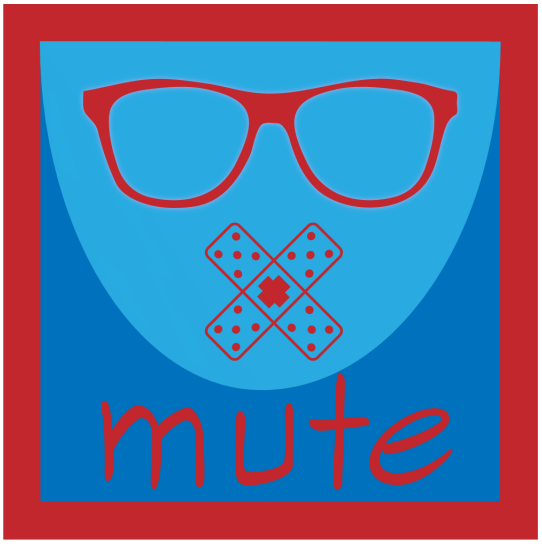
Learning C++ with Esenthel

Mute (http://mutecode.com)

Revision: 5 april 2016



Inhoudsopgave

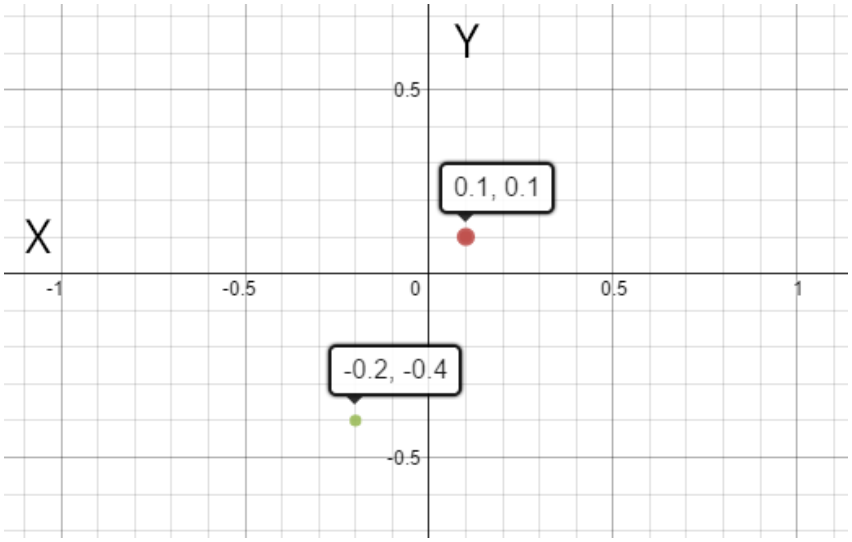
Deel I

2D Concepts

HOOFDSTUK 1

Posities in 2D

Posities in 2D hebben een x- en een y-coördinaat, net zoals een punt in een grafiek. In Esenthel staat het nulpunt voor het midden van het scherm. Positieve waarden op de X-as staan rechts, negatieve waarden links. Positieve waarden op de Y-as staan bovenaan, negatieve waarden beneden.



Figuur 1.1: 2D coördinaten.

In Esenthel gebruiken we de class `Vec2` om een 2D coördinaat weer te geven. Van een `Vec2` kan je zowel de x als de y waarde instellen:

```
Vec2 pos;
pos.x =  0.1; // de positie op de x-as wordt 0.1
pos.y = -0.3; // de positie op de y-as wordt -0.3
```

```
pos   =  0.5; // beide assen krijgen de waarde 0.5
pos.set(0.1, -0.3); // pas beide assen aan via de functie set(float x,
    float y)
```

1.1 Posities Tonen op het Scherm

Alhoewel je de class **Vec2** vooral gebruikt om posities te berekenen, kan je een coördinaat ook op het scherm tonen. Daarvoor bevat de class de functie **draw(Color)**. Als argument geef je de kleur waarin je je punt wil weergeven.

```
Vec2 p1(0.2, 0.4); // maak een punt p1 en stel x en y in via de
    constructor.
Vec2 p2;           // maak een punt p2.

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   p2.set(-0.2, -0.5); // stel p2 in via de set functie
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;

   return true;
}

void Draw()
{
   D.clear(BLACK); // Maak het scherm leeg
   p1.draw(RED  ); // teken p1 rood
   p2.draw(BLUE ); // teken p2 blauw
   Vec2(0 ,0).draw(GREEN); // maak een tijdelijk punt en teken dit groen
}
```

1.2 Rekenen

Je kan met `Vec2` ook rekenen, net zoals je met getallen doet. Het is mogelijk om de x- en de y-coördinaat afzonderlijk aan te passen, maar je kan ook rechtstreeks rekenen met een `Vec2`. In dat geval wordt de gekozen berekening voor zowel de x- als de y-as uitgevoerd:

```
Vec2 p1(0.1,  0.3);
Vec2 p2(0.3, -0.1);
Vec2 p3 = p1 + p2;  // x: 0.4 , y: 0.2
p3 -= 0.1;          // x: 0.3 , y: 0.1
p3 *= 2;            // x: 0.6 , y: 0.2
p3 = p1 / 2.f;      // x: 0.05, y: 0.15
```

1.3 Punten om te onthouden

In Esenthel betekent `Vec2(0,0)` steeds het midden van het scherm. Maar de randen van het scherm zijn minder duidelijk. Niet elk computerscherm heeft immers hetzelfde formaat. Daarom kan je via het object `D` (display) de breedte en de hoogte van het scherm opvragen via de functies `w()` en `h()`. Zo kan je eenvoudig enkele punten berekenen.

```
Vec2 middle(0,0);
Vec2 left(-D.w(), 0);
Vec2 right(D.w(), 0);
Vec2 rightUpperCorner(D.w(), D.h());
Vec2 leftUpperCorner(-D.w(), D.h());
```

Indien je een punt wil tekenen op afstand 0.1 van de linkerbovenhoek, dan kan je dat dus zo doen:

```
Vec2(-D.w() + 0.1, D.h() - 0.1).draw(PINK);
```

1.4 Samengevat

- ❑ Punten in 2D hebben een x- en een y-coördinaat. In Esenthel gebruik je de class `Vec2` om een punt weer te geven.
- ❑ De class `Vec2` heeft een functie `draw(Color)` om het punt op het scherm te tonen.
- ❑ Je kan rekenen met een `Vec2`, net zoals met getallen.
- ❑ `Vec2(0,0)` is steeds het midden van het scherm.
- ❑ De randen van het scherm kan je berekenen via `D.w()` en `D.h()`.

1.5 Oefening

Maak een programma dat de volgende punten toont:

1. Een wit punt in het midden van het scherm.
2. Een rood punt op afstand 0.1 van de linkerkant van het scherm.
3. Een blauw punt op afstand 0.2 van de rechterbovenhoek.
4. Een geel punt op afstand 0.35 van de onderkant van het scherm, op 2/3 van de totale schermbreedte.

HOOFDSTUK 2

Interactie

De meeste interacties met een programma gebeuren via het toetsenbord en de muis. (Mobile devices gebruiken vooral touches, maar die komen later aan bod.) In dit hoofdstuk overlopen we de verschillende mogelijkheden.

2.1 Muis Interacties

De class `Mouse` vind je in Esenthel Engine ⇒ Input ⇒ Mouse. Computers zijn niet voorzien om meer dan één mouse pointer te tonen, dus Esenthel voorziet alvast een object van de class: `Ms`. Het is dus onnodig om zelf een object van de class `Mouse` te maken.

Als je even naar de beschikbare functies kijkt in deze class, dan zie je dat er heel wat mogelijkheden zijn. Meestal zijn we vooral geïnteresseerd in 'clicks' en positie.

2.1.1 Positie

De positie van de muis op het scherm is een `Vec2`. Je kan de huidige positie opvragen via de functie `Ms.pos()`. De volgende code toont een `Vec2` op het scherm. De positie is steeds gelijk aan de positie van de muis.

```
Vec2 mousePos;

void InitPre()
{
    EE_INIT();
```

```
}

bool Init()
{
    return true;
}

void Shut() {}

bool Update()
{
    if(Kb.bp(KB_ESC)) return false;
    mousePos = Ms.pos();
    return true;
}

void Draw()
{
    D.clear(BLACK);
    mousePos.draw(RED);
}
```

Time for Action

Pas het bovenstaande voorbeeld aan, zodat mousePos iets onder de positie van de muis wordt getoond.

2.1.2 Clicks

Je kan via het `Ms` object vanalles te weten komen over de status van de muis. Aangezien we meestal meer dan één knop op een muis hebben, zal je als argument steeds een nummer moeten opgeven. De linkerknop heeft index 0, de rechterknop index 1. Enkele voorbeelden:

```
Ms.bp(0); // true als de linkerknop ingedrukt werd in dit frame
Ms.br(1); // true als de rechterknop los gelaten werd in dit frame
Ms.b (0); // true zolang de linkerknop ingedrukt is
Ms.bd(0); // true wanneer in dit frame een dubbelklik plaatsvond
```

Het verschil tussen `Ms.bp()` en `Ms.b()` is subtiel:

- `Ms.bp()` Betekent dat de knop net werd ingedrukt. In de volgende update kan de muis misschien ook nog ingedrukt zijn, maar dan zal het resultaat van deze functie false zijn.
- `Ms.b()` Deze functie heeft true als resultaat zolang de knop niet los gelaten wordt.

Het volgende programma maakt dit verschil duidelijk:

```
Vec2 mouse1Pos;
Vec2 mouse2Pos;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;

   if(Ms.bp(0)) {
      mouse1Pos = Ms.pos();
   }

   if(Ms.b(1)) {
      mouse2Pos = Ms.pos();
   }

   return true;
}

void Draw()
{
   D.clear(BLACK);
   mouse1Pos.draw(RED  );
   mouse2Pos.draw(GREEN);
}
```

Het rode punt zal enkel een nieuwe positie krijgen op het moment dat je de linker muisknop indrukt. Het groene punt daarentegen blijft de muis volgen zolang de rechter muisknop ingedrukt blijft. Tijdens het ontwikkelen van een programma zal je steeds moeten kiezen welk van beide functies het meest geschikt is.

Time for Action

Pas het bovenstaande voorbeeld aan, zodat de tweede muispositie enkel zichtbaar is wanneer de rechter muisknop ingedrukt is.

2.1.3 Muiswiel

Tegenwoordig heeft een muis meestal ook een wieltje. daarvoor bestaat geen absolute positie, maar esenthel houdt wel bij hoeveel het muiswiel gedraaid werd tijdens de huidige update. Een afstand die binnen een frame werd afgelegd heet een 'delta'. Je kan zo spreken over de delta van de tijd, de delta van een beweging enzovoort. De functie die je nodig hebt om de delta van het muiswiel te weten is `Ms.wheel()`. Het resultaat is een float.

Het volgende voorbeeld laat een punt vertikaal bewegen als via het muiswieltje. In de update functie wordt de positie van het punt aangepast. We stellen nu geen nieuwe positie in zoals bij de vorige oefening. We passen slechts de positie op de y as aan. De functie `Ms.wheel()` geeft enkel de verplaatsing weer sinds de vorige update. Als je naar boven beweegt dan is dat een (zeer klein) positief getal. Beweeg je naar beneden, dan is dit getal negatief.

```
Vec2 mousePos;

void InitPre()
{
    EE_INIT();
}

bool Init()
{
    return true;
}

void Shut() {}

bool Update()
{
    if(Kb.bp(KB_ESC)) return false;
    mousePos.y += Ms.wheel();
    return true;
}

void Draw()
{
    D.clear(BLACK);
    mousePos.draw(RED);
}
```

Time for Action

Pas het bovenstaande voorbeeld aan zodat, wanneer de linker muisknop ingedrukt is, de horizontale positie wordt aangepast. Is de rechter muisknop ingedrukt, dan pas je de vertikale positie aan.

2.1.4 Cursor

De muiscursor is niets anders dan een afbeelding die op het scherm getoond wordt. Je kan aan het begin van je programma (bijvoorbeeld in de Init functie) die afbeelding wijzigen:

```
Ms.cursor(Images( --drop hier de afbeelding-- ));
```

Time for Action

Pas de cursor aan in de vorige oefening. Er staan enkele geschikte afbeeldingen in de map gfx ⇒ mouse. Toon de afbeelding 'BGNormal' wanneer het programma start. In de update functie zorg je er voor dat, enkel wanneer een muisknop ingedrukt is, 'BGMove' getoond wordt.

2.1.5 Overige functies

De class `Mouse` bevat nog veel meer functies. We bespreken ze niet allemaal in dit hoofdstuk. Als oefening zoek je zelf uit waar de volgende functies voor dienen:

- `Ms.hide()`
- `Ms.show()`
- `Ms.eat()`

2.2 Keyboard Interacties

Ook het toetsenbord wordt dikwijls gebruikt om interactie te sturen. Later, in het hoofdstuk over de GUI, zal je zien hoe we een toetsenbord gebruiken om tekst in te typen. In dit hoofdstuk controleren we enkel de status van de toetsen.

Elke toets heeft een naam. Via die naam kan je een bepaalde toets aanspreken. Hierbij moet je wel voor ogen houden dat die naam overeenkomt met de positie van die toets op een qwerty toetsenbord. Werk je op azerty en je wil de status van de 'Z' toets weten, dan zal je naar de status van de 'W' toets moeten vragen. Dit lijkt vreemd, maar bij de aansturing van een spel is vooral de positie van de toets belangrijk. Stel je voor dat de typische WASD aansturing de posities op een azerty toetsenbord zou volgen!

Verder is het controleren van een toets op je keyboard net zoals een muis klik. Je zal zien dat de functies bijna gelijk zijn.

2.2.1 Key Functions

Bekijk de volgende functies even:

```
Kb.bp(KB_N) // true als de N toets werd ingedrukt tijdens de huidige frame.
Kb.br(KB_E) // true als de E toets werd losgelaten tijdens de huidige
    frame.
Kb.b (KB_R) // true als de R toets momenteel ingedrukt is
Kb.bd(KB_D) // true bij een dubbelklik op toets D
```

We gebruiken nu het object `Kb` in plaats van `Ms`, maar de functies zijn precies zoals die voor een muisklik. Waar je bij het `Ms` object het nummer van de toets moest ingeven, gebruik je nu een code. Deze code begint steeds met KB_. Dan volgt meestal een letter of een cijfer. Andere mogelijkheden zijn:

Functions	Control	Modifiers	Arrows	Numpad
KB_F1	KB_ESC	KB_LCTRL	KB_LEFT	KB_NPDIV
KB_F2	KB_ENTER	KB_RCTRL	KB_RIGHT	KB_NPENTER
...	KB_SPACE	KB_LSHIFT	KB_UP	KB_NP1
KB_F12	KB_BACK	KB_RSHIFT	KB_DOWN	KB_NP2
	KB_TAB	...		...

Een volledig overzicht met alle toetsen vind je in de map Esenthel Engine ⇒ Input ⇒ Input Buttons.

Time for Action

Maak een programma met een punt op het scherm. Via de pijltjestoetsen kan je dit punt verplaatsen. Elke keer een pijltjestoets wordt ingedrukt, verplaats je het punt 0.1 units in de gewenste richting.
Het punt zelf teken je in het groen op je scherm, tenzij de spatiebalk ingedrukt is. Dan verschijnt het punt in het rood.

2.2.2 Graduele wijzigingen

De functie `Kb.bp()` gebruik je vooral voor plotse wijzigingen. Je wil bijvoorbeeld een window openen, een object toevoegen, een menu tonen of het programma verlaten. In het volgende voorbeeld zie je hoe je deze functie zou kunnen gebruiken om een help window te tonen:

```
if(Kb.bp(KB_F1)) HelpWindow.show();
```

Stel je voor dat je hier de functie `Kb.b()` zou gebruiken. In elke frame dat de toets F1 ingedrukt is, zou het help window opnieuw getoond worden. Op een snelle computer is dat ongeveer 60 keer per seconde.

Toch is de twee functie heel bruikbaar, maar dan voor waarden die geleidelijk moeten veranderen. Je blijft een waarde dan wijzigen zolang de toets ingedrukt is:

```
if(Kb.b(KB_RIGHT)) point.x += 0.01;
```

De x waarde van het punt zal tijdens elke frame iets groter worden. het punt verplaatst zich dus geleidelijk aan naar rechts.

Time for Action

Maak een programma aan de hand van het laatste voorbeeld. Zorg dat het punt, via de pijltjestoetsen, in vier richtingen kan bewegen.

2.2.3 Delta time

De oefening die je hierboven maakte heeft een groot probleem: elke frame wordt de positie aangepast. Stel nu dat je je programma op twee computers test: de eerste computer heeft een snelle grafische kaart en haalt 60FPS. De tweede computer is al wat ouder en haalt slechts 30FPS. De beweging zal op de eerste computer dubbel zo snel verlopen! Voor een kleine oefening is dat geen probleem, maar bij een echte game is dat niet wenselijk.

De oplossing is de **delta time**. De delta time is gelijk aan de tijd die verstreken is sinds de vorige frame. *(Denk even terug aan de mouse wheel delta: de afstand die het muiswieltje aflegde sinds de vorige frame.)* De delta time kan je opvragen via de volgende functie:

```
Time.d();
```

Wil je een object verplaatsen aan een snelheid van 1 unit per seconde? Dan kan je de volgende code gebruiken:

```
if(Kb.b(KB_RIGHT)) point.x += 1 * Time.d();
```

Time for Action

Plaats de vorige oefening aan, zodat je rekening houdt met de time delta.
Uitbreiding: gebruik in plaats van het getal 1 een float variabele die gelijk is aan 1. Via twee zelf te kiezen toetsen kan je deze waarde verhogen en verlagen.

HOOFDSTUK 3

Tekst

Ongetwijfeld wil je in een programma ook tekst op het scherm tonen. Voor letters, woorden en zelfs hele zinnen bestaat er de class `Str`. Om het duidelijk te houden spreek je dat uit als 'string'.

Je kan op de volgende manieren tekst in een `Str` plaatsen:

```
Str tekst("hello world"); // via de constructor
tekst  = "hello "; // via toekenning
tekst += "world" ; // via de plus assignment operator
```

De tekst tussen de quotes noemen we in het vakjargon een 'string literal': een letterlijke string. Als je niets moet aanpassen aan een tekst, dan kan je dikwijls rechtstreeks met string literals werken.

3.1 Tekst op het scherm tonen

Nu wil je een tekst dikwijls op het scherm tonen. In tegenstelling tot de class `Vec2`, heeft `Str` geen 'draw' functie. Tekst op het scherm plaatsen gaat daarom via het object `D`:

```
Str myString;
myString = "hello world";
D.text(Vec2(0, 0), myString); // plaatst een tekst in het midden van het
    scherm
D.text(Vec2(0, -0.1), "Een string literal"); // het kan dus ook zonder Str
```

Het tweede argument van de functie `text` is de tekst die je op het scherm wil zetten. Het eerste argument is de positie. Je weet ondertussen genoeg over coördinaten om te begrijpen waarom dit een `Vec2` is.

Time for Action

Maak om dit in te oefenen een programma dat de woorden 'links', 'rechts', 'boven' en 'onder' op een logische plaats op het scherm plaatst.

Time for Action

Maak een programma dat de muis verbergt en op de positie van de muis het woord 'mouse' toont.

3.2 Getallen en tekst combineren

Je kan getallen en tekst niet zomaar combineren. Voor je computer zijn 42 en "42" iets helemaal anders. Het eerste is een getal, het tweede een tekst die toevallig de characters 4 en 2 bevat. Dat zie je ook in de volgende code:

```
int i = 42;
Str myString;
mystring = i; // dit genereert een foutmelding
myString = 42; // dit is nogmaals een foutmelding
myString = "42"; // dit is ok!
D.text(Vec2(0, 0), i); // dit is weer fout: je kan geen integer gebruiken
    als het programma een string verwacht
```

Toch wil je vaak ook de waarde van een variabele op het scherm. Hoe pak je dat dan aan? Wel, je kan een getal wel toevoegen aan een `Str` object:

```
int i = 42;
Str myString;
myString += i; // dit is correct
myString += 42; // dit ook
myString += 0.4; // dit ook. De myString bevat nu de tekst "42420.4"
D.text(Vec2(0, 0), myString); // toont de tekst op het scherm
```

Maar we zijn nog niet helemaal klaar. Dikwijls wil je tekst en getallen combineren. Maar kijk eens naar de volgende code:

```
Str myString;
myString = "score: " + 42; // fout!
myString = "score: ";
myString += 42; // correct!
```

De eerste versie werkt niet omdat je 42 wil toevoegen aan een string literal. De compiler zal proberen om eerst 42 toe te voegen aan "score: ". Pas dan wordt de combinatie van die twee in 'myString' geplaatst. Bij het combineren van die twee gaat het dus fout, want "score: " is geen `Str` maar een string literal.

De tweede versie werkt wel, omdat je eerst de string literal in een `Str` object plaats, en pas dan het getal aan dat object toevoegt.

Omdat je een string en een getal vaak gecombineerd worden, bestaat er een shortcut: een `Str` object `S`. Dit is een lege string.

```
Str myString;
myString = S + "score: " + 42; // correct!
```

Hoe komt het dat dit wel werkt? S is een lege `Str`. De string literal wordt toegevoegd aan S. Daarna wordt 42 toegevoegd aan S. Tenslotte wordt S in de variabele 'myString' geplaatst.

Een dergelijk object kan je ook doorgeven aan `D.text()`. De volgende statements zijn dus helemaal in orde:

```
int myScore = 10;
D.text(Vec2(0, 0), S + "score: " + myScore);

// iets complexer
D.text(Vec2(0, 0), S + "score: " + myScore + " op tien");
```

Time for Action

Maak een programma met een int 'score'. Telkens als je op de spatiebalk drukt, dan verhoogt de score met één punt. Je toont de score ook op het scherm.

3.3 Een Vec2 als tekst weergeven

Om de werking van je programma te controleren is het soms handig om de x- en y-coördinaat van een Vec2 op het scherm te zetten. Aangezien dat getallen zijn, kan je dat op de volgende manier doen:

```
Vec2 pos = Ms.pos();
D.text(Vec2(0, 0), S + "x: " + pos.x + " y: " + pos.y);
```

Maar omdat elke programmeur zoiets regelmatig nodig heeft, beschikt de class `Vec2` ook over een functie die dat eenvoudiger maakt:

```
Vec2 pos = Ms.pos();
D.text(Vec2(0, 0), S + pos.asText());
// of ook rechtstreeks:
D.text(Vec2(0, 0), Ms.pos().asText());
```

Time for Action

Maak nog eens een programma met een punt dat je via de pijltjestoetsen kan aanpassen. Toon de positie van dat punt op het scherm.

3.4 Tekstopmaak

Waarschijnlijk wil je de standaardweergave van tekst wel aanpassen als je een eigen game maakt. Dat is op zich niet moeilijk. Je maakt daarvoor een nieuw font, door rechts te klikken in de filetree en 'New Font' te kiezen. Als je dat font opent kan je zowat alles aanpassen. Het meest belangrijke is daar de naam in het vak 'System Font'.

Vervolgens maak je een nieuwe textStyle. Daarmee kan je de kleur en aligment van je font wijzigen. *(In een textStyle zie je onderaan een dropdown menu 'Font'. Daar kan je aangeven welk van je font dat deze stijl moet gebruiken.)* Je kan ook meer dan één textStyle maken van hetzelfde font.

Als je nu je eigen textStyle wil gebruiken om tekst op het scherm te tonen, dan kan je een twee versie van `D.text()` gebruiken:

```
D.text(*TextStyles( -- drop style here -- ), Vec2(0, 0), "tekst");
```

Je sleept je textStyle tussen de haakjes van TextStyles(). Onthoud ook dat er een asterisk voor TextStyles staat. Later leer je waarom dat zo moet, maar het werkt in ieder geval niet als je die vergeet.

Time for Action

Experimenteer met de verschillende mogelijkheden om tekst vorm te geven. Zet minstens vier teksten op het scherm, en gebruik voor elke tekst een andere stijl.

HOOFDSTUK 4

Shapes

Nu je weet hoe je een punt op het scherm kan tonen, zijn meer complexe figuren ook niet moeilijk. (Indien niet, kijk dan nog eens naar hoofdstuk 1.) Je kan de meest voorkomende wiskundige vormen op het scherm tonen. De classes om dat te doen staan in de groene map Esenthel Engine ⇒ Math ⇒ Shapes.

Niet alle figuren kan je in een 2D programma gebruiken. Sommige classes, zoals `Ball` en `Tube`, zijn bedoeld voor 3D toepassingen. In 2D kan je `Circle`, `Edge` (lijn), `Quad`, `Rectangle` en `Triangle` gebruiken.

4.1 Circle

Om een cirkel te tekenen heb je een straal(r) en een positie(pos) nodig. Die straal is een `float`, de positie is een `Vec2`. Je kan die op meer dan 1 manier instellen:

```
// via de constructor, met r(float), pos.x(float), pos.y(float)
Circle c(0.1, 0, 0);

// via de constructor, met r(float), pos(Vec2)
Circle c(0.1, Vec2(0, 0));

// tijdens de duur van het programma
c.set(0.1, 0, 0);
c.set(0.1, Vec2(0, 0));

// zonder de set functie
c.r = 0.1;
c.pos = Vec2(0, 0);
```

4.1.1 Functies

Je kan ook het oppervlak of de omtrek van de cirkel opvragen via functies, of de cirkel op het scherm tonen:

```
// oppervlak en omtrek
float opp = c.area();
float omtrek = c.perimeter();

// een blauwe cirkel op het scherm tonen
c.draw(BLUE);

// enkel een blauwe omtrek tonen
c.draw(BLUE, false);
```

Er is iets opmerkelijks aan de hand met de **draw** functie! Je kan ze zowel met één als met twee argumenten gebruiken. Om te weten hoe dat komt, moet je naar de definitie van die functie kijken (je vindt die in Esenthel Engine ⇒ Math ⇒ Shapes ⇒ Circle):

```
void draw(C Color & color, Bool fill = true, Int resolution = -1) C;
```

Je ziet dat het eerste argument een **Color** moet zijn. Het tweede argument is een **bool** met de naam 'fill'. Je zou dus al kunnen vermoeden dat die dient om de cirkel gevuld te tekenen. Dit argument heeft ook een standaard waarde: true. Daarom is het niet nodig om de draw functie met beide argumenten te gebruiken als je akkoord gaat met de standaard waarde. Enkel wanneer je een cirkel niet gevuld wil tekenen, geef je false door aan de functie.

> **Note**
>
> Ook het derde argument: 'resolution' is optioneel. Experimenteer met verschillende waarden om te ontdekken wat je hier mee kan doen.

4.1.2 Rekenen

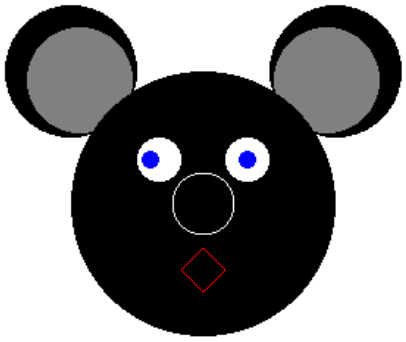
Je kan de operators **+=**, **-=**, **/=** en ***=** gebruiken om te 'rekenen' met cirkels. Hierbij heb je twee mogelijkheden: een **float** past de straal aan, een **Vec2** de positie.

```
Vec2 pos(0.1, 0);
Circle c(0.1, pos);

// verplaats de cirkel 0.1 naar rechts
c += pos;
// verplaats de cirkel 0.2 naar beneden
c -= Vec2(0, 0.2);
```

```
// verdubbel de straal van de cirkel
c *= 2;
```

4.1.3 Oefeningen

Maak de volgende afbeelding na door cirkels op je scherm te plaatsen. Als uitbreiding kan je ook eens naar de functie '**drawPie**' kijken en een meer geschikte mond voorzien.



Figuur 4.1: Let op de oogjes!

4.2 Edge2

Een **Edge2** is een lijnstuk. De '2' is net zoals bij **Vec2** belangrijk. Wanneer je die vergeet is je code wel geldig, maar dient die om een lijnstuk in 3D te tekenen. Om een **Edge2** te definiëren heb je twee punten nodig, het begin en het eind. Esenthel bied je twee mogelijkheden om die in geven:

```
Edge2 lijn1;
Edge2 lijn2;

// via vectoren
lijn1.set(Vec2(0, 0), Vec2(0.1, 0.1));

// dit kan ook met bestaande vectoren
Vec2 pos1(0.3, 0.6);
Vec2 pos2(0.1, 0.2);
lijn1.set(pos1, pos2);

// je kan ook alle x en y waarden afzonderlijk ingeven
lijn2.set(-0.4, 0.2, -0.7, 0.9;
```

Zoals je ziet heb je voor elk punt een x- en een y-coördinaat nodig. Je kan die, zoals in het eerste voorbeeld, ingeven via een `Vec2`. Of je kiest voor de tweede optie en geeft x1, y1, x2 en y2 in.

4.2.1 Functies

Ook een `Edge2` heeft een aantal handige functies. Je kan al verwachten dat er een functie `draw()` bestaat:

```
line1.draw(PURPLE    );
line2.draw(GREEN, 0.1); // stel ook de breedte in
line3.draw(GREEN, RED); // ga over van groen naar rood
```

Daarnaast kan je ook informatie over de lijn opvragen via bijvoorbeeld `center()`, `delta()`, `dir()` en `length()`. Je kan alle functies vinden in Esenthel Engine ⇒ Math ⇒ Shapes ⇒ Edge.

4.2.2 Oefeningen

Via de functies `Sin()` en `Cos()` kan je de sinus en cosinus van een getal opvragen. Dit is vooral handig omdat dat getal bijvoorbeeld de tijd kan zijn. Via de tijd krijg je een geleidelijke verandering in de sinus en cosinus waarde. Beiden bewegen constant tussen -1 en 1.

```
float x = Sin(Time.curTime());
float y = Cos(Time.curTime());
```

1. De bovenstaande code helpt je met berekening van sinus en cosinus van de tijd. Maak nu een programma waarin je een lijn toont van in het midden van je scherm tot de huidige waarde van sinus en cosinus.
2. Pas de dikte van de lijn aan.
3. Zorg er voor dat de lijn dubbel zo snel beweegt.
4. Zorg er voor dat de lijn maar half zo lang is.

Kijk eens in Esenthel Engine ⇒ Math ⇒ Shapes ⇒ Edge. Je kan daar de functie `lerp(float s)` vinden. Deze functie heeft een getal tussen 0 en 1 nodig en geeft een punt in de lijn terug.

1. Definieër globaal een `Edge2` en een `float`, gelijk aan 0;

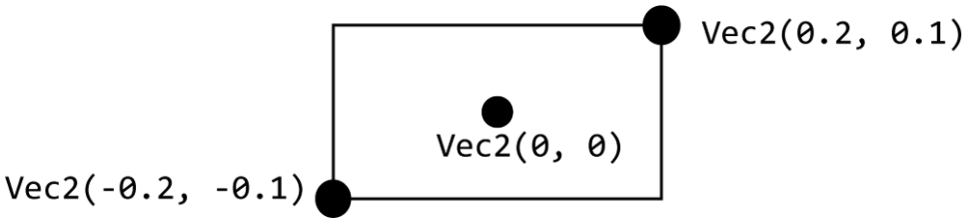
2. In de init functie laat je de lijn van -1, 0 tot 1, 0 lopen.
3. In de update functie verhoog je de float met `Time.d()`. Wanneer de float groter is dan 1, dan stel je hem gelijk aan 0.
4. In de draw functie teken je de lijn in het zwart, met breedte 0.05. Je maakt een `Vec2` die je de waarde geeft van de `lerp()` functie van je lijn, met als argument je float. Dit punt teken je rood, met eveneens een breedte 0.05.

(Uitbreiding) Teken ook eens een driehoek op het scherm.

4.3 Rect

De laatste vorm die we bespreken is de rechthoek: `Rect`. Waarschijnlijk is dit de vorm die je in je eigen programma's het meest zal gebruiken. Een rechthoek ligt immers ook aan de basis van veel andere elementen, zoals een button, een window en een afbeelding.

Na de laatste oefening weet je dat je, om een driehoek te tekenen, 3 posities nodig hebt. Logischerwijs zou je kunnen besluiten dat je voor een rechthoek 4 punten moet ingeven. Maar een rechthoek heeft een bijzondere eigenschap: het is een RECHThoek. Bijgevolg is het genoeg om de linkeronderhoek en de rechterbovenhoek in te geven. De rest kan daaruit afgeleid worden.



Figuur 4.2: De hoeken van een rechthoek

De code voor deze rechthoek ziet er zo uit:

```
Rect(Vec2(-0.2, -0.1), Vec2(0.2, 0.1)).draw(BLACK, false);
```

Zoals je ziet is er ook nu een draw functie, waarin het eerste argument de kleur is, en je daarna optioneel kan aangeven dat je de rechthoek niet wil vullen. Net zoals bij een edge heb je ook nu weer de mogelijkheid om de x- en y-coördinaat afzonderlijk in te geven:

```
Rect(-0.2, -0.1, 0.2, 0.1).draw(BLACK);
```

4.3.1 Een bewegende Rect

Omdat een rechthoek dikwijls aan de basis ligt van figuren die je op het scherm wil tonen, en die figuren in een spel nogal vaak bewegen, is het dikwijls handig om een eigen class te maken die intern een rechthoek gebruikt. In deze sectie maken een rechthoek die je kan bewegen via de pijltjestoetsen.

Rechthoeken hebben één nadeel wanneer je ze vlot wil bewegen: we hebben de positie van de hoeken nodig in plaats van het midden van de rechthoek. Dat betekent dat we telkens twee punten moeten aanpassen voor elke verplaatsing. Daarom is het dikwijls eenvoudiger om een `Vec2` te gebruiken om de positie te onthouden. We declareren ook een kleur, zodat een rechthoek zijn eigen kleur kan onthouden.

```
class movableRect {
  Vec2 pos;
  Color color;
}
```

Deze class kunnen we uitbreiden met een create functie, een update functie en een draw functie. Via de create functie geef je de kleur en de startpositie door. De startpositie heeft een standaard waarde, dus wanneer je later de create functie gebruikt zonder een positie, dan staat de rechthoek in het midden van het scherm.

```
void create(C Color & color, C Vec2 & pos = 0) {
  T.color = color;
  T.pos = pos;
}
```

Note

In deze functie staan nog enkele nieuwigheden. De ampersand geeft aan dat we color en pos als referentie doorgeven. De `C` betekent dat we de waarde niet zullen aanpassen. Je leert hierover meer in een van de volgende hoofdstukken. De letter `T` lost een praktisch probleem op. Zowel het functie-argument als de Color in de class zelf hebben als naam 'color'. De compiler kan daarom niet weten wanneer je welke variabele bedoelt. Je zou ze beiden een verschillende naam kunnen geven, maar dat maakt de code moeilijker leesbaar. De elegante oplossing bestaat er in `T.` voor de variabele van de class toe te voegen. De T staat voor 'this' en daarmee bedoelen we 'deze class'. De versie zonder `T.` is bijgevolg de variabele die we als functieargument doorgeven.

In de update functie kunnen we de positie wijzigen via de pijltjestoetsen. Deze code heb je reeds gezien in hoofdstuk 2.2.2.

```
void update() {
  if(Kb.b(KB_LEFT )) pos.x -= Time.d();
  if(Kb.b(KB_RIGHT)) pos.x += Time.d();
```

```
  if(Kb.b(KB_UP   )) pos.y += Time.d();
  if(Kb.b(KB_DOWN )) pos.y -= Time.d();
}
```

Tot slot is er een draw functie nodig. Het is pas op deze plaats dat we, tijdelijk, een rechthoek maken. In dit geval is dat het meest praktisch, maar dat hoeft niet altijd zo te zijn. Het zou bijvoorbeeld kunnen dat je in de update functie wil controleren of de rechthoek iets raakt. In zo'n geval zou je waarschijnlijk ook een `Rect` aan je class toevoegen.

```
void draw() {
  Rect(pos - Vec2(0.2, 0.1), pos + Vec2(0.2, 0.1)).draw(color);
}
```

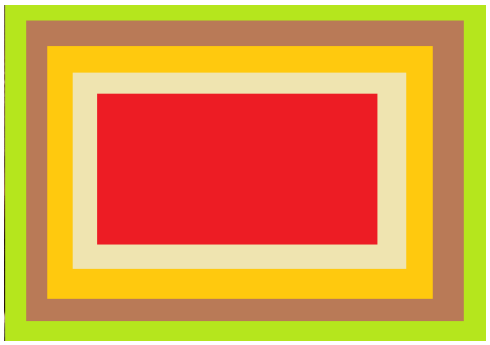
Zoals je ziet gebruiken we de positie 'pos' bij het maken van de rechthoek. Die is namelijk het middelpunt. Aangezien we de hoeken linksonder en rechtsboven nodig hebben bij het maken van een rechthoek, kunnen we eenvoudig een waarde respectievelijk aftrekken en optellen. We zouden deze class nog wat meer flexibel kunnen maken door een 'size' te onthouden.

Omdat dit de eerste maal is dat we een volledige class uitwerken in esenthel krijg je ze hieronder nog eens helemaal te zien. Let ook op de keywords public en private. Die zijn later belangrijk als je je code overzichtelijk wil houden.

```
class movableRect {
private:
  Vec2 pos, size;
  Color color;

public:
  void create(C Color & color, C Vec2 & pos = 0, C Vec2 & size = Vec2(0.1,
      0.1)) {
    T.color = color;
    T.pos   = pos  ;
    T.size  = size ;
  }

  void update() {
    if(Kb.b(KB_LEFT )) pos.x -= Time.d();
    if(Kb.b(KB_RIGHT)) pos.x += Time.d();
    if(Kb.b(KB_UP   )) pos.y += Time.d();
    if(Kb.b(KB_DOWN )) pos.y -= Time.d();
  }

  void draw() {
    Rect(pos - size, pos + size).draw(color);
  }
}
```

4.3.2 Oefeningen

1. Maak in een programma de volgende afbeelding na:



Figuur 4.3: Rechthoeken

2. Voeg een roze vierkant toe van de class `movableRect` die we hierboven voorstelden.
3. **(Uitbreiding)** Zorg dat het vierkant niet buiten het scherm kan bewegen.

4.4 Cuts

Een functie die je vaak gebruikt in verband met shapes is `Cuts`. Er bestaan heel veel varianten op deze functie, maar de betekenis is steeds dezelfde: raken twee objecten mekaar of niet? Zo kan je controleren of een punt en een cirkel mekaar raken, of een cirkel en een rechthoek, twee rechthoeken, een driehoek en een lijn, enzovoort.

Om te controleren o een punt een cirkel raakt, gebruik je bijvoorbeeld de volgende code:

```
Vec2 pos(0.1, 0.1);
Circle area(0.3, Vec2(0));

if(Cuts(pos, area)) area.draw(RED);
```

Natuurlijk is het in dit voorbeeld al duidelijk dat het punt de cirkel steeds zal raken. Maar ook de positie van je muiswijzer is een punt. Je zou dus het volgende kunnen schrijven:

```
Circle area(0.1, Vec2(0));

if(Cuts(Ms.pos(), area)) area.draw(RED);
else area.draw(BLACK);
```

De code hierboven ligt aan de basis van vele interactiemogelijkheden. Dikwijls gebeurt dit in combinatie met andere controles. Probeer zelf eens te bedenken wat de volgende code doet:

```
Rect button(Vec2(-0.2, -0.1), Vec2(0.2, 0.1));
bool hover = false;

// in een update functie:
if (Cuts(Ms.pos(), button)) {
  hover = true;
  if(Ms.bp(0)) exit();
} else hover = false;

// in een draw functie
if(hover) {
  button.draw(Color(0, 255, 0));
} else {
  button.draw(Color(0, 155, 0));
}
```

4.4.1 Oefeningen

1. Maak een programma met 3 cirkels (onder mekaar) waarvan je enkel de rand toont, tenzij de muiscursor zich in een van de cirkels bevindt. In dat geval teken je die cirkel gevuld.
2. Pas het vorige programma aan. Zorg er voor dat een cirkels langzaam naar rechts beweegt wanneer die zich onder de muiscursor bevindt.
3. Maak een integer 'score'. Wanneer een cirkel de rechterkant van het scherm raakt, verhoog je de score met één en plaats je de cirkel terug links.

(Uitbreiding) Maak een eigen class die zich als een button gedraagt. Je kan vertrekken van het voorbeeld hierboven en een hover effect implementeren. (Extra uitdaging: de kleur kan ook geleidelijk veranderen.) Een button zal ook een tekst moeten tonen, dus je voorziet een create functie die de positie en die tekst instelt. Een meer algemene functie om te bepalen wat de button doet wanneer je er op klikt is nog niet voor nu, maar je kan natuurlijk altijd proberen.

HOOFDSTUK 5

Afbeeldingen en Geluid

5.1 Afbeeldingen

Een moderne 2D game bevat bijna altijd afbeeldingen. Wat er ook op het scherm gebeurt, bijna altijd bestaat het uit het tonen en manipuleren van afbeeldingen. Aangezien een afbeelding steeds een rechthoek is, gebruik je een `Rect` om ze op het scherm te tonen.

```
Images(=== drop hier een afbeelding ===).draw(Rect(-0.1, -0.1, 0.1, 0.1));
```

Via `Images()` kan je elke afbeelding in je project gebruiken. Je dropt de gewenste afbeelding tussen de haakjes. Daarna gebruik je de `draw` functie met als argument een `Rect` om de afbeelding te tonen. Een bewegende afbeelding is op die manier heel eenvoudig. Je moet enkel de positie onthouden en je rechthoek daar van afleiden.

```
Vec2 ship(0, -0.8);

// tijdens update:
if(Kb.b(KB_LEFT )) ship.x -= Time.d();
if(Kb.b(KB_RIGHT)) ship.x += Time.d();

// tijdens draw:
Images(=== spaceship ===).draw(Rect(ship - 0.1,  ship + 0.1));
```

Note

Als je afbeeldingen zoals in dit voorbeeld wil gebruiken, dan vind je die makkelijk via Google images. Meestal zal je wel afbeeldingen willen met een transparante achtergrond. Dat kan enkel met een GIF of PNG formaat. Aangezien Esenthel geen GIF ondersteunt, moet je dus op zoek naar PNG afbeeldingen. De search tools op Google images laten je toe om specifiek naar transparante afbeeldingen te zoeken. Vind je niet dadelijk iets bruikbaar? Voeg dan eens de term 'icon' of 'sprites' toe aan je zoekopdracht.
Denk er wel aan dat je deze afbeeldingen enkel tijdens de ontwerpfaze kan gebruiken. Zou je een game publiceren met afbeeldingen die niet van jou zijn, dan krijg je problemen met auteursrechten.

Om alles iets realistischer te maken gebruik je meestal variaties op een afbeelding. In het volgende voorbeeld gebruiken we twee alternatieve versies van de afbeelding 'spaceship' tijdens de verplaatsing.

```
if(Kb.b(KB_LEFT))
{
  Images(=== spaceship ===      ).draw(Rect(ship - 0.1,  ship + 0.1));
} else if(Kb.b(KB_RIGHT))
{
  Images(=== spaceship_left === ).draw(Rect(ship - 0.1,  ship + 0.1));
} else
{
  Images(=== spaceship_right ===).draw(Rect(ship - 0.1,  ship + 0.1));
}
```

Een andere mogelijkheid is het variëren van de afbeelding in tijd. Je kan een animatie maken door telkens te wisselen tussen bepaalde afbeeldingen. In het volgende voorbeeld zie je een player class die drie varianten gebruikt voor elke richting.

```
class player
{
private:

   Vec2 pos;
   float timer = 0.4;

   DIR_ENUM dir = DIRE_DOWN;

public:


   void update()
   {
      // pas de richting aan
      if(Kb.bp(KB_UP   )) dir = DIRE_UP   ;
      if(Kb.bp(KB_DOWN )) dir = DIRE_DOWN ;
      if(Kb.bp(KB_LEFT )) dir = DIRE_LEFT ;
      if(Kb.bp(KB_RIGHT)) dir = DIRE_RIGHT;
```

```
20
        // positie update
        switch(dir)
        {
            case DIRE_UP   : pos.y += Time.d() * 0.5; break;
25          case DIRE_DOWN : pos.y -= Time.d() * 0.5; break;
            case DIRE_LEFT : pos.x -= Time.d() * 0.5; break;
            case DIRE_RIGHT: pos.x += Time.d() * 0.5; break;
        }

30      // timer voor animaties
        timer -= Time.d();
        if(timer < 0) timer = 0.4;
    }

35  void draw()
    {
        // een verwijzing naar een image
        ImagePtr current;

40      // evalueer de richting
        switch(dir)
        {
            case DIRE_UP:
            {
45              // kies een afbeelding afhankelijk van de tijd (wisselt tussen
                    1 - 2 - 3 - 2)
                if      (timer > 0.3) current = Images(=== back1 ===);
                else if (timer > 0.2) current = Images(=== back2 ===);
                else if (timer > 0.1) current = Images(=== back3 ===);
                else                  current = Images(=== back2 ===);
50              break;
            }

            case DIRE_DOWN:
            {
55              if      (timer > 0.3) current = Images(=== front1 ===);
                else if (timer > 0.2) current = Images(=== front2 ===);
                else if (timer > 0.1) current = Images(=== front3 ===);
                else                  current = Images(=== front2 ===);
                break;
60          }

            case DIRE_LEFT:
            {
                if      (timer > 0.3) current = Images(=== left1 ===);
65              else if (timer > 0.2) current = Images(=== left2 ===);
                else if (timer > 0.1) current = Images(=== left3 ===);
                else                  current = Images(=== left2 ===);
                break;
            }
70

            case DIRE_RIGHT:
```

```
            {
                if      (timer > 0.3) current = Images(=== right1 ===);
                else if (timer > 0.2) current = Images(=== right2 ===);
75              else if (timer > 0.1) current = Images(=== right3 ===);
                else                  current = Images(=== right2 ===);
                break;
            }
        }
80
        // toon de afbeelding waar current naar verwijst op het scherm
        current->draw(Rect(pos - 0.05, pos + 0.05));
    }
}
```

Time for Action

De bovenstaande code vind je ook terug in de template. Maak een object van de class player en gebruik die in je programma om te zien wat het resultaat is van deze code.

Note

Op regel 39 zie je een object 'current' van de class `ImagePtr`. Deze class is een 'pointer' naar een `Image`. Via de code die volgt stellen we die pointer gelijk aan de image die we op dat moment willen tonen. Daarna gebruiken we `current->draw()`. Het pijltje wil zeggen dat we van de image waar de pointer naar verwijst, de functie `draw()` uitvoeren. In een volgend hoofdstuk leer je meer over pointers.

5.1.1 Oefeningen

De 'Image' class in Esenthel bevat een overvloed aan functies. De meeste van die functies heb je waarschijnlijk niet nodig om een eenvoudig programma te maken, maar het is goed te onthouden dat, wat je ook met een image wil doen, er waarschijnlijk wel een functie voor bestaat. (Of een combinatie van functies.)

Experimenteer alvast eens met de volgende functies, zodat je weet wat ze doen.

- ❑ De functie `draw` heeft ook een variant waarin je kleuren kan instellen. Probeer die functie uit. (Hint: de tweede kleur is dikwijls `TRANSPARENT`)
- ❑ Teken een afbeelding met de functie `drawFit`. Waarin verschilt deze functie van `draw`?
- ❑ Teken een afbeelding met de functie `drawRotate`. Probeer de afbeelding ook te roteren via de pijltjestoetsen.

- ❑ Teken een afbeelding met de functie `drawFS`
- ❑ (Uitbreiding) Laad een afbeelding, voer er een blur op uit en exporteer de afbeelding als PNG.

5.2 Geluid

Om je game aantrekkelijk te maken voeg je best ook geluid toe. Daarin zijn er twee belangrijke groepen: muziek en effecten (FX). Muziek speel je meestal op de achtergrond, effecten koppel je aan bepaalde acties, zoals het indrukken van een toets of een muisklik. Ook speel je vaak een geluid af wanneer de speler wint, een punt verdient of verliest.

5.2.1 Muziek

Een soundtrack afspelen kan heel eenvoudig met de class `Sound`:

```
Sound soundtrack;

void InitPre()
{
    EE_INIT();
}

bool Init()
{
    soundtrack.create(=== drop hier je audio file ===);
    return true;
}

void Shut() {}

bool Update()
{
    if(Kb.bp(KB_ESC)) return false;

    if(Kb.bp(KB_SPACE))
    {
        if(soundtrack.playing())
        {
            soundtrack.pause();
        } else
        {
            soundtrack.play();
        }
    }
    return true;
}
```

Hierbij moet je wel enkele zaken goed onthouden:

- Voordat je een geluid kan gebruiken, moet je het eerst laden. Dat doe je met de functie `create()`. Die heeft minstens één argument nodig: een audiofile. Je kan die file gewoon van je resources naar de functie slepen. Deze functie gebruik je meestal in de `Init()` functie, want je wil het geluid niet tijdens elke update opnieuw laden.
- De functie `play()` start het geluid.
- De functie `pause()` pauseert het geluid. Wanner je nogmaals de `play()` functie gebruikt, dan speelt dit geluid verder van op de positie waar het stopte.
- Je kan in plaats van `pause()` ook `stop()` gebruiken. Wanneer je na `stop()` opnieuw `play()` aanroept, dan start het geluid terug van het begin.

Overigens heeft de `create()` functie nog enkele mogelijke argumenten. Enkel de audio file is noodzakelijk, maar hier zie je een voorbeeld met alle argumenten:

```
soundtrack.create(=== audio file ===, true, 0.8, VOLUME_MUSIC);
```

Wat betekenen deze argumenten?

1. Het eerste argument ken je al, dat is het audio bestand.
2. Het tweede argument staat voor 'loop'. De waarde kan true of false zijn. Daarmee geef je aan op het geluid in een loop gespeeld moet worden. (Dat wil zeggen dat het op het eind van de track automatisch opnieuw start.) De standaard waarde is false.
3. Het derde argument geeft het volume aan. De standaard waarde is 1.0, wat ook het maximum is. De minimum toegelaten waarde is 0.
4. Het laatste argument is een 'channel'. Esenthel heeft verschillende channels voor het spelen van audio. Als je dit argument niet gebruikt, dan hoort je geluid bij het channel 'VOLUME_FX'. Door hier een andere waarde te gebruiken, wijs je het geluid toe aan een ander channel. Dat heeft als voordeel dat je later het volume voor alle geluiden van een channel samen kan aanpassen. (Het is gebruikelijk dat je in een game de speler toelaat om het volume van de muziek, fx of voice afzonderlijk kan aanpassen.

5.2.2 exercise

Maak een programma waarin je een soundtrack laadt. Je voorziet ook een groene, een oranje en een rode cirkel op het scherm. Zorg er voor dat de track begint te spelen wanneer je in de groene cirkel klikt, pauseert wanneer je in de oranje cirkel klikt en stopt wanneer je in de rode cirkel klikt.

Uitbreiding: Zoek in de class sound naar een functie om de positie binnen de soundfile op het scherm te tonen. Pas op, dit is niet de functie om de positie van het geluid in een 3D wereld te tonen!

Uitbreiding: Dit is al wat moeilijker. Gebruik de functies `fadeInFromSilence()` en `fadeOut()` om de een fade van 3 seconden toe te passen in plaats van het geluid dadelijk te starten of te stoppen.

5.2.3 Playlists (Uitbreiding)

Om meer te doen met muziek kan je ook playlists gebruiken. Zo breng je meer variatie en kan je ook wisselen tussen playlists wanneer je bijvoorbeeld een gevecht start. Wie wil weten hoe dit werkt bestudeert de volgende code:

```
// defined play lists
Playlist Battle , // this is battle playlist used for playing when battles
         Explore, // exploring playlist
         Calm   ; // calm playlist

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   if(!Battle.songs())                          // create 'Battle' playlist
      if not yet created
   {
      Battle += (drop hier dramatische track); // add "battle0" track to
         'Battle' playlist
      Battle += (hier ook                   ); // add "battle1" track to
         'Battle' playlist
   }
   if(!Explore.songs())                         // create 'Explore'
      playlist if not yet created
   {
      Explore+= (hier een rustige track     ); // add "explore" track to
         'Explore' playlist
   }
   if(!Calm.songs())                            // create 'Calm' playlist
      if not yet created
   {
      Calm   += (Hier nog een rustige track ); // add "calm" track to
         'Calm' playlist
   }
   return true;
}
```

```
void Shut()
{
}

bool Update()
{
   if(Kb.bp(KB_ESC))return false;
   if(Kb.c('1'))Music.play(Battle );
   if(Kb.c('2'))Music.play(Explore);
   if(Kb.c('3'))Music.play(Calm   );
   if(Kb.c('4'))Music.play(null   );
   return true;
}

void Draw()
{
   D.clear(TURQ);

   if(Music.playlist()) // if any playlist playing
   {
      D.text(0, 0, S+"time " +Music.time()+" / "+Music.length()+" length");
   }else
   {
      D.text(0, 0, "No playlist playing");
   }
   D.text(0, -0.2, "Press 1-battle, 2-explore, 3-calm, 4-none");
}
```

5.2.4 FX

Korte geluiden kan je spelen met de functie `SoundPlay()`. Die speelt het geluid dat je als argument geeft onmiddelijk af. Aangezien je geen object van de class `Sound` maakt, heb je verder geen controle meer over het geluid. Je gebruikt dit dan ook vooral voor korte geluiden. (We noemen dat 'one-shots'.)

5.2.5 exercise

In het template project vind je enkele geluiden in de folder 'sound'. Speel telkens een 'blip' wanneer je op de down pijltjestoets drukt. Voor de pijltjestoetsen links en rechts gebruik je het geluid 'rotate'. Tot slot speel je 'down' wanneer je op de spatiebalk drukt.

Uitbreiding: Speel ook de soundtrack terug af, maar zorg dat je het volume kan aanpassen via het muiswieltje.

Deel II

Basics

HOOFDSTUK 6

Random

6.1 Gehele getallen

Een game blijft zelden boeiend als die compleet voorspelbaar is. Om die voorspelbaarheid tegen te gaan maken ontwikkelaars gebruik van willekeurige waarden via de `Random` class. Die class laat je toe om variatie in te bouwen. De functie `Random()` geeft je een willekeurig getal tussen 0 en 4.294.967.295. Je kan dat testen via het volgende voorbeeld:

```
uint number = 0;

bool Update() {
  if(Kb.bp(KB_SPACE)) number = Random();
  return true;
}

void Draw() {
  D.clear(BLACK);
  D.text(0, 0, S + number);
}
```

In praktijk zal je zelden zo'n grote getallen willen. Je kan de functie `Random()` daarom ook gebruiken met een of meerdere argumenten. Eén argument zorgt er voor dat je een getal krijgt in de range 0 tot het argument - 1. Dus `Random(5)` geeft je de waarde 0, 1, 2, 3 of 4 als resultaat. Let op: dat zijn 5 verschillende waarden, maar het getal 5 zal niet voorkomen!

Je kan ook twee argumenten gebruiken. `Random(-2, 4)` heeft mogelijk de volgende waarden: -2, -1, 0, 1, 2, 3 of 4. In deze versie zijn de waarden dus wel inclusief.

Time for Action

Probeer de bovenstaande code uit. Test ook de versies van de Random functie met één of twee argumenten.

Note

Als je een random kleur wil, dan kan je de RGB waarden van Color een willekeurige waarde tussen 0 en 255 geven. Dat kan zo:

```
Color myColor;
myColor.set(Random(255), Random(255), Random(255));
```

6.2 Random Float

De bovenstaande code geeft je een willekeurig geheel getal. Dikwijls zal je echter een floating point waarde willen. Daarvoor kan je een andere functie gebruiken: `RandomF()`. Deze functie geeft je een waarde tussen 0 en 1. Ook hier is het mogelijk om argumenten te gebruiken. `RandomF(3)` geeft je een waarde tussen 0 en 3. `RandomF(-1.3, 2.5)` levert een waarde tussen -1.3 en 2.5.

Dikwijls zal je deze functies gebruiken om een object op een willekeurige positie te tonen. Dat kan zo:

```
Circle c;

c.pos.x = RandomF(-D.w(), D.w());
c.pos.y = RandomF(-D.h(), D.h());
```

Zoals je ziet zetten we niet letterlijk getallen in de functie RandomF. Dat kan wel, maar in dit geval is dat niet aangewezen. De hoogte en de breedte van het scherm kunnen namelijk verschillen van toestel tot toestel. Aangezien het middelpunt van het scherm gelijk is aan 0, is de negatieve breedte dus de linkerkant (minimum waarde voor x). De negatieve hoogte is de onderkant (minimum waarde voor y).

Time for Action

1. Maak een programma dat een cirkel op het scherm toont. Telkens je op de spatiebalk drukt, geef je de cirkel een nieuwe positie.
2. Maak een programma dat een klein vierkant op het scherm toont. Elke seconde geef je het vierkant een nieuwe positie.
3. Dit is een uitbreiding op het vorige programma. Voor zie een int 'score' die gelijk is aan nul. Je toont de score ergens op het scherm. Wanneer je de linkermuisknop indrukt, controleer je of de muiswijzer binnen het vierkant zit. Als dat zo is, verhoog je de score met 1.
4. Uitbreiding: zorg dat het vierkant steeds sneller van positie wisselt.

HOOFDSTUK 7

Containers

Tot hier toe moest je altijd aan het begin van je programma alle objecten definiëren die je wil gebruiken. Voor een kleine oefening is dat geen probleem, maar in een groot project wordt dat al snel heel onoverzichtelijk. Ook is het onmogelijk om op deze manier extra objecten bij te maken terwijl je programma loopt.

Wanneer je verschillende objecten van hetzelfde type nodig hebt, dan kan je daar een container voor voorzien. Je declareert dan enkel de container, en voegt er later objecten aan toe. Een container die eenvoudig is in gebruik is `Memc`. De container vereist dat je bij de declaratie aangeeft wat voor objecten je er in wil zetten. Zo kan je in een `Memc<float>` floats opslaan. Wil je rechthoeken in een container? Dan gebruik je `Memc<Rect>`. Zo kan je eender welk soort objecten in een container steken. Hieronder zie je een container voor cirkels.

```
// Declareer een container voor cirkels
Memc<Circle> circles;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
  // genereer 10 cirkels
  for(int i = 0; i < 10; i++)
   {
      // De functie New maakt een nieuwe cirkel. We gebruiken dadelijk de
      // functie set van de class Circle om de straal en de positie in
      // te stellen.
      circles.New().set(0.1, RandomF(-D.w(), D.w()), RandomF(-D.h(),
         D.h()));
   }
```

```
    return true;
}

void Shut() {}

bool Update()
{
    if(Kb.bp(KB_ESC)) return false;
    return true;
}

void Draw()
{
    D.clear(BLACK);

    // Ga alle cirkels in de container af en
    // teken ze op het scherm.
    for(int i = 0; i < circles.elms(); i++)
    {
        circles[i].draw(RED);
    }
}
```

Time for Action

1. Wat gebeurt er als je de code om de cirkels te genereren per vergissing in Update in plaats van in Init zet?
2. Zet de functie terug in Init, maar voorzie in Update code om telkens wanneer je op de spatiebalk drukt één extra cirkel te maken.
3. Toon op het scherm een afbeelding in plaats van een cirkel. *(Te moeilijk? Begin dan met een rechthoek.)*

7.1 New()

De functie `New()` voegt een nieuw element toe aan het eind van de container. Omdat deze functie ook een referentie naar dat nieuwe object als resultaat geeft, kan je daar ook dadelijk een functie van dat object aan toevoegen. Dat is de reden waarom we de `set` functie van Circle kunnen gebruiken in het bovenstaande voorbeeld.

Maar wat als je nu twee functies van het nieuwe object wil gebruiken? Je zou het volgende kunnen proberen:

```
for(int i = 0; i < 10; i++)
{
  circles.New().set(0.1, RandomF(-D.w(), D.w()), RandomF(-D.h(), D.h()));
  circels.New().extend(-0.05);
}
```

...maar dat werkt natuurlijk niet. Je maakt nu bij elke iteratie van de for loop 2 cirkels. Bij de eerste voer je de functie **set** uit, bij de tweede de functie **extend**. De oplossing bestaat er in de referentie naar het nieuwe object even bij te houden. Je leert later meer over referenties, maar voorlopig is het voldoende als je weet dat je een ampersand (&) tussen de class en de naam plaatst:

```
for(int i = 0; i < 10; i++)
{
  Circle & c = circles.New();
  c.set(0.1, RandomF(-D.w(), D.w()), RandomF(-D.h(), D.h()));
  c.extend(-0.05);
}
```

Je maakt met andere woorden een tijdelijke verwijzing naar je nieuwe object. Hierboven gebruik ik de naam 'c'. Daarna kan ja via die referentie de functies uitvoeren die je wil.

7.2 Objecten gebruiken

Meestal zal je in de Update of Draw alle elementen van een container willen overlopen. Bijvoorbeeld om ze op het scherm te tonen. Daarvoor bestaat er de functie **elms()** die je vertelt hoeveel elementen er op dit moment in de container zitten. Met een eenvoudige for-loop kan je die dan allemaal af gaan. Zoals je ziet, geven we tussen vierkante haakjes aan welk element we willen gebruiken, net zoals je dat bij een primitieve array zou doen.

```
for(int i = 0; i < circles.elms(); i++)
{
  circles[i].draw(RED);
}
```

Omdat je zo dikwijls alle elementen van een container moet overlopen, bestaat er ook een 'shortcut'. De macro **REPA**. Je zou de vorige code dus ook zo kunnen schrijven:

```
REPA(circles)
{
  circles[i].draw(RED);
}
```

Onthoud dit als 'repeat all'. Je kiest natuurlijk zelf of je de for-loop dan wel deze shortcut gebruikt.

Je kan meer doen dan objecten op het scherm tonen. Bedenk zelf eens wat de volgende code, geplaatst in Update, doet.

```
REPA(circles)
{
  circles[i].pos.y += Time.d();
  if(circles[i].pos.y > D.h()) {
    circles[i].pos.y -= (2*D.h() + RandomF(1));
  }
}
```

Time for Action

1. Test de bovenstaande code in een progamma.
2. Voeg een functie toe om een cirkel toe te voegen wanneer je op de spatiebalk drukt.
3. In plaats van een vaste straal gebruik je een random waarde tussen 0.01 en 0.1.
4. Teken enkel de omtrek van de cirkels, in het wit. Maak de achtergrond blauw.
5. Roep luid door de klas waar dit je aan doet denken.

7.3 Objecten genereren

Objecten genereren tijdens of aan het begin van een programma is iets wat je best goed beheerst. Daarom zie je hieronder enkele mogelijkheden om je op weg te helpen. Denk er wel aan dat dit slechts voorbeelden zijn. De mogelijkheden waarop je containers gebruikt zijn zowat onuitputtelijk.

7.3.1 Tijdens Init

Tien cirkels op willekeurige posities:

```
for(int i = 0; i < 10; i++)
{
  Circle & c = circles.New();
  c.set(0.1, RandomF(-D.w(), D.w()), RandomF(-D.h(), D.h()));
}
```

Cirkels van de linkerkant tot de rechterkant van het scherm:

```
for(float i = -D.w(); i < D.w(); i += 0.2) {
  circles.New().set(0.1, i, 0);
}
```

Vierkanten over het hele scherm:

```
for(float i = -D.w(); i < D.w(); i += 0.2)
{
  for(float j = -D.h();  j < D.h();  j += 0.2)
  {
     rects.New().set(i - 0.05, j - 0.05, i + 0.05, j + 0.05);
  }
}
```

7.3.2 Tijdens Update

Via interactie:

```
if(Kb.bp(KB_SPACE)) {
  circles.New().set(RandomF(0.05, 0.2), RandomF(-D.w(), D.w()),
     RandomF(-D.h(), D.h()));
}
```

Interactie met de positie van de muis:

```
if(Ms.bp(0)) {
  circles.New().set(0.05, Ms.pos());
}
```

Met een timer:

```
Flt timer = 3; // deze regel zet je boven aan je programma, de rest hoort
    in update

if(timer > 0) timer -= Time.d();
else {
  timer = 3;
  circles.New().set(RandomF(0.05, 0.2), RandomF(-D.w(), D.w()),
     RandomF(-D.h(), D.h()));
}
```

Time for Action

Probeer de bovenstaande mogelijkheden uit, en toon ook steeds de gegenereerde objecten op het scherm.

7.4 Objecten verwijderen

Je kan uiteraard ook objecten verwijderen uit een container. Dit is op zich niet moeilijk:

```
Memc<Vec2> points;

// ... voeg ergens punten toe

points.remove(0); // verwijder het eerste punt
```

Je kan dus met de functie **remove** en als argument de index van het object dat je wil verwijderen, een object in de container wissen. Dikwijls zal je dat echter in een loop willen doen. In dat geval moet je opletten dat je het object niet meer gebruikt nadat je het verwijdert. Dat gebeurt sneller dan je denkt:

```
for(int i = 0; i < points.elms(); i++) {
  if(points[i].y < -D.h()) points.remove(i);
  points[i].y -= Time.d();
}
```

In de bovenstaande code worden alle punten naar beneden verplaatst. Wanneer een punt onder aan het scherm komt, dan wordt het uit de container gehaald. Het probleem stelt zich wanneer je het laatste punt verwijdert. Je probeert daarna het laatste punt te verplaatsen, maar omdat de container nu een element minder bevat is de index van het laatste punt ongeldig geworden. Je kan dat natuurlijk oplossen door **remove** altijd op het eind te plaatsen:

```
for(int i = 0; i < points.elms(); i++) {
  points[i].y -= Time.d();
  if(points[i].y < -D.h()) points.remove(i);
}
```

Moeilijker wordt het wanneer je twee containers combineert. In het volgende voorbeeld kijken we of een punt een cirkel raakt. Hiervoor moeten we alle punten en alle cirkels met mekaar vergelijken.

```
for(int i = 0; i < points.elms(); i++) {
  for(int j = 0; j < circles.elms(); j++) {
    if(Cuts(points[i], circles[j])) {
      points.remove(i);
```

```
        circles.remove(j);
        // op dit moment is er een punt minder, maar zullen de volgende
            cirkels nog
        // steeds met dit punt vergeleken worden. Om dat te voorkomen
            stoppen we hier
        // de binnenste loop met een break:
        break;
      }
    }
}
```

Tot slot kan je een container ook nog in één keer helemaal leegmaken:

```
points.clear();
```

7.5 Een spelletje

1. Maak een driehoek onder aan je scherm die je met de pijltjestoetsen heen en weer kan bewegen.
2. Voorzie een container voor de class `Vec2`. Telkens als je op de spatiebalk drukt voeg je een element toe op de positie van de driehoek.
3. In de update functie verhoog je de y waarde van elk element in de container. (Gebruik `Time.d()`!) Wanneer een element de bovenkant van het scherm bereikt, verwijder je het uit de container.
4. In de `Draw()` functie teken je alle elementen op het scherm.
5. Maak een tweede container voor cirkels. Elke seconde voeg je een cirkel toe aan de bovenkant van het scherm.
6. In de update functie ga je alle cirkels naar beneden bewegen.
7. Toon de cirkels in de `Draw()` functie.
8. Wanneer een cirkel en een punt mekaar raken, dan verwijder je ze beiden.
9. Wanneer een cirkel de driehoek raakt, dan toon je 'Game Over' op het scherm.

Je kan dit concept nog verder afwerken. Zo kan je voorkomen dat er nog nieuwe cirkels bijkomen na een game over, of dat je nog kan bewegen en schieten. Cirkels zouden ook steeds sneller kunnen verschijnen of sneller bewegen naarmate je langer speelt. Je kan ook een score op het scherm tonen of een aantal levens voorzien.

HOOFDSTUK 8

Classes: the basics

Gegevens die samen horen, hoor je te groeperen. Zo heeft `Vec2` (eigenlijk een class!) twee float variabelen x en y om een positie te onthouden. Zonder het bestaan van `Vec2` zou je voor elke positie twee afzonderlijke float variabelen moeten maken. Nu maak je er slechts 1: een `Vec2`.

De ontwerper van de Esenthel engine voorzag dat een class voor een positie, met een x en een y waarde, heel veel gebruikt zou worden. Dus maakte hij die alvast zelf. De meest eenvoudige versie zou er zo kunnen uitzien:

```
class Vec2 {
  float x;
  float y;
}
```

Indien deze code bestaat, kan je die overal in je programma gebruiken:

```
Vec2 pos;
Vec2 pos2 = pos; // wijst de waarden van pos toe aan pos2.
pos.x = 3;       // past de x waarde van pos aan.
pos.y = pos2.x;  // geeft de x waarde van pos2 aan de y waarde van pos
```

Hetzelfde kan je bereiken met je eigen classes. Je hoort classes te maken voor zo ongeveer elk aspect van je code.

Note

Wanneer twee of meer objecten of variabelen samen horen, dan zet je ze samen in één class.

Merk op dat er een verschil is tussen een class en een object. In het voorbeeld hierboven is `Vec2` een class, maar pos en pos2 zijn objecten van de class Vec2. Je kan `Vec2` dus niet gebruiken als een object, maar je kan er wel objecten mee maken:

```
Vec2.x = 3; // dit is fout!
Vec2 pos;
pos.x = 3; // dit is correct.
```

8.1 Een eigen class maken

Stel je voor dat je een bewegende cirkel wil in je programma. Om de cirkel tegen een vaste snelheid laten te bewegen heb je een variabele speed nodig. Die hoort duidelijk bij je cirkel, maar een cirkel heeft zelf geen variabele speed omdat cirkel niet altijd bedoeld zijn om te laten bewegen. Je maakt dus bijvoorbeeld de volgende class:

```
class movingCircle {
  Circle c;
  Vec2 speed;
}

movingCirle mc;

// in de functie init
mc.c.set(0.1, Vec2(-0.4, -0.3));
mc.speed = Vec2(0.3, 0.5);

// in de functie update
mc.c.pos += mc.speed * Time.d();
```

Eenmaal je de class `movingCirle` hebt, kan je zoveel cirkels maken als je wil, die allemaal hun eigen snelheid onthouden. (Verder in de cursus zie je hoe dit nog eenvoudiger kan.)

Time for Action

1. Maak een container voor de class movingCircle. Telkens je op de spatiebalk drukt, voeg je hier een object aan toe, op een willekeurige positie op het scherm. Elke cirkel geef je ook een willekeurige snelheid.
2. Zorg dat alle cirkels bewegen. Wanneer een cirkel buiten het scherm komt, zet je hem terug in het midden.
3. Breidt de class `movingCirle` uit zodat die ook een kleur kan onthouden. Elk object geef je een willekeurige kleur, die je gebruikt wanneer je het op het scherm zet.

Note

In het volgende hoofdstuk werk je verder aan deze oefening. Sla ze dus op!

HOOFDSTUK 9

Functies

Functies zijn instructies die je kan toevoegen aan een eigen class. *(In principe kan je ook functies buiten een class gebruiken, maar in regel probeer je dat best te vermijden.)* Met functies kan je duidelijke opdrachten maken die betrekking hebben op je class. Het is eigenlijk de bedoeling dat je de variabelen in je class nooit rechtstreeks gebruikt buiten de class. Als je ze nodig hebt, of wanneer je ze wil aanpassen, dan schrijf je daar een functie voor.

9.1 Functies zonder argumenten of resultaat

De meest eenvoudige functie doet steeds hetzelfde. Als we even terugdenken aan de class movingCircle, dan zou die misschien een functie kunnen gebruiken die de cirkel terug in het midden van je scherm zet:

```
class movingCircle {
  Circle c;
  Vec2 speed;

  void reset() {
    c.pos = Vec2(0);
  }
}

// ergens in je programma
movingCircle mc;
mc.reset();
```

De functie reset in het vorige voorbeeld zet de cirkel terug in het midden. De functie bestaat uit de volgende delen:

void Een aanduiding dat de functie geen resultaat heeft. (Daarover zodadelijk meer.)

reset De naam van de functie. Deze mag je zelf kiezen.

() Tussen deze haakjes kunnen argumenten staan. Deze eenvoudige functie heeft geen argumenten, dus hier staat niets.

Time for Action

Voeg aan de class die je voor de vorige oefening maakte een functie `reset()` toe, die de cirkel in het van het scherm plaatst. Vervang in de Update functie van je programma de code om een cirkel in het midden te plaatsen door deze functie.
Voeg in dezelfde class ook een functie `draw()` toe. Deze functie kan je cirkel dadelijk in de juiste kleur tonen. In je programma vervang je weer de code om de cirkel te tonen door deze functie.

9.2 Functies met een argument

Je kan een functie ook een argument geven. Dit is handig omdat je dat argument kan gebruiken om een waarde door te geven aan een functie. De functie reset in het vorige voorbeeld heeft geen waarde nodig. We weten dat we met reset de cirkel in het midden zetten. Wil je hem op een andere plaats zetten, dan zou je wel moeten zeggen welke plaats dat dan is. Dat zou zo kunnen:

```
class movingCircle {
  Circle c;
  Vec2 speed;

  void setPos(Vec2 pos) {
    c.pos = pos;
  }
}

// ergens in je programma
movingCircle mc;
Vec2 p(0.1, 0.4);
mc.setPos(p);
```

Nu geef je tussen de haakjes mee dat de functie gebruikt moet worden met een `Vec2` als argument. De `Vec2` die je meegeeft bepaalt dan de nieuwe positie van de cirkel. Je ziet ook dat de `Vec2` 'p' die we in het programma gebruiken een andere naam heeft dan `Vec2` pos die we in de functie gebruiken. Je geeft enkel de waarde van de functie door en niet de naam.

Time for Action

Voeg een functie `setPos` zoals hierboven toe aan je oefening. In het programma zorg je dat, wanneer je op F1 drukt, alle cirkels een nieuwe random positie krijgen.

9.3 Functies met meer argumenten

Het is niet nodig om je tot een enkel argument te beperken. Je kan er ook meer gebruiken, gescheiden door komma's:

```
class movingCircle {
  Circle c    ;
  Vec2   speed;

  void init(float r, Vec2 pos, Vec2 speed) {
    c.r     = r    ;
    c.pos   = pos  ;
    T.speed = speed;
  }
}

// ergens in je programma
movingCircle mc;
Vec2 p(0.1, 0.4);
mc.init(0.1, p, Vec2(0.3, -0.5));
```

Ook hier enkele nieuwigheden:

T Met `T` duiden we aan dat we het over de variabele van de class hebben. Immers, zowel de class als de functie hebben een variabele speed, en we moeten duidelijk maken welke we bedoelen. Je zou ook beide variabelen een andere naam kunnen geven, maar dat is minder duidelijk.

argumenten doorgeven Bij het gebruik van de init functie gebruiken we een vooraf gedefinieerde `Vec2` om de positie door te geven. Maar voor de snelheid maken we een `Vec2` op het moment zelf. Beide opties zijn ok.

Time for Action

Voeg ook een functie init toe aan je class.

9.4 Functies met een resultaat

We stelden in het begin van dit hoofdstuk dat je variabelen van een class eigenlijk niet rechtstreeks zou mogen gebruiken buiten die class. We hebben ondertussen gezien hoe je zo'n waarde aanpast, maar hoe kom je buiten de class te weten wat de huidige waarde van een variabele is? Wel, je maakt een functie met een resultaat:

```
class movingCircle {
  Circle c;
  Vec2 speed;

  void init(float r, Vec2 pos, Vec2 speed) {
    c.r         = r    ;
    c.pos       = pos  ;
    this->speed = speed;
  }

  Vec2 getPos() {
    return c.pos;
  }
}

// ergens in je programma
movingCircle mc;
mc.init(0.1, p, Vec2(0.3, -0.5));
Vec2 pos = mc.getPos();
```

De `Vec2` zal na de laatste regel de waarde 0.1 hebben. Een functie met resultaat maak je op de volgende manier:

- In plaats van void zet je voor de functienaam het soort resultaat dat de functie zal geven. De positie van een cirkel is een `Vec2`, dus in dit geval staat daar `Vec2`.
- In je functie geef je de instructie 'return' gevolgd door de waarde je je wil als resultaat. Die waarde zal dan doorgegeven worden op de plaats waar je de functie gebruikt.

Time for Action

Voeg deze functie toe aan je class. Vervang in je programma de code om de functie van je cirkel op te vragen door de nieuwe functie.

9.5 Functies met argumenten en een resultaat

Ook de combinatie van argumenten en een resultaat is mogelijk. Let wel op: alhoewel een functie meer dan een argument kan hebben, kan er slechts één resultaat zijn.

In het volgende voorbeeld maken we een functie 'move'. Deze functie verplaatst de cirkel door de positie in het argument op te tellen bij de huidige positie. Maar ze doet dat enkel wanneer de nieuwe positie binnen het scherm past. Met een boolean resultaat (true of false) laat de functie weten of dat gelukt is.

```
class movingCircle {
  Circle c    ;
  Vec2   speed;

  void init(float r, Vec2 pos, Vec2 speed) {
    c.r         = r    ;
    c.pos       = pos  ;
    this->speed = speed;
  }

  float getRadius() {
    return c.r;
  }

  bool move(Vec2 pos) {
    if(Cuts(c.pos + pos, D.viewRect())) {
      c.pos += pos;
      return true;
    } else {
      return false;
    }
  }
}

// ergens in je programma
movingCircle mc;
Vec2 p(0, 0);
mc.init(0.1, p, Vec2(0.3, -0.5));
if( !mc.move(mc.speed * Time.d()) ) {
  // doe iets wanneer dit niet gelukt is
}
```

Time for Action

Pas nogmaals je class aan en voeg de functie move toe. In je programma kan je nu deze functie gebruiken om de cirkel te verplaatsen. Omdat de functie zelf aangeeft of dat al dan niet gelukt is, kan je nu ook de controle op de grenzen van het scherm weglaten. Wanneer de functie false als resultaat heeft, zet jet de cirkel terug in het midden.
Je kan nu de functie 'move' nog verder vereenvoudigen. We weten immers dat we altijd op dezelfde manier willen bewegen. Je kan met andere woorden een functie move ook zonder argument maken en de berekening verplaatsen naar de functie.

9.6 De oplossing

Hieronder zie je de volledige oefening zoals je die tot hier toe moest uitwerken. Vergelijk deze code met je eigen uitwerken en vraag uitleg wanneer iets je niet duidelijk is.

De class movingCircle:

```
class movingCircle
{
   Circle c    ;
   Vec2   speed;
   Color  color;

   void create(float radius, Vec2 pos, Vec2 speed, Color color)
   {
      c.r     = radius;
      c.pos   = pos   ;
      T.speed = speed ;
      T.color = color ;
   }

   bool move()
   {
      Vec2 newPos = c.pos + speed * Time.d();
      if(Cuts(newPos, D.viewRect()))
      {
         c.pos = newPos;
         return true;
      } else return false;
   }

   Vec2 getPos()
   {
      return c.pos;
   }

   void setPos(Vec2 pos)
```

```
    {
        c.pos = pos;
    }

    void reset()
    {
        c.pos = 0;
    }

    void draw()
    {
        c.draw(color);
    }
}
```

Het programma:

```
Memc<movingCircle> circles;

void InitPre()
{
    EE_INIT();
}

bool Init()
{
    return true;
}

void Shut() {}

bool Update()
{
    if(Kb.bp(KB_ESC)) return false;

    if(Kb.bp(KB_SPACE))
    {
        circles.New().create(RandomF(0.05, 0.1),
                             Vec2(0),
                             Vec2(RandomF(-1, 1), RandomF(-1, 1)),
                             Color(Random(255), Random(255), Random(255))
                             );
    }

    if(Kb.bp(KB_F1))
    {
        REPA(circles)
        {
            circles[i].setPos(Vec2(RandomF(-D.w(), D.w()), RandomF(-D.h(),
                D.h())));
        }
    }
```

```
35
    REPA(circles)
    {
        if(!circles[i].move()) circles[i].reset();
    }
40
    return true;
}

void Draw()
45 {
    D.clear(BLACK);

    REPA(circles)
    {
50      circles[i].draw();
    }
}
```

HOOFDSTUK 10

Classes in de praktijk

Je weet nu hoe je in class maakt. Maar hoe ga je er in praktijk mee om? In dit hoofdstuk zie je veel gebruikte classfuncties en bekijken we wat je best samen in een class plaatst.

10.1 Wat hoort samen?

De bedoeling van een class is dat je variabelen en functies die samen horen, groepeert. Neem het voorbeeld van een score in een spel. Volgende items horen dan duidelijk samen:

- ❑ een variabele om de score te onthouden
- ❑ een variabele om de highScore te onthouden
- ❑ een functie om de score aan te passen
- ❑ een functie om de score te resetten
- ❑ een functie om te zien of de huidige score groter is dan de highScore
- ❑ een functie om de score op te vragen
- ❑ ...

Afhankelijk van de complexiteit van je spel kan je hier nog veel aan toevoegen. Maar wat er bijvoorbeeld niet bij hoort is een functie om de score op het scherm te tonen. We maken per definitie steeds een onderscheid tussen classes die intern berekeningen doen, en classes die iets op het scherm tonen. Die twee mengen brengt je dikwijls in de problemen.

Wanneer een class duidelijk bedoeld is voor iets dat op het scherm verschijnt, zoals een avatar of een voorwerp, dan hoort die dat natuurlijk wel te doen.

Zo zou het best kunnen dat je de score tijdens het spel op een andere manier toont dan tijdens een game-over. Daarom werk je met afzonderlijke classes om iets op het scherm te tonen. Die classes gebruiken dan wel steeds de functie om de score op te vragen die in het score object zit. Bijvoorbeeld zo:

```
class score {
  int points = 0;

  void reset() {
    points = 0;
  }

  void getPoints() {
    return points;
  }

  void addPoint() {
    points++;
  }
}

score Score; // object

// de volgende class wordt gebruikt om informatie weer te geven
// tijdens het spel
class overlay {
  void draw() {
    D.text(Vec2(0, 0.9), S + "Score: " + Score.getPoints());
  }
}

// de volgende class geeft informatie weer wanneer er geen spel actief is
class gameOver {
  void draw() {
    D.text(Vec2(0,0), S + "Score: " + Score.getPoints());
  }
}
```

10.2 Set en Get Functies

Om variabelen in een class aan te passen of op te vragen, gebruiken we set en get functies. Daarbij gelden de volgende regels:

- De naam van deze functies is steeds de naam van de variable, voorafgegaan door set of get.

- ❏ De functie past de variabele aan met dezelfde naam dan de functienaam.
- ❏ De set functie is een void functie met als argument de nieuwe waarde voor de variabele.
- ❏ De get functie heeft als resultaat het type van de variabele die ze hoort terug te geven.

In dit voorbeeld hebben we een class score met set en get functies voor de variabele points, van het type int.

```
class score {
  int points;

  int getPoints() {
    return points;
  }

  void setPoints(int points) {
    this->points = points;
  }
}
```

10.3 public en private

In het vorige voorbeeld is het nog steeds mogelijk om de variabele points rechtstreeks aan te passen, zonder dat je de get functie gebruikt. Dat is niet zo veilig. Stel je voor dat je ook een counter hebt die bijhoudt hoeveel keer de punten aangepast zijn. Je kan die verhogen telkens wanneer je de functie setPoints gebruikt. Maar wanneer iemand de punten rechtstreeks aanpast, dan klopt je counter niet meer. Aangezien iedereen zich vroeg of laat wel eens vergist, is het beter dat we dit onmogelijk maken. Daarom zullen we meestal de variabelen in een class 'private' maken. Ze kunnen dan enkel binnen deze class gebruikt worden. De functies maken we 'public' zodat die wel buiten de class werken.

```
class score {
private:
  int points  = 0;
  int counter = 0;

public:
  int getPoints() {
    return points;
  }

  void setPoints(int points) {
    this->points = points;
    counter++;
  }
}
```

```
score Score;

// ergens in je programma
Score.points = 3; // dit werkt niet meer
Score.setPoints(3); // dit werkt wel
```

10.4 Globale objecten

Er zijn classes waar je tijdens je programma verschillende objecten van nodig hebt, denk bijvoorbeeld aan een class Vec2. Andere classes zijn slechts bedoeld om één object van te maken, zoals de class score in het voorbeeld. In dit laatste geval is het de gewoonte om het object te maken in hetzelfde bestand als je class. Zo zie je hierboven onder de class score de definitie van het object Score.

10.5 Manager classes

Wanneer je een class maakt met bewegende cirkels, bijvoorbeeld omdat je veel bewegende cirkels in je programma nodig hebt, dan is het dikwijls handig om daarnaast ook een 'manager' class te maken. Een dergelijke class is dan verantwoordelijk voor het maken van nieuwe cirkels, voor het updaten van alle cirkels of voor het tekenen van deze cirkels op het scherm.

Die zou er zo kunnen uitzien:

```
class myCircle {
private:
  Circle c;

public:
  void update() {
    // update code komt hier
  }

  void draw() {
    // draw code komt hier
  }
}

class myCircleManager {
private:
  Memc<myCircle> list;

public:
  void createCircle() {
```

```
    myCircle & temp = list.New();
    // doe nog iets met de nieuwe cirkel
  }

  void update() {
    for(int i = 0; i < list.elms(); i++) {
      list[i].update();
    }
  }

  void draw() {
    for(int i = 0; i < list.elms(); i++) {
        list[i].draw();
      }
    }
  }
}

myCircleManager MyCircleManager;

// ergens in het bestand main (dit is een voorbeeld,
// geen volledig uitgewerkt programma!)
void init() {
  MyCircleManager.createCircle();
}

void update() {
  MyCircleManager.update();
}

void draw() {
  MyCircleManager.draw();
}
```

Time for Action

1. Maak een class 'movingCircle'. Die bevat een cirkel, een Vec2 'direction' en een float 'speed'.
2. Voorzie een functie 'create' die de cirkel een straal en een beginpositie geeft. Voor de variabele 'direction' geef je een random float tussen -1 en 1 aan zowel de x als de y waarde. Speed krijgt een waarde tussen 0.5 en 2.
3. Voorzie een functie 'update' die direction, vermenigvuldigd met `Time.d()` en 'speed', optelt bij de positie van de cirkel. Daarna voer je een controle uit op de nieuwe positie. Wanneer de absolute waarde van de y-coordinaat van de cirkel groter is dan de hoogte van het scherm, dan keer je het teken van de y-coordinaat om. Voor de x-coordinaat doe je hetzelfde, maar dan met de breedte van het scherm.
4. Maak een functie 'draw' die de cirkel op het scherm toont.
5. In je programma maak je een object van de class 'movingCircle'. Je zorgt ook dat de 'create', 'update' en 'draw' functies op de juiste plaats staan om te controleren of je cirkel werkt zoals het hoort.
6. Maak een nieuwe class 'circleManager' met een memory container voor 'movingCircle'.
7. Voorzie in deze class een functie 'add' die een cirkel toevoegt aan de container.
8. Maak een functie 'update' die de functie 'update' van elke cirkel in de container uitvoert.
9. Maak een functie 'draw' die alle cirkels op het scherm toon.
10. Aangezien je van de class meer één object nodig hebt, maak je dat object onderaan in het bestand, net na de class. Je geeft dit object de naam 'CM'.
11. Verwijder het object van de class 'movingCircle' uit je programma. *(Verwijder ook alle functies die je van dat object gebruikte.)*
12. In de update functie van je programma voer je de 'add' functie van CM uit wanneer je op de spatiebalk drukt.
13. Ook in de update functie voer je de update functie van CM uit. De draw functie zet je in de 'Draw' functie van je programma.
14. Uitbreiding: Voeg een functie 'remove' toe aan de circleManager. Deze functie heeft als argument een positie. De circle manager controleert of er een cirkel op die positie staat en verwijdert die cirkel.

10.6 Werken aan bestaande code

Wanneer je iets moet aanpassen aan code die al bestaat, dan is het begin dikwijls het moeilijkst. Hierbij enkele hulpmiddelen.

- ❑ Zorg eerst dat je de weet wat waar staat. Bekijk de verschillende classes in het programma en zorg dat je weet welke class wat doet.
- ❑ Kijk van welke classes er globale objecten bestaan.
- ❑ Zoek naar management classes en onthoudt bij welke class die horen.
- ❑ Zijn er nog andere classes die duidelijk samen horen?
- ❑ Het kan geen kwaad om op een blad papier notities te nemen van het voorgaande. Dat helpt je om alles te onthouden en te structureren.

Wanneer je dan aan de code werkt, houdt dan het volgende in de gaten:

- ❑ Welke variabelen heeft deze class? Welke informatie heb ik nodig om een functie te laten doen wat ze moet doen?
- ❑ Welke andere functies heeft deze class? Kan ik een van deze functies gebruiken?
- ❑ Indien de class een ander object moet wijzigen, welke functies heeft die class al? Moet je er misschien een toevoegen voor dat je kan doen wat je wil doen?
- ❑ Wordt er een set of get functie gevraagd? In dat geval zijn er regels die het je gemakkelijk maken.

HOOFDSTUK 11

referenties

11.1 inleiding

Bekijk even de volgende code:

```
Vec som(Vec pos1, Vec pos2) {
  return pos1 + pos2;
}

// someplace else
Vec p1(0.1, 0.3, 0.5);
Vec p2(1.9, 2.7, 0.5);

Vec p3 = som(p1, p2);
```

Alhoewel de bovenstaande code werkt zoals je verwacht, is dit helemaal niet optimaal. Je programma moet heel wat werk verrichten om de juiste waarde voor `p3` te berekenen.

Uit de leerstof van het 5de jaar heb je (hopelijk) onthouden dat een functie niet weet wat wat er in de rest van het programma gebeurt. In dit geval betekent dat dat de functie som enkel twee waarden ontvangt, die bij mekaar optelt en het resultaat "terugstuurt" naar het programma.

Maar hoe kan de functie `som()` `p1` en `p2` optellen als die onzichtbaar zijn voor deze functie? Het antwoord is eenvoudig: op het moment dat `som()` uitgevoerd moet worden, kopiëert de computer de waarden van `p1` en `p2` naar `pos1` en `pos2` die bestaan in de functie.

Wat terug naar het programma gaat is de waarde na `return`. Maar ook nu weet de functie niet af van `p3`. Dus kopiëert de computer de waarde na `return` naar `p3`.

Maar nu het slechte nieuws: al dat kopiëeren kost tijd. Hieronder zie je wat er werkelijk gebeurt tijdens het uitvoeren van de functie `som()`.

- ❑ kopieer `p1.x` naar `pos1.x`
- ❑ kopieer `p1.y` naar `pos1.y`
- ❑ kopieer `p1.z` naar `pos1.z`
- ❑ kopieer `p2.x` naar `pos2.x`
- ❑ kopieer `p2.y` naar `pos2.y`
- ❑ kopieer `p2.z` naar `pos2.z`
- ❑ reserveer geheugen voor het resultaat (we noemen dit `result`)
- ❑ tel `pos1.x` bij `pos2.x` en sla de som op in `result.x`
- ❑ tel `pos1.y` bij `pos2.y` en sla de som op in `result.y`
- ❑ tel `pos1.z` bij `pos2.z` en sla de som op in `result.z`
- ❑ kopieer `result.x` naar `p3.x`
- ❑ kopieer `result.y` naar `p3.y`
- ❑ kopieer `result.z` naar `p3.z`

En dan gebruikt deze functie nog maar eenvoudige vectoren! Wat als de argumenten containers met 2.000 vectoren zijn? Of misschien gebruik je deze `som()` functie wel op zoveel plaatsen in je code dat ze uiteindelijk 20.000 keer per seconde gebruikt wordt!

Met andere woorden: *wanneer je performante software wil maken, dan moet je in de eerste plaats vermijden dat je objecten kopiëert wanneer dat niet nodig is.*

11.2 Pass by reference

De manier waarop we tot hiertoe waarden doorgeven naar een functie, noemen we **pass by value**. We geven letterlijk de waarde van een object door, we maken met andere woorden een **kopie** van het object (in dit geval een `Vec`).

Een andere manier waarop je waarden kan doorgeven, heet **pass by reference**. Daarmee geven we niet de waarden zelf door, maar een **referentie** (of verwijzing) naar het object dat die waarden bevat.

De code ziet er dan zo uit:

```
Vec som(Vec & pos1, Vec & pos2) {
  return pos1 + pos2;
}

// someplace else
Vec p1(0.1, 0.3, 0.5);
Vec p2(1.9, 2.7, 0.5);

Vec p3 = som(p1, p2);
```

Het is dus slechts de & (ampersand) die het verschil maakt. Dat lijkt misschien niet de moeite, maar het aantal stappen dat nodig is om de functie uit te voeren, kan wel flink dalen:

- ❑ zet een verwijzing naar `p1` naar `pos1`
- ❑ zet een verwijzing naar `p2` naar `pos2`
- ❑ reserveer geheugen voor het resultaat (we noemen dit `result`)
- ❑ tel `pos1.x` bij `pos2.x` en sla de som op in `result.x`
- ❑ tel `pos1.y` bij `pos2.y` en sla de som op in `result.y`
- ❑ tel `pos1.z` bij `pos2.z` en sla de som op in `result.z`
- ❑ kopieer `result.x` naar `p3.x`
- ❑ kopieer `result.y` naar `p3.y`
- ❑ kopieer `result.z` naar `p3.z`

Het is dus bijna altijd een goed idee om een object als referentie door te geven. Enkel bij eenvoudige variabelen, zoals `int`, `float` en `bool` heeft dit geen zin. Het de referentie zou in dat geval niet even veel of zelfs meer geheugen in beslag nemen dan de waarde.

11.3 return by reference

Ongetwijfeld word je na het lezen van de bovenstaande tekst helemaal warm vanbinnen en wil je het voorbeeld nog verder verbeteren. Immers, waarom zou je ook bij de return waarde geen ampersand gebruiken?

Dus zo:

```
Vec & som(Vec & pos1, Vec & pos2) {
  return pos1 + pos2;
}

// someplace else
Vec p1(0.1, 0.3, 0.5);
Vec p2(1.9, 2.7, 0.5);

Vec p3 = som(p1, p2);
```

Helaas. Dit is geen goed idee. Want `p3` is nog steeds een gewone `Vec`, geen referentie naar een `Vec`. Dus dat zou betekenen:

- ❑ zet een verwijzing naar `p1` naar `pos1`
- ❑ zet een verwijzing naar `p2` naar `pos2`
- ❑ reserveer geheugen voor het resultaat (we noemen dit `result`)
- ❑ tel `pos1.x` bij `pos2.x` en sla de som op in `result.x`
- ❑ tel `pos1.y` bij `pos2.y` en sla de som op in `result.y`
- ❑ tel `pos1.z` bij `pos2.z` en sla de som op in `result.z`
- ❑ **geef een referentie naar `result` als functie resultaat**
- ❑ kopieer `result.x` naar `p3.x`
- ❑ kopieer `result.y` naar `p3.y`
- ❑ kopieer `result.z` naar `p3.z`

`p3` is een `Vec`, geen referentie naar een `Vec`. Dus je moet uiteindelijk de referentie toch nog kopieeren naar `p3`. Je dacht je code sneller te maken, maar ze wordt zelfs trager. Balen dus.

Maar wacht! Waarom maken we van `p3` dan niet gewoon een referentie? Ook dat is een no-go. Het probleem is dit: result is een tijdelijk object dat enkel tijdens de uitvoering van de functie bestaat. De `Vec & p3` zou dan na de uitvoering van de functie een verwijzing bevatten naar result, maar result bestaat niet meer op dat moment.

Dit kan dus nooit goed gaan. Gelukkig zal de compiler je waarschuwen, mocht je dit willen proberen.

Kan je dan nooit een return by reference gebruiken? Toch wel. Kijk maar eens naar het volgende voorbeeld:

```
Memc<Vec> points;
Vec & p = points.New();
p.x = 0.1;
...
```

Waarom kan dit wel? **points** is een container voor objecten van het type **Vec**. De functie **New()** maakt een **Vec** in die container en geeft een referentie als resultaat. De **Vec** bestaat dus ook na het uitvoeren van de functie nog steeds, binnen **points**. In dit geval is **p** dus een tijdelijke naam voor die **Vec** in **points**. Zo'n referentie is heel handig, omdat je via **p** een object in de container kan wijzigen waar je anders niet bij kan.

Kan dit dan nooit fout gaan? Jawel, maar dan zoek je het zelf. Als je de referentie gebruikt nadat je het object verwijdert, dan gaat het mis:

```
Memc<Vec> points;
Vec & p = points.New();
points.clear();
p.x = 0.1; // auch!
```

Time for Action

Open opnieuw de oefening die je op het eind van hoofdstuk 10.5 maakte. Bekijk één voor één de functies de je maakte. Vervang waar mogelijk een pass by value door een pass by reference.

HOOFDSTUK 12

Pointers (Uitbreiding)

12.1 Inleiding

Alhoewel references dikwijls heel handig zijn, zijn ze niet in alle omstandigheden bruikbaar. Om het gebruik van een reference zou gemakkelijk mogelijk te maken, verplicht de compiler je om de verwijzing te linken aan een echte variabele op het moment dat ze gemaakt wordt.

Meestal is dat geen probleem. Zo bijvoorbeeld in dit voorbeeld:

```
Vec som(Vec & pos1, Vec & pos2) {
  return pos1 + pos2;
}

// someplace else
Vec p1(0.1, 0.3, 0.5);
Vec p2(1.9, 2.7, 0.5);

Vec p3 = som(p1, p2);
```

Op het moment dat je de functie som gebruikt, ken je dadelijk p1 toe aan de reference pos1 en p2 aan de reference pos2. So far so good. Maar wat met dit voorbeeld?

```
class players {
  Memc<Player> list;

  Circle &  add(Vec2 & pos, Str & name) {
    Player & p = list.add();
    p.set(pos, name);
    return p;
```

```
  }

  Circle & findByName(Str & name) {
    FREPA(list) {
      if(Equal(list[i].getName(), name) {
        return list[i];
      }
    }
  }
}
// globaal object
players Players;

// ergens in je programma
Players.add(Vec2(1,1), "niceGuy");
player & vriend = Players.findByName("coolGuy");
```

De functie **add()** kan je geen problemen geven. De referentie argumenten kunnen niet anders dan bestaan wanneer je de functie gebruikt. Die functie geeft ook de nieuwe player als reference. Die kan in dit geval ook niet anders dan bestaan, want we hebben die net gemaakt.

De functie **findByName()** geeft ook een referentie naar een player als resultaat. Dat gaat goed zolang die player ook bestaat. Maar wat als er geen player bestaat met de naam die we zoeken? Dan is er geen resultaat, maar een referentie die niet dadelijk een waarde krijgt, geeft een fout. En dat is nu het geval met **player & vriend**.

We zouden dit kunnen oplossen door een andere player als resultaat te geven. Maar dan krijgen we de verkeerde informatie, het is immers niet de player die we zochten. In zo'n geval zou het dus beter zijn als er geen resultaat was, maar dat kan niet met een referentie.

12.2 Pointers dus...

Ook een pointer is een verwijzing naar een variabele. Maar wel met een ander symbool: de asterisk (*). Maar waar de compiler een oogje in het zeil houdt wanneer je references gebruikt, ben je bij pointers volledig op jezelf aangewezen. Pointers zijn niets meer dan een variabele waarin je een geheugenadres kan opslaan.

Stel je voor dat je een kast hebt met lades die allemaal precies 1 item kunnen bevatten. Je steekt je GSM in lade 1. Later verplaats je je GSM naar lade 2 en legt in lade 1 een briefje dat je GSM in lade 2 ligt.

Lade 1 is nu een "pointer" naar lade 2. We kunnen nu ook in lade 3 een pointer naar lade 2 steken. Lade 1 en lade 3 verwijzen nu beiden naar lade 2. Maar waarom niet gewoon lade 2 onthouden? In dit geval is het gebruikte systeem met pointers niet zo zinvol, maar het kan wel zinvol zijn.

Stel je voor dat we in alle laden, te beginnen bij 2, een GSM leggen. In lade 1 leggen we een verwijzing naar de GSM in lade 2. Iedereen die moet bellen, doet lade 1 open, en ziet dat de GSM die hij hoort te gebruiken in lade 2 ligt.

Na de GSM in lade twee een tijd gebruikt te hebben, merken we dat hij geen belkrediet meer heeft. We vervangen nu de pointer in lade 1 door een pointer naar lade 3. Iedereen weet nu dat de te gebruiken GSM in lade 3 ligt. Totdat die natuurlijk ook geen belkrediet meer heeft en we een pointer naar lade 4 in lade 1 steken.

In deze situatie heeft de verwijzing, of pointer, al veel zin. Niemand moet nog zoeken naar een bruikbare GSM, tenzij wanneer je de eerste bent die ontdekt dat de huidige GSM geen belkrediet meer heeft.

12.3 Basisbewerkingen

12.3.1 Een pointer declareren

Een pointer is een variabele, net zoals een andere. Alleen, in een pointer variabele sla je geen data op, maar een geheugenadres van data.

Een pointer declareer je zo:

```
int * p1;
int* p2;
int *p3;
Str * textPtr;
Player * playerPtr;
```

Je ziet dat je nogal wat vrijheid hebt. De eerste drie pointer variabelen zijn identiek. Ze kunnen verwijzen naar een integer. Waar de asterisk staat maakt dus niet uit.

De vierde pointer is een pointer naar een `Str` (string), terwijl de vijfde pointer naar een object van de eigen klasse `Player` verwijst.

12.3.2 Een adres toekennen aan een pointer.

Wanneer we een referentie declareren, dan moeten we die dadelijk een waarde geven. Bij een pointer is dat niet nodig.

Maar je weet dat wanneer we een integer gebruiken die we nooit een waarde gegeven hebben, die eender welke waarde kan hebben. Daarom is het een goed idee om een integer dadelijk te initialiseren, bijvoorbeeld met:

```
int i = 0;
```

Ook bij een pointer kan je dat doen. Want zonder initialisatie kan een pointer naar om het even welk geheugenadres verwijzen. En als je vervolgens via de pointer de waarde in dat geheugen wijzigt, dan kan je programma crashen.

Maar je kan niet zomaar het volgende schrijven:

```
int i = 42;
int * p = i;
```

In de code hierboven kennen we de inhoud van i, dus de waarde 42, toe aan p. Maar p verwacht een adres. Je laat p dus het geheugenadres 42 onthouden, niet het adres van i. Om aan te geven dat we het adres –en niet de waarde zelf– willen doorgeven, gebruik je een ampersand (&). Dus…

```
int   i = 42;
int * p = &i; // declaratie en initialisatie
int * p;      // enkel declaratie
p       = &i; // enkel initialisatie
p       =  i; // Fout! Het adres is nu 42
```

…betekent: sla het adres van i op in p. Je ziet dat je dit dadelijk bij het declareren kan doen, maar ook later in het programma. De laatste regel is trouwens een waar je voor moet uitkijken. De compiler zal deze toewijzing niet verbieden, maar je programma zal waarschijnlijk crashen.

12.3.3 Een variabele wijzigen via een pointer

Wat nu als we i willen wijzigen via de pointer p? `p = i;` is fout, dat zagen we hierboven. We moeten aanduiden dat we niet p willen wijzigen, maar het geheugen waar p naar verwijst. Om dat aan te duiden hebben we een extra symbool nodig: de asterisk (*).

```
int   i = 42;
int * p = i ; // het adres in p wordt het adres van i
*p      = 43; // de waarde op het adres in p wordt 43
```

We zetten de twee mogelijkheden nog eens op een rijtje:

```
p = &i; // slaat het adres van i op in p
*p = i; // slaat de waarde van i op in het adres waar p naar verwijst.
```

En het kan nog leuker! Indien p1 en p2 beiden pointers naar integers zijn, kan je de waarde waar p2 naar verwijst, opslaan in het adres waar p1 naar verwijst:

```
*p1 = *p2;
```

12.3.4 pointers naar null

Maar wat met een pointer die nergens naar verwijst? Zonder initialisatie verwijst die naar een willekeurig adres in het geheugen. Dikwijls is het nodig dat je programma weet of een pointer toegewezen is of niet. Daarom bestaat er een speciale waarde die aangeeft dat de pointer nergens naar wijst. We noemen dat een "null pointer". Je kan dat zo schrijven:

```
int * p1 = null;
```

We zouden nu de code aan het begin van dit hoofdstuk kunnen herschrijven:

```
class players {
  Memc<Player> list;

  // referenties geven hier geen probleem, dus blijven we die gebruiken
  Circle &  add(Vec2 & pos, Str & name) {
    Player & p = list.add();
    p.set(pos, name);
    return p;
  }

  // pointer result in plaats van een reference
  Circle * findByName(Str & name) {
    FREPA(list) {
      if(Equal(list[i].getName(), name) {
        return &list[i]; // notice the ampersand!
      }
    }
    // geen player gevonden met deze naam
    return null;
  }
}
// globaal object
players Players;

// ergens in je programma
Players.add(Vec2(1,1), "niceGuy");
player * vriend = Players.findByName("coolGuy");

// misschien is coolGuy niet online?
if(vriend != null) {
  Greet(vriend); // veronderstel Greet(player * p);
}
```

In dit geval kunnen we verder met als de functie `findByName()` geen speler vond. Met een referentie kon dat niet. Maar je moet dan wel controleren of je geen null als resultaat kreeg.

Note

Sommige geheugencontainers, zoals Memc, kunnen tijdens de uitvoering van het programma verplaatst worden. Bijvoorbeeld omdat er op die plaats in het geheugen niet genoeg ruimte was om nieuwe players toe te voegen. Het programma zal dan de hele container verplaatsen naar een andere plaats in het geheugen. De opgevraagde pointer is dan niet meer geldig. Vraag dus elke keer opnieuw het adres aan. Als dat niet kan, of als je zo voortdurend opnieuw hetzelfde adres moet opvragen, dan gebruik je een ander soort container, zoals Memx. Die geeft de garantie dat een gegeven adres nooit wijzigt.

12.4 Alles op een rijtje

Het is je al opgevallen dat pointers en references beiden de symbolen & en * gebruiken. En ze hebben een andere betekenis, afhankelijk van de situatie. hieronder zie je een overzicht van de mogelijkheden.

Aangenomen dat de volgende variabelen reeds bestaan:

```
int   j   = 43;
int & ref = j ;
int * ptr = &j;
```

kunnen we de volgende code gebruiken:

references en pointers

	reference	pointer
Declaratie		`int * i;`
Declaratie en Initialisatie	`int & i = j;`	`int * i = &j;`
Initialisatie		`i = &j;`
Waarde aanpassen	`i = 42;`	`*i = 42;`
Waarde aanpassen	`i = j;`	`*i = j;`
Waarde aanpassen via ref	`i = ref;`	`*i = ref;`
Waarde aanpassen via ptr	`i = *ptr;`	`*i = *ptr;`
Adres aanpassen		`i = &j;`
Adres aanpassen via ref		`i = &ref;`
Adres aanpassen via ptr		`i = ptr;`

Zoals je ziet ogen de references een stuk eenvoudiger. Maar het valt ook op dat die references veel minder mogelijkheden hebben. Daarom is het soms echt noodzakelijk om pointers te gebruiken.

Time for Action

1. Maak een programma met 5 cirkels, gespreid over de onderkant van je scherm. Je gebruikt best een memory container om de cirkels in op te slaan.
2. Als je muis zich boven een cirkel bevindt, dan verplaats je die langzaam naar boven.
3. Maak een functie die als resultaat een pointer geeft naar de hoogste cirkel. Je voert deze functie uit in update en houdt het resultaat bij een een pointer variabele.
4. In Draw teken je alle cirkels op het scherm. Vergelijk voor het tekenen van de cirkel het adres met de pointer variabele die het adres van de hoogste cirkel bevat. Als die gelijk zijn, teken je de cirkel groen, anders rood..

HOOFDSTUK 13

Enumeratie

13.1 Zo moet het niet...

Enumeraties of enum's maken het mogelijk om getallen als tekst weer te geven. Als voorbeeld gebruiken we een class enemy. Die enemy kan een warrior, een rogue of een priest zijn, en afhankelijk daarvan moet er een andere afbeelding getoond worden. Je zou boolean's kunnen gebruiken om te onthouden class de enemy heeft:

```
class enemy {
  bool warrior = false;
  bool rogue   = false;
  bool priest  = false;
  Rect r;

  void setWarrior() {
    warrior = true ;
    rogue   = false;
    priest  = false;
  }

  // de functies setRogue en setPriest zijn gelijkaardig
  // ...

  void draw() {
    if     (warrior) Images(=== warriorImage ===).draw(r);
    else if(rogue  ) Images(=== rogueImage   ===).draw(r);
    else if(priest ) Images(=== priestImage  ===).draw(r);
  }
}
```

Alhoewel bovenstaande code werkt, is ze niet erg efficient. Nu gaat het nog maar om 3 classes, maar hoe meer mogelijkheden je hebt, hoe meer variabelen je moet aanpassen bij het selecteren van een class. Veel beter zou zijn om slechts één variabele te gebruiken, want een enemy kan tenslotte maar 1 class hebben.

13.2 Dit is niet veel beter.

Je zou kunnen beslissen dat een warrior het cijfer 0 krijgt, een rogue het cijfer 1 en een priest het cijfer 2. Dan wordt de code al veel eenvoudiger.

```
class enemy {
  int type = -1;
  Rect r;

  void setType(int type) {
    T.type = type;
  }

  void draw() {
    switch(type) {
      case 0: Images(=== warriorImage ===).draw(r); break;
      case 1: Images(=== rogueImage   ===).draw(r); break;
      case 2: Images(=== priestImage  ===).draw(r); break;
    }
  }
}
```

De bovenstaande code is beter dan de eerste versie, maar het is nogal onwaarschijnlijk dat je niet vergeet welk nummer voor welke class staat. Of misschien gebruik je ergens een getal waarvoor geen class is voorzien.

13.3 Enumeration time!

De oplossing voor dit probleem zijn enum's. Enumeraties zijn lijsten van woorden. Intern wordt het eerste woord gelijk aan 0 en krijgt elk volgend woord een hoger nummer. Je kan echter steeds die woorden gebruiken in plaats van dat nummer.

```
enum ENEMY_TYPE {
  ET_NONE   ,
  ET_WARRIOR,
  ET_ROGUE  ,
  ET_PRIEST ,
}
```

```
class enemy {
  ENEMY_TYPE type = ET_NONE;
  Rect r;

  void setType(ENEMY_TYPE type) {
    T.type = type;
  }

  void draw() {
    switch(type) {
      case ET_WARRIOR: Images(=== warriorImage ===).draw(r); break;
      case ET_ROGUE  : Images(=== rogueImage   ===).draw(r); break;
      case ET_PRIEST : Images(=== priestImage  ===).draw(r); break;
    }
  }
}
```

Het voordeel van deze code is dat je overal in je programma de waarden `ET_WARRIOR` of `ET_ROGUE` kan gebruiken. Je kan de functie setType ook geen getal meegeven van een class die niet bestaat. En bovendien zie je op elk moment duidelijk over welke vrucht je het hebt.

De waarden van een enum schrijven we in hoofdletters. Dit is niet verplicht, maar wel een conventie. Je kan immers nooit het getal waar een enumeratie waarde voor staat, aanpassen. Iets als `ET_WARRIOR = 3` is dus onmogelijk. Aangezien `ET_WARRIOR` het tweede woord in de rij is, is zijn waarde steeds 1.

Het is ook niet nodig om elke waarde met `ET_` te beginnen, maar in een groter programma is het dikwijls handig om een afkorting van de enumeratie te gebruiken. “ENEMY_TYPE” wordt zo ET. Op die manier kan de autocomplete je helpen met het kiezen van een naam zodra je `ET_` getypt hebt.

Note

Esenthel bevat ook een *enumeration editor*. Hiermee kan je grafisch de waarden van een enumeratie type ingeven. Deze waarden zijn dan bruikbaar in zowel je code als de world editor.

HOOFDSTUK 14

Constanten

14.1 Globale Constanten

Dikwijls gebruik je bepaalde waarden doorheen je hele programma. Zo zou je, in een programma dat veel berekeningen met circels moet doen, vaak het getal pi nodig hebben. Dat getal kan je elke keer opnieuw berekenen, maar dat is niet zo'n goed idee omdat de uitkomst van je berekening steeds hetzelfde is. Je zou daarom een globale variabele pi kunnen declareren:

```
int pi = 3.1415926;
```

Nu kan je overal de waarde van pi gebruiken in je berekeningen. Maar stel je voor dat je ergens vergist:

```
int value = 1;
// ... more code ...
if(pi = value) {
  // do something
}
```

Je wil in de bovenstaande code controleren of 'value' gelijk is aan pi. Maar je schrijft een enkele in plaats van een dubbele =. Zo'n fout is snel gemaakt en valt op het eerste zicht niet zo op. Het gevolg is dat na de uitvoering van die code het getal pi niet meer gelijk is aan zijn oorspronkelijke waarde. Al je berekeningen zullen dus fout zijn!

Om dit soort fouten te voorkomen voorzien de meeste programmeertalen in een mogelijkheid om een variabele 'constant' te maken. Dat wil zeggen dat ze na hun declaratie niet meer aangepast

mogen worden. Om dat duidelijk te maken bestaat de afspraak om die variabelen steeds met hoofdletters te schrijven.

Hoe schrijf je zo'n variabele? In C++ doe je dat door voor het type `const` toe te voegen. Esenthel geeft je daarnaast de mogelijkheid om dat af te korten tot `C`. (Net zoals je `this` kan afkorten tot `T`). Je schrijft dus:

```
C PI = 3.1415926;
```

Dit heeft twee voordelen:

1. Je kan de waarde van PI niet langer per vergissing aanpassen.
2. Als je het getal PI in je code wil aanpassen, dan moet je dat maar op één plaats doen. *(In het geval van PI is dat wel héél onwaarschijnlijk, maar bij andere constanten kan dat dikwijls wel. Als je bijvoorbeeld een constante variabele ATTACK_RANGE gebruikt, dan kan je misschien later beslissen dat die toch iets te groot is.)*

> **Note**
>
> Omdat PI een nummer is dat alle programmeurs vaak nodig hebben, bestaat er al een constante PI in Esenthel. Niet enkel dat, er zijn ook al varianten voorzien, zoals PI_2 (de helft van PI) en PI2 (twee maal PI).

14.2 Const Argumenten

Er bestaat nog een andere situatie waarin je constanten gebruikt. Bekijk even de volgende functie:

```
float calculateDistance(Vec2 & pos1, Vec2 & pos2);
```

Je kan deze functie gebruiken om de afstand tussen twee posities te berekenen. Je leerde al in hoofdstuk 11 dat we de argumenten van die functie by reference doorgeven om het programma sneller te maken. Dat heeft één nadeel. Je zou in principe de waarden van pos1 en pos2 kunnen aanpassen in de functie. En dan zijn ook de originele waarden in je programma aangepast. De naam van de functie laat in dit geval vermoeden dat dat niet zal gebeuren. Maar je weet nooit zeker of de progammeur van die functie zich niet vergist heeft.

Als er dus ergens iets fout gaat met de variabele `pos1` in je programma, dan kan je niet anders dan ook de code van de functie `calculateDistance` nakijken. En misschien gebruikt die functie intern nog een andere functie die eveneens pass by reference argumenten heeft. Dat zou

betekenen dat je echt alle onderliggende functies moet nakijken om uit te sluiten dat de fout daar zit.

Zoiets is in grote projecten niet werkbaar. En daarom kunnen we ook een functie argument constant maken, net zoals een globale variabele. Je schrijft de functie dan zo:

```
float calculateDistance(C Vec2 & pos1, C Vec2 & pos2);
```

De gevolgen zijn dat:

1. je tijdens het maken van de functie een foutmelding krijgt wanneer je toch zou proberen pos1 of pos2 aan te passen;
2. de gebruiker van je functie zeker weet dat de waarde nooit aangepast kan zijn in de functie;
3. je bijna zeker weet dat een functie waar de argumenten **niet** constant zijn, die argumenten zal aanpassen.

Vanaf nu volg je dus de regel dat je alle functie argumenten als een const reference doorgeeft, tenzij het de bedoeling is dat de aangepaste waarde in het oorspronkelijke programma terecht komt.

Wat is nu een goede reden om een argument aan te passen? Kijk even naar de Esenthel functie:

```
void Clamp(Vec2 & value, C Vec2 & min, C Vec2 & max);
```

Het is de bedoeling dat deze functie de eerste waarde binnen het gevraagde minimum en maximum houdt. Je gebruikt de functie op deze manier:

```
Vec2 pos = Ms.pos();
Clamp(pos, Vec2(-0.4,-0.4), Vec2(0.4,0.4));
pos.draw(RED);
```

Het tweede en derde argument zijn constant. De functie `Clamp` kan dus niet het minimum of maximum aanpassen. Maar `pos` willen we natuurlijk net wel aanpassen. Hier gebruik je dus geen const reference.

HOOFDSTUK 15

Application States

Een application state is een "status" van het programma. Wanneer twee delen van een programma nooit gecombineerd worden, dan kan je er twee afzonderlijke application states van maken.

Wat veel voorkomt is bijvoorbeeld een game lobby en het eigenlijke spel. Je zal nooit de elementen van een game lobby combineren met het spel, dus kan je die volledig scheiden. Ook een login module kan een afzonderlijke application state zijn.

Elk programma heeft al een default application state. Die bestaat uit de functies `Init()`, `Shut()`, `Update()` en `Draw()`. Wanneer een state actief wordt, dan wordt `Init()` uitgevoerd. Daarna worden `Update()` en `Draw()` afwisselend uitgevoerd, totdat je het programma sluit of overgaat naar een andere state. Op dat moment wordt `Shut()` uitgevoerd.

15.1 Intro

Voor elke state maak je een afzonderlijk bestand. Bijvoorbeeld voor een intro state:

```
bool InitIntro() {return true;}

void ShutIntro() {}

bool UpdateIntro()
{
   if(StateActive.time()>3 || Kb.bp(KB_ESC)) {
      StateMenu.set(1.0);
   }
```

```
    return true;
}

void DrawIntro()
{
    D.clear(BLACK);
    D.text (0, 0, "Intro");
}

State StateIntro(UpdateIntro, DrawIntro, InitIntro, ShutIntro);
```

Je ziet dat dit veel lijkt op de standaard states in je programma. We voegen gewoon het woord Intro toe aan Init, Shut, Update en Draw. Dit houdt het overzichtelijk.

De eigenlijke state zit in de laatste regel:

```
State StateIntro(UpdateIntro, DrawIntro, InitIntro, ShutIntro);
```

Daar geef je aan dat er een nieuwe gamestate is (`StateIntro`) die de typische functies voor een programma bevat. Een `InitPre()` functie kan je niet toelaten, die dient enkel voor de echte start van het programma.

Kijk ook even naar de constructor van `State`:

```
State(Bool (*update)(), void (*draw)(), Bool (*init)()=NULL, void
    (*shut)()=NULL);
```

Komt de asterisk (*) je bekend voor? Inderdaad, we hebben met pointers te maken. Pointers naar functies in dit geval. De constructor verwacht dat we aangeven waar de functies voor deze state staan. We verwijzen dus naar de functies die we net gemaakt hebben: `UpdateIntro()` en `DrawIntro()`. Je zegt eigenlijk "zolang deze state actief is, voer je `UpdateIntro()` uit in plaats van de gewone `Update()` functie.

De volgende twee argumenten, voor de functies `InitIntro()` en `ShutIntro()` zijn optioneel. Je mag ze weglaten als er niets bijzonders moet gebeuren op dat moment.

> **Note**
>
> Indien een functie argument eindigt met `=NULL`, dan mag je het weglaten.

15.2 Menu

De code hierboven bevat ook een verwijzing naar StateMenu:

```
   if(StateActive.time()>3 || Kb.bp(KB_ESC)) {
      StateMenu.set(1.0);
   }
```

Met andere woorden: we wachten tot de huidige state 3 seconden actief is, of totdat de gebruiker op escape drukt. Dan zetten we een nieuwe application state actief met een crossfade van 1 seconde.

Deze nieuwe state zou er zo kunnen uitzien:

```
bool InitMenu() {return true;}
void ShutMenu() {}

bool UpdateMenu()
{
   if(Kb.bp(KB_ESC))return false;
   if(Kb.bp(KB_ENTER))StateGame.set(0.5);
   return true;
}

void DrawMenu()
{
   D.clear(GREY);
   D.text (0,  0  , "Menu");
   D.text (0, -0.3, "Press Enter to start the game");
   D.text (0, -0.5, "Press Escape to exit");
}

State StateMenu(UpdateMenu, DrawMenu, InitMenu, ShutMenu);
```

Deze state lijkt sterk op de vorige. Maar dit maal kunnen we met Enter naar de game zelf. En dat is dan ook weer een nieuwe application state: `StateGame`.

15.3 Game

Deze code kan je voor `StateGame` gebruiken. Maak ook nu weer een afzonderlijk bestand.

```
bool InitGame() {return true;}
void ShutGame() {}

bool UpdateGame()
{
   if(Kb.bp(KB_ESC))StateMenu.set(1.0);
   return true;
}
```

```
10 void DrawGame()
   {
      D.clear(TURQ);
      D.text (0, 0, "Game");
   }
15
   State StateGame(UpdateGame, DrawGame, InitGame, ShutGame);
```

Door tijdens de game op escape te drukken, schakelen we terug naar `StateMenu`. In deze state ga je bij een echte game natuurlijk nog heel veel code moeten toevoegen.

15.4 Default State

Dan rest ons nog het starten van het programma. We hebben nu alle nodige states, maar `StateIntro` moet nog actief worden. Dit gebeurt door in de `Init()` functie van het programma dadelijk door te schakelen naar `StateIntro`. De functies `Update()` en `Draw()` worden in dit programma dus niet gebruikt.

```
   void InitPre()
   {
      EE_INIT();
   }
5
   bool Init()
   {
      StateIntro.set();
      return true;
10 }

   void Shut() {}
   bool Update() {return false;} // unused
   void Draw   () {              } // unused
```

Time for Action

Gebruik de code van dit hoofdstuk om een programma te maken dat wisselt tussen de voorziene application states. Elke state plaats je in een afzonderlijk bestand.

Deel III

Tetris

HOOFDSTUK 16

Inleiding

Je hebt in de vorige hoofdstukken alles gezien om een eenvoudige 2D game te maken. Maar hoe breng je dat nu op een overzichtelijke manier samen in een groot project? Er bestaat eigenlijk niet één antwoord daarop. Je leert dat vooral door ervaring op te doen. Wat voor de ene persoon werkt, vind de andere misschien minder goed. Toch zijn er een aantal regels die het je zeker makkelijker kunnen maken. En wanneer je in groep werkt dan zal de lead programmer meestal ook een aantal regels opleggen die iedereen moet volgen. Dat zijn niet noodzakelijk goede of slechte regels, maar ze werken zolang iedereen hetzelfde doet.

In dit deel van de cursus bouw je een project op voor een tetris kloon. Je leert om stap voor stap een project uit te werken zonder dat je het overzicht kwijt raakt.

> **Note**
>
> Tetris bestaat uit blokken die dan weer bestaan uit vierkanten. In deze cursus bedoelen we met een blok steeds het hele tetris figuurtje. Staat er 'square' of vierkant, dan gaat het om de vierkantjes die samen een blok vormen.

16.1 Setup

Open eerst het project 'Tetris_start'. Daarin vind je alvast de graphics, geluiden en fonts die we gaan gebruiken. Er is ook alvast een lege app 'Tetris' voorzien, maar die gaan we nog niet dadelijk gebruiken.

1. Maak eerst op het hoogste niveau een Library aan. Dat doe je door rechts te klikken en 'new library' te kiezen. Je geeft deze library de naam 'Tetris parts'. Je Een library is een groene map. De code in een library kan je gebruiken vanuit elke applicatie binnen je project, net zoals de library 'Esenthel Engine' die altijd aanwezig is.
2. Maak ook een nieuwe applicatie (blauwe map). Die noem je 'squareTester'.
3. In de applicatie 'squareTester' maak je een code bestand 'main'.
4. In de library 'Tetris parts' maak je een nieuwe folder (geel) met de naam 'definitions'.
5. Maak van 'squareTester' de actieve applicatie.

Je kan in het bestand 'squareTester/main' de code van 'Tetris/initState' overnemen. Verwijder dan wel de regel

```
D.full(true);
```

Dat maakt het makkelijker om je programma te beïndigen wanneer er iets fout gaat.

16.2 Constants

In het vorige hoofdstuk leerde je over constanten. Het is een goed idee om, voordat je aan de echte code begint, enkele belangrijke constanten vast te leggen. Die kan je dan overal in je code gebruiken en later eenvoudig aanpassen als dat nodig blijkt. Je begint daarom in de map 'Tetris parts/definitions' met een nieuw code bestand dat je 'constants'noemt.

Om later de naam van het programma eenvoudig te wijzigen, maken we alvast een constante `Str` met de voorlopige naam.

```
C Str APP_NAME = "Tetris";
```

De grootte van het standaard applicatie window is niet ideaal voor dit spel. Die grootte moet in pixels worden ingegeven. We declareren daar een constante `int` voor, zodat we die later eenvoudig kunnen aanpassen.

```
// the window size on the screen, in pixels
C int WINDOW_WIDTH  = 900;
C int WINDOW_HEIGHT = 800;
```

Tetris bestaat uit rijen en kolommen. Ook die leggen we vast:

```
// this impacts the playing field
C int SQUARES_PER_ROW = 10;
C int ROWS            = 15;
```

Over de score kunnen we ook al iets zeggen. Er moeten een aantal levels zijn, de punten die je per lijn en per level krijgt liggen ook vast.

```
// the score system uses these
C int POINTS_PER_LINE  =  525;
C int POINTS_PER_LEVEL = 6300;
C int NUM_LEVELS       =    5;
```

De snelheid van het spel gaat elk level omhoog. We kunnen die begrippen ook al vastleggen.

```
// the speed will increase every level
C float INITIAL_SPEED = 1.0;
C float SPEED_CHANGE  = 0.1;
```

Als je tetris speelt, dan is er een korte periode waarin je een blok dat de onderkant van het spel bereikt nog opzij kan schuiven. Die periode leggen we ook vast.

```
// the time a block can be slided to the side
// when it hits bottom
C float SLIDE_TIME = 0.25;
```

Dan moeten we bepalen hoe groot het speelveld is. We leggen daarom de linkeronderhoek vast, en de grootte van de rechthoek die we daarop toepassen.

```
// the area reserved for the playing field
C Vec2 GAMEAREA     (-0.8, -0.8);
C Vec2 GAMEAREA_SIZE( 1.0,  1.4);
```

Een nieuw blok verschijnt altijd bovenaan het scherm. Waar dat precies is, dat kunnen we afleiden uit de waarden die we al hebben: SQUARES_PER_ROW en ROWS. Daarnaast is er ook nog een wachtpositie, die je meestal bovenaan rechts naast het speelveld toont. Het valt je misschien op dat we hier geen `Vec2` gebruiken, maar een `VecI2`. Dat is een vector waar enkel gehele getallen in passen. We willen geen tetris waar blokken halverwege tussen twee posities kunnen zitten.

```
// position for the current and next block
C VecI2 STARTPOS(SQUARES_PER_ROW / 2, ROWS - 1);
C VecI2 WAITPOS (SQUARES_PER_ROW + 4, ROWS - 3);
```

Tot slot kunnen we met de voorgaande constante ook de grootte van een vierkant berekenen. Dat heeft het voordeel dat we later de voorgaande constanten kunnen aanpassen en dat de grootte van een vierkant dan vanzelf aangepast wordt.

```
// the size of a square
C float SQUARE_SIZE = GAMEAREA_SIZE.x / SQUARES_PER_ROW;
```

> **Note**
>
> Het zal natuurlijk zelden gebeuren dat je bij de start aan een project al perfect weet welke constanten je nodig zal hebben. In praktijk zal je dus meestal enkele waarden vastleggen en daarna de lijst aanvullen wanneer je merkt dat je nog constanten nodig hebt.

16.3 Enumeraties

Maak in de folder 'Tetris parts/definitions' een nieuw code bestand 'enumerations'. hierin voorzie je alvast twee enum's die je in het project nodig zal hebben. Ten eerste is er het blok type. Tetris heeft vierkante blokken, T blokken enzovoort. Een lijst kan er zo uitzien:

```
enum BLOCK_TYPE
{
    BT_SQUARE     ,
    BT_T          ,
    BT_L          ,
    BT_BACKWARDS_L,
    BT_STRAIGHT   ,
    BT_S          ,
    BT_BACKWARDS_S,
    BT_NUM        , // number of block types used in the game
    BT_BACKGROUND , // special case, only for background
    BT_WALL       ,
}
```

De drie laatste waarden verdienen wat extra aandacht. De waarde 'BT_NUM' is handig omdat het nummer waar die waarde voor staat, gelijk is aan de mogelijkheden + 1. Dat maakt het eenvoudig om een random functie te gebruiken. Die is immers exclusief de hoogste waarde. Zo kunnen we later in het programma eenvoudig de volgende functie gebruiken:

```
blockType type = Random(BT_NUM);
```

Waarom staat BT_NUM dan niet op het eind? Wel, de laatste twee waarden zijn speciale gevallen. Het is handig voor de afwerking van het spel om ook vierkanten te gebruiken om de achtergrond en de randen van het spel te tekenen. En de kleur van zo'n vierkant wordt bepaald door het blok type. Omdat we niet willen dat die types ook in het spel gebruiken, plaatsen we BT_NUM daar voor.

Een tweede enumeratie hebben we nodig om de mogelijke richtingen van een blok te bepalen. Een blok kan nooit naar boven, maar wel naar links, naar rechts, of naar beneden. Blokken die al beneden zijn, hebben geen richting meer.

```
enum DIRECTION
{
   D_LEFT ,
   D_RIGHT,
   D_DOWN ,
   D_NONE ,
}
```

HOOFDSTUK 17

Objecten

In tetris denk je vooral aan blokken. Dat die blokken uit vierkanten bestaan is voor de speler bijkomstig, maar wel belangrijk voor de ontwikkelaar. Elk vierkant kan immers een ander vierkant raken. En wanneer een blok de bodem bereikt, dan wordt het een deel van een hoop vierkanten, die rij per rij verwijderd kunnen worden, ongeacht de vorm van het oorspronkelijke blok.

17.1 Squares

Maak in de library 'tetris parts' een nieuwe folder met de naam 'objects'. Daarin plaats je een code bestand 'square'. We maken in dat bestand een class `square` waarin we een vierkant beschrijven.

Een vierkant moet een positie hebben. Omdat het speelveld in tetris een raster is (we kunnen een vierkant niet eender waar plaatsen!) gebruiken we opnieuw een `VecI2` waarin we dus enkel gehele getallen kunnen opslaan. Verder moet het vierkant een `blockType` hebben om te bepalen in welke kleur het getekend moet worden. In het bestand 'enumerations' hebben we de types al vastgelegd, dus we kunnen die hier gebruiken.

De class `square` zal ook functies nodig hebben. We voorzien een `create` functie om de positie en het type van het blok in te stellen. Daarnaast hebben we een `move` functie nodig om het blok in een bepaalde richting te bewegen. (Die richting is dan weer de tweede enum die we maakten.) We hebben ook een functie nodig om de huidige positie op te vragen en een functie om de positie rechtstreeks te veranderen. En tot slot willen we een functie om het vierkant op het scherm te tekenen.

De class ziet er dan zo uit:

```
class square
{
private:
   VecI2      pos ;
   BLOCK_TYPE type;

public:
   void create (C VecI2 & pos, BLOCK_TYPE type) { }
   void move   (DIRECTION dir)   { }
   VecI2 getPos(             ) C { }
   void setPos (C VecI2 & pos)   { }
   void draw   (             ) C { }
}
```

Voeg de bovenstaande code alvast toe aan je project.

17.1.1 Square Tester

Je verwacht nu waarschijnlijk dat we de inhoud van deze functies gaan toevoegen. In plaats daarvan gaan we eerst naar de applicatie 'squareTester'. Je weet dat je tijdens de ontwikkeling van een programma je code heel regelmatig moet testen. Maar in een groot project is dat moeilijk. We moeten nog heel wat classes schrijven alvorens je de applicatie zou kunnen uitvoeren.

Daarom gebruiken we testprogramma's. Die schrijven we zo dat ze een specifieke class kunnen testen. In dit geval zorgen we er voor dat alle funties van de class **square** getest kunnen worden. De code voor 'squareTester' kan er bijvoorbeeld zo uitzien:

```
Memc<square> squares;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   // hier test je de create functie, met verschillende
   // soorten blokken.
   squares.New().create(VecI2( 2,  2), BT_S          );
   squares.New().create(VecI2( 4,  2), BT_T          );
   squares.New().create(VecI2( 6,  7), BT_L          );
   squares.New().create(VecI2(10, 12), BT_BACKWARDS_S);
   squares.New().create(VecI2( 8,  8), BT_BACKWARDS_L);
   squares.New().create(VecI2( 5,  1), BT_SQUARE     );
   return true;
```

```
20 }

   void Shut() {}

   bool Update()
25 {
      if(Kb.bp(KB_ESC)) return false;

      // we declareren een richting en controleren de pijltjestoetsen
      DIRECTION d = D_NONE;
30    if(Kb.bp(KB_DOWN )) d = D_DOWN ;
      if(Kb.bp(KB_LEFT )) d = D_LEFT ;
      if(Kb.bp(KB_RIGHT)) d = D_RIGHT;

      // vervolgens verplaatsen we alle blokken in deze richting. Wanneer
35    // er geen toets ingedrukt werd, zouden de squares niet mogen bewegen.
      REPA(squares)
      {
         squares[i].move(d);
      }
40
      // De functies getPos en setPos moeten ook getest worden. Dat doen we
      // wanneer de spatiebalk ingedrukt werd. We vragen dan de huidige
         positie
      // van elk vierkant en wijzigen de vertikale waarde. De gewijzigde
         positie wordt
      // terug in het vierkant geplaatst.
45    if(Kb.bp(KB_SPACE))
      {
         REPA(squares)
         {
            VecI2 pos = squares[i].getPos();
50          pos.y += 4;
            squares[i].setPos(pos);
         }
      }

55    return true;
   }

   void Draw()
   {
60    D.clear(BLACK);

      // hier testen we de draw functie van elke square.
      REPA(squares)
      {
65       squares[i].draw();
      }
   }
```

Note

Je hebt dikwijls veel mogelijkheden om een testprogramma te schrijven. Het belangrijkste is dat je op een zo eenvoudig mogelijke manier zoveel mogelijk functies van je class kan testen. Zo voorkom je dat er later fouten opduiken wanneer je alle classes samenvoegt.

17.1.2 Create en Draw

De create functie van square is heel eenvoudig. Je moet er voor zorgen dat de argumenten in de variabelen van de class terecht komen. Werk die functie nu zelf uit.

Iets moeilijker is de draw functie. Daarom overlopen we eerst even wat deze functie moet kunnen.

1. Afhankelijk van het BLOCK_TYPE moet de kleur gekozen worden.
2. De positie van de square is de positie in de grid. We moeten de werkelijke positie op het scherm berekenen.
3. We moeten een afbeelding op het scherm tonen.

De kleur bepalen doen we via een switch statement. We declaren een variabele van het type `Color` en de waarde BLACK. Daarna wijzigen we dit in de gewenste kleur voor elk type:

```
Color color(BLACK);

switch(type)
{
    case BT_SQUARE     : color = RED    ; break;
    case BT_T          : color = PURPLE; break;
    case BT_L          : color = GREY   ; break;
    case BT_BACKWARDS_L: color = BLUE   ; break;
    case BT_STRAIGHT   : color = GREEN  ; break;
    case BT_S          : color = PINK   ; break;
    case BT_BACKWARDS_S: color = YELLOW; break;
    case BT_BACKGROUND : color = Color(50, 50, 50) ; break;
    case BT_WALL       : color = WHITE  ; break;
}
```

Dan moeten we de positie bepalen. In de constante GAMEAREA staat de linkeronderhoek van het speelveld. Alle posities worden vanuit dat punt berekend. De linkeronderhoek van het vierkant op positie (0,0) zou dus gelijk moeten zijn aan de linkeronderhoek van het speelveld. Dus:

```
Vec2 screenpos = GAMEAREA;
```

Stel nu dat je een vierkant op positie (1,0) wil. Dan is de linkeronderhoek van dat vierkant gelijk aan die van het speelveld, plus de breedte van één vierkant. Voor alle andere posities geldt hetzelfde: je vermenigvuldigt de grid positie met de grootte van het vierkant:

```
Vec2 screenpos = GAMEAREA + (pos * SQUARE_SIZE);
```

De rechterbovenhoek van het vierkant is exact één vierkant verder. We kunnen dus een rechthoek maken om op het scherm te tekenen op de volgende manier:

```
Vec2 screenpos = GAMEAREA + (pos * SQUARE_SIZE);
Rect r(screenpos, screenpos + SQUARE_SIZE);
```

Om dan een image op het scherm te tekenen gebruik je de volgende code:

```
Images(=== tetris square ===).draw(color, TRANSPARENT, r);
```

17.1.3 De Test

Op dit moment zou je squareTester de blokken op het scherm moeten tonen, ook al kan je ze nog niet bewegen. Test je programma dus alvast uit.

De functies move, getPos en setPos zijn nog niet uitgwerkt, maar het lukt je zeker om dat zelf te doen. Controleer achteraf of alles werkt door de squareTester uit te voeren.

17.2 Blocks

Nu voeg je in 'tetris Parts/objects' een code bestand 'block' toe. Hierin maken we de class `block`. Net zoals een vierkant heeft een blok een positie en een type. Maar daarnaast bestaat een blok uit squares. Er moeten dus een container voor squares aanwezig zijn.

En dan zijn er nog functies. Ook `block` heeft een create functie nodig, net zoals een vierkant. En we voorzien ook nog een tweede create functie met aangepaste argumenten, zodat we een kopie van een bestaand blok kunnen maken.

Ook zijn er nog de functies move, rotate en draw nodig. En tot slot hebben we nog een functie die het type van een blok kan opvragen en een functie die de lijst met squares in het blok geeft. Deze laatste functies bevatten enkel een return statement. Ze zijn hier al uitgewerkt omdat je testprogramma niet werkt zolang er geen return in deze functies staat. Het resultaat is deze class:

```
class block
{
private:
   VecI2        pos    ;
   BLOCK_TYPE   type   ;
   Mems<square> squares;

public:
   void create(C VecI2 & pos  , BLOCK_TYPE type) { }
   void create(C block & other                 ) { }

   void move  (DIRECTION dir)   { }
   void rotate(             )   { }
   void draw  (             ) C { }

   BLOCK_TYPE       getType   () C { return type   ; }
   C Mems<square> & getSquares() C { return squares; }
}
```

Voeg deze code toe in het bestand 'block' dat je net maakte. Maak daarna een nieuwe applicatie 'blockTester'. Voeg daar alvast code aan toe om een blok te testen. Dit maal hoeft het dus geen container te zijn. Je declareert in de applicatie een blok, gebruikt de create, move, rotate en draw functies. De extra create functie en de functies `getSquares()` en `getType()` moet je nog niet gebruiken.

17.2.1 Squares Toevoegen

Alvorens je begint met de create en draw functies maak je enkele 'helper' functies om het jezelf makkelijker te maken. Deze functies zitten in de class `block`, maar zijn private. De eerste functie is `makeSBlock`. Die ziet er zo uit:

```
void makeSBlock()
{
  //     [0][1]
  // [3][2]

  squares.New().create(VecI2(pos.x    , pos.y    ), BT_S);
  squares.New().create(VecI2(pos.x + 1, pos.y    ), BT_S);
  squares.New().create(VecI2(pos.x    , pos.y - 1), BT_S);
  squares.New().create(VecI2(pos.x - 1, pos.y - 1), BT_S);
}
```

Elk blok bestaat uit 4 squares. De eerste square krijgt dezelfde positie als het blok. Dat is belangrijk om het blok later correct te roteren. De andere squares krijgen een positie die afgeleid is van de eerste positie. Je kan naar het schema in commentaar kijken om een beter beeld van de posities te krijgen. Het tweede argument (BT_S) geeft aan de square door wat

het type van het blok is. Dat is belangrijk om de square in de juiste kleur te tekenen. Je kan nu zelf de functies voor de andere blokken maken. Hieronder zie je de declaraties en een schema voor elk blok.

```
void makeSquareBlock()
{
  // [0][2]
  // [1][3]
}

void makeTBlock()
{
  //    [1]
  // [2][0][3]
}

void makeLBlock()
{
  // [2]
  // [1]
  // [0][3]
}

void makeBackwardsLBlock()
{
  //    [2]
  //    [1]
  // [3][0]
}

void makeStraightBlock()
{
  // [2]
  // [1]
  // [0]
  // [3]
}

void makeBackwardsSBlock()
{
  // [1][0]
  //    [2][3]
}
```

Vervolgens maken we nog een extra private functie: `setupSquares()`. Daarin maak je eerst de `squares` container leeg. Daarna ga je via, afhankelijk van het `BLOCK_TYPE`, de juiste functie aanroepen. Je kan vertrekken van dit voorbeeld en dat zelf verder aanvullen.

```
void setupSquares()
{
   squares.clear();
```

```
    switch(type)
    {
        case BT_SQUARE: makeSquareBlock(); break;
        // vul de rest zelf aan
    }
}
```

17.2.2 Create en Draw

De **create** functie is nu eenvoudig in te vullen. Je stelt **pos** en **type** in en voert daarna de functie **setupSquares** uit. De tweede create functie is wellicht nieuw voor je. Hier krijgen we een referentie naar een ander blok als argument. Het is de bedoeling dat we een kopie maken van dat andere blok. Dat doen we door voor elke variabele de variabele van het andere blok over te nemen. Om dat het de eerste maal is dat je een dergelijke functie nodig hebt, krijg je ze hieronder volledig te zien. Voeg deze functie ook toe aan je project.

```
void create(C block & other)
{
    T.pos  = other.pos ;
    T.type = other.type;

    squares.clear();
    FREPA(other.squares)
    {
        squares.New().create(other.squares[i].getPos(), other.type);
    }
}
```

De **draw** functie mag je weer zelf aanvullen. Om een blok te tekenen moet je gewoon alle squares in de container tekenen.

Op dit moment is het al mogelijk om een test te doen. De **create** en **draw** functies horen alvast te werken, dus je controleert of dat zo is. Voer het programma uit met alle mogelijke bloktypes en corrigeer je code als er iets niet juist blijkt.

17.2.3 Move en Rotate

Nu ga je je blok laten bewegen. De **move** functie heeft als argument een richting. Je gebruikt best een switch argument om, afhankelijk van die richting de x of de y positie aan te passen. (Denk er aan: de positie is de grid positie, niet de positie op het scherm. Je verhoogt dus met 1 en gebruikt geen tijdsdelta.)

Nadat je de positie van het blok hebt aangepast, moet je ook nog de richting doorgeven aan alle squares, zodat die zichzelf ook kunnen aanpassen. Voeg ook die code toe aan de **move** functie.

De rotate functie is iets moeilijker. We moeten hier alle squares draaien rond het punt van het blok. Dat werkt het best als het punt van het blok (0,0) is, want dan kan je roteren rond het nulpunt. Maar het blok zal waarschijnlijk niet op die positie staan. Daarom trekken we eerst het positie van het blok af van de positie van een square. Daarna wisselen we de x en y positie van die square, en we tellen de positie van het blok terug bij het resultaat. Tot slot zetten we de nieuwe positie terug in de square. Dat ziet er zo uit:

```
void rotate()
{
   FREPA(squares)
   {
      VecI2 pos = squares[i].getPos();
      pos -= T.pos;
      VecI2 newPos(-pos.y, pos.x);
      newPos += T.pos;
      squares[i].setPos(newPos);
   }
}
```

Test nu opnieuw de class via je testprogramma. Je moet nu in staat zijn om een blok te verplaatsen en te roteren. Test ook zeker met verschillende startposities, zodat je weet dat het altijd werkt.

17.3 Pile

Wanneer een blok in tetris de bodem bereikt, dan is het geen blok meer: op dat moment worden alle squares van het blok toegevoegd aan een hoop (de 'pile'). Deze class heeft daarom een memory container voor squares nodig. Daarnaast zijn er functies voorzien die de interactie met de pile eenvoudiger maken.

```
class pile
{
private:
   Memx<square> list;

   bool canMove  (C square & s,  DIRECTION dir) C {}
   void removeRow(int row                     ) C {}

public:
   void init() {}

   bool collides(C block & b, DIRECTION dir) C {}
   void add     (C block & b               )   {}
```

```
    int checkLines()   {}
    void draw      () C {}
}

pile Pile;
```

Ook voor deze class kan je een testprogramma maken. Je kan daarin manueel enkele blokken aan de pile toevoegen en die pile op het scherm tonen. De functie **collides** kan je dan testen door in het testprogramma nog een blok te maken en dat proberen te bewegen. Je kan dan bijvoorbeeld deze controle uitvoeren:

```
if(Kb.bp(KB_DOWN) && !Pile.collides(myBlock, D_DOWN)) {
  myBlock.move(D_DOWN);
}
```

Ook de **checkLines** functie valt eenvoudig te testen door ze te linken aan het indrukken van de spatiebalk. Je moet dan natuurlijk eerst genoeg blokken aan de pile toevoegen, zodat er een volle lijn in zit.

17.3.1 Init en Draw

De functies **init** en **draw** kan je ongetwijfeld zelf uitwerken. De init functie moet de container met squares leegmaken. De draw functie tekent alle elementen van de container op het scherm. Als je niet meer weet hoe je dat doet, dan kijk je best het hoofdstuk over containers nog eens na.

17.3.2 Add

De functie **add** heeft een verwijzing naar een blok als argument. Het is de bedoeling dat alle squares in dat blok toegevoegd worden aan de container. Daarom hebben we bij de class **block** een functie voorzien die ons toegang geeft tot alle squares van dat blok. De class **squares** heeft dan weer een functie **create** die als argument een positie en een bloktype heeft. Met die informatie kan je de functie zo uitwerken:

```
void add(C block & b)
{
   C Mems<square> & squares = b.getSquares();
   REPA(squares)
   {
      list.New().create(squares[i].getPos(), b.getType());
   }
}
```

Nu je deze functie klaar hebt, kan je de init, add en draw functie alvast uitproberen in je testprogramma.

17.3.3 canMove & collides

De functie `canMove` is een private functie. Ze zal dus enkel binnen deze class gebruikt worden, en wel door de functie `collides`. De functie moet voor een bepaalde square controleren of het mogelijk is om die square in een bepaalde richting (het tweede argument) te verplaatsen.

Denk er aan dat je de square niet echt mag verplaatsen: je controleert enkel of de verplaatsing mogelijk is. Om die reden is het niet mogelijk om de functie `move` van de class `square` te gebruiken. *(Dat is trouwens ook onmogelijk: precies om deze fout te voorkomen wordt de square als een const reference aan deze functie doorgegeven.)*

De functie bestaat uit de volgende stappen:

- ❑ Maak een lokale `VecI2` die je gelijkstelt aan de positie van de square.
- ❑ Verplaats deze positie in de gevraagde richting. (Kijk nog eens naar de move functie van `square` als je daar problemen mee hebt.)
- ❑ Ga de container met squares af en controleer of er een square in zit met dezelfde positie. In dit geval verlaat je de functie met 'return false'.
- ❑ Na het overlopen van de container kan je het statement 'return true' toevoegen. Immers, indien er al een square op de nieuwe positie zou staan, dan was de functie al verlaten tijdens de controle. Het de square kan dus in de gevraagde richting bewegen.

De publieke functie `collides` heeft ook een richting als argument, maar dan samen met een const reference naar een block. Het is deze functie die het tetris programma zal gebruiken.

Aangezien een `block` een functie heeft om een verwijzing naar de squares in het block te bekijken, gebruiken we die eerst:

```
C Mems<square> & squares = b.getSquares();
```

Het vervolg van de functie zal voor elke square de functie `canMove` uitvoeren. Wanneer zelfs maar één square niet in de gewenste richting kan bewegen, dan botst het block met de pile. We geven in dat geval het resultaat 'false'. Raken we door de controle, dan is het functieresultaat 'true'. Hier zie je de hele functie:

```
bool collides(C block & b, DIRECTION dir) C
{
  // get squares from this block
  C Mems<square> & squares = b.getSquares();

  // check all of them
  REPA(squares)
  {
     if(!canMove(squares[i], dir))
     {
        return true;
     }
  }
  return false;
}
```

Nu je deze functies af hebt, kan je opnieuw je testprogramma gebruiken om je class te controleren.

17.3.4 checkLines & removeRow

Tot slot moeten we de pile kunnen controleren op volle rijen en die dan verwijderen. De functie **removeRow** is niet zo moeilijk. Het argument is de te verwijderen rij. Je dient dus weeral alle elementen van de container af te gaan. Wanneer de y positie van een element gelijk is aan het argument 'row', dan moet je dit argument verwijderen. In het andere geval controleer je of de y positie groter is dan het argument row. In dat geval verplaats je de square naar beneden via de functie **move**.

> **Note**
>
> Omwille van de manier waarop de MemX container werkt, bestaat er geen functie **remove**. Je moet in dit geval **removeValid** gebruiken. De verschillen tussen de verschillende soorten containers komen later aan bod.

De functie **checkLines** moet controleren of er volle rijen in de pile zitten. Als dat zo is, dan moeten die rijen verwijderd worden. Ook moet de functie het aantal verwijderde rijen als resultaat geven, want dat is belangrijk voor de puntentelling.

Je kan de code hieronder overnemen. Zorg wel dat je alle stappen goed begrijpt.

```
int checkLines()
{
  // maak een array voor alle rijen. Hierin voorzien we 3 extra
  // rijen. Op het moment dat het spel gedaan is, zijn er immers
  // blokken in de pile gezet die hoger komen dan het eigenlijke
```

```
  // speelveld.
  int squaresInRow[ROWS + 3];

  // Zet alle waarden op nul.
  REPA(squaresInRow) squaresInRow[i] = 0;

  // Ga de lijst af en verhoog een rij afhankelijk van de \verb|y| positie
  // van de square.
  REPA(list)
  {
     int row = list[i].getPos().y;
     squaresInRow[row]++;
  }

  // Start met 0 volledige lijnen.
  int completedLines = 0;

  // Ga de array met rijen af.
  REPA(squaresInRow)
  {
     // Als het aantal square in deze rij gelijk is aan de
     // constante SQUARES_PER_ROW, dan is de rij vol
     if(squaresInRow[i] == SQUARES_PER_ROW)
     {
        // We verwijderen deze rij, maar moeten er wel rekening
        // mee houden dat er meerdere rijen verwijderd kunnen worden
        // tijdens deze functie. Als dat zo is, dan zijn de overige
        // rijen reeds een positie naar beneden verplaatst.
        removeRow(i - completedLines);

        // tot slot passen we de teller met verwijderde rijen aan.
        completedLines++;
     }
  }

  // geef het aantal verwijderde rijen als resultaat.
  return completedLines;
}
```

17.4 Wall

Het laatste element dat we nodig hebben is de 'Wall', de begrenzing van het speelveld. Dit is geen echt element omdat we de grenzen van het speelveld kunnen afleiden uit de constanten die we gedeclareerd hebben. Toch is het handig om de idee van een wall te gebruiken, zodat we bij een verplaatsing kunnen controleren of die mogelijk is, net zoals we bij de pile deden.

De class bestaat daarom maar uit twee functies. Een private functie `canMove` controleert of een square in een bepaalde richting verplaatst mag worden. Een publieke functie `collides`

controleert of een blok een muur zou raken indien het verplaatst zou worden. De class ziet er zo uit:

```
class wall
{
private:
   bool canMove (C square & s, DIRECTION dir) C {}

public:
   bool collides(C block  & b, DIRECTION dir) C {}
}

wall Wall;
```

Je kan deze class zeker zelf uitwerken. De functie **collides** is identiek aan de functie die je aan de class **pile** toevoegde. De functie **canMove** is bijna gelijk, maar je vergelijkt de nieuwe positie nu niet met de inhoud van een pile. In plaats daarvan is het functieresultaat `false` wanneer de x of de y positie kleiner is dan 0. Ook als de x positie gelijk of groter dan de waarde `SQUARES_PER_ROW` is, dan is het resultaat `false`. In alle andere gevallen is het `true`.

Je kan weer een nieuw testprogramma schrijven, of de controles toevoegen aan het testprogramma voor de pile.

HOOFDSTUK 18

Interface

In dit hoofdstuk werken we de achtergrond en het geluid van de game uit. Ook nu maak je weer een testprogramma om deze classes te testen.

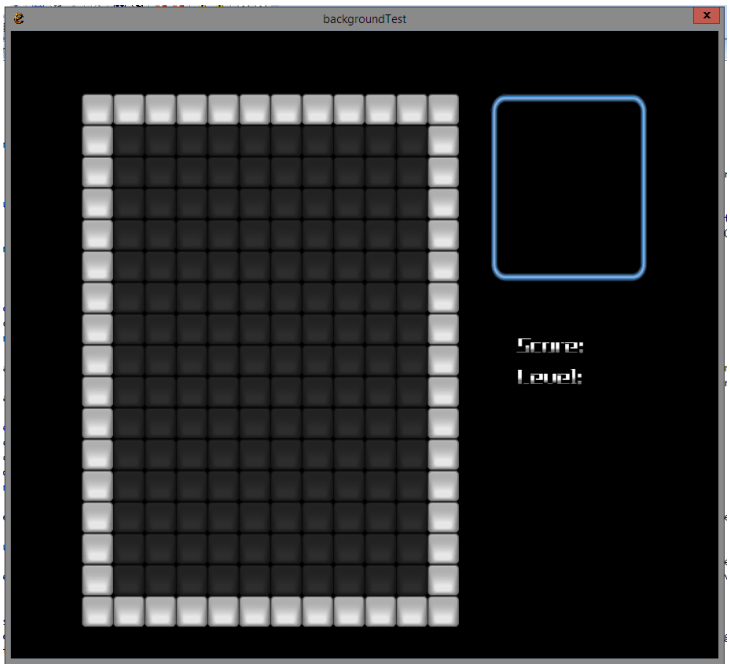
18.1 Background

De class voor de achtergrond houden we eenvoudig. We wijzigen nooit iets aan de achtergrond, dus een `create` en `draw` functie volstaan in dit geval. Ook zullen we nooit meer dan één object van deze class nodig hebben. Daarom maken we er dadelijk een object van.

Het uiteindelijke speelveld zal er uitzien zoals afbeelding 18.1. Gebruik die als referentie tijdens het testen van deze class.

Om het speelveld te tekenen gebruiken we squares. Dat had ook een afbeelding kunnen zijn, maar zo houden we de interface conform met het spel. Daarnaast heb je een `Rect` nodig om een afbeelding te tonen op de plaats waar het volgende blok verschijnt. Tot slot moeten we de positie weten waar we de score en het level tonen.

```
class background
{
   Mems<square> squares;
   Rect blockRect;
   Vec2 scorePos;
   Vec2 levelPos;

   void create()
   {
```



Figuur 18.1: De tetris achtergrond

```
10      }

        void draw()
        {
        }
15  }

    background Background;
```

In je testprogramma kan je nu al de twee functies van deze class tonen. Daarnaast voeg je de volgende regel toe aan de functie `InitPre`:

```
D.mode(WINDOW_WIDTH, WINDOW_HEIGHT);
```

Deze regel zorgt ervoor dat het window van de applicatie de grootte krijgt die we bij de definitie van de constanten ingaven. Bij de vorige testprogramma's was dat nog niet nodig, maar bij de achtergrond willen we wel zien wat het uiteindelijke resultaat is.

18.1.1 Speelveld

Zoals je weet is het speelveld een grid. Omdat we ook een 'muur' rond het speelveld tekenen, beginnen we niet op positie 0 maar op positie -1 met het toevoegen van squares. De squares in

het veld zelf geven we het type **BT_BACKGROUND**. De muren krijgen het type **BT_WALL**. Daardoor zullen ze in een andere kleur op het scherm gezet worden.

De code om de nodige squares toe te voegen krijg je kado. Eigenlijk is ze niet zo moeilijk, maar wel als je er voor het eerst aan moet beginnen. Lees deze code dus goed en vraag uitleg als je niet begrijpt hoe ze werkt. Daarna voeg je ze toe aan de functie **create**.

```
for(int x = -1; x <= SQUARES_PER_ROW; x++)
{
   for(int y = -1; y <= ROWS; y++)
   {
      if(x == -1 || y == -1 || x == SQUARES_PER_ROW || y == ROWS)
      {
         squares.New().create(VecI2(x, y), BT_WALL);
      } else
      {
         squares.New().create(VecI2(x, y), BT_BACKGROUND);
      }
   }
}
```

In de draw functie begin je met het scherm zwart te maken. Daarna toon je alle elementen van de container squares. Test je programma en controleer het resultaat.

18.1.2 Next Block

Op de plaats waar het volgende blok verschijnt plaatsen we een afbeelding. We hebben daar al een constante **WAITPOS** voor gemaakt. Maar dat is een positie in de grid. We moeten die dus steeds vermenigvuldigen met de grootte van een square (de constante **SQUARE_SIZE**). Daar komt ook nog bij dat we pas mogen tellen vanaf het speelveld, en dat is de constante **GAMEAREA**.

Om te beginnen kunnen we daarom de volgende positie berekenen:

```
Vec2 blockPos = GAMEAREA + (WAITPOS * SQUARE_SIZE);
```

Maar zoals je weet heb je voor een rechthoek een minimum en een maximum nodig. Als minimum trekken we van de gevonden positie twee squares af. Het zou logisch zijn om dan voor de maximum positie twee squares extra te rekenen. Later zal je zien dat dat er visueel minder goed uit ziet, maar dat kan je dan zelf aanpassen. De laatste regel gebruikt de twee nieuwe waarden om de eigenlijke rechthoek in te stellen.

```
Vec2 min = blockPos - (2 * SQUARE_SIZE);
Vec2 max = blockPos + (2 * SQUARE_SIZE);
blockRect.set(min, max);
```

Voeg nu aan de functie **draw** code toe om de afbeelding ‘tetris_score’ te tekenen via de rechthoek die je net berekende.

18.1.3 Tekst

Je hebt nog twee posities nodig: `scorePos` en `levelPos`. Voor de positie van de score neem je blockPos als uitgangspunt. Van de vertikale waarde trek je vier maal de `SQUARE_SIZE` af. Voor de level positie vertrek je van de gevonden positie voor de score, maar trekt van de vertikale waarde nog eens één square af.

Als je dat in orde hebt kan je te score op het scherm tekenen. Voorlopig zet je daar enkel een tekst. De echte score voegen we later toe. Er is trouwens ook een speciaal font voorzien om in dit spel te gebruiken. Dat is het bestand gui ⇒ tetrisFont ⇒ tetris Style. Als je niet meer weet hoe je het uitzicht van een tekst moet aanpassen, kan je dat nakijken in hoofstuk 3.4.

Test uiteindelijk weer je programma.

18.2 Het geluid

De volgende class die we maken is `soundManager`. Die is behoorlijk eenvoudig en je kan die zelf uitwerken. Er zijn geen create functies nodig, enkel functies die een bepaald geluid afspelen. Je voorziet de volgende functies:

1. startMusic: start de soundtrack in een loop.
2. blip: speelt het 'blip' geluid.
3. score: speelt het 'rowdone' geluid.
4. win: speelt het 'won' geluid.
5. lost: speelt het 'lost' geluid.
6. rotate: speelt het 'rotate' geluid.
7. moveDown: speelt het 'down' geluid.

Ook van deze class heb je maar één object nodig, dus je maakt onder je class het object `SoundManager`. Daarna kan je een eenvoudig testprogramma maken waarin je door op toetsen te drukken deze geluiden afspeelt. Zorg ervoor dat alle geluiden ongeveer even luid klinken. Indien nodig pas je in je class het volume van bepaalde geluiden aan.

HOOFDSTUK 19

Application States

Nu alle onderdelen klaar zijn beginnen we aan het eigenlijke programma. Daarin zijn er drie application states nodig: de initState waarin het programma start, de gameState waarin de eigenlijke gameplay plaatsvindt en de scoreState waarin je na een spel de score toont.

Maak alvast een applicatie 'tetris' met daarin een folder 'states'. In die folder voorzie je 3 bestanden: initState, gameState en scoreState.

19.1 De Init State

In de init state voorzien we de gewone application functies: **InitPre**, **Init**, **Shut**, **Update** en **Draw**. Je kan die overnemen vanuit eender welk programma dat je eerder maakte. Daarna voorzie je de functies van de nodige code.

Tot nu toe lieten we de **InitPre** functie meestal met rust. Dat is de eerste functie die uitgevoerd wordt wanneer je programma start. Je kan die functie gebruiken om enkele basisinstellingen juist te zeten. In dit geval zijn dat de naam van het programma en de afmetingen van het window. Je kan daarvoor de constanten gebruiken die we eerder ingaven.

```
void InitPre()
{
   EE_INIT();
   App.name(APP_NAME);
   D.mode(WINDOW_WIDTH, WINDOW_HEIGHT);
}
```

De `Init` functie bevat twee lijnen code. Je voert de `create` functie uit van het object `Background` en je voert de `startMusic` functie uit van het object `SoundManager`. Je kan deze functie ongetwijfeld ook zonder voorbeeld uitwerken.

De Update functie slaan we even over. Maar in de `Draw` functie kan je even uitleven. Hierin zorg je voor een mooi startscherm. Je kan zeker de afbeelding gebruiken die voorzien is en enkele teksten tonen in de bijgevoegde tetris fonts. Voor zie ook ergens de tekst "Push space to start", want dat is de manier waarop we het spel zullen starten.

19.2 De Game State

In het bestand 'gameState maak je drie functies: `GameInit`, `GameUpdate` en `GameDraw`. Deze functies laten we voorlopig ongeveer leeg, al kan je in de draw functie wel al de achtergrond tekenen. Je voorziet ook de declaratie van de state. Hier zie je hoe dit bestand er voorlopig uit dient te zien:

```
bool GameInit()
{
   return true;
}

bool GameUpdate()
{
   if(Kb.bp(KB_ESC)) return false;
   return true;
}

void GameDraw()
{
   Background.draw();
}

State GameState(GameUpdate, GameDraw, GameInit);
```

Je voegt nu in de init state een test toe: wanneer je op de spatiebalk klikt, dan schakelt je programma over naar de GameState, met een fade van 0.4 seconden. (Als je niet meer weet hoe dat moet, kijk dan in hoofdstuk 15.

Je kan nu al eens testen of je programma overschakelt naar de gamestate wanneer dat nodig is.

19.3 De Score State

Deze application state is nauw verweven met een class `score`. Het is daarom nodig dat we ze samen uitwerken. Begin met deze code voor de application state:

```
bool scoreUpdate() {
  return true;
}

void scoreDraw() {}

State ScoreState(scoreUpdate, scoreDraw);
```

Voor de class `score` maak je een afzonderlijk code bestand. De basis voor de class `score` ziet er zo uit:

```
class score
{
private:
   int   points;
   int   level ;
   bool  won   ;
   float speed ;

   void checkWin() {}

public:
   void  init      (          )   {}
   void  addPoints (int value)   {}
   float getSpeed  (          ) C {}
   int   getPoints (          ) C {}
   int   getLevel  (          ) C {}
   bool  hasWon    (          ) C {}

   void  gameIsLost(          )   {}
}

score Score;
```

Zoals je ziet bevat deze class 4 waarden: points, level, won en speed. Die laatste heeft niet echt iets met de score te maken. Maar omdat de snelheid afhankelijk is van het level, is het handig om de samen te zetten. Zo kan het score object ook de snelheid verhogen wanneer dat nodig is.

19.3.1 Wat eenvoudig is

We werken de eenvoudige functies eerst uit. De `init` functie geeft alle variabelen een geschikte startwaarde. Je begint op level 1, met 0 punten. En je hebt nog niet gewonnen. De snelheid is gelijk aan de constante `INITIAL_SPEED`.

De functies `getSpeed`, `getPoints`, `getLevel` en `hasWon` geven gewoon de waarde van de bijbehorende variabele als resultaat.

Werk deze vijf functies nu uit.

19.3.2 checkWin & gameIsLost

In `checkWin` moet je controleren of het huidige level groter is dan de constante `NUM_LEVELS`. Als dat zo is, dan wordt 'won' gelijk aan `true`. Je schakelt dan ook over naar de 'score' application state via `ScoreState.set(1);` en laat via de `SoundManager` een feestelijk geluid horen.

De functie `gameIsLost` is gelijkaardig, maar de controle is niet nodig. Je hebt verloren wanneer een blok niet meer in het speelveld past. Die controle kunnen we niet uitvoeren vanuit deze class. Ze zal later ergens anders gebeuren, en daar wordt dan deze functie uitgevoerd. In de functie zorg je dat 'won' gelijk wordt aan `false` en schakel je weer over naar `ScoreState`. En dit maal laat je een minder feestelijk geluid horen.

19.3.3 addPoints

Deze laatste functie voegt punten toe aan de score. Op dat moment gaan we ook kijken of we in een volgend level terechtkomen en zo nodig de snelheid aanpassen. We kunnen hier ook controleren of de speler het spel gewonnen heeft, want dat kan natuurlijk enkel op het moment dat hij punten krijgt.

Je kan hier weer de volgende code overnemen. Nogmaals, zorg dat je de code ook begrijpt. Enkel overtypen is niet genoeg!

```
void addPoints(int value)
{
   if(value > 0)
   {
      points += POINTS_PER_LINE * value;
      SoundManager.score();

      if(points >= level * POINTS_PER_LEVEL)
      {
         level++;
         checkWin();
         speed -= SPEED_CHANGE;
      }
   }
}
```

19.3.4 Score State

Uiteindelijk kan je nu ook de score state uitwerken. Het framework daarvoor heb je al toegevoegd (want anders kon je de state niet gebruiken in de `score` class. Zoals je ziet bevat deze state

enkel een `update` en een `draw` functie. Er moet niets geïnitialiseeerd worden, dus is een `init` functie overbodig.

Om de draw functie uit te werken kan je je inspireren op de draw functie in de Init state die je eerder maakte. Het is waarschijnlijk het mooist als die daar op lijkt. Maar je kan nu wel een licht afwijkende versie maken, afhankelijk van of je al dan niet gewonnen hebt. (Je gebruikt de functie `Score.hasWon()` om dat te controleren. Ook wil je de score op het scherm tonen vraag je de speler of hij nog een spel wil spelen. Voor dat laatste geef je de instructie dat er op y of n gedrukt kan worden.

In de update functie controleer je de toetsen y en n. Wanneer de speler op y drukt, dan schakel je weer over naar de GameState. Drukt hij op n, dan sluit je het programma af.

HOOFDSTUK 20

GameLogic

Er rest ons nog één class, maar dan wel de belangrijkste. We hebben alle onderdelen voor het spel klaar, maar die moeten nu samengebracht worden zodat het spel zich gedraagt zoals we verwachten. De class **gameLogic** dient precies daar voor. Het framework ziet er zo uit:

```
class gameLogic
{
private:
   // blocks in the game
   block currentBlock;
   block nextBlock   ;

   float forceDownCounter = 0         ;
   float slideCounter     = SLIDE_TIME;

   // to move a block completely down
   bool  toBottom      = false;
   float toBottomTimer = 0.05 ;


   bool canRotate            (C block & b               ) C {}
   bool canMove              (C block & b, DIRECTION dir) C {}
   void handleBottomCollision()   {}
   void changeFocusBlock     ()   {}
   void checkLoss            () C {}
   void handleInput          ()   {}

public:
   void create()   {}
   void update()   {}
   void draw  () C {}
}
gameLogic GameLogic;
```

Laten we eerst even de variabelen bekijken:

currentBlock Dit is het blok dat je beweegt tijdens het spel.

nextBlock Dit is het volgende blok, dat klaar staat aan de rechterzijde.

forceDownCounter Wanneer we het blok niet zelf naar beneden bewegen, dan moet dat na een korte tijd vanzelf gebeuren. Met deze timer regelen we hoe lang dat duurt.

slideCounter In tetris kan je wanneer een blok de pile raakt, nog heel even het blok opzij plaatsen. Daar hebben we dus ook een timer voor nodig.

toBottom Wanneer we op de spatiebalk drukken moet het blok helemaal naar beneden bewegen. Maar je moet het wel zien bewegen, dus je mag het niet zomaar in één keer beneden plaatsen. Met deze bool houden we bij of het huidige blok snel naar beneden moet.

toBottomTimer En die beweging heeft ook weer een timer nodig voor elke stap.

20.1 De eenvoudige functies

20.1.1 CheckLoss

Een functie die je zonder problemen kan uitwerken is `checkLoss`. Deze functie moet controleren of de speler het spel verloren heeft. Wanneer gebeurt dat? Wanneer in tetris een nieuw blok bovenaan verschijnt en dat blok kan niet naar beneden verplaatst worden, dan heeft de speler verloren. En wanneer kan een blok niet naar beneden verplaatst worden? Wanneer het zou botsen met de `Pile`.

Pile heeft al een functie `collides`. Aan die functie kan je dus het object `currentBlock` doorgeven en de gewenste richting. Geeft de functie false als resultaat, dan kan je de `Score.gameIsLost()` uitvoeren.

20.1.2 Create

De create functie dient om bij de start van een spel alle variabelen een beginwaarde te geven. Je voegt als eerste regel dit toe:

```
Random.randomize();
```

De bedoeling van deze regel is het volgende. Computers hebben een groot probleem met willekeurige getallen. Dat concept past eigenlijk niet binnen een computerlogica. We lossen dat op met het `Random` object, maar dat object heeft eigenlijk intern een lijstje met getallen die het één voor één af gaat. Elke keer je om een random getal vraagt, krijg je gewoon het volgende getal uit de lijst. Zou je dus je programma elke keer laten beginnen aan het begin van dat lijstje, dan krijg je steeds dezelfde 'willekeurige' getallen. Dat maakt je spel na verloop van tijd wel erg voorspelbaar. De functie `randomize` zorgt er voor dat je naar een willekeurige plaats in de lijst springt. Zo krijg je steeds een andere reeks getallen.

> **Note**
>
> Moest je ooit software ontwikkelen voor een casino, dan zal je nooit de standaard random functies van de programmeertaal mogen gebruiken. Het is namelijk niet zo moeilijk om een programma te schrijven dat, na ingave van de eerste drie resultaten, opzoekt waar de lijst startte. Op dat moment kan je al behoorlijk goed voorspellen wat het volgende getal in de lijst zal zijn. Aangezien een casino toch ook winst wil maken, gebruikt men voor dat soort software een meer complexe random library.

De volgende statements zorgen voor twee willekeurige blokken:

```
currentBlock.create(STARTPOS, (BLOCK_TYPE)Random(BT_NUM));
nextBlock   .create(WAITPOS , (BLOCK_TYPE)Random(BT_NUM));
```

Je ziet dat we de constanten `STARTPOS` en `WAITPOS` gebruiken om de posities in te stellen. Het tweede argument is het blok type. We willen telkens een willekeurig blok, dus we gebruiken de random functie. Het argument van `Random` bepaalt het hoogste getal. Maar die waarde is niet inclusief: dat wil zeggen dat je, als je bijvoorbeeld `Random(3)` schrijft, het resultaat 0, 1 of 2 kan zijn. Niet 3 dus. Waarom schrijven we hier dan `BT_NUM`? Daarvoor moet je even terug in het bestand 'enumerations' kijken. `BT_NUM` is het achtste element in de lijst. Het eerste element is gelijk aan 0, dus het achtste element is gelijk aan 7. De `Random` functie zal hier dus een getal van 0 tot en met 6 teruggeven. En omdat een enumeratie niets anders is dan een naam voor een getal, kunnen we dat getal eenvoudig terug omzetten naar een `BLOCK_TYPE`. Want dat is het type dat de create functie verwacht.

Na deze statements moeten we `forceDownCounter` de waarde 0 geven, en slideCounter gelijk stellen aan `SLIDE_TIME`. Die statements kan je zelf wel verzinnen. Tot slot voer je ook de `init` functies van `Pile` en `Score` uit.

20.1.3 Draw

De draw functie van deze class moet drie elementen op het scherm tonen: het huidige blok, het blok in de wachtpositie en de pile. Voeg de statements toe om dat te doen.

20.2 Iets moeilijker

20.2.1 Can Rotate

We hebben al functies om bij een verplaatsing collisions met de pile of het speelveld te controleren. Nu moeten we ook controleren of het mogelijk is om een blok te roteren. Daarvoor dient de functie `canRotate`.

In deze functie maken we eerst een nieuw blok. We willen namelijk het blok dat we als functie argument binnen krijgen niet roteren, maar enkel controleren of het mogelijk is. Bij dat nieuwe blok voeren we de create functie uit, met als argument het bestaande blok `b`. Daarna roteren we het nieuwe blok.

Nu kunnen we controleren of dit nieuwe blok botst met `Wall` of `Pile`. Het tweede argument is dan de richting `D_NONE`, want we willen geen verplaatsing controleren. Wanneer een van die functies aangeeft dat er een collision is, dan is het functieresultaat `false`. In het andere geval wordt het `true`.

20.2.2 Can Move

De functie `canMove` zorgt er voor dat we een verplaatsing in één keer kunnen controleren. We moeten namelijk zowel collisions met de wall als met de pile in de gaten houden. Als een van deze functies `false` als resultaat heeft, dan is het functieresultaat ook `false`. Is dat niet zo, dan is het resultaat `true`. De argumenten van de functies kan je gewoon doorgeven aan de collide functies van `Wall` en `Pile`.

20.2.3 Change Focus Block

Op het moment dat een blok beneden is, moet je het toevoegen aan de pile en bovenaan een nieuw blok tonen. Het type van de blok moet gelijk zijn aan het blok in de wachtpositie. Daarna moet je ook nog een beslissen wat nu het volgende blok zal worden.

Je kan deze functie in drie stappen uitwerken:

1. Voeg het huidige blok toe aan de pile.
2. Voer opnieuw de create functie van het huidige blok uit. Als eerste argument gebruik je de constante `STARTPOS`. Het tweede argument is het type van het blok op de wachtpositie. (Zoek in de class `block` naar een functie die je dat type geeft.)
3. Voer nu ook opnieuw de create functie van ‘nextBlock’ uit. Die is gelijk aan het statement dat je eerder in de create functie van deze class schreef.

20.2.4 Handle Bottom Collision

Deze functie beschrijft wat er moet gebeuren als een blok de pile raakt. Ook dat zijn vier eenvoudige statements:

1. Voer de functie `changeFocusBlock` uit.
2. Controleer of er rijen verwijderd kunnen worden uit de Pile.
3. Geef het aantal verwijderde lijnen (het resultaat van de vorige regel) door aan het `Score` object.
4. Controleer via de functie `checkLoss` of het spel gedaan is.

20.2.5 Handle Input

We hebben ook een functie nodig die reageert wanneer we een toets indrukken. Dat is de functie `handleInput`. Elke toets die we kunnen indrukken tijdens het spel moet hier behandeld worden. Zo moet, wanneer je de pijltjestoets naar beneden indrukt, eerst gecontroleerd worden of een verplaatsing naar beneden wel mogelijk is. Als dat zo is, dan verplaats je het blok naar beneden en laat je een geluidje horen. Dat kan zo:

```
if(Kb.bp(KB_DOWN))
{
   if(canMove(currentBlock, D_DOWN))
   {
      currentBlock.move(D_DOWN);
      SoundManager.blip();
   }
}
```

De code om een blok naar links of rechts te verplaatsen is gelijkaardig. Die werk je dus weer zelf uit.

De 'UP' toets gebruik je in tetris om een blok te roteren. Voeg dus ook code toe om te controleren of deze toets wordt ingedrukt. Dit maal moet je enkel de functie `canRotate` uitvoeren met het huidige blok. Als het resultaat van die functie `true` is, dan roteer je het blok en laat je weer een geluidje horen.

En als laatste is er de spatiebalk. Bij het indrukken van de spatiebalk moet een blok helemaal tot beneden bewegen. Om dat te doen geven we de variabele 'toBottom' de waarde `true` en de variabele 'toBottomTimer' de waarde 0. En ook hier speelt er weer een geluid.

20.3 Update

En dan komen we bij de laatste functie, die het centrum van het spel vormt: de functie **update**. Die functie voert achtereenvolgens verschillende controles uit.

20.3.1 Force Down

Eerst kijken we of het tijd is om een blok naar beneden te verplaatsen. Daarvoor moeten we de 'forceDownCounter' verhogen. Als die hoger is dan de huidige game speed, dan moet het blok een stap naar beneden:

```
forceDownCounter += Time.d();
if(forceDownCounter > Score.getSpeed())
{
   if(canMove(currentBlock, D_DOWN))
   {
      currentBlock.move(D_DOWN);
      forceDownCounter = 0;
   }
}
```

20.3.2 Slide Counter

De slide counter dient om het blok nog even opzij te kunnen bewegen wanneer het de pile raakt. Daarom moeten we eerst weten of het blok de pile raakt en dat is het geval als het niet meer naar beneden kan. In dat geval zullen we de waarde van 'slideCounter' verlagen. Als de slideCounter nul wordt, dan voeren we de functie **handleBottomCollision** uit.

```
if(!canMove(currentBlock, D_DOWN))
{
   slideCounter -= Time.d();
} else
{
   slideCounter = SLIDE_TIME;
}

if(slideCounter <= 0)
{
   slideCounter = SLIDE_TIME;
   handleBottomCollision();
}
```

20.3.3 To Bottom

De bool 'toBottom' is `true` wanneer de speler op de spatiebalk drukte. We verplaatsen het blok dan snel naar beneden, maar dat moet nog steeds stap voor stap gebeuren om zichtbaar te zijn. We gebruiken dus een counter met een kleine waarde waar we telkens weer de tijdsdelta van aftrekken. Elke keer dat de counter nul wordt verplaatsen we het blok een positie naar beneden. Als dat niet meer mogelijk is, dan schiet de bovenstaande code (Slide Counter) in actie.

Enkel wanneer het blok niet naar beneden glijdt, controleren we de input van de speler.

```
if(toBottom)
{
   toBottomTimer -= Time.d();
   if(toBottomTimer < 0)
   {
      toBottomTimer = 0.05;
      if(canMove(currentBlock, D_DOWN))
      {
         currentBlock.move(D_DOWN);

      } else
      {
         toBottom = false;
      }
   }
} else
{
   handleInput();
}
```

Daarmee is ook deze functie af. Het enige wat je nu nog moet doen is de `create`, `update` en `draw` toevoegen aan de Game State. En daarna natuurlijk alle fouten oplossen tot je programma werkt zoals het hoort.

20.4 Nabespreking

Je hebt nu een volledig project uitgewerkt. Hopelijk zal je onthouden hoe belangrijk het is om alles in classes onder te brengen. Ook het gebruik van testprogramma's is erg belangrijk om een groot project goed uit te werken.

Maar natuurlijk komt deze manier van werken niet helemaal overeen met de realiteit. De auteur van deze cursus wist precies wat er moest gebeuren en in welke volgorde je dat het best kon aanpakken, nog voor je aan deze oefening begon. Als je zelf aan een project begint dan is dat wel anders. Het is heel gewoon dat je de classes die je vooraf maakt meermaals moet aanpassen.

Dikwijls blijken er toch nog functies te ontbreken, of schrijf je functies die je uiteindelijk niet nodig blijkt te hebben. Dat is, zeker voor een beginnende programmeur, heel normaal.

Enkel door ervaring leer je steeds beter inschatten welke functies een class nodig zal hebben. En zelfs dan kan je dat niet altijd exact voorspellen.

Deel IV

Gui

HOOFDSTUK 21

Gui

Een grafische user interface (Gui) kan je visueel ontwerpen via de gui editor. Maar om je interface te gebruiken in je applicatie zal je wel moeten programmeren. Bekijk eerst hoe je de gui editor gebruikt. Je kan daarvoor gebruik maken van deze youtube tutorial:

https://www.youtube.com/watch?v=eFsBxC6pGxE

21.1 Een gui laden

Voor elk window maak je best een afzonderlijk code bestand, dat houdt het overzichtelijk. In dat bestand begin je een class, die bij voorkeur dezelfde naam heeft als de Gui en het bestand.

```
// Een class voor een login window
class loginWindow
{
private:
    // een GuiObjs object kan een Gui object bevatten
    // dat je maakt via de editor.
    GuiObjs objs;

public:
    // Deze create functie zullen we later in het programma
    // uitvoeren om het login window te laden.
    void create()
    {
        // het GuiObjs object kan je gebruiken om een gui te
        // laden. Met de functie load kan je hier via drag and
        // drop een GUI object plaatsen.
```

```
        objs.load( --- Drop Gui Object here --- );

        // Uiteindelijk voeg je deze gui toe aan de Gui Manager
        Gui += objs;
    }

}

// aangezien je normaal gezien maar 1 object nodig hebt van elke gui class,
// kan je dat hier al maken.
loginWindow LoginWindow;
```

De bovenstaande code laad je gui in het geheugen en voegt die toe aan de Gui manager. In je programma ga je dan de **create()** functie uitvoeren en de Gui updaten en tekenen.

```
void InitPre()
{
    EE_INIT();
}

bool Init()
{
    // voor elk gui object voer je de create functie uit
    // tijdens de Init fase
    LoginWindow.create();
    return true;
}

void Shut() {}

bool Update()
{
    // Wanneer je een of meerdere gui classes gebruikt, dan
    // update je de Gui manager tijdens de applicatie update
    Gui.update();
    return true;
}

void Draw()
{
    D.clear(WHITE);

    // Wanneer je een of meerdere gui classes gebruikt,  dan
    // laat je de Gui manager alles op het scherm tekenen. Je
    // doet dit op het einde van de Draw functie,  omdat de
    // de Gui boven op de andere objecten hoort.
    Gui.draw();
}
```

Time for Action

Maak een login window (met naam, wachtwoord textlines en een ok en cancel button) en zorg via bovenstaande code dat het in een programma op je scherm verschijnt.

21.2 Pointers naar elementen

Je hebt nu wel een gui geladen, maar je wil waarschijnlijk ook de elementen van de gui gebruiken. Maar die zitten in `GuiObjs objs`. Je maakt daarom pointers aan naar elk element waar je iets mee wil doen. Na het laden van de gui zoek je naar de elementen en wijs je die toe aan de gewenste pointer.

```
class loginWindow
{
private:
   GuiObjs objs;
   // een pointer naar een Window, met de naam window
   Window * window;
   // een pointer naar een Button, met de naam buttonClose
   Button * buttonClose;

public:

   void create()
   {
      objs.load( --- Drop Gui Object here --- );
      // zoek in objs naar een Window met de naam "window"
      window = objs.findWindow("window");
      // zoek in objs naar een Button met de naam "buttonClose"
      buttonClose = objs.findButton("buttonClose");

      Gui += objs;
   }

}
loginWindow LoginWindow;
```

Je mag je elementen noemen zoals je wil, maar de naam waar je naar zoekt (met findWindow, findButton, …) moet wel gelijk zijn aan de naam die je het element in de Gui Designer hebt gegeven. Als je toch een verkeerde naam zoekt, dan zal je pointer nergens naar verwijzen. Als je dan later die pointer gebruikt in het programma, dan zal je applicatie crashen.

Eens een pointer verwijst naar een element, kan je hem gebruiken in je code. Het volgende voorbeeld gebruikt de pointer naar het window om dit te tonen en te verbergen:

```
class loginWindow
{
private:
   GuiObjs objs;
   Window * window;
   Button * buttonClose;

public:

   void create()
   {
      objs.load( --- Drop Gui Object here --- );
      window = objs.findWindow("window");
      buttonClose = objs.findButton("buttonClose");

      // verberg het window na het laden
      window.hide();

      Gui += objs;
   }

   // deze functie kan je eender waar in je programma gebruiken
   void show() {
      // dit zorgt voor een fade in van je window
      window.fadeIn();
   }
}
loginWindow LoginWindow;
```

De bovenstaande code laat je toe om bijvoorbeeld in de update functie van je applicatie de volgende code te plaatsen:

```
if(Kb.bp(KB_F5)) LoginWindow.show();
```

21.3 Callback functies

Sommige gui elementen, zoals buttons, kan je een functie toewijzen. Die functie wordt dan automatisch uitgevoerd bij een actie. Bij een button is die actie het moment dat iemand klikt op de button. Bij een slider is het het moment waarop de slider verplaatst wordt. Een TextLine heeft dan weer een actie als iemand de waarde aanpast.

De functie die je toewijst aan een element kan eender welke naam hebben, maar het formaat moet wel juist zijn. Je moet de functie declareren als `static void` en met als argument een referentie naar de class die je maakt. Bijzonder aan een `static` functie is dat die geen rechtstreekse toegang heeft tot de objecten in je class. Daarom is de referentie naar de class

noodzakelijk. Wanneer je de functie doorgeeft aan de button, gebruik je `T` om het object door te geven waar de functie bij hoort. Andere functies of elementen van de class moet je dus ook via die referentie benaderen.

```
class loginWindow
{
private:
   GuiObjs objs;
   Window * window;
   Button * buttonClose;

   // Deze functie wordt uitgevoerd wanneer iemand
   // op de button klikt.
   static void myButtonFunction(loginWindow & obj) {
      obj.window.fadeOut();
   }

public:

   void create()
   {
      objs.load( --- Drop Gui Object here --- );
      window = objs.findWindow("window");
      buttonClose = objs.findButton("buttonClose");
      window.hide();

      // Wijs de functie hierboven toe aan de button
      buttonClose.func(myButtonFunction, T);
      Gui += objs;
   }

   void show() {
      window.fadeIn();
   }
}
loginWindow LoginWindow;
```

Note

Pas op. De editor maakt een fout als je de naam van een functie typt: er komen vanzelf haakjes achter. Wanneer je een functie gebruikt hoort dat zo, maar in dit geval moet je enkel de naam doorgeven. Verwijder de haakjes dus.

```
buttonClose.func(myButtonFunction(), T); // fout!
buttonClose.func(myButtonFunction, T);   // juist.
```

Time for Action

Vul de vorige oefening aan met de mogelijkheid om je login window te tonen en te verbergen via de functietoets F1. Zorg ook dat zowel de ok als de cancelbutton het window terug sluiten.

21.4 De inhoud van een element

Dikwijls heeft een element ook een inhoud. Een `TextLine` heeft een tekst als inhoud. Een `Slider` heeft een waarde tussen 0 en 1, afhankelijk van zijn positie. Een `CheckBox` heeft als inhoud `true` wanneer aangevinkt, of `false` indien niet.

Die inhoud opvragen kan complex zijn. Stel je voor dat we een `TextLine` hebben:

```
TextLine tl;
```

De waarde vragen vraag je dan op met:

```
Str waarde = tl();
```

Eenvoudig! Maar… meestal hebben we geen `TextLine` in onze code, maar een **pointer** naar zo'n element. Dus moet je aangeven dat je wil werken met het element waar die pointer naar verwijst. Dat kan zo:

```
Str waarde = *tl();
```

Als je dan ook nog eens vanuit een `static void` functie werkt, zoals een functie die wordt uitgevoerd wanneer je op een knop klikt, dan moet je ook het object van de class meegeven. Bij de class `loginWindow` van hierboven wordt dat dan:

```
Str waarde = (*LoginWindow.tl)();
```

Je kan de inhoud van een element ook wijzigen. Dat is gelukkig een stuk eenvoudiger:

```
Str waarde = "mijn tekst";
tl.set(waarde);
// of in een static void functie
LoginWindow.tl.set(waarde);
```

Een verder uitgewerkt voorbeeld van het loginWindow zou er zo kunnen uitzien:

```
class loginWindow
{
private:
   GuiObjs objs;
   Window * window;
   Button * buttonLogin;
   TextLine * tlName;
   TextLine * tlPass;

   static void tryLogin(loginWindow & obj) {
      Str name = (*obj.tlName)();
      Str pass = (*obj.tlPass)();

      if(Equal(name, "john") && Equal(pass, "secret")) {
         obj.window.fadeOut();
      }
   }

public:

   void create()
   {
      objs.load( --- Drop Gui Object here --- );
      window = objs.findWindow("window");
      buttonLogin = objs.findButton("login");
      tlName = objs.findTextLine("name");
      tlPass = objs.findTextLine("pass");

      window.hide();
      buttonClose.func(tryLogin, T);

      Gui += objs;
   }

   void show() {
      window.fadeIn();
   }
}
loginWindow LoginWindow;
```

Time for Action

Maak een programma met een login window. De achtergrond van het programma is zwart, maar wordt wit wanneer je ingelogd bent. Je toont dan ook de login naam op het scherm via `D.text()`. Als uitbreiding kan je ook eens naar een andere application state overschakelen na het inloggen.

21.5 Data en interface: Never mix!

Een belangrijke regel bij het schrijven van een GUI is de volgende:

> **Note**
>
> GUI en data houd je strikt gescheiden.

Een GUI dient om data aan te passen. Maar die data zelf hoort **NOOIT** (nooit) thuis in de gui class. Je maakt dus steeds een afzonderlijke class voor de data. Je gebruikt dan functies in je gui class om die data te lezen en te wijzigen. Een voorbeeld:

```
// een data class voor een player
class player {
private:
   Str name;
   int age;

public:
   void setName(C Str & name) { T.name = name; }
   void setAge (int     age ) { T.age  = age ; }
   Str  getName(            ) { return   name; }
   int  getAge (            ) { return   age ; }

   void draw() {
      D.text(0,    0, S + "name: " + name);
      D.text(0, -0.1, S + "age : " + age );
   }
}
player Player;
```

```
// een GUI class
class changePlayerGUI {
private:
   GuiObjs objs;
   Window   * window  ;
   TextLine * tlName  ;
   TextLine * tlAge   ;
   Button   * buttonOK;

public:
   void create() {
      objs.load( -- drop gui object here ---);
      window   = objs.findWindow  ("window");
      tlName   = objs.findTextLine("name"  );
      tlAge    = objs.findTextLine("age"   );
      buttonOK = objs.findButton  ("ok"    );

      buttonOK.func(changeData, T);
```

```
      tlName  .set(        Player.getName() );
      tlAge   .set(TextInt(Player.getAge ())); // converteer int naar Str
         met TextInt()

      Gui += objs;
   }

   static void changeData(changePlayerGUI & obj) {
      Str name = (*obj.tlName)();
      Str age  = (*obj.tlAge )();
      int ageInt = TextInt(age);
      Player.setName(name  );
      Player.setAge (ageInt);
   }
}

changePlayerGUI ChangePlayerGUI;
```

HOOFDSTUK 22

Gui Window

Alhoewel je gui elementen ook direct op het scherm kan zetten, plaats je ze meestal in een window. Zelfs al wil je dat window niet tonen (je kan het later onzichtbaar maken) is dit de meest eenvoudige manier om elementen bij mekaar te houden. Wil je bijvoorbeeld verschillende elementen samen tonen of verbergen, dan plaats je ze best samen in een window.

22.1 Definitie

Kijk eens naar de definitie van een `Window`:

```
const_mem_addr struct Window : GuiObj // Gui Window !! must be stored in
    constant memory address !!
{
    Byte      flag       ,
    ...
```

De commentaar en ook het eerste woord `const_mem_addr` laten je weten dat een window een vast geheugen adres nodig heeft. Wat betekent dat? Wel, je mag een window nooit opslaan in een gewone container. De elementen van een `Memc` kunnen wel eens naar een andere locatie in het geheugen verplaatst worden zonder dat je dat zelf merkt. Voor GUI elementen mag zoiets niet gebeuren.

Nu is dat meestal geen probleem. Wanneer je een GUI ontwerpt dan wil je van elk window en de elementen daarin meestal maar één object maken en dat doe je dan onder de declaratie van de class die je daarvoor maakt. (In het vorige hoofdstuk was dat `loginWindow LoginWindow`).

Op die manier heb je automatisch een vast geheugen adres. Heb je toch een container met meerdere objecten van deze class nodig? Gebruik dan de container `Memx` of `Meml`.

Wat ook opvalt is het laatste woord: `GuiObj`. Dat betekent dat een Window gebaseerd is op de class `GuiObj`. Alle eigenschappen van deze class zijn dus ook beschikbaar in `Window`. We zeggen daarom dat `GuiObj` de base class van `Window` is.

22.2 Class Methods

Als je verder in de definitie van de class `Window` kijkt, dan zie je heel wat lidfuncties. En daar komen ook de lidfuncties van `GuiObj` nog bij. Zolang je je gui ontwerpt met de gui editor, zal je de meeste van deze functies niet zo vaak nodig hebben. We bespreken hier enkele functies die toch handig kunnen zijn:

`setTitle(C Str &title)` Via deze functie kan je de titel van het window wijzigen.

`fadeIn()` Samen met de functie `fadeOut()` bepaal je de zichtbaarheid van een window.

`showing()` Samen met de functie `hiding()` controleer je de zichtbaarheid van een window. Je kan bijvoorbeeld de pijltjestoetsen enkel gebruiken om je avatar te bewegen wanneer een window niet zichtbaar is.

`button[3]` is geen functie. Een window bevat altijd die buttons met een specifiek doel: minimize, maximize en close. De typische buttons die je vaak in de rechterbovenhoek ziet. Als je deze buttons wil tonen, dan kan je na het laden van je window de `show()` functie van de gewenste buttons uitvoeren.

Time for Action

Maak in programma waarin je een window toont zolang de spatiebalk ingedrukt is. Wanneer het window zichtbaar is, dan stel je de titel van het window gelijk aan de tijd dat je programma loopt. (Gebruik de functie `Time.appTime()`.)

Er is nog een functie die bijzondere aandacht verdient: de functie `pos(C Vec2 &pos)` van de base class `GuiObj`. Deze functie komt vaak van pas om de positie van je window te corrigeren. Je kan in de gui editor wel de positie van een window bepalen, maar dat is vaak niet voldoende. Je wil bijvoorbeeld een window aan de linkerkant van het scherm, maar niet ieder scherm heeft dezelfde aspect ratio. Daarom bereken je best de positie voordat je een window toont. Voor de linkerbovenhoek zou dat er zo uit zien:

```
void show() {
   window.pos(Vec2(-D.w(), D.h());
   window.fadeIn();
}
```

Andere posities vragen dikwijls iets meer werk. Dan moet je immers ook de afmetingen van het window in rekening brengen. Gelukkig kan dat eenvoudig via de lidfunctie `rect()`. Die geeft je informatie over de grootte van je window, dat natuurlijk een rechthoek is. In dit voorbeeld zie je hoe je een window aan de linkeronderhoek van het scherm plaatst.

```
void show() {
   Vec2 pos;
   pos.x = -D.w();
   pos.y = -D.h() + window.rect().h();
   window.pos(pos);
   window.fadeIn();
}
```

Time for Action

Plaats een window aan de rechterkant van het scherm, in het midden. Maak ook een tweede functie `showCentered()` die het window in het midden van het scherm toont.

22.3 Dialogs

Esenthel bevat ook enkele windows met en meer specifiek doel. Een daarvan is de class `Dialog`. Als je naar de declaratie van die class kijkt, dan zie je dat een `Dialog` de class `Window` als basis heeft. Dat betekent dat ook de functies van `GuiObj` beschikbaar zijn.

Een dialog zal je niet een de gui editor ontwerpen. De lidfunctie `autoSize()` wordt gebruikt om na het instellen van de tekst en de buttons alle elementen voldoende plaats te geven. Dat betekent dat je in dit geval geen `GuiObjs` en ook geen pointers gebruikt. Hier een voorbeeld:

```
class confirmExit {
private:
   Dialog dialog;

   static void cancelFunc(confirmExit & obj)
   {
      obj.dialog.fadeOut();
   }

   static void proceedFunc(confirmExit & obj)
   {
      Exit();
   }

public:
   void create() {
      Mems<Str> buttonTexts;
      buttonTexts.New() = "cancel";
```

```
        buttonTexts.New() = "proceed";
        dialog.create("Exit?", "Ben je zeker dat je dit programma wil
            sluiten?", buttonTexts);
        dialog.autoSize();
        dialog.hide();

        dialog.buttons[0].func(cancelFunc, T);
        dialog.buttons[1].func(proceedFunc, T);

        Gui += dialog;
    }

    void show() {
        dialog.fadeIn();
    }
}

confirmExit ConfirmExit;
```

Time for Action

Voeg deze class toe aan een programma en toon de dialog wanneer de gebruikt op escape drukt, in plaats van daar dadelijk het programma af te sluiten.

HOOFDSTUK 23

Gui Buttons

De meest gebruikte functie van de class button is `func()`. Daarmee koppel je een callback functie aan de button. Die functie wordt dan uitgevoerd wanneer je op de button klikt. In de inleiding werd dit al besproken, maar denk er aan dat dit een statische functie moet zijn, die dus los staat van het eigenlijke object. Alhoewel het niet strikt noodzakelijk is, geef je dus best altijd een referentie naar het huidige object door aan de button.

Enkele andere handige functies zijn:

`enabled(bool enabled)` laat je toe om op bepaalde momenten een button non-actief te maken.

`setText(C Str &text)` wijzigt de tekst van een button.

`bool sound` door deze eigenschap op 'true' te zetten, wordt er een geluid afgespeeld wanneer je op deze button drukt. Het geluid zelf kan je instellen via `Gui.click_sound_id`. Standaard is dat geluid leeg, maar je kan er eender welk geluidsbestand aan toewijzen. (Let op, dit is geen functie maar een property!)

Time for Action

Pas de vorige oefening aan, zodat de buttons in je dialog een geluid afspelen.

23.1 Toggle Buttons

Een button kan ook gebruikt worden als toggle button. In dat geval gedraagt die zich net iets anders. Een enkele toggle button kan je als alternatief voor een checkbox gebruiken. Bijvoorbeeld om bepaalde elementen op het scherm te tonen. Hieronder zie je een voorbeeld van een toggle button die een crosshair aan of uit zet.

```
class crossHair
{
private:
   Button button;
   bool on = false;

   static void buttonFunc(crossHair & obj)
   {
      obj.on = obj.button();
   }

public:
   void create()
   {
      button.create(Rect(-D.w() + 0.1, D.h() - 0.2, -D.w() + 0.6, D.h() -
          0.1), "cross on/off");
      button.mode = BUTTON_TOGGLE;
      button.set(false);
      button.func(buttonFunc, T);
      Gui += button;
   }

   void draw()
   {
      if(!on) return;
      Circle(0.1).draw(RED, false);
      Edge2(0, -0.12, 0, 0.12).draw(RED);
      Edge2(-0.12, 0, 0.12, 0).draw(RED);
   }
}

crossHair CrossHair;
```

Enkele zaken vallen wellicht op in deze class. Ten eerste wordt de gui editor niet gebruikt. Om slechts één button op het scherm te tonen zou dat wat omslachtig zin. Daarom is de Button geen pointer, maar een echt object. We dienen dan wel zelf de functie **create** uit te voeren, met als argument een rechthoek en een tekst.

Daarna wordt de mode aangepast, zodat we een toggle button krijgen. En via de set functie zetten we de huidige stand op 'uit'. De statische functie **buttonFunc** geeft de stand van de button door aan de bool 'on'. Via haakjes na de naam van de button kom je dus zijn huidige

stand te weten. Dit is enkel zinvol bij toggle buttons. Een gewone button heeft immers geen stand.

Time for Action

Voeg deze code toe aan een programma. Kies zelf een geschikte positie voor de button. Wanneer je tijdens de create functie de button zou inschakelen met `button.set(true)` dan zal de crosshair niet toch niet getoond worden. Zoek uit hoe dat komt en hoe je dat kan oplossen.

HOOFDSTUK 24

CheckBox

Een checkbox verschilt niet zoveel van een button in toggle modus. Welk van de twee je gebruikt, maakt enkel visueel een verschil. Net zoals alle gui classes is ook `CheckBox` afgeleid van `GuiObj`, dus je kan ook alle functies van die base class gebruiken.

Time for Action

Maak een window met enkele checkboxes. De eerste checkbox verbind je aan een functie die een bool 'showFPS' controleert. Wanneer die `true` is, dan toon je via de huidige fps op het scherm met behulp van de functie `Time.fps()`. Gebruik de andere checkboxes om enkele windows uit de vorige oefeningen te tonen en verbergen.

HOOFDSTUK 25

Slider

Een slider heeft een waarde tussen 0 en 1, afhankelijk van zijn positie. Je kan die waarde opvragen via de operator `()`. Voor een gewone slider wordt dat:

```
float value = mySlider();
```

Maar wanneer je slider een pointer is, dan moet je wel aangeven dat je niet de pointer maar het object zelf wil aanspreken:

```
float value = (*mySlider)();
```

Doe je dat dan vanuit een static functie die je koppelt aan de slider, dan wordt die variabele automatisch aangepast wanneer je de slider beweegt. Zo kan je het volgende schrijven:

```
class sliderWindow
{
private:
   GuiObjs objs;
   Window * window;
   Slider * speedSlider;
   float currentSpeed = 1;

   static void speedSliderFunction(sliderWindow & obj) {
      currentSpeed = (*obj.speedSlider)() * 5;
   }

public:
   void create()
   {
      objs.load( --- Drop Gui Object here --- );
```

```
        window = objs.findWindow("window");
        speedSlider = objs.findSlider("speedSlider");

        speedSlider.func(speedSliderFunction, T);
        Gui += objs;
    }
}
sliderWindow SliderWindow;
```

Note

Sliders geven altijd een waarde tussen 0 en 1. Je kan die schaal niet aanpassen, maar je kan wel het resultaat vermenigvuldigen tot de range die je nodig hebt. Wil je bijvoorbeeld een waarde tussen 10 en 50, dan schrijf je:

```
float value = 10 + slider() * 40;
```

Time for Action

Maak een window met drie sliders en een `Color`. De color stel je in het begin gelijk aan BLACK, maar de sliders wijzigen respectievelijk de r, g en b waarden van de color. Teken dan ook de achtergrond in deze kleur.

HOOFDSTUK 26

TextLine

TextLine is de enige gui class die je kan gebruiken om tekstinvoer van de gebruiker te krijgen. De visuele functies (grootte, positie, zichtbaarheid enzovoort) zijn identiek aan de andere classes die je zag. Maar de functie **func** werkt anders. Je kan daar op dezelfde manier een functie aan koppelen, maar waar bij een button of een checkbox de functie getriggerd wordt door een mouse click, zal een tekstline deze functie uitvoeren bij elke muisklik. Dat valt eenvoudig te demonstreren met de volgende code:

```
Str myText;

// Om dit voorbeeld kort te houden wordt de gui class verder niet
    uitgewerkt.
// Je ziet enkel de callback functie.

void MyTextLineFunc(guiClass & obj) {
  myText = (*obj.myTextLine)();
}

// .. en de Draw functie
void Draw() {
  D.clear(BLACK);
  D.text(0, 0, myText);
}
```

Als je de invoer enkel wil controleren op een bepaald moment, dan kan je een button gebruiken en pas dan de invoer van de TextLine controleren. Een voorbeeld daarvan vind je in hoofdstuk 21.4.

26.1 Tekst omzetten naar een getal

De inhoud van een TextLine is steeds een tekst, ook al bestaat die tekst uit cijfers. Als je die inhoud als een integer of float wil gebruiken, dan moet je die eerst converteren. Daarvoor bestaan er functies als **TextInt** en **textFlt**.

```
int   i = TextInt( (*obj.myTextLine)() );
float f = TextFlt( (*obj.myTextLine)() );
```

Let wel op: als je textline niet omgezet kan worden naar een getal, dan krijg je geen foutmelding. Het resultaat is dan steeds 0.

26.2 Andere handige functies

Wanneer je een tekst in een TextLine wil plaatsen, dan kan dat met de functie **set**. Die aanvaardt een string als argument:

```
Str purpose = "life";
int meaning = 42;

myTextLine .set(purpose);
myTextLine2.set(S +  42);
```

Met de functie **password** kan je sterretjes in plaats van letters tonen:

```
myTextLine.password(true);
```

Via de functie **clear** maak je een textline leeg:

```
myTextLine.clear();
```

Time for Action

TextLine heeft ook een functie **cursor** om de positie van de cursor op te vragen en aan te passen. Maak een programma met een tekstline waarin je een tekst plaatst. Maak ook functies **moveLeft** en **moveRight** die de cursor een positie naar links of naar rechts kunnen verplaatsen. In de **Update** functie van je class zorg je dat de cursor via F1 en F2 naar links en rechts verplaatst kan worden. Via F3 zet je de password modus aan of uit, en via F4 maak je het hele veld leeg.
Tot slot zoek je in de header file op hoe je tekst selecteert. Zorg er voor dat je via F5 de hele tekst selecteert, en via F6 de selectie ongedaan maakt.

HOOFDSTUK 27

Translations

In een ideale wereld spreekt iedereen dezelfde taal, maar zover zijn we helaas nog niet. Daarom zal je als vaak meerdere talen moeten ondersteunen in je software. In dit hoofdstuk maak je een eenvoudige translation manager.

> **NOTE**
>
> Voor een niet al te uitgebreid programma met enkel korte zinnen is deze aanpak zeker geschikt. Wanneer je een programma met duizenden zinnen of volledige alinea's tekst maakt, dan heb je meer geavanceerde code nodig.

Maak om te beginnen enkele gui objecten die je in deze oefening wil vertalen. Tot hier toe was het zelden nodig om ook gewone teksten in een applicatie van een naam te voorzien. Die tekst veranderde namelijk nooit. Nu we andere versies in verschillende talen willen tonen is dat natuurlijk wel nodig. Met behulp van de uitleg uit de vorige hoofstukken zal je er zeker in slagen om ook een class voor je gui objecten uit te werken.

Maak daarnaast ook een gui met twee buttons, zoals in afbeelding 27.1.

Figuur 27.1: Een Language Gui.

Bij deze gui hoort de volgende code:

```
class languageGui
{
private:
   GuiObjs obj;
   Window * window;
   Button * buttonDutch;
   Button * buttonEnglish;

public:
   void create()
   {
      obj.load(=== drop gui object here ===);
      window        = obj.findWindow("window"       );
      buttonDutch   = obj.findButton("buttonDutch"  );
      buttonEnglish = obj.findButton("buttonEnglish");

      Gui += obj;
   }
}
languageGui LanguageGui;
```

Vergeet niet de create functie uit de voeren in `Init()`.

Maak ook alvast een class `translationManager`. Die ziet er voorlopig zo uit.

```
class translationManager {
private:
  LANG_TYPE language = LANG_ENGLISH;

public:
  void setLanguage(LANG_TYPE language) {
    T.language = language;
  }

  C Str & translate(C Str & text) {
    return text;
  }
}
translationManager TM;
```

In deze class kan je een ingestelde taal onthouden via de enum `LANG_TYPE`. Daarnaast zijn er functies om de taal in te stellen en een string te vertalen. Die laatste functie heeft voorlopig gewoon de input als resultaat. We werken deze class later verder uit, maar kunnen ze nu al gebruiken om de basis van het programma uit te werken.

Nu de translation class bestaat, kunnen we ook de Gui classes afwerken. In de class `languageGui` zullen we callback functies voor de buttons schrijven:

```
static void funcDutch(ptr)
{
  TM.setLanguage(LANG_DUTCH);
}

static void funcEnglish(ptr)
{
  TM.setLanguage(LANG_ENGLISH);
}
```

Vergeet niet deze functies aan de respectievelijke buttons te koppelen, en voeg daarna ook de volgende functie aan de class **languageGui** toe:

```
void translate()
{
  window       .setTitle(TM.translate("Choose Language"));
  buttonDutch  .setText (TM.translate("Dutch"          ));
  buttonEnglish.setText (TM.translate("English"        ));
}
```

De bovenstaande functie maakt al duidelijk hoe we tewerk zullen gaan. We gebruiken Engelse teksten om de content van de Gui elementen in te stellen. Maar die tekst wordt eerst door de translation manager gestuurd. Daar kunnen we dan als resultaat de vertaling geven.

27.1 The translationManager Class

Tijd om aan het echte werk te beginnen: de translation manager. Deze heeft een container nodig om vertaalde strings op te slaan. Maar een container is niet erg geschikt voor dit doel.

Typisch voor deze class is dat we steeds een Engelse tekst hebben en die willen vervangen door een alternatief in een andere taal. In een taal zoals php zou je dat op de volgende manier doen:

```
translations["Life"] = "Leven";
```

...wat toelaat om achteraf de vertaling te bekomen via:

```
void printText(string s) {
   echo translations[s];
}
```

Zoiets kan niet in C++. De index van een container moet steeds een integer zijn. Hoe pak je dat dan aan?

27.2 Map

De held van de dag is de class `Map`. Een map is perfect voor waarden die via een één op één relatie met mekaar gelinkt zijn, zoals een vertaling. Waar we een `Memc` definiëren met enkel zijn data type:

```
Memc<Str> strings;
```

vermelden we in `Map` ook het type van de index:

```
Map<Str, Str> translations;
```

Het eerste argument noemen we KEY, het tweede DATA. Niet enkel strings kunnen een key zijn, maar om het even welke class. En dat schept een nieuw probleem: om efficiënt te zoeken in een Map, moet die intern gesorteerd worden. En om dat te doen, moet `Map` elementen kunnen vergelijken. `Map` moet kunnen beslissen of een key groter, kleiner of gelijk is aan een andere key.

Je zal zelf een functie moeten schrijven die elementen vergelijkt. De definitie van `Map` toont je hoe. Als eerste argument van de constructor map staat er:

```
Int compare(C KEY &a, C KEY &b)
```

Je moet dus een functie schrijven die twee argumenten van hetzelfde type als je key aanvaardt, en die een integer als resultaat heeft. Esenthel verwacht dat dat resultaat -1, 0 of 1 is.

- -1: het eerste argument is kleiner dan het tweede.
- 0: beide argumenten zijn gelijk.
- 1: het eerste argument is groter dan het tweede.

Aangezien we in deze map een key van het type `Str` hebben, is de oplossing heel eenvoudig. Er bestaat namelijk al een functie om strings op deze manier met mekaar te vergelijken. We kunnen de gevraagde functie dus zo schrijven:

```
static int mapCompare(C Str &a, C Str &b)
{
  return CompareCI(a, b);
}
```

Indien je een verschil wil maken tussen hoofdletters en kleine letters, kan je ook `CompareCS` gebruiken.

> **Note**
>
> Merk op dat dit een static functie is, net zoals een callback die je aan een button doorgeeft.

Nu deze functie bestaat, kan je de definitie van je Map verbeteren:

```
Map<Str, Str> textMap(mapCompare);
```

27.3 Load a translation

Nu de map bestaat, maken we een functie om een bestand te laden en de inhoud in de map op te slaan.

We kiezen er in dit voorbeeld voor om te werken met een tekstbestand. In dat bestand staat telkens een Engelse tekst, gevolgd door een vertaling. De tekst en de vertaling worden gescheiden door een dubbele punt:

```
Choose Language: Kies je taal
Dutch: Nederlands
English: Engels
```

Je maakt nu in de class `translationManager` de volgende functie:

```
void loadLanguage() {
  textMap.clear();
  if(language == LANG_ENGLISH) return;
}
```

Daarmee wijs je bij het laden van een vertaling eerst de bestaande vertaling. Wanneer de nieuwe taal Engels is, dan stoppen we de functie omdat er geen vertaling nodig is.

In het andere geval moeten we een bestand laden. Dit bestand zou uit een config directory kunnen komen, maar in dit geval importeren de vertalingen in het project. Dat kan zo:

1. Maak een bestand ‘dutch.txt’ met de inhoud hierboven.
2. Sleep het bestand in de editor.
3. Sleep het bestand naar de read functie, zoals in de code hieronder.

Voeg aan de bovenstaande functie de volgende code toe:

```
FileText ft;
switch(language) {
  case LANG_DUTCH:
  {
      ft.read(=== drop file here ===);
      break;
  }
}
```

Je kan voor elke vertaling een bestand en een case instructie toevoegen. De functie **read** zorgt er voor dat het object **ft** de inhoud van het bestand bevat.

> **Note**
>
> Telkens je content toevoegt aan het oorspronkelijke bestand, kan je in de editor rechts klikken op het bestand en 'Reload' kiezen.

Tot slot moet elke regel in dit bestand gelezen worden, waarbij we de inhoud toevoegen aan de **Map**. We gebruiken daarbij een container 'parts' en de functie **Split**. De Split functie verdeelt elke regel in twee delen en slaat die op in 'parts'. Bijgevolg bevat **parts[0]** de key en **parts[1]** de data.

```
while(!ft.end())
{
  Str line = ft.getLine();
  Mems<Str> parts;
  Split(parts, line, ':');

  (*textMap.get(parts[0])) = parts[1];
}
```

In de laatste regel gebruiken we de **get** functie van **Map**, met als eerste argument de key. Die key bestaat nog niet, dus **Map** maakt een nieuw element aan. De data ken je toe aan dat element.

Nu is de nieuwe vertaling wel geladen, maar de Gui zal die niet vanzelf gebruiken. Wel hebben we in de gui class een functie translate voorzien. Het is dus nodig om na het laden van een vertaling alle Gui's te updaten. We maken daar een afzonderlijke functie voor:

```
void updateGuis()
{
  StatsGui.translate();
  // add other Gui's that must be translated
}
```

En nu kan je de functie **setLanguage** vervolledigen:

```
void setLanguage(LANG_TYPE language)
{
  T.language = language;
  loadLanguage();
  updateGuis();
}
```

Wat nu nog ontbreekt is de werkelijke vertaling. Je past de functie **translate** aan en voegt de volgende code toe:

```
C Str & translate(C Str & text)
{
  Str * result = textMap.find(text));
  if(result != null) return *result;

  return text;
}
```

Deel V

Data

HOOFDSTUK 28

Data opslaan in een bestand

Je kan data op verschillende manieren bewaren. Tekstbestanden zijn eenvoudig om te lezen, maar minder efficient om data in op te slaan. Je kan data ook binair opslaan, wat minder plaats in beslag neemt. Maar die bestanden kan je dan weer niet eenvoudig controleren met een tekst editor.

28.1 Locaties

Wat je ook kiest, je zal eerst moeten bepalen waar je data opslaat. Het is geen goed idee om zelf het hele path naar je bestand vast te leggen in code. Stel je voor dat je beslist om een config bestand op te slaan op `C:\myGame\gameData\config.txt`. Dat is nogal vervelend als de gebruiker je app verplaatst naar een andere directory of schijf. De gebruiker verwacht dat de data op zijn minst in een subdirectory van je game zit. Het is zelfs mogelijk dat een computer geen C drive heeft!

Esenthel laat je toe om bepaalde paden op te vragen. Zo kan je de 'documents' folder van de betreffende computer en gebruiker opvragen, zonder dat je zelf het hele path moet kennen:

```
Str docPath = SystemPath(SP_DOCUMENTS);
```

Als je daarna een bestand wil opslaan in die 'documents' folder, dan kan je het path daarvoor als volgt samenstellen:

```
Str documentPath = S + docPath + "/myfile.txt";
```

Note

Merk op dat je een forward slash (/) gebruikt om folders te schrijven. In Windows gebruik je normaal een backslash, maar Esenthel zal die zelf omzetten. De bedoeling is dat je met deze engine een app kan maken die ook op andere systemen –zoals Linux, mac of Android– werkt. Andere systemen gebruiken steeds een forward slash.

Naast `SP_DOCUMENTS` kan je nog andere systeempaden opvragen:

- **SP_DESKTOP** verwijst meestal naar `C:/Users/*/Desktop`.
- **SP_APP_DATA** verwijst naar `C:/Users/*/AppData/Roaming`. Op mobiles is het het path waar een applicatie toelating heeft om data te bewaren.
- **SP_APP_DATA_EXTERNAL** verwijst ook naar `C:/Users/*/AppData/Roaming`. Maar indien een mobile system een externe SD kaart heeft, dan zal dat path gebruikt worden.
- **SP_ALL_APP_DATA** verwijst meestal naar `C:/ProgramData`

Een volledig overzicht van de mogelijkheden vind je in de header file `Misc $\Rightarrow$ Misc` onder de enum declaratie 'SYSTEM_PATH'. Niet alle paden zijn beschikbaar op elk systeem. Je kan bijvoorbeeld niet naar de desktop schrijven op een mobiel platform.

Time for Action

Controleer waar de bovenstaande paden op jouw computer naar verwijzen. Maak een eenvoudig testprogramma dat de resulterende string voor elk van deze paden op het scherm toont. *(Project Textfiles, ex. 01)*

28.2 TextData

De class `TextData` laat je toe op een eenvoudige manier data als text op te slaan. Deze class is vooral voor het bewaren van config files erg handig.

28.2.1 Content bewaren

Data opslaan is eenvoudig. Voor elke variabele die je wil bewaren maak je een 'node'. Je kent aan die node een waarde toe, en vervolgens sla je het bestand op.

```
TextData data;
data.getNode("name").setValue("John Doe");
data.getNode("highscore").setValue("100");
data.save(S + SystemPath(SP_DOCUMENTS) + "/config.txt");
```

De functie **getNode** zal zoeken naar de naam van de node. Indien er nog geen node bestaat met die naam, dan wordt er een nieuwe node gemaakt. De functie **setValue** geeft de nieuwe node een waarde.

Het bestand zal de volgende inhoud bevatten:

```
name=`John Doe`
highscore=100
```

Time for Action

Maak een programma met de variabelen **name**, **address**, en **age**. Geef de variabelen een default waarde en sla ze op in een bestand. Controleer of het bestand de juiste waarden bevat.

28.2.2 Een bestand laden

Een bestand laden doe je zo:

```
TextData data;
Str docPath = SystemPath(SP_DOCUMENTS);
data.load(S + docPath + "\config.txt");
```

Indien het bestand bestaat, zal het via de bovenstaande code ingelezen worden. *(De functie* ***load*** *geeft ook een bool als resultaat die aangeeft of dit gelukt is. In een echte applicatie controleer je best die waarde.)*

28.2.3 Content lezen

Om de inhoud van een bestand te lezen, moet je het eerst laden. Daarna kan je je zoeken naar een bepaalde node, en de inhoud van die node toewijzen aan een variabele. Omdat het altijd mogelijk is dat een node niet bestaat, neem je twee voorzorgsmaatregelen:

1. Geef je variabele een default waarde. Wanneer het niet lukt om de node te laden, blijft deze waarde gewoon behouden.
2. Haal enkel de waarde uit een node wanneer de node bestaat. Data uit een onbestaande node halen crashed je programma.

```
// config values
Str name = "new player";
int highScore = 0;
int credits = 100;

TextData data;
Str docPath = SystemPath(SP_DOCUMENTS);
if(data.load(S + docPath + "/config.txt")) {
   TextNode * node;

   // try finding the name
   node = data.findNode("name");
   if(node) {
      name = node.asText();
   }

   // try finding the highscore
   node = data.findNode("highscore");
   if(node) {
      highScore = node.asInt();
   }
}
```

Note

Wanneer je een node bewaart, dan kan je er eender welk type aan toekennen met de functie `setValue`. Maar bij het laden van een node moet je wel een onderscheid maken naargelang het type: je gebruikt `asInt` voor een integer, en `asText` voor een string. Meer mogelijkheden vind je in de header file File ⇒ Xml. (`TextNode` is gebaseerd op `TextParam`.)

Time for Action

Pas het bestand uit de vorige oefening aan via een texteditor en laadt het terug in je programma. Toon de waarden van de variabelen op het scherm. Controleer of ze correct zijn. *(Project Textfiles, ex. 02)*

28.2.4 Een class voor een config file

Tot slot krijg je hier nog een volledig uitgewerkt voorbeeld waarop je kan voortbouwen als je je eigen config file wil maken. De bedoeling is dat je de `load()` functie uitvoert bij het starten van je programma en de `save()` functie bij het afsluiten. Daartussen kan je de variabelen wijzigen via de voorziene functies. *(Project Textfiles, ex. 03)*

```
class configFile {
private:
  Str name      = "new player";
  int highscore = 0  ;
  int credits   = 100;

public:
  Str getName     () { return name     ; }
  int getHighscore() { return highscore; }
  int getCredits  () { return credits  ; }

  void setName     (C Str & name   ) { T.name      = name   ; }
  void setHighscore(  int   score  ) { T.highscore = score  ; }
  void setCredits  (  int   credits) { T.credits   = credits; }

  void load() {
    TextData data;
    if(data.load(S + SystemPath(SP_APP_DATA) + "/mygame/config.txt")) {
      TextNode * node;

      if(node = data.findNode("name"     )) name      = node.asText();
      if(node = data.findNode("highscore")) highscore = node.asInt ();
      if(node = data.findNode("credits"  )) credits   = node.asInt ();
    }
  }

  void save() {
    TextData data;
    data.getNode("name"     ).setValue(name     );
    data.getNode("highscore").setValue(highscore);
    data.getNode("credits"  ).setValue(credits  );

    data.save(S + SystemPath(SP_APP_DATA) + "/mygame/config.txt");
  }
}
configFile ConfigFile;
```

28.3 Binary Files

De TextData files hierboven zijn vooral handig voor configuratiebestanden of andere bestanden die je ook wil kunnen openen in een tekst editor. Ze zijn niet de meest efficiënte manier om data op te slaan. Met de class `File` kan het veel compacter. Met `File` sla je enkel de waarde van een variabele op, zonder een ID om die terug te vinden:

```
File f;                    // create file object
f.write("file.dat"    ); // start writing to a file
f.putInt(128          ); // write an integer
f.putFlt(3.14         ); // write a float
f.putStr("Hello world"); // schrijf een tekst naar het bestand
```

Wanneer je nadien de data terug wil inlezen, dan doe je dat in dezelfde volgorde. De functie **readTry** zorgt er voor dat de waarden enkel gelezen worden wanneer het bestand bestaat. Er bestaat ook de functie **read**, maar die crashed het programma wanneer een ongeldig bestand wordt gelezen.

```
File f;
if (f.readTry("file.dat")) {
  int   i    = f.getInt();
  float pi   = f.getFlt();
  Str   text = f.getStr();
}
```

Time for Action

Maak een gui class met een **TextLine**, een **Slider**, een **CheckBox** en twee **Button**'s. De eerste button bewaart de huidige waarde van de gui elementen in een bestand. De tweede button laadt dat bestand opnieuw en kent de opgeslagen waarden toe aan de gui elementen. Wijzig de gui na het opslaan en controleer of na het laden alle elementen terug de juiste waarde hebben.

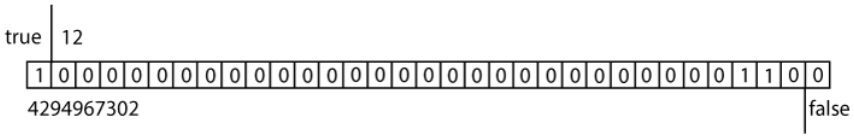
28.3.1 Data Order

Wat als je data in een andere volgorde leest en schrijft? Bekijk even de volgende code:

```
void save() {
  File f;
  f.write("trouble.dat");
  f.putBool(true);
  f.putInt (12  );
}
```

```
void load() {
  File f;
  f.read("trouble.dat");
  int i = f.getInt ();
  int b = f.getBool();
}
```

Bij het laden en opslaan van de data werd een andere volgorde gebruikt. De compiler zal deze fout niet opmerken. Een bool variabele is precies 1 bit groot, maar een integer bestaat uit 32 bits. Er worden dus 33 bits opgeslagen: 1 + 32. Bij het laden wordt eerst de integer gelezen, maar vooraan in het bestand staat de bool variabele. Het programma laadt dus 32 bits, maar dat zijn de bool en 31 bits van de integer. Daarna wordt de laatste bit van de integer gelezen als bool. Figuur 28.1 illustreert dit.



Figuur 28.1: bits in een bestand

> **Note**
>
> De volgorde van de variabelen bij het lezen en het schrijven van een `File` moet exact dezelfde zijn. Als je daar fouten tegen maakt, dan kan je programma crashen of zich anders gedragen dan je zou verwachten.

En er is nog een andere, minder opvallende manier waarop dit mis kan gaan. C++ behandelt de argumenten van een functie van rechts naar links (net zoals de meeste andere programmeertalen, overigens). Als voorbeeld een stukje code.

```
float r = 0.1;
float x = 0.3;
float y = 0.4;

Circle c;
c.set(r, x, y);
```

Argumenten worden gelezen van rechts naar links. Dat betekent dat de instructie `pos2.set` op de volgende manier wordt uitgevoerd:

1. Geef de waarde van y aan het derde argument van set.
2. Geef de waarde van x aan het tweede argument van set.
3. Geef de waarde van r aan het eerste argument van set.

4. Voer de functie set uit.

Waarom is deze volgorde belangrijk? Wel, in het bovenstaande voorbeeld helemaal niet. De data die we gebruiken is niet sequentieel, ze moet niet in de juiste volgorde gelezen worden. We kunnen op elk moment de waarden van x, y of r raadplegen. Maar wat als we data uit een bestand halen?

```
float r = 0.1;
float x = 0.3;
float y = 0.4;

// Put data in an imaginary file
f.putFlt(r);
f.putFlt(x);
f.putFlt(y);

// ... after loading this data
Circle c;
c.set(f.getFlt(), f.getFlt(), f.getFlt());
```

De eerste float in het bestand bevat de waarde voor r. De tweede de waarde voor x en de derde voor y. Dit voorbeeld lijkt op het eerste zicht te werken, maar omdat functieargumenten van rechts naar links verwerkt worden, komt de eerste float rechts terecht. Op de plaats waar we y verwachten! Je lost dit op door er voor te zorgen dat de inhoud stap voor stap gelezen wordt. In dit geval kan dat door vooral de functie set niet te gebruiken.

```
Circle c;
c.r     = f.getFlt();
c.pos.x = f.getFlt();
c.pos.y = f.getFlt();
```

Time for Action

Maak een kort programma dat deze fout demonstreert. (Je gebruikt dus opzettelijk de foute manier.) *(Project Textfiles, ex. 04)*

28.3.2 Objecten Opslaan

Je kan ook meerdere objecten opslaan in hetzelfde bestand. Om dit te doen, geef je een `File` object door als referentie. Zo bijvoorbeeld in de volgende class:

```
class resource {
  int type  ;
  Vec pos   ;
```

```
  int amount;

  void create(int type, int amount, C Vec & pos) {
    T.type   = type  ;
    T.pos    = pos   ;
    T.amount = amount;
  }

  // ... other functions are omitted to keep this example short ...

  void save(File & f) {
    f.putInt(type  );
    f.putFlt(pos.x ).putFlt(pos.y).putFlt(pos.z);
    f.putInt(amount);
  }

  void load(File & f) {
    type   = f.getInt();
    pos.x  = f.getFlt();
    pos.y  = f.getFlt();
    pos.z  = f.getFlt();
    amount = f.getInt();
  }
}
```

Vervolgens voorzie je een manager class om die resources te beheren. Naast functies om ze toe te voegen, te verwijderen, op het scherm te tekenen etc., voeg je functies toe om alle bestaande resources op te slaan of te laden:

```
class resourceManager {
  Memx<resource> resources;

  void save() {
    File f;
    f.write("resources.dat");

    f.putInt(resources.elms());
    FREPA(resources) {
      resources[i].save(f);
    }
  }

  void load() {
    File f;
    if(f.readTry("resources.dat")) {

      int elms = f.getInt();
      for(int i = 0; i < elms; i++) {
        resources.New().load(f);
      }
    }
  }
```

```
}

resourceManager RM;
```

Je kan zelfs alle data voor je programma opslaan in hetzelfde bestand, door ook in de manager class de File als referentie door te geven in plaats van ze ter plaatse te definiëren. In dat geval zou je bijvoorbeeld op de volgende manier je data opslaan:

```
void save() {
  File f;
  f.write("allData.data");
  Config.save(f);
  RM    .save(f);
}

void load() {
  File f;
  f.load("allData.data");
  Config.load(f);
  RM    .load(f);
}
```

Time for Action

Maak een class voor een cirkel die ook zijn kleur kan onthouden. Voorzie een manager class die cirkels kan bevatten. Elke klik met de muis voegt een cirkel toe aan de manager, op de huidige positie van de muis en in een willekeurige kleur. (Herlees hoofdstuk 10.5 indien nodig.)
Zorg dat bij het afsluiten van het programma alle cirkels bewaard worden. Bij de start van het programma laadt je alle cirkels van de vorige sessie. *(Project Textfiles, ex. 05)*

HOOFDSTUK 29

Databases

29.1 Database Servers

Data kan je niet enkel in gewone bestanden opslaan, je kan ook een database gebruiken. De meest voorkomende standaard voor het werken met databases is SQL (sequel). Esenthel Engine ondersteunt 3 verschillende databases: Microsoft SQL, MySQL en SQLite. Elk van deze databases heeft voor- en nadelen.

- **Microsoft SQL:** De database server is zeer performant, maar je hebt wel een Windows OS nodig. Tijdens de ontwikkeling zal dit waarschijnlijk geen probleem zijn, maar wanneer je een server voor je programma laat hosten dan is een Windows host duurder.
- **MySQL:** Dit is waarschijnlijk de meest gebruikte SQL server. MySQL draait op zowat elk OS, is gratis en open source. De installatie en het onderhoud zijn niet erg moeilijk, en de performantie is OK.
- **SQLite:** SQLite gebruikt een gewoon bestand als database. Er moet dus geen server geïnstalleerd worden. Vooral tijdens de ontwikkelfase kan dat een voordeel zijn, maar het zorgt er wel voor dat SQLite heel wat trager is dan een 'echte' database server. SQLite kan ook een goede oplossing zijn als je aan de client side data wil opslaan in een database, zonder dat je daarvoor de gebruiker verplicht om een database server te installeren.

Een belangrijk voordeel van de database class in Esenthel is dat je makkelijk van database kan wisselen. Je kan dus tijdens het ontwikkelen voor SQLite kiezen en pas achteraf overschakelen op een MySQL of MS SQL database.

29.2 Een Database Gebruiken

29.2.1 De Verbinding

Alvorens je een database kan gebruiken, moet je een verbinding maken. De `SQL` class laat je toe een verbinding te maken met elk soort database. In het geval van een SQLite database is de naam van een bestand voldoende, maar bij een echte server zal je de host, de naam van de database, een gebruiker en een wachtwoord moeten voorzien. MS SQL en MySQL laten je ook nog toe een pointer naar een string mee te geven. Indien de verbinding niet lukt, zal die string de foutmelding van de database server bevatten.

```
SQL sql;
Str messages;

// voor Microsoft SQL
if (!sql.connectMSSQL("LocalHost\\SQLExpress", "test_db", "", "",
    &messages)) {
  // verbinding mislukt
  Exit(S + "Database error: " + messages);
}

// voor MySQL
if (!sql.connectMySQL("localhost", "test_db", "user", "password",
    &messages)) {
  // verbinding mislukt
  Exit(S + "Database error: " + messages);
}

// voor SQLite
if (!sql.connectSQLite("test.db")) {
  Exit(S + "Database error");
}
```

Time for Action

Maak een class `database` met een functie `create`. In deze functie maak je een verbinding met een SQLite database met de naam 'mydata.db'. Je voegt een je programma een object van de class toe en je voert de functie create uit. Na het uitvoeren van je programma ga je op zoek naar het bestand 'mydata.db'. Op het bestand met een SQLite browser, te downloaden via http://sqlitebrowser.org . (Je zal een foutmelding krijgen omdat de database leeg is.)

29.2.2 Een tabel maken

Je programma kan bestaande tabellen gebruiken, of je kan met een andere tool je de tabellen in je database aanmaken. Een derde optie is dat je de database aanmaakt op het moment dat je je programma start. Ook dit kan weer handig zijn tijdens de ontwikkelfase: je controleert of een tabel al bestaat, en indien niet dan laat je je programma die tabel maken. Eventueel kan je er ook via je code wat data in zetten. Op die manier kan je je programma eenvoudig testen. En als je de structuur van je data wil wijzigen, dan verwijder je gewoon de hele database zodat die opnieuw aangemaakt wil worden.

Zoals je weet bestaat een tabel uit kolommen met een naam en een type. Bovendien moet in elke tabel één kolom de primary key zijn, eventueel met auto-increment. Je kan ook voor elke kolom een default waarde instellen, maar dat is niet verplicht.

```
if(!sql.existsTable("accounts")) {
  // maak een tabel voor accounts

  Memc<SQLColumn> columns;
  columns.New().set("ID"      , SDT_INT    ).mode = SQLColumn.PRIMARY_AUTO;
  columns.New().set("name"    , SDT_STR, 32); // een string met maximaal
     32 tekens
  columns.New().set("password", SDT_STR, 32);
  columns.New().set("active"  , SDT_BOOL   );
  columns.New().set("score"   , SDT_INT    ).default_val="0";

  if(!sql.createTable("accounts", columns, &messages)) {
    Exit(S + "Can't create table for accounts: \n" + messages);
  }

  // voeg test data toe
  SQLValues values;
  values.New("name"    , "freddy"    );
  values.New("password", "kamerplant");
  values.New("active"  , true        );

  if(!sql.newRow("accounts", values, &messages)) {
    Exit(S + "Can't add data to table accounts: \n" + messages);
  }
}
```

Time for Action

Breidt de functie `create` uit de vorige oefening uit met de bovenstaande code. Voer het programma opnieuw uit en bekijk de database achteraf opnieuw via de SQLite browser. Voeg daarna een extra kolom 'lives' toe. Controleer of deze kolom na het uitvoeren ook in de tabel staat.

29.2.3 Data Lezen

Als je eenvoudig alle data uit een tabel wil lezen, dan kan dat met de functie `getAllRows()`. Je wil die data dan waarschijnlijk wel ergens in het geheugen houden, dus daar gebruik je best een class en een memory container voor. Met de table "accounts" als voorbeeld zou je een class `account` kunnen maken om een rij uit de database in het geheugen te zetten.

```
class account {
  // Om het voorbeeld kort te houden zijn alle variabelen public.
  // In een echt programma kan je meestal beter set en get functies
     gebruiken.
  int  ID    ;
  Str  name  ;
  bool active;
  int  score ;
}
```

De functie `getAllRows()` gebruik je eenvoudigweg met de naam van een tabel. Na het uitvoeren van de functie bevat het `sql` object alle rijen van de gevraagde tabel. Met de functie `getNextRow()` kan je dan een rij met gegevens opvragen totdat er geen volgende rij meer is. Vervolgens kan je de functie `getCol()` gebruiken om de waarde van een kolom te verkrijgen.

> **Note**
>
> Vraag de kolommen van een rij steeds op in de volgorde waarin ze in de tabel staan. Je kan wel kolommen overslaan, maar niet terug gaan naar een vorige kolom.

```
Memc<account> accounts;
sql.getAllRows("accounts");

for( ;sql.getNextRow(); ) {
  account & a = accounts.New();

  sql.getCol(0, a.ID    );
  sql.getCol(1, a.name  );
  // skip loading the password in column 2
  sql.getCol(3, a.active);
  sql.getCol(4, a.score );
}
```

Time for Action

1. Maak een class `account`.
2. Maak een class `accountManager` waarin je een container voor alle accounts voorziet.
3. Voorzie een functie `create` die alle records uit de database leest en toevoegt aan de container.
4. Voeg aan de class een integer 'currentAccount' toe die je op nul zet.
5. Via de functies `next` en `previous` kan je het huidige account wijzigen. Zorg er wel voor dat dit niet kleiner dan nul kan worden, en ook niet groter dan het aantal accounts.
6. Voorzie een functie draw die de inhoud van het huidige account op het scherm toont.

Je wil niet steeds alle data uit een tabel in het geheugen laden. Dikwijls ben je maar geïnteresseerd in een of in enkele records. In dat geval gebruik je de functie `getRows()`. Als argument kan je bij deze functie een voorwaarde opgeven, net zoals je dat in een SQL statement zou doen:

```
sql.GetRows("accounts", "name='freddy'");
if(sql.getNextRow()) {
  // load column data
}
```

Dikwijls zal je dergelijke code in een functie gebruiken, en een bepaalde waarde als resultaat geven. Stel je voor dat we van een account de score willen weten. De functie zou er dan zo kunnen uitzien:

```
int getScore(C Str & name) {
  int result = 0;
  sql.GetRows("accounts", S+ "name='" + name + "'");
  if(sql.getNextRow()) {
    sql.getCol(4, result);
  }
  return result;
}
```

Note

Bij het opstellen van een voorwaarde moet elke waarde tussen enkele quotes staan. Als je dan ook nog variabelen toevoegt aan de vergelijking, dan moet je goed uitkijken dat de quotes op de juiste plaats staan.

Maar je zou ook een hele account als referentie kunnen doorgeven. Dat is vooral bruikbaar in het geval je een volledige record wil laden. We zouden dan eerst de class `account` uitbreiden:

```
class account {
  int  ID      ;
  Str  name    ;
  Str  password;
  bool active  ;
  int  score   ;

  bool load(C Str & name) {
    T.name = name;
    return loadAccount(T);
  }
}
```

Via de functie load kunnen we een account uit de database halen door een naam mee te geven. De return waarde laat ons ook weten of dat gelukt is. De functie `loadAccount()` kunnen we dan zo uitwerken:

```
bool loadAccount(account & a) {
  sql.GetRows("accounts", S+"name='"+a.name+"'");
  if(sql.getNextRow()) {
    sql.getCol(0, a.ID);
    sql.getCol(2, a.password);
    sql.getCol(3, a.active);
    sql.getCol(4, a.score);
    return true;
  }
  return false;
}
```

Time for Action

1. Voeg aan de class `database` een de functie `loadAccount` toe.
2. Maak een Gui class met een TextLine waarin we een naam kunnen schrijven. Voeg ook een Button en Text objects toe om de ID, wachtwoord en score te tonen.
3. Voorzie een functie die, wanneer de gebruiker op de button klikt, zoekt naar een account met de naam in TextLine en dat account toont in de gui.

29.2.4 Records tellen

Het gebeurt dat je niet in de inhoud van een record geïnteresseerd bent, maar enkel wil weten of een record bestaat. Wil je eenvoudigweg weten hoeveel records een tabel bevat, dan gebruik je de functie `getAllRowsNum()`.

```
int accountsInDB() {
  return sql.getAllRowsNum("accounts");
}
```

Time for Action

Voeg aan de class `database` een functie toe die telt hoeveel accounts er zijn. Toon dit ergens op je scherm.

Vaker wil je het aantal accounts dat aan een voorwaarde voldoet. Bijvoorbeeld het aantal actieve accounts. Dan gebruik je de functie `getRowsNum`, die weer een conditie als argument heeft, net zoals `getRows()`.

```
int activeAccountsInDB() {
  return sql.getRowsNum("accounts", "active='true'");
}
```

Met zo'n voorwaarde kan je ook te weten komen of een combinatie van gegevens bestaat. Zo kan je bijvoorbeeld controleren of een wachtwoord juist is.

```
bool validate(C Str & name, C Str & password) {
  int count = sql.getRowsNum("accounts", S + "name='" + name + "' AND
      password='" + password + "'");
  return (count > 0);
}
```

Time for Action

Maak een login gui waarin de gebruiker een naam en een wachtwoord kan ingeven. Zorg (met de bovenstaande code als voorbeeld) dat je kan kontroleren of de combinatie correct is en toon dat op het scherm.

29.2.5 Data Opslaan

Je zal vrijwel nooit alle gegevens in een database willen overschrijven. Wel wil je regelmatig een record die je aangepast hebt, terug opslaan. Dat kan zo:

```
void save(account & a) {
  SQLValues values;
  values.New("name", a.name);
  values.New("password", a.password);
  values.New("active", a.active);
  values.New("score", a.score);
```

```
  Str messages;
  if(!sql.setRow("accounts", S + "ID='" + a.ID + "'", values, &messages)) {
    Exit(S + "Error saving account: \n" + messages);
  }
}
```

Je hoeft ook niet steeds alle data op te slaan. Enkel de waarden die je in **SQLValues** opneemt, worden aangepast. Zo kan je bijvoorbeeld enkel het wachtwoord opslaan.

```
void savePassword(account & a) {
  SQLValues values;
  values.New("password", a.password);
  sql.setRow("accounts", S + "ID='" + a.ID + "'", values);
}
```

Note

Pas nooit de primary key van een record aan!

Time for Action

Voeg een Gui toe waarmee de gebruiker zijn wachtwoord kan aanpassen.

29.2.6 Data Verwijderen

Tot slot zal je ook records willen verwijderen die je niet meer nodig hebt. Daar kan je de functie **delRow** voor gebruiken.

```
void removeAccount(int ID) {
  sql.delRow("accounts", S + "ID='" + ID + "'");
}
```

HOOFDSTUK 30

Over netwerk applicaties

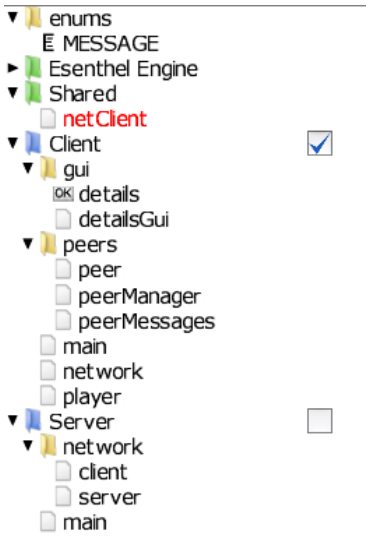
Tegenwoordig heeft bijna elke applicatie wel ergens een netwerk nodig. Bij online games is dat evident, maar ook andere toepassingen houden steeds meer info bij in *'the cloud'*. *(Die cloud is niets meer dan een fancy woord om aan te duiden dat je informatie op een server opslaat.)* Ook wordt een online component dikwijls gebruikt om een licentie te controleren.

Toch is een netwerk applicatie een pak moeilijker te ontwikkelen dan een gewone, stand-alone applicatie. Enkele redenen daarvoor:

- ✓ Je schrijf niet één, maar twee programma's. Er moet namelijk ook een server geschreven worden.
- ✓ Data die je verstuurt over het netwerk bestaat enkel uit bits. Zowel de client als de server moeten die op dezelfde manier interpreteren.
- ✓ Een netwerk kan traag en onbetrouwbaar zijn. Je software moet dat zo goed mogelijk opvangen.
- ✓ Mobile apps hebben dikwijls weinig bandbreedte. Je mag niet meer data versturen dan strikt noodzakelijk is.

Het is dan ook erg belangrijk dat je het overzicht bewaart bij het onwikkelen van een netwerk applicatie. In de volgende hoofdstukken zullen we de delen van zo'n programma bestuderen. Hierin staan verschillende technieken beschreven die je helpen dat overzicht te bewaren. In ieder geval is het belangrijk om je project goed te structureren. In figuur 30.1 zie je een file tree van het voorbeeldproject.

Je ziet dat zelfs een eenvoudig project al snel uit heel wat files bestaat. Je zet die niet allemaal in dezelfde map. Evident is dat we twee applicaties hebben: client en server. Maar het is zeker



Figuur 30.1: Structuur van een client-server project.

niet overbodig om ook binnen die applicaties folders te maken voor code die samen hoort. Zo heeft de client een folder voor alle gui elementen *(zowel layout als code)* en een folder voor code i.v.m. peers *(dat zijn andere spelers op het netwerk)*. We plaatsen dus ook de editor componenten niet zomaar in de root van het project. Immers, de server heeft dat gui element 'details' helemaal niet nodig. Bij een verdere uitwerking zal de server waarschijnlijk zijn eigen gui elementen nodig hebben, waar de client dan weer niets aan heeft.

Daarnaast zie je ook een folder 'enums'. Deze folder is wél beschikbaar voor elke app. We gebruiken enumeraties om berichten door te geven tussen de client en de server. Door ze in de root van het project te plaatsen zijn we zeker dat beide applicaties dezelfde lijst gebruiken.

Ook is er een library (groene folder) met de naam 'Shared'. In deze folder plaatsen we alle code die zowel bij de client als de server hoort.

Time for Action

Maak een nieuw project aan. Daarin maak je alvast alle folders en bestanden zoals in de afbeelding hierboven.

30.1 Messages

Data verzenden over het netwerk doe je via `Files`. Denk hierbij niet aan grote bestanden zoals op een harde schijf: elk bericht dat je verstuurt over het netwerk is in feite in klein bestandje.

Aangezien je bij een file dient te vermelden hoe je het wil noemen, voordat je er data in kan zetten, moet je bij een netwerkfile aangeven dat het enkel om het bestand in het geheugen gaat. En omdat de ontvanger van het bericht moet weten over wat voor bericht het gaat, stuur als eerste byte steeds een enum met het type bericht. Hieronder zie je een voorbeeld van een dergelijk bestand, ditmaal om de positie van een speler te verzenden.

```
File f;
f.writeMem();
f.putByte(M_CLIENT_POS);
f.putFlt(pos.x);
f.putFlt(pos.y);
```

Alle netwerkfuncties in de client en de server zullen dus steeds naar de eerste byte van een bericht kijken om te bepalen welke functie het bericht verder zal afhandelen. Je maakt best een enumeratie die alle mogelijke berichten bevat. In een groot programma kan ook een tweede byte geschakeld worden om een verdere onderverdeling te maken.

Time for Action

Maak in je project een enum 'MESSAGE' met de volgende waarden:

- ❑ M_CLIENT_FULL
- ❑ M_CLIENT_POS
- ❑ M_HELLO
- ❑ M_ADD_CLIENT
- ❑ M_REMOVE_CLIENT

30.2 Gedeelde classes

Je app zal data structuren nodig hebben die zowel bij de client als bij de server bekend zijn. Dat kan bijvoorbeeld een class voor de player zijn, met minstens een naam en een positie. Het zou ook een item in de game wereld kunnen zijn, een chat bericht of een quest.

Aan de andere kant zijn er ook steeds verschillen tussen de client en de server. Zo zal een player bij de client op het scherm moeten verschijnen, en zal de gebruiker hem kunnen verplaatsen via de muis of het toetsenbord. Bij de server is dat niet wenselijk, maar moet een player wel in een database opgeslagen kunnen worden.

We maken voor dit soort classes een zogenaamde 'base class' in de library 'Shared'. Het voorbeeld bevat een dergelijke class voor een speler, die we `netClient` noemen:

```
class netClient
{
    int id;
    Str name = "Player";
    Color color = RED;
    Vec2 pos;

    void writePosToFile(File & f)
    {
        f.putFlt(pos.x);
        f.putFlt(pos.y);
    }

    void readPosFromFile(File & f)
    {
        pos.x = f.getFlt();
        pos.y = f.getFlt();
    }

    void writeDetailsToFile(File & f)
    {
        f.putStr(name);
        f.putByte(color.r).putByte(color.g).putByte(color.b);
        writePosToFile(f);
    }

    void readDetailsFromFile(File & f)
    {
        name = f.getStr();
        color.r = f.getByte(); color.g = f.getByte(); color.b = f.getByte();
        readPosFromFile(f);
    }
}
```

Naast data zoals id, naam, kleur en positie bevat de class ook functies. Als we een bericht over het netwerk versturen, zullen we steeds deze functies gebruiken om de gewenste data aan een `File` toe te voegen. Zo zijn we zeker dat de client en de server dezelfde data verwachten. Mocht je in een later stadium bijvoorbeeld de snelheid van een speler willen onthouden en die ook meesturen in een 'Details' bericht, dan moet je dat enkel hier aanpassen.

Note

Het is soms verleidelijk om deze data rechtstreeks in de client of server applicatie naar een bestand te schrijven. Vroeg of laat zal je je echter vergissen, door bijvoorbeeld de volgorde van de data door mekaar te halen. Dat soort fouten is erg moeilijk te debuggen. Maak er daarom een gewoonte van om een gedeelde base class te maken.

Time for Action

Voeg deze class toe aan het project, in het bestand 'netClient'.

HOOFDSTUK 31

De Server

De server applicatie is verantwoordelijk voor het volgende:

- ✓ connecties van nieuwe clients aanvaarden;
- ✓ berichten van bestaande clients aanvaarden en indien nodig doorsturen naar andere clients;
- ✓ opmerken wanneer een clients offline gaat en dat indien nodig laten weten aan andere clients;

Daarnaast is het meestal zo dat data ook opgeslagen wordt in een database, dat er game-events gegenereerd worden, het gedrag van AI's wordt berekend, etc.

31.1 Main Program Loop

Het hoofdprogramma van de server is meestal behoorlijk eenvoudig: je start een server object, update dat regelmatig en tekent eventueel wat op het scherm. Dat laatste is zelfs dikwijls ongewenst, want een serverprogramma draait dikwijls op een server OS zonder gui. We overlopen even de voorbeeld code.

```
void InitPre()
{
   EE_INIT();
   App.flag = APP_WORK_IN_BACKGROUND|APP_NO_PAUSE_ON_WINDOW_MOVE_SIZE;
}
```

De **InitPre()** functie is niet bijzonder. De Applicatie krijgt enkele flags mee zodat de update functie ook door gaat als het programma geen focus heeft of geminimaliseerd is.

```
bool Init()
{
   if(!Server.create())
   {
      Exit("Can't create Server");
   }
   return true;
}
```

In **Init()** creëren we het **Server** object. Die server class moeten we wel zelf schrijven, wat hieronder aan bod komt. Als het niet lukt om de server te starten, dan wordt het programma afgesloten.

```
void Shut()
{
   Server.del();
}
```

Tot nu toe hebben we de **Shut()** functie vrijwel nooit nodig gehad. Bij een server applicatie moeten we zeker zijn dat de netwerk resources terug vrijgegeven worden als het programma klaar is. Daarom is deze code noodzakelijk.

```
bool Update()
{
   if(Kb.bp(KB_ESC))return false;
   Server.update();
   Time.wait(1);
   return true;
}
```

De **Update()** functie update de server. Later zullen waarschijnlijk ook andere updates toegevoegd worden, zoals bijvoorbeeld een AI manager update. Op het eind van de update functie laten we het programma een milliseconde wachten. Zo belasten we de CPU niet harder dan nodig.

```
void Draw()
{
   D.clear(TURQ);
   D.text(0, 0.7, S + "Server");
   D.text(0, 0.5, S + "Connected clients: " + Server.clients.elms());
   D.text(0, 0.3, S + "Local Addres: " + Server.addressLocal().asText());
}
```

Tot slot is er de **Draw()** functie die ons wat informatie geeft over de server. Het aantal actieve clients en het IP adres van de server.

Note

In dit voorbeeld gebruiken we het lokale IP adres van de server. Daarmee kan je een client op hetzelfde netwerk verbinden met de server, maar externe verbindingen zijn zo niet mogelijk. Wanneer het tijd is om de toepassing te testen over het internet, dan moet je het globale IP adres gebruiken. Het is mogelijk dat je daarvoor ook de router correct moet configureren.

Time for Action

Voeg de functies hierboven toe aan het bestand 'main'.

31.2 De server Class

De server zelf moet je niet zelf ontwikkelen. Esenthel voorziet een base class voor je server, waar je je eigen functies aan toevoegt. Meestal heb je maar enkele functies nodig. We overlopen stap voor stap wat er in deze class dient te staan. Ten eerste is er de class zelf:

```
class server : connectionServer
{
   // andere code
}
server Server;
```

De `server` class heeft als base class `connectionServer`. Die laatste wordt voorzien door de engine. Aangezien er maar één server object actief zal zijn, kunnen we er dadelijk een object van maken.

31.2.1 De Constructor

Een eerste functie binnen de server class is de constructor:

```
server() { clients.replaceClass<client>(); }
```

Constructors komen vaak voor in C++, maar in Esenthel schrijf je ze meestal niet zelf. Het betreft een functie met dezelfde naam als de class, die automatisch wordt uitgevoerd bij het maken van een object. De class `connectionServer` bevat een container clients waarin elke actieve client onthouden wordt. De class van die clients willen we vervangen door onze eigen 'client' class die we zodadelijk zullen ontwerpen. Met de functie `replaceClass` laten we dat weten aan de `connectionServer`.

31.2.2 Data verzenden

```
void sendToClients(File & f, client & sender)
{
  clients.lock();
  FREPA(clients)
  {
     client & c = (client&) clients.lockedData(i);
     if(&c != &sender) // don't send back to sender
     {
        f.pos(0);
        c.connection.send(f);
     }
  }
  clients.unlock();
}
```

Vaak zullen we ontvangen informatie willen versturen naar alle clients, behalve naar de afzender. Omdat eenvoudig te doen vanuit andere delen van het programma voorzien we een functie die de te verzenden informatie *(een File)* en de afzender *(een client)* als argument heeft.

Nieuw hierbij is het *lock* concept. Een server zal dikwijls verschillende processoren tegelijk gebruiken om zo snel mogelijk te gebruiken. Stel je voor dat er clients bijkomen of verdwijnen terwijl we met **FREPA** alle clients afgaan. De server zou dan waarschijnlijk crashen. Om dat te voorkomen wordt de clients container 'gelocked'. Na de loop dien je een 'unlock' te gebruiken, zodat er terug clients toegevoegd kunnen worden.

Elke client wordt vervolgens omgezet naar onze eigen **client** class. Dat gebeurt met het statement:

```
client & c = (client&) clients.lockedData(i);
```

Dit is nodig omdat de connectionServer zijn eigen base class voor een client heeft. We hebben die echter vervangen door een eigen class. Hier geven we aan dat we uit de clients container een referentie naar een client(i) willen gebruiken, maar wel als onze eigen class.

Vervolgens willen we zeker zijn dat we de data niet terug naar de afzender sturen. Om dat te doen vergelijken we het geheugenadres van de huidige client met dat van de afzender. Enkel wanneer die verschillend zijn, versturen we de File.

Het verzenden van die File bestaat uit twee stappen. We beginnen met de positie in de File terug op het begin te zetten. Vervolgens gebruiken we de connectie van de huidige client om de File naar die client te sturen.

Note

Een functie die data uit een File leest, doet dat vanaf de huidige positie en verplaatst die positie stap voor stap *(of beter bit voor bit)* naar het eind van de file. Indien een bestand nogmaals gelezen wordt, moet je de positie dus terug op nul zetten.

31.2.3 Info voor nieuwe clients

Telkens een nieuwe speler zich aanmeld bij de server, heeft die informatie nodig over alle andere spelers. We voorzien daarom een functie `sendAllClientDetails`. De functie ziet er zo uit:

```
void sendAllClientDetails(client & destination)
{
  clients.lock();
  FREPA(clients)
  {
     client & c = (client&) clients.lockedData(i);
     if(&c != &destination) // don't send the client itself
     {
        File f;
        f.writeMem();
        f.putByte(M_ADD_CLIENT);
        f.putInt(c.id);
        c.writeDetailsToFile(f);
        f.pos(0);
        destination.connection.send(f);
     }
  }
  clients.unlock();
}
```

Zoals je ziet wordt ook hier het 'lock' mechanisme gebruikt. Ditmaal is er maar een bestemming: de nieuwe client. Voor alle andere clients maken we een file met de details van die clients. Die file sturen we naar de client 'destination'. Aangezien de inhoud voor elke client anders is, kunnen we niet zoals bij de vorige functie telkens dezelfde file versturen. We maken dus een nieuwe file voor elke client.

Time for Action

Voeg alle code hierboven samen tot de class `server` in het bestand met dezelfde naam.

31.3 De Client Class

Wanneer de server ontdekt dat een nieuwe client een verbinding aanvraagt, zal er automatisch een object van de class `client` gemaakt worden. Ook de update functie van die client zal door de server automatisch uitgevoerd worden. Tenzij je wil dat die client absoluut niets doet, zal je wel een eigen class voor die client moeten voorzien. Deze class heeft twee base classes nodig: enerzijds **moet** de class gebaseerd zijn op `ConnectionServer.Client` om dat ze anders niet de standaard client class kan vervangen, anderzijds willen we ook de shared base class `netClient` gebruiken.

```
int NextClientID = 0;

class client : ConnectionServer.Client, netClient
{
  bool sentHello = false;
  // add code here
}
```

Note

Voor de eigenlijke class staat een `int` 'NextClientID'. Dat is een globale variabele die we gebruiken om elke speler een uniek ID te geven. Daarna begint de class. Alhoewel die leeg lijkt, bevat ze op dit moment reeds alle functies en variabelen van zowel `ConnectionServer.Client` als `netClient`. Een variabele die je alvast bovenaan in de class kan toevoegen is de bool 'sentHello'. Die gebruiken we later in de update functie om te controleren of dit een nieuwe client is.

31.3.1 De Create Functie

Vervolgens moet er in deze class een create functie voorzien worden. De server class zal bij het maken van een nieuwe client automatisch proberen de create functie uit te voeren, maar die moet dan wel dezelfde argumenten hebben als de base class `ConnectionServer.Client`. Als je een create functie maakt zonder die argumenten, zal de server je functie niet gebruiken en gewoon de create functie van de base class uitvoeren.

```
void create(ConnectionServer &server)
{
  // each client needs his own unique ID
  id = NextClientID++;

  // send details for this client to other clients
  File f;
  f.writeMem().putByte(M_ADD_CLIENT).putInt(id);
  writeDetailsToFile(f);
  Server.sendToClients(f, T);
}
```

In deze functie kennen we de nieuwe client een ID toe. Dit is belangrijk omdat we later updates over deze client naar alle andere clients willen sturen. Die clients kunnen enkel weten over welke client het bericht gaat, wanneer elk van die clients een uniek nummer heeft. Daarom voorzien we een globale integer `NextClientID`. We kennen de huidige waarde toe aan de variabele `id` *(aanwezig in netClient)* en verhogen daarna de waarde van NextClientID.

Vervolgens willen we alle bestaande clients laten weten dat er een nieuwe client toegevoegd moet worden. We maken daarom een File waarin we de id van de client plaatsen, gevolgd door de

'details' van deze client. De functie **sendToClients** die we aan de server class toevoegden kan nu gebruikt worden om de **File** te verzenden. We gebruiken **T** *(dit object)* als afzender, zodat de informatie niet naar deze client gestuurd zal worden.

> **Note**
>
> In praktijk zal je de gegevens van een client meestal pas naar andere clients sturen na de controle van een login en wachtwoord. Je zal deze code dan moeten verplaatsen.

31.3.2 De Update Functie

Bijna alles gebeurt verder in de update functie. Die is verantwoordelijk voor drie zaken:

- ✓ indien de client nog geen connectie heeft, moet die opgezet worden;
- ✓ wanneer de client een bericht naar de server stuurt, moet dat bericht behandeld worden;
- ✓ wanneer de client de verbinding verbreekt, moet hij verwijderd worden.

Net zoals de create functie, is ook de update functie aanwezig in de base class. Om er voor te zorgen dat de server ze automatisch uitvoert, moet dit een bool functie zijn, zonder argumenten. Dit maal is het wel belangrijk dat ook de update functie van de base class uitgevoerd wordt. Daarvoor gebruiken we het keyword **super**:

```
bool update()
{
  if(super.update()) {
    // still connected, do something
    return true;
  }

  // connection is lost if this point is reached
  return false;
}
```

Een **return true** laat de server weten dat deze client nog steeds actief is. Bij **return false** veronderstelt de server dat de client offline is, en wordt deze client verwijderd uit het geheugen.

Indien een client niet meer actief is, dan willen we dat aan de andere clients laten weten. Dat gebeurt net voor de lijn 'return false'. Op die plaats moet dit bericht gemaakt en verstuurd worden:

```
File f;
f.writeMem().putByte(M_REMOVE_CLIENT).putInt(id);
Server.sendToClients(f, T);
```

Zolang `super.update()` lukt, blijft de verbinding actief. Het eerste wat dan, bij een nieuwe verbinding, moet gebeuren is de client laten weten dat de verbinding geslaagd is. We hebben om deze reden de bool 'sentHello' toegevoegd aan de class. Standaard is die false, wat wil zeggen dat dit een nieuwe client is. We controleren in dat geval of de status van de verbinding gelijk is aan `CONNECT_GREETED`. Als dat zo is, dan sturen we een kort 'Hello' bericht terug naar de client. Op dat moment informeren we de client ook over de andere clients die al actief zijn. *Hiervoor voegen we een functie* ***`sendAllClientDetails()`*** *toe aan de server class.*

```
if(!sentHello)
{
  if(connection.state() == CONNECT_GREETED) // connection is ready for data
  {
     File f;
     f.writeMem().putByte(M_HELLO).pos(0);
     connection.send(f);
     sentHello = true;

     // send other clients' details to this client
     Server.sendAllClientDetails(T);
  }
}
```

Vervolgens controleren we tijdens elke update of er nieuwe berichten binnenkomen. Als er data ontvangen is, dan zit die in de file 'connection.data'. De eerste byte van elk bericht geeft aan over wat voor bericht het gaat, via een enumeratie 'MESSAGE'. Voor elk van de berichten die een client kan versturen voorzien we dan een functie om het bericht te verwerken, waarbij we 'connection.data' meesturen als bestand.

```
REP(8) if(connection.receive(0)) // if data is recieved
{
  // first byte is type of message
  byte message = connection.data.getByte();

  // get the rest of the data
  switch(message)
  {
     case M_CLIENT_FULL: handleFullUpdate(connection.data); break;
     case M_CLIENT_POS : handlePosUpdate (connection.data); break;
  }
}
```

31.3.3 HandlePosUpdate

Nu dienen we nog de functies te maken om deze messages effectief te verwerken. We beginnen met de functie `handlePosUpdate`, een lidfunctie van de client class.

Op het moment dat de functie uitgevoerd wordt, weten we dat de data die in het bestand zit, gelezen kan worden via `netClient.readPosFromFile()`. Dat moet dan ook eerst gebeuren. In het geval van deze update willen we dat de ontvangen positie verstuurd wordt naar alle andere clients. We maken dus nu op de server een nieuw bericht, bijna zoals het binnenkomende bericht. Met dit verschil dat ook de id van deze client opgenomen wordt. Zo weten de andere clients over welke client het gaat.

```
void handlePosUpdate(File & data)
{
  readPosFromFile(data);

  // send to other clients
  File f;
  f.writeMem().putByte(M_CLIENT_POS).putInt(id);
  writePosToFile(f);
  Server.sendToClients(f, T);
}
```

Een bericht doorsturen naar andere clients is meestal niet moeilijker dan het voorbeeld hierboven. Soms zal er wel meer moeten gebeuren, zoals het aanpassen van gegevens in een database. In dat geval zal je de tijd moeten nemen om een nieuwe functie uit te schrijven.

31.3.4 HandleFullUpdate

Er is nog een andere message mogelijk: `handleFullUpdate`. De client zal dit bericht sturen als de speler zijn naam of kleur heeft aangepast. De behandeling van dit bericht is ongeveer gelijk aan de vorige functie: we lezen de binnenkomende data, en sturen ze door naar alle andere clients.

```
void handlePosUpdate(File & data)
{
  readPosFromFile(data);

  // send to other clients
  File f;
  f.writeMem().putByte(M_CLIENT_POS).putInt(id);
  writePosToFile(f);
  Server.sendToClients(f, T);
}
```

Time for Action

Voeg alle code hierboven samen tot de class `client` in het bestand met dezelfde naam.

HOOFDSTUK 32

De Client

Wat de client betreft beperkt deze cursus zich tot de elementen die betrekking hebben op het netwerk-gedeelte. Andere classes zoals `detailsGui` zijn tijdens de cursus al meermaals aan bod gekomen. Je word verwacht deze classes zelf uit te werken.

32.1 De 'Peer' Class

Met 'Peers' bedoelen we andere clients die zich in je buurt bevinden. Gemakkelijkheidshalve gaan we er van uit dat alle actieve clients hier in mekaars buurt zijn. Bij grotere games zal naar de positie van een client gekeken worden. De server beslist dan welke speler dicht genoeg in mekaars buurt zijn om als peer beschouwd te worden.

De class `peer` heeft als base class `netClient`. Iedere `peer` heeft dus een positie, een kleur en een naam. Die worden enkel via het netwerk aangepast.

Wat een `peer` class bijzonder maakt is het gebruik van interpolatie. Positie updates worden ongeveer 10 keer per seconde verstuurd. Maar om een vloeiende beweging op het scherm te tonen is een fijnere aanpassing van de posities nodig. Meer posities over het netwerk versturen is een optie, maar die belast het netwerk al snel te veel. Daarom gaan we de positiewijzigingen tussen de updates zelf invullen. Daar hebben we interpolators voor nodig. In dit geval is dat een `Interpolator2` voor de interpolatie van de positie, een `Vec2`. Bij een 3D project zou je een `Interpolator3` gebruiken. Daarnaast heb je ook steeds een interpolator voor de tijd nodig: `InterpolatorTime`. De lege class ziet er zo uit:

```
class peer : netClient
{
   Interpolator2 iPos;
```

```
    InterpolatorTime iTime;

    // hier worden functies toegevoegd
}
```

32.1.1 Update

De update functie zal eerst de `InterpolatorTime` updaten. Daarna moet ook de positie geupdate worden, met de interpolatietijd als argument.

```
void update()
{
  iTime.update();
  iPos.update(iTime);
}
```

32.1.2 posities

We voorzien nog twee extra functies om het werken met posities vlot te laten verlopen. De eerste is `recalculatePos`. Deze functie zullen we telkens uitvoeren wanneer we een nieuwe positie via het netwerk ontvangen. Ze zorgt ervoor dat de interpolators hun werk kunnen doen.

```
void recalculatePos()
{
  iPos.step(pos, iTime);
  iTime.step();
}
```

De positie die we gebruiken om de peer op het scherm te tonen krijgen we via `iPos()`. Dit zou verwarrend kunnen zijn, want via `netClient` bestaat er ook al een variabele 'pos'. Om te voorkomen dat we ons vergissen, maken we een extra functie `getPos()`.

```
Vec2 getPos()
{
  return iPos();
}
```

32.1.3 Draw

Tot slot heeft de peer class een `Draw()` functie nodig. Hierin tekenen we een cirkel en een tekst op het scherm. We gebruiken de functie `getPos()` om de geïnterpoleerde positie op te vragen.

```
void draw()
{
  Circle(0.05, getPos()).draw(color);
  Vec2 textPos = getPos();
  textPos.y += 0.1;
  D.text(textPos, name);
}
```

Time for Action

Werk de volledige `peer` class uit aan de hand van de bovenstaande code.

32.2 PeerManager

Aangezien er meer dan één peer actief kan zijn, maken we hiervoor een typische manager class. Die bevat een geheugencontainer om peers te onthouden, evenals functies om een peer toe te voegen, te verwijderen of te zoeken. Ook is er een functie voorzien om alle peers in een keer op het scherm te tekenen.

De lege class ziet er zo uit:

```
class peerManager {
private:
  Memx<peer> peers;

public:
  // add other code
}

peerManager PeerManager;
```

32.2.1 Peer toevoegen

Je hebt een functie nodig om een nieuwe peer toe te voegen. Deze functie is ongeveer gelijk aan functies die je in het verleden al gebruikte om iets aan een manager class toe te voegen.

Een klein verschil is dat je de nieuwe ID van de andere speler als functieargument gebruikt. We stellen in deze functie dadelijk het ID in van de nieuwe speler, maar geven ook een referentie naar dat nieuwe object terug als functieresultaat. Zo kan de code die deze functie gebruikt de peer verder aanpassen.

```
peer & add(int ID)
{
  peer & p = peers.New();
  p.id = ID;
  return p;
}
```

32.2.2 Peer vinden

Er is ook een functie nodig om te zoeken naar een peer met een bepaalde ID. Deze keer geven we geen referentie maar een pointer als resultaat. Het is immers mogelijk dat de peer met een bepaald ID niet bestaat. Maar het is onmogelijk om een lege referentie als resultaat te geven. Een lege pointer kan wel, dat is de 'null' pointer.

```
peer * find(int ID)
{
  FREPA(peers)
  {
     if(peers[i].id == ID)
     {
        return &peers[i];
     }
  }

  return null;
}
```

32.2.3 Peer verwijderen

Wanneer een speler offline gaat, dan moet die ook bij de andere clients verdwijnen. Daarom voorzien we een functie **remove**. In deze functie zullen we de speler met een gegeven ID uit de container verwijderen.

```
void remove(int ID)
{
  FREPA(peers)
  {
     if(peers[i].id == ID)
     {
```

```
            peers.removeValid(i);
            return;
        }
    }
}
```

32.2.4 Peers tellen

We willen ook weten hoeveel spelers er online zijn. Dat getal is gelijk aan het aantal elementen in de container met peers. Maar omdat die container private is, voorzien we ook een functie om deze informatie aan andere classes door te geven.

```
int elms()
{
  return peers.elms();
}
```

32.2.5 Update en Draw

Tot slot zijn er de update en draw functies. Die zullen de gelijknamige functies van elke peer uitvoeren:

```
void update() { FREPA(peers) peers[i].update(); }
void draw  () { FREPA(peers) peers[i].draw  (); }
```

Time for Action

Gebruik de bovenstaande code om de class peerManager uit te werken.

32.3 Peer Messages

Het bestand 'peerMessages' bevat functies om data van het netwerk te verwerken die bedoeld is om peers aan te passen. Eventueel hadden deze functies ook in de class `peerManager` kunnen staan. Maar het is wel overzichtelijk om ze netjes samen in één bestand te plaatsen.

32.3.1 AddPeer

```
void AddPeer(File & f)
{
   int id = f.getInt();
   peer & p = PeerManager.add(id);
   p.readDetailsFromFile(f);
}
```

De functie `AddPeer()` haalt, zoals al deze functies, eerst het id uit het bestand. Vervolgens wordt een nieuwe peer gegenereerd die dan verder de details uit het bestand leest.

32.3.2 GetPeerDetails

```
void GetPeerDetails(File & f)
{
   int id = f.getInt();
   // try to find a peer with this id
   peer * p = PeerManager.find(id);
   if(p != null)
   {
      p.readDetailsFromFile(f);
   }
}
```

In het geval van `GetPeerDetails()` is het mogelijk dat de functie `find()` null als resultaat heeft. Je moet dan ook controleren of dat zo is, voor je probeert de functie `readDetailsFromFile()` uit te voeren.

32.3.3 GetPeerPos

```
void GetPeerPos(File & f)
{
   int id = f.getInt();
   peer * p = PeerManager.find(id);
   if(p != null)
   {
      p.readPosFromFile(f);
      p.recalculatePos();
   }
}
```

Deze functie lijkt sterk op de vorige. In feite zullen alle functies die gebruikt om informatie van het netwerk naar objecten over te brengen, in grote mate op mekaar lijken. In dit geval zullen we na het lezen van de positie ook de functie `recalculatePos()` uitvoeren. Die functie dient om de interpolators te updaten. (Zie uitleg in de vorige sectie.)

32.3.4 RemovePeer

Wanneer er een bericht binnenkomt om een peer te verwijderen, dan bevat dat bericht enkel de ID van die peer. In de class `peerManager` hebben we een functie gemaakt die een peer verwijdert aan de hand van zijn ID. We kunnen die functie hier eenvoudig gebruiken.

```
void RemovePeer(File & f)
{
   int id = f.getInt();
   PeerManager.remove(id);
}
```

Time for Action

Voeg de 4 functies hierboven toe aan het bestand 'peerMessages'.

32.4 De Network Class

Een tweede class controleert alle netwerk messages. De lege class ziet er zo uit:

```
class network
{
private:
   Connection connection;
   float startTime;
   bool connected = false;

public:
   // ... more code ...
}
network Network;
```

Het belangrijkst hier is de class `Connection` die de engine voorziet. Die gebruik je om een verbinding te maken met een server. Je start deze verbinding in de functie `create()`.

32.4.1 Create

```
void create()
{
  startTime = Time.curTime();
  SockAddr serverAddress;
  serverAddress.setIP("127.0.0.1", 65535);
  connection.clientConnectToServer(serverAddress);
}
```

De **startTime** stel je gelijk aan de huidige tijd. Dat is belangrijk om later te controleren hoe lang je al op een verbinding wacht. Vervolgens heb je een IP adres nodig. Dit voorbeeld gebruikt het locale adres van je computer, maar dat zal je uiteindelijk aanpassen naar een publiek IP adres. *(Bij een afgewerkt programma is het zelfs gebruikelijk om eerst ergens op een webserver een bestandje te downloaden dat het huidige IP adres bevat van de game server.)*

32.4.2 Update

De **update()** functie bestaat uit twee delen. Het eerste deel wordt uitgevoerd wanneer de verbinding nog niet in orde is. Het tweede deel kijkt, in het geval van een geldige verbinding, of er nieuwe messages zijn.

```
void update()
{
   if(!connected)
   {
      // code
   } else {
      // more code
   }
}
```

In het deel waarin er nog geen verbinding is, wordt eerst gekeken of er nieuwe messages zijn. Het eerste bericht dat je de server laat terugsturen is **M_HELLO**. Als er een nieuw bericht is, dan controleer je of dat het juiste bericht is en wordt **connected** gelijk aan **true**.

Indien er geen verbinding is en er is ook geen nieuw bericht, dan is het tijd om enkele testen uit te voeren. Je controleert of de state van de connectie wel in orde is. In het geval die gelijk is aan **CONNECT_INVALID** of **CONNECT_VERSION_CONFLICT** dan is het duidelijk dat de verbinding niet zal lukken. We kunnen dat laten weten aan de gebruiker. Ook als we langer dan 5 seconden wachten op een server kunnen we aannemen dat het niet meer in orde komt.

```
if(connection.receive(0))
{
  if(connection.data.getByte() == M_HELLO) connected = true;
```

```
} else
{
  // not connected yet, check for errors
  if(connection.state() == CONNECT_INVALID || connection.state() ==
      CONNECT_VERSION_CONFLICT)
  {
     Exit("Couldn't connect to server");
  }

  if(Time.curTime() - startTime > 5)
  {
     Exit("Connection Timeout");
  }

  Time.wait(1); // wait a bit
}
```

Als er wél een verbinding is, dan controleren we ook op nieuwe messages. De eerste byte laat je weten om welk bericht het gaat. Daarmee beslis je welke functie je uitvoert en geef je de rest van het bestand *(connection.data)* door aan die functie:

```
REP(8) if(connection.receive(0))
{
  byte message = connection.data.getByte();
  switch(message)
  {
     case M_ADD_CLIENT    : AddPeer       (connection.data); break;
     case M_CLIENT_FULL   : GetPeerDetails(connection.data); break;
     case M_CLIENT_POS    : GetPeerPos    (connection.data); break;
     case M_REMOVE_CLIENT: RemovePeer    (connection.data); break;
  }
}
```

Tot slot is er ook nog de functie **send()**. Die kan je overal in je programma gebruiken om data naar de server te sturen. Veiligheidshalve wordt de leespositie van de file eerst terug op nul gezet.

```
void send(File & f)
{
   f.pos(0);
   connection.send(f);
}

bool isConnected() {
  return connected;
}
```

Time for Action

Maak aan de hand van de bovenstaande code de class **network** in het gelijknamige bestand.

32.5 De Player Class

Het programma bevat ook een class voor een player. Net zoals de **client** class bij de server, wordt ook hier **netClient** als basis gebruikt. Die bevat immers variabelen voor de naam, de kleur en de positie van de player.

Hier start je met de volgende class:

```
class player : netClient {
private:
  float timeForUpdate = 0;

public:
  // add functions later
}
player Player;
```

32.5.1 Create

De create functie is in dit geval kort. We stellen enkel een standaard kleur in voor de speler.

```
void create() {
  color = RED;
}
```

32.5.2 Update

In de **update()** functie staat eerst de gebruikelijke code om de speler te bewegen:

```
if(Kb.b(KB_LEFT )) pos.x -= Time.ad();
if(Kb.b(KB_RIGHT)) pos.x += Time.ad();
if(Kb.b(KB_UP   )) pos.y += Time.ad();
if(Kb.b(KB_DOWN )) pos.y -= Time.ad();
Clamp(pos.x, D.viewRect().min.x, D.viewRect().max.x);
Clamp(pos.y, D.viewRect().min.y, D.viewRect().max.y);
```

Daarna volgt de code om de positie regelmatig naar de server te sturen. We doen dit elke 0.1 seconde. In dat geval wordt er een bericht van het type `M_CLIENT_POS` gegenereerd dat via de Network class naar de server verzonden wordt.

```
if(timeForUpdate > 0)
{
   timeForUpdate -= Time.ad();
} else {
   // notify server of new position
   File f;
   f.writeMem().putByte(M_CLIENT_POS);
   writePosToFile(f);
   Network.send(f);

   timeForUpdate = 0.1;
}
```

32.5.3 Draw

De draw functie is eenvoudig. Net zoals in de `peer` class tekenen we een cirkel en een tekst op het scherm:

```
void draw()
{
  Circle(0.05, pos).draw(color);
  Vec2 textPos = pos;
  textPos.y += 0.1;
  D.text(textPos, name);
}
```

32.5.4 setDetails

Ook van belang is de functie `setDetails()`, die wordt uitgevoerd wanneer er op de 'ok'-knop wordt gedrukt in de gui. Op het moment dat we de naam en de kleur van de speler wijzigen, maken we een bericht van het type `M_CLIENT_FULL` dat we naar de server sturen.

```
void setDetails(C Str & name, C Color & color)
{
  T.name = name;
  T.color = color;

  // send details to server
  File f;
  f.writeMem().putByte(M_CLIENT_FULL);
```

```
  writeDetailsToFile(f);
  Network.send(f);
}
```

Note

In tegenstelling tot de server code voor het versturen van de berichten, voegen we nu niet het id van deze client toe. De server weet immers al van welke client dit komt.

Time for Action

Voeg de bovenstaande code samen tot de class `player`.

32.6 Main Program Loop

Een volwaardig programma zal waarschijnlijk uit meerdere application states bestaan. Om dit voorbeeld eenvoudig te houden is enkel de default state aanwezig. We overlopen even de verschillende functies.

32.6.1 InitPre

```
void InitPre()
{
   EE_INIT();
   App.flag=APP_WORK_IN_BACKGROUND|APP_NO_PAUSE_ON_WINDOW_MOVE_SIZE;
}
```

De `InitPre` functie bevat een extra lijn code om enkele application flags in te stellen. Deze flags zijn opties die beinvloeden hoe het programma zich gedraagt. Om de werking van deze app te demonstreren zullen we verschillende instances van deze app gelijktijdig openen. We willen dat die allemaal het scherm updaten, niet enkel de applicatie met de window focus.

32.6.2 Init

```
bool Init()
{
   Network   .create();
   Player    .create();
   DetailsGui.create();

   return true;
}
```

Er is niets bijzonders aan deze code. De verschillende objecten (netwerk, player en gui) worden geïnitialiseerd.

32.6.3 Shut

```
void Shut() {}
```

32.6.4 Update

```
bool Update()
{
   if(Kb.bp(KB_ESC))return false;

   Network.update();

   // the program is not really active as long as the
   // client is not connected to the server
   if(Network.isConnected())
   {
      if(Kb.bp(KB_F1)) DetailsGui.show();
      Gui.update();

      Player.update();
      PeerManager.update();
   }
   return true;
}
```

Hierin is enkel het volgende concept nieuw:

```
if(Network.isConnected())
{
  // ...
}
```

Het `Network` object dat we hieronder uitwerken bevat een functie om te controleren of er een verbinding is. In dit voorbeeld gebruiken wordt het eigenlijke programma niet geupdated of getoond zolang dat niet het geval is. In een echt programma zal je waarschijnlijk wel een loginscherm willen tonen op dat moment.

32.6.5 Draw

```
void Draw()
{
   D.clear(WHITE);

   // Don't do anything if not connected
   if(!Network.isConnected())
   {
      D.text(0, 0, "Waiting for Server...");
      return;
   }

   // draw peers and player on screen
   PeerManager.draw();
   Player.draw();

   // add texts and gui
   D.text(0, -0.9,  S + "Press F1 for options");
   D.text(0, -0.8,  S + "Connected players: " + PeerManager.elms());
   Gui.draw();
}
```

Time for Action

Voeg alle code hierboven toe aan het bestand 'main'.

32.7 De Gui

De gui voor dit project kan je zeker zelf uitwerken. Maak een gui Window met een `TextLine` om je naam in te vullen, een `Button` 'color' die een `ColorPicker` toont en een `Button` 'Ok' om de nieuwe waarden toe te kennen aan de Player.

Je zou met behulp van de code in de hoofdstukken over GUI zelf je gui class moeten kunnen schrijven. In de callback functie voor de button 'Ok' voer je de functie `Player.setDetails()` uit. Als argument zet je daar natuurlijk de nieuwe naam en kleur.

Time for Action

Werk de gui class uit.

Deel VI

3D Worlds

HOOFDSTUK 33

Inleiding

De Esenthel editor bevat een tool om je 3D wereld vorm te geven. Tutorials over het gebruik daarvan vind je op de Esenthel website. In dit deel leer je hoe je die wereld gebruikt in je programma.

33.1 De wereld laden

Meestal gebeurt het laden en updaten van een 3D wereld in een afzonderlijke gamestate. Bij de start van een programma wil je waarschijnlijk eerst enkele menu's en misschien een game lobby tonen. Pas daarna schakel je over naar een gamestate die je wereld toont. Om de code in dit deel eenvoudig te houden, slaan we die stappen over en werken we enkel met een 3D wereld in de standaard gamestate. Daarin zijn 4 functies belangrijk:

- de Init functie,
- de Update functie,
- de Draw functie en
- de Render functie.

33.1.1 Init

In tegenstelling tot een 2D wereld, heeft een 3D wereld enkele standaard vereisten: Physics, een wereld en een camera. (In principe kan het ook zonder de wereld, maar dat is een heel ander soort game.) Deze drie worden geïnitialiseerd in de Init functie:

```
// enable physics engine
Physics.create(EE_PHYSX_DLL_PATH);

// load game world
Game.World.activeRange(D.viewRange());
Game.World.New(=== insert world here ===);
if (Game.World.settings().environment) {
  Game.World.settings().environment->set();
}

// set initial camera
Cam.setSpherical(Vec(16, 0, 16), -PI_4, -0.5, 0, 10);
Cam.set();
```

De eerste functie initialiseert de Physics engine. Die simuleert zwaartekracht en kan gebruikt worden om te detecteren wanneer objecten mekaar raken. Het argument `EE_PHYSX_DLL_PATH` is een constante waarde die de engine laat weten waar het de Physics library can vinden.

Daarna is het de beurt aan de game wereld. Eerst stellen we de active range in. Die is meestal gelijk aan de view range, dat is de afstand tot de camera waarbinnen elementen van de wereld op het scherm getoond worden. Door ze gelijk te stellen worden elementen tot op deze afstand geladen. Wanneer je de later de camera verplaatst, dan zullen ook verder gelegen items in het geheugen worden geladen. Als je verder wil zien in de game, dan zal je best eerst de viewRange aanpassen voordat je deze functie uitvoert. Maar, hoe groter de viewRange, hoe meer je GPU belast wordt.

De functie `Game.World.New()` laadt een wereld. Het argument is een World object dat je in de editor maakt. Vervolgens kan je, indien er een Environment object bij de wereld hoort, deze ook instellen.

Note

De functie `set()` wordt aangeroepen via een pointer notatie. Dit is een van die gevallen waar Esenthel een punt niet vanzelf omzet naar een pointer. Als je via een middelmuisklik naar de declaratie van environment kijkt, dan zie je dat die het type `environmentPtr` heeft. Het is dus slechts een verwijzing (een pointer) naar een environment. Vergeet je de pointer notatie, dan krijg je deze foutmelding:

```
error C2039: 'set': is not a member of
    'EE::CacheElmPtr(TYPE,EE:_Environments>
```

Tot slot volgen twee functies om de camera in te stellen. De functie `setSpherical()` bepaalt waar de camera staat en in welke richting hij kijkt. Daarna is het nodig om deze waarden actief te maken met de functie `set()`.

33.1.2 Update

De update functie moet in ieder geval de camera en de wereld updaten. In de volgende stap zullen we de camera aan de speler koppelen, maar voorlopig koppelen we die aan de muis om alvast wat te kunnen rondkijken:

```
if(Ms.b(1)) {
   Cam.transformByMouse(0.1, 100, CAMH_ZOOM | CAMH_MOVE));
} else
{
   Cam.transformByMouse(0.1, 100, CAMH_ZOOM | CAMH_ROT));
}
```

We bekijken deze functie niet in detail omdat we ze in de volgende stap zullen vervangen. Je kan de betekenis van de argumenten zelf achterhalen door de documentatie te bekijken.

Tot slot is het nodig de wereld te updaten naargelang de camera. Dat kan met:

```
Game.World.update(Cam.at);
```

33.1.3 Render

Een 3D wereld heeft een `Render` functie nodig. Voorlopig is de inhoud van deze functie eenvoudig, maar later kan dit behoorlijk complex worden. Het renderen van een frame gebeurt namelijk niet in één keer. Er zijn verschillende stappen om bijvoorbeeld belichting en schaduw correct te tonen.

In elk geval zal de wereld hier getekend moeten worden. Dat gebeurt zo:

```
void Render() {
   Game.World.draw();
}
```

33.1.4 Draw

Omdat de draw functie van `Game.World` het scherm al leegmaakt, is het onnodig om hier `D.clear()` uit te voeren. Maar de `Render` functie hierboven maakt geen deel uit van de gamestate. Je zal deze functie dus zelf moeten uitvoeren. Dat doe je door de naam van de functie door te geven aan de functie `Renderer()`:

```
void Draw() {
   Renderer(Render);
}
```

Time for Action

1. Open het project 3D Worlds en activeer de applicatie “3D World - Stage 0”. Voeg de hierboven beschreven code toe.
2. Open na het uitvoeren van het programma de app folder met windows explorer. Zoek de physics dll. Vergelijk het path met de waarde van EE_PHYSX_DLL_PATH.
3. Toon de behaalde FPS op het scherm. Pas daarna de active range en view range van het programma aan. Vergelijk de behaalde FPS met verschillende waarden.
4. Experimenteer met de functie `Cam.setSpherical`. Hoe pas je de startpositie aan?

33.2 Player en Camera

De controle over je avatar is essentieel voor je game. In de game wereld staat al een avatar, maar om die te controleren heb je code nodig. Omdat veel 3D games werken met een avatar, kan je een base class gebruiken: `Game.Chr`. Die voorziet alvast een aantal eigenschappen om je avatar te controleren.

33.2.1 De player class

Als je de code van `Game.Chr` bekijkt, dan vind je daar de volgende functie:

```
virtual bool update();
```

In je player class kan je deze functie overschrijven. Zo kan je je eigen update code toevoegen. De basis voor je class ziet er zo uit:

```
class Player : Game.Chr
{
  bool update()
  {
    // add your own code here

    // call update from base class
```

```
    return super.update();
  }
}
```

De code die we zelf toevoegen aan de player class dient om de avatar te verplaatsen. Eerst behandelen we de besturing via het toetsenbord:

```
input.turn.x = Kb.b(KB_Q) - Kb.b(KB_E);
input.turn.y = Kb.b(KB_T) - Kb.b(KB_G);
input.move.x = Kb.b(KB_D) - Kb.b(KB_A);
```

De class **Game.Chr** bevat reeds de eigenschappen turn en move. In de code hierboven ken je die een waarde toe. Kijk even naar **input.move.x**: die kan de waarde -1, 0 of 1 bevatten. De avatar zal naar links bewegen bij de waarde -1, naar rechts bij 1 en niet bewegen bij 0. Je kent die waarde toe door de stand van de toetsen D en A van mekaar af te trekken. Onthoudt dat een bool waarde 'true' gelijk is aan 1 en 'false gelijk aan 0. Dat geeft de volgende mogelijkheden:

1. D is ingedrukt: 1 - 0 = 1
2. A is ingedrukt: 0 - 1 = -1
3. A en D zijn ingedrukt: 1 - 1 = 0
4. A en D zijn niet ingedrukt: 0 - 0 = 0

Vervolgens controleren we beweging op z-as. Dat kan via het toetsenbord gebeuren, maar vaak kan je ook vooruit gaan door een muisknop in te drukken:

```
if(Ms.b(0))
{
   input.move.z = 1;
} else
{
   input.move.z = Kb.b(KB_W) - Kb.b(KB_S);
}
```

Om de avatar te laten springen gebruik je de volgende code:

```
if(Kb.bp(KB_SPACE))
{
  input.jump = 3.5;
}
else
{
  input.jump = 0;
}
```

Tot slot willen we de avatar kunnen roteren wanneer de rechtermuisknop ingedrukt is. We verbergen op dat moment ook de muis pointer.

```
if(Ms.b(1)) {
   Ms.hide();
   angle.x -= Ms.d().x * Time.d() * 50;
   angle.y += Ms.d().y * Time.d() * 50;
   Clamp(angle.y, -PI_4, PI_4);
} else
{
   Ms.show();
}
```

Time for Action

Activeer de applicatie "3D World - stage 1" en voeg een nieuw codebestand "player" toe. Schrijf daarin de class `player` met behulp van de code hierboven.

33.2.2 Link the player

De game wereld bevat al een player object, maar je zal aan je programma duidelijk moeten maken in welke class dat object terecht moet komen. Dat gebeurt in 4 stappen.

1. Om een bepaald object in je game wereld met een class in je programma te linken, gebruik je object classes. In de editor kan je een nieuwe object class aanmaken, maar voor de player is er al een voorzien: OBJ_PLAYER. (Zie afbeelding 33.1)
2. Je selecteert in de wereld het player object, en stelt de juiste class in. (Zie afbeelding 33.2)
3. Je voegt in je code een object map toe voor je class. Dit is een container zoals `Memx` die specifiek dient om game classes in op te slaan. In dit geval doe je dit onder de class `player`:

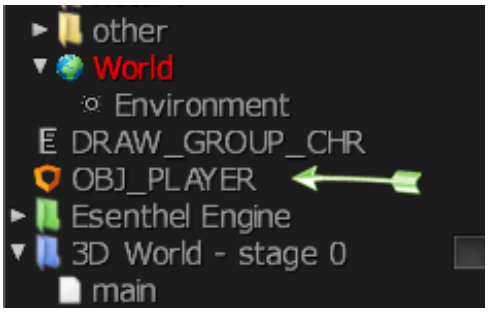
   ```
   Game.ObjMap<player> Players;
   ```

4. Tot slot maak je in de `Init` functie, net voor je je wereld laadt, een koppeling tussen de Object Class en de container die je net toevoegde:

   ```
   Game.World.setObjType(Players, OBJ_PLAYER);
   Game.World.New(=== Your game world ===);
   ```

Time for Action

Voer stap 3 en 4 uit om je class te linken aan het player object.



Figuur 33.1: Een Object Class.

Figuur 33.2: Een Object Class Toewijzen.

33.2.3 Camera

Ook voor de camera maken we een afzonderlijke class. Voorlopig heeft die enkel een update functie nodig, maar dat kan later meer worden. Je kan er van uit gaan dat je maar één camera nodig hebt. Daarom voorzie je onder de class ook een object.

```
class gameCam
{
  void update()
  {

  }
```

```
}
gameCam GameCam;
```

Binnen de functie update bepaal je de positie van de camera en de kijkrichting. Meestal zal de camera op de player gericht zijn. Maar het kan voorkomen dat er tijdelijk geen player object bestaat, bijvoorbeeld bij het einde van het spel, of tijdens een teleport. Daarom starten we de update functie met de mogelijkheid dat er geen speler bestaat. In dat geval voeren we de camera code uit de vorige sectie uit, zodat de camera kan bewegen via de muis.

```
if(!Players.elms())
{
   if(Ms.b(1)) {
      Cam.transformByMouse(0.1, 100, CAMH_ZOOM | CAMH_MOVE);
   } else
   {
      Cam.transformByMouse(0.1, 100, CAMH_ZOOM | CAMH_ROT);
   }
   return;
}
```

Vervolgens update je de afstand van de camera tot zijn target. Dat doen we via de beweging van het muiswiel:

```
Cam.dist -= Ms.wheel() * Time.d() * 10;
Clamp(Cam.dist, 2, 5);
```

En tot slot stel je de positie, yaw, pitch en target distance in:

```
Cam.setSpherical(Players[0].ctrl.center() + Vec(0, 0.5, 0),
   Players[0].angle.x, Players[0].angle.y, 0, Cam.dist);
Cam.updateVelocities().set();
```

Time for Action

1. Werk de class gameCam uit, zoals hierboven. Vergeet niet dat je de `update()` functie ook moet toevoegen aan de main `Update()` functie.
2. Pas enkele waarden in de camera update aan. Voer telkens je programma uit om te zien wat het resultaat is.
3. Pass de avatar controle in `Player.update()` aan aan je wensen. Hoe wil je je avatar besturen?
4. Voeg aan de main `Update()` functie ook de volgende regels toe:

```
D.grassUpdate();
Water.update(Vec2(0.01, 0.01));
```

   Wat is het resultaat?

HOOFDSTUK 34

World Objects Manipuleren

34.1 Object Classes en Code

34.1.1 Object Class

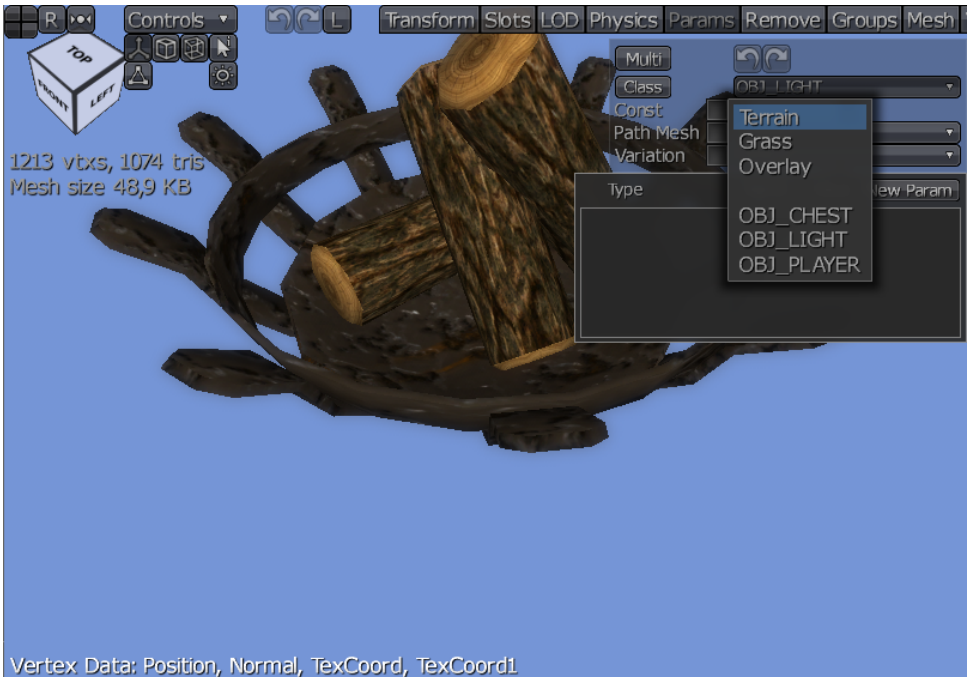
Standaard is elk object in de game wereld een onderdeel van het terein. Dat betekent dat dit object op het scherm getoond zal worden, maar dat interactie met dat object niet mogelijk is. Wanneer je met een object meer wil doen dan het simpelweg tonen, dan is er een connectie nodig tussen het object en je code. Die connectie zag je eigenlijk al in het vorige hoofdstuk: daar maakten we een koppeling tussen een `OBJ_PLAYER` en een de class `player`.

Je gaat steeds op de volgende manier te werk:

1. Maak een Object Class in de editor.
2. Open je object en kies de tab Params. Daar kan je de Object Class instellen. (Zie afbeelding 34.1)
3. Schrijf de code voor je nieuwe class.
4. Maak een `Game.ObjMap` voor deze class.
5. Link de Object Map aan de World.

De eerste twee stappen spreken voor zich, dus we springen direct naar de derde stap.

> **Note**
>
> Start in dit hoofdstuk met de applicatie '3D World - stage 2'.



Figuur 34.1: Een Object Class toewijzen.

34.1.2 Een custom object toevoegen

Wanneer je de object class in de editor instelt, zal je zien dat het object niet meer in de wereld verschijnt als je je game start. Het object is immers geen onderdeel meer van het terrein, maar je hebt nog geen code geschreven om aan te geven wat er dan wel moet gebeuren.

> **Note**
>
> Bij de game assets, in de map 'other', vind je een object 'magic lamp'. Er is ook al een object aanwezig in de game world. Dit object heeft de Object Class `OBJ_MAGIC_LAMP`. In deze oefening gaan we dat object rond zijn as laten draaien.

Wanneer je een class maakt voor een world object, dan moet je een speciale class als base class kiezen. Je hebt de keuze uit de volgende mogelijkheden:

- `Game.Animatable` gebruik je voor objecten met een animatie, zoals een kist die open en dicht kan.
- `Game.Chr` gebruik je voor avatars, zoals de speler en mobs.
- `Game.Destructible` gebruik je voor objecten die uiteen kunnen vallen in delen. (Op dit moment ontbreekt in de Editor een mogelijkheid om een dergelijk object te maken. Het kan wel via code, maar eenvoudig is dat niet.)

- `Game.Door` gebruik je voor deuren.
- `Game.Item` gebruik je voor voorwerpen waarmee interactie mogelijk is.
- `Game.Static` gebruik je voor voorwerpen waarmee interactie mogelijk is, maar die op een vaste plaats blijven staan.
- `Game.Kinematic` gebruik je voor voorwerpen waarmee interactie mogelijk is, maar die enkel via code verplaatst kunnen worden.

In deze oefening gebruiken we `Game.Static`. Voorzie alvast de volgende class:

```
class magicLamp : Game.Static {

}
Game.ObjMap<magicLamp> MagicLamp;
```

Pass in het bestand 'main' de functie `Init()` aan:

```
bool Init()
{
   Physics.create(EE_PHYSX_DLL_PATH);

   Game.World.activeRange(D.viewRange());
   Game.World.setObjType(Players, OBJ_PLAYER);
   Game.World.setObjType(MagicLamp, OBJ_MAGIC_LAMP); // <- New Code!
   Game.World.New(UID(2458107509, 1250513895, 1140948412, 2823156334));
   if(Game.World.settings().environment)Game.World.settings().environment->set();

   return true;
}
```

Wanneer je nu het programma uitvoert, dan zal je zien dat de lamp zichtbaar is.

34.1.3 Virtuele functies

De reden waarom je een object class moet baseren op een van de beschikbare `Game` classes, is dat deze classes virtuele functies bevatten (met het keyword `virtual`). Wanneer je de header file van `Game.Static` bekijkt, dan zie je dat alle functies in de class virtueel zijn. Bovendien is `Game.Static` gebaseerd op `Game.Obj`, die ook nog eens heel wat virtuele functies bevat.

Deze functies zijn nodig omdat de engine je eigen class niet kent. Door je class te koppelen aan de game wereld in de Init functie hierboven, link je je class aan de World Manager. Maar die kan niet weten wat voor functies jij aan die class toevoegt. De World Manager zal daarom functies uitvoeren die wél bekend zijn, zoals `update()` en `drawPrepare()`.

Aangezien deze functies virtueel zijn, kan je ze overschrijven in een afgeleide class. Met andere woorden: de World Manager voert de `update()` functie van `Game.Static` uit, tenzij je een functie met dezelfde naam voorziet in de afgeleide class. In dat geval wordt je eigen functie uitgevoerd.

34.1.4 Rotatie

We willen in deze oefening de lamp roteren rond zijn Y-as. Dat gebeurt door de rotatie aan te passen in de update functie. Je moet dus een eigen update functie toevoegen:

```
class magicLamp : Game.Static
{
   virtual bool update()
   {
      return super.update();
   }
}
```

De functie verwacht een `bool` als resultaat. We voeren in dit geval de `update()` functie van `Game.Static` uit en geven dat resultaat als functieresultaat.

Om het object te roteren maken we gebruik van zijn `Matrix`. Een `Matrix` voor een object is een combinatie van drie vectoren: x, y en z om de rotatie van een object in 3D te bepalen. Daarnaast bevat de `Matrix` ook nog een vector voor de positie. De schaal van het object kan afgeleid worden uit de grootte van de eerste drie vectoren.

Om een `Matrix` te roteren, kan je gebruik maken van `rotatie` functies. In dit geval willen we een rotatie rond de Y-as, dus kan je de volgende code gebruiken:

```
virtual bool update()
{
  Matrix m = matrix();
  m.rotateY(1 * Time.d());
  matrix(m);
  return super.update();
}
```

Deze code doet het volgende:

- ❏ Maak een nieuw object van de class `Matrix`, met de waarden van de huidige object matrix.
- ❏ roteer de matrix, rekening houdend met de tijdsdelta.
- ❏ Geef de object matrix de waarde van de geroteerde matrix.

Start je programma en controleer het resultaat. Je zal zien dat de lamp elke 5 seconden voorbijvliegt. Wat gaat er mis?

Als je een bewerking uitvoert op een matrix, dan is die van toepassing op alle onderdelen van die matrix, dus ook op zijn positie! We roteren dus ook de positie van het object in de 3D wereld. Ik heb je deze fout laten maken omdat je dit goed moet onthouden.

De oplossing bestaat er in de Matrix eerst te verplaatsen naar zijn nulpunt. Zowel schalen als roteren dient steeds op het nulpunt te gebeuren. Daarna verplaats je de matrix terug naar de oorspronkelijke positie:

```
virtual bool update()
{
  Matrix m = matrix();
  m.move(-pos());
  m.rotateY(1 * Time.d());
  m.move(pos());
  matrix(m);
  return super.update();
}
```

De nodige stappen voor een rotatie zijn dus:

- ❑ Maak een nieuw object van de class `Matrix`, met de waarden van de huidige object matrix.
- ❑ Verplaats het object naar zijn negatieve positie.
- ❑ Roteer de matrix, rekening houdend met de tijdsdelta.
- ❑ Verplaats het object naar zijn oorspronkelijke positie.
- ❑ Geef de object matrix de waarde van de geroteerde matrix.

Time for Action

1. Zoek een functie om de matrix zowel op de X als de Y-as te roteren. Gebruik voor beide assen een andere rotatiesnelheid.
2. Zoek een functie om de matrix te schalen. Laat het object langzaam groter en kleiner worden. Hint: je kan de functies `Sin()`, `Time.appTime()` en `Time.d()` gebruiken om de schaal te berekenen.

34.2 Draw Functies

34.2.1 De Renderer

Om een 3D wereld te renderen, gebruiken we de volgende code:

```
void Render() {
   Game.World.draw();
}

void Draw() {
   Renderer(Render);
}
```

De functie `Draw()` is je wel bekend. Maar waar we vroeger zelf objecten op het scherm tekenden, laten we nu de `Renderer()` zijn werk doen. (Je kan natuurlijk later nog steeds een Gui en 2D elementen over het resultaat van de Renderer tekenen.

Die renderer heeft een functie nodig om hem te vertellen wat hij moet doen. In dit geval is dat de functie `Render()`. Daarin geven we de renderer de opdracht om de game wereld te renderen. Ook hier kan je later extra code toevoegen als je game complexer wordt.

Een begrijpelijke misvatting is dat je er van uit gaat dat die wereld in één keer wordt gerenderd. De `Draw()` functie roept de `Renderer()` aan, de `Renderer` voert de functie `Render()` uit, en die voert `Game.World.draw()` uit, niet?

Maar zo eenvoudig is het niet. Het renderen van een 3D beeld gebeurt in verschillende fasen, die allemaal hun eigen doel hebben. Zo zijn er fasen voor het renderen van alle objecten, het toevoegen van licht, het toevoegen van schaduw en nog veel meer. Een volledig overzicht vind je in de engine header Graphics
Renderer:

```
enum RENDER_MODE // Rendering Mode, rendering phase of the rendering
   process
{
   RM_SIMPLE        , // simple
   RM_EARLY_Z       , // early z
   RM_SOLID         , // solid
   RM_SOLID_M       , // solid in mirrors/water reflections
   RM_AMBIENT       , // ambient
   RM_OVERLAY       , // overlay mode for rendering semi transparent
      surfaces onto solid meshes (like bullet holes)
   RM_OUTLINE       , // here you can optionally draw outlines of meshes
      using 'Mesh::drawOutline'
   RM_BEHIND        , // here you can optionally draw meshes which are
      behind the visible meshes using 'Mesh::drawBehind'
```

```
    RM_FUR            , // fur
    RM_BLEND          , // alpha blending
    RM_SHADOW         , // shadow map    , render all shadow casting objects
        here using 'Mesh::drawShadow', if objects will not be rendered in
        this phase they will not cast shadows
    RM_STENCIL_SHADOW, // shadow stencil, render all shadow casting objects
        here using 'Mesh::drawStencilShadow', if objects will not be
        rendered in this phase they will not cast shadows
    RM_CLOUD          , // clouds
    RM_WATER          , // water surfaces
    RM_PALETTE        , // color palette #0 (rendering is performed using
        'Renderer.color_palette'  texture)
    RM_PALETTE1       , // color palette #1 (rendering is performed using
        'Renderer.color_palette1' texture)
    RM_PREPARE        , // render all objects here using 'Mesh::draw', and
        add all lights to the scene using 'Light*::add'

    RM_SHADER_NUM=RM_SHADOW+1, // all modes from RM_SIMPLE to RM_SHADOW are
        included in the 'MeshPart' shader technique lookup list
};
```

In elke fase zal de renderer alle objecten in de World Manager overlopen en kijken of er iets moet gebeuren voor dat object. Voor de objecten waar je zelf een class voor maakt, kan je de renderer vertellen wat er moet gebeuren.

Om dat te doen, moet je eerst de virtuele functie **drawPrepare** overschrijven. Deze functie bepaalt welke extra fases dat op dit object van toepassing zijn. Het resultaat van deze functie is een **uint**, een unsigned integer. Omdat elke game class zelf al enkele fases voorziet, roep je eerst de **drawPrepare()** functie van de base class aan.

34.2.2 Outline Rendering

We zullen nu outline rendering toevoegen aan de lamp uit de vorige sectie. Voeg alvast deze functie toe aan de class **magicLamp**:

```
virtual uint drawPrepare()
{
  uint result = super.drawPrepare();
  \\ add your own rendering phases here
  return result;
}
```

Time for Action

Voeg aan de class magicLamp ook een bool 'selected' toe. In de update functie schrijf je code om te detecteren of de L-toets ingedrukt is. Als dat zo is, dan wordt selected true, in het andere geval is selected false.

Om outline rendering uit te voeren is het nodig om deze mode aan het resultaat toe te voegen. Dat kan door de volgende code aan drawprepare toe te voegen:

```
if(selected) {
   result |= IndexToFlag(RM_OUTLINE);
}
```

Maar dat is niet genoeg. We moeten nu ook de outline draw functie van `Game.Static` overschrijven. Die functie bestaat, maar is leeg. Het is dus niet nodig om `super.drawOutline()` uit te voeren.

```
virtual void drawOutline()
{
  mesh->drawOutline(BLUE, matrixScaled());
}
```

Time for Action

- ❑ Wat gebeurt er als je hierboven de functie `matrixScaled()` door `matrix()` vervangt? Waarom?
- ❑ Maak gebruik van een `Color` om de alpha waarde van de kleur te varieëren met de tijd. Hint: gebruik terug de functie `Sin()` in combinatie met `Time.appTime()`.

34.2.3 Draw Behind

Een andere render mode is `RM_BEHIND`. In deze sectie ga je zelf aan de slag om deze mode toe te passen op de Object Class `OBJ_CHEST`. De stappenlijst vermeldt de hoofdzaken. De details kan je vinden in de secties hierboven.

1. Controleer dat je assets de Object Class `OBJ_CHEST` bevat.
2. Open Assets ⇒ other ⇒ chest en kijk in de tab Params of de class correct ingesteld is.
3. Open de World en zorg dat er ten minste één chest in de wereld staat.
4. Maak een class chest, gebaseerd op `Game.Static`.
5. Voorzie een `Game.ObjMap` voor deze class.
6. Link in de `Init()` functie deze object map aan `OBJ_CHEST`.
7. Voeg aan je class een bool selected toe.
8. Voeg een update functie toe, waarin je selected true of false maakt naargelang de status van de C toets.

9. Voeg een functie `drawPrepare()` toe, waarin je de render mode `RM_BEHIND` toevoegt als selected true is.

10. Voeg de functie `drawBehind()` toe. Daarin schrijf je de volgende code:

```
mesh->drawBehind(Color(64, 128, 255, 255), Color(255, 255, 255,
    0), matrixScaled());
```

11. Experimenteer met verschillende kleuren tot je een origineel resultaat krijgt.

34.3 Mousepointer to 3D

34.3.1 Voorbeeldcode

hierboven 'selecteerden' we een object door een toets in te drukken. Daarbij zag je dat alle objecten van een type geselecteerd werden. Wanneer je slechts één object van een bepaalde class wil selecteren, dan zal je op een andere manier te werk moeten gaan. Je zou bijvoorbeeld de afstand tot de speler kunnen controleren, maar in dit deel bekijken we hoe je een object kan selecteren via de mouse pointer.

Om dat te doen moeten we als uitgangspunt de positie van de muis op het scherm nemen en een denkbeeldige lijn in de diepte van het scherm trekken. Wanneer een object deze lijn raakt, dan is er een selectie mogelijk. Deze methode is behoorlijk rekenintensief. Daarom is het geen goed idee om ze uit te voeren in de class van het object, want dan moet die lijn voor elk object van die class opnieuw gemaakt en gecontroleerd worden.

Time for Action

1. Open de applicatie '3D World - stage 3'.
2. Pas alvast de functie `update()` van de class `magicLamp` aan. De bool 'selected' dient tijdens elke update false te worden.
3. Maak een class `inputControl`, en daaronder een object van deze class.
4. Voeg aan de class een functie `void update()` toe.
5. Voer deze update functie uit in de main `Update()` functie. Het maakt niet zoveel uit waar je de functie plaatst, maar het moet in ieder geval na `Game.World.update()` gebeuren. (Die voert immers de update functie van elk object uit, en daar zet je selected steeds op false.)

We beginnen met een 'mouseover' effect. Dat wil zeggen dat we eenvoudigweg controleren of een muis over een lamp beweegt, zonder extra controles. Voeg in de update functie van inputControl de volgende code toe:

```
void update()
{
   // stop if a gui element is clicked
   if ((Gui.msLit() != null) && (Gui.msLit()->type() != GO_DESKTOP))
      return;

   Vec pos, dir;
   ScreenToPosDir(Ms.pos(), pos, dir);

   PhysHit selector;
   if (Physics.ray(pos, dir * D.viewRange(), &selector))
   {
      if (magicLamp * lamp = CAST(magicLamp, selector.obj))
      {
         lamp.selected = true;
      }
   }

}
```

Wat betekent dit allemaal? Eerst staat er deze regel:

```
if ((Gui.msLit() != null) && (Gui.msLit()->type() != GO_DESKTOP)) return;
```

Op dit moment heeft je project nog geen Gui. Maar als dat wel zo zou zijn, dan wil je geen objecten selecteren als je bijvoorbeeld op een button klikt. Het probleem is dat onder die gui nog altijd je wereld zit, dus een mouseclick zou zonder deze regel zowel de gui als de selectie in de wereld beïnvloeden. Deze regel controleert eerst of er een Gui element actief is. Wanneer het type van dat element gelijk is aan `GO_DESKTOP` dan klik je wat de gui betreft op een leeg element, met andere woorden de wereld. De regel hierboven zorgt ervoor dat de rest van de functie niet uitgevoerd wordt wanneer je niet op die desktop klikt.

```
Vec pos, dir;
ScreenToPosDir(Ms.pos(), pos, dir);
```

Hier maken we twee vectoren die nodig zijn om een denkbeeldige lijn door de wereld te trekken. We gebruiken de functie `ScreenToPosDir()` om de 2D positie van de muis om te zetten in een 3D startpositie en een richting waarin de lijn moet lopen.

```
PhysHit selector;
```

De functie die de denkbeeldige lijn controleert, moet je laten weten wat er geraakt werd. Dat gebeurt via dit object.

```
if (Physics.ray(pos, dir * D.viewRange(), &selector))
```

Hier vraag je aan de Physics engine om een lijn (ray) te trekken. Deze functie heeft 3 argumenten:

pos de startpositie.

dir de richting waarin de lijn moet lopen. We willen geen oneindig lange lijn. Dat zou betekenen dat Physics oneindig ver moet blijven controleren op objecten wanneer er geen object op de lijn zit. Daarom vermenigvuldigen we de richting met de viewRange. De viewRange is de afstand tot waar objecten in je game zichtbaar zijn.

selector dit is een referentie naar het `PhysHit` object dat je hierboven maakte. Physics zal het eerste object dat op deze lijn staat doorgeven aan dit object.

```
if (magicLamp * lamp = CAST(magicLamp, selector.obj))
```

Wanneer Physics iets gevonden heeft, dan zit er een object in de selector. Maar we weten nog niet welk object. Het zou kunnen dat het terrein werd gevonden, of eender welk object in je game. Daarom proberen we het object om te zetten (te 'casten') naar een (pointer naar een) object van het de class `magicLamp`. Als dat lukt, dan weet je zeker dat je met een object van het juiste type te maken hebt.

```
lamp.selected = true;
```

Wanneer een lamp gevonden werd, dan wijzigen we de waarde van selected. Hierboven heb je tijdens elke update alle lampen de waarde `selected = false` gegeven. Door deze functie na de wereld update uit te voeren, wordt enkel de lamp waar nu de muis over zit, true.

34.3.2 Het werkt niet!

Wanneer je nu je app uitvoert, dan zal je merken dat de selectie niet werkt. Om te onderzoeken wat er fout gaat, kijk je in zo'n geval best de Physics na. Dat kan eenvoudig door een regel toe te voegen aan de `Draw()` functie:

```
void Draw() {
   Renderer(Render);
   if(Kb.b(KB_EQUAL)) Physics.draw();
}
```

Als je nu opnieuw je programma uitvoert en door de wereld loopt, dan zie je dat de contouren van alle Physics objecten zichtbaar zijn wanneer je op de = toets drukt. Deze code wil je niet in je uiteindelijke game, maar tijdens de ontwikkeling kan dit wel erg handig zijn. Loop nu opnieuw naar de lamp, en je zal zien dat de Physics ontbreken.

Wanneer je een object importeert in Esenthel, dan wordt er niet vanzelf een Physics object gemaakt. Je kan dat wel eenvoudig zelf doen. Je opent het object in de editor en kiest de tab 'Physics'. Rechts zie je de verschillende opties. Je kan zelf de meest geschikte vorm kiezen, maar denk er wel aan dat meer detail altijd meer rekenkracht vergt.

Eens je object van physics voorzien is, zal de bovenstaande code zonder problemen werken.

Time for Action

- ❑ Voer de Physics ray enkel uit wanneer de L-toets ingedrukt werd.
- ❑ Pas de zoekafstand aan, zodat een lamp enkel gevonden wordt tot op 10 meter afstand.

HOOFDSTUK 35

Object Parameters

De game world bevat ook enkele runestones. Tot hier toe zijn die niet zichtbaar in de game. Selecteer in de game world een runestone en inspecteer de Object Class. Een runestone heeft als class `OBJ_INTERACTIVE`. Ondertussen weet je dat je in je code een class moet voorzien om deze objecten te laden.

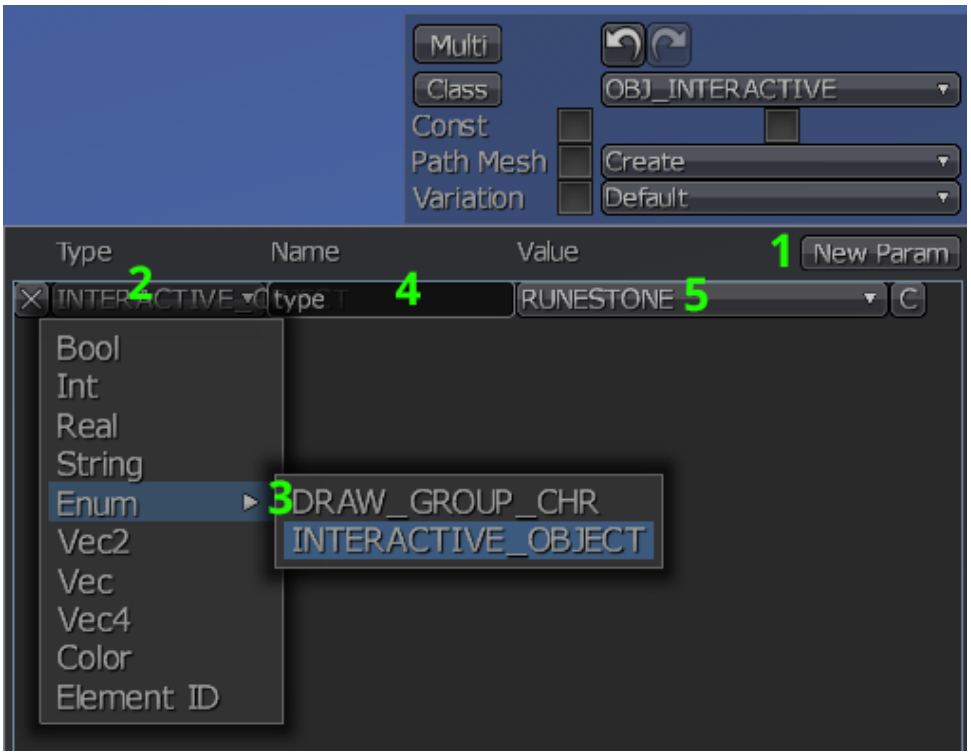
Time for Action

Begin deze oefening met de applicatie '3D World - stage 4'. Zorg dat de runestones zichtbaar zijn in je game. Je voegt de class `interactiveObject` toe, samen met een `ObjMap` en een extra regel bij het laden van de game world. Als je niet meer precies weet wat je moet doen, kijk dan in hoofdstuk 34.1.2.
Open daarna de game en controleer of de runestones zichtbaar zijn.

Object classes in code zijn een enumeratie. Dat betekent dat je maximum 256 verschillende waarden kan gebruiken. Als je game wat groter wordt, dan is dat waarschijnlijk te weinig om alle objecten in je game een eigen object class te geven. Daarom zal je dikwijls een hele groep van objecten van dezelfde object class voorzien. Het onderscheid maak je dan met parameters.

Time for Action

1. Voeg in de editor een nieuwe enumeratie toe in de folder `enums`, met de naam `INTERACTIVE_OBJECT`.
2. Voeg aan de enumeratie de waarde `RUNESTONE` toe.
3. Open de object class `OBJ_INTERACTIVE`.
4. Druk rechts op 'New Param'.
5. Kies als Type Enum⇒INTERACTIVE_OBJECT
6. Geef als Name 'Type' in.
7. Kies als Value 'RUNESTONE'



Figuur 35.1: Custom Parameters for an object class.

Je hebt nu een parameter toegevoegd aan je object class. In dit geval was dat een enumeratie, maar je ziet dat je eender welke variabele kan maken. In je code heb je toegang tot deze waarden, dus je kan ze gebruiken om het gedrag van je object aan te passen.

Time for Action

1. Open Assets ⇒ other ⇒ runestone en kies de tab 'Params'. De parameter type werd overgenomen van de object class.
2. Voeg nog een parameter toe aan je runestone. Deze keer kies je het type 'Int', Name 'ID' en Value '0'.
3. Open de world editor en kies het tabblad 'Object'. Selecteer daarna een runestone.
4. Controleer of elke runestone de gewenste parameters bevat.
5. Pas de waarde van ID voor elke runestone aan, zodat ze de ID's van 0 tot 3 bevatten.

Note

Je ziet dat er een vinkje verschijnt voor een parameter wanneer je die een nieuwe waarde geeft. Daaraan zie je dat deze waarde afwijkt van de oorspronkelijke waarde. Het gebruik van parameters is ietwat gelijk aan het principe van overerving. Je kan parameters ingeven op het niveau van de object class, op het niveau van het object, of op het niveau van de instantie van het object. Elk niveau kan het vorige niveau overschrijven.
Het is ook mogelijk om alle parameters op het niveau van de instantie te schrijven. Dat betekent echter meer werk en een grotere kans op fouten.

35.1 Retrieving Object Params in Code

Om de object parameters in code te gebruiken, dien je de functie `create` van de base class `Game.Static` te overschrijven. Daarin voer je eerst die base class functie uit. Om de parameters in code op te slaan voeg je ook twee variabelen toe. Je class ziet er nu zo uit.

```
class interactiveObject : Game.Static
{
private:
    INTERACTIVE_OBJECT iObjType = IO_NONE;
    int ID = 0;

public:
    virtual void create(Object &obj)
    {
        super.create(obj);

        // every interactive object has a Type parameter
        iObjType = obj.getParam("Type").asEnum();

        if(iObjType == IO_RUNESTONE)
        {
```

```
            // every runestone has an ID parameter
            ID = obj.getParam("ID").asInt();
        }
    }
}

Game.ObjMap<interactiveObject> InteractiveObjects;
```

Zoals je ziet heeft `Object` een lidfunctie `getParam` die je toegang geeft tot de parameters. Aangezien de code niet weet welk type je parameter heeft, moet je zelf aangeven hoe je de parameter aan je code wil doorgeven. Dat kan via functies zoals `asEnum()` of `asInt()`.

35.2 Particles

Om de runestones wat meer uitstraling te geven voegen we aan elke runestone een particle effect toe. Voeg eerst de volgende variabelen toe aan de `interactiveObject` class:

```
Game.ObjParticles fx;
Vec fxPos;
```

Particles zijn kleine afbeeldingen waarvan je de beweging en de levensduur kan instellen. In een game worden ze vaak gebruikt om een vuur te tonen, of als visualisatie van een spell. In Esenthel is een particle een game object met als base class `OBJ_PARTICLE`. Wanneer je bij de parameters van een object deze base class instelt, krijg je automatisch toegang tot alle parameters van een particle.

> **Note**
>
> De object class `OBJ_PARTICLE` moet dan wel al die parameters bevatten. Als je aan een eigen project werkt, dan kan je deze class best copiëren van een voorbeeldproject.

Het oefenproject bij dit hoofdstuk bevat al een uitgewerkte particle in Assets⇒other⇒runestone⇒particle. We zullen deze particle in het object fx laden. Voor we dat doen zullen we eerst een geschikte positie moeten vinden. Het is de bedoeling dat de particles zichtbaar zijn in de holle opening van de runestone.

Met die reden is aan het runestone object een 'slot' toegevoegd. Je kan dat nakijken door het object in de editor te openen en de tab 'slots' te kiezen. Selecteer de optie om een positie aan te passen en hover over het aanwezige slot. Je zal zien dat dit slot een naam heeft: 'particle'. Via deze naam kunnen we de positie van het slot opvragen. Aangezien we in de world editor ook elke runestone een andere schaal zouden kunnen geven, is het belangrijk dat deze positie geschaald wordt.

Ook belangrijk om te onthouden is dat het om een relatieve positie gaat. Het slot heeft geen positie in de wereld, maar een offset ten opzichte van de positie van de runestone. In dit geval komt dat goed uit. We geven fx ook de positie van de runestone, maar de bron van de particles moet relatief zijn ten opzichte van die positie. Als bron stellen we een `Ball` in die de positie van het slot krijgt.

De volledige create functie ziet er dan zo uit:

```
virtual void create(Object &obj)
{
  super.create(obj);

  iObjType = (INTERACTIVE_OBJECT)obj.getParam("Type").asEnum();
  if(iObjType == IO_RUNESTONE)
  {
     ID = obj.getParam("ID").asInt();

     if(mesh->skeleton().findPoint(8"particle") != null)
     {
        fxPos = mesh->skeleton().findPoint(8"particle").pos * scale;
        fx.create(*Objects(=== drop particle object here ===));
        fx.pos(pos());
        fx.particles.source(Ball(0.1, fxPos));
     }
  }
}
```

Time for Action

Voeg deze code toe aan je project.

Particles zijn bewegende afbeeldingen en moeten ook geupdated worden. Daarvoor heeft de class `Game.ObjParticles` een update functie. Het volstaat om deze functie uit te voeren. Maar omdat de `interactiveObject` class niet enkel voor runestones dient, doen we dit enkel wanneer `iObjType` een runestone is:

```
virtual Bool update()
{
  if(iObjType == IO_RUNESTONE)
  {
     // update the particles
     fx.update();
  }

  return super.update();
}
```

Particles worden gerendered in een afzonderlijke draw functie. Afhankelijk van het soort particles heb je één van de volgende render modes nodig:

- ❑ RM_BLEND
- ❑ RM_PALETTE
- ❑ RM_PALETTE1

In de **drawPrepare** functie moet je daarom de gewenste render mode toevoegen. In dit geval gebruiken we RM_PALETTE.

```
virtual UInt drawPrepare()
{
  uint result = super.drawPrepare();
  if(iObjType == IO_RUNESTONE)
  {
     result |= IndexToFlag(RM_PALETTE);
  }
  return result;
}
```

Tot slot moet je ook de functie **drawPalette** overschrijven:

```
virtual void drawPalette()
{
   fx.drawPalette();
}
```

Time for Action

Voeg alle bovenstaande code toe aan je project. Voer je game uit en controleer of je de particles op het scherm ziet.

35.3 Licht

Een andere techniek die de game wat meer sfeer kan geven is het toevoegen van lichtbronnen. Hier moet je wel voorzichtig mee omgaan. Voor elke lichtbron moeten immers ook schaduwen berekend worden. Een scene met veel lichtbronnen kan bijzonder zwaar zijn voor een oudere computer. Ook mobile games zijn meestal beperkt tot één enkele bron.

Een lichtbron toevoegen is alvast heel eenvoudig. Je hebt enkel een positie en een kleur nodig. (Al zijn er extra argumenten om ook de intensiteit en de maximum afstand te bepalen, maar hier gebruik je de defaults.)

Je voegt deze code toe aan de **drawPrepare** functie:

```
if(iObjType == IO_RUNESTONE)
{
   LightPoint(1, pos() + fxPos, GREEN.asVec()).add();
   result |= IndexToFlag(RM_PALETTE);
}
```

Time for Action

Experimenteer met de instellingen van de lichtbron. Wijzig ook de parameters van het particle object, zodat je een beter zicht krijgt op de betekenis van de parameters.

35.4 Toggle the Runes

Op dit moment zijn de particles en het licht steeds actief. Nu ga je er voor zorgen dat je ze in en uit kan schakelen door er op te klikken. Een groot deel van deze code ken je al uit de vorige secties. Zo zullen we eerst een outline toevoegen wanneer je muis over een runestone hovert.

1. Voeg eerst deze variabelen toe aan de `interactiveObject` class:

   ```
   bool belowMouse = false;
   bool active = false;
   ```

2. In de `update` functie zorg je er voor dat `belowMouse` steeds false wordt.

3. Je voegt daarna de outline render mode toe in `drawPrepare`, maar enkel waneer `belowMouse` de waarde true heeft.

4. zorg er ook voor dat het `LightPoint` en de particles enkel getoond worden wanneer 'active' true is.

5. Nu overschrijf je de `drawOutline` functie. Als je niet zeker bent, kan je spieken in de `magicLamp` class.

6. Tot slot voeg je enkele eenvoudige functies toe die je later van pas zullen komen. Je ziet hieronder de declaraties. Je kan deze functies (telkens precies één statement) zeker zelf uitwerken.

   ```
   bool isRuneStone   () { ???; }
   void setBelowMouse () { ???; }
   void toggleActive  () { ???; }
   void deactivate    () { ???; }
   bool isActive      () { ???; }
   int  getID         () { ???; }
   Vec  getParticlePos() { ???; }
   ```

In de class **inputControl** kan je nu een extra controle op de physics selector uitvoeren:

```
if (interactiveObject  * obj = CAST(interactiveObject, selector.obj))
{
  if(obj.isRuneStone())
  {
     obj.setBelowMouse();
     if(Ms.bp(0))
     {
        obj.toggleActive();
     }
  }
}
```

Time for Action

Wanneer je alle code toegevoegd hebt, voer je de game uit en controleer je of alles werkt.

35.5 A Puzzle

In het laatste deel van dit hoofdstuk maak je een puzzel. Het is de bedoeling de 4 runestones in de juiste volgorde activeert. Als ze alle vier actief zijn, komt het hoofd in het midden naar boven. Ook toon je een laser effect tussen de actieve runestones. Je start met een nieuwe class **puzzle**:

```
class puzzle {
private:
  Mems<interactiveObject *> objects;
  Mems<Vec> laserPoints;
  Sound sound;

public:

}
puzzle Puzzle;
```

De **objects** container bevat pointers naar de aanwezige runestones in de game wereld. **laserPoints** zal de punten bevatten waartussen en laser getoond moet worden. En tot slot is er een **sound** om alles wat spannender te maken.

Je voegt nu in het private deel van de class de volgende functies toe:

```
interactiveObject * getObjectWithID(int ID)
{
  REPA(objects)
```

```
  {
5     if(objects[i].getID() == ID) return objects[i];
  }

  return null;
}
10
void turnOffStones()
{
  REPA(objects) objects[i].deactivate();
}
```

Deze functies maken de volgende code eenvoudiger:

getObjectWithID zoek de runestone met het gevraagde ID. De volgorde in de container is immers niet gelijk aan de volgorde van de ID's.

turnOffStones deactiveert alle runestones in de container.

De volgende functies die je aanmaakt zorgen er voor dat je runestones kan toevoegen en verwijderen uit de container. Je moet er namelijk rekening mee houden dat Esenthel oneindig grote werelden ondersteunt. Dat betekent dat niet de hele wereld steeds in het geheugen zit. World Objects kunnen geladen en terug verwijderd worden terwijl je de speler verplaatst doorheen de game world.

```
void registerStone(interactiveObject * obj)
{
  objects.add(obj);
}
5
void unregisterStone(interactiveObject * obj)
{
  REPA(objects)
  {
10    if(objects[i] == obj)
      {
        objects.remove(i);
        return;
      }
15  }
}
```

Nu komt het er op aan om deze functies op de juiste plaats te gebruiken. We laten elke runestone zichzelf registeren bij de puzzel. Dat kan het best in de `create` functie van interactiveObject. Zoek zelf even naar de meest geschikte plaats binnen deze functie en voeg de volgende code toe:

```
Puzzle.registerStone(this);
```

Het object moet zich ook verwijderen uit de puzzel wanneer het niet langer actief is. Nu bestaat er geen functie binnen Game.Static die uitgevoerd wordt wanneer dit het geval is. Maar je kan wel kijken wat het resultaat is van update. Bij 'false' zal de game engine het object verwijderen. Het komt er dus op aan om net ervoor de runestone uit de puzzel te verwijderen. Op het eind van de update functie schreven we tot hiertoe steeds:

```
return super.update();
```

Time for Action

Herschrijf de bovenstaande regel zo dat je het resultaat van `super.update()` kan gebruiken om het object te verwijderen als dat nodig is.

De echte logica van de puzzel zit in de update functie. We overlopen stap voor stap wat hier moet gebeuren. Eerst wordt de container met punten voor de laser leeggemaakt. En wanneer er geen 4 runestones geladen zijn heeft het in elk geval geen zin om de puzzel te controleren.

```
void update() {
  laserPoints.clear();
  if(objects.elms() != 4) return;
}
```

Vervolgens controleren we de huidige stand van zaken. We kunnen er van uit gaan dat je nooit twee runestones op hetzelfde moment kan inschakelen. Daarom is het mogelijk om de volgende logica toe te passen:

1. Stel de hoogst actieve ID gelijk aan nul.
2. Ga alle runes af, beginnend met het hoogste ID.
3. Indien dat ID actief is en de huidige hoogst actieve ID kleiner dan deze, wordt dit de hoogst actieve ID.
4. Indien de ID kleiner is dan de hoogst actieve ID en niet actief is, dan is de puzzel ongeldig en stop je de loop.

De code ziet er zo uit:

```
int highestActive = 0;
bool invalid = false;
for(int i = 3; i >= 0;  i--)
{
  if(highestActive < i && getObjectWithID(i).isActive())
  {
     highestActive = i;
  } else if(highestActive > i && !getObjectWithID(i).isActive())
  {
```

```
      invalid = true;
      break;
    }
}
```

In de volgende stap zijn er twee mogelijkheden: Invalid kan true of false zijn. Wanneer de puzzel ongeldig is, dan kunnen we eenvoudigweg alle stenen uitschakelen. Dat zorgt er voor dat de speler terug opnieuw moet beginnen.

In het andere geval moet je de posities van de actieve stenen toevoegen aan de laserPoints container, maar enkel wanneer de hoogst actieve positie groter is dan 0. (Met slechts 1 actieve positie kan je toch geen lijn maken.)

Wanneer alle stenen actief zijn, dan voeg je op het eind nogmaals de eerste positie toe, zodat de 'cirkel' gesloten is.

```
if(invalid)
{
  turnOffStones();
} else if(highestActive > 0)
{
  // add points to laser
  for(int i = 0; i <= highestActive; i++)
  {
    laserPoints.add(getObjectWithID(i).getParticlePos());
  }

  // close trajectory if all stones are on
  if(highestActive == 3)
  {
    laserPoints.add(getObjectWithID(0).getParticlePos());
  }
}
```

Op het eind van de update functie kan je zelf code toevoegen de nodige geluiden te spelen. Je vertrekt van de volgende regels:

- ❏ Wanneer laserPoints ten minste één element bevat en **sound** nog niet speelt, dan start je het geluid 'erie_ring' in een loop.
- ❏ Wanneer laserPoints leeg is en het **sound** wél speelt, dan stop je het geluid met een fadeout, en je speelt het geluid 'roar' (zonder loop, via de functie **SoundPlay**).

De laatste twee functies in deze class zijn weer eenvoudig. De **draw** functie toont de laser op het scherm en de **solved** functie laat weten of de puzzel al dan niet opgelost is.

```
void draw()
{
  if(laserPoints.elms() > 0)
  {
     DrawLaser(GREEN, WHITE, 0.01, 0.03, false, laserPoints);
  }
}

bool solved()
{
  return laserPoints.elms() == 5;
}
```

Omdat Puzzle geen deel uitmaakt van de Game World en toch een draw functie heeft, moet je die zelf uitvoeren. De declaratie van DrawLaser vertelt je dat er twee render modes nodig zijn: RM_SOLID en RM_AMBIENT. Je voegt die toe aan de **Render** functie in het bestand 'main'.

```
void Render()
{
   Game.World.draw();

   switch(Renderer())
   {
      case RM_SOLID:
      {
         Puzzle.draw();
         break;
      }

      case RM_AMBIENT:
      {
         Puzzle.draw();
         break;
      }
   }
}
```

Tot slot is het de bedoeling om het hoofd tussen de runes te tonen wanneer de puzzel opgelost is. Het hoofd wordt ook geladen als **interactiveObject**, maar heeft als **iObjType** IO_HEAD.

Voeg eerst een variabele toe aan **interactiveObject**.

```
float origHeadPosY;
```

Nu kan je die oorspronkelijke Y positie van het hoofd onthouden in de **create** functie, om daarna het hoofd 2 units naar beneden te plaatsen:

```
if(iObjType == IO_HEAD)
```

```
{
  origHeadPosY = pos().y;
  Vec newPos = pos();
  newPos.y -= 2;
  pos(newPos);
}
```

Daarna kan je in de update functie het hoofd verplaatsen wanneer nodig:

```
if(iObjType == IO_HEAD)
{
  if(Puzzle.solved() && pos().y < origHeadPosY)
  {
     your code here!
  }
  else if(!Puzzle.solved() && pos().y > origHeadPosY - 2)
  {
   .. and here!
  }
}
```

Time for Action

Vul de bovenstaande code verder aan en test of alles werkt.

HOOFDSTUK 36

Animations

De class `Game.Chr` voorziet een aantal standaard animaties. Die zijn echter zelden genoeg. In dit hoofdstuk zie je hoe je deze animaties aanpast, en hoe je nieuwe animaties toevoegt.

Esenthel maakt een onderscheid tussen standaard animaties and custom animaties. De standaard animaties worden automatisch toegepast aan de hand van enkele variabelen in `Game.Chr`. Als je de header file van deze class opent, dan zie je binnen de class een struct `Input`. Deze class regelt hoe je avatar over het scherm beweegt. Je ziet properties zoals crouch, walk, jump en dodge.

Het is eenvoudig om deze waarden aan te passen. Je hebt dat trouwens al gedaan in een van de vorige hoofdstukken. Denk aan code zoals:

```
input.turn.x = Kb.b(KB_Q) - Kb.b(KB_E);
input.turn.y = Kb.b(KB_T) - Kb.b(KB_G);
input.move.x = Kb.b(KB_D) - Kb.b(KB_A);
```

Time for Action

De avatar heeft, naast een run animatie, ook een walk animatie. Die wordt op dit moment niet gebruikt. Pas de code in `player.update` aan zodat je avatar standaard wandelt, maar wel loopt wanneer de linker ctrl toets ingedrukt is. Zoek in de struct `Input` op welke property je hier voor aanpast.

36.1 Jump

Vreemd genoeg bevat **Input** wel een property jump die de avatar tijdelijk omhoog beweegt, maar bestaat er geen standaard animatie voor jump. Dat wil zeggen dat je een custom animatie moet gebruiken. Om dit te doen heb je een **Motion** object nodig. Voeg alvast dit object toe aan de class **Player**.

```
Motion jumpMotion;
```

In de functie **update** kan je de huidige 'jump' code aanvullen:

```
// jumping
input.jump = Kb.bp(KB_SPACE) ?  3.5 :  0;
if(Kb.bp(KB_SPACE))
{
  jumpMotion.set(skel, === drop jump animation ===);
}
jumpMotion.updateAuto(5, 5, 1);
```

Het eerste statement stond al in je code. Dat bepaalt of je avatar al dan niet even omhoog gaat. Daarna stel je de animatie in wanneer de speler op de spatiebalk drukt. Het eerste argument is het skeleton waarop de animatie van toepassing is. De waarde 'skel' is skeleton dat al aanwezig is in **Game.Chr**. Het tweede argument is de verwijzing naar de animatie. Drop daar de jump animatie die je vindt in Assets ⇒ characters ⇒ samurai.

Tijdens elke update moeten ook alle **Motion** object geupdated worden. In dit geval is dat jumpMotion. Je kan de **updateAuto** functie gebruiken om het eenvoudig te houden. De eerste twee argumenten bepalen hoe snel je overschakelt van de standaard animatie naar de jump animatie en omgekeerd. Het derde argument is de algemene snelheid waarmee de animatie getoond wordt.

Wanneer je nu de game uitvoert, zal je zien dat de jump animatie niet gebruikt wordt. We moeten eerst de aanwezige functie **animate** van **Game.Chr** overschrijven. Die functie voert nu enkel de standaard animaties uit. Omdat dit een virtuele functie is (kijk in de header file!) moet je daarin ook de oorspronkelijke functie uitvoeren. *(Tenzij je echt niet wil dat de standaard animaties gebruikt worden.)* Voeg daarom de volgende functie toe aan je player class:

```
virtual void animate()
{
  super.animate();
  skel.animate(jumpMotion, true);
}
```

Deze functie voert eerst de standaard animaties uit. Daarna wordt de **jumpMotion** toegevoegd. Het extra argument 'replace' bepaalt dat de vorige animaties overschreven moeten worden. Wanneer je 'false' gebruikt, zal de impact van de vorige animaties veel groter zijn.

Time for Action

Pas de waarden in `updateAuto` aan een bekijk het resultaat. Bekijk ook hoe anders de animatie is wanneer je `skel.animate(jumpMotion, false)` gebruikt.

HOOFDSTUK 37

Dynamic Objects

In dit hoofdstuk leer je hoe je met tijdelijke items in je game world omgaat. In het algemeen heb je hier twee mogelijkheden:

1. items die altijd aanwezig, maar slechts af en toe zichtbaar zijn.
2. items die je zelf aan de game world toevoegt via code en nadien ook terug verwijdert.

Welk van deze mogelijkheden je gebruikt, hangt af van de situatie. Wil je bijvoorbeeld een power-up op een bepaalde locatie die verdwijnt wanneer de speler hem gebruikt, maar enkele minuten later terug verschijnt, dan gebruik je de eerste optie. Maar wanneer je een voorwerp op eender welke locatie kan opnemen en terug achterlaten, dan is de tweede optie beter.