Project Number: EOA-4901

Real-Time Preview of 3D Image Quality Settings

Sponsoring Agency:
ATI Research Inc.
3D Applications and Research Group (3DARG)

A Major Qualifying Project Report
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

by

William A. Pfeil

Date: January 10, 2006

Approved:

Prof. Emmanuel Agu, WPI Advisor

Mr. Daniel Ginsburg, ATI 3DARG

Mr. Callan McInally, ATI 3DARG Manager

Abstract

ATI Technologies is a leading designer of video cards for computers. This ATI-sponsored project involved implementation of a 3D demo which showcases different 3D image quality settings in real-time. Many tradeoffs were necessary to fit within certain design constraints, including but not limited to: release size, setting-change speed, and quality of demonstration. The report includes unclassified details of code architecture, size/speed/quality/other tradeoffs, and setting implementation details.

Table of Contents

Abstract ii
List of Figures v
List of Tables v
Acknowledgements vi
1. Introduction 1
2. Background 4
2.1. DirectX / Direct3D 4
2.1.1. Meshes 4
2.1.2. 3D Visual Effects 5
2.1.3. Textures 6
2.1.4. Useful Direct3D Libraries 7
2.1.4.1. DXUT 8
2.1.4.2. D3DX 8
2.2. Antialiasing 9
2.3. Mipmaps 10
2.4. Texture Filtering 13
2.5. Normal Mapping 16
2.6. Windows Vista 17
3. Requirements 18
3.1. 3D Graphics API 18
3.2. Target Operating System 18
3.3. Target Platforms 19
3.4. Target Graphics Hardware 19
3.5. Release Size Constraint 19
3.6. Loading Time Constraint 20
3.7. Visual Quality Constraint 20
3.8. Window Size Constraint 20
3.9. 3D Settings for Demonstration 21
4. Programming Decisions 22
4.1. Integrated Development Environment 22
4.2. Useful Direct3D Libraries 22
4.2.1. DXUT 23
4.2.2. D3DX 23
4.3. Commenting System 24
4.4. Coding Standard 25
5. Tradeoffs – Size vs Loading Speed vs Quality 26
6. Implementation 31
6.1. CCP Design Goals 31
6.2. Dual Scene Display 32
6.3. 3D Settings 34
6.3.1. Antialiasing 34
6.3.2. Temporal Antialiasing 35
6.3.3. Adaptive Antialiasing 36
6.3.4. Anisotropic Filtering 37
6.3.5. Advanced Anisotropic Filtering 38
6.3.6. Catalyst AI 38
6.3.7. Mipmap Level of Detail Bias 39
6.3.8. Geometry Instancing 39
6.4. Resource Management 39
6.5. Singletons 40
6.6. Art Assets and Extensibility 41
6.6.1. Art Assets 41
6.6.2. Initialization (.ini) File 41
6.7. Camera Paths 45
6.8. Scene Effects 46
6.8.1. Reflection 46
6.8.2. Fence 50
6.8.3. Normal mapping 52
6.8.4. Sky Effect 54

6.9. Debugging Methods 56
6.10. Communication Overview 58
7. Conclusion 61
8. References 62
Appendix A. CCP Build / Release Overview 64
A.1. Solution Configurations 64
A.1.1. Dashboard / Non-dashboard Builds 64
A.1.2. Win32 / Win64 Builds 65
A.1.3. Debug / Release Builds 65
Appendix B. DXT Compression 66

List of Figures

Figure 1: The Catalyst Control Center (CCC) with the original 3D preview scene....1
Figure 2: A mesh in wireframe mode and the same mesh with all triangles shaded....5
Figure 3. Textures used to encode image data and purely numerical data (surface properties) of a stone wall....7
Figure 4: Aliased and antialiased lines....9
Figure 5: Multisampling antialiasing sampling technique (“4X multisampling” shown)....10
Figure 6: Mipmap levels for a texture....11
Figure 7: Mipmap filtering technique comparison....12
Figure 8: Comparison of texture filtering methods....14
Figure 9: Improvement of quality through anisotropic filtering over linear filtering....16
Figure 10. Multiple solution configurations in VS2005 Professional....19
Figure 11: Zoomed view of uncompressed light map, DXT1-compressed light map, and difference image....29
Figure 12: Sample .ini file ‘ccpconfig.ini’....42
Figure 13: Fountain reflection effect....47
Figure 14: The orientations of the original camera used for the camera fly path, and the reflected camera....48
Figure 15: Fountain view, with reflection texture shown onscreen for debugging....49
Figure 16: Clip map used for the courtyard fence....51
Figure 17: Fence with no adaptive AA versus high quality adaptive AA....52
Figure 18: Screenshots with diffuse lighting only and with specular lighting & normal mapping....53
Figure 19: The courtyard sky....54
Figure 20: The different color components of the one texture used for the sky effect....55
Figure 21: Mipmap debugging mode enabled in the courtyard scene....57
Figure 22. CCP application communication flow chart....59
Figure 23: The new CCC in Windows Vista....61
Figure 24: Supported build configurations....64

List of Tables

Table 1: Size / loading speed tradeoff chart....26
Table 2: Summary of release executable statistics, using best options from Table 1....29
Table 3: Supported AA / ASD combinations....37
Table 4: Colors scheme for mipmap debugging mode....58

Acknowledgements

I would like to thank: Callan McInally (3DARG manager) and Emmanuel Agu (Worcester Polytechnic Institute project advisor) for connecting me with this development opportunity, Dan Ginsburg (ATI supervisor) for the majority of the help and guidance needed for this project, Abe Wiley (ATI lead artist for this project) for supplying a great scene for the demo, and the rest of the 3DARG, driver team, artist team, and other project contributors for their work, support, and contributions to my knowledge and experience.

1. Introduction

Since September 2004, ATI has made publicly available an application entitled the *Catalyst Control Center* (CCC). This application is provided to complement ATI's *Catalyst* video accelerator card drivers, enabling ATI card owners to adjust multiple monitor settings, 3D display settings, color correction settings, video settings, overclocking settings, etcetera, from a unified user interface. The CCC is publicly available on ATI's web site [4].

Figure 1: The Catalyst Control Center (CCC) with the original 3D preview scene

The CCC's ability to allow the user to adjust 3D display settings is especially useful for gamers who would like fine-tuned control over settings such as anisotropic filtering, anti-aliasing, and mipmap level of detail. This allows a user to trade off in-game performance for visual quality, and vice versa. One of the features that make the CCC such a useful tool for making these adjustments is the small *Catalyst Control Preview* (CCP) window (shown in Figure 1) that shows a 3D scene with user-defined 3D settings applied. This allows users to see if 3D setting changes they are making are worthwhile, and also shows exactly where in a 3D scene those setting changes make a difference.

The current implementation of the CCP application is outdated, does not display a very impressive, eye-catching 3D demo, and most importantly, 3D setting adjustments are not easy to visualize in the preview window as settings are altered. The current 3D preview scene shows a racecar driving along a desolate road. The new preview should be niche-independent in subject matter, as opposed to gamer-centric. In addition, the current implementation of the preview window shows only one scene – for the new preview, it is desirable to have two copies of the same scene be displayed side-by-side. The scene on the left would showcase the "current" 3D display settings, and the scene on the right would showcase the "requested" settings. The requested settings are adjusted by moving slider bars in the CCC. When an "Apply" button is pressed in the CCC, the new settings would manifest themselves in the "current" settings scene. It is believed that this is a more intuitive way to adjust settings: a side-by-side comparison between "before" and "after," instead of just one large view of "after."

As for release requirements, the CCP needed to have a small download size and start up and change settings very quickly, while preserving as much visual quality as possible. The CCP application is targeted for Windows Vista and must run on *CPU*s[1] of minimum speed 800MHz and on ATI R300 series *GPU*s[2] as a minimum.

---

[1] *CPU*: Central Processing Unit
[2] *GPU*: Graphics Processing Unit

This project involved making and/or testing the above changes to the CCC and/or CCP, where applicable. In summary, the project involved the following tasks:

- Building a simple 3D engine from the ground up which can handle multiple scenes and windows
- Allowing for fast application of every 3D setting desired for the CCC
- Integrating new art assets provided by ATI's team of artists
- Making sure all setting changes were showcased effectively
- Analyzing release size / loading time / visual quality tradeoffs
- Testing and debugging the CCP application in Windows Vista, on low-end CPUs and GPUs

2. Background

This section provides background on the platforms, software, application programming interface (API), and fundamental graphics knowledge required for the development of the CCP application.

2.1. DirectX / Direct3D

DirectX is a suite of APIs designed by Microsoft in order to standardize integration of 2D and 3D graphics, sound, input, and networking into Microsoft Windows applications. The APIs allow for a common way of communicating with underlying multimedia hardware components. Most relevant to this project is the *Direct3D* (D3D) API, which is the subset of the DirectX API that deals with 3D graphics programming.

2.1.1. Meshes

At the risk of oversimplification, a 3D scene is made up primarily of geometry. A position in a 3-coordinate system (3D space) is known as a *vertex*. A collection of 3 *vertices*[3] define a triangle, commonly known as a *primitive*. A collection of primitives comprise a 3D shape, also known as a *mesh*. An example of a mesh can be seen in Figure 2.

[3] *vertices*: plural of vertex

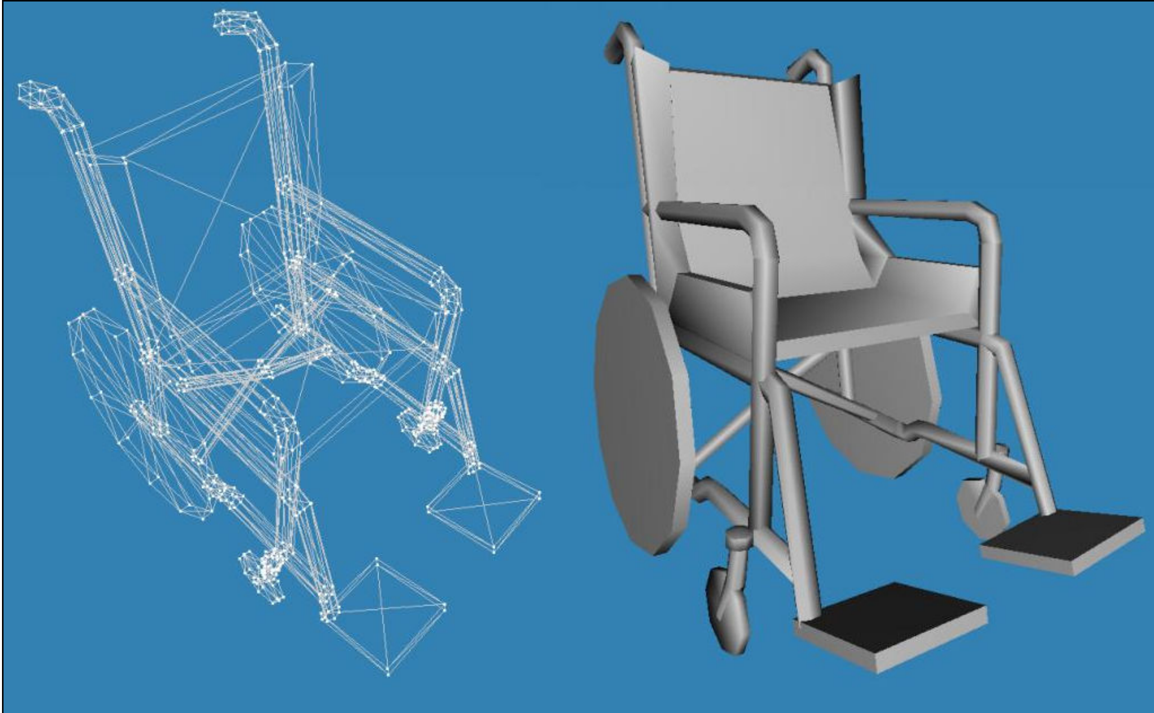

Figure 2: A mesh in wireframe mode (left) and the same mesh with all triangles shaded (right)

Meshes can encode much more information per-vertex than just 3D position. As part of the Direct3D API, Microsoft provides the X mesh format (*.X), which allows for arbitrary per-vertex information, animation information, *texture* and *effect* references, and other arbitrary annotations as desired. More information about the X format can be found at [3] or in the DirectX Software Development Kit (SDK) documentation. The X mesh format is used for the meshes in the CCP application.

2.1.2. 3D Visual Effects

The programmable pipeline built into modern graphics cards allows developers to run arbitrary computations on a *per-vertex* and a *per-pixel*[4] basis using the GPU. Developers can then create fancy special effects and coloring schemes by writing a program that will operate on each vertex ("vertex shader") and a program that will operate on each pixel ("pixel shader"). These shaders are loaded onto the graphics card

[4] *pixel*: the fundamental unit of a picture ("picture element")

where they perform their computations over and over to produce a 3D scene in real-time. Before the emergence of programmable pipelines, graphics cards allowed only fixed-function processing of vertices and pixels on the GPU, meaning developers could only process vertices and pixels using available functions hardwired into the silicon of the graphics card by hardware designers.

It is important to note that the fixed-function pipeline may currently still be used in place of either a vertex shader or pixel shader if desired. It is also important to note that the term “shader” is a misnomer; shaders can do much more than just “shade.” Finally, it should be mentioned that the term “fragment shader” is sometimes used in place of “pixel shader.”

Direct3D provides an encapsulation of vertex shaders, pixel shaders, and pipeline state in one *effect* file. These files are called *FX effects*. The FX effect format allows the developer to decouple the application’s effect management code from the shaders, and from the render state, sampler states, and other states required by those shaders. The FX format also allows for specification of multiple *techniques*, which allow for different shaders to be specified for use in different situations (such as for fallbacks for older hardware). That is, different techniques can specify different pipeline states, or different shaders altogether, or even no shaders at all (fixed-function processing can be enabled from an FX effect as well). In the CCP application, every vertex and pixel shader pair is wrapped inside an FX effect file.

2.1.3. Textures

Textures are images that can be applied to a mesh surface. In a more general sense, textures are used as a lookup table in many different real-time graphics applications. Textures may be used to store surface properties, animation information, or countless other useful pieces of information. See Figure 3 for an example.

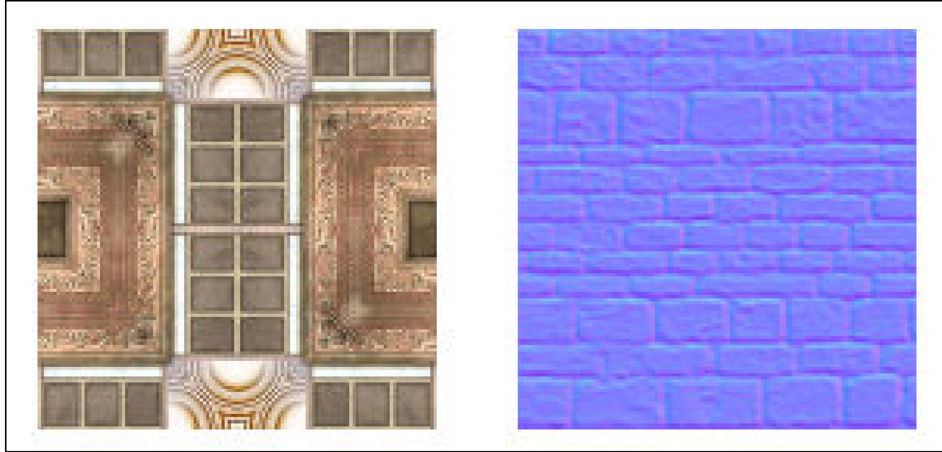

Figure 3. Textures used to encode image data (left) and purely numerical data (surface properties) of a stone wall (right)

The *D3DX* library (see section 2.1.4.2.) supports loading of many different texture formats: .BMP, .DDS, .DIB, .HDR, .JPG, .PFM, .PNG, .PPM, and .TGA. The Windows-specific DDS (DirectDraw Surface) format was chosen for the CCP application since it has a few robust, effective compression methods available, has the ability to store pre-computed *mipmap* chains (see section 2.3.), and supports many image formats ("A8R8G8B8" as an example – 8 bits each of *A*lpha, *R*ed, *G*reen, and *B*lue color information). Effective compression was important to meet size constraints while maintaining high visual quality. The DDS file format supports *DXT*[5] texture compression, which boasts a few different compression options (See Appendix B for DXT compression details). Pre-computed mipmap chains help achieve a fast load time (in some cases). Compression savings and mipmap chain space / load time tradeoffs are discussed in section 5..

2.1.4. Useful Direct3D Libraries

In any field of development, the wheel should not be reinvented. This is especially true with software development. When beginning the CCP application, the

[5] *DXT*: an efficient texture compression method originally developed by *S3 Graphics, Ltd.*

DirectX Utility Toolkit (DXUT) and *D3DX* were a couple of common tools that were considered as candidates for shortening development time.

2.1.4.1. DXUT

DXUT is a framework that abstracts much of the common code required to get a DirectX application up and running. It has simple methods allowing for window creation, *D3D device*[6] creation, timing, camera classes, a suite of *GUI*[7] tools, and *callback*s[8] for many kinds of D3D device events and window messages such as keyboard and mouse input messages, repaint messages, and so on.

2.1.4.2. D3DX

D3DX is a library produced for use with D3D containing many useful utilities for common 3D graphics operations. The library includes structures, functions, interfaces, and macros for loading and/or manipulation of animations, fonts, meshes, shaders, textures, effects, and much more, as well as math functions for manipulating matrices, vectors, quaternions,[9] methods of interpolation, etcetera. Although DirectX-specific formats like .X meshes, .FX effects, and .DDS textures are open formats that can be parsed with custom code [1], there is usually no need to ignore the D3DX library; it is extremely handy in dealing with common operations on these formats.

---

[6] *D3D device*: a D3D software interface to an underlying graphics hardware device
[7] *GUI*: Graphical User Interface
[8] *callback*: a registered function that will be invoked upon some specific event
[9] *quaternion*: a 4-tuple used as an efficient way to represent an orientation in 3-space

2.2. Antialiasing

Antialiasing describes a technique used to correct aliasing artifacts (or "smooth out jaggies") that are sometimes apparent on line edges. Examples of aliased and antialiased lines, with zoomed-in versions, are shown in Figure 4.

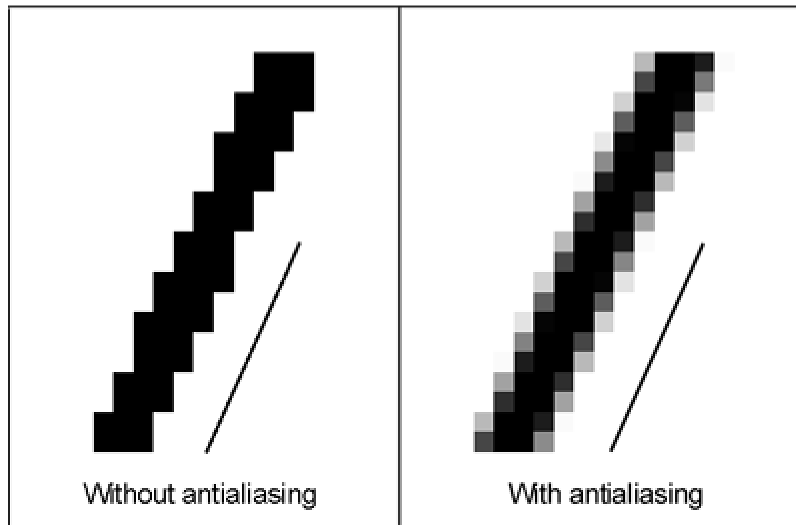

Figure 4 [23]: Aliased and antialiased lines

The antialiased line can be produced in one of many ways, with each way having its own advantages and disadvantages depending on the situation. One method is *supersampling*, in which an image is rendered at a higher resolution (or equivalently, each pixel is divided into sub-pixels), and then the image is resampled down. For example, if one pixel were subdivided into four ("4X supersampling"), the color of each subpixel could be determined via *Bresenham's algorithm.*[10] The four subpixels could then be averaged back together to produce one pixel for rendering.

Supersampling is computationally intensive. In the example above, four times as many pixels must be evaluated. A faster and more popular antialiasing method is *multisampling*. As such, multisampling is supported by most newer graphics cards. Multisampling uses the pixel color, along with multiple sample points within the pixel (arranged in different patterns depending on the GPU vendor), to determine the final

[10] *Bresenham's algorithm* is a common line-drawing algorithm

pixel color. Instead of subdivision into subpixels, multisampling multiplies the color of the pixel by the percentage of pixel sample points that fall within the border of a primitive. See Figure 5 for a pictorial of the idea – the figure shows a zoomed-in view of a pixel that straddles the border of a primitive, with multiple test sample points within the pixel. With multisampling, no extra pixels are processed, but aliasing artifacts are still reduced.

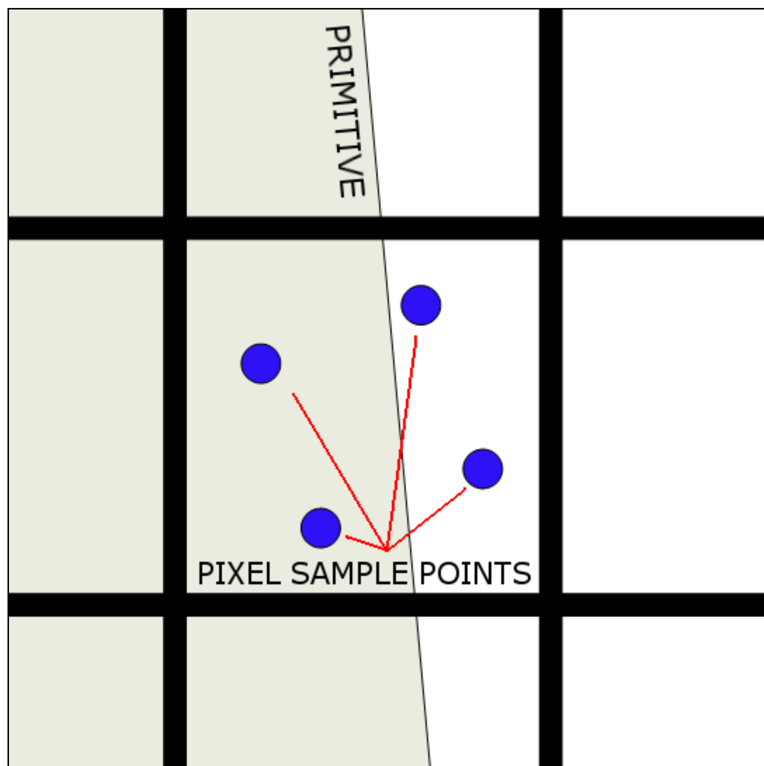

Figure 5: Multisampling antialiasing sampling technique ("4X multisampling" shown)

2.3. Mipmaps

*Mipmap*s are pre-computed collections of bitmaps of different sizes. They are widely used in computer graphics applications to increase texture filtering performance (see section 2.4. on texture filtering). The basic idea is to use a highly detailed version of a bitmap when the detail would be noticeable and a low detail version of the bitmap when high detail would not be noticed. As an example, in 3D games the GPU will choose the

highly detailed bitmap as a texture for a mesh that is close to the player. As the player moves away from the mesh, the GPU will swap in a similar but smaller texture for the mesh, improving performance. If a mesh is very far away, a high detail texture would likely not look any better than a low detail one, due to the finite resolution of computer displays.

Mipmaps are usually square, power-of-2 size textures (e.g. 1x1, 2x2, 4x4, 8x8, etc). Figure 6 shows an example of a bitmap alongside lower-detail versions of the bitmap. Each bitmap is also known as a mipmap "level."

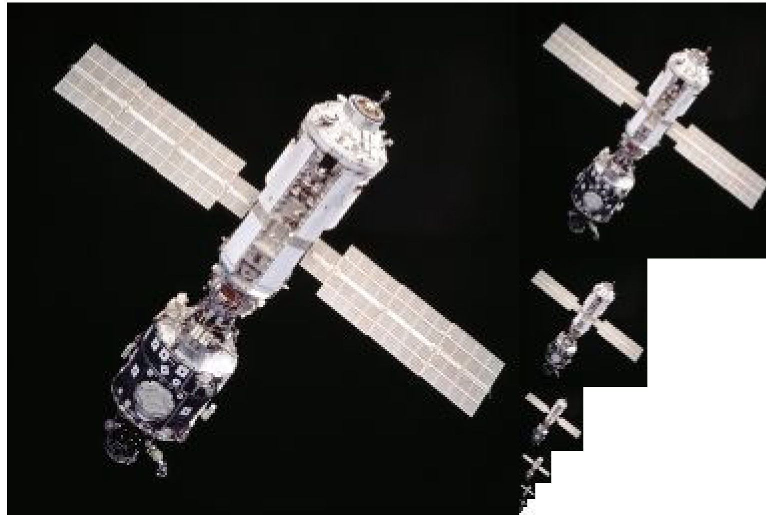

Figure 6: Mipmap levels for a texture

When a mipmap is not the exact same size as a displayed surface (which is often the case), mipmapping can either choose the closest mipmap level to find the color for a pixel, or can blend between the two closest levels. When combined with linear filtering (discussed in section 2.4.), the former is known as *bilinear filtering*, and the latter is known as *trilinear filtering*. Figure 7 shows a comparison of these filtering types. The example (demonstrated using code from [5]) shows a quad[11] with a mipmapped texture applied. The mipmap levels have been individually modified to have different images (in

[11] *quad*: a rectangular primitive

this case, solid colors) so that it can be seen how the levels are being chosen and interpolated.

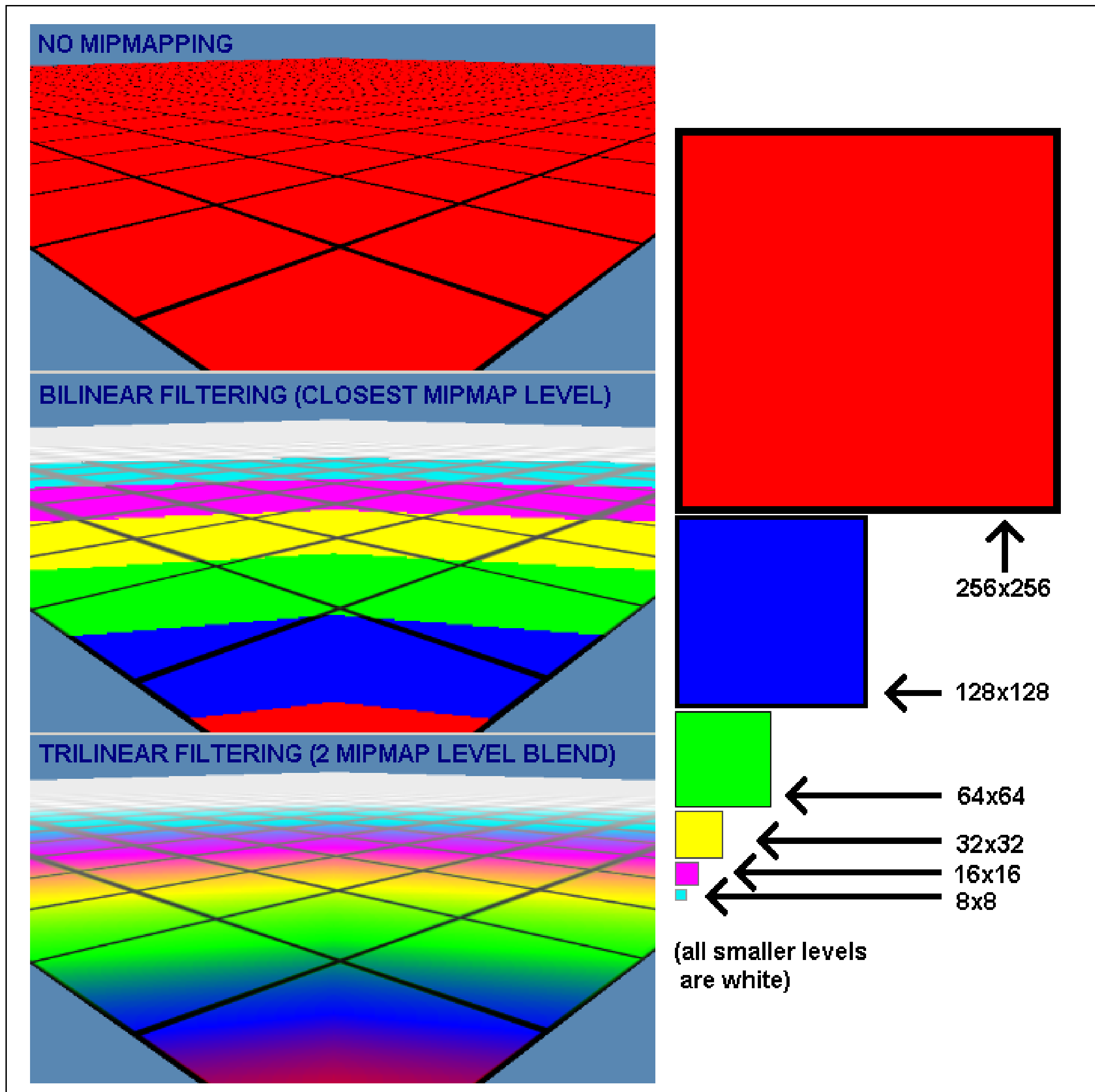

Figure 7: Mipmap filtering technique comparison

Mipmaps are always used in combination with *texture filtering* methods (discussed in section 2.4.).

2.4. Texture Filtering

Texture filtering is a way to account for distortion that occurs when viewing textured surfaces that do not have a 1:1 pixel to texel[12] correspondence. This is a very common occurrence that will take place at most viewing angles. If a texture is too small, then it is expanded using *magnification*. If a texture is too large, then it is shrunk using *minification*. The three texture filtering types supported by Direct3D are: *point* filtering, *linear* filtering, and *anisotropic* filtering. Both minification and magnification may use any of these filtering types. Figure 7 shows a comparison of point, linear, and anisotropic filtering methods used on a checkerboard texture applied to a quad viewed at a grazing angle.

[12] *texel*: the fundamental element of a texture ("texture element")

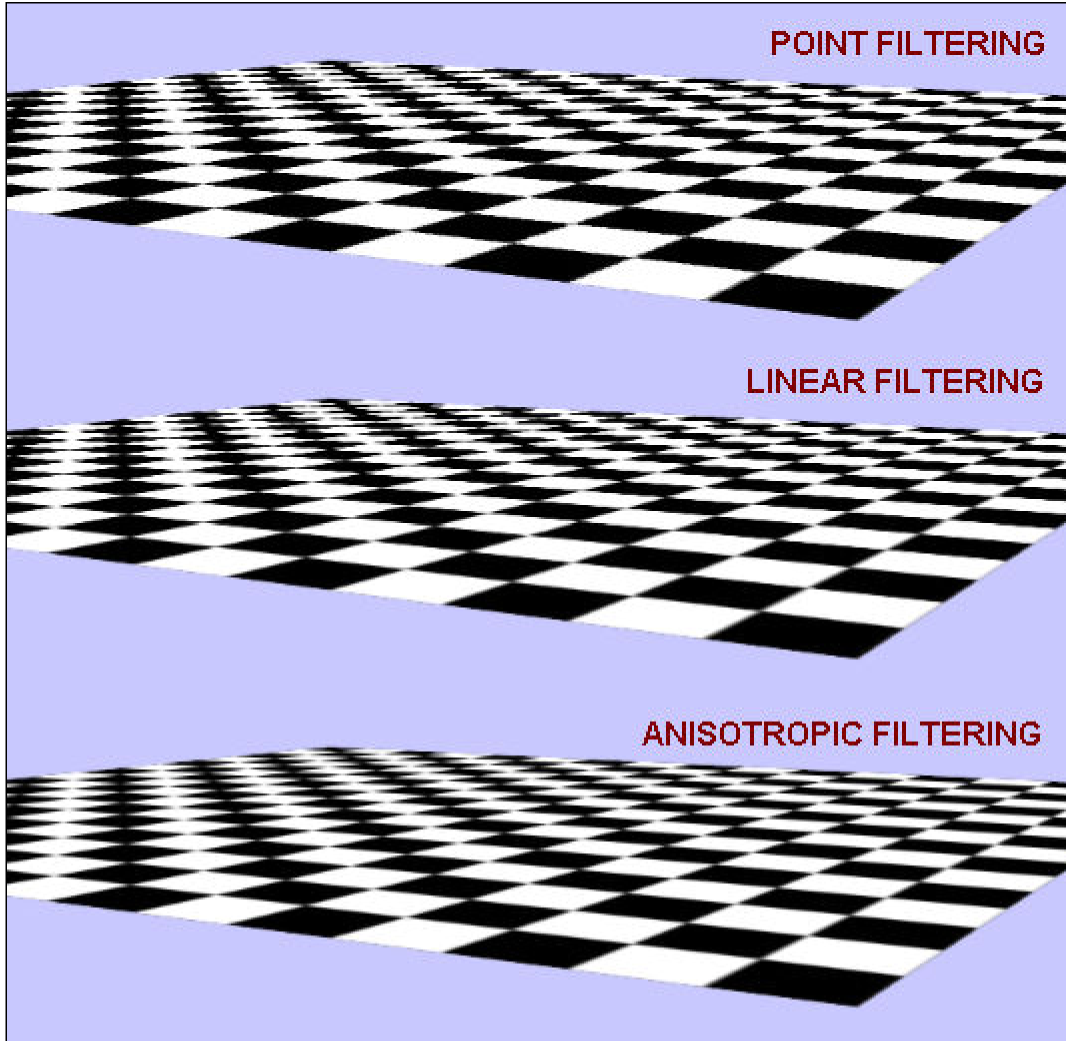

Figure 8: Comparison of texture filtering methods

Point filtering (a.k.a. *nearest point* or *nearest neighbor* filtering) performs no blending of texture colors. This filtering method picks the closest texture element (texel) and uses it as the final color. This produces a blocky-looking texture when the texture is magnified, and produces a low-detail texture when the texture is minified, which is undesirable in most cases.

Linear filtering (a.k.a. bilinear filtering) uses a weighted average of the four nearest texels (weighted according to distance from the sample point) to produce the color of one pixel. If used in conjunction with mipmapping, and the *two* nearest mipmap levels are sampled, producing a weighted average of eight texels, this is known as

trilinear filtering. Bilinear and trilinear filtering produce better image quality than point filtering, but can still produce blurred images.

Anisotropic filtering produces a higher-quality image than other filtering methods, but is more computationally intensive. The filtering method is related to mipmapping. One deficiency of pure mipmapping can be understood through the following example. Assume no texture filtering were available. Now, imagine a mipmapped 64x64 texture mapped to a quad, facing the viewer. Pretend the quad happens to be rendered 64 pixels wide and 64 pixels high. In this case, the texture will appear fine. Now imagine that the quad is tilted backwards, so that a 32x64 quad (approximately) is shown. One could use a 64x64 mipmap level to cover the quad. The problem with this is that, since only 32 rows of pixels are needed, 32 of the 64 rows of pixels in the mipmap level will be discarded. Since the rows that are discarded depend on the angle of the quad, an undesirable *flickering* effect will be apparent if either the camera or the quad is moved up or down. To fix this, mipmapping uses a 32x32 mipmap instead, in this case. This presents a different problem (though not as qualitatively severe as flickering): as there are 32 columns of pixels that must be covered on the quad, the 32x32 mipmap is stretched out horizontally. This stretching creates an undesirable *blurry* image. Although linear filtering may help reduce the flickering artifacts, the blurring problem will not disappear.

The method of anisotropic filtering overcomes these deficiencies. Anisotropic filtering samples and filters a large texture multiple times *in the direction* of the tilt of the quad, and then averages the resulting texel values. Figure 9 shows an image contrasting the use of anisotropic filtering and linear filtering. Notice the lack of blurriness on the ground in the right half of Figure 9.

Figure 9 [6]: Improvement of quality through anisotropic filtering (right) over linear filtering (left)

2.5. Normal Mapping

A *normal* is a 3D vector that is perpendicular to the surface of a triangle of a mesh. Normals are used for lighting calculations (among other things). Typically, mesh normals are stored per-vertex. One drawback of this storage technique is that lighting contributions are then only calculated at the locations of the normals, and are linearly interpolated to find lighting contributions elsewhere. A mesh may be tessellated further to allow for more detailed lighting, but this is inefficient, as further geometric detail is often unneeded.

One solution to the problem is *normal mapping* (related to an older technique called *bump mapping*) where a texture is used to encode one normal per texel. This allows for much greater amounts of surface detail without added geometry. Additionally, the normals do not have to perfectly match the geometry. That is, normals can be cleverly skewed or tilted to artificially create realistic dents or bumps without making an actual deformation to the mesh geometry. The added "geometric" detail is simulated purely by lighting techniques.

2.6. Windows Vista

Windows Vista is Microsoft's next-generation Windows operating system, to be released in early 2007. For the most part, the only Windows Vista changes (from the earlier Windows XP operating system) that affect development of the CCP application were those relating to Direct3D. A summary of changes to the DirectX API in Windows Vista can be found in [2]. Additionally, Windows Vista will be the first operating system to support upcoming Direct3D version 10 class hardware (although the CCP application uses the more conservative, well-tested Direct3D version 9 API).

3. Requirements

Before any software design took place, there were constraints placed on the CCP application by numerous external factors. The 3D graphics API used and the target operating system, platforms, and graphics hardware were hard constraints. All other constraints were optional, and should be optimized, but could be negotiated if necessary.

3.1. 3D Graphics API

Direct3D was the 3D graphics API to be used for the project. The previous preview application used OpenGL, but the new one must use Direct3D, for a few reasons, one of which being that the target operating system, Windows Vista, is guaranteed to have good Direct3D support.

3.2. Target Operating System

The new CCP application targets the Windows Vista operating system only, for a few reasons:

- The CCC has already been developed in Windows.
- Windows has the largest user base of any operating system.
- Windows Vista promises to handle setting changes smoothly.

Although it would be ideal to have the new CCC work on Windows XP as well, there are technical reasons that make setting changes in Windows XP difficult. Instead of making the CCC have second-rate performance on both XP and Vista, it was decided to focus all efforts on development for Vista.

3.3. Target Platforms

The CCP application needed builds for both 32-bit and 64-bit desktop systems, as dictated by the CCC team. Fortunately, the IDE of choice, Visual Studio 2005 Professional [7] (VS2005 Pro), allows for cross-compilation for 64-bit platforms using a 32-bit machine. This has been set up using multiple solution configurations. See Figure 10 for a screenshot of the build configurations. See Appendix A for more detail.

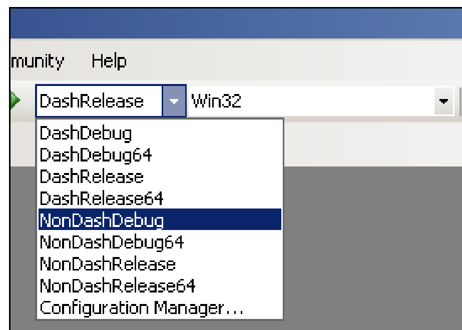

Figure 10. Multiple solution configurations in VS2005 Professional

3.4. Target Graphics Hardware

ATI imposed a constraint that the CCC must run on any ATI R300 model (or newer) video card. This is also the minimum ATI card that Windows Vista's new "Windows Vista Display Driver Model" [8] (WDDM) will support. Additionally, since the R300 and all newer ATI cards support the popular shader model 2.0, this is the shader model used for all vertex and pixel shaders.

3.5. Release Size Constraint

The release size of the CCP application (that is, the size after compression) was a soft constraint. The size of the previous CCP application was 5.5MB, and it was desired

to keep the new CCP application similar in size or smaller. To assist in a smaller release size, objects could be compressed at the cost of visual quality and often load time.

3.6. Loading Time Constraint

The loading time of the 3D scene for the CCP application was desired to be 1 second or less. There are tradeoffs that could be made with preprocessing, compression, and texture resolution to improve loading speed, at the cost of visual quality and/or the release size.

3.7. Visual Quality Constraint

If one were to formulate the constraints of sections 3.5., 3.6., and 3.7. as a knapsack optimization problem, the visual quality is what should be maximized, while keeping the release size and loading time under certain thresholds. Additionally, it should be made clear that although having high visual quality was important, it wasn't as important as making sure that setting adjustments demonstrated *changes* in visual quality effectively. Clearly, demonstrating changes effectively is a requirement – it is what the CCP application is primarily designed for.

3.8. Window Size Constraint

Each of the windows for CCP scene must have a resolution of *192x220*. This resolution was allotted by the CCC team. It is small since the whole CCC panel must fit on a desktop with a resolution of *640x480*. The very small CCP window size greatly influences some CCP application choices. Specifically, high-detail geometry and high-resolution textures are not absolutely necessary.

3.9. 3D Settings for Demonstration

Eight user-selectable 3D settings are integrated into the CCP application. In no particular order, they are:

- Antialiasing (AA)
- Temporal Antialiasing (TAA)
- Adaptive Antialiasing (AAA)
- Anisotropic Filtering (AF)
- Advanced Anisotropic Filtering (AAF)
- Catalyst AI (CatAI)
- Mipmap Level of Detail (MipLOD) Bias
- Geometry Instancing (GI)

More detail on each setting and its implementation is given in section 6.3.

4. Programming Decisions

Before diving into the architecture of the CCP application, some prerequisite development decisions will be described. Good forethought in choosing an integrated development environment (IDE) and seeking out useful libraries saves many hours of development time. Additionally, choosing a documentation format and following a coding standard are two more ways to keep code organized, consistent, readable, and easily extensible.

4.1. Integrated Development Environment

Visual Studio 2005 Professional is used as the IDE for this project, along with *Perforce* [9] for source control. These two intuitive toolsets complement each other very well. Perforce integrates well with Visual Studio, as well as with artist tools such as *Maya* [10] and *3D Studio Max* [11]. Visual Studio 2005 Professional has many desirable features, including a powerful text editor with code completion, advanced project and solution configurations, platform configurations, cross-compilation for 64-bit platforms using a 32-bit platform, and extremely helpful debugging tools. Additionally, it integrates seamlessly with the target 3D API, Direct3D.

4.2. Useful Direct3D Libraries

The libraries mentioned in section 2.3. are undoubtedly useful for typical applications. What must be evaluated is whether or not these libraries support the eccentric requirements of the CCP application (dual windows and D3D devices, as explained in section 6.2.) and are worth their weight in kilobytes, as the release size is an important factor to consider.

4.2.1. DXUT

As DXUT (introduced in section 2.1.4.1.) is meant to be a simple framework, one of its few limitations is that it only supports a single window attached to a single D3D device. As explained later, the preview application needs multiple D3D devices attached to multiple windows. Also, a typical application gains about 400KB from using the framework, mostly due to texture data used for DXUT's GUI system. This is not an insignificant footprint when considering the target release size. For these two reasons, it was decided that DXUT was not worthwhile for the CCP application.

4.2.2. D3DX

In February of 2005, Microsoft began to release the D3DX library (introduced in section 2.1.4.2.) in their DirectX SDK as a dynamically linked library (DLL) instead of as a statically linked library (as it used to be). The DLL approach was adopted so that Microsoft may update their DLLs as necessary (with security fixes, for instance) without requiring applications using the D3DX library to recompile their code (as would be required for an updated statically linked library). The downside of DLLs is their memory footprint. When using a statically linked library, only functions that are used are compiled into a release executable. With a DLL, the whole DLL must be distributed with the release. In the case of the February 2006 SDK used for this project, the size of the D3DX DLL was 2.22MB, or 1.02MB zip-compressed. This is not huge, but when aiming for a ~5MB release size, it accounts for a significant percentage of the allotted memory.

There were six options considered with regard to D3DX:

1) Include the latest D3DX DLL in the release.
2) Convert the latest D3DX DLL to a static library, and statically link.
3) Use an older, statically linked D3DX library.

4) Do not use the D3DX library.
5) Release an installer that downloads any missing D3DX DLL versions at install-time.

Option (1) was certainly plausible, with the sole downside of a large download size for the user. Option (2) may have been possible with the applications *MoleBox Pro* [12] or *DLL to Lib* [13], but the applications cost money, and it is unclear if they would work with the D3DX DLL. Also, it seems that the primary use of these applications is to give developers freedom from having to release DLLs along with their application, but it is unclear if linking with these applications could produce a smaller release size as well. Option (3) was also plausible; however it was possible that some of the required D3DX functionality has been significantly improved since December of 2004 (which is the latest release of a statically linked D3DX library). So, option (3) would be a risky choice. Option (4) was not plausible, as the project required heavy use of DirectX mesh (.X), texture (.DDS), and effect (.FX) formats, for which algorithms for loading and manipulation are not trivial. Option (5) was looked into briefly; however the user may end up downloading more than one DLL, as the installer downloads any missing D3DX DLL versions from Microsoft. This also imposes a requirement that the user must be online at install-time, which is not necessarily acceptable. In addition, if the DLL(s) must be downloaded from Microsoft, then effectively no bandwidth is saved anyhow. It was decided that option (1) was the most reliable option.

4.3. Commenting System

Doxygen [14] commenting was used throughout the CCP project from the start of development. The style is popular, easy to become familiar with, readable, and can produce tidy HTML documentation of any annotated C++ classes, functions, structs, typedefs, code segments, etcetera.

4.4. Coding Standard

ATI's 3D Applications and Research Group had defined a coding standard that includes many common rules and annotations, including a shortened Hungarian notation, upper and lower camel case for different variables, naming conventions, and commenting styles. Overall, the coding standard was all-encompassing and clear. The standard applies mainly to C++ code, which was prominent in this project.

5. Tradeoffs – Size vs Loading Speed vs Quality

One of the most challenging aspects of the project was balancing the constraints of size, quality, and loading speed. The goal was to have as small size release as possible, with load time at a minimum, and visual aesthetics as pleasing as possible. Note that "visual aesthetics" encompasses both *visual quality* and *effective changes in demonstration of visual quality*. As mentioned in section 3., it is desired to have the CCP application be 5.5MB or smaller and load within a second or less.

Table 1 shows size and loading time profiling for various components of the CCP application. The rows outlined in red denote the best options for each component. Note that the *zipped size* in Table 1 is the only important *size* benchmark, since the CCP art assets are uncompressed on a user's computer at install-time to allow for fast run-time loading of the CCP application. Similarly, the *load time* does not take into account any time used for zip uncompressing.

LOADING FROM DISK			
	Load time (secs)	***Size***	***Zipped Size***
Text or Binary or Compressed Binary Mesh			
Text	3.8	25.3 MB	4.00 MB
Binary	0.22	7.21 MB	2.99 MB
Compressed Binary	0.39	2.98 MB	2.91 MB
Text or Binary Effects			
Text	0.09	19 KB	3.9 KB
Binary	0.05	42 KB	5.1 KB
Uncompressed DDSs or DXT1 DDSs			
Uncompressed DDSs	0.08	3.62 MB	2.34 MB
Uncompressed DDSs w/ Pre-generated MIP chain	0.11	5.50 MB	3.16 MB
DXT1 DDSs	0.23	706 KB	419 KB
DXT1 DDSs w/ Pre-generated MIP chain	0.13	940 KB	570 KB
Camera Paths			
Text (also has unnecessary info)	0.03	209 KB	39.9 KB
Binary (with unnecessary info stripped & pre-normalized quats)	0	58.7 KB	30.2 KB

Table 1: Size / loading speed tradeoff chart

X mesh files may be stored in one of three formats: text, binary, or compressed binary. The text format is useful for readability and manual editing, but is the worst option for a release in terms of both size and loading time. The binary format saves both size and loading time over the text format. The compressed binary saves more size, but loads slower than a pure binary format mesh. After zipping the binary format, it is almost the same size as the zipped compressed binary format, but still loads twice as fast at run-time. Thus, it is the best option.

FX effect files may be stored in either text format or binary format. The sum of all zipped binary format effects is slightly larger than all zipped text format effects, but also loads slight slightly faster at run-time. Since the size difference is tiny, it was decided to favor the load time advantage gained from using binary format effects.

After the artist qualitatively determined the required resolutions for textures, it was then necessary to evaluate some compression and pre-calculation options. Firstly, it was determined that all textures used in the scene could be legitimately compressed using DXT1 compression (see Appendix B for more detail on DXT compression). Since DXT compression is the compression option of choice for the CCP application, only tradeoffs between *no* compression and DXT1 compression are presented.

Additionally, since a mipmap chain is usually created at run-time, pre-computing and storing a mipmap chain for a particular texture can speed up load times. In the case of the uncompressed textures, a pre-computed mipmap chain *does not* speed up the load time (possibly due to the large amount of loading from the hard disk required for the large, uncompressed mipmap chain). For the DXT1-compressed textures, pre-computing the mipmap chain *does* save some time. Since DXT1 compression saves a great amount of space, but does not dramatically increase the load time, DXT1 compression is used for all textures. As for mipmap pre-calculation, the technique saves a tenth of a second of load time and adds 151KB to the release size. In this case, it was decided that lessening the load time was more critical.

One downside of DXT compression is that it typically produces undesirable artifacts when applied to *light map*s,[13] which are textures used to encode static lighting information. Figure 11 shows a portion of the light map used for the CCP application, uncompressed and DXT1-compressed. The middle image in the figure shows the differences (brightened 16x) between the two versions. However, these artifacts were hardly apparent when applied to the scene.

Finally, camera paths are exported from *Maya* using a *Sushi*[14] plug-in. They are saved in a text format, which happens to include more information than necessary for camera paths for the CCP application. These files are preprocessed to remove the unnecessary information, pre-normalize the quaternion orientations, and convert the camera path file to binary format. This preprocessing reduces the size and loading time of the camera path files.

[13] *light map:* a texture used as an approximation to static shadows
[14] *Sushi*: ATI 3DARG's internal demo engine

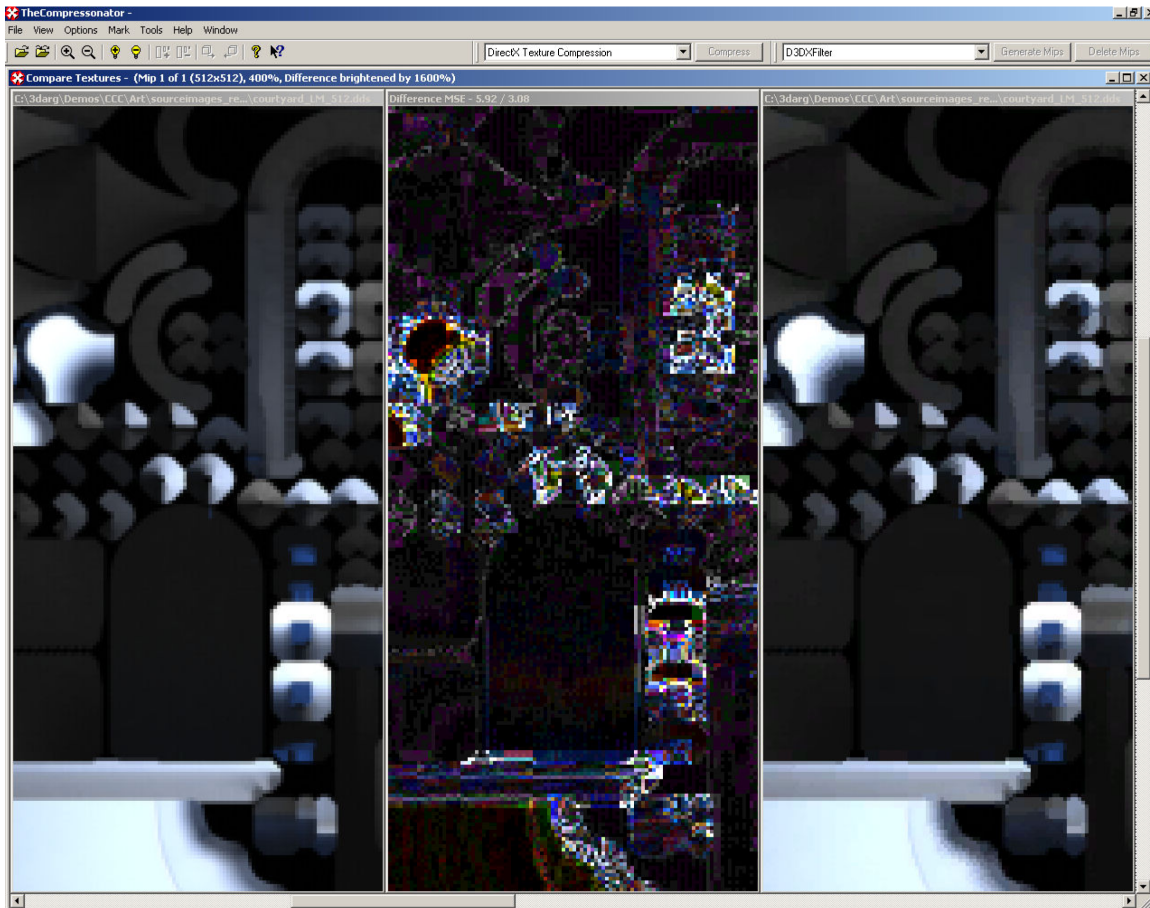

Figure 11: Zoomed view of uncompressed light map (left), DXT1-compressed light map (right), and difference image (middle, brightened 16x)

Table 2 shows a summary of size and loading time of all of the components in the CCP application. Also shown is the time measured for the CCP application to deallocate all of its resources and shut down.

RESULTS, 3GHz PC, ATI Radeon 9800XT	
Release EXE Zipped Size	70.1 KB
D3DX DLL Zipped Size	1.02 MB
Best Option Mesh Zipped Size	2.99 MB
Best Option Effects Zipped Size	5.1 KB
Best Option Textures Zipped Size	570 KB
Best Option Camera Paths Zipped Size	30.2 KB
Total Zipped Size	***4.67 MB***
Total Load Time (not a sum of the individuals, but measured)	***1.05 secs***
Total Shutdown Time (after recv'ing a WM_CLOSE)	***0.03 secs***

Table 2: Summary of release executable statistics, using best options from Table 1

Note a couple things about Tables 1 and 2:

- These are preliminary results, meant as an estimate. Assets were nearly finalized at the time of writing.
- All trials were done with hard disk "Prefetch" cleans (c:\Windows\Prefetch) as a pre-build step, to avoid inconsistent profiling results due to caching. Windows' swap file was also disabled.

6. Implementation

This section describes high-level and low-level implementation details of the CCP application. First, an overview of the goals of the design is presented. Each subsequent section describes the implementation of an aspect of the CCP.

6.1. CCP Design Goals

There is always a balance between hard-coding and data-driven design. If an application were completely hard-coded, it would be an inflexible, hard to debug 3D demo that would probably never be extended. If an application were almost completely data-driven, it would be a 3D engine. The former has the advantage that it is quick to produce, while the latter involves long development cycles. Somewhere in the middle is the CCP application. Given the short development time constraint, the CCP application needed to be as data-driven and extensible as possible, without over-engineering the task at hand.

The following is a list of guidelines indicating the required amount of functionality and generality that the CCP application needed.

- The CCP should use content created by 3D artists. The application may need art assets updated after development is complete. Updates needed not only be possible, but simple and quick to produce and add.

- The CCP should have two identical scenes that can display different settings. The settings could be changed at any time, in either window. Setting changes would be activated by different kinds of inputs, depending on whether debugging or using a release build of the CCP application; so,

it should be easy to accept inputs from multiple sources, and respond to any of them.

- Both scenes should follow the exact same camera paths to make 3D setting differences easy to notice. The camera paths would focus on areas of the scene that best show off a particular setting change.

- The CCP application should be very easy to debug.

With the aforementioned goals in mind, as well as the constraints from section 3., the specification of the CCP project is well-defined. Sections 6.2. through 6.10. detail the satisfaction of the guidelines above and the requirements from section 3.

6.2. Dual Scene Display

One of the first technical challenges imposed was the need for two displays with unique settings. Since different settings are set through different mechanisms, it was necessary to experiment with a few "dual scene display" options before beginning any CCP application coding: multiple *viewports*, multiple *swap chains*, and multiple *D3D devices*.

A *D3D device* is a mechanism used to interface with an *adapter*, which is a representation of an underlying physical 3D hardware device. Each adapter may have multiple D3D devices instantiated. In Direct3D, most graphics objects that are created (textures, effects, meshes, etc.) are bound to a D3D device. To deallocate objects bound to a D3D device, they are usually released with a call to *Release()*. A D3D device also allows for querying of hardware features, such as device capabilities.

A *swap chain* is essentially a *render target*[15] associated with a D3D device. Every D3D device has at least one swap chain, called the *implicit swap chain*. Additional swap chains may be created and associated with a D3D device. Since most resources are bound to a D3D device, swap chains are useful for allowing different views of the same resources, such as different views of the same scene. For instance, multiple swap chains could be used to show the four separate views typically seen in 3D modeling packages such as *Maya* or *3D Studio Max*.

Finally, a *viewport* is very similar to a swap chain – it allows for multiple views of the same scene. The difference between swap chains and viewports is that all viewports must be placed within the same window. Different swap chains may be placed in separate windows.

Each of these three multiple scene view mechanism options (multiple viewports, multiple swap chains, multiple D3D devices) can accomplish the task of showing two views of the same scene. However, only multiple D3D devices allow for *any* setting to differ between each scene. One potential drawback to using multiple D3D devices is that all resources must be loaded twice (bound to each D3D device), which increases both the size of the CPP application in memory as well as the startup time. Another drawback is that the DirectX documentation states that Direct3D is optimized for drawing with one device, as opposed to multiple. However, these drawbacks are not severe, and using multiple D3D devices is the only method that allows for individual D3D render state and other setting changes to occur per scene, since these functions are associated with D3D devices.

Finally, "should multiple windows be used?" was a question to be considered. After some experimentation, it turns out that any number of D3D devices may be used within a single window ("one window per D3D device" is not a requirement). It was decided that using multiple child windows within a single parent window would be the

[15] *render target*: a target surface to render to

most elegant paradigm, in terms of message-passing. This is discussed in detail in section 6.10..

6.3. 3D Settings

There are eight 3D setting changes that the CCP is required to handle, as mentioned in section 3.9.. Some setting changes require simple D3D API calls, while others more advanced techniques. The settings are enumerated and their implementations are explained in this section.

6.3.1. Antialiasing

As explained in section 2.2., antialiasing is a method used to eliminate aliasing or "jaggies" along the edges of primitives. For the CCP, the antialiasing technique used is multisampling antialiasing (MSAA). The following settings are supported.

- *No AA*
- *2X MSAA*
- *4X MSAA*
- *6X MSAA*
- *8X MSAA*

The multisampling type to use is one of many parameters passed to the D3D device creation function, *IDirect3DDevice9::CreateDevice()*. As such, the multisampling type must be specified up front. This presented a problem. To change multisampling settings, a device reset is necessary, which destroys and recreates a device using a call to *IDirect3DDevice9::Reset()*. All resources bound to the device must be released before the reset, and must be reloaded after the reset. The upshot is: changing multisampling settings takes a long time to do (nearly a second).

A better option using render targets was devised to allow for fast MSAA setting changes. A render target may have MSAA options specified for it upon creation. First, a D3D device for each scene is created with *no AA*. Then, an offscreen render target is created for each scene, with the requested MSAA setting. When an MSAA setting change is requested, the render target is destroyed and recreated with the new setting. Destroying and recreating an offscreen render target is very fast, as no D3D devices (and thus no resources) need to be released and reloaded.

For each scene, the application renders completely to the offscreen render target. This is possible since *render-to-texture*[16] is a supported feature of all ATI R300+ video cards. The application then copies the contents of the render target (using *IDirect3DDevice9::StretchRect()*) to the *back buffer*[17] for display.

6.3.2. Temporal Antialiasing

Temporal AA is an ATI-specific setting that may be set to either *enabled or disabled*. Whereas regular AA uses a fixed sampling pattern, temporal AA uses a different sampling pattern each frame. Given a high enough frame rate, the alternating sampling patterns cannot be perceived by the human eye, and the end result is an effectively higher AA level, at no extra performance cost. One constraint is that the frame rate must be high in order to fool the human eye, preventing perception of a flickering effect at the edges of primitives.

This option requires vertical sync[18] (VSync) to be forced on. It is undetermined whether this presents a problem for a windowed application like the CCP. The CCP application enables the technique, but this setting may not show correctly, and thus is not currently demonstrated by the CCP.

---

[16] *render-to-texture*: a feature allowing a texture to be used as a render target
[17] *back buffer*: the default render target for a D3D device
[18] *vertical sync*: limits the framerate to the monitor's refresh frequency

6.3.3. Adaptive Antialiasing

While standard AA techniques take care of aliasing around the edges of primitives, there remain situations in which aliasing can occur on the internals of primitives. For example, an *alpha-tested*[19] primitive (such as a quad with a chain link fence texture applied) would suffer from aliasing artifacts that could not be corrected by standard AA techniques. Adaptive AA overcomes these problems by selectively supersampling only alpha-tested textures.

Although ATI's driver typically seeks out alpha-tested textures and automatically applies adaptive AA to them, the effect was emulated in the CCP application, for various undisclosed reasons. To emulate, multiple rendering passes were used, while masking off different samples per pass, in combination with *centroid sampling*, for each object that demonstrates adaptive AA. Essentially, centroid sampling allows texture coordinates (and other pixel shader *interpolants*[20]) to vary in each pass, based on the varying sample pattern mask (see [15] for more on centroid sampling). In the end, the technique produces the average value of the texture lookups, producing an antialiasing effect on the alpha-textured primitives.

In addition to an *enable* / *disable* setting for adaptive AA, there is a quality versus performance setting dubbed *Adaptive Sampling Divisor* (ASD). ASD specifies how many samples to render per pass, which is used in conjunction with the number of samples to determine the number of passes required, as follows:

- *Adaptive AA disabled:* 1 geometry pass
- *Adaptive AA enabled:* [Number of samples / ASD] geometry passes

[19] *alpha-tested*: transparency in some areas, to a degree, according to per-pixel "alpha" values
[20] *interpolants*: per-vertex values that are interpolated in a pixel shader to produce per-pixel values

Therefore, ASD = 1 produces the highest quality at the cost of performance (many passes are required). Table 3 shows the supported combinations of sample counts and ASD settings.

AA Setting ***(# of samples)***	***ASD Setting*** ***(# of samples per pass)***	***Number of passes***
(no AA)	1	1
2X	1	2
2X	2	1
4X	1	4
4X	2	2
6X	1	6
6X	2	3
6X	3	2
8X	1	8
8X	2	4
8X	4	2

Table 3: Supported AA / ASD combinations.

Each of the combinations shown in Table 3 is implemented as a different FX effect *technique*, each of which must be defined in any FX effect file that is to demonstrate adaptive AA. In the CCP application, the fence happens to be the only object in the scene that demonstrates adaptive AA.

6.3.4. Anisotropic Filtering

Anisotropic filtering (AF), as introduced in section 2.4., is a setting that allows for angle-independent texture sampling to produce high quality textured surfaces, no matter what the viewing angle is with respect to the surface. For the CCP application, the following texture filtering modes are supported:

- *No AF*

- *2X AF*
- *4X AF*
- *8X AF*
- *16X AF*

When anisotropic filtering is disabled, trilinear filtering is enabled. This filtering method is the baseline filtering method that we wish to compare against. Enabling different AF levels is done easily through the D3D API.

6.3.5. Advanced Anisotropic Filtering

Advanced anisotropic filtering (AAF) is a feature specific to a small subset of ATI video cards (ATI R500 series). This setting allows for higher quality anisotropic filtering. The setting may be either *enabled* or *disabled*.

6.3.6. Catalyst AI

Catalyst AI is an ATI-specific mechanism used to improve performance behind the scenes in subtle ways. Catalyst AI has three settings: *disabled*, *standard*, and *advanced*. When *disabled*, no optimizations are used. When *standard*, some liberties are taken with compression of textures, among other things, to improve performance. These optimizations are meant to be hardly noticeable to the user. When *advanced*, many more liberties are taken, and a reduction in visual quality is usually noticeable. In order to display any changes in the CCP application, all textures must be released and reloaded after a Catalyst AI setting change.

Since textures in the CCP are already compressed, the Catalyst AI setting makes a minimal difference in the CCP application visuals. However, small changes are noticeable if the camera is extremely close to some surfaces.

6.3.7. Mipmap Level of Detail Bias

As explained in section 2.3., mipmaps are collections of pre-computed, different-sized versions of the same bitmap, used for fast lookups during texture filtering. For a given textured primitive, the GPU performs calculations to choose the mipmap level that best fits the primitive, and will then use that level when applying the texture. The mipmap level of detail (LOD) bias setting influences the mipmap level choice calculation, persuading the GPU to choose a smaller or larger mipmap level, improving performance or quality, respectively. The four choices allowed for the level of detail bias setting are *high performance*, *performance*, *quality*, and *high quality*. Additionally, if the High Performance option is specified, then some of the larger levels in a mipmap chain are ignored (dropped from the chain).

6.3.8. Geometry Instancing

Geometry instancing is the concept of having multiple copies of the same geometry or mesh in a scene. Each instance could have different state, such as transparency, hierarchical transformations, etcetera. Clearly, the world space position of a multiply-instanced object would be one desirable state change between instances. GI can speed up applications by precompiling graphics commands so that they may be passed quickly to the GPU, instead of being processed one at a time on the CPU.

The geometry instancing setting was eventually decided to be excluded from the CCP demonstration. However, in the CCP, although the function to enable GI is only a stub, support for the calling that stub was added.

6.4. Resource Management

For generic resource management, a templated interface *CCPIManager* was made. The interface exposes generic methods for loading, fetching, and removing resources from the base management data structure. The only two methods that need be overwritten by derived class are the loading and unloading functions *Add()* and *CleanByName()*, as their implementations are specific to the resource being managed. Once a class is derived from the templated interface, extra functionality can be added to that specific manager. When a resource is loaded, a string identifier is associated with the resource such that the resource can be fetched by its identifier. In the CCP application, this feature was used to fetch a loaded resource by its file name.

6.5. Singletons

In some cases, instances of managers may suffice, but sometimes classes are well-suited as singleton objects. Managers are a great example – usually, only one manager is needed for a given application.

In rare cases, it is useful to make a class, and then be able to use it as both a global singleton instance, and at the same time, have multiple local instances of that same class. For instance, the CCP application has a camera class. A singleton camera was useful, since only one camera defined the view for both scenes. However, it was necessary to have instances of the camera to allow for copying of the singleton camera, to make modifications to its view for rendering reflections (see section 6.8.1. for discussion of the reflection rendering technique).

The templated singleton design used for the CCP application allows for both a global singleton of a class as well as instances of the class, if desired. If the developer would like *no* local instances to be allowed (a true singleton), then a derived singleton class need only have its constructors declared *private*.

6.6. Art Assets and Extensibility

As with most 3D demos, this project was a combination of artist and programmer contributions. As such, it was necessary to provide the artist with any tools he needed to quickly and easily ascertain the quality of the scene as shown in the CCP application, as development progressed.

6.6.1. Art Assets

The art for the CCP application was created, edited, and/or applied by ATI artist Abe Wiley. The art assets include meshes (the complete courtyard), textures (*color maps*[21], a *clip map*[22], *light maps*[23], and *normal maps*[24]), and camera flight paths. The main tool used for modeling and creating camera paths was Maya. For texture creation and tweaking, *Modo* [16] and *Photoshop* [17] were used.

6.6.2. Initialization (.ini) File

During the design of the CCP, it was necessary to tweak settings and art assets to quickly and easily produce the most effective visuals and features, without requiring recompilation of the application. Integrating a scripting engine would be an excessive solution. However, a program initialization file (*.ini* file) serves as an easy way to test out new features and art assets. Figure 12 shows a sample .ini file. Comments are preceded with a semicolon. All options shown in Figure 12 are required, with the exception of the *MIPDEBUG** options.

[21] *color map*: the traditional use of a texture – encoding of a mesh's color
[22] *clip map*: 1-bit "on" or "off" data in a texture, used to determine what parts of a surface are transparent
[23] *light map*: a texture used to store lighting contributions from static light sources
[24] *normal map*: a texture used to encode normal information for a surface

```
; Resolution of each of the two windows.
RES: 640x480

; Reflection texture size.
REFLECTION_RES: 320x240

; Height of the water level in the main lower basin of the fountain. This is only used for reflection calculations.
FOUNTAIN_WATER_LEVEL: 65.619

; Show the reflection texture, picture-in-picture, for debugging?
DEBUG_SHOW_REFLECTION: false

; .X files' paths. These .X files should reference a number of .FX files and textures.
XFILE: ./Meshes/final_mesh.x  ; Final courtyard scene revision.

; Whether or not to reset the path location to the beginning of a path upon a path switch.
CAMPATH_RESET_ON_SWITCH: true

; Camera path files.
CAMPATH_DEFAULT: AmbCamera.bth
CAMPATH_AA: AntiAliasCam.bth
CAMPATH_AF: AnisoCam.bth
CAMPATH_TAA: AntiAliasCam.bth
CAMPATH_AAF: AnisoCam.bth
CAMPATH_AAA: AdaptiveAACam.bth
CAMPATH_CAI: AmbCamera.bth
CAMPATH_MIPLOD: MipMapCam.bth
CAMPATH_GI: AmbCamera.bth

; Camera path FOVs.
CAMPATH_DEFAULT_FOV: 55.0
CAMPATH_AA_FOV: 55.0
CAMPATH_AF_FOV: 55.0
CAMPATH_TAA_FOV: 55.0
CAMPATH_AAF_FOV: 55.0
CAMPATH_AAA_FOV: 55.0
CAMPATH_CAI_FOV: 55.0
CAMPATH_MIPLOD_FOV: 55.0
CAMPATH_GI_FOV: 55.0

; Whether or not to use mipmap debugging mode (different colored mips).
MIPDEBUG: false

; Paths to mipmap debugging textures (colored levels).
MIPDEBUGBASE1024: ./Mipdebug_textures/texbase1024.bmp
MIPDEBUG1024: ./Mipdebug_textures/tex1024.bmp
MIPDEBUG512: ./Mipdebug_textures/tex512.bmp
MIPDEBUG256: ./Mipdebug_textures/tex256.bmp
MIPDEBUG128: ./Mipdebug_textures/tex128.bmp
MIPDEBUG64: ./Mipdebug_textures/tex64.bmp
MIPDEBUG32: ./Mipdebug_textures/tex32.bmp
MIPDEBUG16: ./Mipdebug_textures/tex16.bmp
MIPDEBUG8: ./Mipdebug_textures/tex8.bmp
```

Figure 12: Sample .ini file 'ccpconfig.ini'

The .ini options are explained below.

RES:

This option allows for adjustment of the resolution (and size) of each of the two CCP windows.

REFLECTION_RES:

This option specifies the resolution of the texture to use for creating the reflection shown in the fountain. A higher resolution produces a more realistic reflection See section 6.8.1. for explanation of the reflection effect.

FOUNTAIN_WATER_LEVEL:

This option must be a floating-point constant, which tells the application the vertical height of the water in the scene. This is used for reflection calculations. Although there are multiple pools in the fountain, some with different heights than others, only one reflection is done and applied to all water surfaces, for reasons of efficiency. This value can be adjusted to make the reflection look the best for all pools, and as such, is a matter of aesthetics. Refer to section 6.8.1. for more detail on the reflection effect.

DEBUG_SHOW_REFLECTION:

This option must be either “true” or “false.” It specifies whether or not to show the reflection texture for the fountain in the top-left corner of the window, for debugging purposes. See Figure 15 in section 6.8.1. for an example of this.

XFILE:

This option specifies a full or relative path to an X mesh file, to be loaded into the scene. For the CCP application, there is only one mesh. However, multiple meshes may be loaded by simply adding more *XFILE: meshname.x* declarations to the .ini file. However, note that all loaded meshes are simply positioned in the center of the scene.

CAMPATH_RESET_ON_SWITCH:

This option must be either "true" or "false." There are multiple looping camera flight paths that are switched between when choosing 3D setting categories in the CCC GUI. If this setting is "true," then when the CCP switches to a new camera path, the camera's flight will begin from the start of the new path's animation loop. If this setting is "false," then, upon a switch, the camera will immediately be placed as far along the new path as it has reached along the old path. Which option is better is a matter of taste, but "true" was chosen for the CCP release.

CAMPATH_*:

This set of options specifies the full or relative paths to camera path files. These may either be camera paths taken straight from the *Sushi* camera path exporter (.pth files) or preprocessed, stripped-down binary camera paths (.bth files). There is a camera path for each of the settings that the CCP demonstrates. Note that some camera paths are used more than once in Figure 12. For instance, it was desirable to use the same camera path for demonstrating AA setting changes (*CAMPATH_AA*) as for temporal AA setting changes (*CAMPATH_TAA*).

CAMPATH_*_FOV:

This set of options specifies the field of view (FOV) to use for each camera path's camera. Note that if two different paths use the same path (as in the case of *CAMPATH_AA* and *CAMPATH_TAA*), then they must also have the same FOV specified.

MIPDEBUG:

This option must be either "true" or "false." If true, this option enables a debugging mode that allows for visualization of different mipmap levels in the scene, and how they are affected by changing the mipmap level of detail setting. See Figure 21 in section 6.9. for a screenshot of this debugging mode. If "false," the scene is loaded as normal. Note that if this is "true," a different mesh ("test_mesh_at_work_mipdebug.x") should be loaded with the *XFILE* option. The

mesh should reference effect files developed exclusively for this mipmap debugging mode.

MIPDEBUG:*

This set of options specifies the full or relative paths to textures to be used in the mipmap debugging mode. The textures are different sizes and colors, as explained in section 6.9..

To handle loading of the .ini file settings, a singleton class, *CCPConfigManager*, reads and interprets the .ini file, and grants access to the settings to any application code that needs the information.

6.7. Camera Paths

Different camera fly paths are used in the CCP application to demonstrate different settings. For instance, to demonstrate temporal AA setting changes, it was found to be most effective to use a camera path that flies alongside the alpha-textured fence (see section 6.8.2. for more detail on the fence effect). All paths were created in *Maya* and exported using a *Sushi* plug-in for *Maya*. As explained in section 5, the path files are then preprocessed to make run-time loading of the paths as fast as possible.

The camera system in the CCP application is fairly straightforward. A camera class, *CCPCamera*, defines a basic camera representation, allowing for basic placement, movement, and orientation of a camera. Since it was desired to move this camera along a path, a camera path class was created called *CCPCameraPath*. This class holds many 3D position and orientation value pairs, as well as the sampling period (in seconds) between each pair. Given the current application time, a function will return the current position and orientation of the camera by *LERP*ing[25] and *SLERP*ing[26] between position and

[25] *LERP*: linear interpolation
[26] *SLERP*: spherical linear interpolation

orientation values, respectively. Finally, a camera controller class, *CCPCameraController*, was created. This class holds a *CCPCamera* and a *CCPCameraPath* as members. *CCPCameraController* pulls together all camera functionality, including switching between different camera paths, disabling camera path following in order to move the camera manually, querying for the application time, finding the camera position/orientation for a particular time, and setting the camera position/orientation of the *CCPCamera*.

Since it was desired to have each scene's camera path remain in sync, it was simplest to use an instance of the *CCPCameraController* class as a Singleton object. This was possible by using the following type definition:

```
typedef CCPSingleton<CCPCameraController> CCPGlobalCameraController;
```

Note that the design of the templated *CCPSingleton* class still allows a *CCPCameraController* to be instantiated multiple times if desired, as is required for the fountain's reflection rendering technique (see section 6.8.1.).

6.8. Scene Effects

A few notable effects were used in the courtyard scene. Some were used to demonstrate specific settings, and others were used to make the scene more attractive. This section discusses the implementation of the various effects.

6.8.1. Reflection

The fountain at the center of the courtyard scene shows pools of water with a realistic reflection on the water's surface. This effect is meant to make the scene look more realistic (as opposed to demonstrate a particular setting). Figure 13 shows a screenshot of the fountain's reflection.

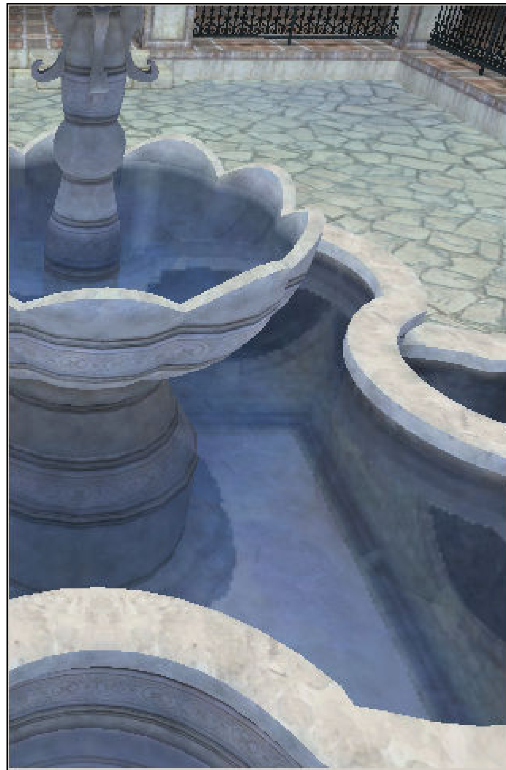

Figure 13: Fountain reflection effect

To create the reflection visuals seen on the surface of the water, the application renders the scene to a texture, from the point of view of a camera that is under the water. For example, if the camera were looking straight down into the fountain, we would want to reflect the camera about the plane defined by the surface of the water, producing a camera facing straight up at the sky. Figure 14 shows an example of the idea. A function *CCPCamera::ReflectAboutPlane(...)* flips the camera as desired.

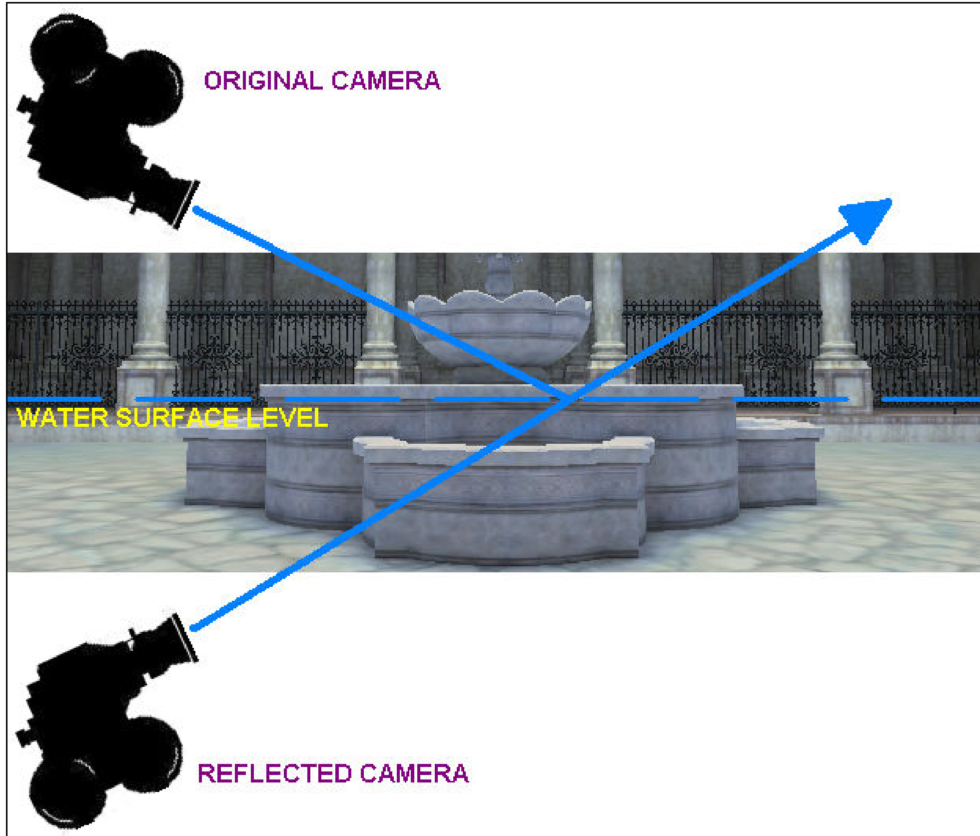

Figure 14: The orientations of the original camera used for the camera fly path, and the reflected camera

From this point of view, it is necessary to remove any objects at or below the level of the water, since they would not ever be seen in a reflection. This is done using a *user clip plane*, which is a standard component of the D3D API. It allows us to prevent portions of the scene from being rendered. The user clip plane is set to be slightly higher than the surface of the water, so that the scene may be seen "from the water's point of view." A rendering pass is then performed from the reflected camera's position, storing the result in a texture. The top-left corner of Figure 15 shows the scene from the point of view of the reflected camera, after the user clip plane has been applied.

Figure 15: Fountain view, with reflection texture shown onscreen for debugging.

The only remaining step is to correctly position the texture shown at the top-left of Figure 15 onto the surface of the fountain. To do this, the application uses *projective texture mapping*, which is a method used to project a texture onto a scene as if the texture were a slide in a slide projector. More information on projective texture mapping can be found at [18]. By distorting vertex positions or texture coordinates, a moving surface could be simulated, if desired. However, it was desired to have an unmoving surface, as the fountain has no circulation or cascading of water.

The water is semi-transparent, due to alpha-blending, or transparency, applied to the water. The amount of transparency is dependent on the viewing angle of the camera with respect to the water's surface. This effect is a partial approximation of the *Fresnel Effect*, which is a real-life phenomenon: when looking at a water surface at a small grazing angle, more reflectivity is apparent than when looking straight down at the surface. The water also has a light map applied, allowing for explicit specification of lighter or darker areas of the surface.

Although the resolution for the reflection render target is configurable via the .ini file (see section 6.6.2.), a smaller resolution is usually used than for the resolution of the CCP window. This is because less detail will be apparent in the reflection, and thus the performance optimization will show little to no decrease in visual quality.

Since the complete scene had to be rendered an extra time to accomplished a reflection, only one reflection was performed for efficiency reasons. As such, this method only produces a perfectly accurate reflection for *one* of the fountain pool levels. However, the reflection still looks realistic from all viewing angles in all camera fly paths.

6.8.2. Fence

Although each section of the fence in the courtyard is only composed of a couple of primitives, its pattern detail is simulated through the use of a *clip map*. A clip map is a simple bitmap specifying which texels of a texture should be opaque and which should be transparent. Figure 16 shows the clip map used for the fence.

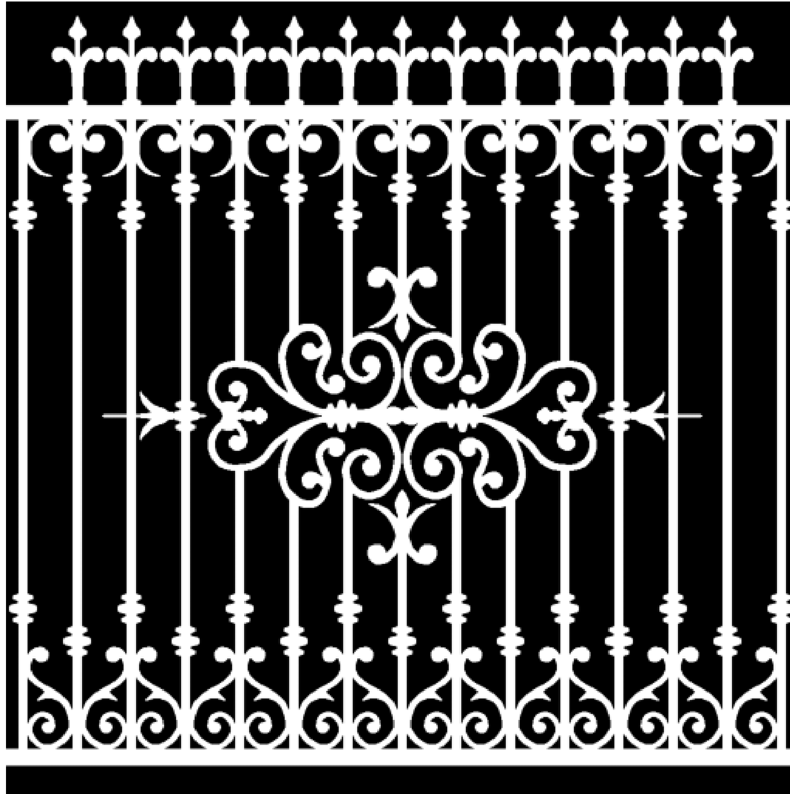

Figure 16: Clip map used for the courtyard fence

The clip map in Figure 16 is stored as the 1-bit alpha channel of a DXT1-compressed normal map texture for the fence. A separate texture stores the color map for the fence.

The fence is used as a way to demonstrate adaptive AA. Figure 17 shows two views of a portion of the fence. The left scene has MSAA enabled with adaptive AA disabled, and the right scene has MSAA enabled, as well as adaptive AA. With only MSAA enabled, the fence's outlines still show aliasing artifacts. With temporal AA enabled, the aliasing artifacts are greatly reduced. Note that mipmapping must be disabled for any textures applied to a surface that demonstrates adaptive AA, or else undesirable artifacts appear.

Figure 17: Fence with no adaptive AA (left) versus high quality adaptive AA (right)

6.8.3. Normal mapping

As explained earlier in section 2.5., normal mapping encodes normal information in the R, G, B color channels of a texture. Each color component encodes one component of a 3D vector.

It was decided early on that *diffuse lighting*[27] effects could be encoded into a light map, avoiding expensive run-time lighting calculations as well as storage of normal maps. This is because conventional diffuse lighting contributions are calculated via the following equation:

$\vec{N} \bullet \vec{L}$, where $\vec{N}$ is a normal, $\vec{L}$ is a vector pointing towards a light source

[27] *diffuse lighting*: lighting from diffuse (rough-surface) reflections

and both $\vec{N}$ and $\vec{L}$ are static. That is, the normals of all scene objects are static with respect to the (also static) light source (the sun).

Since *specular lighting*[28] contribution calculation requires a vector pointing towards the camera (among other things), and the camera position is constantly changing, specular lighting contributions must be computed at run-time. Since detailed normal information provides substantial benefits when calculating specular lighting, normal maps would be used if specular lighting were to be used. Additionally, it would be beneficial to use other features like *gloss map*s[29] or *specular exponent map*s[30] for each surface. However, CCP release size constraints as well as project time constraints prevented these per-pixel lighting techniques from being used in the CCP application. Normal mapping is not used, but the technique is implemented within applicable shaders. The right half of Figure 18 shows an example of the use of normal maps combined with specular lighting.

Figure 18: Screenshots with diffuse lighting only (left) and with specular lighting & normal mapping (right)

Within any shader that utilizes normal mapping, in order to obtain valid normal values from a normal map, the color component ranges (0.0 to 1.0) are first scaled and shifted, to be in the valid normal ranges (-1.0 to 1.0). Then, since all vectors that are used for lighting calculations need to be in the same coordinate space, every vector must be transformed (from *world space*) to the same space as the vectors encoded in the normal

[28] *specular lighting*: lighting from specular (smooth-surface) reflections
[29] *gloss map*: a texture used to control the shininess of different areas of a surface
[30] *specular exponent map*: a texture used to control the focus or tightness of specular reflections on different areas of a surface

map (*tangent space*). *World space* is the global coordinate system that all geometry is stored in, relative to one origin, and *tangent space* is a local coordinate system used at the surface of any given primitive. The basis of a tangent space coordinate system consists of the *tangent*, the *normal*, and the *binormal*. These tangent space vectors are stored within the courtyard mesh. Although it would be possible to translate the normal map tangent space vectors to be in world space (and this is probably more intuitive), this would be more computationally intensive, since the translation would need to be performed per-pixel instead of per-vertex.

6.8.4. Sky Effect

The courtyard sky shows moving clouds to increase the realism of the scene. The sky geometry is composed of one large quad, and uses only one texture to produce the unique animated effect. The sky is demonstrated in Figure 19.

Figure 19: The courtyard sky

The animated cloud technique is a variation on a technique found in [19]. Since only one texture is needed for the effect, the technique it is very economical in terms of storage space, which is a perfect fit for the CCP application's requirements. The effect uses the red and green channels of a texture to store horizontal and vertical texture coordinate perturbation values, respectively, and the blue channel is used to store the cloud texture itself. Although each channel can only store 8 bits of information, this is

enough to produce convincing animated clouds. Figure 20 shows the different channels of the sky texture, each interpreted as 8-bit grayscale textures. The tool *RGBAviewer* [20] (shown in Figure 20) was developed to allow for simple viewing and editing of texture color components.

Figure 20: The different color components of the one texture used for the sky effect

The way the sky shader works is as follows. First, horizontal and vertical perturbations are fetched from the texture (red and green channels, respectively). These perturbation values are scaled by different amounts and are summed with texture coordinate values. Then, the application time is scaled and summed to the *x*-component of the texture coordinates. Next, the cloud texture (blue channel) is fetched with these new texture coordinates. Since these coordinates have been disturbed (horizontally and vertically) and scrolled (horizontally), and this behavior will continue each frame, this

produces a moving, distorting cloud. This same algorithm is used to produce a second cloud that scrolls in the opposite direction.

Finally, the colors of the two clouds are linearly interpolated to produce a final cloud color, which is then interpolated with a sky color, according to an “overcast factor.” Other options were added for tweaking, including cloud color, sky color, sun glare, brightness, tint, and individual cloud speed and perturb factors. The combined effect of the distortion and the clouds passing at different speeds produces a seemingly slithery, cloudy sky (best appreciated when in motion).

6.9. Debugging Methods

To help shorten development time and solve any future issues that could arise, it was helpful to make the CCP application easy to debug. As such, a logging class was one of the first classes implemented. The logger outputs warning and error messages to both the debug console using *OutputDebugString()*, and to a log file. To capture messages sent to the debug console, one can either use Visual Studio’s *output* tab, or the program *DebugView* [21].

A unique debugging issue arose from the embedded nature of the CCP application. The final build of the CCP application needed to run inside a small window provided by the CCC. This makes stepping through code difficult. More importantly, the CCP could only be launched by launching the CCC, which is an undesirable level of indirection when debugging. For this reason, both “dashboard” and “non-dashboard” builds were created. The two builds use conditional compilation to include either code paths to create an executable to be launched by the CCC (“dashboard” build) or a standalone executable with a larger window size, movable windows, and other debugging features (“non-dashboard” build). Since both builds include code that is mostly common to the two, debugging can successfully be carried out in a “non-dashboard” build in most

cases. Additionally, switching between the two build types is very simple (see Appendix A).

To verify that different settings were changing as desired, it was useful to zoom in on certain parts of the scene, instead of flying along preconfigured paths. To do this, a "manual camera movement" mode was implemented to let developers move and face the camera anywhere in the scene.

Moving around the scene does not help verify *every* setting, however. For instance, to confirm that the mipmap LOD bias setting was working as expected, it was necessary to create a mipmap debugging mode. When this mode is enabled, every texture in the scene has its mipmap levels individually modified to have different images (solid colors) so it can be seen that the levels are being blended and biased correctly. Figure 21 shows what the mode looks like for the courtyard scene.

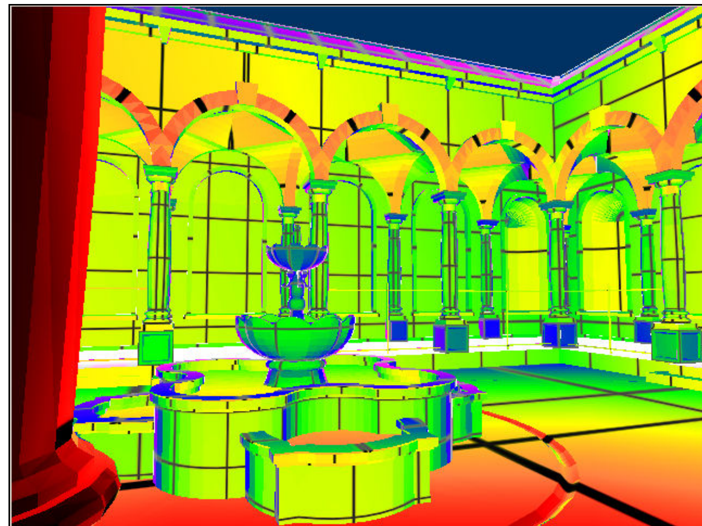

Figure 21: Mipmap debugging mode enabled in the courtyard scene

Table 4 denotes what each color represents. Note that the mipmap levels chosen are dependent on the resolution chosen for the window. That is, if the same view of the scene is shown with a smaller window, the mipmap levels chosen will be different (specifically, they will be closer to purple).

Color	Texture size
Black	1024x1024
Dark Red	512x512
Red	256x256
Orange	128x128
Yellow	64x64
Green	32x32
Blue	16x16
Purple	8x8

Table 4: Colors scheme for mipmap debugging mode

6.10. Communication Overview

This section discusses the overall flow of the CCP application, both execution and communication. The flow chart in Figure 22 summarizes the essential steps of the CCP application. The flow chart distinguishes between functionality existing only in the dashboard build, functionality existing only in the non-dashboard build, and functionality that is common to both builds.

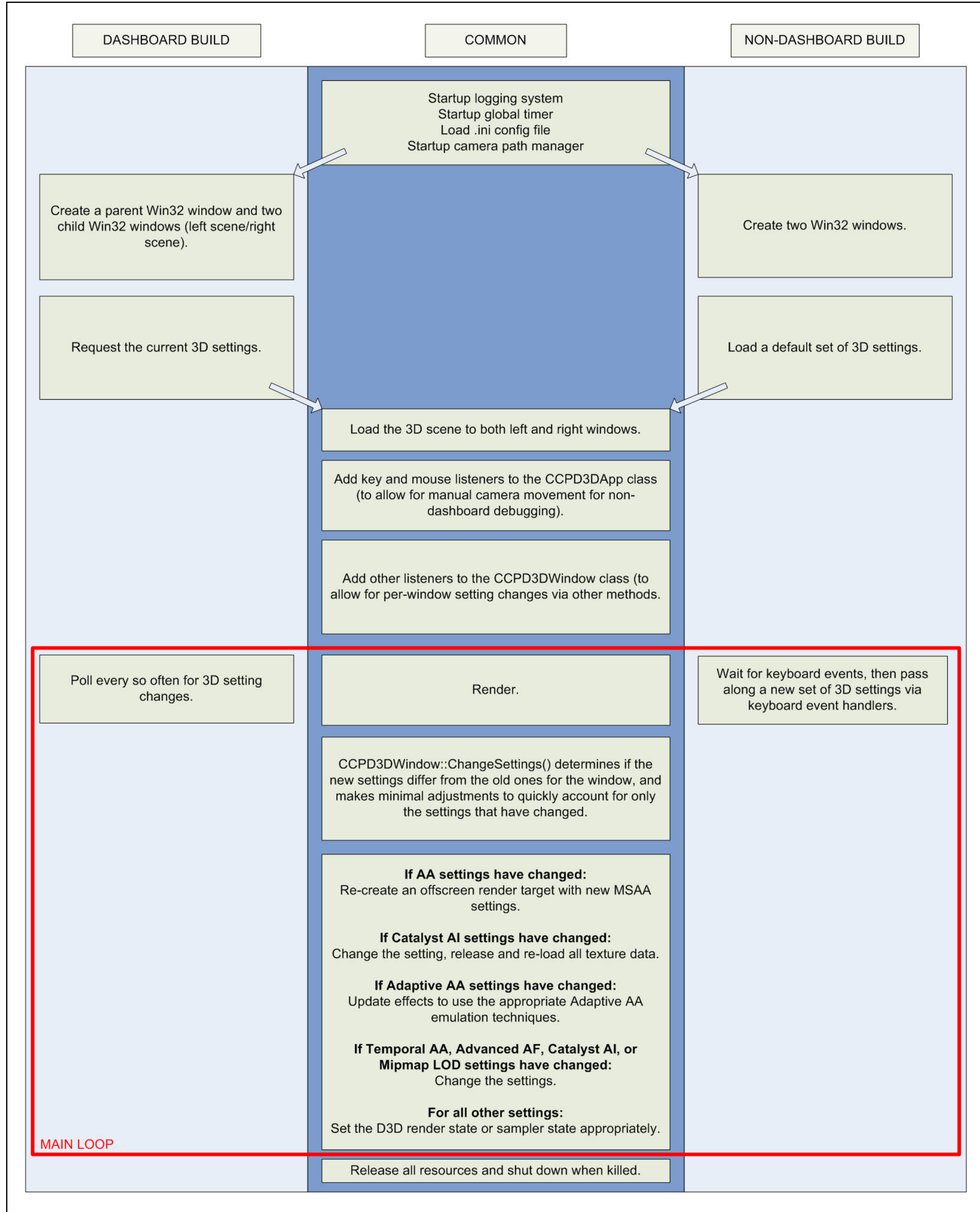

Figure 22. CCP application communication flow chart (main loop outlined in red)

First, the CCP application initializes global components such as the logging system and timer. This initialization takes place regardless of the build chosen.

Next, if a non-dashboard build is being used, two separate windows are created. The application then loads a default set of 3D settings for both windows.

Due to the embedded nature of the dashboard build, a different message-passing scheme is required, and thus a different hierarchy of windows is required. Two child windows are spawned below a parent window, and are similar in functionality to the two windows created in the non-dashboard build.

After the preceding window specifics have been taken care of, the application flow is almost identical between builds. Direct3D is initialized and all art assets for the scene are loaded. Listeners are added to allow for application control via any input device. Finally, the program's main loop is reached, outlined in red. This loop is where scene rendering takes place in both windows.

One final difference between the dashboard and non-dashboard builds is how 3D settings are adjusted. In the non-dashboard build, the CCP application accepts input from the keyboard for testing of all CCP options. In the dashboard build, the application communicates with the CCC to determine 3D setting changes.

Aside from the preceding setting change mechanism difference, the main loop is similar for both dashboard and non-dashboard builds. If any settings have changed, the application updates the scene display for the appropriate window. Different settings are changed using different techniques, as explained earlier in section 6.3.

7. Conclusion

The new CCP application was designed with many goals in mind, including extensibility, readability, effective showcasing of 3D settings, quick loading time, quick transitioning between camera paths and 3D settings, small release size, ease of use (for both developers and artists), among many other desirables. Above all, it is hoped that the new CCP application demonstrates setting changes effectively and provides a unique, intuitive 3D setting tweaking experience for end users.

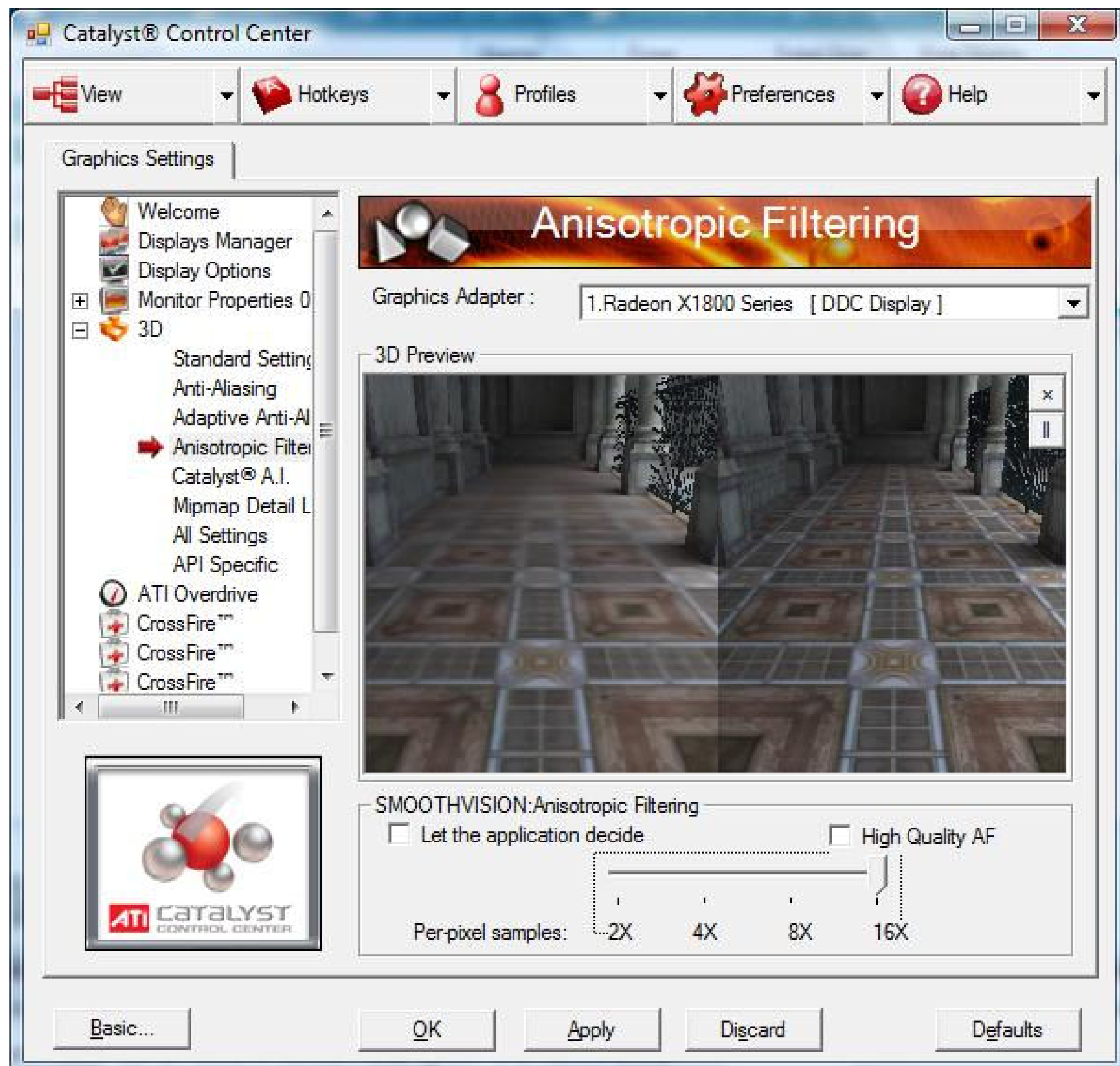

Figure 23: The new CCC in Windows Vista

8. References

[1] Coppens, Paul. "Loading and displaying .X files without DirectX." Gamedev.net. Accessed: June 2006. <http://www.gamedev.net/reference/programming/features/xfilepc>.

[2] Microsoft Corporation. "DirectX for Windows Vista." MSDN Library. Accessed: May 2006. <http://msdn2.microsoft.com/en-us/library/ms681824.aspx>.

[3] "Direct-X File Format." Compiled by Bourke, Paul. Created: January 1999. Accessed: May 2006. <http://local.wasp.uwa.edu.au/~pbourke/dataformats/directx/>.

[4] ATI Technologies, Inc. Accessed: May 2006. <http://www.ati.com>.

[5] Harris, Kevin. "Texture Mip-mapping" sample code. Codesampler.com. Created: February 2005. Accessed: May 2006. <http://www.codesampler.com/dx9src/dx9src_3.htm>.

[6] Weinand, Lars. "ATI's Optimized Texture Filtering Called Into Question." Tomshardware.com. Created: June 2004. Accessed: May 2006. <http://www.tomshardware.com/2004/06/03/ati/index.html>.

[7] Microsoft Corporation. "Visual Studio 2005 Professional Edition" product page. Microsoft Products. Accessed: May 2006. <http://msdn2.microsoft.com/en-us/vstudio/aa718668.aspx>.

[8] Walbourn, Chuck. Microsoft Corporation. "Graphics APIs in Windows Vista." MSDN Library. Modified: August 2006. Accessed: May 2006. <http://msdn2.microsoft.com/en-us/library/ms681824.aspx>.

[9] Perforce. "Perforce Windows Client" product page. Accessed: May 2006. <http://www.perforce.com/perforce/products/p4win.html>.

[10] Autodesk, Inc. "Autodesk Maya" product page. Accessed: May 2006. <http://www.autodesk.com/maya/>.

[11] Autodesk, Inc. "Autodesk 3ds Max" product page. Accessed: May 2006. <http://www.autodesk.com/3dsmax/>.

[12] Teggo. "MoleBox Pro" product page. Accessed: May 2006. <http://www.molebox.com/>.

[13] Binary Soft, Inc. "DLL To Lib" product page. Accessed: May 2006. <http://www.binary-soft.com/dll2lib/dll2lib.htm>.

[14] van Heesch, Dimitri. "Doxygen: Source code documentation generator tool." Modified: December 2006. <http://www.stack.nl/~dimitri/doxygen/>.

[15] Microsoft Corporation. "Antialias Sample." MSDN Library. Modified: October 2006. Accessed: June 2006. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Antialias_Sample.asp>.

[16] Luxology LLC. "Modo" product page. Accessed: May 2006. <http://www.luxology.com/whatismodo/>.

[17] Adobe Systems Incorporated. "Adobe Photoshop" product page. Accessed: May 2006. <http://www.adobe.com/products/photoshop/>.

[18] Everitt, Cass. "Projective Texture Mapping." Nvidia Corporation. Accessed: May 2006. <http://developer.nvidia.com/object/Projective_Texture_Mapping.html>.

[19] Isidoro, Jon and Riguer, Guennadi. “Texture Perturbation Effects.” ATI Research, Inc. Accessed: July 2006. <http://ati.amd.com/developer/shaderx/ShaderX_TexturePerturbationEffects.pdf>.

[20] Pfeil, William A. "RGBAviewer" application. Modified: July 2006. <http://users.wpi.edu/~wap/downloads/RGBAviewer_1.0_bin.zip>.

[21] Russinovich, Mark. "DebugView for Windows v4.63." Microsoft TechNet. Modified: November 2006. Accessed: May 2006. <http://www.sysinternals.com/Utilities/DebugView.html>.

[22] "UnrealWiki: DXT." The Unreal Engine Documentation Site. Accessed: June 2006. <http://wiki.beyondunreal.com/wiki/DXT/>.

[23] "Antialiasing." Meko Ltd. Accessed: May 2006. <http://www.meko.co.uk/antialias.shtml>.

Appendix A. CCP Build / Release Overview

Switching between the many builds required for development / release, 32-bit / 64-bit platforms, and embedded / floating window configurations is greatly simplified by Visual Studio's configuration manager.

A.1. Solution Configurations

There are 8 solution configurations for the CCP application. There is one each for all 8 combinations of (dashboard / non-dashboard), (Win32 / Win64), and (Debug / Release). Simply choose a solution configuration, and then compile.

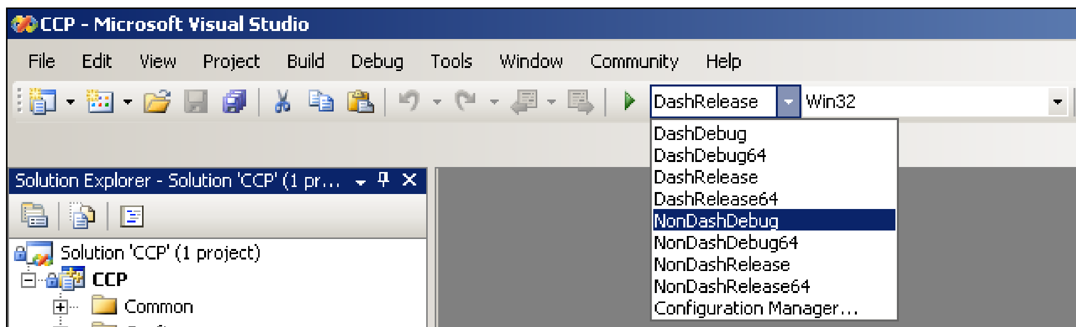

Figure 24: Supported build configurations

A.1.1. Dashboard / Non-dashboard Builds

Dashboard and non-dashboard build configurations were made to allow for easy switching between a version of the application that gets embedded within the CCC dashboard, and a version of the application that is standalone, useful for controlled development and debugging. The latter build configuration proved critical for controlled testing.

A.1.2. Win32 / Win64 Builds

The CCC is meant for both 32-bit and 64-bit platforms, so builds of each were required for both testing and release. Although the development machine for this project is a 32-bit machine, Visual Studio 2005 allows for 64-bit cross-compilation.

A.1.3. Debug / Release Builds

Debug / Release builds are standard configurations within Visual Studio. The debug configuration allows for stepping through the application, and the release configuration is optimized for speed.

Appendix B. DXT Compression

DXT, or DXTC, is an effective texture compression method originally developed by *S3 Graphics, Ltd.*. DirectX version 9 supports 5 types of DXT-compression: *DXT1*, *DXT2*, *DXT3*, *DXT4*, and *DXT5*.

"In DXT compression, images are divided into a block of 4x4 texels. For each texel, two color values are chosen to represent the range of pixel colors within that block, and then each pixel is mapped to one of four possible colors (two bits) within that range. The compressed texel is represented by the two 16-bit color values, and 16 2-bit pixel values, totaling 64 bits of data per texel, amounting to an overall image size of 4 bits per pixel. [22]

"Alpha (transparency) information in DXT is handled in one of several ways, depending on the DXT format. In DXT1, each texel can either be defined as having four possible color values within the range (as described above), or alternately three color values and one value indicating "this pixel is transparent". Thus, in DXT1, one can have at most 1-bit (on or off) transparency in the image, but even this is done at the expense of some color information. In DXT2/3/4/5, alpha information is specified using a second 64-bit block for each texel (thus doubling the image size). In DXT2/3, for each pixel, four bits are used to indicate its alpha, providing 16 different transparency levels. DXT4/5 uses a method similar to the way color data is stored to provide "interpolated" alpha information: Two (8-bit) alpha values are chosen representing the range of transparency in that texel, and then for each pixel, three bits are used to represent its transparency within that defined range (This allows much better for subtle gradations, but can have less precision for large ranges within a texel). [22]

"In DXT2 and DXT4, the pixel color values are multiplied by the alpha values before compressing (so partially transparent pixels have a color value stored darker than it shows onscreen, and completely transparent pixels always have a color value of black

in the compressed texture). This can speed up some types of compositing operations, but it has the side-effect of losing color information, and can result in uglier DXT compression for some types of textures." [22]

For the CCP application, DXT1 is used exclusively for space-saving reasons, since alpha channels are rarely needed. In the case where the alpha channel is needed (the courtyard fence texture), only 1 bit of alpha is needed, which DXT1 satisfies.

Glossary of Acronyms

AA – Antialiasing

AAA – Adaptive Antialiasing

AAF – Advanced Anisotropic Filtering

AF – Anisotropic Filtering

API – Application Programming Interface

ASD – Adaptive Sampling Divisor; a divisor used to tradeoff adaptive antialiasing performance / quality

CatAI – Catalyst AI

CCC – Catalyst Control Center

CCP – Catalyst Control Preview

CPU – Central Processing Unit

D3D – Direct3D

D3DX – a library for use with Direct3D, with many common graphics operations

DDS – DirectDraw surface

DLL – Dynamically-Linked Library

DX – DirectX

DXT / DXTC – an effective texture compression method originally developed by *S3 Graphics, Ltd.*

DXUT – DirectX Utility Toolkit, a framework for useful for typical Direct3D applications

GI – Geometry Instancing

GPU – Graphics Processing Unit

GUI – Graphical User Interface

IDE – Integrated Development Environment

LOD – Level of Detail

MipLOD – Mipmap Level of Detail

MSAA – Multisampling Antialiasing

R*00 – a particular series of ATI GPU hardware

R300 – the minimum generation of ATI GPU hardware supported by the CCP application

R500 – ATI's X1xxx series of GPU hardware

SDK – Software Development Kit

TAA – Temporal Antialiasing

UI – User interface

VS – Visual Studio

VSync – vertical sync

WDDM – Windows Vista Display Driver Model