Particles and Graphics Effects in Qt Quick 2

Release 1.0

Digia, Qt Learning

February 28, 2013

Contents

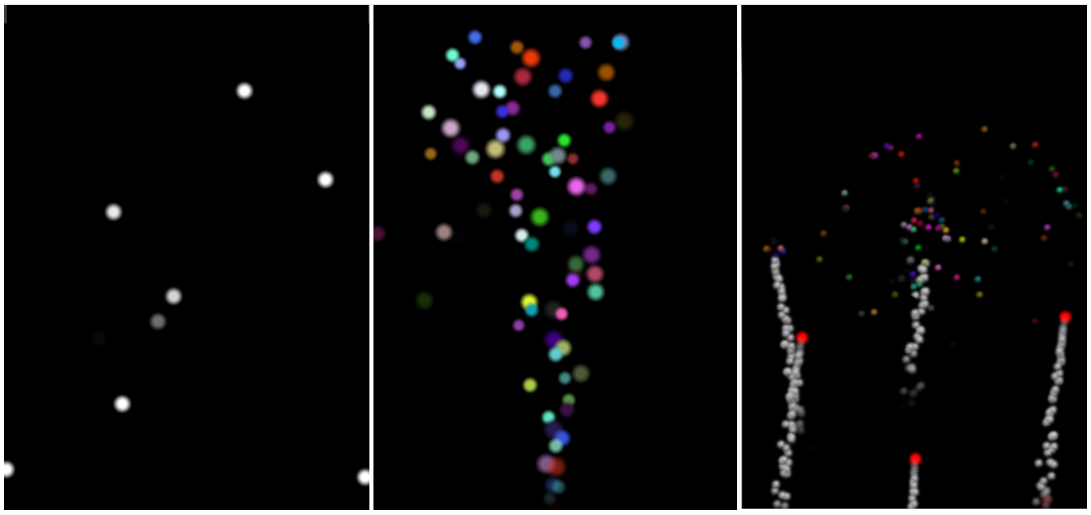
CHAPTER 1

About this Tutorial

1.1 Why Would You Want to Read this Guide?

The goal of this tutorial is to introduce you to some of the features of Qt Quick 2 for implementing animations and graphic effects. Mainly, this tutorial provides an overview of how to use the *Particles* module in Qt Quick 2 as well as *ShaderEffects* for advanced graphic effects.

The tutorial is split into three main chapters. In the first chapter, you will be introduced to the *Particles* module. We will provide some basic setup code illustrating the use of the main types. The second chapter provides a quick overview of how to use *Shader* programs within QML through a simple example. The last chapter will focus on implementing a demo application step-by-step using *Animations*, *Particles*, and *Shaders*.



1.2 Get the Source Code and the Tutorial in Different Formats

A .zip file that contains the full code source of the tutorial's examples is provided:

Source code[1]

The guide is available in the following formats:

- PDF[2]
- ePub[3] for ebook readers.
- Qt Help[4] for Qt Assistant and Qt Creator.

1.3 License

What's Next?

Next chapter provides an overview of the *Particles* module in Qt Quick 2 with simple examples that will cover the most basic types.

[1] http://releases.qt-project.org/learning/developerguides/qteffects/particles_src.zip
[2] http://releases.qt-project.org/learning/developerguides/qteffects/ParticlesTutorial.pdf
[3] http://releases.qt-project.org/learning/developerguides/qteffects/ParticlesTutorial.epub
[4] http://releases.qt-project.org/learning/developerguides/qteffects/ParticlesTutorial.qch

CHAPTER 2

Particles

2.1 Overview

Qt Quick 2 comes with the *Particles* module for making nice visual particle effects, which can be used by many applications that require a lot of tiny moving particles such as fire simualtion, smoke, stars, music visualization, and so on.

The Particles[1] module is based on four major components:

2.2 Basic Setup

Let's start with a simple example that illustrates how we can use those different elements together to make particle effects.

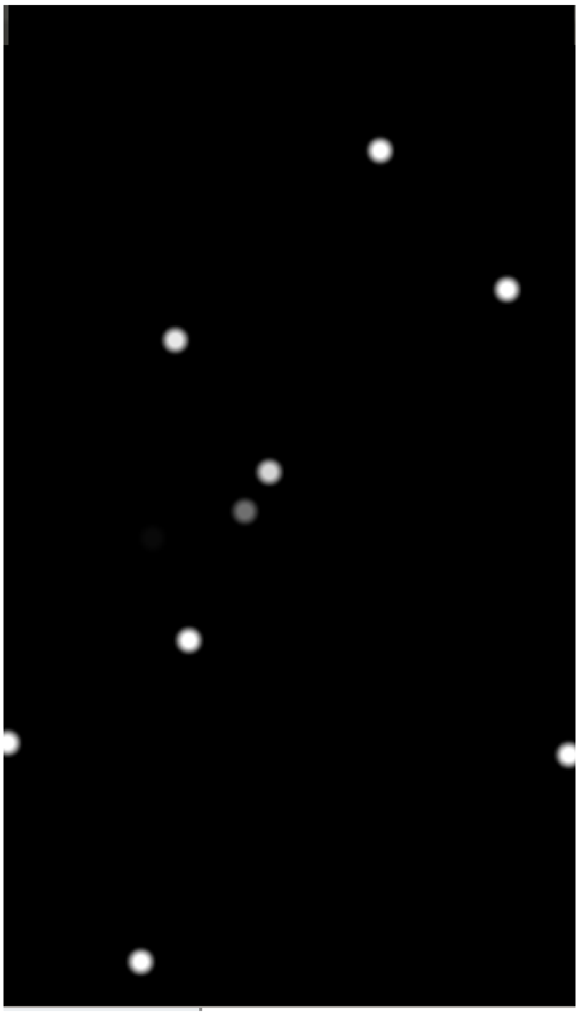
The following example implements a simple rectangle with a `ParticleSystem` type that contains an ImageParticle[2] to render particles based on an image, and an `Emitter` to create and emit particles.

```
// particles_example_02.qml

import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {

    width: 360
    height: 600
    color: "black"

    ParticleSystem {
        anchors.fill: parent

        // renders a tiny image
```

[1] http://qt-project.org/doc/qt-5.0/qtquick/qtquick-particles2-qtquick-effects-particles.html
[2] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-imageparticle.html

```
        ImageParticle {
            source: "resources/particle.png"
        }

        // emit particle object with a size of 20 pixels
        Emitter {
            anchors.fill: parent
            size: 20
        }
    }
}
```

If you run the code shown above, you will see a couple of tiny particles (based on the image source) blinking on a black background.



The particles are emitted all over the entire area of the parent because we set the emitter's anchors to fill the entire area of the root element (that is, the rectangle).

To make the animation more intersting, we may want to make all particles emit from the bottom of the window and spread out with an increased lifeSpan[3].

First we set the emitter's anchors and specify where we want the particles to be emitted from.

[3]http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-emitter.html#lifeSpan-prop

```
Emitter {
    height: 10; width: 10
    anchors.bottom: parent.bottom
    anchors.horizontalCenter: parent.horizontalCenter
}
```

Then we set the trajectory and speed of the particles using AngleDirection[4] QML type.

```
Emitter {
    ...
    velocity:  AngleDirection {
        // Make particles spread out vertically from the bottom
        angle: 270
        // make the movement of the particles slighly different from
        // one another
        angleVariation: 10
        // set speed to 150
        magnitude: 100
        }
    ...
}
```

As the default `lifeSpan` for a particle is one second, we will increase its value so that we can visualize the particles path:

```
Emitter {
    ...
    // 8 seconds may be enough
    lifeSpan: 8000
}
```

We can also set the particles to emit in various sizes by using the sizeVariation[5] property in the `Emitter` component:

```
Emitter {
    ...
    // set the variation up to 5 pixels bigger or smaller
    sizeVariation: 5
}
```

The colorVariation[6] property in the `ImageParticle` type enables us to apply color variation to the particles:

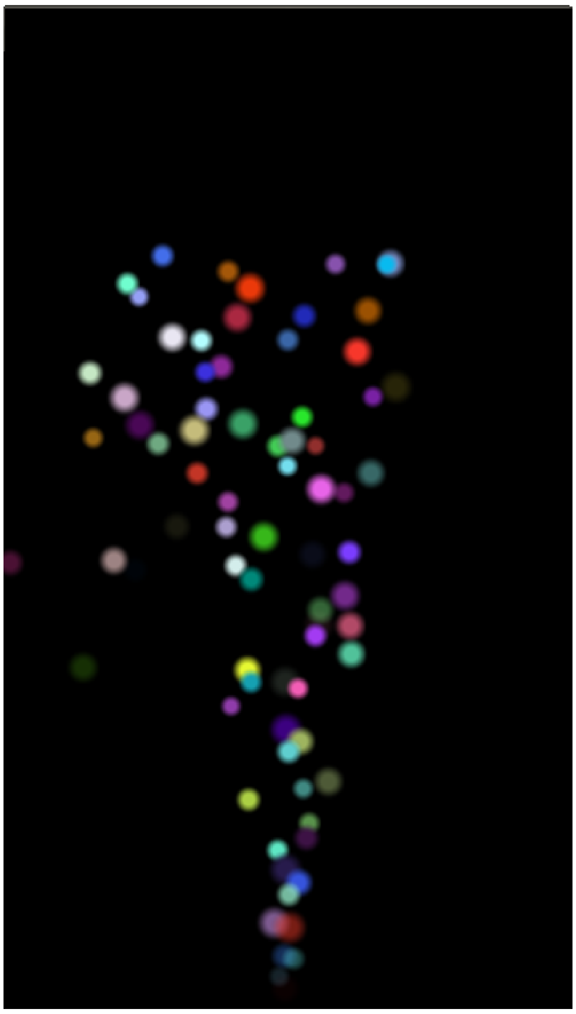
```
ImageParticle {
    ...
    //Color is measured, per channel, from 0.0 to 1.0.
    colorVariation: 1.0
}
```

Then we can use the Gravity[7] affector to make our particles fall back down.

[4] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-angledirection.html
[5] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-emitter.html#sizeVariation-prop
[6] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-imageparticle.html#colorVariation-prop
[7] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-gravity.html

```
ParticleSystem {
    ...
    Gravity {
        anchors.fill: parent
        // apply an angle of acceleration when the particles hit
        // the affector
        angle: 90
        // accelerate with 15  pisxels/second
        acceleration: 15
    }
    ...
}
```

If you now run the code, you will see an animation displaying particles of different sizes and colors spreading out from the bottom to the top of the window and then falling back down.



Note: The complete code is available in the *particles_example_02.qml* file.

2.3 ParticleGroups and Transitions

The *Particles* module also provides a ParticleGroup[8] type that enables us to set timed transitions on particle groups. This could be very helpful if we want to implement animations with special behavior that require many transitions.

To illusrate how we can use `ParticleGroup`, let's implement a simple fireworks animation. The particles should be emitted from the bottom of the window. We'll also add some TrailEmitters[9] that simulates smoke produced by flames as well as explosions in mid-air.

In our fireworks animation we proceed as follows:

> Within the main `Rectangle`, we add a `ParticleSystem` that will be used by all components to run the animation.
>
> Add the main `Emitter` that emits firework particles from the buttom to the top of the window and specify a logical group identifier so that we can later assign an `ImageParticle` to render the flame particles.
>
> Add a `TrailEmitter` that will simulate the smoke produced by the flame. We also specify a logical group so that we can later assign the corresponding `ParticlePainter` to the emitter.
>
> Add a `ParticleGroup` to simulate the explosion using a `TrailEmitter` type.
>
> Add a `GroupGoal` in the main `Emitter` to tell where or when to apply the transition we define in the `ParticleGroup`.

Note: A logical group enables us to paint particles emitted by different `Emitters` using different `ImagePartilces` within the same `ParticleSystem` as we will see later in the *four seasons* demo application.

So first, we declare one main `Emitter` that emits firework particles from the bottom to the top:

```
import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {

    width: 360
    height: 600
    color: "black"


    // main particle system
    ParticleSystem {id: particlesSystem}

    // firework emitter
```

[8]http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-particlegroup.html
[9]http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-trailemitter.html

```
    Emitter {
        id: fireWorkEmitter
        system: particlesSystem
        enabled: true
        lifeSpan: 1600
        maximumEmitted: 6
        // Specify the logical group that
        // the emitter belongs to
        group: "A"
        // we want to emit particles
        // from the bottom of the window
        anchors{
            left: parent.left
            right: parent.right
            bottom: parent.bottom
        }

        velocity:  AngleDirection {
                     angle: 270
                     angleVariation: 10
                     magnitude: 200
                 }
    }
}
```

Then we add a `TrailEmitter` type to simulate the smoke produced by the firework before exploding in the air.

```
TrailEmitter {
    system: particlesSystem
    group: "B"
    // follow particle emitted by fireWorkEmitter
    follow: "A"
    size: 12
    emitRatePerParticle: 50
    velocity: PointDirection {yVariation: 10; xVariation: 10}
    acceleration: PointDirection {y:  10}
}
```

Then we add a `ParticleGroup` type to set a transition and simulate the explosion of particles in the air. We will be using a `TrailEmitter` with an `AngleDirection` to display the exploding effect.

```
ParticleGroup {
    name: "exploding"
    duration: 500
    system: particlesSystem

    TrailEmitter {
        group: "C"
        enabled: true
        anchors.fill: parent
        lifeSpan: 1000
        emitRatePerParticle: 80
        size: 10
        velocity: AngleDirection {angleVariation: 360; magnitude: 100}
        acceleration: PointDirection {y:  20}
    }
}
```

In order to know exactly where to apply the transition, we add a GroupGoal[10] type inside the *fireWorkEmitter* that tells the emitter what the aimed state is and when/where the particles should switch to it.

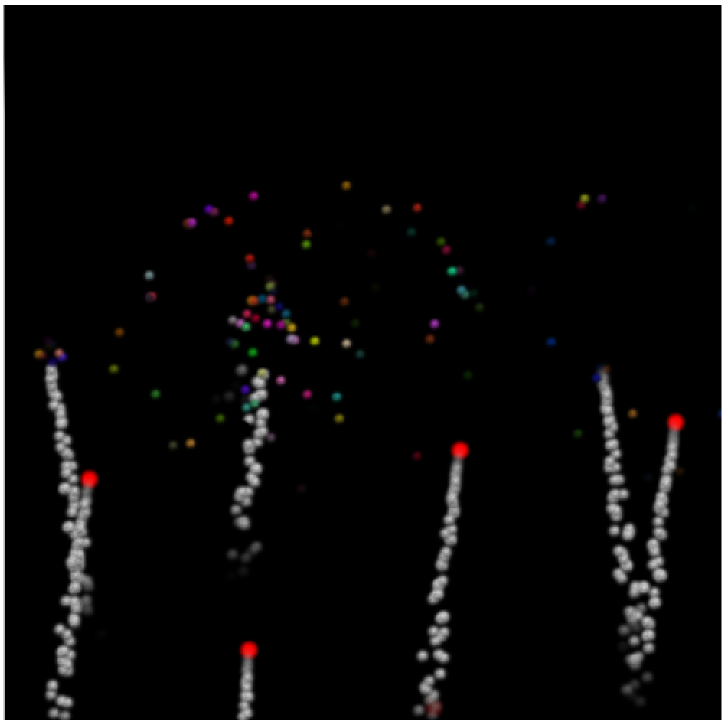
```
Emitter {
    id: foreWorkEmitter
    ...
    GroupGoal {
        // on which group to apply
        groups: ["A"]
        // the goalState
        goalState: "exploding"
        system: particlesSystem
        // switch once the particles reach the window center
        y: - root.height / 2
        width: parent.width
        height: 10
        // make the particles immediately move to the goal state
        jump: true
    }
}
```

Next, we just add the `ImageParticle` types to visualize particles for each group defined above.

```
// ParticlePainter for the main emitter
ImageParticle {
    source: "resources/particle.png"
    system: particlesSystem
    color: "red"
    groups: ["A"]
}

//  ParticlePainter for the trailEmitter smoke
ImageParticle {
    source: "resources/smoke_particle.png"
    system: particlesSystem
    groups: ["B"]
    color: "white"
}

// ParticlePainter for the trailEmitter in the ParticleGroup
ImageParticle {
    source: "resources/smoke_particle.png"
    system: particlesSystem
    groups: ["C"]
    color: "red"
    colorVariation: 1.2
}
```

And now if you run the code, you should have a simple animation that displays particles emitted from the window bottom and exploding once they reach the window center:

[10] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-groupgoal.html



2.4 what's next?

In the next article, we introduce the `ShaderEffect` type used for more advanced graphic effects. We will also implement a demo application that uses *Particles* and *Shaders*.

CHAPTER 3

Shader Effects

3.1 Overview

In order to perform advanced graphical effects, Qt Quick 2 enables you to use vertex and fragment shader programs with your QML local properties via the ShaderEffect[1] QML type.

This type enables you to combine your GLSL program with your QML code to control the graphics at a much lower level using custom shaders. *ShaderEffect* enables you to implement a vertex or fragment shader program in your QML code via the vertexShader[2] and fragmentShader[3] properties. When you specify a QML item as variant property in your *ShaderEffect*, the item is provided to your vertex or fragment shader as *Sampler2D*.

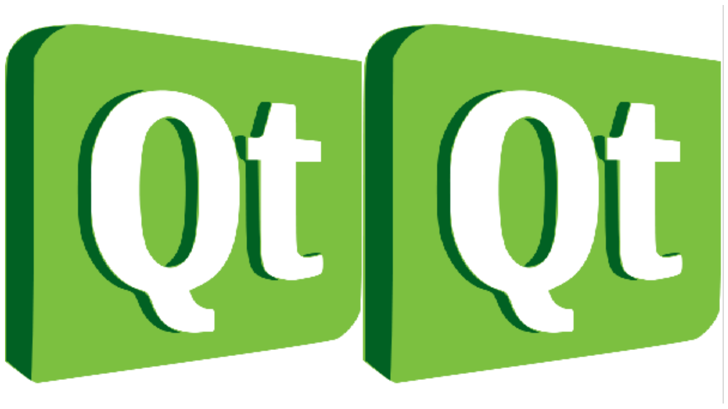
Consider the following example:

```
import QtQuick 2.0

Rectangle {
    id: root
    color: "white"
    width: 600
    height: 300

    Image {
        id: background
        width: parent.width/2
        height: parent.height
        source: "resources/Qt.png"
        anchors {
            right: parent.right
            top: parent.top
        }
    }

    ShaderEffect {
        id: shaderEffect
```

[1]http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-shadereffect.html
[2]http://qt-project.org/doc/qt-5.0/qml-qtquick2-shadereffect.html#vertexShader-prop
[3]http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-shadereffect.html#fragmentShader-prop

```
        width: parent.width/2
        height: parent.height
        anchors {
            left: parent.left
            top: parent.top
        }

        property variant source: background

    }

}
```

The *ShaderEffect* type takes the background QML item, provides it as a *Sampler2D* to the fragment shader and paints the result on the screen (at the position of the *ShaderEffect*). In the above example, we did not specify any fragment or vertex shader. So the default shaders that apply no special effects are used.



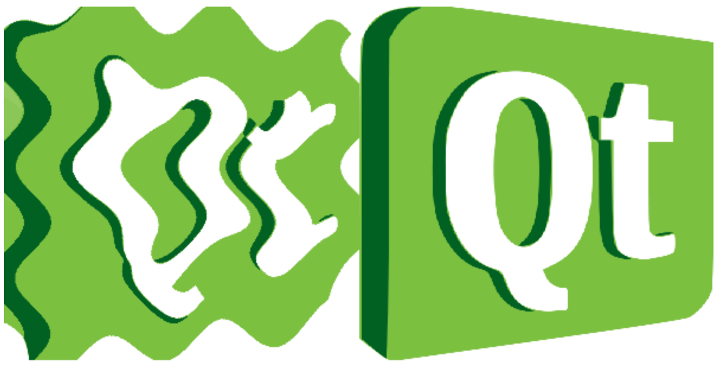
You can then add your vertex or fragment shader program using *fragmentVertex* or *fragmentShader*. For example, we can add an effect using a fragment shader as follows

```
import QtQuick 2.0

Rectangle {
    id: root
    color: "white"
    width: 600
    height: 300

    Image {
        id: background
        width: parent.width/2
        height: parent.height
        source: "resources/Qt.png"
        anchors {
            right: parent.right
            top: parent.top
        }
    }

    ShaderEffect {
        id: shaderEffect
        width: parent.width/2
        height: parent.height
        anchors {
            left: parent.left
            top: parent.top
        }

        property variant source: background
        property real frequency: 20
        property real amplitude: 0.05
        property real time

        NumberAnimation on time {
            from: 0; to: Math.PI * 2
            duration: 1000
            loops: Animation.Infinite
        }

     fragmentShader:
            "varying highp vec2 qt_TexCoord0;
            uniform sampler2D source;
            uniform lowp float qt_Opacity;
            uniform highp float frequency;
            uniform highp float amplitude;
            uniform highp float time;
            void main(){
                  vec2 p= sin(time + frequency * qt_TexCoord0);
                  gl_FragColor = texture2D(source, qt_TexCoord0 + amplitude *vec2(p.y, -
                       }";

    }

}
```

Again the background QML item is provided as *Sampler2D* in the fragment shader. Another very important feature that is introduced in the code above is the automatic property binding between QML and GLSL code.

If an *uniform* variable in the vertex or fragment shader program has the same name as a property defined in the *ShaderEffect*, the value of this property is bound to the uniform.

In the above code snippet we are using this feature in conjunction with a *NumberAnimation* to produce a animated wobbling effect. The effect is shown in the screenshot below:



For more details concerning GLSL and the use of Shaders in QML, refer to the related links listed at the end of this tutorial.

3.2 What's next?

Next, we will be implementing a demo application that illustrates the use of the `Particles` module and `ShaderEffect` type in QML.

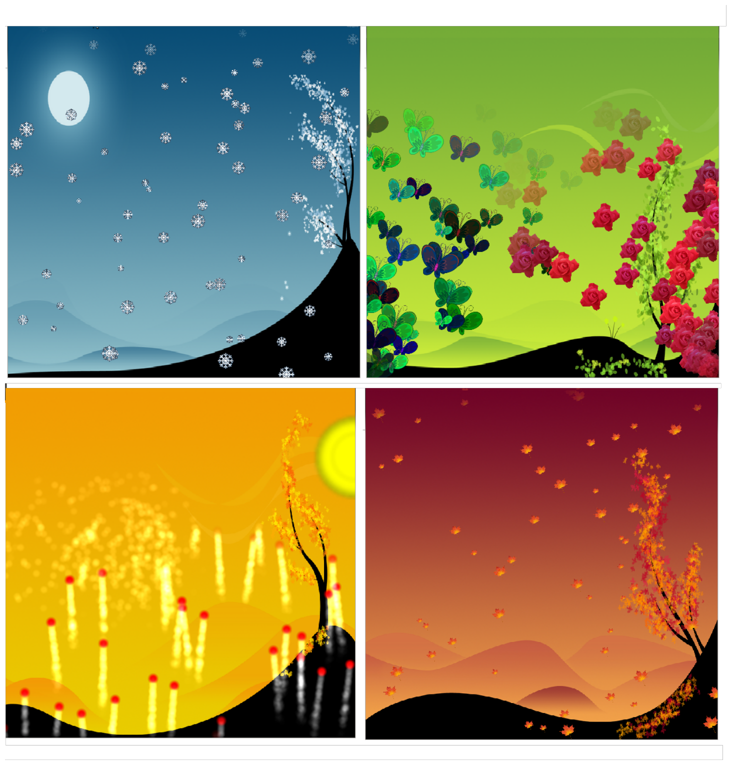
CHAPTER 4

Demo Application

In order to have a better understanding of how to use `Particles` and `ShaderEffect`, we will implement an example that illustrates their use to create some animations and graphical effects.

In this chapter, we will implement a simple demo that consists of four background images corresponding to one of the four seasons with special animations for each season. The idea is to show you different ways and techniques of making nice animations and graphic effects using particles and shaders.

Almost all animations are implemented using particle effects. However, we add some animations using shader effects. We will create four special animations, each having its own `Emitter` and `ParticlesImage` types with different settings that correspond to the four seasons of the year. The following figure presents a screenshot of the final implementation:



In order to easily follow the steps of our implementation, this tutorial is split into several sections. Each section covers the `Emitter` and `ParticlesImage` types (as well as other elements) associated with each season.

4.1 The Main Element

The application consists of a `Rectangle` type with an `Image` that displays different backgrounds. Each background image corresponds to a season. For each season, we associate special animations based on *Particles*. We additionally add an animation when switching from one season to another.

4.2 Background

The main rectangle displays an image for each season. Let's start by implemeting a simple animation once the user switches from one season to another.

```
Rectangle {
    id: root

    property int  numberVal: 4

    width: 600
    height: 600
    // enable keyboard events
    focus: true

    Image {
        id: background
        anchors.fill: parent
        source: "resources/winter.png"

        Behavior on source {
            SequentialAnimation {
                ParallelAnimation {
                    NumberAnimation { targets: background;
                                      properties: "opacity";
                                      to: 0 }
                    NumberAnimation { target: background;
                                      property: "scale";
                                      to : 2 }

                }

                PropertyAction {target: background;
                                property: "source"}

                ParallelAnimation {
                    NumberAnimation { targets: background;
                                      properties: "opacity";
                                      to: 1 }
                    NumberAnimation { target: background;
```

```
                                        property: "scale";
                                        to : 1}

                }
            }
        }
    }
}
```

The default season is winter in the code shown above. To manage different seasons, we define a `State` for each season as follows:

```
states :[
    State {
        name: "summer"
        PropertyChanges { target: background;
                          source: "resources/summer.png" }
    },

    State {
        name:"spring"
        PropertyChanges{ target: background;
                         source: "resources/spring.png" }
    },

    State {
        name:"autumn"
        PropertyChanges{ target: background;
                         source: "resources/autumn.png" }
    }
]
```

In every `State`, we just apply the corresponding background image to the active season.

Then we define a function to switch between seasons. Each function should set the corresponding state and should later apply the related animation.

```
function toSpring()
{
    state = "spring"
    // Apply spring animation later ...

}

function toSummer() {
    state = "summer"
    // Apply summer animation later ...

}

function toAutumn() {
    state = "autumn"
    // Apply autumn animation later ...

}

function  toWinter (){
    // default state
    state = ""
    // Apply winter animation later ...

}
```

Once the background image has been changed, we add a `NumberAnimation` that modifies the image's scale and opacity. For more details concerning animations in QML, refer to the NumberAnimation Documentation[1].

To switch between the season's background, the user can simply press the `space` key on the keyboard:

```
Keys.onPressed: {
    if (event.key == Qt.Key_Space){
        switch(state) {
            case  "":
                toSpring();
                break;
            case "spring":
                toSummer();
                break;
            case "summer":
                toAutumn() ;
                break;
            case "autumn":
                toWinter();
                break;
        }

    }
}
```

4.3 Winter Animation

In the *winter* state, we want to display some snow particles falling down from the top of the window. So first we declare a `ParticleSystem` that paints the particles and runs the emitters:

```
ParticleSystem { id: sysSeason  }
```

Then we add a `ParticleImage` type that visualizes logical particles using an image. In our case, the image should correspond to a snow particle. We also specify the system whose particles should be visualized and a group property to specify which logical particle group will be painted. This is helpful if we want to use different emitters within the same `ParticleSystem`:

```
ImageParticle {
    id: snow
    system: sysSeason
    source: "resources/snow.png"
```

[1] http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick2-numberanimation.html

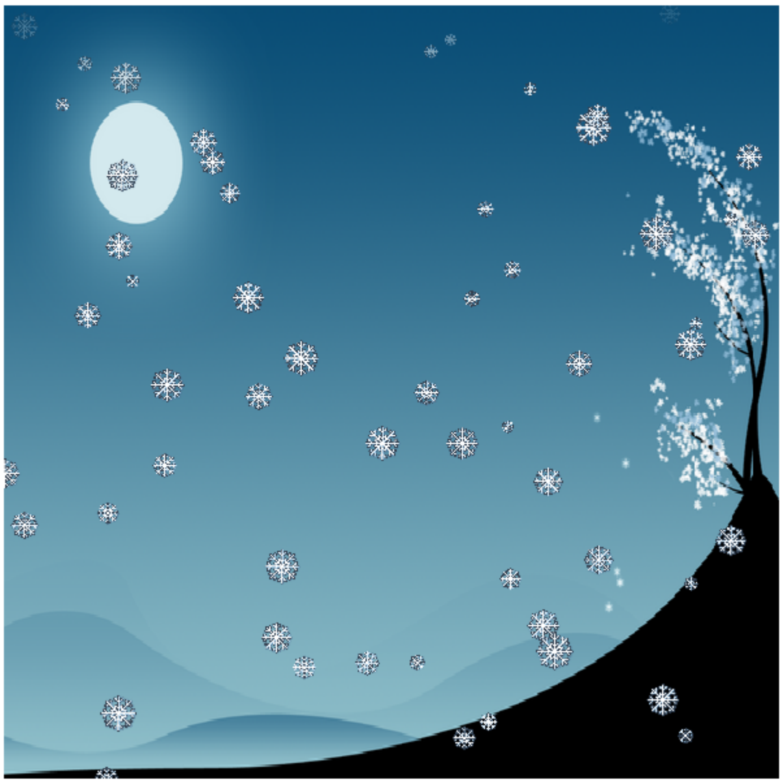
```
    groups: ["A"]
}
```

To emit particles, we add an `Emitter` type that emits our snow particles from the top window down to the bottom using an `AngleDirection` with a 90° angle:

```
Emitter {
    id: snowEmitter
    // Enable the emitter as winter is the default state
    enabled: true
    system: sysSeason
    group: "A"
    lifeSpan: 8000
    anchors{
        left: parent.left
        right: parent.right
        top: parent.top
    }
    velocity: AngleDirection { angle: 90;
                                angleVariation : 20;
                                magnitude: 100 }
    size: 20
    sizeVariation: 10
}
```

We also specify the logical particle group that corresponds to the *snowImage*, with a `lifeSpan` of 8 second.

The following screenshot shows what the particles will look like:



4.4 Spring Aniamtion

In the Spring season, we want to display some flower and butterfly particles from the bottom corners of the window. So first we define the `ImageParticles` to vizualize flower and butterfly particles.

```
// ImageParticle for butterfly
ImageParticle {
    id: butterfly
    system: sysSeason
    source: "resources/butterfly.png"
    colorVariation: 1.0
    groups: ["C"]
}

// ImageParticle for flowers
ImageParticle {
    id: flower
    system: sysSeason
    source: "resources/flower.png"
    colorVariation: 0.4
    groups: ["B"]
}
```

As the particles should be emitted from different places, we will be using two emitters. In each `Emitter`, we specify the logical particles group.

In the butterFly `Emitter`, we specify a group and emit the particles from the bottom right corner:

```
Emitter {
    id: butterFlyEmitter
    enabled: false
    system: sysSeason
    lifeSpan: 5000
    group: "C"
    anchors.bottom: parent.bottom
    velocity : AngleDirection { angle : 300;
                                angleVariation: 30;
                                magnitude: 100 }
    size: 50
    sizeVariation: 20
}
```

In `flowerEmitter`, we use the same code as in `butterFlyEmitter`, but with a different group and from the opposite corner:

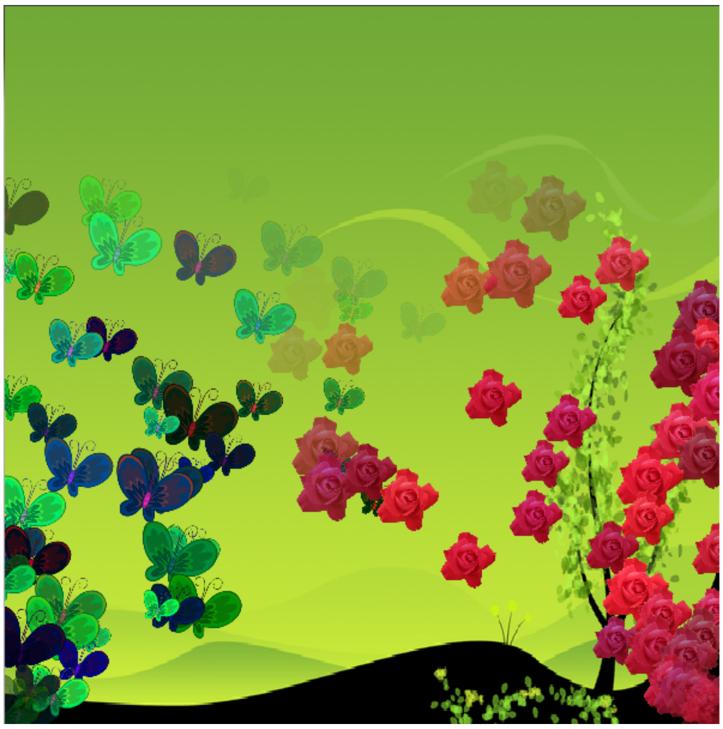
```
Emitter {
    id: flowerEmitter
    enabled: false
    system: sysSeason
    lifeSpan: 5000
    group: "B"
    anchors.bottom: parent.bottom
    anchors.right: parent.right
    velocity : AngleDirection { angle : 250;
```

```
                                        angleVariation: 40;
                                        magnitude: 100 }
    size: 50
    sizeVariation: 10
}
```

In the `toSpring` function, once we switch to the *spring* season, we disable the `snowEmitter` and enable the *butterFly* and *flower* emitters.

```
function toSpring()
{
    state = "spring"

    snowEmitter.enabled = false
    butterFlyEmitter.enabled = true
    flowerEmitter.enabled = true
}
```

If you now run the code, you should be able to visualize flower and butterfly particles as shown on the following screen:



4.5 Summer Animation

In the summer state, we will be adding two major animations: one to simulate the sun movement and the other to launch some fireworks.

For the sun animation, we define an `Emitter` that emits particles using `AngleDirection`. We also want the emitter to move from left to right, so we add a `SequentialAnimation` on the `x` and `y` properties:

```
Emitter {
    id: summerEmitter
    enabled: false
    system: sysSeason
    lifeSpan: 200
    group: "G"
    y: parent.height / 4
    emitRate: 1600
    velocity : AngleDirection { angleVariation : 360 ;
                                    magnitude: 80}
    size: 100
    sizeVariation: 50

    SequentialAnimation {
        id: sunAnimation

        ParallelAnimation
        {
            NumberAnimation { target: summerEmitter;
                              property: "x" ;
                              from: 0;
                              to: root.width/2;
                              duration: 10000;
                              running: false }

            NumberAnimation { target: summerEmitter;
                              property: "y" ;
                              from: root.height/4;
                              to: 0;
                              duration: 10000;
                              running: false }
        }

        ParallelAnimation
        {
            NumberAnimation { target: summerEmitter;
                              property: "x" ;
                              from: root.width/2;
                              to: root.width;
                              duration: 10000;
                              running: false }

            NumberAnimation { target: summerEmitter;
                              property: "y" ;
                              from: 0;
                              to: root.height/4;
                              duration: 10000;
                              running: false }
        }
    }
}
```

We add the *ImageParticle* to paint the particle using an image.

```
ImageParticle {
    id: particle
    system: sysSeason
    source: "resources/particle.png"
    color:" yellow"
    groups: ["G"]
}
```

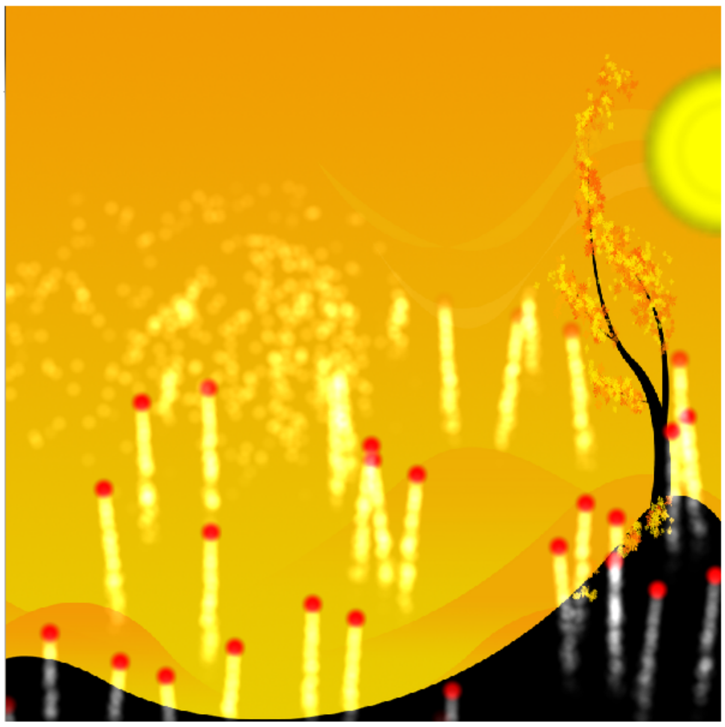
Then we add the firework animation effect using the `Emitter`, `TrailEmitter`, `GroupGoal`, `ParticlesGroup` and `ImageParticles` types as we have seen before in the `Particles` article.

```
// ImageParticle to render the firework particles
ImageParticle {
    system: sysSeason
    id: fireWorkParticle
    source: "resources/particle.png"
    color: "red"
    groups: ["D"]
}

//Emitter to creates and emits the firework particles
Emitter {
    id: fireworksEmitter
    enabled: false
    group: "D"
    system: sysSeason
    lifeSpan: 3000
    anchors.bottom: parent.bottom
    width: parent.width
    velocity : PointDirection {y: -120 ; xVariation: 16}
    size: 20
    GroupGoal {
        groups: ["D"]
        goalState: "lighting"
        jump: true
        system: sysSeason
        y: - root.height / 2
        width: root.width
        height: 10
    }

// TrailEmitter to simulate the smoke
TrailEmitter {
    id: trailEmitter
    system: sysSeason
    group: "E"
    follow: "D"
    enabled: false
    anchors.fill: parent
    emitRatePerParticle: 80
    velocity: PointDirection {yVariation: 16; xVariation: 5}
    acceleration: PointDirection {y: -16}
}


// ParticlesGroup to simulate the explosion
ParticleGroup {
    name: "lighting"
    duration: 300
    system: sysSeason

    TrailEmitter {
        enabled: true
        anchors.fill: parent
        group: "F"
        emitRatePerParticle: 80
        lifeSpan: 2000
        velocity: AngleDirection {magnitude: 64; angleVariation: 360}
    }

}
```

In the *toSummer* function, we disable previous emitters and enable the *sunEmitter*, *firework-sEmitter* and *trailEmitter*, and run *runAnimation* to move the emitter.

```
function toSummer() {
    state = "summer"

    butterFlyEmitter.enabled = false
    flowerEmitter.enabled = false

    sunEmitter.enabled = true
    fireWorksEmitter.enabled = true
    trailEmitter.enabled = true
    sunAnimation.running = true
}
```

The result should look like this:



4.6 Autumn Animation

In Autumn, we want to display some leaves falling down from the top of the window with a wind effect. To achieve this, we first add an *autumnEmitter* that emits the particles from the top of the window. This is quite similair to the *snowEmitter* we saw above:

```
Emitter {
    id: autumnEmitter
    enabled: false
    system: sysSeason
    group: "H"
    lifeSpan: 8000
    anchors{
        left: parent.left
        right: parent.right
        top: parent.top
    }
    velocity : AngleDirection { angle: 90;
                             angleVariation : 20;
                             magnitude: 100 }
    size: 40
    sizeVariation: 20
}
```

Then we add an *ImageParticle* to render the leaf particles using an image. The *ImageParticle* should belong to the same logical group as our *autumnEmitter*:

```
ImageParticle {
    id: leaf
    system: sysSeason
    source: "resources/autumn_leaf.png"
    groups: ["H"]
}
```

To add some effects, we will use an *Affector* that will generate a wind effect. For this, we will be using the Wander[2] affector that allows particles to randomly vary their trajectory:

```
Wander {
       id: wanderer
       enabled: false
       system: sysSeason
       anchors.fill: parent
       xVariance: 360;
       pace: 300;
   }
```

And That's it! Now we just need to disable the previous emitter and enable the *autumnEmitter* and the *wanderer* affector in our *toAutumn()* function:

```
    function toAutumn() {

    state = "autumn"

    summerEmitter.enabled = false
```

[2]http://qt-project.org/doc/qt-5.0/qtquick/qml-qtquick-particles2-wander.html

```
    fireworksEmitter.enabled = false

    autumnEmitter.enabled = true
    wanderer.enabled = true
}
```

Note: We created a similar animation for Winter, but with a different background, different particles displayed and some wand effects

Our Autumn animation will look like this:

In order to keep the same animation in the Winter, once we switch seasons, we need to disable the emitter and affecter above and enable the *snowEmitter* as follows:

```
function  toWinter (){
    state = ""

    autumnEmitter.enabled = false
    wanderer.enabled = false

snowEmitter.enabled = true
}
```

4.7 Shader Effect

Now we want to display a hot air balloon moving up from the bottom of the window and have a flag attached to it. For this we need two images:

One to simulate the hot air balloon with a `NumberAnimation` to make it move from the bottom to the top of the window:

```
Image {
    id: ballon
    x: root.width / 2 - width/2
    y: root.height
    source: "resources/ballonAir.png"

    NumberAnimation on y {  id: ballonAnimation;
                            running: false;
                            from: root.height;
                            to: - height * 2;
                            duration: 15000 }
}

 A second for the flag to be attached to the balloon:

Image {
    id: welcome_flag
    anchors.top: ballon.bottom
    anchors.horizontalCenter: ballon.horizontalCenter
    source: "resources/welcome.png";
}
```

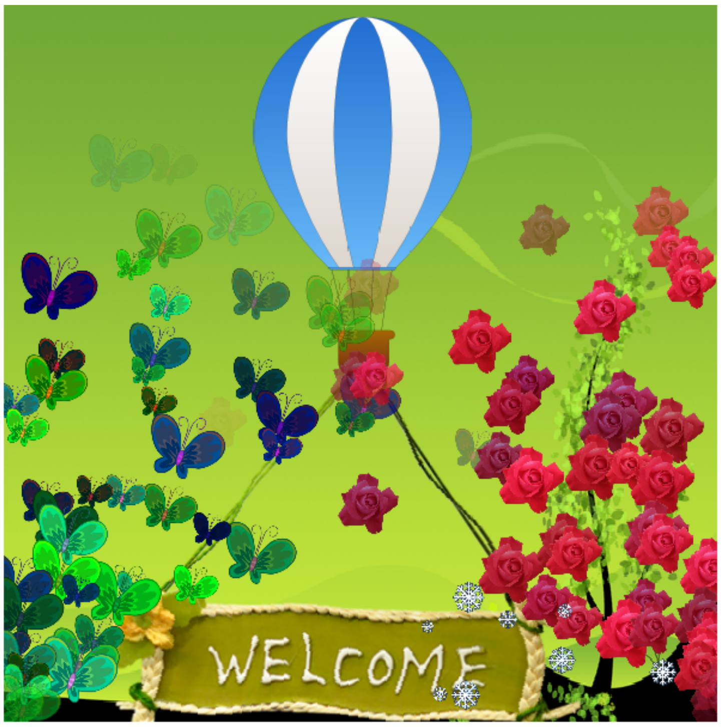
To simulate the wind effect on the flag, we add a fragment shader program via the `ShaderEffect` type:

```
ShaderEffect {
    id: shaderEffect
    anchors.fill: welcome_flag
    property variant source: welcome_flag
    property real amplitude: 0.01
    property real frequency: 20
    property real time: 0

    NumberAnimation on time { loops: Animation.Infinite;
                              from: 0;
                              to: Math.PI * 2;
                              duration: 600 }

    fragmentShader:

        "uniform lowp float qt_Opacity;
         uniform highp float amplitude;
         uniform highp float frequency;
         uniform highp float time;
         uniform sampler2D source;
         varying highp vec2 qt_TexCoord0;
         void main() {
            highp vec2 p = sin(time + frequency * qt_TexCoord0);
            gl_FragColor = texture2D(source, qt_TexCoord0  +
                        amplitude * vec2(p.y, -p.x)) * qt_Opacity;
         }";
}
```

We want to display the balloon with the flag in the Spring season so in the related function, we run the animation related to the balloon image.

```
function toSpring()
{
    //...
    balloonAnimation.running = true
}
```

Now if you run the code, you should be able to visualize the air balloon animation.



Note: The full source code of this chapter is provided in the *particles_seasons.qml* file.

4.8 Summary

In this tutorial, we went through the *Particles* module in Qt Quick and the use of *Shaders* to apply advanced animation effects. We also provided an example combining those technics. For more details concerning *Particles* and *Shaders* effects, refer to these links:

- http://qt-project.org/doc/qt-5.0/qtquick/qtquick-particles2-qml-particlesystem.html
- http://qt-project.org/doc/qt-5.0/qtmultimedia/multimedia-video-qmlvideofx.html
- http://www.lighthouse3d.com/opengl/glsl/