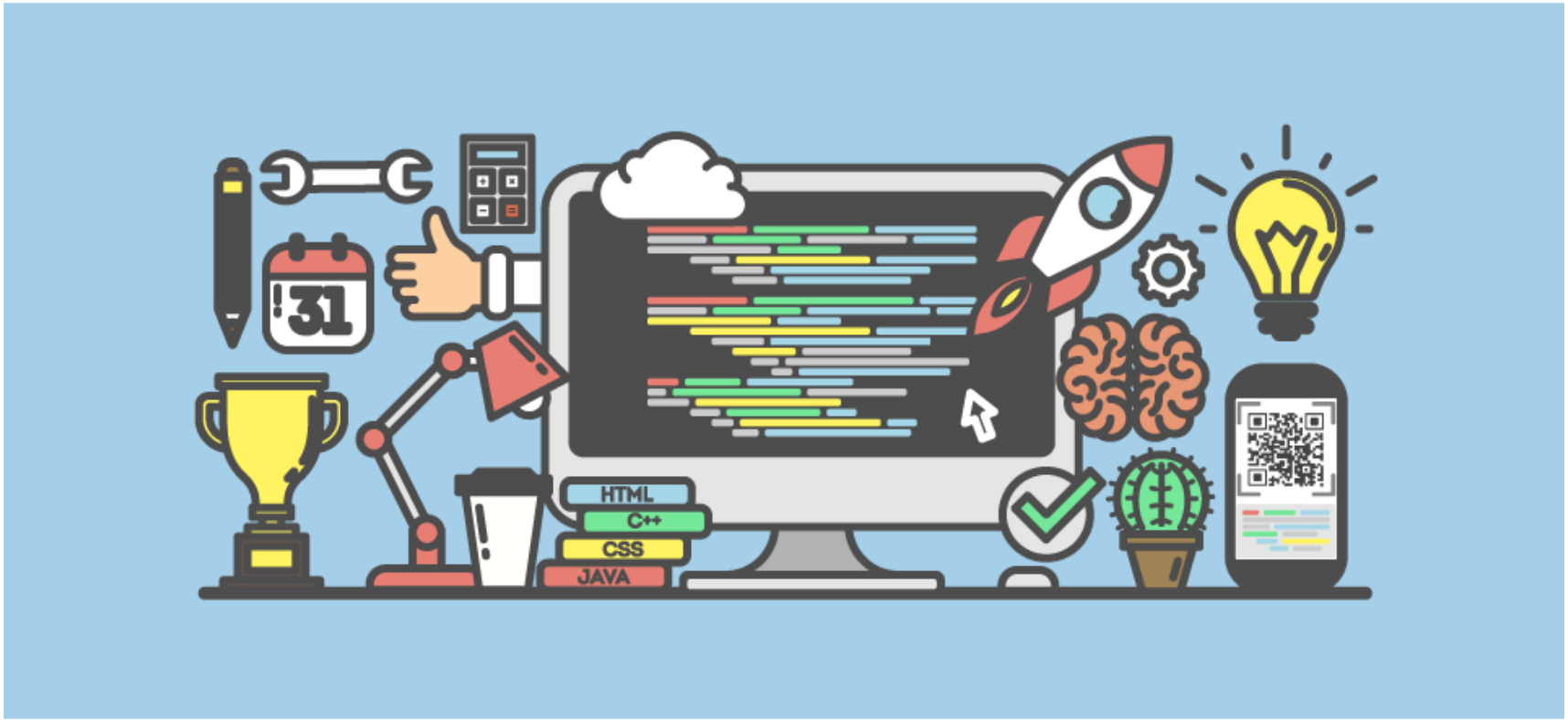
Basics of rendering - Algorithms with WPF



Gaëtan Blaise-Cazalet - Head of programming section

Basics of Rendering - 2d

Why WPF ?

- We want a easy to use tool to test 2d (and 3d) rendering algorithms and concepts
- A simple tool for simple ideas. We will use more complex tools further in the year.
- WPF uses C# (and XAML). C# is a widely used language, and is a the main language for Unity scripts.
- It is an efficient tool to create windows application, compatible with older versions of windows.

- Nevertheless : Microsoft wants to replace WPF by UWP. WPF years are counted.
  (link)
- WPF uses an old version of DirectX (DirectX 9).

Historical vocabulary

- Old painting applications, that would paint pixels on rectangular canvas, are called raster graphics applications
- The fact to convert something to pixels is called rasterization
- Those applications were drawing primitives, ie shapes that have a brush (inside the shape) and pen (edge of the shape)
- Two modes of rendering :
  - Immediate mode when the application regenerate a set of primitives to update a scene.
  - Retained mode when the content of the scene is represented as a scene graph. Updating the scene graph make the application update the rendering

Let's start with WPF

- Create a new WPF application in Visual Studio. It starts with a window.
- Create a canvas that fill the window
- We want to draw a clock.
- Add an Ellipse to figure your clock background

- MainWindow.xaml

```
<Window x:Class="_01.Canvas.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_01.Canvas"
        mc:Ignorable="d"
        Title="Intro to rendering" Height="600" Width="800">
    <Canvas ClipToBounds="False">
        <Ellipse Canvas.Left="-10" Canvas.Top="-10" Width="20" Height="20" Fill="LightGray" />
    </Canvas>
</Window>
```

Using transforms

- We want to keep our coordinate system on 100 units, but we also want the clock to fill the window
- We'll use transforms to adapt our drawing

- MainWindow.xaml

```
<Canvas ClipToBounds="False">
   <Ellipse Canvas.Left="-10" Canvas.Top="-10" Width="20" Height="20" Fill="LightGray" />

   <Canvas.RenderTransform>
      <TransformGroup>
         <ScaleTransform ScaleX="4.8" ScaleY="4.8" CenterX="0" CenterY="0" />
         <TranslateTransform X="48" Y="48" />
      </TransformGroup>
   </Canvas.RenderTransform>
</Canvas>
```

Use a template as a clock-hand

- We want to reuse the clock-hand
- We use a control template for that

- MainWindow.xaml

```
<Canvas ClipToBounds="False">
   <Canvas.Resources>
      <ControlTemplate x:Key="ClockHandTemplate">
         <Polygon Points="-0.3,-1 0.2,8 0,9 0.2,8 0.3,1" Fill="Navy" />
      </ControlTemplate>
   </Canvas.Resources>

   <Ellipse Canvas.Left="-10" Canvas.Top="-10" Width="20" Height="20" Fill="LightGray" />
   <Control Name="MinuteHand" Template="{StaticResource ClockHandTemplate}" />

   <Canvas.RenderTransform>
      <TransformGroup>
         <ScaleTransform ScaleX="4.8" ScaleY="4.8" CenterX="0" CenterY="0" />
         <TranslateTransform X="48" Y="48" />
      </TransformGroup>
   </Canvas.RenderTransform>
</Canvas>
```

The hour clock-hand

- MainWindow.xaml

```
...
<Control Name="MinuteHand" Template="{StaticResource ClockHandTemplate}" />
<Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
   <Control.RenderTransform>
      <TransformGroup>
         <ScaleTransform ScaleX="1.7" ScaleY="0.7" CenterX="0" CenterY="0" />
         <RotateTransform Angle="45" />
      </TransformGroup>
   </Control.RenderTransform>
</Control>
...
```

- Now add a red thin clock-hand for the seconds

Add animation

- Set the hour handle to top (12)
- We add an additional rotation with a name (ActualTimeHour)
- We set an animation for this variable
- MainWindow.xaml

```
<Control Name="HourHand" Template="{StaticResource ClockHandTemplate}">
    <Control.RenderTransform>
        <TransformGroup>
            <ScaleTransform ScaleX="1.7" ScaleY="0.7" CenterX="0" CenterY="0" />
            <RotateTransform Angle="180" />
            <RotateTransform x:Name="ActualTimeHour" Angle="0" />
        </TransformGroup>
    </Control.RenderTransform>
</Control>

<Canvas.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetName="ActualTimeHour" Storyboard.TargetProperty="Angle"
                        From="0" To="360" Duration="00:00:10.00" RepeatBehavior="Forever" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Canvas.Triggers>
```

- Make every handle turn
- Make the clock turn at real time rate
- Find a way to initialize the clock at correct hour

Exercice - The covered wagon

Your goal: to modify the clock application to make and animate a 2D model of a cart (viewed from the side), consisting of a rectangle resting atop two wheels. Each wheel should have four spokes constructed via instantiations of the clock-hand template that we've already defined for you. Feel free to rename things to match the new semantics, e.g. "SpokeStencil" instead of "ClockHandStencil".

1. First build one wheel, complete with the four spokes, but don't worry about rotating it yet. It's preferable to use a coordinate system that is comfortable for the modeling process, i.e. with the center of the wheel at (0,0).

2. Wrap the entire wheel model up in a canvas and make it be a new template resource, as was previously done for the clock hand. You will end up with a structure like that shown below. To make sure you didn't damage its functionality, test by instantiating it onto the visible canvas.

- MainWindow.xaml

```
<Canvas.Resources>
 <ControlTemplate x:Key='SpokeStencil'>
  <Polygon ... />
 </ControlTemplate>

 <ControlTemplate x:Key='WheelStencil'>
  <Canvas>
   <Ellipse Width='2.0' ... />
   <Control Name='Spoke1'
          Template='{StaticResource SpokeStencil}'>

    <Control.RenderTransform> ...
   </Control>

   ... Three more spoke declarations will be here ...

  </Canvas>
 </ControlTemplate>
```

3. Make the wheel rotatable by attaching a transform to the element that is instantiating it, via the same technique that was used to make the minute hand rotatable.

4. Modify the animation element that originally rotated the minute hand, to make it rotate the wheel instead.

5. Add to the scene a rectangle and another instance of the wheel template to make a simple model of a cart or wagon. Of course, the second wheel instance must be translated differently to ensure it doesn't simply lie right on top of its sibling. Make sure the new wheel is separately animatable, and create a new animation element to handle its rotation.

6. It's rather unrealistic for a cart with two spinning wheels to appear to be standing still! Use a canvas element to wrap up the entire model of the cart and attach a translate transform to this new canvas element; this transform will thus control the entire cart model as a whole. Connect an animator element to the translate transform's x value, to achieve the goal of having the entire model move horizontally to simulate the effect of the spinning wheels.

7. If you've completed the above step successfully, your cart probably zips off the screen the moment the movie begins, leaving your audience with nothing to be awed by! Set the AutoReverse property on all of the animators to simulate a cart that simply goes back and forth on a short track, thus staying on screen throughout the movie.