TDP005 Projekt: Objektorienterad Programmering

Designspecifikation

Författare
Alexander Stolpe, alest170
Fredrik Jonsén,frejo105

Höstterminen 2014
Version 1.0

Innehållsförteckning

1. Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första utskick av designspecifikationen	141217

2. Detaljbeskrivning klasser

2.1. Player klassen

Player klassen håller reda på det som rör spelaren när play_state är igång. Det kan endast vara en spelare på spelplanen samtidigt. Konstruktorn för en spelare tar in textur för när han står stilla i spelet och en konstant rektangel som bestämmer hans AABB.

Nedan listas klassens funktioner upp med en kort beskrivning. Efter funktionens namn sägs även om den ligger i public eller private i klassen :

reset_y_velocity(), public.

> Hanterar gravitationen för spelaren. När han kolliderar i y-led så sätts hastigheten i y-led till 0 samt tar bort hopp flaggan.

Update(), public.

> Hanterar allting som kontinuerligt skall uppdateras i play state, som t.ex. inmatning från tangentbordet, kollisionshantering osv.

order_player(), public.

> Hanterar inmatning från användaren och översätter den för update funktionen. En spelare kan springa och hoppa med piltangenterna och skjuta med mellanslag.

handle_gravity(), public.

Ökar y hastigheten utefter den globala gravitationskonstanten som tas in som parameter.

set_stunned(), public.

Används när spelaren kolliderar med en fiende. En stun representeras i och med att spelaren inte kan ta emot input under en kort period, utöver detta så knuffas spelaren bort från fienden i x led. Tiden i frames som spelaren är stunnad utgörs utav inparametern time. Velocity_x som är funktionens andra inparameter och ger spelaren en hastighet bort från fienden som kolliderades med.

Fire(), public.

Säger åt spelaren att börja skjuta med sitt nuvarande vapen. När spelaren skjuter skapas projektilpekare som sedan returneras i en lista, detta för att ett av vapnen skapar tre projektiler istället för en. Om spelaren är stunnad eller att vapnet är på ”cooldown”, som används för att vapnen skall ha ett konstant avfyrningsintervall, så sätts projektilerna till nullptr.

Jump(), public.

Sätter jumping flaggan till true för att spelaren inte skall kunna dubbelhoppa och flyga t.ex. Dessutom får spelaren en hastighet uppåt.

randomize_weapon(), public.

Används när spelaren går över en kaffekopp, en låda som ger lite liv och en vapenuppgradering. Det funktionen gör är att man får ett slumpat vapen som ersätter det nuvarande. Vapnet kan inte vara samma som spelaren har för tillfället.

handle_move(), private.

Används för att flytta spelaren på banan. Kallas när en inmatning har registrerats. Sätter spelaren nya position utefter hur han skall röra sig samt kollar om spelaren kan röra sig eller om denne är stunnad.

handle_animation(), private.

Används när spelaren rör sig och hanterar i vilken följd och hastighet som animeringsbilderna ritas ut.

Variablerna för klassen ligger alla i private. Utöver variabler så finns det enum klasser, vektorer , maps samt en strukt för vapen. Nedan förklaras dessa mer i detalj:

- **WeaponName** är en enum klass som innehåller namn av de vapen som spelaren kan använda.
- **Weapon** är en struktur av vapnen som som används. Den innehåller en konstruktor som tar emot texturen för projektilerna som avfyras, hur många projektiler som avfyras, projektilernas horisontella hastighet, hur mycket skada som ges när en fiende träffas, cooldown för hur snabbt vapnet kan skjuta samt bredd och höjd på projektilerna i pixlar. Dessa sparas även som variabler i strukturen.
- **weaopns_** är en map över de olika vapnens egenskaper. Den innehåller vapnens namn från WeaponName, en sträng av namnet på bilden för projektilbilden, int variabler för hur många

projektiler som avfyras, deras hastighet, skada, cooldown samt höjd och bredd på själva projektilen.

- **kJumpVelocity** är en const int är hastigheten som spelaren får i y-led när denne hoppar.
- **frames_since_firing_** är en int som säger hur många frames det var sedan vapnet sist avfyrades.
- **jumping_** är en bool som säger om spelaren hoppar eller inte.
- **stunned_** är en bool som säger om spelaren är stunnad eller inte.
- **stunned_timer_** är en size_t som säger hur länge spelaren är stunnad om kollison med fiende sker.
- **current_weapon_** är av typen Weapon, som säger vilket vapen spelaren har för tillfället.
- animations_ är en vector över de bilder som spelaren använder sig av när denna springer.
- **current_animation_** är en int som används för att få ut vilken animationsbild som används just nu.
- **animation_timer_** är en size_t som håller koll på hur länge den nuvarande animationsbilden har varit aktiv.
- **animation_change_frequency_** är en size_t som säger hur ofta en animationsbild skall bytas.
- **MovementCommand** är en enum klass som innehåller namnet på de kommandon som säger hur spelaren förflyttas.

2.2. GameEngine klassen

GameEngine är den klass som har hand om hur hela programmet körs. Den har variabler som bestämmer hur fönstret skall se ut, skapar alla tillstånd, states, tar in all input via SDL funktioner samt kör programmets while loop där vad som skall göras bestäms utifrån vilket tillstånd som är aktivt.

Nedan listas de funktioner som finns i GameEngine med en kort beskrivning samt om den ligger i private eller public delen av klassen:

GameEnging(), public.

Detta är klassens konstruktor. Den initierar de olika tillstånden samt skapar graphics_engine_ så fönstret skapas.

run(), public.

Funktionen kör hela programmet. Den innehåller loopen som körs för det aktiva tillståndet samt beräknar hur lång tid varje frame har.

handle_input_translation(), private.

Funktionen tar in en tom lista som den fyller med alla inputs från användaren.

handle_state_command(), private.

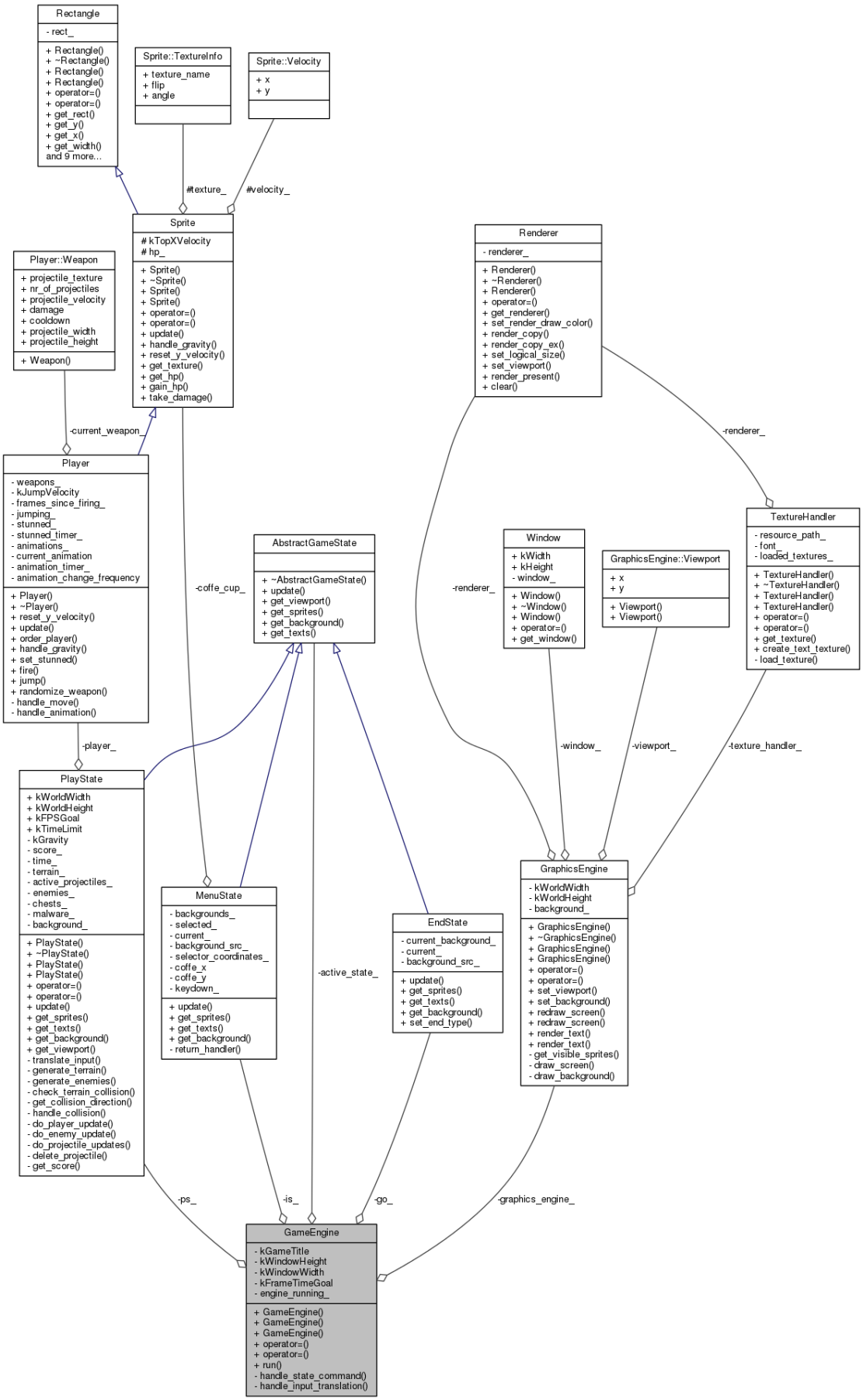
Funktionen hanterar StateCommands som returneras av varje tillstånd efter efter en updatering av denna och byter tillstånd om så är fallet.

Variablerna för klassen ligger alla i private. Nedan följer en lista som kortfattat förklarar varje variabel:

- **graphics_engine_** är av typen GraphicsEngine och används för att rendera skärmen.
- **ps_ är** av typen PlayState och är instansen utav det tillståndet i GameEngine.
- **go_** är av typen EndState och är instansen utav det tillståndet i GameEngine.
- **is_** är av typen MenuState och är instansen av det tillståndet i GameEngine.
- **active_state_** är en pekare av typen AbstractGameState och pekar på det det nuvarande aktiva tillståndet.
- **engine_running_** är en bool som säger om while loopen i run skall köras eller inte. Alltså om spelet körs eller inte.
- **kFrameTimeGoal** är av typen size_t och säger vilket minimum är på tid per frame.
- **kWindowWidth** är en const int och säger vilken bredd det skapade fönstret skall ha.
- **kWindowHeight** är en const int och säger vilken höjd det skapade fönstret skall ha.
- **kGameTitle** är en sträng och sätter titeln för det skapade fönstret.

3. Klassdiagram

Se nästa sida.


Rectangle
- rect_
+ Rectangle()
+ ~Rectangle()
+ Rectangle()
+ Rectangle()
+ operator=()
+ operator=()
+ get_rect()
+ get_y()
+ get_x()
+ get_width()
and 9 more...
Sprite::TextureInfo
+ texture_name
+ flip
+ angle
Sprite::Velocity
+ x
+ y
#texture_
#velocity_
Sprite
kTopXVelocity
hp_
+ Sprite()
+ ~Sprite()
+ Sprite()
+ Sprite()
+ operator=()
+ operator=()
+ update()
+ handle_gravity()
+ reset_y_velocity()
+ get_texture()
+ get_hp()
+ gain_hp()
+ take_damage()
Player::Weapon
+ projectile_texture
+ nr_of_projectiles
+ projectile_velocity
+ damage
+ cooldown
+ projectile_width
+ projectile_height
+ Weapon()
-current_weapon_
Player
- weapons_
- kJumpVelocity
- frames_since_firing_
- jumping_
- stunned_
- stunned_timer_
- animations_
- current_animation
- animation_timer_
- animation_change_frequency
+ Player()
+ ~Player()
+ reset_y_velocity()
+ update()
+ order_player()
+ handle_gravity()
+ set_stunned()
+ fire()
+ jump()
+ randomize_weapon()
- handle_move()
- handle_animation()
-player_
-coffe_cup_
AbstractGameState
+ ~AbstractGameState()
+ update()
+ get_viewport()
+ get_sprites()
+ get_background()
+ get_texts()
PlayState
+ kWorldWidth
+ kWorldHeight
+ kFPSGoal
+ kTimeLimit
- kGravity
- score_
- time_
- terrain_
- active_projectiles_
- enemies_
- chests_
- malware_
- background_
+ PlayState()
+ ~PlayState()
+ PlayState()
+ PlayState()
+ operator=()
+ operator=()
+ update()
+ get_sprites()
+ get_texts()
+ get_background()
+ get_viewport()
- translate_input()
- generate_terrain()
- generate_enemies()
- check_terrain_collision()
- get_collision_direction()
- handle_collision()
- do_player_update()
- do_enemy_update()
- do_projectile_updates()
- delete_projectile()
- get_score()
MenuState
- backgrounds_
- selected_
- current_
- background_src_
- selector_coordinates_
- coffe_x
- coffe_y
- keydown_
+ update()
+ get_sprites()
+ get_texts()
+ get_background()
- return_handler()
EndState
- current_background_
- current_
- background_src_
+ update()
+ get_sprites()
+ get_texts()
+ get_background()
+ set_end_type()
-active_state_
Renderer
- renderer_
+ Renderer()
+ ~Renderer()
+ Renderer()
+ operator=()
+ get_renderer()
+ set_render_draw_color()
+ render_copy()
+ render_copy_ex()
+ set_logical_size()
+ set_viewport()
+ render_present()
+ clear()
-renderer_
-renderer_
Window
+ kWidth
+ kHeight
- window_
+ Window()
+ ~Window()
+ Window()
+ operator=()
+ get_window()
GraphicsEngine::Viewport
+ x
+ y
+ Viewport()
+ Viewport()
TextureHandler
- resource_path_
- font_
- loaded_textures_
+ TextureHandler()
+ ~TextureHandler()
+ TextureHandler()
+ TextureHandler()
+ operator=()
+ operator=()
+ get_texture()
+ create_text_texture()
- load_texture()
-window_
-viewport_
-texture_handler_
GraphicsEngine
- kWorldWidth
- kWorldHeight
- background_
+ GraphicsEngine()
+ ~GraphicsEngine()
+ GraphicsEngine()
+ GraphicsEngine()
+ operator=()
+ operator=()
+ set_viewport()
+ set_background()
+ redraw_screen()
+ redraw_screen()
+ render_text()
+ render_text()
- get_visible_sprites()
- draw_screen()
- draw_background()
-ps_
-is_
-go_
-graphics_engine_
GameEngine
- kGameTitle
- kWindowHeight
- kWindowWidth
- kFrameTimeGoal
- engine_running_
+ GameEngine()
+ GameEngine()
+ GameEngine()
+ operator=()
+ operator=()
+ run()
- handle_state_command()
- handle_input_translation()

4. Diskussion

Till projektet användes SDL, som är ett bibliotek skrivet i C. För att få detta att passa bättre med vårat i övrigt objektorieterade system så valde vi att skriva ”wrapper-klasser” till de viktigaste klasserna; Window, Renderer, Texture och Surface. Vi använder även SDL_Rect som ”container” i vår Rectangle-klass. Vilket vi anser är en relativt bra lösning, då vi inte behövde komma ihåg att använda SDLs ”lösa” funktioner, så som SDL_DestroyTexture, utan kunde istället hantera dem på samma sätt som alla övriga klasser. Nackdelen med detta är såklart det tog lite extra tid innan vi väl kunde börja med själva spelet, eftersom vi först var tvungna att komma fram till exakt hur SDL fungerade och vilka av dess funktioner vi behövde. Vi kände dock ändå att det var värt den extra tiden, för att kunna arbeta med alla klasser på samma sätt, och inte behöva har två olika sätt att göra saker på, beroende på om det var en av våra egna klasser eller något från SDL.

Översiktligt så är vårat system uppbyggt med en GameEngine som driver själva spelet, en GraphicsEngine som hanterar allt det grafiska från SDL, så som fönstret och renderaren, samt tre olika tillstånd beroende på i vilken fas spelet är. GameEngine hanterar både vilket tillstånd som är aktivt, samt grafikmotorn, vilket gör att denna klassen kan ses som våran ”huvudklass”. I denna finns funktionen run, som startar hela spelet. När spelet är igång, körs en loop som ber det nuvarande tillståndet av spelet att uppdatera sig, varpå den returnerar ett StateCommand till GameEngine. Detta StateCommand används exempelvis av menyn för att säga när spelaren har valt ”Start Game”-alternativet, eller när spelaren dog i PlayState och spelmotorn ska gå vidare till EndState. Detta ger väldigt bra abstraktion, eftersom det gör att de tre olika faserna av spelet är helt separerade från varandra. Det ger dock även nackdelen att väldigt mycket data måste skickas med getters och setters via GameEngine, för att tillstånden ska veta vilken information som ska visas.

Eftersom det är PlayState som är själva spelet, är detta tillståndet också mycket större än de andra. Den har mycket information att hantera, och väldigt lite som kan abstraheras till andra klasser. Den är därför rätt svår att få översikt över, men vi anser ändå att det var den mest optimala lösningen för vårat relativt lilla spel. Om vi hade gjort ett större spel, t.ex. om vi haft några veckor till att jobba, hade vi antagligen behövt dela upp själva spelfasen, och då fått fler tillstånd. Möjligtvis hade vi även kunnat göra en klass för själva banan för att hålla reda på saker som terräng, spelaren, monster och annat som är specifikt för bara kartan, men då vi endast hade en karta kändes det inte särskilt meningsfullt att försöka abstrahera ut dessa delar, vi kände att det skulle ha tagit en hel del tid, och vi insåg inte hur stor PlayState-klassen blev förrän i projektets slutskede, när det började bli knappt med tid.

5. Externa filformat

Utöver de vanliga kodfilerna så finns det bildfiler av formatet png som används för att representera spelets grafik. Vi har en ttf font som används när vi skriver ut text på skärmen i systemet, t.ex. poäng, liv, highscore osv. Highscore sparas i en textfil av formatet txt och hanteras av systemet.