Nomad

Nicky van Urk

(Realisatie)

Inhoudsopgave

Inleiding

In dit document wordt mijn aanpak beschreven hoe fouten in mijn code worden opgelost. Tevens beschrijf ik twee programmeer concepten die door mij zijn geïmplementeerd in Nomad.

Realisatie

Voordat ik begin met programmeren zorg ik er eerst voor dat ik een goed beeld heb van wat er moet worden gerealiseerd. Een groot deel van het spel moet van te voren daarom al duidelijk zijn. Een spel hoeft niet tot in de kleinste details van te voren worden uitgewerkt, sommige dingen moet je simpelweg testen in de praktijk. Het idee wordt in kaart gebracht door een ontwerp document te schrijven. Dit document dient als referentie materiaal gedurende de realisatie periode.

Het ontwerp document wordt gebruikt om het programma in grote lijnen te kunnen uitstippelen. Tijdens deze periode wordt er zorgvuldig nagedacht over hoe allerlei ideeën daadwerkelijk kunnen worden geïmplementeerd. Een programma zoals een computerspel valt natuurlijk op te splitsen in veel kleinere concepten. Tijdens het bouwen maak in gebruik van een methode genaamd 'bottom-up design' dit wil zeggen dat ik eerst een module van het spel ontwikkel om deze hierna vervolgens uit te breiden met nog een module. Het programma begint klein maar naarmate de tijd vordert groeit deze in complexiteit en volledigheid.

Tijdens het bouwen van de applicatie wordt er gedebuged bij elke verandering in de code die plaatsvind. Dit wil zeggen dat er wordt gekeken of het programma fouten bevat en zo ja waar deze fouten zich bevinden zodat de ik oftewel de programmeur deze kan herstellen. Als er geen fouten zijn gevonden zal het programma worden gebouwd waarna ik deze zal uitvoeren. Hierna kijk ik of de aanpassing werkt zoals ik het voor ogen heb. Tijdens het testen probeer je eigenlijk altijd de code te breken. Zo kom je achter je logica fouten, deze fouten worden niet gedetecteerd door de debugger omdat hier de syntaxis niet incorrect is maar de logische werking ervan. De applicatie zal hierdoor onvoorspelbaar gedrag vertonen. Door frequent te testen verklein je de kansen op fouten en maakt het je programma betrouwbaarder.

Als de applicatie is voltooid zal ik deze grondig testen. Dit houdt in dat ik de applicatie in het geheel controleer probeer te breken, ik zal zo veel mogelijk handeling uitproberen en zo de applicatie op alle mogelijkheden testen. Ook wordt de code onder de loop genomen om te kijken wat hier nog valt te verbeteren en eventuele fouten op te lossen. Als ik helemaal tevreden ben over de applicatie pas dan is deze voltooid en wordt de bijbehorende documentatie verder uitgebreid.

Concept

Nomad wordt gerealiseerd door middel van een concept genaamd tile-based. Dit wil zeggen dat het spel wordt opgebouwd uit allemaal kleine tiles oftewel tegels. Tiles hebben over het algemeen precies dezelfde breedte als hoogte en zijn daarom dus vierkant. Dit is echter niet altijd het geval. In Nomad heb ik het gehouden bij de klassieke vierkante tiles. De gehele functionaliteit wordt rondom dit concept gebouwd en heet daarom tile-based.

Implementatie

Het implementeren van een tile-based computerspel is doorgaans vrij eenvoudig. Men maakt simpelweg een variabel aan met een tile grote. Vervolgens wordt de gehele functionaliteit rondom dit variabel gebouwd. In het geval van Nomad is deze tile grote 16 waar het getal staat voor het aantal pixels horizontaal en verticaal. De reden waarom ik een tile grote van 16x16 heb gekozen is omdat zo de stijl van het spel simpel blijft en er een retro gevoel aan het spel wordt gegeven.

De implementatie begint met het extern inladen van de level grote (in tiles) en level data. Deze data bestaat uit getallen. Elk getal staat voor een unieke tile. Vervolgens wordt deze uitgelezen data opgeslagen in een vector oftewel een dynamische array. De level grote wordt apart gehouden. Door de variabelen te combineren kan er een tile-based spel worden geïmplementeerd.

We beginnen met de weergave van een level. Hiervoor heb ik een class aangemaakt genaamd: TileMap. Behalve het weergeven van een level doet deze class niets anders. Ik zal van deze class een voorbeeld geven in pseudocode. Pseudocode betekend dat de code op zo'n manier wordt weergeven dat het meer op een gewone taal lijkt dan een computer taal en dus gemakkelijk te begrijpen is.

```
Class TileMap {
   functie load(tileSize, tiles, width, height)
   functie render()

   variabel _vertices;
}
```

De functies spreken voor zich. Hieronder staan de argumenten van de functie load uitgelegd en tevens de enige variabel in deze class.

tileSize: de grote van de tiles.
tiles: de vector met daarin de level data.
width: de breedte van een level uitgedrukt in tiles.
height: de hoogte van een level uitgedrukt in tiles.

_vertices: een soort array (vertexarray) waarin vertices in worden opgeslagen, een vertex is een knooppunt en bestaat uit een x en y positie.

Op de volgende pagina staat de class definitie uitgelegd.

De load functie heeft als doel vertexen te construeren op een bepaald punt om deze vervolgens te weergeven.

```
Functie load(tileSize, tiles, width, height) {
   zet de vertexarray naar het type quads (vierkant).
   vergroot de vertexarray naar width * height * 4.

   for (i = 0; i < width; ++i)
      for (j = 0; i < height; ++j)
         tile_nummer = tiles[i + j * width]

         if (tile_nummer is gelijk aan 0)
            continue

         pointer quad = &_vertices[(i + j * width) * 4]

         quad[0].positie = positie( i * tileSize,       j * tileSize)
         quad[1].positie = positie((i + 1) * tileSize,  j * tileSize)
         quad[2].positie = positie((i + 1) * tileSize, (j + 1)* tileSize)
         quad[3].positie = positie( i * tileSize,      (j + 1) * tileSize)
}

functie render() {
   teken _vertices naar het scherm
}
```

Dit is in feite de TileMap class. Vanaf dit punt kan er voor worden gekozen een extra argument mee te geven bijvoorbeeld een afbeelding en die afbeelding te gebruiken in combinatie met de vertices. Ik heb er echter voor gekozen om het simpel te houden en een array met kleuren mee te geven als extra argument. Deze kleuren worden aan de vertices gegeven afhankelijk van het nummer in de level data (tiles)

Voorbeeld:

```
quad[0].kleur = kleuren[tile_nummer];
quad[1].kleur = kleuren[tile_nummer];
quad[2].kleur = kleuren[tile_nummer];
quad[3].kleur = kleuren[tile_nummer];
```

dit wordt vervolgens geplaatst onder het stuk code wat de quad positie bepaalt.

Nadat we het level hebben kunnen tekenen naar het scherm implementeer ik de entiteiten. De entiteiten in Nomad bestaan uit de speler, monsters en schatkisten. We laden wederom externe data in, de positie en grote van de entiteiten. De ingeladen data plus een kleur worden als argumenten meegeven aan de entiteiten object. Vervolgens construeert dit object een vierkant gebaseerd op de meegegeven data. Deze worden vervolgens naar het scherm getekend. Let wel dat eerst het level wordt getekend en dan de entiteiten. Zo worden de entiteiten als het ware over het level getekend en zijn deze altijd goed zichtbaar. Ook is het belangrijk om van de entiteiten eerst de schatkisten te tekenen gevolgd door de monsters en tot slot de speler. Op deze manier wordt voorkomen dat de speler achter een schatkist of monster terechtkomt.

Nadat alles wordt getekend naar het scherm implementeer ik de besturing van de speler en daarmee ook de beweging. Op de volgende pagina zal het gaan hebben over de collision detection.

Collision

Concept

Zoals in het ontwerp document te zien is (in-game wireframe) heb de collision opgesplitst in vier losse stukken. Eerst controleer ik of de speler collision heeft met het level oftewel de tiles. Hierna doe ik hetzelfde voor alle monsters. Als laatst controleer ik of de speler contact maakt met een monster of schatkist. Deze controle wordt gedaan door te controleren of een entiteit met een tile of andere entiteit overlappen en vervolgens acties te ondernemen afhankelijk van de uitkomst.

Implementatie

De collision met het level wordt wederom gebaseerd of de vector met level/tile data. Ik heb het op de volgende manier geïmplementeerd:

(de entiteit positie wordt voor collision berekent vanaf de linkerbovenhoek van het object)

int tx = entiteit positie van coördinaten naar tiles op de x as
int ty = entiteit positie van coördinaten naar tiles op de y as

bool nx = true als entiteit met de tile rechts overlapt
bool ny = true als entiteit met de tile beneden overlapt

De cell variabel slaat het nummer op van de tile waar de entiteit zich op dat moment in bevind. Hier wordt ook weer de breedte van het level gebruikt (in tiles).

*int cell = tiles[tx + ty * levelWidth]*

Vervolgens worden ook de tile nummers rondom de cell opgeslagen.

```
int cellRight  = _tiles[(tx+1)+ ty      * levelWidth];
int cellDown   = _tiles[ tx   +(ty+1)   * levelWidth];
int cellDiag   = _tiles[(tx+1)+(ty+1)* levelWidth];
```

Door middel van al deze variabelen kunnen we de collision gaan controleren. Als eerste doen we dit op de verticale as oftewel de y as:

```
if (entiteit snelheid op y as > 0) {
   if ((cellDown and !cell) or (cellDiag and !cellRight and nx)) {
      Corrigeer de positie van de entiteit
      Zet de entiteit snelheid op y as naar 0
      ny = 0
   }
} else if (entiteit snelheid op y as < 0) {
   if ((cell and !cellDown) or (cellRight and !cellDiag and nx)) {
      Corrigeer de positie van de entiteit
      Zet de entiteit snelheid op y as naar 0
      cell = cellDown
      cellRight = cellDiag
      ny = 0
   }
}
```

Nadat we de collision checks voor de verticale as hebben uitgevoerd voeren we bijna precies dezelfde checks uit voor de horizontale as oftewel de x as.

```
if (entiteit snelheid op x as > 0) {
   if ((cellRight and !cell) or (cellDiag and !cellDown and ny)) {
      Corrigeer de positie van de entiteit
      Zet de entiteit snelheid op x as naar 0
   }
} else if (entiteit snelheid op y as < 0) {
   if ((cell and !cellRight) or (cellDown and !cellDiag and ny)) {
      Corrigeer de positie van de entiteit
      Zet de entiteit snelheid op x as naar 0
   }
}
```

En dat is het! Deze collision checks worden elke update (60 keer per seconde) uitgevoerd voor de speler en de monsters. Dan nu de collision tussen de speler en de overige entiteiten:

```
for (elke schatkist) {
   if ((absolute waarde(entiteitPos.x – schatkistPos.x) <  (entiteitBreedte/2 + schatkistBreedte/2)) and
       (absolute waarde(entiteitPos.y – schatkistPos.y) <  (entiteitHoogte/2 + schatkistHoogte/2))) {
      Verwijder de schatkist van het spel
      Verhoog de schatkist score, het aantal schatkisten de speler heeft verzameld
   }
}
```

Dit is het voorbeeld van hoe de collision werkt tussen speler en schatkist maar deze zelfde code wordt ook gebruikt om de collision tussen speler en monster te controleren. Het enige verschil is dat er een extra check in staat die controleert of de speler contact heeft gemaakt met het bovenste gedeelte van de monster of niet. Is dit het geval dan zal de monster worden verwijderd en de monster score worden verhoogd. Als dit niet het geval is dan heeft de speler dus niet op de monster gesprongen en sterft de speler en verliest hierbij één leven. Ook wordt de speler teruggezet naar zijn startpositie zolang er nog levens zijn. Zijn ze op? Dan wordt het hele level gereset.

Conclusie

Een tile-based platformer is simpeler dan ik in eerst instantie in gedachten had. Zoals hierboven is beschreven hoeft er alleen maar wat data te worden opgeslagen en vervolgens het level tekenen naar het scherm, gooi hier een paar bewegende entiteiten in en collision detection en viola een tile-based computerspel! Van te voren dacht ik hier veel te moeilijk over.

Tot slot de conclusie van de realisatie periode. De gehele periode is vrij soepel verlopen en tot mijn verbazing duurde de realisatie periode maar een korte twee weken. Ik heb mezelf overtroffen. Af en toe zat ik even vast met een probleem maar door goed na te denken of simpelweg op het internet te kijken kwam ik al snel met een oplossing. Ik ben tevreden over de gehele realisatie periode.