Developing an arcade game with Qt3D

Release 0.1 (default)

DIgia, Qt Learning

February 28, 2013

Contents

CHAPTER 1

About this Guide

1.1 Why should you read this guide?

The Qt3D module is a set of APIs that use OpenGL and aim on making 3D development easier and more platform independent. It includes features like asset loading, shapes, texture management and shaders.

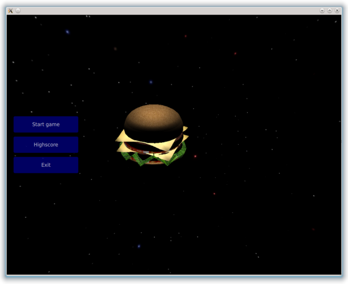
Qt3D has both, a C++ and a Qt Quick API.

The main aim of the C++ API is to make 3D programming platform independent. You don't have to worry anymore what 3D architecture in use. It should be no matter whether it is desktop or embedded OpenGL or whether a fixed or dynamic pipeline is used.

The Qt3D Qt Quick module provides more abstraction and makes it possible to write complete 3D applications in Qt Quick only. Using the Qt3D Qt Quick module, you can very easily mix 2D and 3D elements, implement head up displays overlaying a 3D scene, or even embed 3D elements in a 2D Qt Quick user interface.

Qt3D was initially developed as an add-on project in Qt4. Today, it is a part of Qt5. It is a good time to take a closer look on this in an hands-on example and get ready to use it in the future.

The purpose of this guide is to give a brief overview of the Qt3D Qt Quick by walking the reader through the development of an arcade game:



This guide will show how the 2D and 3D worlds can be combined in one application. Additionally we will talk about basic lighting, material and texture topics. While developing the game, you will get an overview of major Qt Quick elements[1] provided by the Qt3D module and learn how to use them.

This guide uses QML only without any C++ code. A solid Qt Quick knowledge is a prerequisite for reading this guide. Additionally, some basic understanding of computer graphics, OpenGL[2] in general, OpenGL shaders and GLSL (OpenGL Shading Language) is required. If you do not have sufficient knowledge in OpenGL, you should strongly consider reading the OpenGL tutorial[3] first.

In this guide, we will focus on 3D and graphic aspects. Generic aspects of application development for desktop and mobile are in focus in other guides. If you are interested in this, consider reading them first[4].

After completion of this guide, you should have a good understanding of how the Qt3D Qt Quick API works as well as how to develop basic desktop and mobile applications with this technology.

1.2 Get the source code and the guide in different formats

A .zip file that contains the source code of each chapter is provided here:

Source code[5]

The guide is available in the following formats:

- PDF[6]
- ePub[7] for ebook readers.
- Qt Help[8] for Qt Assistant and Qt Creator.

1.3 License

[1]http://qt-project.org/doc/qt-5.0/qt3d-qml3d.html
[2]http://www.opengl.org
[3]http://qt-project.org/wiki/Developer-Guides/index.html
[4]http://qt-project.org/wiki/Developer-Guides/
[5]http://releases.qt-project.org/learning/developerguides/gamedevelopmentqt3d/completegame_src.zip
[6]http://releases.qt-project.org/learning/developerguides/gamedevelopmentqt3d/GameDevelopmentQt3D.pdf
[7]http://releases.qt-project.org/learning/developerguides/gamedevelopmentqt3d/GameDevelopmentQt3D.epub
[8]http://releases.qt-project.org/learning/developerguides/gamedevelopmentqt3d/GameDevelopmentQt3D.qch

What's Next?

In the next chapter we will explain the application concept and look at small first examples.

CHAPTER 2

How we Proceed

In the following chapters, we will start with the development of our game. We will proceed step-by-step and take a deeper look on Qt3D Qt Qtuick API and related technologies. The source code archive accompanying this guide contains the final version of the application. Since this guide in on an advanced level, we will discuss relevant code samples directly taken from this version and omit making small applications for each topic.

First, we will cover basics of the Qt Quick Qt3D API. After this, we learn how to load models and textures. We will also cover dynamic model creation, making Head up Displays, states, a game menu, and the usage of GLSL shaders.

In the course of the guide, we will explain basics of Qt Quick Qt3D API. Nevertheless, you should read the detailed description provided in the Qt documentation[1].

As previously mentioned, we use only QML in the development in order to stay focused on Qt Quick. This is sufficient for many basic use cases. Qt3D's C++ API provides more features and a better control over the elements imported in Qt Quick. When developing a more complex game or a real 3D application, you will sooner or later need to define your own modules in C++ and exposing them to Qt Quick. This is a more advanced topic out of scope of this guide.

[1]http://qt-project.org/doc/qt-5.0/qt3d-reference.html

CHAPTER 3

Overview of the “SpaceBurger” Game

3.1 Application idea

Our game will be called *SpaceBurger*. Playing this game, your mission is to steer a hamburger through the darkness of the space and to try to hit as many onion rings as possible. Each onion ring brings additional scope. During the flight, it is possible increase maneuverability and firepower by hitting small power-ups. The player has to pass multiple levels including a special challenge at the end of every level: a fight against a boss* enemy. Once the boss has been defeated, the next level begins.

3.2 Controls

While having to hit several targets along the way, the hamburger is shown from behind. It can be controlled using the `A` (left), `S` (down), `D` (right) and `W` (up) keys. We will use basic movement equations to achieve realistic flight behavior.

The fight against the boss enemy at the end of every level is shown from the top view. During the fight, the hamburger can then only be moved left and right. Weapons can be fired using the `space` key.

3.3 Game menu

When starting the application, a game menu will be displayed. The player can start a new game and review the achieved results in a highscore table.

CHAPTER 4

Qt3D Basics

In order to starting using Qt3D API in Qt Quick you need to import it in your application with the statement:

```
import Qt3D 1.0
```

After this, you can load items provided by the Qt3D module. Most important from them are:

*Viewport** The Viewport[1] element specifies a viewport for the whole scene. It is the root element and defines the camera and (scene-)lights as well as rendering parameters. It is usually the outermost element in a 3D scene.

*Camera** The Camera[2] element is assigned to the camera property of the viewport. It defines a viewing position and direction as well as the projection. Furthermore stereo projections are supported.

*Item3D** The Item3D[3] is used for creating visible 3D objects in a scene. It defines basic parameters and methods for manipulating an object. To create a visible object, a mesh has to be specified for the Item3D. Futhermore tree structures can be built out of Item3Ds which allows the creation of logical groups. Children of an Item3D are placed relatively to their parent object in the 3D scene. I.e. if the parent object is moved or rotated, all the children will also be rotated.

*Mesh** The Mesh[4] is used to load geometry in such a way that it can be used in Qt3D. File loading is done automatically after a filename is specified. The mesh element chooses the appropriate model loader from the file ending. The supported model formats are e.g. 3ds, dae, bez and obj.

*Effect** An Effect[5] defines a very basic and simple way on how an item is rendered on the screen. With it simple lighting, material and texture effects can be achieved.

*ShaderProgram** The ShaderProgram[6] element is derived from Effect and gives

[1]http://qt-project.org/doc/qt-5.0/qml-viewport.html
[2]http://qt-project.org/doc/qt-5.0/qml-camera.html
[3]http://qt-project.org/doc/qt-5.0/qml-item3d.html
[4]http://qt-project.org/doc/qt-5.0/qml-mesh.html
[5]http://qt-project.org/doc/qt-5.0/qml-effect.html
[6]http://qt-project.org/doc/qt-5.0/qml-shaderprogram.html

the user the means for creating custom shader programs in GLSL. You can specify a fragment and a vertex shader. The texture property inherited from the Effect element will be maped to the *qt_Texture0* in the shader program code. The ShaderProgram automatically binds custom properties to your fragment and vertex shaders if they exist under the same name. If you want e.g. to specify more then one texture you may do so by adding a string property with the path to your texture.

*Material** Materials[7] provide some information for an effect, like lighting properties and colors.

*Transformations** There are currently four transformations available in Qt3D: Rotation3D[8], Translation3D[9], Scale3D[10] and LookAtTransform[11]. All of these can be applied to an Item3D and rotate, translate, scale or change the orientation of an item. The order in which the transformations are specified are very important for the result. It makes a difference if an item is rotated first and then translated or the other way round.

4.1 Before you try the first example

Qt3D is available for Qt 4.8 as an add-on. Qt5 is the first Qt version where Qt3D is available as an Essential Module[12]. In this guide, we will use Qt5 only. If required, the game can be ported to Qt4.8 with a minimal effort. If you already have installed Qt5, please make sure that OpenGL is supported. More information where to download Qt5 and how get it installed is available on the Qt Project homepage[13].

In-line examples and the final game are available as QML files. Please use the program *qmlscene* from the *qtbase/bin* directory to try the code.

What's Next?

Next we will be using the elements explained above in a very simple example.

[7]http://qt-project.org/doc/qt-5.0/qml-material.html
[8]http://qt-project.org/doc/qt-5.0/qml-rotation3d.html
[9]http://qt-project.org/doc/qt-5.0/qml-translation3d.html
[10]http://qt-project.org/doc/qt-5.0/qml-scale3d.html
[11]http://qt-project.org/doc/qt-5.0/qml-lookattransform.html
[12]http://qt-project.org/wiki/Qt-Essentials-Modules
[13]http://qt-project.org/wiki/Qt_5.0

CHAPTER 5

`Hello world` in Qt3D

Every scene is rendered into a *Viewport* element which can be used as any Qt Quick element. It can be anchored, have a size and contain any other items:

```
// game.qml
import QtQuick 2.0
import Qt3D 1.0

// A Viewport element with defined geometries
Viewport {
    id: root
    width: 300
    height: 200
}
```

When running the code above, a black rectangle is displayed on the screen. It is not that exciting yet... :-)

5.1 Loading a model

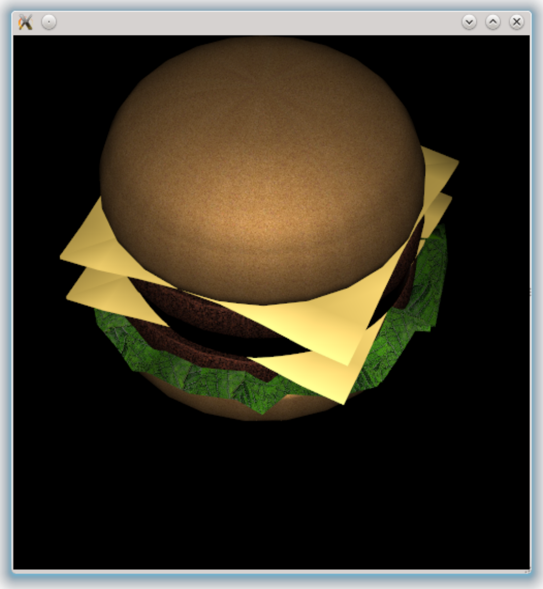
In order to fill our empty scene with some exciting stuff, a model will be loaded and placed at the origin (where the *x*,‘y‘ and *z* coordinates are set to *0*) of the scene.

Qt3D has support for some of the most common model formats. For displaying them into a scene, we simply create a *mesh* and apply it to an *Item3D’s* mesh property:

```
// game.qml
import QtQuick 2.0
import Qt3D 1.0

// A Viewport element with defined geometries
Viewport {
    id: root
    width: 300
    height: 200

    Item3D {
      id:hamburger
      scale: 0.1
      mesh: Mesh {
          source: "hamburger/models/hamburger.dae"
      }
    }
}
```

In this example, we have loaded the geometry from *hamburger.dae* and scaled it to 10% of its original size. After we apply the *source* property, the model-loading process is started. Note that the loaded geometry is not only restricted to vertices and indices, it can also include materials and textures.

When you execute this code, a hamburger should be displayed in front of the camera. You can viewed from a different perspective by dragging and scrolling the mouse. It looks like this:



5.2 Where to get 3D models?

Making 3D models requires quite some effort and sufficient skill. You might want to re-use general or simple 3D models from other projects. Even though there are plenty of sources on the Internet where one can get 3D models, you should proceed carefully check license conditions. Make sure that the author on download page is the real author and you may use this model for your purposes.

Here are some good sources where you search for free good-quality 3D models:

http://archive3d.net/ : Lots of free and high-quality 3D models, but sometimes the models have too many polygons for a real time application.

http://sketchup.google.com/3dwarehouse/ : The Google 3D warehouse is probably the best source to get free 3D models on the Internet. Users can publish their selfmade models that are usually made using Google Sketchup[1]. Most models are therefore offered in the Google Sketchup format (.skp). If you want to use them in your application, you first have to convert them to Collada (.dae). The best way to do this is by downloading Google Sketchup and exporting the files from there. There is also an import plugin for 3D Max 2010 that should work fine. The models in the 3D warehouse are of variable quality, but the license is free for commercial use.

http://thefree3dmodels.com/ : Lots of free high-quality 3D models under the Attribution-Noncommercial-Share Alike 3.0 license.

What's Next?

In the next chapter, we will see how to use the *Camera* element.

[1]http://sketchup.google.com/

CHAPTER 6

Using a Camera

In the previous chapter we used the *Viewport* element to render a scene with a model placed at the origin. When you look at this 3D scene rendered on your monitor, you still see a 2D picture. Quite some computing is required for this. It is done by OpenGL. A mapping of a 3D scene geometry onto a 2D area on your computer monitor (this area is also called the *viewport*) requires a transformation matrix. This matrix consists of a camera position, a type of projection and a viewing vector. The OpenGL term for this matrix is *modelview-projection matrix*. Please have a look at the OpenGL Tutorial[1] for more details.

Qt3D offers a very convenient way to deal with projections as described above, the Camera[2] element. The *Camera* element has the properties to define position, projection type, near and far planes. These properties are used as parameters to calculate according modelview-projection matrix and and projection itself.

6.1 Animating the Camera's position

In the next example, we want to use the camera to rotate around our model, which has been placed at the origin.

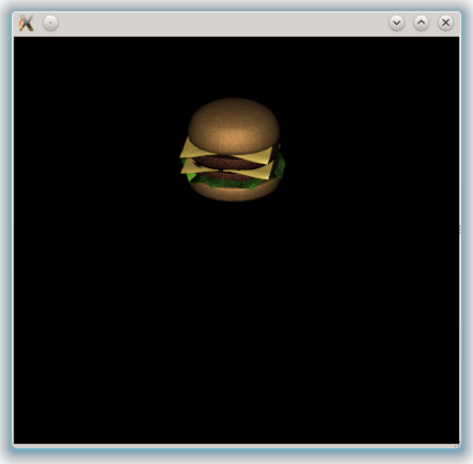
The only thing we have to do for this is creating a new *Camera* element and assign it to the *camera* property of the Viewport[3]. The *eye* property is the position of the camera and the *center* is basically the point in the 3D space at which the camera is looking. The default value is the origin so the property would be dispensable in this example:

```
//game.qml
...
Viewport {
...
  //The game camera
  camera: Camera {
        id: cam
      property real angle:0
```

[1]http://qt-project.org/wiki/Developer-Guides/
[2]http://qt-project.org/doc/qt-5.0/qml-camera.html
[3]http://qt-project.org/doc/qt-5.0/qml-viewport.html

```
        center: Qt.vector3d(0,0,0)
      eye: Qt.vector3d(20    Math.sin(angle), 10, 20*Math.cos(angle))
      NumberAnimation on angle{
          to: 100
          duration: 1000000
      }
  }
...
}
```

The rotation around the origin has been created in this example by using the sine and cosine together with a *NumberAnimation* on a custom property called *angle*. The angle is increasing from 0 to 100 over a big enough timespan.



What's Next?

Next we will see how to use a *Skybox* in order to visualize stars in space.

CHAPTER 7

Skybox

At the current stage of the development, we have just a hamburger* and a camera which moves around it. Lets create a feeling of being in the space and add some stars.

There are several ways of accomplishing this. One way is to use Qt Quick particles, specifically one for each star. We can use normal spheres for planets and suns. All these however, are very involved topics that can push us into the realm of topics beyond the scope of this guide. We will go another way and use a technique called skybox* to model stars and suns.

Skybox is a cube around the camera that creates the illusion of a distant 3D surrounding. A texture is projected on every side of the cube. The skybox usually does not move with the viewer in the 3D scene so it creates the illusion of being very far away. It can be used to render very far away mountains, clouds or, in our case, stars when flying through space. Because a skybox is a 6-sided cube, 6 textures are required - one for every side of the cube.

There are several sources of skybox textures on the internet. The problem is however, that a texture is projected onto a huge area on the screen and therefore has to be in a very high resolution (e.g. 1024x1024) to be of good quality. Because of this, we recommend using applications which are specialized for creating skybox textures:

> Terragen[1]: A terrain generator for photo realistic terrains. It is very easy to use and is available as a feature limited freeware application.
>
> Spacecape[2]: An open source project for creating space skyboxes containing several layers of nebulas, suns and stars.

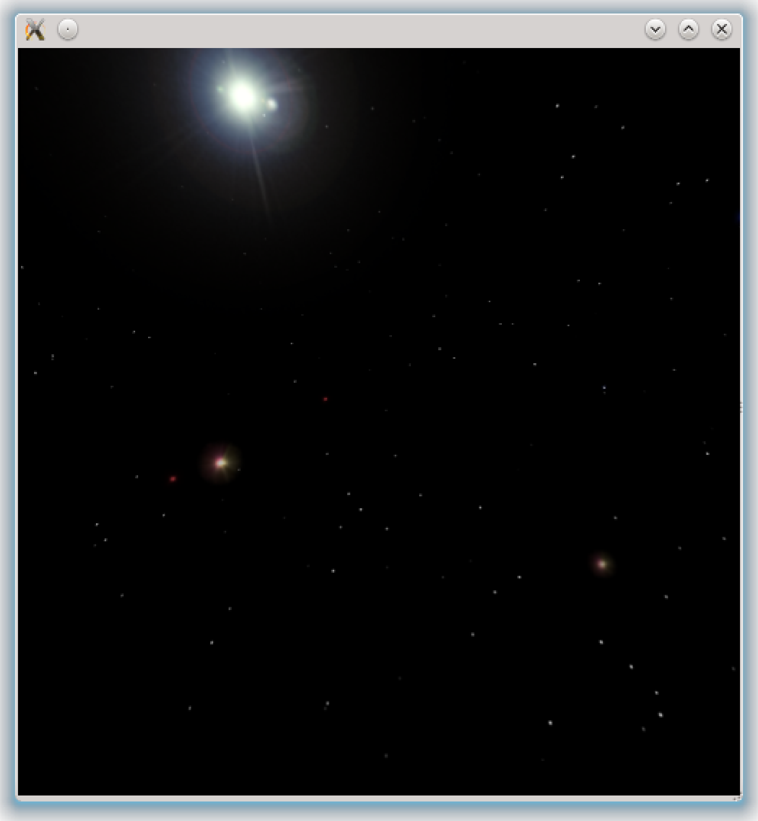
Fortunately, Qt3D has a built-in element called Skybox[3] that does exactly what we are looking for. It takes a source folder which should contain 6 textures. The textures can have a random name but must contain the orientation in the skybox (north*, *south*, *east*, *west*, *up* and *down*). A texture could have the name *space_west.png*.

```
// game.qml
...
Viewport {
...
```

[1] http://www.planetside.co.uk/
[2] http://sourceforge.net/projects/spacescape/
[3] http://qt-project.org/doc/qt-5.0/qml-skybox.html

```
  Skybox{
    //The folder containing the skybox textures
    source: "space"
  }
...
}
```

The following skybox was made using the *Spacescape*:



What's Next?

Next we will see how to create the player object and move it in the 3D world.

CHAPTER 8

Player Movement

In our game, we want the player to control the hamburger* movement using the keyboard input. It should be possible to move it from left to right and up and down. Furthermore, the orientation should depend on the current acceleration.

In order to achieve realistic flight behavior, the movement is controlled using some basic movement equations. For the sake of good order, we will be implementing the logic code of our game into a new *Gamelogic.qml* file.

8.1 Update-timer

Before we implement the movement equations, we do need an *update* timer that periodically updates the hamburger*'s position and acceleration. An interval value of *50ms* should be sufficient for achieving fluent movement.

Later on, within the *onTriggereed* signal handler, the movement equations will be processed and the hamburger* position should be updated.

```
//Gamelogic.qml

import QtQuick 2.0

Item {

    Timer {
       id: gameTimer
       running: true
       interval: 50
       repeat: true
       onTriggered: {
       ...
        // update position...
       }
    }
}
```

8.2 Keyinput

To handle the keybord input, we first need to set the value of the *focus* property of the root item to *true* to handle the key events. We also need four new variables (one for each key), which are either set to *true* or *false* depending on the press state of the keys. Within *onPressed* and *onReleased* we handle the eventual key events as shown in the code below:

> This whole construct is necessary because we want to allow the user to press and hold more then one key at a time. We also need four new variables (one for each key), which are either set to *true* or *false* depending on the press state of the keys. Movement processing is then performed in the update-timer's *onTriggered* signal:

```
Item {
  ...
  focus: true
  property bool upPressed: false
  property bool downPressed: false
  property bool leftPressed: false
  property bool rightPressed: false

  //Handling of basic key events
  Keys.onPressed: {
      if(event.key == Qt.Key_A)
          leftPressed = true
      if(event.key == Qt.Key_D)
          rightPressed = true
      if(event.key == Qt.Key_W)
          upPressed = true
      if(event.key == Qt.Key_S)
          downPressed = true
      if(event.key == Qt.Key_Space)
          fireLaser();
  }
  Keys.onReleased: {
      if(event.key == Qt.Key_A)
          leftPressed = false
      if(event.key == Qt.Key_D)
          rightPressed = false
      if(event.key == Qt.Key_W)
          upPressed = false
      if(event.key == Qt.Key_S)
          downPressed = false
  }
}
```

Later we will perform the movement processing in the update-timer's *onTriggered* signal handler.

Then we have to instantiate the *Gamelogic* component in the main *game.qml* file.

```
//game.qml

Viewport {
  ...
  Gamelogic {id: gameLogic}
  ...
}
```

8.3 Basic motion equations

In the our *SpaceBurger* game, the hamburger* will be seen from the back (if there is any for a *hamburger*). So we set the camera's eye position to (*0, 0,-30*). The player can then move it on the *y* and *x* axes. To make sure that the *hamburger* will remain in the screen view, we define *x* and *y* boundaries that will restrict the movement. The *x* and *y* bounds could be calculated from the camera parameters, but a we can simply set *4.5* value for the x-bound and *5* value for the y-bound.

Note: The *y* and *x* bound parameters will change with the aspect ratio of the viewport you are using and in general with the camera parameters!

```
//game.qml
...
Viewport {
  ...
  property real x_bound: 4.5
  property real y_bound: 5
  ...
}
```

To move the hamburger* object, we will be using the two basic motion equations for constant acceleration[1] . The motion equations are based on the acceleration, the current speed and the position values.

> //Velocity is acceleration multiplied with time plus the initial speed v = a t + v0
> //Distance is velocity multiplied with time plus the initial distance s = v t + s0

We create a new *Player.qml* file to define the *Hamburger* as a separate component, and calculate its speed and acceleration for the *x* and *y* axes an. Those values are then saved in the *vx*, *vy*, *ax* and *ay* properties as shown in the code below:

```
//Player.qml
import QtQuick 2.0
import Qt3D 1.0

Item3D {

    property real vx: 0
    property real vy: 0

    property real ax: 0
    property real ay: 0

    mesh: Mesh { source: "hamburger/models/hamburger.dae" }
```

[1] http://en.wikipedia.org/wiki/Motion_equation#Constant_linear_acceleration

```
    scale: 0.1
}
```

Since we can build a tree structure with an *Item3D*, we will define a root *Item3D* for the top level which contains all the visible 3D items of the scene. The *player* object will then be a child of an *Item3D* element. Furthermore, we set the *camera* to a position behind the burger:

```
//game.qml

Viewport {
    ...
    Item3D {
        id: level

        Player {
            id: player
        }
    }

    camera: Camera {
       id: cam
       eye: Qt.vector3d(0, 0,-30)
    }
    ...
}
```

We will also define a variable called *maneuverability* in the *Gamelogic.qml* in order to have better control over the flight parameters. A convenient value for the *maneuverability* will be 0.3:

```
// Gamelogic.qml
...
property real maneuverability: 0.3
//The game timer is our event loop. It processes the key events
//and updates the position of the hamburger
Timer {
    id: gameTimer
    running: true
    interval: 50
    repeat: true
    onTriggered: {
        //Velocity is updated
        player.vx+=player.ax    0.05
        player.vy+=player.ay    0.05
        //Acceleration is updated
        player.ax=(player.ax+maneuverability    leftPressed
      + maneuverability*rightPressed)/1.1
        player.ay=(player.ay+maneuverability    downPressed
      + maneuverability*upPressed)/1.1
        //Position is updated
        player.position.x += player.vx    0.05
        player.position.y += player.vy    0.05
        //If the player exceeds a boundary, the movement is stopped
        if (player.position.x>x_bound) {
            player.position.x = x_bound
            player.vx = 0;
            if (player.ax>0)
```

```
                player.ax = 0
        }
        else if (player.position.x<-x_bound) {
            player.position.x = -x_bound
            player.vx = 0
            if (player.ax<0)
                player.ax = 0
        }
        else if (player.position.y<-y_bound) {
            player.position.y = -y_bound
            player.vy = 0
            if (player.ay<0)
                player.ay = 0
        }
        else if (player.position.y>y_bound) {
            player.position.y = y_bound
            player.vy = 0
            if (player.ay>0)
                player.ay = 0
        }
    }
}
...
```

Now we should be able to move the hamburger* smoothly over the screen and the movement should stop on the viewport boundaries.

Note: For a realistic flight behavior, the hamburger* should turn into the flight direction.

8.4 Transformations

There are currently four transformation types available in the *Qt3D* module: *Rotation3D*, *Scale3D*, *Translation3D* and *LookAtTransform*. The names should be fairly self-explanatory.

One or more transformations can be applied to an *Item3D*'s *transform* or *pretransform* properties. The *pretransform* property however is intended to transform the model before all other transformations, because it may be in an unconventional scale, rotation or translation after loading.

As explained above, we want the hamburger* to rotate in the flight direction, so we need to achieve three things:

- When moving *hamburger* along the *x* axis (left or right), the *hamburger* should roll a bit into flight direction. (the rotation axis is the *z* axis)
- When moving *hamburger* along the *x* axis (left or right), it should move the nose in flight direction. (the rotation axis is the *y* axis)
- When moving *hamburger* along the *y* axis (up or down), the *hamburger* should move its front up or down. (the rotation axis is the *x* axis)

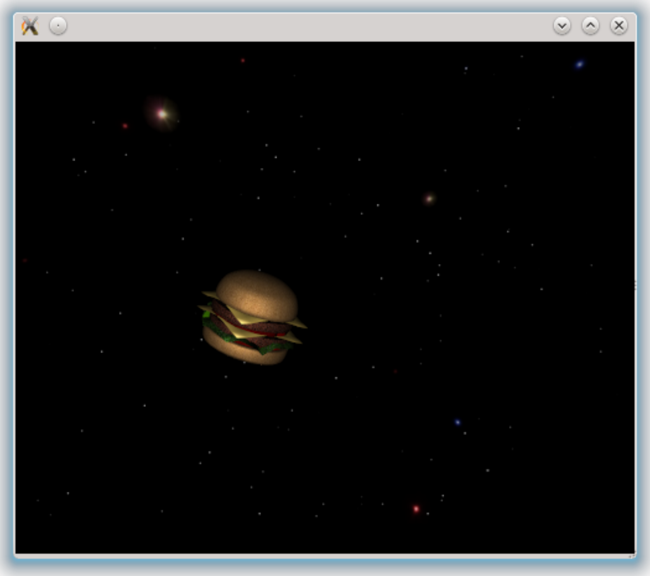
Now we can add the different transformations to the *transform* property in the *Player.qml* and specify their axis. We are connecting the angle of each rotation directly to the acceleration, which will have a fairly good-looking result. The scalar factors have been obtained by trial and error:

```
//Player.qml
...
transform: [
    Rotation3D {
        angle: -10    ay
        axis: "1, 0, 0"
    },
    Rotation3D {
        angle: 5    ax
        axis: "0, 1, 0"
    },
    Rotation3D {
        angle: -20    ax
        axis: "0, 0, 1"
    }
]
...
```

When moving the hamburger*, you might notice that the rolling behavior is a bit strange. That is because the balance point of the object is not at the origin. We can however correct this very easily by applying a *Translation3D* to the *pretransform* property. In addition to this, the scaling was moved into *pretransform* as well (i.e we have to remove the scale property in the *Player*). Furthermore a rotation of *45°* on the *y* axis was added for aesthetic reasons.

```
//Player.qml
pretransform: [
    Scale3D {
        scale: 0.1
    },
//Moving the objects origin into the balance point
    Translation3D {
        translate: "0,-1,0"
    },
    Rotation3D {
        angle: 45
        axis: "0, 1, 0"
    }
]
...
```

The *hamburger** object could now be controlled by the player:



What's Next?

Next we add the *onion rings* to be hit by the player in our game. For this, we will introduce dynamic object creation, collision detection and how to use textures and predefined shapes.

CHAPTER 9

Moving targets

In our *SpaceBurger* game, the flying hamburger* should hit *onion rings* moving toward the player as targets to increase the player's score. In this chapter, we will see how to implement the *onion rings*.

9.1 Onion Rings

Every onion ring* will be rendered using a *quad* with a semi transparent texture. To create the *quad*, we'll be using the *Qt3D Shapes* module, which comes with many predefined shapes such as *quads*, *cylinders*, *teapots* etc.

So first, we create a new *Target.qml* file to implement the onion ring component* which consist of a *Quad* element: .. code-block:: js

```
//Target.qml

import QtQuick 2.0
import Qt3D 1.0
import Qt3D.Shapes 1.0

Quad {
  id: root
}
```

The *Quad* element is lying on the *(x,z)* plane by default. However, to face the camera, we need to apply a pretransform as follows:

```
//Target.qml
...
Quad {
  id: root

  pretransform: [
     Rotation3D { axis: "1,0,0"; angle: 90}
  ]
}
```

We also want to apply a semi transparent texture onto the *quad* where only the onion ring* part of the texture is visible. That means, however, that we need an image format that supports transparency. *PNG* format is a convenient choice.

Furthermore, we want to have some transparency on the non transparent parts of the onion ring*. For that we add a Material[1] with a *diffuseColor* that has an *alpha* value of *0.9* so that *onion ring* is slightly transparent. We also want to have the *onion ring* glowing a bit, so we add a red *emittedLight*:

```
//Target.qml
...
Quad {
    ...
  effect: Effect {
       blending: true
       material: Material {
         textureUrl: "onion.png"
         emittedLight: Qt.rgba(1,0.8,0.8,1)
         diffuseColor: Qt.rgba(1,1,1,0.9)
       }
    }
    ...
  }
```

Since we are using blending for the transparent objects, we have to consider few things: First of all the blending property in the *Effect* has to be set. This will also override the viewport specific setting for *alpha* blending. When using blending, items have to be painted from back to front. This means that items which are farther away from the viewer have to be painted first, which requires to sort the items. Fortunately, *Qt3D* does this for us automatically if we set the *sortChildren* property to *BackToFront* in the parent *Item3D* element.

```
//game.qml
...
Item3D {
     id: level
     sortChildren: Item3D.BackToFront
     ...
}
...
```

Note: *BackToFront* sorting works only for one hierarchy level. This means only direct children of an *Item3D* are sorted and not the children's children.

Once a *Target* is created, it should immediately start moving toward the player. We can achieve this by a adding a *NumberAnimation* on the *z* property of the *Quad*.

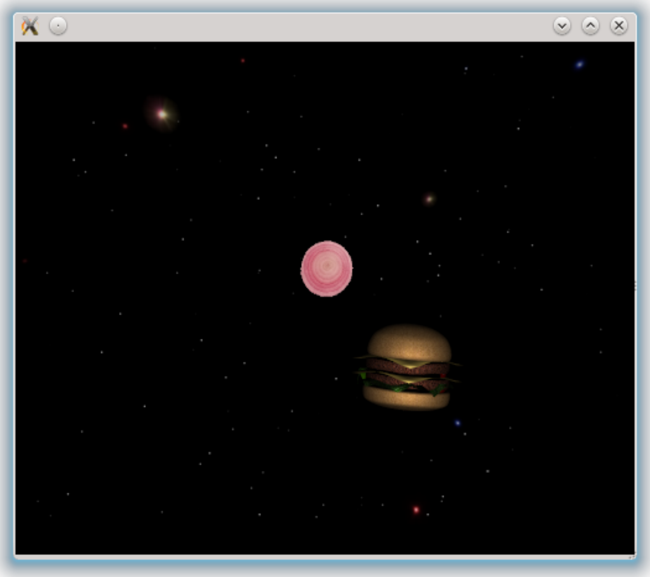
```
  //Target.qml
  ...
  Quad{
    ...
    NumberAnimation on z{
         running: true
```

[1] http://qt-project.org/doc/qt-5.0/qml-material.html

```
            duration: 10000
            from: 200
            to: -30
        }
        ...
}
```

You can now test the *Target* component by manually adding it to the level. It should then be created in the distance and fly towards the player. However, later we should create *Target* objects dynamically.

```
//game.qml
...
Item3D {
     id: level
     ...
     Target { }
     ...
}
...
```



9.2 Collision-detection

Collision-detection is not yet supported by *Qt3D* and it is not possible to get a bounding box of an *Item3D*. But still, we can implement a simple collision detection on our own.

A collision test is only performed between two objects (i.e. in our game, a collision only occurs between the onion ring* and the *hamburger* and between the weapon fire and the enemy or player). But since we will also be using collision detection for other items, we will create a new component in a *BasicGameItem.qml* file, which implements the collision detection. This component will be used as a parent item for all the components that need to implement a collision detection.

To archieve the detection we will proceed as follows:

> A target is specified for which the collision test is performed.
>
> The target element has to define a *radius* property that specifies the size of the object.
>
> The *BasicGameItem* should define a *radius* that specifies the size of the item.
>
> Every time a *positionChanged* signal is emitted, a test for collision takes place.
>
> If a collision is detected, a *collisionDetected* signal is emitted and *BasicGameItem* is destroyed afterwards.

And here is how our code looks like:

```
//BasicGameItem.qml
import QtQuick 2.0
import Qt3D 1.0

Item3D {
    id: gameItem

    signal collisionDetected(variant object)

    property variant collisionTarget: 0

    property real radius: 0.5

    //Test for a collision between the item and the target
    function testCollision(){
    if (Math.pow(x-collisionTarget.x,2)+Math.pow(y-collisionTarget.y,2)
        + Math.pow(z-collisionTarget.z,2)
        < Math.pow(radius+collisionTarget.radius,2)) {
        return true;
    }
    return false;
    }

    onPositionChanged: {
    if (collisionTarget!=0) {
        if (testCollision()) {
        collisionDetected(gameItem)
        gamenItem.destroy()
        }
    }
    }
}
```

Now, the *Target.qml* file will look like this:

```
// Target.qml

BasicGameItem {
    id: root
    Quad {
    pretransform: [
        Rotation3D { axis: "1,0,0"; angle: 90}
    ]
    effect: Effect {
        blending: true
        material: Material {
        textureUrl: "onion.png"
        emittedLight: Qt.rgba(1,0.8,0.8,1)
        diffuseColor: Qt.rgba(1,1,1,0.9)
        }
    }
    }
    NumberAnimation on z{
    running: true
    duration: 10000
    from: 200
    to: -30
    onRunningChanged: {
        if (running == false)
        root.destroy()
    }
    }
}
```

Make sure you use the *NumberAnimation* on the *BasicGameItem* and not on the *Quad*. Otherwise the detection will fail.

The collision target of our *Target* component will be the *Player* object. So we have to define a *radius* property for in the *Player* component.

```
//Player.qml

Item3D {
  ...
  property real radius: 1.5
  ...
}
```

9.3 Dynamic Object Creation

As explained above, the onion ring* targets need to be created dynamically. For that we will use a timer in *Gamelogic.qml* to create new target every 4 seconds that flies towards the player.

To create new *Target* objects, we need first to load the *Target* component. Then, we can create an instance of our *Target* component using the *createObject* method. Since we want to reuse the component several times, we will load it when starting the application in *GameLogig.qml*.

Note: If the component is loaded over the network, we first need to wait for the component to be ready before calling *createObject*

First, we define two properties in *game.qml* to store the *score* and to count the number of targets:

```
//game.qml
...
```

```
property int score: 0
property int targetCount: 0
...
```

Then we implement the target timer in the *GameLogic.qml*.

```
//GameLogic.qml
...
property variant targetComponent: Qt.createComponent("Target.qml");
....
//Timer creates targets in a certain interval
Timer {
    id: targetTimer
    interval: 4000
    repeat: true
    running: true
    onTriggered: {
    targetCount++
    var object = targetComponent.createObject(level,
         {"position.x": (Math.random()-0.5) *8,
        "position.y":  (Math.random()-0.5) *6,
        "scale": 3-0.2*targetCount, "collisionTarget": player})
    object.collisionDetected.connect(targetCollision)
    }
}
```

Once the object is created, we connect the *collisionDetected* signal to a function called *target-Collision* where the *score* property defined earlier is incremented by one.

```
//GameLogic.qml
...
Item{
  function targetCollision(sender) {
       score++;
  }
  ...
}
```

What's Next?

Next we will see how to use *States* to handle the flow of our game.

CHAPTER 10

States

In previous chapters, we have implemented most of our game logic and added many new components. In this chapter we will see how to use the *states* concept in Qt Quick to define every major event, that requires e.g. camera adjustment, or changes important parameters.

Let's summarize, once again, the game flow. Once the application is started, a game menu is shown and the player has two options: View the highscore table or start the game. When starting the game, the camera has to move to the back of the hamburger* and the keyboard controls will be enabled. The *onion rings* start to fly toward the player who has to try to hit them. After a certain number of *onion rings*, the boss enemy should appear. The final fight will take place from the birds eye view so the camera has to be moved first. Once the boss or player have been destroyed, a dialog appears to enable the player to enter his name with his score to store them in the highscore table.

We define the following states:

> The **Menu** state: it's the initial state where only the 3 buttons of the menu are shown.
>
> The **Highscore** state: it's an extension to the **Menu** state where the highscore table is also displayed.
>
> The **Enter Highscore** state: Provides a *textfield* to enter the player's name in the highscore table. The whole game scene will be frozen.
>
> The **Game** state: Moves the camera behind the *hamburger* and starts the game. The keyboard controls are enabled and the game timer starts running.
>
> The **Boss Rotation** state: Rotates the camera around the boss enemy then to a position above the scenery.
>
> The **Boss Fight** state: Adjusts the *x-bound* after the camera has moved to a different point and starts the fight against the boss enemy.

```
//game.qml
...
state: "Menu"

states:[
    State{
        name: "Menu"
        PropertyChanges {target: player; ax: 0; ay: 0; vx: 0; vy:0;
                            position: Qt.vector3d(0, 0, 0); restoreEntryValues: false}
        PropertyChanges {target: root; score: 0; targetCount:0;
                            restoreEntryValues: false}
        PropertyChanges {target: cam; center: Qt.vector3d(0, 0, 0) }

    },
    State{
        name: "Highscore"
        extend: "Menu"
    },
    State{
        name: "EnterHighscore"
    },
    State{
        name: "Game"
        PropertyChanges {target: player; position: Qt.vector3d(0, 0, 0) }
    },
    State{
        name: "BossFight"
    PropertyChanges {target: player; ay: 0; vy:0;
      position: Qt.vector3d(0, 0, 0); restoreEntryValues: false}
    },
    State{
        name: "BossRotation"
        PropertyChanges {target: player; position: Qt.vector3d(0, 0, 0) }
    }
]
...
```

Note: We will cover the implementation of the game menu and the highscore dialog in the next chapter.

The *targetTimer* and *gameTimer* should not start until either *Game* or *BossFight* state are reached:

```
//Gamelogic.qml
...
id: targetTimer
running: root.state=="Game"
...
id: gameTimer;
running: root.state=="Game"||root.state=="BossFight"
...
```

Also, The player will not be able to move on the *y* axis during the fight against the boss enemy and shooting lasers obviously must only be possible when there is a target to shoot. Later, we will be implementing a *fireLaser* function.

```
//Gamelogic.qml
...
if(event.key == Qt.Key_W && root.state == "Game")
   upPressed = true
```

```
if(event.key == Qt.Key_S && root.state == "Game")
   downPressed = true
if(event.key == Qt.Key_Space && root.state == "BossFight")
   fireLaser();
...
```

Note: Note that the default state of our game is *Menu*. As we didn't yet implement the game menu, at this stage of the implementation the user can not start the game as it's supposed to be.

What's Next?

Next we will implement the main menu for our game.

CHAPTER 11

Game Menu

In this chapter we will be implementing the game menu. With Qt Quick it's easy to mix 2D and 3D elements which enables us to add basic UI to our game.

11.1 Head-up display

A Head-up display (HUD) usually shows information to the player about the current game state and the player's conditions. There are actually three things we want to display: the level of the laser's energy, the hit points and the score.

So first we add the following properties to the player:

```
//Player.qml
...
property int hitpoints
property real maxHitPoints: 10

property int energy
property int maxEnergy: 2000
...
```

Then we want to display two *energy* bars. A red one in the center to show the player's hit points and a blue one to show the laser's energy left. The current score is displayed in the upper left corner of the viewport. To archieve that, we add a new *Hud.qml* file that consist of an *Item* containing two *Rectangles* that present the bars, and a *Text* element to display the score.

```
//Hud.qml
import QtQuick 2.0

Item {
    id: hud
      anchors.fill: parent

      Text {
          anchors.left: parent.left
          anchors.top: parent.top
          anchors.margins: 10
```

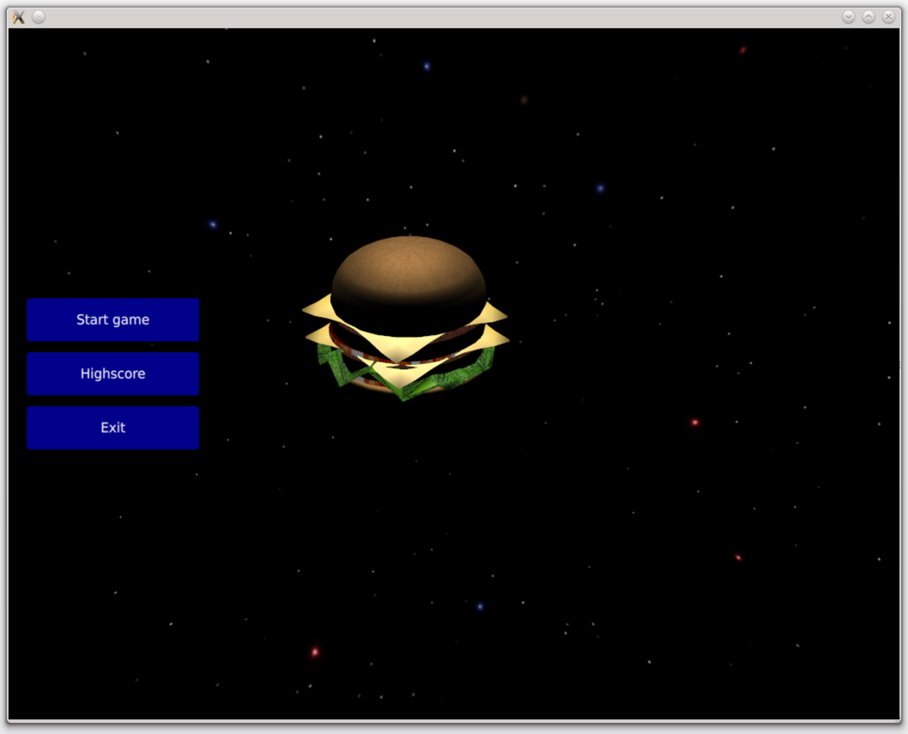
```
            text: "Score: " + score;
            style: Text.Raised
            font.pixelSize: 20
            color: "green"
        }

        Rectangle {
            anchors.top: parent.top
            anchors.topMargin: 20
            anchors.horizontalCenter: parent.horizontalCenter
            width: parent.width/2
            height: 15
            color: "transparent"
            border.color: "red"
            Rectangle{
                  anchors.left: parent.left
                  anchors.top: parent.top
                  anchors.bottom: parent.bottom
                  width: parent.width*player.hitpoints/player.maxHitPoints;
                  color: "red"
            }
        }

        Rectangle {
            anchors.right: parent.right
            anchors.rightMargin: 20
            anchors.verticalCenter: parent.verticalCenter
            height: parent.height/3
            width: 10
            color: "transparent"
            border.color: "blue"
            Rectangle{
                  anchors.right: parent.right
                  anchors.left: parent.left
                  anchors.bottom: parent.bottom
                  height: parent.height*player.energy/player.maxEnergy;
                  color: "blue"
            }
        }
}
```

Then we instantiate the *HUD* in *game.qml* as follows:

```
//game.qml
...
Viewport {
  ...
  //Head up display
  Hud {id: hud}
  ...
}
```

11.2 Game menu

Once the game is started, a menu should be displayed. This menu consists of a button group containing three buttons: *start*, *highscore* and *exit*. While the menu is displayed, the hamburger* is rotating in the background. When clicking on the “start” button, the game starts and the camera is moved behind the *hamburger*. When clicking on the “highscore” button, a new rectangle will appear and displays the highscores in a *ListView*. To exit the game the player can simply click on the *exit* button.



Before we start implementing the menu, we first have to define two missing camera movements. One is the rotation of the hamburger* while the game menu is displayed and the other moves the camera behind the *hamburger* when we start the game:

```
//game.qml
...
//The game camera
camera: Camera {
    id: cam
    property real angle:0;
    eye: Qt.vector3d(20    Math.sin(angle), 10, 20*Math.cos(angle))
    NumberAnimation on angle{
        id: hamburgerRotation
        to: 100
        running: false
        duration: 1000000;
    }
    PropertyAnimation on eye {
        id: moveBehindHamburger
        to: Qt.vector3d(0, 0,-30)
        duration: 2000
        running: false
    }
}
...
```

Then we define a new button component in a new *Button.qml* file:

```
//Button.qml
import QtQuick 2.0

//Creates a simple button that has an attribute buttonText
Rectangle {
      id:rootI
      width: 200;
      height: 50;
      signal buttonClicked();
      property variant buttonText;
      radius: 5
      border.color: "black"
      border.width: 2
      color: "darkblue"
      opacity: 1
      MouseArea {
          hoverEnabled: true;
          anchors.fill: parent;
          onClicked: buttonClicked();
          onEntered: border.color="white"
          onExited: border.color="black"
      }
      Text {
          anchors.centerIn: parent;
          text:  buttonText;
          color: "white"
      }
}
```

Next we create our menu component in a *Menu.qml* file. The menu consists of an *Item* with a *Column* containing three buttons. When a button is clicked, the appropriate state will be set in the root element (the viewport):

```
//Menu.qml
import QtQuick 2.0

Item {
    visible: false
    anchors.fill: parent
    //The button group
    Column {
        id: buttonGroup
        anchors.verticalCenter: parent.verticalCenter;
        anchors.left: parent.left;
        anchors.leftMargin: 20
        spacing: 10

      Button {
          buttonText: "Start game"
          onButtonClicked: root.state="Game"
      }

      Button {
          buttonText: "Highscore"
          onButtonClicked: root.state="Highscore"
      }

      Button {
          buttonText: "Exit"
          onButtonClicked: Qt.quit()
      }

  }
}
```

Then we add the menu to the *Viewport* in *game.qml*.

```
//game.qml
...
Viewport {
  ...
  Menu {id: gamemenu}
  ...
}
```

To save the highscore table, we will use an *SQLite* database. We will avoid discussing the detail how to *SQLite* in QML. For more detail please refer to the Qt Quick Desktop Guide[1].

For that, we create a new *gameDB.js* Stateless JavaScript library. This means that only one instance will be created for all QML file including it. The library defines the database logic as shown in the code below:

```
// gameDB.js

//making the gameDB.js a stateless library
.pragma library

.import QtQuick.LocalStorage 2.0 as Sql

// declaring a global variable for storing the database instance
var _db

//Opens the database connection
function openDB() {
    print("gameDB.createDB()")
    _db = Sql.openDatabaseSync("SpaceburgerDB","1.0","The Spaceburger Database"
      ,1000000);
    createHighscoreTable();
}

//Creates the highscore table
function createHighscoreTable() {
    print("gameDB.createTable()")
    _db.transaction( function(tx) {
  tx.executeSql("CREATE TABLE IF NOT EXISTS "
```

[1] http://qt.nokia.com/learning/guides

```
      +"highscore (score INTEGER, name TEXT)");
            });
}

//Reads the first 10 elements of the highscoretable and returns them as an array
function readHighscore() {
    print("gameDB.readHighscore()")
    var highscoreItems = {}
    _db.readTransaction( function(tx) {
            var rs = tx.executeSql("SELECT name, score FROM "
           +"highscore ORDER BY score DESC LIMIT 0,10");
            var item
            for (var i=0; i< rs.rows.length; i++) {
        item = rs.rows.item(i)
        highscoreItems[i] = item;
            }
        });

    return highscoreItems;
}

//Saves an element into the highscore table
function saveHighscore(score, name) {
    print("gameDB.saveHighscore()")
    _db.transaction( function(tx){
        tx.executeSql("INSERT INTO highscore (score, name) "
       +"VALUES(?,?)",[score, name]);
            });

}
```

Next we create the highscore table in *Menu.qml*:

```
//Menu.qml
Item {
  ...
  ListModel {
      id: highscoreModel;
  }

  Component.onCompleted: {
      GameDB.openDB();
  }

  Rectangle {
      visible: root.state=="Highscore"
      anchors.left: buttonGroup.right
      anchors.right: parent.right
      anchors.bottom: parent.bottom
      anchors.top: parent.top
      anchors.margins: 50
      radius: 5
      border.color: "black"
      border.width: 2
      color: "darkblue"
      opacity: 0.7
      Text {
          id: title
          anchors.top: parent.top
          anchors.horizontalCenter: parent.horizontalCenter
          anchors.topMargin: 20
          text: "Highscore"
          font.bold: true
          font.pointSize: 15
          color: "white"
      }
      //The highscore table
      ListView {
            id: highscore
          anchors.top: title.bottom
          anchors.topMargin: 50
          anchors.verticalCenter: parent.verticalCenter
          width: parent.width-70
          height: parent.height-title.height-50
          model: highscoreModel;
          delegate: Item {
              anchors.left: parent.left; anchors.right: parent.right
                  anchors.margins: 40
              height: 30
              Text{anchors.left: parent.left;   text: name;  font.bold: true;
                  font.pointSize: 20; color: "white"}
              Text{anchors.right: parent.right; text: score; font.bold: true;
                  font.pointSize: 20; color: "white"}
          }
      }
   }
}
```

As you might have noticed, we have created an empty *ListModel* and used it in the *ListView*. Next we are going to populate this model with the data we get out of the SQL table through the *readHighscore()* function.

The first thing to do is to import the library:

```
//Menu.qml
import "gameDB.js" as GameDB
```

Now we can read the data from the highscore table. We will do that in the *onVisibleChanged* signal handler of the highscore item, so that an update will occur every time the highscor is displayed.

We use the *GameDB's readHighscore()* function to read the highscore table from the databse parse it into the *ListModel* we have already defined:

```
//Menu.qml
...
        onVisibleChanged: {
            if (visible == true) {
                var highscoreTable=GameDB.readHighscore();
                highscoreModel.clear();
                for (var i in highscoreTable) {
                    print(highscoreTable[i])
                    highscoreModel.append(highscoreTable[i]);
                }
        }
    }
```

Adding a new highscore into the SQL table is possible once the game has been finished. A dialog is displayed that asks the player to enter his name. The name and the score will then be saved. The code of the dialog is implemeted into *HighscoreDialog.qml* as follows:

```
//HighscoreDialog.qml
import QtQuick 2.0
import "gameDB.js" as GameDB

Rectangle{
    anchors.verticalCenter: root.verticalCenter
    anchors.horizontalCenter: root.horizontalCenter
    height:170
    width:270
    radius: 5
    border.color: "black"
    border.width: 2
    color: "darkblue"
    opacity: 0.7
    visible: false
    Text{
        id: title
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: parent.top
        anchors.topMargin: 15
        text: "Enter your name:"
        font.pointSize: 17
        color: "white"
    }

    Rectangle{
        id: input
        anchors.horizontalCenter:  parent.horizontalCenter
        anchors.top:  title.bottom
        anchors.topMargin: 15
        height: 40
        width: 200
        radius: 2
        color: "lightgray"
        clip: true
        TextInput{
            id: inputField
            anchors.fill: parent
            color: "black"
            text: "Name..."
            font.pointSize: 17
        }
    }

    Button {
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 15
        anchors.right: parent.right
        anchors.rightMargin: 15
        buttonText: "OK"
        onButtonClicked: {
            GameDB.saveHighscore(score, inputField.text)
            root.state="Menu"
        }
    }
}

//main.qml
...
HighscoreDialog {id: highscoreDialog}
...
```

We now can update our states:

```
states:[
    State{
        name: "Menu"
        PropertyChanges {target: player; ax: 0; ay: 0; vx: 0; vy:0;
                            position: Qt.vector3d(0, 0, 0);  hitpoints: 2;
                            energy:2000;  restoreEntryValues: false}
        PropertyChanges {target: root; score: 0; targetCount:0;
                            restoreEntryValues: false}
        PropertyChanges {target: cam; center: Qt.vector3d(0, 0, 0) }
        PropertyChanges {target: gamemenu; visible: true;}
        PropertyChanges {target: hamburgerRotation; running: true;}
        PropertyChanges {target: hud; visible: false;}
    },
    State{
        name: "Highscore"
        extend: "Menu"
    },
    State{
        name: "EnterHighscore"
        PropertyChanges {target: hud; visible: true;}
        PropertyChanges {target: highscoreDialog; visible: true;}
    },
    State{
        name: "Game"
        PropertyChanges {target: moveBehindHamburger; running: true;}
        PropertyChanges {target: hud; visible: true;}
    },
    State{
        name: "BossFight"
        PropertyChanges {target: hud; visible: true;}
        PropertyChanges {target: player; ay: 0; vy:0;
    position: Qt.vector3d(0, 0, 0); restoreEntryValues: false}
    },
    State{
        name: "BossRotation"
    }
]
```

What's Next?

Next we implement the boss enemy that should appear at the final level.

CHAPTER 12

Boss enemy

The boss enemy appears at the end of a level after a certain number of targets has been passed.

Unlike in the first part of the level, the player observes the fight not from the back of the hamburger*, but from the top. That also means the *hamburger* (and the enemy) can only be moved on the x-axis. At the beginning of the fight, we have to set the *y* value to *0*, which we already did when we defined the states.

12.1 Camera movement

The camera will start moving when the enemy has arrived at its final position, in front of the player. To accomplish this, a *SequentialAnimation* is started which moves the camera's *center* to the enemy's position. After that, the camera pans around the enemy and at the end moves the *eye* to the top of the scene and adjusts the camera's *center* to the middle of the fighting scene.

After the animation is finished, the start of the fight is triggered by setting a new state for the root item:

```
//game.qml
...
Viewport {
  ...
  SequentialAnimation {
  id: rotateAroundBoss
  running: false
  PropertyAnimation{
      target: cam
      properties: "center"
      to: enemy.position
      duration: 400
  }
  PropertyAnimation{
      target: cam
      properties: "eye"
      duration: 2000
      to: Qt.vector3d(30,5,50);
  }
```

```
    PropertyAnimation{
        target: cam
        properties: "eye"
        duration: 2000
        to: Qt.vector3d(-30,5,50);
    }
    PropertyAnimation{
        target: cam
        properties: "eye"
        duration: 1000
        to: Qt.vector3d(0,5,0);
    }
    ParallelAnimation {
        PropertyAnimation{
        target: cam
        properties: "eye"
        duration: 2000
        to: Qt.vector3d(0, 140, -1);
        }
        PropertyAnimation{
        target: cam
        properties: "center"
        running: false
        duration: 1000;
        to: Qt.vector3d(0,0,20);
        }
    }
    onRunningChanged: {
        if (running==false) {
        root.state="BossFight"
        }
    }
    }
}
```

We can also add the animation to the states:

```
//game.qml
...
     State{
         name: "BossRotation"
     PropertyChanges {target: rotateAroundBoss; running: true }
     }
...
```

12.2 Movement

For the boss enemy, we create *Enemy.qml*.

It uses the *Fruits.3ds* model, which has to first be pretransformed in order for it to fit into our scene:

```
//Enemy.qml
import QtQuick 2.0
import Qt3D 1.0
```

```
//Creates an enemy
Item3D {
id: enemy

//Size of the object for the collision detection
property real radius: 1.5

mesh: Mesh { source: "Fruits/Fruits.3ds"; options: "ForceSmooth";}

pretransform : [
    Rotation3D {
    angle: -180
    axis: Qt.vector3d(0, 1, 0)
    },
    Scale3D {
    scale: 0.01
    }
]
}
```

The enemy will be created after ten targets have been passed so we have to extend the *target-Timer* code and add the boss enemy component:

```
 //Gamelogic.qml
 ...
 property variant bossEnemyComponent: Qt.createComponent("Enemy.qml")
 ...
 Timer {
    id: targetTimer
...
onTriggered: {
    var component;
    //After a certain amount of targets were created the boss enemy appears
    if (targetCount>10) {
        targetTimer.stop()
        enemy = bossEnemyComponent.createObject(level)
    }
    //Targets are constantly created and fly towards the player
    else {
        targetCount++
        var object = targetComponent.createObject(level,
        {"position.x": (Math.random()-0.5) * 8,
        "position.y":  (Math.random()-0.5) * 6,
        "scale": 3-0.2*targetCount,
        "collisionTarget": player})
        object.collisionDetected.connect(targetCollision)
    }
}
}
...
```

Furthermore, we are adding a property called enemy* to *main.qml* to be able to easily access the object:

```
//game.qml
...
Viewport {
  ...
  property variant enemy
  ...
}
```

When the enemy is created, it will approach the player, stop at a distance of 40, and afterwards, set a new state for the root element which will trigger the camera movement.

```
//Enemy.qml
...
//Animation which moves the the enemy towards the player
NumberAnimation on z{
running: true
duration: 10000
from: 200
to: 40
onRunningChanged: { if (running == false) root.state="BossRotation" }
}
...
```

The enemy will simply move from left to right and fire in constant intervals. Both the *SequentialAnimation* and the *Timer* will only run if the root item is in the *BossFight* state.

```
//Enemy.qml
...
Item3D {
  ...
  //The enemy movement
  SequentialAnimation {
  id: bossMovement
  running: root.state=="BossFight"
  loops: Animation.Infinite
  PropertyAnimation{
      target: enemy
      properties: "x"
      duration: 5000
      to: -16
      easing.type: Easing.InOutSine
  }
  PropertyAnimation{
      target: enemy
      properties: "x"
      duration: 5000
      easing.type: Easing.InOutSine
      to: 16
  }
  }

  Timer {
  id: shootTimer
  interval: 1000
  repeat: true
  running: root.state=="BossFight"
  onTriggered: {
      shootLaser()
  }
  }
}
```

Because we use a *SequentialAnimation* here, more complex movements could be implemented (for example the enemy flying in circles or at altering speed).

12.3 Weaponfire

We use a very popular technique called Billboarding* for bullets that are fired from and at the enemy. It adjusts an item's orientation so that it always faces the camera. *Billboarding* is very often used for particle effects, (distant) vegetation or just to cut down polygons on far away 3D Objects. Usually a billboard consists of a rectangle that is always facing the camera, but any arbitrary 3D Object could be used for that.

In Qt3D there are two methods available that create billboard items. One of them is the *BillboardItem3D* which uses a very fast way for creating billboards that face the camera plane. This element however has some restrictions, whereas scaling and rotating of an item is not possible. Because of that we take the *LookAtTransform* for creating a billboard that faces the camera.

```
//Bullet.qml
Quad{
    //defines the shadereffect, that should be used for the item
    effect: lasereffect
    transform: [
        Rotation3D{
            angle: 90
            axis: Qt.vector3d(1, 0, 0)
        },
        LookAt{ subject: camPos}
    ]
    //wrapper around the camera position
    Item3D { id: camPos
    position: cam.eye
}
}
```

We are using a *Quad* for our particle effect, that has to be rotated first, because it is lying in the x,z plane. Afterwards the *LookAt* transformation is applied, which takes an *Item3D* as *subject*. That is why we have to embed the camera's position into a *Item3D* before assigning it to the *LookAt* transform. The subject is the item, that should be looked at.

For now, we will just create a simple *Effect* for each bullet, i.e. a semitransparent texture is mapped on top of the quad.

```
Effect {
    id: lasereffect
    blending: true
    material: Material {
  textureUrl: "bullet.png"
    }
}
```

We will reuse the collision detection, which we already built in the previous section, for the bullets. The difference between the bullet and the onion rings is that a bullet has a direction and a velocity that can both depend on the entity that shoots the bullet or power ups that the player has collected. We therefore have to implement a new animation that handles the movement of the bullet. Again, it is very important to only animate the position of the *BasicGameItem* and not the *Quad*. Otherwise collision detection will not work.

```
//Bullet.qml
import QtQuick 2.0
import Qt3D 1.0
import Qt3D.Shapes 1.0

//This item represents a laser particle
BasicGameItem{
id: bullet
property variant dir: Qt.vector3d(0,0,1)
property real speed: 100;
Quad{
    //defines the shadereffect, that should be used for the item
    effect: lasereffect
    transform: [
    Rotation3D{
        angle: 45
        axis: Qt.vector3d(1, 0, 0)
    },
    LookAt{ subject: Item3D { position: cam.eye} }
    ]
    Effect {
    id: lasereffect
    blending: true
    material: Material {
      textureUrl: "laser2.png"
      emittedLight: Qt.rgba(1,0.8,0.8,1)
    }
    }
}
//The movement of the bullet
PropertyAnimation on position {
    to: Qt.vector3d(x+speed*dir.x, y+speed*dir.y, z+speed*dir.z);
    duration: 10000
    onRunningChanged:  {
    //When the bulletanimation is finished and no target has been hit
    if (running==false) {
        bullet.destroy();
    }
    }
}
}
```

We have now got bullets with a working collision detection that move in a direction that can be specified. The feature that is still missing is the firing mechanism for the bullets. We need to make it possible for the player to shoot a bullet when pressing the space key, also the enemy should be able to shoot back. Furthermore, the player and the enemy need a property which holds the hit-points that are left and a function connected to the *collisionDetected* signal of the bullets so that the hit-points can be subtracted.

We first implement the latter for the enemy:

```
//Enemy.qml
property int hitpoints: 10
property real maxHitPoints: 10
....
function hit() {
    hitpoints--
    if (hitpoints <= 0) {
        explode();
    }
}

function explode () {
    enemy.destroy()
    root.state="EnterHighscore"
}
....
```

It is nearly the same for the player except that we do not delete the player after it explodes:

```
//Player.qml
property int hitpoints: 10
....
function hit() {
    hitpoints--
    if (hitpoints <= 0) {
        explode();
    }
}

function explode () {
    root.state="EnterHighscore"
}
....
```

The firing mechanism for the enemy is very simple. We just create a new *Bullet* object with the player as the target. Then we connect the *collisionDetected* signal to the player's *hit* function:

```
//Enemy.qml
...
//Shoots a bullet
function shootLaser() {
    var component = Qt.createComponent("Bullet.qml")
    var object = component.createObject(level, {"position": enemy.position,
      "radius": 0.2, "dir": Qt.vector3d(0,0,-1),
      "collisionTarget": player});
    object.collisionDetected.connect(player.hit)
    object.collisionDetected.connect(object.destroy)
}
...
```

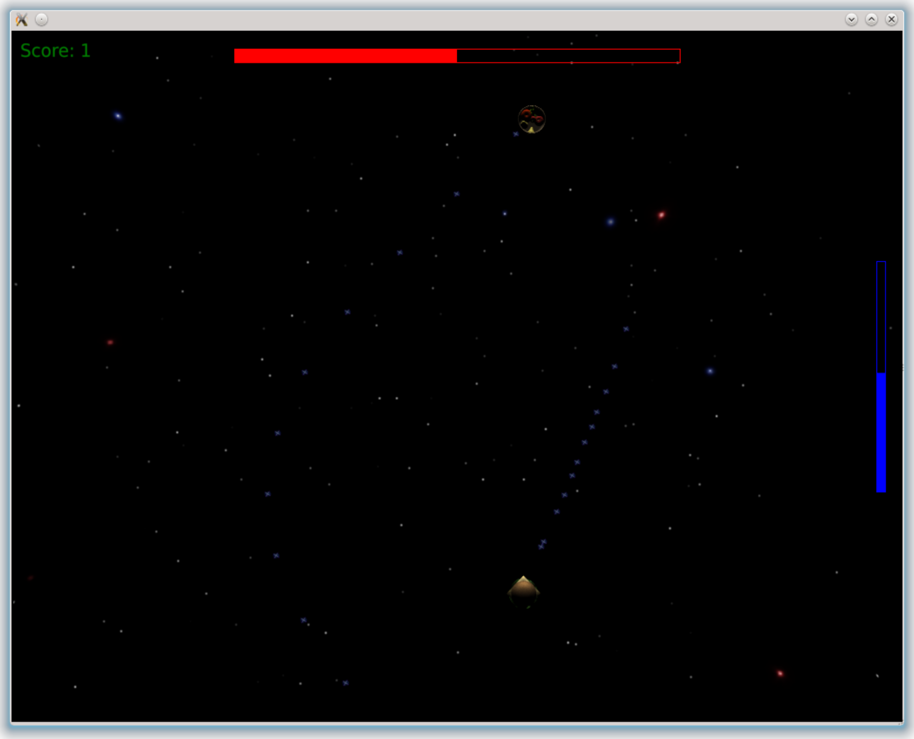
We implement the firing of the player's bullet in *Gamelogic.qml*, where the *fireLaser* function is executed after the space key has been pressed. Every time a bullet is fired, a certain amount of energy is subtracted from the player, which we refill in the *gameTimer*:

```
//Gamelogic.qml
...
id: gameTimer
onTriggered: {
if(player.energy<player.maxenergy)
    player.energy++;
...
function fireLaser() {
    if (player.energy>=40) {
        print(player.y)
        player.energy -=40
        var component = Qt.createComponent("Bullet.qml")
        var laserObject = component.createObject(level,
    {"position": player.position,
    "collisionTarget": enemy})
        laserObject.collisionDetected.connect(enemy.hit)
    }
}
...
```

The fight against the enemy should work fine now. One thing you have probably noticed is that the area in which the hamburger* can be moved is fairly small. This is because of the new perspective. That is why we have to expand the *x_bound* value during the fight against the boss enemy.

```
//game.qml
...
property real x_bound: state == "BossFight" ? 16: 4.5;
...
```

You should now be able to fight against the boss enemy.



What's Next?

Next we will talk about shaders and see how to create particle effects with it.

CHAPTER 13

Shaders

In the previous chapters, we have seen that we can define *Effects* with a texture and different material properties for our geometry in QML/3D. For most areas of applications, we do not need more than that, but if we want to achieve a custom effect, this technique has its limits. It is, however, possible in Qt Quick 2.0 and QML/3D to define custom effects as shader programs that allow you to extend the functionality of the built in effects.

The basics of shader programming are not covered in this guide. There are several tutorials out there, some of them even Qt specific. If you do not yet have any experience with shaders, we recommend that you first read the OpenGL-Tutorial[1]. In this section, we will only give you advice on how to use shaders in QML/3D and show you how to use them.

In QML/3D, shader programming is possible using the *ShaderProgram* element, which is derived from the more general *Effect* element. The *ShaderProgram* element has been extended by two properties: *fragmentShader* and *vertexShader*. Both take a string that contains the GLSL shader code.

13.1 Bulletshader

The shader for the fired bullets will mix two textures, rotate them and adjust the interpolation level over time. The result should be the impression of a rotating and blinking object.

For rotation and interpolation, we are defining two new properties in the *ShaderProgram* element. A special feature of the *ShaderProgram* is the automatic property to uniform binding. This means that if we define a uniform variable in either of the shaders (fragment or vertex), the uniform is automatically bound to the *ShaderProgram's* property when they have the same name. The following code serves as an example:

```
//Lasershader.qml
import QtQuick 2.0
import Qt3D 1.0

ShaderProgram {
```

[1] http://qt.nokia.com/learning/guides

```
    blending: true
    property real angle : 1.0
    property real interpolationFactor : 1.0
    property string texture2: "texture2.png"
    ...
    fragmentShader: "
    uniform mediump float angle;
    uniform mediump float interpolationFactor;
    uniform sampler2D texture2;
    ...
    "
}
```

What you should also notice is that you can not only bind simple integer and float variables to uniforms, but also textures and matrices. Textures are therefore defined as string properties, however, the first texture can be defined using the *Effect's texture* property and is bound to the *qt_Texture0* uniform.

First we want to define the vertex shader, because it is a fairly simple task:

```
//Lasershader.qml
ShaderProgram {
...
vertexShader: "
attribute highp vec4 qt_Vertex;
uniform mediump mat4 qt_ModelViewProjectionMatrix;

attribute highp vec4 qt_MultiTexCoord0;
varying mediump vec4 texCoord;

void main(void)
{
gl_Position = qt_ModelViewProjectionMatrix * qt_Vertex;
texCoord = qt_MultiTexCoord0;
}
"
}
```

There are the predefined attributes *qt_Vertex* (vertex position), *qt_MultiTexCoord0* (texture coordinates) of the currently processed vertex and the *qt_ModelViewProjectionMatrix*. We want to pass the texture coordinates to the next stage so we define a varying *texCoord* that we assign the texture coordinates to.

The fragment shader will be a bit more involved. There are two major tasks that we have to accomplish. Firstly, we need to rotate the texture according to the *angle* value and then we have to interpolate between the two *Sampler2Ds* that have been assigned to the shader. The rotation will be accomplished by rotating the texture coordinates with a 2D rotation matrix. Afterwards, we will be using the built in *mix* function in order to mix two color values and hand over the interpolation factor as a third parameter:

```
//Lasershader.qml
ShaderProgram {
...
fragmentShader: "
varying highp vec4 texCoord;
uniform sampler2D qt_Texture0;
uniform sampler2D texture2;
uniform mediump float angle;
uniform mediump float interpolationFactor;
uniform mediump float hallo;
void main()
{
//The rotation matrix
mat2 RotationMatrix = mat2( cos( angle ), -sin( angle ),
        sin( angle ),  cos( angle ));

vec2 textureC = RotationMatrix*(texCoord.st-vec2(0.5))+vec2(0.5);

mediump vec4 texture1Color = texture2D(qt_Texture0, textureC);
mediump vec4 texture2Color = texture2D(texture2, textureC);
mediump vec4 textureColor = mix(texture1Color, texture2Color,
      interpolationFactor);
gl_FragColor = textureColor;
}
"
}
```

Now we also want to animate the *interpolationFactor* and *angle* properties:

```
//Lasershader.qml

ShaderProgram {
  ...
  SequentialAnimation on interpolationFactor
  {
running: true; loops: Animation.Infinite
NumberAnimation {
    from: 0.3; to: 0.7;
    duration: 800
}
PauseAnimation { duration: 200 }
NumberAnimation {
    from: 0.7; to: 0.3;
    duration: 800
}
PauseAnimation { duration: 500 }
  }


  NumberAnimation on angle{
from:0
to: Math.PI
duration: 1000;
running: true; loops: Animation.Infinite;
  }
}
```

For enabling this *Effect* on our bullets, we have two options. The first option would be to directly assign the *Lasershader* to the *effect* property of the bullet, which would mean that whenever a new bullet is created, a new *ShaderProgram* is also created:

```
//Bullet.qml
...
```

```
effect: Lasershader { }
...
```

The second option would be to create it globally in *game.qml* and assign the id of the effect to the bullet's *effect* property. The latter method saves more resources, but as you might notice, the *angle* and *interpolationFactor* stay the same for all bullets that are shot, and therefore, do not look as good as in the first method:

```
//game.qml
...
Viewport {
  ...
  Lasershader {id:bulleteffect}
}

//Bullet.qml
...
Quad {
  effect: bulleteffect
...
```

13.2 Explosion

There are many ways to create explosions. Most of them, however, are quite difficult to implement. Our approach will be a very simple one, but quite aesthetic and realistic looking. We use the Billboarding* technique again and combine it with an animation. When an object explodes, one or more quads are created on which an explosion is shown that has been created before with a special program for example. In this context, *Animated* means that several pictures of an explosion are shown after each other (the same concept, as when watching a movie).

For a good explosion animation, we need at least 10 to 16 pictures to shown one after the other. We can, however, not include them separately in the vertex shader because we only have a certain amount of texture slots available on the graphic card. That is why we merge all explosion frames together into one big texture. This texture will be uploaded to the GPU and the fragment shader chooses which parts of the texture to use according to a time value. But first of all we create a new file called *Explosion.qml*. This will contain one *BillboardItem3D* that uses a quad as a mesh:

```
//Explosion.qml
import QtQuick 2.0
import Qt3D 1.0
import Qt3D.Shapes 1.0

Quad{
    id: explosionItem
    scale:5
    transform: [
    Rotation3D{
        angle: 90
        axis: Qt.vector3d(1, 0, 0)
    },
        LookAt{ subject: camPos}
    ]
    //wrapper around the camera position
    Item3D { id: camPos
    position: cam.eye
    }
}
```

As already mentioned, we need a *lifetime* property for our explosion that has to be available in the fragment shader:

```
//Explosion.qml
...
Quad{
  ...
  NumberAnimation
  {
  running:true
  target: program
  property: "lifetime"
  from: 0.0
  to: 1.0;
  duration: 1000
  onRunningChanged: {
      if(running==false)
      explosionItem.enabled= false;
  }
  }
}
```

The *ShaderProgram* consists of the *lifetime* property used above, the *explo.png* texture, which has 16 explosion frames, a vertex and a fragment shader:

```
//Explosion.qml
...
Quad{
  ...
  effect: program
  ShaderProgram {
  id: program
  texture: "explo.png"
  property real lifetime : 1.0
  blending: true
  vertexShader: "
  attribute highp vec4 qt_Vertex;
  uniform mediump mat4 qt_ModelViewProjectionMatrix;

  attribute highp vec4 qt_MultiTexCoord0;
  uniform mediump float textureOffsetX;
  varying mediump vec4 texCoord;

  void main(void)
  {
  gl_Position = qt_ModelViewProjectionMatrix * qt_Vertex;
  texCoord.st = qt_MultiTexCoord0.st;
  }
  "
```

```
    ...
  }
}
```

The vertex shader is not really exciting because it just computes the position of the vertex and passes on the texture coordinates. The fragment shader, however, looks a bit more involved. We first multiply the *lifetime* by the number of frames we have in our texture and then try to find out which row and column position is the closest to our current *lifetime* value:

```
//Explosion.qml
...
ShaderProgram{
  ...
  fragmentShader: "
  varying highp vec4 texCoord;
  uniform sampler2D qt_Texture0;
  uniform mediump float lifetime;

  void main(void)
  {
  mediump int life = int(lifetime    16.0);
  mediump int row = life % 4;
  mediump int column = life / 4;
  mediump vec4 textureColor = texture2D(qt_Texture0,
    vec2(texCoord.s/4.0+0.25*float(row) ,
    1.0-texCoord.t/4.0-0.25*float(column)));
  gl_FragColor = textureColor;
  }
  "
}
```

Suitable animated explosions can be found everywhere on the internet. There is also software that can produce these textures from scratch. This technique is not only limited to displaying explosions. Thunderbolts and fire can also be animated.

The last thing needed for our explosion to work is the integration into our game. There are several ways of doing this. Either we define a global explosion which can be moved to the position of the exploding object or we implement the explosion in the objects. We now create a completely new component that can handle all possible explosions. For that the new component *ExplosionSystem.qml* is created:

```
//ExplosionSystem.qml
import QtQuick 2.0

Timer {
    id: explosion
    running: true
    property int loops: 40
    property variant position: Qt.vector3d(0,0,0)
    property variant explosionComponent: Qt.createComponent("Explosion.qml")
    property real variation: 3

    signal finished()

    interval: 200
    repeat: true
    onTriggered: {
    loops--
    var object = explosionComponent.createObject(level,
      {"x": position.x+(Math.random()-0.5) * variation,
      "y": position.y+(Math.random()-0.5) * variation,
      "z": position.z+(Math.random()-0.5) * variation})
    if (loops==0) {
        finished()
        explosion.destroy()
    }
    }
}
```

Note: Because we destroy the object after the *loops* property reaches *0*, the *ExplosionSystem* component may only be created dynamically with the *createObject* function.

The *ExplosionSystem* is created if the player has no hitpoints anymore:

```
//Player.qml
...
Item3D {
...
function explode () {
    root.state="EnterHighscore"
    var component = Qt.createComponent("ExplosionSystem.qml")
    var object = component.createObject(level, {"position": position})
    object.finished.connect(enemy.exploded)
}
}
```

The same applies for the enemey:

```
//Enemy.qml
...
Item3D {
...
function explode () {
    root.score+=20
    var component = Qt.createComponent("ExplosionSystem.qml")
    var object = component.createObject(level, {"position": position})
    object.finished.connect(enemy.exploded)
    shootTimer.running=false
    bossMovement.running=false
    root.state="EnterHighscore"
}
}
```

What’s Next?

For this tutorial, this will be the final version of the game. Next we will however talk about how we can extend and improve it and give some ideas and instructions for further enhancement.

CHAPTER 14

Finalizing the game

Although the game is already playable, a few things are obviously still missing to bring it up to a round figure. We have to extend the current game for that - add new levels and enemies and make small adjustments to enhance game play.

More levels can easily be added. We need, however, a level counter and to exend the *GameLogic.qml*. After the *BossFight*, the highscore dialog will be shown but the *Game* state will be revoked again. In further levels, the targets could for example change into something else and not fly straight towards the player but maybe move on the way on the y- or x-axis. This makes it more difficult for the player to hit them.

Furthermore, small power-ups could be collected that could improve the hitpoints, energy level or enhance maneuverability. In future levels, there could also be targets that have to be avoided because they might cause damage to the player or decrease abilities.

As mentioned before, the boss enemy could also be greatly improved by giving it a different moving pattern and other weapons. It should be able to move back and forth, avoid the players weaponfire and perform some unexpected movements.

All of this can be accomplished by just extending the current structure.