ludli839, lukvi423

TDP005

I WANNA BE THE SCRUB

Designspecifikation

Detaljbeskrivning av Player

Player-klassen representerar den karaktär som spelaren har kontroll över under spelets gång. Spelaren kan på så sätt få karaktären att hoppa, röra sig i sidled samt skjuta projektiler i x-led. Riktningen av projektilen bestäms av spelarens riktning vid det tillfället som projektilen avfyras, som i sin tur bestäms av vilket håll denne rörde sig senast. Player-objekt har bara ett HP vilket innebär att denne dör direkt då någon slags skada tas.

Player-klassen, likt alla andra utritade objekt i spelet, ärver från klassen GameObject. Då GameObject innehåller en pekare till en Sprite som medlemsvariabel så har även Player en sån Sprite-pekare. Sprite-klassen tar då hand om den grafiska logiken.

Player-konstruktorn tar emot x-position, y-position samt en SDL_Renderer-pekare som krävs för att rita ut spelarobjektet på skärmen. I konstruktorn så kallar vi på metoden idle() som säger vilka frames av dess spritesheet som ska användas när karaktären är stillastående. Ett spritesheet är då en samling av bilder som visar karaktären i olika tillstånd, som t.ex. springandes och stillastående. I konstruktorn specificerar vi även hur stort det utritade objektet ska vara samt sökvägen till dess spritesheet och hur många tillstånd detta spritesheet innehåller. Till slut så kallar vi loadSprite() på den Sprite som Player har tillgång till för att ladda in bilden från sökvägen.

Player ärver följande variabler och funktioner från GameObject:

- void updatePosition()

Uppdaterar det utritade objektets position med objektets hastighet i både x- och y-led.

- void kill()

Uppdaterar en medlemsvariabel som specificerar att objektet ska dö i nästa uppdatering och då tas bort från spelytan.

- void takeDmg()

Sänker objektets hälsa med ett HP eller kallar på kill() om HP är ett. Denna funktion kallas då vid t.ex. kollision med projektil eller spikblock. Då spelaren bara har ett HP så innebär detta att funktionen i spelarens fall då direkt kallar på kill().

- void updateGravity()

Uppdaterar objektets y-position enligt simplifierade principer av gravitet. Detta innebär att spelaren, och andra liknande objekt som fiender, faller mot marken när det inte finns något under att förhindra det.

- Getters för x_vel, y_vel, direction, type och die_next_update.
- Setters för x_vel och y_vel.
- bool intersect(GameObject* const& a, std::array<std::string, 4> &result)

Funktionen för kollisionshantering som kallas på objekt med någon slags hastighet varje uppdatering. Därefter loopas alla objekt på spelytan igenom för att hitta möjlig kollision mellan objektet och objektet med hastighet. De argument som skickas med är en GameObject-pekare som då pekar på objektet som objektet med hastighet kanske kolliderar med. Dessutom skickas en referens till en array med som då ska innehålla eventuella kollisionsriktningar för objektet med hastighet. Kollisionshanteringen består av två delar. Den första delen kollar om det över huvud taget sker en kollision mellan de två objekten. Den andra delen preciserar kollisionen till en av fyra riktningar (top, right, bottom, left) och lägger då resultatet i result-arrayen.

- int direction

Objektets riktning där -1 är vänster och 1 är höger.

- bool die_next_update

Specificerar om objektet ska dö nästa uppdatering.

- double x_vel

Objektets hastighet i x-led.

- double y_vel

Objektets hastighet i y-led.

- int health

Objektets hälsopoäng.

- double airTime

Objektets tid i luften som ökar då objektet inte vidrör något markobjekt. Använder vid gravitationsuträkningen.

- string type

Objektets typ som då kan vara t.ex. “player” eller “projectile”. Används bl.a. vid kollisionshantering för att skilja på kollision mellan olika objekt.

- int x_pos

Objektets x-position vid initiering, t.ex. då spelet startar eller vid omstart.

- int y_pos

Objektets y-position vid initiering, t.ex. då spelet startar eller vid omstart.

- bool collidedThisUpdate

Sant om objektet har kolliderat med ett block eller en border under denna uppdatering. Används för att grafiskt förbättra hur kollision med två objekt samtidigt ser ut.

Players egna funktioner:

- void moveLeft()

Kallas på när spelaren ska röra sig i vänsterled. Funktionen sätter karaktärens hastighet i x-led till ett negativt värde.

- void moveRight()

Kallas på när spelaren ska röra sig i högerled. Funktionen sätter karaktärens hastighet i x-led till ett positivt värde.

- void jump(float deltaTime)

Kallas på när spelaren ska hoppa. Funktionen sätter karaktärens hastighet y-led till ett negativt värde beroende på dess nuvarande y_vel.

- void shoot()

Kallas på när spelaren vill avfyra en projektil. Funktion sätter medlemsvariabeln shoot_next_update till true och skottet kommer då skapas vid nästa uppdatering.

- void handle(SDL_Event event, float deltaTime)

Funktion som kallas på från ett gamestate. SDL_Event innehåller händelser som till exempel knapptryck. Handle() använder SDL_Eventen för att ta reda på vilken av knapparna som trycktes på. Beroende på vilken knapp det var kallas så kallas det på funktioner relaterade till knapptrycket.

- void running()

Sätter start och slut index för karaktärens springanimation. Vid kallelse av funktionen innebär det att karaktären ska röra på sig, och på så sätt visa den tillhörande animationen.

- void idle()

Sätter start- och slutindex för karaktärens stillastående animation.

- void willCollide(std::vector<GameObject*> objects)

Använder intersect() för att ta reda på om det sker en kollision med ett annat denna uppdatering, görs genom att iterera över vektorn med objekt på spelytan. Vid kollision kallas handleCollision().

- void handleCollision(std::vector<std::pair<GameObject*, std::array<std::string, 4>>> collidingObjects)

Funktionen tar emot en vektor med par av GameObject-pekare samt en sträng-array. Vektorn innehåller pekare till objekten som detta objekt har kolliderat med. Sträng-arrayen innehåller kollisionsriktningen för det objektet som den är parad med. handleCollision() bestämmer vad som ska hända vid kollision med ett specifikt objekt, beroende på kollisionsriktningen.

- void update(float deltaTime)

Funktionen som kallas på från ett GameStates uppdateringsloop, denna update funktion bestämmer vilka funktioner som ska kallas på vid varje update.

- void createObject(std::vector<GameObject*> &map_objects)

Funktionen tar emot en referens till en vektor med GameObject pekare som representerar alla objekt på spelytan. Den kallas på när ett objekt, t.ex. spelaren, vill avfyra en projektil. Funktionen skapar objektet och lägger in dess pekare i map_objects vektorn.

Detaljbeskrivning av PlayState

PlayState är den klass som representerar det huvudsakliga stadiet av spelande där spelaren rör på karaktären, skjuter projektiler, osv. PlayState innehåller funktioner som har hand om banskapandet, objektskapande, uppdatering av alla objekt, event-hantering och upprensningen vid spelavslut. PlayState innehåller en vektor med GameObject-pekare som representerar de objekt som finns på spelytan. Dessa GameObject skapas vid spelstart enligt den information som finns i en extern level-fil.

PlayState ärver från den abstrakta klassen GameState som då representerar de olika stadierna av spelande.

PlayState-konstruktorn sparar SDL_Window pekare och SDL_Renderer pekare från StateMachine till medlemsvariabler för att kunna använda de vid utritningen av objekt.

- void init()

Åkallar funktionen loadLevel() för att skapa objekten som ska finnas på spelytan.

- void loadLevel(std::string level);

Tar emot en sträng med sökvägen till en fil som specificerar objekt som ska ritas ut samt dess position. Sedan loopar den igenom denna fil rad för rad och kallar på funktionen createObject() med information från den inlästa filraden.

- void createObject(std::string name, int x, int y, SDL_Renderer* renderer);

Tar emot information som loadLevel() läste in från filen, och utifrån denna information skapar createObjects() objekten i fråga. Sedan läggs pekare till dessa objekt in i en vektor som ska innehålla alla spelytans objekt.

- void cleanup();

Åkallas från StateMachine när det är dags för spelet att avslutas och då avallokeras minnet som spelobjektet använde. Görs genom att iterera igenom objektvektorn.

- void update()

Update står för att uppdatera allt på spelytan som ska förändras. Den står även för att rensa den tidigare ritade skärmen.

Uppdateringen av varje objekt sker genom att iterera igenom vektorn med alla spelobjekten och kallar på antal av dess funktioner. t.ex. om objektet har en hastighet i x-led eller en hastighet i y-led så kallas objektetets willCollide() funktion för att se om en kollision sker. Den kallar även på createObject för att skapa relevanta objekt (t.ex. projektiler), update() för att uppdatera alla dess variabler och värden, draw() för att rita ut på skärmen och eventuellt ta bort objektet om det ska dö.

- SDL_Window* window
- SDL_Renderer* renderer

- std::vector<GameObject*> objects

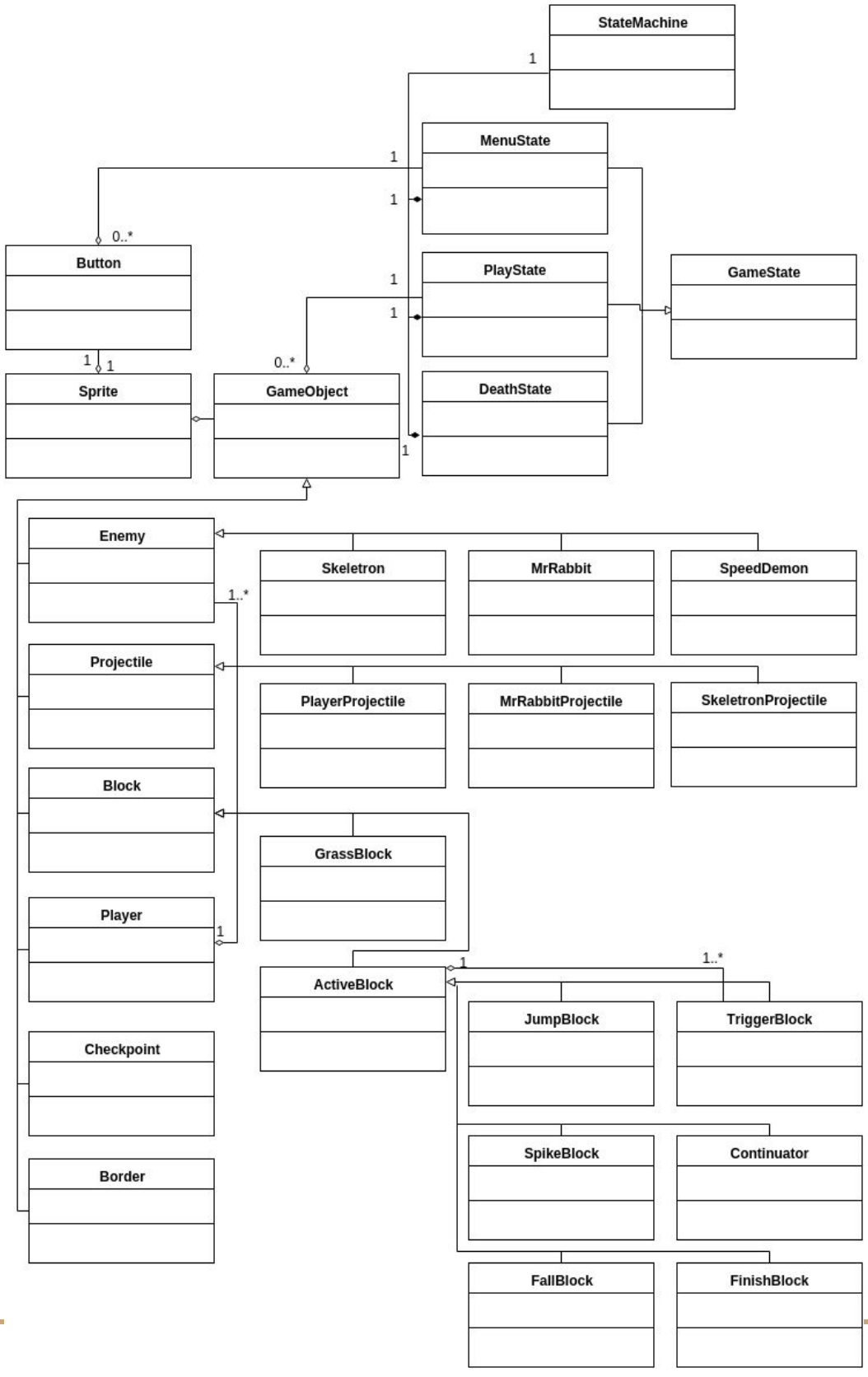
En vektor där alla objekt som finns på spelytan samlas.

Beskrivning av designen

Vår design bygger på idén att hela spelet styrs av det vi kallar en StateMachine. StateMachine har hand om initiering av så kallade GameStates samt byte mellan dem. GameStates är då t.ex. spelstadiet eller menystadiet. Beroende på vilket stadie som är aktuellt för tillfället så vidarebefordras de nödvändiga variablerna och de events som krävs för att styra t.ex. spelarkaraktären eller menyn.

Våra GameStates har hand om logiken som är relaterad till det stadiet av spelande som vi är i för tillfället. Här ligger uppdateringsloopen för alla objekt som existerar på spelytan samt logiken som bestämmer hur events ska hanteras. I PlayStates fall så är det Player-klassen som hanterar events och ser till att de exekveras för spelarobjektet.

Översikt av förhållandet mellan våra klasser (nästa sida):


StateMachine
1
MenuState
1
1
0..*
Button
PlayState
GameState
1
1
1
1
0..*
Sprite
GameObject
DeathState
1
Enemy
Skeletron
MrRabbit
SpeedDemon
1..*
Projectile
PlayerProjectile
MrRabbitProjectile
SkeletronProjectile
Block
GrassBlock
Player
1
ActiveBlock
1
1..*
JumpBlock
TriggerBlock
Checkpoint
SpikeBlock
Continuator
Border
FallBlock
FinishBlock

Spelobjeketen ärver alla från klassen GameObject som innehåller den gemensamma logik som då är positions- och hastighetsuppdatering, skadehantering, gravitationsberäkning, kollisionshantering samt dödshantering. Varje GameObject har även en pekare till ett eget Sprite-objekt som då används för att ladda in samt rita ut den relevanta bilden till skärmen.

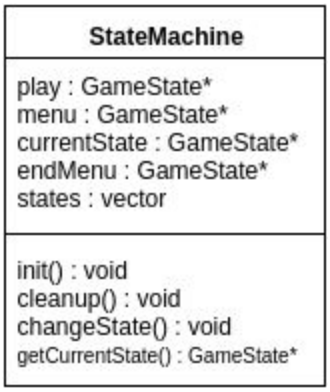
Vår generella design gällande olika typer av ett objekt är att vi har en överklass, t.ex. Block och Enemy, som agerar mall åt de mer specifika objekten, t.ex. Grass Block och Speed Demon. Detta gör vi för att abstrahera ordentligt och få en pålitlig struktur på alla våra klasser.

Tankar kring designen

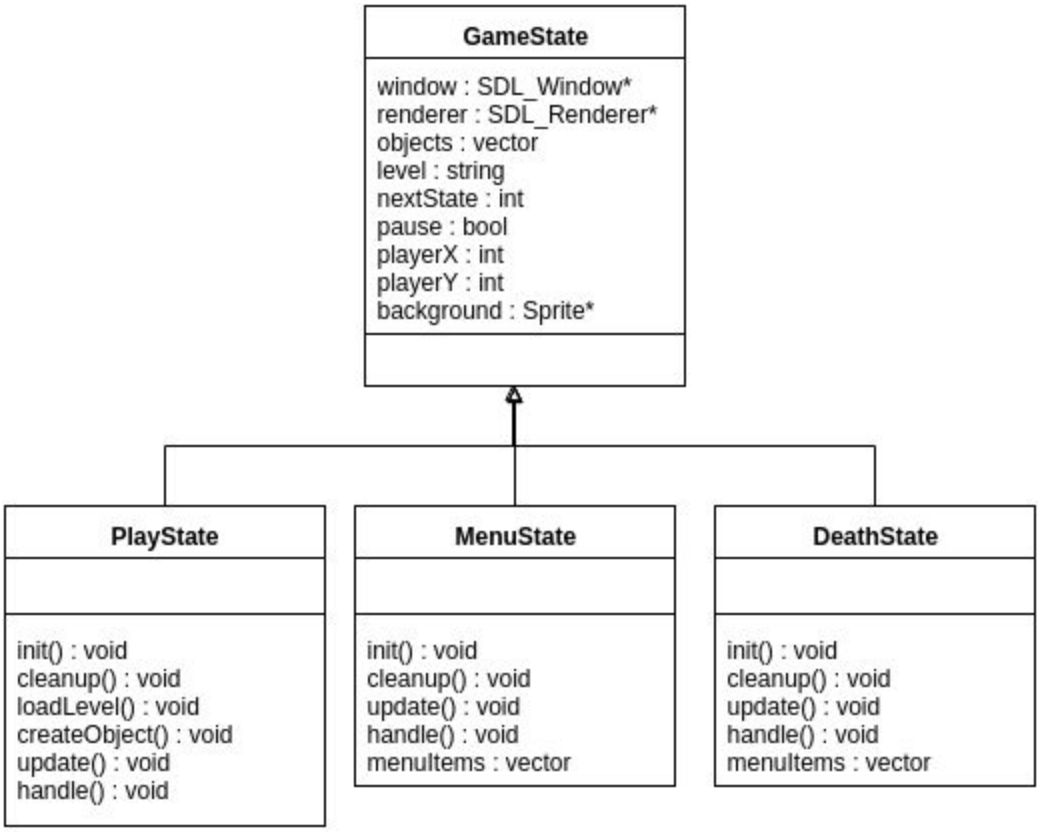
- Då vi använder en vektor för att lagra pekare till alla objekt på banan så uppstår det potentiella problem med storleksallokering. Då en vektor vanligtvis bara har plats för 16 objekt så måste vi manuellt expandera vektorn i källkoden för att kunna hantera de objekt vi skapar i våra banor. Detta gör att vi allokerar minne som vi kanske inte använder.
- Strukturen som den ser ut nu är oeffektiv vid vissa ställen då vi har haft problem med arv och hur vi ska lösa det i specifika situationer. T.ex. finns det en funktion vid namn willCollide() som egentligen ser likadan ut i alla GameObject-subklasser men trots det så lyckades vi inte implementera den i GameObject utan behövde duplicera den i subklasserna. Likt det så har vi även tomma funktioner i vissa klasser då alla GameObjects funktioner inte används av alla subklasser.
- Ett problem som är relevant till hel vår kodbas är kopplingen mellan funktioner. Så som spelet är programmerat så är många funktioner starkt beroende av andra vilket ibland gör det svårt att göra små förändringar utan att behöva göra de på andra ställen också.
- Vi har inte haft tid att optimisera menyn för förändring av fönsterstorlek vilket gör att den utritade knappen och det faktiska området man ska klicka på inte stämmer överens. Detta gör att det svårt att navigera menyn med vissa fönsterstorlek.

Nedan följer bilder av våra klasser, subklasser och deras funktioner samt variabler.

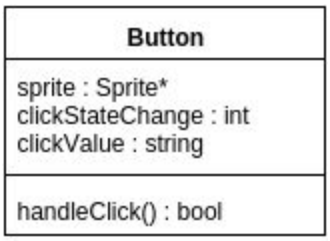
StateMachine



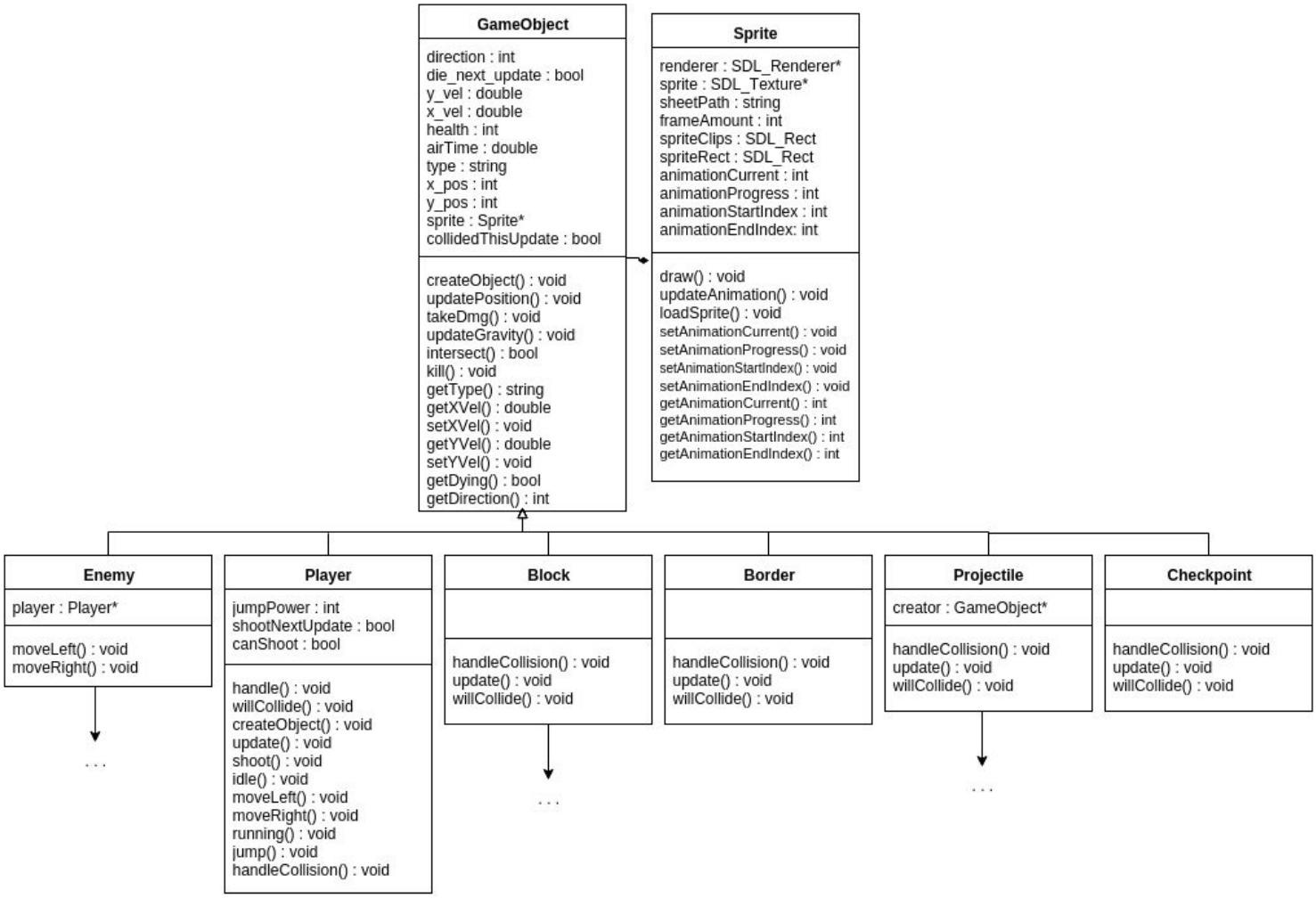
GameState och dess underklasser:



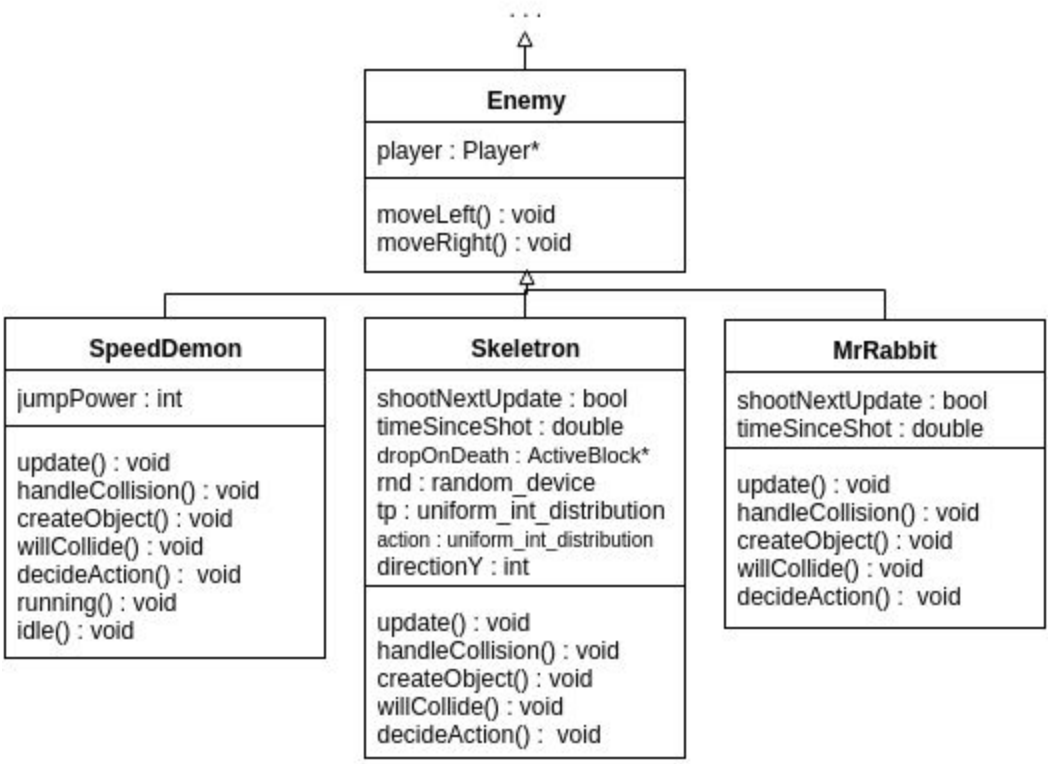
Button



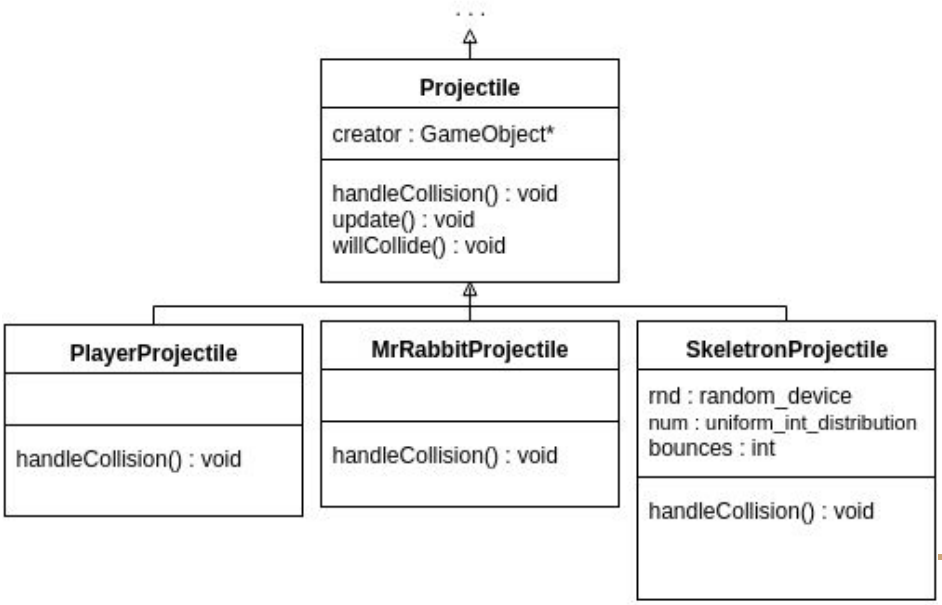
GameObject och dess underklasser samt Sprite (kompakt):



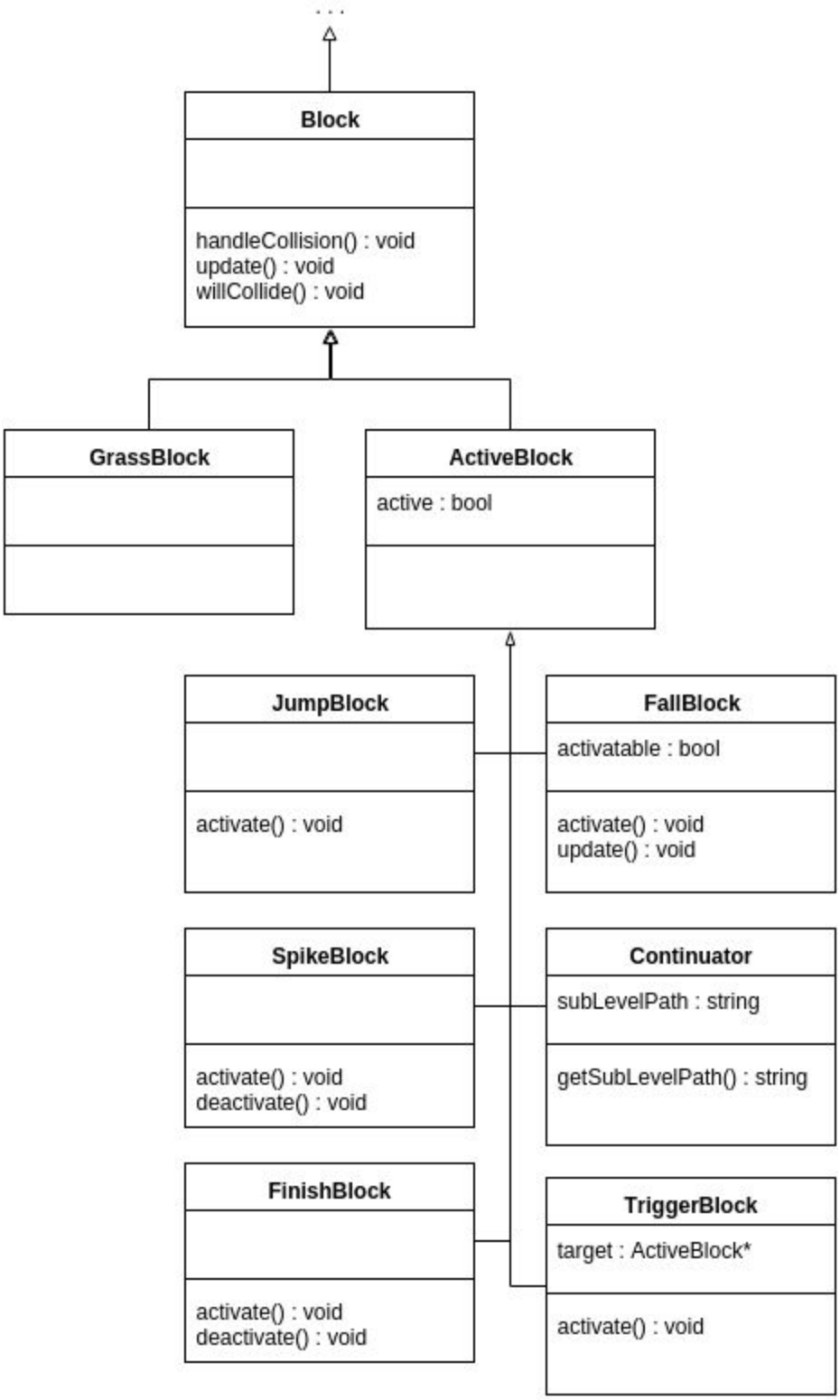
Enemy och dess underklasser:



Projectile och dess underklasser:



Block och dess underklasser:



Externa filformat

Det externa filformatet som vi använder är av typen .lvl. Detta är i princip en .txt fil som representerar de objekt som ska ritas ut på skärmen. Olika spelytor har olika .lvl filer. Så som inladdningen av filen ser ut för tillfället så litar vi för det mesta på att användaren skriver in rätt data och följer det angivna formatet. Om användaren specificerar en klass som inte finns så kommer de bli varnade men inladdningen fortsätter som vanligt genom att strunta i raden med ogiltiga värden.

Inkorrekt data i filen skapar dock problem med mer avancerad funktionalitet som t.ex. TriggerBlock då de använder vad man skulle kunna kalla “GoTo” satser för att bindas till andra objekt. Detta kan orsaka spelkrascher.