Developing 2D games in a declarative way

Sérgio Correia, Rodrigo Gonçalves de Oliveira, Roger Zanoni
{sergio.correia, rodrigo.goncalves, roger.zanoni}@openbossa.org
Nokia Institute of Technology (INdT)
Av. Torquato Tapajós, 7200, 69093-415, Manaus/AM – Brazil

Abstract

This work presents an open source game framework that provides ready-to-use QtQuick elements representing basic game entities needed by most of games, such as layers, sound, physics, viewports and sprites. It is named Quasi-Engine and it eases the task of writing 2D games, as we are now able to write such games declaratively. Quasi-Engine is also tightly coupled with the popular Box2D physics library, and as such, making use of its simulation capabilities is also provided, thus allowing the games to be more realistic.

Keywords:: Game, engine, 2D, declarative language

1 Introduction

Quasi-Engine is a framework that intends to be a complete toolset to ease 2D game development, providing ready-to-use QML (QtQuick, a declarative programming language, such as Prolog and SQL) elements representing basic game entities needed by most games. This framework is primarily built upon the upcoming major version of Qt framework (Qt5) [Digia 2012], which will ship with many improvements on the graphical and declarative modules, especially the new SceneGraph engine. Since the absolute majority of Qt-based platforms run Qt4, we also maintain a port for this version.

This project was born from our desire to implement a simple 2D game using the QML language. Although its core components were enough to put together most of our ideas, a significant effort would be needed to develop more complex elements and concepts, things that could be potentially reused by other similar games.

Since there were no public engines targeted to Qt/QML developers to date, we decided to roll out our own set of components to aid development of common aspects of 2D games, and named it Quasi-Engine. This paper focuses on the core framework and its application in a real world use case. Among its current features, there are:

- Physics support through the Box2D library.
- Static and animated layers, including parallax scrolling.
- Support to state-based sprite animations.

The project roadmap also includes support for background music, audio effects, networking and cutscenes.

2 Related Work

There are some declarative programming languages which provide game development support, such as Inform 7 and SuperGlue. Each one has its own peculiarities and they are listed below:

- Inform [Graham Nelson 2006] is a design system for interactive fiction based on natural language, and its newer version, Inform 7, focuses on declarative programming. It's used to develop text based games, letting the player explore worlds, stories, historic simulations and experimental digital art. Inform 7 differs substantially from QML on its source code, which reads like English sentences, as stated in the following example:

```
"Hello World" by "I.F. Author"

The world is a room.

When play begins, say "Hello, world."
```

- SuperGlue is a "textual live language", based on reactive values known as signals, that are supported with declarative dataflow connections and dynamic inheritance [McDirmid 2007; McDirmid and Hsieh 2006]. The concept of this programming language is to simplify component assembly by hiding event handling details from glue code. A simple example is shown below:

```
class GhostShape ▷ Shape {
  port gs ▷ Shape;
  gs = (Pie + Rectangle.scale(height = 0.495).
       translate(y = 25) −
    Eye.translate(x = 10) − Eye.translate(x = 30));
  this.shape = gs.shape;
  class Eye ▷ Shape;
  Eye.shape = Pie.scale(width = 0.2, height = 0.2).
       translate(y = 10).shape;
}
```

As shown above, there are just a few game engines/frameworks that can be used in declarative game development. Unfortunately, these libraries are also very specific on their usage, not being useful or adequate for developing different kinds of game styles, limiting the potential of their language.

For a general context, there are some notorious game engines, written in many different programming languages, such as C, C++, C#, Java, Objective C, Python and a plenty of others. We list here some usual game engines:

- Blender [Blender Foundation 2012]
- CryEngine [Crytek 2012]
- GameMaker [YoYo Games 2012]
- id Tech [id Software 2012]
- RAGE [Rockstar Games 2012]
- Unity3D [Unity Technologies 2012]
- Unreal Engine [Epic Games 2012]

3 The Quasi Framework

The Quasi-Engine framework is built as a set of elements that handle basic game items. These items are best described below:

3.1 Architecture Overview

As the QML engine loads Qt plugins, there is a QML plugin interface that provides types and functionalities for the use in C++ (through Qt), which we used to write this framework.

The entire codebase is developed in the C++ programming language (as image manipulation methods, audio handling routines and the Box2D physics library access), so these defined functions and types are exposed to QML, being wrapped on a plugin file that can be dynamically loaded later into the game.

Quasi-Engine can be virtually ported to any platform that supports the Qt application framework and its declarative module, QML.

The game loop (also known as the update-render loop), an essential component of any game or game engine, has the following general structure (shown in pseudocode)

```
function gameLoop() {
    update()
    render()
}
```

The *update* call is responsible for updating the state of the game entities, and the *render* call is responsible for rendering such game entities, making use of their state information.

Quasi-Engine does not have a game loop with this classic structure, however the same basic logics are followed to update and render its entities. The *Game* element (more on Quasi elements in the next subsection) contains properties to indicate how many times per second the call to update must be performed, and it is responsible for notifying the current *Scene* element that its entities must be updated. The *Scene* element, on the other hand, execute the update call for each of its active entities, which in turn update their own state.

Quasi does not manage the rendering process, delegating this task to the Qt Framework; the engine worries only with the management of its own entities.

3.2 Elements

Quasi-Engine provides the basic elements needed to represent a game, and we are now going to describe them:

- the **Game** element represents a game in its entirety. Its role is to inform the current scene that its elements should be updated.

```
QuasiGame {
  id: game

  width: 400
  height: 250

  onUpdate: console.log("update", delta)

  Component.onCompleted: {
    console.log("fps", game.fps)
  }
}
```

- the **Entity** element is Quasi-Engine's basic entity. It has all the basic properties a QML item does, such as position, rotation, visibility and opacity, for instance, and may contain another Entity element, as well as any other QML element. In Quasi, every element that may be used in QML inherits from the Entity element, and they also have a routine – *updateScript* – which is called whenever an update is needed.

- a **Scene** is a logical subdivision of a game, and each of them contains a group of Entity elements. For instance, each game level could be a scene. Also, each of the configuration screens could be a scene. When a game requests the scene to update its elements, the update routine is called separately for each of them.

```
QuasiGame {
  id: game

  width: 400
  height: 250

  currentScene: scene

  QuasiScene {
    id: scene

    entities: QuasiEntity {
      updateScript: {
        console.log("update")
      }
    }
  }
}
```

- **Viewport** is a rectangular area on which is projected part of a scene. This exhibited part may vary, creating the impression of movement, and if one changes the scale of the projected area, the impression of approximation is created.

- a **Layer** in Quasi is an area that contains the items in the scenario which are to be displayed. The layers have a presentation order that determines which scenario elements superimpose the remaining ones. There are a few different layer types, described below:

  – a **StaticLayer** exhibits one or more layers with static images. It is a kind of layer very useful in the creation of platform games, requiring the elements to have different presentation ordering.

```
QuasiLayers {
  anchors.fill: parent
  drawType: Quasi.PlaneDrawType

  layers: [
    QuasiStaticLayer {
      id: layer

      source: "image.jpg"
      order: Quasi.BackgroundLayerOrdering_01
    }
  ]
}
```

  – an **AnimatedLayer** has the same properties a StaticLayer does, such as the level control, and the property of displaying the movement of the layers' images indefinitely. This effect can be used for creating side-scrolling games, very common in touchscreen cell phones lacking a physical keyboard.

```
QuasiLayers {
  anchors.fill: parent
  drawType: Quasi.TiledDrawType
  tileWidth: 32
  tileHeight: 32

  layers: [
    QuasiAnimatedLayer {
      source: "image.png"
      order: Quasi.BackgroundLayerOrdering_01

      horizontalStep: 5
      direction: Quasi.BackwardDirection
      type: Quasi.InfiniteType
    }
  ]
}
```

  Parallax is the difference in the apparent positioning of objects, according to the change in the observers' position. Quasi is able to produce this effect by using the AnimatedLayer element.

```
QuasiLayers {
  anchors.fill: parent
  drawType: Qiasi.TiledDrawType
  tileWidth: 40
  tileHeight: 40

  layers: [
    QuasiAnimatedLayer {
      source: "images/space.png"
      factor: 0.3
      order: Quasi.BackgroundLayerOrdering_01

      horizontalStep: 1
      type: Quasi.MirroredType
    },
    QuasiAnimatedLayer {
      source: "images/planet.png"
      factor: 0.5
      order: Quasi.BackgroundLayerOrdering_02

      horizontalStep: 1
      type: Quasi.InfiniteType
    },
    QuasiAnimatedLayer {
      /* ... */
    }
  ]
}
```

3.3 Physics

Applying the physics concepts and laws to games is done with the intention of making the effects appear more real to the observer. Quasi-Engine uses *Box2D* [Catto 2009], an open-source C++ engine for simulating rigid bodies in 2D. Through *Box2D*, Quasi provides the following elements:

- in physics, a body is a set of masses and some other properties, such as position and rotation, for instance. In the *Box2D* library, a **Body** is an element that represents a body, containing information such as a set of masses (Shapes and Fixtures, explained below), position, current angle and it also provides an interface for applying a force and rotation.
- a **Fixture** associates a Body to a Shape, and it contains useful information for the simulation, such as density, and coeficients of friction and restitution.
- a **Shape** is a geometric shape attached to a Body. They respond to collisions and are used for calculating the mass of the Body element.

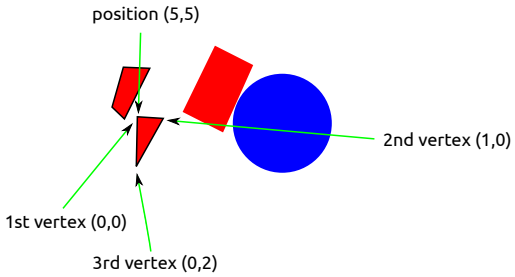

Figure 1: *Examples of shapes*

- in *Box2D*, a **Chain** is a shape used to define linked line segments that may or may not form a polygon. They are useful, for instance, to create the representation of the environment terrain in a scene. In Quasi, the use of these elements is simplified, reunited in a single element named **QuasiBody**.
- *Box2D* **joints** unite the elements and change their behavior in the simulation. There are several joint types in *Box2D*, but currently Quasi supports only the distance and mouse joints.
  - **QuasiDistanceJoint** is a joint that unite two bodies in determined points and keeps fixed this distance between them.
  - **QuasiMouseJoint** is a joint that links a body and a point and is always in the same position as the mouse, causing the linked body to try to follow the current cursor position.

4 Sample Demos

As new features are implemented in Quasi, new examples and demos are developed to show off these features. We list below some examples and demos available in the framework.

- *basketball*: Figure 2 shows the use of a force applied to a ball, simulating a simple basketball game.

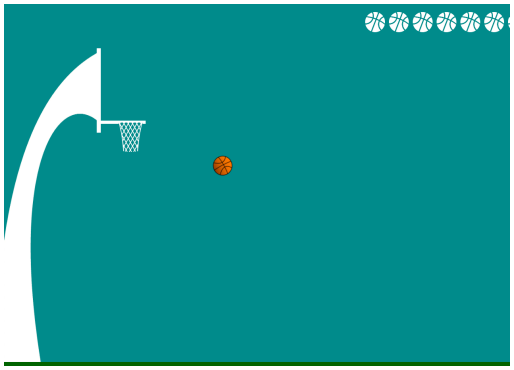

Figure 2: *Basketball game demo*

- *paratrooper*: Figure 3 shows the use of a force too, applied to a human being trying to land on the target surface, similar to the classic Atari game *Lunar Lander* [Atari, Inc. 1979].

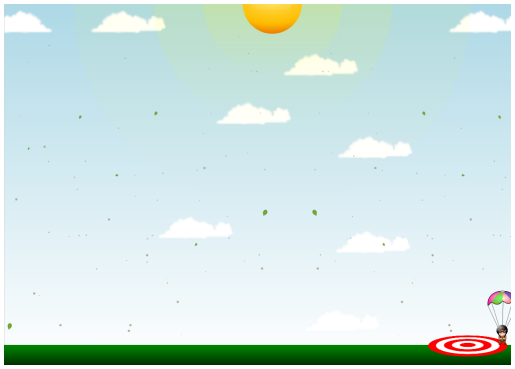

Figure 3: *Paratrooper game demo*

- *parallax*: Figure 4 shows off the use of multiple layers, performing a parallax visual effect of a spaceship through the space.

Figure 4: *Parallax layers example*

- *viewport parallax*: Figure 5 shows the use of two layers, forming the parallax effect. This example uses the concept of a viewport, where the developer sets a fixed rectangular area for displaying the game scene. When moving to its borders (left and right), the viewport automatically scrolls the layers, moving the scene until the character reaches the viewport limits.

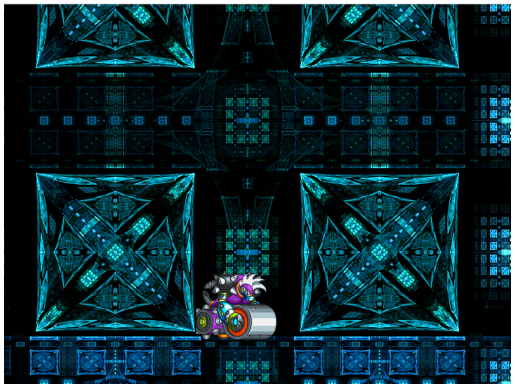

Figure 5: *Viewport parallax example*

5 Conclusion

This paper presents the base elements of a game framework that still is in heavy development, but already has some game elements ready to use. The framework itself was developed using the C++ programming language on top of the popular user interface framework Qt. This framework has a declarative module named QtQuick,

also known as QML, which is the programming language the final user will develop in. Quasi also supports physics simulation, provided by the Box2D library, merged on its core C++ counterpart.

From the project roadmap, audio support (background music and audio effects) is being currently addressed and should be part of the first stable release of Quasi. It is worth mentioning Quasi-Engine is open-source and publicly available at [INdT – Nokia Institute of Technology 2012] for people to download and contribute to, if so they wish.

References

ATARI, INC., 1979. Lunar Lander. http://www.atari.com/lunarlander.

BLENDER FOUNDATION, 2012. Blender. http://blender.org/.

CATTO, E., 2009. Box2D Physics Engine. https://code.google.com/p/box2d.

CRYTEK, 2012. CryENGINE. http://mycryengine.com/.

DIGIA, 2012. The Qt Framework. http://qt.digia.com/.

EPIC GAMES, 2012. Unreal Engine. http://www.unrealengine.com/.

GRAHAM NELSON, 2006. Inform 7. http://inform7.com/.

ID SOFTWARE, 2012. id Tech. http://www.idsoftware.com/.

INDT – NOKIA INSTITUTE OF TECHNOLOGY, 2012. Quasi-Engine on GitHub. https://github.com/INdT/Quasi-Engine.

MCDIRMID, S., AND HSIEH, W. C. 2006. Superglue: component programming with object-oriented signals. In *Proceedings of the 20th European conference on Object-Oriented Programming*, Springer-Verlag, Berlin, Heidelberg, ECOOP'06, 206–229.

MCDIRMID, S. 2007. Living it up with a live programming language. *SIGPLAN Not. 42*, 10 (Oct.), 623–638.

ROCKSTAR GAMES, 2012. RAGE. http://www.rockstargames.com/.

UNITY TECHNOLOGIES, 2012. Unity 3D. www.unity3d.com.

YOYO GAMES, 2012. GameMaker. http://www.yoyogames.com/gamemaker/studio.