Av: bjoki902,
danro824

Detaljbeskrivning av Player

Syftet med Playerklassen är att representera det skepp som spelaren styr. Spelarens skepp styrs till skillnad från övriga skepp av spelaren både när det kommer till vilken riktning den kör i och om den skjuter eller ej. Spelarens skepp kan inte heller lämna spelplanen.

Player-klassen ärver av Ship-klassen som i sin tur ärver av Sprite-klassen.

Players konstruktor tar en pekare till en SDL_Surface som representerar skärmen, en pekare till en SDL_Surface som representerar spelarskeppets bild samt två koordinater x och y. Samtliga parametrar anropar Ships konstruktor. I Players konstruktor sätts antalet liv till 3 samt starthastigheten till 0.

Player har en metod, update, som kontrollerar om spelarens nästa position fortfarande är inom skärmens gränser och om den är det så uppdateras positionen. Den kollar även om spelaren har slut på liv, om så är fallet så sätts kill_sign_ till true och spelaren får game over vid nästa kontroll i PlayState.

Från Ship ärver Player följande metoder och variabler:

- ➢ void decreaseHP() - Minskar skeppets hit_points_ med 1. Den kallas på av PlayState om spelaren kolliderar med en fiende eller ett fiendeskott.
- ➢ bool shoot_sign_ - En variabel som representerar om skeppet skjuter eller ej. Den kollas av PlayState för att avgöra om nya skott ska läggas till i skottvektorn.
- ➢ int hit_points_ - En variabel för att hålla koll på hur mycket liv skeppet har kvar. Om denna någonsin når noll sätts kill_sign_ (från Spriteklassen) till sannt vilket gör så att PlayState kallar på changeGameState (som finns i GameEngine) och byter till GameOverState.
- ➢ double shot_x_speed_ - En variabel som anger vilken hastighet skeppet skjuter i x-led. Det är något tveksamt att spelaren har denna då den alltid är 0 (se diskussionen om Shipklassen i “Beskrivning av designen”).
- ➢ double shot_y_speed_ - En variabel som anger vilken hastighet skeppet skjuter i y-led.
- ➢ int score_ - En variabel som anger hur mycket poäng som skeppet ger när det blir nedskjutet. Det är även tveksamt att spelaren har denna.
- ➢ speed_ - Återigen inget spelaren använder då den får sin input från tagentbordet.
- ➢ angle_ - Återigen inget spelaren använder då den får sin input från tagentbordet.

Ships konstruktor anropar Sprites konstruktor med pekaren till screen och pekaren till image. I konstruktorn så sätter den x_ till x och y_ till y. Konstruktorn är protected då vi endast vill skapa instanser av underklasser till ship.

Från Sprite ärver Player följande metoder och variabler:

- bool collides(Sprite const& s) const - Returnerar sant om skeppet kolliderar med s.
- bool outOfScreen() const - Returnerar sant om skeppet är utanför spelplanen. Spelaren använder inte denna då spelaren aldrig kan åka utanför planen.
- void draw - Ritar ut skeppet på skärmen.
- double x_ - X-positionen för det övre vänstra hörnet på skeppet.
- double y_ - Y-positionen för det övre vänstra hörnet på skeppet.
- double x_speed_ - Hastigheten i x-led.
- double y_speed_ - Hastigheten i y-led.
- SDL_Surface* screen_ - En pekare till en skärmyta.
- SDL_Surface* image_ - En pekare till en bildyta.
- bool kill_sign_ - En variabel som avgör om skeppet ska förstöras eller ej vid nästa loopiteration i PlayState. I spelarens fall triggar denna ett byte till GameOverState om den är sann.
- int width_ - Bredden på skeppet.
- int height_ - Höjden på skeppet.

Spriteklassen används till att samla alla egenskaper som alla rörliga objekt har i spelet. Spriteklassens konstruktor sätter screen_ till screen, image_ till image, kill_sign_ till false, width_ till image->w och height till image->h. Konstruktorn är protected då vi endast vill skapa instanser av underklasser till ship.

Spriteklassen inkluderar main.h för att komma åt SDL och vår utritningsfunktion draw_surface.

Av: bjoki902,
danro824

Detaljbeskrivning av GameEngine

GameEngine är den klass som håller koll på vilket stadie spelet befinner sig i. Den innehåller även funktioner som kan påverka alla spelstadier och den kod som ser till att spelet överhuvudtaget körs.

Klassen har en kompositionsrelation till IntroState, MenuState, PlayState, HighscoreState och GameOverState (vilka ärver av AbstractGameState) då den innehåller en vektor med instanser av alla dessa klasser (se UML diagram under “Beskrivning av designen” för en fullständig förklaring av klassförhållandena). De källkodsfiler som hör samman med klassen inkluderar även SDL, SDL_image, SDL_ttf och classes.h (vars syfte är att lösa cirkulärberoenden).

GameEngines konstruktor sätter active_state_ till 0, running_ till true och game_on_ till false. Konstruktorn fyller inte vektorn med spelstadier, utan detta görs i run() innan spelloopen börjar.

Destruktorn frigör det minne som hör samman med alla spelstadier.

GameEngine har följande metoder och variabler:

- void run() - Skapar textfonter, färger, skärmpekaren, event-hanterar och lägger in alla spelstadier i en vektor. Initialiserar även SDL och sätter titeln på fönstret till SPACE RANDOM. Efter att allt som behovs har skapats och initialiserats sätter spelloopen igång, vilken körs 100 gånger i sekunden. Denna körs fram tills något spelstadie kallar på funktionen quit(), vilken sätter running_ till false och avslutar loopen. När loopen avslutats frigörs det minne som allokerats i run().
- void changeGameState(int to_state) - Byter värde på integern aktive_state_ till värdet på parametern. De olika spelstadierna kallar på den här funktionen för att byta spelstadie.
- void callForReset() - Kallar på en funktion i PlayState som ställer om alla värden i PlayState till grundtillståndet för att starta ett nytt spel.
- void initialize() - Kallas på av run() för att initialisera SDL.
- void quit() - Sätter running till false och avslutar spelloopen.
- int active_state_ - ID för nuvarande gamestate. Används i run() för att kalla på draw() och update() funktionerna i rätt stadie. Detta sker genom att run() kallar på funktionerna i fråga i det state som har detta index i vektor. Stadierna har alltid samma index i förhållande till varandra (0 - IntroState, 1 - MenuState, 2 - PlayState, 3 - HighscoreState, 4 - GameOverState).
- bool running_ - Är vilkoret i spelloopen, så länge den är sann så körs spelet. Om den sätts till false bryts spelloopen.

- bool game_on_ - Är sann om ett aktivt spel körs för tillfället. Denna variabel används för att avgöra om knappen “Resume Game” ska visas i menyn eller inte.
- std::vector<AbstractGameState*> states_ - Vektorn där vi sparar de gamestates som finns i spelet.

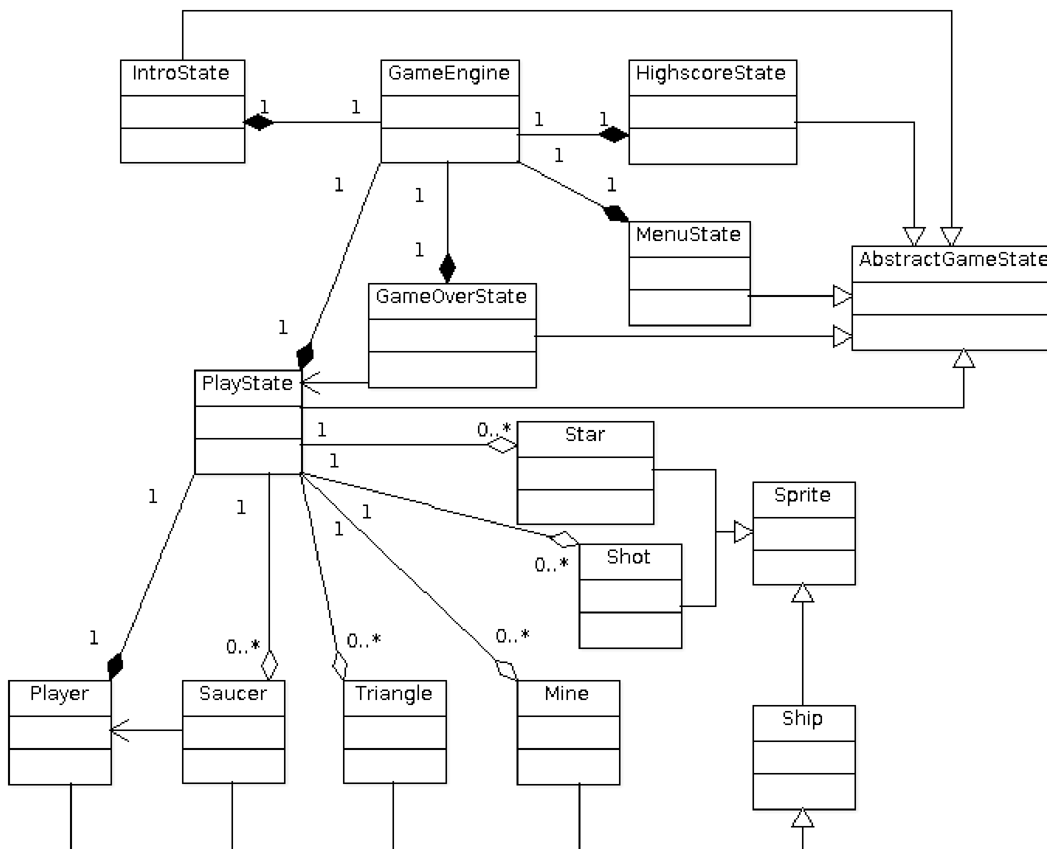
Utöver detta skapas ett antal variabler lokalt i run() som används av alla instanser i spelet. Exempel på dessa är SDL färgerna grön och svart, samt TTF fonts.

Beskrivning av designen

Vår design bygger på en spelmotor som har kontroll över 5 spelstadier, IntroState , MenuState, GameOverState, HighscoreState och PlayState. Stadierna skapas direkt när spelet startas och ligger alltid sparade i en vektor tills programmet avslutas. Detta gör det mycket enkelt att pausa spelet och sedan forsätta exakt där man var innan man pausade. De olika stadierna ärver av klassen AbstractGameState, som innehåller de metoder och attribut som de har gemensamt.

Stadierna har även tillgång till ett antal funktioner i GameEngine (de publika), främst för att kunna byta aktivt spelstadie men även för att kunna hålla koll på ett antal variabler som alla stadier måste ha tillgång till (vilket innebär att de inte kan ligga lokalt i ett stadie). Detta sköts genom att alla stadier innehåller en pekare till den GameEngine som skapats.

En detaljerad beskrivning av förhållandet mellan klasserna kan ses nedan:



Av: bjoki902,
danro824

Stadierna existerar mestadels oberoende av varandra, förutom GameOverState som innehåller en pekare till PlayState. Denna har syftet att ge GameOverState tillgång till PlayStates publika funktioner för att kunna ta fram poängsumman för att kunna spara denna i highscore filen.

Alla Spelobjekt ärver från den abstrakta klassen Sprite som ger dem ett antal gemensamma metoder och attribut. Under spriteklassen finns ännu en abstrakt klass, Ship, som innehåller de saker som delas av spelarskeppen och fiendeskeppen. Denna klass utgör en av designens svagheter då den egentligen kanske borde ha delats upp eftersom den innehåller attribut och metoder som inte används av alla underklasser.

Alla sprites i spelet lagras i PlayState, vilken alltid har samma position i den vektor som lagrar alla stadier i GameEngine. Valet av STL-container är ett resultat av en felaktig åtgärd till ett segmenteringsfel som var ett problem tidigt i utvecklingen. Inledningsvis lagrades alla stadier i en map, vilket är en överlägsen container då vi inte hade tänkt att ändra på den överhuvudtaget. Vi försökte dock lösa problemet genom att byta till en vektor, och den har stannat i programmet även efter att vi kommit på att det var en felaktig lösning då vi helt enkelt haft annat att göra i utvecklingen än att byta tillbaka till en map.

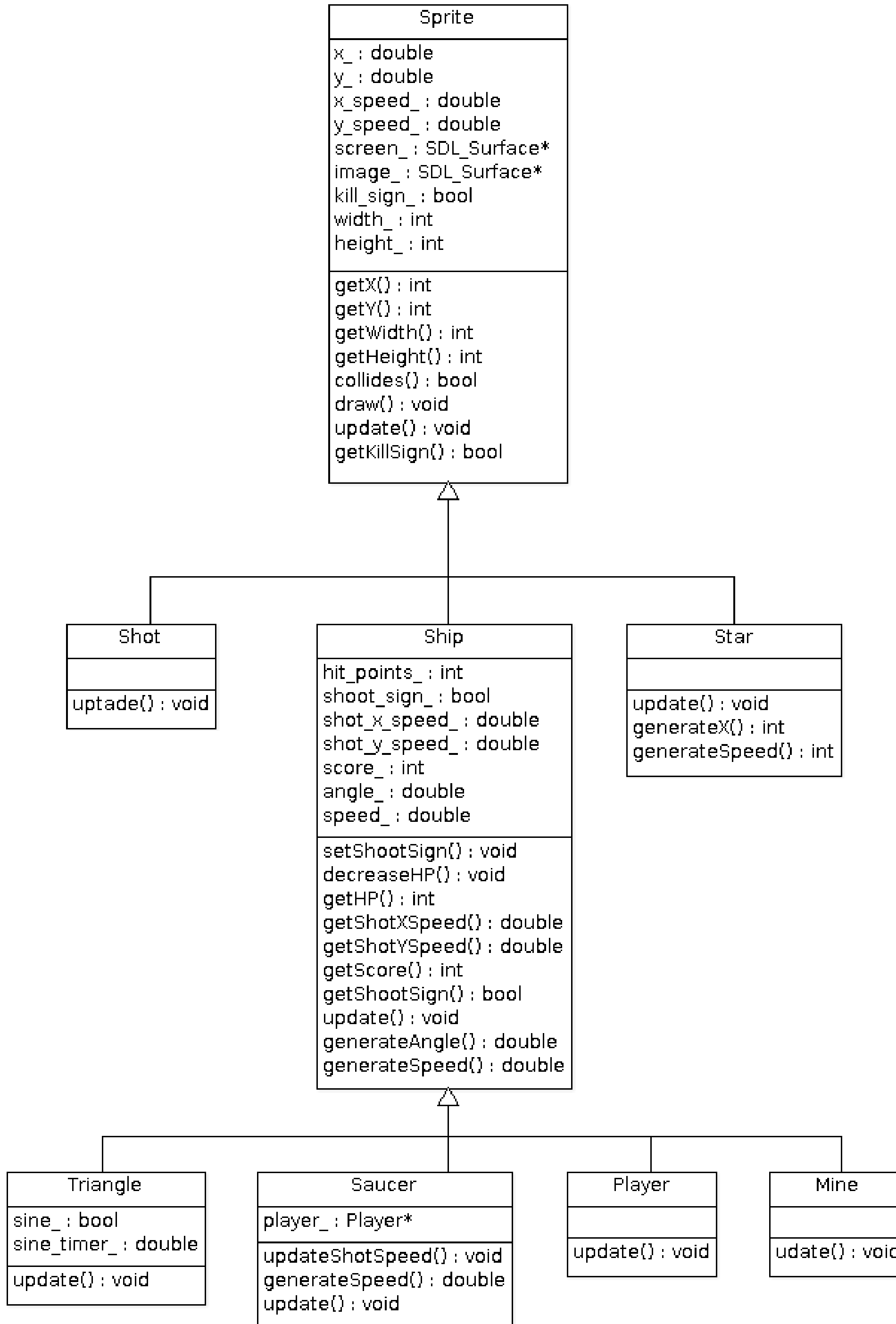
Även de olika Spriteklasserna existerar mestadels oberoende av varandra utom Saucer, som tar en pekare till en Player för att fråga efter dennes position och skjuta mot densamma.

Ytterligare en svaghet i designen är det höga beroendet mellan de olika klasserna. Många klasser är tvungna att ta emot pekare till andra klasser vilket innebär att vissa delar av designen flyter ihop till en viss del. Detta gör så att ändringar blir svårare att genomföra då även en liten ändring kan ha en större effekt.

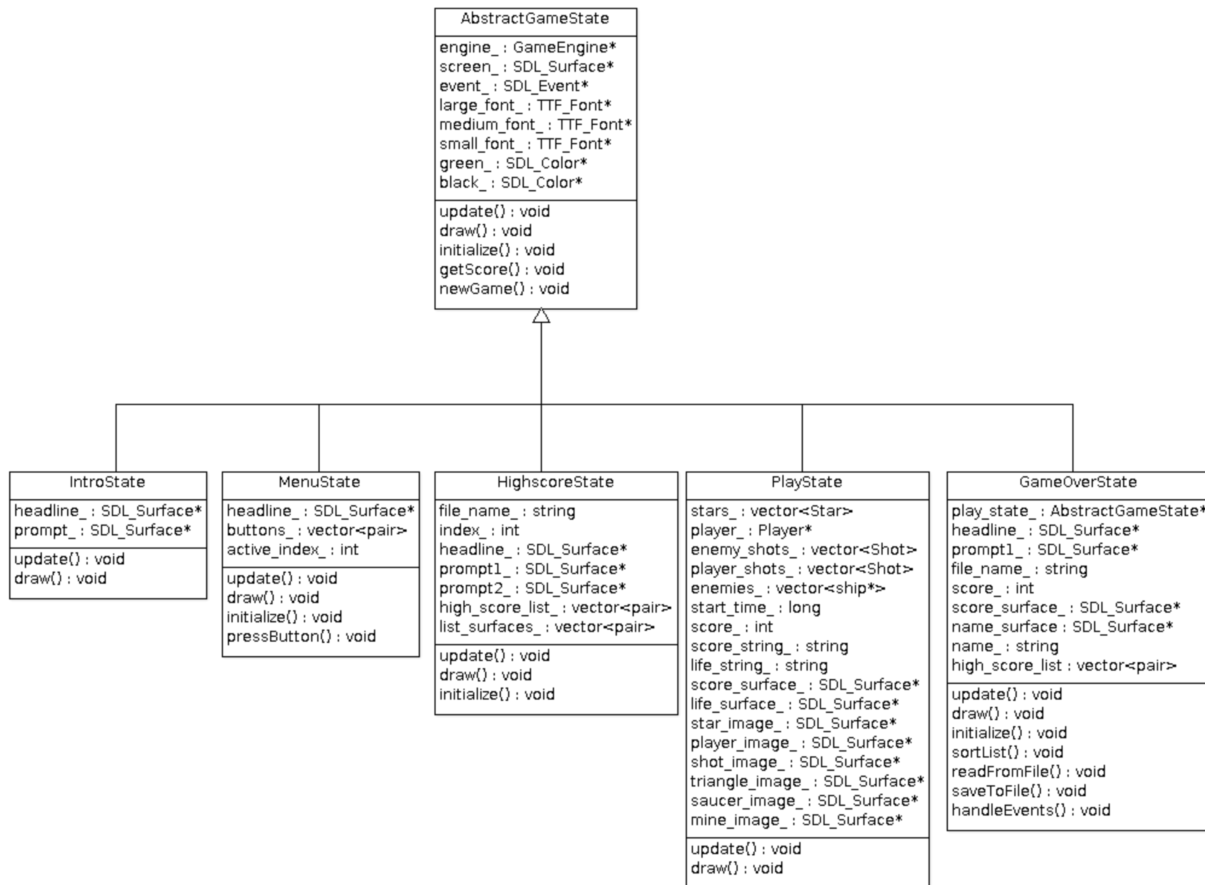
Vissa klasser (till exempel PlayState) är väldigt stora och skulle möjligtvis kunna delas upp i mindre beståndsdelar, vilket hade gjort det hela mer lättöverskådligt.

Av: bjoki902,
danro824

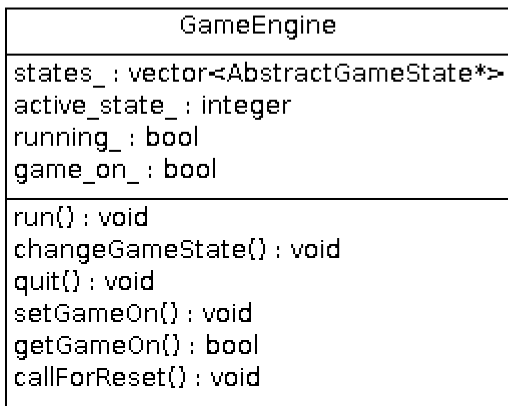
Hur Sprite och dess underklasser är uppbyggda kan ses nedan:



Hur AbstractGameState och dess underklasser associerar till varandra visas nedan:



Innehållet i GameEngine visas nedan:



Externa filformat

Det enda externa filformatet som används i Space Random är helt enkelt en vanlig textfil(.txt) som används för att spara highscore listan. Värdena (poäng och namn) separeras med blanktecken och läses in med formatterad inmatning (samt skrivs till fil med formatterad utmatning). Anledningen att inget mer avancerat använts har varit dels att behovet inte varit så stort och dels att tid inte funnits till att implementera någonting mer avancerat.

En svaghet med detta är att hanteringen av informationen saknar felhantering fullständigt och att det i nuvarande läge kräver att textfilen är strukturerad på precis rätt sätt (alltså att användaren inte gör dumma ändringar i filen).