Yaron Guez

CPSC 490b: Special Project

Written Report

Combining Data Sources in a 3D Rendering of the Yale Campus

My CPSC 490 Special Project senior requirement was to create a navigable 3D rendering of a portion of the Yale University campus in order to learn the C# programming language, the .NET framework, the Direct3D graphical language, the creation and manipulation of computer aided design (CAD) files, advanced 3D modeling, digital photo editing and preparation, and the integration of various data from each genre in order to create a seamless experience for the user.

The first stage of my project was to purchase and read Apress’ C# and the .NET Framework by Andrew Troelson, 900+ pages. C# itself is an object oriented program language with a syntax that is similar to C++ and Java. I found the language to be extremely intuitive and robust despite its simplified symbolic requirements. It also helps to enable Rapid Application Development (RAD). Through the course of reading the book I went through several programming exercises covering basic syntax, object oriented programming, .NET interfaces, collections, and GUI design.

C# functions as one of the implementing languages of Microsoft’s .NET initiative utilizing the Common Language Runtime (CLR). The CLR is, in a sense, a virtual machine capable of reflexive execution (an awareness and an ability to modify its program structure and execution during runtime) that supports over 40 “.NET aware” languages. Using the .NET runtimes installed on a user’s computer, the CLR reads and runs Microsoft Intermediate Language (MSIL, often referred to as Common Intermediate
Language CIL) outputted by a .NET capable compiler. The Intermediate Language is a stack based language similar to assembly code and is the lowest-level programming language in the .NET framework that is still human readable. The compiler will output the same Intermediate Language for a given procedure regardless of what language it was coded in, provided that the language is “.NET aware.” .NET “awareness” is a term given to a language that satisfies the specification of Microsoft’s Common Language Infrastructure (CLI). The CLI provides a set of specifications, including a Common Type System (CTS) of types and their respective supported operations, a set of rules for interacting with other .NET aware languages (Common Language Specification: CLS), and a standard for providing Metadata about program structure. Languages satisfying these requirements are able to interact with each other on any platform with a corresponding CLR, regardless of computer architecture. The separate, language independent components are loaded, combined, and executed as machine code by the Virtual Execution System (VES) according to the outputted Metadata and the platform/architecture specifications provided in the CLI. Languages that are compatible with the CLI allow a programmer to write programs for several different platforms as well as work with other code written in other .NET aware languages without having to rewrite instructions for different architectures.

In addition to the benefits of the CLR, the .NET framework itself serves as an API, providing numerous “canned” implementations of common programming requirements such as graphical user interfaces (GUI), operating system events, data access, cryptography, hash functions, network communications, and numeric algorithms. The framework also manages the execution of .NET programs, handling memory
management, exceptions, and security issues. Version 1.1 of the .NET runtimes is installed by default with the latest versions of Windows and/or Windows update. Version 2.0 is an optional Windows update component and is usually bundled with software written with .NET 2.0 compilers.

Initially I used Microsoft Visual Studio 2005 for my project. By default, VS 2005 compiles using .NET 2.0. In order for users to execute my program, they would have to download and install the .NET 2.0 runtimes, a lengthy (20+ minute) process. Additionally, Managed DirectX 2.0 (as defined later) was significantly less stable in trials among peers than Managed DirectX 1.1. Accordingly, I installed and used Microsoft Visual Studio .NET 2003, which compiles using .NET 1.1, for the rest of my project.

The next stage of my project was to learn Direct3D, the 3D graphical language and API component of DirectX, Microsoft’s game programming API. I learned that the standard versions of Direct3D, and DirectX in general, were not supported by the .NET framework. Instead, Microsoft had produced a new version of the API referred to as Managed DirectX, or MDX. MDX is relatively new and does not support all of the features of standard DirectX. In December 2005, around the time that I started this stage of my project, Microsoft released MDX 2.0 in order to support version 2.0 of the .NET framework. This was a significant obstacle for several reasons. The distinctions between DirectX and MDX were not clear to begin with, let alone those between MDX and MDX 2.0. Thus, I found myself lost between different standards, documentation, usability, stability, and features. Furthermore, a majority of the MDX online community, which I was depending on for tutorials and user support, was unavailable as sites began switching over to the new format. Since I was using the latest version of Visual Studio I was
compiling in MDX 2.0 by default without realizing it and ran into countless problems getting tutorials to compile based on the format difference. When the site which I was using as a learning center finally returned with MDX 2.0 tutorials, I learned that MDX 2.0 was still considered to be in the beta stages of development, which explained the instability I was finding and the inability for my program to run on any computer but my own. At that point I considered reverting to standard unmanaged DirectX, or even the open source graphical language OpenGL, considered a standard by many in the graphical community but used less and less in the game development community. In order to justify the time and effort spent on learning C# and the .NET framework I opted only to abandon the tutorials and MDX 2.0 game engine I was learning, install Visual Studio .NET 2003, and find tutorials and a game engine utilizing MDX 1.1

Though Direct3d itself is an API, when working in the Direct3d environment it is often recommended to utilize a second layer of abstraction. Sometimes referred to as graphic frameworks or game engines, these APIs avoid having to “recreate the wheel” and streamline the development process, though there is often a performance cost. The first MDX 2.0 engine I worked with was entitled the C-Unit Framework and was geared more towards learning MDX then for game development. I then found a site entitled “The Z-Buffer” dedicated to MDX resources with a list of game engines that utilized MDX. There were dozens. Some were written in MDX, others MDX 2.0. There were some with open source, partially open source, and libraries only. I had to consider library documentation, tutorials, community support, stability, mesh compatibility, MDX version, and learning curve. Each took a sizeable investment of time in order to accurately determine if it met my needs or not. The three most notable graphic engines
that I worked with were Irrlicht, Purple#, and Artificial Engines. Irrlicht supported a very wide array of mesh types including 3DS, .X (the DirectX mesh format, which is nearly impossible to google), and Quake 3 maps among others. It was open source, well supported and maintained, fast, and stable. Unfortunately it was not actually written in .NET but has .NET libraries that were ported from C++. While documented well, these MDX libraries did not have all the features of their unmanaged DirectX counterpart. Furthermore there was only one tutorial written for the .NET version, as opposed to 14 in unmanaged DirectX. Purple# had adequate documentation, a strong community, and a wizard to set up a new application, but did not support .x file meshes.

I eventually settled on Artificial Engines for its integration into the Visual Studio IDE, its strong and quick-to-respond community, its stability and ease of use, and the fact that it was completely native to .NET. However there were only four tutorials, it was more geared towards VB.net than C#, there wasn’t any documentation, it wasn’t open source, and near the end of my project when I needed help the most, the support boards took a long time to respond and were not very helpful.

My desire to learn C# and the .NET framework are now disjoint from my ambition to create a 3D rendering of a portion of campus. Knowing what I now know about the environment, had I simply wanted to do the latter, I would surely do it in unmanaged DirectX and use the Irrlicht engine.

While dealing with the issues of MDX I was concurrently dealing with getting the information for my model of cross campus. I made an appointment with the facilities office and met with David Kula, CAD Manager. He burned me a CD several DWG files, the AutoCad drawing format. I found working with AutoCad far more challenging than I
had foreseen. Each of the files contained a component of a map of campus. One file contained the roads, another had the layout of campus building, another had New Haven buildings, etc. Working with the help file, at first I joined the files together by created external references, or X-Refs. This resulted in circular dependencies. I traced the debugging output to find the source of the circular dependence and found that one file already had references formed to the other files. Executing a `bind` command imbedded the various maps into one resulting in a huge file that was slow to work with and hard to break apart. Additionally the map was 2D and without height coordinates. I was in semi frequent e-mail correspondence with Mr. Kula who recommended the `explode` command to separate the different splines on the map in order to work with them. Because of the size of the map, every `explode` command took considerable time to execute and often splines were still grouped by some other reference. Because the files were organized by component of the map, and not area, I could not block off a select area to work with. Eventually I was able to isolate one building at a time in order to export it to a modeling program. However, I still needed the height values. After I e-mailed Mr. Kula asking for the location of the height values, he responded with a new set of DWG files. He said that these were from an older map and that he had no idea how they were set up, but they had height values on one of the layers.

The zip file that he sent me contained dozens of numbered DWG files, some with external references to others. By sorting through them all I found the base files and found that they were in fact organized by chunks of campus. At this point I decided to model cross campus since I found one file that was included that area of campus. The 2D drawing contained building layouts, height markers, the layout of trees, walkways, stairs,
roof tops, lamp posts, fire hydrants and more. The resulting map was cluttered to say the least.

At the same I was going through the lengthy but extremely helpful tutorials that came with 3D Studio Max. I went through lessons on basic modeling, dealing with objects and sub-objects, materials, and the complicated user interface. One tutorial was on cleaning up, importing, and building off of CAD files. I followed it and was able to clean up the cross campus map by freezing unnecessary layers, and setting the settings correctly when importing the DWG file into 3D Studio Max. The import was too small, too far from the origin to find, and not aligned with the axes. I returned to Autocad and to the Autocad help file and found how to move the origin, rotate and align the world axes, and create new blocks consisting of individual buildings to import one at a time with the correct relative distances. I read about units setup in 3D Studio Max and set the System Units to feet, while importing as inches. The block files could then be imported separately, aligned close to the origin yet with the correct relative distance, and aligned properly relative to the axes.

At first I attempted to use the existing splines that were imported and extruding them based on the height values. This resulted in missing faces, discoloring, and skewed models. Breaking apart the splines by vertex led me to discover that the vertexes were not connected and that the file was just sketches, not accurately constructed layouts. I used the sketches as a blueprint, or stencil, and constructed new lines over the existing ones. During the process of modeling I learned several techniques to aid in the effort. A factor that had to be considered was that of relative heights. Ground level had a minimum height of 36 feet so all the values had to subtract 36 before being used. The sketch was
also not entirely accurate, especially when dealing with structures such as windows on
the roof, which had to be constructed line by line according to a picture, then giving an
extrusion of 0, then converted to an editable polygon in order to texture map. Snaps could
be configured to aid in the drawing, forcing the line to “snap” to existing vertices, edges,
faces, or grid lines. The down side of this was that drawing could not be restricted to a
single plain, and often times vertices in a flat structure such as a walkway would have
varied Z coordinates and thus needed to be set individually after the fact. Modeling could
be sped up by cloning existing structures. 3D Studio Max gives the option for a reference
or a copy clone. Curved items such as walkways corners could be rounded off using the
fillet tool on individual vertices. 3D Studio gives the option to select an object, e.g.
spline, polygon, or mesh, as well as sub-objects such as segment, vertex, face, edge, or
border. When transforming objects or sub-objects there is the option of relative versus
absolute transform. When drawing complicated 3D objects, vertices could be fused
together using the fuse tool, simplifying the mesh drastically. There were numerous
modifiers which could be applied to objects including extrude, taper, bend, and UVW
map. Before learning the concept of constructing 3D objects with a 2D line, an extrusion
of 0, vertex snaps, and fused vertices, I would often use taper and bend modifiers on
boxes to form roof tops. I also learned that taper and bend only work when the object
being modified is divided up into several segments within the object parameters. 3D
Studio had multiple pre-made objects. I used stairs and doors several times, both of which
were completely customizable.

Since the sketch was not 100 percent accurate, I would often need to use digital
photographs as a guide. Since I was not at the texturing stage of modeling yet I did not
have pictures on hand. Online image databases such as Google image search at
http://image.google.com and the Yale Digital Manuscripts and Archives database at
http://mssa.library.yale.edu became invaluable resources. The Yale office public affairs
also had a number of valuable images however their database was down and they were
inaccessible. I was able to get around this by using the Way Back Machine at
http://www.archive.org, a company which runs regular archival scans of most of the
known internet.

In order to display the mesh in my graphics engine I needed to export it as a .x file
mesh. The DirectX SDK (software development kit) had originally included an exporter
tool for 3DS Max but it did not support the latest version. The website for Artificial
Engines, the graphics engine I was using, had an exporter on their site, however it would
often fail. I tried several others before finding a free Pandasoft DirectX exporter plug-in
which supported the latest version of 3DS Max and had numerous settings for animation,
texture mapping, coordinate systems, normals, and file settings, including the option to
output in ASCII or binary. For awhile my models would display from the bottom instead
of straight on. I used a rotation matrix in the graphic engine to adjust this but the result
would still be backwards. Looking into the problem I discovered that 3DS Max uses a
right hand coordinate system whereas DirectX uses a left hand one. After coming up
with several ways to accommodate this I found a feature in the exporter that would
convert coordinate systems automatically. Often times an exported mesh would fail to
load in the graphics engine or in the DirectX mesh viewer which came with the SDK.
Through trial and error I found several issues that could cause a crash. Object names
could not contain symbols such as colons. Multi-Object textures would display but using
separate mapping channels for separate sub-objects, each having its own UVW map (such as a one UVW map for the side of a building and another for the roof) would cause a crash. Under this circumstance I had to convert the object into an editable polygon, select one polygon as a sub-object (such as the face of a building) and use the detach function to make that into a separate object which could then have its own disjoint UVW map. As a result, my final model is comprised of several separate faces which had been taken from an editable polygon which had been converted from an extruded line.

In order to work effectively and avoid system slow down during the modeling of Berkeley North, Berkeley South, WLH, Sprague, and the surrounding area (grass, walkways, CCL, and the sidewalk), I utilized 3D Studio Max’s selection set feature and created selection sets for BK North, BK North Markers (the imported Autocad sketch), BK South, BK South Markers, WLH/Sprague, and WLH/Sprague Markers. I incorporated the surrounding area into the selection set that was closest and therefore could export three separate meshes (BK North, BK South, WLH/Sprague) to be displayed by the graphics engine. This would allow me to incorporate level of detail features such as loading a mesh only when it’s either directly visible or if the camera is within it had it been necessary. However the program never slowed down or had difficulty loading the three meshes at startup so this feature did not have to be implemented after all. Also, the graphics engine already uses an is-visible test before processing any part of a loaded mesh.

I often consulted with several online forums regarding issues with 3D Studio Max, Autocad, Photoshop, or the graphics engine. The most notable example was with regard to a problem I was having with zooming and panning within 3D Studio. Often
times and without warning I would suddenly be unable to zoom and pan quickly in the
perspective view port. The further I would zoom the slower the rate of zoom would be as
well as panning. In order to change angles or look at something else I would have to
zoom out far, pan, then zoom back in. What started out as an occasional nuisance
became more common as my mesh grew to the point that I could hardly work. I scoured
the help file and found nothing and tried various tricks to speed up viewport rendering
thinking it was a question of processor use to no avail. The problem occurred only in
perspective view, and occurred regardless of whether using display wireframe or shading
or using adaptive degradation. After a day of troubleshooting and an extensive search
online I found a post on a 3D modeling website from 2002 with a long chain of users
talking about the same issue without solution. The last post solved the problem, advising
the user to select something near to what you are looking at, and then pressing “z” which
forces the program to zoom on that object, stopping the slowdown.

The last stage of my project was that of texture mapping. I first completed several
tutorials on the topic. 3D Studio lets you create materials which can be applied to
objects. As long as the material is standard, or a multi-material/sub-object material made
up of standard sub-materials, the exported mesh would display properly. I had a problem
displaying textures on a computer other than my own at first. I solved this by disabling a
feature that exports the full pathname to the texture as opposed to the relative one within
the exporter plug-in. After creating and applying a material to an object or sub-object, I
had to apply a UVW map modifier. The modifier had several types including plane, face,
spherical and box. Using a box type was the most convenient as it allowed you to move a
box gizmo representing the texture’s location, making it easy to position and size the
texture exactly how I wanted it, though it took some getting used to. As described earlier, objects requiring different UVW maps for different sub-objects (sides of a building) needed to be broken up into separate objects due to a limitation of the exporter. However if one UVW map was suitable for all sides then sub-objects could be texture mapped using multi-object materials. Each sub-object, such as a polygon, was given a set ID within the sub-object parameters. Then instead of creating a standard material I would create a multi-object material and define the number of materials needed. Next I would drag-and-drop previously or separately defined materials into slots corresponding to the appropriate set ID. Once the new multi-object material was dropped onto the object, each sub-object automatically received its corresponding material without needing to be broken up into distinct objects. Again, this method only worked if all sub-objects used the same UVW mapping.

The greatest challenge was preparing the actual textures. I took pictures with my digital camera of the buildings and other objects to be mapped, however they were often obstructed by trees, adulterated with shadow, skewed due to the perspective of the camera looking up at the building, and with uneven image levels. I spent a huge amount of time using Adobe Photoshop to compensate and correct for these effects. I also discovered ways to improve and make the process easier.

First I would use the vanishing point filter to correct the effects of perspective. While there was a perspective transform tool, it didn’t work nearly as good or as consistent as the vanishing point filter. Using the filter, you first outline a plane within the skewed image, such as that of a window. You can compare the accuracy of your outline by stretching and/or moving the plane around the picture to see if lines up with
other lines within the image. The program adjusts the plane accordingly, skewing it as
you move it based on the perspective you defined. If the plane was inaccurate, or formed
an impossible perspective the plane would glow yellow, or red, depending on how far one
was from a possible perspective. Next I would create another plane with perfectly
straight dimensions. I would use the outline of the picture as a guide and then shrink the
new plane representing the camera’s view so that it was out of the way. Next I used the
rectangle selection tool within the first plane, representing the frame of reference of the
image. The selection tool follows the same skew as the plane did as you moved and
resized it. Once the entire image is covered, you press alt and click and then drag the
selection into the target camera plane. The image is automatically adjusted to the frame
of reference of a dead-on camera. Often additional resizing and skewing would be
necessary to fine tune the edges so that they were all parallel with the screen. Next I
would fill in the gaps that resulted from the filter as well as any part of the image blocked
by foliage or shadow. This was done with a combination of the clone stamp tool, which
would copy from one target to another based on the mouse location, as well as basic
rectangular selection copy and pasting. I used a mirror effect to prevent noticeable tiling
within the image and would merge and duplicate layers along the way so as to
exponentially decrease the number of copy/pastes needed. If part of a window was
blocked, I would copy the other half of the window, and flip it over. This also helped to
deal with perspective issues that could not be solved with the filter, such as if an angled
window were captured from the side, hiding the inset of the opposite side. To cover up
bricks that would not line up perfectly I would use a low opacity clone, or blur tool to
smooth over the intersection. Finally I would flatten the image and use image levels,
As I progressed, I picked up other techniques. Planning ahead was extremely crucial. I would first take a QuickTime video clip with my camera that I would narrate so as to get a general idea of the placement of items within the scene, and then would take isolated pictures of individual components, such as an area of blank brick, or a single window or door. I created templates for different buildings, consisting of a background base of tiled brick and several layers for different kinds of windows that the building incorporated in its design. In this way I could piece together an accurate portrayal of a side of a building without needing a perfect picture of it. Often a video was suitable if I had a good enough template formed from a previous side. I spent a great amount of effort on this part of the project as it was important to me to create an accurate portrayal of the buildings modeled, not just a rough estimate. I would often change the structure of a building to be more consistent with the image, such as moving roof windows to line up with the texture, or moving a gate or side walk to line up with the image of a door. While the process was painstakingly long, it was tremendously rewarding to see the building come to life little by little with every mapping. I tried to maintain this sense of realism in elevation as well, consistently maintaining subtle inclines and declines that corresponded with the ground heights indicated in the AutoCad file.

One problem I ran into during texture mapping was that of mirrored tiling. 3D Studio Max gave the option for mirrored tiling of a material to hide the tiling effect, however this feature was not exported and therefore maps of grass, dirt, or other tiling on the order of 20x20 would look extremely tiled. I first read up on the .x file format so that
I could find a way to add the mirror effect manually. However I figured out a shortcut instead. I did a 4x4 mirrored tiling with Photoshop of the original texture before importing it as a material. In that way even though the tiling within 3D Studio Max was straight (not mirrored), the resulting texture would have the mirrored effect and appear smooth.

To create the background sky I added a 2D sprite within the graphic engine, mapped with an image of a blue sky, that remained locked in the background regardless of camera position and was not shaded.

For camera movement I utilized class enumerations defined by the graphics engine as well as .NET functions for getting and responding to input. Unfortunately, I learned that in order to add collision detection to prevent walking through walls and falling through the ground once gravity was added, would require a complete overhaul of the program and mesh. This was a flaw of the graphics engine as it recognizes each .x file as a single object and is unable to form bounded boxes and spheres around individual components of the mesh. A week after I inquired about this fact in the message board, a developer told me that I could get around this by using the available physics engine, Artificial Reality, and add forces based on gravity, and geometric constraints based on the layout of the mesh. This would essentially be recreating the model with abstract shapes recognized by the engine corresponding to walls and the ground and is equivalent to another project all together. Since I’ve become so wrapped up in the project, I intend to port it over to the unmanaged DirectX and the Irrlicht engine which utilizes a more straightforward approach to collision detection.
Overall this project has been a tremendous learning experience. While the resulting program looks great it is not a fair representation of the sum of the project’s parts over the course of the semester. Besides learning C#, .NET and Managed DirectX, a project in and of itself, I gained a thorough understanding of advanced modeling techniques, image alteration, architectural and design methods, relying on an open community for technical support, and most importantly, the integration of all these techniques, methods, and data sources towards a cohesive and tangible end product. I plan on continuing this project after graduation in the hopes of both finishing what I started and eventually having a program portable enough so that Yale can utilize it on its website.