TDDE71 Seminarie 4

UPP-gruppen

2024-11-11

Space Invaders

Bakgrund

Det nystartade svenska spelföretaget Oskho Games (en hyllning till tidigare kursledare Oskar Holmström) vill lansera en riktig succé. Företaget vill skapa en modern version av det klassiska spelet Space Adventure. De har själva en bra idé av vad som ska vara i spelet, men har dålig koll på hur de ska implementera de olika delarna i spelet - speciellt representationen av alla objekt. Oskho Games vänder sig därför till er, 71:an Consulting, för experthjälp.

Mail från Oskho Games:

Hej 71:an Consulting,

Tack för att ni kan hjälpa oss. Här är en beskrivning av hur vi tänker oss nivå 1 i spelet.

Space Adventure är ett spel där spelaren styr sitt rymdskepp upp/ned piltangenterna. Skeppet flyter framåt (åt höger på skärmen) genom rymden i en bestämd hastighet som inte går att påverka. Vi ser skeppet och rymden omkring ovanifrån. Målet är att ta sig fram och överleva hela banan som slutar med en planet som vaktas av en boss längst till höger.

Längs vägen måste spelaren väja för rymdgrus och aliens. Rymdgrus ligger stilla på samma plats i spelvärlden och försvinner om rymdskeppet kolliderar med det, men det nöter ned skeppets sköld. Om skölden nöts bort helt förlorar spelaren. Aliens pendlar mellan spelvärldens övre och nedre kant. En kollision med en alien nöter mycket på skeppets sköld och knuffar skeppet ur kurs (i samma riktning som alien). Bossen är inaktiv och osynlig tills spelaren kommit tillräckligt nära. När bossen aktiveras släpper den ut en målsökande missil. Missilen rör sig hela tiden mot rymdskeppet.

Om skeppet kolliderar med en missil eller med bossen har spelaren förlorat och spelet är slut. Kolliderar missilen med något annat sker en explosion och båda objekten försvinner. Om skeppet når fram till (kolliderar med) planeten i slutet av banan spelas en fanfar och spelaren har vunnit. Till spelarens hjälp finns statusbar högst upp på skärmen. Där visas en liggande stapel med sköldens status och där finns en nödknapp. Klickar spelaren på nödknappen aktiveras en dödspuls som omedelbar sprider sig över hela spelvärlden. Endast inaktiva objekt, rymdskeppet och planeten överlever en kollision med dödspulsen.

Innan spelet startas möts spelaren av en välkomstskärm med två alternativ: starta spelet och avsluta. Under spelets gång kan spelaren när som helst trycka på “P” för att sätta spelet på paus. Skärmen indikerar att spelet är på paus. Ett nytt “P” gör att spelet fortsätter som om inget har hänt. Vid vinst eller förlust återgår spelet till välkomstskärmen.

Vi behöver förslag på vilka klasser som behövs och hur implementationen skulle fungera med dessa.

Med vänliga hälsningar,
Oskho Games

Objektorienterad analys

1. Finn objekten (leta substantiv)
2. Klassificera objekten (namn, ansvar, samarbeten)
3. Beskriv relationer (Arv eller association)
4. Identifiera aktörer och användningsfall

Utförligare beskrivning av stegen

1. Ett förslag till hur man ska finna objekt är att läsa kravspecifikationen och notera ofta förekommande substantiv.
2. Alla objekt som identifierats i steg 1 ska klassificeras. Detta innebär att bestämma vilka klasser som objekten är instanser av. Beskriv varje klass kortfattat med:

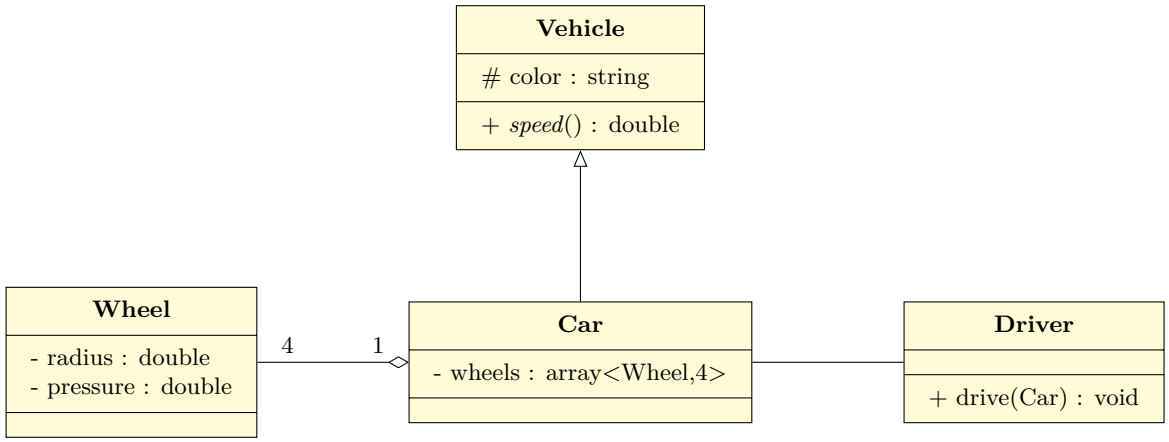
   - klassens namn – välj namn med omsorg! Välj engelska namn som substantiv i singularis.
   - ansvar – operationer som kan utföras på eller av objekt av klassen ifråga.
   - samarbetspartners – vilka andra klasser som klassen samarbetar med för att utföra sina åtaganden.

   Vilka andra objekt/klasser ett objekt behöver känna till kan upptäckas genom att ställa frågor som ”varifrån får objektet denna information”, ”till vem ska objektet leverera denna information”. Ibland kommer kunskaper att representeras av en datamedlem i objekten ifråga, ibland i form av utdata från anrop av metoder i andra objekt.
3. I detta steg bestäms de relationer som finns mellan klasserna i systemet. En relation är antingen arv (x är en specialisering av y), aggregation/komposition (x består av komponenten y) eller association (x kan använda en y).
4. Ett användningsfall är en interaktion som kan inträffa under systemets exekvering. Syftet med att identifiera aktörer och användningsfall och utföra olika scenarier för användningsfall är att få en djupare insikt om samarbetet mellan objekt. Under denna aktivitet kan man till exempel upptäcka nya aktörer och användningsfall, nya objekt/klasser, nya samarbetspartners och nya relationer mellan klasser och objekt. Omvänt gäller att varje objekt/klass/ansvar/samarbete ska ingå i något användningsfall för att dess existens ska vara motiverad.

UML

Exemplet nedan visar hur de viktigaste relationerna ritas med UML.

- `Vehicle` är en basklass med en skyddad datamedlem `color` och en *polymorf* funktion `speed`. Pilen från `Car` till `Vehicle` visar att bilen ärver(specialiserar) fordonet.
- `Car` är en klass härledd från `Vehicle` där 1 `Car` har 4 `Wheel` som privat datamedlem. Diamantpilen från `Wheel` till `Car` visar att hjulen är en komponent som bygger upp bilen och att det behövs 4st hjul för varje 1st bil. Ibland syns detta förhållande med en fylld diamant och kallas då komposition och används när komponenten aldrig existerar självständigt.
- `Driver` har en publik funktion `drive` för att köra en `Car`. Strecket från `Car` till `Driver` visar att föraren kan använda (köra) en bil.



Klasser

Se separat fil med `basegame.drawio.png` klassdiagram.

Game

Håller reda på aktivt spelstate i en stack. Anropar aktivt states `handle_input()`, `update()` och `draw()` varje spelframe tills spelet avslutas. Hanterar eventuellt byte av aktivt state efter varje frame.

Level

Håller reda på alla spelobjekt. Vidarebefordrar anrop av `handle_input()`, `update()` och `draw()` till varje spelobjekt. `update()` anropar även `collides()` plus ev `perform_collision()` på varje objektkombination.

Objekt	**Start X**	**Start Y**	**Misc**
Skeppet	Längst till vänster.	I mitten.	
Grus	25% in från vänster.	I mitten.	
Alien	I mitten.	Längst ned.	Inledande riktning uppåt.
Boss	75% in från vänster.	I mitten.	
Hemplanet	Längst till höger.	I mitten.	
Missil	Framför skaparens objekt.		Rör sig alltid mot Skeppet.
Explosion	Samma som skaparens.		300ms livstid.
Sköld	Längst till vänster.	Högst upp.	Visar nivå på Skeppets sköld.
Nödknapp	Till höger om skölden.	Högst upp.	
Dödspuls	I mitten.	I mitten.	1 frame.

Table 1: Objekten startinformation

Beteenden för objekt i spelvärlden

Objekt	Input	Update	When collide?	Do collide
Skepp	Reagerar på UPP/NED	Flytta ett steg till höger.	Standard.	Minska sköld. Noll-sköld? Missil? Boss aktiv? Spelslut förlust.
Grus	Noop	Noop	Standard.	Försvinner själv.
Alien	Noop	Byt riktning vid kant. Flytta ett steg.	Standard.	Försvinner själv.
Boss inaktiv	Noop	Noop	Trippel storlek.	Aktivera. Lägg till missil. Lägg till aktiv Boss. Försvinner själv.
Boss aktiv	Noop	Noop	Standard.	Dödspuls? Försvinner själv.
Missil	Noop	Flytta närmare skepp	Standard.	Försvinner själv. Lägg till explosion.
Explosion	Noop	Försvinner själv efter 300ms.	Aldrig.	Noop.
Planet	Noop	Noop.	Överlapp X-led.	Skepp? Spelslut vinst!
Dödspuls	Noop	Försvinner själv efter en frame.	Alltid.	Noop.

Table 2: Beteenden för kolliderande spelobjekt

Beteenden för permanenta skärmobjekt (ritas ut sist)

Objekt	Input	Update
Nödknapp.	Klick inom knapp lägger till dödspuls.	Noop.
Sköld.	Noop.	Noop.

Table 3: Beteenden för permanenta skärmobjekt

Användningsfall

Programmet startas

Ett Game-objekt skapas. Game-objektet skapar ett SFML fönster med tillhörande händelsekö, samt ett Menu-objekt som placeras på en tillstånds-stack. Därpå påbörjas spelets huvudloop som på aktivt tillstånd anropar `handle_input()` (för varje händelse i händelsekön), `update()` och `draw()` samt efter detta byter tillstånd vid behov.

Användaren trycker F2 medan programmet visar huvudmenyn

Menu-objektets `handle_input()` reagerar på tangenttryckningen F2 och skapar ett nytt tillstånds-objekt av typen Level. När Level skapas fyller den på sin vektor med spelobjekt med nya objekt enligt deras startposition mm (som är lagrat på fil). Level-objektet placeras så att Game-objektet byter till Level-objektet som aktivt tillstånd. Nästa iteration av Game-objektets huvudloop sker nu på Level. Level-objektets `handle_input()`, `update()` och `draw()` vidarebefordrar anropet till varje spelobjekt i vektorn. Level-objektets `update()` utför även kollisionstestning och hantering av kollision för alla par av spelobjekt.

Användaren trycker F5 medan programmet kör spelbanan (Level-tillståndet aktivt)

Level-objektets `handle_input()` reagerar på tangenttryckningen F5 genom att skapa ett Pause-objekt som placeras så att Game-objektet byter till det nya Pause-tillståndet.

Användaren trycker F5 medan programmet är i paus (Pause-tillståndet aktivt)

Pause-objektets `handle_input()` reagerar på tangenttryckningen F5 genom att signalera till Game att återgå till det tidigare aktiva tillståndet.

Användaren trycker UPP medan programmet kör spelbanan (Level-tillståndet aktivt

Tangenttryckningen UPP plockas från händelsekön i Game, skickas till aktivt tillstånds `handle_input()`, i detta fall Levels. Level skickar händelsen vidare till varje spelobjekt. Spelobjektet `Player` är det enda objektet vars `handle_input` reagera på händelsen genom att modifiera sin position uppåt. Vid nästa anrop av `draw()` kommer den nya positionen synas på skärmen.

Positionen för Player och Alien överlappar (Level-tillståndet aktivt)

Game-objektet anropar `update()` på aktivt tillstånd. Level gör en fullständig kollisionskontroll som resulterar i två kollisioner. Player kolliderar med Alien vilket resulterar i att Player-objektet får hantera att det kolliderat med ett

Alien-objekt. Players `perform_collision()` minskar sin sköldsstyrka. Alien kolliderar med Player vilket resulterar i att Alien-objektet får hantera kollision med ett Player-objekt. Aliens `perform_collision()` markerar att objektet självt ska tas bort från spelet. Efter all kollisionshantering och alla objekt kört sin `update()` placerar Level-objektet alla objekt som ska tas bort sist i vektorn. Därpå minskas vektorns storlek så att rätt objekt försvinner.

Positionen för Player och inaktiv Boss överlappar (Level-tillståndet aktivt)

Game-objektet anropar `update()` på aktivt tillstånd. Level gör en fullständig kollisionskontroll som resulterar i två kollisioner. Player kolliderar med inaktiv Boss vilket resulterar i att Player-objektet får hantera att det kolliderat med ett Boss-Inactive-objekt. Players `perform_collision()` gör ingenting. Boss-Inactive kolliderar med Player vilket resulterar i att Boss-Inactive-objektet får hantera kollision med ett Player-objekt. Boss-Inactive `perform_collision()` markerar att objektet självt ska tas bort från spelet och lägger till dels ett nytt Boss-Active-objekt och ett Missile-objekt i en vektor för nya objekt. Efter all kollisionshantering och alla objekt kört sin `update()` flyttar Level-objektet alla objekt i vektorn för nya objekt till vektorn med aktiva spelobjekt. Nästa frame kommer dessa att börja behandlas.

Simulering

När vi kommit fram till klasserna (eller tillräckligt nära) kan vi göra en simulering av spelet. Ordna fram markörer (t.ex. postit-lappar) för respektive objekt och använd en tavla eller en öppen bordsyta som spelplan. Vi behöver hålla reda på både vilka objekt vi har i vektorerna och var på spelplanen de befinner sig, så det behövs antagligen två markörer per objekt. Begränsa mängden objekt så mycket det går, objekt som ska göra intressanta saker är viktigast att få med och tänka igenom.

Utför sedan spelets huvudloop ett steg i taget.

1. Förmedla varje indata i händelsekön till aktivt tillstånd `handle_input`).
2. Förmedla förfluten tid till aktivt tillstånd (`update`).
3. Förmedla skärmuppdatering till aktivt tillstånd (`draw`).

När Level är aktivt tillstånd:

1. `handle_input` låter varje spelobjekt i vektorn reagera på indata, samt hanter indata Level själv ska reagera på.
2. `update` testar för kollisioner, genomför inträffade kollisioner, uppdaterar varje spelobjekt, tar bort spelobjekt som ska försvinna, lägger till nyskapade spelobjekt.

3. `draw` ritar ut bakgrund, ritar ut varje spelobjekt och ritar sist ut objekt som ska ligga över allt annat.

Flytta runt objekten i vektorer och på spelplanen och tänk igenom vad som ska hända i olika situationer och vilket objekt som ska utföra vilka delar av händelsen.

Tips

Det saknas tips för just mitt problem

Hör av dig och fråga! Har du löst det? Bidra med din lösning!

Ändra parameterlistan till hande_input, update eller draw

Dessa tre funktioner överlagras i **alla** härledda klasser. Att göra en ändring kommer därför innebära att **alla** deklarationer måste gås igenom och ändras. **Alla** filer måste ändras. Det här är ju riktigt tråkigt. Ett viktigt mål med hur vi organiserar koden är att ändringar ska kunna göras på **ett** ställe och påverka så lite annat som möjligt.

Lösningen är att flytta in hela eller en del av parameterlistan i en separat klass eller struct och ta in ett sådant objekt som parameter. Det är denna funktion Context fyller. Context samlar referenser till de objekt som ska skickas som argument till handle_input, update och draw. Sedan skickas det Context-objektet som referensargument till handle_input, update och draw. En ändring i Context påverkar inte deklarationerna av handle_input, update och draw, men kommer samtidigt med till dem.

Det går att upprepa detta mönster. Data som finns i Game och behöver skickas med kan levereras i GameContext, och om Level behöver skicka med egna andra saker kan den skicka med en extra LevelContext. Det är förstås lägligt om behov av detta upptäcks innan alla klasser är gjorda så att det från start är bestämt vilka parameterlistor som behöver den ena, andra eller båda context. Annars landar vi ändå i många ställe att ändra när ett extra Context ska införas.

Långsam inläsning av texturer

Skapa en "cache" av redan inlästa texturer. När cachen ombeds att ta fram en textur hämtas den från cachen om texturen finns där, annars läses texturen först in och placeras i cachen. En `std::map` lämpar sig utmärkt för detta ändamål. Detta löser även problemet att texturer måste finnas kvar minst lika länge som alla sprites som använder den, och garanterar att texturen bara finns i en kopia i minnet.

Spelvärld större än skärmen

Genom att använda SFML vyer är det möjligt att visa bara ett litet utsnitt av spelvärlden på skärmen. Det gör att spelvärlden kan vara mycket större än skärmytan och att saker kan hända utanför skärmen så länge alla objekt i spelvärden hela tiden blir förmedlade vad som händer.

Spelvärld med mycket väggar och andra fasta hinder

Om spelplanen består av många väggar eller likanande bakgrundsobjekt som hindrar spelaren kan dessa samlas i ett kart-objekt med en förenklad representation (t.ex. rutnät) av vart det finns väggar. Vid kollisions-detektering transformeras spelarens koordinater till rad och kolumn i rutnätet. Sedan kontrolleras om det är en vägg där. Detta kan förenkla hur kartor med väggar skapas (det går då att göra kartor med t.ex. ascii-art och läsa in dem). Det kan även minska antalet objekt så kollisionshantering snabbas upp.

Oändlig spelvärld med flera scenarier

Om spelaren hela tiden ska röra sig framåt i en oändligt lång spelvärld går det att hantera genom att bryta upp spelvärlden i lagom långa delar, t.ex. "nuvarande" och "nästa", som båda två är aktiva. Spelarobjektet överförs sömlöst till "nästa" när det når gränsen och när "nuvarande" helt försvunnit ur bild kan det tas bort, "nästa" göras till "nuvarande" och en ny "nästa" läsas in utan att spelaren märker annat än kanske en momentan fördröjning den frame "nästa" läses in. På detta vis kan förberedda "vågor" av spel-scenarier ganska enkelt kombineras ihop i slumpvisa ordningar.

Många uppgraderingar som dessutom kan vara aktiva samtidigt

Om spelet har uppgraderingar och flera uppgraderingar kan vara aktiva samtidigt går det att dekorera spelarobjektet med egenskaper från uppgraderingarna. När spelarobjektet kolliderar med en uppgradering som gör att spelaren skjuter iväg mer potenta missiler under en viss tid skapas ett egenskapsobjekt som får spelarobjektet som datamedlem och ersätter spelarobjektet i spelvärlden. Vid anrop av senare `handle_input()`, `update()` och `draw()` sker då anropet till egenskapsobjektet. Egenskapsobjekt kan göra egna saker både innan och efter det anropar motsvarande funktion i spelarobjektet, eller kanske helt låta bli att vidarebefordra vissa händelse. I detta fall kanske egenskapsobjektet genererar en mer kraftfull missil när CTRL hålls ned men skickar all annan indata till spelarobjektet. Om nu egenskapsobjektet i sin tur kolliderar med en ny uppgradering som ska dubblera hastigheten placeras det gamla egenskapsobjektet som datamedlem i det nya. Det nya egenskapsobjektet ersätter

det gamla i spelvektorn och vid anrop av `update()` ser den nya "yttre" egenskapen till an anropa den gamla "inre" egenskapen två gånger. Den gamla "inre" egenskapen förmedlar i sin tur varje anrop till `update()` i spelarobjektet. Vid anrop av `handle_input()` vidarebefordra även detta till en "inre" egenskapen som fortfarande fångar CTLR och släpper en kraftful missil. När en uppgraderad egenskap ska sluta vara aktiv ersätter vi dess plats i vektorn med spelobjekt med det objekt egenskapen har som datamedlem. Detta är ett *mycket klurigt* men *väldigt kraftfullt* designmönster och mer finns att läsa på `https://refactoring.guru/design-patterns/decorator`.

Långsam kollisionshantering på grund av för många objekt

Om du delar upp spelvärlden i p sektioner *och spelobjekten är ungefär jämnt fördelade över sektionerna* kommer kollisionshantering som är $O(n^2)$ transformeras till $p * O((n/p)^2)$, dvs $O(n^2/p)$. Det tillkommer extra logik att hantera objekt som ligger på gränsen mellan sektioner. Väl skrivet med sektioner som är stora i förhållande till total gränsyta kan det ge upp till p gånger snabbare kollisionshantering.