Developing a 3d interactive adventure game using Unity

Game development concepts with emphasis on texture compression and anti-aliasing

Peter Broderick and Joe Sweeney

BACHELOR OF SCIENCE(HONOURS) IN COMPUTING IN SOFTWARE DEVELOPMENT

Galway-Mayo Institute of Technology
Ireland
28/05/15

Contents

Abstract

The aim of this project was to examine how 3d games are designed and the mechanisms behind the development of these types of games. We look at how games are developed using the Unity 3D game engine. This game engine is one of the most popular game engines on the market for developing 2d and 3d games. Some popular games developed in Unity 3D include Assassin's Creed Identity, Temple Run Trilogy and Wasteland 2. Before we chose Unity 3d as our game engine we looked at how we would be able to develop our game and deploy it to a multitude of platforms like android, windows, IOS etc. Today most people have a mobile device, laptop or tablet and this is why we put a lot of research into finding the right game engine that would support this. After conducting research on Anti-Aliasing and Texture compression, the results of which can be found in their respective sections we tried to run the game on android. We did this using various Texture compression techniques but found that if a particular android device didn't support our chosen metho dof compression it would default to Ericsson's ETC texture compression method. The, alternative was to deploy the game in many different formats to the Google Store so that each device would automatically download the texture compresison technique it best supports and failing this would use ETC. But without an Alpha channel(ETC doesn't support this) the game would not look optimal. As this was a research project and the game was developed as proof of concept and not meant to be of merchantable quality, the fact that the aesthetic quality wasn't perfect was not an issue for us. However, we opted against deploying on android because the control system lended itself better to a PC gaming environment, so we deployed on PC instead. This option allows you to change the graphics settings by default at Run-Time, High Quality will sacrifice speed of performance but reduce MIP-Maps(Which are discussed later) and use Anti-Aliasing to smooth edges etc.

In terms of how the game turned out as a finished product we are happy to have met our expectations and on the whole though Unity was a good option to have chosen as beginners as opposed to some of it's alternatives and we feel that moving forward we have an in-depth knowledge of the Unity architecture and a much greater understanding of all aspects of Video Game development.

1 Introduction

For our final year project we decided to develop a 3d adventure game using Unity 3D for windows. Up until the last few years licensing a premier game engine would have been very expensive, but in the last few years Unity has released a free version of their gaming engine. This has led to prices for high end game engines plummeting in recent years. The Unity game engine supports a vast array of platforms which includes Windows, Mac, Linux,Android, iOS, Blackberry, Playstation and many more. There is a multitude of reasons to why we chose Unity to develop our 3D adventure game but the main reason was because of its multi-platform capabilities. With mobile apps becoming more

and more popular every year we needed a way to deploy our 3d adventure game on as many platforms as possible. We aim to provide an in depth look at what Unity has to offer from it's easy to use mechanisms to its multi-platform deployment capabilities. The programming language used for the game that we have developed is C-Sharp. We use this programming language specifically for scripting in the game.The project demonstrates the basic features of a First Person Adventure game and the process of the 3D game designing with the Unity game engine. We will address all aspects of the development process that we undertook, including coding for Unity using monodevelop, model creation, the architecture of the IDE and pay particular attenition to the way 3d games are developed with regard to Textuure Compression and Anti-Aliasing, which can be easily done through UNity through various methods. We will describe the Unity game engine and show the basic structure and layout of the development environment. Our goal was to create an adventure game that included characters the player could interact with, enemies that could attack the player, an inventory that would hold the items collected during gameplay, we also wanted to experiment with model creation and animation. The game's story wasn't an important factor for us as we were more concerned with the development process from a low-level standpoint.

Texture Compression Techniques in 3d gaming.

Peter Broderick

November 10, 2014

Abstract

The aim of this Literature Review is to look at different methods of texture compression with particular attention to those that are used in the ever expanding world of 3d gaming. We will seek to examine different types of texture compression algorithms and contrast them to try to understand the reason why certain techniques are preferred to others. Our goal is to determine the most effective texture compression algorithm and perhaps identify more that may have a future in gaming. We will also attempt to demonstrate some of the challenges that are encountered when developing texture compression algorithms.

1 Introduction

When creating 3d games, developers are constricted by hardware limitations. Developing for a particular system demands that certain techniques be employed to utilize the machine efficiently in order to render games seamlessly; the ideal outcome is the user experience being maximised without inordinate pressure being put on the underlying hardware. Components such as main memory, CPU, GPU need to be considered. One of the biggest challenges presented is that of texture mapping; texture mapping is a process that allows images to be mapped onto objects in a gaming environment to give the illusion of complexity. This rapid rendering can consume a lot of memory, effecting performance as a result. This has prompted the introduction of high quality texture compression algorithms. These algorithms allow a given system to render compressed versions of textures as opposed to the textures themselves. This allows the developer to use higher resolution textures, which would not be possible without texture compression. With the extra space that there is in memory as a result of compression you can use it to perform other functions which will be discussed in detail further on. Online games are almost impossible to create without texture compression, if we consider the fact that even PC graphics systems find it difficult to provide enough bandwidth to games(Video Ram needs to draw images at extremely high speeds), then it's not hard to imagine how internet applications struggle to run games.

2 Benefits of freeing texture memory

Textured models look much more visually appealing than non-textured ones, however creating these presents drawbacks in the form of high data volume and considerable processing times[1].Using a good texture compression algorithm has benefits on both hardware and software computational elements in a system. Below I will describe how freeing up texture memory can improve performance.

2.1 MIP-Mapping

MIP-Mapping is a process that has become prevalent in 3d gaming in recent years and consumes a considerable portion of texture memory, while it is usually an optional feature in PC gaming, it couldn't even be considered without texture compression. The process is performed using a series of layers. The first layer which is the one nearest the main camera, is drawing on compressed textures, however, to be more economical the further layers have a decreasing degree of complexity relative to the main camera. This is achieved by duplicating the main texture and scaling it down rapidly All filters are approximated by linearly interpolating a set of square box filters, the sides of which are powers-of-two pixels in length[2]. Put simply, a block of 4 pixels will be divided into Red, Green, Blue and and 1 unused block and that unused block in turn will represent 4 more pixels with same contents and so on until there are a series of image copies with diminishing accuracy. 1

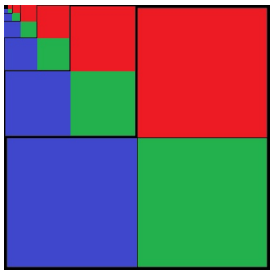


Figure 1: Oraganisation of a MIP-Map

As the camera location moves forward these layers will scale back accordingly. Without MIP-Maps everything is being rendered at once, hence it is computationally expensive for the graphics processing unit. 2

2.2 Triple Buffered Rendering

Double buffering is a computer graphics standard. With double buffering however, you are forced to wait until data is moved from one buffer to another before either can be accessed. This causes a delay of a few milliseconds, which can lead to screen tear in gaming, which is where an image can briefly appear to have been torn into two sections. Triple buffering uses 2 back buffers, and a

Figure 2: The left image which is using MIP-Maps is rendering textures fully in the foreground but is using scaled down images in the background. Note the background trees which look blocky in contrast to those on the right which are fully rendered.

front buffer [3]. The front buffer is constantly interacting with the GPU. While one of the 2 back buffers is copying to the front buffer the other is drawing, this is illustrated in the image below 3. When the front buffer is finished it's transaction with the Graphics card it switches to the ready buffer. This process is tied to the refresh rate of the associated monitor, so that if both buffers contain images that haven't been shown by the monitor, then drawing is not done which means there are less CPU invocations.

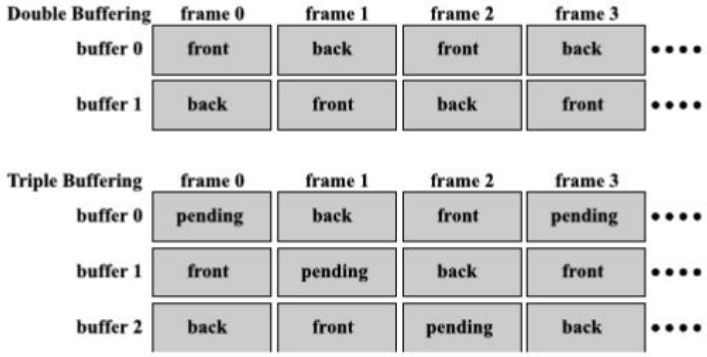


Figure 3: The Double buffering system on the top is constantly switching buffers from front to back whereas the triple buffered system has an extra buffer that is getting ready to be invoked. This image has been adapted from Haines et al.[3]

2.3 Improved Sustained Fill Rate

The amount of pixels a graphics card can render and then output to memory in a single second is called the fill rate. The way your system is developed

from a hardware standpoint effects the amount of memory assigned to your graphics engine. Textures that have been compressed take up about a third of the bandwidth of uncompressed data so can be fetched faster. This increases the fill rate.

3 Literature review

When choosing a texture compression technique, there are challenges to consider. Beers et al (1996)[4] state four challenges they see and how a technique called Vector Quantisation(VQ) approaches them. The four challenges they identified are Decoding Speed(1), Random Access(2), Compression Rate and Visual Quality (3), Encoding Speed (4).

1. Decoding Speed: Vector Quantisation has a very high decoding speed as it uses a table lookup. Table lookups use a simple index array to save on processing time. The decoding speed must be extremely fast for any image compression technique associated with gaming. The decoder must not be complex.

2. Random Access: Texture compression must make it possible for texels to be quickly randomly accessed. The problem of random accessibility is also solved using indexing by VQ. When textures are being rendered in real time it is impossible to predict when and where texture pixels will be accessed from. VQ (as with other techniques such as FTC which will be discussed later) has a fixed compression ratio, this means it uses a fixed number of bits to represent each block of pixels so you can directly access that block and subsequently the pixel. Other schemes such as JPEG can force you to decompress entire blocks to locate a particular pixel.

3. Compression Rate and Visual Quality: While there are lossless compression schemes available [5], that will accurately replicate a texture, these aren't a priority in 3d rendering as they can be hardware intensive, and for the most part unnecessary. While it is important for a compressed replication of an image to remain faithful to the original for many endeavors, this in not necessarily the case for textures in 3d gaming as losses aren't as easily distinguishable under the guise of a full, complex, 3d environment. In these cases the images are viewed as a collective and are changing so quickly that it isn't worth draining the already limited supply of texture memory.

4. Encoding Speed: VQ Encoding can be slow. Fortunately, it is more important that the decoding be quick than the encoding and although fast encoding has it's benefits, it is much more important to encode accurately as the majority of textures used can be generated at authorship.

3.1 Tree structured VQ

VQ has been used for decades. [6] The main principle behind it is that instead of using the entire spectrum of colours, a colour codebook is generated which will index a smaller number of colours that represent the actual image's colours. The point of texture encoding is to compress images so that they can be retrieved

in reasonably high quality. The more arduous task is to allow this data to be retrieved quickly. There are 2 ways this can be done in VQ, one is 'the Generalised Lloyd Algorithm' proposed by Katsavounidis et al[7], and the other is 'Tree structured VQ'. Both of these methods generate a codebook(so called because it decodes index locations into vector values), this is a set of pixel blocks which are derived from a larger set of pixel blocks. The smaller set of pixels is then indexed and each set of pixels will have it's own unique index on an index map. 4 They can be used later to lookup specific larger blocks. This eradicates the need to search for each pixel block of a texture individually. Out of the two methods mentioned, 'Tree Structured VQ' is preferable for 3d gaming as it uses a binary tree to organise codebooks. This makes Decompression extremely fast when compared to GLA or other VQ techniques such as Spatial Vector Quantisation(SVQ)[8] which is used in Image based rendering(IBR) , and has such a poor compression ratio that it uses a technique called qzip [9] to compress images further and as a result the qzipped SVQ index isn't randomly accessible.

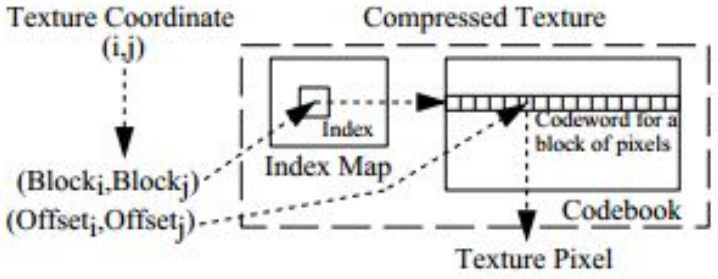


Figure 4: Codebook. This image has been adapted from Beers et al.[4]

TSVQ consists of an encoder and a decoder. It works by parsing an image into pixel groups. These are usually 2x2 squares. Then the encoder produces a binary vector(Which is comparable to a node on a binary tree data structure). The decoder(Table lookup), later receives a binary vector then outputs a stored codeword, which is a word in memory indexed by the binary vector. If there is access to a all possible codewords via a codebook(binary tree)then the decoder is completely described.

The Codebook is generated by creating a tree in a 'Greedy' manner [10]. You start with the root and from this two child nodes are derived. Each of these child nodes is labelled by a separate vector to produce a 2-word code. A clustering algorithm is then run to create a codebook from all the 2-word codes in the tree.

Vector quantisation aims to create a less accurate representation of an image without sacrificing too much of it's aesthetic. One way it does this is by scaling down (Quantizing) things such as colour intensities. With scalar quantisation we can represent a colour in graphics that has a floating-point value which

is between [0.0, 1.0] (Potentially a very large number) with an integer value that's between 0 and 255 (A comparatively much smaller number). This greatly decreases the value to an 8-bit number. Using this method we can then make a colour that is comprised of RGB elements into the standard 24-bit format(8-bits for red, green and blue).

3.2 S3TC

As previously discussed, lossy texture compression algorithms are commonly used in 3d gaming, the reason for this is that they don't need complex hardware systems to run on. S3 Texture Compression(S3TC) is widely used in the gaming world, with Open GL and DirectX(S3TC is called DXTC in DirectX) and has become an industry standard in recent years, with hardware support from large companies including Microsoft. It is used by the Unreal Gaming engine on which popular games such as Battlefield 3 (The game's two 4096 x 2048 terrain textures are compressed to bring memory usage down to 8mb from 32mb), Team Fortress 2, Batman: Arkham Asylum, Dishonored and various other new games are built. The engine has become ubiquitous in gaming in recent years and it's games are far less hardware intensive than games with similar graphical detail.

S3TC doesn't rely as much on codebooks to compress textures as other algorithms [11]. It instead represents each texel with a 4x4 pixel block, which amounts to 16-bits, there is a 2-bit index assigned to each pixel block which amounts to 32-bits. Two 16-bit colour values are also included to create a compressed block size that is just 64-bits in total. If you were to replicate this without any compression an RGB565 image would have 256-bits, comprising of; a 4x4 pixel block which is 16bits each pixel therein with a colour value of a further 16-bits. [12] So, as these blocks are self contained it is not necessary to use codebooks as with VQ, therefore only one fetch is required which greatly helps performance. S3TC decodes quickly and decompresses almost flawlessly(for a lossy compression scheme). It allows game developers to compress during installation, in contrast to older compression schemes which force the developer to compress during authorship.

The fact that S3TC has a fast decompression process as a result of it's method of dividing texture blocks into equal sizes and segregating them, is perhaps the reason it has become an industry standard in 3d gaming and colour codeword dependant systems have not. When using S3TC you generally compress textures as they are being created. However, if you decide not to do this and want to compress them later S3TC can compress textures very quickly and still maintain 95 percent of the texture's quality. This is a major factor in it's employment by game developers; where some require you to compress textures at authorship, S3TC allows game developers to compress when a game is being installed, when the game begins or as levels load.

3.3 Prospective alternatives to S3TC

Floating Texture Compression: Philipp Klaus Krause, describes FTC as having the ability to produce superior image quality(To S3TC) with the same compression ratio or comparable quality at twice the compression ratio [5]. FTC as with most modern compression systems are designed to utilise the same decompression hardware that is used on S3TC as this is so hugely supported in industry, which helps alleviate any difficulties trying to garner support for a new system.

FTC uses variable precision when generating codewords. For example, older techniques that use colour codewords hold codewords with fixed accuracy and they are all stored directly. FTC tries to put emphasis on making the colour codewords that are near to each other as accurate as possible but the ones that are far away less so. It does this by not storing all the codewords directly, conversely it will only store the first codeword and the other codeword is stored as an offset of the first one, relative to it's distance in colour space. When the codewords are close to each other, more bits are used to hold the first one than the second.

As is the diagram below 5, there are 10-bits for each of the colours RGB and 2-bits represent the exponent e. The main colour will be assigned (5+e) bits, and the secondary colour will be assigned (5-e) bits.

J.Nystad et al. [13] present another system of texture compression called Adaptive Scalable Texture Compression which provides better quality of 3d image compression than most in use today and substantially better than that of S3TC. In fact after Nystad et al. compared ASTC to DXT(S3TC) they concluded the following: "The Resulting image quality is competitive with the most advanced formats in use today, and better than that of industry standards such as DXT and PVRTC". As with S3TC it also uses the block based system that is also used in other modern techniques.

ASTC has the potential to become ubiquitous going forward, as it covers a broad range of bit rates. For example PVRTC(Which is not block based) supports only 2-4 bits per pixel and S3TC supports 4-8 bpp which is more suited to higher end devices but isn't ideal for mobile devices. ASTC on the other hand supports from 8-bpp down to as little as 1-bpp. PVRTC is useful for mobile devices and is used for all iOS devices because of it's low bitrate support and because of S3TC's support for higher bitrates it is widely used in gaming. ASTC covers a broad range and has recently been picked up by devices such as the Sony Playstation 4. It has been tested against PVRTC and S3TC at similar bitrates to theirs and outperformed them both. 6

Figure 5: ftc1 data layout. This image has been adapted from Krause et al.[5]

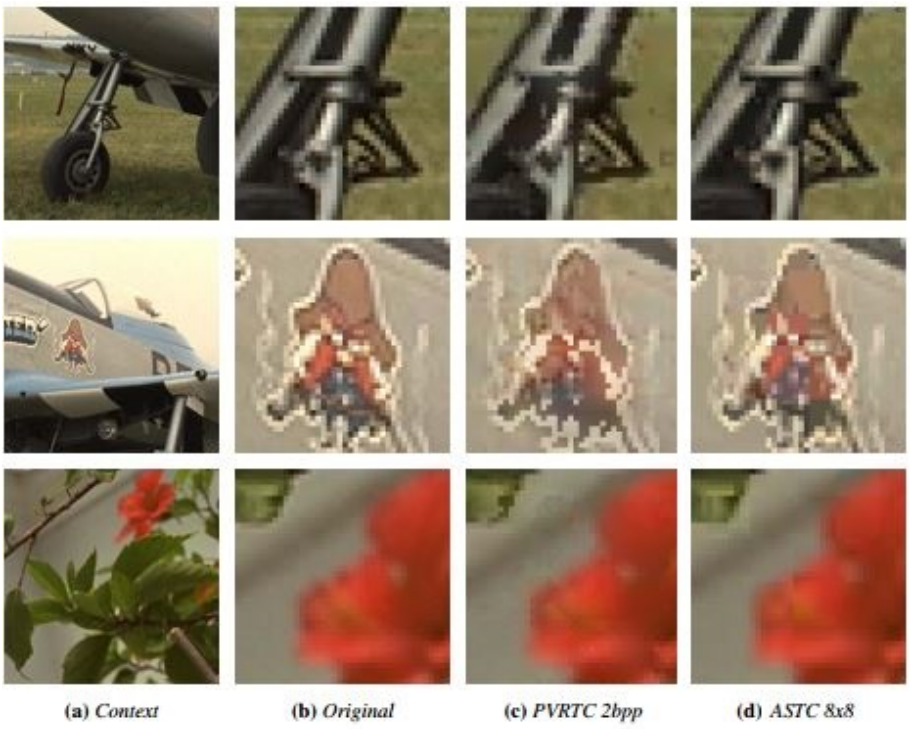


Figure 6: ASTC compared with PVRTC. Regard images (b) and (e) which depict the compressed original image (b), They both compress at 2-bpp and have similar quality. This image has been adapted from Krause et al.[13]

4 Conclusion

Our goal at the beginning of this review was to understand more about texture compression algorithms and find out which are used most widely in the gaming world and which may become popular in the future. We have identified numerous methods of texture compression. Based on what we have discussed it is clear there are considerable differences between those methods. Some are better at compressing textures but take a long time to decompress albeit in higher quality. Lossy techniques are extremely fast and have become ubiquitous in gaming. A huge factor in whether or not a texture compression technique should be used is how well it is supported in industry. Microsoft's massive support for DXTC ensures that it has a long term role in the gaming industry and the openGL equivalent S3TC is also hugely popular in the gaming world. However, in the future we may see other techniques such as the aformentioned Floating Texture Compression or Adaptive Scalable Texture Compression gaining popularity but it may be difficult for developers to switch to other techniques even if they are superior if there is a lack of hardware support from major companies.

References

[1] "ISPRS Journal of Photogrammetry and Remote Sensing", Bo Mao, Yiffang Ban, Volume 79, pp. 68-79 (2013)

[2] "Pyramidal Parametrics", Lance Williams, Volume 17, Number 3, Page2.

[3] "Real-Time Rendering - Second Edition", Haines, Eric, Akenine-Mller pp. 675-677 (2002).

[4] "Rendering from Compressed Textures", Andrew C. Beers, Maneesh Agrawala, Navin Chaddha, Stanford University.

[5] "Floating Precision Texture Compression", Phillip Klaus Krause, Goethe-Universitat, Frankfurt, Germany.

[6] "Texture Compression in Memory- and Performance-Constrained Embedded Systems", J. Ogniewski, A. Karlsson, I Ragnemalm, Linkpings University.

[7] "A new initialization technique for generalized Lloyd iteration", I. Katsavounidis,Zhen Zhang, Signal and Image Process. Inst., Univ. of Southern California, Los Angeles, CA, USA.

[8] "Ramamurthi, Bhaskar, and Allen Gersho. "Classified vector quantization of images." Communications, IEEE Transactions on 34.11: pp. 1105-1115. (1986)

[9] "On the Compression of Image Based Scene Rendering", Jin Li, Harry Shum and Ya-Qin Zhang.

[10] "Classification and Regression Trees", L.Breiman, J.H. Friedman, R.A, Olshen, C.J. Stone, Belmont CA; Wadsworth, (1984)

[11] "Texture Compression using Low-Frequency Signal Modulation", Simon Fenney, Graphics Hardware (2003)

[12] "Heirarchical Approach for Texture Compression", Anton V. Pereberin, M.V. Keldysh Institute of Applied Mathematics RAS, Moscow, Russia, pp. 1-2.

[13] "Adaptive Scalable Texture Compression", J. Nystad, A. Lassen, A. Pomianowski, S. Ellis and T. Olsen. High Performance Graphics (2012)

[14] "Real-Time Rendering", Haines, Eric, Akenine-Mller, pp. 137-138 (2002).

Anti-Aliasing Techniques in 3D Games Development

Joseph Sweeney

G.M.I.T

Abstract. One of the major problems games developers have when developing games is the image quality of their product. Aliasing is one area which developers look at to resolve. This literature review provides readers with an in-depth look at some popular anti-aliasing techniques for computer games development. Anti-aliasing techniques have been around for quite a while now but developers are still developing ways to advance already developed methods of handling aliasing problems. The reason they try to enhance the previous methods is for efficiency and quality purposes. Some developers want a specific technique to be more efficient and not use as much processing power or they might want to enhance the quality of the gaming images so the user has a better gaming experience. Different techniques may suit different needs when developing computer games and this is why there are a variety of different methods for resolving the problem. An image that is at a distance does not need an anti-aliasing technique as powerful as an image that is near by. I want to go into more detail with three types of anti-aliasing techniques which are (i) *Super-sampling Anti-aliasing (SSAA);* (ii) *Multi-sample Anti-aliasing (MSAA);* (iii) *Morphological Anti-aliasing(MLAA).*These techniques have been the gold standard when it came to anti-aliasing for games development.

1 Introduction

Aliasing is a term used quite often in computer games development. It is a common problem when developing computer games. It is used to describe the distorted or jagged appearance of computer generated images. This is due to the pixel colours on the screen and gives images edges a stair case effect. This problem really decreases the image quality and doesn't look pleasing to the eye. There has been many techniques developed over the years to combat this problem of aliasing and they call these techniques anti-aliasing techniques. The main reason I have chosen this topic for my literature review is to give the readers a better understanding of the different options and techniques that are on offer when it comes to dealing with aliasing. From reading many articles and journals I have picked out three types of anti-aliasing techniques that I found the most popular when it came to computer games development. In section 1 we discuss the super-sampling anti-aliasing (SSAA) method and how developers have taken different approaches to enhance its efficiency and quality. In section

2 discusses the Multi-sample anti-aliasing (MSAA) method and how different methods of anti-aliasing was born from the traditional multi-sample approach. In my final section 3 I will discuss Morphological Anti-aliasing (MLAA) and outline the advantages and disadvantages of this particular technique.

2 Super-Sampling Anti-Aliasing

The traditional Z-Buffer method was a popular image generation technique but it had problems when it came down to its output. It produced aliased images when outputted. [2] discusses an adapted anti-aliasing technique based on the super-sampling method. Super-sampling is a popular anti-aliasing technique. This technique is one of the most popular when developers are looking for the best image quality. The only problem that super-sampling anti-aliasing (SSAA) has is that it can be very slow and inefficient. It puts a lot of strain on the graphics card and uses a lot of video memory. This is the main reason that we have such a choice when it comes to anti-aliasing techniques because developers are constantly trying to advance the technique to enhance the efficiency for the users benefit. There are however adaptive methods of super-sampling ,[1] considers a different approach that enhances the super-sampling method. They look at how the different adaptations that effect the overall performance of the super-sampling technique.[2] also talks about a similar adaptive method of super-sampling which produces anti-aliased images that has minimal memory consumption.
The original super-sampling technique using the Z-buffer method stores the depth and colour values of each sub-pixel; this takes up a lot of the systems memory allocation. The main problem is that a majority of the super-sampled pixels are not needed because the aliasing problems arise around the edges and where different surfaces intersect. The adapted super-sampling technique is based around the Z-Buffer method. It uses far less memory to produce anti-aliased images. When a polygon is detected the method checks and declares if the polygon is covering the entire pixel or just partly of the pixel. If it is fully covered by the pixel then we sample pixel and if the polygon is just part covering the pixel we super-sample the pixel. A technique is used here so we can handle both types of resolution and it's similar to the A-Buffer (anti-aliased, area-averaged, accumulation buffer); this method basically breaks down a polygon into pixel fragments. A pixel fully covered uses a normal 2D buffer for the regular Z-Buffer scan (Sampled Pixels) and when the edges of a polygon are encountered a sub-pixel sample is stored in a memory block. [2] shows a figure and how the traditional super-sampling differs from the new and improved super-sampling technique.

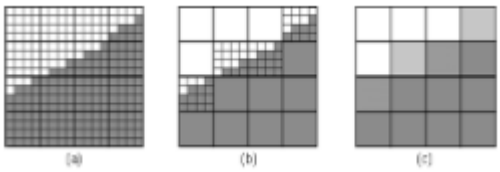


Figure 1: [2]
(a) Traditional Super-sampling method.
(b) Adapted super-sampling method.
(c) The result of both methods.

[3] shows that with the A-Buffer there are two questions asked. The first being the number of edge pixels in an image and the second question being the number of fragments in the edge pixels. This technique can deal with the surface intersection by using the sub-pixel level. [3] pares the memory usage of the four methods mentioned above which are the A-Buffer, Z-Buffer, super-sampled Z-Buffer and the adaptive super-sampling buffer.

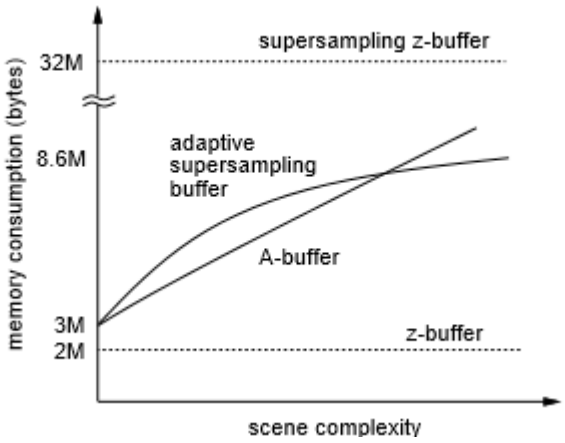


Figure 2: From [3]
A comparison of the memory consumption.

From this memory test you can see how all four methods differ from each other. You can see that the Z-Buffers performance are independent of the scenes complexity. The A-Buffer increases the more complex the scene is. The adapted super-sampling method however performs the best out of the four methods- It gradually approaches to a constant value no matter how complex the scene.

3 Multi-Sample Anti-Aliasing

Multi-sample Anti-aliasing (MSAA) has been a popular method for anti-aliasing over the years. It has a solid balance between efficiency and also the quality of image that is outputs. While super-sampling anti-aliasing (SSAA) gathers the same amount of samples from each resource. [4] gives us an illustration of MSAA that

takes four samples per pixel and it computes each samples depth and value but each sample receives the same colour samples at the pixels centre. The shaded computation of each sampled pixel is only calculated once. This saves valuable processing power even if it uses the same amount of memory. [5] remarks that although MSAA offers very high quality anti-aliased images in real-time it uses a huge amount of memory. They go on to say that the majority of alternative approaches cannot compete with the image quality outputted by MSAA but their approach offers quality images at 16x MSAA (which is a gradient quality) and it only uses minimal memory and less time consumption. This method detects borders and tries to find patterns. They then begin to blend the image by changing the pixels colours of the edges of the image this gradually blends the image edges. In [4] they discuss multiple anti-aliasing techniques that have been inspired by MSAA.

1 Coverage-Sampled Anti-aliasing (CSAA)

2 Sub-Pixel Reconstruction Anti-aliasing (SRAA)

3 Directionally-Adaptive Edge Anti-Aliasing (DAEAA):

Coverage-Sampled Anti-aliasing (CSAA) uses MSAA and combines this method with extra pixels to better capture the pixel. Unlike MSAA not all of the sampled pixels capture the colour and depth of the pixel. The samples not measuring the colour and depth capture the fragment coverage. With this technique the image quality is improved because it has a better scope of the pixel area. Moreover, by using CSAA we can accurately determine how to colour each individual pixel. The main advantage of CSAA is that it uses far less memory that MSAA. One disadvantage [4] discusses about is that the coverage samples could be overridden because they are not subject to a depth test. Sub-Pixel Reconstruction Anti-aliasing (SRAA) is another technique that was inspired by MSAA. [6] explains how SRAA was developed and why it was developed- because MSAA was so successful developers wanted to expand on this specific technique. SRAA is a post process operation. SRAA was designed for real-time applications and specifically for games. This technique super-samples the geometry and depth. They then use these calculations to fill the gaps among pixels. [4] simply describes this technique by saying that it builds a virtual super-sampled image and then down-samples to an output resolution. In figure 3 you can see how the two techniques described where developed around the same idea as MSAA. We can see clearly from [4] diagram below how these three techniques that we discussed work and how they compare and contrast with each other. Directionally-Adaptive Edge Anti-Aliasing (DAEAA) is a technique that consists of a Multi-sampled anti-aliasing technique that has improved image filtering in the final steps of the MLAA technique. DAEAA's image quality compares to MSAA when its 4x DAEAA then it is compared to 16x MSAA this is after four full screen passes.

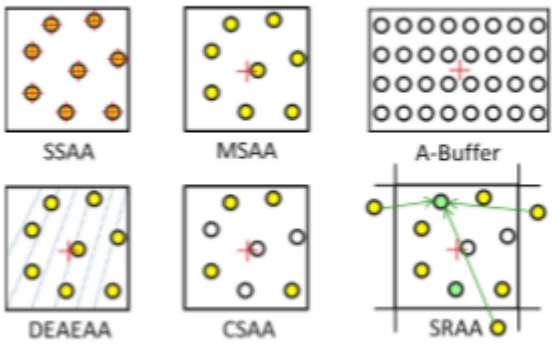


Figure 3: From [4]
MSAA with 8 colour and coverage samples and central shading per pixel. CSAA with 4 colour and coverage samples and 4 coverage-only samples per pixel. SRAA with 4 colour and coverage samples, 2 geometry samples, and the colour reconstruction from other samples. DAEAA with 8 MSAA samples to estimate isolines passing through the pixel.

4 Morphological Anti-Aliasing

I wanted to review Morphological Anti-aliasing (MLAA) because this technique was the reason a lot of techniques were developed. MLAA gave birth to a multiple of real-time anti-aliasing techniques that gave the gold standard techniques some competition like the other two techniques that we discussed earlier, supersampling(SSAA) and multi-sampling (MSAA). [5] describes how MLAA belongs to a family of data dependant filters that allows anti-aliasing at a post processing step. This technique identifies discontinuity patterns and then tries to blend the patterns colours. In [4] they discuss the reason why MLAA was developed . MLAA minimises aliasing of images from the images edges and silhouettes as seen in figure 5 below. The technique removes aliasing from ray-tracing-generated images. Step by step MLAA processes the image buffer in three steps :

* First it finds discontinuities between the pixels of the image
* Second it finds U-Shaped, L-Shaped and Z-Shaped patterns
* Finally it blends the neighbourhood of the patterns

[7] explains how the Z-Shapes and also the U-Shapes can be split into two L-Shapes in the final step of the algorithm. Figure 4 below is a diagram from [4] and gives us a better understanding of the U, L and Z shaped patterns that MLAA detects in the second step above.

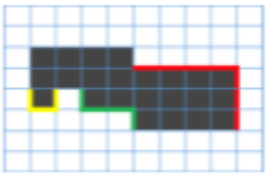


Figure 4: From [4]
Examples of U(yellow),Z(green) and L(red)shapes detected in MLAA technique.

In Figure 5 below is a step by step diagram of how MLAA works.

Figure 5: From [5]
morphological anti-aliasing improves the quality of the rendered image without having a significant impact on performance. The algorithm uses separation lines falling between perceptually different pixels to infer silhouette lines and then blend colours around such silhouettes.

[5] Gives us the key advantages when it comes to MLAA.

* The algorithm is universal and can run on minimal data pixel colours.
* Its image quality is as good as 4x super-sampling
* It is independent from the rendering pipeline, allowing efcient implementation on modern hardware.

With MLAA it can quickly reduce aliasing of ray-traced images because it only needs to use the colour buffer. [4] talks about MLAA's disadvantages or limitations. MLAA does not have the ability to handle sub-pixel features for example blurs of edge pixels with no aliasing. When [7] discusses about MLAA's weak points they point out that as a post-processing technique MLAA is unable to handle small scale geometry aliasing. This leads to problems for example flickering animated artifacts. Furthermore, [8] also explains how MLAA has limitations in sub-pixel features. They also discuss another limitation of MLAA and this is its handling of border pixels. This is because they do not have much information about the neighbourhood beside the edge pixels. Another disadvantage of MLAA would be that is can mangle small text. Below is an example of step 3 of MLAA.

Figure 6: From [7]
Example of Step 3 in MLAA.
"In MLAA, the bottom red pixel blends with the top red pixel weighted by the area of the yellow trapeze. The technique consists in detecting L shapes in green and blending pixels along the green triangle with pixels oppos- ing the L."

[7] gives us an insight into a modified version of MLAA called Practical Morphological Anti-Aliasing (PMLAA). This technique works the same as MLAA and has the same three steps but the steps are improved. The step when detecting edges masks the pixels requiring anti-aliasing and this avoids unnecessary processing. The blending step makes use of the fast filtering offered by the GPU hardware. When compared PMLAA is far more efficient but still maintains MLAA's original limitations and may cause excessive blur in sharp corners.

5 Conclusions

This literature review presents us with a broad range of anti-aliasing techniques. At the start the concept of anti-aliasing seems like a simple idea, but the further we dug, the more and more topics I found that could be discussed in deeper. We now know of a broad range of anti-aliasing techniques. We know how some

techniques might output the best quality images but with that they have a very inadequate performance as they use up a lot of processing power. This problem triggered a majority of developers to start developing different techniques to give us the most efficient technique in anti-aliasing. When reviewing these different techniques we have learned that some methods of anti-aliasing did not impress from a image quality point of view, its performance may have been high but the whole idea of anti-aliasing is to get the best quality image onto the screen. With different textures we have learned how different anti-aliasing techniques can have different results on these textures. Some showed positive results and some not so much and this was a big issue when reviewing this topic. We also learned how even some of the gold standard anti-aliasing techniques even have limitations but still hold their ground in being the most popular of anti-aliasing techniques, as they have a balance of image quality and also performance. From reviewing a multitude of anti-aliasing techniques we learned how each anti-aliasing techniques has their own strengths and weaknesses. With MLAA we we can see when this technique was tested that it does a reasonable job of cleaning up aliasing artefacts on some transparent textures. This technique can be a very effective method of anti-aliasing but it would best suit larger images or text because it struggles with image quality when small text is involved in the gameplay. SSAA was shown to output the best quality by any of the anti-aliasing techniques but it does take a major performance hit and from reading these journals they seem to wonder if it is worth the performance hit. Finally with MSAA we learned how it gave birth to a multiple of different techniques that based their methods around the original version of MSAA.

Bibliography

[1] “A parallel rendering approach to the adaptive super-sampling method. ”, SAM LIN, RYNSON W.H. LAU, XIAOLA LIN, P. Y. S. C.,(1997).

[2] “ An anti-aliasing method for parallel rendering. ”, Lin, S, Lau, R.W.H, Cheung, P.Y.S.,p:228235.

[3] “ An Adaptive Supersampling Method. ”, Lau, R. W. H.,p:205214.,(1995).

[4] “ Transparency and Anti-Aliasing Techniques for Real-Time Rendering. ”, Maule, M., Comba, J. L. D., Torchelsen, R., and Bastos, R.,p:5059.,(2012).

[5] “ Filtering approaches for real-time anti-aliasing. ”, Jimenez, J., Lottes, T., Malan, H., Persson, E., Andreev, D., Sousa, T., McGuire, M.,p:1329.,(2011).

[6] “ Subpixel reconstruction antialiasing for deferred shading. ”,Chajdas, M. G., McGuire, M., and Luebke, D. .,(2011).

[7] “ MORPHOLOGICAL ANTIALIASING AND TOPOLOGICAL. ”

[8] “ Morphological Antialiasing. ”, Reshetov, A .,(2003).

4 Unity 3D Interface

Unity is a powerful engine with a variety of tools. The editor is easily customised and this gives you plenty of freedom to pick and choose what you want in your work space. There are five main views you can have in your editor. Project view, Scene view, Game view, Hierarchy view and Inspector view. We will go into more detail about all these views below:

4.1 Project view

The project view is a very important part of every unity project. In this view you store all your assets that make up your game. It is just like a directory on a computer where you have a hierarchy of folders that hold different types of information. In the project view you can store assets like scenes, scripts, prefabs, 3d objects, textures and audio clips.

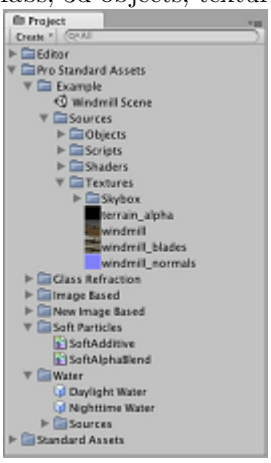


Figure 1. Project View Example

The easiest way to import an asset into your project view would be to go to Assets then Import new Asset. When you look through your project view you will see your new asset included in your project view and you can use it in your game.

While we where developing our 3d adventure game I noticed a slight problem while importing 3d graphics. The scale of the 3d models would be scaled down to 0.1 while they should be scaled at 1. If you where not aware of this problem you would think that the 3d object was not there but it is just very small and you have to manually adjust the scale by finding the asset in the project view and changing the scale of the object. You can create assets very easily in the project view by clicking on Create this will give you a drop down list of every asset that you are able to create.

Note: Do not move assets around using folders in your operating system because It will interfere with the meta-data associated with the specific asset. If you need to move assets to different folders just use the project view in unity to do so.

4.2 Scene View

The Scene View is where you can interact with your game environment and everything that is inside of it. The Scene view makes it easy to position game objects for example the player, camera, buildings etc. When you are developing games you want to be able to move objects around quickly without any hassle and unity's scene view makes it very easy to manipulate every kind of game object.

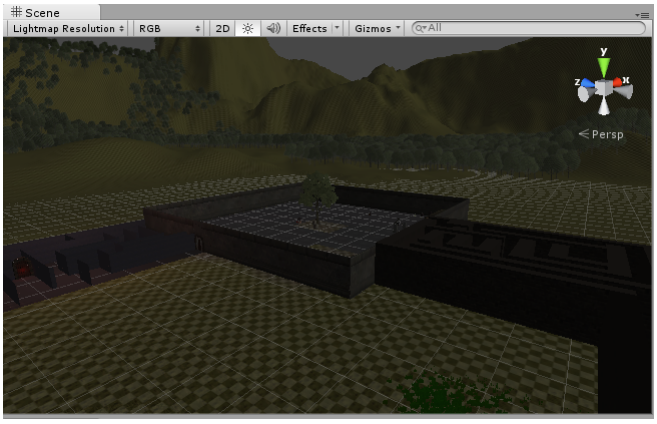


Figure 2 . Scene View example.

In the scene view there are an abundant of different tools that let you navigate through the scene view. There is a toolbar that lets you easily navigate through the scene.

Figure 3. Navigational toolbar.

In the upper right hand side of the scene view there is a gizmo which you can toggle with. This is for the cameras orientation and lets you modify the viewing angles. Each arm is colored on the gizmo and this represents the different axis.

Figure 4. Gizmo to manipulate camera angle.

4.3 Hierarchy View

The Hierarchy view is basically a view of every object that is currently in the scene of your game. You can drag assets from your project view directly to

the hierarchy. If you deleted an object from your scene view you would see it disappear from you hierarchy also. A very useful aspect of the hierarchy view is its ability to parent game objects. By doing this the child objects can inherit all the functionality of its parent object. Any object which has an arrow next to it can be expanded to show its child objects. The hierarchy view is also very useful when you want to find a specific object in the scene view. By clicking on the object you want and then putting your cursor over the scene view and pressing the F Key the scene view will zoom in on the object you selected.

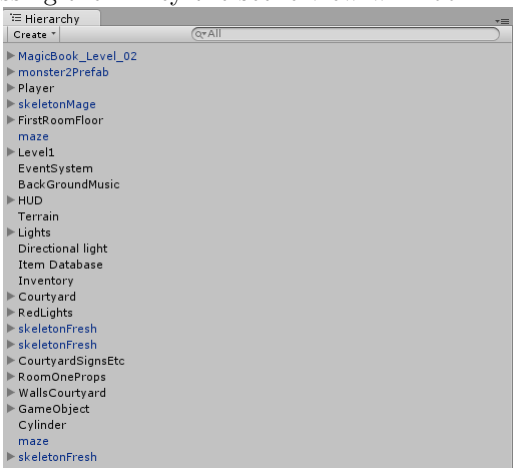


Figure 5. Hierarchy View Screen Shot

4.4 Game View

The game view is the view you will have as you play the game. This is the position the player will be in when you start the game. There is a play button at the top of the unity environment that will direct you to the game view and this runs your game. The game can be started, paused and restarted and this helps when you are testing the game for bugs. There is a log file attached to the game view that will show debugging messages this is very useful when you want to find any bugs that are in your scripts.

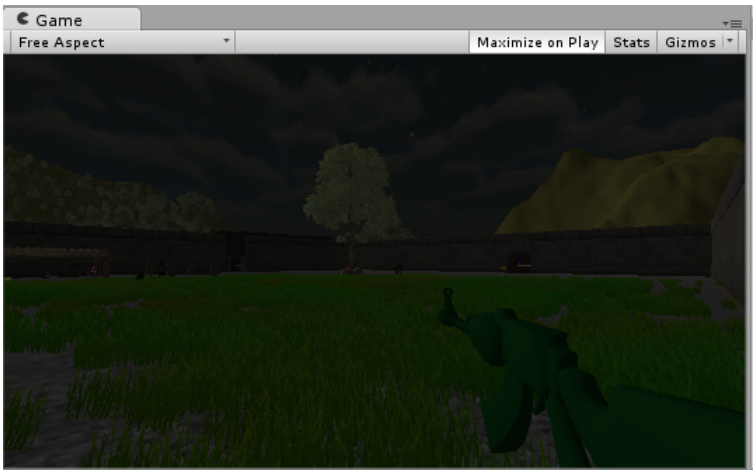


Figure 6. Game View Screen Shot

4.5 Inspector View

Every game object that is in your scene has different scripts, meshes, audio clips and other elements that are linked to that specific game object. The inspector gives you a detailed view of what the current game object has. It is in the Inspector where you can modify you game objects functionality. Any property that is displayed in the inspector view can be modified. Even if you wanted to change variables in a script that is in the inspector view of a game object you can easily change the variable name from the inspector without going into the script itself.

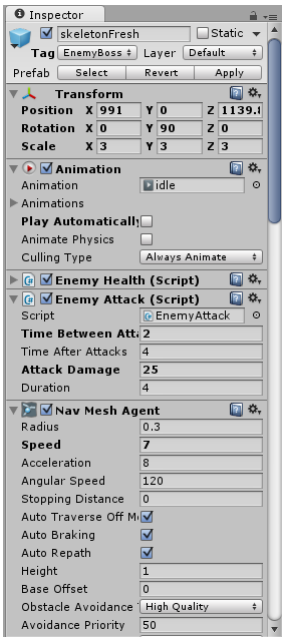


Figure 7. Inspector View.

5 Unity 3d deployment capabilities

Deploying a Unity game on Windows, Mac or Linux is very simple. There are no specific requirements or extra downloads needed when deploying to these three platforms, Unity 3d does all the required compiling for them. However for android , IOS and windows 8 devices there are required downloads. SDK's are needed for android, Xcode for IOS mobile devices and Windows SDK's for windows 8 devices. We will go into more detail about the requirements for deploying a unity game on android, IOS mobile devices and also Windows 8 devices.

Deploying a Unity game on Windows, Mac or Linux is a very simple. There are no specific requirements or extra downloads needed when deploying to these three platforms, Unity 3d does all the required compiling for them. However for android , IOS and windows 8 devices there are required downloads. SDK's are needed for android, Xcode for IOS mobile devices and Windows SDK's for windows 8 devices. We will go into more detail about the requirements for deploying a unity game on android, IOS mobile devices and also Windows 8 devices.

5.1 Deploying a Unity game to an Android device

As mentioned above the android SDK is needed if you want to deploy your game for android. You can find an android SDK preferably the most current

one online. Once you have downloaded the SDK the update manager needs to update any packages that needs to be updates. When the update is complete you need to add the path of the SDK to the build properties that is in Unity. To do this you need to go to Edit Preferences External Tools . You will then see where the android sdk's path has to be entered.

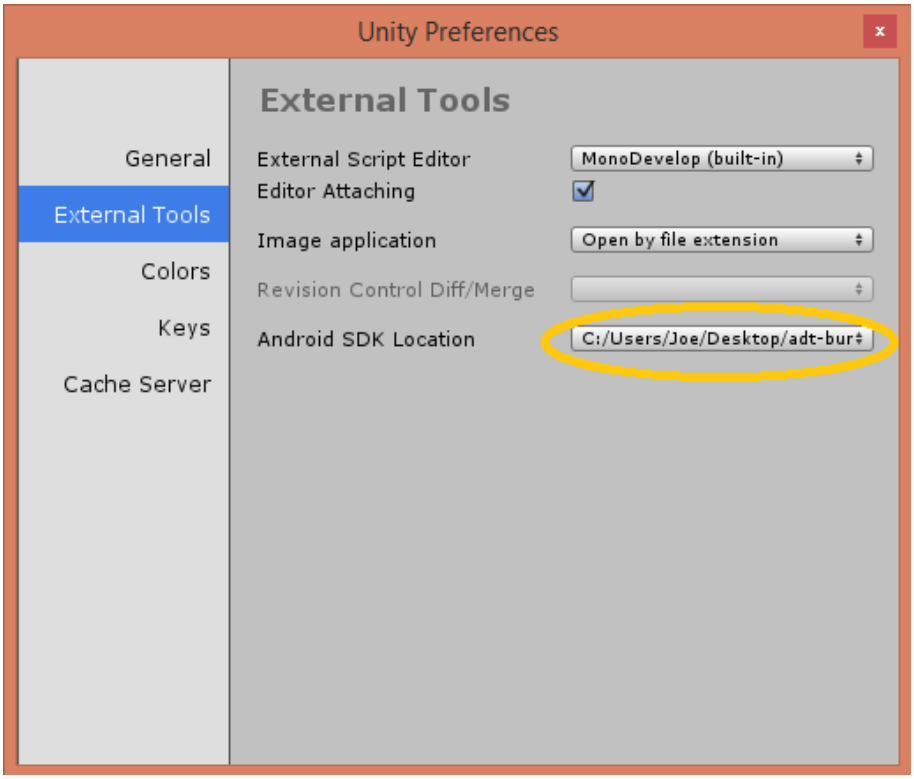


Figure 8. Adding SDK path

As mentioned above the android SDK is needed if you want to deploy your game for android. You can find an android SDK preferably the most current one online. Once you have downloaded the SDK the update manager needs to update any packages that needs to be updates. When the update is complete you need to add the path of the SDK to the build properties that is in Unity. To do this you need to go to Edit then Preferences then External Tools. You will then see where the android sdk's path has to be entered.

5.2 Deploying a Unity game to an IOS device

Deploying a Unity 3d game for devices like the iPhone and iPad is a more difficult job than deploying for desktop PC or even android. Unlike the PC market the target hardware for iPhone and iPad is standardized and not as powerful as a desktop computer or laptop. For this reason you will have to deploy your game a little different for IOS devices. The first job you need to

do is to set up your very own apple developer account. All this can be done on apple's developer website.

When you have successfully set up your apple account and it has been approved you can then proceed to the next step which is generating the Xcode project. When you build the Unity IOS game the Xcode project is generated for you. The Xcode project will be inside a folder in your project. This project This project is required to sign, compile and prepare your game for distribution.

Deploying a Unity 3d game for devices like the iPhone and iPad is a more difficult job than deploying for desktop PC or even android. Unlike the PC market the target hardware for iPhone and iPad is standardized and not as powerful as a desktop computer or laptop. For this reason you will have to deploy your game a little different for IOS devices. The first job you need to do is to set up your very own apple developer account. All this can be done on apple's developer website.

When you have successfully set up your apple account and it has been approved you can then proceed to the next step which is generating the Xcode project. When you build the Unity IOS game the Xcode project is generated for you. The Xcode project will be inside a folder in your project. This project This project is required to sign, compile and prepare your game for distribution.

5.3 Deploying a Unity game to an Windows device

Currently if you want to develop and build a windows store app you will have to develop it on windows 8 or windows 8.1.Unfortunately you cannot use Mono but instead we use .Net together with WinRT and this allows you to code your scripts using visual studios. The windows store app targets that unity supports are ARM, X86 and X64. Before you can proceed to developing and deploying your game for windows you first have to create a windows 8 developers account. You can create a developers account by going to microsoft.com.

6 Game Mechanics of Unity

Today's games all have similar game mechanics that lets you trigger different objects to do specific functions. Collision detection is a very important mechanic in games development. These game mechanics have been around for some time now and have been improved throughout the years.

6.1 Collider Mechanics

When creating objects in the unity game engine we can add components to the specific game object that will let you manipulate the object. The collision detectors can be added to the inspector view. The colliders are located under the physics tab.

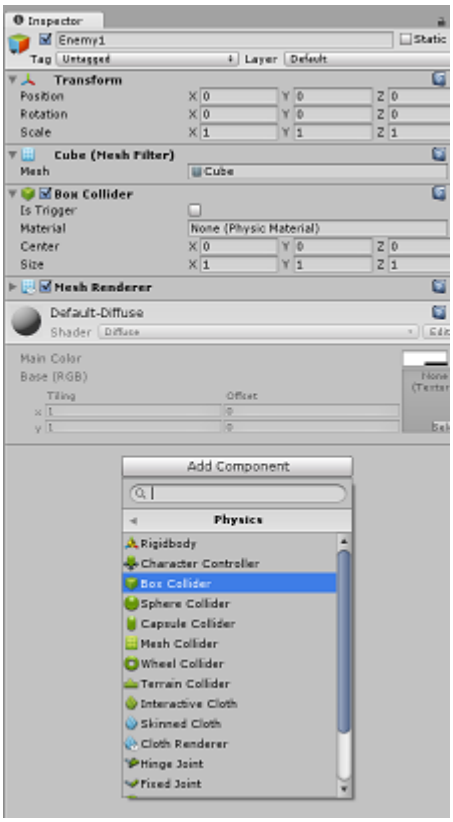


Figure 9. Adding a collider to game object

The most popular and most used collision detectors are box, sphere and capsule collider's. The box collider has a cube shaped collision area. You can toggle with the box's size and center using the inspector the collider is added to. Box colliders are used for objects with sharp edges like doors windows etc. The sphere collider is very similar to the box collider, it has a spherical shape rather than cube shape. You can toggle with the radius of the sphere collider to suit the size of the object. You can use this type of collider for rounded objects like footballs etc. And finally the causule collider defines a capsule volume for collision. You can toggle with the height and axis of the capsule. You would use these types of colliders for character game objects. The collider size and volume does not have to correspond with the game objects it can be bigger so that the trigger will happen when in the colliders area. The colliders are triggered when some other object has entered the collision detectors area. If you are using the collider as a trigger area you need to use the OnTriggerEnter method.

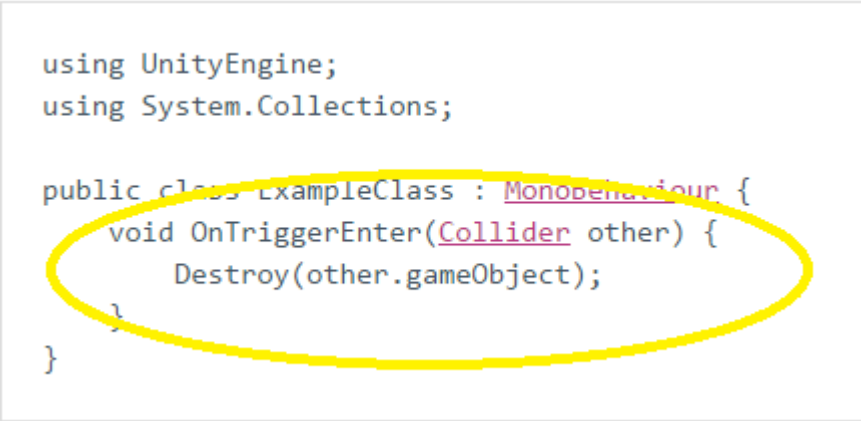


Figure 10. OnTrigger method example.

You can also check to see what object it is that triggered the collider and this way you can have different outcomes for different objects.

6.2 RigidBody Mechanics

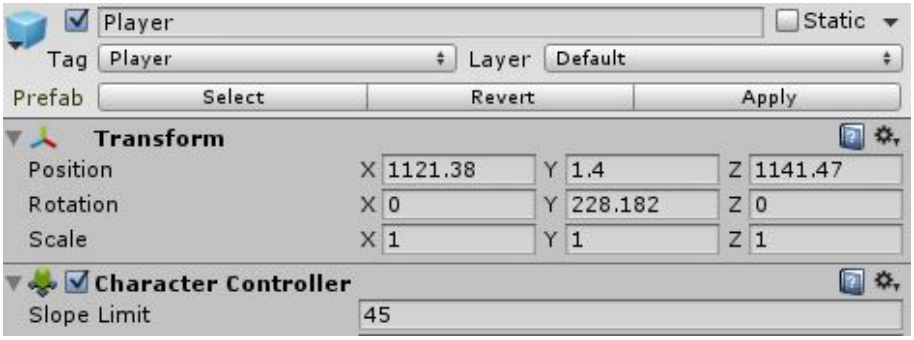
To have an object react to a physical collision from another physical object you will need to add a rigidbody component. By adding a rigidbody to your game object the object will then be influenced by gravity and external forces.

If you want your object to react to physical collision with other objects and the game world, you'll need to add a RigidBody component. A game object with a rigid body will be influenced by gravity and external forces. You can customize your game object if you want a discrete or continuous collision detection. The reason you would use a continuous collision detection would be for fast moving objects. Discrete collision detection may not detect some collisions and by default this is set to discrete as a continuous collision detection could greatly effect your games efficiency.

7 Unity scripting API

This section will explain how scripting is implemented in Unity using excerpts from the scripts we used.

The Unity API provides numerous interfaces which will be discussed later. The main reason for these interfaces is to help to connect scripts to the inspector window in the Unity IDE so that it's easy to attach objects such as 3d models and sounds to the variables declared in the script. They also provide us with a large amount of resources with which to control all of the game elements while the game is running. This image is from our games inspector and depicts two of the components we've added to our Player object in our heirarchy. The transform component exists to give the Player a position in 3d space and the Character Controller component is the one that we apply behaviours to.



The line below references the CharacterController class we added in the Unity editor. We then instantiate various float variables such as 'mouseSensitivity' etc. to manipulate the character during gameplay. These variables can later be adjusted in the editor window.

```
CharacterController characterController = GetComponent<CharacterController>();
```

We manipulate these variables in the Update() method which is called every frame. Below is a number of lines we used in the FirstPersonController.cs script to manipulate the GameCharacter's location in 3d space via mouse input. They allow the mouse to move the character's view up and down a certain amount before stopping. The Quaternion.Euler line controls movement in the (Z,X,Y) directions ie., it controls the rotation of an object in 3d space by rotating it x degrees on the x axis, y degrees on the y axis and z degrees on the z axis.

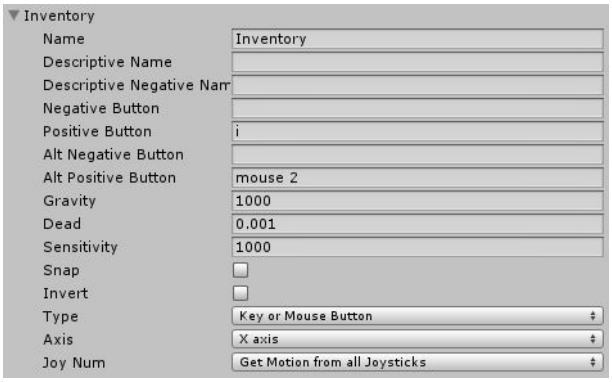
```
rotUpDown -= Input.GetAxis ("Mouse Y") * mouseSensitivity;
rotUpDown = Mathf.Clamp (rotUpDown, -upDownRange, upDownRange);
Camera.main.transform.localRotation = Quaternion.Euler (rotUpDown, 0,0);
```

Unity allows you to assign keys globally through the Unity editor and they can then be accessed through the script. The image below is a screenshot of the Inspector in the IDE where we set up our inventory button.

The 'name' field is how we reference the input in our script and the 'Positive Button' text box is where we define the key that must be pressed to use it. When the game is running, the user is able to press 'i' on their keyboard to draw the inventory system on their screen. The code underneath shows how we invoke the 'Jump' key.

```
if (characterController.isGrounded && Input.GetButtonDown ("Jump")) {
verticalVelovicty = jumpSpeed;
forwardSpeed = (Input.GetAxis("Vertical")*movementSpeed)/2;
}
```

All mouse inputs are handled by Unity API's Input interface which provides Input.GetAxis and so on so that the programmer doesn't have to decide how



to interpret mouse movements by the user, instead they have to decide the limitations of these inputs, controlling the mouse's sensitivity, how they affect the game character etc. After you understand how to read user inputs you can begin coding the actions of the game character. As our game runs in first person mode, we are mainly using mouse inputs to control the position of the camera in real time, thus simulating the standard movements of the characters' head and body. For example: If you use the mouse to make your character look up, the character will do so in real time but then stop movement at 60 degrees to simulate the limitations of a human being tilted back, otherwise the character could tilt backwards to 360 degrees. We used Mathf.Clamp(Sets a maximum and minimum float value and clamps a third value within that range.) to achieve this. In the following way:

```
rotUpDown = Mathf.Clamp (0, -60, 60);
```

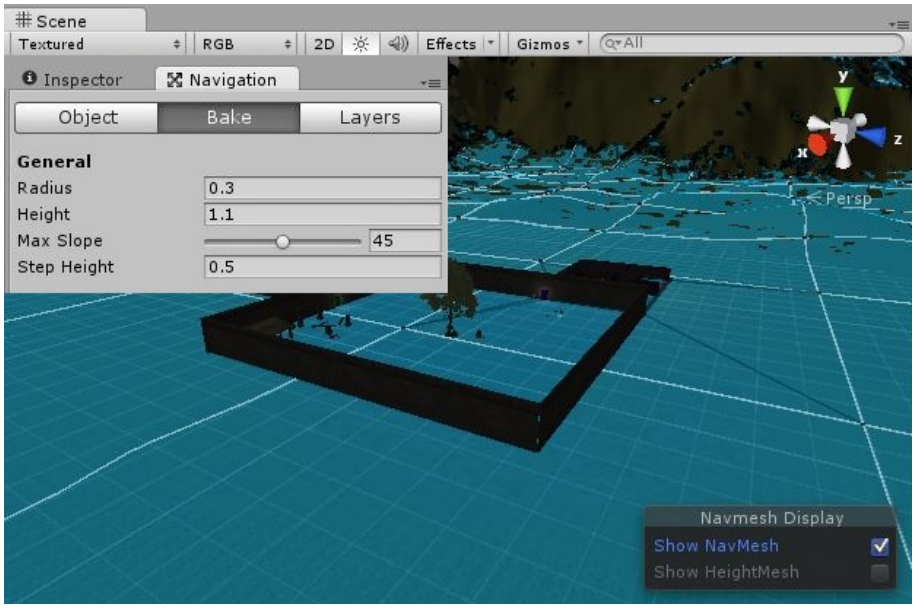
We also control the character's walking speed, jumping power and so on through our scripts. The full details of their implementation can be found in the FirstPersonController.cs script in the scripts section.

As the game's enemies aren't controlled by user input there needs to be a different way to move them arount the 3d environment. To do this we add a 'NavMeshAgent' to the enemy character in the Unity IDE and then in the script we initialise the NavMeshAgent and decide where to move the enemy. In our case the enemy stays in its position until our character is in range and then it moves towards the the player's position.

```
//Creating NavMesh Variable
NavMeshAgent nav;
//Initialising in the Awake();
nav = GetComponent <NavMeshAgent> ();
//Moving to player in Update();
nav.SetDestination (player2.position);
```

Once the NavMeshAgent has been added and the script dealing with movement has been attached, the next step is to define what areas in 3d space you

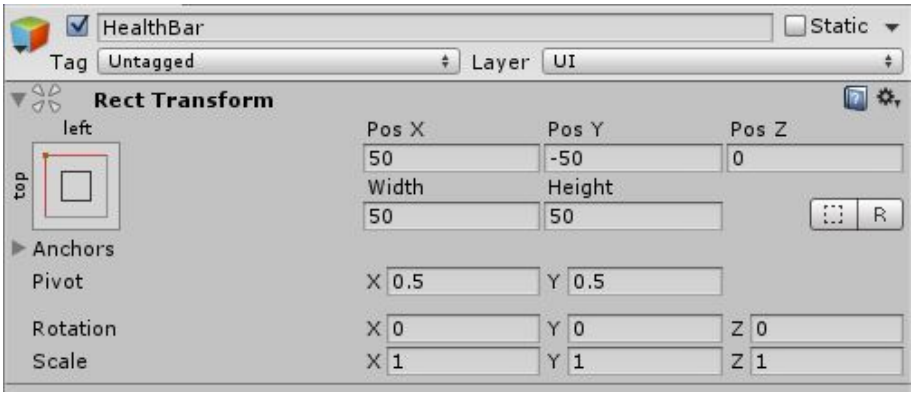
want to be traversable by the enemy characters. You do this in the navigation window of Unity. The step height will control how high a step the enemy can overcome and the max slope will decide how steep a hill the enemy can climp. If the Max Slope was 90 degrees for example the enemy would be able to scale walls. When you have adjusted your settings appropriatly; the final step is to 'Bake' the scene, a process which can take between a few minutes to a few hours depending on you terrain size. The blue areas in the diagram below are walkable. The green areas in the background hills are not covered by the NavMesh therefore their slope must be greater than 45 degrees, so the characters cannot walk the background.



7.1 User Interface

The UI system is relatively new to Unity, with it only being added to version 4.6. Previous to it's inclusion OnGUI was the only way to add a User Interface and this requires a lot of coding, it also makes it extremely difficult to design aesthetically appealing menus etc. the new UI sytem is much more forgiving when it comes to design and implementation. For example, the text boxes and inventory system in our game are created using the legacy(OnGUI) system and we struggled to display the information we wanted, whether it was text or icons in the correct screen position or at the right size for varying screen sizes. In contrast, the GameCharacter icon and Health Bar which appear on the top left side of the screen were easier to implement and look better and there was virtually no coding. The health bar is simply a 'slider' that is common to any software that deals with the creation of forms and it's position on the screen is

simply defined in the Unity inspector as in the diagram below.



The position of our inventory had to be defined through our C-sharp script. This involved a lot of trial and error to see which positions actually looked best on screen.

```
Vector3 GeneratedPosition(){
int x,y,z;
x = 100;
y = 25;
z = 120;
return new Vector3(x,y,z);
}
```

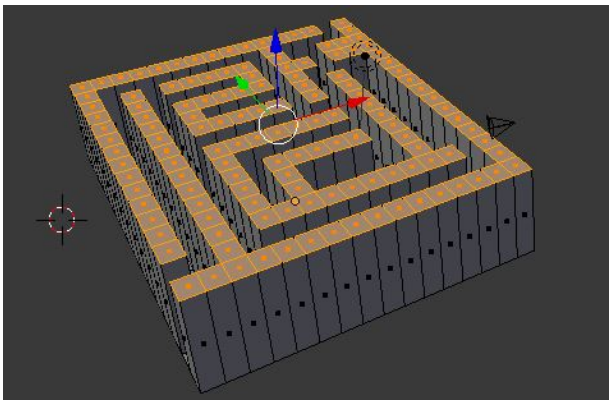
7.2 First Person Controller

7.3 Textures in Unity

The 3d game we developed was initially going to be deployable on android devices. But as the game progressed it became increasingly complex for a mobile platform. Texture rendering, as we expected, is the largest constraint we have when considering deploying on different hardware. Unity supports many different types of compression, However ETC1 is the only one guaranteed to work on all android systems. The problem with ETC1 however, is that it does not support an alpha channel. Therefore it can only be used for opaque textures. Given that our project contains a large number of transparent .PNG images, this texture compression is unsuitable. There are various other types of compression that Unity can use but they are only supported by specific devices, for examplec PVRTC(PowerVR texture compression) is used in devices such as the Nexus One that have an Adreno GPU, S3TC is used by the NVIDIA chipset and so on. This would require seperate consideration for each device and extensive testing across those platforms. Unity has extensive documentation on how to perform Per-Platform Overrides on it's website. You can change change

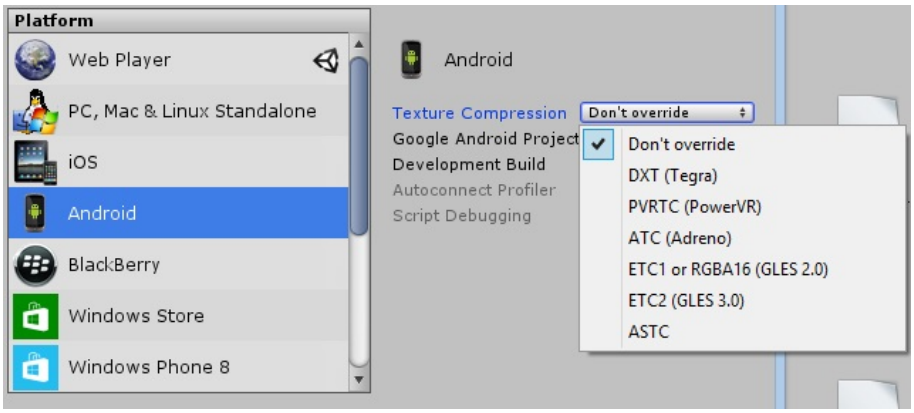
your android manifest to accomodate multiple .apk files if you want to generate them in different texture compression formats and allow the android market's filtering system to select the correct one for you. If you ignore doing this, and compress in DXT or similar and the device doesn't support it, it will automatically decompress all the textures into standard RGB(A) format at runtime. For a game that has as many textures as the game we developed this would be a catostrophic result as the system's GPU might not be able to render the textures on time and cause lag or dropped frames. An image compressed with DXT5 RGB(A) has a memory consumption of 1 bpp while a standard 16bit RGB(A) consumes twice as much memory(2 bpp) and a 32bit RGB(A) is double that again(4 bpp).

The formula for image storage size is the width of the image multiplied by the height and bits per pixel(w*h*bpp). We then add a third of that amount to any given texture as we are using mip-maps, which even though they help performance while the game is running, rely on stored smaller copies of the images so that images are only drawn in full quality when they are close to the player's position.



The majority of compression occurs on import with Unity, i.e any textures or models created outside of Unity can be manually compressed when you are

importing them to the game's assets folder. We used a 3d rendering software called 'Blender' to create the maze, we exported it directly into Unity's assets folder with generic textures and then decided how it would be compressed using the 'Import Settings' window in Unity.



Mip-Maps can be seen throughout the game as it is running. Especially on the walls of the courtyard and maze. For instance if you are looking at one one these walls from a distance, they appear blurry and the detail isn't sharp but as you approach they become clearer as the higher quality textures are loaded from memory.

7.4 Unity graphical quality

The Unity game engine allows you to customize the graphical quality on your game. With a greater graphical quality you are sacrificing the efficiency of the game. Depending on the type of hardware you have on your device that will also have an effect on what types of graphic quality you may want to use. The Unity game engine has a quality settings inspector where you can select the quality level for different platforms as show in the figure below.

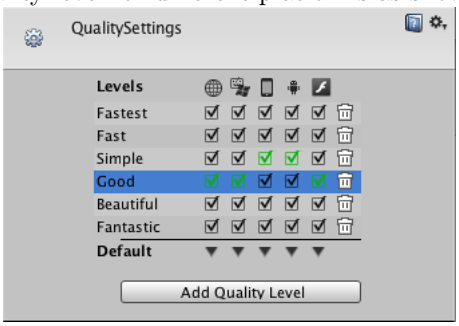


Figure 11. Quality settings example.

So from the figure above you can see how easy it is to choose what kind of quality you want for each platform. If you click on a quality setting like simple,

fast, fastest etc it will let you edit that particular quality type. The editing is done in the panel below.

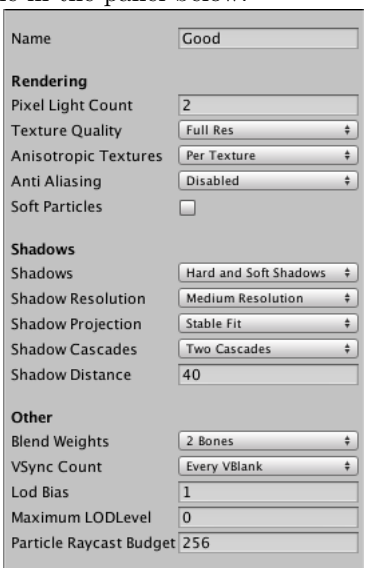


Figure 12 As you can see it is very easy to customize the quality of your game. You can even disable some of the rendering options if you are more interested in the quality of the game play rather than a high quality image.

7.5 Anti aliasing in Unity

Anti aliasing is one of the rendering options that you can customize to your liking. Anti aliasing is a technique that diminishes jaggies or in other words the stair step effect of round objects. Anti aliasing gets rid of these jaggies by surrounding the stair step with an intermediate color between the object color and the background color. This takes the pixelated look off of the images.

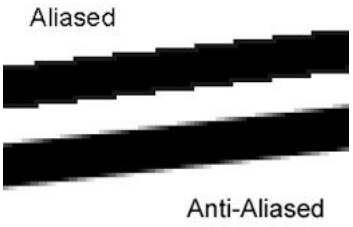


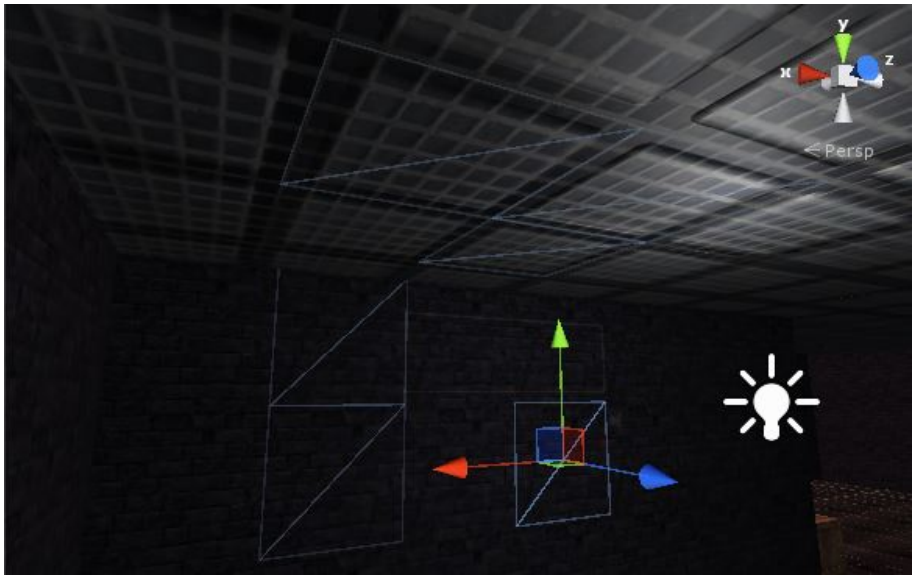
Figure 13

The level of anti aliasing in your game will determine how smooth your game objects will be. However using anti aliasing does incur a performance cost and can put a lot of strain on the graphics card and video memory. Unity supports a range of anti aliasing techniques which include NVIDIA's, FXAA, FXAA 2, FXAA 3, NFAA, SSAA. From these techniques SSAA or super sampling anti aliasing is the fastest followed by NFAA or normal filter anti aliasing and then

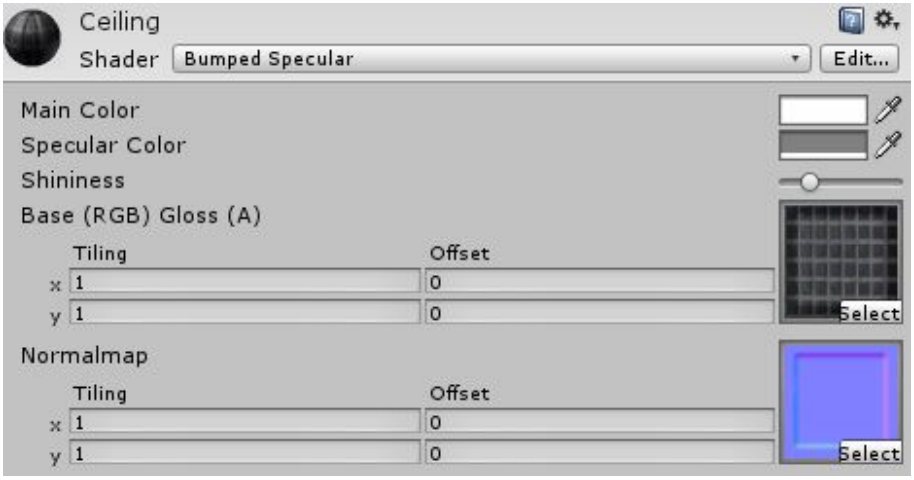
FXAA 2 which is short for fast approximate anti aliasing and so on. Usually the better the anti aliasing technique the slower the speed but that's where a good choice of technique can be beneficial. For a well rounded anti aliasing technique FXAA 3 has a great balance between quality and speed and most games developers would choose it.

8 3d Model Creation in Unity

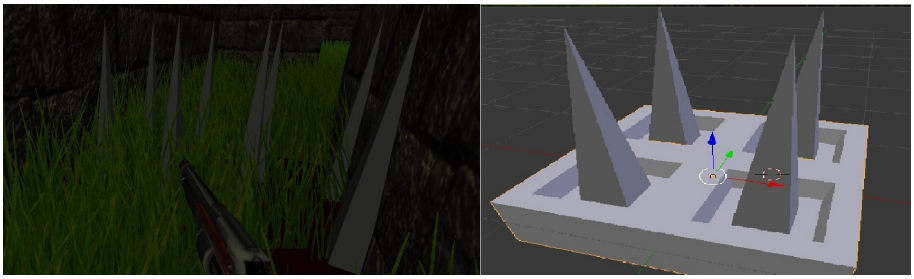
While some of the models used in our game were downloaded from https://www.assetstore.unity3d.com/ and http://www.blendswap.com/, we tried to understand the basics of creating these models ourselves. We started by making a room in Unity. We created a new project and then created a 3d object plane. In photoshop, we then created textures with a base(RGB) and a Normalmap to make the textures appear three dimensional. We then duplicated these tiles one at a time until we had created the entire first room and courtyard.



This was extremely time consuming, to create an entire maze in this manner would not have been feasible, so to develop the maze section of our game we used a 3d modelling software to create the object in 3d and then exported it. This is the same maze depicted in the 'Textures in Unity' section. Although there is a steep learning curve with Blender, learning to use it(or other 3d modelling software) is vital for development on Unity if you want to be able to populate your game environment with unique characters and objects. We also developed the spike traps in the game using Blender. These were created with the help of an online tutorial, we first created the base of the spike trap and then spikes seperately so that they could be moved independantly of the base of the trap at



run-time. Once this was done the trap was exported to Unity where we added the textures and sounds and also colliders that determine whether or not the spikes have connected with the character. If they do, they remove the characters health accordingly.



8.1 Animation

The animation on the spikes was relatively basic. The spikes start underneath the base and then over the course of a few seconds, extend and then sloely retract. To do this we created an animator in Unity and a subsequent animation and applied them to our model. We animated via the 'Transform' Unity class that we discussed earlier. This basically means we animated the spikes position in 3d space. So, in the editor we select a keyframe at time X, and at X we move the spikes in the Unity editor to position Y. When we run the animation the spikes will move from their starting position to point Y over the space of X amount of seconds. The we reverse this so the spikes go back in. These actions are solely based on time and happen continuously as the game is running.

```
void Start () {
```

```
StartCoroutine(Go());
}
IEnumerator Go(){
while(true)
{
animation.Play();
AudioSource.PlayClipAtPoint(spikeNoise, transform.position);
yield return new WaitForSeconds(3f);
}
}
```

The enemies on the other hand were downloaded from the Unity Assets store and their animations were already configured so we just needed to program when to run each particular animation. For example, the 'idle' animation would run by default, but when the Player got within a certain range the animation would switch to the 'run' animation and when the enemy intersected with the Player's sphere collider the 'attack' animation plays. These assets are extremely useful when developing as they allow you to define attack strength, movement speed, line of sight, the enemies size and much more without having to animate the characters and create the models yourself. All of this would need to be done form scratch however if you were planning to bring a game to the market for a fee, otherwise the original creators would be owed royalties.

9 Software Evaluation

Unity's main attraction for us was it's interface. We had never developed a 3d game before or used a gaming IDE so we felt learning a completely new technology might be a monumental task, but after a few days of working with Unity we felt we were competent at perfroming simple tasks. We also noticed a large consideration for texture compression and deployment on virtually any platform. We eventually decided to deploy our game for PC and produced a standalone .exe file that can be run from virtually any computer. Although, this could just as easily have been deployed on Android, iOS, Windows Phone 8 and BlackBerry. We downloaded the android SDK from the android website and installed earlier versions of our game on our phones, but after testing we found that it was extremely difficult to move a character around and control the camera at the same time using only touch input, so we decided against this option and settled on a standalone PC version instead, which runs from an executable file on the desktop. This also allows mouse and keyboard inputs as the game follows the more conventional PC gaming criteria. Console development is also simple but requires access to SDK's which are often difficult for beginner developers to attain as you often need to havve a proven track record of development before being allowed to develop for consoles, whereas with PC you can develop a game and submit it to outlets such as 'Steam: Greenlight' which is an online gaming forum that can approve your game concept and in turn, if it is received well be distributed via the Steam online store. You have 2 coding options with Unity,

C-sharp or Javascript. Development for Unity is also made easier thanks to the wealth of tutorials on the Unity website which include documentation and videos.

One of the major let downs with Unity is it's sub-par modeling capabilities. It is competent for creating generic shapes, which we did for numerous parts of the game such as the entire first room which is just a large set of 3d planes that are joined together, and the planks littering the floor in the courtyard. However, for more detailed models it is necessary to employ a third party software such as Blender(The one we chose) or 3ds Max. Despite this failing though, Unity does allow almost any type of model format, such as .fbx to be imported simply. Unity's asset store is also a good alternative to creating your own models and there is a vast catalogue there, the downside is that for a beginner developer there isn't alot of free resources, and the prices on the paid resources can vary greatly.

The used the free version of Unity. This meant that we were limited in some ways, for example, if you want to use a certain type of light you can use only a certain amount of them because they are extremely high quality and require a lot of real-time rendering, you can get around this issue by 'Baking' the scene, which means to effectively 'paint' the environment in the lighted condition so that it appears that these lights are everywhere in the environment. However, this 'Baking' can only be done in the Pro version of Unity which costs 1,500 dollars. Also, in recent months Unity have released Unity 5 personal edition which is similar to the professional version and is for small studios that earn under 100,000 dollars per annum.

9.1 Alternatives to Unity

Unreal Engine 4 (UE4) Has recently been released by Epic Games. It is capable of using advanced dynamic lighting capabilities and has a much better particle system than Unity's, boasting 1 million particles at a time. The scripting language for Unreal is C++. UE4 used to cost 19 dollars a month with a 5 percent royalty fee when you ship the game but is now completely free. If you earn 3000 dollars per quarter on a particular title you still need to pay the royalty fee. UE4 also has tutorials on it's site, but not as many as Unity.

CryENGINE is another alternative that is extremely powerful but has an extremely steep learning curve, an wasn't really an optionfor us because we didn't need an engine with such high performance capabilities and CryENGINE doesn't have as much tutorials as Unity or UE4. Below is a list of some PROS and CONS of Unity:

PROS:

- Free
- Relatively easy to use
- Large assets store

- Can easily import 3rd party models
- Good tutorials and resources

CONS:

- Some features not available on free version
- Can't develop detailed models in editor

9.2 Blender

As discussed earlier, Blender is the third-party software we used to create the spike traps and the entire maze section. This was extremly useful in conjunction with Unity as it allows you to export directly into your Unity assets folder. However, we found that when we exported some models with textures we had to re-scale the textures on the models as they became out of synch on import. THis was a minor problem that was relatively easy to fix through Unity but with a more complex model this might be troublesome. Blender is quite easy to use and is completely free.

10 Conclusion

We set out to develop a basic game where the Player character can move around the environment, picking up keys to escape a room with some enemies and a basic inventory. We also wanted to make the game somewhat immersive by adding music and lighting. We are satisfied that we met these goals and feel that in some regards exceeded them by having animated enemies and traps, 3d models that we created from scratch, level loading, multiple scenes, a game-world that was entirely created by us using Unity and Blender, an inventory system that is affected by a character you speak to, a character who gives you different dialogue options depending on what you have in your inventory with menus and credits etc.

The main aim of the project was to get a good understanding of the entire development process from coding to model design and animation and also deployment considerations such as texture compression and anti-aliasing. Although our game is nowhere near what we would deem to be of merchantable quality, I feel we did this sufficiently during the course of the project and feel we have an extensive knowledge of Unity which would allow us to make a much better, more cohesive game in the future, allowing us to clearly define exactly how we want the game to perform, have complete artistic freedom and then be able to carry out our goals given enough time, this didn't seem achievable before we started our research and online tutorials on receipt of our project brief.

During the process of creating The Adventures of Milan we have gained substantial knowledge of texture compression and it's value in the ever expanding

mobile market and anti-aliasing in the PC gaming market. To have this knowledge at the outset of a project would allow us to tailor our development to a particular platform with great efficiency.

Unity is a good IDE with many perks and areas it exceeds in. Minor flaws, that have been discussed earlier, don't prevent this from being an excellent choice for beginner developers that want to have access to a myriad of discussion forums and online tutorials and have the ability to deploy quickly to virtually any platform.

11 Scripts

Below is a list of the C-Sharp scripts used in the project.

11.1 Acknowledgments.CS

This script controls the game's credits. The Update() method makes the camera move upwards slightly faster than the text to create the effect you can see when the game runs.

```
using UnityEngine;
using System.Collections;

public class Aknowlegments : MonoBehaviour {
public GameObject text;
public GameObject camera2;
public int speed = 10;
public string level;

void Update () {
text.transform.Translate(0, 0.08f,0);
camera2.transform.Translate(0, 0.005f,0);
}

IEnumerator waitFor(){
yield return new WaitForSeconds (5);
Application.LoadLevel (level);
}
}
```

11.2 BlueKeyPickUp.CS

This script controls everything that happens when you collide with the blue key. The key is destroyed in the gameworld, added to your inventory and a sound plays.

```
using UnityEngine;
using System.Collections;

public class BlueKeyPickUp : MonoBehaviour {
public AudioClip keyGrab; // Audioclip to play when the key is picked up.
public GameObject door;
private GameObject player;// Reference to the player.
private PlayerInventory playerInventory;        // Reference to the player's inventory.

void Awake ()
{
// Setting up the references.
player = GameObject.FindGameObjectWithTag("Player");
playerInventory = player.GetComponent<PlayerInventory>();

}
void OnTriggerEnter (Collider other)
{
// If the colliding gameobject is the player...
if(other.gameObject == player)
{ // ... play the clip at the position of the key...
AudioSource.PlayClipAtPoint(keyGrab, transform.position);
// ... the player has a key ...
playerInventory.hasKeyBlue = true;
// ... and destroy this gameobject.
Destroy(gameObject);
playerInventory.blueKeyUp =true;

}
}
}
```

11.3 BookPickup.CS

```
using UnityEngine;
using System.Collections;

public class BookPickup : MonoBehaviour {
public AudioClip keyGrab; // Audioclip to play when the key is picked up.
public GameObject door;
private GameObject player;// Reference to the player.
private PlayerInventory playerInventory;        // Reference to the player's inventory.

void Awake ()
{
```

```
// Setting up the references.
player = GameObject.FindGameObjectWithTag("Player");
playerInventory = player.GetComponent<PlayerInventory>();

}

void OnTriggerEnter (Collider other)
{
// If the colliding gameobject is the player...
if(other.gameObject == player)
{
AudioSource.PlayClipAtPoint(keyGrab, transform.position);
playerInventory.hasBook = true;
Destroy(gameObject);
playerInventory.hasBookStill =true;

}
}
}
```

11.4 BulletGrenade.CS

After you fire the gun a bullet is created, this script makes that bullet disappear after 3 seconds.

```
using UnityEngine;
using System.Collections;

public class BulletGrenade : MonoBehaviour {

float makeBulletDisappear = 3.0f;
// Use this for initialization
void Start () {
}
// Update is called once per frame
void Update () {
makeBulletDisappear -= Time.deltaTime;
if(makeBulletDisappear <= 0){
destroy();
}
}
void destroy()
{
Destroy (gameObject);
```

```
}
}
```

11.5 DoorOpener.CS

This script makes the door open IF you have the key. Controls sounds also.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class DoorOpener : MonoBehaviour {

public AudioClip doorOpen;
public AudioClip accessDenied;
public GameObject door;
public GameObject light;
private GameObject player;// Reference to the player.
private PlayerInventory playerInventory;        // Reference to the player's inventory.
private ItemDatabase itemdb;
public List<Item> items = new List<Item>();
private Inventory inventory;

void Awake ()
{
// Setting up the references.
player = GameObject.FindGameObjectWithTag("Player");
light = GameObject.Find ("DoorMaster/Point light");
playerInventory = player.GetComponent<PlayerInventory>();
door = GameObject.Find ("DoorMaster/RedDoorOpen/Cube");
}

void OnTriggerEnter (Collider other)
{
if (other.gameObject == player) {

// If the colliding gameobject is the player...
if (playerInventory.hasKey == false) {
AudioSource.PlayClipAtPoint (accessDenied, transform.position);

} else {
AudioSource.PlayClipAtPoint (doorOpen, transform.position);
Destroy (door);
Destroy(light);

}
```

```
}

}
}
```

11.6 DoorOpenerBlue.CS

```
using UnityEngine;
using System.Collections;

public class DoorOpenerBlue : MonoBehaviour {

public AudioClip doorOpen;
public AudioClip accessDenied;
public GameObject light;
public GameObject door;
private GameObject player;// Reference to the player.
private PlayerInventory playerInventory;        // Reference to the player's inventory.

void Awake ()
{
// Setting up the references.
player = GameObject.FindGameObjectWithTag("Player");
light = GameObject.Find ("DoorMasterBlue/BlueDoorOpen/Point light");
playerInventory = player.GetComponent<PlayerInventory>();
door = GameObject.Find ("DoorMasterBlue/BlueDoorOpen/Cube");
}

void OnTriggerEnter (Collider other)
{
if (other.gameObject == player) {
// If the colliding gameobject is the player...
if (playerInventory.hasKeyBlue == false) {
AudioSource.PlayClipAtPoint (accessDenied, transform.position);
} else {
AudioSource.PlayClipAtPoint (doorOpen, transform.position);
Destroy (door);
Destroy (light);

}
}

}
}
```

11.7 EnemyAttack.CS

Controls when the enemy can attack. Is the player in range, if so follow, if the sphere colliders intesect, attack. Timer determines cool off timebetween attacks.

```
using UnityEngine;
using System.Collections;


public class EnemyAttack : MonoBehaviour
{
public float timeBetweenAttacks = 2f;     // The time in seconds between each attack.
public float timeAfterAttacks = 4f;
public int attackDamage = 10;               // The amount of health taken away per attack.
public float duration = 4f;
AudioSource spear;
Animator anim;                              // Reference to the animator component.
GameObject player;                          // Reference to the player GameObject.
PlayerHealth playerHealth;                  // Reference to the player's health.
//EnemyHealth enemyHealth;                    // Reference to this enemy's health.
bool playerInRange;                         // Whether player is within the trigger collider
float timer;                                // Timer for counting up to the next attack.
//AudioSource spear;

void Awake ()
{
// Setting up the references.
player = GameObject.FindGameObjectWithTag ("Player");
playerHealth = player.GetComponent <PlayerHealth> ();
//enemyHealth = GetComponent<EnemyHealth>();
anim = GetComponent <Animator> ();
spear = GetComponent <AudioSource> ();

}


void OnTriggerEnter (Collider other)
{
// If the entering collider is the player...
if(other.gameObject == player)
{
// ... the player is in range.
playerInRange = true;
}
}
```

```
void OnTriggerExit (Collider other)
{
// If the exiting collider is the player...
if(other.gameObject == player)
{
// ... the player is no longer in range.
playerInRange = false;
}
}

void Update ()
{
// Add the time since Update was last called to the timer.
timer += Time.deltaTime;

// If the timer exceeds the time between attacks, the player is in range and this enemy is a
if((timer >= timeBetweenAttacks) && (playerInRange) )//&& enemyHealth.currentHealth > 0
{
// ... attack.

Attack ();
}

// If the player has zero or less health...
if(playerHealth.currentHealth <= 0)
{
// ... tell the animator the player is dead.
anim.SetTrigger ("PlayerDead");
}
}

IEnumerator Wait(float duration)
{
//This is a coroutine
yield return new WaitForSeconds(duration);   //Wait
}

void Attack ()
{
// If the player has health to lose...
if(playerHealth.currentHealth > 0)
{
animation.Stop("run");
animation.Play("attack");
Wait(duration);//yield return new WaitForSeconds(duration);
playerHealth.TakeDamage (attackDamage);
```

```
spear.Play ();
}
timer = 0f;
}
}
```

11.8 EnemyMove.CS

This controls how the enemies move, if in range etc. Switches animations also.

```
using UnityEngine;
using System.Collections;

public class EnemyMove : MonoBehaviour {

public float timeBetweenAttacks = 5f;
public float allowableDist = 3f;
public int attackDamage = 10;
public Transform player2;
GameObject player;
Animator anim;
NavMeshAgent nav;
bool playerInRange;
bool yoke =true;
float timer;

void Awake ()
{
player2 = GameObject.FindGameObjectWithTag ("Player").transform;
player = GameObject.FindGameObjectWithTag ("Player");
nav = GetComponent <NavMeshAgent> ();
anim = GetComponent <Animator> ();
}


//Test
void OnTriggerEnter (Collider other)
{
// If the entering collider is the player...
if(other.gameObject == player)
{
// ... the player is in range.
playerInRange = true;
yoke =false;
}
}
```

```
void OnTriggerExit (Collider other)
{
//animation.Play("run");
// If the exiting collider is the player...
if(other.gameObject == player)
{
// ... the player is no longer in range.
playerInRange = false;
}

}

void Update ()
{
float dist = Vector3.Distance(player2.position, transform.position);

if (dist < allowableDist) {

if (playerInRange == false && yoke == true) {
animation.Play ("run");
}
timer += Time.deltaTime;

// If the timer exceeds the time between attacks, the player is in range and this enemy is a
if (timer >= timeBetweenAttacks && playerInRange) {//&& enemyHealth.currentHealth > 0
//

Attack ();
} else if (playerInRange == false) {
//animation.Stop("attack");
//animation.Play("run");

Run ();
}


nav.SetDestination (player2.position);

} else if (dist > allowableDist) {
animation.Play ("idle");
}
}

void Attack ()
{
```

```
timer = 0f;
animation.Stop("run");
animation.Play("attack");

}

void Run ()
{
animation.Stop("attack");
animation.Play("run");

}
}
```

11.9 ExplosionGrenade.CS

This controls whether the bullet connects and destroys the enemy or not.

```
using UnityEngine;
using System.Collections;

public class ExplosionGrenade : MonoBehaviour {



//public AudioClip explodeGrenade;
float lifespan = 3.0f;
public GameObject fireEffect;
public GameObject fireEffect2;
public GameObject rabbit;
public AudioClip explodeGrenade;
public GameObject cube;
private PlayerInventory playerInventory;

void Start () {
}

void Awake(){
}

void Update () {

lifespan -= Time.deltaTime;
```

```
if(lifespan <= 0) {
Explode ();
}
}

void OnCollisionEnter(Collision collision)
{

if (collision.gameObject.tag == "Enemy")
{
//rabbit = GameObject("Zombunny");
collision.gameObject.tag = "Untagged";
Instantiate(fireEffect, collision.transform.position, Quaternion.identity);
Destroy (collision.gameObject);

AudioSource.PlayClipAtPoint (explodeGrenade, transform.position);

Explode ();
}

else if (collision.gameObject.tag == "EnemyBoss")
{
//rabbit = GameObject("Zombunny");
collision.gameObject.tag = "Untagged";
Instantiate(cube, collision.transform.position, Quaternion.identity);
Destroy (collision.gameObject);

AudioSource.PlayClipAtPoint (explodeGrenade, transform.position);

Explode ();
}
}

void Explode() {
Destroy (gameObject, 2f);
playerInventory.enemyDead = true;
}
}
```

11.10 FirstPersonController.CS

First person controller, controlls all player movement and keyboard inputs.

```
using UnityEngine;
using System.Collections;
```

```
[RequireComponent (typeof(CharacterController))]
public class FirstPersonController : MonoBehaviour {
public float movementSpeed = 6.0f;
public float mouseSensitivity = 5.0f;
public float jumpSpeed = 20.0f;
public float rotUpDown = 0;
public float upDownRange = 60.0f;
float verticalVelovicty = 0;
CharacterController characterController;
private PlayerInventory playerInventory;

void Start(){
Screen.lockCursor = true;
characterController = GetComponent<CharacterController>();
}

// Update is called once per frame
void Update () {
CharacterController characterController = GetComponent<CharacterController>();
float rotLeftRight = Input.GetAxis ("Mouse X") * mouseSensitivity;
transform.Rotate (0, rotLeftRight, 0);

rotUpDown -= Input.GetAxis ("Mouse Y") * mouseSensitivity;
rotUpDown = Mathf.Clamp (rotUpDown, -upDownRange, upDownRange);
Camera.main.transform.localRotation = Quaternion.Euler (rotUpDown, 0,0);

float forwardSpeed = Input.GetAxis("Vertical")*movementSpeed;
float sideSpeed = Input.GetAxis("Horizontal")*movementSpeed;

verticalVelovicty += Physics.gravity.y * Time.deltaTime;

if (characterController.isGrounded && Input.GetButtonDown ("Jump")) {
verticalVelovicty = jumpSpeed;

forwardSpeed = (Input.GetAxis("Vertical")*movementSpeed)/2;
sideSpeed = (Input.GetAxis("Horizontal")*movementSpeed)/2;
}

Vector3 speed = new Vector3 (sideSpeed,verticalVelovicty,forwardSpeed);

speed = transform.rotation * speed;

characterController.Move(speed*Time.deltaTime);

}
```

```
}
```

11.11 greenKeyPickup.CS

```
using UnityEngine;
using System.Collections;

public class greenKeyPickup : MonoBehaviour {

public AudioClip keyGrab;

public GameObject door;
private GameObject player;
private PlayerInventory playerInventory;

void Awake ()
{

player = GameObject.FindGameObjectWithTag("Player");
playerInventory = player.GetComponent<PlayerInventory>();
}

void OnTriggerEnter (Collider other)
{
if(other.gameObject == player)
{
AudioSource.PlayClipAtPoint(keyGrab, transform.position);
playerInventory.hasKeyGreen = true;
Destroy(gameObject);
playerInventory.greenKeyUp =true;
}
}
}
```

11.12 Heart1Pickup.CS

```
using UnityEngine;
using System.Collections;

public class Heart1Pickup : MonoBehaviour {

public AudioClip keyGrab; // Audioclip to play when the key is picked up.

public GameObject door;
```

```
private GameObject player;// Reference to the player.
//private GameObject doorM;// Reference to the player.
private PlayerInventory playerInventory;        // Reference to the player's inventory.
private ItemDatabase database;
private Inventory inventory;
// Use this for initialization
void Awake () {
player = GameObject.FindGameObjectWithTag("Player");
playerInventory = player.GetComponent<PlayerInventory>();
}

void OnTriggerEnter (Collider other)
{
if(other.gameObject == player)
{
// ... play the clip at the position of the key...
AudioSource.PlayClipAtPoint(keyGrab, transform.position);
Destroy(gameObject);
playerInventory.hasHeart1 =true;
}
}
}
```

11.13 Inventory.CS

This controls how things are added to the inventory and draws the inventory on the screen.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Inventory : MonoBehaviour {
public int slotsX, slotsY;
public GUISkin skin;
private KeyPickUp keyPickup;
public PlayerHealth playerHealth;
public List<Item> inventory = new List<Item>();
private ItemDatabase database;
public List<Item> slots = new List<Item>();
private bool showInventory;
private bool showTooltip;
private string tooltip;
private PlayerInventory playerInventory;
private GameObject player;
```

```
private bool draggingItem;
private Item draggedItem;
private int prevIndex;
public Transform prefab;
public GameObject cube;

void Awake ()
{

player = GameObject.FindGameObjectWithTag("Player");

// Setting up the references.
playerInventory = player.GetComponent<PlayerInventory>();
//doorM = GameObject.FindGameObjectWithTag("Door Master");

}
// Use this for initialization
void Start () {

for(int i=0; i< (slotsX * slotsY); i++)
{
slots.Add(new Item());
inventory.Add(new Item());
}
//inventory.Add ();
database = GameObject.FindGameObjectWithTag ("Item Database").GetComponent<ItemDatabase>();

}
void Update()
{
if(Input.GetButtonDown("Inventory"))
{
if (playerInventory.blueKeyUp == false) {

} else if (playerInventory.blueKeyUp == true){
AddItem (0);
}
if (playerInventory.redKeyUp == false) {

} else if (playerInventory.redKeyUp == true){
AddItem (2);
}
if (playerInventory.hasHeart1 == false) {

} else if (playerInventory.hasHeart1 == true){
AddItem (3);
```

```
}


if (playerInventory.hasBook == false) {

}
else if (playerInventory.hasBook == true){

AddItem (5);
}


if (playerInventory.greenKeyUp == false) {

} else if (playerInventory.greenKeyUp == true){
AddItem (4);

}

if (playerInventory.hasMapStill == false) {

} else if (playerInventory.hasMapStill == true){
AddItem (6);

}

showInventory =! showInventory;

playerInventory.redKeyUp = false;
playerInventory.blueKeyUp = false;
playerInventory.hasHeart1 = false;
playerInventory.greenKeyUp = false;
playerInventory.hasBook = false;
playerInventory.hasMapStill  = false;

}


}


Vector3 GeneratedPosition()
{
int x,y,z;
x = 1110;
y = 2;
```

```
z = 1054;
return new Vector3(x,y,z);
}

void PlaceCubes()
{
Instantiate(cube, GeneratedPosition(), Quaternion.identity);

}

void OnGUI()
{
tooltip = "";

GUI.skin = skin;
if(showInventory)
{
DrawInventory();
if(showTooltip)

GUI.Box(new Rect(Event.current.mousePosition.x,
Event.current.mousePosition.y, 200, 200), tooltip, skin.GetStyle("Tooltip"));

}

if (draggingItem) {
GUI.DrawTexture(new Rect(Event.current.mousePosition.x,
Event.current.mousePosition.y, 100, 100), draggedItem.itemIcon);

}

}

void DrawInventory()
{


Event e = Event.current;

int i = 0;
for(int y=0; y < slotsY; y++){
for(int x = 0; x < slotsX; x++){

Rect slotRect = new Rect(x*125, y*125,100,100);
GUI.Box(slotRect, "", skin.GetStyle("Slot"));
slots[i] = inventory[i];
```

```
if(slots[i].itemName != null)
{
GUI.DrawTexture(slotRect, slots[i].itemIcon);

if(slotRect.Contains(e.mousePosition))
{
tooltip = CreateTooltip(slots[i]);
showTooltip = true;
if(e.button == 0 && e.type == EventType.mouseDrag && !draggingItem)
{
draggingItem = true;
prevIndex = i;
draggedItem = slots[i];
inventory[i] = new Item();
}

if(e.type == EventType.mouseUp && draggingItem)
{
inventory[prevIndex] = inventory[i];
inventory[i] = draggedItem;
draggingItem = false;
draggedItem = null;
}

if(e.type == EventType.mouseDown && !draggingItem)
{
//RemoveItem (0);
playerInventory.hasHeart1 = false;
Update();
}
}
} else{

if(slotRect.Contains(e.mousePosition))
{
if(e.type == EventType.mouseUp && draggingItem)
{
inventory[i] = draggedItem;
draggingItem = false;
draggedItem = null;
}
}
}


if(tooltip == "")
```

```
{
showTooltip = false;
}
i++;
}
}
}

string CreateTooltip(Item item)
{

tooltip = "<color=#ffffff>"+ item.itemName + "</color>\n\n" + item.itemDesc;
return tooltip;
}

void AddItem(int id)
{
for (int i=0; i < inventory.Count; i++) {
if (inventory [i].itemName == null) {

for(int j=0; j < database.items.Count; j++)
{
if(database.items[j].itemID == id)
{
inventory[i] = database.items[j];
}

}
break;

}
}
}

void RemoveItem(int id)
{
for (int i = 0; i < inventory.Count; i++) {
if (inventory [i].itemID == i) {
inventory [i] = new Item ();
break;
}
}
}

bool InventoryContains(int id)
{
```

```
bool result = false;
for (int i=0; i< inventory.Count; i++) {
result = inventory[i].itemID == id;
if(result){

break;
}
}

return result;
}
}
```

11.14 Item.CS

```
using UnityEngine;
using System.Collections;

[System.Serializable]

public class Item{

public string itemName;
public int itemID;
public string itemDesc;
public Texture2D itemIcon;
public int itemPower;
public int itemSpeed;
public ItemType itemType;

public enum ItemType{
Weapon,
Consumable,
Quest
}

public Item(string name, int id, string desc, int power, int speed, ItemType type)
{
itemName = name;
itemID = id;
itemDesc = desc;
itemIcon = Resources.Load<Texture2D>("Item Icons/"+ name);
itemPower = power;
itemSpeed = speed;
itemType = type;
```

```
}

public Item()
{

}
}
```

11.15 ItemDatabase.CS

A database of items, defined in this class, they can be accessed from the inventory script.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class ItemDatabase : MonoBehaviour {

public List<Item> items = new List<Item>();
private PlayerInventory playerInventory;
private GameObject player;

void Awake()

{
items.Add (new Item ("I_Key05",0, "Opens Blue Doors",0,0, Item.ItemType.Quest));
items.Add (new Item ("S_Shadow02",1, "Explode when they hit enemies",2,2, Item.ItemType.Weap
items.Add (new Item ("redKey",2, "Opens Red Doors",0,0, Item.ItemType.Quest));
items.Add (new Item ("S_Holy01",3, "Restores Health",0,0, Item.ItemType.Consumable));
items.Add (new Item ("I_Key02",4, "Opens Green Doors",0,0, Item.ItemType.Quest));
items.Add (new Item ("W_Book03",5, "Maybe I should bring this back to the mage.",0,0, Item.I
items.Add (new Item ("I_Map",6, "A map! this might help me escape the maze. (press M to use)
items.Add (new Item ("I_Torch02",7, "(press F to place).",0,0, Item.ItemType.Quest));
}
void Update()
{

}


}
```

11.16 MazeDoorScript.CS

```
using UnityEngine;
using System.Collections;

public class MazeDoorScript : MonoBehaviour {

public AudioClip doorOpen;
public AudioClip accessDenied;
public GameObject light;
public GameObject door;

private GameObject player;// Reference to the player.
private PlayerInventory playerInventory;        // Reference to the player's inventory.


void Awake ()
{
// Setting up the references.
player = GameObject.FindGameObjectWithTag("Player");
light = GameObject.Find ("DoorMasterGreen/GreenDoorOpen/Point light");
playerInventory = player.GetComponent<PlayerInventory>();
door = GameObject.Find ("DoorMasterGreen/GreenDoorOpen/Cube");
}

void OnTriggerEnter (Collider other)
{
if (other.gameObject == player) {

if (playerInventory.hasKeyGreen == false) {

AudioSource.PlayClipAtPoint (accessDenied, transform.position);


} else {
AudioSource.PlayClipAtPoint (doorOpen, transform.position);

Destroy (door);

}
}

}
}
```

11.17 PauseMenu.CS

This displays UI buttons that allow the player to switch music off, view controls etc.

```
using UnityEngine;
using System.Collections;

public class PauseMenu : MonoBehaviour {

public GUISkin myskin;
public GUIStyle customButton;
public GUIStyle customButton1;
private Rect windowRect;
private bool paused = false , waited = true, option = false;

private void Start()
{
windowRect = new Rect(Screen.width / 2 - 100, Screen.height / 2 - 100, 200, 300);
AudioListener.pause = false;
}

private void waiting()
{
waited = true;
}

private void Update()
{
if (waited)
if (Input.GetKey(KeyCode.Escape) || Input.GetKey(KeyCode.P))
{
GameObject.Find("Player").GetComponent<FirstPersonController>().enabled=false;
GameObject.Find("GunBarrelEnd").GetComponent<PlayerShooting>().enabled=false;

if (paused)

paused = false;
else
paused = true;

waited = false;
Invoke("waiting",0.3f);
}
if (paused)
{
Time.timeScale = 0;
```

```
}
else
{
Time.timeScale = 1;
}
}

private void OnGUI()
{
if (paused)

windowRect = GUI.Window(0, windowRect, windowFunc, "Pause Menu",customButton1);

if (option)
{
windowRect = GUI.Window(0, windowRect, windowFuncOption, "Option",customButton1);

}
}

private void windowFunc(int id)
{
if (GUILayout.Button("Resume",customButton))
{
GameObject.Find("Player").GetComponent<FirstPersonController>().enabled=true;
GameObject.Find("GunBarrelEnd").GetComponent<PlayerShooting>().enabled=true;
paused = false;

}
if (GUILayout.Button("Options",customButton))
{
option = true;
//paused = false;


}
if (GUILayout.Button("Quit",customButton))
{
Application.LoadLevel("MenuMain");
}
}

private void windowFuncOption(int id)
{
if (GUILayout.Button("Audio On",customButton))
```

```
{
AudioListener.pause = false;
option = false;
paused = true;

}
if (GUILayout.Button("Audio Off",customButton))
{
AudioListener.pause = true;
option = false;
paused = true;

}

}
}
```

11.18 PlayerHealth.CS

Controls what happens when the player is hit. It puts the health slider down an amount relative to the enemy's strength.

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class PlayerHealth : MonoBehaviour
{
public int startingHealth = 100;
// The amount of health the player starts the game with.
public int currentHealth;
 // The current health the player has.
public Slider healthSlider;
 // Reference to the UI's health bar.
public Image damageImage;
// Reference to an image to flash on the screen on being hurt.
public AudioClip deathClip;
 // The audio clip to play when the player dies.
public float flashSpeed = 5f;
public float duration = 5f;
// The speed the damageImage will fade at.
public Color flashColour = new Color(1f, 0f, 0f, 0.1f);
// The colour the damageImage is set to, to flash.
public int getHealth = 5;
public GameObject heart;
public PlayerInventory playerInventory;
```

```
// Reference to the player's inventory.
private ItemDatabase database;
private Inventory inventory;
Animator anim;
 // Reference to the Animator component.
AudioSource playerAudio;
 // Reference to the AudioSource component.
PlayerMovement playerMovement;
 // Reference to the player's movement.
//PlayerShooting playerShooting;
 // Reference to the PlayerShooting script.
bool isDead;
 // Whether the player is dead.
bool damaged;
// True when the player gets damaged.
public string gameOverScreen;

void Awake ()
{
// Setting up the references.
anim = GetComponent <Animator> ();
playerAudio = GetComponent <AudioSource> ();
playerMovement = GetComponent <PlayerMovement> ();
//playerShooting = GetComponentInChildren <PlayerShooting> ();
heart = GameObject.Find ("SimpleHeart1");

// Set the initial health of the player.
currentHealth = startingHealth;
}


void Update ()
{
// If the player has just been damaged...
if(damaged)
{
// ... set the colour of the damageImage to the flash colour.
damageImage.color = flashColour;
}
// Otherwise...
else
{
// ... transition the colour back to clear.
//damageImage.color = Color.Lerp
(damageImage.color, Color.clear, flashSpeed * Time.deltaTime);
}
```

```
// Reset the damaged flag.
damaged = false;
}


IEnumerator Wait(float duration)
{
//This is a coroutine
yield return new WaitForSeconds(duration);   //Wait
}

public void TakeDamage (int amount)
{
Wait(duration);
//yield return new WaitForSeconds(2);
// Set the damaged flag so the screen will flash.
damaged = true;

// Reduce the current health by the damage amount.
currentHealth -= amount;

// Set the health bar's value to the current health.
healthSlider.value = currentHealth;

// Play the hurt sound effect.
playerAudio.Play ();

// If the player has lost all it's health
and the death flag hasn't been set yet...
if(currentHealth <= 20 && !isDead)
{
// ... it should die.
Death ();
}
}

void OnTriggerEnter (Collider other)
{
// If the colliding gameobject is the player...
if (other.gameObject == heart)
{
if(currentHealth <= 95)
{
// Reduce the current health by the damage amount.
currentHealth = currentHealth + getHealth;
```

```
// Set the health bar's value to the current health.
healthSlider.value = currentHealth;

Destroy(heart);
}
else{
//currentHealth = currentHealth;
//Destroy(heart);
//playerInventory.hasHeart1 = true;

}

}
if (other.transform.tag == "Spike") {
//damaged = true;
if(currentHealth <= 20 && !isDead)
{
// ... it should die.
Death ();
}
// Reduce the current health by the damage amount.
currentHealth -= 50;

// Set the health bar's value to the current health.
healthSlider.value = currentHealth;

// Play the hurt sound effect.
playerAudio.Play ();
}
}

void Death ()
{
isDead = true;
Application.LoadLevel (gameOverScreen);
playerMovement.enabled = false;
}
}
```

11.19 PlayerInventory.CS

```
using UnityEngine;
using System.Collections;

public class PlayerInventory : MonoBehaviour {
```

```
public bool hasKey = false;
public bool redKeyUp =false;
public bool hasKeyBlue = false;
public bool blueKeyUp = false;
public bool hasKeyGreen = false;
public bool greenKeyUp = false;
public bool hasHeart1 = false;
public bool hasBook = false;
public bool hasBookStill = false;
public bool enemyDead = false;
public bool hasMap = false;
public bool hasMapStill = false;
}
```

11.20 PlayNewScene.CS

```
using UnityEngine;
using System.Collections;

public class PlayNewScene : MonoBehaviour {
private GameObject player;// Reference to the player.
public string creds;
// Use this for initialization
void Start () {

}
void Awake ()
{
// Setting up the references.
player = GameObject.FindGameObjectWithTag("Player");


}
void OnTriggerEnter (Collider other)
{
// If the colliding gameobject is the player...
if(other.gameObject == player)
{
Application.LoadLevel (creds);
}
}

// Update is called once per frame
void Update () {
```

```
}
}
```

11.21 ShootingFPS.CS

This script is responsible for firing bullets, defines their speed and movement with respect to gravity etc.

```
using UnityEngine;
using System.Collections;

public class ShootingFPS : MonoBehaviour {

// Use this for initialization
public GameObject bullet_prefab;
public GameObject grenade_prefab;
float bulletSpeed = 100f;
public AudioClip throwGrenade;
public GameObject smoke;

void Start()
{


}

void Update () {
//gun_prefab = null;
Camera c = Camera.main;

if (Input.GetButtonDown ("Fire1")) {
GameObject fpsGrenade= (GameObject)Instantiate (grenade_prefab,
 c.transform.position, c.transform.rotation);
fpsGrenade.rigidbody.AddForce
(c.transform.forward * bulletSpeed, ForceMode.Impulse);
//fire.rigidbody.AddForce(c.transform.forward *
 bulletSpeed, ForceMode.Impulse);
AudioSource.PlayClipAtPoint (throwGrenade, transform.position);
//Screen.lockCursor = true;
}
else if (Input.GetButtonDown ("Fire2")) {

Screen.lockCursor = true;
```

```
}

else if (Input.GetButtonDown ("F")) {
Instantiate(smoke,  transform.position, transform.rotation);
AudioSource.PlayClipAtPoint (throwGrenade, transform.position);

}
}
}
```

11.22 Traps.CS

Controls how the spikes behave and their sound.

```
using UnityEngine;
using System.Collections;

public class Traps : MonoBehaviour {
public float delayTime;
// Use this for initialization
public AudioClip spikeNoise;

void Start () {
StartCoroutine(Go());
}

IEnumerator Go()
{
while(true)
{
animation.Play();
AudioSource.PlayClipAtPoint(spikeNoise, transform.position);
yield return new WaitForSeconds(3f);

}


}
}
```

11.23 TrapTrigger.CS

```
using UnityEngine;
```

```
using System.Collections;

public class TrapTrigger : MonoBehaviour {

public GameObject cubeDisappear;
public GameObject cube;
private GameObject player;
public AudioClip BoulderSound;

void Awake ()
{
// Setting up the references.
player = GameObject.FindGameObjectWithTag("Player");
cubeDisappear = GameObject.Find("CubeToDisappear");
cube = GameObject.Find("CubeTrigger");

}

void OnTriggerEnter (Collider other)
{
// If the colliding gameobject is the player...
if(other.gameObject == player)
{
// ... play the clip at the position of the key...

AudioSource.PlayClipAtPoint(BoulderSound, transform.position);


// ... and destroy this gameobject.
Destroy(cube);
Destroy(cubeDisappear);


}
}
}
```