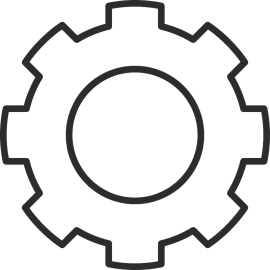
**Get it in, bro!**

**Ontwikkeld door Kodi Coding**

**C:\Users\Dion\Documents\Visual Studio 2010\Projects\Getitinbro\Press Kit\logo.jpg**

**Technisch ontwerp**

Inhoud

[Introductie 3](#_Toc373160971)

[Projectopdracht 3](#_Toc373160972)

[Doel van de mobile game 3](#_Toc373160973)

[Obstakels 4](#_Toc373160974)

[Ontwikkelmethode 4](#_Toc373160975)

[Multi platform 6](#_Toc373160976)

[Graphics Engine 7](#_Toc373160977)

[Physics Engine 8](#_Toc373160978)

[Momentum & valversnelling & Angular momentum 8](#_Toc373160979)

[Collision Detection 9](#_Toc373160980)

[Opslagstructuur 10](#_Toc373160981)

# **Introductie**

Kodi Coding is een onofficieel bedrijf opgericht door Koen Klever en Dion van ’t Veer. Dit team, bestaande uit twee studenten Technische Informatica op de Hogeschool Rotterdam, is een team van twee softwareontwikkelaars met ervaring in C#, Java, Pascal, Visual Basic, HTML, JavaScript, PHP, ASP.NET, C, en C++.

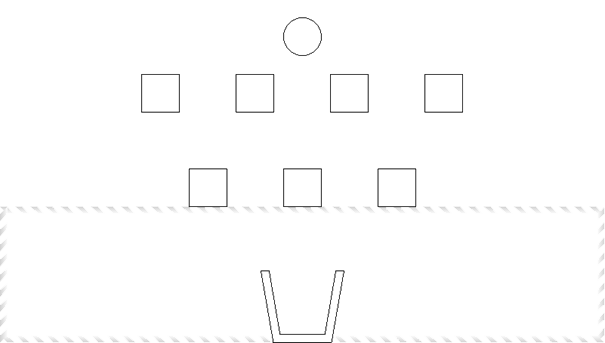
# **Projectopdracht**

In het eerste kwartaal van de minor Game Design en Development is ons een opdracht toegewezen. In deze opdracht werd ons gevraagd binnen 6 weken een retrogame een nieuwe draai te geven en te publiceren op een mobile device.

Wij hebben deze opdracht aangenomen en na overleg besloten een remake te maken van het onder de meesten bekende spel ‘Line Rider’. Aangezien ons team bestaat uit twee softwareontwikkelaars en niet uit grafische ontwerpers, hebben we ervoor gekozen het grafische ontwerp simpel te houden. We hebben ons als opdracht opgelegd om een werkende ‘physics engine’ te maken. Dit houd in dat we zelf een wereld creëren en hierin zwaartekracht, wrijving en andere natuurkundige wetten verwerken.

# Doel van de mobile game

Zodra de spelomgeving wordt gestart, krijgt de gebruiker een ‘wereld’ te zien met hierin een bal, een emmer en obstakels. Het doel van het spel is om de gebruiker zelf een weg te laten tekenen, om zo de bal in de emmer te krijgen. In figuur 1 is een voorbeeld van een (simpel lijkend) level.



Figuur 1: Een simpel lijkend level.

# Obstakels

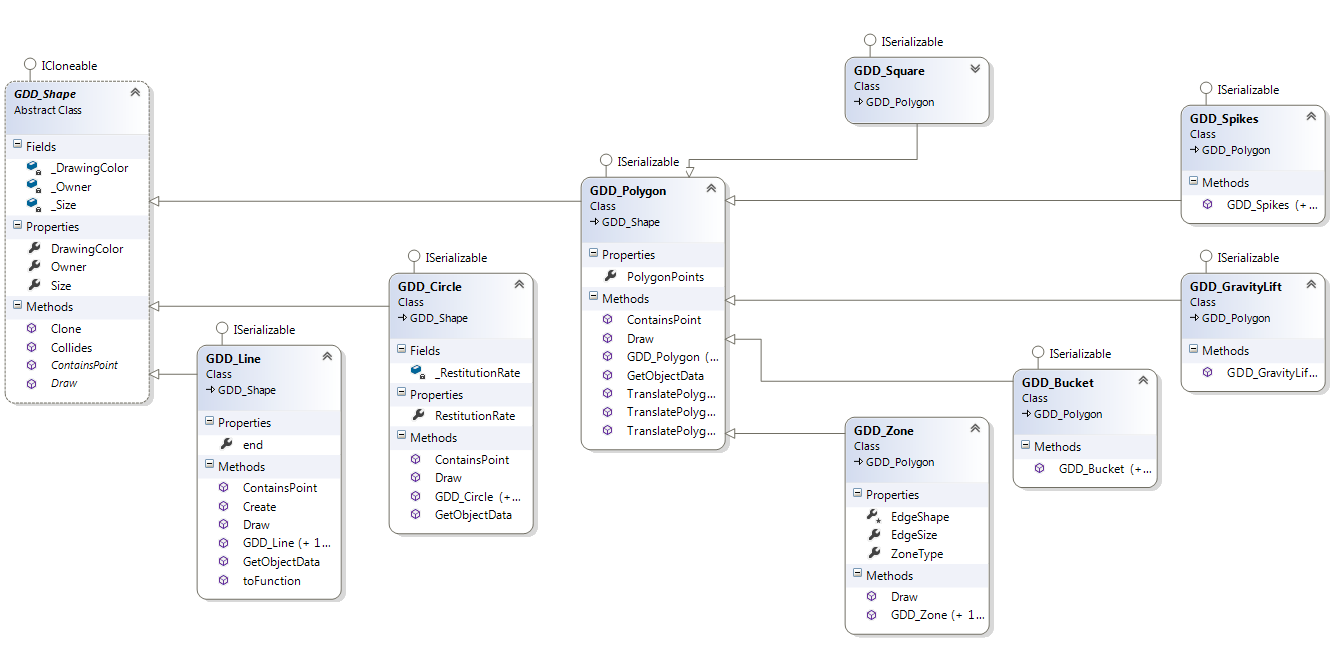
De obstakels in deze mobile game zijn simpel weergegeven. Onderstaand staat een lijst met de te implementeren obstakels.

* ‘Muren’ in de vorm van vierhoeken welke niet gepasseerd kunnen worden.
* ‘Spikes’ welke bij aanraking de bal lek prikt.
* ‘Gravity Lift’ welke de bal in een bepaalde richting schiet.
* ‘No-Draw zone’ waar niet in getekend kan worden.

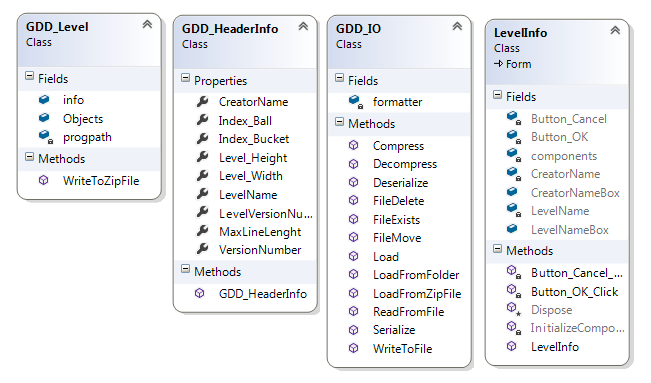
In Figuur 1 zijn de ‘muren’ en ‘No-Draw zone te zien’.

# Ontwikkelmethode

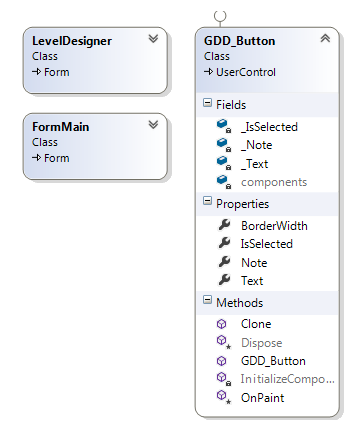
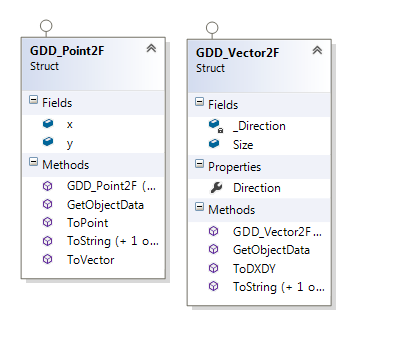
In dit project is gekozen om te ontwikkelen in C#, met als ontwikkelomgeving Visual Studio 2012. Deze keuze geeft de mogelijkheid de mobile game te laten draaien op elk apparaat dat Windows XP of hoger draait, inclusief tablets. In Figuur 2.1 tot en met Figuur 2.7 zijn klasse-diagrammen te zien, welke de structuur van de broncode bevat.



Figuur 2.1: De klasse-diagram van Shapes en zijn afgeleiden

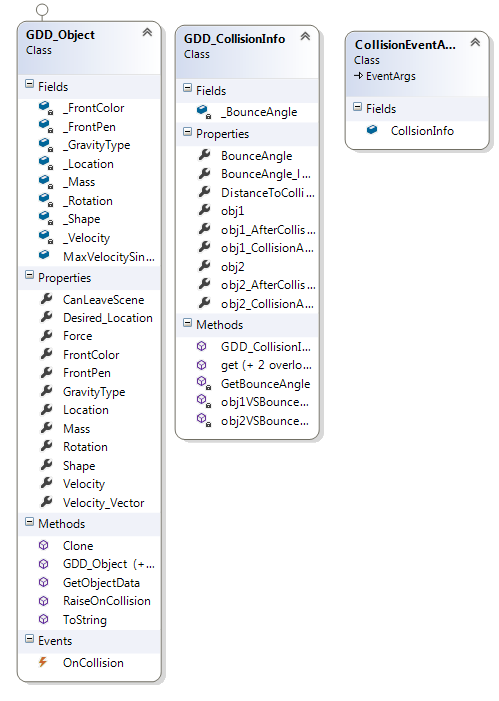
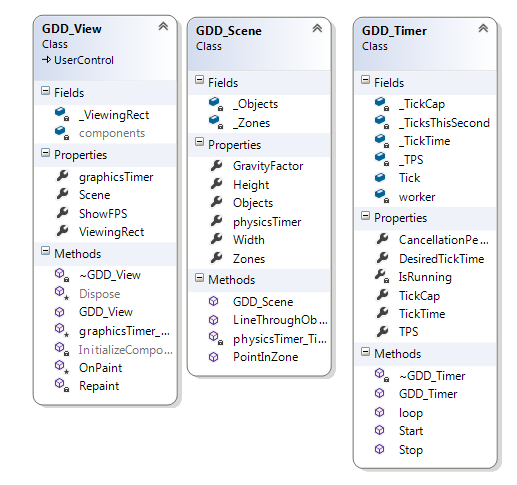


Figuur 2.2: De klasse diagrammen van de Level-klassen



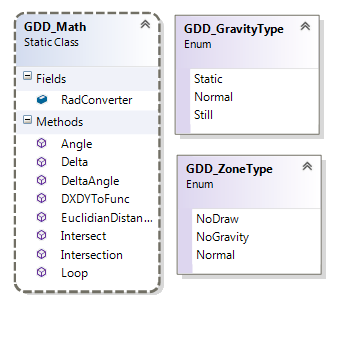
Figuur 2.3: De Types

Figuur 2.4: De zelfgemaakte User Controls



Figuur 2.6: De klassen waarin wordt getekend

Figuur 2.5: De klasse Object en Collision



Figuur 2.7: Overige klassen

# Multi platform

Om te zorgen dat er (in de toekomst) meerdere platformen ondersteund wordt hebben we onze eigen types gemaakt die de basis vormen van de Physics / Graphics engine zie hiervoor Figuur 2.3

# Graphics Engine

De graphics engine achter ‘Get it in bro’ is zelf geschreven en wordt getekend door middel van een graphics object die standaard geleverd wordt met C#.

GDD\_Objects zijn de fundering van zowel de graphics als physics engine. De GDD\_Objects zijn verantwoordelijk voor de locatie, size en rotatie terwijl ‘GDD\_Shape’ verantwoordelijk zijn voor de vorm van het object ( Zie Figuur 2.1 )

GDD\_Shapes hebben een ‘Draw’ functie en zal zichzelf tekenen op het Graphics object dat een parameter is van deze functie en d.m.v. het bovenliggende GDD\_Object. Veel GDD\_Shapes bestaan uit Polygons; een verzameling aan punten die met elkaar verbonden worden. Hieronder is een voorbeeld hoe zo’n polygon getekend wordt:

/// <summary>

/// Draws this shape on a Graphics g

/// </summary>

/// <param name="G"></param>

public override void Draw(Graphics G)

{

//Getting a translated polygon

PointF[] poly = TranslatePolygonPoints(Owner.Rotation.Direction, Size / 100f, Owner.Location);

//Draws the shape using the polygon data

G.FillPolygon(DrawingColor, poly);

G.DrawPolygon(Owner.FrontPen, poly);

}

Op deze manier kunnen we door alle objecten itereren en ze allemaal tekenen:

//Creating graphics

Graphics g = Graphics.FromImage(b);

//Looping each GDD\_Object and drawing them

for (int i = 0; i < this.Scene.Objects.Count; i++)

{

GDD\_Object obj = this.Scene.Objects[i];

Obj.Draw(g);

}

# Physics Engine

De phycis engine achter ‘Get it in bro’ is zelf geschreven en kan doordat heel veel GDD\_Shapes uit polygons bestaan kan er vrij gemakkelijk mee gerekend worden. Twee belanrijke factoren uit de physics engine zijn het toepassen van snelheid/kracht en het berekenen van collisions ( botsingen )

## Momentum & valversnelling & Angular momentum

Snelheid wordt op twee manier gedefinieert in de physics engine; als een DeltaX&DeltaY en als een vector, omdat de snelheid ook als DeltaX&DeltaY wordt opgeslagen kunnen we elke frame gemakkelijk de valversnelling toepassen en dus een nieuwe momentum en een gewenste locatie berekenen die vervolgens gebruikt wordt voor de ‘Collision Detection’:

//Doing a calculation which spits out ‘d’; a factor of seconds (0.05 equals 1/20th of a second )

float t = graphicsTimer.TickTime;

if (t == 0) { t = graphicsTimer.DesiredTickTime; }

float d = t / 1000;

//Applying gravity

obj.Velocity = new GDD\_Point2F(obj.Velocity.x, obj.Velocity.y + (this.Scene.GravityFactor \* d));

//Determining the end location

obj.Desired\_Location = new GDD\_Point2F(obj.Location.x + (obj.Velocity.x \* d), obj.Location.y + (obj.Velocity.y \* d));

//Applying angulair momentum

obj.Rotation = new GDD\_Vector2F(obj.Rotation.Direction + (obj.Rotation.Size \* 360f) \* d, obj.Rotation.Size);

## Collision Detection

Zoals eerder genoemd, worden heel veel shapes opgemaakt uit polygons, elke polygon kan weer in lijnen opgedeeld worden. Om collision te berekenen kunnen we derhalve lijnen met elkaar laten botsen, dit vragen we de shape klasse. CollisionInfo zal ook berekenen wat er met de objecten moet gebeuren:

//List of collision

List<GDD\_CollisionInfo> Collisions = new List<GDD\_CollisionInfo>();

//Looping each object to see if it’s colliding with ‘obj’

for (int j = 0; j < this.Scene.Objects.Count; j++)

{

GDD\_Object obj1 = this.Scene.Objects[j];

//Calculating the (possible) collision

GDD\_CollisionInfo collision = GDD\_Shape.Collides(obj.Shape, obj1.Shape);

//Did we have a collision?

if (collision != null)

{

//Adding the collision

Collisions.Add(collision);

}

}

Nu we een lijst hebben met collisions, gaan we kijken of eer meerdere zijn; zo ja wordt de dichtstbijzijnde berekend door middel van de Euclidean Distance ( Pythagoras )

//Do we have more than 1 collision?

if (Collisions.Count > 1)

{

//Sorting the collisions by distance to collision

Collisions = Collisions.OrderBy(c => c.DistanceToCollision).ToList();

}

//Applying the collision with the closest object

GDD\_CollisionInfo collision = Collisions[0];

Omdat eigenlijk alleen maar de bal met lijnen botst, kan je ‘hoek van inval is gelijk aan de hoek van uitval’ regel toepassen. Dit klinkt heel makkelijk maar de implementatie kost een flink aantal regels. Voor een volledige schriftelijke uitwerking verwijs ik naar de broncode, zie de methode ‘public static GDD\_CollisionInfo get(GDD\_Circle circle, GDD\_Line line)’ bestand ‘CollisionInfo’ (regel 179)

# Opslagstructuur

De mobile game bevat een bepaalde opslagstructuur. Levels die door de gebruiker worden gemaakt in de daarvoor geleverde Level Designer worden automatisch opgeslagen in de map:

**(Game directory)/Saved levels/Custom**

Als (levelnaam).zip bestand. Van tevoren meegeleverde levels zijn te vinden in:

**(Game directory)/Levels/Chapter(chapternummer)/ch(chapternummer)lev(levelnummer).zip**

Een voorbeeld hiervan is, uitgaande van Chapter 1 en Level 1:

**(Game directory)/Levels/Chapter1/ch1lev1.zip**

Custom games worden geladen vanuit de bovenstaande map.