Izrada platforme za izradu video igara

Vilagoš, Leon

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni preddiplomski studij Matematika i računarstvo

Izrada platforme za izradu video igara

Završni rad

Mentor:

Domagoj Ševerdija

Kandidat:

Leon Vilagoš

Osijek, 2022

Sadržaj

1 | Uvod

Video igrice danas su jedan od najrasprostranjenijih načina zabave u svijetu. U suštini video igrice nisu ništa drugo osim interaktivne aplikacije. Za izradu takvih aplikacija koriste se specijalizirani skupovi alata. Takav skup alata koji je specijaliziran za izradu igrica građen je kao platforma za izradu video igrica [1]. Kompleksnost i cijena izrađivanja video igrica raste svake godine. Stoga postoji konstanta potreba za sve moćnijim i kompleksnijim platformama za izradu video igrica. Danas je standard da igrice imaju brzo i efikasno prikazivanja i 2D i 3D grafike što uključuje teksture, simulaciju svijetlosti, kaustike, simulaciju različitih materijala i efekata itd. Također video igrice moraju imati: zvuk, simulaciju fizike, efikasnost pri procesiranju ogromnih količina podataka, stabilan rad na različitima platformama i još mnogo zahtjeva koji nisu trivijalni za implementirati. Kako kompleksnosti svih ovih zahtjeva raste tako naravno raste i kompleksnost platformu za izradu video igrica.

Jedno pitanje koje ima smisla postaviti jest ako su video igrice interaktivne aplikacije koja svaka ima svoje jedinstvene zahtjeve, zašto uopće praviti platformu za izradu igrica ako možemo jednostavno napraviti igricu i odmah ukomponirati sve te funkcionalnosti. Uzmimo za primjer da igrica treba imati sunce u sceni. Osoba koja pravi igricu u platformi za izradu igrice će uzeti sunce kakvo je implementirano u platformi, promijeniti par parametara preko korisničkog sučelja i to je sve. Dok je problem samog implementiranja izvora svijetla puno složeniji jer zahtjeva poznavanje matematike i općenito teorije o računalnoj grafici. Tj. osoba koja pravi funkcionalnosti potrebne za igricu i osoba koja pravi igricu imaju veoma drugačija područja stručnosti [5]. Također iako svaka igrica ima svoje jedinstvene zahtjeve, mnogo tih zahtjeva se preklapaju, stoga ima smisla imati platformu koja može brzo i efikasno proizvoditi igrice. Jedna od glavnih uloga platforme za izradu igrica je prevođenje podataka u oblik koji je čovjeku razumljiv. Na primjer kako bi ljudi koji prave igrice mogli bez problema raditi sa 3D objektima, platforma za izradu igara mora: znati prikazati taj 3D objekt na ekranu, znati nacrtati ga u određenoj poziciji u svijetu, moći dodijeliti mu odgovarajući materijal i obojati ga odgovarajućom teksturom. Sve su to problemi koje osoba koja pravi platformu za izradu igrice mora riješiti.

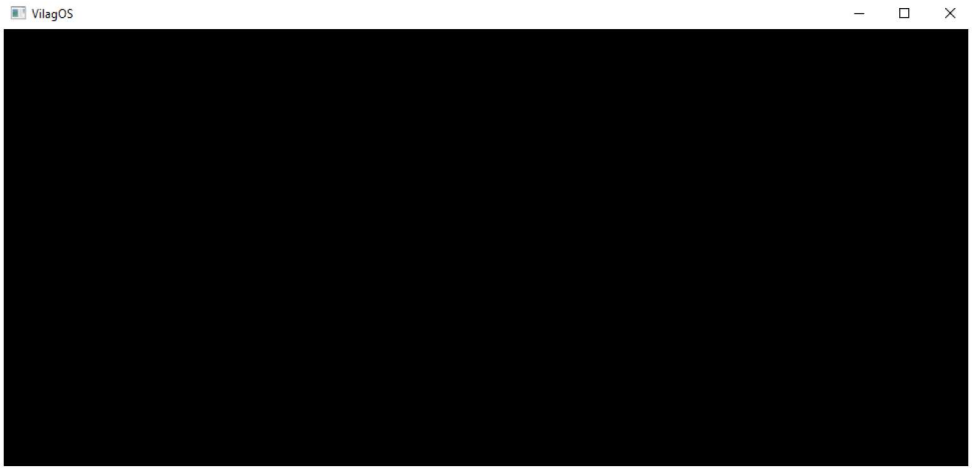
U ovom radu pokazat ćemo izradu jednostavne platforme za izradu 2D video igrica.

2 | Struktura projekta

Kako bi se nešto moglo zvati platformom za izradu video igrica, to nešto mora omogućavati korištenje svih osnovnih funkcionalnosti bez kojih igrica jednostavno nije igrica. Te funkcionalnosti su: prepoznavanje unosa s miša i tipkovnice, sustav događaja, bilježenje unosa, sustav slojeva, crtač, kamera, sjenčari i transformacija objekata na sceni. Također trebao bi imati funkcionalnosti koji nisu i vezanu uz samu igrice, poput vođenja dnevnika ili radnjama nad prozorom aplikacije (pomicanje, približavanje, smanjivanje i slično) [4]. Već ovo nije lagan posao, ima dobar broj funkcionalnosti koje sadržavaju svoj niz zahtjeva, problema i funkcionalnosti koje treba riješiti. Stoga prođimo kroz svaku funkcionalnost jedna po jednu.

2.1 Prozor

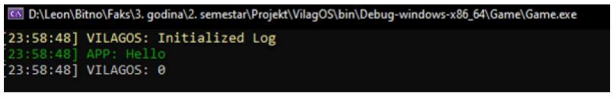
Prva i najosnovnija stvar koju bi imalo smisla implementirati jest prozor. Kod igrica je najčešće slučaj da dok živi prozor živi i sama igrica, tj. kada se prozor zatvara, zatvara se i igrica. Kakav će biti taj prozor ovisiti će o platformu za koju će igrica namijenjena. Na slici 2.1 prikazan je primjer Windows prozora.



Slika 2.1: Prozor aplikacije

2.2 Dnevnik

Funkcionalnost koja je jako korisna i za izradu platforme i za izradu samih igrica jest dnevnik. Prilikom izrade same platforme on će služiti kao pomoć pri otklanjanju pogreška i testiranju, a pri izradi igrice korisnik ga može koristiti kako god želi, bitno je samo da dnevnik postoji. Stoga bi bilo bi dobro da se ispisi u dnevnik pri izradi platforme i ispisi u dnevnik koje će korisnik koristiti pri izradi igre razlikuju. Na slici 2.2 prikazan je primjer Windows konzole u kojoj su ispisane poruke dnenvika.



Slika 2.2: U dnevniku ispisane poruke sa strane aplikacije i platforme

2.3 Sustav događaja

Kako je implementiran dnevnik ispisa implementacija sustava događaja bit će puno jednostavnije, jer je moguće jednostavno ispisati u konzolu na primjer je li se događaj dogodio ili ne. Ideja sustava događaja je da aplikacija ima tzv. dispečer događaja. Dispečer će biti zadužen za to da za svaki događaj koji dobije pozove odgovarajuću funkciju za taj događaj. Dok će sami događaji biti vezani za dio aplikacije na koji se događaj odnosi, funkcije koje dispečer poziva za neki događaj bit će vezane za aplikaciju. U prozor klasi definiramo događaj, a u klasi aplikacije definiramo funkciju za taj događaj i povežemo tu funkciju i događaj pomoću dispečera.

Na primjer treba razvući prozor. U klasi prozora definiramo događaj, a u klasi aplikacije definiramo funkciju koja će zapravo promijeniti veličinu prozora. Ovo je lako jer će prozor biti vezan za aplikaciju. Analogno definiramo događaje za pritisak na tipkovnici i/ili mišu i sve ostalo što bi nam trebalo.

2.4 Ispit unosa

Sustav događaja nam omogućava da znamo kada se neki događaj dogodio i onda pozovemo odgovarajuću funkciju za taj događaj. Iako ono što je poželjno imati je sustav koji će u svakom trenutku izvršavanja aplikacije provjeravati je li dan neki unos. To omogućava da se na primjer drži pritisnuta tipka ALT i na pokretanje miša se upravlja okretanje kamere, dok inače okretanje miša okreće igrača na primjer.

Za ljude koji rade igrice vrlo efikasan način za implementirati bilo kakve kontrole jest preko ispita unosa. Na primjer u svakom trenutku provjeravaju je li gumb

za kretanje stisnut, ako jest onda pomakni igrača, puno jednostavnije nego da se korisnik mora patiti sa sustavom događaja za nešto ovako jednostavno.

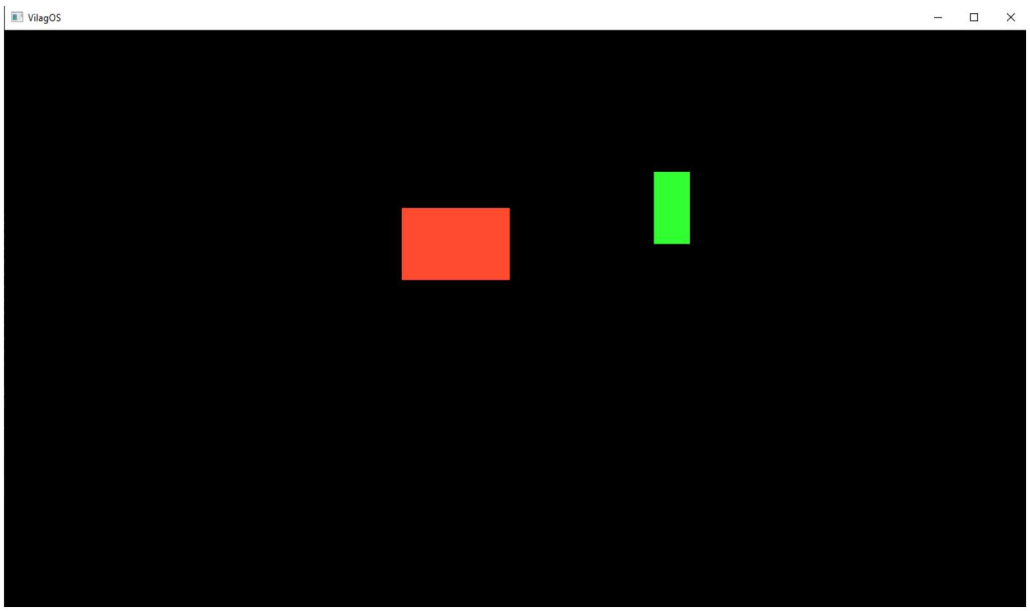
2.5 Slojevi

Strukturu slojeva treba zamisliti kao slojeve iz PhotoShopa. Ova struktura jako je korisna za dvije stvari. Prva je redoslijed iscrtavanja. Ono što je na najgornjem sloju će se iscrtavati iznad svega što je na slojevima ispod. Druga vrlo bitna stvar jest to što isto kao i kod prozora, svaki sloj može imati svoje događaje. Što znači da svaki sloj ima zasebnu događaj za pritisak miša ili tipkovnice i slično.

Primjer koji jako dobro demonstrira ove dvije ključne značajke jest korisničko sučelje u igrici. Korisničko sučelje bi se uvijek trebalo crtati zadnje jer mora biti preko svega ostaloga, stoga će svijet igrice biti na sloju ispod sloja korisničkog sučelja. Također ako i korisničko sučelje i igrica imaju događaj klika miša svaki sloj može imati svoju funkciju za taj događaj. Vrlo lako se napravi da ako je sloj korisničkog sučelja odradio događaj da tada sloj igrice ignorira taj događaj.

2.6 Prikazivač

Trenutno aplikacije kakve bi stvorila ovakva platforma ne mogu zapravo imati išta nacrtano na njima, što je očito veliki problem. Za prikazivanje grafike može se koristiti proizvoljno sučelje za programiranje aplikacija koje služi za prikazivanje grafike. Prikazivač će zapravo biti apstrakcija naredbi korištenog sučelja za programiranje aplikacija koju su potrebne za crtanje objekata na ekran. Slika 2.3 prikazuje prozor s nacrtanim objektima od prikazivača.

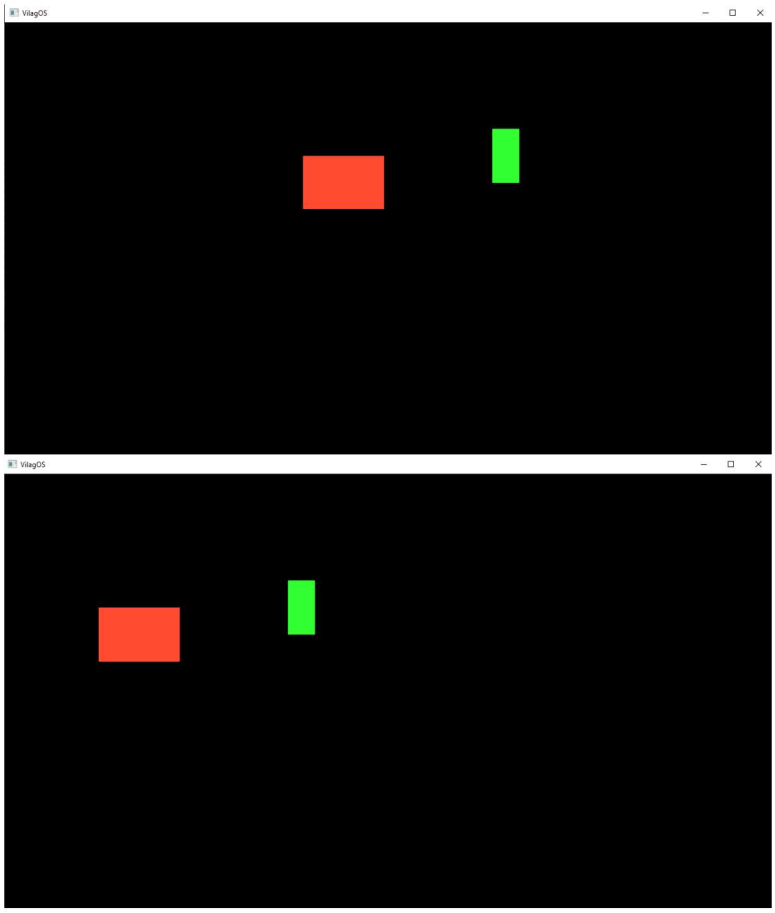


Slika 2.3: Prikazani objekti na prozoru

2.7 Kamera

Ovakva platforma sad već ima niz osnovnih funkcionalnosti, sada kada može zapravo nešto i prikazati ima smisla razmišljati o kameri. Kamera je pogled u svijet igrice i bitno je da ima mnogo opcija poput omjera stranica, zum, rotacije i pozicije i slično.

Pri implementiranju kamere treba je zamisliti više kao ideju ili konstrukciju nego objekt na sceni. Ideja je kamere da je vidljiv samo određeni dio svijeta. Dakle htjet ćemo iscrtavati objekte samo one na koje kamera gleda, ali pošto kamera ne postoji kao sami objekt, kretanje kamere će morati biti simulirano. To će raditi tako da ako se kamera pomjeri gore za 2 jedinice prostora, zapravo će se cijeli svijet pomaknuti dolje za dvije jedinice prostora kako bi taj dio koji kamera vidi došao na prozor [2]. Slika 2.4 prikazuje prozor prije i poslije pomicanja kamere.



Slika 2.4: Prozor prije i poslije pomicanje kamere udesno

2.8 Transformacija

Trenutno kada treba nacrtati objekt potrebno mu je dati točke koje čine taj objekt i prikazivač ga nacrta. Ono što je poželjno da se taj objekt može nacrtati bilo gdje

u prostoru, bilo koje veličine i s bilo kakvom rotacijom.

To se postiže matricom transformacije. Objekt se i dalje crta preko točaka koje čine taj objekt, ali onda na svaku se točku primjeni matrica transformacije koja će skalirati, rotirati i pomaknuti tu točku u prostoru [3, str. 135].

2.9 Sijenčar

Sijenčar će u suštini biti apstrakcija naredbi sučelja za prikazivanje grafike koje služe za bojanje objekata. Omogućava bojanje objekata teksturama i bojama što uključuje i interpolaciju boja.

S ovime platforma ima dovoljno funkcionalnosti da se naprave neke jednostavne igrice. Pokazat ćemo na kraju dva primjera koja su u potpunosti napravljena koristeći ovakvu platformu. Slika 2.5 prikazuje Windows prozor s objektima nacrtanim prikazivačem i obojani teksturama pomoću sijenčara.

Slika 2.5: Objekti s teksturama

3 | Implementacija funkcionalnosti

Prije bilo čega drugoga treba smisliti kako će ovaj projekt izgledati. Iako platforme za izradu igrica imaju grafičko sučelje, ono služi za lakše korištenje funkcionalnosti koje sadrži platforma pa se neće pričati o tome već o samim funkcionalnostima. Stoga platforma za izradu igrica se može implementirati kao statična biblioteka koju će koristiti aplikacija (tj. igrica). Radi jednostavnosti pričat ćemo o platformi za izradu igrica samo za platformu Windows. To znači da će igrice koju stvara platforma biti Windows aplikacija.

3.1 Aplikacija

Aplikacija je sama igrica kakvu stvara platforma. Glavna stvar za imati na umu pri implementaciji jest da aplikacija mora moći koristiti sve funkcionalnosti koje pruža platforma. Stoga ima smisla da će aplikacija kao objekt enkapsulirati te funkcionalnosti u sebi.

Aplikacija treba enkapsulirati instance prozora, prikazivača i vektora slojeva, pošto će sve to biti jedinstveno vezano za aplikaciju.

Implementacija klase za aplikaciju:

```
1  class Application
2  {
3  public:
4    Application();
5    virtual ~Application();
6    void run();
7    void OnEvent(Event &e);
8    void PushLayer(Layer *Layer);
9    void PushOverlay(Layer *Layer);
10
11   inline WindowMaster &GetWindow() { return *m_Window; };
12   inline static Application &GetApp() { return *s_Instance; };
13
14 private:
15   bool OnWindowClose(WindowCloseEvent &event);
16   bool OnWindowResize(WindowResizeEvent &event);
17   static Application *s_Instance;
18
19   bool m_Running = true;
```

```
20   bool m_Minimized = false;
21
22   std::unique_ptr<WindowMaster> m_Window;
23
24   LayerStack m_LayerStack;
25   DeltaTime m_TimeOfLastFrame = 0.0f;
26 };
```

Funkcije:

- **Application - konstruktor:** stvara instancu prozora aplikacije i početni sloj pošto igrica mora imati barem jedan sloj da bi radila te inicijalizira prikazivač.
- **Run:** tu se nalazi glavna while petlja za izvođenje aplikacije. Za svaki prolazak petlje prolazi kroz sve slojeve i prozor da ih ažurira. Također mjeri koliko vremena je potrebno za jedan prolazak petlje što daje podatak o tome koliko je vremena trebalo da se izvršni jedan prolazak petlje tj. koliko milisekundi je trajala jedna sličica aplikacije.
- **PushLayer:** stvara novi sloj i dodaje ga u vektora slojeva kao najniži sloj tj. onaj koji će iscrtavati prvi
- **PushOverlay:** dodaje sloj u vektor slojeva koji će biti na samom vrhu.
- **OnEvent:** prima neki događaj i pomoću dispečera događaja svakom događaju dodjeljuje odgovarajuću funkciju odaziva.
- **OnWindowClose:** funkcija odaziva za događaj zatvaranja prozora.
- **OnWindowResize:** funkcija odaziva za događaj promjene veličine prozora.

3.2 Prozor

Windows prozor može se jednostavno stvoriti s proizvoljnim sučeljem za programiranje aplikacija, kako god dobro je imati zaseban objekt tipa prozora koji će enkapsulirati i razne potrebne podatke i funkcije za taj prozor. Većina tih funkcija bit će apstrakcija funkcija sučelja za programiranje aplikacija, ali poanta je da korisnik koji pravi igricu ne mora znati ništa o sučelju koje se koristi već koristi funkcije same platforme.

Implementacija klase za prozor:

```
1 class Window
2 {
3 public:
4   using EventCallbackFn = std::function<void(Event &)>;
5   Window(const WindowProps &props);
6   virtual ~Window();
7
8   void OnUpdate() override;
```

```
9
10    inline unsigned int GetWidth() const override { return m_Data.
       Width; };
11    inline unsigned int GetHeight() const override { return m_Data.
       Height; };
12
13    inline void SetEventCallback(const EventCallbackFn &callback)
       override
14    {
15      m_Data.EventCallback = callback;
16    }
17    void SetVSync(bool enabled) override;
18    bool IsVSync() const override;
19
20  private:
21    void Init(const WindowProps &props);
22    void Shutdown();
23
24    GLFWwindow *m_Window;
25
26    struct WindowData
27    {
28      std::string Title;
29      unsigned int Width, Height;
30      bool VSync;
31      EventCallbackFn EventCallback;
32    };
33
34    WindowData m_Data;
35  };
```

Funkcije:

- **Window - konstruktor:** poziva Init funkciju prozora.
- **OnUpdate:** koristi funkcije sučelja za prozor koje se trebaju izvršavati svaki prolazak while petlje aplikacije (tj. svaku sličicu).
- **GetWidth i GetHeight:** vraćaju širinu i visinu prozora.
- **SetVSync i IsVSync:** postavlja i provjerava je li uključena vertikalna sinkronizacija.
- **GetNativeWindow:** vraća instancu ovog prozora.
- **Init:** postavi sve potrebne podatke na početne vrijednosti (koje bira osoba koja implementira platformu) i definira odaziv funkcije za događaje.
- **ShutDown:** zatvara aplikaciju i briše iz memorije sve što treba.

3.3 Dnevnik

Za dnevnik najlakše je koristiti bilo koje sučelje za programiranje aplikacije za dnevnik. Jedini problemi koje ovdje treba riješiti su apstrakcija naredbi sučelja

koje se koristi i napraviti posebne instance dnevnika. Jedna koja će služiti kao dnevnik za izradu same platforme i druga koju će koristit korisnik pri izradi igrice.

Implementacija klase za dnevnik:

```
class Log
{
public:
  static void Init();

  inline static std::shared_ptr<spdlog::logger> &GetCoreLogger() {
   return s_CoreLogger; }
  inline static std::shared_ptr<spdlog::logger> &GetClientLogger()
   { return s_ClientLogger; }

private:
  static std::shared_ptr<spdlog::logger> s_CoreLogger;
  static std::shared_ptr<spdlog::logger> s_ClientLogger;
};
```

Funkcije:

- **GetCoreLogger:** vraća dnevnik namijenjenu za onoga tko razvija platformu.
- **GetClientLogger:** vraća dnevnik namijenjenu za onoga tko razvija igricu.

3.4 Događaji

Ideja je stvoriti tip objekta događaj koji će imati svoj tip, ime i kategorije. Slojevi, prozori i sama aplikacija će definirati instance svojih događaja. Zatim potreban je tip objekta dispečer događaja koji će biti zadužen da za svaki događaj pridruži odgovarajuću funkciju odaziva. Razlikovanje događaja po tipu i kategoriji je korisno zato što različiti tipovi događaja (zatvaranje prozora, pritisak tipke miša, pritisak tipkovnice itd.) i različite kategorije događaja (aplikacija, miš, tipkovnica itd.) trebaju enkapsulirati različite podatke.

Implementacija klase za događaje:

```
class VOS_API Event
{
  friend class EventDispatcher;

public:
  virtual EventType GetEventType() const = 0;
  virtual const char *GetName() const = 0;
  virtual int GetCategoryFlags() const = 0;
  virtual std::string ToPrint() const { return GetName(); }

  inline bool IsInCategory(EventCategory category)
  {
```

```
13      return GetCategoryFlags() & category;
14    }
15
16    bool m_Handled = false;
17  };
```

Funkcije:

- **GetEventType:** vraća tip događaja.
- **GetName:** vraća ime događaja.
- **GetCategoryFlags:** vraća kategorije događaja.
- **ToPrint:** ispisuje ime događaja u konzolu.
- **IsInCategory:** za proslijeđenu kategoriju provjerava je li događaj u toj kategoriji.

Implementacija klase za dispečer događaja:

```
1  class EventDispatcher
2  {
3      template <typename T>
4      using EventFn = std::function<bool(T &)>;
5
6  public:
7      EventDispatcher(Event &event) : m_Event(event) {}
8
9      template <typename T>
10     bool Dispatch(EventFn<T> func)
11     {
12         if (m_Event.GetEventType() == T::GetStaticType())
13         {
14             m_Event.m_Handled = func(*(T *)&m_Event);
15             return true;
16         }
17         return false;
18     }
19
20 private:
21     Event &m_Event;
22 };
```

Funkcije:

- **Dispatch:** prima bool funkciju. Ako primljena funkcija true tada ne radi ništa zato što to znači da je ovaj događaj već odrađen na nekom sloju koji je iznad trenutnog. Ako je primljena funkcija false onda se poziva funkcija odaziva za taj događaj na trenutnom sloju. Više o slojevima u 3.6

3.5 Ispit unosa

Za ispit unosa najlakše je koristiti bilo koje sučelje za programiranje aplikacije za ispit unosa. Najčešće sučelja za prozore već imaju svoje funkcionalnosti za ispit unosa. Stoga implementacija ispita unosa se svodi na apstrakciju funkcija ispita unosa sučelja koje se koristi. Ovo je bitno zato što korisnik koji pravi igricu ne mora znati ništa o sučelju koje se koristi već koristi funkcije same platforme.

Implementacija klase za ispit unosa:

```
class Input {
public:
  inline static bool IsKeyPressedStatic(int keycode) { return
   s_Instance->IsKeyPressed(keycode); };
  inline static bool IsMouseButtonPressedStatic(int button) {
   return s_Instance->IsKeyPressed(button); };
  inline static std::pair<float, float> GetMousePositionStatic() {
   return s_Instance->GetMousePosition(); };
protected:
  virtual bool IsKeyPressed(int keycode) = 0;
  virtual bool IsMouseButtonPressed(int button) = 0;
  virtual std::pair<float, float> GetMousePosition() = 0;
private:
  static Input* s_Instance;
};
```

Funkcije:

- **IsKeyPressed:** vraća je li tipka tipkovnice pritisnuta.
- **IsMouseButtonPressed:** vraća je li tipka miša pritisnuta.
- **GetMousePosition:** vraća poziciju miša kao uređeni par x i y koordinate prozora.

3.6 Slojevi

Sloj će biti glavni gradivni dio aplikacije. Što se tiče implementacije treba zamisliti kao da se cijela aplikacija dijeli na dijelove. Svaki dio aplikacije ima dio koji se izvodi svaku sličicu (tj. svaki prolazak while petlje aplikacije)

Implementacija klase za sloj:

```
class Layer
{
public:
  Layer(const std::string &name = "Layer");
  virtual ~Layer();

```

```
7   virtual void OnAttach() {}
8   virtual void OnDetach() {}
9   virtual void OnUpdate(DeltaTime DeltaTime) {}
10  virtual void OnUIRender() {}
11  virtual void OnEvent(Event &e) {}
12
13  inline const std::string &GetName() const { return m_DebugName; }
14
15 protected:
16  std::string m_DebugName;
17 };
```

Funkcije:

- **Layer - konstruktor:** konstruktor koji samo postavlja ime sloja.
- **OnAttach:** poziva se kada se sloj stavi u vektor slojeva.
- **OnDetach:** poziva se kada se sloj skine s vektora slojeva.
- **OnUpdate:** poziva se na svakom na svakom prolasku while petlje aplikacije.
- **OnUIRender:** funkcija za crtanje korisničkog sučelja ovog sloja.
- **OnEvent:** koristi dispečer događaja da pozove odgovarajuće funkcije odaziva za proslijeđeni događaj.

3.7 Prikazivač

U suštini implementacija prikazivača se svodi na apstrakciju funkcija sučelja za programiranje aplikacija koje služi za prikazivanje grafike. Potrebno je pomoću funkcija sučelja napisati funkcije koje će biti od platforme i pomoću kojih se može lako obavljati operacije za prikaz grafike na prozoru.

Implementacija klase za prikazivač:

```
1 class Renderer
2 {
3 public:
4   Renderer();
5   ~Renderer();
6
7   void static Init();
8   void static OnWindowResize(uint32_t width, uint32_t height);
9
10  void static BeginScene(OrthographicCamera &Camera);
11  void static EndScene();
12
13  void static SubmitData(const std::shared_ptr<Shader> &shader,
     const std::shared_ptr<VertexArray> &vertexArray, const glm::mat4
      &transform = glm::mat4(1.0f));
```

```
14
15 private:
16   struct SceneData
17   {
18     glm::mat4 m_ViewProjectionMatrix;
19   };
20   static SceneData *m_SceneData;
21 };
```

Funkcije:

- **OnWindowResize:** funkcija odaziva za događaj promjene veličine prozora. Koristi funkcije sučelja za prozor.
- **BeginScene:** inicijalizira kameru i postavlja sve potrebne parametre kamere.
- **EndScene:** sadrži funkcije sučelja za prikazivanje grafike.
- **SubmitData:** prima podatke o točkama objekata i indeksima točaka i pomoći funkcija sučelja za prikazivanje grafike prikazuje sve objekte na prozor.

Implementacija klase za crtanje objekata:

```
1 class Renderer2D
2 {
3 public:
4   static void Init();
5   static void Shutdown();
6
7   static void BeginScene(const OrthographicCamera &camera);
8   static void EndScene(){};
9   static void DrawQuad(const vec2 &position, vec2 size, const vec4
     &color);
10  static void DrawQuad(const vec3 &position, vec2 size, const vec4
     &color);
11  static void DrawQuad(const vec2 &position, vec2 size, const std::
     shared_ptr<Texture2D> &texture, float tiling = 1.0f);
12  static void DrawQuad(const vec3 &position, vec2 size, const std::
     shared_ptr<Texture2D> &texture, float tiling = 1.0f);
13
14  static void DrawRotatedQuad(const vec2 &position, vec2 size,
     float rotation, const vec4 &color);
15  static void DrawRotatedQuad(const vec3 &position, vec2 size,
     float rotation, const vec4 &color);
16  static void DrawRotatedQuad(const vec2 &position, vec2 size,
     float rotation, const std::shared_ptr<Texture2D> &texture, float
      tiling = 1.0f);
17  static void DrawRotatedQuad(const vec3 &position, vec2 size,
     float rotation, const std::shared_ptr<Texture2D> &texture, float
      tiling = 1.0f);
18  static void DrawRotatedQuadZ(const vec3 &position, vec2 size,
     float rotation, const std::shared_ptr<Texture2D> &texture, float
      tiling = 1.0f);
19 };
```

Ova klasa sadrži funkcije koje su apstrakcija procesa crtanja objekata pomoću korištenog sučelja za prikazivanje grafike. Što znači da korisnik koji pravi igricu mora samo pozvati jednu od ovih funkcija da nešto nacrta, što je inače dosta kompliciraniji proces.

3.8 Kamera

Implementacija kamere svodi se na implementiranje matematičkih funkcija za transformaciju objekata u prostoru. Za to se koriste matrice te je najjednostavnije koristiti sučelja za programiranje aplikacija koje ima već definiran rad s matricama. Potrebno je definirati matrice pomaka, skaliranja i rotacije te neke dodatne vrijednosti koje su bitne za rad s kamerom pri pravljenju igrice.

Implementacija klase za kameru:

```
1  class OrthographicCamera
2  {
3  public:
4    OrthographicCamera() {}
5    OrthographicCamera(float left, float right, float bottom, float
       top);
6
7    void SetProjection(float left, float right, float bottom, float
       top);
8
9    glm::vec3 GetPosition() { return m_Position; }
10   inline void SetPosition(const glm::vec3 &position)
11   {
12     m_Position = position;
13     RecalculateViewMatrix();
14   }
15
16   inline void SetRotation(float rotation)
17   {
18     m_Rotation = rotation;
19     RecalculateViewMatrix();
20   }
21
22   const inline glm::mat4 GetProjectionMatrix() const { return
       m_ProjectionMatrix; }
23   const inline glm::mat4 GetViewMatrix() const { return
       m_ViewMatrix; }
24   const inline glm::mat4 GetViewProjectionMatrix() const { return
       m_ViewProjectionMatrix; }
25   const inline float GetRotation() const { return m_Rotation; }
26
27 private:
28   void RecalculateViewMatrix();
29   glm::mat4 m_ProjectionMatrix;
30   glm::mat4 m_ViewMatrix;
31   glm::mat4 m_ViewProjectionMatrix;
32   float m_Rotation = 0.0f;
```

```
33
34   glm::vec3 m_Position = glm::vec3(0.0f, 0.0f, 0.0f);
35 };
```

Funkcije:

- **OrthographicCamera:** računa matricu pogledprojekcije.
- **RecalcalculateViewMatrix:** računa matricu pogledprojekcije.
- **SetProjectionMatrix:**
- **SetPosition:** mijenja poziciju kamere na proslijeđenu i ponovno računa matricu pogledprojekcije.
- **GetPosition:** vraća poziciju kamere.
- **GetRotation:** vraća rotaciju kamere.
- **GetProjectionMatrix:** vraća matricu projekcije.
- **GetViewMatrix:** vraća matricu pogleda.
- **GetViewProjectionMatrix:** vraća matricu pogledprojekcije.

3.9 Sijenčar

Sijenčar se uobičajeno piše u zasebnoj datoteci čiji nastavak i sintaksa ovise i sučelju prikazivača koji se koristi. Sijenčar kao funkcionalnost platforme treba enkapsulirati kod koji se nalazi u datoteci sijenčara i funkcije potrebne za prenošenje podataka u kod sijenčara.

Implementacija klase za sijenčar:

```
1 class Shader
2 {
3 public:
4   // Takes source code for vertex and fragment shaders
5   Shader(const std::string &name, const std::string &vertexSource,
     const std::string &fragmentSource);
6   Shader(const std::string &filepath);
7   ~Shader();
8
9   void Bind() const;
10  void Unbind() const;
11
12  void UploadUniformMat4(const glm::mat4 &matrix, const std::string
      &name);
13  void UploadUniformVec4(const glm::vec4 &color, const std::string
     &name);
14  void UploadUniformVec2(const glm::vec2 &texcord, const std::
     string &name);
```

```
15   void UploadUniformInt(const int tex, const std::string &name);
16   void UploadUniformFloat(const float ft, const std::string &name);
17
18   inline uint32_t GetRendererId() { return m_RendererID; }
19   inline const std::string &GetName() { return m_Name; }
20
21 private:
22   std::string ReadFile(const std::string &filepath);
23   std::unordered_map<unsigned int, std::string> PreProcess(const
      std::string &source);
24   void Compile(std::unordered_map<unsigned int, std::string>
      ShaderSources);
25   uint32_t m_RendererID;
26   std::string m_Name;
27 };
```

Funkcije:

- **Shader - konstruktor:** prima put do datoteke sijenčara i uzima sve potrebne podatke i obrađuje ih pomoću funkcija sučelja za prikaz grafike.
- **Bind:** koristeći funkcije sučelja za prikaz grafike započinje korištenje ovog sijenčara.
- **Unbind:** koristeći funkcije sučelja za prikaz grafike prekida korištenje ovog sijenčara.
- **UploadUniform:** prenosi proslijeđene podatke na sijenčar.
- **GetRendererId:** vraća jedinstvenu oznaku sijenčara što je prirodni broj.
- **GetName:** vraća ime sijenčara.

U nastavku ćemo pokazati kako se samo uz ove opisane funkcionalnosti mogu napraviti jednostavne igrice. Igrica se stvara tako što se napravi instanca aplikacije i za svaki sloj 3.6 te aplikacije se definira što se logički događa na tom sloju, koje događaje 2.3 on uzima i što se treba prikazivati na tom sloju.

4 | Primjer igrice - Asteroidi

Ideja ove igrice je vrlo jednostavna. Mali svemirski brod koji mora izbjegavati asteroide. Svakih 15 sekundi započinje nova runda i svaku novu rundu broj rundi se poveća za jedan. Ova igrica može bez problema raditi na jednom sloju. Pošto nema interaktivno korisničko sučelje, ono se može nalaziti na istom sloju kao i ostatak igrice. Jedina kompliciranija stvar što je potrebno implementirati jest detekcija sudara dva objekta. Osim toga sve ostalo je jednostavno: treba napraviti klasu za igrača koja će enkapsulirati podatke poput teksture, pozicije, veličine i brzine kretanja, dalje treba napraviti kontrole za igrača, logiku stvaranja i putovanja asteroida te napraviti neke objekte koji će služiti samo estetici. Slika 4.5 prikazuje gotovu igricu.

4.1 Značajke

4.1.1 Igrač

Igrač će u suštini biti samo pravokutnik s teksturom kojeg se može pomjerati. Sve što je potrebno za njega jest klasa koja enkapsulira teksturu, varijablu koja predstavlja brzinu kretanja i varijablu koja predstavlja veličinu igrača. Igračeva brzina namijenjena je da se povećava do maksimalne brzine što se više kreće, tj. dok se igrač ne kreće brzina kretanja se stalno smanjuje do nule. Na slici 4.1 prikazana je tekstura igrača.

Slika 4.1: Tekstura igrača

4.1.2 Asteroidi

Asteroidi su prepreke koje igrač mora izbjegavati. Stvaraju se nasumično na rubu vidnog polja igrača i nasumičnom brzinom se kreću prema dnu vidnog polja igrača gdje se briše iz memorije i stvara se novi asteroid na njegovo mjesto. Brzina i broj asteroida koji lete su nasumični ali unutar nekih granica. Na slici 4.2 prikazana je tekstura koju će imati asteroidi.

Slika 4.2: Tekstura asteroida

4.1.3 Planeti

Planeti su pravokutnici koji se ne mogu sudariti s igračem. Kada se oni pojave na ekranu znači da je započela nova runda i da se broj i brzina asteroida povećavaju. Na slici 4.3 prikazane su teksture za planete.

Slika 4.3: Teksture planeta

4.1.4 Zvijezde

Zvijezde su samo mali bijeli pravokutnici koji lete prema dolje kako bi dali efekta prolaska svemirskog broda kroz svemir. Ne mogu se sudariti s igračem i postaju sve brže svaku rundu.

4.1.5 Korisničko sučelje

Korisničko sučelje je ovom smislu je samo tekst koji prikazuje podatke o rundama i vremenu koje je prošlo od početka igranja. Na slici 4.4 prikazano je korisničko sučelje kakvo će izgledati u igri.

Slika 4.4: Korisničko sučelje

4.2 Implementacija značajki

4.2.1 Igrač

Igrača nije teško implementirati. Kontrole će se lako implementirati pomoću ispita unosa 2.4. Pomoću funkcija prikazivača 2.6 i sijenčara 2.9 se jednostavno može

nacrtati igrač s njegovom teksturom.

```
void OnUpdate(DeltaTime dt)
{
  if (CheckInput())
  {
    m_Speed = clamp(m_Speed * dt.GetMiliseconds(), m_Speed -= 8.0f
   * dt.GetMiliseconds(), 600.0f * dt.GetMiliseconds());
  }
  if (IsKeyPressedStatic(VOS_KEY_A))
  {
    m_Speed = clamp(m_Speed * dt.GetMiliseconds(), m_Speed += 8.0f
   * dt.GetMiliseconds(), 600.0f * dt.GetMiliseconds());
    m_Position.x -= m_Speed * dt.GetMiliseconds();
  }
  if (IsKeyPressedStatic(VOS_KEY_D))
  {
    m_Speed = clamp(m_Speed * dt.GetMiliseconds(), m_Speed += 8.0f
   * dt.GetMiliseconds(), 600.0f * dt.GetMiliseconds());
    m_Position.x += m_Speed * dt.GetMiliseconds();
  }
  if (IsKeyPressedStatic(VOS_KEY_W))
  {
    m_Speed = clamp(m_Speed * dt.GetMiliseconds(), m_Speed += 8.0f
   * dt.GetMiliseconds(), 600.0f * dt.GetMiliseconds());
    m_Position.y += m_Speed * dt.GetMiliseconds();
  }
  if (IsKeyPressedStatic(VOS_KEY_S))
  {
    m_Speed = clamp(m_Speed * dt.GetMiliseconds(), m_Speed += 8.0f
   * dt.GetMiliseconds(), 600.0f * dt.GetMiliseconds());
    m_Position.y -= m_Speed * dt.GetMiliseconds();
  }

  m_Position.x = clamp(m_Position.x, -4.75f, 4.75f);
  m_Position.y = clamp(m_Position.y, -8.0f, 8.0f);

  DrawRotatedQuad(m_Position, m_Size, 180.f, m_ShipTexture);
}
```

Funkcije:

- **OnUpdate:** funkcija će se pozivati svaki prolazak glavne while petlje aplikaciju (tj. svaku sličicu). U njoj je pomoću ispita unosa definirano koliko se jedinica prozora igrač kreće za trenutnu brzinu i prikazuje se igrač na njegovoj trenutnoj poziciji.

4.2.2 Asteroidi

Pomoću funkcija prikazivača 2.6 i sijenčara 2.9 se jednostavno može nacrtati asteroid s njegovom teksturom.

MoveAsteroid:

```
for (auto &asteroid : m_Asteroids)
{
    if (asteroid.position.y < -8.5f)
    {
        CreateAsteroid(asteroid.index);
    }
    asteroid.position.y -= asteroid.speed;
    asteroid.rotation += asteroid.rotationSpeed;
    DrawRotatedQuad(vec3{asteroid.position}, asteroid.size, (float)
    asteroid.rotation, m_Texture);
}
```

Svaki prolazak glavne while petlje asteroidi se pomiču prema dnu prozora za njihovu brzinu koja se određuje nasumično pri njihovom stvaranju. Kada je jedan asteroid došao do dna prozora, on se briše i stvara se jedan novi asteroid na vrhu prozora pomoću funkcije CreateAsteroid.

CreateAsteroid:

```
void CreateAsteroid(int index)
{
    Asteroid &asteroid = m_Asteroids[index];
    asteroid.position = vec3(Dist() * 5.0f * myRandom(), 10.f,
    index * 0.1f - 0.5f);
    asteroid.speed = std::max(Dist() * m_AsteroidSpeed, 0.5f *
    m_AsteroidSpeed);
    asteroid.rotation = Dist() * 360 * myRandom();
    asteroid.rotationSpeed = Dist() * 3;
    asteroid.size = vec2(std::max(Dist() * 1.0f, 0.5f));
    asteroid.index = index;
}
```

Funkcija stvara novi asteroid u vektoru asteroida i nasumično mu dodjeljuje brzinu i poziciju unutar zadanih granica.

4.2.3 Planeti

Pomoću funkcija prikazivača 2.6 i sijenčara 2.9 se jednostavno može nacrtati planet s njegovom teksturom.

MovePlanet:

```
if (m_Planet.position.y > -10.0f && m_Planet.show)
{
    m_Planet.position.y -= m_Planet.speed;
    DrawQuad(m_Planet.position, {2.0f, 2.0f}, m_Planet.
    planetTexture);
}
else
{
```

```
8       m_Planet.show = false;
9   }
```

Svaki prolazak glavne while petlje, ako je prošlo 15 sekundi od zadnje pojave planeta, planet se pomiče prema dnu prozora za njegovu brzinu koja se dodjeljuje pri njegovom stvaranju. Kada je jedan planet došao do dna prozora, i za 15 sekundi stvara se jedan novi planet na vrhu prozora pomoću funkcije CreatePlanet.

CreatePlanet:

```
1  void CreatePlanet(int index)
2  {
3       m_Planet.planetTexture = m_Textures[index];
4       m_Planet.position = vec3(Dist() * 5.0f * myRandom(), 10.f, -0.7
       f);
5       m_Planet.speed = 0.3f * m_AsteroidSpeed;
6       m_Planet.show = true;
7  }
```

Funkcija stvara novi planet u vektoru planeta i nasumično mu dodjeljuje poziciju unutar zadanih granica.

4.2.4 Zvijezde

Pomoću funkcija prikazivača 2.6 i sijenčara 2.9 se jednostavno može nacrtati zvijezda s njezinom teksturom.

MoveStar:

```
1  for (auto &asteroid : m_Asteroids)
2  {
3       if (asteroid.position.y < -8.5f)
4       {
5           CreateAsteroid(asteroid.index);
6       }
7       asteroid.position.y -= asteroid.speed;
8       asteroid.rotation += asteroid.rotationSpeed;
9       DrawRotatedQuad(vec3{asteroid.position}, asteroid.size, (float)
       asteroid.rotation, m_Texture);
10 }
```

Svaki prolazak glavne while petlje zvijezde se pomiču prema dnu prozora za njihovu brzinu koja se određuje nasumično pri njihovom stvaranju. Kada je jedna zvijezda došla do dna prozora, ona se briše i stvara se jedna nova zvijezda na vrhu prozora pomoću funkcije CreateStar.

CreateStar:

```
void CreateAsteroid(int index)
{
    Asteroid &asteroid = m_Asteroids[index];
    asteroid.position = vec3(Dist() * 5.0f * myRandom(), 10.f,
    index * 0.1f - 0.5f);
    asteroid.speed = std::max(Dist() * m_AsteroidSpeed, 0.5f *
    m_AsteroidSpeed);
    asteroid.rotation = Dist() * 360 * myRandom();
    asteroid.rotationSpeed = Dist() * 3;
    asteroid.size = vec2(std::max(Dist() * 1.0f, 0.5f));
    asteroid.index = index;
}
```

Funkcija stvara novu zvijezdu u vektoru zvijezda i nasumično joj dodjeljuje brzinu i poziciju unutar zadanih granica.

4.2.5 Detekcija sudara

Koristeći transformaciju 2.8 lako se može provjeriti jesu li se dva objekta na prozoru 2.1 sudarila. Postoji vrlo jednostavan matematički postupak koji provjerava nalazi li se neka točke prvog pravokutnika unutar stranica drugog pravokutnika. Ovo je najjednostavniji način za detektirati jesu li se dva objekta sudarila pošto su objekti za koje se provjerava sudar uvijek pravokutnici. U ovom slučaju provjerava se je li se igrač sudario s bilo kojim od asteroida.

OnCollision:

```
bool OnCollision()
{
    vec4 Verticies[4] = {
        {-0.5f, 0.5f, 0.0f, 1.0f},
        {-0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, 0.5f, 0.0f, 1.0f}};

    vec4 playerTransformedVerticies[4];

    const auto &position = m_Player.GetPosition();
    const auto &size = m_Player.GetSize();

    for (int i = 0; i < 4; i++)
    {
        playerTransformedVerticies[i] = translate(mat4(1.0f),
    position) * scale(mat4(1.0f), vec3(size.x, size.y, 1.0f)) *
    Verticies[i];
    }

    for (auto &asteroid : m_Asteroids)
    {

        vec2 asteroidVertecies[4];

```

```
24          for (int i = 0; i < 4; i++)
25          {
26              asteroidVertecies[i] = translate(mat4(1.0f), asteroid.
    position) * rotate(mat4(1.0f), radians(asteroid.rotation), vec3
    (0.0f, 0.0f, 1.0f)) * scale(mat4(1.0f), vec3(asteroid.size.x,
    asteroid.size.y, 1.0f)) * Verticies[i];
27          }
28
29          for (auto vert : playerTransformedVerticies)
30          {
31              if (Collided(vec2(vert.x, vert.y), asteroidVertecies
    [0], asteroidVertecies[1], asteroidVertecies[2],
    asteroidVertecies[3]))
32              {
33                  return true;
34              }
35          }
36      }
37      return false;
38  }
```

Na točke koje čine vrhove igrača se primjeni transformacija 2.8. Zatim se prolazi kroz sve asteroide kako bi se i na njihove početne točke primijenila transformacija. S takvim transformiranim točkama asteroida se za svaku transformiranu točku igrača provjerava je li došlo do sudara pomoću funkcije Collided.

Collided:

```
1  bool Collided(vec2 vert, vec2 vertOne, vec2 vertTwo, vec2 vertThree
       , vec2 vertFour)
2  {
3       float DOne = (vertTwo.x - vertOne.x) * (vert.y - vertOne.y) - (
       vert.x - vertOne.x) * (vertTwo.y - vertOne.y);
4       float DTwo = (vertThree.x - vertTwo.x) * (vert.y - vertTwo.y) -
        (vert.x - vertTwo.x) * (vertThree.y - vertTwo.y);
5       float DThree = (vertFour.x - vertThree.x) * (vert.y - vertThree
       .y) - (vert.x - vertThree.x) * (vertFour.y - vertThree.y);
6       float DFour = (vertOne.x - vertFour.x) * (vert.y - vertFour.y)
       - (vert.x - vertFour.x) * (vertOne.y - vertFour.y);
7       if (DOne >= 0 && DTwo >= 0 && DThree >= 0 && DFour >= 0)
8       {
9           return true;
10      }
11      else
12      {
13          return false;
14      }
15 }
```

Funkcija prima jednu točku igrača i sve točke asteroida pa provjerava nalazi li se točka igrača unutar svih stranica asteroida.

Slika 4.5: Gotova igrica

5 | Primjer igrice - Platformer

Ideja igre je simulacija života studenta na fakultetu. Igrač treba skupljati skripte za ispit i izbjegavati društvene mreže koje uzrokuju prokrastinaciju. Treba skupiti svih sedam skripti koje su u nivou, a ako se igrač sudari s društvenom mrežom izgubi jedan od tri života. Kada izgubi sve živote igrač neslavno diplomira i igra prestaje. Upravljanje igračem je takvo da može skakati po platformama i da ga kamera prati kako bi se mogao kretati u većem prostoru. Jedini veći problemi koji treba riješiti jesu skakanje igrača i detekcija sudara igrača i prepreke. Slika 5.5 prikazuje gotovu igricu.

5.1 Značajke

5.1.1 Igrač

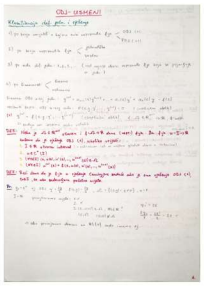
Igrač će u suštini biti samo pravokutnik s teksturom kojeg se može pomjerati. Njegova klasa enkapsulira teksturu, varijablu koja predstavlja brzinu kretanja, varijablu koja predstavlja veličinu igrača i varijablu brzine kretanja igrača prema dolje. Problem skakanja može se riješiti primitivnom simulacijom gravitacije. Igračeva brzina namijenjena je da se povećava do maksimalne brzine što se više kreće, tj. dok se igrač ne kreće brzina kretanja se stalno smanjuje do nule. Gravitacija će biti brzina kojom igrač ide prema dolje, kada se igrač odmakne od zemlje ta brzina počinje raste dok ne prestigne brzinu kojom se igrač kreće prema gore i igrač se počne spuštati prema dolje dok ne udari u neku podlogu. Kada igrač dotakne podlogu brzina kojom igrač ide prema dolje se vraća na početnu. Na slici 5.1 prikazana je tekstura igrača.

Slika 5.1: Tekstura igrača

5.1.2 Skripte

Skripte su pravokutnici s teksturom koji se, radi komunikacije važnosti objekta igraču, vrte oko y osi. Kada se pokupi neka skripta bodovi igrača se povećavaju za

1, a cilj igre je skupiti sedam bodova. Na slici 5.2 prikazana je tekstura za skripte.



Slika 5.2: Tekstura skripte

5.1.3 Prepreke

Prepreke su pravokutnici s teksturama koje imaju definirane obrasce kretanja. Kada se igrač sudari s njima gubi jedan od tri života, ako igraču životi dođu na nulu, igra prestaje. Na slici 5.3 prikazane su teksture za prepreke.

Slika 5.3: Teksture prepreka

5.1.4 Korisničko sučelje

Korisničko sučelje je ovom smislu je samo tekst koji prikazuje podatke o bodovima, životima i vremenu koje je prošlo od početka igranja. Na slici 5.4 prikazano je korisničko sučelje kakvo će izgledati u igri.

Slika 5.4: Korisničko sučelje

5.2 Implementacija značajki

5.2.1 Igrač

Igrača nije teško implementirati. Kontrole će se lako implementirati pomoću ispita unosa 2.4. Pomoću funkcija prikazivača 2.6 i sijenčara 2.9 se jednostavno može nacrtati igrač s njegovom teksturom.

```
1  void Player::OnUpdate(DeltaTime dt)
2  {
3    if (CheckInput() && !(m_InAir))
4    {
5      m_Speed = glm::clamp(m_Speed * dt.GetMiliseconds(), m_Speed -=
     8.0f * dt.GetMiliseconds(), m_SpeedCap * dt.GetMiliseconds());
6    }
7
8    if (m_InAir)
9    {
10     float temp = m_Speed;
11     m_FallSpeed = glm::clamp(m_FallSpeed * dt.GetMiliseconds(),
     m_FallSpeed += 40.0f * dt.GetMiliseconds(), (m_SpeedCap + 100.0f
     ) * dt.GetMiliseconds());
12     if (m_FallSpeed > m_Speed || !(IsKeyPressedStatic(VOS_KEY_W)))
13     {
14       m_Position.y -= (m_FallSpeed + 4.0f) * dt.GetMiliseconds();
15       m_EnableJump = false;
16     }
17   }
18   else
19   {
20     m_EnableJump = true;
21     m_FallSpeed = 0.0f;
22   }
23
24   if (IsKeyPressedStatic(VOS_KEY_A))
25   {
26
27     if (m_InAir)
28     {
29       m_Speed = glm::clamp(m_Speed * dt.GetMiliseconds(), m_Speed
     += 8.0f * dt.GetMiliseconds(), m_SpeedCap * dt.GetMiliseconds())
     ;
30       m_Position.x -= (m_Speed + 4.0f) * dt.GetMiliseconds();
31     }
32     else
33     {
34       m_Speed = glm::clamp(m_Speed * dt.GetMiliseconds(), m_Speed
     += 8.0f * dt.GetMiliseconds(), m_SpeedCap * dt.GetMiliseconds())
     ;
35       m_Position.x -= m_Speed * dt.GetMiliseconds();
36     }
37     if (m_Size.x < 0.0f)
38       m_Size.x *= -1.0f;
39   }
40   else if (IsKeyPressedStatic(VOS_KEY_D))
41   {
42     if (m_InAir)
43     {
44       m_Speed = glm::clamp(m_Speed * dt.GetMiliseconds(), m_Speed
     += 8.0f * dt.GetMiliseconds(), m_SpeedCap * dt.GetMiliseconds())
     ;
45       m_Position.x += (m_Speed + 4.0f) * dt.GetMiliseconds();
46     }
47     else
```

```
48      {
49        m_Speed = glm::clamp(m_Speed * dt.GetMiliseconds(), m_Speed
      += 8.0f * dt.GetMiliseconds(), m_SpeedCap * dt.GetMiliseconds())
      ;
50        m_Position.x += m_Speed * dt.GetMiliseconds();
51      }
52      if (m_Size.x > 0.0f)
53        m_Size.x *= -1.0f;
54    }
55    if (IsKeyPressedStatic(VOS_KEY_W) && m_EnableJump)
56    {
57      if (m_Speed <= 5.0f)
58      {
59        m_Speed = 5.0f;
60        m_Position.y += (m_Speed)*dt.GetMiliseconds();
61      }
62      else
63      {
64        m_Position.y += (m_Speed + 4.0f) * dt.GetMiliseconds();
65      }
66    }
67    else if (IsKeyPressedStatic(VOS_KEY_S))
68    {
69      if (m_InAir)
70      {
71        m_Speed = glm::clamp(m_Speed * dt.GetMiliseconds(), m_Speed
      += 8.0f * dt.GetMiliseconds(), m_SpeedCap * dt.GetMiliseconds())
      ;
72        m_Position.y -= (m_Speed + 4.0f) * dt.GetMiliseconds();
73      }
74      else
75      {
76        m_Speed = glm::clamp(m_Speed * dt.GetMiliseconds(), m_Speed
      += 8.0f * dt.GetMiliseconds(), m_SpeedCap * dt.GetMiliseconds())
      ;
77        m_Position.y -= m_Speed * dt.GetMiliseconds();
78      }
79    }
80    if (IsKeyPressedStatic(VOS_KEY_LEFT_SHIFT))
81    {
82      m_SpeedCap = 1400.0f;
83    }
84    else
85    {
86      m_SpeedCap = 1000.0f;
87    }
88    m_Position.x = glm::clamp(m_Position.x, -15.75f, 57.0f);
89    m_Position.y = glm::clamp(m_Position.y, -4.75f, 35.0f);
90    Renderer2D::DrawRotatedQuad(m_Position, m_Size, 180.f,
      m_ShipTexture);
91  }
```

Funkcije:

- **OnUpdate:** funkcija će se pozivati svaki prolazak glavne while petlje apli-

kaciju (tj. svaku sličicu). U njoj je pomoću ispita unosa definirano koliko se jedinica prozora igrač kreće za trenutnu brzinu i prikazuje se igrač na njegovoj trenutnoj poziciji. Također upravlja logikom skakanja tj. postavlja brzinu padanja i regulira brzinu kretanja u zraku.

5.2.2 Skripte

Skripte su samo pravokutnici s teksturom koji se vrti oko svoje osi. Potrebno ih je samo nacrtati i prepoznati kada se igrač sudari s njima kako bi se obrisali i povećao broj bodova igraču.

OnUpdate:

```
for (auto &el : m_Scripts)
{
    el.rotation += 1.0f;
}
```

Skripte se za svaku prolazak glavne while petlje (tj. svaku sličicu) okreću oko svoje y osi za zadanu vrijednost. To signalizira igraču da su ovi objekti bitniji od ostalih.

CreateScript:

```
void CreateScripts()
{
    m_Scripts.resize(7);

    for (int i = 0; i < m_Scripts.size(); i++)
    {
        m_Scripts[i].size = vec2(1.5f, 1.4f * 1.5f);
        m_Scripts[i].rotation = 0.0f;
        m_Scripts[i].scriptTexture.reset(new Texture2D("assets/
    textures/script.png"));
        m_Scripts[i].index = i;
    }
    m_Scripts[0].position = vec3(-3.0f, -3.5f, -0.2f);
    m_Scripts[1].position = vec3(55.0f, -3.5f, -0.2f);
    m_Scripts[2].position = vec3(-12.0f, 17.5f, -0.3f);
    m_Scripts[3].position = vec3(23.0f, 12.5f, -0.3f);
    m_Scripts[4].position = vec3(35.0f, 8.5f, -0.3f);
    m_Scripts[5].position = vec3(23.0f, 27.0f, -0.3f);
    m_Scripts[6].position = vec3(53.0f, 32.0f, -0.3f);
}
```

Skripte se ne kreću, ali potrebno im je zadati podatke o teksturi, poziciji i brzini rotacije. Sve skripte su spremljene u vektor skripti. Jednom kada su skripte stvorene potrebno ih je samo obrisati kada se pokupe.

5.2.3 Prepreke

Prepreke su pravokutnici s teksturom koji imaju zadani smjer kretanja. Kada se sudare s igračem, igrač gubi jedan život. **OnUpdate:**

```
for (auto &obs : m_Obstacles)
{
    obs.rotation += obs.speed * 90.0f * dt.GetMiliseconds();
    obs.position += obs.toChange * obs.speed * dt.GetMiliseconds();
    if (obs.orientation)
    {
        obs.travel = distance(obs.position, obs.startPosition);
    }
    else
    {
        obs.travel = distance(obs.position, obs.destination);
    }

    if (obs.toTravel < obs.travel)
    {
        obs.speed *= -1;
        obs.orientation = !obs.orientation;
    }
}
```

Prepreke se za svaku prolazak glavne while petlje (tj. svaku sličicu) okreću oko svoje z osi za zadanu vrijednost i pomiču u određenom smjeru za svoju brzinu. Njihovo brzo okretanje signalizira igraču opasnost.

CreateObstacle:

```
void Level::CreateObstacles()
{
    m_Obstacles.resize(8);
    m_Obstacles[0].position = vec3(56.5f, -4.0f, 0.0f);
    m_Obstacles[0].destination = vec3(-15.0f, -4.0f, 0.0f);
    m_Obstacles[0].toChange = vec3(1.0f, 0.0f, 0.0f);
    m_Obstacles[0].speed = 10.0f;

    m_Obstacles[1].position = vec3(-1.5f, 7.3f, 0.0f);
    m_Obstacles[1].destination = vec3(-4.0f, 25.0f, 0.0f);
    m_Obstacles[1].toChange = vec3(0.0, 1.0f, 0.0f);
    m_Obstacles[1].speed = 12.0f;

    m_Obstacles[2].position = vec3(35.6f, 26.5f, 0.0f);
    m_Obstacles[2].destination = vec3(55.5f, 26.5f, 0.0f);
    m_Obstacles[2].toChange = vec3(1.0f, 0.0f, 0.0f);
    m_Obstacles[2].speed = 12.0f;

    for (int i = 0; i < m_Obstacles.size(); i++)
    {
        m_Obstacles[i].size = vec2(1.6f, 1.6f);
        if (i % 3 == 0)
```

```
23          {
24              m_Obstacles[i].obstacleTexture.reset(new Texture2D("
    assets/textures/insta.png"));
25          }
26          if (i % 3 == 1)
27          {
28              m_Obstacles[i].obstacleTexture.reset(new Texture2D("
    assets/textures/whap.png"));
29          }
30          if (i % 3 == 2)
31          {
32              m_Obstacles[i].obstacleTexture.reset(new Texture2D("
    assets/textures/fb.png"));
33          }
34
35          m_Obstacles[i].rotation = 0.0f;
36          m_Obstacles[i].startPosition = m_Obstacles[i].position;
37          m_Obstacles[i].travel = 0.0f;
38          m_Obstacles[i].toTravel = distance(m_Obstacles[i].
    startPosition, m_Obstacles[i].destination);
39          m_Obstacles[i].orientation = true;
40          if (dot(m_Obstacles[i].startPosition, m_Obstacles[i].
    toChange) > dot(m_Obstacles[i].destination, m_Obstacles[i].
    toChange))
41          {
42              m_Obstacles[i].speed *= -1.0f;
43          }
44      }
45  }
```

Preprekama je potrebno zadati podatke o početnoj i završnoj točki kretanja, brzini rotacije, brzini kretanje i teksturi. Prepreke se stvore na početku i one ostaju tamo do kraja igre, nema potrebe za brisanjem ili stvaranjem novih.

5.2.4 Detekcija sudara

Koristeći transformaciju 2.8 lako se može provjeriti jesu li se dva objekta na prozoru 2.1 sudarila. Postoji vrlo jednostavan matematički postupak koji provjerava nalazi li se neka točke prvog pravokutnika unutar stranica drugog pravokutnika. Ovo je najjednostavniji način za detektirati jesu li se dva objekta sudarila pošto su objekti za koje se provjerava sudar uvijek pravokutnici.

U ovom slučaju provjerava se je li igrač sudario s bilo kojom od prepreka da bi se znalo da treba izgubiti život i treba provjeriti je li se igrač sudario sa skriptom kako bi se znalo da treba skriptu maknuti i povećati igračev broj bodova. Također treba provjeriti je li igrač sudario s nekom podlogom (tj. podom ili platformom) kako bi se prestao kretati prema dolje.

Collided with Script:

```
1 bool Level::OnPickup()
```

```
{
    vec4 Verticies[4] = {
        {-0.5f, 0.5f, 0.0f, 1.0f},
        {-0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, 0.5f, 0.0f, 1.0f}};

    vec4 playerTransformedVerticies[4];

    const auto &position = m_Player.GetPosition();
    const auto &size = m_Player.GetSize();

    for (int i = 0; i < 4; i++)
    {
        playerTransformedVerticies[i] = translate(mat4(1.0f),
    position) * scale(mat4(1.0f), vec3(size.x, size.y, 1.0f)) *
    Verticies[i];
    }

    for (auto &script : m_Scripts)
    {

        vec2 floorVerticies[4];

        for (int i = 0; i < 4; i++)
        {
            floorVerticies[i] = translate(mat4(1.0f), script.
    position) * rotate(mat4(1.0f), radians(0.0f), vec3(0.0f, 0.0f,
    1.0f)) * scale(mat4(1.0f), vec3(script.size.x, script.size.y,
    1.0f)) * Verticies[i];
        }

        for (int j = 0; j < 4; j++)
        {
            if (Collided(vec2(playerTransformedVerticies[j].x,
    playerTransformedVerticies[j].y), floorVerticies[0],
    floorVerticies[1], floorVerticies[2], floorVerticies[3]))
            {
                m_Index = script.index;
                return true;
            }
        }
    }

    return false;
}
```

Na točke koje čine vrhove igrača se primjeni transformacija 2.8. Zatim se prolazi kroz sve skripte kako bi se i na njihove početne točke primijenila transformacija. S takvim transformiranim točkama skripti se za svaku transformiranu točku igrača provjerava je li došlo do sudara pomoću funkcije Collided.

Collided with Obstacle:

```
bool Level::OnHit()
{
    vec4 Verticies[4] = {
        {-0.5f, 0.5f, 0.0f, 1.0f},
        {-0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, 0.5f, 0.0f, 1.0f}};

    vec4 playerTransformedVerticies[4];

    const auto &position = m_Player.GetPosition();
    const auto &size = m_Player.GetSize();

    for (int i = 0; i < 4; i++)
    {
        playerTransformedVerticies[i] = translate(mat4(1.0f),
    position) * scale(mat4(1.0f), vec3(size.x, size.y, 1.0f)) *
    Verticies[i];
    }

    for (auto &script : m_Obstacles)
    {
        vec2 floorVerticies[4];

        for (int i = 0; i < 4; i++)
        {
            floorVerticies[i] = translate(mat4(1.0f), script.
    position) * rotate(mat4(1.0f), radians(0.0f), vec3(0.0f, 0.0f,
    1.0f)) * scale(mat4(1.0f), vec3(script.size.x, script.size.y,
    1.0f)) * Verticies[i];
        }

        for (int j = 0; j < 4; j++)
        {
            if (Collided(vec2(playerTransformedVerticies[j].x,
    playerTransformedVerticies[j].y), floorVerticies[0],
    floorVerticies[1], floorVerticies[2], floorVerticies[3]))
            {
                return true;
            }
        }
    }

    return false;
}
```

Na točke koje čine vrhove igrača se primjeni transformacija 2.8. Zatim se prolazi kroz sve skripte kako bi se i na njihove početne točke primijenila transformacija. S takvim transformiranim točkama skripti se za svaku transformiranu točku igrača provjerava je li došlo do sudara pomoću funkcije Collided.

Collided with floor:

```
bool OnCollision()
{
    vec4 Verticies[4] = {
        {-0.5f, 0.5f, 0.0f, 1.0f},
        {-0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, -0.5f, 0.0f, 1.0f},
        {0.5f, 0.5f, 0.0f, 1.0f}};

    vec4 playerTransformedVerticies[4];

    const auto &position = m_Player.GetPosition();
    const auto &size = m_Player.GetSize();

    for (int i = 0; i < 4; i++)
    {
        playerTransformedVerticies[i] = translate(mat4(1.0f),
    position) * scale(mat4(1.0f), vec3(size.x, size.y, 1.0f)) *
    Verticies[i];
    }

    for (auto &floor : m_Floors)
    {

        vec2 floorVerticies[4];

        for (int i = 0; i < 4; i++)
        {
            floorVerticies[i] = translate(mat4(1.0f), floor.
    position) * rotate(mat4(1.0f), radians(0.0f), vec3(0.0f, 0.0f,
    1.0f)) * scale(mat4(1.0f), vec3(floor.size.x, floor.size.y, 1.0f
    )) * Verticies[i];
        }

        for (auto vert : playerTransformedVerticies)
        {
            if (Collided(vec2(vert.x, vert.y), floorVerticies[0],
    floorVerticies[1], floorVerticies[2], floorVerticies[3]))
            {
                return true;
            }
        }
    }
}
```

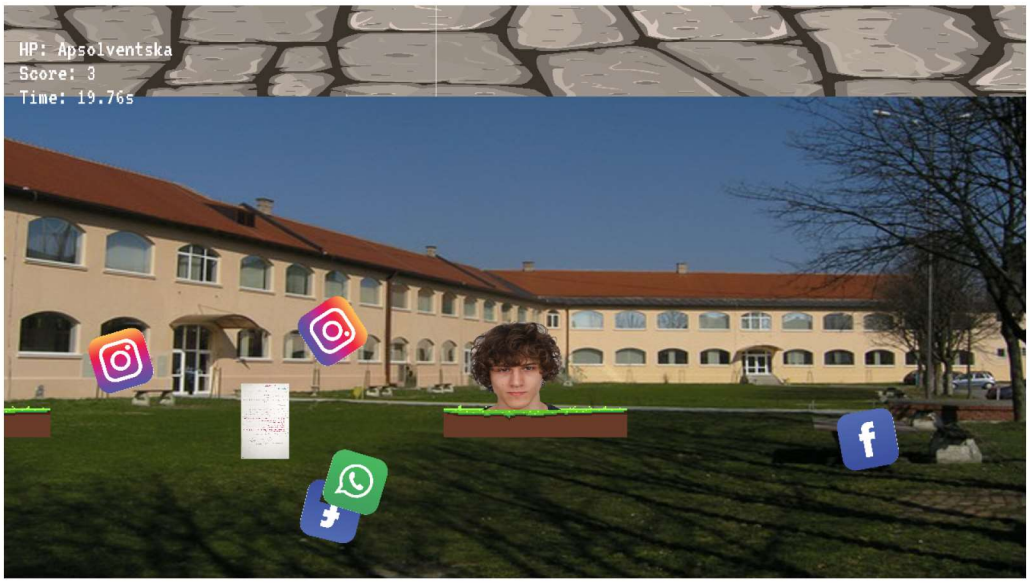
Na točke koje čine vrhove igrača se primjeni transformacija 2.8. Zatim se prolazi kroz sve skripte kako bi se i na njihove početne točke primijenila transformacija. S takvim transformiranim točkama skripti se za svaku transformiranu točku igrača provjerava je li došlo do sudara pomoću funkcije Collided.

Collided:

```
bool Collided(vec2 vert, vec2 vertOne, vec2 vertTwo, vec2 vertThree
    , vec2 vertFour)
```

```
{
    float DOne = (vertTwo.x - vertOne.x) * (vert.y - vertOne.y) - (vert.x - vertOne.x) * (vertTwo.y - vertOne.y);
    float DTwo = (vertThree.x - vertTwo.x) * (vert.y - vertTwo.y) - (vert.x - vertTwo.x) * (vertThree.y - vertTwo.y);
    float DThree = (vertFour.x - vertThree.x) * (vert.y - vertThree.y) - (vert.x - vertThree.x) * (vertFour.y - vertThree.y);
    float DFour = (vertOne.x - vertFour.x) * (vert.y - vertFour.y) - (vert.x - vertFour.x) * (vertOne.y - vertFour.y);
    if (DOne >= 0 && DTwo >= 0 && DThree >= 0 && DFour >= 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Funkcija prima jednu točku igrača i sve točke drugog objekta pa provjerava nalazi li se točka igrača unutar svih stranica drugog objekta.



Slika 5.5: Gotova igrica

Literatura

[1] Web izvor dostupan na https://broken-bytes.medium.com/building-a-game-engine-from-scratch-6a45c9f21317, Marcel Kulina.

[2] Web izvor dostupan na http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/, OpenGL-tutorials.

[3] Peter Shirley and Steve Marschner and Michael Ashikhmin, *Fundamentals of Computer Graphics, 2nd Edition*, A K Peters, 2005.

[4] Web izvor dostupan na https://github.com/TheCherno/Hazel, TheCherno.

[5] Web izvor dostupan na https://www.youtube.com/c/TheChernoProject, TheChernoProject.

Sažetak

Kako video igrice postaju sve kompliciranije tehnologija za njihovu izradu također postaje sve moćnija i kompliciranija. Kako bi se olakšala proizvodnja igrica prave se platforme koje koriste nove tehnologije da implementiraju specijalne funkcionalnosti. Te funkcionalnosti se koriste za izradu igrica. Uz samo par tih jednostavnih funkcionalnosti može se napraviti platforma pomoću koje se može lako i efikasno praviti jednostavne 2D igrice. Te funkcionalnosti su: prozor, dnevnik, sustav događaja, ispit unosa, slojevi, prikazivač, transformacije i sijenčar. Za implementaciju gotovo svih tih funkcionalnosti koriste se sučelja za programiranje aplikacija koje uvelike olakšavaju pravljenje platforme za stvaranje igrica. Neki primjeri tih igrica bile bi igrice u stilu broda koji izbjegava asteroide ili igrica u stilu platformera.

Ključne riječi

platforma, igrica, implementacija, događaji, slojevi, prikazivač, igrač, platformer, 2D

How to build a game engine

Summary

Video games are becoming more complex by the day and so the technology for their development is becoming more powerful and complex. To make creating games easier, engines are being developed that are using the new technology to implement special functions. These functions are used in creation of video games. With just a few of these simple functions, game engine that has the ability to create simple 2D games can be made. Those functionalities are; window, log, event system, input polling, renderer, transforms and shader. To implement almost all of those functionalities application programming interfaces are used to greatly simplify the creation of a game engine. Some examples of such games would be games made in style of an ship avoiding asteroids or a game in a style of an platformer.

Keywords

engine, game, implementation, events, layers, renderer, player, platformer, 2D