Taakgroep: WPF

Context

De afgelopen 6 levels heb je oprecht veel mooie dingen gemaakt. Het ging misschien niet altijd even makkelijk en was de opdracht soms niet helemaal logisch. In het werkveld is het veelal niet anders. Tegenstrijdige opdrachten waarbij elke collega wel een mening heeft hoe je het zou moeten aanpakken. Uiteindelijk zul je het toch zelf moeten doen en je keuzes moeten kunnen onderbouwen.

Ook het tijdens het schrijven van testen heb je wellicht vreemde keuzes moeten maken. Bijvoorbeeld een methode public maken terwijl die eigenlijk private zou moeten zijn. Waar doe je dan goed aan? Belangrijkste is ook hier dat je bewust bent van je keuzes en deze kan onderbouwen.

Waarschijnlijk ben je de console ondertussen wel een beetje zat. Zeker bij de vorige taak toen je ineens gevraagd werd om nog even wat extra informatie te tonen. Gelukkig ga je nu de overstap maken naar WPF en ga je de race simulator grafisch weergeven.

Gelukkig heb je alles volgens de MVC structuur gemaakt. Bij de volgende taken zal je merken hoe waardevol dit is. Je kunt je volledig richten op een nieuwe visualisatie zonder dat je bestaande code hoeft aan te passen.

Taak: First Window

In deze taak ga je een nieuw project opzetten voor de nieuwe view en het eerste scherm vormgeven.

Aanpak

De eerste stap is om je solution uit te breiden met een nieuw project. Zorg er wel voor dat het project wel in dezelfde folder geplaatst wordt bij alle andere projecten. Dit kan je aangeven bij het aanmaken van het project. Als je dit niet goed doet wordt dit project niet meegenomen in de source control.

- Breid je solution uit met een WPF App project. Let op dat je hetzelfde framework (.NET Core) en versie gebruikt hiervoor.

  Het nieuwe project heeft een aantal standaard bestanden. Bijvoorbeeld `App.xaml` en `MainWindow.xaml`. Met het bestand `App.xaml` moet je niet verwijderen of aanpassen, dit is namelijk de start van je nieuwe view. Hierin is bijvoorbeeld vastgelegd wat je startscherm zal zijn. Niet geheel verrassend is op dit moment je startscherm `MainWindow.xaml`.

  Wanneer je dubbelklikt op de naam `MainWindow.xaml` in de solution explorer opent zich de `designer`. Bovenaan is de grafische weergave van je scherm en direct daaronder de `XAML` die daarbij hoort. Aan

de linkerkant heb je de `toolbox` waar je verschillende componenten kan vinden bijvoorbeeld knoppen, labels, tabellen, etc..

In WPF kennen componenten een gelaagdheid vergelijkbaar met JAVA. Een component kan 1 of meerdere subcomponenten bevatten. Zo kan je een `Label`, een `TextBox` en een `Button` bij elkaar plaatsen in een overkoepelende `Grid` component omdat ze bij elkaar horen. De `MainWindow.xaml` heeft al een `Grid` component. Ondanks dat je dit niet ziet in de grafische weergave kan je dit herkennen door de `<Grid> </Grid>` in de bijbehorende `XAML`.

In de grafische weergave kan je componenten selecteren. Doordat componenten soms erg klein, of zelfs op de achtergrond zijn, is het niet altijd even makkelijk selecteren. Door op een `XML` element te klikken in de `XAML` kan je ook een component selecteren.

Als je een component selecteert kan je aan de rechter kans het `Properties` scherm openen. Dit kan je doen door onderaan de tab `Properties` te kiezen. Je kunt weer terug naar de `Solution Explorer` door deze tab weer te selecteren. Het properties scherm kent een aantal algemene properties die alle componenten hebben, denk bijvoorbeeld aan `Layout` en `Common`. Een aantal properties horen specifiek bij dat type component.

- Selecteer het component `Grid` en geef de achtergrond een andere achtergrond kleur. Tip: Dit doe je bij het onderdeel `Brush`.

  De gekozen achtergrondkleur kan je uiteraard straks weer aanpassen wanneer blijkt dat de kleur toch niet zo mooi is.

  Het doel van deze view is om de race simulator grafisch weer te geven. Dit kan je doen door op component te "tekenen". Als je de race visueel gaat weergeven op de `Grid` component heb je geen ruimte meer voor andere informatie zoals de naam van de track. Je zult dus een component moeten uitkiezen om op te kunnen tekenen. Tijdens deze opdracht ga je de `Image` component hiervoor gebruiken.

- Voeg aan het `Grid` component een `Image` component toe.

  Het `Image` component moet vervolgens wel een juiste plek en formaat krijgen. Wanneer je het `Image` component selecteert zal je wellicht opvallen dat je aan de randen van het venster, aan elke zijde 1, een klein icoontje ziet. Of er staat 2 kleine ovaaltjes of een gebroken ovaaltje. Bij 2 kleine ovaaltjes staat het component vastgezet. Wanneer de gebruiker straks het venster groter of kleiner maakt dan blijft de afstand tussen de rand en het component hetzelfde. Door op het icoontje te drukken kan je switchen tussen de 2 opties.

- Geef het `Image` component een gewenst formaat en zet het component bij elke kantlijn "vast".

  Bij de volgende taken worden er nog meer componenten geplaatst maar voor nu is het `Image` component voldoende.

Ondersteunende informatie

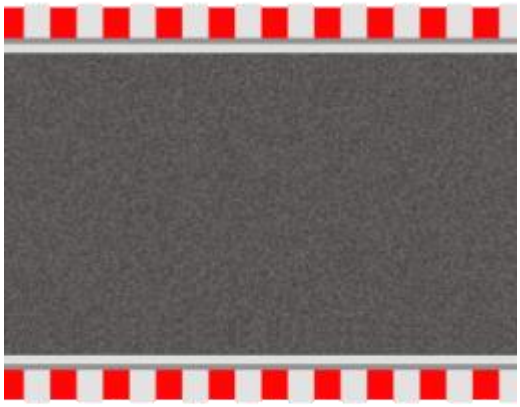
Meer informatie over de `XAML` editor kan je hier lezen.
Uitleg over het gebruik van de `designer` is hier te vinden.

Taak: Graphics

Nu je een scherm hebt met daarop een component op je racebaan te plaatsen is het tijd voor het leukste onderdeel; het grafisch visualiseren van je racebaan. Voordat je dit kan gaan doen zal je eerst wat moeten voorbereiden. Die voorbereiding ga je doen tijdens deze taak.

Aanpak

Voordat je verder gaat met programmeren zal je wat graphics moeten hebben. Net zoals bij de console view is het verstandig om de hoogte en de breedte per sectie hetzelfde te laten zijn. Waar het bij de console gaat over character gaat het bij de graphics over de pixels. Hieronder een aantal voorbeelden die gebruikt zijn bij de demo.



Uiteraard ben je volledig vrij om je eigen graphics te gebruiken. Je kunt op veel verschillende manieren aan je grapghics komen. Wat altijd helpt is een goed programma die je kan helpen je graphics te bewerken en in het juiste formaat te krijgen. Een fijn, en gratis, programma die je hierbij kan helpen is Paint.NET.

Het doel is dat je voor elke type sectie een los plaatje maakt waarbij elk plaatje dezelfde hoogte en breedte heeft. Kijk bij je console view welke type secties jij met characters hebt aangemaakt onder de `graphics region`.

De plaatjes hebben natuurlijk wel eerst een plek nodig in je project.

- Maak in je WPF project een folder, via de solution explorer, een folder aan.

Nu je een plek voor de plaatjes hebt.

- Maak voor elke type sectie een plaatje met dezelfde hoogte en breedte. Plaats deze plaatjes in de aangemaakte folder.

  Tijdens het implementeren van het klassendiagram heb je ook de enum `TeamColors` aangemaakt. Je zult voor elke kleur ook een visualisatie moeten hebben.

- Maak voor elke teamkleur een plaatje. Omdat 2 deelnemers naast elkaar moeten kunnen racen moeten de plaatjes zeker de helft kleiner zijn dan een sectie.

  De laatste visualisatie is wanneer een deelnemer even pech heeft.

  In Visual Studio kan je elk bestand een eigen `Build Action` geven. Plaatjes moeten uiteraard niet gecompileerd worden maar moeten wel gekopieerd worden naar waar de uiteindelijke executable geplaatst wordt.

- Click met je rechtermuisknop, in de solution explorer, op een plaatje.
- Selecteer `Properties` in het context menu.
- Verander de `Build Action` naar `Content` bij de `Properties`
- Voer de laatste 3 stappen voor alle andere plaatjes ook uit.

  Wanneer je nu de solution opnieuw build zullen alle plaatjes ook mee gekopieerd worden naar de build folder.

Ondersteunende informatie

Meer informatie over `Build Actions` kan je hier vinden.

Taak: Loading and Caching

Misschien speel je wel eens een spelletje op je PC, XBOX, PS4, etc.. Allemaal moeten ze eerst de wereld inladen voordat je kan spelen. Hoe frustrerend wachten kan zijn, het heeft zeker een reden. Een spel dat telkens elk plaatje opnieuw gaat laden wordt enorm traag. Soms is de laadtijd er kort en hebben de ontwikkelaars het voor elkaar gekregen om het laden tijdens het spelen te doen.

De ingeladen plaatjes worden opgeslagen in een `Cache`. In het kort is `caching` het opslaan van objecten in het geheugen van je computer. Soms doe je het omdat het orginele medium erg traag is en soms omdat de bewerking CPU intensief is.

In deze taak ga je ook een cache bouwen en ga je die gebruiken voor je plaatjes. Uiteraard ga je een intelligente cache maken waarbij de plaatje pas geladen worden wanneer ze nodig zijn.

Aanpak

Het werken met afbeeldingen is redelijk specifiek en verdient daarom een eigen klasse.

- Maak een static klasse aan, in je WPF project, welke de taak krijgt om afbeeldingen aan te maken, afbeeldingen in te laden en te bewerken (bv roteren).

  Het is idee is om een string, met de url naar het plaatje, om te zetten naar een afbeelding. Binnen C# zijn er verschillende types welke een plaatje kunnen bevatten. Een veelgebruikte type is `Bitmap`.

  Voor ophalen van plaatjes ga je hetzelfde doen als bij `GetSectionData` in de `Race` klasse.

- Geef de aangemaakte klasse een private `Dictionary` waarbij de string de key is en de bitmap de value.

  De aangemaakte dictionary gaat je cache worden. Deze vul je wanneer er een plaatje opgevraagd wordt.

- Maak nu een methode aan met een string als parameter en een `Bitmap` als return type. De string bevat straks de url naar het plaatje.
- Implementeer de methode verder waarbij je de meegegeven string gebruikt om de `Bitmap` op te halen in de dictionary. Wanneer de meegegeven string nog niet aanwezig is als key maak dan een nieuwe Bitmap en gebruik de meegegeven string als parameter voor de `Bitmap`. Plaats de nieuwe `Bitmap` en de bijbehorende string in de dictionary.

  Wanneer er weer een nieuwe race begint wil je snel de cache kunnen legen. Dit kan je doen door een nieuwe instantie aan te maken van de dictionairy maar je kan hiervoor ook de methode `Clear` gebruiken.

- Maak een methode aan, zonder parameters of return type, waarin je de cache leeg maakt.

  Je hebt nu effectief een slimme cache gemaakt zodat je straks niet elke keer het plaatje gaat inladen bij elke refresh.

Ondersteunende informatie

Wat `caching` is kan je o.a. hier lezen.

Informatie over `Bitmaps` kan hier vinden

Taak: Empty track

Net zoals bij de console visualisatie ga je stap voor stap aan de slag. Het maakt het ontwikkelproces makkelijker en prettiger. De eerste stap is plaatsen van een leeg plaatje in de aangemaakte component in je WPF window.

Aanpak

Voordat je de track kan tekenen zul je eerst een groot plaatje moeten aanmaken met de juiste dimensies. In de vorige level heb je een vaste hoogte en breedte, in pixels, gekozen voor elke sectie. Met deze informatie kan je uitrekenen wat het formaat moet worden van het plaatje waarop je de track gaat tekenen.

Breedte in pixels **=** Breedste punt van de track geteld in secties **x** breedte van een sectie in pixels
Hoogte in pixels **=** Hoogste punt van de track geteld in secties **x** hoogte van een sectie in pixels

Omdat deze informatie per track verschillend is zal je een methode moeten schrijven die hiermee rekening houdt.

Wat even handig is om te weten is dat je niet direct op een Bitmap kan tekenen. Hiervoor moet je gebruik maken van de `Graphics` klasse uit de namespace `System.Drawing`. Je kan de de methode `Graphics.FromImage` waarbij je de aangemaakte Bitmap mee geeft. De geretourneerde waarde, van het type `Graphics` kan je gebruiken om op de Bitmap te tekenen.

- Implementeer in de static klasse een methode met als parameters 2 int's en een Bitmap als return type.

  Je kan je voorstellen dat het redelijk CPU intensief is als je bij elke update een nieuwe lege Bitmap moet aanmaken. In deze methode ga je daarom actief gebruik maken van de aangemaakte cache bij de vorige level. Bedenk voor jezelf een waarde voor een string dat je gaat gebruiken als key in de dictionary, bijvoorbeeld "empty".

- Haal de waarde op bij de gekozen key (bv "empty"). Wanneer er nog geen waarde beschikbaar is maak dan een lege Bitmap met de juiste dimensies. Gebruik hiervoor de 2 parameters. Geef deze lege Bitmap een achtergrondkleur passend bij de tracks. Tip: gebruik hiervoor een `SolidBrush`. Voeg de nieuwe aangemaakte Bitmap toe aan de cache met de gekozen string (bv "empty") als key.

  Nu ga je iets doen wat erg belangrijk is. Als je de lessen nog kan herinneren over `call by value` en `call by reference` is het onverstandig om de lege Bitmap direct te retourneren. Immers je geeft de referentie mee van de Bitmap welke in de cache staat. Als je hierop gaat tekenen, dan teken je eigenlijk op de Bitmap in de cache. Hierdoor is de Bitmap geen lege Bitmap meer. Dus wat je wilt is een kopie van de Bitmap. Gelukkig heeft jouw aangemaakte Bitmap daar een methode voor, namelijk `Clone`.

- Tenslotte retourneer de `Clone` van de gevonden of aangemaakte Bitmap. Je hoeft geen parameters aan de methode `Clone` mee te geven Bitmap.

Het vervelende van oude en nieuwe technologie is dat er soms een mismatch is. In het geval van het WPF image component kan je niet zomaar een `Bitmap` gebruiken. Gelukkig zit het internet vol met verschillende oplossingen die je hiermee kunnen helpen. Belangrijk is dat je altijd de code moet begrijpen voordat je zomaar vreemde code in je project gebruikt. In dit geval krijg je de volgende methode welke je over kan nemen in je static klasse.

- Voeg de onderstaande methode toe aan de static klasse.

```
public static BitmapSource CreateBitmapSourceFromGdiBitmap(Bitmap bitmap)
{
    if (bitmap == null)
        throw new ArgumentNullException("bitmap");
    var rect = new Rectangle(0, 0, bitmap.Width, bitmap.Height);
    var bitmapData = bitmap.LockBits(
        rect,
        ImageLockMode.ReadWrite,
        System.Drawing.Imaging.PixelFormat.Format32bppArgb);
    try
    {
        var size = (rect.Width * rect.Height) * 4;
        return BitmapSource.Create(
            bitmap.Width,
            bitmap.Height,
            bitmap.HorizontalResolution,
            bitmap.VerticalResolution,
            PixelFormats.Bgra32,
            null,
            bitmapData.Scan0,
            size,
            bitmapData.Stride);
    }
    finally
    {
        bitmap.UnlockBits(bitmapData);
    }
}
```

Deze methode converteert de Bitmap en naar een `BitmapSource`. Een `BitmapSource` kan je wel gebruiken voor de image component op je WPF window.

Ondersteunende informatie

Wil je meer weten over de type `BitmapSource` open dan deze pagina.

Overzicht wat je allemaal met het `Graphics` object kan doen is hier te vinden.

Taak: First look

Nu je iets hebt staan om te tonen, al is het maar een leeg plaatje, is het tijd om de functionaliteiten in je controller te koppelen. In deze taak ga je de events in de controller gebruiken om je WPF image component scherm te updaten.

Aanpak

In de console view moest je alles initialiseren in de static methode main. In je WPF project zal je geen static methode main vinden, deze wordt automatisch gegenereerd wanneer je het WPF project build.

In WPF heeft elk scherm een eigen constructor, zoals gebruikelijk bij andere klassen. De constructor van een scherm wordt pas aangeroepen wanneer het scherm geopend wordt. Op dit moment wordt de `MainWindow` automatisch geopent wanneer je het WPF project start. Daarmee wordt ook de constructor van de `MainWindow` aangeroepen.

In de constructor van de `MainWindow`, te vinden in `MainWindow.xaml.cs`, staat al regel code.
Met `InitializeComponent();` wordt het scherm met alle bijbehorende componenten aangemaakt. Selecteer de regel `InitializeComponent();` en druk dan eens op `F12`. Hiermee "spring" je naar de methode `InitializeComponent`. Bovenaan, in het tabje, zie je dat de methode bestaat in het bestand `MainWindow.g.i.cs`. Dit cs bestand is **automatisch gegenereerd** en daarom mag je het **nooit wijzigen**. Uiteraard ben je vrij om het te bestuderen. Sluit dit bestand en open weer `MainWindow.xaml.cs`.

Nu je weet wat het start punt is van de WPF applicatie kan je beginnen met coderen. Het idee is om al iets te tonen op je WPF scherm. Je gaat dezelfde aanpak gebruiken zoals bij de console view.

- Maak een static klasse aan die verantwoordelijk voor de visualisatie, vergelijkbaar als in de console view.

  Bij de static klasse, in de console view, tekende je nog rechtstreeks op de console. Alleen bij WPF kan je niet zomaar rechtstreeks op een component tekenen. Dit doe je door een waarde te zetten.
  De `DrawTrack` methode bij de WPF view is daarom iets anders.

- Maak in de static klasse de methode `DrawTrack` aan met `Track` als parameter en `BitmapSource` als return type.

  In het kort ga je straks in de `DrawTrack` methode een plaatje maken met de complete visualisatie. Dit plaatje wordt vervolgens als waarde voor de image component gebruikt.

- Implementeer de `DrawTrack` methode waarbij je een leeg plaatje aanmaakt. Dit lege plaatje retourneer je vervolgens. Tip: Gebruik hiervoor de static klasse en methodes uit de vorige taak.

Nu je een begin hebt gemaakt met de `DrawTrack` methode is het tijd om het image component het plaatje te geven. Omdat de controller al klaar is kan je meteen gebruik maken van de beschikbare events.

- Gebruik de constructor, in `MainWindow.xaml.cs`, om de controller te initialiseren. Tip: kijk bij je console project wat je hier gedaan hebt.
- Maak vervolgens een eventhandler aan en koppel deze aan het `DriversChanged` event uit je controller.

Wanneer je nu het WPF project zal starten en een breakpoint zet in de eventhandler zal je zien dat deze nu telkens aangeroepen wordt. Hiermee kan je vaststellen dat het voorwerk goed is gegaan.

Nu je eventhandler aangeroepen wordt en er een `DrawTrack` methode is kan je eindelijk het image component bijwerken met het plaatje..... Was het maar zo eenvoudig.

In je console project kon je rechtstreeks op de console tekenen. Bij elke `Console.Write` werd de tekst meteen op het scherm geplaatst. Je had volledige controle. Microsoft heeft bij WPF besloten om deze vorm van controle niet toe te staan. Daarom kies je bij het maken van spellen meestal een `WinForms` project, daar heb je wel volledige controle over het scherm.

Leuk feitje: `UWP` is afgeleid van `WPF`. UWP is door Microsoft neergezet als Windows Store project. Doordat ontwikkelaars geen volledige controle hebben over de schermen, vanwege de UWP verplichting, was het erg moeilijk om constante FPS te behalen in spellen (voornamelijk 3D spellen). Hierdoor is de Windows Store nooit echt succesvol geworden als game store, zoals een Steam. Vanaf 2019 heeft Microsoft besloten om ook `Win32` (WinForms) projecten toe te staan in de Windows Store. UWP is hiermee eigenlijk om zeep geholpen, al zal Microsoft dat uiteraard ontkennen.

De reden waarom je niet direct kan tekenen op het scherm is omdat het scherm in een andere `thread` draait. Een `thread` is een stuk code dat bij elkaar hoort en wordt door je computer gezien als een los programma. Je kunt niet zomaar vanuit het ene programma dingen veranderen in een ander programma. In het geval van jouw race simulator draait de code uit de controller in een andere `thread` dan het WPF scherm.

Hoe kan je dan informatie in het WPF scherm aanpassen dan? Bij WPF moet je hiervoor een "verzoek" indienen. Wanneer jouw computer zin en tijd heeft dan wordt jouw verzoek uitgevoerd. Elk WPF component heeft hiervoor een `Dispatcher`. Deze `Dispatcher` ken een aantal methodes die je wellicht nog herkent bij het maken van de events zoals `Invoke`.

Bij de events heb je gebruik gemaakt van `Invoke`. Bij het aanpassen van de WPF componenten ga je `BeginInvoke` gebruiken. Het grote verschil is dat `Invoke` synchroon is en `BeginInvoke` asynchroon. Met `BeginInvoke` wacht je code niet op de computer om het scherm aan te passen.

De `BeginInvoke` heeft een aantal overloads waaronder `BeginInvoke(DispatcherPriority, Delegate)`. Deze ga je straks gebruiken. De `DispatcherPriority` parameter biedt je de mogelijkheid om de computer te vertellen hoe belangrijk de taak is. De 2e parameter is een `Delegate`. In plaats van dat je zelf een delegate gaat schrijven ga je nu een `Action` gebruiken.

Voordat je de `Dispatcher` kan gebruiken moet je wel het component in je code kunnen benaderen. Standaard geeft Visual Studio componenten geen `Name` waardoor je de componenten ook niet rechtstreeks kan benaderen.

- In het property scherm geef het image component een `Name`. Let op, ook hier moet je de naam met een hoofdletter laten beginnen.

  Nu je het image component een naam heeft kan je die nu ook benaderen vanuit je code. Dit kan je doen met `this`.

- Implementeer nu de eventhandler en gebruik de onderstaande code. Vervang `Xxx` met de naam van jouw component. Op de 3 puntjes roep je de aangemaakte `DrawTrack` methode aan.

```
this.Xxx.Dispatcher.BeginInvoke(
    DispatcherPriority.Render,
    new Action(() =>
    {
        this.Xxx.Source = null;
        this.Xxx.Source = ...;
    }));
```

  Als je de eventhandler goed hebt geïmplementeerd zal bij het uitvoeren van je WPF project het image component veranderen met het lege plaatje.

Ondersteunende informatie

Uitleg over wat `multi threading` is kan je hier lezen.
Wil je meer weten over `multi threading` bij WPF kan je dit lezen.
Informatie over het gebruiken van de `Dispatcher` en `BeginInvoke` is hier te vinden.
Nieuwschierig over de `Action` lees dan dit;

Taak: Graphical Visualisation

Met deze taak ga je het leukste doen, het grafisch weergeven van je race simulator. Wees vrij om lekker creatief aan de slag te gaan. Doordat je bij de vorige levels het meeste denkwerk al gedaan hebt kan je nu richten op de grafische kant. Eventuele oplossingen bij de console view kan je bij deze taak, met soms een kleine aanpassing, ook weer gebruiken.

Aanpak

Bij de console view heb je een region aangemaakt met daarin string arrays. In deze string arrays staan de secties uitgebeeld in characters. Bij de grafische weergave heb je natuurlijk geen characters maar plaatjes.

- Maak in de static klasse een region. Hierin maak je voor elk plaatje een string aan met als waarde de url naar het plaatje. Tip: Gebruik relatieve paden voor de url, bijvoorbeeld `".//images//finish.png"`. Doordat de string nooit aangepast gaat worden mag je de string ook `constant` maken, dit doe je door het keyword `const` te gebruiken.

  De volgende stap mag je weer naar eigen inzicht uitvoeren. Houd je daarbij wel aan de SOLID principes en zorg voor testbare code. Tip: Denk aan het `Graphics` object, hiermee kan je alle teken operaties doen die je nodig hebt.

- Implementeer de `DrawTrack` methode zodat de volledige track wordt weergegeven. Tip: Je kunt de methode, waarmee je een leeg plaatje maakt, uitbreiden waardoor je alvast aankleding plaatst zoals gras, bomen, vijver, etc... Leef je uit!

  Nu de track te zien is hoef je alleen nog maar de deelnemers te plaatsen. Waar je bij de console view nog een string replace deed moet je dit voor de grafische versie anders aanvliegen. Je moet nu, op de pixel nauwkeurig, uitrekenen waar je de deelnemers plaatst.

- Breid de visualisatie verder uit zodat je nu ook alle deelnemers op de track ziet bewegen. Vergeet niet de pechgevallen uit te beelden.

  De laatste stap is de event implementeren waarmee de huidige race wordt afgerond en een nieuwe wordt gestart.

- Implementeer de functionaliteit waarmee je de huidige race afrond en een nieuwe race start. Je kunt afkijken hoe je dit gedaan hebt bij de console view. Hint: Vergeet het legen van de cache niet.

  Bij het uitvoeren van de taken in deze level zal je wel gemerkt hebben hoe snel het ineens kan gaan. Ineens kan je gebruik maken van allerlei oplossing die je al hebt moeten maken voor de console view. Hopelijk maakt het ook duidelijk waarom een MVC structuur en het houden aan de SOLID principes zo belangrijk is. Het kost iets meer voorbereiding maar levert later in je project enorm veel tijdswinst op.

Ondersteunende informatie

Hoe en wanneer je `const` kunt gebruiken kan je hier lezen.