Game engine

VilagOS

Leon Vilagoš
29.8.2022.

Uvod

U suštini igrice nisu ništa drugo nego interaktivne aplikacije. Za izradu interaktivnih aplikacija koriste se skupovi alata koji su sami predočeni kao interaktivna aplikacija koji prvenstveno služe kao interpreter podataka, tj. konvertiranje podataka u čovjeku lako čitljiv obliku u svrhu lako rado s podacima. Takvi skupovi alata mogu imati ili jako široku ili jako specijaliziranu primjenu. Na primjer oni mogu biti namijenjeni za simulaciju fizike, prikaz 3D prostora ili nešto općenitije poput igrica. Skup alata koji je specijaliziran za izradu igrica so općenit zove Game Engine (hrv. Pogon za igrice). Jednostavnije, na game engine možemo gledati kao na platformu na izradu igrica. (Kao platforma game engine ima svoju vrijednost komercijalizacije).

Igrice su svakoga dana sve kompleksnije što prvobitno omogućava tehnologija koja služi za njihov razvoj, stoga lako je zaključiti da su i sami game engine-i vrlo kompleksni. Sve što bi igrica mogla zahtijevati game engine bi trebao podržavati. Na primjer prikaz kompleksne grafike u realnom vremenu, simuliranje fizike, reproduciranje zvukova, mogućnost brzog i efikasnog izvršavanja skripti, animacije objekata, učitavanje i interpretiranje podatak itd.

U ovom radu dati ću opis mojih rješenja ovog problema izrađenih u C++, koristeći OpenGl za prikaz grafike za platformu Windows.

Strukturiranje problema

Kod izrade ovako kompleksnog projekta planiranje i organizacija su ključni.

Radi jednostavnosti sam Game engine neće biti aplikacija već skup funkcija, klasa i resursa koji će se koristiti za izradu igrice, ali neće biti predstavljeni kao aplikacija nego kao Static Library pomoću kojeg ću napraviti igricu.

Mjesto gdje se pokreće aplikacija zvat će se Entry Point (hrv. Točka ulaza). Prva stvar koju će trebati implementirati jest Window (hrv. Prozor) za aplikaciju za što ću koristiti glfw library (hrv. Biblioteka). Zatim ima smisla napraviti Event System (hrv. Sistem za događaje), tj. aplikacija treba prepoznavati klikove miša, tipkovnice i akcije nad prozorom. Tu treba riješiti problem da na primjer da svaki pritisak miša apsolutno svi objekti koji se koji se nalazi ispod miša, dok bi klik miša trebao prihvatiti samo najgornji element aplikacije. Rješenje toga biti će struktura Layer-a (hrv. Slojevi) koja će biti korisno i za puno drugih stvari. U ovom dijelu razvoju engine već ima dobar broj funkcija i bilo bi dobro imati neki Log sustav za ispisivanje poruka u konzolu. Treba postojati neki UI za to ću koristiti ImGui library. Imamo input (hrv. Unos) baziran na event-ovima, bilo bi dobro da imamo input za koji se provjerava svaki frame je li nešto stisnuto već se poziva samo kada se nešto pritisne. U ovom dijelu puno sistema je funkcionalno i možemo preći na to da nešto prikažemo, za to nam treba renderer. Za renderanje koristit ćemo OpenGl library Glad. Nakon toga dodati kameru je lako. Pošto imamo kameru i crtanje objekata puno bi značilo kada bi objekti imali svoju poziciju i širinu u prostoru što ćemo dobiti implementacijom Transform-a (hrv. Transformacija). Uz to još napraviti Shader-e (hrv. Sjenčare) i može se napraviti dobra jednostavna igrica.

Core

Klase i datoteke koje se tiče osnovnih funkcija i upravljanja engine-a.

EntryPoint.h

Header (hrv. Zaglavlje) u kojemu se nalazi samo main funkcija, tj. ovaj header služi kao početak same aplikacije.

```
extern VilagOS::Application* VilagOS::CreateApplication();

int main(int argc, char** argv) {
    VilagOS::Log::Init();
    VOS_CORE_WARN("Initialized Log");
    VOS_CLIENT_INFO("Hello");

    auto App = VilagOS::CreateApplication();
    App->run();
    delete App;
}
```

Slika 1 Entry point

Funkcije:

- **int main(int argc, char** argv):** inicijalizira Log, stvara instancu aplikacije i poziva Run funkciju iz klase Application koja je zadužena za izvođenje aplikacije tijekom cijelog njezinog života i nakon toga je briše iz memorije.

Core.h

Služi za spremanje makroa.

```
#ifdef VOS_PLATFORM_WINDOWS
#if VOS_DYNAMIC_LINK
    #ifdef VOS_BUILD_DLL
        #define VOS_API __declspec(dllexport)
        #define IMGUI_API __declspec(dllexport)
    #else
        #define VOS_API __declspec(dllimport)
        #define IMGUI_API __declspec(dllimport)
    #endif // VOS_BUILD_DLL
#else
    #define VOS_API
#endif
 #endif // VOS_PLATFORM_WINDOWS

#ifdef VOS_ENABLE_ASSERTS
    #define VOS_ASSERT(x, ...) {if(!(x)) { VOS_ERROR("Asserion Failed: {0}", __VA_ARGS__); __debugbreak()};}
    #define VOS_CORE_ASSERT(x, ...) {if(!(x)) { VOS_CORE_ERROR("Asserion Failed: {0}", __VA_ARGS__); __debugbreak()};}
#else
    #define VOS_ASSERT(x, ...)
    #define VOS_CORE_ASSERT(x, ...)
 #endif // VOS_ENABLE_ASSERTS


 #define BIT(x) (1 << x) //shift 1 by x places

 #define VOS_BIND_EVENT_FN(fn) std::bind(&fn, this, std::placeholders::_1)
```

Slika 2 Core.h

Application.h

Klasa koja enkapsulira cijelu aplikaciju, uključujući window i layer-e. Zadužena je za rukovanje s layer-ima, window-om, event-ovima i glavnu petlju same aplikacije.

```
class VOS_API Application
{
public:
    Application();
    virtual ~Application();
    void run();
    void OnEvent(Event& e);
    void PushLayer(Layer* Layer);
    void PushOverlay(Layer* Layer);

    inline WindowMaster& GetWindow() { return *m_Window; };
    inline static Application& GetApp() { return *s_Instance; };

private:
    bool OnWindowClose(WindowCloseEvent& event);
    bool OnWindowResize(WindowResizeEvent& event);
    static Application* s_Instance;

    bool m_Running = true;
    bool m_Minimized = false; //It makes sense that this is in the app

    std::unique_ptr<WindowMaster> m_Window;

    ImguiLayer* m_ImGuiLayer;

    LayerStack m_LayerStack;
    DeltaTime m_TimeOfLastFrame = 0.0f;
    std::shared_ptr<Texture2D> m_Texture;
};
```

Slika 3 Application.h

Funkcije:

- **Application() - konstruktor:** stvara window, inicijalizira renderer i stvara UI layer koji je namijenjen da uvijek bude najgornji te ga stoga zovem overlay (hrv. Gornji sloj).
- **void Run():** tu se nalazi glavna while petlja za izvođenje aplikacije, poziva se u main funkciji koja se nalazi u EntryPoint.h. Za svaki prolazak petlje prolazi kroz sve layer-e da pozove njihove OnUpdate i OnImGuiRender funkcije i poziva OnUpdate funkciju za Window. Također mjeri koliko vremena je potrebo za jedan prolazak petlje što nam daje DeltaTime.
- **void PushLayer(Layer* layer)**: klasa Application enkapsulira vektor m_LayerStack u kojemu će biti spremljeni svi layer-i ova funkcija prima novostvoreni layer i dodaje ga u m_LayerStack na početak.
- **void PushOverlay(Layer* layer)**:: dodaje layer na kraj vekotra m_LayerStack, pošto se overlay (hrv.) mora zadnji nacrtat, elementi vektora se crtaju od prvog do zadnjeg.
- **void OnEvent(Event& event):** prima neki Event i dodjeljuje ga odgovarajućim funkcijama za event-ove Window-a ili ga prosljeđuje dalje svakom layer-u da on s njim rukuje kako treba.
- **bool OnWindowClose(WindowCloseEvent& e):** poziva se ako OnEvent dobije event za zatvaranje prozora. Prekida rad aplikacije.
- **bool OnWindowResize(WindowResizeEvent& e):** poziva se ako OnEvent dobije event za promjenu veličine prozora. Poziva OnWindowResize funkciju iz klase Renderer.

Napomene:

Ovu klasu pruža engine, ali instanca ove klase treba biti stvorena na strani klijenta, pošto sama aplikacija bi trebala naš klijent. Tako je ovo zamišljeno da se ova klasa naslijedi od klase Game na strani klijenta.

U ovom slučaju Application će nasljeđivati klasa Game, ali recimo da ne pravimo igricu već neku 3D simulaciju, onda će ta klasa nasljeđivati klasu Application.

DeltaTime.h

Sadrži samo klasu DeltaTime koja služi za spremanje vremena koje prođe između izvršavanja svake petlje.

```
class DeltaTime {
public:
    DeltaTime(float time) : m_Time(time) {}
    float GetMiliseconds() const { return m_Time / 1000.0f; }
private:
    float m_Time = 0.0f;
};
```

Slika 4 DeltaTime.h

Funkcije:

- **DeltaTime(float time) - konstruktor:** dodjeljuje izračunati vrijednost u milisekundama varijabli m_Time.
- **float GetMiliSeconds() :** ovo je malo zavaravajuće pošto funkcija zapravo vraća vrijeme u sekundama pošto m_Time sprema vrijeme u milisekundama.

Log.h

Koristi spdLog library za ispis poruka na konzolu.

```
namespace VilagOS {
    class VOS_API 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;

    };
}

//Core log macros
#define VOS_CORE_ERROR(...)     :: VilagOS::Log::GetCoreLogger()->error(__VA_ARGS__)
#define VOS_CORE_WARN(...)      :: VilagOS::Log::GetCoreLogger()->warn(__VA_ARGS__)
#define VOS_CORE_INFO(...)      :: VilagOS::Log::GetCoreLogger()->info(__VA_ARGS__)
#define VOS_CORE_TRACE(...)     :: VilagOS::Log::GetCoreLogger()->trace(__VA_ARGS__)
#define VOS_CORE_FATAL(...)     :: VilagOS::Log::GetCoreLogger()->fatal(__VA_ARGS__)

//Client log macros
#define VOS_CLIENT_ERROR(...)   :: VilagOS::Log::GetClientLogger()->error(__VA_ARGS__)
#define VOS_CLIENT_WARN(...)    :: VilagOS::Log::GetClientLogger()->warn(__VA_ARGS__)
#define VOS_CLIENT_INFO(...)    :: VilagOS::Log::GetClientLogger()->info(__VA_ARGS__)
#define VOS_CLIENT_TRACE(...)   :: VilagOS::Log::GetClientLogger()->trace(__VA_ARGS__)
#define VOS_CLIENT_FATAL(...)   :: VilagOS::Log::GetClientLogger()->fatal(__VA_ARGS__)
```

Slika 5 Log.h

MouseButtonCodes.h i KeyCodes.h

Sadržavaju Macro-e za tipke miša i tipke tipkovnice.

Input.h

Input klasa biti će samo interface (hrv. Sučelje) koje će naslijediti window. Ona služi za postizanje event polling-a (hrv. bilježenje događaja). Event polling omogućuje da znamo u bilo kojem trenutku izvođenja programa kada je nešto pritisnuto. Za implementacija ovoga koristit ćemo glfw library pošto on već ima event polling samo ga treba iskoristiti u našem kodu.

```
class VOS_API 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;
};
```

Slika 6 Input.h

Layer.h

Konstrukcija layer se može shvatiti isto kao i layer-i u Photoshopu na primjer. Ono što se nalazi na najgornjem layer-u se crta prvo i na klik ako layer ne propusti event na elemente na nižim layer-ima klik će biti registriran samo od najgornjeg elementa.

```
class VOS_API Layer {
public:
    Layer(const std::string& name = "Layer");
    virtual ~Layer(); //Need a virtual destroctor because this is going to be subclassed

    virtual void OnAttach() {}
    virtual void OnDetach() {}
    virtual void OnUpdate(DeltaTime DeltaTime) {}
    virtual void OnImGuiRender() {}
    virtual void OnEvent(Event& e) {}

    inline const std::string& GetName() const { return m_DebugName; }

protected:
    std::string m_DebugName;
};
```

Slika 7 Layer.h

Funkcije:

- **Layer(const string& name)** – konstruktor.
- **void OnAttach():** poziva se kada se layer stavi na Layer Stack.
- **void OnDetach():** poziva se kada se layer skine sa Layer stack-a.
- **void OnUpdate():** poziva se na svakom frame-u.
- **void OnImGuiRender():** za crtanje ImGui elemenata.
- **void OnEvent(Event& e):** za stvaranje event dispatcher-a.

LayserStack.h

Klasa LayerStack enkapsulira vektor layer-a i funkcije za rad nad vektorom.

```
class VOS_API LayerStack{
public:
    LayerStack();
    ~LayerStack();

    void PushLayer(Layer* layer);
    void PushOverlay(Layer* overlay);
    void PopLayer(Layer* layer);
    void PopOverlay(Layer* overlay);

    std::vector<Layer*>::iterator begin() { return m_Layers.begin(); }
    std::vector<Layer*>::iterator end() { return m_Layers.end(); }

private:
    std::vector<Layer*> m_Layers;
    unsigned int m_LayerInsertIndex = 0;
};
```

Slika 8 LayerStack.h

Funkcije:

- **void PushLayer(Layer* layer):** stavlja layer na početak vektora m_Layers.
- **void PushOverlay(Layer* layer):** stavlja overlay na kraj vektora m_Layers.
- **void PopLayer(Layer* layer):** uklanja overlay iz vektora m_Layers.
- **void PopOverlay(Layer* layer):** uklanja proslijeđeni layer iz layer vektora m_Layers.

Napomene:

Kao struktura podatak layer i overlay su identični i kod ih ne razlikuje, ali za lakše razumijevanje imamo različite funkcije za layer i overlay.

Events

Skup Header-a i file-ova koji sadržavaju sve vezano za

Event.h

Sadržava klase Event i EventDispatcher i enum klase EventType i Event Category.

EventType enum

Enum klasa koja sadržava sve moguće tipove evenata.

```
enum class EventType {
    None = 0,
    WindowClose, WindowResize, WindowFocus, WindowLostFocus, WindowMoved,
    AppTick, AppUpdate, AppRender,
    KeyPressed, KeyReleased, KeyTyped,
    MouseButtonPressed, MouseButtonReleased, MouseMoved, MouseScrolled
};
```

Slika 9 Event types

EventCategory enum

Pošto jedan event može biti u više kategorija, kategorije su organizirane kao bit field tako da pomoću binarni operator or (hrv. ili) mogu pristupiti više kategorija.

```
enum EventCategory {
    None = 0,
    EventCategoryApplication =  BIT(0),
    EventCategoryInput =        BIT(1),
    EventCategoryKeyboard =     BIT(2),
    EventCategoryMouse =        BIT(3),
    EventCategoryMouseButton =  BIT(4)
};
```

Slika 10 Event Categories

Event klasa

Služi kao interface za klase specifičnih evenata.

```
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) {
        return GetCategoryFlags() & category;
    }

    bool m_Handled = false;
};
```

Slika 11 Event

Funkcije

- EventType GetEventType(): vraća EventTpye iz enum class EventType koji je dodijeljen toj instanci event-a.
- const char* GetName(): vraća ime event-a.
- int GetCategoryFlags(): vraća kategorije instance event-a.
- string ToPrint(): ispisuje ime event-a.
- bool IsInCategory(): provjerava je li instanca event-a u prosljeđenoj kategoriji ili ne.

EventDispatcher klasa

Služi za dodjeljivanje funkcija proslijeđenim event-ima.

```
class EventDispatcher {
    template<typename T>
    using EventFn = std::function<bool(T&)>;
public:
    EventDispatcher(Event& event) : m_Event(event){}

    template<typename T>
    bool Dispatch(EventFn<T> func) {
        if (m_Event.GetEventType() == T::GetStaticType()) {
            m_Event.m_Handled = func(*(T*)&m_Event);
            return true;
        }
        return false;
    }

private:
    Event& m_Event;
};
```

Slika 12 Event dispatcher

Funkcije

- **bool Dispatch(std::function<bool(T&)> func):** prima neku bool funkciju func koja prima event m_Event i vraća true ako se neki event smatram obavljenim i ne treba ga proslijediti dalje na sljedeći layer ili false ako funkcija treba biti proslijeđena i na ostale layer-e.

Napomene

Macro-i definirani za jednostavniji kod.

```
#define EVENT_CLASS_TYPE(type)  static EventType GetStaticType() {return EventType::##type;}\
                                virtual EventType GetEventType() const override {return GetStaticType();}\
                                virtual const char* GetName() const override {return #type;}

#define EVENT_CLASS_CATEGORY(category) virtual int GetCategoryFlags() const override {return category;}
```

Slika 13 Event macros

Funkcije:

- **EventType GetStaticType():** vraća EventType instance event-a.
- **EventType GetEventType():** virtualna funkcija za implementirati u klasi koja nasljeđuje klasu Event.
- **cont char* GetName():** vraća ime tipa event-a.
- **int GetCategoryFlags():** vraća kategorija event-a.

ApplicationEvent.h

Enkapsulira event-e koji se tiče same aplikacije, tj. prozora.

WindowResizeEvent klasa

Event za promjenu veličine prozora. Enkapsulira visinu i širinu prozora sa overload-anim (hrv. preop-terećenim) operatorom ispisa.

```
class VOS_API WindowResizeEvent : public Event {
public:
    WindowResizeEvent(unsigned int width, unsigned int height): m_width(width), m_height(height){}

    std::pair<unsigned int, unsigned int> getSizes() const {
        std::pair sizes = std::make_pair(m_width, m_height);
        return sizes;
    }

    std::string ToPrint() const override {
        std::stringstream ss;
        ss << "WindowResizedEvent: (" << getSizes().first << ", "<< getSizes().second<<")";
        return ss.str();
    }

    EVENT_CLASS_CATEGORY(EventCategoryApplication)
    EVENT_CLASS_TYPE(WindowResize)

private:
    unsigned int m_width, m_height;
};
```

Slika 14 Window Resize Event

Funkcije:

- std::pair<unsigned int, unsigned int> GetSizes(): vraća par trenutne visine i širine window-a.
- std::string ToPrint(): ispis visini i širine window-a.

Napomene:

Macro naredbe su definirane i objašnjenje u Event.h napomenama.

WindowCloseEvent klasa

Ova klasa ne treba ništa posebno, sve se implementira u funkciji na koju se event bind-a.

```
class VOS_API WindowCloseEvent : public Event {
public:
    WindowCloseEvent() {}
    EVENT_CLASS_CATEGORY(EventCategoryApplication)
    EVENT_CLASS_TYPE(WindowClose)
};
```

Slika 15 Window Close Event

KeyBoardEvent.h

Tipkovnica će imati event za pritisak na tipkovnici i za kada se pusti tipka na tipkovnici kako bi mogli imati držanje gumba. Kako će ta dva event-a imati zajednička svojstva imat ćemo jednu klasu koja će biti općenito za event tipkovnice koju će ostali event-i tipkovnice nasljeđivati.

KeyboardEvent

Interface za keyboard event klase.

```
class VOS_API KeyboardEvent : public Event {
public:
    inline int GetKeyCode() const { return m_KeyCode; }
    EVENT_CLASS_CATEGORY(EventCategoryKeyboard | EventCategoryInput)

protected:
    int m_KeyCode;
    KeyboardEvent(int keycode): m_KeyCode(keycode) {}
};
```

Slika 16 Keyboard Event

Funkcije

- **int GetKeyCode():** vraća oznaku tipke s tipkovnice.
- **KeyBoardEvent() – konstruktor:** konstruktor je protected zato što ne želim stvoriti instancu klase Keyboard event zato što je uostalom samo interface.

KeyPressedEvent

Event koji za pritisak tipke na tipkovnici.

```
class VOS_API KeyPressedEvent : public KeyboardEvent {
public:
    KeyPressedEvent(int keycode, int repeatCount) : KeyboardEvent(keycode), m_RepeatCount(repeatCount) {}
    inline int GetRepeatCount() const { return m_RepeatCount; }

    std::string ToPrint() const override {
        std::stringstream ss;
        ss << "KeyPressedEvent" << m_KeyCode << " (" << m_RepeatCount << " repeats).";
        return ss.str();
    }

    EVENT_CLASS_TYPE(KeyPressed)

private:
    int m_RepeatCount;
};
```

Slika 17 Key Pressed Event

Funkcije

- **KeyPressedEvent(int keycode, int repeatCount) - konstruktor:** instanciranje KeyboardEvent klase i postavljanje m_RepeatCount varijable.
- **int GetRepeatCount():** vraća vrijednost varijable m_RepeatCount.
- **std::string ToPrint():** ispis pritisnute tipke s tipkovnice i koliko dugo je pritisnuta.

KeyReleasedEvent

Event za kada je tipka na tipkovnici puštena.

```
class  VOS_API KeyReleasedEvent: public KeyboardEvent{
public:
    KeyReleasedEvent(int keycode) : KeyboardEvent(keycode) {}

    std::string ToPrint() const override {
        std::stringstream ss;
        ss << "KeyReleasedEvent: " << m_KeyCode;
        return ss.str();
    }

    EVENT_CLASS_TYPE(KeyReleased)

};
```

Slika 18 Key Released Event

Funkcije

- **KeyReleasedEvent(int keycode, int repeatCount) - konstruktor:** instanciranje KeyboardEvent klase.
- **std::string ToPrint():** ispis puštene tipke s tipkovnice.

KeyTypedEvent

Event za tipkanje po tipkovnici.

```
class VOS_API KeyTypedEvent : public KeyboardEvent {
public:
    KeyTypedEvent(int keycode) : KeyboardEvent(keycode) {}

    std::string ToPrint() const override {
        std::stringstream ss;
        ss << "KeyTypedeEvent" << m_KeyCode;
        return ss.str();
    }

    EVENT_CLASS_TYPE(KeyTyped)
};
```

Slika 19 Key Typed Event

Funkcije

- **KeyReleasedEvent(int keycode, int repeatCount) - konstruktor:** instanciranje KeyboardEvent klase.
- **std::string ToPrint():** ispis pritisnute tipke s tipkovnice.

MouseEvent.h

Svi event-ovi za miš.

MouseMovedEvent

Event za pokret miša.

```
class VOS_API MouseMovedEvent : public Event{
public:
    MouseMovedEvent(float x, float y) : m_MouseX(x), m_MouseY(y) {}

    inline float GetX() const { return m_MouseX; }
    inline float GetY() const { return m_MouseY; }

    std::string ToPrint() const override{
        std::stringstream ss;
        ss << "MouseX: " << GetX() << " MouseY: " << GetY();
        return ss.str();
    }

    EVENT_CLASS_TYPE(MouseMoved)
    EVENT_CLASS_CATEGORY(EventCategoryMouse)

private:
    float m_MouseX, m_MouseY;
};
```

Slika 20 Mouse Moved Event

Funkcije

- **MouseMovedEvent(float x, float y):** postavlja pozicije miša na m_MouseX, m_MouseY varijable.
- **float GetX():** vraća x poziciju miša.
- **float GetY():** vraća y poziciju miša.
- **std::string ToPrint():** ispis pozicije miša.

MouseScrolledEvent

Event za scroll-anje miša.

```
class VOS_API MouseScrolledEvent : public Event {
public:
    MouseScrolledEvent(float x, float y): m_offsetX(x), m_offsetY(y) {}

    inline float GetX() const { return m_offsetX; }
    inline float GetY() const { return m_offsetY; }

    std::string ToPrint() const override {
        std::stringstream ss;
        ss << "MouseScrolledEvent " << GetX() << " by x axis and for " << GetY() << " by y axis.";
        return ss.str();
    }

    EVENT_CLASS_TYPE(MouseScrolled)
    EVENT_CLASS_CATEGORY(EventCategoryMouse)

private:
    float m_offsetX, m_offsetY;
};
```

Slika 21 Mouse Scrolled Event

Funkcije

- **MouseScrolledEvent(float x, float y):** postavlja offset-a (pomaka) scroll tipke miša na m_MouseX, m_MouseY varijable.
- **float GetX():** vraća x offset scroll tipke miša.
- **float GetY():** vraća y offset scroll tipke miša.
- **std::string ToPrint():** ispis offset scroll tipke miša.

MouseButtonEvent

Isto kao i kod event-a tipkovnice ovo je interface za event-ove pritiska miša.

```
class VOS_API MouseButtonEvent : public Event {
public:
    inline int getButton() const { return m_button; }

    EVENT_CLASS_CATEGORY(EventCategoryMouse | EventCategoryMouseButton)
protected:
    MouseButtonEvent(int button): m_button(button) {}

    int m_button;
};
```

Slika 22 Mouse Button Event

Funkcije

- **int GetButton():** vraća gumb koji je pritisnut.
- **MouseButtonEvent(int button) - konstruktor:** postavlja varijablu m_Button na vrijednost pritisnute tipke s miša. Konstruktor je protected zato što ne želim stvoriti instancu klase Keyboard event zato što je uostalom samo interface.

MouseButtonPressedEvent

Event za pritisak gumba miša.

```
class VOS_API MouseButtonPressedEvent : public MouseButtonEvent {
public:
    MouseButtonPressedEvent(int button) : MouseButtonEvent(button) {}

    std::string ToPrint() const override {
        std::stringstream ss;
        ss << "MouseButtonPressedEvent" << getButton();
        return ss.str();
    }

    EVENT_CLASS_TYPE(MouseButtonPressed)

};
```

Slika 23 Mouse Button Pressed Event

Funkcije

- **MouseButtonPressedEvent(int button):** kreira instancu klase MouseButtonEvent().
- **std::string ToPrint():** Ispisuje pritisnutu tipke s miša.

MouseButtonReleasedEvent

Event za puštanje gumba miša.

```
class VOS_API MouseButtonReleasedEvent : public MouseButtonEvent {
public:
    MouseButtonReleasedEvent(int button) : MouseButtonEvent(button) {}

    std::string ToPrint() const override{
        std::stringstream ss;
        ss << "MouseButtonReleasedEvent:" << getButton();
        return ss.str();
    }

    EVENT_CLASS_TYPE(MouseButtonReleased)
};
```

Slika 24 Mouse ButtonReleased Event

Funkcije

- **MouseButtonReleasedEvent(int button):** kreira instancu klase MouseButtonEvent().
- **std::string ToPrint():** Ispisuje puštenu tipku miša.

Window

Skup header-a i file-ova koji sadržavaju sve za upravljanje i rad s prozorima.

WindowMaster.h

Interface za window.

```
class VOS_API WindowMaster {
public:
    using EventCallbackFn = std::function<void(Event&)>;
    virtual ~WindowMaster() {}

    virtual void OnUpdate() = 0;

    virtual unsigned int GetWidth() const = 0;
    virtual unsigned int GetHeight() const = 0;

    virtual void SetEventCallback(const EventCallbackFn& callback) = 0;
    virtual void SetVSync(bool enabled) = 0;
    virtual bool IsVSync() const = 0;

    virtual void* GetNativeWindow() const = 0;

    static WindowMaster* Create(const WindowProps& props = WindowProps());
};
```

Slika 25 Window Master

Funkcije:

- **void OnUpdate():** poziva se svaki frame.
- **int GetWidth():** vraća širinu prozora.
- **int GetHeight():** vraća visinu prozora.
- **void SetEventCallback():**
- **void SetVsync():** uključi ili isključi vsync, uključeno po default-u.
- **bool IsVsync():** vraća je li vsync uključen ili ne.
- **void* GetNativeWindow():** vraća pointer (hrv. pokazivač) na window.

- **WindowMaster* Create(cosnt WindowProps& props):** stvara instancu WindowMaster klase?? Gdje je WindowProps struct koji sadržava podatke o window-u.

```
struct WindowProps {
    std::string Title;
    unsigned int Width;
    unsigned int Height;

    WindowProps(const std::string& title = "VilagOS", unsigned int width = 1280, unsigned int height = 720) :
        Title(title), Width(width), Height(height) {}
};
```

Slika 26 Window Props

Window.h

Nasljeđuje WindowMaster klasu. Sadrži struct WindowData koji je sadržava sve potrebne podatke window-a poput naziva (Title), visine i širine (Height i Width), je li VSync uključen ili isključen (VSync) i event callback funkciju tj. funkcija koja se poziva na neki proslijeđeni event.

```
class Window : public WindowMaster{
public:
    using EventCallbackFn = std::function<void(Event&)>;
    Window(const WindowProps& props);
    virtual ~Window();

    void OnUpdate() override;

    inline unsigned int GetWidth() const override { return m_Data.Width; };
    inline unsigned int GetHeight() const override { return m_Data.Height; };

    inline void SetEventCallback(const EventCallbackFn& callback) override {
        m_Data.EventCallback = callback;
    }
    void SetVSync(bool enabled) override;
    bool IsVSync() const override;

    inline void* GetNativeWindow() const { return m_Window; };

private:
    void Init(const WindowProps& props);
    void Shutdown();

    GLFWwindow* m_Window;

    struct WindowData
    {
        std::string Title;
        unsigned int Width, Height;
        bool VSync;
        EventCallbackFn EventCallback;

    };

    WindowData m_Data;
};
```

Slika 27 Window

Funkcije

- **Window() – konstruktor:** poziva Init funkciju.
- **void OnUpdate():** poziva funkcije glfwSwapBuffers() i glfwPollEvents().
- **unsigned int GetWidth():** vraća širinu prozora.
- **unsigned int GetHeight():** vraća visinu prozora.
- **void SetVSync(bool enable):** postavlja varijablu VSync na vrijednost proslijeđene varijable enable i ako je enable true onda uključi swap-anje (hrv. zamjenu) glfw buffera inače je isključi.
- **bool isVSync():** vraća vrijednost varijable VSync iz WindowData.
- **void* GetNativeWindow():** vraća pokazivač na GLFWwindow što je baš prozor stvoren u Init funkciji.
- **void Init():** postavi vrijednosti u m_Data struct-u. Zatim inicijalizira glfw, stvori novi glfw prozor i kontekst, postavlja VSync na true uvijek tako da je po default-u uključen i postavi callback funkcije za glfwWindowSize, glfwWindowClose, glfwSetKey, glfwSetMouse, glfwSetScroll, glfwSetCursorPosition i glfwSetChar funkcije.
- **void Shutdown():** uništava glfwWindow, tj. zatvori prozor.

WindowMasterInput.h

Nasljeđuje klasu Input. Klasa za čitanje inputa tek kada je input dan tj. za event polling. Koristi glfw library funkcije da prepoznaje kada je dan input i koji je input dan.

```
class WindowMasterInput: public Input{
protected:
    virtual bool IsKeyPressed(int keycode) override;
    virtual bool IsMouseButtonPressed(int button) override;
    virtual std::pair<float, float> GetMousePosition() override;
};
```

Slika 28 Window Master Input

Funkcije

- **bool IsKeyPressed(int keycode):** uzima referencu na aktivni prozor za proslijeđeni keycode gleda je li pritisnuta ta tipka ili se drži i da vrati true inače false.
- **bool IsMouseButtonPressed(int button):** uzima referencu na aktivni prozor za proslijeđeni gumb i ako je gumb pritisnut vrati true inače false.
- **std::pair<float, float> GetMousePosition():** Uzima referencu na aktivan prozor i vraća poziciju miša.

Renderer

Uključuje sve potrebno za crtanje

Renderer.h

Sadrži samo static Renderer klasu koja priprema sve potrebno za crtanje na ekranu.

Funkcije

- **void Init():** poziva Init funkcije od RenderCommand i Render2D.
- **void OnWindowResize(uint32_t width, uint32_t height):** poziva SetViewport funkciju iz RenderCommand.
- **BeginScene(OrthographicCamera& camera):** iz kamere dohvaća ViewProjectionMatrix.
- **EndScene():** does nothing at all.
- **SubmitData(const std::shared_ptr<Shader>& shader, const std::shared_ptr<VertexArray>& vertexArray, const glm::mat4& transform = glm::mat4(1.0f)):** funkcija za lako prosljeđivanje podataka za crtanje. Prvo bind-a proslijeđeni shader, zatim proslijedi shader-u

ViewProjecionMatrix i transform objekta koji se crta, na kraju bind-a vertexArray i poziva DrawElements na taj vertexArray iz RenderCommand.

Renderer2D.h

Apstrahira proces crtanja 2D objekata na window.

```
class Renderer2D {
public:
    static void Init();
    static void Shutdown();

    static void BeginScene(const OrthographicCamera& camera);
    static void EndScene() {};
    static void DrawQuad(const glm::vec2& position, glm::vec2 size, const glm::vec4& color);
    static void DrawQuad(const glm::vec3& position, glm::vec2 size, const glm::vec4& color);
    static void DrawQuad(const glm::vec2& position, glm::vec2 size, const std::shared_ptr<Texture2D>& texture, float tiling = 1.0f);
    static void DrawQuad(const glm::vec3& position, glm::vec2 size, const std::shared_ptr<Texture2D>& texture, float tiling = 1.0f);

    static void DrawRotatedQuad(const glm::vec2& position, glm::vec2 size, float rotation, const glm::vec4& color);
    static void DrawRotatedQuad(const glm::vec3& position, glm::vec2 size, float rotation, const glm::vec4& color);
    static void DrawRotatedQuad(const glm::vec2& position, glm::vec2 size, float rotation, const std::shared_ptr<Texture2D>& texture, float tiling = 1.0f);
    static void DrawRotatedQuad(const glm::vec3& position, glm::vec2 size, float rotation, const std::shared_ptr<Texture2D>& texture, float tiling = 1.0f);
};
```

Slika 29 Renderer2D

Funkcije

- **void Init():** pripremi shader i vertex array za rad.
- **void BeginScene(cosnt OrthographicCamera& camera):**
- **void DrawQuad():** overload-ana funkcija koja može primati ili glm::vec3 ili glm::vec2 i crta na screen quad sa proslijeđenom bojom glm::vec4& color. Ili se umjesto boje može proslijediti pointer na Texture2D objekt ako treba crtati quad s teksturom s opcionalnim parametrom float tiling koji služi za ponavljanje teksture, po defaultu tekstura se ne ponavlja.
- **void DrawRotatedQuad():** isto kao i DrawQuad() samo što ima dodatni parametar za rotaciju u stupnjevima.

RendererCommand.h

Apstrahira neke OpenGL operacije za jednostavnije korištenje.

```
class RenderCommand {
public:
    void  static Init() { //what does this do?
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        glEnable(GL_DEPTH_TEST); //check which pixel is in front or back
    }
    inline static void DrawElements(const std::shared_ptr<VertexArray>& vertexArray) {
        glDrawElements(GL_TRIANGLES, vertexArray->GetIndexBuffer()->GetCount(), GL_UNSIGNED_INT, nullptr);
        glBindTexture(GL_TEXTURE_2D, 0); //clearing 0 texture slot
    }

    inline static void SetViewport(uint32_t x, uint32_t y, uint32_t width, uint32_t height) {
        glViewport(x, y, width, height);
    }

    static void Clear(const glm::vec4& color) {
        glClearColor(color.r, color.g, color.b, color.a);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    }
};
```

Slika 30 Renderer Command

Funkcije

- **void Init():** uključuje blend-anje tekstura i testiranje dubine, tako da blend-anje tekstura normalno funkcionira.
- **void DrawElements():** crta elemente od proslijeđenih točaka (vertexArray) i miče zadnju teksturu.
- **void SetViewport(uint32_t x, uint32_t y, uint32_t width, uint32_t height):** postavlja viewport.
- **void Clear():** postavlja boju za clear-anje screen-a i clear-a screen.

VertexArray.h

Sadrži klasu VertexArray koja enkapsulira sve podatke potrebne za renderer. Operacije izvršava u potpunosti pomoću OpenGl-a.

```
class VertexArray {
public:
    VertexArray();
    ~VertexArray() {};

    void Bind() const;
    void Unbind() const;

    void AddVertexBuffer(const std::shared_ptr<VertexBuffer> vertexBuffer);
    void AddIndexBuffer(const std::shared_ptr<IndexBuffer> indexBuffer);

    inline std::vector<std::shared_ptr<VertexBuffer>> GetVertexBuffers() { return m_VertexBuffers; }
    inline std::shared_ptr<IndexBuffer> GetIndexBuffer() { return m_indexBuffer; }

    inline VertexArray* GetThisVertexArray() { return this; }

private:
    std::vector<std::shared_ptr<VertexBuffer>> m_VertexBuffers;
    std::shared_ptr<IndexBuffer> m_indexBuffer;
    uint32_t m_rendererID;
};
```

Slika 31 Vertex Array

Funkcije

- **VertexArray() – konstruktor:** stvara jedan vertex array za aktivni renderer pomoću.
- **void Bind():** bind-a vertex array.
- **void Unbind():** unbind-a vertex array.
- **void AddVertexBuffer():** dodaje proslijeđeni vertex buffer za vertex array.
- **void AddIndexBuffer():** dodaje proslijeđeni index buffer za vertex array.
- **std::vector<std::shared_ptr<VertexBuffer>> GetVertexBuffers():** vraća listu svih vertex buffer-a. Napomena: ovaj engine trenutno supporta samo jedan vertex buffer.
- **std::shared_ptr<IndexBuffer> GetIndexBuffer():** vraća index buffer za ovaj VertexArray.
- **VertexArray* GetThisVertexArray():** vraća pointer na instancu ove klase.

Buffer.h

Sadrži klase VertexBuffer i IndexBuffer koje služe kao struktura za buffer-e točaka i njihovih indeksa.

OrthographicCamera.h

```
class OrthographicCamera {
public:
    OrthographicCamera() {}
    OrthographicCamera(float left, float right, float bottom, float top); //dont need near and far
    void SetProjection(float left, float right, float bottom, float top);

    glm::vec3 GetPosition() { return m_Position; }
    inline void SetPosition(const glm::vec3& position) {
        m_Position = position;
        RecalculateViewMatrix();
    }

    inline void SetRotation(float rotation) {
        m_Rotation = rotation;
        RecalculateViewMatrix();
    }

    const inline glm::mat4 GetProjectionMatrix() const { return m_ProjectionMatrix; }
    const inline glm::mat4 GetViewMatrix() const { return m_ViewMatrix; }
    const inline glm::mat4 GetViewProjectionMatrix() const { return m_ViewProjectionMatrix; }
    const inline float GetRotation() const { return m_Rotation; }


private:
    void RecalculateViewMatrix();
    glm::mat4 m_ProjectionMatrix;
    glm::mat4 m_ViewMatrix;
    glm::mat4 m_ViewProjectionMatrix; //so that I dont have to do the calculations every time I need this matrix
    float m_Rotation = 0.0f;

    glm::vec3 m_Position = glm::vec3(0.0f, 0.0f, 0.0f);

};
```

Slika 32 Orthographic Camera

Funkcije

- **OrthographicCamera(float left, float right, float bottom, float top) – konstruktor:** računa ViewProjectionMatrix.
- **void RecalculateViewMatrix():** za ponovno računanje ViewMatrix i ViewProjectionMatrix.
- **void SetProjection(float left, float right, float bottom, float top):** promjeni ProjectionMatrix i ponovno izračuna ViewProjectionMatrix.
- **void SetPositon(glm::vec3& position):** postavlja m_Position na proslijeđeni position i poziva RecalculateViewMatrix.
- **glm::vec3 GetPosition():** vraća m_Position.
- **glm::Mat4 GetProjectionMatrix():** vraća m_ProjectionMatrix.
- **glm::Mat4 GetViewMatrix():** vraća m_ViewMatrix.
- **glm::Mat4 GetViewProjectionMatrix():** vraća m_ViewProjectionMatrix.
- **float GetRotation():** vraća m_Rotation.

OrthographicCameraController.h

Upravljanje kamerom.

```
class OrthographicCameraController {
public:
    OrthographicCameraController() {}
    OrthographicCameraController(float aspectRatio);

    void OnUpdate(DeltaTime dt);
    void OnEvent(Event& e);

    inline OrthographicCamera GetCamera() { return m_Camera; }
private:
    bool OnMouseScrolledEvent(MouseScrolledEvent& e);
    bool OnWindowResizedEvent(WindowResizeEvent& e);
private:
    float m_AspectRatio;
    float m_ZoomLevel = 1.0f;
    OrthographicCamera m_Camera;

    glm::vec3 m_CameraPosition = glm::vec3(0.0f);
    float m_CameraRotation = 0.0f;

    float m_CameraMovementSpeed = 1.0f, m_CameraRotationSpeed = 90.0f;
};
```

Slika 33 Orthographic Camera Controller

Funkcije:

- **OrthographicCameraController(float aspectRation):** stvara novu kameru.
- **void OnUpdate(Deltatime dt):** tu se nalaze kontrole za kameru.
- **void OnEvent(Event& e):** dodjeljivanje callback funkcija event-ovima.
- **OrthographicCamera GetCamera():** vraća kameru.
- **bool OnMouseScrolledEvent(MouseScrolledEvent& e) :** callback funkcija za mouse scrolled event. Zoom in i out za kameru.
- **bool OnWindowResizedEvent(WindowResizeEvent& e):** callback funkcija za Window resize event. Resize-a kameru s prozorom.

Texture.h

Sadrži klase Texture i Texture2D.

Texture

Služi kao interface za Texture2D.

```
class Texture {
public:

    virtual ~Texture() = default;
    virtual uint32_t GetWidth() const = 0;
    virtual uint32_t GetHeight() const = 0;

    virtual void SetData(void* data, uint32_t size) = 0;

    virtual void Bind(uint32_t slot = 0) const = 0;
};
```

Slika 34 Texture

Texture2D

Klasa teksture.

```
class Texture2D : public Texture {
public:
    Texture2D(const std::string& path);
    Texture2D(uint32_t width, uint32_t height);
    ~Texture2D();

    inline uint32_t GetHeight() const override { return m_Height; }
    inline uint32_t GetWidth() const override { return m_Width; }

    void SetData(void* data, uint32_t size) override;

    void Bind(uint32_t slot = 0) const override;
private:
    std::string m_Path;
    uint32_t m_Width, m_Height;
    uint32_t m_RendererId;
    GLenum m_OpenGlFormat, m_DataFormat;
};
```

Slika 35 Texture2D

Funkcije

- **Texture2D() – konstruktor:** overload-ani konstruktor. Može primati ili path to teksture ili visinu i širinu teksture. Napravi OpenGl stvari koje treba za teksture.
- **void SetData():** OpenGl stuff.
- **void Bind():** bind-a texture slot za renderer.
- **uint32_t GetHeight():** vraća visinu teksture.
- **uint32_t GetWidth():** vraća širinu teksture.

Shader.h

Sadrži klase Shader i ShaderLibrary.

Shader

Enkapsulira shader file i rukovanje njime.

```
class Shader {
public:
    //Takes source code for vertex and fragment shaders
    Shader(const std::string& name, const std::string& vertexSource, const std::string& fragmentSource);
    Shader(const std::string& filepath);
    ~Shader();

    void Bind() const;
    void Unbind() const;

    void UploadUniformMat4(const glm::mat4& matrix, const std::string& name);
    void UploadUniformVec4(const glm::vec4& color, const std::string& name);
    void UploadUniformVec2(const glm::vec2& texcord, const std::string& name);
    void UploadUniformInt(const int tex, const std::string& name);
    void UploadUniformFloat(const float ft, const std::string& name);

    inline uint32_t GetRendererId() { return m_RendererID; }
    inline const std::string& GetName() { return m_Name; }

private:
    std::string ReadFile(const std::string& filepath);
    std::unordered_map<GLenum, std::string> PreProcess(const std::string& source); // Idk how many files il
    void Compile(std::unordered_map<GLenum, std::string> ShaderSources);
    uint32_t m_RendererID;
    std::string m_Name;
};
```

Slika 36 Shader

Funkcije

- **Shader():** konstruktor: overload-ani konstruktor koji može primati ili put do shader file-a ili zasebno stringove imena, vertexSource kod i fragmentSource kod.
- **void Bind():** pomoću OpenGl-a bind-a shader.
- **voind Unbind():** pomoću OpenGl-a unbind-a shader.
- **UploadUniform**() funkcije:** upload-a proslijeđene podatke u shader.
- **uint32_t GetRendererId():** vraća renderer id.
- **std::string& GetName():** vraća ime shader-a.

ShaderLibrary

Zamišljen kao kontejner za Shader-e.

```
class ShaderLibrary {
public:
    void Add(const std::shared_ptr < Shader >& shader);
    void ShaderLibrary::Add(const std::string& name, const std::shared_ptr<Shader>& shader);
    std::shared_ptr < Shader > Load(const std::string& filepath);
    std::shared_ptr < Shader > Load(const std::string& name, const std::string& filepath);

    std::shared_ptr < Shader > Get(const::std::string& name);
private:
    std::unordered_map < std::string, std::shared_ptr<Shader>> m_Shaders;
};
```

Slika 37 Shader Library

Funkcije

- **void Add(share_ptr <Shader> shader):** dodaj proslijeđeni shader u unordered map shader-a m_Shaders. Overload-ano tako da može primiti i ime shader-a, inače ime postaje ime file-a.
- **void Load():** stvara novi shader i dodaje ga u unordered map shader-a m_Shaders. Overload-ano tako da može primiti i ime shader-a, inače ime postaje ime file-a.
- **share_ptr <Shader> Get(const std::string& name):** za proslijeđeno ime name vraća shader iz m_Shaders.

Game

Za koristiti sve ovo potrebno je u klijentu samo include-at header file VilagOS.h. Ideja je naslijediti klasu Application i stvoriti layer-e. Za svaki layer reći što mora crtati i raditi svaki frame. Sve funkcionalnosti iz engine-a su po meni intuitivne i lako pristupačne, ako nešto nije jasno konzultirati se s ovom dokumentacijom ili me kontaktirati.