# Programmeren Bulletin

23ste jaargang december 2016 Nummer 4

hcc! programmeren

Interessegroep

# Inhoud


Veel applicaties geven de mogelijkheid om de gebruiker zowel met de muis te laten werken als met de code. GML, een script in Game Maker, is daar een voorbeeld van. VBA kunnen we daar niet mee vergelijken, hoewel het toch als een script wordt aangezien. Wat is het verschil?

Wie XNA kent weet dat kennis hebben met Visual Studio een pré is. De overstap naar Unity bespaart een hoop tijd in het ontwikkelen van grafische programma's. Het nadeel is wel dat Visual Basic niet samen kan met Unity, maar dat wel kan met XNA. Men moet goed onthouden dat alleen in Visual Basic 2010 met XNA geprogrammeerd kan worden.

Nu we kennis konden maken met Python, zouden we natuurlijk willen weten wat we er nog meer mee kunnen doen. Kunnen we echte programma's schrijven? Jazeker, en zelfs met de OOP structuur erbij. Maar de editor is en blijft een Interpreter. Er moet dus een oplossing zijn om geen last te hebben van de directe mode.

**Marco Kurvers**

# Kennismaken met Game Maker Studio en GML.

Programma's hebben scripttalen om de gebruiker meer functionaliteit te geven. Excel heeft dat ook. Met VBA werken is echter niet nodig, maar het kan het werkboek uitbreiding geven. GML, de afkorting van *Game Maker Language*, is ook een taal die niet speciaal nodig, maar wel handig is voor extra mogelijkheden.

Men denkt vooral dat zulke achtergrond-programmeertalen scripttalen zijn. Hoewel GML een scripttaal is, is VBA dat echter niet, ook al kunnen we VBA niet vergelijken met Visual Basic. De reden is hoe we de code in het programma moeten implementeren. GML kent geen OOP structuur. We kunnen alleen een scriptbestand aanmaken en die via een event aan laten roepen. VBA heeft een apart IDE venster waarmee we een heel project in kunnen maken. Klassen kunnen in aparte bestanden worden geschreven en de werkbladen hebben events waar we de klassen in aan kunnen sturen.

## Game Maker Studio IDE

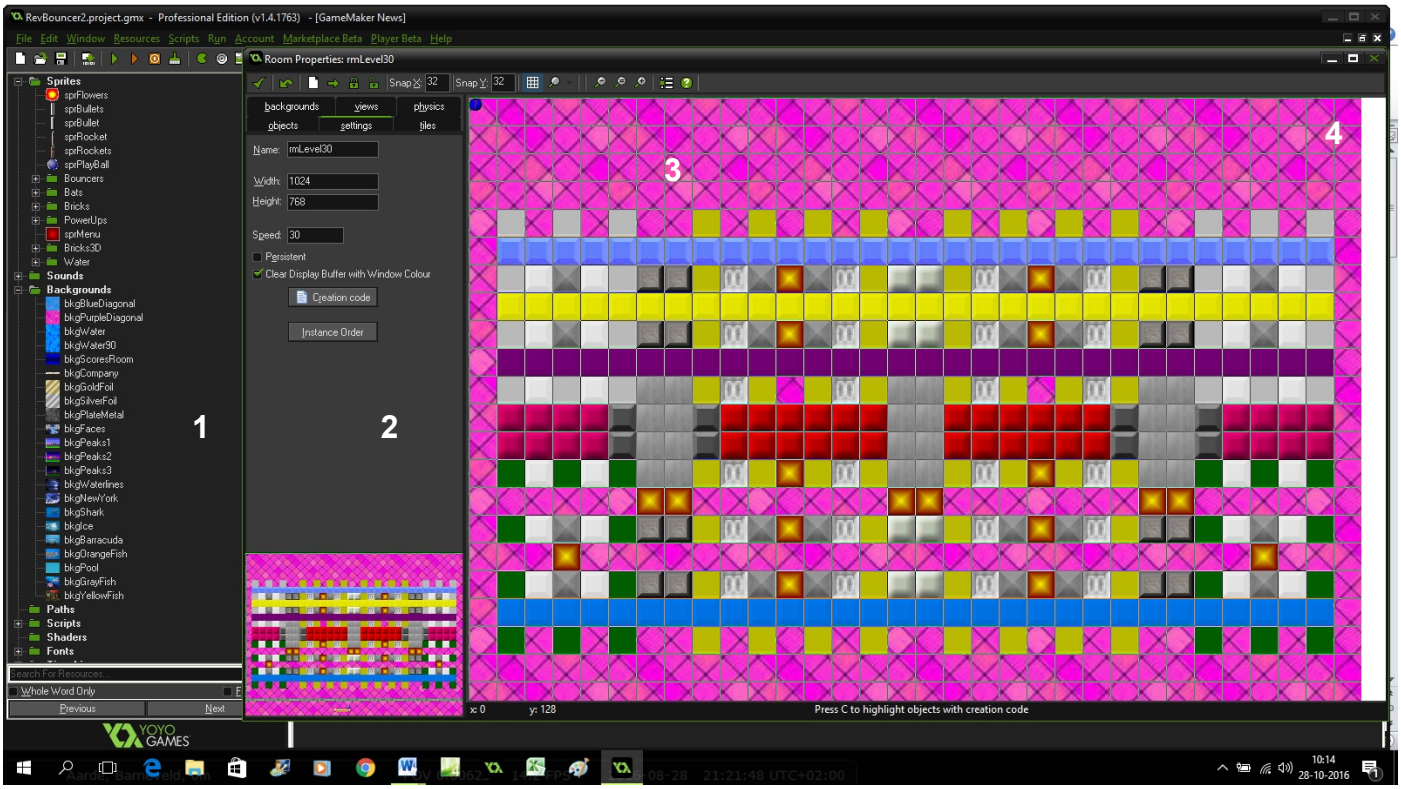



**Fig. 1 Game Maker Studio IDE**

1. Dit deel is te vergelijken met de Windows Verkenner. Hier kunt u kiezen voor sprites, achtergronden, fonts, sounds, rooms, objecten, enzovoort. Zie figuur 2 voor een betere weergave van de hiërarchie.
2. Hier kunt u de instellingen vinden van het speelscherm of ook wel de room genoemd. U kunt zoveel rooms gebruiken als u wilt, bijvoorbeeld een room voor elke game level. Meer rooms kost wel meer geheugen en neemt meer ruimte in beslag in het uitvoerbare bestand.
3. Dit scherm is het deel dat de speler ziet als de game wordt gestart. Het bevat inhoud, zoals alle objectinstanties van de objecten die in punt 1 worden toegevoegd.
4. De room kan geminimaliseerd, gemaximaliseerd en gesloten worden. Achter het roomvenster kunt u informatie en reclame vinden over Game Maker en YoYo Games.

## Het linkerdeel groter gezien

Er kan van alles worden toegevoegd. De sprites kunnen jpg, png en gif bestanden zijn. Game Maker heeft ook een sprite-editor. Hiermee kunnen zelf sprites gemaakt worden en kunnen zelfs eigen strips worden gemaakt voor animaties. Om de sprites op een room te kunnen plaatsen, hebben we objecten nodig, zie later. De objecten hebben zelf events die acties kunnen ondernemen. Sprites kunnen dat niet, want die bevatten alleen maar textuur-informatie. Toch kunnen de sprites worden gebruikt zonder objecten, maar dan moet er gebruik worden gemaakt van GML om acties te kunnen ondernemen.

Rechts ziet u een deel van het roomvenster. Elke room kan ook een script aanroepen, zie de knop **Creation code**. Op het **settings** tabblad geeft u de naam van de room op en stelt u de grootte in.

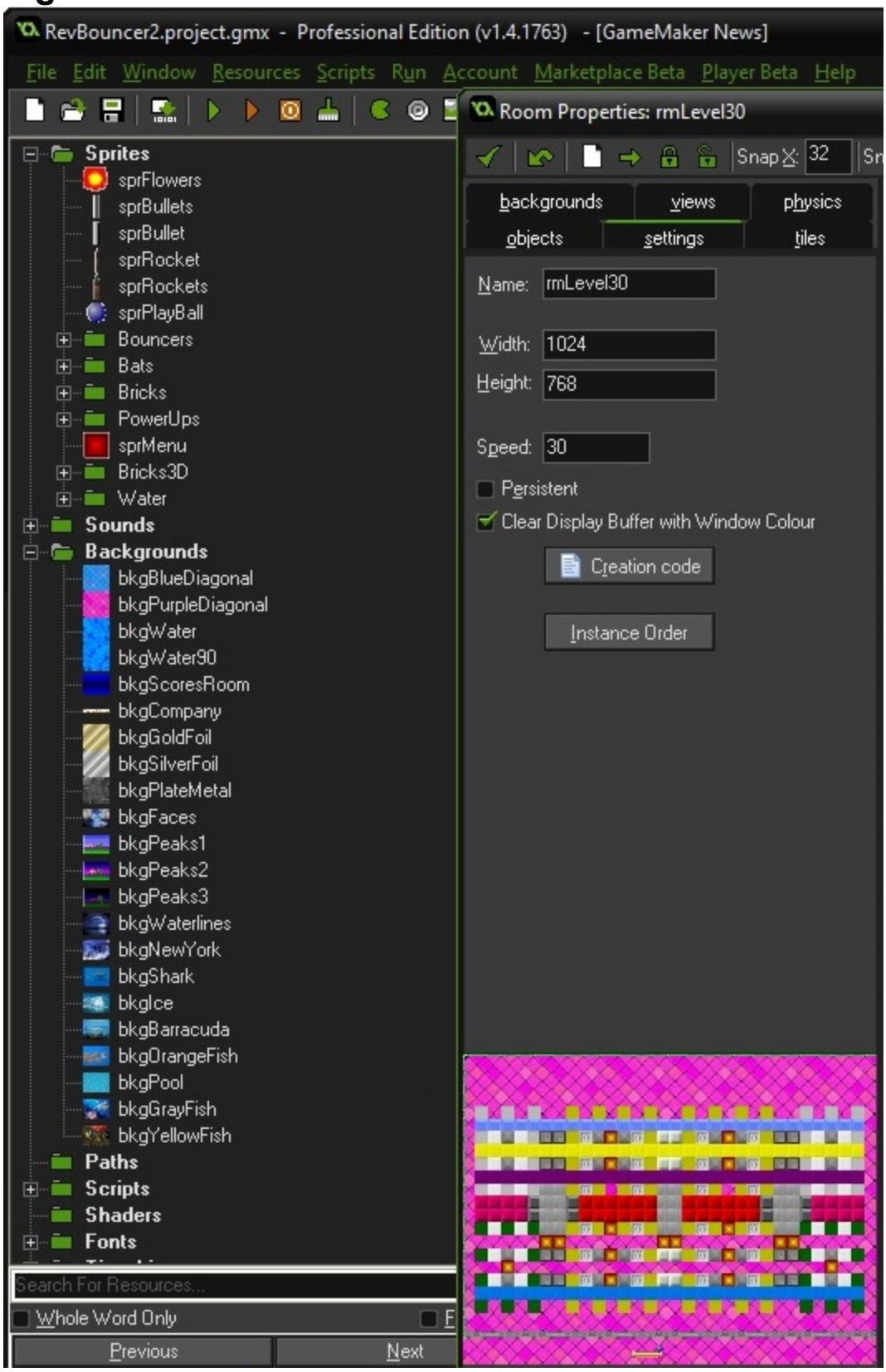



**Fig. 2 De hiërarchie en de room instellingen**

**De objecten**

Figuur 3 laat een deel van de objecten zien met een object dat geopend is. In het geopende venster zijn de eigenschappen en events van het object aanwezig. Alle objectinstanties die op de room geplaatst worden, gebruiken deze ingestelde eigenschappen van het aangemaakte object; in code zouden we het ook wel het *klassenobject* kunnen noemen waarvan we de instanties declareren.

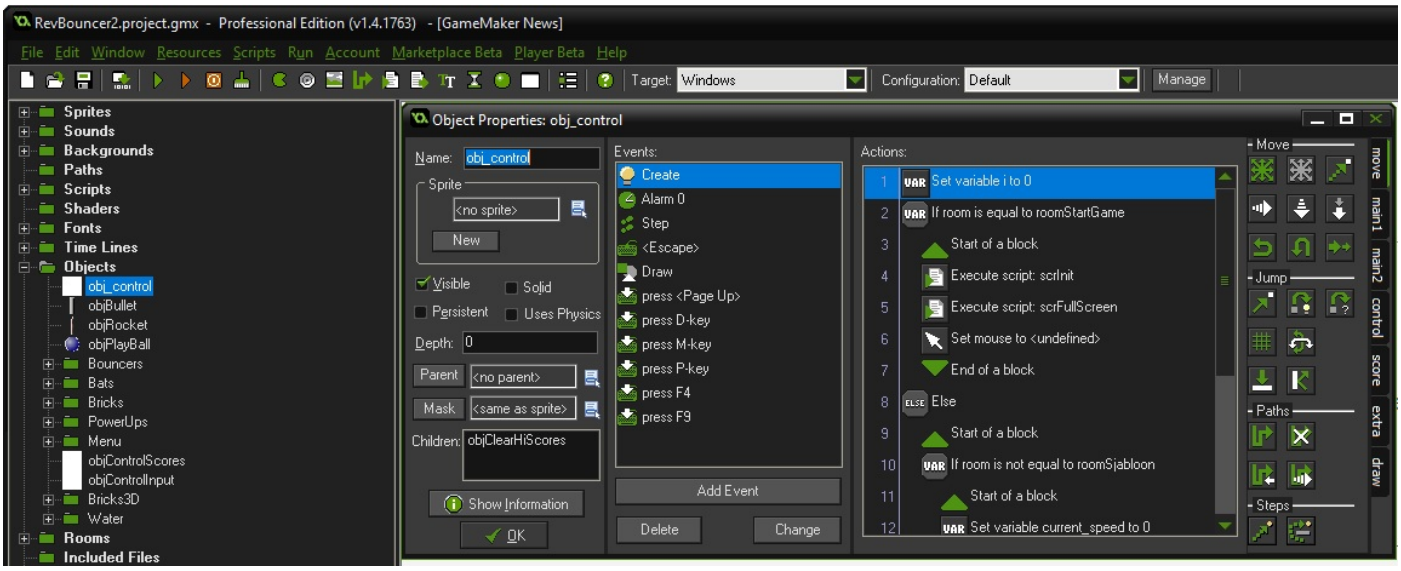



**Fig. 3** Een object geopend met de inhoud (eigenschappen, events en acties)

Figuur 4 laat een afbeelding zien van alleen het objectvenster, zodat de inhoud nog duidelijker te zien is.

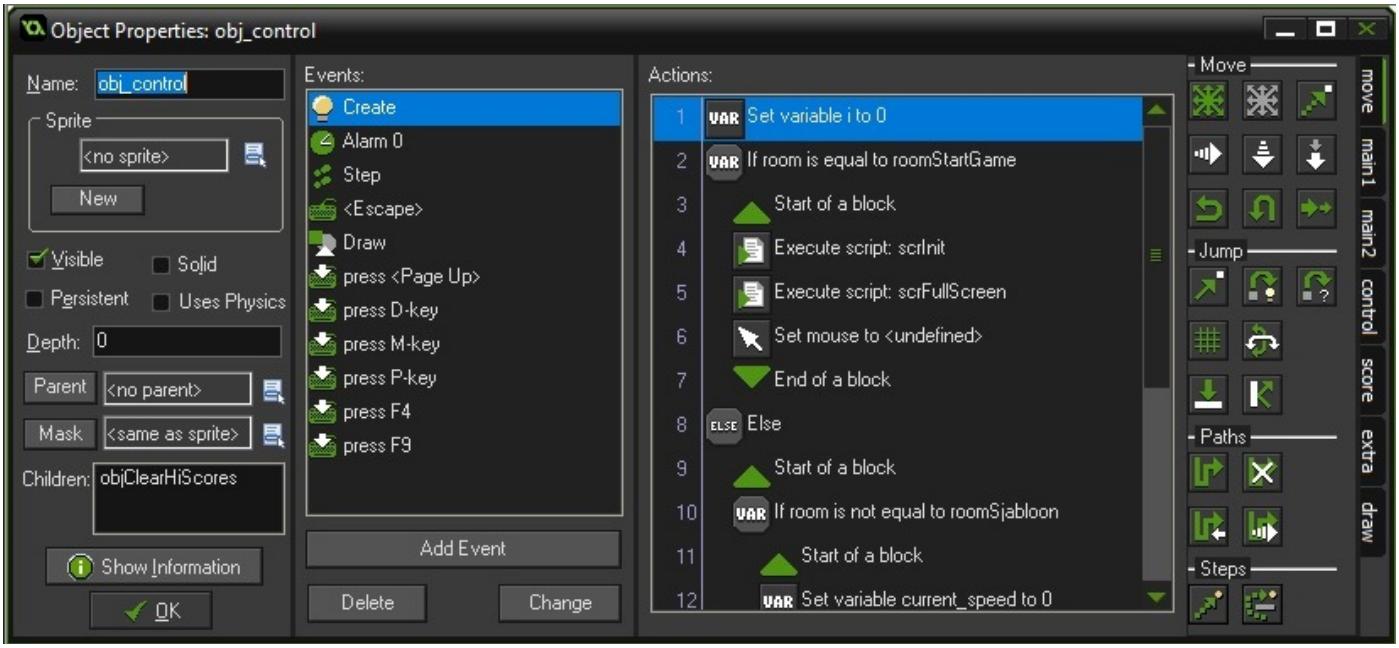
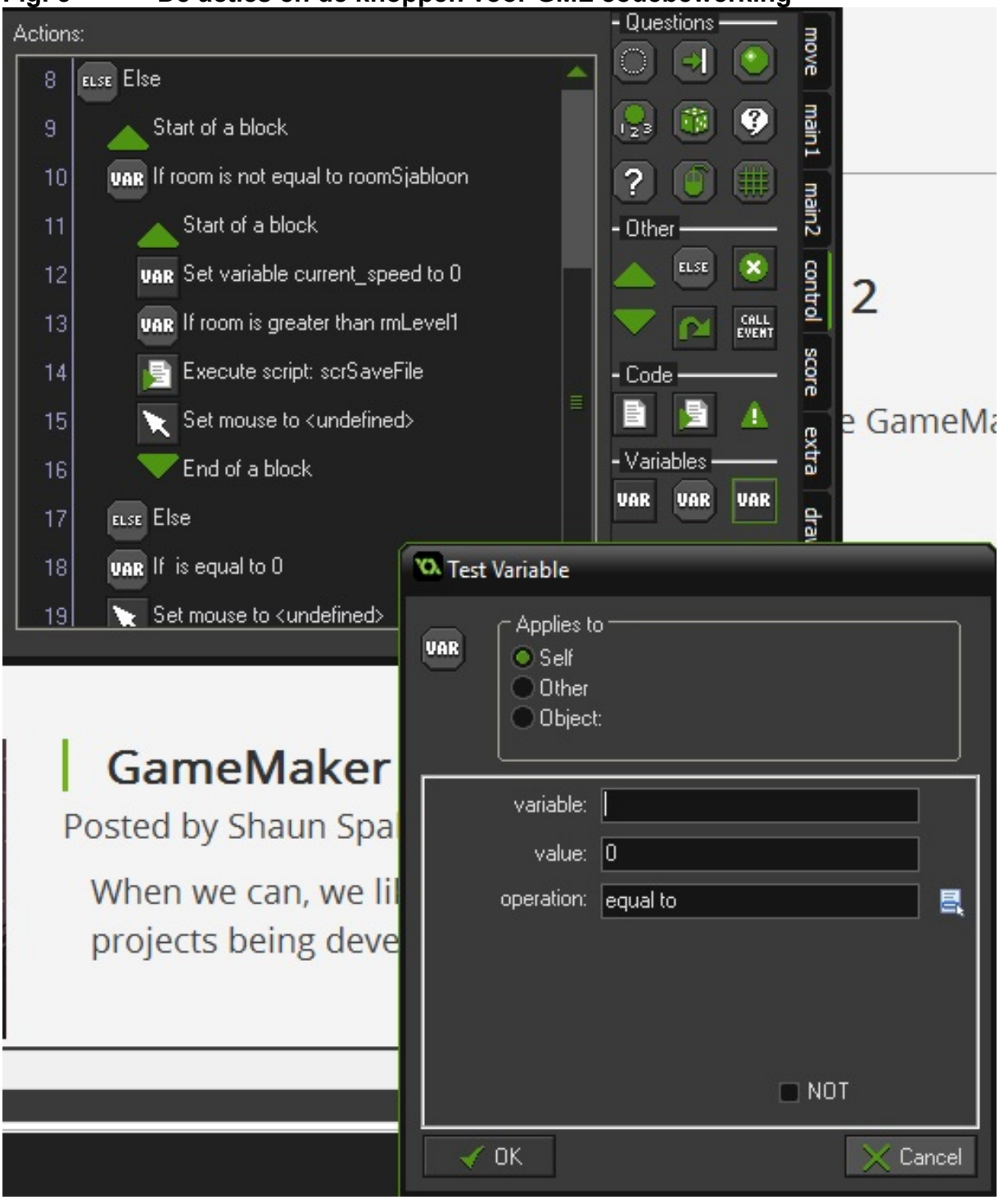



**Fig. 4 Het objectvenster van obj_control**

Dat dit object *obj_control* heet is geen verplichte naam. Ik heb hem zelf zo genoemd. De naam had evengoed obj_start mogen zijn. Dit object is het hoofdobject dat voor de gameloop in elke room moet zorgen. Zonder dit object is het lastig een game te starten, hoewel het wel kan. Een game maken zonder gebruik van de IDE betekent een goede kennis hebben in GML. Het is daarom niet verkeerd om eerst goed te leren hoe de IDE van Game Maker werkt en hoe de scriptcode in elkaar zit. Vergelijk het eens met: eerst de Excel macro's, dan pas het VBA project.

**Het muiswerk**
U kent het vast wel – programma's die de mogelijkheid geven met de muis te kunnen draggen en droppen. Aan de rechterkant van figuur 4 ziet u knoppen en tabbladen. Deze knoppen zijn actieknoppen die naar de actielijst van een event gesleept kunnen worden. De actielijst laat een duidelijke structuur zien en het lijkt wel programmeercode. In principe is dat ook zo. Wat we normaal in code een **if** statement noemen, werkt het in Game Maker IDE als een IF knop. We zien echter een achthoekig symbool met de tekst VAR. De andere VAR ervoor staat op een vierkant (eerste regel). De teksten achter de symbolen vertellen wat de actieknoppen doen. Zodra we een actieknop aanklikken (drag) en naar de actielijst slepen en loslaten (drop), verschijnt er een venster van de actieknop, zie hieronder.

**Fig. 5 De acties en de knoppen voor GML codebewerking**

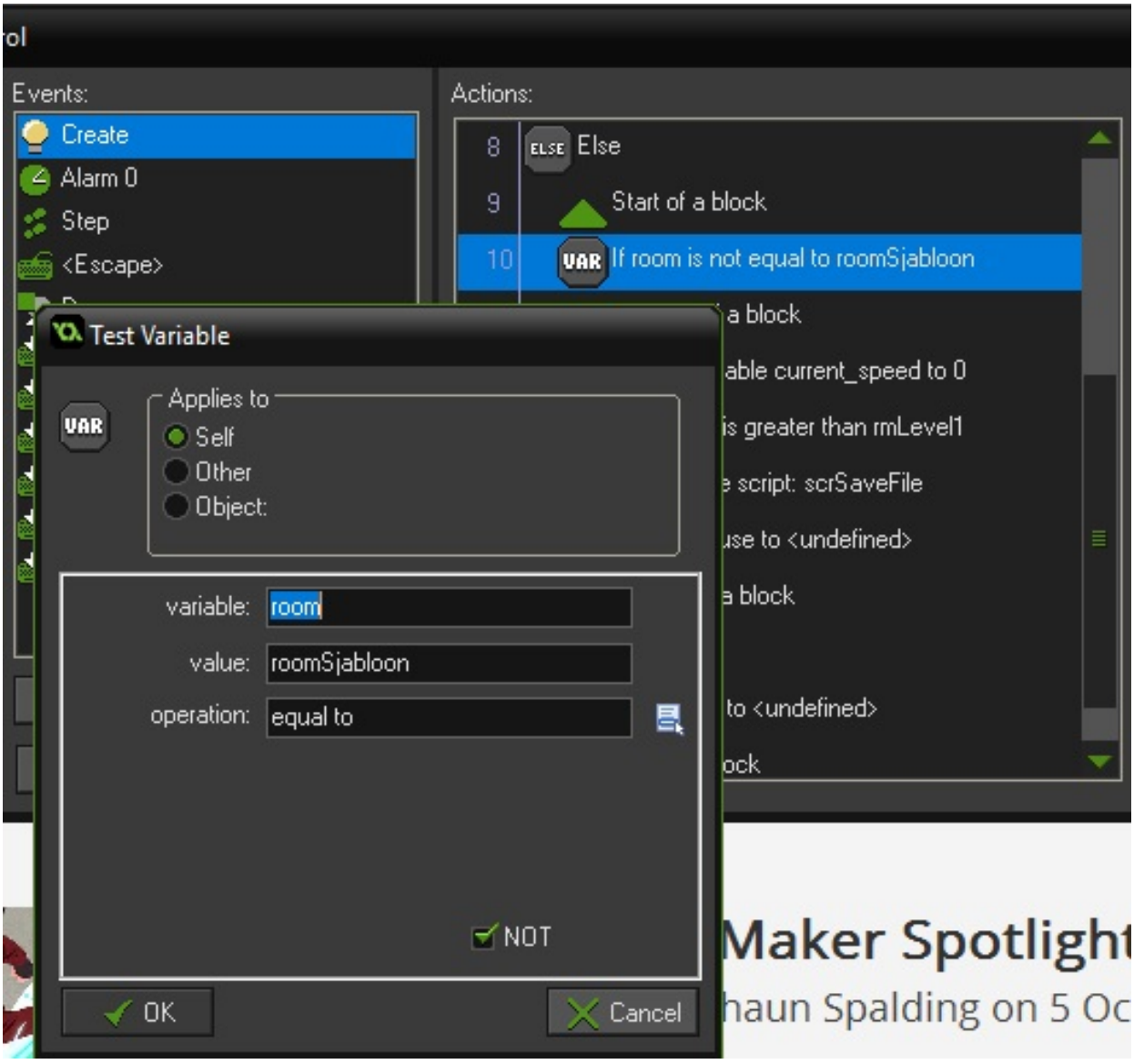



In regel 18 staat de nieuwe actie die we alleen in het **control** tabblad kunnen vinden. Met deze actie kunnen we variabelen en objecten controleren op een bepaalde voorwaarde (*operation*) en of het wel

of niet zo is (*NOT*). Wie programmeren moeilijk vindt, kan met deze acties veel doen en begrijpen wat dit doet.

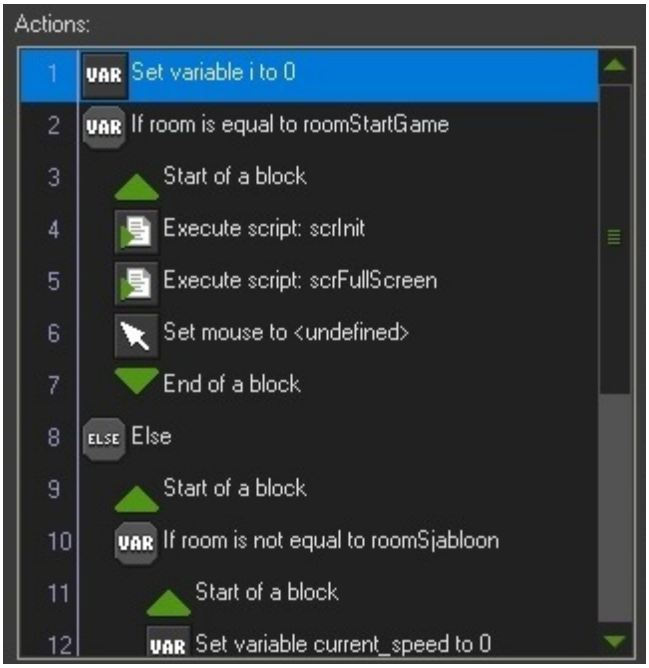
Elke variabele die we hier invoeren heeft te maken met het object dat eraan toebehoort (*Applies to*). In plaats van zichzelf (this of *Self*) kan het ook van een ander object zijn (*Other*), bijvoorbeeld een botsingcontrole, of een vreemd object dat met de andere twee nergens mee te maken heeft. Zodra we die aanklikken, wordt er om de naam van het object gevraagd.

Dubbelklikken we echter op de actie in regel 10, dan zien we hetzelfde venster met de voorwaarde wat het moet doen.



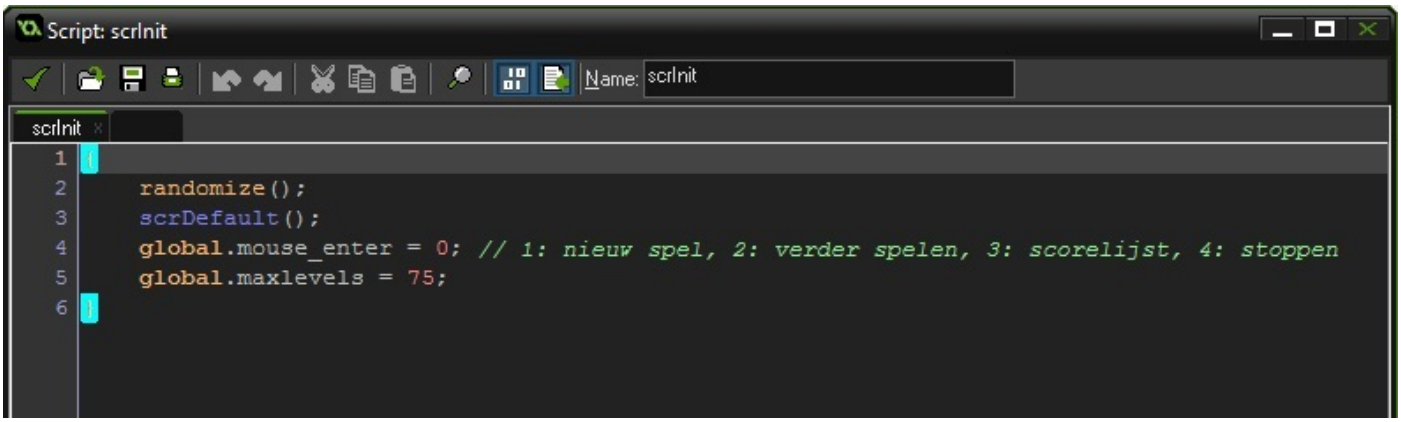

Het lijkt alsof hier de variabele **room** met de waarde **roomSjabloon** van object **Self** vergeleken wordt, maar dat is echter niet het geval. De variabele **room** is een voorgeprogrammeerde globale GML variabele die alleen gelezen kan worden. Het bepaalt zelf welke room op dit moment gebruikt wordt. Als het roomSjabloon niet de huidige room is, wordt het resultaat **true** gegeven. Vandaar dat het vinkje aan staat bij **NOT**. Daarna kan het hele blok, dat tussen de groene driehoeken, maar ingesprongen van regel 10 staat, worden uitgevoerd, anders het andere eventuele ingesprongen blok of een enkele regel. De volgende keer kom ik uitgebreid terug over het objectvenster.

**Een GML script nader bekijken**




**Hier is een deel van de actielijst te zien van het Create event in obj_control. In het blok ziet u twee scripts (regel 4 en 5) die uitgevoerd worden, alleen als de room het roomStartGame object is.**

Het script **scrInit** heb ik gemaakt om de game te initialiseren met de juiste gegevens. Merk op dat ik een ander script in de Init script aanroep. De reden is dat het andere script **scrDefault** aangeroepen wordt als de speler op de New Game knop klikt. Het **scrInit** script mag maar één keer worden aangeroepen als het programma wordt gestart, en dat is dus de obj_control instantie in het roomStartGame object.




Elk script is een codeblok. Het kan een procedure zijn, maar het kan ook als een functie werken. In GML kan een script maximaal 16 parameters hebben. De parameters worden echter niet als argumenten bij **Name** gegeven. De namen van de argumenten heten altijd argument0, argument1, enzovoort.

Variabelen die globaal in de game gedeclareerd worden, moeten altijd beginnen met het sleutelwoord **global** gevolgd door een punt.

Naast globale variabelen kent GML ook lokale variabelen. Het kunnen (alfa)numerieke variabelen zijn, maar ook objectvariabelen. Deze variabelen zijn alleen in het script te gebruiken waarin ze gedeclareerd zijn.

Het script scrBatMove heeft één argument. Het is in GML verplicht om de argumenten toe te kennen aan lokale variabelen voor gebruik. Het argument krijgt als parameterwaarde het paddle-object mee, zodat verder met object **s** de paddle bestuurd kan worden. Eventueel had ook de bal als parameter meegegeven mogen worden maar dat vond ik niet nodig, omdat alleen de paddle in variabele lengte kan zijn.

Variabele **i**, die de lengte van de paddle bepaald, is belangrijk om er voor te zorgen dat de paddle niet buiten het venster zal komen, ongeacht of de muis buiten het venster kan komen.

Zoals u ziet hebben beide scripts dezelfde lokale variabelen **i**, die niets met elkaar te maken hebben. Variabele i in scrBatMove krijgt de waarde die in scrCheckBatLength bepaald zal worden. Wat daadwerkelijk de juiste lengte van de paddle bepaalt is het event dat bestuurd wordt in één van de power-up objecten, en daar de globale variabele global.bat_length de waarde krijgt die u hier in de case regels ziet staan.

Script: scrBatMove

Name: scrBatMove

scrBatMove

```
{
    var i, s;
    s = argument0;
    i = scrCheckBatLength();
    s.y = 736;
    if (mouse_x <= i + 3)
        s.x = i + 3
    else
        if (mouse_x >= room_width - (i + 4))
            s.x = room_width - (i + 4)
        else
            s.x = mouse_x;
    if (global.ball_sticky)
    {
        objPlayBall.x = s.x + 4;
        objPlayBall.y = s.y - 19;
    }
}
```

18/18: 2

Script: scrCheckBatLength

Name: scrCheckBatLength

scrCheckBatLength

```
{
    var i;

    switch (global.bat_length)
    {
        case 0: i = 12; break;
        case 1: i = 19; break;
        case 2: i = 38; break;
        case 3: i = 50; break;
        case 4: i = 76; break;
    }
    return i;
}
```

13/13: 2 INS 10 pt

**Meer mogelijkheden van GML**
GML code kan op twee manieren worden aangemaakt: intern en extern. Tot nu toe hebt u scripts extern gezien. Deze scripts zijn aparte bestanden die in elk game object toegevoegd kunnen worden – ze kunnen dus hergebruikt worden.

GML code kan ook intern worden gebruikt als directe actiecode. Voor elke actie een apart script aanmaken hoeft niet per se, soms is dat zelfs niet aan te raden. Directe actiecode heeft een voordeel: het werkt sneller en het werkt lokaal in het event dat gekozen is. Vaak worden zelfs complete acties in een event alleen maar gemaakt in GML code, waardoor er maar één actieknop in zal staan. Hoe meer actieknoppen gebruikt worden, hoe meer de compiler de code moet verwerken voor de uitvoer. Zie Figuur 5 voor de twee codeknoppen die u kunt kiezen.

De derde knop die ernaast staat is een uitroepteken. Deze knop laat een bericht zien die u in kunt toetsen zodra u de knop naar de actielijst sleept. Het bericht zal dan tijdens de uitvoer worden getoond. Echter raad ik het af om deze knop te gebruiken. In GML code bestaat er een **draw_text()** statement die meer mogelijkheden heeft om tekst op het scherm te kunnen tonen.

Een andere typische mogelijkheid van GML is zijn karakter. GML ziet er niet alleen uit als C code, het ondersteund ook een Pascal structuur. Pascal gebruikers kunnen daardoor gemakkelijk GML gebruiken. Onderstaande GML code ziet u nog eens in GML, maar dan in Pascal structuur.

**C structuur**
```
{
	if (a == 0)
	{
		b = 1;
		klaar = true;
	}
}
```

**Pascal structuur**
```
begin
	if a = 0 then
	begin
		b := 1;
		klaar := true
	end
end
```

De GML compiler accepteert de Pascal code goed. De haakjes bij een voorwaarde zijn optioneel. Ook de puntkomma achter de regel (klaar := true) is niet nodig, net zoals in Pascal deze ook niet nodig is – denk aan het *null-statement,* dat ontstaat als de puntkomma achter het statement wel wordt ingetoetst.
De laatste *end* heeft ook geen puntkomma, maar ook geen punt. GML accepteert dat ook niet. Toetst u het toch in, dan geeft de compiler direct een foutmelding.

GML is ook *case-sensitief*. U kunt een statement niet met een hoofdletter beginnen, ook niet een hoofdletter bij een Pascal statement, zoals *then*. Zodra u het statement met een hoofdletter laat beginnen, verandert direct de kleur van het statement. Oranje is een sleutelwoord kleur. Grijs is de kleur van een variabele (in standaardkleuren). De kleuren kunnen ook zelf ingesteld worden.

**Rommelig werken**

Kijk uit wat u doet in GML. Bepaal zelf welke taalstructuur u kiest, maar onthoud wel dat GML het niet erg vind als u het beide gebruikt. Zo kunt u beginnen met **begin**, maar mag u eindigen met een sluit-accolade **}** of andersom. Ook het toekennen van waarden aan variabelen mag in Pascal structuur, terwijl u geen **then** gebruikt in een **if** regel. GML protesteert niet, maar de code kan rommelig uitkomen en onleesbaar worden.

**Samenvatting**
Game Maker Studio is één van de makkelijkste programma's om games te maken, geschikt voor beginners. Ook geavanceerde programmeurs maken er gebruik van. Ze maken bijvoorbeeld de code in de programmeertaal C en importeren het in Game Maker Studio. Het hoeft in C niet aangepast te worden in GML structuur. Er kunnen zelfs klassenstructuren worden ontworpen en in GML worden gebruikt. GML herkent dan automatisch de klassenmethoden en eigenschappen die in de lijst verschijnen, zodra u begint met de code. Zelf heb ik die manier nog niet uitgeprobeerd, omdat ik dat niet nodig vind wanneer ik leuke spellen ga maken.

Kijk eens op YoYoGames op internet voor meer informatie. In Google kunt u de website van YoYoGames vinden.

Wilt u echte FPS shooter games maken? Dan is **Unity**, zie het volgende onderwerp, een zeer geschikt programma. Ook daar maakt u gebruik van drag en drop en een room. In Unity wordt een room een *scene* genoemd.

---

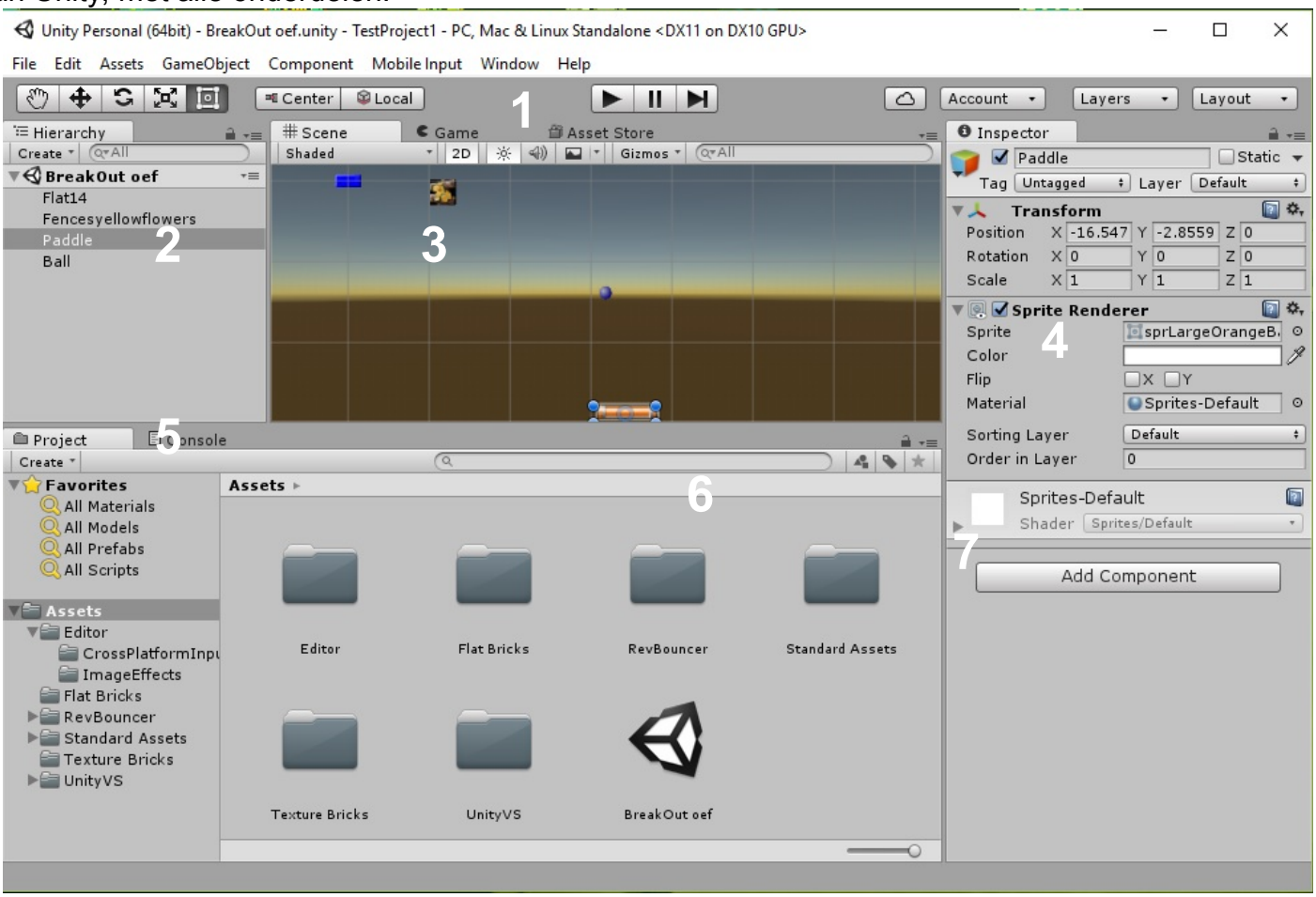
# Van XNA Game Studio 4.0 naar Unity.

In een aantal Bulletins kon u vele onderwerpen volgen over XNA Game Studio, een deel van het grote .NET Framework 4.0. XNA was de instap voor game programmeurs. Ik heb laten zien hoe we een paddle met een bal over het scherm konden laten bewegen en hoe de muis samen met de paddle kon werken.

Hoewel XNA veel mogelijkheden biedt, waardoor het gebruik van DirectX objecten en OpenGL verleden tijd is, was de besturing voor 2D objecten en 3D modellen niet optimaal. Beginners hebben er veel moeite mee. XNA kent bijvoorbeeld geen gemakkelijke manier om een automatische box-collider bij een model te gebruiken, als vaste eigenschap instelling. Zodra een model moet bewegen, moeten alle boxen (meshes) doorlopen worden en op elke nieuwe positie worden gezet. De matrix, het hoofdobject van een model, houdt alle meshes bij elkaar. Maar het hele model kan niet één hele box-collider hebben.

Modellen moeten in aparte tekenprogramma's gemaakt worden. De gratis programma's geven niet altijd de beste mogelijkheden en goede 3D model tekenprogramma's zijn vaak duur.

Unity heeft de mogelijkheid dat er met een IDE gewerkt kan worden. Hieronder ziet u een afbeelding van Unity, met alle onderdelen.




1. Dit is de werkbalk. Hiermee kan de camera worden verplaatst, het scherm worden geroteerd voor een andere kijk, de objecten worden verplaatst en de scene worden bekeken door deze te starten.
2. Dit is de hiërarchie van de scene die in het game project op dit moment wordt gebruikt. Een game project kan namelijk meerdere scenes hebben. In de scene worden alle onderdelen toegevoegd, zoals de Game Objecten.
3. Dit scherm is de scene waar de Game Objecten op geplaatst kunnen worden. Deze sleept u van punt 2 naar het scherm. Zodra u de muisknop loslaat, wordt de Inspector, zie punt 4, ingeschakeld met alle instellingen.
4. Punt 4 bevat alle gegevens van het aangeklikte object op de scene. De Inspector kan variërende gegevens bevatten, zoals de scripts die vanuit de Assets lijst naar het juiste game object gesleept kunnen worden, zie punt 5.
5. De Assets lijst bevat alle gegevens, zoals de sprite bestanden, model bestanden, texturen, sounds, animaties, de scenes en nog veel meer. U kunt ze vanuit de Windows Verkenner in de Assets lijst toevoegen of zelf in de Assets venster aanmaken, zie punt 6.
6. Hier kunt u zelf toevoegen wat u wilt, zoals de mappen om de bestanden te ordenen, en de scenes in op te slaan. Voorbeelden van mappen zijn de materials en de scripts.
7. Materialen en onderdelen moeten in sommige hoofdobjecten samenwerken. Deze moeten eerst in punt 6 worden gemaakt en kunnen dan als componenten worden toegevoegd. Daardoor zal de Inspector (punt 4) variëren door het aantal eigenschappen dat er bij komt.
   De componentenlijst bevat niet alleen componenten van de programmeur. Unity heeft zelf ook componenten die gebruiksklaar zijn, zoals de particles. Die kunnen aangepast worden met uw eigen materialen.

Het voordeel van Unity is dat u zelf uw eigen 3D modellen kunt maken. Elk onderdeel maakt u eerst apart. Wanneer u de onderdelen klaar hebt, kunt u ze op de juiste plaats aan elkaar zetten. Maar als u per ongeluk een onderdeel verplaatst, gaat de rest niet mee. U zou dat zeker wel willen. De oplossing is door een leeg Game Object, via het menu GameObject, toe te voegen. Geef het object de naam die bij de juiste onderdelen bekend moet zijn. Verplaats dan de onderdelen naar het nieuwe object. Nu is er een nieuw model dat zelf onderdelen heeft, en daar heeft u een ander tekenprogramma niet meer voor nodig.

In de volgende Bulletin zal ik laten zien hoe een 3D klok gemaakt wordt en hoe de cube-wijzers met behulp van C# script code kunnen draaien op de juiste seconde.

---

# Liberty BASIC API reference

**Naar het register schrijven**
Windows heeft een archief van informatie over de manier hoe het werkt. Dit wordt Het Register genoemd. Windows kijkt in het register om te zien welke snelkoppelingen er op het bureaublad getoond moeten worden, een recordlijst van eerder geopende bestanden, een recordlijst van keuzes uit bureaublad kleurenschema's, schermresolutie en allerlei soorten ander informatie. Applicaties kunnen uit het register lezen en erin schrijven. Als het register beschadigd raakt, kan Windows niet correct functioneren of sommige applicaties kunnen niet goed functioneren. In het ergste geval is de computer mogelijk niet meer bruikbaar voor alles! Gezien de risico's heeft het opslaan van informatie in het register weinig zin wanneer een ini-bestand prima werkt. Ini bestanden worden later uitgelegd. Indien het noodzakelijk is om direct te werken met het register, zijn de functies als volgt. Voor gedetailleerde informatie voor gebruik van onderstaande functies, bekijk de Microsoft Developers Network Library.
http://msdn.microsoft.com/library/default.asp

| | | |
|---|---|---|
| RegCloseKey | RegConnectRegistry | RegCreateKey |
| RegCreateKeyEx | RegDeleteKey | RegDeleteValue |
| RegEnumKey | RegEnumKeyEx | RegEnumValue |
| RegFlushKey | RegGetKeySecurity | RegLoadKey |
| RegNotifyChangeKeyValue | | RegOpenKey |
| RegOpenKeyEx | RegQueryInfoKey | RegQueryMultipleValues |
| RegQueryValue | RegQueryValueEx | RegReplaceKey |
| RegRestoreKey | RegSaveKey | RegSetKeySecurity |
| RegSetValue | RegSetValueEx | RegUnloadKey |

**INI bestanden**
"Ini" is afgekort voor "initialisatie". Liberty BASIC heeft een INI bestand met als inhoud de lettertype-voorkeuren, de lijst van laatst geopende bestanden, de lijst van externen en andere voorkeuren en records. Het heet lbasicxxx.ini en het bevindt zich in dezelfde map waar ook Liberty BASIC zich in bevindt. Wanneer Liberty BASIC opstart, leest het de informatie uit dit bestand om het gewenste lettertype in te stellen, de lijst van de recente bestanden in het File menu te tonen, enzovoort. Het is mogelijk om dit bestand te openen in een tekstverwerker en het te lezen.

**De API gebruiken om een INI bestand te maken**
De twee functies, WritePrivateProfileStringA en GetPrivateProfileStringA bieden een gemakkelijke en precieze manier om te schrijven naar en te lezen in initialisatie bestanden. Deze "ini" bestanden zijn simpele tekstbestanden. Gebruik deze methode als een alternatief om informatie naar het Windows register te schrijven. Knoeien met het register kan vervelende resultaten hebben, zelfs Windows kan er onbruikbaar door worden. Schrijven naar een privé INI bestand kan alleen invloed hebben op het ene programma. Een programma zou INI bestanden op kunnen nemen van iemands recent geopende bestandenlijst, of zijn voorkeuren voor gebruik van het programma, zoals de syntax kleuren "aan" of "uit", of dat hij zelfs een geregistreerde gebruiker is van het programma.

**Een INI bestand schrijven in API**
WritePrivateProfileStringA wordt gebruikt voor informatieopslag in INI bestanden. Hier is de syntax van de functie, dat een deel is van kernel32.dll:

```
calldll #kernel32, "WritePrivateProfileStringA", _
     Section$ as ptr, _       'sectienaam
     Entry$ as ptr, _         'ingangsnaam
     String$ as ptr, _        'actuele ingang
     FileName$ as ptr, _      'naam van het INI bestand
     result as boolean        'niet-nul = succes
```

**Section$**
Verwijst naar een op null eindigende tekenreeks waarmee de naam van de sectie naar een tekenreeks moet worden gekopieerd. Liberty BASIC voegt automatisch het nodige null-teken toe, dat chr$(0) is, dus het programma hoeft dat niet zelf te doen. Als de sectie niet bestaat, wordt die gemaakt. Een bestand kan uit zoveel secties bestaan als gewenst, maar elke sectie moet een unieke naam hebben. Als het bestand geopend was in een tekstverwerker, ziet de ingang er zo uit – de sectienaam tussen rechte haken:
[section]
Als de sectie genoemd wordt als “user”, ziet het er zo uit:
[user]

**Entry$**
Verwijst naar een op null eindigende tekenreeks met het item of de sleutel die is verbonden met de waarde van de tekenreeks. Als het ingangsitem niet bestaat, zal het worden gemaakt. Is deze NULL (“”), dan zal de hele sectie worden gewist. Een bestand kan zoveel Entry$ sleutels hebben als wenselijk is, maar elk een moet een unieke naam hebben. In een tekstverwerker moet de Entry$ gevolgd worden met een = teken, gevolgd door een string$.
Entry$=String$
Als de Entry$ “naam” heet en zijn waarde is “Carl Gundel”, zal het eruit zien als:
naam=Carl Gundel

**String$**
Verwijst naar een null eindigende tekenreeks die geschreven wordt naar het bestand. Als deze NULL (“”) is, zal de Entry$ ingang gewist worden van het bestand.
Entry$=String$
Als de Entry$ “naam” heet en zijn waarde is “Carl Gundel”, zal het eruit zien als:
naam=Carl Gundel

**FileName$**
Verwijst naar een null eindigende tekenreeks dat het INI bestand heet. Als de naam een volle gekwalificeerde map met de bestandsnaam is, zal die worden gebruikt. Is er geen map, dan zoekt Windows naar het bestand in de Windows directory. Als het niet bestaat, zal het worden gemaakt.

**result**
Resulteert in nul als het niet lukt of succesvol als het niet nul is.

**Voorbeeld**
Hier is een voorbeeld. Na het uitvoeren van deze kleine routine, kunt u kijken in de Windows map naar het bestand “testme.ini”. Open het in kladblok om het resultaat te zien. Merk op dat het programma niet het bestand hoeft te openen of te sluiten, zoals het normaal gesproken wel gedaan wordt als er naar het bestand geschreven moet worden in Liberty BASIC.

```
Section$="User" : Entry$="Name"
String$="Carl Gundel" : FileName$="testme.ini"

calldll #kernel32, "WritePrivateProfileStringA", _
     Section$ as ptr, Entry$ as ptr, String$ as ptr, _
     FileName$ as ptr, result as boolean
```

Het bestand “testme.ini” bevat nu:

```
[User]
Name=Carl Gundel
```

**Een INI bestand lezen in API**

GetPrivateProfileStringA leest een soortgelijk bestand. Veel argumenten zijn hetzelfde als die in WritePrivateProfileStringA. Er zijn wat extra argumenten, die later uitgelegd worden. Section$ en FileName$ zijn eerder uitgelegd in de alinea van WritePrivateProfileStringA.

```
nSize=100
lpReturnedString=space$(nSize)+chr$(0)
calldll #kernel32, "GetPrivateProfileStringA", _
     lpAppName$ as ptr, _     'sectienaam
     lpKeyName$ as ptr, _     'sleutelnaam
     lpDefault$ as ptr, _     'teruggegeven standaard string als er
                              'geen ingang is
     lpReturnedString$ as ptr, _   'bestemming buffer
     nSize as long, _         'grootte van bestemming buffer
     lpFileName$ as ptr, _    'INI bestandsnaam
     result as ulong          'aantal tekens gekopieerd naar de buffer
```

**Entry$**

Verwijst naar een op null eindigende tekenreeks die verbonden is met de stringwaarde. Als deze parameter leeg is, bevat de geretourneerde tekenreeks een lijst van alle waarden in de opgegeven sectie, gescheiden door null-tekens, en afgesloten met dubbele null-tekens.

**Default$**

Verwijst naar een op null eindigende tekenreeks die de sleutelwaarde zal hebben, opgeslagen in het INI bestand. Als de sleutelwaarde niet gevonden kan worden, zal de buffer de inhoud bevatten van de Default$ waarde. Er is een belangrijk verschil in het gebruik van dit string argument in GetPrivateProfileStringA. Het programma zal de gegevens lezen die geplaatst zijn in deze variabelen bij de API functie. Dit betekent dat dit argument doorgegeven: “By Reference” moet zijn. Het op deze manier doorgeven betekent dat het niet de waarde van de string doorgeeft, maar het geheugenadres. De functie kan dan de inhoud van het adres bepalen. Om Liberty BASIC te laten weten dat het doorgegeven moet worden “By Reference”, moet er een null-einde aan de string worden toegevoegd, zoals hier:

```
ReturnString$ = space$(100) + chr$(0)
```

De regel hierboven maakt een buffer of locatie in het geheugen dat groot genoeg is om de gegevens in op te kunnen slaan die dan geplaatst worden bij de functie.

**SizeString**

Deze parameter bepaalt de lengte van de buffer ReturnString$.

**result**

De teruggegeven waarde bepaalt het aantal bytes gekopieerd naar de opgegeven buffer, zonder het null-einde teken. Dit betekent dat het de lengte van de tekst bepaalt, die de functie in de buffer heeft

geplaatst. Het programma kan deze informatie gebruiken om de waarde in het INI bestand in te lezen zonder afwijkende tekens mee te nemen tot het einde.

Hier is een werkend voorbeeld, dat de geschreven waarden uit “testme.ini” leest, die we geschreven hebben in de routine hierboven:

```
Section$="User" : Entry$="Name"
FileName$="testme.ini" : Default$ = "no name" + chr$(0)
SizeString=100
ReturnString$=space$(SizeString)+chr$(0)

calldll #kernel32, "GetPrivateProfileStringA", _
     Section$ as ptr, Entry$ as ptr, Default$ as ptr, _
     ReturnString$ as ptr, SizeString as long, _
     FileName$ as ptr, result as ulong

print "De sleutel is "
print left$(ReturnString$, result)
end
```

Wanneer het gevonden bestand “testme.ini” geopend wordt in Kladblok, ziet het er zo uit:

```
[User]
Name=Carl Gundel
```

In de volgende Bulletin komt het API onderwerp over het klembord. Door gebruik te maken van de API zal het besturen van het klembord veel sneller werken. Het is zelfs mogelijk te werken met afbeeldingen. In Bulletin nummer 1 volgend jaar zullen er mooie voorbeelden bij komen.

---

# Turbo Pascal – Gegevens structureren

In plaats van de normale gegevenstypes dat Pascal kent, zoals Integer en String, kunnen we ook onze eigen gegevenstypes definiëren. In Pascal kunnen we zoveel verschillende soorten gegevenstypes maken en de gegevens ervan structureren, dat alleen uw verbeelding een beperking is.

Wie Pascal niet heeft kan FreePascal downloaden van internet. De mogelijkheden van Turbo Pascal 6 werken ook in de console editor van FreePascal.

Een type-declaratie vertelt de compiler wat meer over nieuwe gegevenstypes die opgenomen moeten worden. Nadat een gegevenstype gemaakt is, kunt u variabelen van dat type declareren om in expressies of als parameters in procedures en functies te gebruiken.

Hieronder ziet u de syntaxis van het geserveerde woord **type**.

**type** <identifier> = <typenaam> [**record**<inhoud> **end**];

- identifier: Een geschikte eigen naam voor het nieuwe type.
- typenaam: Geef hier een typenaam op die al bestaat. Een bereik of een array mag ook.
- inhoud: geef hier een inhoud, zoals velden bij records.

In Pascal kunnen we subranges gebruiken om het grensbereik te beperken. Een normale Integer variabele kan een groter getal aannemen dan u zou willen. Onderstaand programma geeft een voorbeeld hoe we een bereik type kunnen gebruiken.

```
{$R+}
program Grenzen;
type
     Index = 1 .. 10;
var
     i: Index;
begin
     i := 1;
     while i < 10 do
     begin
          Write(i:5);
          i := i + 1
     end;
     Writeln;
     Readln
end.
```

In plaats van een bereik als een nieuw type te maken, hadden we ook gelijk een variabele kunnen declareren met een bereik. Bovenstaande nieuwe type Index heeft echter voordelen. U kunt gemakkelijk het bereik veranderen, en alle variabelen die gedeclareerd zijn van het type nemen automatisch het nieuwe bereik over. Nog een ander belangrijk voordeel van deze types is het doorgeven als parameter naar een argument in procedures en functies. U kunt namelijk geen bereik opgeven als argumenttype. Argumenten moeten eenvoudige types hebben. Onderstaande procedure met argument is dus niet juist:

```
     procedure Werk(i: 1 .. 15);   { ??? }
```

In plaats daarvan is een nieuw gegevenstype noodzakelijk:

```
     type Index = 1 .. 15;
     procedure Werk(i: Index);
```

Een andere soort gegevenstypes zijn enumeraties, ook wel scalaire types genoemd. Een enumeratietype ziet er zo uit:

**type** <eumeratietype> **=** (identifier, identifier, identifier, …);

Onderstaand programma Regenboog laat een voorbeeld zien hoe een enumeratietype werkt.

```
program Regenboog
type
     Kleuren = (Rood, Oranje, Geel, Groen, Blauw, Indigo, Paars);
var
     Kleur: Kleuren;
begin
     Kleur := Geel;
     Kleur := Paars;
```

```
     if Kleur = Paars then
          Writeln('Kleur is Paars');
     Readln
end.
```

De scalaire elementen zijn geen variabelen. Ze worden door de compiler vertaald in sequentiële getallen te beginnen bij nul. In het gecompileerde programma wordt Rood nul, Oranje wordt één, enzovoort. Daarom worden scalaire types enumeratietypes genoemd. Het is niet toegestaan om aan variabele Kleur de waarde 4 toe te kennen. In plaats daarvan moet u schrijven:

```
     Kleur := Blauw;
```

Met de functie Ord() kunnen we de waarden van de scalaire elementen opvragen. Onderstaand programma geeft de opsomming van een variabele enumeratie.

```
program ScalaireWaarden;
type
     Kleuren = (Rood, Oranje, Geel, Groen, Blauw, Indigo, Paars);
var
     Kleur: Kleuren;
begin
     for Kleur := Rood to Paars do
          Writeln(Ord(Kleur));
     Readln
end.
```

We kunnen het ook andersom doen. Het type gebruiken met het scalair element tussen haakjes, zoals:

```
     Kleur := Kleuren(3);
```

Het getal representeert het scalair element. Variabele Kleur krijgt dus de scalaire waarde Groen. De compiler geeft echter een foutmelding als we een waarde gebruiken die buiten het bereik ligt:

```
     Error 76 : Constant out of range
```

De compiler accepteert dus niet:

```
     Kleur := Kleuren(30);
```

Dit kunnen we ook vergelijken met de ASCII waarden. We kunnen bijvoorbeeld een ASCII waarde converteren in de tekenvariabele Kar:

```
     Kar := Chr(67);
```

De variabele Kar wordt hierdoor gelijk aan het teken met de ordinale ASCII waarde 67 – dus een C. We kunnen ook echter de ingebouwde identifier Char() gebruiken:

```
     Kar := Char(67);
```

Zelfs al lijken beiden hetzelfde te doen, toch is er een verschil. Chr() is een ingebouwde Pascal functie. Char() is een enumeratietype.

**Arrays**

Een array is een verzameling waarden van hetzelfde type. Veel waarden kunt u verzamelen in een array zodat u geen last hebt van allemaal losse variabelen.

**type** <arraytype> **= array**[<indexbereik>] **of** <typenaam>

U kunt bijvoorbeeld een arraytype declareren:

```
type
     IntegerArray = array[1 .. 10] of Integer;
```

Arrays kunt u ook direct als variabelen declareren:

```
var
     KarArray: array[0 .. 79] of Char;
```

U kunt zelf uw eigen bereik opgeven, zoals [100 .. 200]. Pascal is slim genoeg om te bepalen dat de posities [1 … 99] niet opgeslagen moeten worden, wat in veel andere programmeertalen wel gedaan wordt, omdat de index bij 0 of 1 moet beginnen. Ook negatieve indexwaarden mogen gebruikt worden. Het volgende is bijvoorbeeld ook correct:

```
var
     Score: array[-10 .. +10] of Integer;
```

U kunt elke integerwaarde toekennen aan een integerarray, zoals deze:

```
Score[4] := 100;
```

**Records**

Een record is een verzameling van variabelen die van verschillende soorten types kunnen zijn. Zoals de typedeclaratie hierboven laat zien, heeft het een inhoud dat uit veldvariabelen bestaat. Het begint met **record** en het eindigt met **end**. De identifiers en types vertonen een gelijkenis met een variabelendeclaratie.

Hieronder is een programma DatumTest met een record dat drie identifiers bevat, Dag, Maand en Jaar.

```
program DatumTest;
type
     DatumRec = record
          Maand     : 0 .. 12;     { 0 = geen datum}
          Dag       : 1 .. 31;
          Jaar      : Integer
     end;
var
     Datum: DatumRec;
begin
     Writeln('Datum Test');
     Datum.Dag := 16;
     Datum.Maand := 5;
     Datum.Jaar := 72;
     Writeln(Datum.Dag, '-', Datum.Maand, '-', Datum.Jaar);
     Readln
end.
```

Voor de toekenningen van de waarden aan de recordvelden is er een kortere methode. Met het **with** statement hoeven we niet meer elke keer met de recordvariabele te beginnen. Het gedeelte van het programma kunnen we verkorten tot:

```
with Datum do
begin
     Dag := 16;
```

```
     Maand := 5;
     Jaar := 72;
     Writeln(Dag, '-', Maand, '-', Jaar)
end;
```

Door het **with** statement weet de compiler dat Dag, Maand en Jaar bij het record Datum horen.

**with** <record identifier>[, …] **do** <statement>|[<statements> **end**];

**Records als bouwstenen**
Een identifier als recordveld kan zelf ook van een recordtype zijn, bijvoorbeeld van een echtpaar waaruit we de naam van de man willen weten.

```
type NaamRec = Record
          Achternaam: string[20];
          Voornaam: string[20]
     end;

     EchtpaarRec = Record
          Man, Vrouw: NaamRec
     end;
```

Merk op dat ik bij het tweede recordtype niet meer het sleutelwoord **type** heb opgegeven. Bij elke volgende typedeclaratie mag **type** weggelaten worden.

We declareren een variabele van het type EchtpaarRec:

```
     var Echtpaar: EchtpaarRec;
```

We kunnen dan op de volgende manier de naam van de man opvragen:

```
     Writeln(Echtpaar.Man.Voornaam);
```

We kunnen dat weer verkorten met een **with** statement:

```
     with Echtpaar.Man do
          Writeln(Achternaam, ', ', Voornaam);
```

**With statements nesten**
We kunnen kiezen uit twee soorten With stijlen. Bekijk onderstaande recordstructuur:

```
Student: record
               Naam: NaamRec;
               Leeftijd: Integer
          end;
```

Om de velden Achternaam en Voornaam af te drukken, kunt u het volgende statement gebruiken:

```
with Student do with Naam do
     Write(Achternaam, ' is ', Leeftijd, ' jaar oud');
```

Het scheiden van de records op verschillende niveaus door komma's is hetzelfde als het gebruik van meerdere **with** statements. In het algemeen kan de vorm:

```
     with r1 do with r2 do with r3 do
```

vereenvoudigd worden tot:

```
    with r1, r2, r3 do
```

# Python – De code leren

## De programmastructuur in Python

**If statements**
Het meest gebruikte statement is wel het If statement, waarmee we op van alles kunnen controleren, op waarden en of er iets bestaat, is de invoer juist, enzovoort.

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...       x = 0
...       print('Negative changed to zero')
... elif x == 0:
...       print('Zero')
... elif x == 1:
...       print('Single')
... else:
...       print('More')
...
More
```

Er kunnen meer elif delen of helemaal geen elif delen staan en het else deel is optioneel. Het sleutelwoord 'elif' is afgekort voor 'else if' en is nuttig om buitensporig inspringen te voorkomen. Een if … elif … elif … is een substitutie voor de switch of case statements, aanwezig in andere programmeertalen.

**For statements**
Het for statement in Python verschilt een beetje van hoe u het gebruikt in C of Pascal. Dit is anders dan telkens te lopen over een aantal nummers (zoals in Pascal en Basic) en geeft de gebruiker de mogelijkheid om zowel de iteratiestap en stopvoorwaarde op te geven (als in C en Basic). Het for statement itereert over de items van een willekeurige reeks (een lijst of een string), in de volgorde waarin ze worden weergegeven. Bijvoorbeeld (geen bedoelde woordspeling):

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...       print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Als u de reeks wilt veranderen terwijl u in de loop itereert (bijvoorbeeld om geselecteerde items te dupliceren), dan is het aanbevolen eerst een kopie te maken. Het itereren over een reeks maakt niet impliciet een kopie. De slice notatie maakt dit bijzonder geschikt:

```
>>> for w in words[:]:   # Loop over a slice copy of the entire list.
...       if len(w) > 6:
...            words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

Als u over een getallenreeks wilt itereren, is de ingebouwde functie range() zeer handig. Het genereert aritmetische rijen:

```
>>> for i in range(5):
...       print(i)
...
0
1
2
3
4
```

Het gegeven eindpunt is nooit het deel van de gegenereerde reeks; range(10) genereert 10 waarden, de legale indices voor items van een reeks met een lengte van 10. Het is mogelijk om het bereik te starten met een ander nummer of een verschillende optelling (zelfs negatief); soms wordt dit de ‘stap’ genoemd:

```
range(5, 10)
     5 tot en met 9

range(0, 10, 3)
     0, 3, 6, 9

range(-10, -100, -30)
     -10, -40, -70
```

Om over de indices van een reeks te itereren, kunt u een range() en een len() als volgt combineren:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...       print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Hoewel het vaker beter is de enumerate() functie te gebruiken, zie de lus techniek volgend jaar in het onderwerp ‘Python en de gegevensstructuren’.

Een vreemd geval gebeurt als u een bereik afdrukt als dit:

```
>>> print(range(10))
range(0, 10)
```

In veel opzichten dat door het object geretourneerd wordt door range(), gedraagt het zich alsof het een lijst is, maar in feite is dat het niet. Het is een object wat de opeenvolgende items van het gewenste reeks resulteert zodra u het itereert, maar het maakt niet echt een lijst, waardoor het ruimte bespaart.

We zeggen: een object is *itereerbaar*, dat wil zeggen, geschikt als een doelwit voor functies en constructies die verwachten dat ze iets waar ze opeenvolgende items tot de levering krijgen kunnen. We hebben gezien dat het for statement een iterator is. De functie list() is anders; het maakt lijsten van iterabels:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Later zullen we meer functies zien die iterabels retourneren en iterabels maken als argument.

**De break en continue statements, en de else clausules over lussen**
Het break statement, als in C, breekt uit de smalste ingesloten for of while lus.

Lus statements mogen een else clause hebben; het wordt uitgevoerd wanneer de lus beëindigd wordt van de lijst (met for) of wanneer de voorwaarde onwaar wordt (met while), maar niet wanneer de lus wordt beëindigd door een break statement. Dit wordt geïllustreerd door de volgende lus, die priemgetallen zoekt:

```
>>> for n in range(2, 10):
...       for x in range(2, n):
...            if n % x == 0:
...                 print(n, 'equals', x, '*', n//x)
...                 break
...       else:
...            # loop fell through without finding a factor
...            print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Jazeker, dit is de juiste code! Kijk goed: de else clause behoort tot de for lus, **niet** het if statement.

De else clause heeft meer gemeen in Python. Niet alleen bij de lussen, zoals in bovenstaande code, maar ook bij het try statement. Een else clause bij een try wordt uitgevoerd als er geen uitzondering wordt waargenomen. Bij de lussen wordt de else clause uitgevoerd als er geen break wordt waargenomen.

Het continue statement, evenzo aanwezig in C, gaat verder met de volgende iteratie van de lus:

```
>>> for num in range(2, 10):
...       if num % 2 == 0:
...            print("Found an even number", num)
...            continue
...       print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

**De pass statements**
Het pass statement doet niets. Het kan worden gebruikt wanneer een statement nodig is, maar het programma niet echt een actie nodig heeft. Bijvoorbeeld:

```
>>> while True:
...       pass      # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

Dit wordt ook gebruikt voor het maken van minimale klassen:

```
>>> class MyEmptyClass:
...       pass
...
```

Het pass statement kan ook als een place-holder voor een functie of conditie-body worden gebruikt wanneer u werkt in nieuwe code, zodat u kunt blijven denken op een meer abstract niveau. De pass wordt genegeerd:

```
>>> def initlog(*args):
...       pass      # Remember to implement this!
...
```

De volgende keer meer over het definiëren van functies.