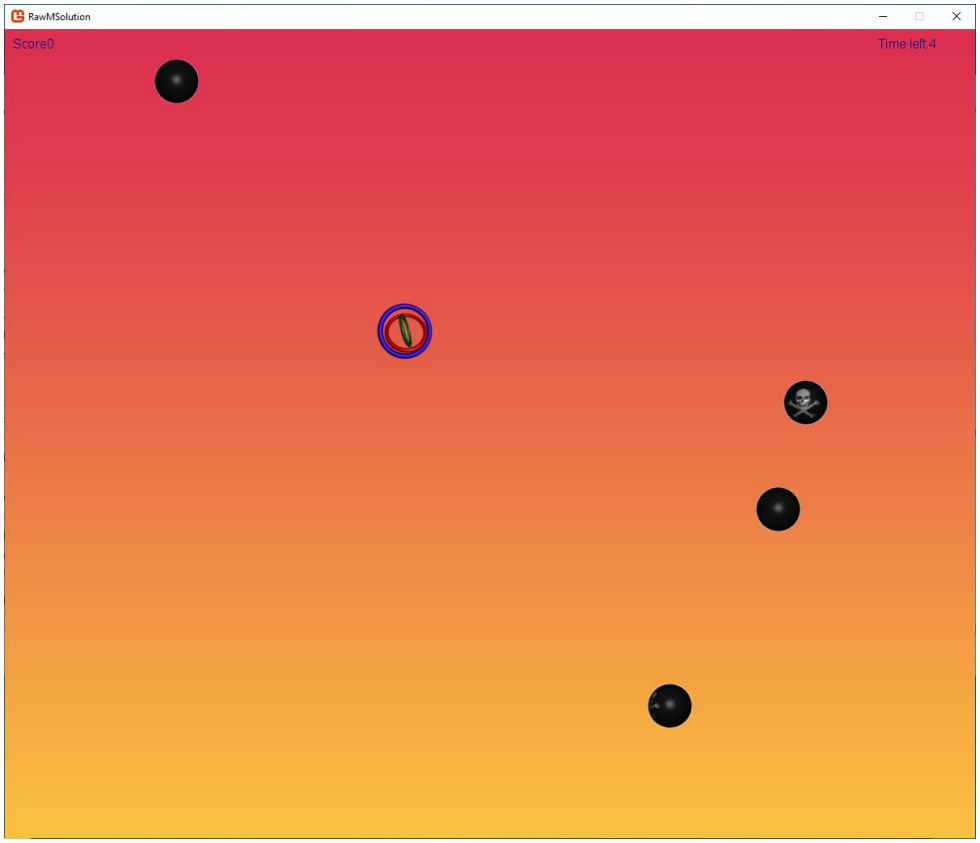
U sljedećim poglavljima bit će opisane komponente i implementacija dvodimenzionalne igre pod nazivom „Uhvati Loptu“. Igra je rađena s ciljem da se prikažu implementacije funkcionalnosti komponenti prikazanih u nastavku rada. Prilikom pokretanja igre prvi put korisnik je suočen s početnim zaslonom prikazanim na slici 5.



Slika 5. Početni zaslon računalne igre „Uhvati loptu”

Iako zaslon estetski ne izgleda najbolje, za potrebe prototipa je ovaj zaslon bio više nego dovoljan. Cilj ovog početnog zaslona je informirati igrača o cilju ove igre koji je uhvatiti što više lopti te pauzirati računalnu igru sve dok igrač ne pritisne bilo koju tipku kako bi potvrdio da je spreman. Klikom bilo koje tipke pokreće se iduća faza igre te igrač počinje

aktivno igrati igru. Na slici 6. prikazan je izgled zaslona računalne igre. Valja naglasiti da se u gornjem lijevom kutu nalazi vrijednost „Score“ koja ima za zadaću brojati bodove koje je igrač ostvario, a u desnom gornjem kutu se nalazi vrijednost „Time Left“ koja označava broj sekundi koje su preostale igraču prije nego igra završi. Igrač ostvaruje bodove kolizijom s crnim loptama od kojih svaka vrijedi 100 bodova.

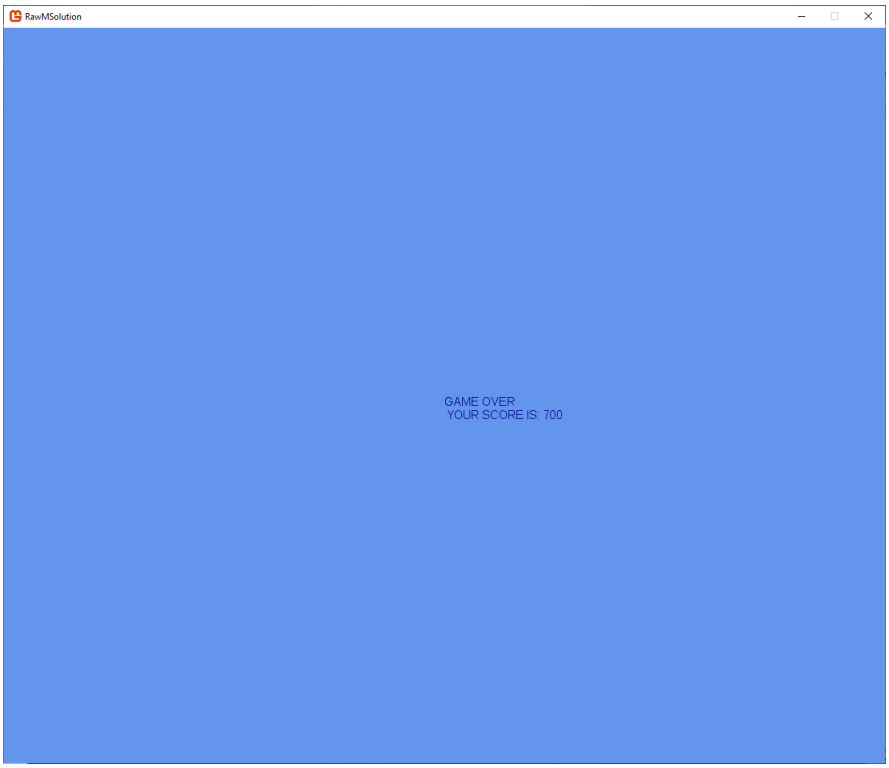


Slika 6. Prototip 2D igre

Također prilikom kolizije crne lopte ispuštaju zvuk koji signalizira da je igrač dodirnuo loptu te je prototip moguće igrati pomoću miša, tipkovnice ili kontrolera. Budući da se igrač može kretati samo unutar granica početno definiranog zaslona, sve lopte koje izađu iz tog prostora potrebno je uništiti kako se performanse računalne igre nakon dužeg igranja ne bi narušile. Svaka lopta unutar igre posjeduje animacijsku komponentu koja daje dodatan vizualni ugođaj igri prilikom igranja. Lopte također posjeduju primitivnu umjetnu inteligenciju te posjeduju 3 moguća stanja ponašanja. Prvo stanje je stanje bježanja u kojem lopta pokušava otići što dalje od igrača. Drugo stanje je stanje lovljenja

u kojem lopta pokušava uhvatiti igrača, treće stanje je stanje mirovanja u kojem lopta stoji na mjestu te se ne pokušava micati.

Nakon što vrijeme istekne, faza igranja završava te se iscrtava finalni zaslon koji signalizira da je igra završila. Na njemu se ispisuje tekst koji govori da je igra završila te ukupan broj ostvarenih bodova. Na slici 7. prikazan je finalni zaslon.



Slika 7. Zaslon koji označava kraj igre

6. Detekcija ulaznih naredbi

Svaka računalna igra mora posjedovati određenu razinu interakcija s igračem, što je u suštini i čini drugačijom od drugih medija poput filmova i muzike. Bio to klik miša, pritisak tipke ili jednostavno pokret ruke (kad pričamo o virtualnoj realnosti), ulazna naredba mora utjecati na razvoj događanja unutar računalne igre. Ulazne naredbe moraju biti jednostavne i responzivne kako bi igračevo iskustvo bilo što ljepše i ugodnije. U ovom projektu implementiran je generički ulazni sustav (engl. *input system*) koji će nam kroz nastavak igre omogućiti što brže i lakše čitanje korisnikovih instrukcija. Ulazni sustav je statičan za cijeli projekt te sadrži implementaciju prikazanu u primjeru koda 4.

```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Input;
3.
4.  namespace Kernel
5.  {
6.      public static class InputController
7.      {
8.          private static KeyboardState keyboardState { get { return Keyboard.GetState();
    } }
9.
10.         private static MouseState mouseState { get { return Mouse.GetState(); } }
11.
12.         private static MouseState lastMouseState { get; set; }
13.
14.         public static bool GetKeyDown(Keys key)
15.         {
16.             return keyboardState.IsKeyDown(key);
17.         }
18.
19.         public static bool GetKeyUp(Keys key)
20.         {
21.             return keyboardState.IsKeyUp(key);
22.         }
23.
24.         public static bool IsMouseActive()
25.         {
26.             return lastMouseState != mouseState;
27.         }
28.
29.         public static Vector2 GetMousePosition()
30.         {
31.             if (lastMouseState != mouseState)
32.             {
33.                 lastMouseState = mouseState;
34.             }
35.
36.             return new Vector2(mouseState.X, mouseState.Y);
```

```
37.         }
38.
39.         public static bool GetLeftMouseDown()
40.         {
41.             return mouseState.XButton1 == ButtonState.Pressed;
42.         }
43.
44.         public static bool GetRightMouseDown()
45.         {
46.             return mouseState.XButton2 == ButtonState.Pressed;
47.         }
48.
49.         public static bool GetGamePadKeyDown(Buttons button, PlayerIndex playerIndex =
    PlayerIndex.One)
50.         {
51.             var gamePadState = GamePad.GetState(playerIndex);
52.
53.             return gamePadState.IsButtonDown(button);
54.         }
55.
56.         public static bool GetGamePadKeyUp(Buttons button, PlayerIndex playerIndex = Pl
    ayerIndex.One)
57.         {
58.             var gamePadState = GamePad.GetState(playerIndex);
59.
60.             return gamePadState.IsButtonUp(button);
61.         }
62.
63.         public static Vector2 GetGamePadLeftStickPosition(PlayerIndex playerIndex = Pla
    yerIndex.One)
64.         {
65.             var gamePadState = GamePad.GetState(playerIndex);
66.
67.             return gamePadState.ThumbSticks.Left;
68.         }
69.
70.         public static Vector2 GetGamePadRightStickPosition(PlayerIndex playerIndex = Pl
    ayerIndex.One)
71.         {
72.             var gamePadState = GamePad.GetState(playerIndex);
73.
74.             return gamePadState.ThumbSticks.Left;
75.         }
76.     }
77. }
```

Primjer koda 4 - Implementacija komponente InputController

Kao što možemo vidjeti iz primjera koda iznad, klasa InputController.cs služi za detekciju ulaznih podataka pomoću miša, tipkovnice i kontrolera. Vrijednosti keyboardState služi za dohvaćanje trenutnog stanja tipkovnice, odnosno trenutno pritisnute tipke na tipkovnici. Valja napomenuti da će ovo polje isključivo vraćati zadnju pritisnutu tipku u slučaju da su istovremeno dvije tipke pritisnute. Vrijednosti mouseState i lastMouse state

služe za dohvaćanje vrijednosti trenutnog stanja miša i prijašnjeg stanja miša. Prijašnje stanje miša koristimo kako bismo izračunali deltu odnosno razliku pozicije miša iz prijašnjeg ažuriranja i trenutnog ažuriranja. Funkcija GetKeyDown kao ulazni parametar prima šifru tipke te kao rezultat vraća zastavu (engl. flag) koja nam govori je li ta tipka pritisnuta ili ne. Funkcija GetKeyUp kao ulazni parametar prima šifru tipke te kao rezultat vraća vrijednosti koje nam govore je li ta tipka prestala biti pritisnuta ili ne. IsMouseActive funkcija radi provjeru koristi li se miš tijekom igranja na način da uspoređuje prijašnje i trenutno stanje miša. U slučaju da je prijašnje i trenutno stanje različito, možemo pretpostaviti da se miš koristi. Funkcija GetMousePosition nam vraća trenutno poziciju miša na ekranu. Funkcija GetLeftMouseDown vraća bool vrijednost koja govori da li je lijeva tipka na mišu pritisnuta ili ne. Funkcija GetRightMouseDown vraća zastavu koja govori je li desna tipka na mišu pritisnuta ili ne. Funkcija GetGamepadKeyDown kao ulazne parametre prima šifru tipke na kontroleru te indeks igrača za kojeg se vrši provjera. Kao rezultat vraća zastavu koja nam govori je li na specifičnom kontroleru pritisnuta specifična tipka. Funkcija GetGamepadKeyUp kao ulazni parametar prima šifru tipke i indeks igrača za kojeg se vrši provjera. Kao rezultat vraća vrijednosti koje nam govore je li na specifičnom kontroleru prestala biti pritisnuta određena tipka. Funkcija GetGamePadLeftStickPosition kao ulazni parametar prima indeks igrača za kojeg se provjera vrši te vraća dvodimenzionalni vektor koji nam govori koja je normalizirana pozicija lijevog analognog štapa (engl. *analog stick*). Funkcija GetGamePadRightStickPosition kao ulazni parametar prima indeks igrača za kojeg se provjera vrši te vraća dvodimenzionalni vektor koji nam govori koja je normalizirana pozicija desnog analognog štapa.

Primjer korištenja ovih funkcija nalazi se u primjeru koda 5 (bez dodatnih funkcionalnosti koje će biti opisane u sljedećim poglavljima).

```
1.  if (InputController.GetKeyDown(Keys.A))
2.  {
3.  }
4.  if (InputController.GetKeyDown(Keys.D))
5.  {
6.  }
7.  if (InputController.GetKeyDown(Keys.W))
8.  {
```

```
9.  }
10. if (InputController.GetKeyDown(Keys.S))
11. {
12. }
13.
14. if (InputController.IsMouseActive())
15. {
16. }
17.
18. var position = InputController.GetGamePadLeftStickPosition();
19. if (InputController.GetGamePadKeyDown(Buttons.A))
20. {
21. }
```

Primjer koda 5 - Korištenje komponente InputController

7. Objekt unutar računalne igre

Svaku računalnu igru možemo gledati kroz objekte, odnosno svaka slika, tekstura, zvuk i sl. može biti definirana kao jedinstveni objekt. U našem dizajnu objekt je temelj sve logike u računalnoj igri odnosno svaka komponenta koja nije dio nekog objekta neće biti inicijalizirana i ažurirana. U ovom projektu objekt možemo definirati kao skup komponenti koje dijele isti cilj/svrhu, tako da će objekt igrača na sebi uvijek imati komponente isključivo potrebne za igrača, dok će objekt neprijatelja na sebi imati uvijek isključivo komponente potrebne za rad neprijatelja. Implementacija klase GameObject unutar razvojnog okruženja opisana je u primjeru koda 6.

```
1.  using Kernel.Components;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.  using System.Collections.Generic;
5.
6.  namespace Kernel
7.  {
8.      public class GameObject : Component
9.      {
10.         public Transform transform { get; set; }
11.
12.         public string Name { get; set; }
13.
14.         internal List<Component> Components { get; set; }
15.
16.         public GameObject()
17.         {
18.             this.Components = new List<Component>();
19.             transform = new Transform();
20.             this.AddComponent(transform);
21.         }
```

```
22.
23.         public GameObject(string name)
24.             :this()
25.         {
26.             this.Name = name;
27.         }
28.
29.         public override void LoadContent(GraphicsDevice graphicsDevice)
30.         {
31.             for(int i = 0; i< Components.Count; i++)
32.             {
33.                 Components[i].LoadContent(graphicsDevice);
34.             }
35.         }
36.
37.         public TComponent GetComponent<TComponent>() where TComponent : Component
38.         {
39.             for(int i = 0; i< this.Components.Count; i++)
40.             {
41.                 if(this.Components[i] is TComponent)
42.                 {
43.                     return (TComponent)this.Components[i];
44.                 }
45.             }
46.
47.             return null;
48.         }
49.
50.         public void AddComponent(Component component)
51.         {
52.             this.Components.Add(component);
53.             component.GameObject = this;
54.
55.             for (int i = 0; i < Components.Count; i++)
56.             {
57.                 this.Components[i].UpdateReferences();
58.             }
59.         }
60.
61.         public void RemoveComponent(Component component)
62.         {
63.             this.Components.Remove(component);
64.             component.Destroy();
65.
66.             for (int i = 0; i < Components.Count; i++)
67.             {
68.                 this.Components[i].UpdateReferences();
69.             }
70.         }
71.
72.         public override void Update(GameTime gameTime)
73.         {
74.             if (!Enabled) return;
75.
76.             for (int i =0; i< Components.Count; i++)
77.             {
78.                 Components[i].Update(gameTime);
79.             }
80.         }
81.
```

```
82.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
83.         {
84.             if (!Enabled || !Visiable) return;
85.
86.             for (int i = 0; i < Components.Count; i++)
87.             {
88.                 Components[i].Draw(spriteBatch, camera);
89.             }
90.         }
91.
92.         public override void UpdateReferences()
93.         {
94.         }
95.
96.         public override void Destroy()
97.         {
98.             for (int i = 0; i < Components.Count; i++)
99.             {
100.                        Components[i].Destroy();
101.                    }
102.                    Components.Clear();
103.                }
104.            }
105.        }
```

Primjer koda 6 - Implementacija komponente GameObject

Budući da svaki objekt može biti postavljen unutar prostora računalne igre, definirali smo dodatnu klasu Transform, čija je implementacija prikazana u primjeru koda 7.

```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Graphics;
3.
4.  namespace Kernel.Components
5.  {
6.      public class Transform : Component
7.      {
8.          public Vector2 Position2D;
9.
10.         public Vector3 Position3D;
11.
12.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
13.         {
14.             if (!Enabled || !Visible) return;
15.         }
16.
17.         public override void LoadContent(GraphicsDevice graphicsDevice)
18.         {
19.         }
20.
21.         public override void Update(GameTime gameTime)
22.         {
23.             if (!Enabled) return;
24.         }
```

```
25.
26.         public override void UpdateReferences()
27.         {
28.         }
29.
30.         public override void Destroy()
31.         {
32.             this.GameObject.Components.Remove(this);
33.         }
34.     }
35. }
```

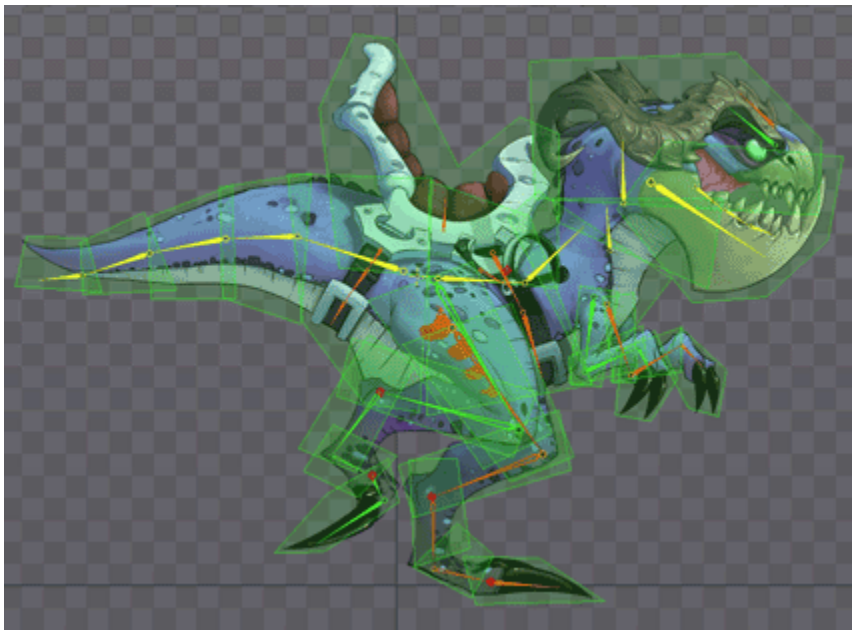
Primjer koda 7 – Implementacija komponente Transform

Transform je komponenta koja služi da određivanje pozicije objekta unutar prostora scene. Vrijednosti Position2D i Position3D služe za određivanje pozicije objekta u 2D i 3D prostoru. Funkcije koje Transform nasljeđuje od klase Komponent nemaju neku bitnu funkcionalnost osim funkcije Destroy koja služi za brisanje komponente s objekta na kojem se koristi.

U konstruktoru klase GameObject inicijaliziramo novu praznu listu komponenti kojoj zatim dodajemo novu instancu Transform klase. Funkcija LoadContent prima klasu GraphicDevice koja služi za crtanje tekstura, kompleksnih i primitivnih oblika u prostoru računalne igre. Bez klase GraphicDevice niti jedan objekt u računalnoj igri ne bi bio vidljiv. Funkcija GetComponent<T> je generička funkcija zadužena za dohvaćanje komponente specifičnog tipa koji se nalazi na objektu. Generička vrijednost uvijek mora biti tipa Component, a u slučaju da komponenta tog tipa ne postoji vraća se prazna vrijednost (*null*), a u slučaju da postoji više komponenti istoga tipa, uvijek će se vratiti komponenta koju iteracija prva dohvati. Funkcija AddComponent kao parametar prima objekt tipa Component te ga dodaje u listu komponenti objekta. Funkcija RemoveComponent briše određenu komponentu iz liste komponenti na objektu. Također, poziva se funkcija Destroy nad komponentom kako bi se počistile sve reference i relacije s komponentom prije nego se ona izbriše iz memorije. Na preostalim komponentama se zatim poziva funkcija UpdateReferences koja ažurira relacije između komponenti kako ne bi došlo do pokušaja pristupa objektu koji ne postoji. Funkcije LoadContent, Update, Draw i Destroy pozivaju ekvivalentne funkcije unutar komponenti.

8. Animacije, teksture i zvukovi

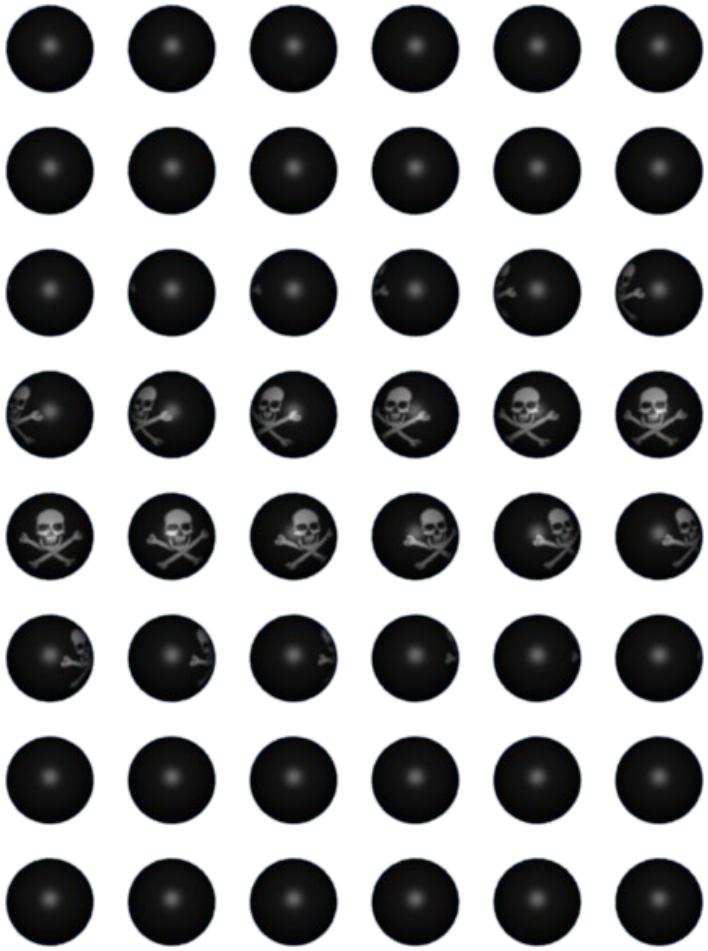
Vizualni aspekt računalnih igara je jedan od bitnijih marketinških i prodajnih aspekata računalne igre. Slika je prva stvar koju će protunacionalni kupac vidjeti prije nego uopće počne razmišljati o kupnji računalne igre. Za 2D razvoj koriste se slike različitih rezolucija ovisno o potrebi i temi koju računalna igra želi prisvojiti. Postoje dvije osnovne tehnike kreiranja animacije za 2D okruženje a to su: animiranje pomoću kičme (engl. *spine animation*) i animiranje pomoću slike (engl. *sprite animation*). Kod animacije pomoću kičme definiramo kosti tijela koje zatim vežemo za određene dijelove objekta. *Esoterics software* u svom članku „What is spine?“ govori da se prilikom micanja kostiju proporcionalno pomiču dijelovi objekta direktno povezani s tom specifičnom kosti. Na slici 8. možemo vidjeti izgled kostiju unutar objekta animiranog pomoću kičme.



Slika 8. Kosti korištene pri animaciji objekta (*Esoterico Software*: What is Spine?)

Budući da je korištenje animacije pomoću kičme dosta kompleksno te se rijetko koristi za 2D razvoj, u ovom radu neće biti opisana implementacija. Tehnika kojoj se većina developera koji razvijaju računalne igre okreće je animiranje pomoći slika koje funkcionira pomoću brze izmjene te se na taj način ostvaruje prividni osjećaj kretanja ili interakcije objekata unutar scene. Animiranje pomoću slike funkcionira na način da se u memoriju

računala učitava samo jedna slika koja u sebi sadrži više manjih slika potrebnih za animaciju određene radnje. Na slici 9. prikazana je animacija kugle s lubanjom.



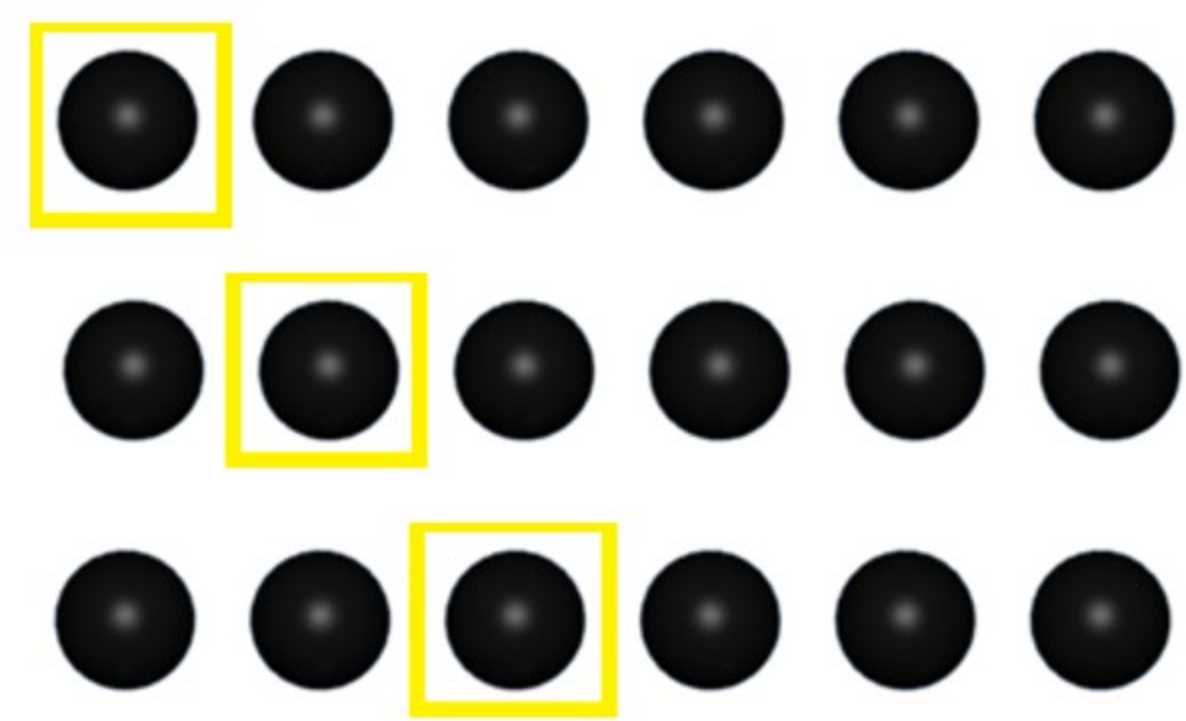
Slika 9. Animacija kugle s lubanjom (Learning XNA 4.0)

Prilikom animacije pomoću slike poželjno je da svi dijelovi animacije budu iste rezolucije na jednakom razmaku jedni od drugih radi olakšanog pristupa i kontrole. Način na koji animacije pomoću slike funkcionira je sljedeći:

- definiranje okvira koji će se koristiti za odabir određene slike unutar animacije
- dohvaćanje dijela slike na koju okvir trenutno pokazuje
- iscrtavanje slike na ekranu
- pomak okvira na sljedeću poziciju/sliku

- u slučaju da je okvir na zadnjoj animaciji, vrši se resetiranje odnosno vraćanje na početnu sliku.

Na slici 10. vizualiziran je okvir te njegovo kretanje unutar slike korištene za animaciju.



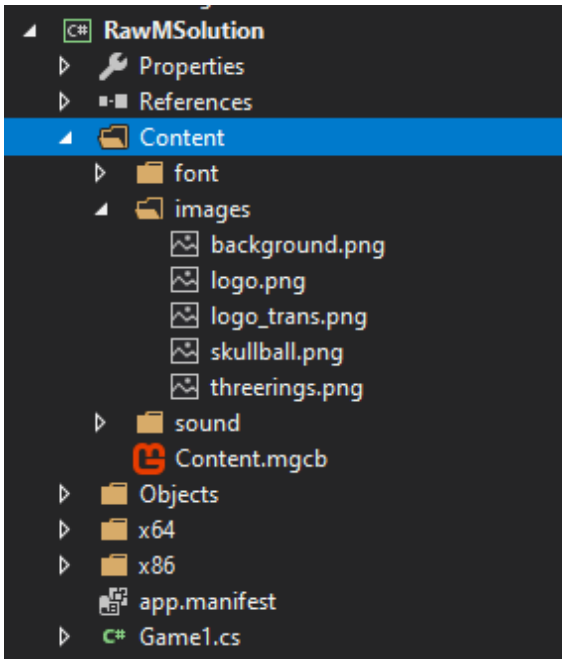
Slika 10. Vizualizacija okvira selekcije

8.1 Implementacija tekstura i animacija

Da bismo mogli vidjeti animaciju unutar scene računalne igre, potrebno je prvo kreirati teksturu na kojoj će se pojedine slike animacije iscrtavati. Tekstura je u osnovi podloga/platno koje služi za vizualizaciju objekata unutar scene. Da bismo mogli koristiti teksturu unutar računalne igre, potrebno je prvo konvertirati u format razumljiv našem razvojnom okruženju. Naše razvojno okruženje ne može direktno pročitati teksturu u PNG formatu stoga je potrebno koristiti konvergirajući alat (engl. *Pipeline tool*) kako bismo iz PNG formata kreirali XNB format.

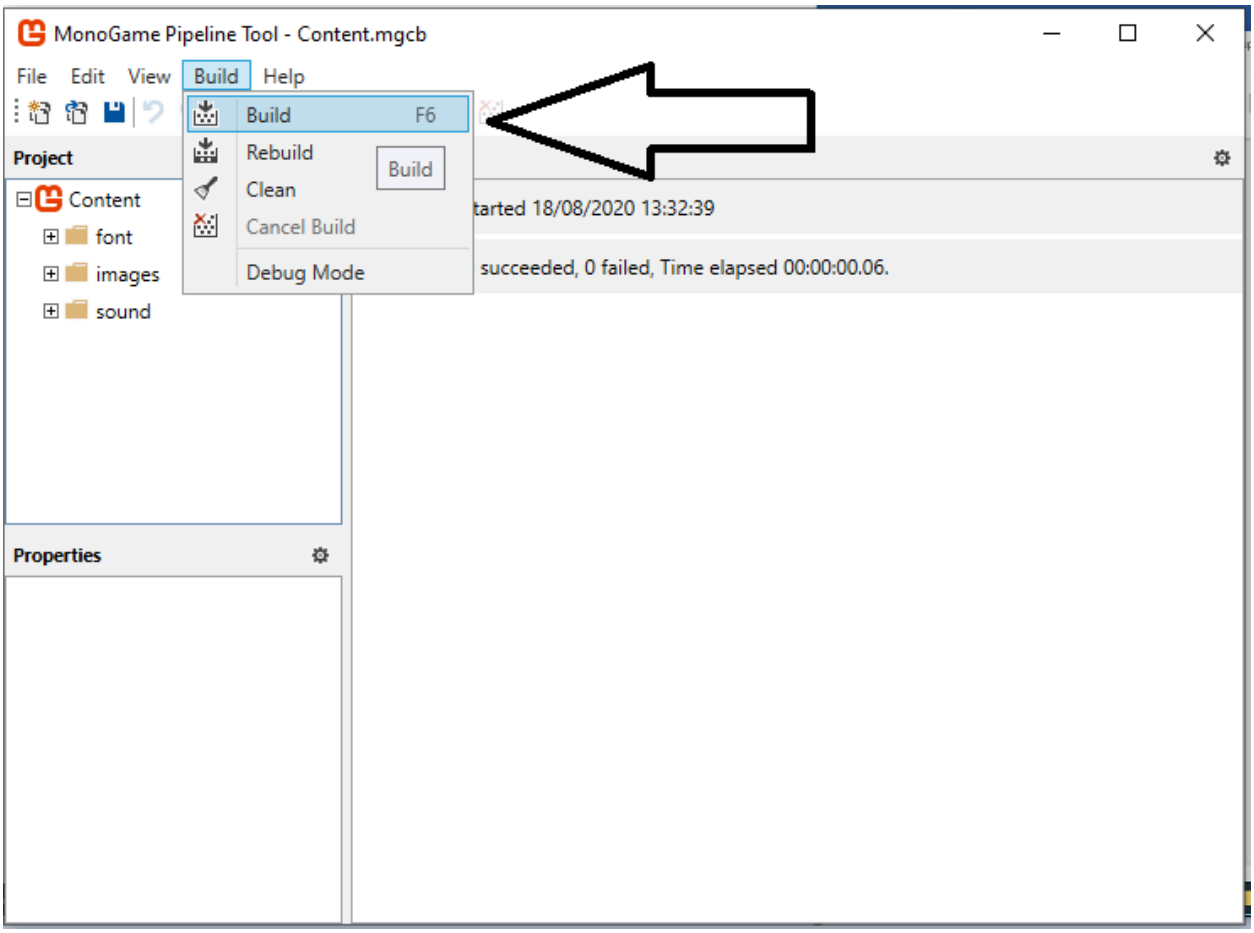
Postupak konverzije je sljedeći:

- U rješenju koje smo kreirali u prošlim poglavljima, locirati Content datoteku prikazanu na slici 11.



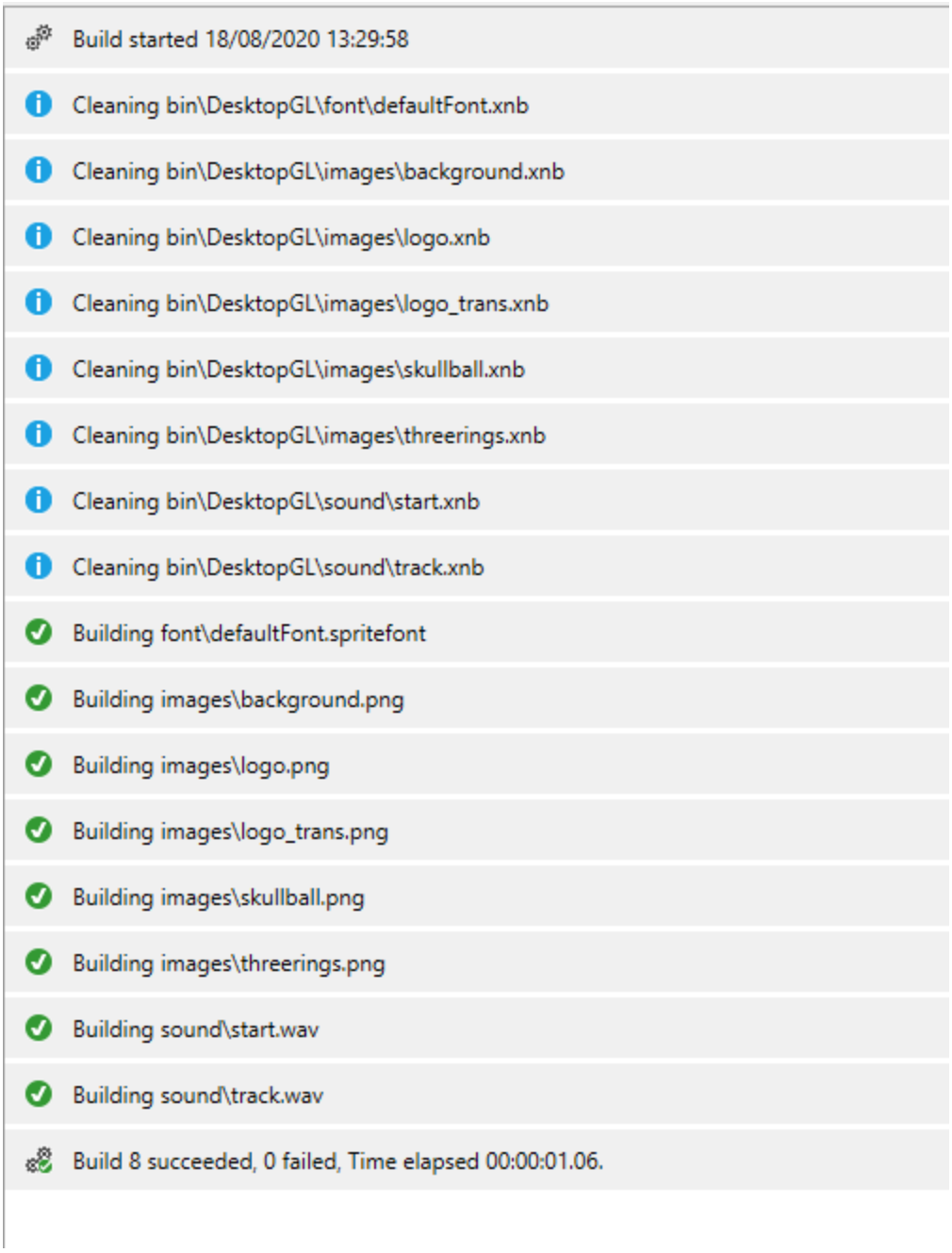
Slika 11. Content datoteka unutar projekta

- U datoteku Content dodati sav sadržaj koji želimo koristiti (slike, zvukove, fontove i sl.).
- Locirati Content.mgcb aplikaciju kroz pretraživač datoteka (engl. *file explorer*) te ga pokrenuti (aplikaciju nije moguće direktno pokrenuti iz Visual Studija).
- Prilikom pokretanja Content.mgcb aplikacije trebao bi se prikazati prozor prikazan na slici 12.



Slika 12. Početni prozor Content aplikacije

- Pritiskom na Build opciju s alatne trake te ponovnim pritiskom na Build iz padajućeg izbornika pokreće se konverzija sadržaja u xnb format. U Build Output prozoru bi trebale biti prikazane sve uspješne i neuspješne konverzije sa dodatnim porukama i razlozima kao što je prikazano na slici 13.



Slika 13. Izlazne poruke nastale nakon konverzije datoteka u XNA formate

8.2 Teksture unutar igre

Nakon što je tekstura uspješno konvergirana, možemo je učitati i koristiti unutar računalne igre. Često se zna dogoditi da se pojam slike i teksture koristi kao semantički jedan pojam, tekstura je skup jedne ili više slika koje skupa definiraju određenu površinu. Komponenta SpriteRenderer zadužena je za učitavanje tekstura, a implementacija je prikazana u primjeru koda 8.

```
1.  using Kernel.Manager;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.  using System;
5.  using System.IO;
6.
7.  namespace Kernel.Components
8.  {
9.      public class SpriteRenderer : Component
10.     {
11.         protected string TexturePath { get; set; }
12.
13.         public Texture2D Texture { get; set; }
14.
15.         public Point Size { get; set; }
16.
17.         public int OrderInLayer { get; set; }
18.
19.         public Action<SpriteBatch> DrawAction { get; set; }
20.
21.         public SpriteRenderer(string texturePath, Point size = default(Point), int orde
    rInLayer = 0, Action<SpriteBatch> drawAction = null)
22.         {
23.             this.TexturePath = texturePath;
24.
25.             if (size == default(Point))
26.             {
27.                 this.Size = new Point(1, 1);
28.             }
29.             else
30.             {
31.                 this.Size = size;
32.             }
33.             this.DrawAction = drawAction;
34.             this.OrderInLayer = orderInLayer;
35.         }
36.
37.         public override void LoadContent(GraphicsDevice graphicsDevice)
38.         {
39.             using (var stream = new FileStream(TexturePath, FileMode.Open))
40.             {
41.                 Texture = Texture2D.FromStream(graphicsDevice, stream);
42.             }
43.         }
44.
```

```
45.         public override void Update(GameTime gameTime)
46.         {
47.             if (!Enabled) return;
48.         }
49.
50.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)

51.         {
52.             if (!Enabled || !Visible) return;
53.
54.             if (DrawAction != null)
55.             {
56.                 DrawAction(spriteBatch);
57.             }
58.             spriteBatch.Draw(Texture,
59.                this.GameObject.transform.Position2D,
60.                new Rectangle(0,0,Size.X,Size.Y),
61.                Color.White,
62.                0,
63.                Vector2.Zero,
64.                1,
65.                SpriteEffects.None,
66.                OrderInLayer);
67.         }
68.
69.         public override void UpdateReferences()
70.         {
71.         }
72.
73.         public bool IsOutOfBounds(Rectangle clientRect)
74.         {
75.             if(this.GameObject == null)
76.             {
77.                 return true;
78.             }
79.
80.             if (this.GameObject.transform.Position2D.X < -Size.X ||
81.             this.GameObject.transform.Position2D.X > clientRect.Width ||
82.             this.GameObject.transform.Position2D.Y < -Size.Y ||
83.             this.GameObject.transform.Position2D.Y > clientRect.Height)
84.             {
85.                 return true;
86.             }
87.
88.             return false;
89.         }
90.
91.         public override void Destroy()
92.         {
93.             this.Texture = null;
94.         }
95.     }
96. }
```

Primjer koda 8. Implementacija komponente SpriteRenderer

Vrijednost TexturePath služi za definiranje relativnog puta do teksture. Ova vrijednost je nužna iz razloga što nekad nije optimalno učitati teksturu u memoriju, već je potrebno čekati optimalan trenutak za učitavanje te specifične teksture. Klasičan primjer ovog problema nastaje prilikom inicijalizacije objekata koji nisu vidljivi unutar scene (privremeno ili stalno). Nije potrebno učitavati teksture koje nisu vidljive igraču jer teksture u suštini služe samo za igračev vizualni dojam računalne igre, sva logika ostaje ista te se može obavljati i bez učitavanja tekstura. Vrijednost Texture sadrži učitanu teksturu. Vrijednosti Size označava veličinu pojedine slike unutar učitane teksture. Vrijednost OrderInLayer označava prioritet iscrtavanja pojedine teksture odnosno tekstura s većim prioritetom će biti iscrtana iznad teksture s manjim prioritetom. Akciju DrawAction koristimo kao predprocesorsku naredbu crtanja u kojoj definiramo logiku koju želimo izvršiti prije nego što počnemo iscrtavati objekt u scenu. Uglavnom je koristimo onda kada želimo ažurirati neke vrijednosti poput veličine slike, rezolucije i slično. Funkciju LoadContent koristimo za učitavanje teksture koristeći vrijednosti TexturePath. Funkcija Draw služi za crtanje teksture unutar scene računalne igre. Parametri koje funkcija Draw prima su:

- tekstura
- pozicija iscrtavanja
- kvadrat koji simbolizira startnu poziciju i dimenzije slike koju želimo iscrtati
- dodatna boja sjenčanja (bijela boja označava bez dodatnog efekta)
- rotacija teksture
- startna pozicija te dimenzije slike s teksture koju iscrtavamo
- skaliranje (1 je normalno skaliranje, 2 je tekstura duplo većih dimenzija itd.)
- dodatni efekt nad slikom
- indeks sloja (vrijednost OrderInLayer).

Funkcija IsOutOfBounds služi za provjeru nalazi li se objekt koji implementira komponentu SpriteRenderer unutar granica predanih kroz ulazni parametar clientRect.

Funkcija Destroy čisti referencu na teksturu te dozvoljava GC-u čišćenje teksture iz memorije u optimalnom trenutku.

8.3 Animacija

Klasa SpriteAnimation funkcijski i implementacijski sliči klasi SpriteRenderer. Glavna razlika između njih je ta što SpriteAnimation svakim novim prolazom mijenja sliku koju iscrtava. U primjeru koda 9 prikazana je implementacija komponente SpriteAnimation.

```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Graphics;
3.  using System;
4.
5.  namespace Kernel.Components
6.  {
7.      public class SpriteAnimation : SpriteRenderer
8.      {
9.          private int millisecondsPerFrame { get; set; }
10.
11.         private int timeSinceLastFrame { get; set; }
12.
13.         public Point SheetSize { get; set; }
14.
15.         public Point CurrentFrame;
16.
17.         public SpriteAnimation(
18.             int millisecondsPerFrame,
19.             string texturePath,
20.             Point currentFrame = default(Point),
21.             Point size = default(Point),
22.             int orderInLayer = 0,
23.             bool autoAnimation = false,
24.             Action<SpriteBatch> drawAction = null)
25.             : base(texturePath, size, orderInLayer, drawAction)
26.         {
27.             this.millisecondsPerFrame = millisecondsPerFrame;
28.             this.timeSinceLastFrame = 0;
29.             this.CurrentFrame = currentFrame;
30.         }
31.
32.         public override void LoadContent(GraphicsDevice graphicsDevice)
33.         {
34.             base.LoadContent(graphicsDevice);
35.             this.SheetSize = new Point(Texture.Width / NumberOfSprites.X, Texture.Heigh
    t / NumberOfSprites.Y);
36.         }
37.
38.         public override void Update(GameTime gameTime)
39.         {
```

```
40.             if (!Enabled) return;
41.
42.             this.timeSinceLastFrame += gameTime.ElapsedGameTime.Milliseconds;
43.
44.                 if (this.timeSinceLastFrame > this.millisecondsPerFrame)
45.                 {
46.                     this.timeSinceLastFrame -= this.millisecondsPerFrame;
47.                     this.CurrentFrame.X += 1;
48.
49.                     if (this.CurrentFrame.X >= this.SheetSize.X)
50.                     {
51.                         this.CurrentFrame.X = 0;
52.                         ++this.CurrentFrame.Y;
53.
54.                         if (this.CurrentFrame.Y >= this.SheetSize.Y)
55.                         {
56.                             this.CurrentFrame.Y = 0;
57.                         }
58.                     }
59.                 }
60.         }
61.
62.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
63.         {
64.             if (!Enabled || !Visible) return;
65.
66.             spriteBatch.Draw(this.Texture,
67.                this.GameObject.transform.Position2D,
68.                new Rectangle(this.CurrentFrame.X * this.NumberOfSprites.X,
69.                this.CurrentFrame.Y * this.NumberOfSprites.Y,
70.                this.NumberOfSprites.X,
71.                this.NumberOfSprites.Y),
72.                Color.White,
73.                0,
74.                Vector2.Zero,
75.                1,
76.                SpriteEffects.None,
77.                this.OrderInLayer);
78.         }
79.
80.         public override void UpdateReferences()
81.         {
82.         }
83.
84.         public override void Destroy()
85.         {
86.             base.Destroy();
87.         }
88.     }
89. }
```

Primjer koda 9. - Implementacija komponente SpriteAnimation

Vrijednost milisecondsPerFrame služi za određivanje razmaka između pojedine animacije. U slučaju da postavimo vrijednost na 50 slika animacije će se promijeniti svakih 50 milisekundi. Vrijednost timeSinceLastFrame služi za računanje vremena koje je prošlo

od zadnjeg ažuriranja slike. Ako vrijednost timeSinceLastFrame postane veća od vrijednosti milisecondsPerFrame, vrši se ažuriranje slike te se vrijednost timeSinceLastFrame postavlja na 0. Koristeći se vrijednošću SheetSize definiramo broj, širinu slike koja će se koristiti prilikom animacije. Vrijednost CurrentFrame služi kao brojač te označava indeks trenutne slike koju označavamo. Pošto SpriteAnimation nasljeđuje SpriteRenderer kroz konstruktor je potrebno poslati parametre baznoj klasi kako bi inicijalizacija bila uspješna. U funkciji LoadContent pozivamo prvo baznu funkciju kako bismo učitali teksturu, a zatim dolazimo do veličine slike koja će se prikazivati tako što širinu i dužinu teksture podijelimo s brojem slika po X i Y osi. U funkciji Update svakom novom iteracijom dodajemo u vrijednost timeSinceLastFrame vrijednost gameTime.elapsedGameTime.Miliseconds koja označava vrijeme između trenutnog i zadnjeg poziva metode ažuriranja. U slučaju da je vrijednost timeSincelastFrame veća ili jednaka od vrijednosti milisecondsPerFrame inkrementiramo X vrijednost vrijednosti CurrentFrame. U slučaju da vrijednost CurrentFramea prelazi vrijednost maksimalnog broja slika po redu, vrijednost X vraćamo na 0, a iteriramo vrijednost Y, što u osnovi znači da se iduća slika nalazi na prvom mjestu u drugom redu. U slučaju da i Y vrijednost prelazi vrijednost maksimalnog broja slika po stupcu, vrijednost Y vraćamo na 0 što znači da će iduća slika koju ćemo učitati biti na poziciji (X:0 Y:0), što je u teoriji najgornja lijeva slika koja se nalazi na teksturi. Funkcija Draw se ponaša slično i bazna funkcija samo se u ovom slučaju koristi vrijednošću CurrentFrame kako bi se ostvario pravilan pomak selektirajućih okvira.

9. Zvuk

Pored vizualnog dojma neki žanrovi igara mogu imati puno veću i bitniju ulogu. Računalne igre kao Guitar Hero ili Osu privlače igrača isključivo radi zvuka, a ne radi kompleksnih mehanika ili vizualnog dojma. U računalnim igrama natjecateljske naravi (ESports), a pogotovo u žanru pucačina iz prvog lica (FPS), zvuk se ističe kao najbitniji aspekt stjecanja prednosti nad neprijateljem. Računalne igre kao što je Doom koriste zvuk kao adrenalinski poticaj igraču da se što bolje uklopi u atmosferu i karakter koji igra. U

borbenim igrama (engl. *fighting games*) zvuk se koristi kao indikator udarca koji daje priliku protivniku da pravodobno reagira i osmisli protunapad. U prototipu igre koja je urađena u ovome diplomskom radu zvuk se koristi za pozadinski ugođaj te kao indikator kolizije u slučaju da igrač dotakne loptu.

9.1. Implementacija zvuka

Implementacija zvuka je dosta jednostavna te se zasniva na dohvaćanju resursa iz memorije te kreiranju instance zvuka koja se zatim po potrebi okida te se zvuk pušta preko izlaznog medija (slušalice, zvučnici ili sl.). Primjer koda 10 prikazuje implementaciju komponente zvuka unutar razvojnog okruženja.

```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Audio;
3.  using Microsoft.Xna.Framework.Graphics;
4.  using System.IO;
5.
6.  namespace Kernel.Components
7.  {
8.      public class AudioSource : Component
9.      {
10.         public SoundEffect SoundSource { get; set; }
11.
12.         public SoundEffectInstance SoundInstance { get; set; }
13.
14.         public AudioSource(string sourcePath)
15.         {
16.             using (var stream = new FileStream(sourcePath, FileMode.Open))
17.             {
18.                 this.SoundSource = SoundEffect.FromStream(stream);
19.             }
20.
21.             this.SoundInstance = this.SoundSource.CreateInstance();
22.         }
23.
24.         public void Play()
25.         {
26.             if(SoundInstance != null)
27.             {
28.                 SoundInstance.Play();
29.             }
30.         }
31.
32.         // Not needed on sound
33.         public override void LoadContent(GraphicsDevice graphicsDevice)
34.         {
35.         }
36.
37.         public override void Update(GameTime gameTime)
```

```
38.         {
39.             if (!Enabled) return;
40.         }
41.
42.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
43.         {
44.             if (!Enabled || !Visible) return;
45.         }
46.
47.         public override void UpdateReferences()
48.         {
49.         }
50.
51.         public override void Destroy()
52.         {
53.             this.SoundInstance = null;
54.             this.SoundSource = null;
55.         }
56.     }
57. }
```

Primjer koda 10 - Implementacija komponente AudioSource

Vrijednost SoundSource služi za pohranu učitanog zvuka iz memorije računala u radnu memoriju. Vrijednost SoundInstance sadrži novokreiranu instancu zvuka u sebi. Unutar konstruktora pomoću relativnog puta učitava se zvuk iz memorije u vrijednost SoundSource te se zatim pomoću nje kreira instanca zvuka koja se pohranjuje u vrijednost SoundInstance. Funkcija Play ima zadaću pokrenuti zvuk trenutno spremljen u vrijednosti SoundInstance. Funkcija Destroy postavlja vrijednosti SoundInstance i SoundSource na *null* što omogućuje GC-u čišćenje istih iz radne memorije računala.

10. Kolizija

Kolizija u računalnim igrama odnosi se na događaje koji nastaju prilikom dodira dvaju objekata unutar scene. Kolizija je jedna od kritičnih komponenti u bilo kojoj današnjoj igri. Budući da se kolizija mora konstantno provjeravati na svakom objektu, onda mora biti brza i optimizirana. Iz toga razloga developeri se često koriste pravilnim geometrijskim oblicima poput kocke, kružnice i trokuta kako bi detektirali koliziju. U slučaju da je broj kolizijskih operacija prevelik, može doći do pada performansi igre što rezultira kasnim ažuriranjem kolizijskih granica. Loša implementacija kolizijskog sustava može se primijetiti u računalnim igrama u kojima dolazi do prolaženja objekata jednih kroz druge ili neokidanja potrebnih događaja prilikom kolizije dvaju objekata. U ovom projektu

urađena je bazna klasa za koliziju te je implementirana osnovna kvadratna kolizija. Na slici 14. prikazan je vizualizirani primjer okvira (granica) kvadratne kolizije.

Slika 14. Vizualizacija kolizijskog okvira

Bazna apstraktna klasa Collider definira kalup koji korisnik razvojnog okruženja može naslijediti u bilo kojem trenutku te izvesti svoj proizvoljni kolizijski oblik. Primjer koda 11 prikazuje implementaciju komponente Collider.

```
1.  using Kernel.Enums;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.  using System;
5.  using System.Collections.Generic;
6.
7.  namespace Kernel.Components
8.  {
9.      public abstract class Collider : Component
10.     {
11.         public Layers Layer { get; set; }
12.
13.         protected static List<Collider> colliderPositions { get; set; }
14.
15.         public Rectangle rect { get; set; }
16.
17.         public Action<Collider> OnCollisionEnter { get; set; }
18.
19.         static Collider()
20.         {
21.             colliderPositions = new List<Collider>();
22.         }
23.
24.         public Collider(Layers layer = Layers.Layer1)
25.         {
26.             Layer = layer;
27.             colliderPositions.Add(this);
28.         }
29.
```

```
30.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
31.         {
32.             if (!Enabled || !Visible) return;
33.         }
34.
35.         public override void LoadContent(GraphicsDevice graphicsDevice)
36.         {
37.         }
38.
39.         public override void Update(GameTime gameTime)
40.         {
41.             if (!Enabled) return;
42.         }
43.
44.         public override void UpdateReferences()
45.         {
46.         }
47.
48.         public void CheckCollision()
49.         {
50.
51.             for(int i = colliderPositions.Count-1; i>= 0; i--)
52.             {
53.                 var col = colliderPositions[i];
54.
55.                 if (col == this && col.Layer == this.Layer)
56.                 {
57.                     continue;
58.                 }
59.
60.                 if (this.rect.Intersects(col.rect))
61.                 {
62.                     if (OnCollisionEnter != null)
63.                     {
64.                         OnCollisionEnter(col);
65.                     }
66.                     if (col.OnCollisionEnter != null)
67.                     {
68.                         col.OnCollisionEnter(this);
69.                     }
70.                 }
71.             }
72.         }
73.
74.         public override void Destroy()
75.         {
76.             colliderPositions.Remove(this);
77.         }
78.     }
79. }
```

Primjer koda 11 - Implementacija komponente Collider

Vrijednost Layer služi za slojevitu podjelu kolizijskih objekata, odnosno objekti mogu samo vršiti koliziju s objektima na istom kolizijskom sloju. Statička vrijednost colliderPositions je kolekcija koja sadrži reference na sve trenutno aktivne kolizijske

objekte unutar scene. Vrijednost *rect* služi za definiciju kolizijskog okvira te se kao takav koristi za detekciju kolizije s drugim objektima. Akcija OnCollisionEnter definira svu logiku koja će se izvršavati prilikom kolizije objekta, detaljniji primjer rada ove akcije će biti opisan u sljedećim poglavljima. Klasa posjeduje statički konstruktor koji se okida kao predprocesorska naredba prilikom dohvaćanja bilo koje statičke vrijednosti ili funkcije u klasi. U primjeru koda jedina statička vrijednost je colliderPosition te će ona biti inicijalizirana u praznu listu prilikom prvog poziva. Funkcija CheckCollision služi za provjeru kolizije gdje iteriramo kroz cijelu kolekciju colliderPosition te provjeravamo dolazi li do preklapanja ijednog okvira s lokalnim rect okvirom. U slučaju da dođe do kolizije poziva se akcija OnCollisionEnter na oba kolizijska objekta. U konstruktoru klase novokreirani objekt se dodaje u kolekciju colliderPosition dok se u funkciji Destroy isti objekt briše iz kolekcije.

Klasa BoxCollider2D služi za implementiranje bazične kvadratne kolizije. Primjer koda 12 prikazuje implementaciju klase BoxCollider2D.

```
1.  using Kernel.Enums;
2.  using Microsoft.Xna.Framework;
3.
4.  namespace Kernel.Components
5.  {
6.      public class BoxCollider2D : Collider
7.      {
8.          public int OffsetX { get; set; }
9.
10.         public int OffsetY { get; set; }
11.
12.         private SpriteRenderer spriteRenderer { get; set; }
13.
14.         public BoxCollider2D(Layers layer = 0, int offsetX = 0, int offsetY= 0)
15.             : base(layer)
16.         {
17.             this.OffsetX = offsetX;
18.             this.OffsetY = offsetY;
19.         }
20.
21.         public override void Update(GameTime gameTime)
22.         {
23.             if (!Enabled) return;
24.
25.             var newRect = new Rectangle((int)this.GameObject.transform.Position2D.X + O
    ffsetX,
26.                                         (int)this.GameObject.transform.Position2D.Y + OffsetY,
27.                                         spriteRenderer.Size.X - (OffsetX * 2),
28.                                         spriteRenderer.Size.Y - (OffsetY * 2));
29.
```

```
30.             if(newRect != rect)
31.             {
32.                 rect = newRect;
33.                 CheckCollision();
34.             }
35.         }
36.
37.         public override void UpdateReferences()
38.         {
39.             spriteRenderer = this.GameObject.GetComponent<SpriteRenderer>();
40.         }
41.
42.         public override void Destroy()
43.         {
44.             base.Destroy();
45.             this.spriteRenderer = null;
46.         }
47.     }
48. }
```

Primjer koda 12 - Implementacija klase BoxCollider2D

Vrijednosti OffsetX i OffsetY služe za proizvoljni pomak (engl. *offset*) kvadrata kolizije od korijena objekta. Vrijednost spriteRenderer služi za definiranje duljine, širine i pozicije kvadrata kolizije. U slučaju da su vrijednosti OffsetX i OffsetY jednake nuli, kvadrat kolizije će odgovarati dimenzijama slike koja je trenutno iscrtana na objektu. Funkcija Update služi za kreiranje novog kolizijskog okvira te provjeru kolizije. U linije 25 kreira se novi kolizijski okvir koristeći se trenutnom pozicijom objekta, dimenzijama slike zapisanim unutar spriteRenderer komponente, te vrijednostima OffsetX i OffsetY. Zatim se novokreirani okvir uspoređuje s trenutnim okvirom. U slučaju da okviri nisu isti, pretpostavljamo da je došlo do pomaka okvira te je potrebno ažurirati kolizijski okvir te zatim provjeriti postoje li nove kolizije pozivom funkcije CheckCollision. U funkciji UpdateReferences ažuriramo referencu spriteRenderer u slučaju da je prijašnja referenca maknuta ili zamijenjena novom referencom.

11. Tekst

Za prototip igre bilo je potrebno uraditi tekstualnu komponentu kako bi igrač mogao pratiti postignuti rezultat i potrebne bodove. Da bi se tekst mogao koristili unutar razvojnog okruženja, potrebno je definirati font koji će se koristiti. Font mora biti u XNB formatu što znači da mora biti unesen u projekt pomoću konverzijskog alata. Primjer koda 13 prikazuje implementaciju komponente Text.

```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Graphics;
3.
4.  namespace Kernel.Components.UI
5.  {
6.      public class Text : Component
7.      {
8.          private SpriteFont font { get; set; }
9.
10.         public string text { get; set; }
11.
12.         public Text(SpriteFont font)
13.         {
14.             this.font = font;
15.             this.text = string.Empty;
16.         }
17.
18.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
19.         {
20.             spriteBatch.DrawString(font, text,
21.             this.GameObject.transform.Position2D, Color.DarkBlue, 0, Vector2.Zero,
22.             1, SpriteEffects.None, 1);
23.
24.         }
25.
26.         public override void LoadContent(GraphicsDevice graphicsDevice)
27.         {
28.         }
29.
30.         public override void Update(GameTime gameTime)
31.         {
32.         }
33.
34.         public override void UpdateReferences()
35.         {
36.         }
37.
38.         public override void Destroy()
39.         {
40.             this.GameObject.Components.Remove(this);
41.         }
42.     }
43. }
```

Primjer koda 13 - Implementacija komponente text

Vrijednost SpriteFont sadrži vrstu fonta koja će se koristiti za ispis teksta. Vrijednost *text* sadrži vrijednost koja će biti prikazana na igračevom ekranu. Obe vrijednosti mogu biti promijenjene tijekom izvršavanja komponente. Funkcija Draw poziva funkciju DrawString koja kao parametre prima: font, *text*, poziciju, boju slova, rotaciju, pomak od početne pozicije, faktor skaliranja, posebne efekte iscrtavanja i broj sloja u kojem se tekst nalazi.

12. Definiranje generalnog dizajna igre

Nakon što razvojno okruženje posjeduje sve potrebne komponente za kreaciju prototipa, potrebno je odrediti generalni dizajn igre koji će omogućiti krajnjem developeru da što lakše i jednostavnije kreira nove funkcionalnosti. Koristeći dosad napisan kod, krajnji developer nema baš striktno definiran način pisanja koda što može biti i prednost i nedostatak razvojnog okruženja ovisno o znanju koje developer posjeduje. Stoga razvojno okruženje definira osnovni kalup za pisanje logike unutar igre bez da zabranjuje krajnjem developeru da proizvede/izmisli svoj vlastiti dizajn. Klasa MonoBehaviour posjeduje osnovnu definiciju dizajna unutar razvojnog okruženja te je klasa GameLoop zadužena za ažuriranja svih aktivnih objekata na sceni.

U primjeru koda 14 prikazana je implementacija komponente MonoBehaviour.

```
1.  using Kernel.Manager;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.
5.  namespace Kernel.Design
6.  {
7.      public abstract class MonoBehaviour
8.      {
9.          public GameObject gameObject;
10.
11.         protected GraphicsDevice graphicsDevice;
12.
13.         protected MonoBehaviour(GraphicsDevice graphicsDevice)
14.         {
15.             this.gameObject = new GameObject(this.GetType().Name);
16.             this.graphicsDevice = graphicsDevice;
17.             GameManager.singleton.gameObjects.Add(this);
18.             this.LoadContent();
19.         }
20.
21.         protected abstract void InitComponents();
22.
23.         private void LoadContent()
24.         {
25.             this.InitComponents();
26.             GameLoop.WaitingForAdding.Enqueue(this);
27.             this.gameObject.LoadContent(graphicsDevice);
28.         }
29.
30.         public abstract void Update(GameTime gameTime);
31.
32.         public virtual void Destroy()
33.         {
34.             GameLoop.Destroy(this);
```

```
35.             GameManager.singleton.gameObjects.Remove(this);
36.             this.gameObject.Destroy();
37.         }
38.     }
39. }
```

Primjer koda 14 - Implementacija komponente MonoBehaviour

Vrijednost GameObject definira osnovni objekt kroz koji će implementirati sve funkcionalnost objekta unutar scene. Vrijednost graphicDevice služi za lokalnu pohranu instance objekta grafičkog uređaja koji je potreban komponentama koje posjeduju funkcionalnost u Draw funkcijama kao što su SpriteRenderer, SpriteAnimation, Text i sl. Pozivom konstruktora kreira se nova instanca gameObjecta te mu se kao ime postavlja tip objekta koji implementira MonoBehaviour. Objekt se zatim u liniji 17 dodaje na listu svih aktivnih objekata unutar scene te se poziva funkcija LoadContent. Funkcijom LoadContent se prvo inicijaliziraju komponente MonoBehaviour objekta te se on zatim dodaje u red za čekanje unutar klase GameLoop. Razlog što se objekt dodaje u red za čekanje, a ne direktno u listu za iteraciju je taj što ne želimo u istoj iteraciji u kojoj smo stvorili objekt iterirati po njegovim funkcijama iz razloga što se može dogoditi da se određene komponente još nisu učitale iz memorije te kao takve nisu spremne za korištenje. Nakon što smo dodali objekt u red za čekanje nad vrijednosti gameObject, pozivamo funkciju LoadContent koja bi do iduće iteracije trebala učitati sve komponente te ih pripremiti za korištenje. Funkcije InitComponents i Update su apstraktne funkcije te ne sadrže tijelo u klasama u kojima se definiraju, već obvezuju klasu koja ih nasljeđuje da ih implementira (više u sljedećim poglavljima). Funkcija Destroy služi za uništavanje objekta iz scene te njegovo čišćenje iz memorije.

U primjeru koda 15 prikazana je implementacija komponente GameLoop.

```
1.  using Kernel.Components;
2.  using Kernel.Design;
3.  using Microsoft.Xna.Framework;
4.  using Microsoft.Xna.Framework.Graphics;
5.  using System;
6.  using System.Collections.Generic;
7.
8.  namespace Kernel.Manager
9.  {
10.     public static class GameLoop
11.     {
12.         public static List<MonoBehaviour> MonoBehaviours { get; set; }
13.
```

```
14.         public static Queue<MonoBehaviour> WaitingForAdding { get; set; }
15.
16.         public static Queue<MonoBehaviour> WaitingForDestruction { get; set; }
17.
18.         public static int DefaultMillisecondsPerFrame { get; set; }
19.
20.         static GameLoop()
21.         {
22.             DefaultMillisecondsPerFrame = 50;
23.             MonoBehaviours = new List<MonoBehaviour>();
24.             WaitingForDestruction = new Queue<MonoBehaviour>();
25.             WaitingForAdding = new Queue<MonoBehaviour>();
26.         }
27.
28.         public static void Destroy(MonoBehaviour monoBehaviour)
29.         {
30.             WaitingForDestruction.Enqueue(monoBehaviour);
31.         }
32.
33.         public static void UpdateObjects(GameTime gameTime)
34.         {
35.             foreach (var mono in MonoBehaviours)
36.             {
37.                 mono.Update(gameTime);
38.                 mono.gameObject.Update(gameTime);
39.             }
40.
41.             while (WaitingForAdding.Count > 0)
42.             {
43.                 MonoBehaviours.Add(WaitingForAdding.Dequeue());
44.             }
45.
46.             while (WaitingForDestruction.Count > 0)
47.             {
48.                 MonoBehaviours.Remove(WaitingForDestruction.Dequeue());
49.             }
50.         }
51.
52.         public static void DestroyAll()
53.         {
54.             foreach (var mono in MonoBehaviours)
55.             {
56.                 WaitingForDestruction.Enqueue(mono);
57.             }
58.         }
59.
60.         public static void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
61.         {
62.             foreach (var mono in MonoBehaviours)
63.             {
64.                 mono.gameObject.Draw(spriteBatch, camera);
65.             }
66.         }
67.     }
68. }
```

Primjer koda 15 - Implementacija komponente GameLoop

Vrijednost MonoBehaviours sadrži kolekciju svih aktivnih MonoBehaviour objekata unutar scene. Vrijednost WaitingForAdding sadrži kolekciju svih MonoBehaviour objekata koji čekaju da budu dodani unutar scene. Vrijednost WaitingForDestruction sadrži kolekciju svih MonoBehaviour objekata koji čekaju brisanje sa scene. Vrijednost DefaultMilisecondsPerFrame služi za određivanje otkucaja ažuriranja, odnosno s njom određujemo svako koliko će se vršiti ažuriranje scene. U konstruktoru je ova vrijednost postavljena na 50 milisekundi što znači da će se igra ažurirati 20 puta po sekundi (20fps). Funkcija Destroy dodaje objekt koji prima kao parametar u kolekciju WaitingForDestruction. Funkcija DestroyAll dodaje sve objekte na sceni u kolekciju WaitingForDestruction. Funkcija UpdateObjects vrši ažuriranje svih objekata unutar scene te nakon toga prvo dodaje sve objekte iz kolekcije WaitingForAdding, a zatim briše sve objekte iz kolekcije WaitingForDestruction. Funkcija Draw služi za iscrtavanje svih objekata unutar scene.

U primjeru koda 16 prikazana je implementacija komponente GameManager.

```
1.  using Kernel.Design;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.  using System;
5.  using System.Collections.Generic;
6.  using System.Threading.Tasks;
7.
8.  namespace Kernel.Manager
9.  {
10.     public class GameManager
11.     {
12.         private static GameManager _singleton;
13.
14.         public static GameManager singleton { get { return Init(); } }
15.
16.         public HashSet<MonoBehaviour> gameObjects { get; set; }
17.
18.         public GameManager()
19.         {
20.             this.gameObjects = new HashSet<MonoBehaviour>();
21.         }
22.
23.         private static GameManager Init()
24.         {
25.             if(_singleton == null)
26.             {
27.                 _singleton = new GameManager();
28.             }
29.
30.             return _singleton;
31.         }
32.
```

```
33.         public MonoBehaviour FindObjectByName(string name)
34.         {
35.             foreach(var item in gameObjects)
36.             {
37.                 if(item.gameObject.Name == name)
38.                 {
39.                     return item;
40.                 }
41.             }
42.
43.             return null;
44.         }
45.
46.         public Type FindObjectByType<Type>()
47.             where Type : MonoBehaviour
48.         {
49.             foreach (var item in gameObjects)
50.             {
51.                 if (item is Type)
52.                 {
53.                     return (Type)item;
54.                 }
55.             }
56.
57.             return null;
58.         }
59.     }
60.
61. }
```

Primjer koda 16 - Implementacija komponente GameManager

Vrijednost singleton koristi se kao statička instanca klase GameManager. Kolekcija gameObjects sadrži sve aktivne objekte unutar scene. Funkcija Init() vraća instancu klase GameManager te u slučaju da instanca još nije kreirana, funkcija će kreirati novu instancu. Funkcija FindObjectByName kao ulazni parametar prima tekst koji simbolizira ime objekta te kao rezultat vraća objekt unutar scene s tim specifičnim imenom ili *null* vrijednost. Funkcija FindObjectByType<Type>, prima generički parametar koji simbolizira tip te kao rezultat vraća objekt unutar scene koji odgovara tom specifičnom tipu ili *null* vrijednost.

13. Implementacija prototipa 2D igre

Sve komponente koje su opisane u prijašnjim poglavljima bit će prikazane i implementirane unutar prototipa igre „Uhvati loptu“. U prvom primjeru koda nalazi se

implementacija objekta Player koja zapravo ima zadaću primati igračeve ulazne naredbe i reagirati odgovarajućim naredbama. U primjeru koda 17 prikazana je implementacija objekta Player.

```
1.  using Kernel;
2.  using Kernel.Components;
3.  using Kernel.Design;
4.  using Microsoft.Xna.Framework;
5.  using Microsoft.Xna.Framework.Graphics;
6.  using Microsoft.Xna.Framework.Input;
7.  using System;
8.
9.  namespace RawMSolution.Objects
10. {
11.     public class Player : MonoBehaviour
12.     {
13.         public float speed = 6;
14.
15.         public Player(GraphicsDevice graphicsDevice) : base(graphicsDevice)
16.         {
17.         }
18.
19.         public override void Update(GameTime gameTime)
20.         {
21.             if (InputController.GetKeyDown(Keys.A))
22.             {
23.                 gameObject.transform.Position2D.X -= speed;
24.             }
25.             if (InputController.GetKeyDown(Keys.D))
26.             {
27.                 gameObject.transform.Position2D.X += speed;
28.             }
29.             if (InputController.GetKeyDown(Keys.W))
30.             {
31.                 gameObject.transform.Position2D.Y -= speed;
32.             }
33.             if (InputController.GetKeyDown(Keys.S))
34.             {
35.                 gameObject.transform.Position2D.Y += speed;
36.             }
37.
38.             if (InputController.IsMouseActive())
39.             {
40.                 gameObject.transform.Position2D = InputController.GetMousePosition();
41.             }
42.
43.             var position = InputController.GetGamePadLeftStickPosition();
44.             if (InputController.GetGamePadKeyDown(Buttons.A))
45.             {
46.                 gameObject.transform.Position2D.X += speed * 2 * position.X;
47.                 gameObject.transform.Position2D.Y -= speed * 2 * position.Y;
48.                 GamePad.SetVibration(PlayerIndex.One, 1f, 1f);
49.             }
50.             else
51.             {
52.                 gameObject.transform.Position2D.X += speed * position.X;
53.                 gameObject.transform.Position2D.Y -= speed * position.Y;
54.                 GamePad.SetVibration(PlayerIndex.One, 0, 0);
```

```
55.             }
56.         }
57.
58.         protected override void InitComponents()
59.         {
60.             gameObject.AddComponent(new BoxCollider2D(0, 10, 10));
61.             gameObject.AddComponent(new AudioSource(@"Content/sound/bgmusic.wav"));
62.             gameObject.AddComponent(new SpriteAnimation(50, @"Content/images/threerings
    .png",new Point(0,0), new Point(75, 75), 1, true));
63.             gameObject.transform.Position2D = new Vector2(100, 100);
64.
65.             var audioSource = gameObject.GetComponent<AudioSource>();
66.             audioSource.SoundInstance.IsLooped = true;
67.             audioSource.Play();
68.         }
69.     }
70. }
```

Primjer koda 17 - Implementacija objekta Player

Vrijednost speed označava brzinu pomicanja objekta unutar scene. Funkcija InitComponents dodaje sljedeće komponente na objekt: BoxCollider2D, AudioSource, SpriteAnimation te određuje početnu poziciju objekta unutar scene. BoxCollider2D pozicionirana je u središtu objekta s duljinom i širinom umanjenom za 10 piksela. Komponenta AudioSource učitava bgmusic.wav iz pozadine te je pokreće u liniji koda 67. SpriteAnimation komponenta učitava sliku threerings.png te od nje kreira animaciju u kojoj je svaka slika visoka i široka 75 piksela. U funkciji Update učitavamo ulazne vrijednosti koje igrač pritišće te pomičemo objekt u odgovarajućem smjeru, za kretnju je moguće koristiti tipkovnicu (21. – 37. linija koda), miš (38. – 42. linija koda) i kontroler (42. – 56. linija koda).

Sljedeći objekt koji je bilo potrebno uraditi vezan je za loptu koju igrač mora dotaknuti kako bi osvojio bodove. U primjeru koda 18 prikazana je implementacija objekta Ball.

```
1.  using Kernel.Components;
2.  using Kernel.Design;
3.  using Kernel.Manager;
4.  using Microsoft.Xna.Framework;
5.  using Microsoft.Xna.Framework.Graphics;
6.  using System;
7.
8.  namespace RawMSolution.Objects
9.  {
10.     public enum TypeOfEnemyMovement
11.     {
12.         Random,
13.         ChasePlayer,
```

```
14.         Evade
15.     }
16.
17.     public class Ball : MonoBehaviour
18.     {
19.         private Random rnd { get; set; }
20.
21.         private int direction { get; set; }
22.
23.         private SpriteAnimation sprAnim { get; set; }
24.
25.         public TypeOfEnemyMovement TypeOfMovement { get; set; }
26.
27.         public Player Player { get; set; }
28.
29.         public int ScoreValue { get; set; }
30.
31.         public Ball(GraphicsDevice graphicsDevice, int scoreValue, TypeOfEnemyMovement
    typeOfMovement = TypeOfEnemyMovement.Random) : base(graphicsDevice)
32.         {
33.             rnd = new Random();
34.             direction = rnd.Next(0, 4);
35.             this.TypeOfMovement = typeOfMovement;
36.             this.ScoreValue = scoreValue;
37.         }
38.
39.         public override void Update(GameTime gameTime)
40.         {
41.             if (TypeOfMovement == TypeOfEnemyMovement.Random)
42.             {
43.                 switch (direction)
44.                 {
45.                     case 0:
46.                         gameObject.transform.Position2D.X++;
47.                         break;
48.                     case 1:
49.                         gameObject.transform.Position2D.X--;
50.                         break;
51.                     case 2:
52.                         gameObject.transform.Position2D.Y++;
53.                         break;
54.                     case 3:
55.                         gameObject.transform.Position2D.Y--;
56.                         break;
57.                 }
58.             }
59.             else if(TypeOfMovement == TypeOfEnemyMovement.ChasePlayer)
60.             {
61.                 if(Player != null)
62.                 {
63.                     if (Player.gameObject.transform.Position2D.X < gameObject.transform
    .Position2D.X)
64.                         gameObject.transform.Position2D.X--;
65.                     else if (Player.gameObject.transform.Position2D.X > gameObject.tran
    sform.Position2D.X)
66.                         gameObject.transform.Position2D.X++;
67.                     if (Player.gameObject.transform.Position2D.Y < gameObject.transform
    .Position2D.Y)
68.                         gameObject.transform.Position2D.Y--;
69.                     else if (Player.gameObject.transform.Position2D.Y > gameObject.tran
    sform.Position2D.Y)
```

```
70.                         gameObject.transform.Position2D.Y++;
71.                 }
72.                 else
73.                 {
74.                     Player = GameManager.singleton.FindObjectByType<Player>();
75.                 }
76.             }
77.             else if (TypeOfMovement == TypeOfEnemyMovement.Evade)
78.             {
79.                 if (Player != null)
80.                 {
81.                     if (Player.gameObject.transform.Position2D.X < gameObject.transform
    .Position2D.X)
82.                         gameObject.transform.Position2D.X++;
83.                     else if (Player.gameObject.transform.Position2D.X > gameObject.tran
    sform.Position2D.X)
84.                         gameObject.transform.Position2D.X--;
85.                     if (Player.gameObject.transform.Position2D.Y < gameObject.transform
    .Position2D.Y)
86.                         gameObject.transform.Position2D.Y++;
87.                     else if (Player.gameObject.transform.Position2D.Y > gameObject.tran
    sform.Position2D.Y)
88.                         gameObject.transform.Position2D.Y--;
89.                 }
90.                 else
91.                 {
92.                     Player = GameManager.singleton.FindObjectByType<Player>();
93.                 }
94.             }
95.
96.             if (sprAnim.IsOutOfBounds(Game1.singleton.Window.ClientBounds))
97.             {
98.                 Destroy();
99.             }
100.            }
101.
102.            protected override void InitComponents()
103.            {
104.                var audioSource = new AudioSource(@"Content/sound/collected.wav");
105.                audioSource.SoundInstance.IsLooped = false;
106.                gameObject.AddComponent(audioSource);
107.                gameObject.AddComponent(new BoxCollider2D(0, 10, 10));
108.                gameObject.GetComponent<BoxCollider2D>().OnCollisionEnter = a =>
109.                {
110.                    if (a.GameObject.Name == nameof(Player))
111.                    {
112.                        audioSource.Play();
113.                        Destroy();
114.                        Manager.TotalScore += 100;
115.                    }
116.                };
117.                sprAnim = new SpriteAnimation(50, "Content/images/skullball.png", ne
    w Point(0, 0), new Point(75, 75), 1, true);
118.                gameObject.AddComponent(sprAnim);
119.                gameObject.transform.Position2D = new Vector2(250, 250);
120.            }
121.        }
122.    }
```

Primjer koda 18 - Implementacija objekta Ball

Vrijednost rnd služi kao slučajni generator smjera koji se pohranjuje u vrijednost direction. Vrijednost sprAnim služi kao pomoćna referenca pri dohvaćanju komponente SpriteAnimation s objekta. Vrijednošću TypeOfMovement definiramo način ponašanja lopte koje može biti: nasumično kretanje, bježanje od igrača, kretanje za igračem. Vrijednost Player sadrži referencu na igračev objekt, a vrijednost ScoreValue definira broj bodova koje će igrač osvojiti ako uspije uhvatiti tu loptu.

U funkciji InitComponent na objekt se dodaju sljedeće komponente: AudioSource, BoxCollider2D, SpriteAnimation. Unutar AudioSource komponente vrijednost IsLooped postavljamo na laž što znači da se zvuk neće ponavljati nego će se odsvirati samo jedanput. U komponenti BoxCollider2D postavljamo središe kolizijskog okvira u centru objekta sa širinom i dužinom manjom za 10 piksela od stvarne dužine objekta. Akciji OnCollisionEnter definiramo funkciju koja provjerava je li objekt kolizije tipa Player te u slučaje da jest, pokrećemo zvuk iz AudioSource komponente, uništavamo objekt iz scene te povećavamo ukupni rezultat koji je igrač ostvario za 100 bodova. Komponenta SpriteAnimation učitava sliku skullball.png te od nje kreira animaciju gdje je svaka slika velika 75 piksela po širini i dužini. Prvi parameter u SpriteAnimation konstruktoru je brzina izvođenja animacije koja je 50 milisekundi. Funkcija Update služi za implementaciju osnovne logike kretanja lopti koje je ovisno o stanju koje im se definira vrijednošću TypeOfMovement.

Budući da objekti tipa Ball moraju biti stvoreni unutar scene, definirali smo objekt BallSpawner koji će imati zadaću stvarati lopte unutar scene u nasumičnim periodima i na nasumičnim pozicijama. U primjeru koda 19 prikazana je implementacija objekta BallSpawner.

```
1.  using Kernel.Design;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.  using System;
5.
6.  namespace RawMSolution.Objects
7.  {
8.      public class BallSpawner : MonoBehaviour
9.      {
10.         public Random rand { get; set; }
11.
12.         private int ballSpawnMinMilliseconds = 300;
13.
```

```
14.         private int ballSpawnMaxMilliseconds = 1500;
15.
16.         private int nextSpawnTime = 0;
17.
18.         int numberOfSpawns = 100;
19.
20.         public BallSpawner(GraphicsDevice graphicsDevice) : base(graphicsDevice)
21.         {
22.             this.rand = new Random();
23.         }
24.
25.         public override void Update(GameTime gameTime)
26.         {
27.             if (numberOfSpawns > 0)
28.             {
29.                 nextSpawnTime -= gameTime.ElapsedGameTime.Milliseconds;
30.                 if (nextSpawnTime < 0)
31.                 {
32.                     var test = new Ball(graphicsDevice,100, TypeOfEnemyMovement.Evade);

33.                     nextSpawnTime = rand.Next(ballSpawnMinMilliseconds, ballSpawnMaxMil
    liseconds);
34.
35.                     var x = rand.Next(0, 1000);
36.                     var y = rand.Next(0, 1000);
37.                     test.gameObject.transform.Position2D = new Vector2(x, y);
38.                     numberOfSpawns--;
39.                 }
40.             }
41.         }
42.
43.         protected override void InitComponents()
44.         {
45.         }
46.     }
47. }
```

Primjer koda 19 - Implementacija objekta BallSpawner

Vrijednost rand služi za nasumično generiranje pozicije i vremena kreiranja nove lopte unutar scene. Vrijednosti nextSpawnTime je nasumična vrijednost koja se nalazi u granicama vrijednosti ballSpawnMinMiliseconds i ballSpawnMaxMiliseconds te definira iduću točku stvaranja nove vrijednosti. Vrijednost numberOfSpawns služi za provjeru nalazi li se na sceni maksimalan broj lopti odnosno je li moguće dodati više lopti na scenu. U funkciji Update provjeravamo uvjete za stvaranje nove lopte unutar scene. U slučaju da su uvjeti zadovoljeni, stvara se nova lopta na nasumičnoj poziciji te se brojač za kreiranje nove lopte resetira.

Objekti ScoreBox,TimerDisplay, MainMenu i GameOverScreen definirani su na isti način te se jedino razlikuju u tekstu koji prikazuju i u poziciji. U primjeru koda 20 prikazana je implementacija objekta ScoreBox.

```
1.  using Kernel.Components.UI;
2.  using Kernel.Design;
3.  using Microsoft.Xna.Framework;
4.  using Microsoft.Xna.Framework.Graphics;
5.
6.  namespace RawMSolution.Objects
7.  {
8.      public class ScoreBox : MonoBehaviour
9.      {
10.         Text text;
11.
12.         public ScoreBox(GraphicsDevice graphicsDevice) : base(graphicsDevice)
13.         {
14.         }
15.
16.         public override void Update(GameTime gameTime)
17.         {
18.             text.text = "Score" + Manager.TotalScore;
19.         }
20.
21.         protected override void InitComponents()
22.         {
23.             SpriteFont font = Game1.singleton.Content.Load<SpriteFont>("font/defaultFon
    t");
24.             text = new Text(font);
25.             this.gameObject.AddComponent(text);
26.             this.gameObject.transform.Position2D = new Vector2(10, 10);
27.         }
28.     }
29. }
```

Primjer koda 20 - Implementacija objekta ScoreBox

Vrijednost text sadrži tekst koji će se ispisivati na ekranu igre. Funkcija InitComponents dodaje Text komponentu na objekt te postavlja poziciju u gornji lijevi kut ekrana. Funkcija Update služi za ažuriranje teksta tako da rezultat odgovara trenutnom broju osvojenih bodova od igrača.

Objekt Background služi za definiciju pozadinske slike unutar igre koja uvijek mora popunjavati cijeli ekran i nakon što se rezolucija promijeni. U primjeru koda 21 prikazana je implementacija objekta Background.

```
1.  using Kernel.Components;
2.  using Kernel.Design;
```

```
3.  using Microsoft.Xna.Framework;
4.  using Microsoft.Xna.Framework.Graphics;
5.  using System;
6.  using System.Collections.Generic;
7.  using System.Linq;
8.  using System.Text;
9.  using System.Threading.Tasks;
10.
11. namespace RawMSolution.Objects
12. {
13.     public class Background : MonoBehaviour
14.     {
15.         public Background(GraphicsDevice graphicsDevice) : base(graphicsDevice)
16.         {
17.         }
18.
19.         public override void Update(GameTime gameTime)
20.         {
21.         }
22.
23.         protected override void InitComponents()
24.         {
25.             var spriteRen = new SpriteRenderer(@"Content/images/background.png");
26.             spriteRen.DrawAction = new Action<SpriteBatch>(a => {
27.                 a.Draw(spriteRen.Texture,
28.                 new Rectangle(0, 0, Game1.singleton.Window.ClientBounds.Width,
29.                 Game1.singleton.Window.ClientBounds.Height), null,
30.                 Color.White, 0, Vector2.Zero,
31.                 SpriteEffects.None, 0);
32.             });
33.
34.             this.gameObject.AddComponent(spriteRen);
35.         }
36.     }
37. }
```

Primjer koda 21 - Implementacija objekta Background

Funkcija InitComponents dodaje komponentu SpriteRenderer na objekt te definira akciju DrawAction s funkcijom koja će uvijek teksturu unutar SpriteRenderer komponente razvući ili suziti tako da popunjava cijeli ekran.

Nakon što smo definirali sve objekte koje želimo koristiti, potrebno je kreirati upravitelja računalne igre (engl. *game manager*) koji će upravljati sa svim objektima ovisno o stanjima u kojima se igra nalazi. U primjeru koda 22 implementiran je objekt GameStateManager zadužen za upravljanje stanjima igre.

```
1.  using Kernel.Manager;
2.  using Microsoft.Xna.Framework;
3.  using Microsoft.Xna.Framework.Graphics;
4.  using RawMSolution.Objects.Enums;
```

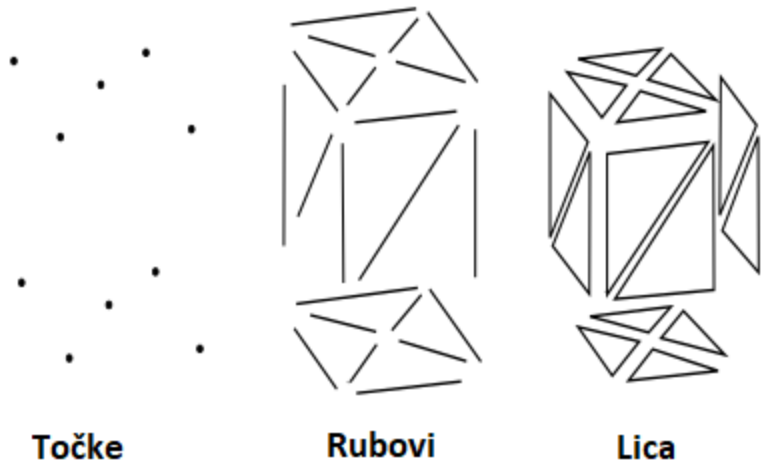
```
5.
6.  namespace RawMSolution.Objects
7.  {
8.      public class GameStateManager
9.      {
10.         public GameState state { get; set; }
11.
12.         public MainMenu mainMenu { get; set; }
13.
14.         public const int MaximumTime = 10000;
15.
16.         public int timeLimit { get; set; }
17.
18.         public GraphicsDevice graphicsDevice { get; set; }
19.
20.         public GameStateManager(GraphicsDevice graphicsDevice)
21.         {
22.             this.graphicsDevice = graphicsDevice;
23.             mainMenu = new MainMenu(graphicsDevice);
24.         }
25.
26.         public void StartGame()
27.         {
28.             mainMenu.Destroy();
29.             state = GameState.InGame;
30.             new ScoreBox(graphicsDevice);
31.             new Player(graphicsDevice);
32.             new BallSpawner(graphicsDevice);
33.             new Background(graphicsDevice);
34.             new TimerDisplay(graphicsDevice);
35.         }
36.
37.         public void GameOver()
38.         {
39.             GameLoop.DestroyAll();
40.             new GameOverScreen(graphicsDevice);
41.             state = GameState.GameOver;
42.         }
43.
44.         public void Update(GameTime gameTime)
45.         {
46.             timeLimit += gameTime.ElapsedGameTime.Milliseconds;
47.
48.             if (MaximumTime - timeLimit < 0)
49.             {
50.                 GameOver();
51.             }
52.
53.             GameLoop.UpdateObjects(gameTime);
54.         }
55.
56.         public void Draw(SpriteBatch spriteBatch)
57.         {
58.             GameLoop.Draw(spriteBatch);
59.         }
60.     }
61. }
```

Primjer koda 22 - Implementacija objekta GameStateManager

Vrijednost *state* definira stanja u kojima se igra može nalaziti. Moguća stanja igre su: Glavni izbornik, Igranje, Kraj igre. Vrijednost mainMenu sadrži referencu na glavni izbornik koji se inicijalizira na sceni kad je igra u stanju *Glavni izbornik*. Vrijednost timeLimit označava vrijeme preostalo do kraja igre. U funkciji StartGame postavljamo stanje igre na InGame te prvo uništavamo objekt mainMenu iz scene, a zatim inicijaliziramo sve objekte koji će se koristiti unutar igre. Valja primijetiti da objekt Ball nije inicijaliziran budući da se s njim upravlja preko BallSpawner objekta. U funkciji GameOver prvo uništavamo sve objekte u sceni, a zatim inicijaliziramo GameOverScreen na kojem se ispisuje razultat koji je igrač ostvario. U funkciji Update brojimo vrijeme te u slučaju da je brojač prešao maksimalno dopušteno vrijeme, pozivamo funkciju GameOver. Također u funkcijama Update i Draw pozivamo ekvivalente funkcija iz klase GameLoop definirane u prijašnjim poglavljima.

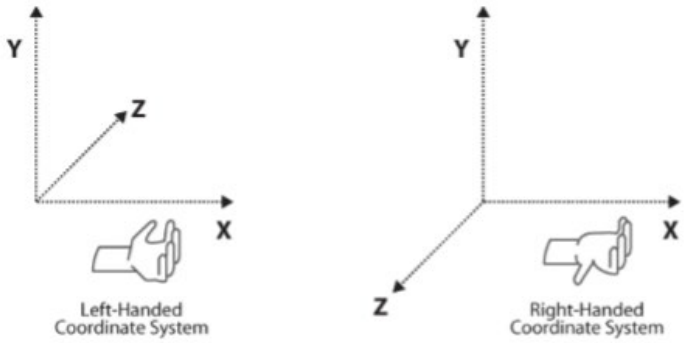
14. Razvoj 3D računalnih igara

Iako se 2D i 3D računalne igre razlikuju u samo jednoj dodatnoj osi. Implementacija funkcionalnosti unutar 3D računalnih igara je puno kompleksnija. Slike se zamjenjuju modelima koji se prema Wikipedijinom članku *Poligonske mreže* sastoje od skupa točki i rubova koji se u svakom ažuriranju ponovno iscrtavaju unutar prostora te se zatim povezuju, a prostor između njih se popunjava licima. Na slici 15. vizualno su prikazani izgledi točki rubova i lica.



Slika 15. Točke, rubovi i lica (Learning XNA 4.0)

Kao i u 2D okruženju i 3D okruženje posjeduje koordinatne osi. Budući da X i Y os označuju dužinu i visinu unutar prostora, treća os Z služi za definiranje širine objekta. Jason Gregory u svojoj knjizi *Arhitektura razvojnog okruženja računalnih igrara* govori o tome da se na današnjem tržištu razvijaju okruženja isključivo sa cilindričnim i kockastim koordinatnim sustavima. Iako kockasti koordinatni prevladava na današnjem tržištu, cilindrični sustav je puno korisniji u igrama koje imaju puno cirkularnih gibanja. Kockasti koordinatni sustavi se također dijele na dvije osnovne podvrste, a to su ljevoruki i desnoruki koordinatni sustavi. Na slici 16. prikazan je izgled ljevorukog i desnorukog koordinatnog sustava.



Slika 16. Ljevoruki i desnoruki koordinatni sustav (Learning XNA 4.0)

Velik problem koji je stopirao 3D razvoj računalnih igara bio je isključivo vezan za optimizaciju. Budući da su računala davnih 1980-ih posjedovala dosta malu količinu memorije i procesorske snage, developeri su konstantno izmišljali nove tehnike kako bi što bolje iskoristili računalne resurse. Kompanija Id Software 1996. godine revolucionira industriju računalnih igara kreirajući računalnu igru pod nazivom *Quake* koja postaje prva računalna igra u kojoj su svi karakteri i objekti unutar scene bili 3D modeli. Na slici 17. prikazan je izgled Quake računalne igre preuzete sa Steam *online* trgovine.

Slika 17. Quake računalna igra (Quake Steam trgovina)

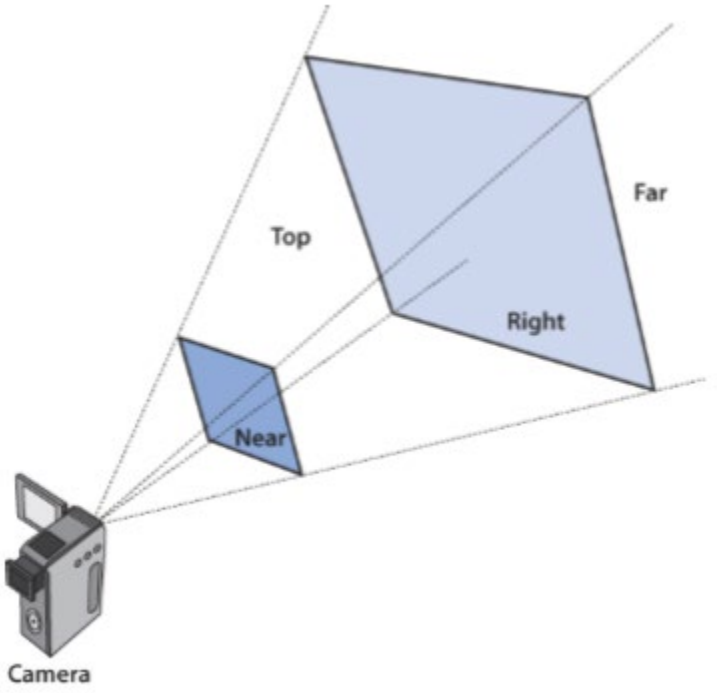
Danas računalne igre rijetko pate od manjka računalne snage, a razvojna su okruženja toliko uznapredovala da neka od njih posjeduju svoje vlastite optimizacijske algoritme koji u pozadini čiste, pripremaju i raspoređuju resurse unutar računalne igre. Iduća poglavlja opisuju osnovne komponente potrebne za kreiranje osnovne 3D računalne igre te se njihova implementacija primjenjuje u primjeru prototipa igre. Funkcionalnosti koje će se opisivati su:

- kreiranje 3D modela unutar scene
- pozicioniranje i rotacija 3D modela unutar scene
- 3D kolizija

- sustav čestica
- kamera.

15. Kamera

Kamera je osnovna komponenta svake računalne igre. Kamera ima zadaću kreiranja crtanja slike koja se prikazuje igraču. U pravilu će samo objekti koje kamera aktivno promatra biti iscrtani dok su svi objekti koje kamera ne promatra nevidljivi. Također, iz razloga što se prostor unutar 3D svijeta kreće u beskonačnost, kamera bi trebala iscrtati sve objekte od pozicije na kojoj se nalazi pa do beskonačnosti. Iz tog razloga uvedena je tehnika pod nazivom Clipping plane s kojom se određuje minimalna i maksimalna točka iscrtavanja objekata. Na slici 18. prikazana je kamera koja definira minimalnu i maksimalnu matricu iscrtavanja.



Slika 18. Minimalna i maksimalna matrica iscrtavanja (Learning XNA 4.0)

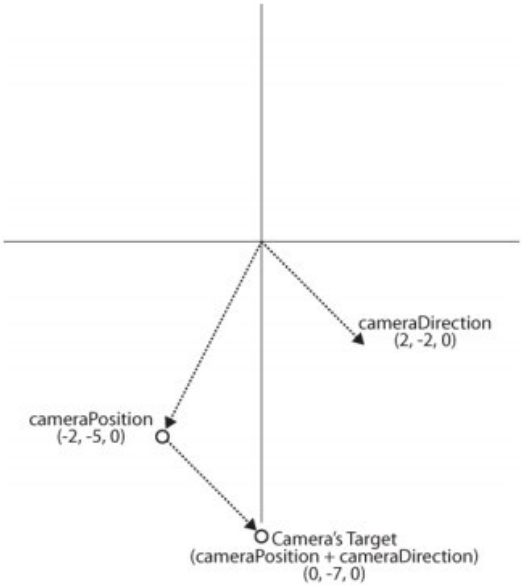
Primjer koda 22 prikazuje implementaciju komponente Camera korištenu za definiranje matrice prostora promatranja unutar scene.

```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Graphics;
3.  using Microsoft.Xna.Framework.Input;
4.  using System;
5.
6.  namespace Kernel.Components
7.  {
8.      public class Camera : Component
9.      {
10.         public Matrix view { get; set; }
11.
12.         public Viewport viewPort { get; set; }
13.
14.         public Matrix projection { get; set; }
15.
16.         public Vector3 LookPoint { get; set; }
17.
18.         public Vector3 Direction { get; set; }
19.
20.         public Vector3 cameraUp { get; set; }
21.
22.         public Camera(int clipNear, int clipFar, Viewport viewport)
23.         {
24.             projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2,
25.                 (float)viewport.Width
26.                 /(float)viewport.Height,
27.                 clipNear,
28.                 clipFar);
29.
30.             this.viewPort = viewport;
31.         }
32.
33.         public override void Destroy()
34.         {
35.         }
36.
37.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)

38.         {
39.         }
40.
41.         public override void LoadContent(GraphicsDevice graphicsDevice)
42.         {
43.             Direction = LookPoint - this.GameObject.transform.Position3D;
44.             Direction.Normalize();
45.             CreateLookAt();
46.         }
47.
48.         public override void Update(GameTime gameTime)
49.         {
50.             CreateLookAt();
51.         }
52.
53.         public override void UpdateReferences()
54.         {
55.         }
```

```
56.
57.         private void CreateLookAt()
58.         {
59.             var position = this.GameObject.transform.Position3D;
60.             view = Matrix.CreateLookAt(position, position + Direction, cameraUp);
61.         }
62.     }
63. }
```

Primjer koda 22 - Implementacija komponente Camera

Vrijednost view definira matricu prostora promatranja kamere. Objekti koje se nalaze unutar te matrice bit će iscrtani na ekranu. Vrijednost viewPort definira početni prikaz kamere. Vrijednost projection definira granice prostora koji kamera promatra uzimajući u obzir vrijednosti: prostor gledanja (engl. *field of view*), početni prikaz te minimalnu i maksimalnu točku iscrtavanja. Vrijednost LookPoint služi za određivanje početne točke gledanja, a vrijednost Direction služi za određivanje kuta gledanja. Vrijednost cameraUp služi za definiranje gornje osi kamere koja je uglavnom Y os. U konstruktoru klase definiramo vrijednosti projection i viewport. U funkciji LoadContent definiramo vrijednost Direction tako što od točke promatranja oduzmemo trenutno poziciju objekta koja se zatim normalizira. Na slici 17. prikazuje se pronalazak direkcijskog vektora unutar koordinatnog sustava.



Slika 19. Direkcijski vektor unutar koordinatnog sustava (Learning XNA 4.0)

Funkcija Update poziva funkciju CreateLookAt koja kreira novu matricu promatranja uzimajući u obzir poziciju objekta, direkciju gledanja te gornje osi kamere.

16. Dodavanje 3D modela na scenu

Budući da su MonoGame i XNA izgrađeni na temeljima DirectX-a svi modeli koji se žele koristiti unutar računalne igre moraju biti dodani u projekt u .x formatu. Primjer koda 23 prikazuje implementaciju komponente Model3D koja se koristi za dodavanje 3D modela unutar scene.

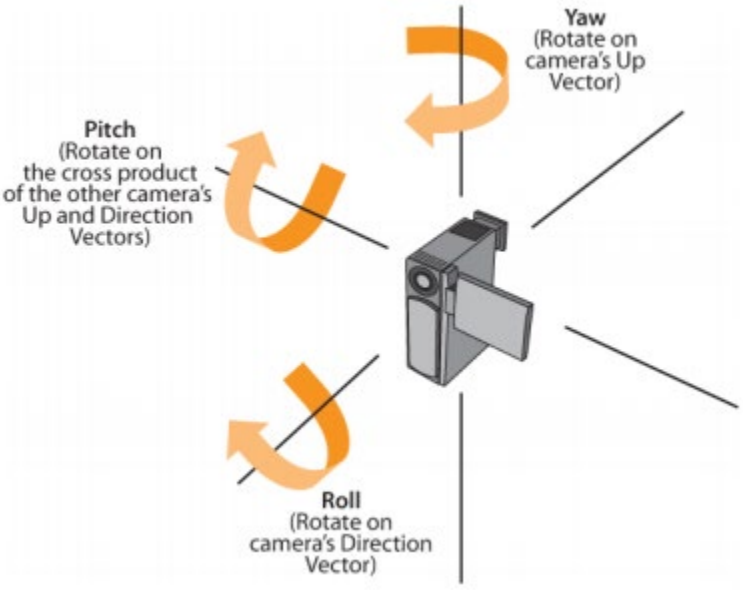
```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Graphics;
3.  using System;
4.  using System.Collections.Generic;
5.  using System.Linq;
6.  using System.Text;
7.  using System.Threading.Tasks;
8.
9.  namespace Kernel.Components
10. {
11.     public class Model3D : Component
12.     {
13.         public Model model { get; protected set; }
14.
15.         public Matrix world = Matrix.Identity;
16.
17.         public Matrix rotation = Matrix.Identity;
18.
19.         public float yawAngle { get; set; }
20.
21.         public float pitchAngle { get; set; }
22.
23.         public float rollAngle { get; set; }
24.
25.         public Vector3 direction { get; set; }
26.
27.         public Model3D(Model model, Camera camera,Vector3 position = default(Vector3))

28.         {
29.             this.model = model;
30.             this.world = Matrix.CreateTranslation(position);
31.         }
32.
33.         public override void LoadContent(GraphicsDevice graphicsDevice)
34.         {
35.         }
36.
37.         public override void Update(GameTime gameTime)
38.         {
39.             if (!Enabled) return;
40.
```

```
41.             rotation *= Matrix.CreateFromYawPitchRoll(yawAngle, pitchAngle, rollAngle);
42.
43.             world *= Matrix.CreateTranslation(direction);
44.         }
45.
46.         public override void Draw(SpriteBatch spriteBatch = null, Camera camera = null)
47.         {
48.             if (!Enabled || !Visible) return;
49.
50.             var transforms = new Matrix[model.Bones.Count];
51.             model.CopyAbsoluteBoneTransformsTo(transforms);
52.
53.             foreach (var mesh in model.Meshes)
54.             {
55.                 foreach(BasicEffect basicEffects in mesh.Effects)
56.                 {
57.                     basicEffects.EnableDefaultLighting();
58.                     basicEffects.Projection = camera.projection;
59.                     basicEffects.View = camera.view;
60.                     basicEffects.World = GetWorld() * mesh.ParentBone.Transform;
61.                 }
62.                 mesh.Draw();
63.             }
64.         }
65.
66.         public virtual Matrix GetWorld()
67.         {
68.             return rotation * world;
69.         }
70.
71.         public override void UpdateReferences()
72.         {
73.         }
74.
75.         public override void Destroy()
76.         {
77.             Enabled = false;
78.             Visible = false;
79.         }
80.
81.         public void SetPosition(Vector3 position)
82.         {
83.             this.GameObject.transform.Position3D = position;
84.             world = Matrix.CreateTranslation(position);
85.         }
86.
87.         public bool IsColiding(Model otherModel, Matrix otherWorld)
88.         {
89.             foreach(var thisModel in model.Meshes)
90.             {
91.                 foreach(var otherMesh in otherModel.Meshes)
92.                 {
93.                     if (thisModel.BoundingSphere.Transform(GetWorld()).
94.                         Intersects(otherMesh.BoundingSphere.Transform(otherWorld)))
95.                     {
96.                         return true;
97.                     }
98.                 }
99.             }
```

```
100.
101.                     return false;
102.                 }
103.
104.             }
105.         }
```

Primjer koda 23 - Implementacija komponente Model3D

Vrijednost model definira odabrani model koji će se iscrtavati unutar scene. Vrijednost world definira matricu prostora u kojoj se objekt nalazi, također u slučaju da objektu dodamo smjer gibanja, ova vrijednost će zapravo biti zaslužena za pomicanje objekta unutar scene. Vrijednost rotation definira rotaciju objekta unutar scene. Vrijednosti yawAngle, pitchAngle i rollAngle služe za definiranje rotacije objekta unutar scene. Na slici 20 možemo vidjeti da yawAngle definira rotaciju oko Y, pitchAngle rotaciju oko X i rollAngle rotaciju oko Z osi. (Slika preuzeta s Learning XNA 4.0.)



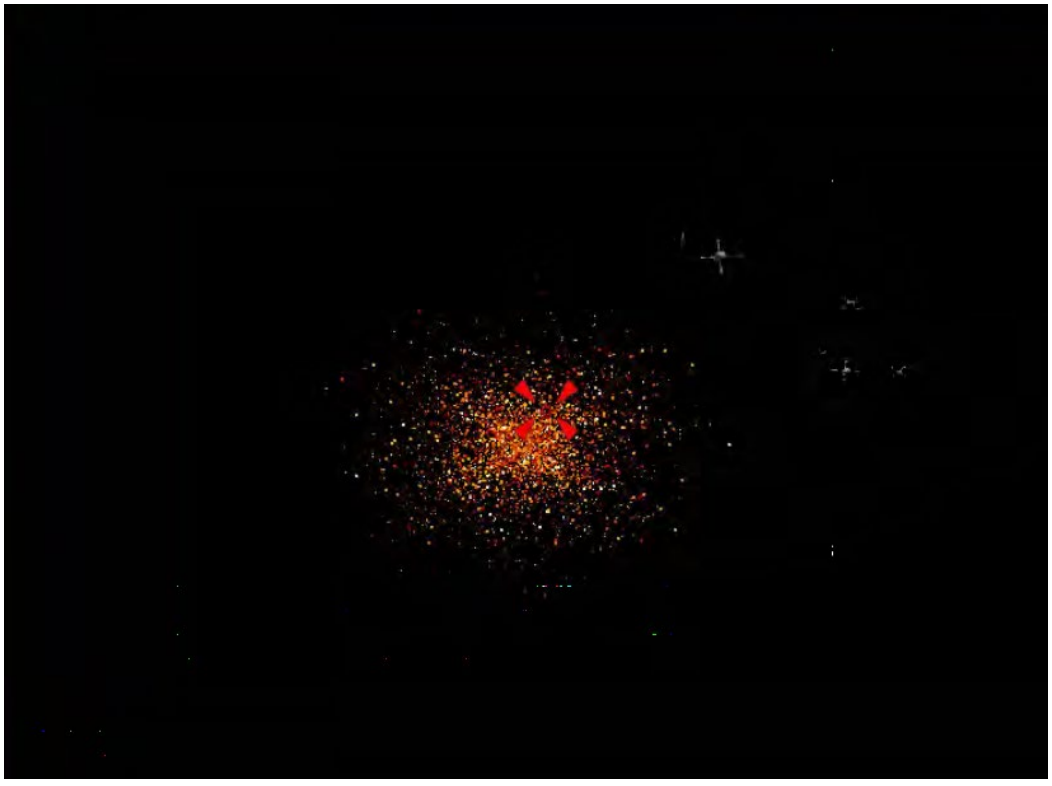
Slika 20. Rotacija kamere po X, Y i Z osi (Learning XNA 4.0)

Funkcija Update služi za ažuriranje pozicije i rotacije objekta unutar scene. Funkcija Draw iscrtava objekt unutar scene iteracijom kroz mreže (engl. *Meshes*). Vrijednosti mreže sadrže informacije o relacijama između pojedinih točki i linija koje definiraju objekt, stoga se može reći da skup mreža čini objekt. Zatim se svakoj mreži unutar objekta definira

efekt koji zavisi o kutu gledanja kamere i izboru svjetlosti. Razlog je to što elementi mreže mogu drugačije reagirati ili biti prikazani ovisno o kutu i udaljenosti iz koje se promatraju. Funkcija SetPosition mijenja poziciju objekta na vrijednost position koju prima kao parametar. Funkcija IsColiding služi za provjeru kolizije sa specifičnim objektom.

17. Sustav čestica

Sustav čestica je tehnika u računalnim igrama koja koristi ogroman broj manjih slika, 3D modela ili drugih grafičkih objekata za simulaciju određenih fenomena koje inače ne bi bilo moguće simulirati normalnim tehnikama iscrtavanja. Na slici 21. prikazan je izgled sustava čestica.



Slika 21. Izgled sustava čestica unutar prototipa igre

Za implementaciju sustava čestica potrebno je prvo definirati vrijednosti koje želimo koristiti unutar igre. Primjer koda 24 definira postavke korištene za definiciju sustava čestica

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.  using System.Threading.Tasks;
6.
7.  namespace Kernel.Components
8.  {
9.      public class ParticleSettings
10.     {
11.         public int maxSize = 2;
12.     }
13.
14.     public class ParticleExplosionSettings
15.     {
16.         // Life of particles
17.         public int minLife = 1000;
18.         public int maxLife = 2000;
19.         // Particles per round
20.         public int minParticlesPerRound = 100;
21.         public int maxParticlesPerRound = 600;
22.         // Round time
23.         public int minRoundTime = 16;
24.         public int maxRoundTime = 50;
25.         // Number of particles
26.         public int minParticles = 2000;
27.         public int maxParticles = 3000;
28.     }
29. }
```

Primjer koda 24 - Implementacija postavki sustava čestica

Klasa ParticleSetting definira samo jednu vrijednost maxSize koje služi za definiranje maksimalne veličine pojedine čestice. Klasa ParticleExplosionSettings definira minimalne i maksimalne granice pojedine čestice. Vrijednosti minLife i maxLife definiraju minimalan maksimalan životni vijek čestice. Vrijednost minParticesPerRount i maxParticlesPerRound definiraju minimalan i maksimalan broj čestica po jednoj rundi (otkucaju). Vrijednosti minRoundTime i maxRoundTime definiraju minimalan i maksimalan broj rundi. Runde označavaju broj ponavljanja ispuštanja čestica. Vrijednosti minParticles i maxParticles označavaju minimalan i maksimalan broj čestica ukupno. Ove vrijednosti postoje da maksimalan broj čestica ne bi bio premalen odnosno neprimjetan ili prevelik kako ne bi naštetio performansama unutar igre.

Primjer koda 25 definira implementaciju klase ParticleExplosion koja služi za kreiranje i raspršivanje čestica unutar scene.

```
1.  using Microsoft.Xna.Framework;
2.  using Microsoft.Xna.Framework.Graphics;
3.  using System;
4.  using System.Collections.Generic;
5.  using System.Linq;
6.  using System.Text;
7.  using System.Threading.Tasks;
8.
9.  namespace Kernel.Components
10. {
11.     public class ParticleExplosion : Component
12.     {
13.         public ParticleExplosion(GraphicsDevice graphicsDevice, Vector3 position,
14.              int lifeLeft, int roundTime, int numParticlesPerRound, int maxParticles,
15.              Texture2D particleColorsTexture, ParticleSettings particleSettings,
16.              Effect particleEffect)
17.         {
18.             this.lifeLeft = lifeLeft;
19.             this.numParticlesPerRound = numParticlesPerRound;
20.             this.maxParticles = maxParticles;
21.             this.roundTime = roundTime;
22.             this.graphicsDevice = graphicsDevice;
23.             this.particleSettings = particleSettings;
24.             this.particleEffect = particleEffect;
25.             this.particleColorsTexture = particleColorsTexture;
26.         }
27.         // Particle arrays and vertex buffer
28.         VertexPositionTexture[] verts;
29.         Vector3[] vertexDirectionArray;
30.         Color[] vertexColorArray;
31.         VertexBuffer particleVertexBuffer;
32.
33.         // Life
34.         int lifeLeft;
35.
36.         // Rounds and particle counts
37.         int numParticlesPerRound;
38.         int maxParticles;
39.         static Random rnd = new Random();
40.         int roundTime;
41.         int timeSinceLastRound = 0;
42.
43.         // Vertex and graphics info
44.         GraphicsDevice graphicsDevice;
45.
46.         // Settings
47.         ParticleSettings particleSettings;
48.
49.         // Effect
50.         Effect particleEffect;
51.
52.         // Textures
53.         Texture2D particleColorsTexture;
54.
```

```
55.         // Array indices
56.         int endOfLiveParticlesIndex = 0;
57.         int endOfDeadParticlesIndex = 0;
58.
59.         public bool IsDead
60.         {
61.             get { return endOfDeadParticlesIndex == maxParticles; }
62.         }
63.
64.         private void InitializeParticleVertices()
65.         {
66.             // Instantiate all particle arrays
67.             verts = new VertexPositionTexture[maxParticles * 4];
68.             vertexDirectionArray = new Vector3[maxParticles];
69.             vertexColorArray = new Color[maxParticles];
70.             // Get color data from colors texture
71.             Color[] colors = new Color[particleColorsTexture.Width *
72.             particleColorsTexture.Height];
73.             particleColorsTexture.GetData(colors);
74.             // Loop until max particles
75.             for (int i = 0; i < maxParticles; ++i)
76.             {
77.                 float size = (float)rnd.NextDouble() * particleSettings.maxSize;
78.                 // Set position, direction and size of particle
79.                 verts[i * 4] = new VertexPositionTexture(this.GameObject.transform.Posi
    tion3D, new Vector2(0, 0));
80.                 verts[(i * 4) + 1] = new VertexPositionTexture(new Vector3(this.GameObj
    ect.transform.Position3D.X,
81.                 this.GameObject.transform.Position3D.Y + size, this.GameObject.transfor
    m.Position3D.Z), new Vector2(0, 1));
82.                 verts[(i * 4) + 2] = new VertexPositionTexture(new Vector3(this.GameObj
    ect.transform.Position3D.X + size,
83.                 this.GameObject.transform.Position3D.Y, this.GameObject.transform.Posit
    ion3D.Z), new Vector2(1, 0));
84.                 verts[(i * 4) + 3] = new VertexPositionTexture(new Vector3(this.GameObj
    ect.transform.Position3D.X + size,
85.                 this.GameObject.transform.Position3D.Y + size, this.GameObject.transfor
    m.Position3D.Z), new Vector2(1, 1));
86.                 // Create a random velocity/direction
87.                 Vector3 direction = new Vector3(
88.                 (float)rnd.NextDouble() * 2 - 1,
89.                 (float)rnd.NextDouble() * 2 - 1,
90.                 (float)rnd.NextDouble() * 2 - 1);
91.                 direction.Normalize();
92.                 // Multiply by NextDouble to make sure that
93.                 // all particles move at random speeds
94.                 direction *= (float)rnd.NextDouble();
95.                 // Set direction of particle
96.                 vertexDirectionArray[i] = direction;
97.                 // Set color of particle by getting a random color from the texture
98.                 vertexColorArray[i] = colors[(
99.                 rnd.Next(0, particleColorsTexture.Height) * particleColorsTexture.Width
    ) +
100.                     rnd.Next(0, particleColorsTexture.Width)];
101.                 }
102.                 // Instantiate vertex buffer
103.                 particleVertexBuffer = new VertexBuffer(graphicsDevice,
104.                 typeof(VertexPositionTexture), verts.Length, BufferUsage.None);
105.
106.             }
107.
```

```
108.                public override void LoadContent(GraphicsDevice graphicsDevice)
109.                {
110.                    InitializeParticleVertices();
111.                }
112.
113.                public override void Update(GameTime gameTime)
114.                {
115.                    // Decrement life left until it's gone
116.                    if (lifeLeft > 0)
117.                        lifeLeft -= gameTime.ElapsedGameTime.Milliseconds;
118.                    // Time for new round?
119.                    timeSinceLastRound += gameTime.ElapsedGameTime.Milliseconds;
120.                    if (timeSinceLastRound > roundTime)
121.                    {
122.                        // New round - add and remove particles
123.                        timeSinceLastRound -= roundTime;
124.                        // Increment end of live particles index each
125.                        // round until end of list is reached
126.                        if (endOfLiveParticlesIndex < maxParticles)
127.                        {
128.                            endOfLiveParticlesIndex += numParticlesPerRound;
129.                            if (endOfLiveParticlesIndex > maxParticles)
130.                                endOfLiveParticlesIndex = maxParticles;
131.                        }
132.                        if (lifeLeft <= 0)
133.                        {
134.                            // Increment end of dead particles index each
135.                            // round until end of list is reached
136.                            if (endOfDeadParticlesIndex < maxParticles)
137.                            {
138.                                endOfDeadParticlesIndex += numParticlesPerRound;
139.                                if (endOfDeadParticlesIndex > maxParticles)
140.                                    endOfDeadParticlesIndex = maxParticles;
141.                            }
142.                        }
143.                    }
144.                    // Update positions of all live particles
145.                    for (int i = endOfDeadParticlesIndex;
146.                    i < endOfLiveParticlesIndex; ++i)
147.                    {
148.                        verts[i * 4].Position += vertexDirectionArray[i];
149.                        verts[(i * 4) + 1].Position += vertexDirectionArray[i];
150.                        verts[(i * 4) + 2].Position += vertexDirectionArray[i];
151.                        verts[(i * 4) + 3].Position += vertexDirectionArray[i];
152.                    }
153.
154.                }
155.
156.                public override void Draw(SpriteBatch spriteBatch = null, Camera camera
    = null)
157.                {
158.                    graphicsDevice.SetVertexBuffer(particleVertexBuffer);
159.                    // Only draw if there are live particles
160.                    if (endOfLiveParticlesIndex - endOfDeadParticlesIndex > 0)
161.                    {
162.                        for (int i = endOfDeadParticlesIndex; i < endOfLiveParticlesInde
    x; ++i)
163.                        {
164.
165.                            // Draw particles
```

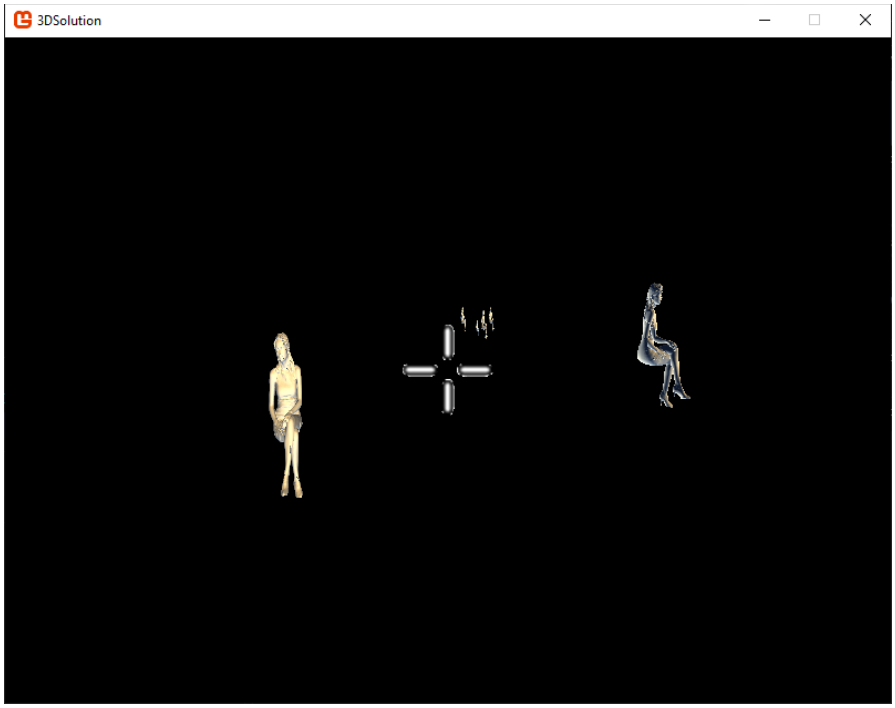
```
166.                             foreach (EffectPass pass in particleEffect.CurrentTechnique.
    Passes)
167.                             {
168.                                 pass.Apply();
169.                                 graphicsDevice.DrawUserPrimitives<VertexPositionTexture>
    (
170.                                 PrimitiveType.TriangleStrip,
171.                                 verts, i * 4, 2);
172.                             }
173.                         }
174.
175.                     }
176.                 }
177.
178.                 public override void UpdateReferences()
179.                 {
180.                     InitializeParticleVertices();
181.                 }
182.
183.                 public override void Destroy()
184.                 {
185.                 }
186.             }
187.         }
```

Primjer koda 25 - Implementacija komponente sustava čestica

U konstruktoru se inicijaliziraju postavke potrebne za rad sustava čestica. Učitavaju se vrijednosti iz objekta tipa ParticleSettings koje će se koristiti prilikom stvaranja čestica. Funkcija InitializeParticleVertices definira matricu koja će sadržavati informacije o svim trenutno aktivnim česticama. Zatim u for petlji se iterira sve dok maksimalna iterativna vrijednost ne dosegne vrijednost maxParticles. U iteraciji se definira vrijednost te se zatim definiraju 4 osnovne točke čestice koje će se koristiti prilikom iscrtavanja čestice. Zatim se određuje nasumična normalizirana direkcija čestice kako bi svaka čestica išla u različitom smjeru gotovo jednakom brzinom, na samom kraju se određuje boja između točki radi vizualnog ugođaja. Funkcija Update iterira kroz matricu te ažurira sve vrijednosti poput smjera i pozicije dok istovremeno stvara nove i uništava stare čestice unutar scene. Funkcija Draw iscrtava čestice unutar scene te koristi vrijednosti endOfDeadParticlesIndex kako bi odredila koje su čestice završile sa svojim životnim vijekom te ih nije potrebno više iscrtavati.

18. Implementacija prototipa 3D računalne igre

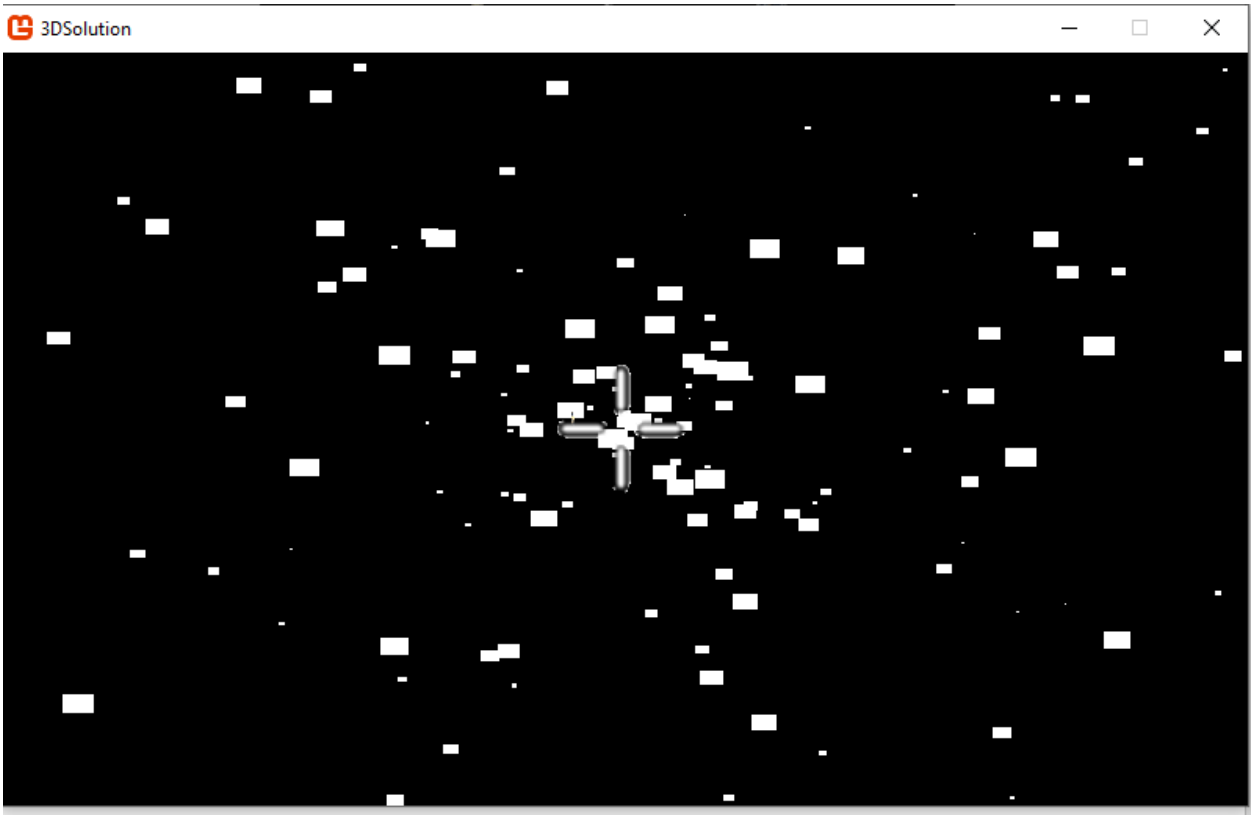
Budući da je implementacija korisničkog sučelja opisana u 2D prototipu računalne igre, ovaj prototip će biti isključivo fokusiran na igranje unutar računalne igre. Na primjeru prototipa 3D igre pod nazivom „Pogodi kraljicu“ opisat ćemo implementaciju svih komponenti navedenih u prošlim poglavljima. Iako igra ne posjeduje kompleksne mehanike, brojna ograničenja MonoGame biblioteke zahtijevaju njihovu jako kompleksnu implementaciju. Cilj igre je pogoditi kraljicu prije nego ona dodirne igrača. Na slici 22. prikazan je finalni izgled računalne igre.



Slika 22. Izgled prototipa 3D računalne igre

Igrač je statički pozicioniran na nultoj poziciji unutar scene. U sredini ekrana posjeduje ciljnik koji označava smjer gledanja (poziciju miša). Igrač pritiskom na lijevi klik miša ispaljuje metak u smjeru gledanja. Na nasumičnoj poziciji i u nasumičnim vremenskim intervalima stvaraju se objekti (kraljice) koji putuju prema igraču. Igrač ima cilj pogoditi objekte prije nego što ga dodirnu kako bi nastavio igrati i napredovati na sljedeću razinu. Svaka nova razina stvara nove objekte bliže igraču i u kraćim vremenskim intervalima. U

slučaju da se postigne određeni broj objekata koje igrač nije uspio uništiti, igra završava. Kad metak dodirne objekt, objekt se uništava te se sustav čestica aktivira koji daje dodatan vizualni ugođaj igri. Na slici 23. prikazana je aktivacija sustava čestica prilikom pogotka objekta unutar prototipa računalne igre.



Slika 23. Aktivacija sustava čestica prilikom pogotka

Za početak je definirana klasa pod nazivom CameraObject koju će igrač koristiti za promatranje događanja unutar scene računalne igre. Primjer koda 26 prikazuje implemetaciju objekta CameraObject.

```
1.  using Kernel.Components;
2.  using Kernel.Design;
3.  using Microsoft.Xna.Framework;
4.  using Microsoft.Xna.Framework.Graphics;
5.  using Microsoft.Xna.Framework.Input;
6.  using System;
7.
8.  namespace _3DSolution.Scripts
9.  {
10.     public class CameraObject : MonoBehaviour
```

```
    {
        public Camera camera { get; set; }

        public MouseState prevMouseState { get; set; }
        private readonly float totalYaw = MathHelper.PiOver4 / 2;
        private float currentYaw = 0;
        private readonly float totalPitch = MathHelper.PiOver4 / 2;
        private float currentPitch = 0;

        private SpriteRenderer spriteRenderer { get; set; }

        public CameraObject(GraphicsDevice graphicsDevice) : base(graphicsDevice)
        {
        }

        public override void Update(GameTime gameTime)
        {
            ////if (Keyboard.GetState().IsKeyDown(Keys.W))
            ////    gameObject.transform.Position3D += camera.LookDirection * speed;
            ////if (Keyboard.GetState().IsKeyDown(Keys.S))
            ////    gameObject.transform.Position3D -= camera.LookDirection * speed;
            ////if (Keyboard.GetState().IsKeyDown(Keys.A))
            ////    gameObject.transform.Position3D += Vector3.Cross(Vector3.Up ,camera.LookDirection) * speed;
            ////if (Keyboard.GetState().IsKeyDown(Keys.D))
            ////    gameObject.transform.Position3D -= Vector3.Cross(Vector3.Up, camera.LookDirection) * speed;

            // Yaw
            float yawAngle = (-MathHelper.PiOver4 / 1000) *
                (Mouse.GetState().X - prevMouseState.X);

            if (Math.Abs(currentYaw + yawAngle) < totalYaw)
            {
                camera.Direction = Vector3.Transform(camera.Direction,
                    Matrix.CreateFromAxisAngle(camera.cameraUp, yawAngle));

                currentYaw += yawAngle;
            }

            float pitchAngle = (MathHelper.PiOver4 / 1500) *
                (Mouse.GetState().Y - prevMouseState.Y);

            if (Math.Abs(currentPitch + pitchAngle) < totalPitch)
            {
                var directionNorm = camera.Direction;
                directionNorm.Normalize();
                camera.Direction = Vector3.Transform(camera.Direction,
                    Matrix.CreateFromAxisAngle(Vector3.Cross(camera.cameraUp, directionNorm), pitchAngle));

                currentPitch += pitchAngle;
            }

            prevMouseState = Mouse.GetState();

        }

        protected override void InitComponents()
        {
            var vp1 = graphicsDevice.Viewport;
```

```
69.
70.             vp1.Height = (graphicsDevice.Viewport.Height / 2);
71.
72.             camera = new Camera(1, 3000, vp1)
73.             {
74.                 LookPoint = new Vector3(30, 100, 5),
75.                 cameraUp = Vector3.Up,
76.             };
77.
78.             this.gameObject.AddComponent(camera);
79.             camera.GameObject.transform.Position3D = new Vector3(30, 100, 20);
80.
81.             prevMouseState = Mouse.GetState();
82.             Mouse.SetPosition(Game1.singleton.Window.ClientBounds.Width / 2,
83.             Game1.singleton.Window.ClientBounds.Height / 2);
84.
85.             spriteRenderer = new SpriteRenderer(@"Content/images/crosshair.png");
86.
87.             spriteRenderer.DrawAction = new Action<SpriteBatch>((a) =>
88.             {
89.                 a.Draw(spriteRenderer.Texture,
90.                     new Vector2((Game1.singleton.Window.ClientBounds.Width / 2)
91.                     - (spriteRenderer.Texture.Width / 2),
92.                     (Game1.singleton.Window.ClientBounds.Height / 2)
93.                     - (spriteRenderer.Texture.Height / 2)),
94.                     Color.White);
95.             });
96.
97.             this.gameObject.AddComponent(spriteRenderer);
98.
99.         }
100.        }
101.    }
```

Primjer koda 26 - Implementacija objekta CameraObject

Vrijednost camera sadrži referencu na kameru koja se koristi od strane objekta za kreiranje matrice prostora koji se prikazuje. Vrijednost prevMouseState sadrži prijašnje stanje miša. Vrijednosti totalYaw i totalPitch prikazuju maksimalnu dopuštenu rotaciju objekta oko Y i X osi. Vrijednosti currentYaw i currentPitch predstavljaju trenutnu rotaciju kamere po X i Y osi. Vrijednost spriteRenderer sadrži referencu na sliku koja će se koristiti kao ciljnik unutar scene. U funkciji InitComponents inicijaliziraju se vrijednosti te se na vrijednost spriteRenderer dodaje akcija koja će uvijek neovisno o promjeni rezolucije ekrana pozicionirati ciljnik na sredinu ekrana. U funkciji Update prvo se vrši kalkulacija rotacije po X i Y osi tako što se izračuna delta između prijašnjeg i trenutnog stanja miša, a zatim se iste vrijednosti primjenjuju na transformaciju kamere kako bi se dobila ispravna pozicija točke gledanja u odnosu na miš.

Iduća komponenta koja je potrebna za realizaciju igre je neprijatelj kojeg će igrač moći gađati. Objekt Enemy sadrži logiku neprijatelja, a implementirana je u primjeru koda 27.

```
1.  using Kernel.Components;
2.  using Kernel.Design;
3.  using Microsoft.Xna.Framework;
4.  using Microsoft.Xna.Framework.Graphics;
5.  using System.Collections.Generic;
6.
7.  namespace _3DSolution.Scripts
8.  {
9.      public class Enemy : MonoBehaviour
10.     {
11.         private Camera camera { get; set; }
12.
13.         public Model3D model { get; set; }
14.
15.         public ParticleExplosionSettings particleExplosionSettings = new ParticleExplos
    ionSettings();
16.         public ParticleSettings particleSettings = new ParticleSettings();
17.         public Texture2D explosionTexture;
18.         public Texture2D explosionColorsTexture;
19.         public Effect explosionEffect;
20.
21.         public Enemy(GraphicsDevice graphicsDevice, Camera camera)
22.             :this(graphicsDevice)
23.         {
24.             this.camera = camera;
25.         }
26.
27.         public Enemy(GraphicsDevice graphicsDevice) : base(graphicsDevice)
28.         {
29.         }
30.
31.         public override void Update(GameTime gameTime)
32.         {
33.
34.             if (model.GetWorld().Translation.Z > camera.GameObject.transform.Position3D
    .Z + 100)
35.             {
36.                this.Destroy();
37.             }
38.         }
39.
40.         protected override void InitComponents()
41.         {
42.             model = new Model3D(Game1.singleton.Content.Load<Model>(@"Models\lowpoly_ma
    x"), camera, this.gameObject.transform.Position3D);
43.             this.gameObject.AddComponent(model);
44.
45.             var spriteRenderer = new SpriteRenderer(@"Content/images/crosshair.png");
46.             spriteRenderer.LoadContent(graphicsDevice);
47.             explosionTexture = spriteRenderer.Texture;
48.             explosionColorsTexture = spriteRenderer.Texture;
49.             explosionEffect = new BasicEffect(graphicsDevice);
50.
51.         }
```

```
52.     }
53. }
```

Primjer koda 27 - Implementacija objekta Enemy

Vrijednost camera služi kao referenca odredišne točke, a budući da je igrač ujedno i kamera, svaki neprijatelj će se uvijek referencirati na njega. Vrijednost model sadrži referencu na model koji definira neprijatelja. Vrijednosti particleSettings, explosionTexture, explosionColorTexture i explosionEffect definiraju postavke sustava čestica koji će se koristiti u slučaju da neprijatelj bude pogođen. U funkciji InitComponents inicijaliziramo model neprijatelja pod nazivom lowpoly_max te inicijaliziramo teksturu koja će se koristiti za sustav čestica. U funkciji ažuriranja provjeravamo je li neprijatelj došao dovoljno blizu igraču te ga uništavamo u slučaju da je.

Budući da će se neprijatelji stvarati u sceni u nasumičnom vremenu, potrebno je uraditi komponentu koja će to omogućiti. Objekti EnemySpawnerData i EnemySpawner sadrže logiku za stvaranje neprijatelja unutar scene, a implementirane su u primjeru koda 28.

```
1.  using Kernel.Components;
2.  using Kernel.Design;
3.  using Microsoft.Xna.Framework;
4.  using Microsoft.Xna.Framework.Graphics;
5.  using System;
6.  using System.Collections.Generic;
7.  using System.Linq;
8.  using System.Text;
9.  using System.Threading.Tasks;
10.
11. namespace _3DSolution.Scripts
12. {
13.     public class EnemySpawnerData
14.     {
15.         public int minSpawnTime { get; set; }
16.         public int maxSpawnTime { get; set; }
17.
18.         public int numberEnemies { get; set; }
19.         public int minSpeed { get; set; }
20.         public int maxSpeed { get; set; }
21.         // Misses
22.         public int missesAllowed { get; set; }
23.
24.         public EnemySpawnerData(int minSpawnTime, int maxSpawnTime, int numberEnemies,
    int minSpeed, int maxSpeed, int missesAllowed)
25.         {
26.             this.minSpawnTime = minSpawnTime;
27.             this.maxSpawnTime = maxSpawnTime;
28.             this.numberEnemies = numberEnemies;
29.             this.minSpeed = minSpeed;
```

```
            this.maxSpeed = maxSpeed;
            this.missesAllowed = missesAllowed;
        }
    }

    public class EnemySpawner : MonoBehaviour
    {
        public Random rnd { get; set; }

        private readonly Vector3 maxSpawnLocation = new Vector3(500, 100, -3000);

        private int nextSpawnTime = 0;

        private int timeSinceLastSpawn = 0;

        private float maxRollAngle = MathHelper.Pi / 40;

        private int enemiesThisLevel = 0;

        private int missedThisLevel = 0;

        private int currentLevel = 0;

        public List<EnemySpawnerData> enemySpawnersData { get; set; }

        public List<Enemy> enemies { get; set; }

        private Camera camera { get; set; }

        public List<ParticleExplosion> particleExplosionsOfDeadEnemies { get; set; }

        public EnemySpawner(Camera camera, GraphicsDevice graphicsDevice) : base(graphi
csDevice)
        {
            this.camera = camera;
            this.enemies = new List<Enemy>();
            this.particleExplosionsOfDeadEnemies = new List<ParticleExplosion>();
        }

        public override void Update(GameTime gameTime)
        {
            for (int i = 0; i < particleExplosionsOfDeadEnemies.Count; ++i)
            {
                // If explosion is finished, remove it
                if (particleExplosionsOfDeadEnemies[i].IsDead)
                {
                    this.gameObject.RemoveComponent(particleExplosionsOfDeadEnemies[i])
;
                    particleExplosionsOfDeadEnemies.RemoveAt(i);
                    --i;
                }
            }


            if (enemiesThisLevel < enemySpawnersData[currentLevel].numberEnemies)
            {
                timeSinceLastSpawn += gameTime.ElapsedGameTime.Milliseconds;
                if (timeSinceLastSpawn > nextSpawnTime)
                {
                    SpawnEnemy();
                }
```

```
89.             }
90.         }
91.
92.         private void SetNextSpawnTime()
93.         {
94.             nextSpawnTime = rnd.Next(enemySpawnersData[currentLevel].minSpawnTime,
95.                 enemySpawnersData[currentLevel].maxSpawnTime);
96.             timeSinceLastSpawn = 0;
97.         }
98.
99.         private void SpawnEnemy()
100.                {
101.                    var position = new Vector3(rnd.Next(
102.                        -(int)maxSpawnLocation.X, (int)maxSpawnLocation.X),
103.                        rnd.Next(-
    (int)maxSpawnLocation.Y, (int)maxSpawnLocation.Y), maxSpawnLocation.Z);
104.
105.                    var enemy = new Enemy(graphicsDevice, camera);
106.                    enemy.model.SetPosition(position);
107.
108.                    enemy.model.direction = new Vector3(
109.                        rnd.Next(-(int)maxSpawnLocation.X, (int)maxSpawnLocation.X),
110.                        rnd.Next(-(int)maxSpawnLocation.Y, (int)maxSpawnLocation.Y),
111.                        30000);
112.                    enemy.model.direction = new Vector3(0, 0, rnd.Next(enemySpawnersData
    [currentLevel].minSpeed, enemySpawnersData[currentLevel].maxSpeed));
113.                    // enemy.model.rollAngle = (float)rnd.NextDouble() * maxRollAngle -
    (maxRollAngle / 2);
114.
115.                    enemies.Add(enemy);
116.
117.                    enemiesThisLevel++;
118.                    SetNextSpawnTime();
119.                }
120.
121.                protected override void InitComponents()
122.                {
123.                    rnd = new Random();
124.                    enemySpawnersData = new List<EnemySpawnerData>();
125.                    enemySpawnersData.Add(new EnemySpawnerData(1000, 3000, 20, 2, 6, 10)
    );
126.                    enemySpawnersData.Add(new EnemySpawnerData(900, 2800, 22, 2, 6, 9));

127.                    enemySpawnersData.Add(new EnemySpawnerData(800, 2600, 24, 2, 6, 8));

128.                    SetNextSpawnTime();
129.                }
130.            }
131.        }
```

Primjer koda 28 - Implementacija objekta EnemySpawner i EnemySpawnerData

U klasi EnemySpawner dane vrijednosti minSpawnTime i maxSpawnTime služe za određivanje minimalnog i maksimalnog razmaka između stvaranja novih neprijatelja unutar scene. Vrijednost numberEnemies označava maksimalan broj neprijatelja unutar

scene. Vrijednosti minSpeed i maxSpeed služe za određivanje minimalne i maksimalne brzine kretanja neprijatelja unutar scene. Vrijednost missesAllowed označava maksimalan dozvoljeni broj promašaja. U klasi EnemySpawner vrijednost rnd služi kao nasumični generator vrijednosti. Vrijednost maxSpawnLocation određuje najdalju moguću lokaciju na kojoj se neprijatelj može stvoriti. Vrijednost nexSpawnTime označava vrijeme idućeg stvaranja neprijatelja unutar scene. Vrijednost timeSinceLastSpawn označava vrijeme koje je prošlo od stvaranja zadnjeg neprijatelja. Vrijednost maxRollAngle označava maksimalnu rotaciju objekta po Z osi. Vrijednost enemiesThisLevel označava broj neprijatelja koji su se stvorili od početka igranja, a vrijednost missedThisLevel označava broj promašenih neprijatelja. Vrijednost currentLevel označava trenutnu razinu igra, povećanjem razine proporcionalno raste i težina igranja. Kolekcija enemySpawnersData sadrži podatke o razinama unutar igre, a kolekcija enemies sadrži popis svih neprijatelja unutar scene. Vrijednost camera sadrži referencu na kameru unutar scene odnosno na poziciju igrača. Kolekcija particleExplosionsOfDeadEnemies sadrži sve čestice koje su trenutno aktivne unutar scene. U funkciji Update prvo iteriramo kroz kolekciju particleExplosionsOfDeadEnemies te brišemo čestice koje su završile svoj životni ciklus, zatim provjeravamo jesu li uvjeti zadovoljeni da se stvori novi neprijatelj te u slučaju da jesu, stvaramo novog neprijatelja. U funkciji SpawnEnemy stvaramo novog neprijatelja te mu određujemo brzinu, smjer i poziciju gibanja dok u funkciji SetNextSpawnTime određujemo iduće vrijeme stvaranja novog neprijatelja. U funkciji InitComponents inicijaliziramo kolekciju enemySpawnersData te joj dodajemo 3 razine težine.

Klasa Bullet stvara hitac koji ako pogodi neprijatelja uništava ga te aktivira njegov sustav čestica. Primjer koda 29 prikazuje implementaciju objekta Bullet.

```
1.  using Kernel.Components;
2.  using Kernel.Design;
3.  using Microsoft.Xna.Framework;
4.  using Microsoft.Xna.Framework.Graphics;
5.  using Microsoft.Xna.Framework.Input;
6.  using System;
7.  using System.Collections.Generic;
8.  using System.Linq;
9.  using System.Text;
10. using System.Threading.Tasks;
11.
12. namespace _3DSolution.Scripts
```

```
13. {
14.     public class Bullets : MonoBehaviour
15.     {
16.         public List<Model3D> bullets { get; set; }
17.         float shotMinZ = -3000;
18.
19.         private CameraObject cameraObject { get; set; }
20.
21.         private EnemySpawner enemySpawner { get; set; }
22.
23.         private float shotSpeed = 10;
24.
25.         private float shotDelay = 300;
26.
27.         private float shootCountdown = 0;
28.
29.         public Bullets(EnemySpawner enemySpawner, CameraObject cameraObject, GraphicsDe
    vice graphicsDevice) : base(graphicsDevice)
30.         {
31.             this.cameraObject = cameraObject;
32.             bullets = new List<Model3D>();
33.             this.enemySpawner = enemySpawner;
34.         }
35.
36.         public override void Update(GameTime gameTime)
37.         {
38.             for (int i = 0; i < bullets.Count; ++i)
39.             {
40.                 if (bullets[i].GetWorld().Translation.Z < shotMinZ)
41.                 {
42.                     var destroyBullet = bullets[i];
43.                     bullets.RemoveAt(i);
44.                     this.gameObject.RemoveComponent(destroyBullet);
45.                     i--;
46.                 }
47.                 else
48.                 {
49.                     for(int j =0; j< enemySpawner.enemies.Count(); j++)
50.                     {
51.                         var enemy = enemySpawner.enemies[j];
52.                         if (bullets[i].IsColiding(enemy.model.model, enemy.model.GetWor
    ld()))
53.                         {
54.                             var particle = new ParticleExplosion(graphicsDevice, enemy.
    model.GetWorld().Translation,
55.                                 enemySpawner.rnd.Next(enemy.particleExplosionSettings.m
    inLife, enemy.particleExplosionSettings.maxLife),
56.                                 enemySpawner.rnd.Next(enemy.particleExplosionSettings.m
    inRoundTime, enemy.particleExplosionSettings.maxRoundTime),
57.                                 enemySpawner.rnd.Next(enemy.particleExplosionSettings.m
    inParticlesPerRound, enemy.particleExplosionSettings.maxParticlesPerRound),
58.                                 enemySpawner.rnd.Next(enemy.particleExplosionSettings.m
    inParticles, enemy.particleExplosionSettings.maxParticles),
59.                                 enemy.explosionColorsTexture, enemy.particleSettings, e
    nemy.explosionEffect);
60.
61.                             this.enemySpawner.gameObject.AddComponent(particle);
62.
63.                             enemySpawner.enemies[j].Destroy();
64.                             var destroyBullet = bullets[i];
65.                             bullets.RemoveAt(i);
```

```
66.                             this.gameObject.RemoveComponent(destroyBullet);
67.                             i--;
68.
69.
70.                             break;
71.                         }
72.                     }
73.                 }
74.             }
75.
76.             if (shootCountdown <= 0)
77.             {
78.                 Shoot();
79.             }
80.             else
81.             {
82.                 shootCountdown -= gameTime.ElapsedGameTime.Milliseconds;
83.             }
84.
85.
86.         }
87.
88.         protected override void InitComponents()
89.         {
90.         }
91.
92.         private void Shoot()
93.         {
94.             if (Keyboard.GetState().IsKeyDown(Keys.Space) ||
95.                 Mouse.GetState().LeftButton == ButtonState.Pressed)
96.             {
97.                 var bullet = new Model3D(Game1.singleton.Content.Load<Model>(@"Models\l
    owpoly_max"), null, this.gameObject.transform.Position3D);
98.                 this.gameObject.AddComponent(bullet);
99.
100.                        bullet.SetPosition(cameraObject.camera.GameObject.transform.Posi
    tion3D + new Vector3(0, -5, 0));
101.                        bullet.direction = cameraObject.camera.Direction * shotSpeed;
102.
103.                        bullets.Add(bullet);
104.                        shootCountdown = shotDelay;
105.                    }
106.                }
107.            }
108.        }
```

Primjer koda 30 - Implementacija objekta Bullets

Kolekcija bullets sadrži sve metke koji su aktivni unutar scene. Vrijednost shotMinZ određuje maksimalnu udaljenost koju metak može postići prije nego se uništi sa scene. Vrijednost cameraObject sadrži referencu na kameru unutar scene. Vrijednost enemySpawner sadrži referencu na objekt koji stvara neprijatelje unutar scene. Ovu vrijednost koristimo da bismo provjerili je li metak pogodio nekoga od neprijatelja unutar scene. Vrijednost shotSpeed označava brzinu ispaljenog metka. Vrijednost shotDelay

označava razmak između ispaljenih metaka kako bi se izbjegao efekt lasera (uzastopnih ispaljivanja) i pada performansi igre. Vrijednost shootCountdown služi kao brojač koji kad udari vrijednost 0 dozvoljava da se novi metak ispali. U funkciji Update provjeravamo i uništavamo sve metke koji su prešli maksimalnu dopuštenu udaljenost te za sve preostale metke provjeravamo jesu li u koliziji s nekim od neprijatelja. U slučaju da jesu, aktiviramo sustav čestica na neprijatelju te uništavamo neprijatelja i metak unutar scene. U slučaju da je pritisnut razmak na tipkovnici ili lijevi klik miša, u funkciji Shoot se stvara novi metak kojem se definira početna pozicija i direkcija gibanja.

Kao i u 2D prototipu, i 3D prototip igre posjeduje upravitelja koji inicijalizira početne objekte unutar scene te vrši njihovu iteraciju. U primjeru koda prikazana je implementacija objekta GameStateManager.

```
1.  using _3DSolution;
2.  using _3DSolution.Scripts;
3.  using Kernel.Manager;
4.  using Microsoft.Xna.Framework;
5.  using Microsoft.Xna.Framework.Graphics;
6.  using System;
7.  using System.Collections.Generic;
8.
9.  namespace Solution.Objects
10. {
11.     public class GameStateManager : DrawableGameComponent
12.     {
13.         public GraphicsDevice graphicsDevice { get; set; }
14.
15.         public CameraObject camera { get; set; }
16.
17.         public EnemySpawner enemySpawner { get; set; }
18.
19.         public GameStateManager(GraphicsDevice graphicsDevice, Game game)
20.             : base(game)
21.         {
22.             this.graphicsDevice = graphicsDevice;
23.             camera = new CameraObject(graphicsDevice);
24.             enemySpawner = new EnemySpawner(camera.camera, graphicsDevice);
25.             new Bullets(enemySpawner, camera, graphicsDevice);
26.
27.         }
28.
29.         public void StartGame()
30.         {
31.         }
32.
33.         public void GameOver()
34.         {
35.             GameLoop.DestroyAll();
```

```
36.         }
37.
38.         public override void Update(GameTime gameTime)
39.         {
40.
41.             GameLoop.UpdateObjects(gameTime);
42.
43.             base.Update(gameTime);
44.         }
45.
46.         public override void Draw(GameTime gameTime)
47.         {
48.             Game1.singleton.spriteBatch.Begin();
49.             GameLoop.Draw(Game1.singleton.spriteBatch, camera.camera);
50.             base.Draw(gameTime);
51.             Game1.singleton.spriteBatch.End();
52.         }
53.
54.     }
55. }
```

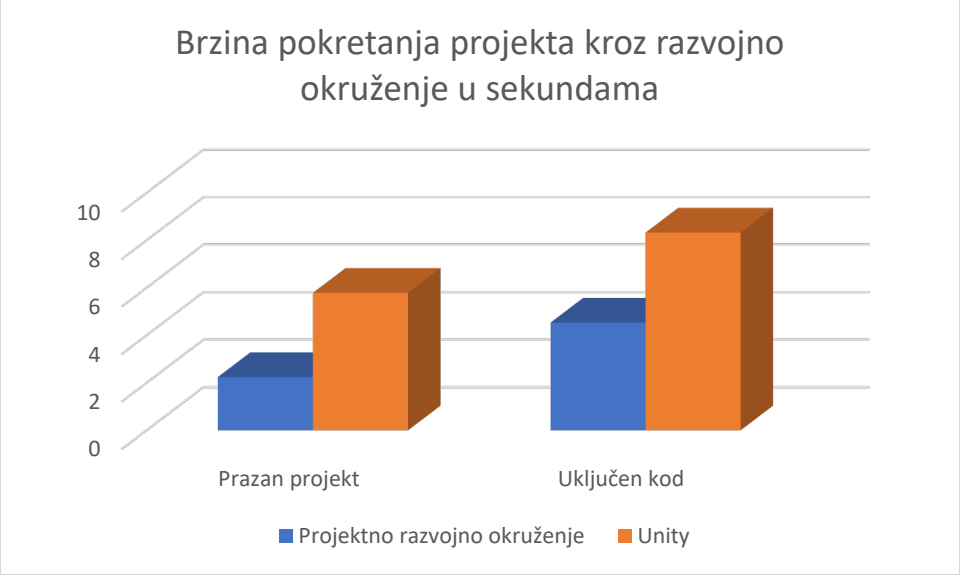
Primjer koda 30 - Implementacija objekta GameStateManager

Kao i u prošlom primjeru klase GameStateManager se u svojim funkcijama Update i Draw brine da se sve stvari unutar scene pravilno ažuriraju i iscrtavaju. U konstruktoru se inicijaliziraju početni objekti unutar scene te se referenca na kameru i grafički uređaj (vrijednosti camera i graphicDevice) koriste za iscrtavanje istih objekata unutar scene.

19. Konkurentnost projektnog rješenja na tržištu

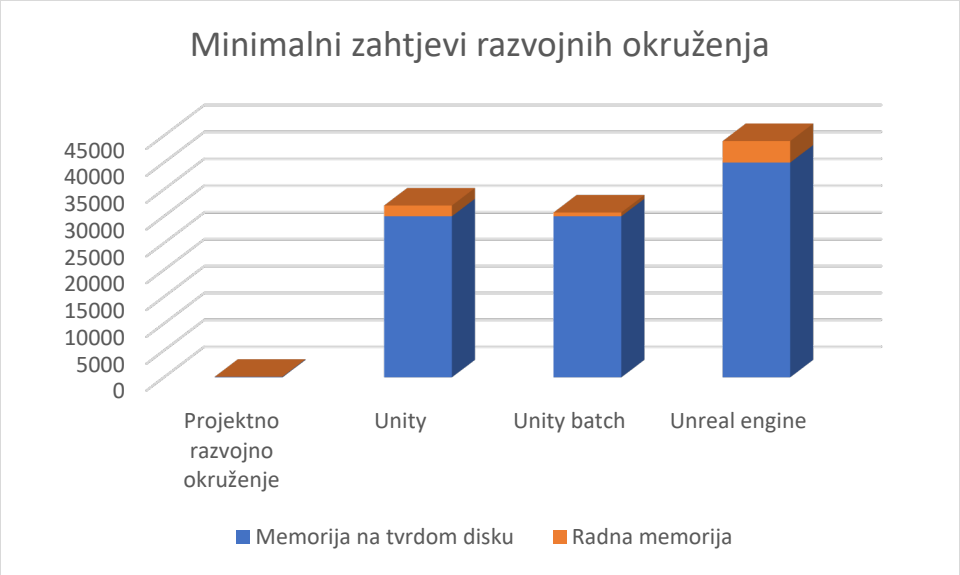
Iako po broju funkcionalnosti, dokumentaciji i dugoročnoj održivosti projektno rješenje teško može konkurirati bilo kojem komercijalno primjenjivom rješenju, postoji par stavki s kojima se projektno rješenje može natjecati. Prva i najjača stavka ovog rješenja su performanse prilikom razvoja igre. Budući da rješenja poput Unityja i Unreal Engina koriste korisničko sučelje te jako puno refleksije prilikom dohvaćanja metoda, kompilacijski dio projektnog razvojnog okruženja je osjetno brži od komercijalnih rješenja. Na slici 24. vidimo graf s ispisom brzine pokretanja projekta kroz projektno i Unity razvojno okruženje. Razlog ovome je to što sastavljač (engl. *compiler*) IL2CPP koji Unity koristi prevodi sav kod iz C# programskog jezika u C++ što je dosta skup i dugotrajan proces.

Isto tako Unity razvojno okruženje koristi jako puno refleksije kako bi dohvatilo skripte i metode sa svakog pojedinog objekta, stoga se često preporučuje da se broj klasa u Unityju drži na minimumu.



Slika 24. Usporedba brzine pokretanja projekta kroz razvojno okruženje u sekundama

Također, pošto je projektno razvojno okruženje praktički skup biblioteka te ne posjeduje nikakvo značajno korisničko sučelje, za pokretanje razvojnog okruženja je dovoljno 100 MB memorije na tvrdom disku i 2 MB radne memorije dovoljne za otvaranje programa za pisanje teksta (*notepad*). Prema informacijama koje se mogu pronaći na službenoj Unity *web*-stranici, za pokretanje Unity razvojnog okruženja potrebno je imati minimalno 30 GB memorije na tvrdom disku i 2GB radne memorije. Postoji i opcija da se Unity pokrene bez korisničkog sučelja što smanjuje korištenje radne memorije na 700 MB, ali u tom slučaju moguće je pokretati samo funkcije kao što su testovi te ne postoji direktan ulaz u računalnu igru. Na slici 25. prikazan je graf s usporedbom podataka minimalnih zahtjeva razvojnih okruženja.



Slika 25. Minimalni zahtjevi razvojnih okruženja

Kao što je već navedeno u prijašnjim poglavljima, projektno i Unity razvojno okruženje su strukturalno nalik jedno drugome; programer koji je prethodno koristi Unity ili projektno razvojno okruženje ne bi trebao imati nekih većih problema snalazeći se u drugom razvojnom okruženju. Prednost projektnog razvojnog okruženja je također što je otvorenog koda te ne zahtijeva licencu za korištenje dok razvoj računalne igre za prodaju u Unity razvojnom okruženju u vrijeme pisanja ovog rada košta 150 dolara mjesečno po osobi koja sudjeluje u razvoju.

Kako je već spomenuto u prijašnjim poglavljima, velika prednost projektnog razvojnog okruženja je to što ne ograničava korisnika da koristi specifični dizajn arhitekture igre već mu na izbor daje jedno od beskonačno mnogo mogućih rješenja. Unity striktno definira pravilo da svaka komponenta koja želi biti prisutna na sceni mora nasljeđivati komponentu MonoBehaviour. Budući da je MonoBehaviour skripta koja ima slabe performanse, programeri su prisiljeni osmišljavati načine kako balansirati performanse s lakoćom održavanja koda u budućnosti, odnosno manje MonoBehaviour skripti rezultirati će boljim performansama koda, ali će isto tako skripte postajati sve kompleksnije tako da će u jednom trenutku postati preteške za održavanje i pronalaženje problema. Postoje dizajni arhitekture koda kao što je Naredbeni dizajn koji može smanjiti kompleksnost

koda, a performanse ostaviti gotovo istima, ali onda se počinje javljati problem kutije (engl. *box*).

Manji broj funkcija projektnog razvojnog okruženja je i prednost i nedostatak. Nedostatak je zato što mali broj funkcionalnosti znači manji broj žanrova računalnih igara koji se mogu napraviti. Zbog manjeg broja funkcionalnosti svaki programer može relativno brzo savladati sve komponente razvojnog okruženja te shvatiti kako ga maksimalno iskoristiti za određeni žanr igre. Većina razvojnih okruženja je nastala kao produkt izrade računalne igre stoga se dosta razvojnih okruženja ističe u posebnom žanru igara.

20. Zaključak

U svrhu diplomskog rada napravljen je projektni zadatak kako bi se čitatelju pokazala tehnika i način kreiranja razvojnog okruženja za izgradnju računalnih igara koristeći se programskim jezikom C# i bibliotekom MonoGame. Budući da MonoGame posjeduje veoma lak i razumljiv skup funkcija, programeri se mogu više fokusirati na arhitekturu i dizajn igre nego što bi se inače trebali baviti raznim drugim komponentama poput grafičkih jezika, alokacije memorije i slično. Iako je C# kao jezik dosta sporiji po performansama od jezika poput C++,C i Pythona, prednosti C# su jednostavnost, čistač memorije i runtime generički pozivi.

Na samom početku rada definirana je ideja dizajna da pisanje koda unutar projektnog razvojnog okruženja što više naliči pisanju koda u Unity razvojnom okruženju. Također, programer s osnovnim znanjem programiranja računalnih igara u C# programskom jeziku može koristiti ovaj projekt kao uvodnu lekciju za kreiranje svog vlastitog razvojnog okruženja. Razlog je to što projektno razvojno okruženje predstavlja dobre prakse i tehnike korištene za kreiranje razvojnog okruženja, ali isto tako ne zabranjuje proširenje, modifikacije i primjene vlastitog dizajna prilikom razvoja.

Diplomski rad pokriva veliku količinu područja unutar razvoja računalnih igara. U prvom dijelu rada predstavljen je razvoj okruženja za dvodimenzionalne računalne igre. S komponentama poput SpriteRenderer i SpriteAnimation prikazan je način kako se slike mogu iscrtavati i animirati unutar scene. Također, s komponentama poput Collider i BoxCollider2D prikazana je osnovna implementacija kolizije među objektima unutar scene te prepoznavanje potrebe za apstrakcijom. Komponenta AudioSource korištena je za stvaranje i puštanje zvuka unutar scene koja je za cilj imala omogućiti krajnjem igraču zvučni odgovor na njegove akcije. Kod trodimenzionalnog razvoja komponenta Camera ima zadaću stvoriti matricu prostora promatranja, a komponenta Model3D služi za iscrtavanja trodimenzionalnih modela unutar scene. Na samom kraju opisan je sustav čestica koji ima zadaću stvoriti ogroman broj slika u malom vremenu te paziti da pritom ne ošteti performanse igre. Implementacija svih komponenti opisana je na dva prototipa pod nazivom „Uhvati loptu“ i „Pogodi kraljicu“.

Najveći problemi s kojima se današnji programeri susreću je neznanje o procesima koji se obavljaju u pozadini razvojnih okruženja što često dovodi do neželjenih greški i sporog razvoja računalne igre. Stoga velik broj programera koji se bavi razvojem računalnih igara pokušava izraditi svoju inačicu razvojnog okruženja kako bi utvrdili svoje znanje te se upoznali s procesima koji se događaju u pozadini.

Iako danas sva okruženja izrađena od jednog programera teško mogu konkurirati na tržištu razvojnih okruženja za računalne igre, svakim novim okruženjem se akumulira sve više i više znanja o radu i izradi razvojnih okruženja što u konačnici pomaže novim programerima uhvatiti korak s konkurencijom te pomažu novim komercijalnim projektima da uhvate korak s konkurencijom u industriji.

Stoga iako projektno razvojno okruženje nikad neće postati komercijalno prihvatljivo rješenje, niti je ikad imalo u cilju to postati, znanje koje ono donosi poslužit će budućim generacijama programera kao temelj u razumijevanju i shvaćanju osnovnih komponenti razvojnih okruženja te će kao takvo u budućnosti pomoći pri stvaranju novih i boljih komercijalno prihvatljivih razvojnih okruženja.

Literatura

1. Aaron Reed – Learning XNA 4.0 (2010), < raspoloživo na: https://books.google.hr/books/about/Learning_XNA_4_0.html?id=HtF2n8aZlSYC&printsec=frontcover&source=kp_read_button&redir_esc=y#v=onepage&q&f=false

2. Esoterico Software- What is Spine? (7. 7. 2020.) < raspoloživo na http://en.esotericsoftware.com/spine-in-depth#:~:text=Animation%20in%20Spine%20is%20done,frame%2Dby%2Dframe%20animation%3A&text=Attachments%20Images%20attached%20to%20bones,with%20different%20items%20and%20effects>

3. Poligonska mreža (15.8.2020.) <raspoloživo na: https://en.wikipedia.org/wiki/Polygon_mesh.>

4. Quake (17.8.2020.) <raspoloživo na: https://store.steampowered.com/app/2310/QUAKE/>

5. Jason Gergory, Arhitektura videoigara, Treće izdanje <raspoloživo na https://www.amazon.com/Engine-Architecture-Third-Jason-Gregory/dp/1138035459/ref=rtpb_3/146-5969764-5779020?_encoding=UTF8&pd_rd_i=1138035459&pd_rd_r=10aaae69-01f3-4393-86df-94760b40f44e&pd_rd_w=MyDqj&pd_rd_wg=eJ7vS&pf_rd_p=1060fc32-cc06-48f1-987d-6b74a57cd8f2&pf_rd_r=XZ4Q99D3VZ09QQJ9GW3Q&psc=1&refRID=XZ4Q99D3VZ09QQJ9GW3Q >

6. MonoGame službena stranica<raspoloživo na https://www.monogame.net/>

7. Xna Game Studio 4.0 Refresh <raspoloživo na https://docs.microsoft.com/en-us/previous-versions/windows/xna/bb200104(v=xnagamestudio.41)?redirectedfrom=MSDN>

8. MonoGame GitHub repozitorije <raspoloživo na: https://github.com/MonoGame/MonoGame>

9. Sistemski zahtjevi potrebni za pokretanje Unity razvojnog okruženja (6. 9. 2020.) <raspoloživo na: https://docs.unity3d.com/2019.1/Documentation/Manual/system-requirements.html>

10. Sistemski zahtjevi potrebni za pokretanje Unreal razvojnog okruženja (6. 9. 2020.) <raspoloživo na

Popis slika

Popis programskih kodova