Rendering Textures Up Close in a 3D Environment Using Adaptive Micro-Texturing

Thesis
Submitted in Partial Fulfillment
of the Requirements for the

Degree of
Master of Arts in Interdisciplinary Computer Science
Mills College
Fall 2012

By
Joseph Klein

Thesis advisors:

Almudena Konrad
Associate Professor of Computer Science
Mills College

Michael Kingery
Concept Designer
SkyVu Entertainment

Susan Wang
Professor of Computer Science
Mills College

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Abstract

Two problems with all 3D representations of environments are the memory limit and rendering time of textures placed on a 3D plane. Software developers usually have to limit the size of resolutions for their textures. With these resolution limits, textures can be blurry up close when looking at them. In this thesis, I propose a new texturing technique that will replace the texture's large stretched pixels with repeatable micro-textures that represent the material that the overall texture is representing. So when the textures are seen up close now, it will display a crisp detailed representation of those materials at a micro level thus giving the illusion of much higher resolution textures within the 3D environment.

To my Mother and Father,

who always support my endeavors no matter what they are.

Table of Contents

I. Introduction

There are many computational applications that push the limits of our latest hardware. First there are the scientists that use massive supercomputers to process large amounts of data and to study and understand our world. Then there are filmmakers who create movies completely or partially using computer generated worlds and characters. And then there are video game developers who create virtual worlds, characters, and intelligence all in a single package. The main difference between the scientist or filmmaker, and the game developers, is the relative time frame they have to complete all of their tasks. Interactive video games have to do what the scientist strives to understand and the filmmaker endeavors to create, in real time, and on limited hardware. This thesis follows in that artistic endeavor.

One of the major problems I have always noticed in 3D rendered video games is the limits of textures' fidelity when scrutinized up close. Textures can only be as large as the hardware allows at render time to keep the rendering engine at a smooth 30 - 60 frames a second. This means that textures are limited in size, and thus when displayed up close, tend to be stretched and blurred which then takes away form the illusion of the environment created by the engine. The idea for my new texturing technique, Adaptive Micro-Texturing, is to replace the stretched blurry pixels that is shown, with crisp detailed micro-versions of the material that the texture is displaying. So instead of a large blurry brick texture, the engine displays micro versions of brick and mortar in place of the stretched texture. Utilizing visual trickery, and intelligent algorithms, I aspire to help create an engrossing "fake" world with my own new graphical technique which will enhance how textures on objects look at very close distances.

1.1 Background

In this section I will be explaining the basics of rendering 3D environments. The basics consist of setting up systems to organize information which describe a virtual environment which will be rendered to a computer monitor. This includes the practice of creating 3D graphics, the coding libraries which make the process much easier, and the advanced graphical shaders which have revolutionized modern 3D rendering techniques.

1.1.1 3D Graphics

While there are many methods of rendering 3D graphics, such as voxels[1] and raytracing[2], the most widely used method is vector graphics. The basis of vector graphics comes from the use of vertices and textures. Vertices are points of data that hold a position. These positions are strung together in a vertex buffer which is used to place and scale the vertices in a virtual 3D space. The buffer also holds normal and texture coordinate data for textures, which are then deformed and stretched between three or four vertices at a time. After everything is placed in its proper location, a virtual 3D space is shown to the user as shown in Figure 1.

---

[1] A representation of 3D space consisting of a regular array of cubes. [5]

[2] The ray tracing algorithm renders an image by casting rays into the scene. A ray is cast through every pixel of the image, tracing the light coming from that direction. [8]

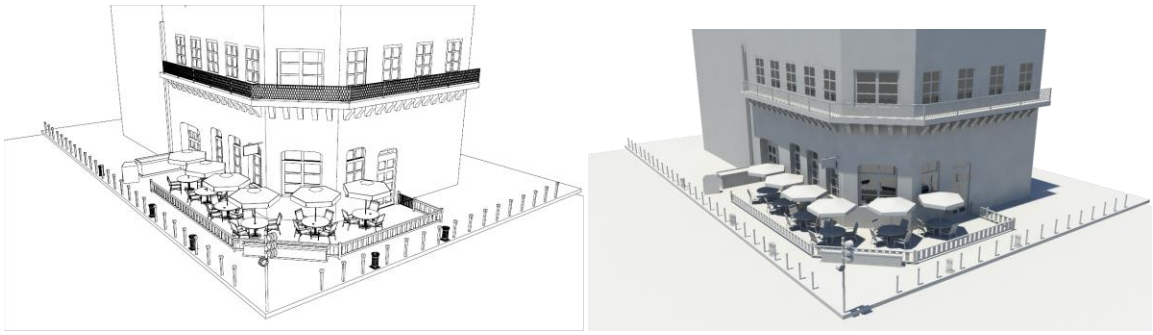


Figure 1: 3D Vector Graphics

1.1.2 Graphical API

Modern graphic Application Programming Interfaces (APIs) are used to speed up the process of creating these 3D spaces. Concurrently, these APIs have also been optimized to get the best performance out of current hardware. There are two mainstream graphical APIs that are in current use, DirectX[3] and OpenGL[4].

DirectX is the most widely used graphical library for interactive video games. It is developed by Microsoft and exclusively used on their platforms: Windows, Xbox Consoles, and Windows Phone. It was first developed for the release of Windows 95, and made available in the concurrent SDK[5] release [10]. Since then there have been numerous iterations. As of 2012, the most current release is DirectX 11 which added tessellation, ambient occlusion, and extra post-processing effects [4].

OpenGL is another popular graphical library widely used. Unlike DirectX, OpenGL is open source and free to use on almost every computer system and device. OpenGL was first

[3] DirectX provides graphics support for animation, music and sound effects support for games, and multiplayer support for network games. [10]
[4] The industry's most widely used and supported 2D and 3D graphics application programming interface. [9]
[5] Software Development Kit

released in 1992 by Silicon Graphics and the specification was written by Mark Segal and Kurt Akeley. The aim of the specification was to "formalize the definition of a useful graphics API and made cross platform non-SGI 3rd party implementation and support viable" [7]. Since OpenGL is available on all platforms, there are numerous iterations of the product, like Apple's AGL, Apple Graphics Library, uses OpenGL as well as WebGL, which is an implementation that runs within internet browsers for graphical rich internet applications.

1.1.3 Graphical Shaders

Another component of modern graphical applications is the graphical shader. While the overall code of an application will decide on the objects consisting within a 3D environment, and how they may interact with each other, the code that renders the world is processed within a video card that is designed to display and enhance the world into a monitor at very fast speeds.

The rendering of polygons to create a 3D environment is split between placing the vertices of the polygons within the environment, and then deciding the final color of every pixel on the screen by taking into account the textures on the polygons, lighting of the environment, and other visual enhancements. Modern video cards consist of a pipeline that places the code for rendering objects to the screen in two distinct locations.

First there is the vertex shader which takes all of the vertices in a scene and then displaces them and scales them according to the information in the Vertex Array Object (VAO). The VAO is a long array that holds all of the vertices as well as the normals and texture coordinate for them. The normals decide how light will bounce off of an object. The texture coordinates are the coordinates that decide how a bitmap is stretched and placed onto a polygonal object. The vertex shader is run once for every vertex that is to be rendered in the

scene. This means that performance will drop using this shader only when there is an exorbitant amount of vertices on screen.

The other shader is the Fragment Shader. The fragment shader is the shader that computes the color value of every pixel on screen. This means that the fragment shader will be run once for every pixel of the current resolution. If the display resolution of the program is 1680x1050, then the pixel shader will be run 1.764 million times. This means that the programmer must be careful with the amount of operations placed in the fragment shader since performance can take a massive hit when rendering the scene. Some of the effects that can be processed in the fragment shader is bump/normal mapping, shadows, and visual effects like blurring and depth of field.

1.2 Challenges

Challenges with creating virtual environments have always stemmed from the hardware limitations presented to the graphical artist and the computer programmer. While the artist can create extremely detailed objects and environments, the programmer has no way to directly translate those visuals in front of the consumer with the hardware they own. This is when visual trickery is brought into the fold to 'fake' a look. It comes down to knowing what something 'should' look like, rather than what it actually looks like. For example, if one looks down from their window seat from their plane over the ocean, they will see billions of waves and light sparkles from the sun's reflection. Each of those waves are unique. Each sparkle a different photon bent and reflected from different water particles. In a real time visualization of the world, that type of rendering is impossible with current or any foreseeable hardware. The graphical artist instead creates a small patch of ocean, which is repeatable in a grid like fashion, with

animated waves repeating in a loop. This will create an approximation of what an ocean should look like, though not perfect, it is generally close enough, and a good balance between performance and polish. The best part comes at runtime, when the user is rewarded with a silky smooth performance on multiple platforms.

One problem that has plagued games that use 3D environments is blurry textures when seen up close. In the 3D environment, the user usually controls the camera which moves around in the 3D space. Because of this freedom, the user can get pretty close to many different objects in the virtual world. When these objects are analyzed up close, the textures placed on them are stretched to the resolution of the current rendering window. This means that a 512x512 texture on an object might be stretched to four times its size. What the user sees then is stretched, blurry, and pixelated texture as demonstrated in Figure 2.

Figure 2. Blurry textures up close in Battlefield: Bad Company 2 [1]

This problem stems from the limitations of memory. With unlimited memory, textures equal to the resolution of the screen could be used on all of the objects in the environment. With textures that high in resolution, the pixels would never be stretched when seen up close. This though is impossible with current hardware. Problems like this are somewhat remedied by

different techniques that can soften the blow that these visual artifacts can present. Video game programmers use visual trickery and complex algorithms to maximize performance, and enhance visuals in their 3D engines.

II. Related Work

This section will introduce related works that influenced Adaptive Micro-texturing. Within each technique there are sub-techniques that are utilized in this thesis. All of them have been used in commercial products and have advanced the fidelity of 3D graphical technology.

2.1 Mipmaps

One way that is utilized to tackle the problem of texture scaling is a technique called Mipmapping. In 1983, Lance Williams wrote a paper called “Pyramid Parametrics” which summarizes the idea of mipmapping that creates a sequence of the same texture at multiple resolutions as seen in Figure 3, which is displayed hierarchically dependant upon the distance the user is from the object [17].

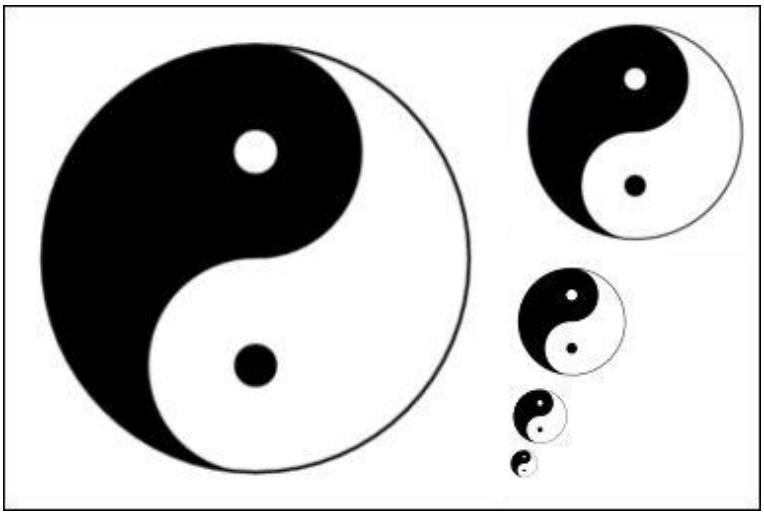


Figure 3. Mipmap

This greatly helps rendering time because rendering higher resolution textures take more time than lower resolution textures. It would be a waste of time to render a 256x256 pixel texture, if only 16x16 pixels of the screen are displaying the object. Up close, a texture will be rendered using the highest resolution, and the program will dynamically switch to lower resolutions depending on the distance the 3D camera is from the object [11]. Quality wise, Mipmapping also removes any artifacting that is inherent in scaling photos to smaller resolutions. This technique has been added to graphical APIs and can automatically be switched on in OpenGL just through code.

2.2 Texture Splattering

Texture splattering is a technique that is used to remedy the problem with repeatable texture mapping. If you create a texture like terrain or a brick wall, you can get a "tiled" look on it as illustrated in Figure 4.

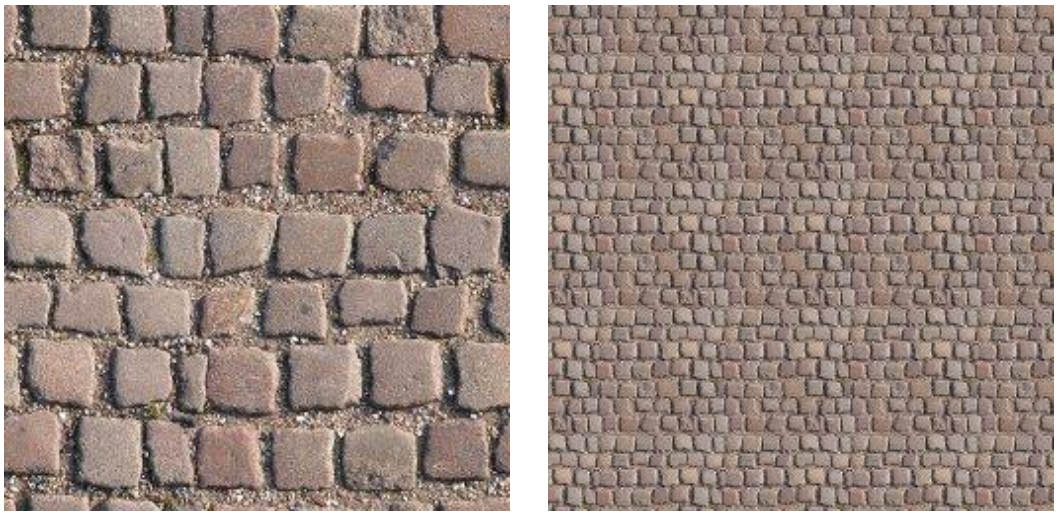


Figure 4. Tiled Map

A solution many programmers have used is multiple texture overlays. In 2000, Charles Bloom coined the term "Splattering" to describe this technique. "Splatting is a technique for texturing a terrain using high resolution localized tiling textures which transition nonlinearly" [3]. This utilizes multiple textures and places them on top of each other with different alpha channels to blend the textures together. This can enhance boring textures and as Bloom describes, "With 'noise' splatting texture transitions can be made irregular, and the tiled texture look is almost totally eliminated" [3]. Splattering presents a great technique to add detail to texture up close as seen in Figure 5.

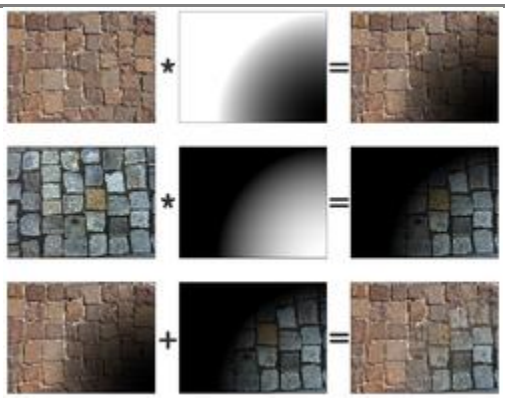


Figure 5. Texture Splattering [6]

So instead of using a large detailed texture of size 2048x2048, which would take up 4 million bytes, texture splattering techniques can use four 512x512 textures which would only use a fourth the amount of memory as the large texture as well provide a better range of differentiating visuals. Texture splattering uses this gain in memory savings as well as removing the repeatable instances of textures which can manifest in 3D environments. Overall, splattering is a great technique for creating different and detailed textures in the environment.

2.3 Bump / Normal Mapping

Bump Mapping is a method that was developed to simulate detail on mesh surfaces to take the place of extreme numbers of vertices of polygonal objects. This is achieved by creating a monochromatic height map. James F. Blinn wrote it as " a method of using a texturing function to perform a small perturbation on the direction of the surface normal" [2]. That then creates the bumps and the illusion of high detail as seen in Figure 6.

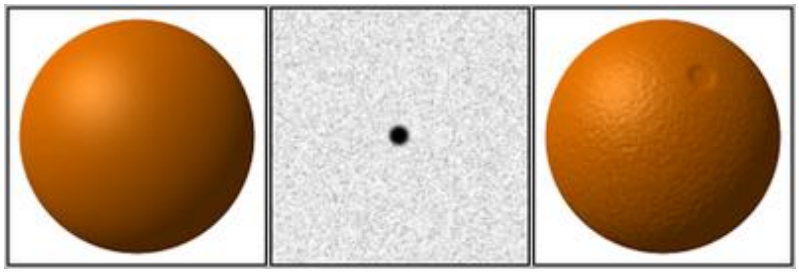


Figure 6. Bump Map Application [15]

The problem with bump maps is that the normals the bump creates is based upon the texture coordinates. This approximates the normals and can give unreliable values when placed in a 3D space. Bump mapping was iterated upon by creating a normal map. Instead a

monochromatic height map, a normal map is a heightmap based upon the values of the xyz coordinate system. When used correctly, the normals of the polygon surface are calculated and then perturbed by the normal map to create accurate normal values for the surface in world space. World space is the actual coordinate system of the 3D environment. This is illustrated in Figure 7.

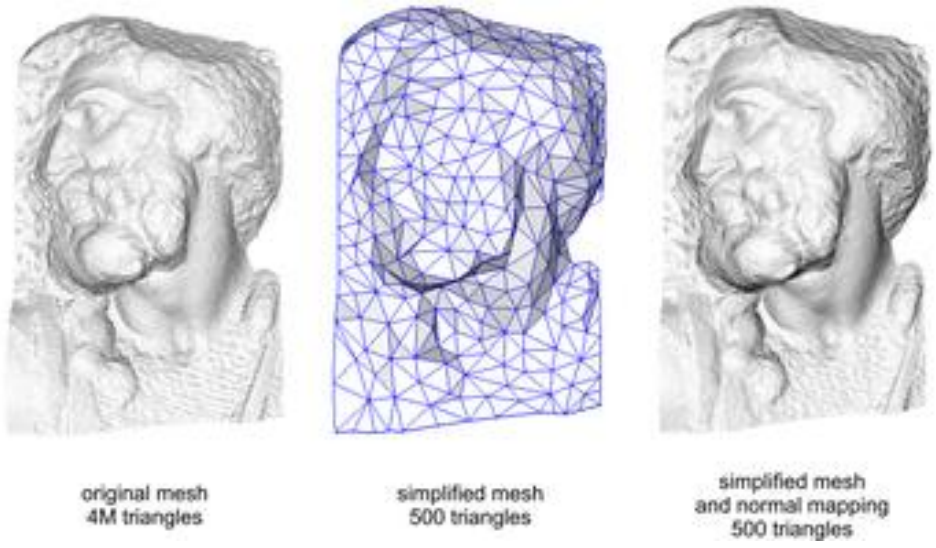
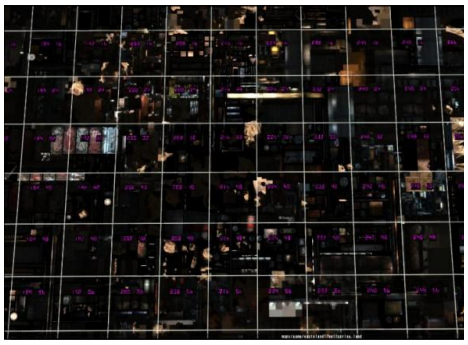


Figure 7. Normal Mapping Application [16]

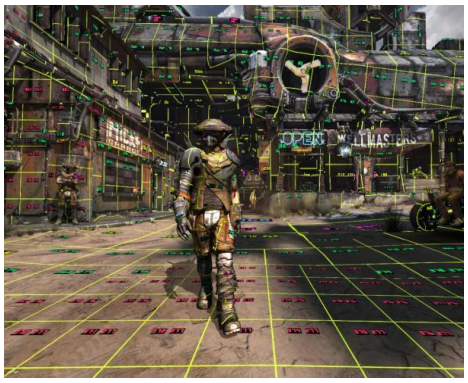
2.4 Rage's Megatextures and Virtual Texturing

A new technique that has been utilized by id Software's latest game engine, id Tech 5, is the use of Megatexturing. Megatexturing is a technique coined by John Carmack of id software, where textures are streamed into the 3D environment and placed correctly on the models within the virtual space. The technique allows the use of textures up to the size of 128000x128000 pixels. A texture of that size could never be loaded directly into memory as current hardware does not provide the necessary space. Instead, the texture is loaded in tiles of a subset size while the

environment is broken up into a grid to enable the display of the subset textures shown in Figure 8. This means artists do not have to compromise on the amount of detail they place in the 3D virtual space since a massive problem is broken up into smaller tasks.



- A small portion of a 128000x128000 pixel Megatexture which illustrates the subsets of tiles used for environments.



- Virtual texturing grid on Rage's 3D environment.

- End result in Rage when all of the subset tiles are places in the locations specified by the environmental artists.

Figure 8. Example of Virtual Texturing with a Megatexture [14]

The game Rage was the first game to use id Tech 5. The game was marred by poor texture pop-in, where textures would seem to load slowly and a blurry untextured version of the

world could be seen before all of the computational tasks had been finished. The developer blamed this on unpatched drivers. After patches and driver updates, Rage did end up showing an amazing detailed world with a very powerful new engine.

III. Implementation

This section presents the components of my thesis. It will start with how the concept was envisioned. After that, the first steps that were necessary to take to build up the capabilities for implementation will be discussed. And then the construction of the program that will demonstrate the thesis technique will be described.

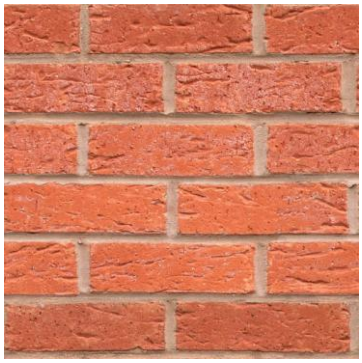
3.1 Concept

To fix the problem with blurry textures up close to a camera in a 3D environment, I developed a new technique called Adaptive Micro-Texturing. The idea of Adaptive Micro-Texturing, AMT, is to be able to see the small substructures of what makes up materials when viewed up close. When one looks very close at a shirt, it is possible to see the tiny threads that make up the textile fabric. When viewing beach sand up close, small rocks and bits of shell can be seen. With the memory constraints of texture sizes, at the moment this isn't possible. The engine would have to use a ridiculously high definition texture to allow this detailed to be seen in the environment. So to fake it, AMT will be used.

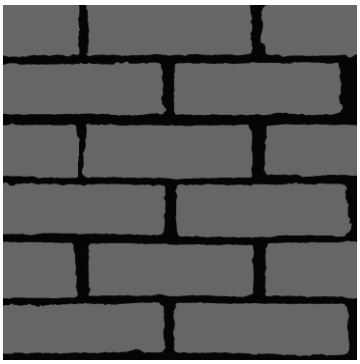
The technique AMT is implemented in three distinct steps. The first is to create a Micro-Texture palette which is a single large bitmap that contains multiple micro-textures which is similar to Rage's subset textures. The second is to produce a monochromatic texture map for a texture, similar to a bump map, which will map out where each microtexture will be placed. With the art assets produced, the last step will be to create shaders that will activate at a certain distance from the object like mipmapping. The shaders will splatter the micro-textures onto the base texture and as the user gets closer to the object, the micro-texture's opacity will increase until the micro-texture transitions completely and replaces the would be blurry stretched base

texture. The texture map will be used to allow multiple micro-textures to be placed in the pre-determined locations which will result in fine detail being shown on the object as illustrated in Figure 9.

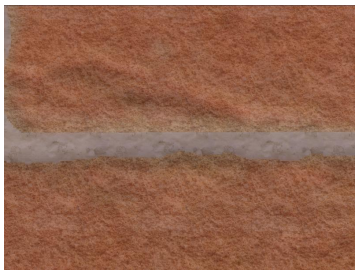
- Micro-Texture Palette. The second entry is of mortar. The third is brick. This example palate could use up to 16 micro-textures, only 4 entries are used at this time. Larger palettes could be used as necessary as well.



- A Brick texture



- AMT Map. This will pre-determine where brick or mortar micro-textures will be used.



- Micro-Texturing Application. This is an example of the texturing technique at half opacity of the micro-textures. The fine detail starts to show as the user gets closer.

Figure 9. Micro-Texture Placement

3.2 First Steps

The first steps I had to make was to choose a language to use. Most 3D games are programmed in C or C++ because of their low level capabilities allowing games to squeeze every ounce of power from a platform. Since I want to demonstrate the low amount of processing power needed and the small memory space allocation required from using this new technique, I chose to use Java and the cross compatible library libgdx which is maintained by Badlogic Games. Libgdx allows me to write source code that can be run on Windows, Mac, Web Browsers, and the Android Platform. Since memory is very limited on mobile devices, I think game engines on those platforms could greatly benefit from this technique.

Libgdx uses the Lightweight Java OpenGL library, lwjgl, which is an OpenGL wrapper for coding in Java. OpenGL is run in native C++ code, so the wrapper allows a Java programmer to call functions in Java to simplistically set up the necessary OpenGL functions. OpenGL also follows my focus on cross compatibility. With these libraries, a simplistic 3D engine will be constructed. The focus of it will just to display the AMT technique, and judge performance solely on that. For shaders, OpenGL uses the OpenGL Shading Language, GLSL, which composes of a vertex shader and a pixel shader. The AMT code will mostly be housed in

the pixel shader since everything is based upon values contained in the texture map and micro-texture palette.

3.3 Construction

For my first foray into 3D programming, constructing the basic engine was trickier than I thought it would be. Constructing the 3D camera and placing objects into the world worked perfectly at first. Since I planned on making a demo for users, I knew I had to somehow use a 3D modeling application to construct the rooms. Since I already had experience with the application, I chose to use Autodesk's Maya 3D modeling software. The tricky part was importing the environments into my program. I ended up having to construct a custom object loader which formatted the 3D scene data into the Vertex Buffer Object, which houses all of the vertex and texture data.

When rendering a 3D environment with multiple objects there are multiple ways you can utilize the Vertex Buffer Object. "A Vertex Buffer Object (VBO) is a Buffer Object which is used as the source for vertex array data" [12]. You can use multiple VBOs which has an advantage if the environment requires objects to changed, added, or removed from the Vertex Array Object, which houses all of the data that will be fed into the graphical shaders. Since my world is static, I chose to only use one VBO which makes populating the Vertex Buffer Array very easy. Another advantage was when I wrote a custom object loader for the scenes made in Maya, I could format the data at the time of execution, which takes away computational time during runtime. According to OpenGL's specification, "As a general rule, you should use interleaved attributes wherever possible" [12]. Interleaving is the formatting of data in the VBO where instead of all of the vertex data, then the normal data, and then the textcoord data, you

place a vertex's data next to each vertex in the array. Figure 10 explains the different ways to populate a VBO. When the object loader is run at the time of execution, it interleaves all of the data and places it correctly into the VBO before any interactivity from the user happens.

VBO Data Organization

(VVVV)(NNNN)(CCCC)	• Three VBOs. One for vertices, one for normals, and one for texture coordinate data
(VVVVNNNNCCCC)	• A single VBO. Listing the vertices, then the nornals, and then the texture coordinates.
(VNCVNCVNCVNC)	• Interleaved Data. Each vertice with it's associated normal and texture coordinate in sequence.

Figure 10. Formats of VBO [13]

3.4 Hardware and Software

This section shows the hardware and software used for this thesis.

3.4.1 Hardware

All benchmarks were run on the system detailed below:

Windows PC	**CPU:** Intel Core Duo E8400 @ 3.0 Ghz **RAM:** 4GB DDR2 @ 800 Mhz **HDD:** 1 TB @ 7200 rpm **Graphics:** Sapphire HD 6870 1GB GDDR5 PCIE

3.4.2 Software

All coding, editing, and art assets were created by the software below:

Software	**Operating System:** Windows 7 64bit **IDE:** Eclipse Indigo Service Release 2 **Photo Editor:** Gimp 2.6 **3D Modeling:** Maya 2012 Student Edition

IV. Evaluation

The evaluation section will go over how the tests for AMT will be measured. Afterwards, the results will be presented and analyzed with helpful visual examples.

4.1 Testing Benchmarks

Since the focus of AMT is to enhance the fidelity of textures in a real time setting, frames per second will be the main point of contention. There is no point of this technique if it requires a massive amount of overhead. So to test the technique under heavy load, there will be multiple settings for the testing demo. The demo will consist of a room with objects that have micro-textures placed on them. The rooms will go up in complexity by having more objects within them as demonstrated in Figure 11. The objects will have the same vertex counts but be smaller in scale. The widely used software Fraps will be used to obtain the average frame rates during runtime.

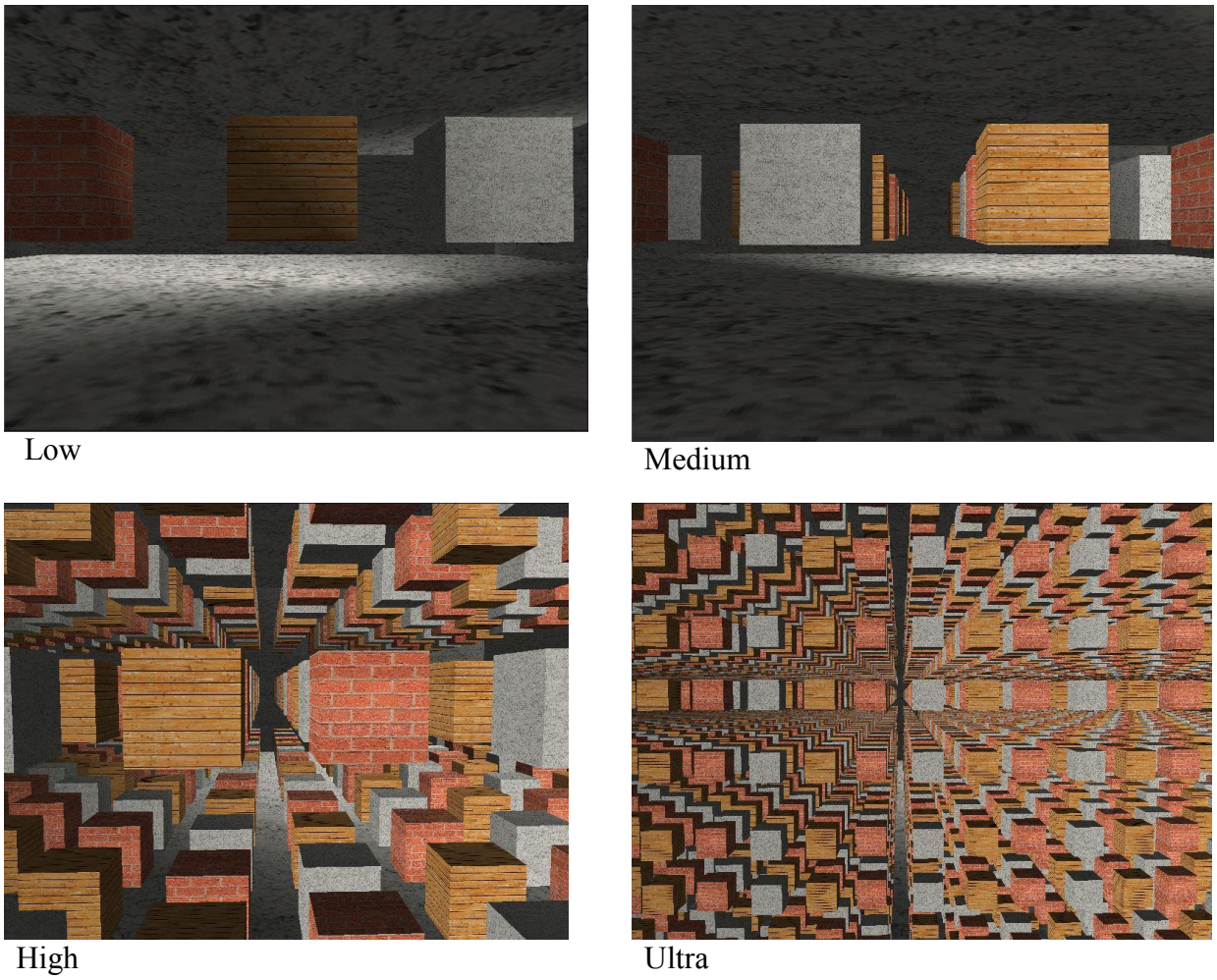


Figure 11: Testing Environments

There will be three shaders that will be tested for average frames per seconds; Plain Shader, Bump Shader, and AMT Shader. The plain shader is a shader that only displays the objects and their textures to the screen, thus illustrating the bare minimum in graphical fidelity and the highest performance possible with the hardware the technique is being tested on.

The bump shader is a basic normal mapping implementation. It takes the lighting and height maps, and creates an illusion of roughness upon the textures. The bump shader is similar to a beginning 3D programmer's first graphical shader. It utilizes a modern technique, but no where near the fidelity of modern 3D engines.

The AMT shader is an iteration of the bump shader. The final AMT shader for the graphical technique started as the bump shader and then was modified to include the AMT technique. Comparing the performance from the AMT shader to the bump shader will best illustrate the extra performance hit that the new graphical technique will take.

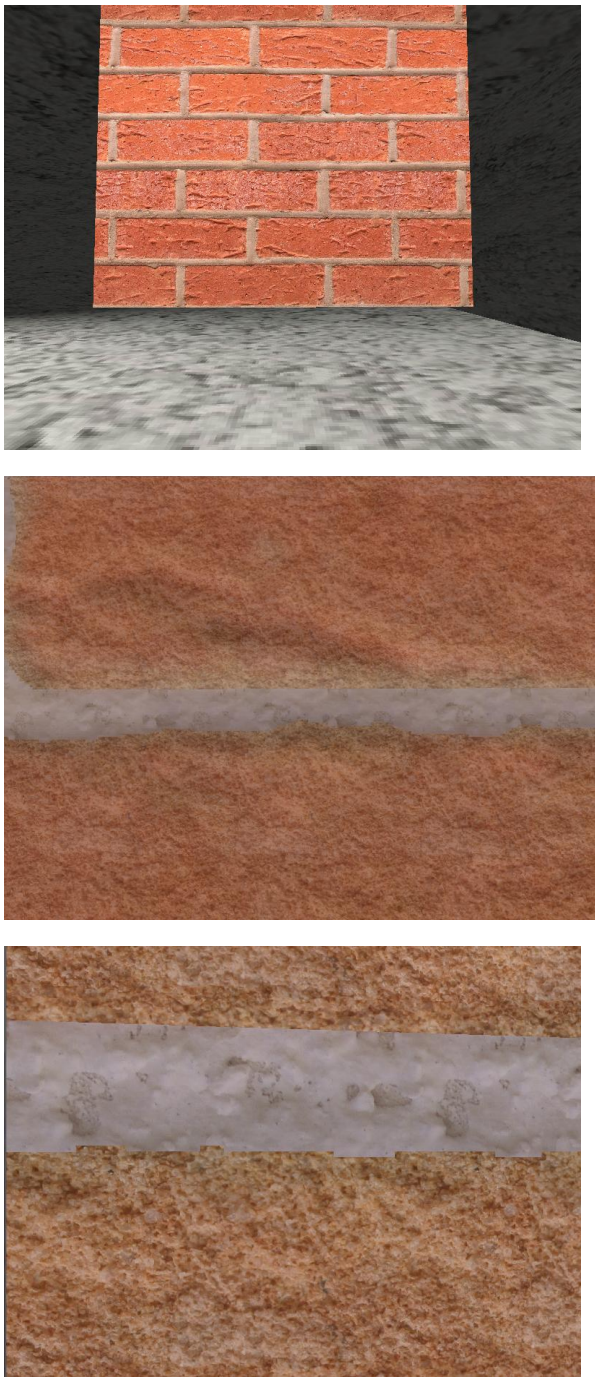
4.2 Results

This section will present the results in two distinct ways. First the fidelity will be discussed to describe the overall presentation of AMT which can only be measured by a user's opinion of how an object looks. The other test will be the framerate benchmark, will describe how well the technique works at runtime and different graphical loads.

4.2.1 Fidelity

The extra fidelity on the textures up close can be seen in Figure 12. The technique shows that with proper micro-texturing, pixelation will never be shown within the 3D engine. Moving the camera up close to the object has a nice transition that blends the original texture into the micro texture so the eye refocusing illusion works out pretty well. The one problem which was noticed was the blur used for transitioning into the AMTs, especially if the camera is flush with a wall and it is showing the full transition from the micro-textures to the normal texture. The blur effect can sometimes end abruptly displaying a line between shading techniques and breaking the illusion as shown in Figure 13. This could be remedied with the use of a more complex blurring technique or better lighting code. Displacement maps, which are similar to bump maps though they actually add geometry to the object, could also hide the transition completely by adding a

lot more detail to the polygons rather than my plain cubes. This extra detail would distort the line where the AMT starts, hiding the obvious transition.



- Far Away. AMT is not activated at all. Only base texture is seen.
- Transitioning. Base texture is blended with micro-textures.
- Up Close. Micro-textures at full opacity.

Figure 12: AMT Technique at Different Distances

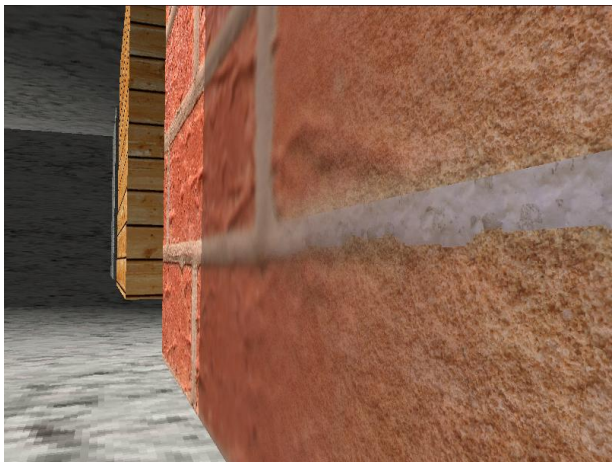


Figure 13: Noticeable Blur Transition

4.2.2 Performance

The results of the average frame rates are listed in Figure 14. Overall, the technique showed a performance loss of roughly 20% when compared with the similar bump shader without the technique implemented. Figure 15 illustrates that there is a higher performance loss when the micro-texturing is taking up all of the screen space which means the performance hit is mostly coming from the pixel shader. Another piece of information to note is the performance loss based upon the complexity of the space. As shown in Figure 16, the performance drop dips as the scene gets more complex. This is a great trend to have since modern 3D engines are showing environments hundreds of times more complex than my test environments. This means that AMT takes the greatest hit in performance when the whole screen is taken up by the micro-texture swaps. When this happens though, only a couple of objects will be shown on screen which means the loss in performance will most likely not be noticed.

	Low	Medium	High	Ultra	Avg Performance Loss (%)
Plain Shader	4608.6	1766.6	222.2	9.3	
Bump Shader	4308.7	1574.2	187.5	8	11.75 %
AMT Shader	2667.1	1096.3	158.9	7.8	31.17 %

Figure 14: Average Frames Per Second and Average Performance Loss

	Frames Per Second	Avg Performance Loss (%)
Far Away	2693.1	
Up Close	747.6	72.2 %
Up Close, Half Screen	1272.1	52.8 %

Figure 15: Performance Loss based on Allotment of Screen

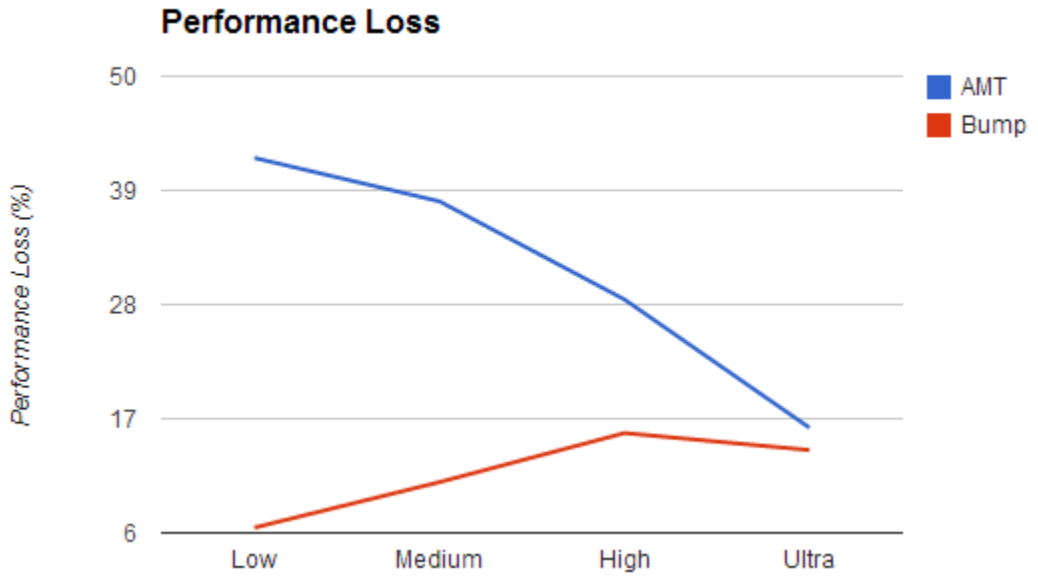


Figure 16: Performance Loss according to Environment Complexity (Lower is better)

V. Conclusion

Adaptive Micro-Texturing accomplishes the goal I set out to tackle. When textures are viewed up close in a 3D environment, instead of showing stretched and blurred pixels an enlarged texture, the screen shows crisp micro-textures replacing the base texture. The technique gives the illusion of a much higher resolution texture than the one that is actually used. This section will discuss the conclusions that resulted from the experimentations. Included will be the effectiveness of the technique to achieve the overall goal of enhancing textures up close, the limitations of the technique, and the future work that can be done to enhance and optimize AMT.

5.1 Effectiveness

Visually the technique is very effective as demonstrated in Figure 12. It greatly enhances the fidelity of the texture when viewing an object up close by removing the pixelation so prevalent in modern 3D engines. Though I was only able to test it in simplistic 3D environments, because of artistic skill and time constraints, a more complex environment created by a professional 3D artist would most likely illustrate the technique even better.

Performance wise, AMT works very effective as well. When it comes to interactive 3D rendering engines, frame rate is king. A developer targets as a minimum, 30 frames a second, which is the speed the human eye renders at. Keeping an engine at 60 frames a second, which is the average refresh rate of monitors, presents a silky smooth experience which never dips down to a frame rate that is noticeable to the user. Figure 14 demonstrates a very respectable frame rate is achieved. Even more important, Figure 16 illustrates how the loss in performance lessens as a result of more complexity in a 3D environment. This will mean that the technique should

work even better in real world settings when millions of vertices are on screen and hundreds of textures are used.

5.2 Limitations

The main limitation of AMT is when the texture itself is defined by the details in the bitmap rather than the material it is meant to look like. While the majority of a 3D engine is spent on rendering walls, floors, and basic objects found in the real world, there are objects that are defined by the content of them like a poster, photograph, or sign. When it comes to these contextual objects, no amount of micro-texturing would help. They must be rendered with a high resolution texture. Since AMT requires less memory space for most of the textures, more space can be used for these objects that would gain no benefit from the AMT technique.

5.3 Future Work

I definitely see the AMT technique as very much in its infancy. The edge of the blurring effect is easily noticeable when looking down a long wall as seen in Figure 12. I'm sure this can be made to look much better just by using perhaps a better technique than the simple gaussian blur I added to the edge. AMTs would also look alot better with the additions of Normal Maps, perhaps displacement maps, and a much more dynamic lighting engine. Since 3D rendering is still new to me, I don't have the capabilities at the moment to build into the demo more contemporary 3D engine enhancements. This thesis demonstrates a new way of handling the pixelation that plagues every video game made with a 3D rendering engine and provides a new way to remedy some of those limitations.

References

[1] Battlefield: Bad Company 2. V 1.5
EA Digital Illusions CE. Electronic Arts. March 2, 2010

[2] Blinn, J. F. (1978, August). Simulation of Wrinkled Surfaces. *SIGGRAPH '78*, 286 - 292. Retrieved October 10, 2012, from http://research.microsoft.com/pubs/73939/p286-blinn.pdf

[3] Bloom, Charles (2000). Terrain Texture Compositing by Blending in the Frame-Buffer from http://www.cbloom.com/3d/techdocs/splatting.txt

[4] Case, L. (2010, July 7). What DirectX 11 is, and What It Means to You | Maximum PC. *Maximum PC*. Retrieved October 5, 2012, from http://www.maximumpc.com/article/features/directx_11_deconstructed

[5] Fisher, R. (1999, September 12). Edinburgh Online Graphics Dictionary. *Informatics Homepages Server*. Retrieved October 28, 2012, from http://homepages.inf.ed.ac.uk/rbf/GRDICT/grdict.htm

[6] Glasser, N. (2005, April 23). Texture Splatting in Direct3D - Game Programming - Articles. *GameDev.net Developer Community*. Retrieved October 10, 2012, from http://www.gamedev.net/page/resources/_/technical/game-programming/texture-splatting-in-direct3d-r2238

[7] History of OpenGL. (n.d.). *OpenGL - The Industry Standard for High Performance Graphics*. Retrieved October 28, 2012, from http://www.opengl.org/wiki/History_of_OpenGL

[8] Nikodym, Tomas. (2010, May 20). *Ray Tracing Algorithm For Interactive Applications.* Archive theses CTU. *Czech Technical University in Prague*. Retrieved October 28, 2012, from https://dip.felk.cvut.cz/browse/pdfcache/nikodtom_2010bach.pdf

[9] OpenGL Overview. (n.d.). *OpenGL - The Industry Standard for High Performance Graphics*. Retrieved October 28, 2012, from http://www.opengl.org/about/

[10] Sink, Keith. *DirectX 8 and Visual Basic development*. Indianapolis, Ind.: SAMS, 2002. Print.

[11] Texture Filtering with Mipmaps (Direct3D 9). (2012, October 16). *MSDN – Explore Windows, Web, Cloud, and Windows Phone Software Development*. Retrieved October 28, 2012, from http://msdn.microsoft.com/en-us/library/windows/desktop/bb206251(v=vs.85).aspx

[12] Vertex Specification - OpenGL.org. (n.d.). *OpenGL - The Industry Standard for High Performance Graphics*. Retrieved October 5, 2012, from http://www.opengl.org/wiki/Vertex_Specification

[13] Vertex Specification Best Practices - OpenGL.org. (n.d.). *OpenGL - The Industry Standard for High Performance Graphics*. Retrieved October 5, 2012, from http://www.opengl.org/wiki/Vertex_Specification_Best_Practices

[14] Waveren, J. v. (Director) (2009, August 6). id Tech 5 Challenges From Texture Virtualization to Massive Parallelization.*Beyond Programmable Shading*. Lecture conducted from Siggraph, New Orleans. Retrieved October 8, 2012, from http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf

[15] Wikipedia. 2010. *Bump map demo full*. [PNG] Retrieved from http://upload.wikimedia.org/wikipedia/commons/0/0a/Bump-map-demo-full.png

[16] Wikipedia. 2006. *Normal map example*. [PNG] Retrieved from http://upload.wikimedia.org/wikipedia/commons/3/36/Normal_map_example.png

[17] Williams, L. (1983). Pyramidal Parametrics. *Computer Graphics*, *17*(3), 1 - 11. Retrieved October 10, 2012, from http://staff.cs.psu.ac.th/iew/cs344-481/p1-williams.pdf