International Journal of Trend in Scientific Research and Development (IJTSRD)
International Open Access Journal
ISSN No: 2456 - 6470 | www.ijtsrd.com | Volume - 2 | Issue – 3

Learning Game Development Life Cycle through Project-Based Approach

Advait Maindalkar, Sarvesh Gharat, Prof. Vinod Rathod

Department of Computer Technology, Bharati Vidyapeeth Institute of Technology, Kharghar, Navi Mumbai, India

ABSTRACT

In the field of software engineering, game development has been, and continues to remain a significant and promising sector. The growth of this field is hard to predict, but there is no denying that it has potential. A tremendous research has been going in the gaming industry mainly in the areas like game development, game designing, game AI engines, educational games. The emergence of game engine tools like Game Maker, Unity 3D made life easier to design and develop more sophisticated games with the reduced amount of time. Smartphones are one of the most widely owned and used consumer electronic devices all over the world, amongst which, Android smartphones are the most widely used, due to their various features and economic pricing. And Google's Play Store has rather friendly developer policies has caused many developers to migrate to Android. Hence the Android Marketplace is home to a plethora of games, of all sorts of categories. However, upon closer inspection, it's observable that users prefer casual, time pass and fun-oriented games more. Due to this, we wanted to keep our game simplistic, hence we opted to make a 2D game, and decided to pick the puzzle genre, because people tend to like simplistic yet challenging games, and get addicted to it.

Keywords: *Unity, Game Engine, Component-based Coding, C#, GUI, Sprite, GameObject*

INTRODUCTION

Games play an important role in the daily lives of many. Video game has come a long way since its origin. Video games could be designed for different types of platforms ranging from computer to smart phones. A tremendous research has been going in the gaming industry mainly in the areas like game development, game designing, game AI engines, educational games. The emergence of game engine tools like Game Maker, Unity 3D made life easier to design and develop more sophisticated games with the reduced amount of time. The growth of this field is hard to predict, but there is no denying that it has potential.

Games are a medium of learning and entertainment, which have been widely enjoyed and played by several people over the world. The gaming industry has come far ahead from the times of small, old, 2D, monochrome or 8bit coloured games, to immense, 3D games with photorealistic 4K textures, high-polygon count detailed models, comprehensive tessellation, accurate physics, multiplayer elements, etc.

Technically, a game is a software that is designed and developed with a certain characteristic goal to provide entertainment. Similar to software products, there is a development life cycle for delivering a successful game, namely Game Development Life Cycle (GDLC). For a good quality game, both development and design has to be balanced. Designing computer games requires adequate experience, and great attention to detail to describe the rules & aesthetics that compose the interactive experience.

Worflow of Game Design in Unity

Unity Game Engine

Unity is a multi-platform game engine,created by Unity Technologies, which is used in the development of games, both3D and 2D, along withvarious simulations and experiences for computers, consoles, and mobile devices. It supports 2D and 3D graphics, drag-and-drop functionality and scripting using C#.

Fig 1: *Unity, the Game Engine*[1]

Within 2D games, Unity allows importation of sprites and houses a feature-rich 2D world renderer. For 3D games, Unity allows texture compression, mipmaps, and resolution settings for each platform that the game engine supports, and provides addiotional support for "bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects[1]„

Unity supports building to 27 different platforms, including Android.

Component-Based Game Design of Unity

Object-Oriented Programming is used de-facto in game development, and it *can* be used in Unity to create highly-productive workflows. However, Unity functions better in component-based design.

In the world of programming, the notions of components and decoupling go hand in hand. A component is visualised as a smaller piece of a larger machine. Each component is assigned its own specific job, and can generally (and optimally) accomplish its task or purpose without the help of any outside sources. Furthermore, components rarely belong to a single machine, and can be joined with various systems to accomplish their specific task, but achieve different results when it comes to the bigger picture. This works since components neither care about the bigger picture beyond, nor know it exists.

The hardest part about working with components, however, is learning how to assemblerelated projects when using them. In most cases, this usually means in the creation of a lot more scripts – eachdoing smaller, more specific tasks.

How communication between scripts occurs is also a significant issue for consideration, as there will be a lot of tinierpieces and lesser giant classes where every object knows about every other object. There are ways around this, such as static variables for core components of a particular game, such as the instances of Player.However, thisseldom works for everything, and it is unwise to do so.There are several advanced methods to structure your components properly, and to stay decoupled.

Although, given the factthat Unity has been built with the focus on components, it has a number of built-in tools that help accomplish this. There are functions to get references to a specific component of a specific object, to check all objects to see which contain a specific component, etc.

With these types of various useful functions, relevant information can easily be retrieved, to create that magical one-way street of knowledge where components can communicate with objects they affect, but the component itself has no visibility of what exactly that object is. All these tools, combined with the use of interfaces, can give superior programming capability and flexibility, allowing any tasks, big or small, to be tackled efficiently.

Since Unity works on Object-Oriented Programming, it treats everything as objects internally. Everything can be referred to in the form of a GameObject, which in itself is a Component, which can then be used to gain reference to any of the attached component(s) and its respective properties.

For 2D Games, which was the scope of our project, the first step is designing, by creating sprites for the in-game objects. *Sprites* are 2D images, which represent some object in your game. E.g. for your player, the image of a person holding a gun could be a sprite. Once you have created the Sprites, you load them into the Unity project, apply required compressions, and then the assets are ready to be used. It is advised to get this done before anything

else, as alignment of the objects in the game world is done based on the visual part of the objects. This holds true especially since Unity has a drag-and-drop interface.

After this step, the sprites need to be dragged into the game world space, and positioned appropriately. Doing so automatically creates and object, with a Sprite Renderer component attached to it, which contains the source as the image you chose to drag in.

Now, appropriate "components" can be attached to this object, via the new component button available within the Unity Editor UI. It can be observed that Unity features a plethora of different components, each with its own specific, small functionality, but can be combined with other components to create a dynamic and feature-rich ecosystem of objects.

After this, usually scripting begins. The objects need logical code to be able to perform certain behaviours, without which the given object is merely a static object residing somewhere in the game world. Again, each specific function needs to be separately programmed into a script. E.g. if say a game consists of a player and several enemy units, each having health, aiming functionality, and a certain death condition and/or animation, there needs to be a separate script for health mechanism, death mechanism, and similarly for aiming. Health, e.g. could manage operations related to health, like returning the current health value, affecting and/or changing it, and deciding when to invoke death upon the player, etc. The Death script could handle death invocations, and play a certain animation based on whatever object it is attached to, and the aiming mechanism can define how the character aims, plus an another script for how enemy AI aims.

These individual scripts, as components, can be applied on different objects, their values can be tweaked for each object separately, and already this is way simpler than re-creating the same code differently for objects like player, enemy type I, enemy type II, etc.

Creating Luminux

While creating Luminux, we first made a level draft, creating necessary visual assets. This makes further development easier as the interface is based on drag-and-drop functionality for the arrangement of objects or elements. Creating assets beforehand makes it easy to set up collision detection masks on the sprites. It is more helpful having this part done beforehand, as then you can focus on the development part and bringing everything together, and wiring the components together appropriately.

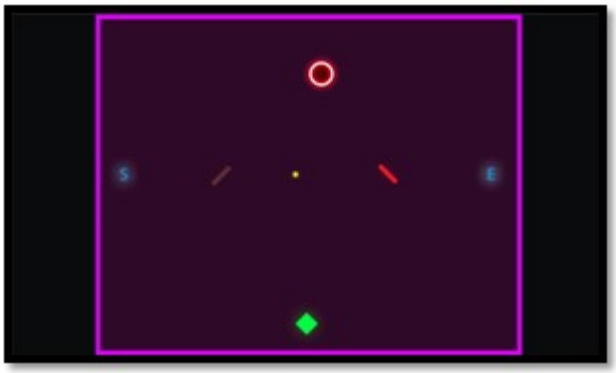

Fig. 2: Level Draft of Luminux

Then, we built each object from ground-up, coding the functionality of each individual element, and ensuring that there is proper communication between elements.

Unity allows accessing of any component via its "GetComponent" function, which can be used to gain reference to the script component attached to another GameObject. Then, to communicate info between objects, the "SendMessage" function can be used, which supports one parameter, to ensure flow of data from one object to another while still maintaining abstraction and protecting encapsulated data.

Slowly, as each component was built, we integrated them into objects, and performed unit testing on each individual unit. This ensured that any bugs that surfaced due to coding errors were fixed right away. Then, these unit tested components were bought together and assembled according to the game's design, and then Integration Testing was conducted, testing the functionality and inter-communication of the various objects and components, and ensuring consistency across several usage scenarios.

Unity is a rather powerful and flexible game engine, and it looks after garbage collection and memory leakage itself, so the developer is relieved of these concerns and can focus on the game logic instead.

The aforementioned incremental development and testing method helped us create the specific and various functional components of the game quickly, by assembling together the smaller component and wiring their required and provided data sources.

The end result was a fully functional game, with several gameplay elements, and properly-usable UI, along with suitably tested components which resulted in a complete and professional experience for the users who played the game.

CONCLUSION

After creating the game for our project, we learnt game development approaches and fundamentals, as well as how to use and code with game engines like Unity. This helped us understand game development life cycle, and how to do component-based programming for Unity. It also helped kick-start our game development phase.

REFERENCES

1. https://en.wikipedia.org/wiki/Unity_(game_engine)
2. "Game Design Research", Annakaisa Kultima,https://www.researchgate.net/publication/282185969_Game_Design_Research
3. Rido Ramadan and Yani Widyani, "Game Development Life Cycle Guidelines", http://ieeexplore.ieee.org/document/6761558/