Gamification-Based E-Learning Strategies for Computer Programming Education

Ricardo Alexandre Peixoto de Queirós
Polytechnic Institute of Porto, Portugal

Mário Teixeira Pinto
Polytechnic Institute of Porto, Portugal

A volume in the Advances in Game-Based Learning (AGBL) Book Series

IGI GLOBAL
DISSEMINATOR OF KNOWLEDGE

www.igi-global.com

Published in the United States of America by
IGI Global
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Names: Queiros, Ricardo Alexandre Peixoto de, 1975- | Pinto, Mario Teixeira, 1967- author.
Title: Gamification-based e-learning strategies for computer programming education / Ricardo Alexandre Peixoto de Queiros and Mario Teixeira Pinto, editors.
Description: Hershey PA : Information Science Reference (an imprint of IGI Global), 2016. | Series: Advances in game-based learning | Includes bibliographical references and index.
Identifiers: LCCN 2016032941| ISBN 9781522510345 (hardcover) | ISBN 9781522510352 (ebook)
Subjects: LCSH: Simulation games in education. | Computer programming--Study and teaching. | Computer games.
Classification: LCC LB1029.S53 Q45 20116 | DDC 371.39/7--dc23 LC record available at https://lccn.loc.gov/2016032941

This book is published in the IGI Global book series Advances in Game-Based Learning (AGBL) (ISSN: 2327-1825; eISSN: 2327-1833)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 10
Using Game Frameworks to Teach Computer Programming

Alberto Simões
Instituto Politécnico do Cávado e do Ave, Portugal

ABSTRACT

Teaching computer programming is an important task in the formation of computer scientists. Being a subject taught in the first years of student degrees, need to properly motivate students, so they try, at home, to learn by themselves, complementing that way their classes. This chapter proposes an approach to computer programming teaching based on the construction of videogames, using state of the art game frameworks. The author will show how the task of writing a game using a common framework deals with the basic programming concepts that are usually taught on a first course on computer programming, namely on object oriented programming languages like C# or Java: algebraic operations with variables, methods declaration, objects definition, objects hierarchy and multidimensional arrays. As it will be shown, even the common order of concepts presentation during the course can be kept, although applying them to computer games instead of the usually requested exercises.

INTRODUCTION

Teaching computer programming is a complex task. Not just because the need of complex reasoning needed for a student to learn how to develop an algorithm and code it, but also because of the lack of catchy feedback.

We all know that currently teenagers are in the smartphone era, used to touch in an icon and see a nice application appear. Later, in the computer programming classes, they are asked to implement an algorithm to sort numbers or strings, and print them in the console. This makes the student struggle, on whether programming is too hard, or the teacher too lame.

More recently, teachers use visual tools, like Forms. Nevertheless, we are still speaking on standard grey windows, feeling like accounting applications.

DOI: 10.4018/978-1-5225-1034-5.ch010

Trying to tackle this problem, some teachers already try to ask students to perform more interesting tasks, like the generation of images, web pages and, a minority, computer games. I am involved in teaching at two different universities in Portugal, and in both there are approaches related to programming games.

For example, in the last years, at University of Minho (Braga, Portugal), students are asked to implement a game in their functional course, when learning Haskell. The projects were first based in an implementation of Carcassonne (Schend, 2007), using mostly the console; in the next year the project was slightly different, with the implementation of LightBot (Yaroslavski, 2014) game, simulating it using X3Dom (Behr, Eschler, Jung, & Zöllner, 2009); and this year, the project was the implementation of the Sokoban (Falcon Corporation, 2016) game, using the Gloss Haskell library (Lippmeier, 2010) for simple game development. Although these projects (mostly the latest two) resulted in some interesting projects, by better students, the whole class was dedicated to implement simple functions to process strings (filters, from a board and a move, generating a new board) and only in the final assignment students really see something moving. Therefore, when those better students get to the final assignment, their motivation raises exponentially, but those we have difficulties never get to see anything fancy, and get demotivated. I still defend this approach to be better than in previous years, but it is not the best (but probably the possible when learning a language like Haskell).

In the other hand, at Instituto Politécnico do Cávado e do Ave (Barcelos, Portugal), there is a grade dedicated to the development of computer games. When teaching to those students, that are highly motivated for computer games development, I used some game programming frameworks, and it seems that their use might be a good approach to teach object oriented programming basics to first years students. And this statement is not based only on the eye candy feedback students receive, but because it is easy to focus on standard learning concepts when developing a computer game.

In this chapter I will focus on my point of view on how teaching object oriented programming can be done resourcing to a game-programming framework. To illustrate how I see it, I will use two different programming languages (although based on a same paradigm and a virtual machine): C# and Java. And for each one I chose a game development framework: the MonoGame Framework (Chamillard, 2015) (the open-source fork of XNA (Sung, 2013)) for the C# language, and LibGDX Framework (Cook, 2015) for the Java language. This way I will be able to demonstrate how basic OO programming concepts can be taught using different languages, and different platforms: from Windows native applications to Android applications.

The remaining of this chapter will be divided in the following sections:

- **Game Frameworks:** In this section I expect to answer a set of questions regarding technology. First, make a distinction between game engines and game-programming frameworks, explaining why the second are best suited to teaching programming basics. Then, I will list a set of game frameworks that are being used nowadays, and focus on their similarities and differences.
- **The Game's Main Loop:** One of the big advantages on using a game engine is that it hides the main program structure. There is no need to write a main method, explain why it should or not be static (depending on the language of your choice) or even discussing if it is an object or not. Also, the basic *'out of the box'* game already runs!
- **First Exercises:** Usually a programming language is taught starting with the concept of variables and doing some arithmetic and conditional expressions for different simple tasks. This kind of approach is also possible when writing game. Just add a sprite to the empty game and then make it move randomly, or in specific directions. Later, for conditional expressions, keyboard events

can be processed, and coordinates checked to guarantee the sprite does not goes out of the game window.

- **Classes and Hierarchy:** To illustrate the concept of classes or objects, the sprite can be converted into the player object, and it can be replicated to give life to enemies. All these share a lot of information, from their position, their bitmap, their possible animation and orientation. Adding more objects is also easy. A sprite object can also represent a wall or a bullet. They can be, then, analyzed, and similarities detected, creating an object structure that distinguishes from the movable and non-movable sprites, the characters, and the projectiles.
- **Arrays and Matrices**: The next natural step on teaching programming basics is the introduction to arrays and matrices. Although these languages include more interesting data structures, like lists or dictionaries, arrays and matrices are always a key component to tech. Fortunately, most common games out there are based on grids, or matrices. For this, the game world should be represented by a matrix filled with walls and paths. Then, an array can be created to define enemies' positions. To manage these structures loops can be introduced, and given the bidimensionality of matrices, nested loops will be heavily used. Later, to compute the game logic some other algorithms that deal with matrices are needed, like checking for a cell neighbors, or to compute line. And if you wish, graphs can also be introduced for path checking.
- **Closing Remarks:** With this chapter I do not intend to write a course on how to use any of these frameworks, or to give any guideline to follow. In fact, although its contents are technical, it is more like an opinion than a scientific document. I expect that with this brainstorming of ideas I can convince other lecturers to use this approach to teach OO programming, being to standard computer science students or computer game development students. In this final section I will discuss possible directions from the proposed approach, and give some insights on how my students behave.

GAME FRAMEWORKS AND GAME ENGINES

This chapter main contribution is an overview on how Game Frameworks can be used to teach basic computer programming skills. In order to contextualize the remaining text, and make the read easier for those who are less aware of the game development world, this section focuses on defining what are game frameworks and distinguish them from standard libraries and computer game engines. At the end, a brief list of game frameworks is presented.

Game Frameworks

In the last years there was an interesting evolution both on the resources available for computer games development, but also, on how these resources are named. Note that there is not any standardization on the used names, and different authors might see this panoply of resources from other perspectives.

For years, applications were developed using one or more programming languages and, when in the need of some specific functionality, one or more external libraries. These libraries range from mathematical or scientific libraries, file formats readers and writers, image processing tools, machine learning algorithms, computer graphics, and a wide variety of other tasks.

The same happened in the development of Computer Games. For some years we can find libraries to draw graphics, reproduce sound, simulate physics, perform networking operations, and so on. Nevertheless, game programming required all these aspects, and therefore programmers needed to choose one library for each one of these tasks (graphics, sound, artificial intelligence, physics), and then make them fit and work together.

This need lead to the advent of game frameworks: a set of libraries chosen and configured to work together out of the box. Thus, a game framework includes, at least, a library for graphics rendering, and a set of other (probably smaller) libraries for some other tasks. Some of these frameworks are very complete, including libraries for sound, reproducing, physics simulation, artificial intelligence or networking. Some others are somewhat more limited, including only the graphics rendering library and a couple of auxiliary functions/modules for other tasks.

More recently, game frameworks include even more interesting features. Some are developed using a common backend, making games available for different platforms and operating systems. Some other already include auxiliary tools, either to bootstrap the game code architecture, or as a game world editor.

Game Engines

There is another trending type of tool around on game development: Game Engines. A game engine is, basically, a generic game, that can be configured for different similar games. Or, at least, this was its original definition. Probably the best examples of game engines are the Doom and Quake engines (ID Tech engines), by ID Software (Griliopoulos, 2011). Given the success of these games, their engines were sold to other companies that created their own titles, like Hexen or Half Life. Another similar example is the Cry Engine, the result of the Crysis game series (Farokhmanesh, 2014).

This abstraction on game development can be pushed to the limit, making the used engine more and more generic. As an example, we have Unreal Engine (Tach, 2014), developed initially for the Unreal game. But the direction is not necessarily from a concrete game to an abstract engine. Other companies started developing their own engines, from scratch, and without any game in mind, starting from something that could be a framework, and enriching it with extra features, to make game development easy. An example is the Unity game engine (Lee, 2014). This makes the distinction harder, and depending on the author, the division line can be drawn at different places. But game engines, as I understand them, are frameworks that include a complete integrated development environment (IDE) prepared to write games using the resources available in that framework (and only those), together with a world editor, to allow the graphic deployment of graphic objects in the game arena.

Game Engines vs. Frameworks on Computer Teaching

It is relevant to teach both the usage of game engines and of game frameworks to students, namely for those studying computer graphics and computer game development. But, from my point of view, game frameworks can be very interesting to teach generic object oriented computer programming concepts, while allowing the student to apply them in games, making the learning process, itself, enjoyable. Almost as a game!

In the other hand, game engines are not very suitable for computer programming teaching given they have a lot of focus on design and 2D or 3D model construction, as well as a very disperse way to include the game logic, making it more confusing for students still learning how to program. Also, the way concepts are used is not so transversal to out-of-games world. On the contrary, and how it will be shown, game frameworks include most of the main concepts used in "standard" programming tasks.

Game Frameworks for Common Programming Languages

I do not intend to include here an exhaustive list of game frameworks, but to help the reader to distinguish between game engines and frameworks as, for example, Wikipedia lists them all together without any distinction.

For C# the widest used framework is MonoGame (http://monogame.net/), an open source version of the old XNA library from Microsoft. There are some other smaller frameworks. OpenTK (http://www.opentk.com/) is an abstraction level over OpenGL graphics library, OpenAL audio library and OpenCL virtual engine. SlimDX (https://slimdx.org/) is a very thin wrapper around DirectX making it easier to write game. Duality (http://duality.adamslair.net/) is a 2D game framework that uses OpenGL and includes audio and physics functionalities. It even includes an editor inspired in Unity, that approaches this framework to a minimalist game engine.

For Java, and namely for portable devices, there are some interesting game frameworks. One of the most used is LibGDX () that includes OpenGL/ES support, ships with Box2D physics engine, and has easy integration with external physics libraries, like Bullet. It includes some interesting features, supporting both 2D and 3D games. AndEngine () was quite active in the last years, and is a complete framework for 2D game development. Other, yet smaller, framework, is LWJGL (), intended to be a lightweight framework to write Java games with OpenGL.

Although there are frameworks for other languages, like Python, C or C++, I will not list or refer to them. The main reasons for that are because they do not support natively portable devices, and because some of them are not completely OO, especially for teaching purposes.

In the remaining of this chapter, my examples will be illustrated using MonoGame and LibGDX. The choice was based on my previous experience, and the fact that they have different global approaches that, as we will see, can be matched and similarities detected.

THE GAME MAIN LOOP

Game frameworks usually include, either explicitly or implicitly, the game main loop. This loop hides the process of creating the application environment, and running the user code, calling some specific callbacks (user defined methods) for some events. Which events are available depends on the framework, but some of them are available on most of them: an initialization function, where resources can be loaded and variables initialized (instead of using the constructor of the class); an update function, where the game state should be updated, and the game logic should be included; and a draw function, responsible for drawing the screen for every frame.

This skeleton code is usually generated in some way by the framework. For MonoGame there are a couple of templates, for different platforms. For LibGDX, the Gradle download includes a tool to create an application skeleton. Two classes usually constitute these skeletons: a static one, which the programmer can safely ignore and that creates an instance of the other class, the game class, where the real implementation takes place.

The game class usually hides the game main loop: cycle updating the game state and drawing each frame, taking care of exiting when the player requests. To have this main look hidden has some advantages when teaching computer programming:

- There is no need to create the *main* function, or *main* class. This is especially useful when teaching object oriented programming. Teachers usually have a hard time to explain the existence of a *main* class, as it is not a standard class, representing an object, and does not have any instances, as it is a *static* class. Explaining the benefits of OO programming by creating something that does not fit the paradigm is counterproductive. The framework, itself, creates the *main* class in a separate file, where the student does not need to look (at least in the first weeks of classes). This *main* class will use a *game* class. But this one is a standard class, although with one instance only. But as students are developing one game, the class can be seen as an instance of the game. And as the main class does its instantiation, it does not need to be declared as *static*. Thus, variables are created as standard class variables, and can be used directly. In fact, most of these details can be hidden during the first weeks, in order to introduce only imperative programming constructions.
- The frameworks usually hide the *game* class constructor from the user – MonoGame does not hide it by default, although it is possible. This is great if the teacher does not want to start digging objects right from the first class. But as variables need to be initialized, the frameworks usually expose an initialization method.
- The existence of separate methods to update the game state and to draw it can be used to motivate the students to the need of an abstract model of the game (a data structure) that is managed during the update function, and traversed during the draw phase. Yet again, this is a big improvement over the usual way to teach imperative programming, where students are invited to write code to read variables from keyboard, process them, and print results, all in a single *main* function. Unfortunately some of the frameworks (like LibGDX) just include a draw method, and no update. In any case, if the teacher desires, he can call two methods there, one for the state update code, and another for the state drawing.
- Depending on the framework, these methods can hide the requirement on calling its parent method. For MonoGame, most methods should end by calling its counterpart in the parent class, which does not help to hide the OO details. For LibGDX, the methods can be written completely without any explanation on objects and hierarchy, as they are not required to call their parent class counterpart methods.

Table 1 summarizes the main methods offered by MonoGame and LibGDX for the user to subclass and implement. Note that we only list methods that are created automatically by the framework tools or templates, as there are other less used methods that can be sub-classed by the user if needed.

To conclude this section, the skeleton of the game class as created by the MonoGame templates is presented below. The LibGDX skeleton is very similar.

Table 1. Game class methods explicitly offered by each game framework

MonoGame	LibGDX	Description
Initialize()	create()	Users should not replace the class constructor but instead, to use this method to initialize variables.
LoadContent()		Used to load content, as pictures or sounds, when game starts.
UnloadContent()		Used to dispose loaded content then the game exits.
Update(GameTime)		Called at each frame. Should be used for the logic game code.
Draw(GameTime)	render()	Called at each frame to draw the game. Some frameworks require it to be used only for rendering where others allow code for the game logic.
	pause()	For mobile devices, called when app loses focus.
	resume()	For mobile devices, called when app is reactivated.
	resize(int, int)	Called if game area changed its size.
	dispose()	Dispose resources and exit.

```
public class Game1: Game
{
     GraphicsDeviceManager graphics;
     SpriteBatch spriteBatch;
     public Game1 ()
     {
          graphics = new GraphicsDeviceManager (this);
          Content.RootDirectory = "Content";
          graphics.IsFullScreen = true;
     }
     protected override void Initialize ()
     {
          // TODO: Add your initialization code here
          base.Initialize ();
     }
     protected override void LoadContent ()
     {
          spriteBatch = new SpriteBatch (GraphicsDevice);
          // TODO: Add your update logic here
     }
     protected override void Update (GameTime gameTime)
     {
          if (Keyboard.GetState ().IsKeyDown (Keys.Escape))
               Exit ();
          // TODO: Add your update logic here
          base.Update (gameTime);
     }
     protected override void Draw (GameTime gameTime)
```

```
    {
        graphics.GraphicsDevice.Clear (Color.CornflowerBlue);
        //TODO: Add your drawing code here
        base.Draw (gameTime);
    }
}
```

THE FIRST EXERCISES

The first exercises when teaching a new programming languages usually encompass the definition of variables for specific types, and compute and check their values, accordingly with some purpose. So, examples of first common exercises include reading a pair of values and computing the higher, computing the minimum common divisor, or just doing some other kind of mathematical computations.

In the context of computer games, reading values is not very common, and rarely useful for real-time games. Common input for a game is to check for the status of some keys, knowing if they are pressed or released; or to check the position of the mouse cursor, and which mouse button was clicked.

Checking the status of a key does not return a value, by itself, but it can be used to increment or decrement values, making objects move, for example. For the second input type, the coordinates of the mouse (or the touch, for mobile devices) correspond to a pair of coordinates that can be used for different types of computation.

So, in the context of computer games, my suggestion as a first exercise would be the movement of a sprite in the screen. First, define a pair of global coordinates (or a Vector2 variable, a type usually available in game frameworks) and initialize it with the middle value of the screen (width and height of the window divided by two). Then, in the Draw method call the appropriate framework method to draw the sprite in the screen, in that position.

At this point, movement can be added taking into account the pressed keys and changing the sprite coordinates accordingly. Note that the first games are really light, and will run at top speed, usually getting the 60 frames per second. This could mean that, if you increment a pixel at each frame, you will get a 60 pixel per second movement, which will be too fast. So, two different aspects can also be introduced: either define coordinates as real values, and increment a little at each frame, or define conditions over the elapsed game time, and increment values only in 10 frames per second, for example.

If the students grasp this concept easy, and have some background on physics, try to change the movement from the computation of a position directly from the keyboard input, and try to compute an acceleration vector from the key presses, updating the position at each frame with the acceleration value. This can, then, include drag easily. All these behaviors are accomplished using simple computations that use simple algebraic formulae.

The next construct to be included in exercises is, usually, conditions. For the example suggested earlier, conditions can be added for a simple task, like checking if the sprite has a valid position, not allowing it to exit the screen area. Note that different frameworks behave differently to invalid positions. Where some of them will cope with them, just not showing the sprite, some other will throw exceptions. As usual, class preparation is still needed.

With little more, a first real game can be implemented. Add two objects controlled by the keyboard independently, and some other object to move, and the result is a simple football game (1 against 1). If

the math to compute intersections between rectangles and circles do not fright the students, even the well-known Pong game can be implemented (ball and two rackets). Any of these two games would imply to distinguish between the position of the two players, the position of the ball, and the detection of collision between the ball and the players. The action on collision will require a simple physics that can be easily implemented, just computing angles accordingly with the collision direction.

Depending on the class goals and syllabus, and in case arrays can be introduced early in the course, then it should be possible to write some methods to deal with keyboard key states. Some game frameworks, like LibGDX and MonoGame, just have two different states for a key: it is either pressed or not. Usually, in games, it is useful to know if the key is pressed or not, but also to know when it got pressed or when it got released. For this, my usual approach is to define a method that, given a key, returns a value from an enumeration: *pressed*, *released*, *being pressed* and *being released*. For this to be possible to implement, arrays will be needed, as at each frame the complete list of keys used in the game need to be checked for their state change, and its state update accordingly.

CLASSES AND HIERARCHY

Whether to start with classes or arrays depends a lot on the personal point of view of the teacher. And although the approach here presented visits first the OO details, it would be possible to work first with arrays and then creating new objects.

The examples from the previous section can be easily converted to include objects, defining the ball as an object, and the rackets other objects. In fact, they share a lot of information. Both include a position, where in the screen to be drawn, the bitmap to draw, and possibly a rotation if desired. Students can start by creating two different objects, and then detecting the similarities. These similarities can be put in evidence, creating a generic sprite object from which the other will inherit all the information.

Usually my definition of *sprite* goes together with the definition of a *scene.* The *scene* is, basically, a list of all the *sprites* in the game. The *game class* initialization can create the sprites and add them to the scene. Then, the *game class* update method can call the update method for the *scene* that, basically, cycles through all the *sprites* in it and update them (calling their own update method). The same approach is done for the *draw* method: the *game class* calls the *scene* draw method, and it calls all the draw method for all existing *sprites*. This kind of approach allows that each game object behavior can be cleanly isolated from the remaining code.

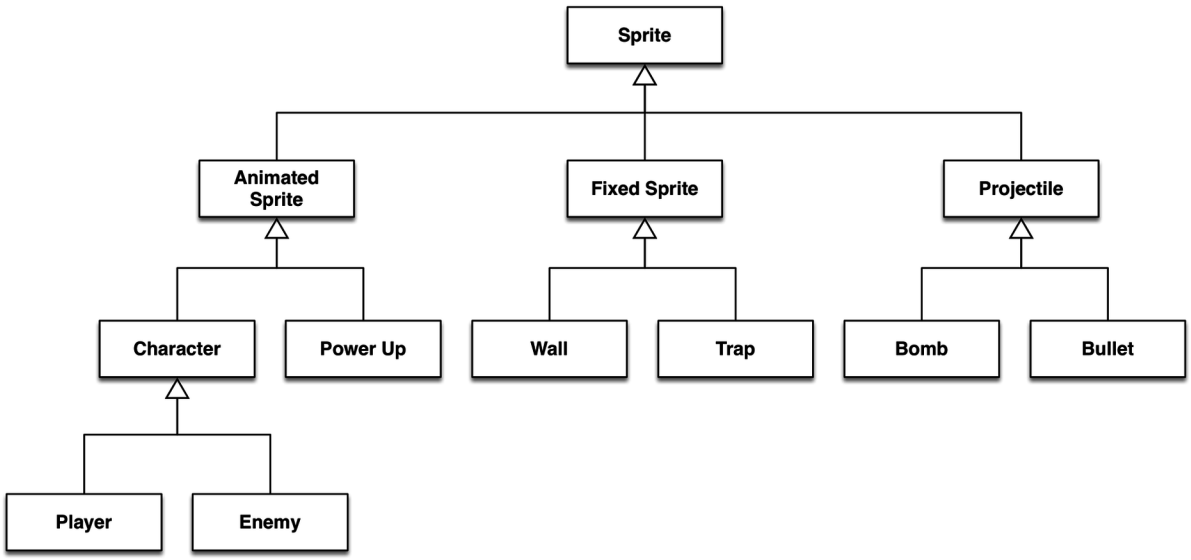
Follows the draft structure for a generic sprite object that can be easily combined with a scene object, and the draft structure for a generic scene object, that controls all the sprites currently being displayed.

```
class Sprite
{
    public Sprite(string name, Vector2 position, float rotation)
    {
          // save the name and load bitmap data
    }
    public virtual void Draw(GameTime gameTime)
    {
         // draw bitmap in current position
```

```
        }
        public virtual void Update(GameTime gameTime)
        {
             // add here the basic sprite logic
        }
        public virtual void SetScene(Scene scene)
        {
            Scene = scene;
        }
    }
    class Scene
    {
        private List<Sprite> sprites = null;
        public Scene()
        {
            this.sprites = new List<GameObject>();
        }
        public void Update(GameTime gameTime)
        {
            foreach (var sprite in sprites.ToList())
                sprite.Update(gameTime);
        }
        public void Draw(GameTime gameTime)
        {
            if (sprites.Count > 0) {
                  foreach (var sprite in sprites)
                       sprite.Draw(gameTime);
            }
        }
        public void Add(Sprite sprite)
        {
            sprites.Add(sprite);
            sprite.SetScene(this);
        }
        public void Remove(Sprite sprite)
        {
            sprites.Remove(sprite);
        }
}
```

Starting from this motivation, a more complicated game can expose more classes, and some more hierarchy. For example (see Figure 1 UML diagram), an animated sprite (that changes the bitmap every n seconds) can be a subclass of the generic sprite object. Each character in the game, being the player or the enemies can inherit from the animated sprite. In the other hand, walls, power-ups and other in-game

Figure 1. Class structure to relate different types of objects in a game. This approach allows the addition of specific behavior at each level, while allowing a common scene class to manage active sprites



objects that do not move and does not have any animation can inherit directly from the sprite object. Finally, another kind of object very common on games are projectiles and, yet again, they can be objectified, and some functionalities can be put in evidence: for example, if there are two kind of projectiles (from the player or from the enemies) they can share a projectile information: origin, direction, time to live and other game specific information. These projectiles would be added in run time to the scene object that would then be responsible for updating its position, and drawing it. Note that the only code that would need to be outside the game objects is for collision detection, as it will need to compare a sprite position with all other relevant sprites in the game. Collision detection, by itself, is a *complicated matter that will not be discussed here. Usually, in these introductory courses, I try to stick to circular or rectangular areas, making the collision detection easier. It is also possible to then compute pixel-by-pixel intersection if needed, after knowing two sprite bounding-boxes (or bounding-circles) are colliding.*

I confess that my will, as teacher, is to make this structure as versatile as possible. In the last year the class project included a tank, enemy soldiers, and projectiles from the tank and from the soldiers. Then, collision between the objects was added. And finally, some artificial intelligence was coded in the soldier objects, so they only move with the tank in a specific range. With all this in mind I started to create a generic model of classes and after 5 classes I noticed the students enthusiasm from the first class project has declined. And it was easy to spot the problem: a lot of classes with math (for rotation, speed of projectiles, etc.), concepts from OO programming (creating classes, inheritance, etc.) and no real gain in the final game (no changes on the visual or the mechanics of the game). That is, students were not able to foresee what was being built and how that abstract structure could help when creating a new game. Fortunately, some students, when developing their own projects, realized that they could use the same classes defined in the class project with little adaptations, and get a working game in a few hours. This was not bad, but made me realize that during next years I need to take into account how much of the code changes will affect the game, and try to motivate changes in the code structure with specific visual goals in the game.

In the other hand, if the students were not from the first year, the generalization of this hierarchy could be interesting as an exercise for the discussion of data structures, and showing that sometimes objects could benefit from inheriting from different classes. This is not possible in Java or C# directly, but some workarounds can be done with interfaces. From other languages I use for some research projects, like Perl, I miss the concept of *trait* or *role* in Java and C#. But this could be easily implemented as components that objects can aggregate (see Figure 2 UML diagram). As a simple example of such a component, note that a lot of objects have a life timespan, like projectiles or power-ups. These two kinds of objects are not directly related (other than being sprites). But is a sprite object could have a set of components aggregated, it would be easy to implement a timer, that will make the object trigger its destroy method. There are a lot of behaviors that can be implemented as components, and that would make the development of games easier and more reusable. But, just like I pointed in the previous paragraph, this extra work can demotivate students.

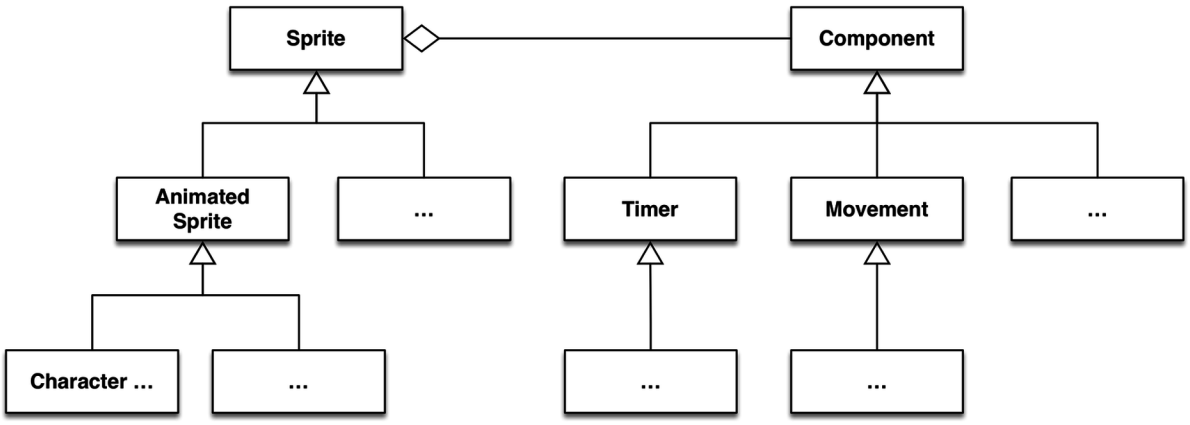
GRID ORIENTED GAMES

As referred in the previous section, teachers can choose if they desire to present OO concepts before or after explaining the concept of array and multidimensional arrays. In fact, I might argue that the process can be mixed up, presenting arrays at first, then exploiting the OO hierarchy, and then get back to arrays, explaining multidimensional ones.

In the common teaching approach, exercises with unidimensional arrays are quite easy to create with different backgrounds. But when creating multidimensional arrays exercises, teacher often need to resource to math exercises, like the addition or multiplication of matrices. As multidimensional arrays are homogeneous, storing only one object type, they are only useful for very simple exercises.

For games, it is quite easy to use a bidimensional array to store the structure of a game board, being it a typical board game, like chess, but also for other games whose world can be defined as an orthogonal grid, where on each cell a specific object is stored (like walls, powerups, and other objects).

Figure 2. UML class structure for sprite elements with components that will affect sprites with specific behaviors



Although at first this can seem a little limited, the number of games using grid oriented worlds, and their diversity is huge. The list can start with minesweeper, visiting Pac-Man, Sokoban, Tetris, Naval Battle, 4 in line, Snake or Reversi. In fact, lot of other, new games, can be implemented, either using a top-down view, or a 2D-platform like world. Grids even allow the development of a game where the world is a grid, but the characters walk continuously between cells.

Knowing what to store in each grid cell depends a lot of the game. Looking into a specific example, Pac-Man, there is a set of different entities that need to be represented: the walls, the pellets, the super-pellets, the ghosts and Pac-Man itself. All these objects have a position in the game state, and can be represented as a different value in a matrix (for example a character matrix, where different characters represent different objects). Nevertheless, this approach might give some trouble if everything is stored there. For example, some caution will be needed when a ghost walks through cells with pellets, because when the ghost leaves the cell, the pellet should remain there. This kind of problem will make the students try to find a solution. Some will store that information in each ghost object: "the cell where I am had a pellet before?" Some others will duplicate the matrix, and store in one the pellets, in the other the ghosts' position. Finally, other might come with storing each ghost object in an array, where the position is stored as a coordinate. So, a simple problem has very different solutions that can be discussed and exploited in order to analyze their efficiency and memory consumption.

Any of the games mentioned before will require students to perform some typical tasks. In the draw method, students will need to nest cycles in order to sweep through the matrix and draw each object correctly on the screen. To see if a character can move to a specific place, direct lookups will be needed. To understand if there is a line of objects, or a path, more complex algorithms will be needed.

One task that I find very interesting to propose to students (but that does not fit in all games) is to compute the neighbor of a cell. For instance, checking if a specific cell is surrounded by some type of objects. This is an interesting task because most students will go to the brute force solution, as there are, usually, only eight neighbors for a cell. Nevertheless, this is a good task to make students understand that cycles are not used only when sweeping arrays or lists, but can be used for other interesting tasks, like coupling two nested cycles, from -1 to 1:

```
for (int i = -1; i <= 1; i++) {
    for (int j = -1; j <= 1; j++) {
        if (i==0 && i == 0) continue;
        // check neighbor  (x+i, y+j)  for a specific condition
    }
}
```

The final task would be the implementation of a path finding algorithm. Again, this can be done in different ways. The Dijkstra (Frana & Misa, 2010) algorithm will be easy for the students to define by themselves. Every cell has at maximum 3 exists (as the other is the entrance point) and the algorithm can do the usual "turn right at each corner" rule, doing some backtracking when there is no exit. As this algorithm is quite straightforward, after knowing is basic rules (save the current path and each visited node), the introduction of A* (E. Hart, Nilsson, & Raphael, 1972) can be done easily, taking into account direct distance weight heuristic.

Final Remarks

In this chapter my main point was to motivate computer science teachers, especially those who are responsible for introductory courses on programming languages, to use computer games as their projects, both as do-at-home student project and as class projects. The kind of problems that arise are enough to focus the most important aspects of an introductory course on object oriented programming, ranging from the simple exercises with arithmetic computations, going through classes and objects, and ending in arrays and matrices.

I am aware that my suggestions are quite trivial, and any teacher would easily find these suggestions by themselves. My goal is more to show such an approach is possible, and allow the teachers reading this to think about this approach without the efforts on looking to these frameworks and understand how they could be used in the classroom.

My Experience

This entire chapter describes my experience on different courses, but mostly in Game Development Techniques, a course in a degree on Electronic Game Development. For this course, the MonoGame framework was chosen, as the students had a prior contact with C# using Visual Studio. Instead of starting with concepts, the first classes were dedicated to a grid-oriented game: Tetris. This implementation did not use objects, just local variables to define objects, and arrays (to define the grid area and the array of Tetris pieces.

During the last classes dedicated to this task, students started their first home project (in groups of up to three students). Their first project was the implementation of a Pac-Man clone. Although the idea frightened the students, most groups ended with good games, adding extra complexity over the original game, like bullets, teleportation, or extra speed power-ups.

After the Tetris project, the second project developed during classes was started, in a top-down game, with a tank, soldiers, bullets and bombs. As already described, this approach was not the best, as I took a lot of time in the construction of a flexible hierarchy of objects to make the game implementation easier. At this same time, students started their own project, with their game idea. The final games were very diverse, and some with very good quality.

Together with the C# and MonoGame projects, students were also forced to use Git for version control. For that, they requested a student account to GitHub, and created their projects there. This was interesting as I could see the log messages and each commit code, having an idea of how much each student had contributed to the project.

So, my approach was not exactly the one proposed in this chapter. The reason is, mostly, because the students had, already, basic knowledge on C#, arrays and classes. But as some students had these concepts not very clear, in this next year I would, mostly sure, use the approach described in this chapter.

FUTURE DIRECTIONS

There are a lot of further improvements that can be included in order to focus students on other tasks and techniques:

- Grids can be easily stored in textual files. Students can be asked to define more than one level for their games, storing that information on a text file. Then, at run time, the games would read each level from the game.
- Games have points, thus, every game should compute a player score somehow. And what makes gamification interesting is the desire of players to perform better. Students' projects can store a high-score list in a file, and update it whenever somebody breaks the record.
- Instead of using a grid, a game can use an open world, placing sprites at pixel positions. For this, sprites will need information on bounding boxes (rectangular or circular bounding boxes) so that collisions can be detected. For smoother collision detection, information about the pixels of each sprite and their alpha (transparency) pixels.

REFERENCES

Behr, J., Eschler, P., Jung, Y., & Zöllner, M. (2009). X3DOM: a DOM-based HTML5/X3D integration model. In S. N. Spencer (Ed.), *Proceedings of the 14th International Conference on 3D Web Technology* (pp. 127-135). New York: ACM. doi:10.1145/1559764.1559784

Chamillard, A. T. (2015). *Beginning C# Programming with MonoGame.* Burning Teddy.

Cook, J. (2015). *LibGDX Game Development By Example*. Birmingham, UK: Packt Publishing.

Falcon Corporation. (2016). *Sokoban Rules*. Retrieved January 29, 2016, from Sokoban: http://www.sokoban.jp/rule.html

Farokhmanesh, M. (2014, May 28). *Polygon*. Retrieved January 29, 2016, from Crytek releases CryEngine software via Steam: http://www.polygon.com/2014/5/28/5759208/crytek-releases-cryengine-software-via-steam

Frana, P. L., & Misa, T. J. (2010). An interview with Edsger W. Dijkstra. *Communications of the ACM*, *53*(8), 41–47. doi:10.1145/1787234.1787249

Griliopoulos, D. (2011, April 28). *IGN*. Retrieved January 29, 2016, from A History of ID Tech: http://www.ign.com/articles/2011/04/28/a-history-of-id-tech

Hart, P., Nilsson, N. J., & Raphael, B. (1972). Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *ACM SIGART Bulletin,* (37), 28-29.

Lee, K. (2014, March 18). *Techradar*. Retrieved January 29, 2016, from Unity 5 engine unveiled with better lighting, sound and browser gaming: http://www.techradar.com/news/gaming/unity-5-unveiled-with-better-lighting-sound-and-browser-gaming-1235019

Lippmeier, B. (2010, May 5). *Gloss: Painless 2D vector graphics, animations and simulations.* Retrieved January 29, 2016, from Hackage: https://hackage.haskell.org/package/gloss-1.0.0.1

Schend, S. E. (2007). Carcassonne. In J. Lowder (Ed.), *Hobby Games: The 100 Best* (pp. 46–48). Seattle, WA: Green Ronin Publishing.

Sung, K. (2013). *Learn 2D Game Development with C*. New York: Apress.

Tach, D. (2014, March 29). *Polygon*. Retrieved January 29, 2016, from Watch Epic's just-released Unreal Engine 4 in action: http://www.polygon.com/2014/3/19/5526910/unreal-engine-4-video-tutorial

Yaroslavski, D. (2014, June). *How does Lightbot teach programming?* Retrieved January 29, 2016, from Lightbot: https://lightbot.com/hoclearn.html