2017

GameMaker

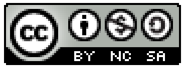
F. Vonk

versie 2

2-8-2017

inhoudsopgave

1. inleiding .... - 3 -
2. de wereld .... - 4 -
3. player character .... - 17 -
4. interactie met de wereld .... - 37 -
5. zwaartekracht .... - 47 -
6. A(rtificial) I(ntelligence) .... - 57 -
7. alarms .... - 70 -
8. meerdere rooms .... - 74 -
9. hoe verder .... - 78 -
10. wat heb je geleerd .... - 79 -



Dit werk is gelicenseerd onder een Creative Commons Naamsvermelding – NietCommercieel – GelijkDelen 3.0 Unported licentie

De afbeelding op het voorblad is verkregen via INFOwrs. Copyright © 2010 INFOwrs Serviços em informatica.

1. inleiding

Niet alleen het spelen van games is leuk, ook het maken ervan is leuk en divers. Deze film laat je zien wat je allemaal moet kunnen om een goede game te maken. Er bestaan tegenwoordig veel ontwikkelomgevingen die het leven gemakkelijker maken door veel van het saaie en echt moeilijke werk uit handen te nemen. Bekende ontwikkelomgevingen voor 3D games zijn Unreal Development Kit, Cry Engine en Unity 3D. Om daarmee te werken is voldoende programmeerervaring, inzet en doorzettingsvermogen echter wel een vereiste. Maar gelukkig zijn er ook ontwikkelomgevingen waarvoor je geen of weinig programmeerervaring nodig hebt en die je wel de kans geven om op een leuke manier te leren programmeren. GameMaker is zo'n omgeving.

Welkom bij de module *GameMaker*. Deze module gaat je laten zien hoe de ontwikkelomgeving werkt en hoe je er spellen mee kunt maken. De makers van GameMaker hebben geprobeerd de echte wereld zo goed mogelijk na te bootsen zodat jouw gamewereld op een natuurlijke manier te maken is en werkt. De invalshoek van deze module is dan ook om je GameMaker zo veel mogelijk te laten leren vanuit dingen die je kent uit de echte wereld.

In deze module kom je opgaves tegen die je moet maken om de lesstof te verwerken. De antwoorden kunnen in de les besproken worden.

opgave

Opgaves in blauw moet je maken.

Let op, links in dit document hebben een rode kleur.

Veel plezier en succes.

2. de wereld

Wij als mensen ervaren onze wereld als dynamisch. Om ons heen gebeuren heel veel dingen en daar reageren we op. Als je je smartphone hoort dan wil je hem pakken. Als je op de fiets zit en er toetert een auto achter je dan heb je de neiging om achterom te kijken. Als de deurbel gaat dan loop je naar de deur en doet hem open. We noemen dit reactief gedrag. Er treedt een gebeurtenis (event) op en we reageren daarop (de actie of action).

GameMaker werkt op een vergelijkbare manier. Je kunt er een gamewereld mee maken waarin je dingen gaat neerzetten. De gamewereld noemen we in GameMaker de room. Dingen in de room noemen we objecten. Alles wat we in de gamewereld zetten is een object. Dat is anders dan in de echte wereld want daar noemen we mensen en dieren geen objecten. In de gamewereld is een poppetje dat door de speler wordt bediend echter gewoon een object. Zo'n poppetje noemen we in game termen een player character (PC) maar vaak noemen we het ook wel gewoon "de speler" omdat het de representatie van de speler in de gamewereld is. Ook poppetjes die door de game zelf worden bediend zijn objecten. Zo'n poppetje noemen we in game termen een non-player character (NPC).

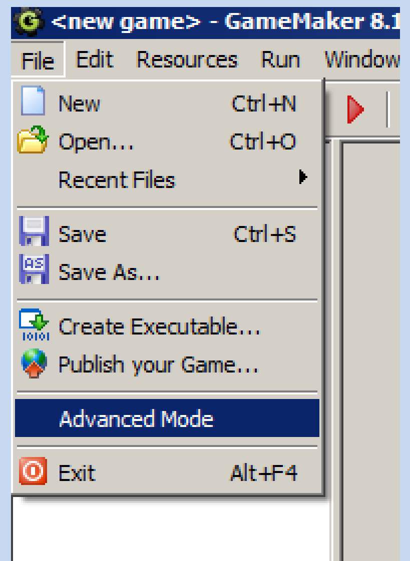
In GameMaker kunnen we, net als in de echte wereld, objecten laten reageren op gebeurtenissen (events) en ze acties uit te laten voeren. Dat is wat onze gamewereld dynamisch en interessant maakt.

Voor je aan de slag gaat met GameMaker, moeten we eerst nog één belangrijk aspect bespreken. We willen objecten in de room natuurlijk ook graag zien. Als we om ons heen kijken in de echte wereld zien we immers ook van alles. Om in GameMaker een object te visualiseren gebruiken we zogenaamde sprites. Een sprite is een afbeelding of serie afbeeldingen. We zullen met de gemakkelijkste vorm beginnen en dat is dat we één afbeelding aan een object koppelen.

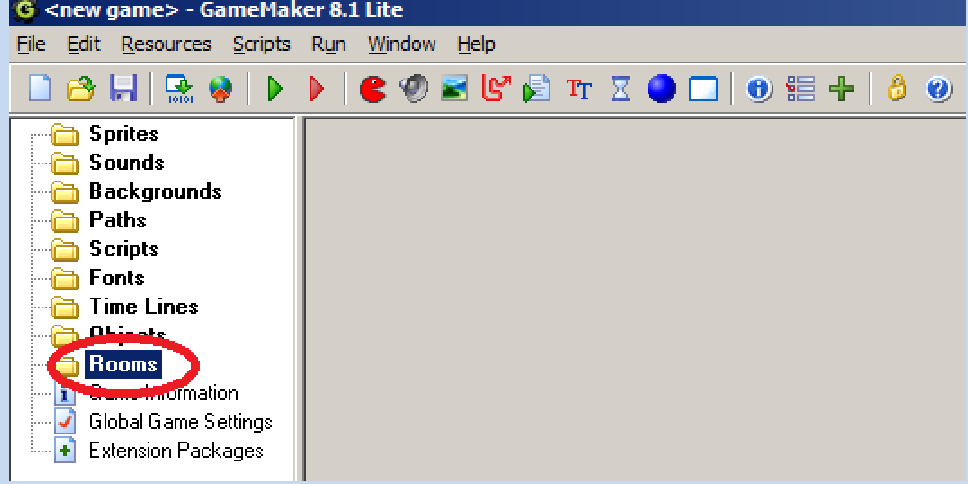
opgave 2.1

Als het goed is staat de full version van GameMaker 8.1 op je computer geïnstalleerd. Je kunt deze vinden via het startmenu onder vakken, informatica.

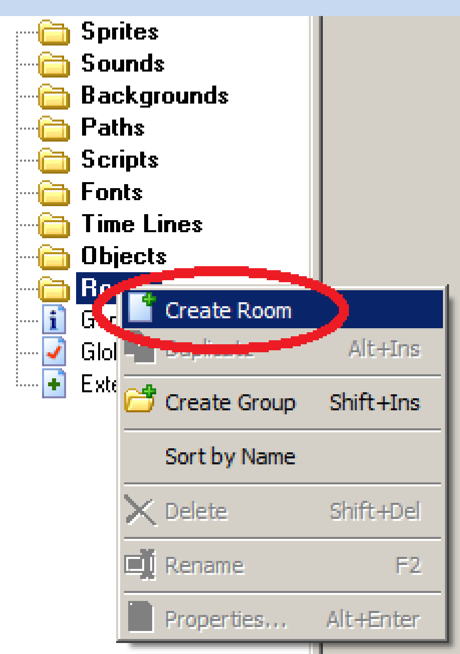
Als je GameMaker opstart krijg je mogelijk de vraag of je het programma in advanced mode wilt opstarten. Klik bij deze vraag op "ja". Als je per ongeluk op "nee" hebt gedrukt kun je *advanced mode* aanzetten via het GameMaker menu zoals hierna is afgebeeld.



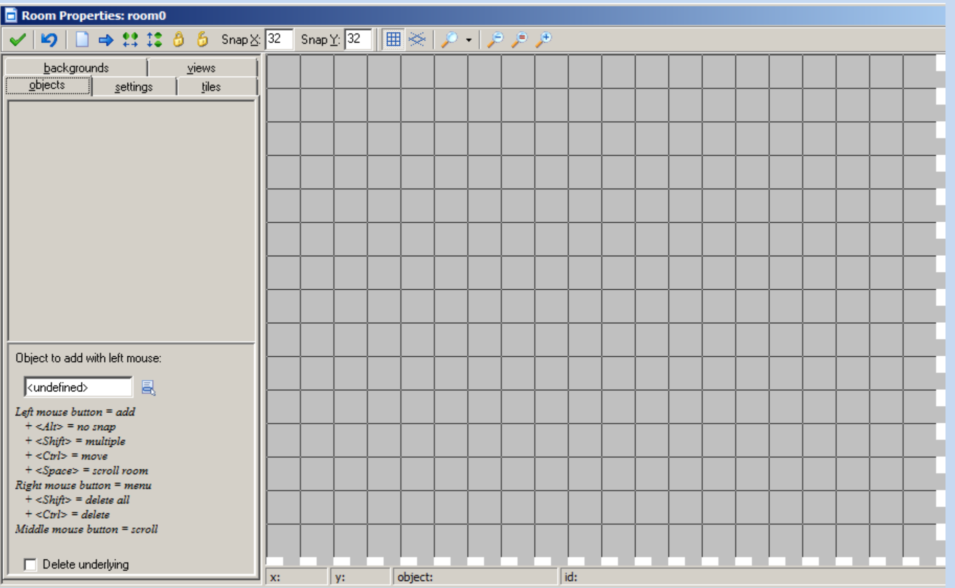
Als eerste maken we een gamewereld aan, een room in GameMaker. Aan de linkerkant in de ontwikkelomgeving staat een lijst met **re-sources** (het **resource paneel**). Eén van de resources is *Rooms* zoals je in de volgende afbeelding.



Als je met je rechter muisknop op *Rooms* klikt kun je een nieuwe room aanmaken met *Create Room* zoals je in de afbeelding hierna ziet.

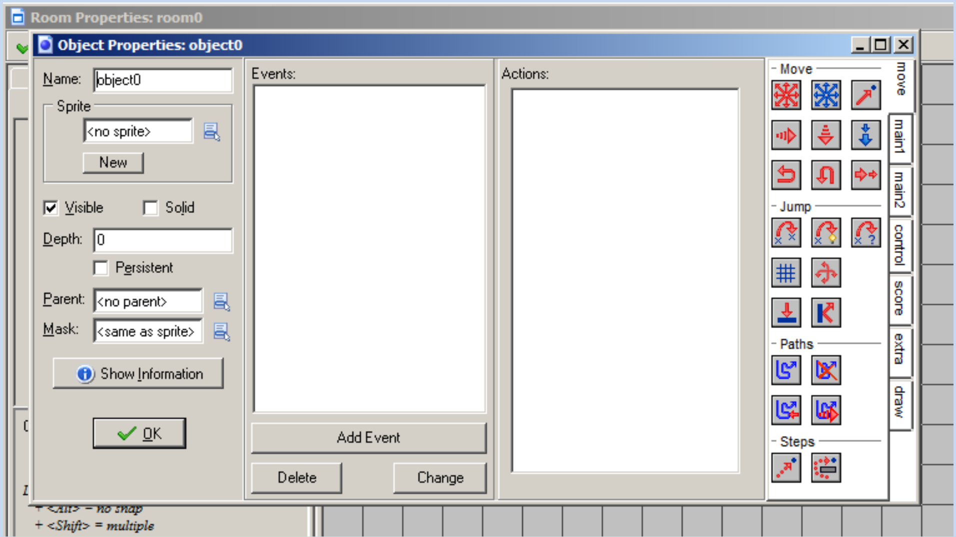


Maak een room aan. Je krijgt dan een venster te zien dat *Room Properties: room0* heet zoals je hierna ziet. Dit is het **Room venster**.



Je ziet dat er 5 tabs zijn aan de linkerkant in dit venster. De op dit moment getoonde tab is *objects* en dat is voorlopig de belangrijkste tab. De andere tabs negeren we voorlopig. We willen namelijk graag objecten in onze wereld zetten. Maar dan moeten we wel eerst objecten hebben. Dus we gaan een object maken. **Laat het *Room venster* open staan**.

Objecten kunnen we op dezelfde manier aanmaken als rooms door met de rechter muisknop in het *resource paneel* op *Objects* te klikken. Als je een object aanmaakt dan krijg je als het goed is een nieuw venster te zien dat *Object Properties: object0* heet zoals hierna afgebeeld. Dit is het **Object venster**. Je ziet dat dit venster gewoon over het *Room venster* komt te staan.

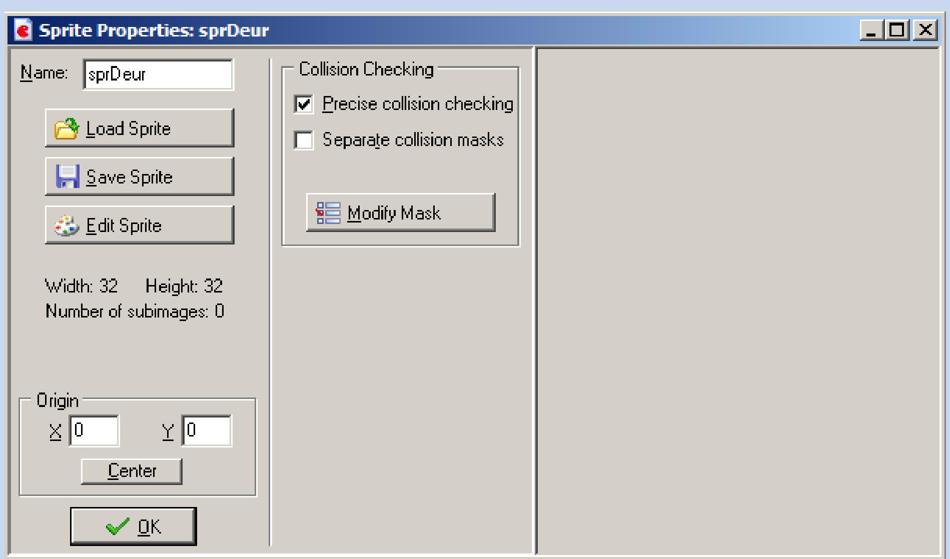


Je ziet dat GameMaker automatisch namen geeft aan resources. Deze namen zijn echter nietszeggend. Voor de room is dat nu nog minder erg, maar objecten moeten we **altijd** een zinvolle naam geven. Wat we willen doen is een deur in onze wereld zetten. Daarom ga je het nieuwe object *objDeur* noemen in plaats van *object0*.[1] **Je gebruikt "obj" als prefix omdat het een object is.**

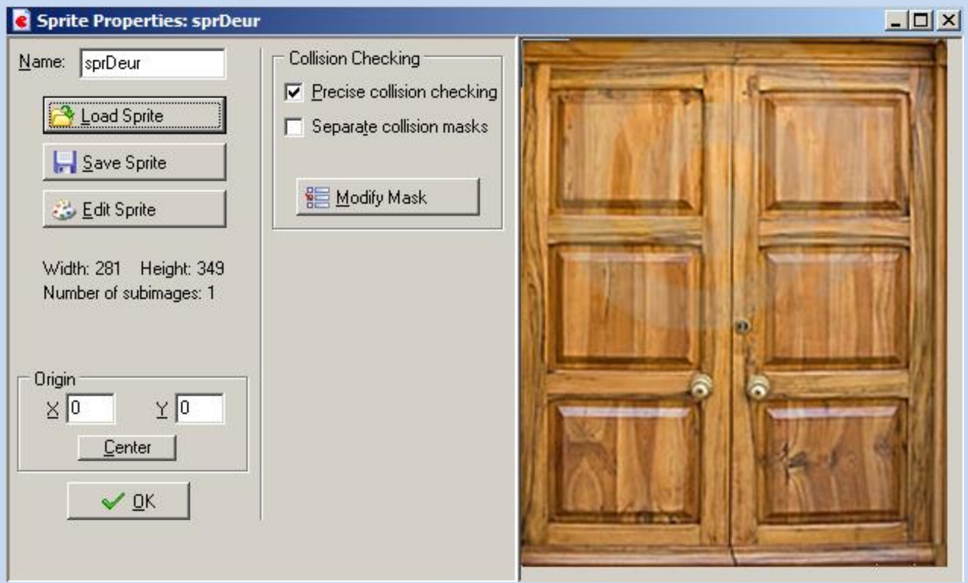
Vervolgens moeten we een *sprite* aan het object koppelen zodat het zichtbaar is als we het in de room plaatsen. Het maken van een sprite werkt hetzelfde als voor een room en object. **Laat het *Object venster* open staan**. Maak nu een sprite aan en als het goed is krijg je dan een venster dat *Sprite Properties: sprite0* heet. Dit is het **Sprite venster**. Ook sprites moeten altijd een zinvolle naam hebben. Dit wordt de sprite voor de deur en daarom noemen we de sprite *sprDeur* in plaats van *sprite0*. **Je gebruikt "spr" als prefix omdat het een sprite is.** Als het goed is, ziet je sprite venster er nu als volgt uit.

---

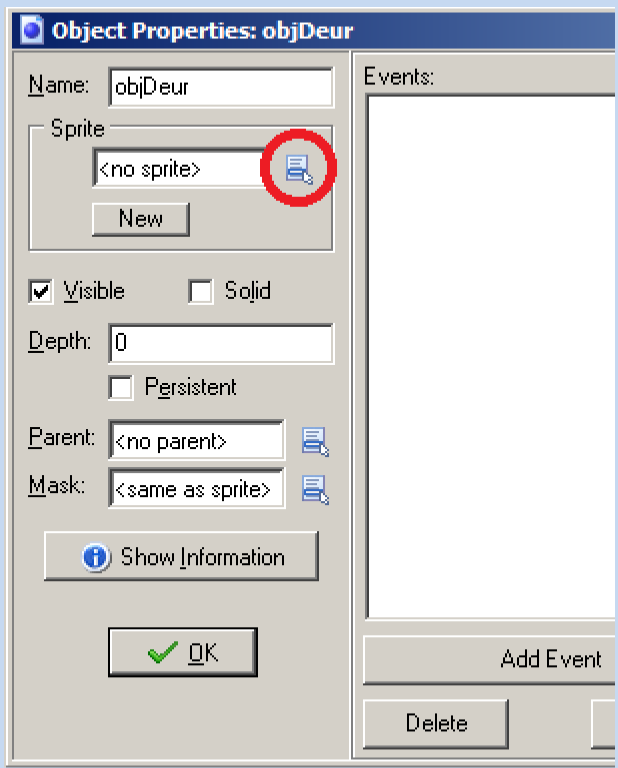
[1] De versie van GameMaker die je gebruikt kan een bug bevatten die ervoor zorgt dat je steeds een foutmelding krijgt als je een letter intypt bij het hernoemen van je object in het *Object venster*. Sluit in dat geval het *Object venster* en hernoem het object in het *resource paneel*.



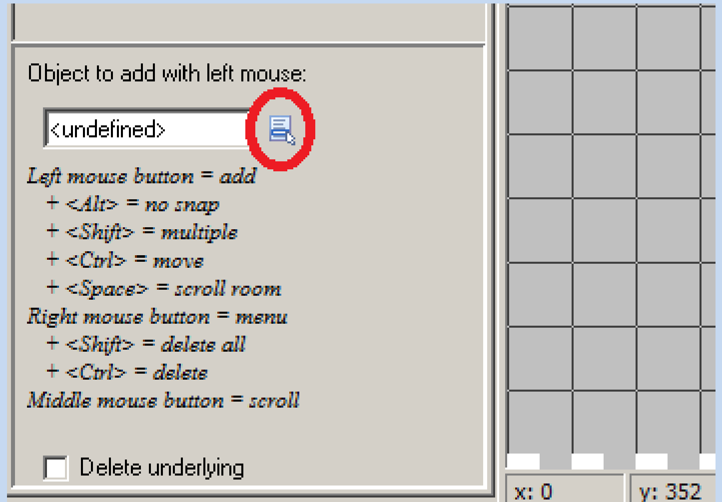
In GameMaker kun je je eigen sprites maken, maar nu gaan we even een kant en klare afbeeldingen laden. Dit doen we door op de *Load Sprite* knop te drukken die onder de naam van de sprite staat. Je krijgt dan het *Open a Sprite Image* venster te zien. Hiermee kun je de *deur.jpg* afbeelding opzoeken en laden. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort. Zoek de afbeelding en laad deze. Als het goed is, ziet je *Sprite venster* er dan als volgt uit.



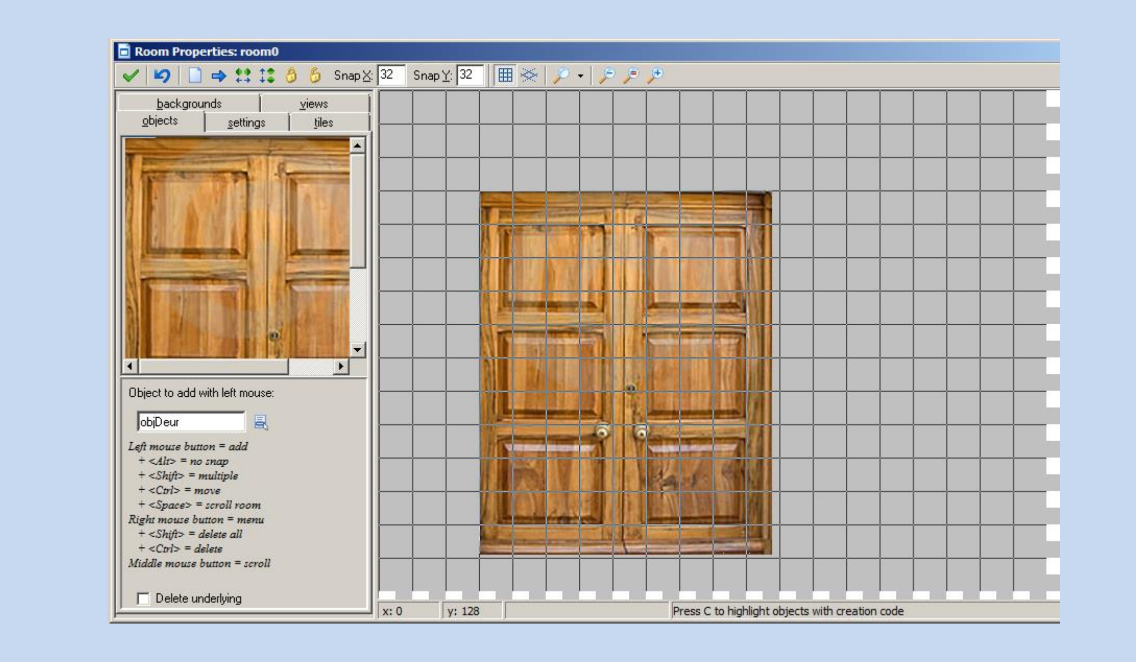
Druk nu op *OK* en als het goed is wordt het *Sprite venster* gesloten. Vervolgens kun je in het *Object venster* de sprite aan het object koppelen via het *Sprite* selectiegedeelte onder de naam van het object. We hebben op het moment maar één sprite dus de keuze is makkelijk. Klik op de *selecteer sprite* knop zoals je in de volgende afbeelding ziet.



Selecteer de sprite *sprDeur* en klik op *OK* in het *Object venster*. In de *objects* tab van het *Room venster* zit een *selecteer object* knop zoals je ziet in de volgende afbeelding.



Selecteer het object *objDeur*. Nu kun je, door met je linker muisknop in de room te klikken, *deur* objecten plaatsen in je wereld. Het is verstandig om even met de interface voor het plaatsen van objecten te spelen. Kijk goed naar de beschrijvingen in het venster zodat je weet welke toets-muis combinaties wat doen. Uiteindelijk is het de bedoeling dat je één deur object in je wereld hebt staan zoals je kunt zien in de volgende afbeelding.

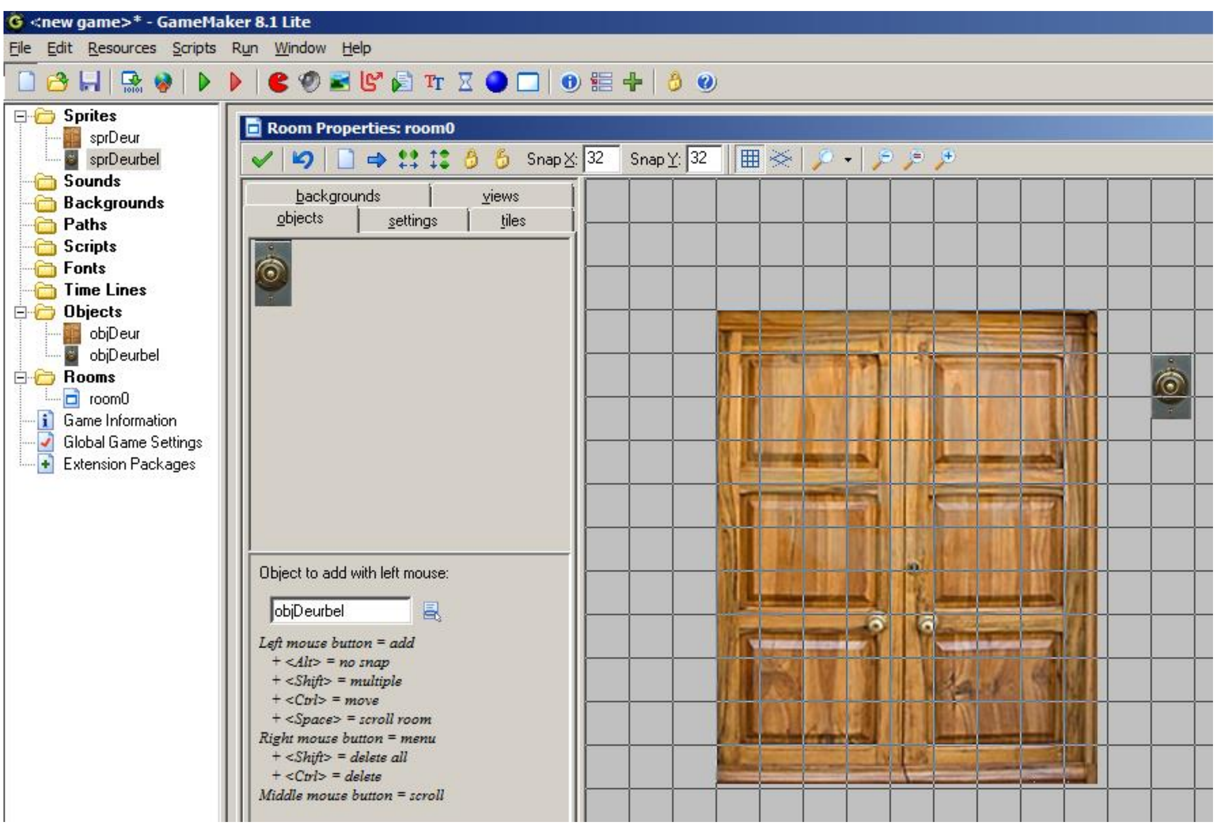


Zo nu heb je geleerd hoe je een room, object en sprite maakt. En je weet hoe je een sprite aan een object koppelt en hoe je een object in een room zet, verplaatst en weghaalt.

opgave 2.2

Herhaal wat je hebt geleerd in opgave 2.1 voor een deurbel. De afbeelding van de deurbel heet *deurbel.jpg* en kun je vinden in het ZIP bestand dat bij deze module hoort. Plaats de deurbel rechts naast de deur in je wereld.

Als je opgave 2.2 hebt gedaan dan zou je nu een sprite genaamd *sprDeurbel* moeten hebben, een object genaamd *objDeurbel* met daaraan gekoppeld de sprite *sprDeurbel* en een GameMaker omgeving die er als volgt uitziet.



opgave 2.3

Run je spel en kijk wat je ermee kunt doen. Dit doe je door op de groene *play* knop te klikken. Waar je deze knop vindt zie je hierna afgebeeld.

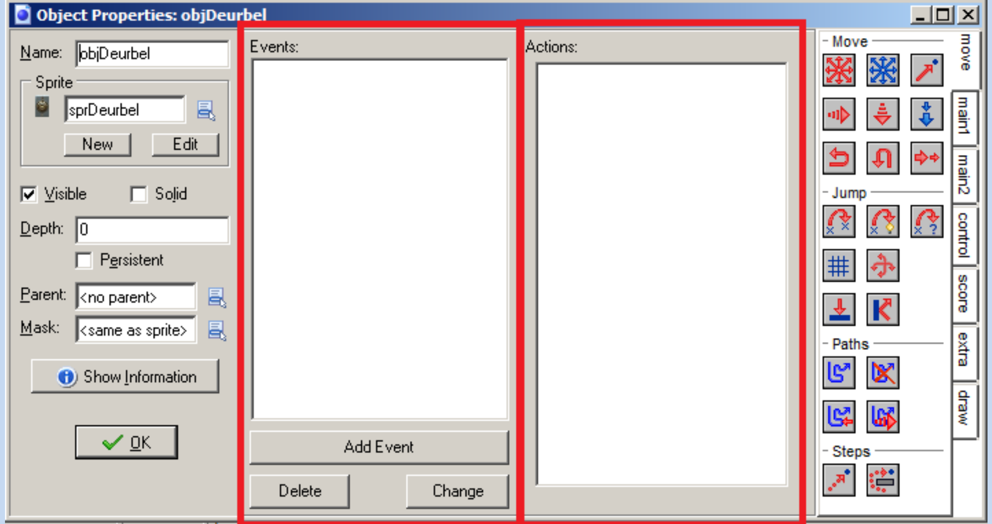
Dit is allemaal wel leuk en aardig maar zoals je waarschijnlijk gemerkt hebt kun je nog niks met de objecten in het spel. Dat komt omdat je nog geen enkel object

hebt verteld op welke gebeurtenissen (events) het moet reageren en hoe. Dat ga je nu doen. We willen namelijk dat we geluid horen als je op de bel drukt.

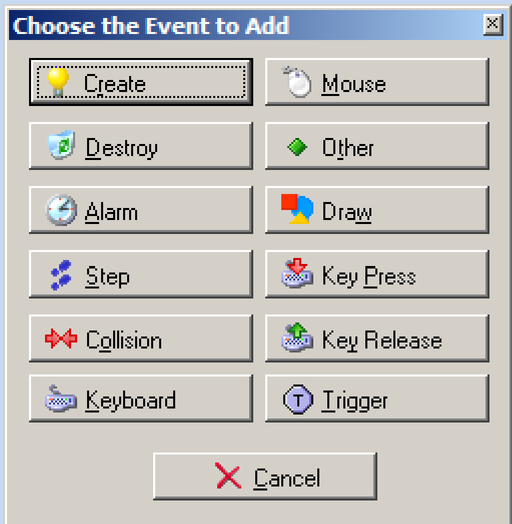
opgave 2.4

Om interacties (events & acties) met een object te definiëren moeten we het *Object venster* van dat object openen. Dat doen we door te dubbelklikken op de naam van het object in het *resource paneel*. Open het *Object venster* voor *objDeurbel*.

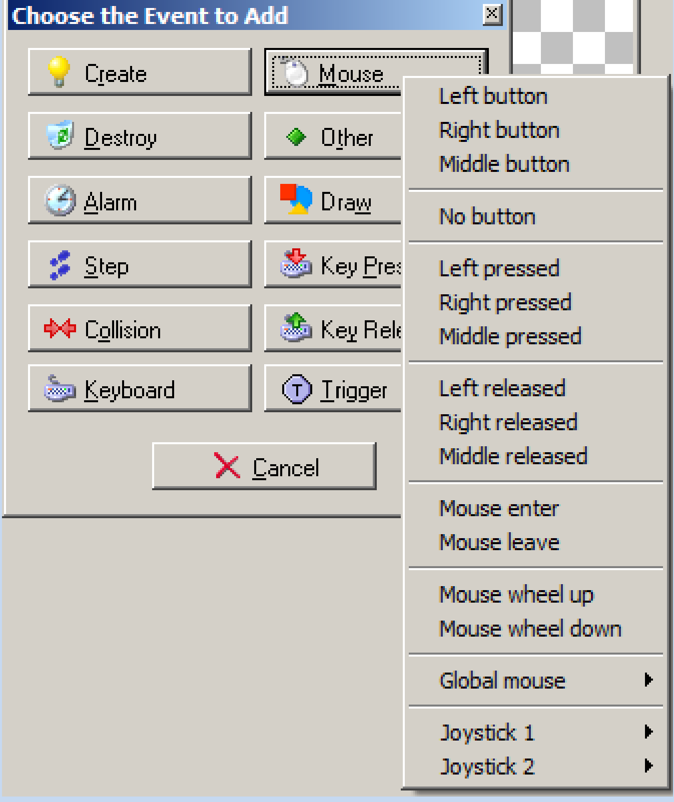
We gaan naar de *Events* en *Actions* gedeeltes kijken die je in de afbeelding hierna gemarkeerd ziet.



Het is belangrijk om te snappen dat actions bij events horen. Dus als je geen events hebt kun je geen actions definiëren. Daarom ga je eerst een event toevoegen. Klik op de *AddEvent* knop. Je ziet dan de volgende dialoog.



We willen dat er geluid klinkt als we op de deurbel klikken en daarom moeten we een *Mouse* event (de knop rechtsboven) toevoegen. Als je op de *Mouse* knop drukt krijg je de volgende pull-down lijst.

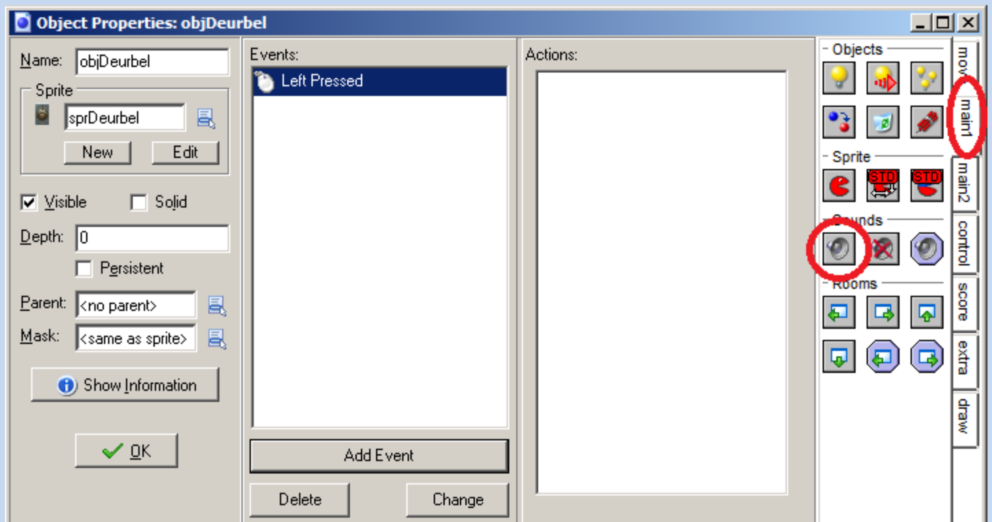


Je gaat de deurbel op de linker muisknop laten reageren. We zien zowel een *Left button* event als een *Left pressed* event. **Het is belangrijk dat je het verschil snapt tussen deze twee events.** Het *Left button* event blijft optreden vanaf het moment dat de linker muisknop wordt ingedrukt totdat deze weer wordt losgelaten. Het *Left pressed* event treedt exact één keer op wanneer de linker muis-

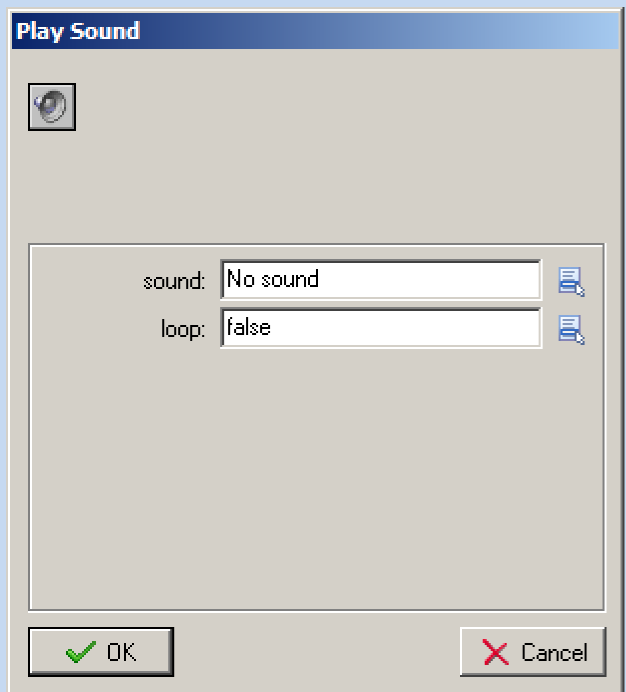
knop ingedrukt wordt, ongeacht hoe lang je de knop daarna ingedrukt houdt. Daarom is er ook nog een apart *Left released* event.

We gaan voor de deurbel met het *Left pressed* event werken. Voor we het event en de bijbehorende actions toevoegen, is het verstandig om eerst het geluid in onze omgeving te zetten. Daarvoor maken we een sound aan in het *resource paneel* bij *Sounds*. Een sound toevoegen doe je hetzelfde als een sprite toevoegen. Dus voeg nu het geluid van een deurbel toe. Het bestand met het geluid heet *deurbel.mp3* en kun je vinden in het ZIP bestand dat bij deze module hoort. Geef het geluid de naam *sndDeurbel* in plaats van *sound0*. **Je gebruikt "snd" als prefix omdat het een sound is.**

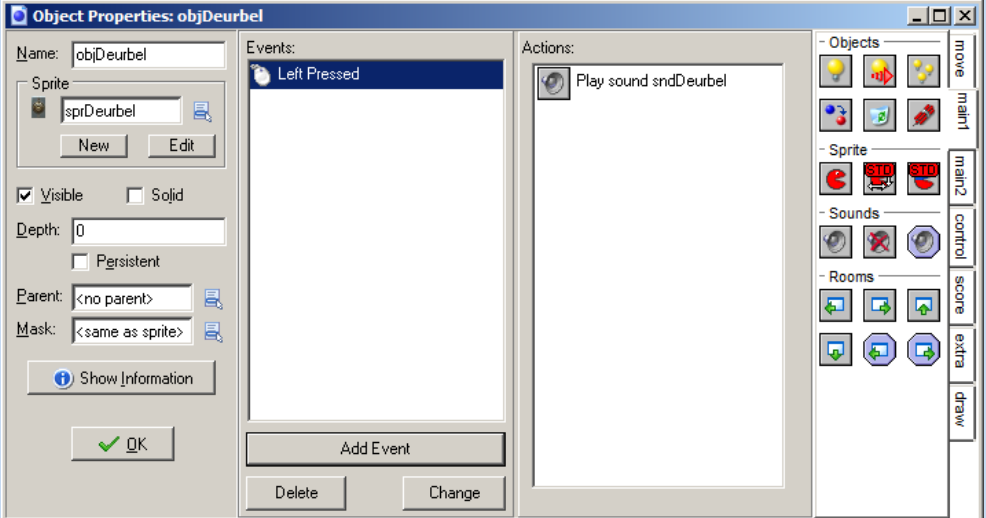
Nu je het geluid in je ontwikkelomgeving hebt zitten kun je het event en de actions gaan toevoegen. Voeg eerst het *Left pressed* event toe. Nu kun je via de *main1* tab aan de rechterkant van het *Actions* vak de *Play Sound* action toevoegen. Waar je dit allemaal kunt vinden zie je in de volgende afbeelding.



Je kunt nu de *Play Sound* action in het *Actions* vak slepen met je muis en dan krijg je de dialoog die je hierna ziet.



Selecteer *sdnDeurbel* bij *sound* en laat *loop* op *false* staan. Dat laatste betekent dat het geluid niet steeds herhaald wordt en dat is precies wat we willen. Klik op *OK*. Je object venster ziet er dan als volgt uit.



Klik op *OK* en run je spel. Zorg op school dat je een koptelefoon aangesloten hebt op je computer anders kan GameMaker vreemd reageren. GameMaker wil namelijk geluid produceren en mag dat op school niet altijd van Windows als er geen koptelefoon is aangesloten. Thuis heb je dit probleem niet.

Klik op de deurbel met je muis en kijk (of beter gezegd luister) wat er gebeurt.

Goed, je hebt intussen al veel geleerd. In je programma heb je een room gemaakt met daarin twee objecten en daaraan gekoppeld twee sprites. Aan één van de objecten heb je een event met een bijbehorende action gekoppeld, aan het andere niet. Je kunt in je gamewereld dus objecten hebben die nergens op reageren. Bovendien heb je met geluid gewerkt.

opgave 2.5

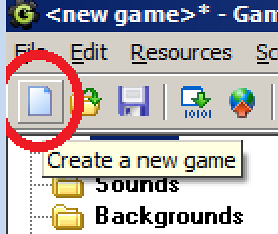
Sla je spel op met *hoofdstuk2.gm81* als naam op je H schijf (netwerk schijf). Je doet dit met Save As uit het File menu van GameMaker zoals je ziet in de volgende afbeelding.

3. player character

Wat je in het vorige hoofdstuk hebt gemaakt is niet echt een spel. Wat een spel interessant maakt is vaak dat je een poppetje in je gamewereld hebt dat je kunt besturen. Dat poppetje wordt dus vaak het player character (PC) genoemd. Laten we eens kijken hoe we dat kunnen doen. Dit hoofdstuk gaat ervan uit dat je goed snapt wat je in het vorige hoofdstuk hebt geleerd en dit ook kunt toepassen.

opgave 3.1

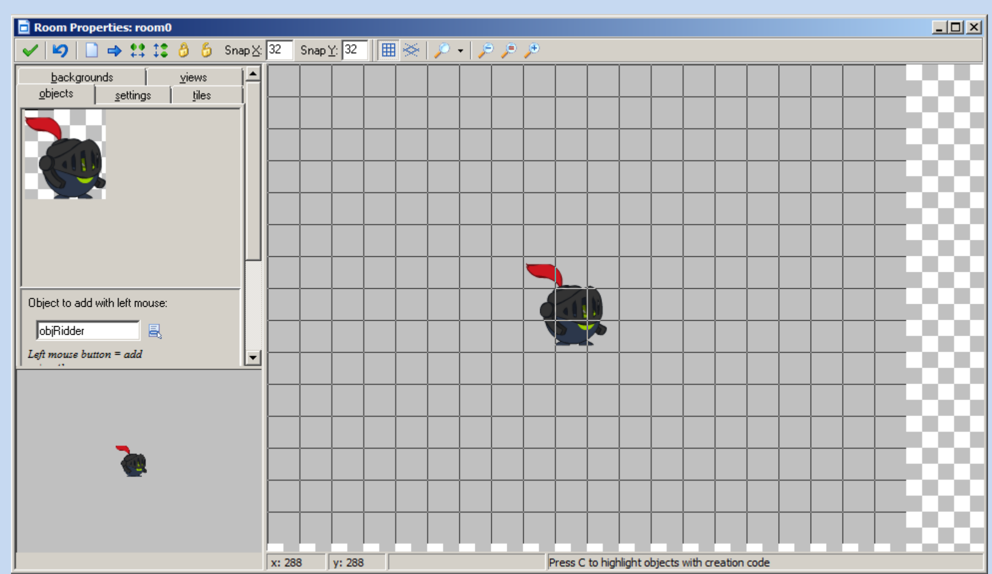
Start een nieuwe game. Dit kun je doen door op *Create a new game* te klikken onder het menu zoals je kunt zien in de volgende afbeelding.



Doe de volgende dingen:

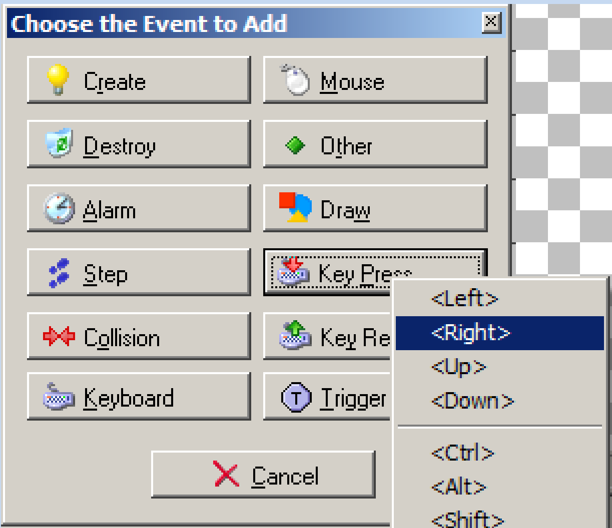
- Maak een room.
- Maak een sprite die *sprRidder* heet en laad daarin de afbeelding *ridder.png*. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort.
- Maak een object dat *objRidder* heet en koppel hieraan *sprRidder*.
- Zet het object *objRidder* in de room.

Dit zou er als volgt uit moeten zien.

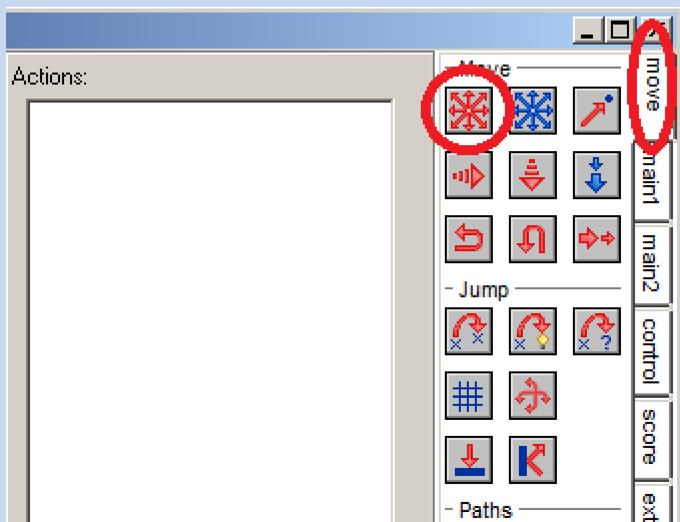


Open nu het *Object venster* voor *objRidder*. We gaan nu zorgen dat we de ridder kunnen besturen met de pijltjestoetsen. Er zijn een aantal manieren om dit voor elkaar te krijgen. Er is een moeilijke en een makkelijke manier als het gaat over wat je van GameMaker moet weten. De makkelijke manier heeft echter een nadeel dat de moeilijke manier niet heeft.

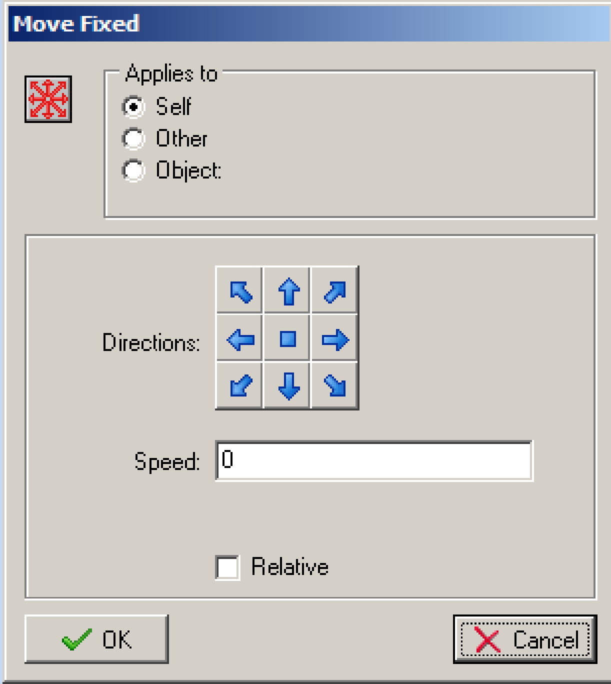
Laten we toch eerst naar de makkelijke manier kijken. Je wilt de ridder naar rechts bewegen wanneer je op de pijltjestoets naar rechts drukt. Hiervoor voeg je het *Key Press - <Right>* event toe zoals je hierna ziet afgebeeld.



Vervolgens koppel je aan dit event de action *Move Fixed* uit de *move* tab zoals je hierna in de afbeelding ziet aangegeven.



Je krijgt dan de volgende dialoog te zien.



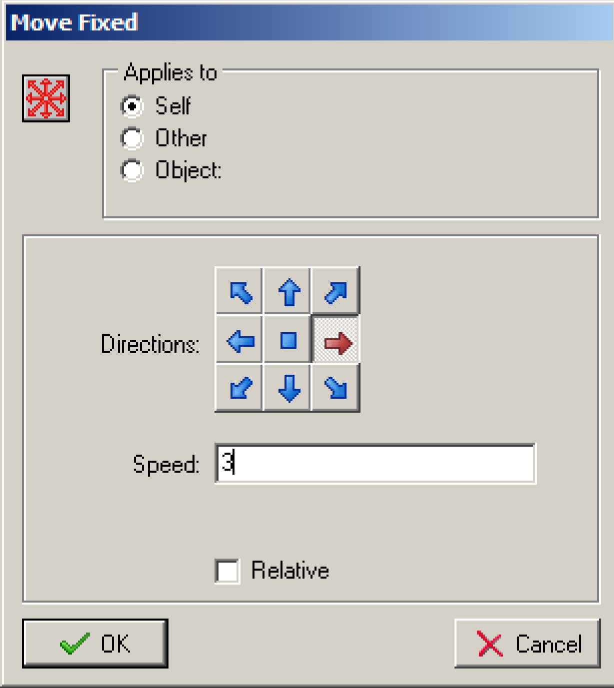
Het *Applies to* gedeelte geeft aan waarop de action van toepassing is. Meestal hoef je hieraan niets te veranderen en kun je de waarde op *Self* laten staan. In Hoofdstuk 4 leer je wat de andere opties betekenen. Nu ben je bezig met het bewegen van de ridder zelf en kies je voor de optie *Self*. Dit is ook de standaard (**default**) waarde.

We zitten in het *Key Press - <Right>* event dus je klikt het pijtje naar rechts aan.

Naar wat *Speed* precies doet kijken we ook later. Nu kies je de waarde 3.

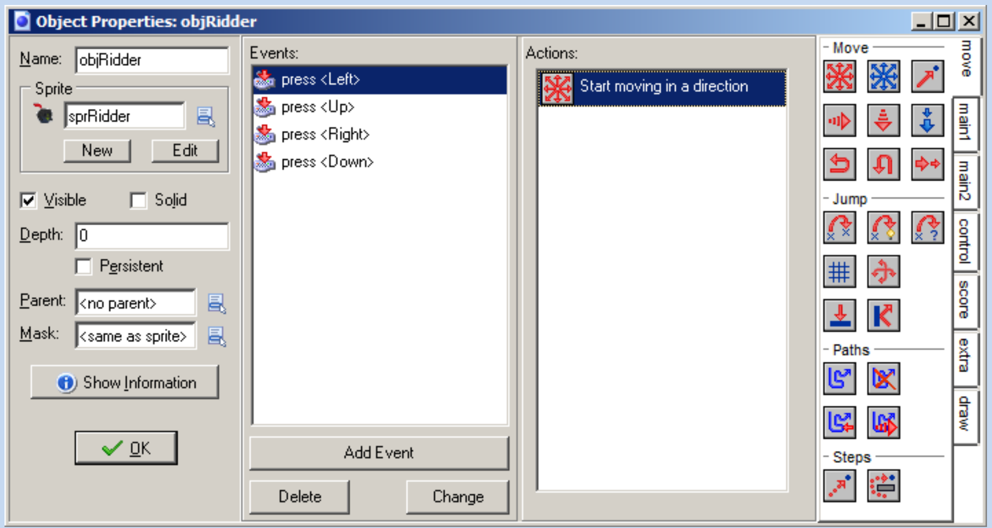
De optie *Relative* vink je niet aan. Ook hier komen we later op terug.

De dialoog komt er dan als volgt uit te zien.



Je bent nu klaar en kunt op *OK* klikken. Vervolgens doe je hetzelfde voor de pijltjestoetsen naar links, boven en beneden. Natuurlijk kies je hier een andere richting dan bij de pijltjestoets naar rechts.

GameMaker bepaalt zelf de volgorde waarin de events in het *Events* vak komen te staan. Daar hoef je je geen zorgen over te maken. Je object venster ziet er dan uiteindelijk als volgt uit.



Je kunt het object venster gewoon open laten staan en je spel runnen. De veranderingen zijn al doorgevoerd. Dus run je spel en kijk wat er gebeurt als je op de pijltjestoetsen drukt.

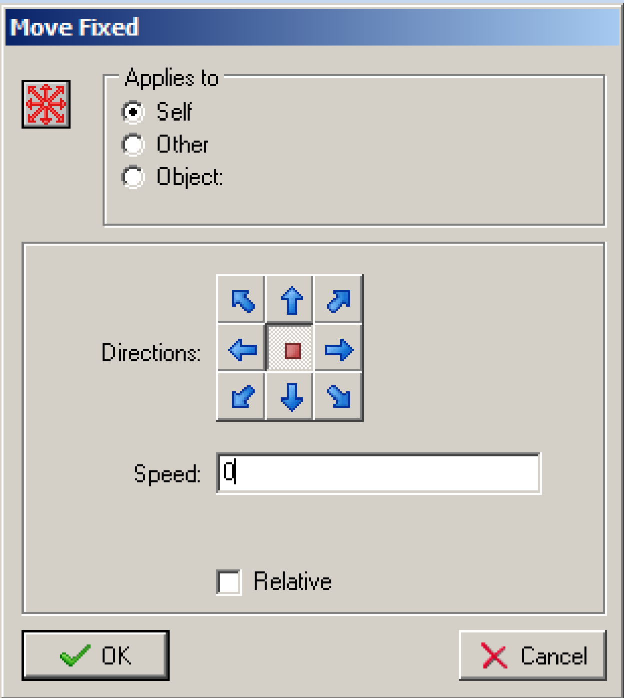
Hopelijk vallen je een aantal dingen op:

1. Als je een pijltjestoets loslaat dan blijft de ridder lopen.
2. Als je naar links beweegt dan lijkt het alsof de ridder achteruit loopt.
3. De ridder kan niet diagonaal lopen door twee pijltjestoetsen in te drukken.
4. De ridder kan de wereld uitlopen.

Deze dingen ga je natuurlijk verhelpen (fixen). We hebben onze ridder verteld dat hij een richting in moet lopen als we een pijltjestoets indrukken. Maar we hebben hem nooit verteld dat hij moet stoppen. Je moet computers alles expliciet vertellen want ze kunnen niet denken zoals mensen.

opgave 3.2

We gaan de ridder laten stoppen als we een pijltjestoets los laten. Daar hebben we gelukkig *Key Release* events voor. Voor iedere richting zeggen we tegen de ridder dat hij moet stoppen en dat doen we weer met behulp van de *Move Fixed* action en wel op de volgende manier.

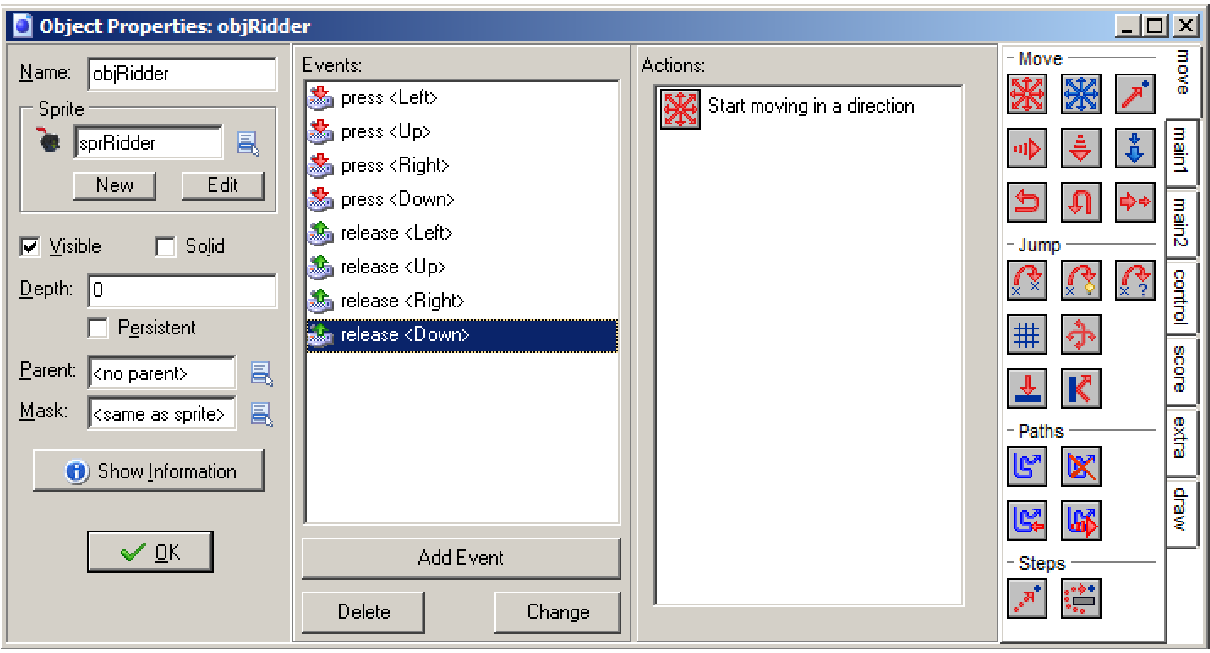


De knop in het midden van de *Directions* betekent dus stoppen. De

Speed is nu niet relevant, maar het is netjes om hem op 0 te zetten.

Voeg voor alle relevante *Key Release* events deze action toe. Omdat de action telkens hetzelfde is kun je de action, die je voor het eerste event maakt, kopiëren en in de andere events plakken. Test daarna of het stoppen werkt door je spel te runnen.

Als je alles goed hebt gedaan stopt de ridder nu met lopen als je een pijltjestoets loslaat. Je object venster zou er nu als volgt uit moeten zien.



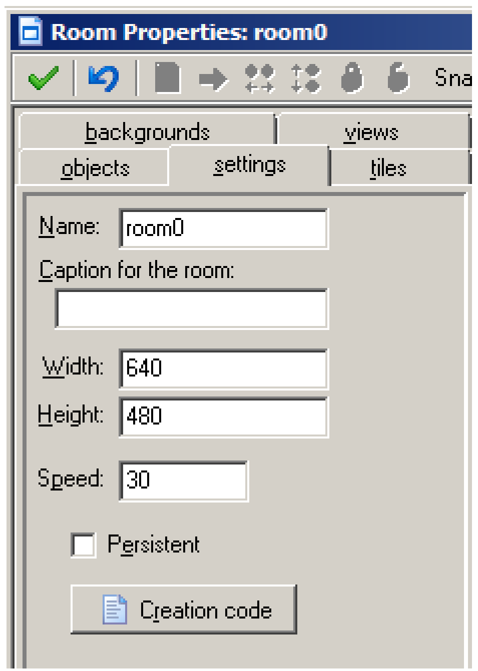
Zoals beloofd moeten we nog uitleggen wat *Speed* en *Relative* doen. Laten we beginnen met *Speed* (snelheid).

In GameMaker wordt de snelheid van objecten bepaald aan de hand van de snelheid van de gamewereld. Je denkt misschien: "Dat is raar. Hoe kan een wereld nu een snelheid hebben?" Toch is het niet zo heel vreemd als je in termen van tijd denkt. In onze echte wereld lijkt alles om ons heen continu en vloeiend te gebeuren. Maar is dat ook echt zo?

Onze ogen kunnen maximaal zo'n 100 beelden per seconde naar onze hersenen doorsturen. Als er per seconde bijvoorbeeld 200 veranderingen in onze wereld optreden dan missen we in het beste geval dus de helft van de veranderingen en gemiddeld genomen nog meer. Als onze ogen dus optimaal werken, dan is de snelheid waarmee de echte wereld voor ons mensen dus "beweegt/verandert" maximaal 100 hertz (100 beelden per seconde).

Voor onze gamewereld geldt iets soortgelijks. Alleen werkt de gamewereld niet met beeldjes maar met events. De snelheid van onze gamewereld is het aantal keer per

seconde dat alle events voor alle objecten in de wereld afgehandeld worden. We noemen deze snelheid de room speed. Standaard staat de room speed op 30. Het is alsof er een klok tikt die, bij room speed 30, iedere één dertigste (1/30) seconde een puls afgeeft aan alle objecten in de gamewereld. Deze puls noemen we in het vervolg de event puls (in andere bronnen kom je ook wel de term klok tik of tick tegen). De room speed kun je vinden in het room venster bij *Speed* in de tab *settings* zoals je in de afbeelding hierna ziet.



Wat betekent het dan als we tegen onze ridder zeggen dat hij met snelheid 3 moet bewegen? Daarvoor moeten we eerst nog weten wat de eenheid van bewegen is in onze wereld. In de *settings* tab van het room venster zien we twee andere belangrijke waardes. Namelijk de *Width* en de *Height*. Deze geven de grootte van onze wereld aan uitgedrukt in pixels (de eenheid). Simpel gezegd is een pixel een puntje op het beeldscherm van je computer. Dit is niet helemaal waar maar voldoende waar om niet verder in detail te treden. De standaard grootte van onze wereld is dus 640x480 pixels. Gemakshalve is voor de eenheid van bewegen in onze wereld ook voor pixels gekozen. Dus met snelheid 3 zeggen we dat er iedere event puls 3 pixels bewogen wordt. Bij een room speed van 30 betekent dat dus dat onze ridder $30 * 3 = 90$ pixels per seconde beweegt.

Relatief (relative) betekent dat je iets uitdrukt ten opzichte van iets anders of ten opzichte van een vorige toestand. Het tegenovergestelde van relatief is absoluut. Een voorbeeld zal het één en ander duidelijk maken.

We zetten op dit moment de snelheid van de ridder op drie en vinken *Relative* niet aan. Dat is dus absoluut; oftewel de waarde die we invullen is de waarde die exact gebruikt wordt voor de snelheid.

Op het moment dat we *Relative* aanvinken, dan gebeurt er iets anders. De uiteindelijke snelheid wordt dan niet de waarde die we invullen, maar de waarde die we invullen wordt opgeteld bij de huidige snelheid. Als we dat in ons geval zouden doen, dan zouden we een foutmelding krijgen, want we hebben nog geen huidige snelheid.

Maar goed, we hebben intussen alleen ons eerste probleem opgelost. Er zijn er nog drie over, namelijk:

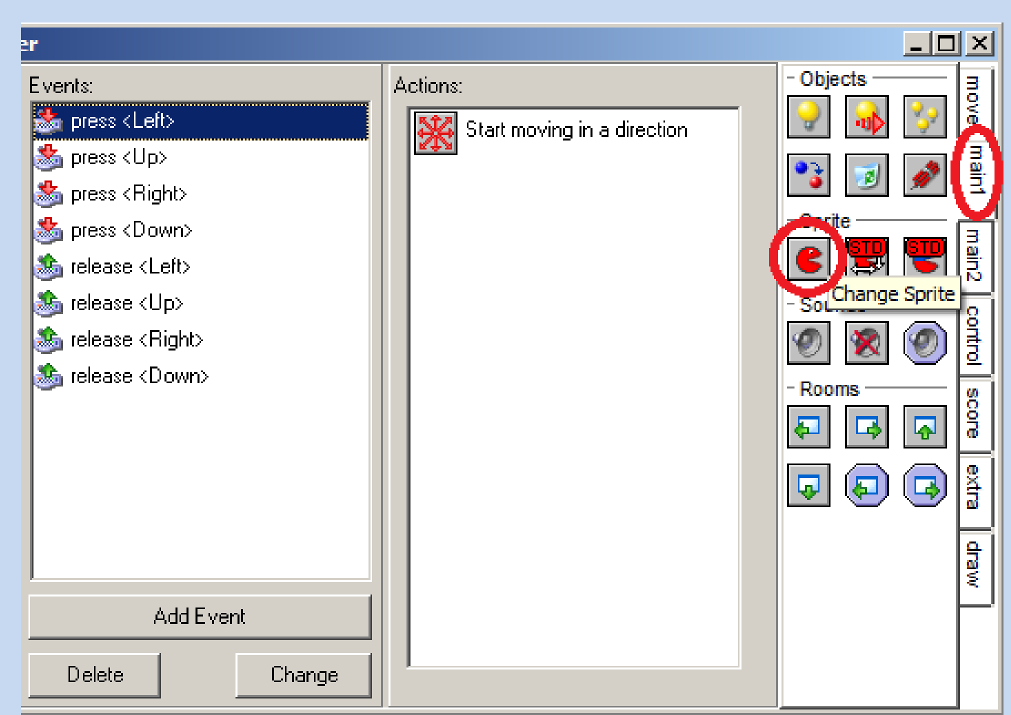
1. ~~Als je een pijltjestoets loslaat dan blijft de ridder lopen.~~
2. Als je naar links beweegt dan lijkt het alsof de ridder achteruit loopt.
3. De ridder kan niet diagonaal lopen door twee pijltjestoetsen in te drukken.
4. De ridder kan de wereld uitlopen.

Het tweede probleem oplossen is gelukkig makkelijk. We moeten gewoon een andere afbeelding gebruiken bij het naar links lopen. Dat gaan we nu regelen.

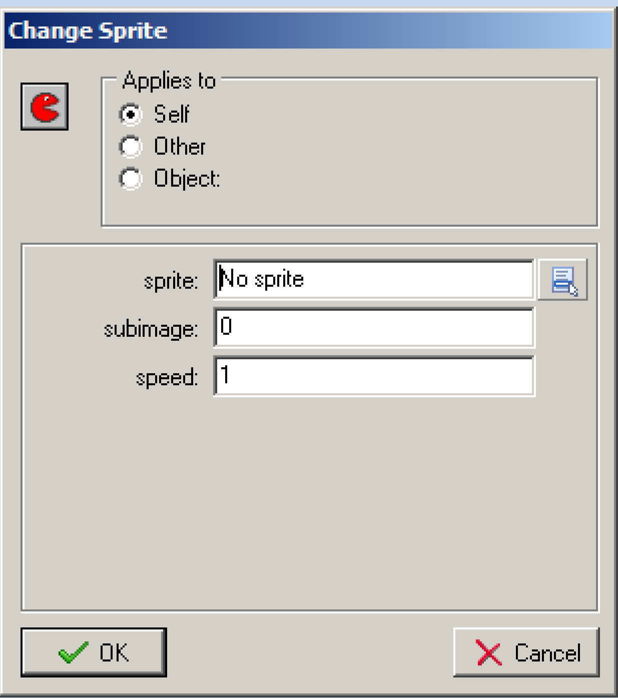
opgave 3.3

Nu ga je ervoor zorgen dat de sprite verandert als we op de linker pijltjestoets drukken. Daarvoor moet je eerst een nieuwe sprite *sprRidderLinks* maken en daarin de afbeelding *ridder_links.png* laden. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort.

Als je dit gedaan hebt dan voeg je bij het *Key Press - <Left>* event de action *Change Sprite* toe. Deze vind je in de *main1* tab zoals je in de afbeelding hierna kunt zien.



Als je de *Change Sprite* action toevoegt dan krijg je de volgende dialoog.



Kies nu de sprite *sprRidderLinks* bij *sprite*. Bij *subimage* en *speed* hoef je voorlopig niks te veranderen. Run vervolgens je spel en kijk wat er gebeurt.

Hopelijk heb je opgemerkt dat de ridder de goede kant indraait als je op de pijltjestoets naar links drukt. De ridder draait echter niet terug als je weer op de rechter pijltjestoets drukt. Aan de slag dus.

opgave 3.4

Bij het *Key Press - <Right>* event moeten we dus ook een sprite verandering doen ook al staat de ridder daarvoor initieel (in het begin) gewoon goed.

Om alles netjes te maken doen we het volgende. Open het sprite venster voor *sprRidder*. Verander de naam in *sprRidderRechts* en laad de afbeelding *ridder_rechts.png*. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort. Klik dan op *OK*.

Kijk eens goed wat er in *objRidder* gebeurd is met de sprite die aan het object gekoppeld is. Hopelijk zie je dat deze veranderd is. GameMaker heeft automatisch het veranderen van *sprRidder* in *sprRidderRechts* doorgevoerd voor *objRidder* zodat alles blijft werken. Handig!

Verander in *objRidder* de naam in *objRidderRechts*. Voeg vervolgens de *Change Sprite* action toe bij het *Key Press - <Right>* event zoals je dat ook bij het *Key Press - <Left>* event hebt gedaan in de vorige opgave. Run je spel en kijk of alles werkt.

Zo het tweede probleem is ook opgelost. Op naar het derde dat ons bij de moeilijke manier van het besturen van een object brengt. Daarvoor heb je eerst een stukje theorie nodig. Weet je nog dat we het over *Relative* hadden en een voorbeeld ervan gaven met de snelheid van de ridder? *Relative* is niet alleen van toepassing op snelheid maar ook op positie. Het werkt daar als volgt.

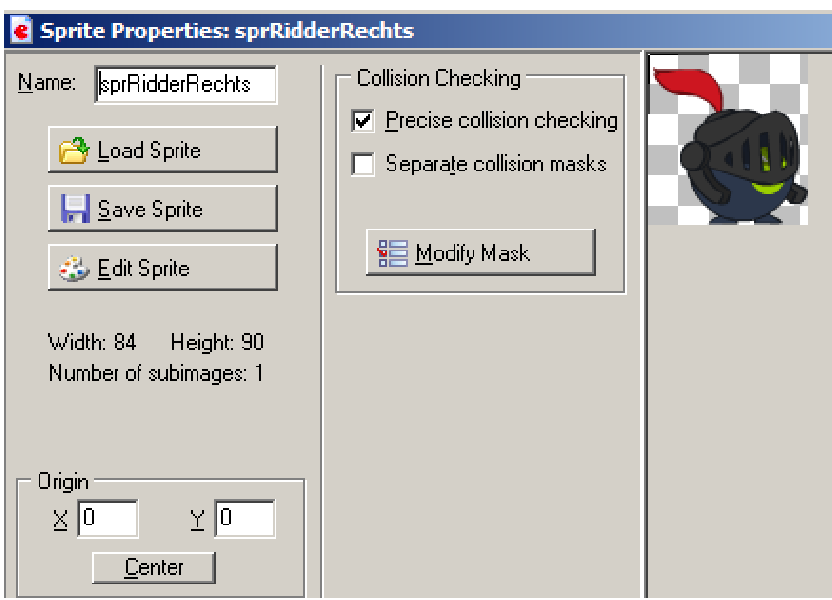
Onze room is tweedimensionaal. Dat betekent dat we de posities uitdrukken in x- en y-waardes en noteren als (x, y). De room heeft een beginpunt dat we oorsprong (origin) noemen. Deze oorsprong noteren we als positie (0, 0) en het is de linkerbovenhoek van de room. Hopelijk herinner je je nog dat een room ook een breedte en hoogte heeft. In GameMaker worden ook alle objecten gedefinieerd door middel van een oorsprong, een breedte en een hoogte. Alle positionering in GameMaker wordt gedaan op basis van de oorsprong van de room en de objecten.

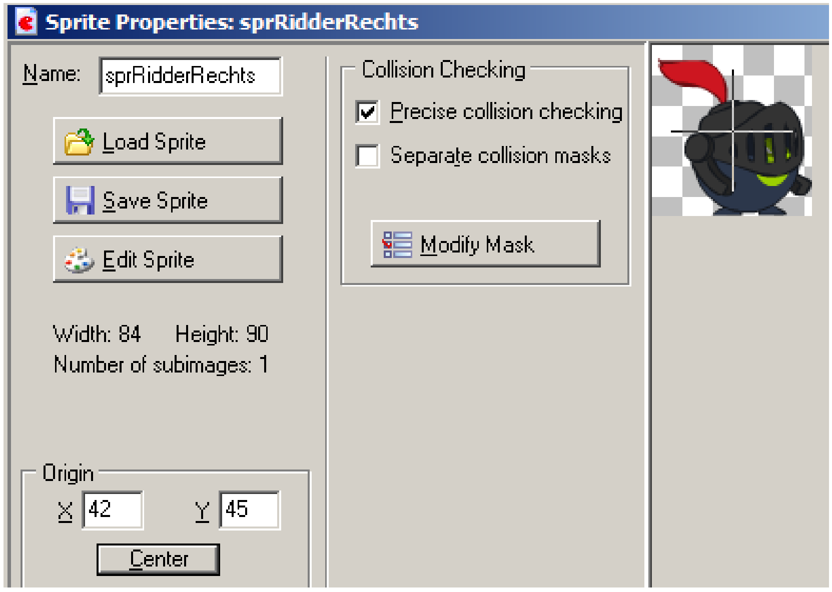
Nu wordt het een beetje vervelend, want als je een object positioneert ten opzichte van de oorsprong van de room dan noemen we dat absolute positionering. Dit klinkt vreemd want we positioneren het object relatief ten opzichte van de room

oorsprong. Albert Einstein zei het al: "Everything is relative." Relatieve positionering is positionering ten opzicht van het object zelf of een ander object.

Ook hier zal een voorbeeld weer helpen. Stel we zetten onze ridder op positie (200, 100), dan staat de oorsprong van onze ridder 200 pixels rechts van en 100 pixels onder de oorsprong van de room.

De oorsprong van een object kun je echter zelf instellen via de bijbehorende sprite en staat default op (0, 0). Hier is (0, 0) de linkerbovenhoek van de sprite. Hieronder zie je twee afbeeldingen waarin we laten zien dat je de oorsprong voor een sprite kunt veranderen.



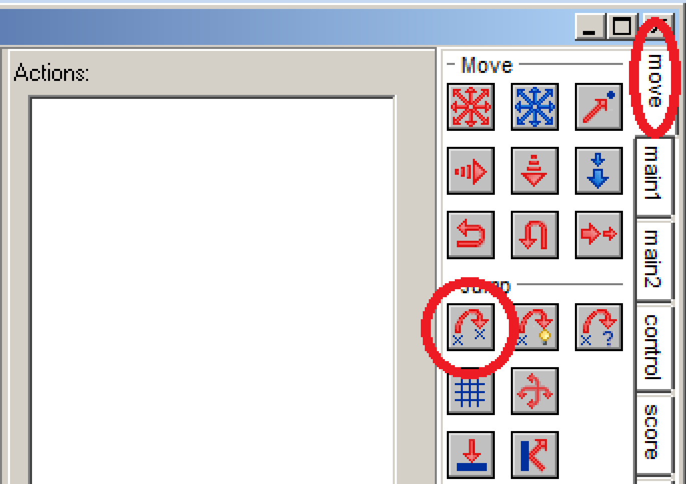


In de onderste afbeelding is er op de *Center* knop gedrukt. Je ziet nu een crosshair in het midden van de sprite staan en de waardes bij *Origin* staan op (42, 45) in plaats van (0, 0) zoals in de bovenste afbeelding. In de bovenste afbeelding staat de crosshair in de linkerbovenhoek zodat je hem nauwelijks ziet.

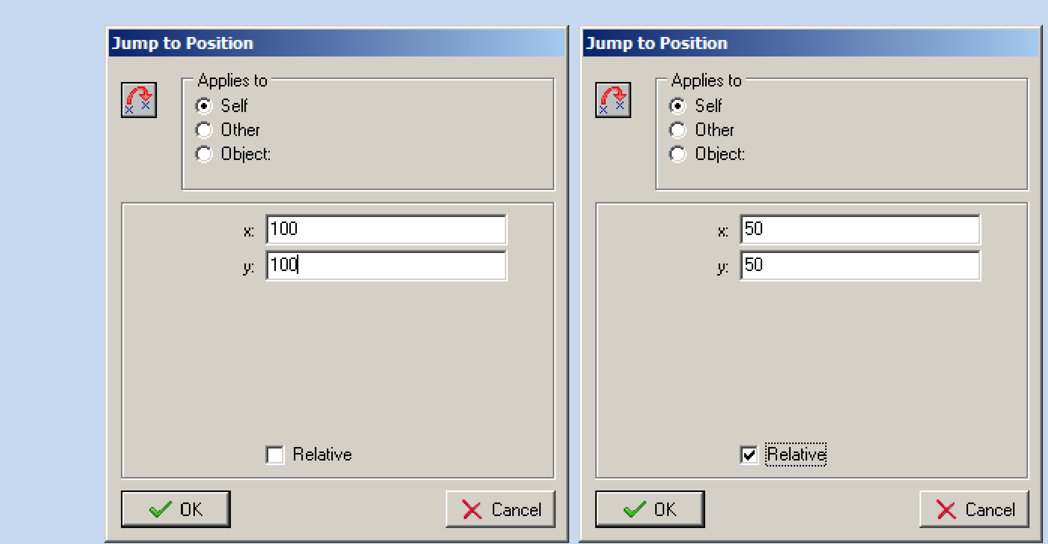
Maar terug naar het voorbeeld. De oorsprong van onze ridder staat dus 200 pixels rechts van en 100 pixels onder de oorsprong van de room. Als we de ridder nu een nieuwe absolute positie van (300, 150) geven dan ligt de oorsprong van de ridder nu dus 300 pixels rechts van en 150 pixels onder de oorsprong van de room. Maar als we een relatieve verplaatsing van (100, 50) ten opzicht van zichzelf doen, dan bereiken we hetzelfde.

opgave 3.5

Laten we eens met absolute en relatieve positionering gaan spelen. Hiervoor gaan we de *Mouse* events *Left pressed* en *Right pressed* gebruiken. Aan beide events koppelen we de action *Jump to Position* uit de *move* tab. Waar deze action zit zie je in de volgende afbeelding.



Voor *Mouse - Left pressed* (linker afbeelding hierna) doen we absolute positionering op (100, 100) en voor *Mouse - Right pressed* (rechter afbeelding hierna) relatieve positionering op (50, 50). Dat ziet er als volgt uit.



Run nu je spel en kijk wat er gebeurt als je de ridder met de linker of rechter muisknop aanklikt.

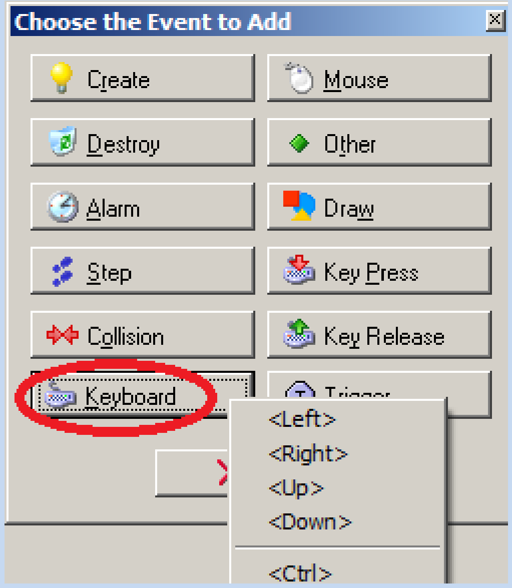
Hopelijk maakt dit het verschil tussen absolute en relatieve positionering duidelijk. Is het je trouwens opgevallen dat nu je bij de y-waarde een relatieve verplaatsing van 50 invult dat de ridder naar beneden beweegt? Misschien zou je verwachten dat de ridder bij een verplaatsing met een positieve waarde (hier dus met +50) naar boven zou bewegen. Dat is dus niet het geval. Als je een object naar boven wilt verplaatsen moet je een negatieve waarde gebruiken. Dat is logisch omdat de oorsprong van de room (0, 0) is en in de linker bovenhoek ligt. Hier moet je mogelijk even aan wennen.

Waar waren we ook alweer mee bezig? O ja, we wilden het derde probleem oplossen: "De ridder kan niet diagonaal lopen door twee pijltjestoetsen in te drukken." Stiekem heb je net de action, die je daarvoor nodig hebt, al gebruikt. Aan de slag dan maar weer.

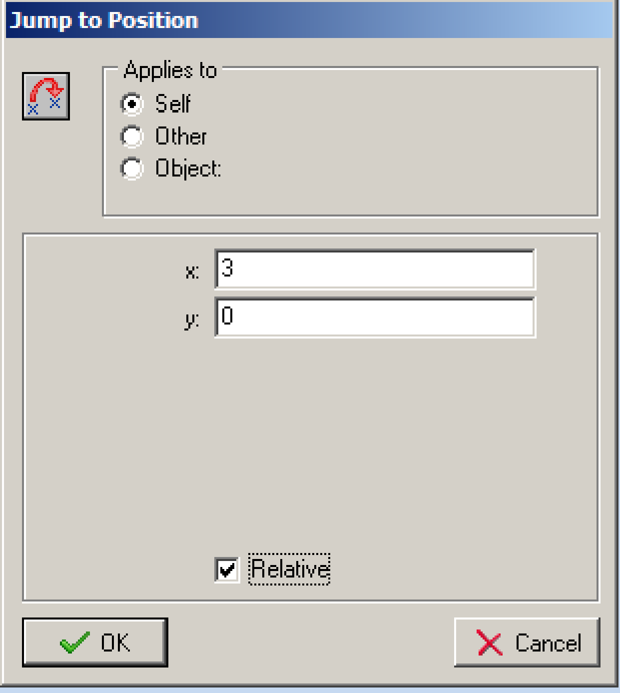
opgave 3.6

De andere manier van besturen is zo anders dat je alle events uit *objRidder* kunt verwijderen. Doe dat eerst dan kun je met een schone lei beginnen.

We gaan nu geen gebruik maken van *Key Press* en *Key Release* events maar van *Keyboard* events. In de afbeelding hierna zie je waar die zitten.



Laten we beginnen met het *Keyboard - <Right>* event. Aan dit event koppelen we de *Jump to Position* action die we in de vorige opgave ook al gebruikt hebben. Net als bij de vorige manier van besturen gaan we onze ridder een speed van 3 geven. Dit doen we in dit geval door de ridder relatief 3 pixels de goede kant in te laten springen. Om naar rechts te bewegen, moeten we dus 3 pixels optellen bij de huidige x-waarde van de ridder. De ridder mag niet naar boven of naar beneden bewegen, dus de y-waarde laten we op 0 staan. Omdat we relatief verplaatsen betekent dit dat er in de y-richting niets gebeurt. Dat ziet er als volgt uit.



Maak het besturen van de ridder nu zelf af voor de andere richtingen. Zet ook de sprite wisselingen terug bij het naar links en rechts bewegen. Run vervolgens je programma en kijk of alles werkt. Kijk dus ook of je nu diagonaal kunt bewegen.

Hopelijk heb je er meteen aan gedacht dat je een negatieve waarde moet gebruiken om omhoog te bewegen en heb je overal *Relative* meteen aangevinkt. Zo niet dan heb je het hopelijk gecorrigeerd.

Voordat we het vierde probleem "de ridder kan de wereld uitlopen" op gaan lossen gaan we de ridder wat geanimeerder maken. Dit doen we door de sprites van de ridder uit meerdere afbeeldingen te laten bestaan en die af te spelen terwijl de ridder beweegt. Hiervoor hoeven we alleen dingen in de sprites aan te passen, de objecten blijven onveranderd.

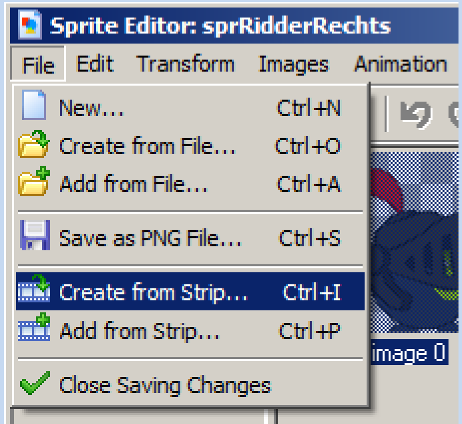
opgave 3.7

Laten we met *sprRidderRechts* beginnen. Open het sprite venster en druk op de *Load Sprite* knop. Vervang de huidige afbeelding door de inhoud van *ridder_rechts_strip.png*. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort. In dit PNG bestand zitten 8 verschillende afbeeldingen van onze ridder in plaats van maar 1. Laad het bestand en kijk eens wat er gebeurd is wanneer je op de *Edit Sprite* knop klikt.

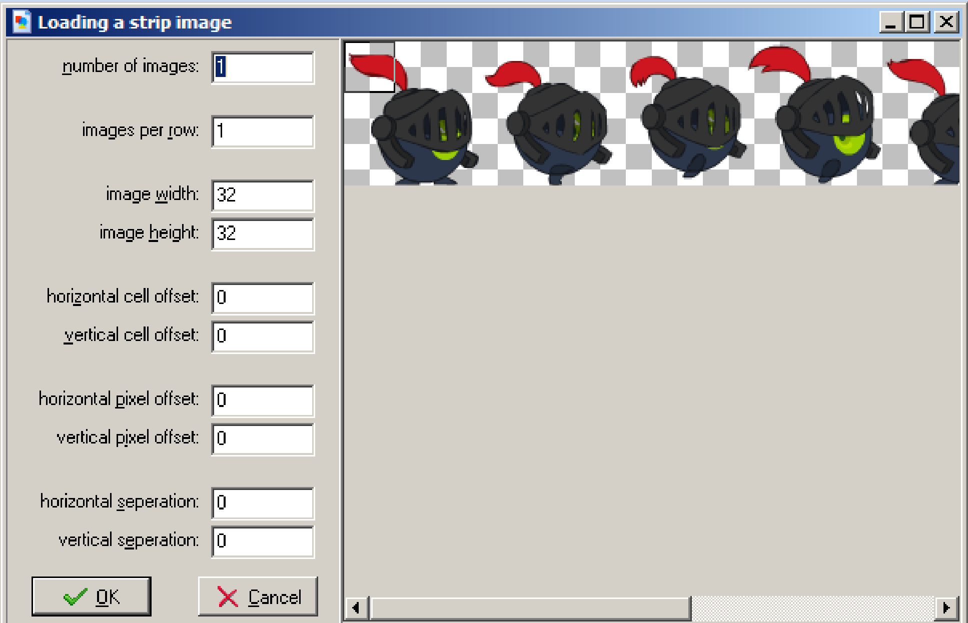
GameMaker heeft nu de hele strip als 1 afbeelding opgeslagen[2]. Dat is natuurlijk niet de bedoeling. We willen niet 1 maar 8 afbeeldingen. Dit kunnen we doen door *Create from Strip* te kiezen in het *File* menu van de sprite editor zoals hierna is afgebeeld.

---

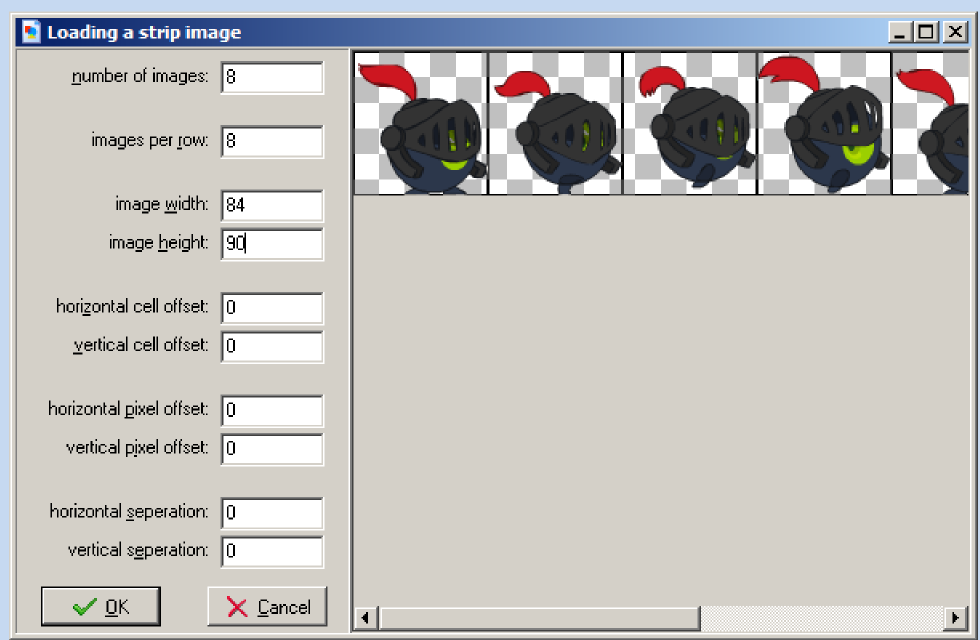
[2] In sommige gevallen herkent GameMaker dat het een strip is en splitst deze automatisch op in 8 aparte afbeeldingen. Het is de auteur helaas nog steeds niet bekend onder welke omstandigheden dit precies gebeurt.



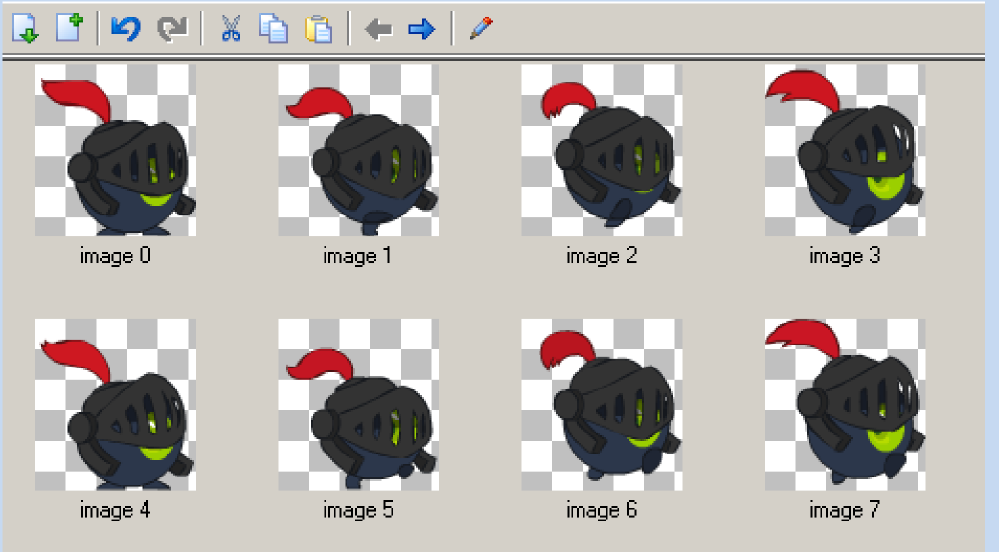
Als je dan *ridder_rechts_strip.png* laadt, krijg je de volgende interface te zien.



We weten dat er 8 afbeeldingen in zitten en kunnen erachter komen dat elke afbeelding 84 pixels breed en 90 hoog is. Deze informatie kunnen we als volgt invullen.



Als je op de *OK* knop klikt, dan krijg je het volgende te zien als het goed is.

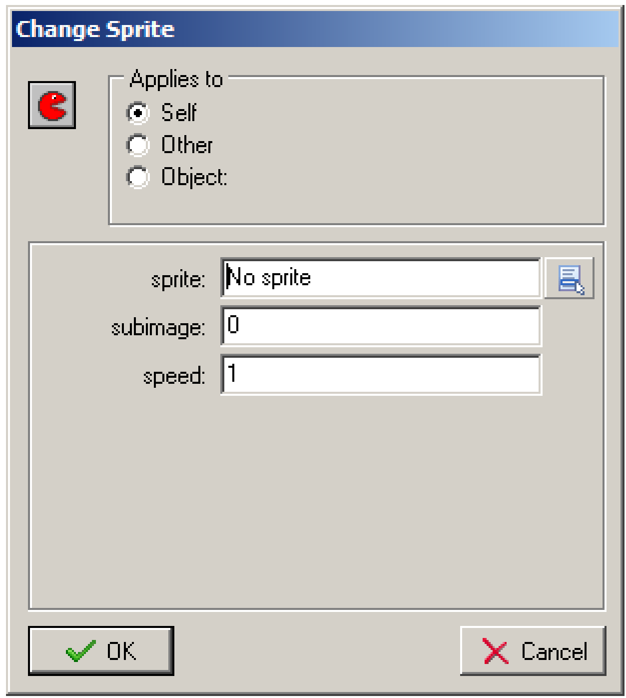


Je hebt nu dus in één sprite meerdere afbeeldingen. We zeggen dan dat de sprite sub-images heeft. GameMaker kan deze sub-images als een strip afspelen. Je kunt in de sprite editor aan de linkerkant de preview aanzetten en kijken hoe het er uitziet.

Als je op de *OK* knop van de sprite editor klikt krijg je in je sprite venster aan de linkerkant te zien dat je nu 8 *subimages* van 84 bij 90 hebt.

Tijd om het spel te runnen en te kijken wat er gebeurt.

Het eerste dat je hopelijk opvalt is dat de animatie stopt als we naar rechts of links bewegen. Na de volgende uitleg ga je dit ook logisch vinden. Herinner je je nog dat je bij de *Change Sprite* action twee dingen onder de keuze van de sprite had staan? Misschien niet, daarom volgt hier nog een keer de dialoog voor de *Change Sprite* action.



Hier zien we de term *subimage* terug. Het getal dat hierachter staat is de index in de strip. Het eerste sub-image staat op index 0 en bij onze ridder staat dus het laatste sub-image op index 7 omdat we in totaal 8 sub-images hebben. Het is doorgaans prima om bij het eerste sub-image te beginnen.

De *speed* is de snelheid waarmee de strip wordt afgespeeld. Als je snelheid 0 invult wordt de strip niet afgespeeld. De snelheid is ook weer afhankelijk van de room speed.

We hebben de *Change Sprite* action gekoppeld aan de *Keyboard - <Right>* en *Keyboard - <Left>* events. Als we de rechter of linker pijltjestoets ingedrukt houden dan treedt het bijbehorende event dus 30 keer per seconde op. Iedere keer wordt onze sprite daardoor teruggezet op sub-image 0 en daarom zien we geen beweging. Dus dat ga je oplossen.

opgave 3.8

In plaats van de *Change Sprite* action te koppelen aan de *Keyboard - <Right>* en *Keyboard - <Left>* events, koppelen we deze action aan de *Key Press - <Right>* en *Key Press - <Left>* events. Dit zou je nu zelf voor elkaar moeten kunnen krijgen. Probeer het eens.

Run vervolgens het spel en kijk of het werkt.

We zeiden hiervoor in de uitleg dat de snelheid waarmee de strip wordt afgespeeld afhangt van de room speed. Dit gaan we testen.

opgave 3.9

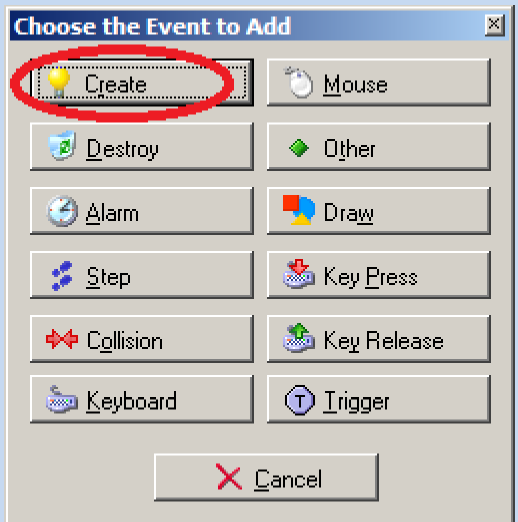
Zet de room speed op 3 in plaats van 30, run het spel en kijk wat er gebeurt.

Zet vervolgens de room speed weer terug op 30.

Tot slot ziet het er raar uit dat onze ridder huppelt terwijl hij niet beweegt.

opgave 3.10

Als we het spel starten dan begint onze ridder al met huppelen. De beste manier om dit te voorkomen is door de ridder stil te zetten op het moment dat hij gecreëerd wordt in de room. Hiervoor bestaat het speciale *Create* event. In de afbeelding hierna zie je waar je dit event vindt.



Zorg nu zelf dat de animatie van de ridder niet wordt afgespeeld als hij niet beweegt door de juist action te koppelen aan de juiste events.

Houd er rekening mee dat we alleen een strip hebben voor het naar rechts en links lopen en niet voor het naar boven en beneden lopen. In de laatste twee gevallen is het dus prima dat de ridder stilstaat.

En dan rest ons nog steeds het vierde probleem: "De ridder kan de wereld uitlopen." Maar dat lossen we in het volgende hoofdstuk op. Ook heb je nog een stukje theorie tegoed over het *Applies to* gedeelte bij actions. Ook daar komen we op terug in het volgende hoofdstuk. Maar eerst.

opgave 3.11

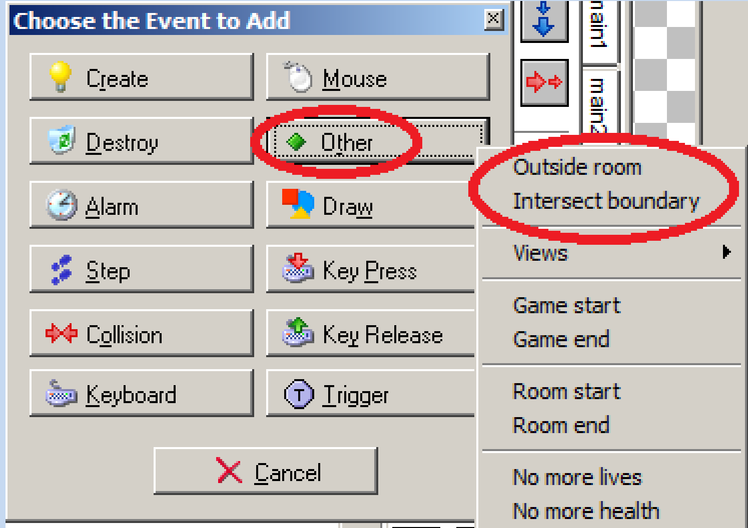
Sla je spel op met *hoofdstuk3.gm81* als naam op je H schijf (netwerk schijf). Je doet dit wederom met Save As uit het File menu van GameMaker.

4. interactie met de wereld

Als we een player character eindelijk zo gek hebben gekregen om in de gamewereld te kruipen willen we natuurlijk niet dat hij er zomaar uit kan wandelen. Een heel eenvoudige manier gaan we nu maken.

opgave 4.1

GameMaker kan detecteren of een object buiten de room is of de rand van de room raakt. Hiervoor worden de *Other – Outside room* en *Other – Intersect boundary* events gebruikt. Je vind ze zoals aangegeven in de volgende afbeelding.



In ons geval nemen we het *Other – Intersect boundary* event en koppelen daar de action *Jump to Start* aan. Dit zet het object terug op de positie waarop het in de room gecreëerd is.

Voeg het event en de action toe aan *objRidder*, run het spel en kijk of het werkt.

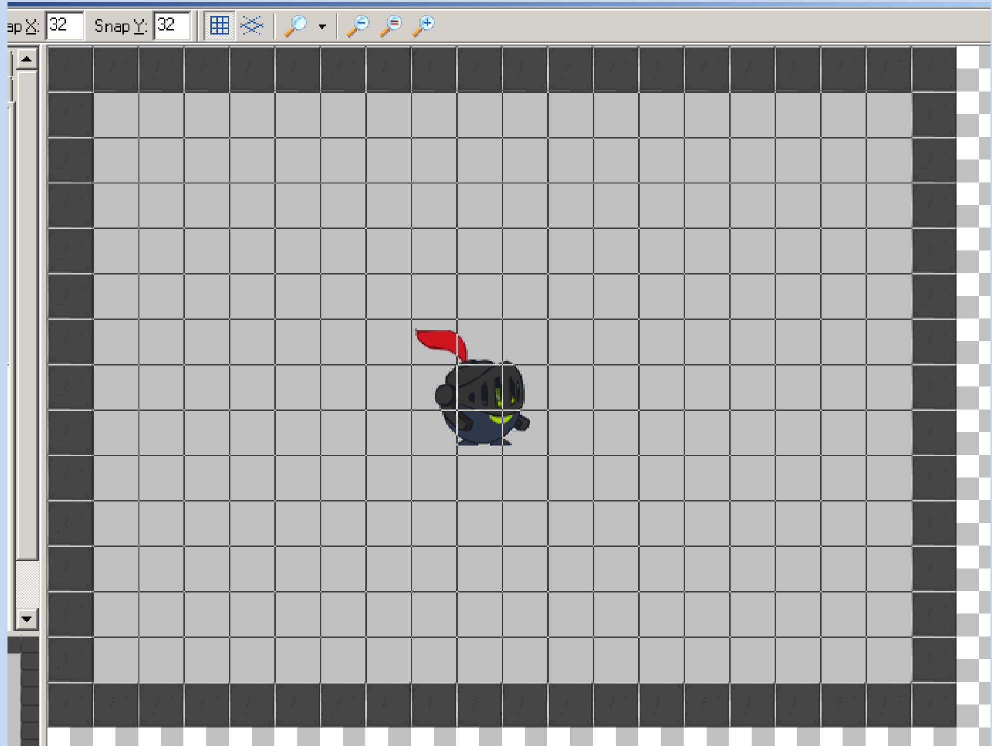
Dit is echter geen fijne oplossing voor de speler omdat hij steeds weer opnieuw moet beginnen als hij tegen de rand van de room aanloopt. Wat we eigenlijk willen is de ridder tegen de rand van de room laten botsen alsof het een muur is. Als je in het echt tegen de muur aan loopt dan "stuiter" je een stukje terug. Dit kan in GameMaker ook. Je gebruikt hiervoor de *Bounce* action. Het probleem bij deze action is echter dat je alleen kunt stuiteren tegen objecten en de rand van de room is geen object. Daarom zie je in de meeste spellen dat de makers ervan eigen gemaakte muren in het spel zetten. De rand van de room is een prima plek om muren neer te zetten. Laten we daar zelf eens mee aan de slag gaan.

opgave 4.2

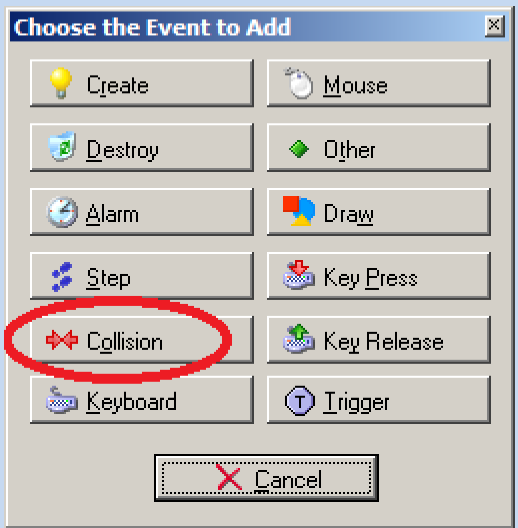
Doe de volgende dingen:

- Delete het *Other - Intersect boundary* event.
- Maak een sprite die *sprMuur* heet en laad daarin de afbeelding *muur.png*. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort.
- Maak een object dat *objMuur* heet en koppel hieraan *sprMuur*.
- Zet het object *objMuur* langs alle randen van de room. Door de shift toets ingedrukt te houden en met de linker muisknop ingedrukt te slepen kun je gemakkelijk een hele rand van muur objecten voorzien.

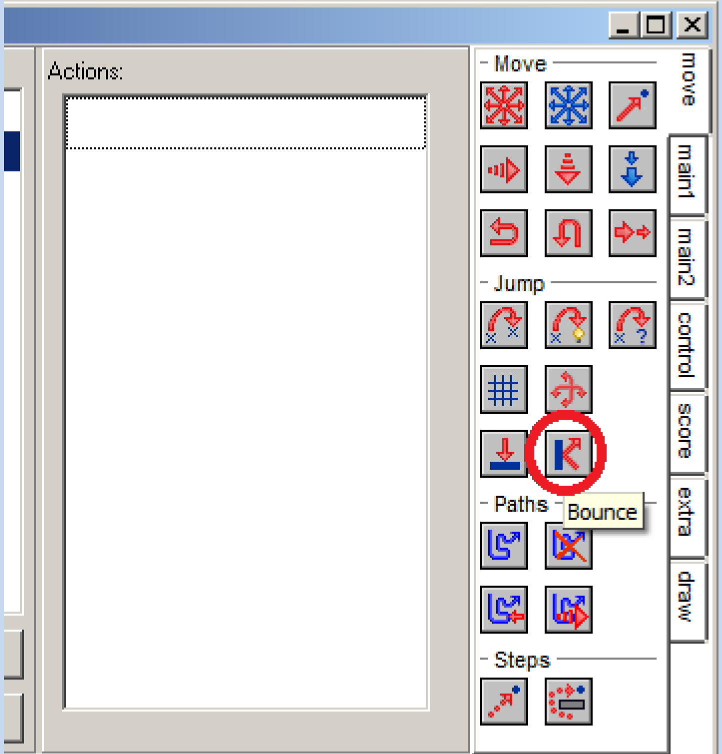
Dit zou er als volgt uit moeten zien.



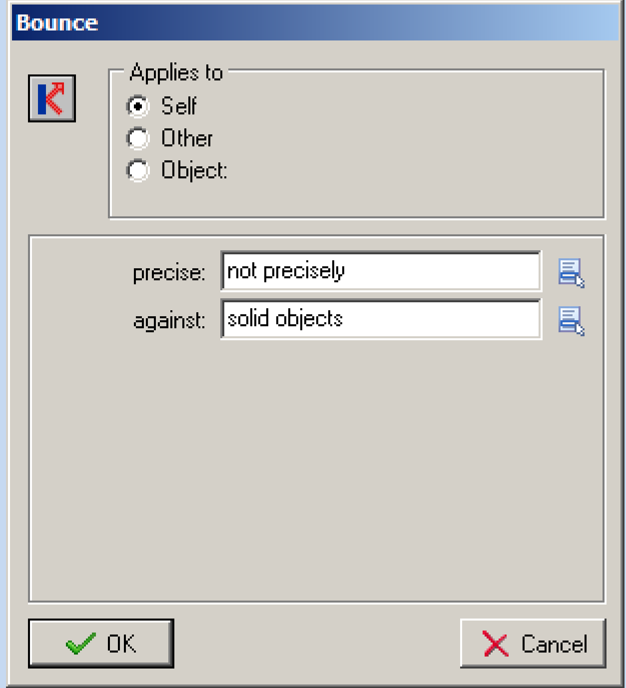
Nu gaan we ervoor zorgen dat er iets gebeurt als onze ridder tegen een stuk muur aanloopt. Hiervoor gebruiken we het *Collision* event. In de volgende afbeelding zie je waar je het event vindt.



Voeg het event in *objRidder* toe en koppel er de action *Bounce* aan. Deze action vind je in de *move* tab die je intussen zou moeten kunnen vinden. De afbeelding hierna laat zien waar de *Bounce* action staat.



Als je de *Bounce* action toevoegt krijg je de volgende dialoog.



Bij *Applies to* laat je de optie *Self* staan.

Als je bij *precise* de waarde *not precisely* kiest, dan ziet het botsen er alleen natuurlijk als je tegen een recht vlak botst. Aangezien we alleen maar mooie rechte muren in ons spel hebben is dat voor nu prima. Je kunt de waarde ook op *precisely* zetten. In dat geval ziet de botsing er altijd natuurlijk uit. Dit gaat wel ten koste van rekenkracht van GameMaker tijdens het runnen van het spel.

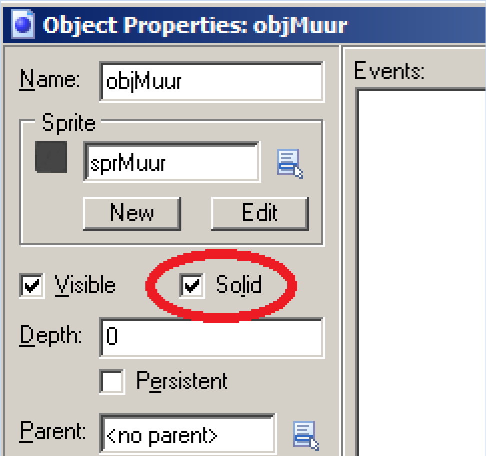
Daarna kunnen we aangeven of we moeten stuiteren tegen alleen **solid** objecten (waarde *solid objects*) of ook tegen niet solid objecten (waarde *all objects*). Standaard zijn objecten niet solid. Het wel of niet solid zijn kan handig zijn als je bepaalde objecten wel door het ene object wilt laten bewegen maar niet door een ander.

Kies om te beginnen de waarde *solid objects* bij *against* en kijk wat er gebeurt. Dus run je spel en observeer.

Je ziet dat de ridder nog steeds door de muur loopt en dat klopt, want standaard zijn objecten niet solid. Onze muur dus ook niet en we zeggen, bij de *Bounce* action, dat we alleen stuiteren tegen solid objecten. Oftewel, GameMaker doet precies wat we hem verteld hebben. Er zijn nu twee mogelijkheden. Of we zeggen bij *against* van de *Bounce* action dat er tegen alle objecten gestuiterd moet worden. Of we maken muren solid.

Het eerste is gemakkelijk. Het tweede laten we even zien.

Open het object venster voor *objMuur* en vink Solid aan zoals je in de volgende afbeelding ziet.



Run nu je spel weer en kijk wat er gebeurt.

Als het goed is kan je ridder nu niet meer uit de room lopen. Ook niet als je diagonaal loopt. In veel gevallen maakt het bij het stuiteren niet uit of je objecten nu solid maakt of niet. Er zijn echter situaties waarin je door het slim gebruiken van solid problemen kunt oplossen. Het is doorgaans een goede gewoonte om objecten solid te maken als je er in de echte wereld ook niet doorheen zou kunnen lopen. In ons geval zouden we dus zowel *objRidder* als *objMuur* solid maken.

We hebben nu gezorgd dat onze ridder iets niet kan, namelijk uit de room lopen. Het is natuurlijk veel leuker om te zorgen dat onze ridder juist iets wel kan. Onze ridder is namelijk dol op gebraden kip. Daarom gaan we nu een aantal gebraden kippen in onze room neerzetten zodat de ridder ze op kan peuzelen.

opgave 4.3

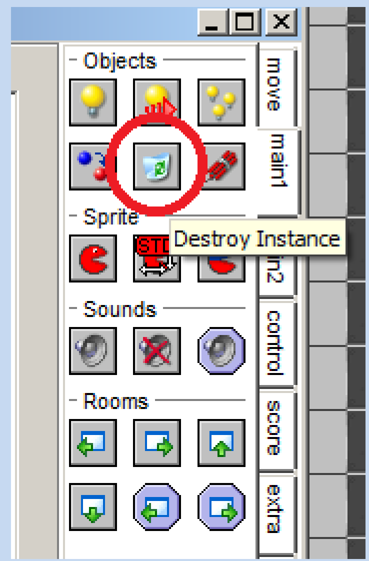
Doe de volgende dingen:

- Maak een sprite die *sprGebrKip* heet en laad daarin de afbeelding *gebradenkip.png*. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort.
- Maak een object dat *objGebrKip* heet en koppel hieraan *sprGebrKip*. Maak het object solid.
- Zet vier keer *objGebrKip* in de room.

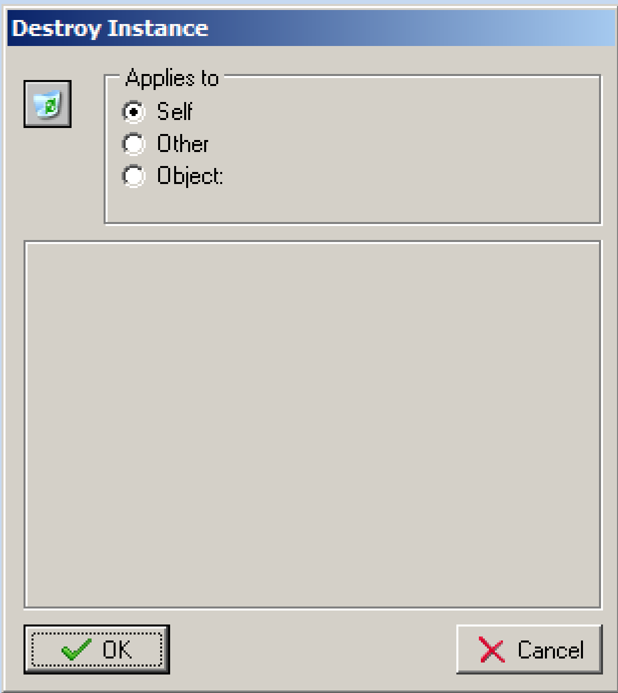
Dit zou er als volgt uit kunnen zien.

Nu gaan we ervoor zorgen dat er iets gebeurt als onze ridder tegen een gebraden kippetje aanloopt. Hiervoor gebruiken we weer het *Collision* event. Wat er moet gebeuren als onze ridder tegen een gebraden kip loopt is dat de kip moet verdwijnen. De ridder eet hem per slot van rekening op.

Voeg het *Collision* event in *objRidder* toe en koppel er de action *Destroy Instance* aan. Deze action vind je in de *main1* tab die je intussen zou moeten kunnen vinden. De afbeelding hierna laat zien waar de *Destroy Instance* action staat.



Als je de *Destroy Instance* action toevoegt krijg je de volgende dialoog.



Je hoeft bij deze action niet veel in te stellen, maar we kunnen nu wel goed laten zien wat elk van de opties in het *Applies to* gedeelte precies doet. Het stuk theorie hierover vind je na deze opgave.

Laat allereerst *Applies to* op *Self* staan, run de game en kijk wat er gebeurt.

Dat is een sterk stukje kip, het eet zowaar onze ridder op. Dat is natuurlijk niet wat we willen maar wat we GameMaker wel verteld hebben. We "programmeren" in *objRidder* en zeggen dat *Self* (de ridder dus) vernietigd moet worden.

Zet *Applies to* op *Object* en kies als object *objGebrKip*, run de game en kijk wat er gebeurt.

Zo, die ridder heeft honger. Hij eet ze allemaal in één keer op. Het lijkt al meer op wat we willen maar het is nog steeds niet goed. Toch is het wel wat we GameMaker verteld hebben. We zeggen namelijk dat GameMaker dingen die een *objGebrKip* zijn moet vernietigen en dat zijn er in onze room toevallig vier.

Zet *Applies to* op *Other*, run de game en kijk wat er gebeurt.

Kijk dat is precies wat we willen. Als de ridder tegen een gebraden kip botst moet alleen de kip waar de ridder tegenaan botst verdwijnen.

En dan nu het beloofde stuk theorie over het *Applies to* gedeelte van actions. Het is je hopelijk opgevallen dat veel actions zo'n gedeelte hebben. Daarom is het belangrijk dat je goed snapt wat het doet. Zoals gezegd, geeft het *Applies to* gedeelte van een action aan waarop de action van toepassing is. Voor we precies en in detail uit kunnen leggen hoe dit werkt is het belangrijk dat je eerste het begrip instantie (instance) snapt.

In de vorige opdracht heb je vier gebraden kippen in de room gezet. Zonder het toen te beseffen heb je daarmee meerdere instanties van het object *objGebrKip* aangemaakt. Elk voorkomen (occurrence) van een object in een room heet een instantie.

Iedere instantie van een object gedraagt zich hetzelfde maar heeft typisch wel andere eigenschappen. Zo zullen verschillende instanties van hetzelfde object typisch een andere positie in de room hebben. De vier kippen staan immers allemaal op een eigen plek in de room. Ook kunnen verschillende instanties bijvoorbeeld een verschillende snelheid hebben.

Als er in het object echter staat dat het indrukken van de rechter pijltjestoets betekent dat er naar rechts bewogen wordt dan zullen alle instanties naar rechts bewegen bij het indrukken van deze toets.

opgave 4.4

Zet nog een ridder in je room en run je spel. Kijk wat er gebeurt als je op de pijltjestoetsen drukt.

Stop je spel weer en kijk in je room venster. Ga met je muis boven de instanties van beide ridders staan en kijk welk *id* ze hebben. Het ID vind je in de balk onder je room. Als het goed is hebben de ridders een verschillend ID. Dit is het **instance ID** en hieraan zie je dus dat de ridders voor GameMaker daadwerkelijk verschillende instanties van hetzelfde object zijn.

Laat de tweede ridder nog even in de room staan.

Nu je weet wat een instantie is kunnen we uitleggen hoe de opties van het *Applies to* gedeelte werken.

- De optie *Self* betekent dat de action wordt toegepast op iedere instantie van het object waarin je aan het "programmeren" bent. In ons geval dus iedere instantie van *objRidder*.
- De optie *Other* betekent dat de action wordt toegepast de instantie van het object waarmee het object waarin je aan het "programmeren" bent interacteert. In ons spel tot nu toe is de optie *Other* dus zinloos, want alleen *objRidder* is bij de events betrokken.
- De optie *Object* betekent dat de action wordt toegepast op alle **instanties** van een zelf te kiezen object.

opgave 4.5

Zet in *objRidder* bij de *Destroy Instance* action van het event *Collision* met *objGebrKip* de waarde van *Applies to* weer op *Self*. Run vervolgens het spel en laat één ridder tegen een kip aanlopen en de andere niet. Zorg dat je goed snapt wat er gebeurt.

Verwijder nu de tweede ridder en speel met de opties bij *Applies to* van de *Destroy Instance* action zodat je goed snapt wat de verschillende opties precies doen.

En tot slot zoals in elk hoofdstuk.

opgave 4.6

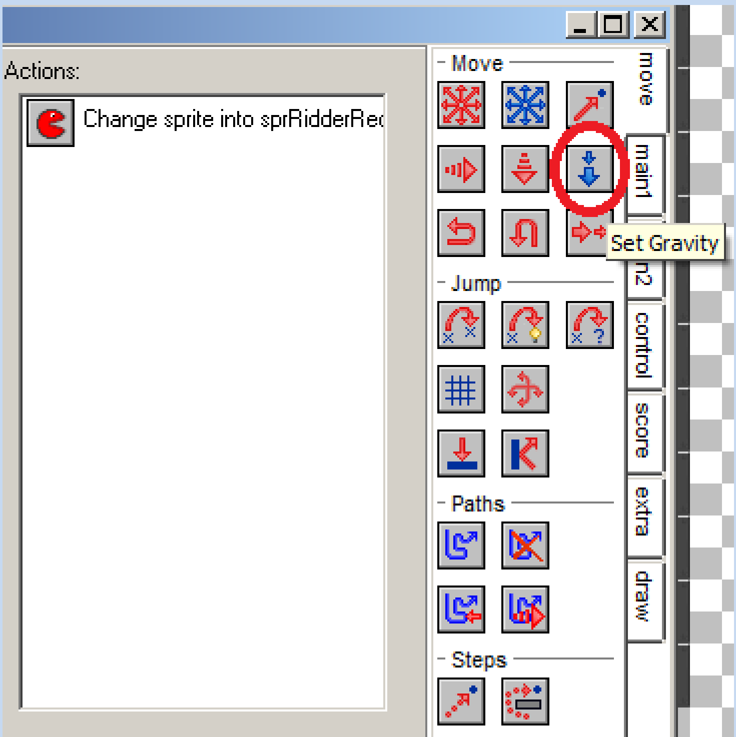
Sla je spel op met *hoofdstuk4.gm81* als naam op je H schijf (netwerk schijf).

5. zwaartekracht

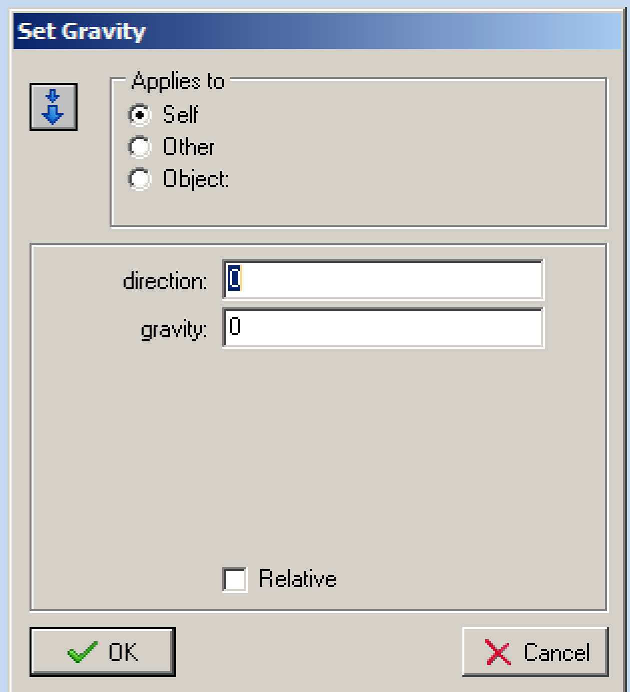
Net als in de echte wereld kun je met GameMaker in je gamewereld met zwaarte-kracht werken. Met name in platform games zorgt dit ervoor dat het bewegen en springen van je player character er natuurlijk uitziet. Laten we onze ridder eens blootstellen aan zwaartekracht.

opgave 5.1

Open het object venster voor *objRidder*. Als het goed is heb je daar al een *Create* event in staan. Je gaat de actions voor dit event uit-breiden. Voeg een *Set Gravity* action toe. Deze vind je in de *move* tab zoals je in de volgende afbeelding ziet.



Als je deze action toevoegt krijg je de volgende dialoog te zien.



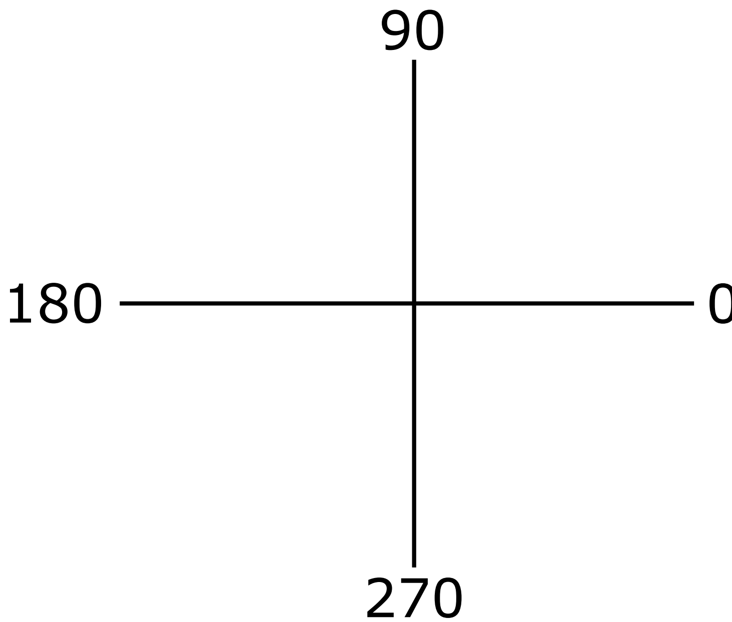
Bij *Applies to* zou je nu zelf moeten kunnen beslissen wat je daar moet kiezen.

De *direction* geeft de richting aan waarin de zwaartekracht werkt. De eenheid is graden. Je kunt hier dus een waarde vanaf 0 tot 360 graden invullen. Als je grotere waardes invult dan wordt er modulo 360 gewerkt. Anders dan in de echte wereld kun je in GameMaker dus zwaartekracht in elke willekeurige richting creëren. Dit is onder andere handig bij reversed gravity of bij space games. Gebruik voorlopig 270 als waarde. Na deze opgave leggen we uit hoe graden werken in GameMaker.

De *gravity* is het aantal pixels dat elke event puls bij de huidige snelheid van een instantie wordt opgeteld. Je kunt hier decimale getallen gebruiken. Let wel goed op dat het decimale scheidingsteken een punt (.) is en geen komma (,). We programmeren immers in het Engels en in die taal worden decimalen gescheiden door een punt. Kies hier bijvoorbeeld de waarde 0.5.

Run het spel en kijk wat er gebeurt.

We zien de ridder nu inderdaad naar beneden vallen door de zwaartekracht. Als richting hebben we 270 graden ingesteld. Dat vind je misschien niet logisch. Laten we daarom eens kijken hoe GameMaker met richting in graden omgaat. Je ziet dit uitgelegd in Figuur 1. Hier zie je het assenstelsel van GameMaker met daarbij de graden die bij iedere as horen. Zoals je ziet is 270 graden inderdaad naar beneden.

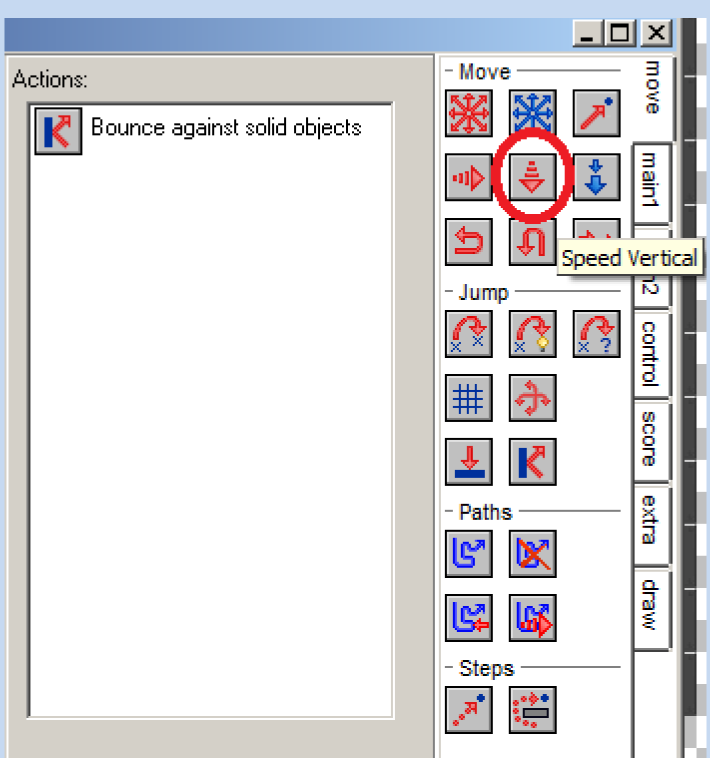


Figuur 1: het assenstelsel van GameMaker

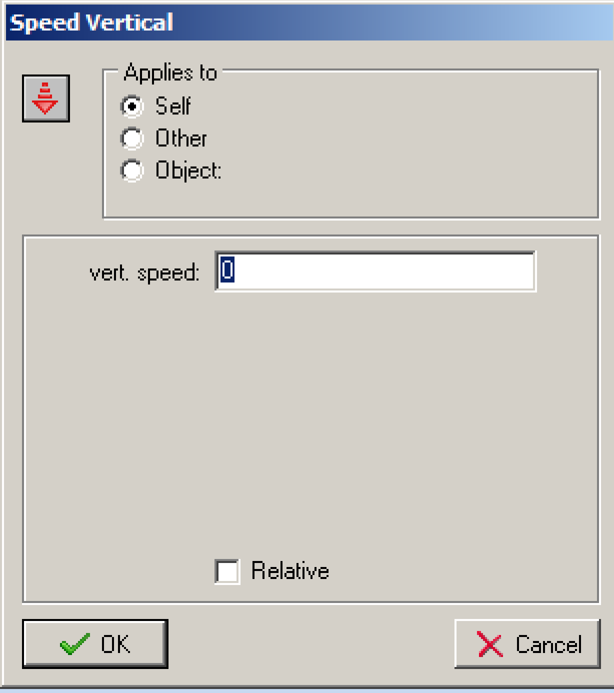
Leuk, nu hebben we een stuiterende ridder. Niet echt wat we willen, maar wel wat we GameMaker verteld hebben. Immers, als de ridder een muur raakt moet hij stuiteren. Wat er onder water door GameMaker gebeurt als je zwaartekracht aanzet is dat er automatisch snelheid opgebouwd wordt in instanties van het betreffende object. Wat we dus zouden kunnen doen is de snelheid van de ridder op 0 zetten als hij een muur raakt.

opgave 5.2

Open het object venster voor *objRidder*. Voeg bij het *Collision* met *objMuur* event de action *Speed Vertical* toe. Deze vind je in de *move* tab zoals je in de volgende afbeelding ziet.



Als je deze action toevoegt krijg je de volgende dialoog te zien.



Bij *Applies to* zou je nu zelf moeten kunnen beslissen wat je daar moet kiezen.

Aangezien we de ridder willen stoppen is het logisch dat we de verticale snelheid (de ridder valt immers naar beneden) op 0 zetten.

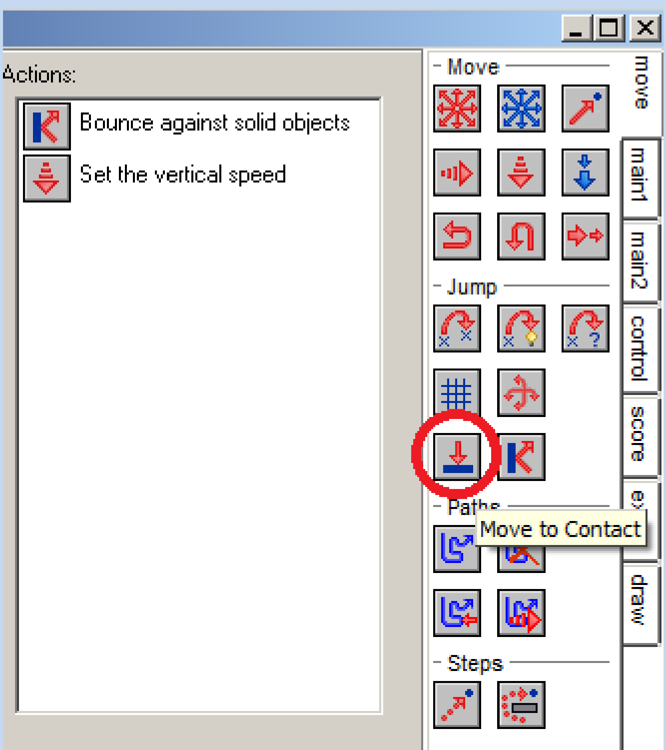
Run het spel en kijk wat er gebeurt.

Je ziet dat de ridder nu inderdaad niet meer stuitert. Dat is mooi, maar heb je ook geprobeerd te lopen? Zo nee, probeer dat dan eens. Je zult merken dat de ridder niet van zijn plaats komt. Wat wel werkt is diagonaal lopen. Bij diagonaal lopen trek je de ridder telkens een beetje los van de ondergrond en dan kun je bewegen. Als het ridder object namelijk op een muur object staat dan treedt er iedere event puls een collision op waardoor het systeem overbelast raakt. Hierdoor kan de ridder dus niet meer bewegen. Dit kunnen we op twee manieren oplossen en we gaan ze allebei bekijken.

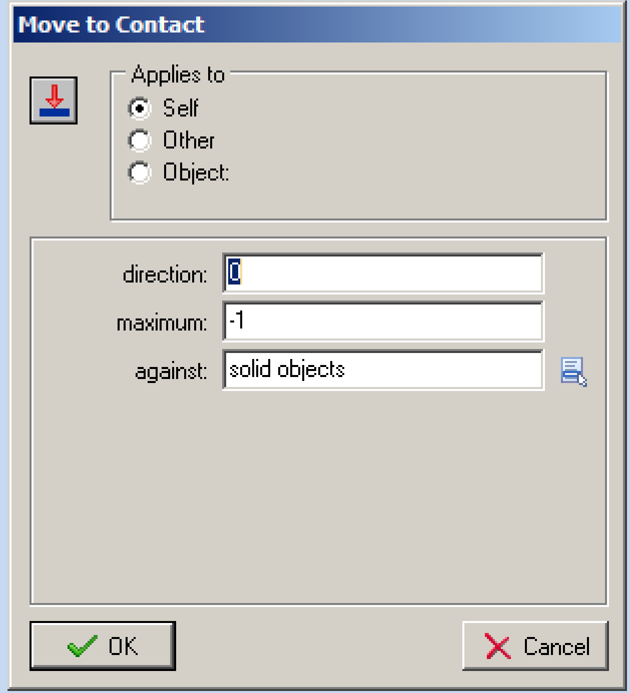
opgave 5.3

Bij de eerste oplossing gebruiken we de *Move to Contact* action.

Open het object venster voor *objRidder*. Voeg bij het *Collision* met *objMuur* event de action *Move to Contact* toe. Deze vind je in de *move* tab zoals je in de volgende afbeelding ziet.



Als je deze action toevoegt krijg je de volgende dialoog te zien.



Kies zelf de juiste waardes voor *Applies to* en *direction*.

Bij *maximum* moet je het maximale aantal pixels invullen dat de instantie mag bewegen om in contact te komen met de andere instantie. Een afstand is altijd een positief getal. Toch zie je hier standaard -1 staan. Een negatieve waarde betekent in dit geval dat de instantie zo ver mag bewegen als nodig is. Het is een ietwat vreemde schrijfwijze voor oneindig.

Je kunt de *Bounce* action laten staan maar feitelijk is deze nu overbodig. Het is daarom beter om hem weg te halen.

Run het spel en kijk wat er gebeurt.

Nu werkt het nog niet. Dat is nou vervelend. Kun je zelf bedenken wat er aan de hand is?

..@..#..$..%.. ..@..#..$..%.. ..@..#..$..%.. ..@..#..$..%..

Natuurlijk, de zwaartekracht staat nog aan. Dus ook al zetten we de verticale snelheid op 0, onze ridder zal toch weer naar beneden vallen als we hem hebben losgemaakt van de muur door de *Move to Contact* action. Dus we moeten ook de zwaartekracht uitzetten. Doe dat en run het spel weer om te kijken of het nu wel werkt

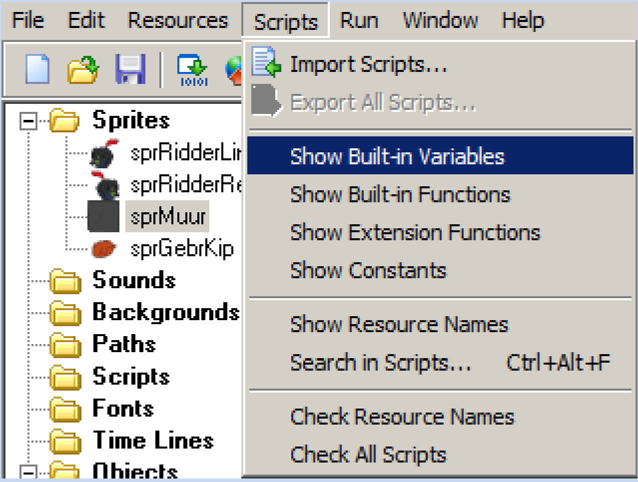
Zo nu kan onze ridder weer lopen en dat vindt hij wel zo fijn. We hebben echter wel een ander probleem geïntroduceerd maar daar kijken we naar nadat we de

tweede oplossing voor het vastzitten hebben gezien. De tweede oplossing introduceert namelijk hetzelfde probleem.

opgave 5.4

Bij de tweede oplossing gebruiken we de speciale variabelen *xprevious* en *yprevious*.

GameMaker houdt voor ons voor elke instantie de positie bij. We kunnen deze altijd bekijken. In GameMaker wordt heel veel bijgehouden en gebruikt daarvoor variabelen. Welke variabelen GameMaker allemaal bijhoudt kun je zien als je in het menu naar *Scripts* gaat en daar *Show Built-in Variables* aanklikt zoals in de volgende afbeelding is aangegeven.

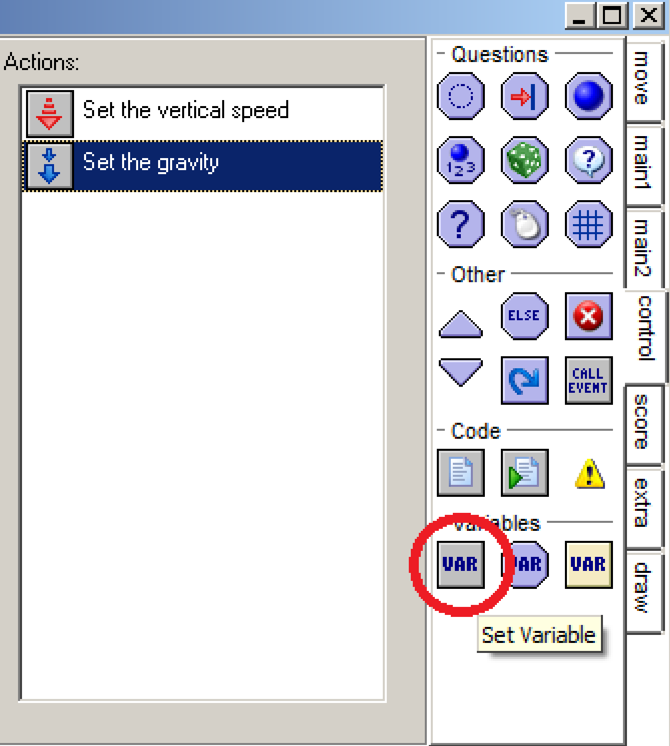


Helemaal onderaan de lijst zie je de variabelen staan waar we mee gaan werken. De variabelen *x* en *y* geven de huidige positie van een instantie aan, de variabelen *xprevious* en *yprevious* de vorige positie van de instantie.

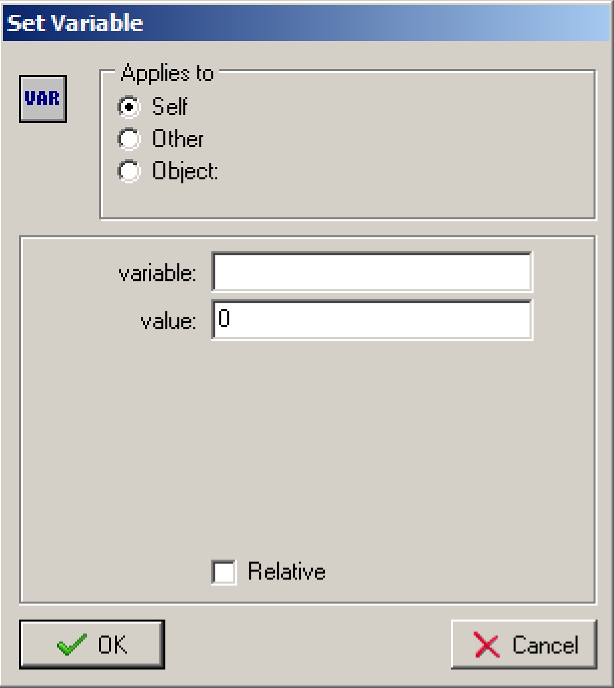
Op het moment dat de ridder met de muur botst kunnen we de huidige x en y waardes dus vervangen door de vorige x en y waardes. Daarmee komt de ridder weer los van de muur. Uiteraard moeten we bij deze oplossing ook de verticale snelheid en zwaartekracht op 0 zetten.

Om de x en y waardes aan te passen gebruiken we de *Set Variable* action in de *control* tab. Omdat we twee variabelen aan moeten passen moeten we de action twee keer gebruiken. Voor we dat doen gooi je eerst de *Move to Contact* action weg. Die hoort namelijk bij de andere oplossing. In de volgende afbeelding zie je waar je de *Set*

Variable action kunt vinden.



Als je deze action toevoegt krijg je de volgende dialoog te zien.



Kies zelf de juiste waarde voor *Applies to*.

Bij *variable* moet je de naam van de variabele invullen die je wilt veranderen. Laten we beginnen met x. Bij *value* komt de nieuwe

waarde van de variabele te staan. In dit geval *xprevious*. We vinken *Relative* niet aan want we willen x gewoon gelijk maken aan *xprevious*.

Doe dit zelf voor de variabele y.

Als het goed is staan de actions voor de verticale snelheid en zwaartekracht er nog, dus run het spel en kijk wat er gebeurt.

Valt je bij de tweede oplossing iets op? De ridder staat niet mooi op de muur. Dat komt omdat *xprevious* en *yprevious* afhankelijk zijn van de snelheid van de instantie. De mooiste oplossing is daarom de eerste, maar deze kan bewerkelijk zijn als de collision niet altijd in dezelfde richting optreedt. Je moet dan namelijk eerst de richting bepalen om te weten wat je bij de *Move to Contact* action als *direction* in moet vullen.

Goed, we zeiden dat we een nieuw probleem hebben geïntroduceerd. Weet je al welk? Wat gebeurt er als je op de pijltjestoets naar boven drukt? Precies, onze ridder loopt naar boven en valt niet meer naar beneden. En dat klopt. We hebben immers de zwaartekracht uitgezet. De oplossing is simpel. Als we op de pijltjestoets naar boven drukken dan moet de zwaartekracht weer aan.

opgave 5.5

Bouw *objRidder* weer om zodat de *Move to Contact* action wordt gebruikt als de ridder een muur raakt.

Zorg nu dat de zwaartekracht weer aan wordt gezet als de ridder naar boven beweegt.

Als je dat voor elkaar hebt dan merk je dat onze ridder niet echt bij de bovenste kippen kan. Speel met de y waarde van de *Jump to Position* action bij het *Keyboard - <Up>* event zodat de ridder ook de bovenste kippen op kan eten.

Hopelijk heb je gemerkt dat als de ridder tegen de bovenste muur aankomt er rare dingen gebeuren. Zo niet, zet de y waarde van *Jump to Position* maar eens op -20 of lager en run het spel. Je ziet de ridder dan ineens naar beneden gegooid worden als hij de bovenste muur raakt. Hetzelfde gebeurt als de ridder tegen een zijmuur botst. In een platform game zullen we het platform (in ons geval de onderste muur) dan ook niet maken van dezelfde objecten als de zijkanten en bovenkant

van de room. Het is echter wel belangrijk dat je snapt wat er in ons geval nu precies gebeurt en waarom.

opgave 5.6

Leg het gedrag van de instantie van *objRidder* in je eigen woorden uit als de instantie tegen de bovenste muur of een zijmuur aankomt.

En ook in dit hoofdstuk sluiten we af met het bewaren van ons spel.

opgave 5.7

Sla je spel op met *hoofdstuk5.gm81* als naam op je H schijf (netwerk schijf).

6. A(rtificial) I(ntelligence)

Ondanks dat onze ridder dol op gebraden kip is voelt hij zich best eenzaam zo helemaal alleen in de gamewereld. We gaan hem daarom een speelkameraad geven.

opgave 6.1

Doe de volgende dingen:

- Maak een sprite die *sprVogelLinks* heet en laad daarin de afbeelding *vogel_links_strip.png* als een strip met 24 aparte sub-images van 48 bij 48 pixels. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort.
- Maak een sprite die *sprVogelRechts* heet en laad daarin de afbeelding *vogel_rechts_strip.png* als een strip met 24 aparte sub-images van 48 bij 48 pixels. Deze afbeelding zit in het ZIP bestand dat bij deze module hoort.
- Maak een object dat *objVogel* heet en koppel hieraan *sprVogelRechts*. Maak het object solid.
- Zet één instantie van *objVogel* in de room.

Als je niet meer precies weet hoe je met sprites en strips van afbeeldingen omgaat kijk dan even terug naar opgave 3.8 om te kijken hoe dat ook alweer zit.

Run het spel en kijk wat er gebeurt.

Zo nu heb je een fladderende vogel in de room staan. Het zou natuurlijk wel leuk zijn als er wat meer leven in deze vogel zit anders hebben we er nog niks aan. De vogel wordt een NPC (non-player character). Het gedrag van een NPC wordt bepaald door de AI (Artificial Intelligence) die we eraan koppelen. Deze kan heel eenvoudig zijn maar ook erg complex. Laten we eens beginnen met een zeer eenvoudige AI. We laten de vogel heen en weer vliegen.

opgave 6.2

Open het object venster van *objVogel*. Je ziet dat daar nog geen events in staan dus die ga je nu toevoegen.
We hebben ervoor gekozen om de vogel naar rechts te laten kijken. Daarom laten we de vogel eerst naar rechts vliegen. Dit doen we in het *Create* event want dat is het eerste event dat meteen optreedt

als het object in de room komt. Aan het *Create* event koppelen we daarom een *Move Fixed* action met de juiste richting en snelheid 3. Dat kun je intussen helemaal zelf doen, dus ga je gang.

Probeer te voorspellen wat er gaat gebeuren, run daarna het spel en kijk wat er gebeurt.

Je ziet dat de vogel nu gewoon de room uitvliegt. Dat is jammer want nu is de ridder weer alleen. Wat we willen is dat de vogel zich omdraait en de andere kant invliegt als hij tegen een zijmuur aanbotst. Je kunt hiervoor de *Reverse Horizontal* action gebruiken in de *move* tab. Voeg een *Collision* event met *objMuur* toe aan *objVogel* en koppel daar de *Reverse Horizontal* action aan.

Probeer te voorspellen wat er gaat gebeuren, run daarna het spel en kijk wat er gebeurt.

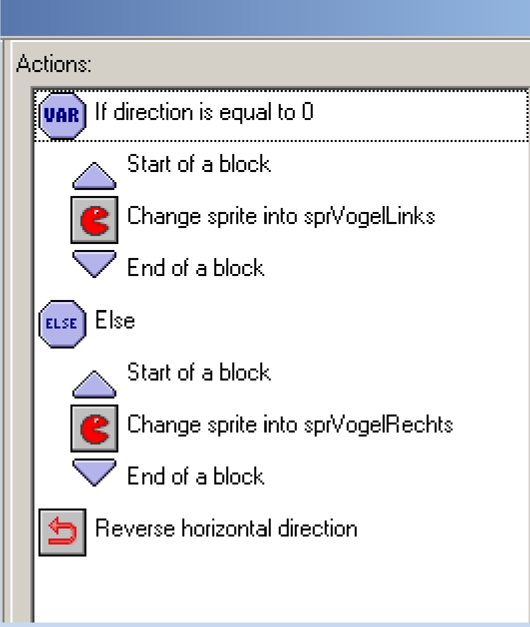
De vogel vliegt nu inderdaad de andere kant in maar hij vliegt achteruit. Dat ziet er onnatuurlijk uit dus we willen ook graag de goede sprite gebruiken als de vogel een zijmuur raakt. Maar hoe doen we dat? We kunnen na de *Reverse Horizontal* action de sprite in *sprVogelLinks* veranderen maar dan gaat het na de botsing met de linker zijmuur weer niet goed.

Om dit probleem op te lossen moeten we een beetje gaan programmeren. Allereerst moet je beseffen dat een bewegende instantie van een object een richting (direction) heeft. De instantie heeft immers ook snelheid. Richting wordt in GameMaker uitgedrukt in graden en die werken ook hier volgens het assenstelsel afgebeeld in Figuur 1. Iedere instantie heeft daarom een variabele die *direction* heet. Als de vogel naar rechts vliegt dan is zijn richting dus 0 graden en de richting is 180 graden als de vogel naar links vliegt. Hier kunnen we gebruik van maken op het moment dat er een botsing optreedt. Als de richting namelijk 0 graden is en de vogel botst dan wordt de nieuwe richting 180 graden en moeten we dus *sprVogelLinks* hebben. Als de richting 180 graden is bij de botsing hebben we *sprVogelRechts* nodig. Aan de slag dus.

opgave 6.3

Open het object venster van *objVogel* en selecteer het *Collission* event met *objMuur*. Maak nu de code die je in de afbeelding hierna ziet na. Je hebt hiervoor actions nodig uit de *control* en *main1* tabs.

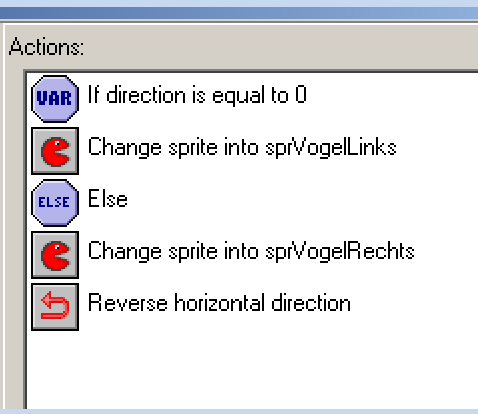
De *Reverse Horizontal* action had je als het goed is al.



Wat hier gebeurt is het volgende.

- Het programma kijk eerst of de richting van de vogel gelijk is aan 0 (oftewel controleert of de vogel naar rechts vliegt).
- Als dit het geval is dan gaat de vogel na de botsing naar links vliegen en moeten we *sprVogelLinks* gebruiken.
- Als dit niet het geval is (else) dan gaat de vogel na de botsing dus naar rechts vliegen. Immers, de vogel kan maar twee kanten invliegen in ons geval. Nu moeten we *sprVogelRechts* gebruiken.

Je ziet dat we *Start Block* en *End Block* actions gebruiken om een duidelijke structuur aan de code te geven. Dit is niet verplicht maar wel sterk aan te raden. De volgende code werkt ook maar ziet er een stuk minder overzichtelijk uit. Hopelijk vind je dat zelf ook.



Run het spel en kijk wat er gebeurt.

Ondanks dat de vogel nu niet meer de room uitvliegt is onze ridder toch nog niet blij. Op deze manier heeft hij namelijk niks aan zijn speelkameraad. Hij heeft graag een aanhankelijke vogel. En die gaat de ridder krijgen.

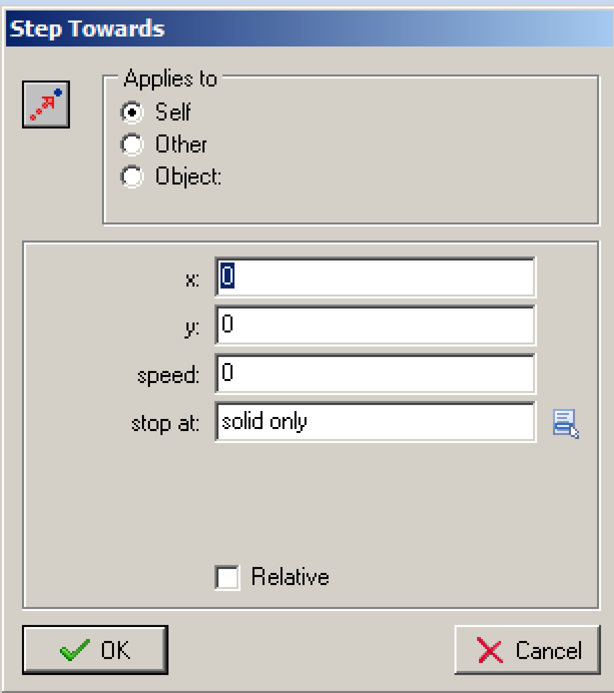
opgave 6.4

Open het object venster van *objVogel* en verwijder alle events.

Vervolgens ga je een bijzonder event toevoegen, een *Step* event. Er bestaan hier drie varianten van maar in de meeste gevallen heb je alleen het *Step – Step* event nodig. Pas als je heel ingewikkelde spellen gaat maken zijn de andere twee varianten nodig.

Een *Step* event is een event dat iedere event puls optreedt voor instanties van het object waarin dit event is gebruikt. Hiermee kunnen we onze vogel dus iedere stap (step) een stukje richting de ridder laten bewegen. Hiervoor gebruiken we de *Step Towards* action die we onderin de *move* tab vinden.

Voeg een *Step - Step* event toe en koppel daarin een *Step Towards* action. Je krijgt dan de volgende dialoog.



Wat er bij *Applies to* moet staan moet je intussen zelf snappen.

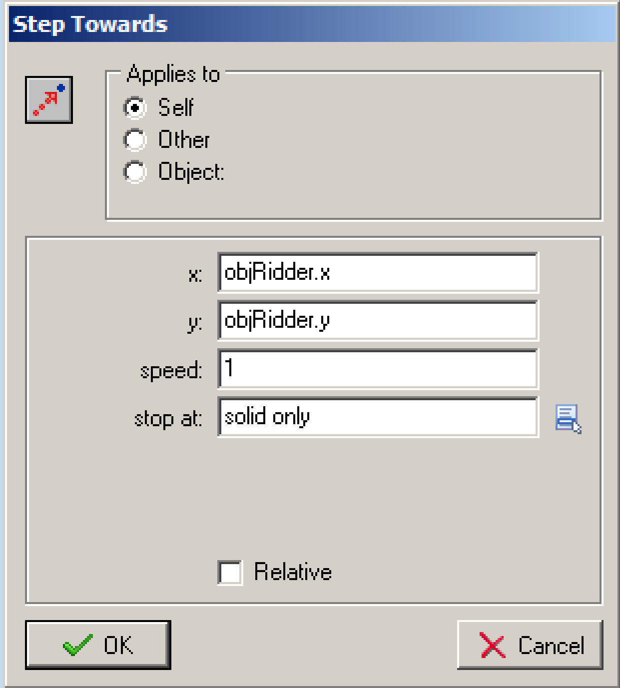
Bij de *x* en de *y* geef je op richting welke positie er bewogen moet worden. In ons geval willen we naar de ridder bewegen en moeten

we daarom de x en y waarde van de positie opgeven. Hiervoor gebruiken we *objRidder.x* en *objRidder.y*. Je kunt *objRidder.x* lezen als "de x waarde van het object *objRidder*".

Het is geen supersnelle vogel dus laten we hem *speed* 1 geven.

De waarde bij *stop at* geeft aan wanneer de vogel moet stoppen. Hier is *solid only* prima omdat alles in onze room solid is.

Je dialoog zou er dan als volgt uit moeten zien.



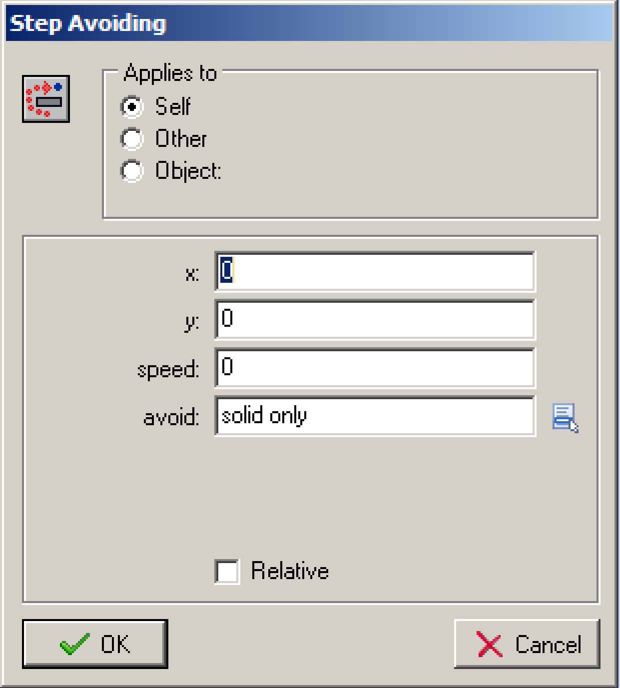
Klik op *OK*, run het spel en kijk wat er gebeurt.

Zoals je ziet achtervolgt de vogel de ridder nu. Toch kan de ridder zich verschuilen voor de vogel en wel achter een gebraden kip. Op het moment dat er een gebraden kip tussen de vogel en de ridder in staat, dan stop de vogel bij de gebraden kip. Dat komt omdat er bij *stop at* in de *Step Towards* action dialoog *solid only* staat. We kunnen daar ook *all instances* invullen maar dat verandert in ons geval niets. Onze ridder wilde een aanhankelijke vogel en dan krijgt hij die ook.

opgave 6.5

Open het object venster van *objVogel* en verwijder de *Step Towards* action bij het *Step - Step* event.

In plaats van *Step Towards* gaan we de *Step Avoiding* action gebruiken. Als je deze toevoegt zie je de volgende dialoog.



Deze dialoog is vrijwel identiek aan die van de *Step Towards* action alleen is er nu geen *stop at* maar een *avoid* optie. Hier kunnen we dus aangeven dat we botsingen met instanties van objecten juist willen vermijden.

Vul in de dialoog dezelfde waardes in als in de vorige opgave. Klik op *OK*, run het spel en kijk wat er gebeurt.

Je ziet dat de ridder nu helemaal nergens meer veilig is. De vogel komt altijd bij hem en vliegt om de gebraden kippen heen als dat nodig is.

opgave 6.6

Tijd voor experimenten. Zet nog een ridder in de room en voeg ook nog twee vogels toe.

Run het spel, kijk wat er gebeurt en verklaar in je eigen woorden wat je ziet gebeuren.

Je heb nu een aantal mechanismes gezien om AI te programmeren. Er zijn er natuurlijk veel meer. Een veel gebruikte AI in spellen is om een NPC pas naar de player character te laten bewegen als deze in de buurt van de NPC komt. Om dit voor elkaar te krijgen moeten we de afstand tussen objecten kunnen berekenen. Hier hebben we het begrip absolute waarde voor nodig. De absolute waarde van een getal is de lengte (of grootte) van dat getal. Daarom is de absolute waarde altijd groter dan of gelijk aan 0, een lengte kan immers nooit negatief zijn. We noteren de absolute waarde van getal N als |N|. Dus |4| is gelijk aan 4 en |-4| is ook gelijk aan 4. In GameMaker bestaat een functie om de absolute waarde uit te rekenen en deze heet *abs*. We kunnen in GameMaker dus zeggen *abs(-4)*, *abs(objRidder.x)* en belangrijker nog *abs(objRidder.x - objVogel.x)*. Dit laatste voorbeeld kunnen we namelijk gebruiken om de afstand tussen de objecten *objRidder* en *objVogel* te bepalen.

Als we heel netjes de exacte afstand tussen twee punten willen bepalen hebben de Stelling van Pythagoras nodig. Hopelijk ken je die nog, zo niet kijk dan op Wikipedia. Met behulp van deze stelling kun je een cirkelvormig gebied om een punt definiëren. Dit doen we als volgt.

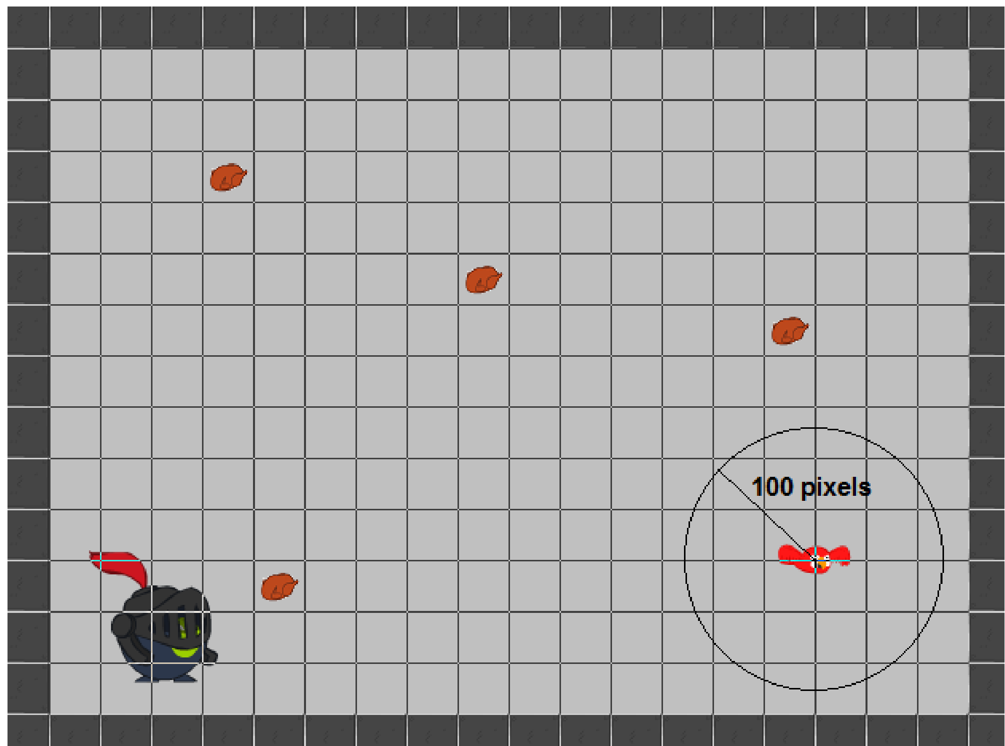
Zorg dat de oorsprong van beide objecten in het midden van het object ligt. Dit is belangrijk bij het netjes bepalen van de afstand. De horizontale afstand tussen de middelpunten van twee objecten (dus over de x-as) hebben we hiervoor al gezien. Voor de vogel en de ridder is dat *abs(objRidder.x - objVogel.x)*. Voor de verticale afstand doen we hetzelfde maar dan voor de y waardes van de middelpunten, dus *abs(objRidder.y - objVogel.y)*. Nu geldt, op basis van de stelling van Pythagoras, voor de afstand tussen de vogel en de ridder het volgende.

$$afstand = \sqrt{(objRidder.x - objVogel.x)^2 + (objRidder.y - objVogel.y)^2}$$

We kunnen hier de absolute waarde weglaten omdat we geen negatieve uitkomsten kunnen krijgen als we getallen kwadrateren. Als we nu willen bepalen of de vogel en de ridder bijvoorbeeld binnen 100 pixels van elkaar verwijderd zijn dan kunnen we de volgende conditie gebruiken.

$$\sqrt{(objRidder.x - objVogel.x)^2 + (objRidder.y - objVogel.y)^2} < 100$$

Hiermee definieer je het cirkelvormige gebied om de vogel dat je in de volgende afbeelding ziet, mits je de conditie in *objVogel* implementeert bij het bewegen van de vogel.

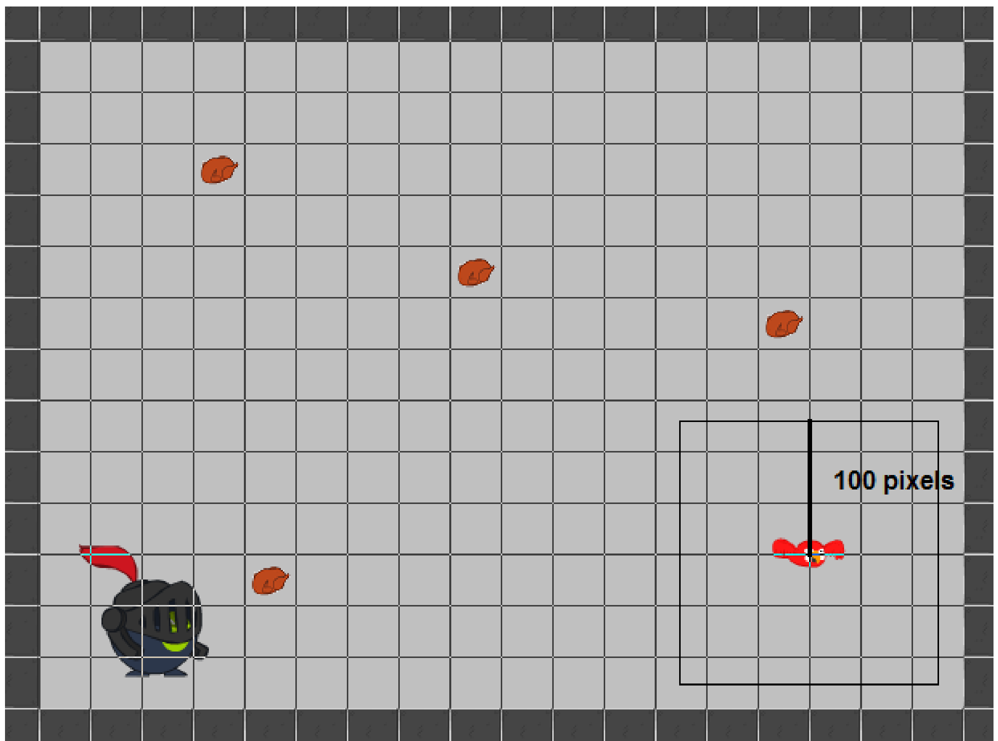


Doorgaans is dit een dure berekening en deze kan ervoor zorgen dat je spel niet vloeiend loopt, zeker als je veel objecten hebt waartussen je de afstand in de gaten moet houden. Daarom wordt er in games vaak een alternatieve methode gebruikt die minder nauwkeurig is maar wel voldoet. Deze methode definieert geen cirkelvormig maar een vierkant gebied om een punt. Als we nu willen bepalen of de vogel en de ridder bijvoorbeeld binnen 100 pixels van elkaar verwijderd zijn dan kunnen we de volgende conditie gebruiken.

$$|objRidder.x - objVogel.x| < 100 \ \&\& \ |objRidder.y - objVogel.y| < 100$$

Hierbij staat && voor de EN (AND) operator; oftewel zowel de expressie links als die expressie rechts van de EN moet waar zijn. Hier hebben we de absolute waarde nodig want deze conditie werkt niet voor negatieve getallen.

Hiermee definieer je het vierkante gebied om de vogel dat je in de volgende afbeelding ziet, mits je de conditie in *objVogel* implementeert bij het bewegen van de vogel.



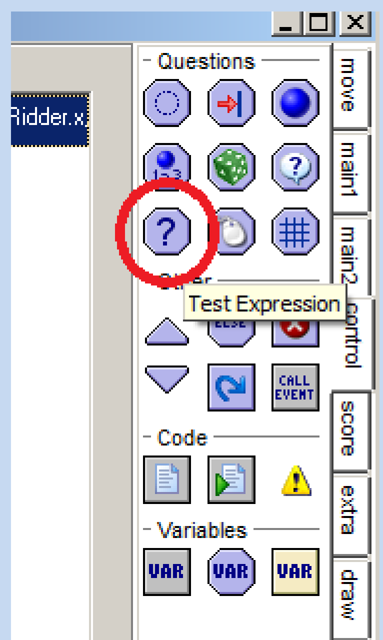
Laten we dit laatste eens gaan maken in GameMaker.

opgave 6.7

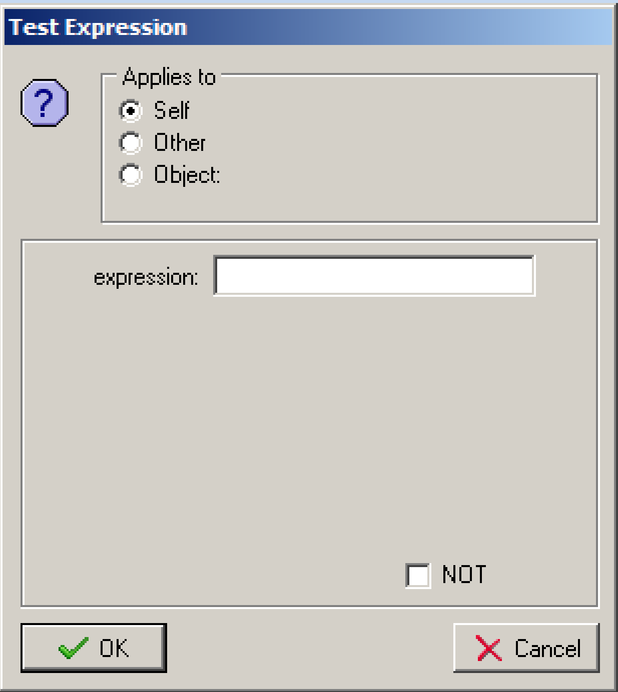
Zorg dat je één ridder en één vogel in de room hebt. Zet de ridder helemaal links en de vogel helemaal rechts in de room neer. Centreer de oorsprong voor de sprites (dus zowel die van links als rechts) van beide objecten.

We willen dat de vogel op de ridder reageert als de ridder in de buurt komt. We gaan daarom onze code in *objVogel* zetten. Dus open het venster van dit object.

Iedere step moeten we nu controleren of de ridder in de buurt is en als dat het geval is activeren we de *Step Avoiding* action. Dit controleren doen we met de *Test Expression* action die je in de *control* tab vindt zoals je in de volgende afbeelding ziet



Als je de *Test Expression* action gebruikt dan krijg je de volgende dialoog te zien.

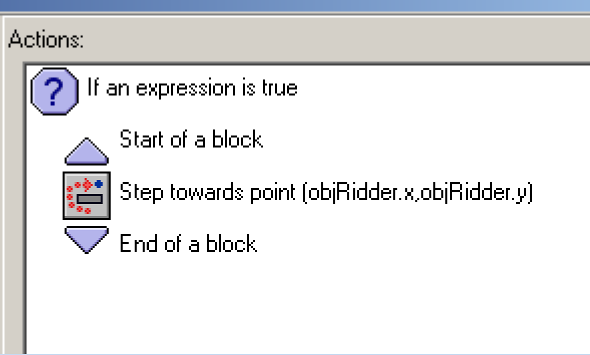


Bij *expression* moet je nu de conditie invullen die we voor deze opgave hebben laten zien en uitgelegd. In GameMaker ziet deze er als volgt uit.

abs(*objRidder.x* - *objVogel.x*) < 100 && *abs*(*objRidder.y* - *objVogel.y*) < 100

Zorg weer dat je netjes programmeert zodat je het volgende in je

Actions gedeelte hebt staan.



Het werkt ook zonder de *Start Block* en *End Block* actions maar met deze actions is het netter.

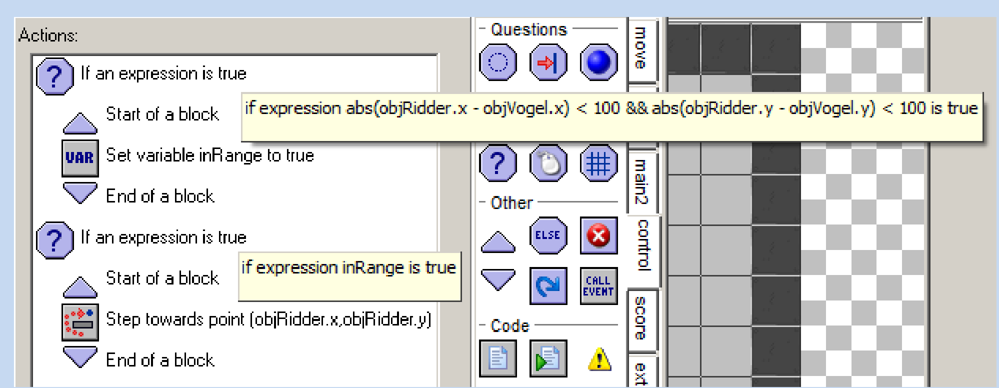
Run het spel, kijk wat er gebeurt en verklaar in je eigen woorden wat je ziet gebeuren.

Zo gauw de ridder in de buurt komt begint de vogel naar de ridder te bewegen. Als de ridder echter weer buiten het bereik van de vogel (in ons geval 100 pixels) komt, dan stop de vogel. Dat komt omdat iedere event puls de conditie wordt gecheckt. Als deze niet meer waar is wordt de bijbehorende action ook niet meer uitgevoerd en dus stopt de vogel. Vaak is dit wat je wilt maar stel dat je wilt dat als de ridder eenmaal in de buurt van de vogel is geweest dat de vogel hem blijft volgen.

opgave 6.8

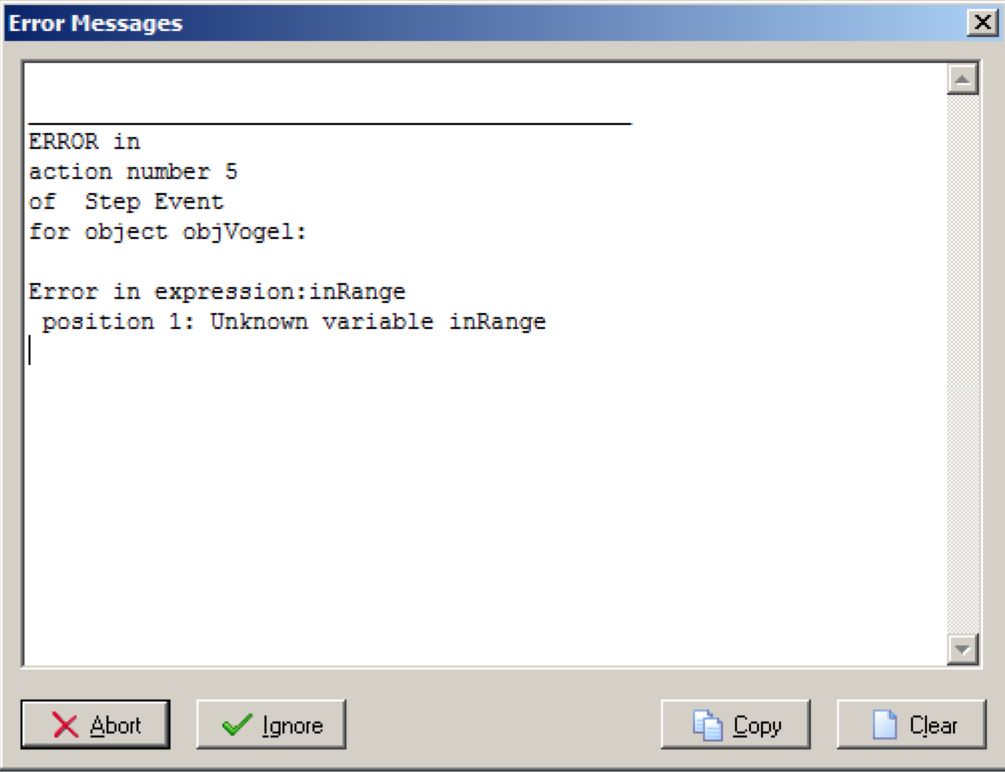
Om ervoor te zorgen dat de vogel de ridder blijft volgen als deze eenmaal in de buurt is geweest, moet je dus onthouden dat de ridder in de buurt is geweest. Typisch gebruiken we variabelen om dingen te onthouden. Daarom introduceren we een variabele genaamd *inRange*.

Deze zetten we in het *Step Step* event op de waarde *true* (waar) als de ridder in de buurt is. We zetten deze variabele niet meer op *false* (onwaar). Vervolgens laten we de vogel bewegen als *inRange* waar is. Aangezien deze variabele waar wordt de eerste keer als de ridder in de buurt van de vogel komt en daarna niet meer onwaar wordt gemaakt heeft dit het beoogde effect. Dat ziet er als volgt uit.



Run het spel en kijk wat er gebeurt.

Je krijgt nu de volgende foutmelding.



Hiermee bedoelt GameMaker dat hij de variabele *inRange* niet kent. En dat is ook wel logisch, want de eerste event puls is de conditie van de eerste *Test Expression* action niet waar. Daarom wordt er geen waarde aan de variabele *inRange* gegeven en is de variabele dus nog niet bekend in ons programma. Vervolgens willen we de variabele in de volgende *Test Expression* action wel gebruiken en dat kan dus niet want de variabele bestaat nog niet.

We kunnen de variabele *inRange* echter geen vaste waarde geven in het Step - Step event want dit event wordt elke event puls uitgevoerd. De meest gebruikte oplossing hiervoor is de variabele een

waarde te geven in het *Create* event aangezien dit event maar één keer aangeroepen wordt en wel helemaal aan het begin van het spel. Dat doen we als volgt.

We kiezen de waarde *false* (onwaar) als beginwaarde. We willen immers niet dat de vogel meteen naar de ridder beweegt ook al is de ridder nog niet in de buurt geweest en dat zou wel gebeuren als we bijvoorbeeld de waarde *true* (waar) nemen.

Run het spel en kijk wat er gebeurt.

We hebben nu genoeg voorgedaan. Tijd om zelfstandig een probleem aan te pakken.

opgave 6.9

Pas het spel zo aan dat het de Stelling van Pythagoras gebruikt om de vogel te laten bewegen als de ridder in de buurt is geweest.

Let op dat je voor deze opdracht ook het een en ander op internet op moet zoeken omdat je bijvoorbeeld nog niet alle functies hebt gehad die je nu nodig hebt.

En tot slot zoals we intussen gewend zijn

opgave 6.10

Sla je spel op met *hoofdstuk6.gm81* als naam op je H schijf (netwerk schijf).

7. alarms

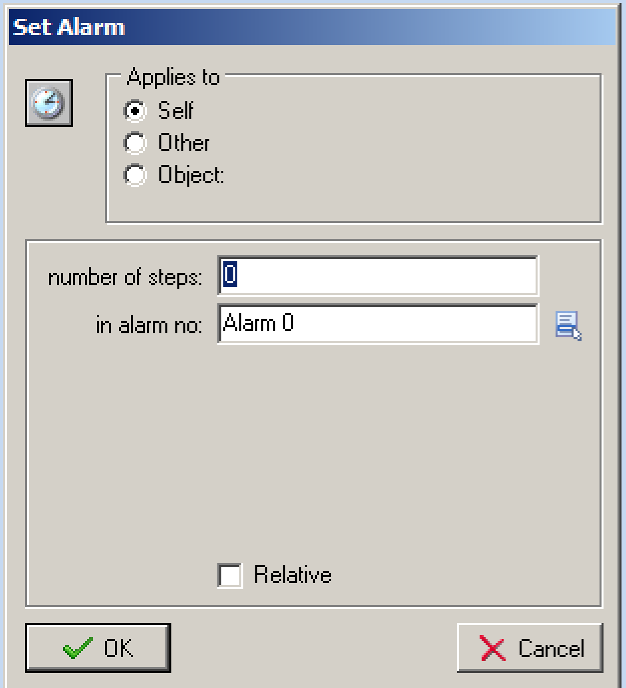
Weet je nog dat onze ridder honger had en dat we hem gebraden kip voerden? We hebben toen een stuk of vier kippen in de room gezet, maar die heeft de ridder zo op en daarna is het weer honger leiden voor hem. Het is veel handiger als een opgegeten kip na verloop van tijd ververst wordt, dan hoeft onze ridder nooit honger te leiden.

Wat we willen is dat een kip bijvoorbeeld iedere 10 secondes nadat deze opgegeten is wordt ververst. Dit verversen noemen we in game termen spawn. Om dit voor elkaar te krijgen gaan we met een alarm (wekker) werken. Als de ridder een kip op eet dan moeten we een wekker zetten die na 10 secondes af gaat. Als het alarm afgaat moeten we een nieuwe gebraden kip in de room zetten. Dat gaan je nu doen.

opgave 7.1

Het klinkt logisch zijn om dit te programmeren in *objGebrKip*. Toch is dit niet handig, want als de ridder de kip opeet wordt de instantie van objGebrKip vernietigd. Een wekker zetten in iets dat niet meer bestaat heeft totaal geen zin want de wekker kan niet meer afgaan. Wat we kunnen doen is dit implementeren in het object *objRidder*.

Open het object venster voor *objRidder*. We hebben daar al een *Collision* met *objGebrKip* event. We gaan daar de *Set Alarm* action aan toevoegen. Deze action staat links bovenin de *main2* tab. Je krijg dan de volgende dialoog.

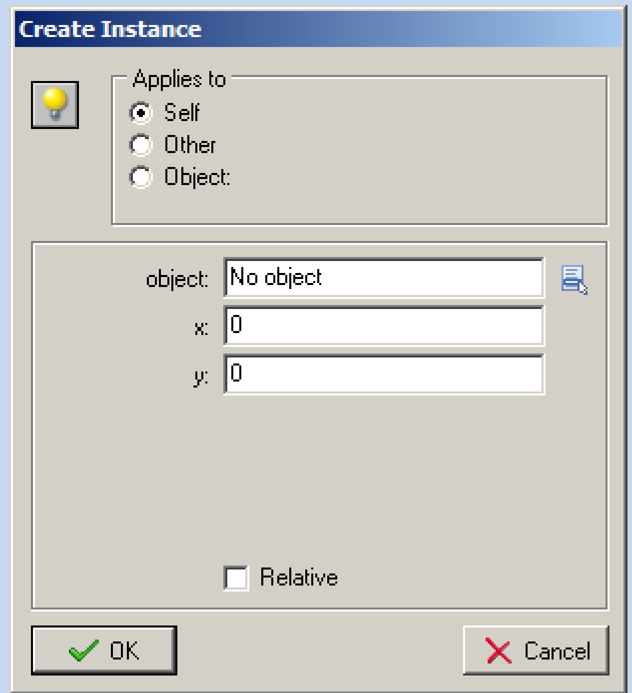


Je geeft op over hoeveel tijd het alarm af moet gaan met *number of steps*. De eenheid daarvan is steps. We weten dat de room speed aangeeft hoeveel events, en dus steps, er per seconde optreden. Gelukkig is er een variabele die *room_speed* heet en die kunnen we dus gebruiken om de tijd in secondes uit te drukken. We kunnen 10 secondes dus opschrijven als *10 * room_speed*.

In totaal kun je per object 12 alarms definiëren. Aangezien we er nog geen hebben gebruikt kunnen we *Alarm 0* gebruiken.

We hoeven *Relative* niet aan te vinken want deze optie doet niets bij een alarm.

Vervolgens moeten we als het alarm afgaat een nieuwe gebraden kip in de room zetten. Dit doen we via het *Alarm - Alarm 0* event omdat we zojuist alarm 0 hebben gezet bij de collision. Hiervoor gebruiken we de *Create Instance* action die links bovenin de *main1* tab staat. Je krijgt dan de volgende dialoog te zien.



We moeten een gebraden kip creëren dus als *object* nemen we *objGebrKip*. Vervolgens moeten we opgeven waar de nieuwe gebraden kip moet komen te staan. Laten we hiervoor een willekeurige waarde nemen. Daarvoor hebben we een aantal functies ter beschikking. Laten we beginnen met de functie *irandom_range*. Hieraan geven we een ondergrens en een bovengrens voor de random waarde mee.

Onze room is 640 bij 480 pixels. Het is echter geen goed gebruik om die getallen te gebruiken. Als we onze room groter of kleiner maken dan werkt ons spel misschien niet meer goed. We kunnen de breedte en hoogte van een room opvragen via de variabelen *room_width* en *room_height*. In onze room staat ook een muur en daar willen we vanaf blijven. Laten we daar 50 pixels voor reserveren dat is redelijk veilig. We willen dus voor de *x* een random waarde tussen *50* en *room_width-50* en voor *y* tussen *50* en *room_height-50*.

Vul alles in, druk op *OK*, run het spel en kijk wat er gebeurt.

Heb je geprobeerd de ridder meerdere kippen binnen 10 secondes op te laten eten? Zo ja, dan heb je gemerkt dat er na 10 secondes maar één kip bijkomt ongeacht hoeveel de ridder er opeet. Dit probleem oplossen met de drag & drop functionaliteit (de manier van programmeren die je tot nu toe gebruikt hebt) is omslachtig. GameMaker heeft echter ook een tekstuele taal waarin je kunt programmeren. Deze heet GML (GameMaker Language). Hierin kun je complexe problemen makkelijk oplossen. In deze module gaan we GML echter niet behandelen.

We kunnen echter wel een workaround gebruiken. Dat wil zeggen dat om het probleem heen gaan werken en het niet echt oplossen. Dat doen we als volgt.

opgave 7.2

Verwijder de *Collision* met *objGebrKip* en *Alarm - Alarm 0* events uit *objRidder*. We gaan deze onderbrengen in objGebrKip. Maar in plaats van het object *objGebrKip* bij een collision te vernietigen gaan we het alleen verbergen.

Open het object venster voor *objGebrKip*. Voeg een *Collision* met *objRidder* event toe. We gaan eerst de kip onzichtbaar maken en daarna buiten de room zetten zodat de ridder er niet per ongeluk tegenaan beweegt. Dat doen we als volgt.

Voeg aan het *Collision* met *objRidder* event een *Change Sprite* action toe en gebruik *sprMuur* als *sprite*. Voeg daarna een *Jump to Position* action toe met als coördinaten (0, 0), want daar staat al een muur. Zet tot slot nog een alarm voor over 10 secondes.

Als het alarm afgaat moeten we een nieuwe gebraden kip in de room zetten. Dit doen we weer via het *Alarm - Alarm 0* event omdat we zojuist alarm 0 hebben gezet bij de collision. Ook gebruiken we weer de *Create Instance* action waarin we een object *objGebrKip* creëren met voor de *x* een random waarde tussen *50* en *room_width-50* en voor *y* tussen *50* en *room_height-50*. Ook moeten we tot slot de oude instantie vernietigen.

Vul alles in, druk op *OK*, run het spel en kijk wat er gebeurt.

Om te zien wat er gebeurt kun je ook de *Change Sprite* action weglaten. Je ziet dan in de linker bovenhoek van de room telkens de opgegeten kip staan tot er een nieuwe in het veld verschijnt.

En ook dit hoofdstuk slaan we het resultaat weer op.

opgave 7.3

Sla je spel op met *hoofdstuk7.gm81* als naam op je H schijf (netwerk schijf).

8. meerdere rooms

Tot slot gaan we kijken hoe je meerdere rooms maakt. Het is immers leuk als de speler van je game telkens een nieuwe, mooiere en moeilijkere uitdaging krijgt.

opgave 8.1

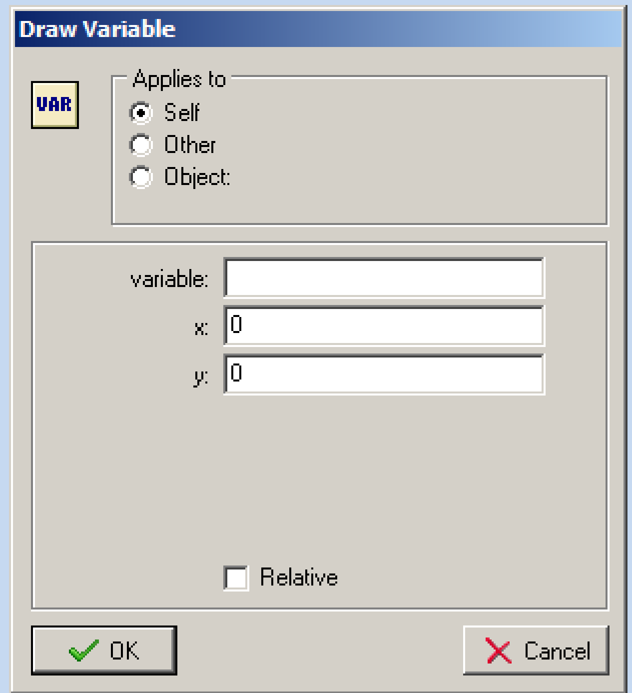
Maak een nieuwe room aan. Dit kun je het gemakkelijkst doen door je eerste room te dupliceren (rechtsklik op *room0* bij de resources en kies *duplicate*).

We moeten nu een reden vinden waarom de ridder naar de volgende room mag. Dat kan zijn als hij bijvoorbeeld 10 kippen op heeft gegeten. Daarom gaan we eerst zorgen dat we bijhouden hoeveel kippen de ridder op heeft.

Zoals je al eerder gezien hebt kunnen we iets onthouden in variabelen. Voor het aantal gegeten gebraden kippen gebruiken we bijvoorbeeld de variabele *eaten*. Deze hebben nodig in het object *objRidder* want de ridder eet de kip op. We kunnen de variabele maken en een waarde geven in het *Create* event. Doe dit zoals je ook met *inRange* bij *objVogel* hebt gedaan. Gebruik de waarde nul want als de ridder gecreëerd wordt heeft hij nog geen kip gegeten.

Vervolgens willen we de waarde van de variabele ergens laten zien. Dat doen we met de *Draw Variable* action in de *control* tab. *Draw* actions kunnen alleen gebruikt worden *Draw* events, net zoals *Step* actions alleen gebruikt kunnen worden in *Step* events.

Voeg een *Draw* event toe en koppel daar de *Draw Variable* action aan. Je krijgt dan de volgende dialoog te zien.



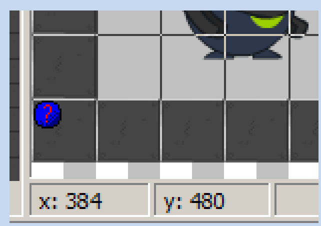
Als *variable* vul je *eaten* in. Neem als positie bijvoorbeeld de coördinaten (50, 50). Het maakt niet heel veel uit waar de waarde van de variabele staat op dit moment, het gaat om de oefening.

Vul alles in, druk op *OK*, run het spel en kijk wat er gebeurt.

Dat is nou vervelend. We zien onze ridder ineens niet meer. Dat komt omdat we een *Draw* event hebben opgenomen in het *Event* gedeelte van *objRidder*. Dat betekent dat we nu alles zelf moeten tekenen dus ook de *sprite* van de ridder. Dat is heel veel werk dat we nu ineens zelf moeten doen terwijl GameMaker dat eerst voor ons deed. En dat om de waarde van een variabele af te drukken. Dat is het niet waard. Dus dat gaan we anders oplossen.

opgave 8.2

Maak een object aan waarin je geen sprite koppelt. Noem dit object *objRoomCtrl*, wat staat voor room controller. Dit object gebruik je om globale dingen voor de room te regelen en onthouden zoals het aantal gegeten kip waar we in de vorige opgave mee bezig zijn geweest. Zet *objRoomCtrl* ergens in de eerste room neer, bijvoorbeeld links onderin de room. Dit ziet er als volgt uit.



We kunnen nu het laten zien van de variabele *eaten* in *objRidder* doen in *objRoomCtrl*. Gebruik als positie voor het getal de coördinaten (50, 50). Probeer dit zelf te realiseren, je kunt vanuit *objRoomCtrl* bij de variabele *eaten* van *objRidder* via de volgende constructie:

```
objRidder.eaten
```

Vervolgens moeten we zorgen dat de variabele *eaten* opgehoogd wordt als de ridder een kip opeet. Hiervoor moet je in *objGebrKip* kijken. Gebruik hiervoor de *Set Variable* action. Zoek verder zelf uit hoe je dit aanpakt en test je spel.

We hebben aan het begin van dit hoofdstuk een nieuwe room aangemaakt en zoals gezegd willen we dat de ridder naar de volgende room gaat als hij 10 kippen opgegeten heeft. Dat doen we als volgt.

opgave 8.3

De plaats om te controleren of er 10 kippen opgegeten zijn is dezelfde plek als waar we onze teller (variabele *eaten*) verhogen. De action om naar de volgende room te gaan kun je vinden in de tab *main1*. Zoek zelf uit hoe je dit aanpakt en test je spel.

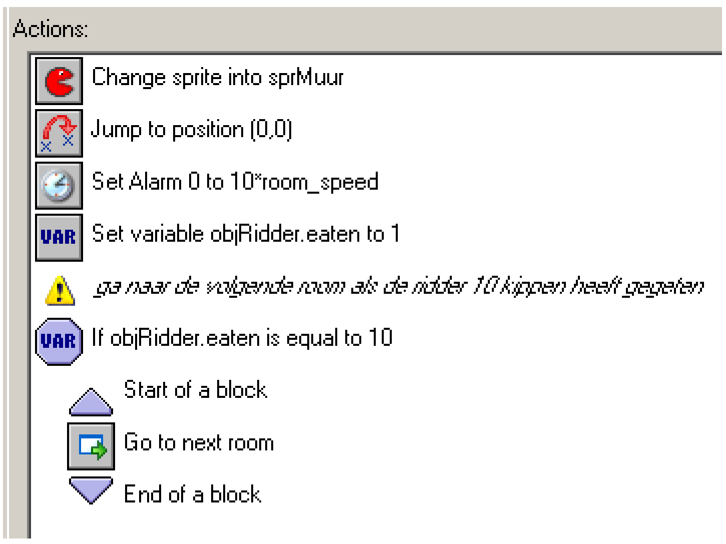
Het is je ongetwijfeld gelukt om in de volgende room te komen. Maar waarschijnlijk is de ridder niet meegekomen. Dat komt omdat onze ridder niet persistent is.

opgave 8.4

Je maakt een object persistent in het object venster. Je kunt hier de optie *Persistent* aanvinken zoals je de volgende afbeelding ziet.

Vink de optie *Persistent* aan, run je spel en kijk wat er gebeurt.

Tot nu toe hebben we nog geen echt complexe dingen gedaan. Als je stukken code maakt die moeilijker te begrijpen zijn dan moet je dingen gaan uitleggen. Dit doen we door commentaar in de code te zetten. Als je met drag & drop code werkt kun je commentaar toevoegen aan je actions met de *Comment* action in de *control* tab. Dat ziet er als volgt uit.



9. hoe verder

Zo, nu heb je in principe voldoende geleerd om je eigen spel te gaan maken en nieuwe dingen te ontdekken in GameMaker. Er zijn nog een aantal concepten die we in deze module niet uitgelegd hebben en die wel handig kunnen zijn. Deze worden uitgelegd op de website van Michel Fiege. Op deze website zijn met name spellen 3 (Pacman) en 6 (Super Mario) interessant om te bekijken. In het spel Pacman wordt uitgelegd wat recursie is. In Super Mario leer je wanneer en waarom je groepering van resources toepast. Verder leer je wat toestandsdiagrammen (STD; State Transition Diagram) zijn. Door de al bestaande code te bestuderen kun je verder zelf uitzoeken hoe overerving (inheritance) werkt.

Als je echt niet kunt wachten met je eigen spel te maken dan kun je nu van start gaan maar het is verstandig om eerst nog naar Pacman en Super Mario te kijken wat ons betreft.

Veel plezier en succes.

10. wat heb je geleerd

Je hebt intussen al veel geleerd over GameMaker en je bent nu goed bekend met de gebruikersinterface. Je kunt nu de volgende dingen:

- rooms maken
- sprites maken
- objecten maken
- nieuwe sprites aan objecten koppelen
- events toevoegen
- actions koppelen aan events
- geluiden toevoegen en gebruiken
- een player character laten besturen op verschillende manieren
- omgaan met snelheid
- omgaan met eventpulsen
- omgaan met de oorsprong van rooms, objecten en sprites
- bewegende sprites maken
- reageren als objecten de room verlaten
- reageren als objecten botsen
- omgaan met de solid optie
- verschil maken tussen objecten en instanties
- omgaan met zwaartekracht
- een player character laten springen
- hoe het assenstelsel werkt
- hoe een player character kan landen op een platform
- AI programmeren voor objecten op verschillende manieren
- wiskunde gebruiken in games
- omgaan met alarms
- meerdere rooms maken
- de player character meenemen naar een nieuwe room