Kodgranskning

TDP005: Objektorienterade System

Gruppmöte

Vi kom överens om att granska varandras kod och sedan träffas på labbet den 10 December. På labbet turades vi sedan om att gå igenom varandras kod och kommentera på det som uppmärksammats under den första granskningen. Till slut så lämnade vi över våra anteckningar till varandras grupper, och avslutade mötet.

Kommentarer till grupp thebl297/andan604 (repo)

Struktur

Generellt var det hög kvalitet på koden som jag fick läsa igenom. Strukturen var fin, kod som hörde samman var strukturerade i block, och det var lagom mycket luft i koden så att det var lätt att läsa. Indenteringen var konstant (förutom ett fall i PlayState.cpp: rad 50-54, vilket ser ut som en temporär miss), och gruppen har undvikit breda rader (förutom ett undantag i Player.cpp: rad 12 där konstruktorn är 236 tecken{!!} lång).

Varje header innehöll headerskydd (bra!), fast inkluderingarna kunde vara bättre strukturerade. Det är bra praxis att inkludera i följande ordning:

1. Standardbibliotek
2. Tredjepartsbibliotek
3. Egna lokala filer

Så följande inkluderingar (från Entity.h)
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
#include <map>
#include "block.h"
Bör istället skrivas:
#include <map>

#include <SDL/SDL.h>
#include <SDL/SDL_image>

#include "block.h"

Det fanns inga kommentarer i headerfilerna, vilket kunde behövas, men de som fanns i källkodsfilerna förekom lagom ofta och var mycket hjälpfulla.

Variabler

Generellt var det mycket bra namngivna variabler. När jag exempelvis kollade igenom Player.h så förstod jag allting direkt vid första anblicken, vilket är mycket positivt.

Namngivningen bör bli lite mer konsistent i ett fall, då samtliga privata medlemsvariabler slutar på understreck (exempel keystate_) i samtliga klasser förutom PlayState.h. Förutom detta så var variablerna bra och tydligt namngivna.

Medlemsvariablerna var privata, och åtkomsten skedde genom publika funktioner (bra!) förutom i ett fall, Block.size. Size var en konstant variabel, vilket gör det försvarbart, men generellt bör man hålla variablerna lokala.

I ett fall hade gruppen använt endast stora bokstäver för en icke-konstant variabel, vilket är mot standarden men inte en allvarlig miss.

Metoder

Klassmetoderna följde samma namngivningsstandard (sma_bokstaver_med_understreck), vilket är bra. Jag tyckte dock att namnen kunde vara tydligare.
Exempel:

- ***void Player::angle_GL(bool)***
  - Mycket svårt att förstå
- ***bool Entity::collision_entity(Entity*)***
  - Vad gör denna metod? Kollar om två Entities kolliderar? Modifierar två Entities som kolliderat? Vad innebär returvärdet true eller false?

Bättre exempel skulle kunna vara ***bool Entity::hasCollidedWith(Entity*)*** eller ***Entity::handleCollisionWith(Entity*).***
Jag tyckte också personligen att överlagring skulle skett lite oftare, nu fanns det till exempel två funktioner: ***Entity::collision_entity(...)*** och ***Entity::collision_block(...)*** där en var för klassen Entity och en för typen Block.

Det saknades mycket ***const***-deklareringar bland konstanta metoder. Exempelvis en ***get_texture()***-metod som endast returnerar texture_ får gärna vara konstant.
Det fanns också en del icke-konstanta referenser till privata medlemsvariabler, vilket inte är bra.
Exempel:
Entity hade följande privata variabel: ***SDL_Rect sheet_*** och följande publika metod ***SDL_Rect& get_sheet()***. Ifall man nu skriver ***Entity.get_sheet().w = 4*** så kommer det privata värdet att ändras! En bättre version vore: ***const SDL_Rect &get_sheet() const***

Övrigt

Jag hittade en potentiell minnesläcka i GameEngine.game()
Rad 22: ***SDL_Surface* screen = nullptr;***
Rad 29: ***screen = SDL_SetVideoMode(...);***
Rad 81: ***screen = SDL_SetVideoMode(...);***
Rad 82: ***SDL_Quit();***
Screen från rad 29 hinner aldrig friges innan en ny screen allokeras. Detta är dock inget jag är säker på, utan kan vara ett, för mig, missförstånd i hur SDL_SetVideoMode fungerar.

Det fanns en del fall där gamla C-casts användes istället för C++-casts
Exempel:
PlayState.cpp: rad 75: ***player -> set_speed((short int)3, (short int)0);***
skrivs i c++ ***player->set_speed(dynamic_cast<short int>(3), dynamic_cast<short int>(0));***

Jag var också osäker på vad jag tycker om designen av "Sprite-klassen" Entity.
Entity-klassen hanterar sprites, men Block (som är ett grafiskt block) var inte en del av den klassen, utan endast Player var. Tvärtom så inkluderade Entity klassen Block för att fungera, vilket är lite konstigt.

Kommentarer från grupp thebl297/andan604

Jag fick väldigt bra kritik angående min kod, vilket är en lättnad eftersom jag inte fått någon feedback på koden förut. Jag fick höra att det hade varit mycket svårt att hitta något att klaga på. Följande punkter fick jag, se mina tankar efter:

Varför använder du Random Device när du kan använda True Random

Anledningen till detta är att jag inte riktigt har bra koll på random i C++. Efter lite testande använder jag en random som påverkas av datorklockan, vilket är gör att början av banan får en speciell karaktär (många stjärnor ligger på en rad), vilket jag ser som mer positivt än en sann slumpgenerator.

Kopieringskonstruktorn kan sättas till delete om den inte ska användas

Detta är något helt nytt för mig, och självklart något som jag ska börja använda från och med nu!

Lite lång if-sats i Player.cc: rad 45-47

Typiskt en sådan sak som man blir blind för i sin egen kod, och som jag ska förändra genom att deklarera några extra variabler.

Varför inte använda while(true) istället för för(;;)

Ingen stor skillnad, jag ser for(;;) som mer idiomatiskt.