MULTIPLATFORM GAME DEVELOPMENT USING THE UNITY ENGINE

Roman Jašek
Master Programme (3), FIT BUT
E-mail: xjasek01@stud.fit.vutbr.cz

Supervised by: Rudolf Kajan
E-mail: ikajanr@fit.vutbr.cz

Abstract: This paper deals with the aspect of developing games for multiple platforms concurrently. It uses the game engine Unity 3D to explore the possibilities and issues that occur in multiplatform development nowadays. The focus of this paper is the development for both the desktop computers and mobile devices.

Keywords: Unity 3D, multiplatform, games, optimalization, Windows, Windows Phone, Windows RT, Android

1 INTRODUCTION

There has been a spread of various platforms in the computer world in recent years. Hand-in-hand with this came the evolution of games. What was once a field of interest only among a specific group of people became a respected industry. The games have evolved from difficult concepts played on specialized devices to casual time-killers available on computers and mobile phones used by billions of people daily. In order to react to these new conditions new tools were presented to allow developers to easily target different types of devices and platforms simultaneously. We are going to take a look at an example of this approach - a game engine called Unity 3D [2]. Moreover we are going to examine the approaches needed in order to use this engine for multiplatform development and deal with the challenges that come along with this process.

2 RENDERING OPTIMIZATIONS

In order to be able to launch a game on mobile devices the developers need to deal with low performance hardware. So if we want to develop for both the desktop and mobile we need to take advantage of a few tricks that are offered by Unity.

The biggest performance killer is the amount of times that the grahpics API has to draw something on the screen. So we will explore the possibilities of reducing the number of draw calls.

Independent meshes usually need separate draw calls. This can be however avoided for certain meshes. If there are multiple meshes that do not move independently and share the same material they can be combined into one larger mesh. Unity can partially do this automatically with batching. It can combine meshes that are marked as static and are just copies of one another. This approach however brings some limitations. The meshes to be combined cannot have different sizes and only up to 900 vertices can be put together this way. Moreover we cannot specify which meshes are batched.

Another way to achieve combining of meshes is to use a script to combine such meshes. By using this option we can specify which meshes we want to put together. We can also combine skinned meshes which are not supported by the standard batch script and meshes not marked as static and even meshes with different sizes. The meshes with the same material are arranged as a group under a game object within the Unity's hierachy. Then we can attach our script to this group and call it on all of the meshes. The usual approach is that we combine the meshes in the Start function of our game. This way it is called only once at the beginning and does not slow down the game loop while the game is running. This approach has been tested in a scene with several crates and barrels staged in the corner of a room. The results of applying the script to combine meshes can be seen in table 1. We can see that the draw calls dropped by 59 when we applied this technique from 159 to 100. This is a significant improvement considering the relatively small testing scene.

Technique used	**Draw calls**
Automatic mesh batching	159
Combining meshes using script	100

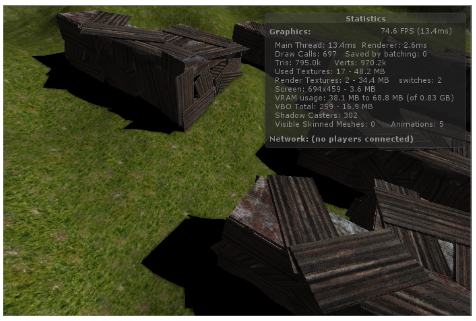
Table 1: Comparison of combining meshes using different techniques.

We can also combine meshes with different textures. This however requires more work. In order to combine these meshes we need to create a texture atlas. It is basically a large texture that contains smaller textures used by various meshes. This needs to be prepared outside of Unity and imported as an asset. Fore example this can be used to draw a whole building which originally contained many different textures for windows, walls roofs and other parts. By combining the meshes and using atlas textures we can achieve greatly enhance the performance of our game and make it run even on mobile devices.

3 BAKING LIGHTS

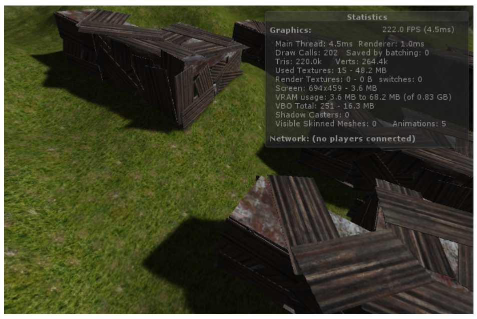
Another technique that we can use to make the rendering more fluid is baking of lights. It means that we precompute the lighting attributes of the surface before we launch the game. Then we store the shadows in the textures.

Another test scene can be seen in figure 1 with the results of light baking. Images show a look at a few sheds and both are lit by 1 directional light. The light in the scene 1a has hard shadows turned on and renders at around 74 frames per second. However the scene 1b does not have any shadows turned on. The shadows are baked into the environment by the lightmapping feature. This brings the frame counter to 222 frames per second which is almost 3 times as much as in the first scene. The edges of the baked shadows are even smoother than the ones computed on the go as can be seen in the figure 1. While 74 frames from the first scene are fluent, the example shows only a simple scene and this issue would greatly influence a more complex scene.



```
Statistics
Graphics:                    74.6 FPS (13.4ms)
  Main Thread: 13.4ms  Renderer: 2.6ms
  Draw Calls: 697   Saved by batching: 0
  Tris: 795.0k     Verts: 970.2k
  Used Textures: 17 - 48.2 MB
  Render Textures: 2 - 34.4 MB   switches: 2
  Screen: 694x459 - 3.6 MB
  VRAM usage: 38.1 MB to 68.8 MB (of 0.83 GB)
  VBO Total: 259 - 16.9 MB
  Shadow Casters: 302
  Visible Skinned Meshes: 0     Animations: 5
Network: (no players connected)
```

(a) Dynamic hard shadows.



```
Statistics
Graphics:                    222.0 FPS (4.5ms)
  Main Thread: 4.5ms  Renderer: 1.0ms
  Draw Calls: 202   Saved by batching: 0
  Tris: 220.0k     Verts: 264.4k
  Used Textures: 15 - 48.2 MB
  Render Textures: 0 - 0 B  switches: 0
  Screen: 694x459 - 3.6 MB
  VRAM usage: 3.6 MB to 68.2 MB (of 0.83 GB)
  VBO Total: 251 - 16.3 MB
  Shadow Casters: 0
  Visible Skinned Meshes: 0     Animations: 5
Network: (no players connected)
```

(b) Baked lights

Figure 1: Light baking demonstration.

Light baking can be controlled from the Lightmapping window in Unity. Various parameters can be set there to influence the light baking. Resolution and Final Gather Rays are the settings with the biggest influence on the speed of baking. The resolution affects the final resolution of the generated texture and the other parameter states the number of rays shot from every final gather point [1]. In order to get a quick bake the values of these parameters should be set low. This can be used for debugging purposes, when we just want to see the general way of lighting in our scene.

4 USER INTERFACE

Working with user interfaces on various form factors and taking different user controls in account can be tricky. Unity provides two approaches to creating user interface.

The first way is by using Unity's game objects. GUIText is a simple text field that allows text data to be displayed. Its basic properties can be modified including the color and font. GUITexture can be used to create more controls. It displays a texture on the screen and can react to various events such as a mouse click. Both GUIText and GUITexture are positioned relatively onto to screen. Therefore it is not needed to care about the specific resolution and they are always placed where intended even if the screen resolution or orientation changes. Moreover they do not effect the performance of the game much. While testing we created a game filled with 72 buttons and it ran at 60 frames per second even on a mobile device. Game objects are therefore usually used for in-game menus, which need to be displayed while the game is running and are not desired to take up significant performance.

On the other hand we can create the user interface using scripts. This approach allows us to use more advanced controls like sliders and scrollable text areas out of the box. They are easy to use and support various scenarios. However they are very different to using the game objects. Their position on the screen is absolute so we need to recompute their position on the go manually. On top of that they take up a lot of computing power. The aforementioned test was done and the frame rate dropped to only 40 when running on a mobile device. This would be a significant drawback if it happened during an active game. Therefore this approach is usually used within the menus, while the actual game is not running.

5 CONCLUSION

This paper dealt with the challenges that developers face when trying to develop for multiple platforms using the Unity engine. It presented issues that are commonly present especially in the mobile development and provided solutions to these issues. The intention of this paper is to inform Unity developers about the options they have when focusing on multiple systems at once.

As there is no comprehensive set of solutions to these issues a tutorial is being created to help other developers to see these solutions and use them for themselves. It will be available online once it is done and will be offered to the Unity developers on the Unity's web forums. It will cover all the challenges mentioned in this paper and will add even more if requested.

The market of the mobile devices is currently spreading rapidly and developers want to create games for as many of them as possible. Therefore I believe that the theme of multiplatform development will be very interesting for the community around Unity as this engine supports and encourages this approach.

REFERENCES

[1] Unity Technologies. Unity - lightmapping in-depth. `http://unity3d.com/Documentation/Manual/LightmappingInDepth`, 2013.

[2] Unity Technologies. Unity - game engine. `http://unity3d.com`, 2014.