Opdracht 9 Ballen en Bumpers

– Algoritmisch Denken en Gestructureerd Programmeren in Greenfoot –

1 Inleiding

In de komende opdrachten zullen we steeds minder code en tips voorgeven en zul je het grootste deel van het scenario zelf moeten ontwerpen en implementeren.

In deze opdracht ga je een soort pinball automaat maken. Daarvoor ga je kijken hoe je de beweging van één of meerdere ballen in een tweedimensionale ruimte kunt simuleren.

De simulatie bevindt zich in een Greenfoot-wereld en is dus begrensd. Als een bal zich in deze wereld in een rechte lijn beweegt dan zal hij op een gegeven moment een van de wanden raken. Dan moet deze terug stuiteren.

Maar wat als er meerdere ballen in het spel zijn en de ene bal een andere bal raakt? Of als we andere voorwerpen in de wereld plaatsen waar ballen tegenaan kunnen botsen? Dan wordt het lastiger.

We proberen dit zo realistisch mogelijk te simuleren. Dat wil zeggen, we gebruiken hiervoor de botsingswetten zoals we in natuurkunde geleerde hebben, namelijk de *wet van behoud van impuls* en de *wet van behoud van energie*.

2 Leerdoelen

Na het voltooien van deze opdracht kun je:

- een (2D) simulatie maken voor een echte (3D) situatie;
- formules uit een ander vak(gebied) uit programmeren en toepassen;

3 Instructies

Voor deze opdracht heb je scenario **'MadagaskarOpdr7'** nodig.

4 Opgaven

4.1 Een begin maken

We gaan de begin maken voor onze deze botsende-ballen-simulatie. Je krijgt drie Java klassen cadeau, die voegen we ook aan het scenario toe.

1. Maak zelf een nieuw scenario voor deze botsende-ballen-simulatie. Bedenk zelf een geschikte naam.

2. Voeg de Java klassen toe die je cadeau krijgt:
   (a) Sluit Greenfoot af.
   (b) Haal de klasses `BallWorld`, `Vector` en `SmoothMover` op.
   (c) Plaats deze drie klassen in de map die hoort bij het scenario dat je zojuist hebt aangemaakt.
3. Open het scenario weer.
4. Bekijk de klassediagram. Als alles goed is verschijnen nu de drie klassen in het Greenfoot project-venster. Zoals je ziet komt de klasse `Vector` in het klassevenster helemaal onderin te staan en niet bij de `Actor`-klassen. Het is ook geen `Actor`. Er komen nooit objecten van deze klasse zelfstandig in de wereld voor. De klasse `SmoothMover` is een hulpklasse die wél een uitbreiding vormt van `Actor`. We bespreken deze klasse zo meteen.
5. Compileer het scenario. Runnen heeft nu nog geen zin, daarvoor moet je zelf eerst nog wat programmeren!

4.2 Vloeiende bewegingen

Om de simulatie realistisch te maken moeten de ballen vloeiend over het scherm bewegen. In Mimi's wereld (die maar uit 12 bij 12 cellen bestaat) kan dat niet. De acteurs in die wereld springen van de ene cel naar de andere. De kleinste stap die een acteur kan maken is die van cel naar cel, tussen twee cellen in staan gaat niet. Daardoor worden de bewegingen schrokkerig. Dat is trouwens niet specifiek voor Mimi's wereld maar geldt voor Greenfoot scenario's in het algemeen.

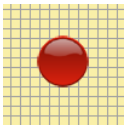
4.2.1 Cel grootte

Om bewegingen vloeiender te maken moeten we kleinere cellen gebruiken. De celgrootte wordt uitgedrukt in *pixels*: puntjes op je beeldscherm. In Mimi's wereld zijn de cellen 60 bij 60 pixels groot. En in 'Mimi de magazijnwerkster' (opdracht 8) waren de cellen al ietsje kleiner gemaakt zodat alle levels zonder problemen op het scherm pasten. De celgrootte die hier werd opgegeven was 50 bij 50 pixels. Deze grootte is gespecificeerd in `Madagaskar`.

1. Open het scenario voor opdracht 8 (Sokoban: 'Mimi de magazijnwerkster').
2. Bekijk de code van `Madagaskar`. Waar staat de celgrootte aangegeven?
3. Waar wordt de celgrootte gebruikt?
4. Waarom staat dit in hoofdletters? Hoe heet zoiets? Tip: Kijk eventueel terug bij de uitleg van opdracht 7.

De celgrootte die we voor deze simulatie willen gebruiken is 1 bij 1. Iedere cel bestaat precies uit één pixel. In scenario's met grote cellen passen de afbeeldingen van de acteurs (meestal) precies in een cel. Bij kleine cellen is dat niet het geval: afbeeldingen bedekken vaak meerdere cellen. Dit verschil kun je in de volgende plaatjes zien:

Figuur 1: Wereld met lage resolutie.



Figuur 2: Wereld met hoge resolutie.

Bij een hoge resolutie heb je veel cellen nodig voor een wereld die groot genoeg is om een simulatie in uit te voeren. Een wereld die uit 12 bij 12 cellen bestaat, zoals bij Mimi, is natuurlijk veel te klein. Wellicht is 600 bij 500 cellen al een stuk realistischer. Door klasseconstantes te gebruiken kun je later de grootte heel eenvoudig aanpassen.

Resolutie van de wereld De resolutie geef je aan in de World-constructor aanroep, door de grootte van de cellen aan te geven en de hoogte en breedte van de wereld.
Voorbeeld: Het aangeven van de resolutie zie je in de volgende uitbreiding van de `World`-klasse:

```
public class BallWorld extends World
{
    private static final int WIDTH    = 600;
    private static final int HEIGHT   = 500;
    private static final int CELLSIZE = 1;

    /**
     * Maak de ballenwereld. Deze wereld is WIDTH x HEIGHT cellen groot
     * waarbij iedere cel overeenkomt met CELLSIZE pixels
     */
    public BallWorld( ) {
        super( WIDTH, HEIGHT, CELLSIZE );
    }
}
```

We passen nu de resolutie van onze scenario aan:

1. Open de code voor `BallWorld`.
2. Pas de code aan zoals in het bovenstaand theorieblok is voorgesteld.

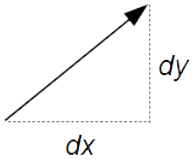
4.2.2 Vector

We bekijken de klasse `Vector` die je cadeau hebt gekregen. Als je deze begrijpt, kun je hier straks in jouw bal-klasse handig gebruik van maken.

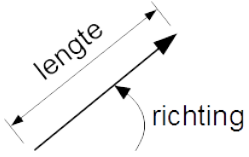
Vectoren Vectoren kun je op twee verschillende manieren voorstellen:

1. De *Cartesische* vorm: als een paar met afstanden, uitgedrukt in $x-$ en $y-$ componenten;
2. De *Polaire* vorm: als een paar met een richting en een afstand, uitgedrukt in een richting en een lengte.

Hieronder zie je eenzelfde vector op deze twee verschillende manieren weergegeven.



Figuur 3: Vector in Cartesische vorm.



Figuur 4: Vector in Cartesische vorm.

Voor sommige operaties is het handiger om de Cartesische vorm te gebruiken, en voor andere is juist de Polaire vorm gemakkelijker. In de voorgegeven Java-implementatie worden

beide vormen gebruikt en wordt er (intern) automatisch voor gezorgd dat de informatie van een Vector-object altijd up-to-date is.

We bekijken nu welke methoden er in de gegeven `Vector` klassen staan zodat je daar straks handig gebruik van kunt maken:

1. Open de klasse `Vector`.
2. Bekijk de 2 constructors van deze klasse. Wat is het verschil tussen deze constructors? Hoe zorg je ervoor dat je bij het maken van een nieuwe `Vector` instantie de juiste constructor wordt aangeroepen?
3. Ga na welke operaties op vectoren gedefinieerd zijn. Wat doen deze operaties?

4.3 Ball

Om een bal op het scherm te tonen en deze te laten rollen, heb je natuurlijk een klasse `Ball` nodig. Je gaat nu de klasse `Ball` ontwerpen en implementeren. Omdat een bal soepel moet rollen wordt deze een subklasse van `SmoothMover`. Deze klasse bekijken we straks.

4.3.1 De klasse `Ball`

Als eerste stap voeg je de klasse `Ball` aan jouw scenario toe.

1. Breid de `SmoothMover` klasse uit met de klasse `Ball`:
   (a) Klik in het klassediagram (met je rechtermuisknop) op `SmoothMover` en selecteer *New subclass ....*
   (b) Vul in de dialoog die dan verschijnt de naam voor de klasse in en kies een geschikt plaatje voor jouw bal.
   (c) Klik op *OK*.
2. Compileer jouw scenario. Je krijgt waarschijnlijk een foutmelding die er op neerkomt dat je in jouw Ball constructor de `SmoothMover` constructor niet aanroept. Dit los je op door:
   (a) Met **`super`** roep je de constructor aan vanuit de klasse die je uitbreidt.
   (b) Aan **`super`** geef je dezelfde parameters mee als de `SmoothMover` constructor verwacht. Zo'n aanroep ben je ook al eerder tegengekomen in uitbreidingen van de wereld klasse. Kijk bijvoorbeeld naar de code van `BallWorld`.
   (c) Voeg een aanroep van **`super`** toe aan jouw Ball-constructor en controleer opnieuw of je het scenario nu wel compileert.
3. Het is ook handig om de straal van de bal expliciet bij te houden.
   (a) Voeg hiervoor een geschikte instantievariabele toe.
   (b) De waarde van de straal kun je aan de hand van het plaatje, dat je hebt gekozen voor een bal, bepalen.
   (c) Het plaatje kun je ophalen met `getImage( )` en de breedte vervolgens met `getWidth( )`. Zoek in de Greenfoot documentatie op hoe je deze methodes dient te gebruiken.

4.3.2 De bal gaat rollen...

De `Ball` klasse is een subklasse van `SmoothMover`. We gaan nu de klasse `SmoothMover` bekijken.

1. Open de klasse `SmoothMover` in de editor.
2. Hierin zie je bovenaan drie instantievariabelen:
   - `myXCoord` en `myYCoord` van het type double: hiermee wordt de positie van de een `SmoothMover` bijgehouden;
   - `myVelocity` van het type vector: hiermee wordt de snelheid bijgehouden.
3. In vorige opdrachten hebben we voor Mimi en eieren ook coördinaten opgevraagd. Toen hebben we gebruik gemaakt van de `getX( )` en `getY( )` methodes uit `Actor`.
   (a) Wat is het verschil tussen het type van `myXCoord` en het returntype van `getX( )`?
   (b) Waarom zouden we hier liever `myXCoord` gebruiken?
4. De constructor van de klasse verwacht de initiële snelheid van de `SmoothMover` uitgedrukt in polaire vorm. (Tip: kijk eventueel terug naar het theorieblok over vectoren)
5. Met de methode `move` wordt de `SmoothMover` één stap verplaatst.
   - Als de huidige positie van de bal (x, y) is en de snelheid bedraagt $\vec{v}$, dan is de nieuwe positie $(x + v_x, y + v_y)$.
   - Zoals je ziet, wordt hierbij gebruik gemaakt van de Cartesische vorm van vector $\vec{v}$. (Tip: kijk eventueel terug naar het theorieblok over vectoren)
6. Bekijk van de overige methodes nog de *getters* van de instantievariabelen.

4.3.3 Stuiterbal

We gaan nu een hele eenvoudige implementatie maken van de bal klasse om deze vervolgens stapsgewijs uit te breiden totdat hij probleemloos werkt.

1. Open de code voor `Ball`.
2. Roep allereerst de `move( )` methode aan vanuit `act( )`.
3. Compiler en run het scenario.
4. Er zou nu wel iets moeten gebeuren. Beschrijf wat je ziet.
5. De bal houdt nu nog geen rekening met het feit dat de wereld begrensd is. Zoals we ook al met Mimi zagen is in Greenfoot ingebouwd dat een `Actor` niet van de wereld af kan vallen, maar wordt tegengehouden. Schrijf een methode die controleert of een de wereldgrenzen bereikt is. Tip: Ga dit na door te controleren of de afstand van het middelpunt van de bal tot de wand kleiner is dan de straal van de bal.
6. We gaan een methode schrijven zodat de bal terugstuitert als deze tegen een wand opbotst:
   (a) Ga na wat er precies aan een vector (de snelheid dus) verandert als de bal (recht) terugstuitert. Tip: de bewegingsrichting keert om.
   (b) Schrijf een methode **`void`** `handleWallCollision()` die allereerst controleert of een van de wereldgrenzen bereikt is en, indien dat het geval is, de snelheid van de bal aanpast.

4.4 Bumpers

We gaan nu stilstaande obstakels aan het scenario toevoegen waar de bal tegenop kan botsen. Bij pinball heten deze obstakels *bumpers*. Hiervoor voegen we een klasse `Bumper` toe.

1. `Bumper`-objecten zijn rond van vorm en kunnen zelf niet bewegen. Bedenk welke klasse je hiervoor het beste met de klasse `Bumper` kunt uitbreiden.
2. Breid de klasse `Actor` uit met een klasse genaamd `Bumper`.
3. Welke instantievariabele is handig om aan de klasse `Bumper` toe te voegen? Tip: bekijk hiervoor opgave 4.3.1 onderdeel 3.
4. Voeg deze instantievariabele toe en initialiseer die op dezelfde wijze als bij de ballen.
5. Welke klasse is verantwoordelijk voor (regelt) het terugkaatsen van de ballen? Wat blijft er dan nog over voor de `Bumper`. Tip: De definitie van `Bumper` mag uiterst eenvoudig zijn.

4.4.1 Botsen tegen bumpers

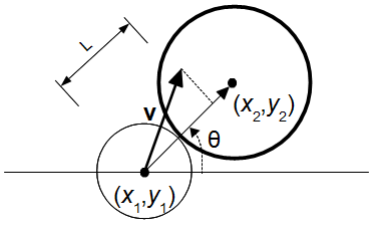
Om de ballen op juiste manier op bumpers te laten reageren is het goed om eerst te analyseren wat er precies dient te gebeuren als een bal een bumper raakt.

In feite verschilt dit niet veel van de situatie waarin de bal een van de muren raakt: de component van de snelheid in de richting van de muur klapt om.

Voor bumpers geldt hetzelfde: zodra een bal een bumper raakt bepaal je de component van de snelheid van de bal in de richting van de bumper, om deze om te keren. Dit wordt toegelicht in het volgende theorieblok.

Bal botst tegen bumper Als een bal een bumper raakt dan klapt de component van de snelheid in de richting van de bumper om.

Dit wordt toegelicht in het volgende plaatje:



- De bal (middelpunt (x_1, y_1)) raakt de bumper (middelpunt (x_2, y_2)) onder een hoek θ.
- Om het effect van de botsing op de bal te bepalen moeten we L uitrekenen, de component van de balsnelheid $\vec{v}$ in de richting van de bumper. Dit kan relatief eenvoudig door de vector $\vec{v}$ met de wijzers van de klok mee te roteren over een hoek $-\theta$. L is dan hetzelfde als de $x-$component van de geroteerde vector.
- Door de botsing dient de $x-$component geïnverteerd te worden. De uiteindelijke snelheid krijgen we door de veranderde vector weer terug te roteren over een hoek θ.

We werken dit algoritme stap voor stap in Java uit.

1. Voeg aan de klasse `Ball` een methode **`void`** `handleBumperCollision`() toe die:

(a) Eerst controleert of de bal een bumper raakt. Hoe bepaal je of een bal een bumper raakt? Tip: Je kunt alle bumper objecten ophalen met de `world` methode `getObjects( Class cls )`. Deze levert je een lijst van bumpers op die je kunt doorlopen. Per bumper kun je bekijken of de bal dicht genoeg in de buurt van de bumper is.

(b) Als de bal een bumper raakt, dan voer je achtereenvolgens de volgende stappen uit:

i. bepaal de hoek θ. Tip: Maak een nieuwe vector met als begin- en eindpunt de middelpunten van resp. de bal en de bumper.); Hoe kom je nu makkelijk aan θ?

ii. roteer de snelheid van de bal over een hoek $-\theta$;

iii. inverteer de $x-$component van deze snelheid;

iv. roteer de snelheid wederom, ditmaal over een hoek θ.

2. Voeg een aanroep van deze methode toe aan de `act` methode van `Ball`.

3. Compileer en voer het scenario uit. Werkt het zoals je verwacht? Pas zo nodig jouw code aan.

4.5 Bewegende ballen botsen tegen elkaar

We gaan nu het botsen van de ballen zelf op de juiste manier implementeren.

In feite lijkt de situatie veel op wat we met gedaan hebben met de ballen en de bumpers. Het enige verschil is dat het voorwerp waarmee de bal in botsing komt niet stilstaat, maar zelf ook beweegt en door deze botsing zelf ook een andere snelheid zal krijgen.

Energiewet bij frontale botsing Met de energiewetten kun je uitrekenen dat wanneer ballen van gelijke massa recht op elkaar botsen ze na afloop elkaars snelheid hebben uitgewisseld.

Dus als bal 1 een snelheid v_1 heeft en bal 2 een snelheid v_2 dan zal na de botsing gelden dat bal 1 snelheid v_2 en bal 2 snelheid v_1 heeft.

Maar in algemeen zullen de ballen elkaar niet frontaal raken, maar is er, net zoals bij het bal-bumper geval, sprake van invalshoek θ (die overigens op dezelfde wijze berekend kan worden).

We passen de code nu aan zodat twee ballen onder een hoek tegen elkaar kunnen botsen:

1. Ga nu zelf na hoe je vervolgens de nieuwe snelheden van de botsende ballen kunt bepalen (Hint: gebruik weer rotaties om de juiste componenten van beide snelheden te bepalen en wissel die vervolgens uit).

2. Werk dit geheel uit in een methode genaamd **`void`** `handleBallCollision`() die je weer vanuit je `act` () methode aanroept.

3. Compileer en run je scenario en verbeter, indien nodig, je implementatie totdat deze naar behoren werkt.

4.6 Optioneel: Verschillende maten ballen

Pas jouw programma aan zodat die een simulatie maakt met ballen van verschillende groottes.

Een aantal tips:

- Je kunt zelf een plaatje maken waaar je later op kunt 'tekenen' en dit koppelen aan een actor met behulp van de Greenfoot klasse `GreenfootImage`. Gebruik dan de constructor **`public`** `GreenfootImage(` **`int`** `width,` **`int`** `height )` om een rechthoekig plaatje te maken waarin de bal precies past. Hoe groot zijn `width` en `height`.

- Teken een cirkel. Het beste kun je eerst de cirkel inkleuren en daarna de rand tekenen.

1. Bedenk eerst wat de kleur van je bal wordt. Gebruik `setColor(Color c)` om de pen waarmee getekend wordt de gekozen kleur te geven.
2. Met de `GreenfootImage` methode `fillOval(int x, int y, int width, int height)` kun nu de cirkel inkleuren. Bekijk de Greenfoot API om te zien wat de bedoeling van de parameters is.
3. Als je de cirkel een zwart randje wil geven moet je eerst de pen (weer) op zwart zetten. Daarna teken je de rand met `drawOval( int x, int y, int width, int height )`.
4. Gebruik verder de `Greenfoot.getRandomNumber(int limit)` om een willekeurige grootte en willekeurige kleur te gebruiken.

4.7 Optioneel: Ballen van verschillende massa's

Het laatste onderdeel van de deze opdracht is facultatief.

Tot nu toe hebben alle ballen dezelfde massa. De simulatie wordt iets complexer (en realistischer) als we ballen met verschillende massa's toestaan.

Energiewet bij botsing met verschillende massa's Bij twee botsende ballen (zeg met massa m_1 en m_2), waarbij m_2 stilstaat en m_1 met snelheid v recht op m_2 botst, kunnen we eenvoudig bepalen wat de snelheden u_1 en u_2 van beide massa's na de botsing zijn. Deze worden gegeven door de volgende formules:

$$u_1 = \frac{m_1 - m_2}{m_1 + m_2} v \qquad u_2 = \frac{2m_1}{m_1 + m_2} v$$

We breiden nu de code uit voor ballen van verschillende massa's:

1. Evenals in het vorige onderdeel bewegen natuurlijk beide ballen en hoeft er geen sprake te zijn van een frontale botsing. Ook nu kun je dit algemene geval weer terug brengen tot het ééndimensionale geval d.m.v. een rotatie en een transformatie. Ga precies na hoe dit werkt. Tips:
   - Druk eerst de snelheid van bal 1 ($\vec{v_1}$) uit t.o.v. bal 2 (met snelheid $\vec{v_2}$). Dit is heel simpel: $\vec{v_1} - \vec{v_2}$.
   - Roteer dit resultaat over een hoek $-\theta$.
   - Bereken nu met bovenstaande formule de snelheden na de botsing en converteer alles weer terug naar de oorspronkelijke situatie.
2. Werk dit vervolgens uit in Java code.
3. Compileer en test je toevoeging.
4. Pas nu de code aan zodat de massa overeenkomt met de grootte van de bal. Hoe groter de bal, hoe zwaarder die is.
5. Compileer en test je toevoeging.

5 Samenvatting

Je hebt geleerd:

- complexe wetten uit de natuurkunde te simuleren met een zelf geschreven programma.
- formules uit te drukken in code.

6 Opslaan en inleveren

6.1 Opslaan

Je bent klaar met de negende opdracht. Sla je werk op, want je hebt het nodig voor de volgende opdrachten.

1. Kies in Greenfoot 'Scenario' in het bovenste menu, en dan 'Save As ...'.
2. Vul de bestandsnaam aan met jouw eigen naam en het opgavenummer, bijvoorbeeld `Opdr9_Michel`.

Alle onderdelen van het scenario bevinden zich nu in een map die dezelfde naam heeft als de naam die je hebt gekozen bij 'Save As ...'.

6.2 Inleveren

Lever de opdracht in via Blackboard.

1. Ga naar het map waar je jouw werk hebt opgeslagen.
2. Standaard wordt er bij ieder scenario een 'README.TXT' bestand gegenereerd. Open het bestand 'README.TXT' en vul hier jullie namen en studentnummers in.
3. Comprimeer de hele map tot één `.zip` bestand. Onder Windows kun je dit doen door er op de klikken met je rechtermuisknop en uit het pop-up menu 'Send to' en dan 'Compressed (zipped) folder' te kiezen.
4. Lever dit ene gecomprimeerde bestand in via Blackboard: ga naar 'assignments'. Druk op 'uploaden' en vervolgens op 'Submit'.

Hou je aan de deadline die in Blackboard voor deze opdracht staat aangegeven.