

Driving Simulator Use in the Roadway Design and Planning Process



SAFETY RESEARCH USING SIMULATION

UNIVERSITY TRANSPORTATION CENTER

Shawn Allen, BFA
Program Mgr., Transportation
Visualization
NADS
University of Iowa

Driving Simulator Use in the Roadway Design and Planning Process

Shawn Allen, BFA
Program Mgr., Transportation Visualization
NADS
University of Iowa

Amanda Beadle
Undergraduate Student
Engineering
University of Iowa

Vincent Horosewski
Application Programmer/Analyst
NADS
University of Iowa

A Report on Research Sponsored by SAFER-SIM

University Transportation Center, The University of Iowa

May 2016

Table of Contents

Table of Contents.....	ii
List of Figures.....	iv
List of Tables.....	vi
Abstract.....	vii
1 Texture Mapping Algorithm & Tool Development	1
1.1 Texture Mapping Overview	1
1.2 Texture Mapping Test Methods.....	1
1.3 Source Model File Format	8
1.4 Development Approach.....	9
1.4.1 Programming Environment.....	9
1.4.2 Test Models	9
1.4.3 Geometry Coordinate System	12
1.4.4 Texture Test Map	13
1.4.5 Texture Application Methods	13
1.5 Texture Mapping Algorithm Solution	21
1.6 Future Work.....	22
2 Tile Model Integrator Tool	24
2.1 Tile Model Library.....	24
2.2 Tile Model Library Overview	25
2.3 Content to Simulator Overview.....	25
2.4 Tile Mosaic Tool	26
2.5 Tile Model Integrator Tool Description	28
2.6 Integrator Tool Dependencies	28
2.7 Integrator Tool User Interface	30
2.8 System Configuration Requirements.....	32

2.9	Python Requirements.....	32
2.9.1	Python Modules.....	32
2.9.2	OpenFlight™ API Requirements.....	32
2.10	Integrator Tool User Interface	33
2.11	Tile Model.....	38
2.11.1	Tile Model Files	38
2.12	Associated Model File Set.....	39
2.12.1	Import Model Requirements	40
2.12.2	Modelling Conventions	41
2.13	Future Work.....	41
	References.....	43
	Appendix A: Sample OBJ File.....	44
	Appendix B: OBJ File Format Specification	46

List of Figures

Figure 1.1 – Geometric and texture coordinates	1
Figure 1.2 – Imported geometry sample	2
Figure 1.3 – Flow texture on imported mesh shows a tangle of UVs	3
Figure 1.4 – Flow texture model results with texture landmark	3
Figure 1.5 – Planar texture mapping UVs	4
Figure 1.6 – Planar texture mapping simple noise image	4
Figure 1.7 – Simple texture pattern driver view	5
Figure 1.8 – Texture skew artifact	6
Figure 1.9 – Manual UV adjustments	7
Figure 1.10 – Complex ribbon model	7
Figure 1.11 – Single (left) and multi-span (right) geometry ribbons	10
Figure 1.12 – Test texture map	13
Figure 1.13 – Texture mapping planar/stamp projection	14
Figure 1.14 – Side (elevation) view of model	14
Figure 1.15 – Test model 2 top (plan) view	15
Figure 1.16 – Test model 2 side view and segments	15
Figure 1.17 – Texture mapping on variable width geometry	16
Figure 1.18 – Model with negative X-axis coordinates, top and side views	16
Figure 1.19 – Texture mapping distortion test	17
Figure 1.20 – Texture mapping face pairs	19
Figure 1.21 – Texture mapping face groups	20
Figure 1.22 – Texture mapping universal min X, Y	20
Figure 1.23 – Texture mapping algorithm	22
Figure 1.24 – Texture mapping algorithm results	22
Figure 2.1 – Roundabout tile model and configuration layout	24

Figure 2.2 – NADS content pipeline: source to simulator.....	26
Figure 2.3 – Category operations panel after successful initialization	33
Figure 2.4 – Color picker	34
Figure 2.5 – User feedback for selected category.....	34
Figure 2.6 – New comm_2 category inserted.....	35
Figure 2.7 – Import model panel	36
Figure 2.8 – Data entry in progress.....	36
Figure 2.9 – Data entry completed	37

List of Tables

Table 1.1 – Test case geometry	10
Table 1.2 – OBJ face geometry definition	18
Table 2.1 – 3D file formats supported by the converter engine.....	29

Abstract

This research project was the Initial Collaboration Project for the University of Wisconsin-Madison and the University of Iowa National Advanced Driving Simulator (NADS) as proposed to the USDOT in the SAFER-SIM proposal under *Theme Areas: 1. Using driving simulators to conduct virtual road safety audits, and 2. Using simulation in the roadway design process to drive the road before it is built.*

The purpose of this project was to demonstrate that safety-centered road designs and evaluations that rely on human-in-the-loop simulation through the use of driving simulators could be conducted using different simulator platforms. This was accomplished by demonstrating that a core scenario compatible across multiple driving simulation platforms could be created using standard 3D modelling practices and custom software tools and by leveraging existing simulator scenario authoring tools. For the purposes of this research, the core scenario was defined as the visual environment and the road surface definition required for the subject to ‘drive’ the scenario.

This project utilized two different simulator architectures:

1. At University of Wisconsin Madison (UWM) – Realtime Technologies, Inc (RTI).
2. At University of Iowa National Advanced Driving Simulator – NADS MiniSim™.

The primary development performed by NADS included development of texture algorithms and software to apply texture to arbitrary ribbon geometry (e.g., a road surface), and development of a software tool and workflow to facilitate managing the NADS tile model library, which is a collection of 3D visual models with associated meta-data used to create driving simulation environments. This tool would automate the management of tile categories and permit importing new models into the tile model library without requiring the end-user of the tool to manually manipulate or edit key configuration files. The goal for this project was to allow simulator end-users to quickly

import non-native model files into the Tile Mosaic Tool (TMT). Imported files would then be available for use as any other standard asset.

This report is divided into two sections: one describing the texture mapping algorithm and tool development, and one describing the model integrator tool for importing third party models into the NADS virtual asset pipeline.

1 Texture Mapping Algorithm & Tool Development

1.1 Texture Mapping Overview

Texture mapping is a computer graphics technology that enhances the apparent visual detail of a geometric model through the application of detailed images in the form of 2D bitmaps. Images used in this fashion are called ‘texture maps’. Each pixel in the picture has a unique texture address coordinates (U, V). UV coordinates are normalized units, in the range of 0 to 1. Texture mapping is the process of mapping the 2D picture onto 3D geometry using UV coordinates, as shown in Figure 1.1.

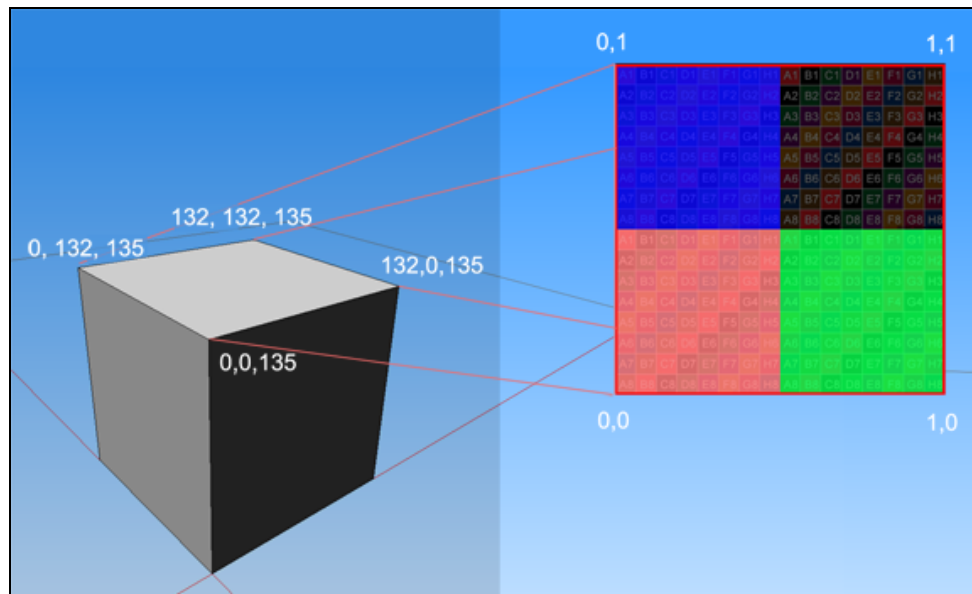


Figure 1.1 – Geometric and texture coordinates

1.2 Texture Mapping Test Methods

Texture mapping during the creation of geometry is a common operation in 3D software. However, models which are imported from third party sources (i.e., the geometry is constructed in one program, then imported into a different application) often cannot be relied upon to present a well-ordered polygon mesh. This has significant

implications for texture mapping since the appearance of textured geometry depends on minimizing visual artifacts across multiple surfaces such that the result appears to be a unified and cohesive whole. Figure 1.2 below shows a sample of imported geometry in the form of a polygon ribbon, which constitutes a roadway.

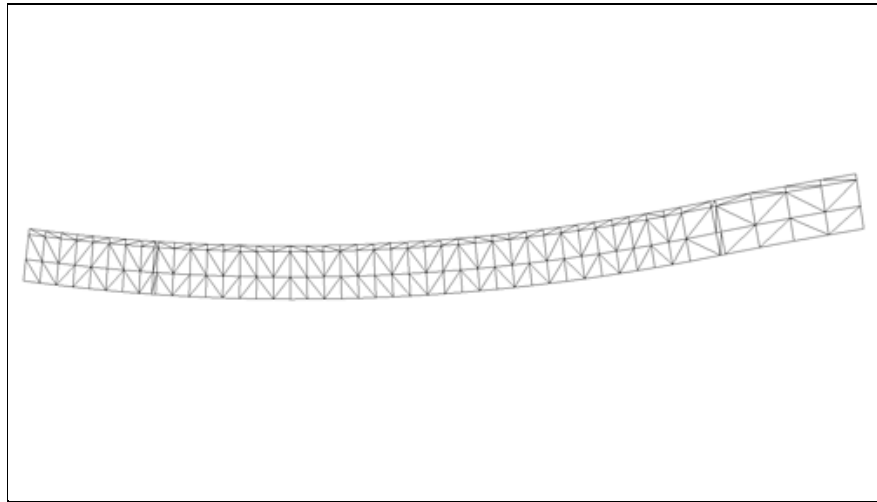


Figure 1.2 – Imported geometry sample

Flow texture mapping is a technique that applies texture onto one polygon, and then the remaining polygons are textured based on their orientation in 3D coordinate space relative to the originally mapped polygon. The process is similar to water flowing downstream; although it is reasonable to expect that the texture (water) flows smoothly despite varying topology of the streambed, in reality texture mapping is highly dependent on how the model geometry is defined.

This becomes clear when a texture containing some landmark feature, such as a stripe, is mapped to the model geometry. In figure 1.3, one triangle has been textured with the right side of the texture aligned with the right edge of the polygon. As the UV mapping shows, flowing texture from the original triangle onto adjacent polygons resulted not in a smooth, stream-like mapping, but in what looks like an arbitrary tangle

of UVs. In this example, the imported model geometry was used as-is, without editing of any kind.

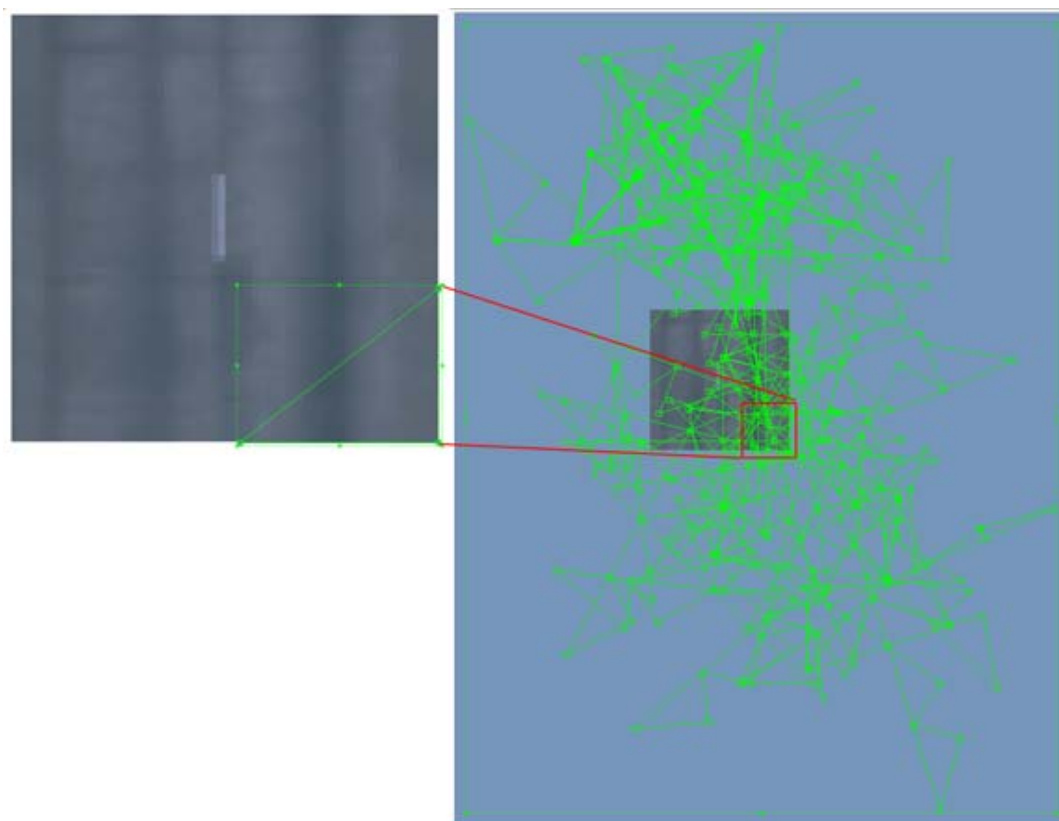


Figure 1.3 – Flow texture on imported mesh shows a tangle of UVs

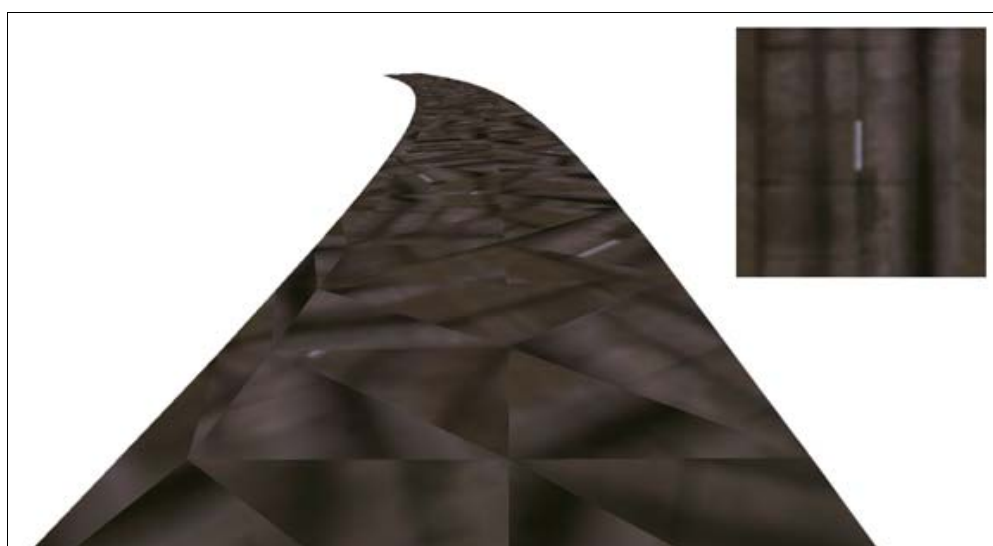


Figure 1.4 – Flow texture model results with texture landmark

Addressing the issue of arbitrary geometry orientation is possible by using a single UV mapping for all geometry, irrespective of the orientation of the geometry. The simplest way to accomplish this uses a planar or stamp projection where the texture is mapped without regard to the shape of the model, and all geometry uses the same UV mapping. This method works best with textures that are generic and lack identifiable features.

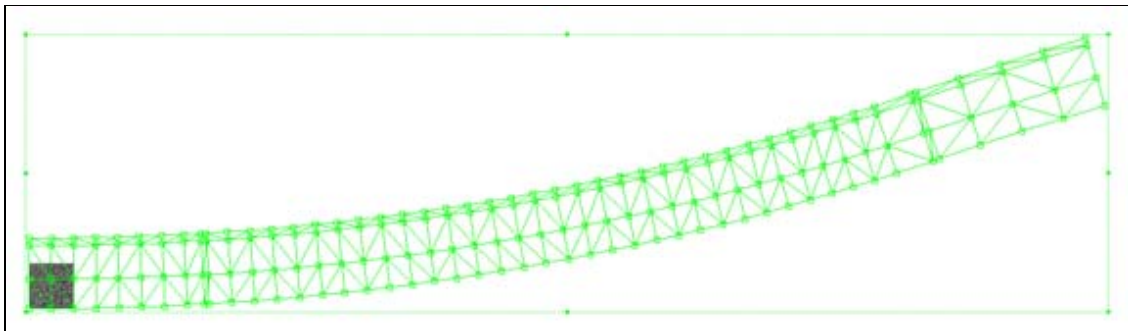


Figure 1.5 – Planar texture mapping UVs

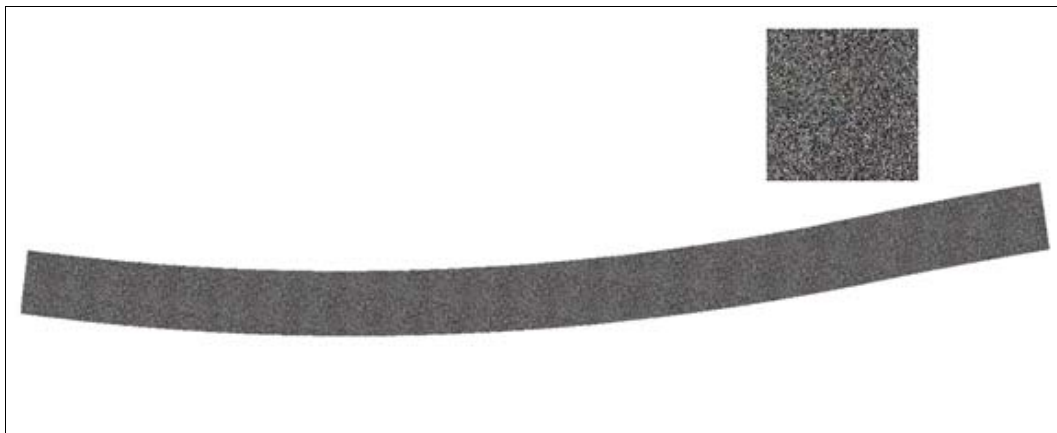


Figure 1.6 – Planar texture mapping simple noise image

Although this method of texture application is simple and fast, it relies on the general character of a texture to minimize texture artifacts and therefore the resulting textured model lacks detail. This becomes problematic for situations where surface

details are important contributors to scene quality, such as a driving simulation environment. Features and details within the environment contribute to the ability of the simulator driver to manage key attributes of situational awareness such as speed perception, depth cueing, and lane keeping. Because the generic texture lacks detail, these important cues are missing or misleading during the simulation experience.

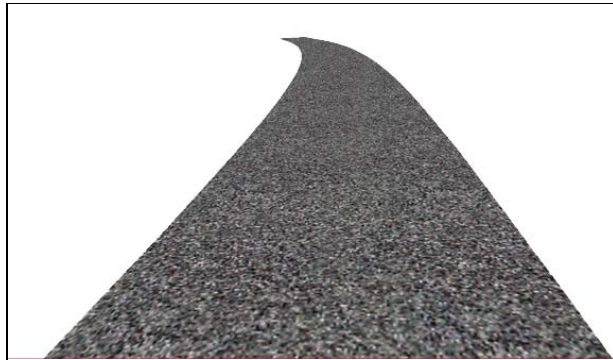


Figure 1.7 – Simple texture pattern driver view

When the texture does contain detail or repeating features, the projection method is no longer sufficient to produce a workable model unless the model is aligned on one axis. Because projection mapping does not accommodate model variations, a complex texture shows skewing, as illustrated in Figure 1.8 below.



Figure 1.8 – Texture skew artifact

Using a texture with detailed or repeating features requires a well-ordered geometry mesh, or some means to impose a consistent texture mapping to the mesh. A well-ordered mesh contains a consistent polygon definition that allows a texture to ‘flow’ along the geometry with minimal artifacts.

Correcting texture skew is generally straightforward, depending on the complexity of the model. Manually aligning UVs will ensure that texture artifacts are minimized and produce a generally pleasing model. One UV is used as an anchor and the remaining UVs are adjusted to fit the texture and the model, as shown in Figure 1.9. In Figure 1.9, item 1 shows the original mapping, item 2 shows the lower UVs aligned along their lowest V, and item 3 shows the results of manual UV alignment.

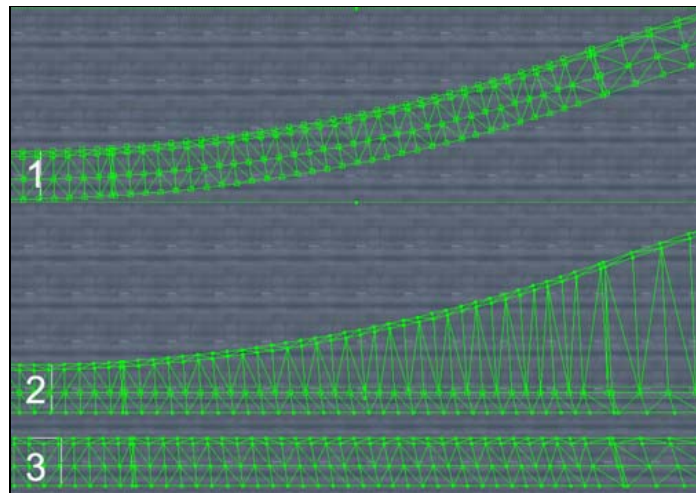


Figure 1.9 – Manual UV adjustments

The process of manually aligning UVs is very straightforward. However, manually editing massive amounts of UVs is not a scale neutral task and can quickly become tedious as the number of UVs increases. Manually editing a large scale, life-size model that spans miles or contains complex geometry is a non-trivial task because model scale and detail introduce additional complexity to the editing process.

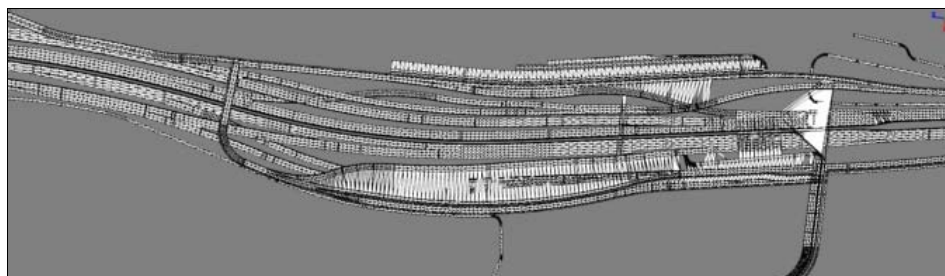


Figure 1.10 – Complex ribbon model

Applying texture to geometry without regard to geometric orientation results in obvious skew artifacts when the texture contains recognizable features, such as a stripe or other repeating element. The ideal texture mapping process applies texture to geometry while maintaining a consistent scale and minimizes texture artifacts while

respecting model geometry. Such a process would be relatively scale neutral, suitable for processing small or large models and producing results similar to manual UV mapping.

1.3 Source Model File Format

The OBJ file format for the source model was chosen based on several criteria. The primary consideration was that the source format had to be supported by both UWM's and NADS' simulator tools and resources. Because the NADS MiniSim™ render engine is based on the graphics library OpenSceneGraph (OSG), compatibility with OSG was also an important consideration. OSG supports many other 3D file formats; this means it will be possible to leverage OSG capabilities to support those 3D model formats once the initial software has been implemented for OBJ. Secondly, the file format needed to be well documented and preferably open source or publicly available.

Although OBJ is an old file format, it is well supported by free or open source modelling tools (Blender, Sketchup) as well as commercial modelling applications such as AutoDesk Civil3D™, Bentley Microstation™, Presagis Creator™, Rhino3D™, Autodesk 3DSMax™, Maya™, and many others. Lastly, in anticipation of the experience level of the application developer, the chosen file format should ideally be available in human readable form. The Wavefront OBJ file format satisfied all these requirements.

Unlike modern file formats, OBJ consists of a dual file structure: one file stores geometric data (vertices, texture vertices, vertex normals, polygons, and references to surface texture), and another file contains associated material definitions for surface attribute definition. This takes the form of "fileA.obj, fileA.obj.mtl". A simple OBJ model is included in this report in Appendix A. Links to the OBJ file format specification are in Appendix B.

1.4 Development Approach

Software available on the internet was identified as having relevance to the programming objectives:

- a) objloader ([Google Code](#))
- b) tinyobjloader ([GitHub](#))

Ultimately, these programs proved to be too difficult for the developer to integrate into this project and the decision was made to create a simple command-line program to read a simple source OBJ file. The program writes to a new output file, preserving the original file in unaltered form. The processed file is differentiated from generic OBJ files by the addition of a comment string that includes the date the file was processed:

```
#created by NADS_ObjectGo Mon Jun 29 14:37:33 2015.
```

There was an initial assumption that use of the texture tool might require iterative processing, and thus needed to maintain all previously textured data within the model. In practice, to ensure the surface decoration is applied uniformly, the model is simply re-textured using a consistent texture scale.

1.4.1 *Programming Environment*

The initial programming environment was Code::Blocks. This is an integrated development environment in use at the University of Iowa College of Engineering and one with which the developer should have been familiar and comfortable. Ultimately, this tool was abandoned for Visual Studio in response to suggestions from other NADS programmers. The texture application was coded in C++.

1.4.2 *Test Models*

The approach to testing included many geometric configurations intended to encompass the broadest variety of geometric configurations that might be encountered within a road network. The assumption was that being capable of processing these test models would result in an algorithm capable of processing real-world models without

incurring the processing penalty of large models during development. The use of single- and multi-span geometry (see Figure 1.11), axis and off-axis, single and multiple ribbons, split, flat/level, and vertically articulated ribbons covered all single-level configurations. Small test cases meant faster development cycles and were crucial for debugging purposes.

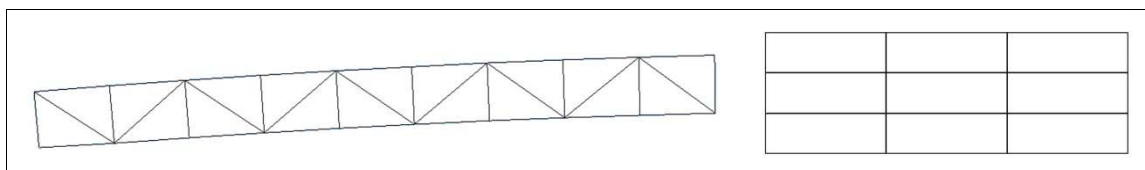
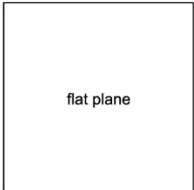

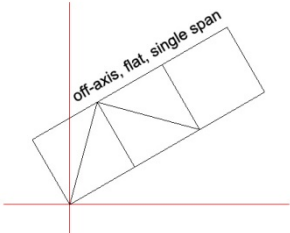

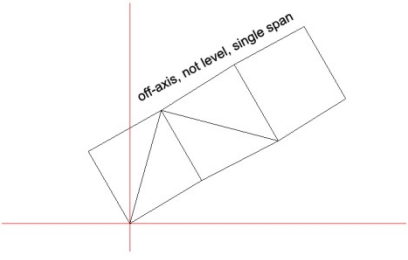

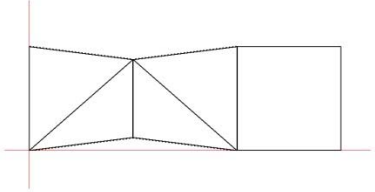

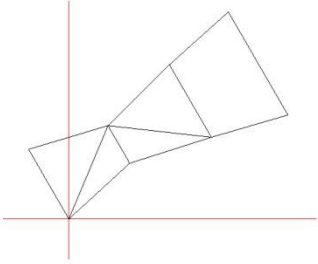

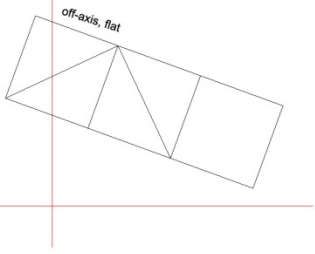

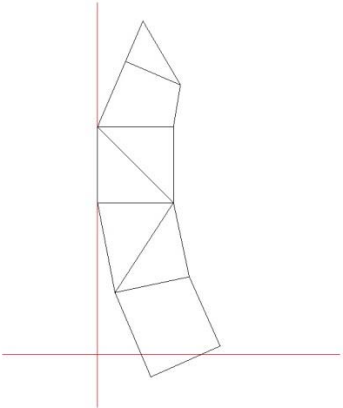



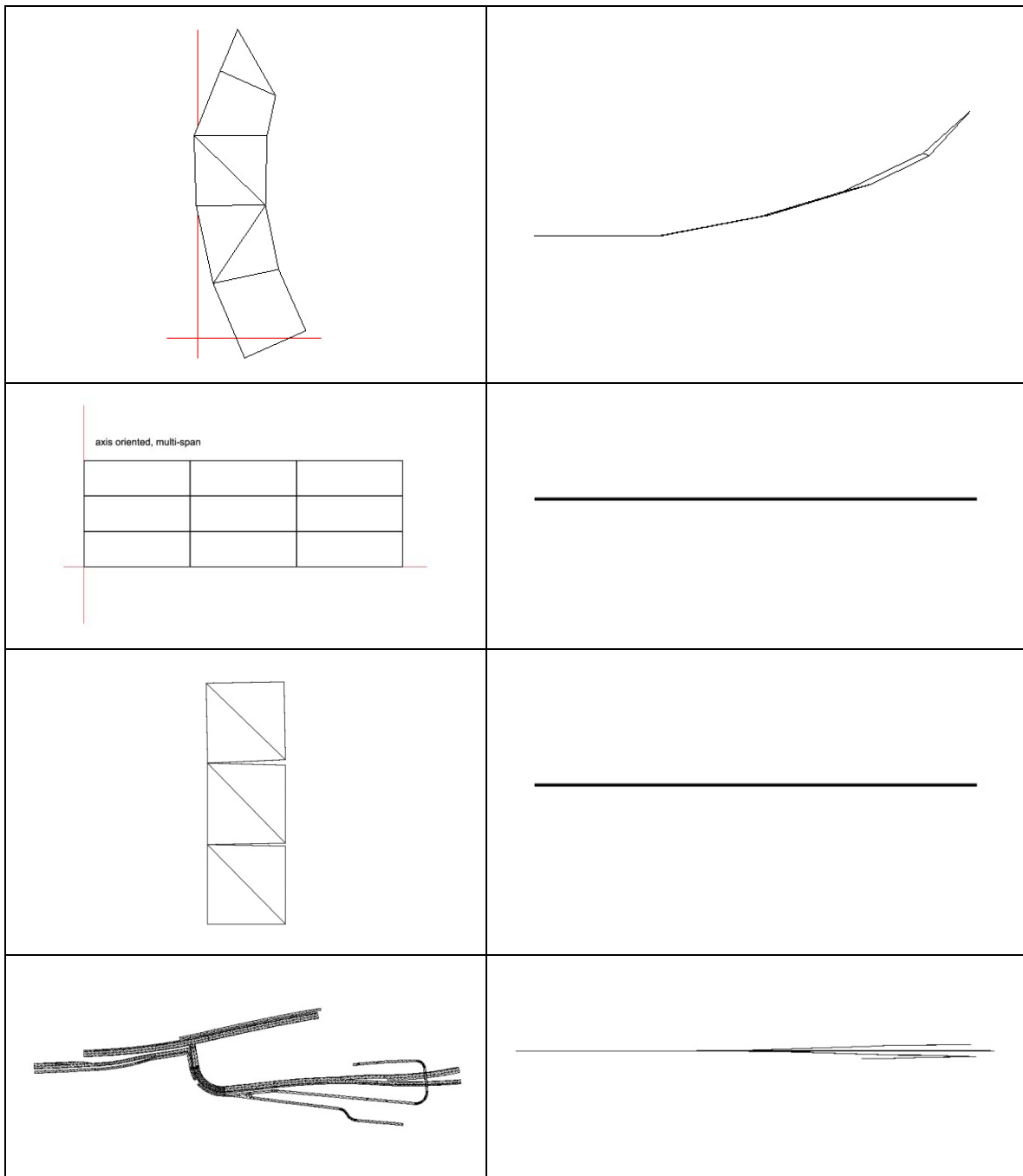
Figure 1.11 – Single (left) and multi-span (right) geometry ribbons

The tested geometric configurations are listed in Table 1.1, which shows top (plan) and side (elevation) views.

Table 1.1 – Test case geometry

Top (plan) view	Side (elevation) view
	
	

 <p>off-axis, not level, single span</p>	
	
	
 <p>off-axis, flat</p>	
	



1.4.3 Geometry Coordinate System

The texture mapping algorithm follows a coordinate convention of X (increasing to the right), Y (increasing forward). Z (elevation) was not considered for the current

version of the tool given that, in keeping with standard roadway design standards, road network elevation changes are typically proportional to road length.

1.4.4 Texture Test Map

The test texture utilized a feature-dense texture map to facilitate evaluation of texture mapping generated by the tool, as shown in Figure 1.12.



Figure 1.12 – Test texture map

1.4.5 Texture Application Methods

The simplest form of texture application is global planar surface projection, sometimes referred to as a ‘stamp texture’. This method uses the model geometry to determine the texture mapping. This method applies texture relative to the geometry mesh without respect to mesh orientation, as shown in Figure 1.13. The mapping algorithm process started with simple test cases to acquire familiarization with the processes and mechanisms necessary to create textured OBJ model files. Success in this phase was measured by ensuring that model geometry was not a limitation of the tool; therefore, test cases were constructed for various conditions likely to be encountered in real world models.

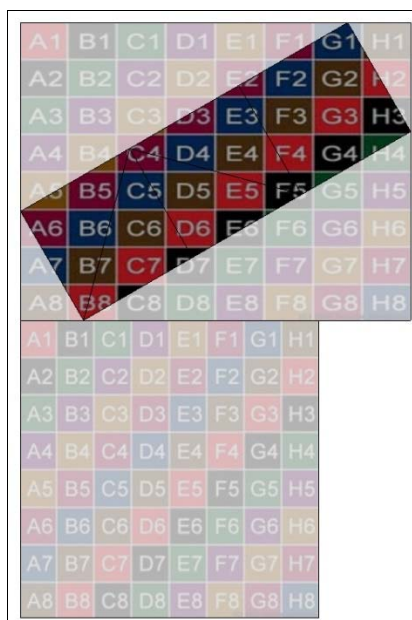


Figure 1.13 – Texture mapping planar/stamp projection

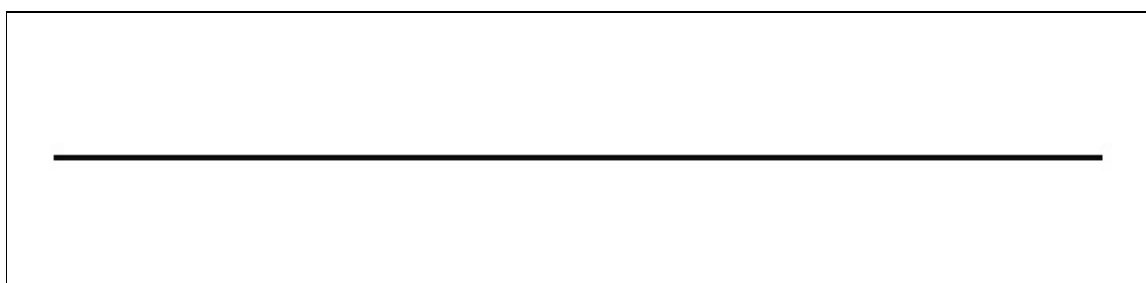


Figure 1.14 – Side (elevation) view of model

In this example, texture UVs were assigned based on the minimum and maximum vertex coordinates of the test model geometry, as shown in the upper portion of Figure 1.13. Because the geometry determines the UV extents without regard for texture map proportions, some horizontal distortion is introduced when the model and texture proportions are different. This is most noticeable when comparing the mapped texture to a non-distorted, uni-directional mapping, as shown in the lower portion of Figure 1.13.

Processing a similar test case with elevation changes showed results consistent with the flat model.

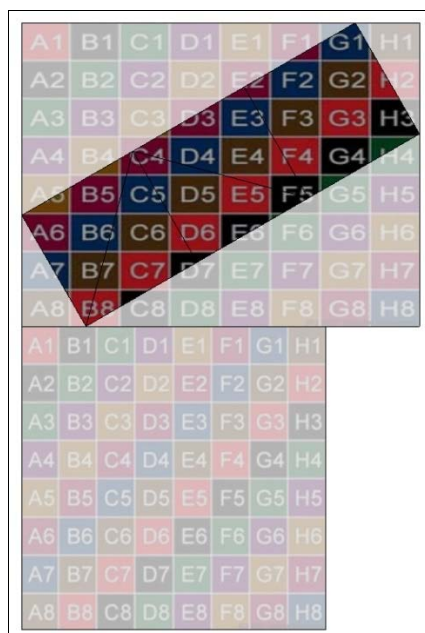


Figure 1.15 – Test model 2 top (plan) view

The elevation changes were more evident in the side view, as shown in Figure 1.16. Reading left to right, the elevation changes on each of the three segments were consistent with slopes of 8, 24 and 16 percent.

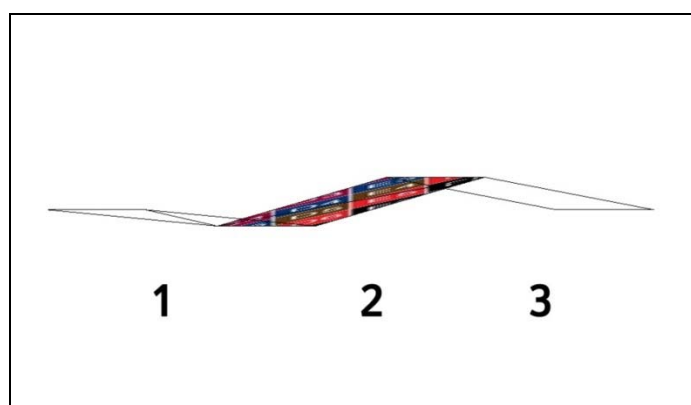


Figure 1.16 – Test model 2 side view and segments

The planar projection or stamp method of texture application ignores geometric variability in the model (see Figure 1.17). While the geometry changed width, the texture did not, and thus visual anomalies were introduced along the geometry edge boundaries.

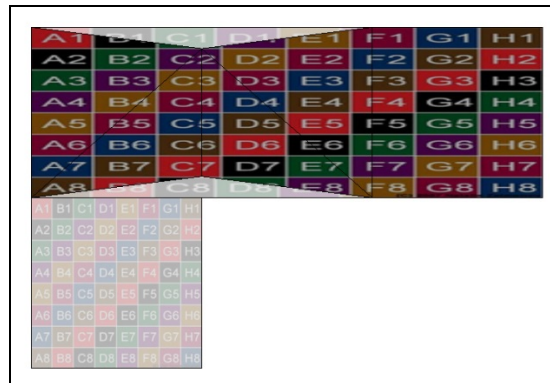


Figure 1.17 – Texture mapping on variable width geometry

Additional tests were conducted to ensure the mapping algorithm supported model geometry with negative coordinates, as shown in Figure 1.18. All boundary conditions were supported: positive, on-axis, and negative X, Y, and Z planes.

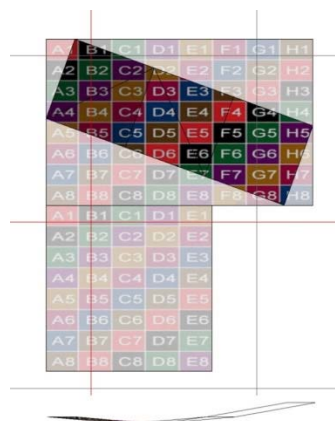


Figure 1.18 – Model with negative X-axis coordinates, top and side views

To ensure the mapping algorithm supported model geometry with significant distortion, additional tests were conducted. In the following example (Figure 1.19), the test model contained negative coordinates in the X and Y planes (shown in the top view). The numbers in Figure 1.19 indicate the side views, which illustrate the vertical geometry distortion present in the model. In side view 1, texture distortions within the top three grid rows (A3, B2, C1) are evident. This was caused by mapping distortion due to the model geometry orientation and the lack of Z coordinate support in the planar projection.

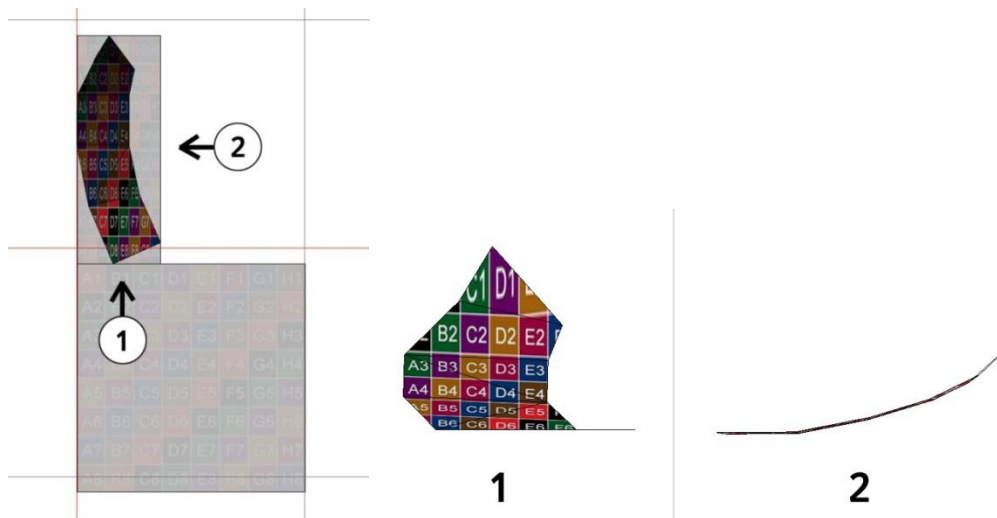
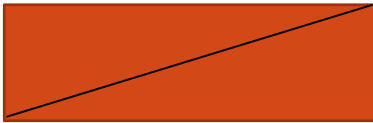


Figure 1.19 – Texture mapping distortion test

Because supporting detailed, feature dense textures is more desirable than simple noise textures, the planar mapping was considered inadequate and the method changed from processing the model as a whole to processing individual polygon faces. Faces within the OBJ file are defined using the format “face / vertex / texture vertex / normal vertex”. Data fields are separated by the back slash character. Fields that are empty are represented by null characters, as shown in the code block below (Table 1.2). In this example, only geometry vertices were present.

Table 1.2 – OBJ face geometry definition

f 1// 2// 3// f 2// 4// 3// ...	
---------------------------------------	--

All test case files showed face geometry is ordered by adjacent (paired) faces, such that every two faces describe a rectangle, as shown in Table 1.2. Therefore, by processing two faces at a time (i.e., assumed rectangle) and calculating texture vertices using the same planar projection, each rectangle regardless of size would have the texture applied to it. This method resulted in texture stretch in some locations and normal appearance in other locations, as expected by disregarding geometric scale.

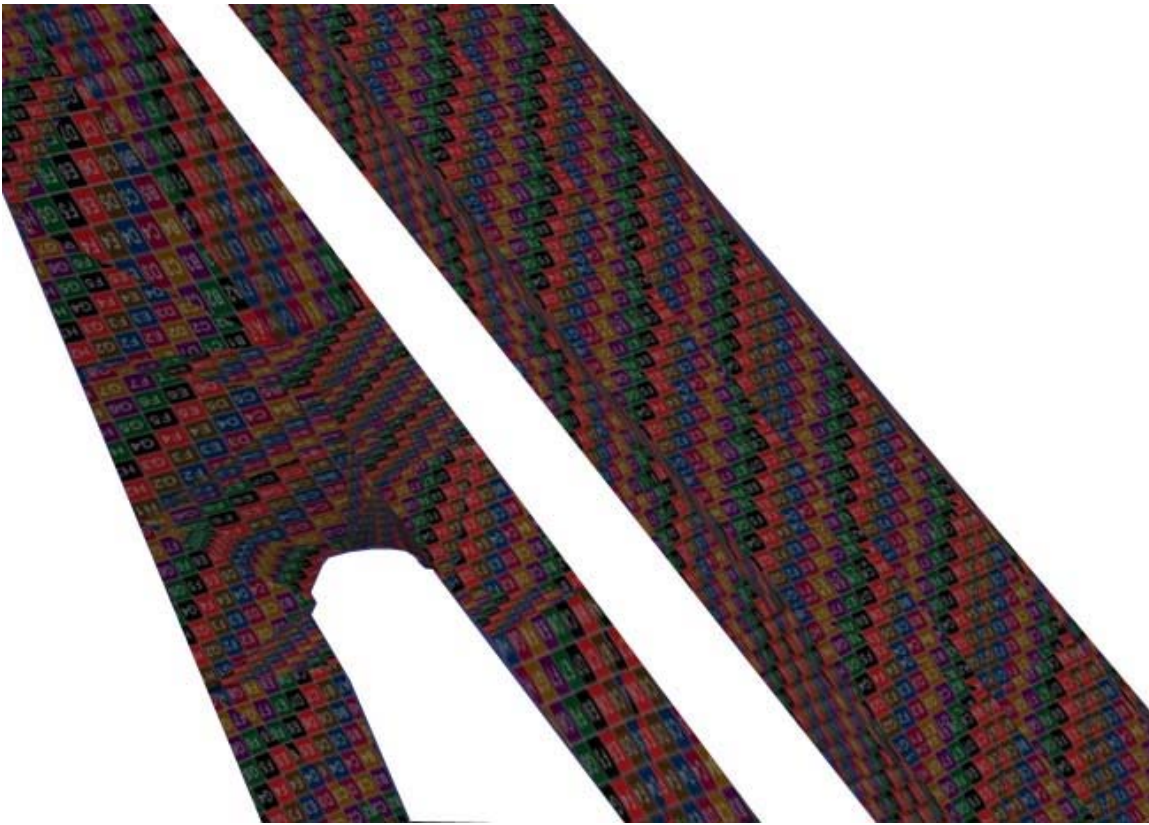


Figure 1.20 – Texture mapping face pairs

The paired face approach resulted in mapping that did not align correctly to geometry edges and contained significant stretch/squash; larger faces used large mappings and smaller faces used small mappings. This was most evident in the curve regions of Figure 1.20. In order to ensure a consistent texture application, it is necessary to map texture using global model coordinates. This method would prevent texture scale problems by using a common mapping scale for all polygons.

Using global model coordinates, the input model was processed by looping through all model vertices and determining the overall average width and length for each face within the model based on the vertex coordinates. Using an average width and length ensured a uniform texture mapping; however, the edge conditions were not processed using this method. In addition, the assumption of paired faces presented an

unrealistic condition. It seemed reasonable to assume that non-paired face conditions would break the mapping algorithm. This became an overriding concern, and the algorithm was extended to use groups of faces instead of pairs. Each group had an average width and length, but there was no uniform scale applied across the entire model.



Figure 1.21 – Texture mapping face groups

Figure 1.21 shows a model with 18 faces and multiple groups. Each group contained three faces. Although the results appeared closer to the desired goal, some discontinuities existed between groups. These discontinuities were evident as breaks in the feature flow. Proper flow would show as consistent diagonal features – instead, diagonal features showed clear discontinuities.

Addressing this issue required calculating texture UVs using a universal minimum X and Y instead of the local minimum calculated from each group of faces. As shown in Figure 1.22, this method did resolve texture discontinuities between groups as evident by the texture features flowing across the geometry without any mismatches. This method ensured both continuity across model geometry and a consistent texture scale.



Figure 1.22 – Texture mapping universal min X, Y

At this point the algorithm was tested against a larger model, where it became evident the mapping was not successful on curved geometry. Processing the larger file, it also became evident that using an average width and length by itself created proper mapping for texture repetition without regard for groups. Reverting to calculating individual texture vertices for each face produced the same results as the previous step. However, this method still lacked a way to conform to the geometry edges in a curved model. Adjustments to the algorithm showed that curves could be supported in a limited fashion by introducing rotation into the mapping, but this introduced problems in formerly working regions.

1.5 Texture Mapping Algorithm Solution

On reviewing progress to date, it became clear the missing ingredient(s) involved addressing geometry edge conditions while applying texture at a uniform scale. The method chosen to address edge conditions was the use of a transformation applied to each model face, orienting each face to a UV baseline axis while maintaining the geometric relationships present in the model, similar to unwrapping a parcel. The test case chosen was a slightly curved ribbon with a small elevation change over the length of the model.

The baseline UV axis chosen was determined by the geometry. Cases where width was greater than length used the U axis. Cases where length was greater than width used the V axis. Each face was translated such that the minimum vertex was positioned on the texture axis. Width and length (UV) were calculated by using the geometric delta between minimum and maximum X and Y as the texture delta. This method satisfied proportional texture mapping while supporting edge following, with results comparable to manually textured geometry.

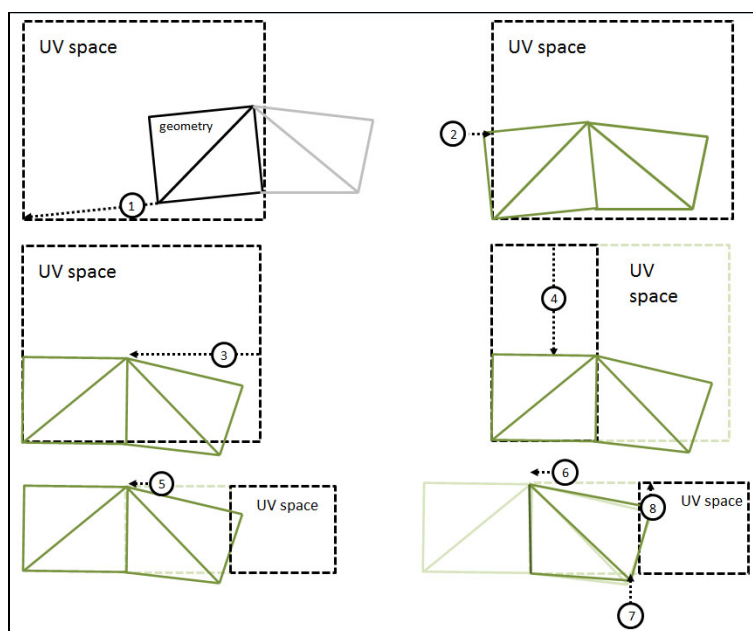


Figure 1.23 – Texture mapping algorithm

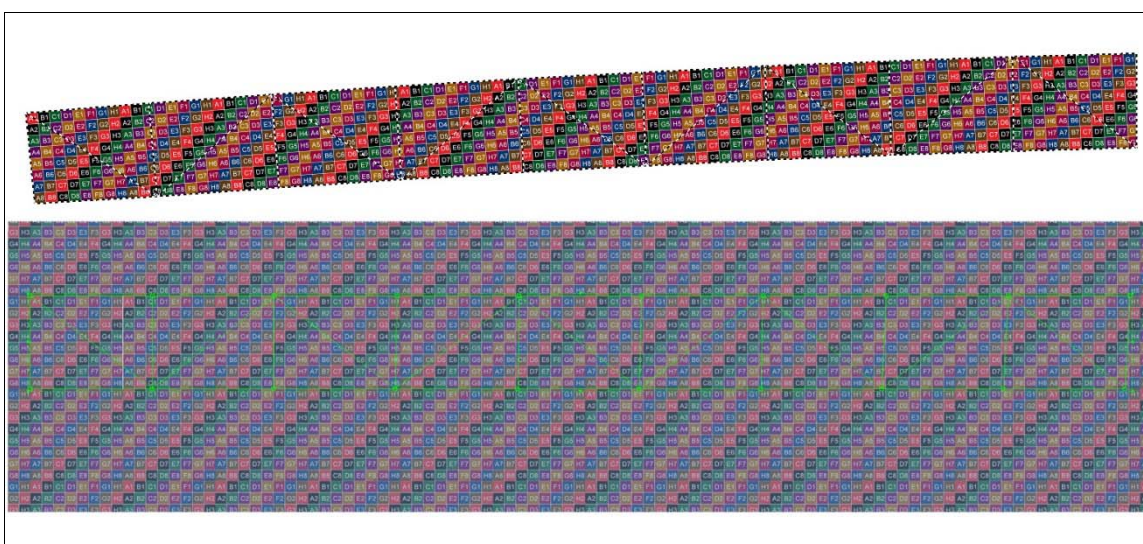


Figure 1.24 – Texture mapping algorithm results

1.6 Future Work

The texture mapping algorithm processes single- and multi-span ribbon geometries, but its current form is capable of processing only simple ribbon cases such as single roadway ribbons. Future enhancements should include geometry input

segmentation and elevation discrimination, as well as provide additional intelligence for the tool to discriminate more complex surfaces, perhaps based on polygon orientation. This capability would prove useful for models that contain side walls and curbing where the geometry does not lay on a single coordinate plane. Another useful capability would be the ability to process overlapping geometry, such as occurs at overpasses.

2 Tile Model Integrator Tool

The second part of this project involved the creation of a tool to manage integrating third party tile models into the TMT tile model library. Prior to this tool, all integration tasks were manual and required specialized knowledge of the 3D model and NADS virtual asset library and pipeline. The tool does not completely eliminate these requirements, but it does facilitate model integration into the tile model library through a graphical user interface and guided data entry steps to allow non-experts to successfully import new tile models. Once the imported file has been successfully integrated, it becomes part of the standard library and may be used the same as any native tile model.

2.1 Tile Model Library

The NADS MiniSim™ relies on a collection of 3D models known as ‘tile models’ that represent environments and environment components within the simulation scene environment. A simulation scene environment is created by assembling tiles into a configuration, which is typically designed to accommodate specific research requirements such as simulation drive time (duration) or locale (place). Configurations may be altered to create additional configurations. When a final design has been reached, the configuration is published using a build process to generate files necessary for scenario authoring and simulation.



Figure 2.1 – Roundabout tile model and configuration layout

2.2 Tile Model Library Overview

Central to the model library is the concept of extensibility and re-use. There is no real limit to the number and type of models that may be imported into the library; as project requirements grow, models are constructed using modelling standards and conventions for tile models and then imported into the library. Once a model has been integrated into the library, it can be re-used as needed.

Tile models are not just 3D model geometry and textures – they are composite objects that include various meta-data (objects and attributes). In addition to model-specific meta-data, there are also attribute and configuration files that are global across the entire library of models. All model-specific and library configuration files must be managed when importing new models into the library. Currently, importing new models requires making edits to library configuration files, adding the tile model and associated data files into the library file system, and ensuring all required files are present. There is a library constraint that all models must be unique; in a manual process, this constraint is highly dependent on the diligence of the person integrating new models into the library.

The process of integrating new models is second nature for persons having daily contact with the system. However, ensuring that novice simulator users are able to integrate new models requires some automation and error checking in the workflow. This functionality has been built into the tile model integrator tool.

2.3 Content to Simulator Overview

This section describes the various stages to transfer source data into a simulation. The process of integrating models follows the schematic shown in Figure 2.2. Data resources on the left side are processed, imported, or created by 3D modelling tools and staff. Additional data files are generated or created and then imported into the tile model library. Tile Mosaic Tool configuration files must be updated to include new model resources, and at this point the newly imported model may be used by the

system. The schematic does not show various modelling conventions that apply to the contents of the tile model library.

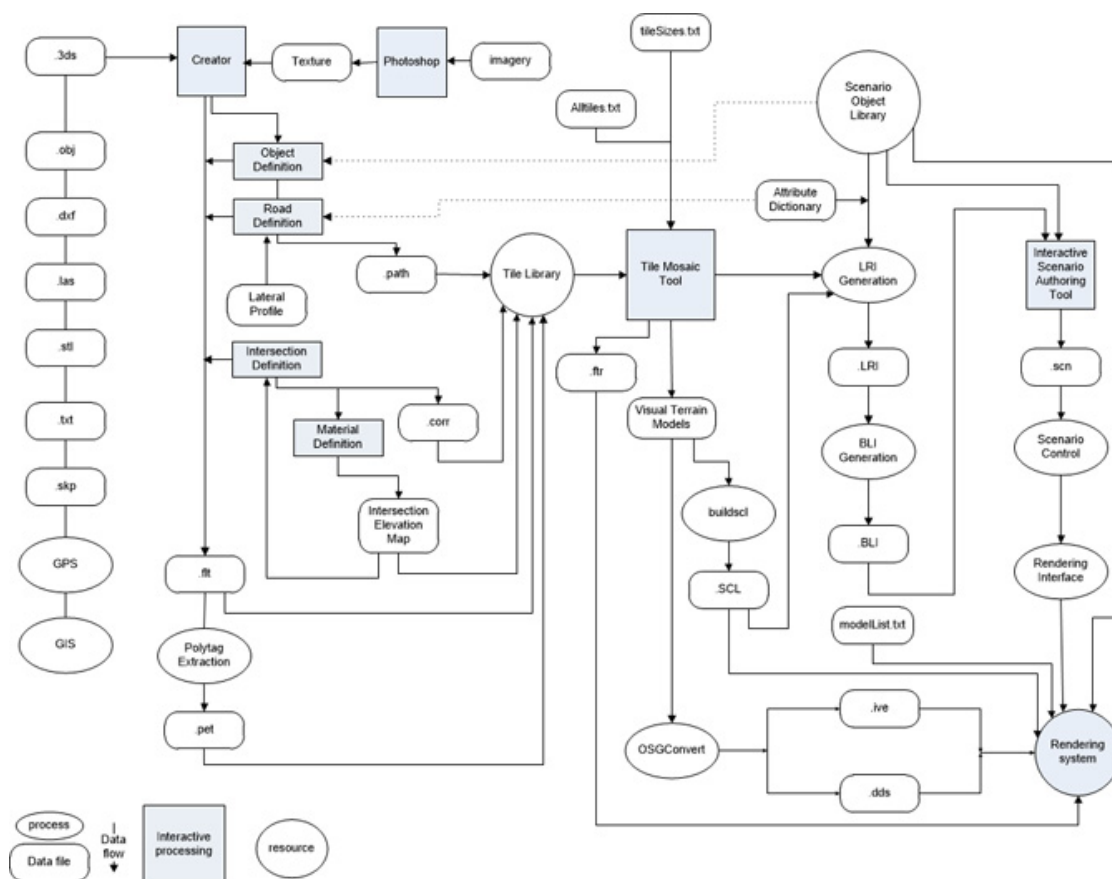


Figure 2.2 – NADS content pipeline: source to simulator

2.4 Tile Mosaic Tool

The Tile Mosaic Tool (TMT) is a graphical user interface to the tile model library for creating simulation environment configurations. The TMT is a simple two-dimensional top-view map editor that contains an inventory of tile models grouped by association or function, as specified in the configuration file “allTiles.txt”. Models are grouped into categories that include a color specification for the category. When the TMT first encounters a new tile, it processes that model by opening the “model.flr” file to process attributes located in the model header. The results of this step are stored in the tile

model folder as “model.txt”, where *model* is the same base file name as the actual model. This processing occurs when the model is new to the library, and any time in the future when the timestamp differs between the “model.flt” and “model.txt” files. If there is no header data present in the “model.flt” and no “model.txt” file is present, the TMT does not recognize the model and it will not be present in the tile model library.

In order to implement these model header attributes, it is necessary to edit the “model.flt” file using a 3D modelling application capable of modifying specific OpenFlight™ hierarchy nodes or some other tool capable of injecting the required attributes into the model file header. Alternatively, it is possible to generate these model attributes manually by creating a “model.txt” file using a text editor. Other tile models can be used as a template to ensure all required fields are present, but this method does not guarantee that correct data is entered; for example, duplicating a similar model and failing to update the record fields for the new model will result in a mismatch between the model attributes and the model.

Model size is a required attribute and is specified in terms of tile model units. A tile unit is 660 feet. Models must be at least 1 unit in width and length (X and Y coordinate planes) and rectilinear. In general, model size accurately reflects the dimension of the model, but in special cases it is desirable to implement non-standard dimensions instead. Model dimensions are also stored in a library configuration file that is used during the environment publication process. Failure to include correct model data results in invalid builds that affect scenario authoring. Although the environment generates files for simulation, some objects in the build will not respond to authored commands.

Currently, importing a new model into the library requires manually

- a) creating the tile model geometry, textures, object definitions and attributes;
- b) editing the tile model library configuration file to include the new model;

- c) editing the tile dimension configuration file to include the new model; and
- d) editing additional tile model library configuration files as needed (e.g., if new road profiles or intersection elevation maps are defined in the new model).

The process of importing a new model into the tile model library presents a significant challenge to MiniSim™ users of any experience level. Managing the tile model library requires a thorough understanding of the MiniSim™ and TMT as well as familiarity with all associated meta-data files (parameter, configuration, object definition, and environment data files).

2.5 Tile Model Integrator Tool Description

The model integrator tool is a simple graphical user interface (GUI) application written in Python 2.7.8 that processes Wavefront OBJ and OpenFlight™ files and guides the user during data entry for importing new models into the tile model library. Currently the tool imports OBJ format model files, but the file format is a function of the “OSGConv.exe” converter engine, which supports a variety of file formats. Subsequent changes to the tool for expanding supported file formats are therefore possible by extending “OSGConv.exe”.

2.6 Integrator Tool Dependencies

The integrator tool has several important dependencies. The current content to simulator workflow requires OpenFlight™ format files for tile model source files, so the Presagis OpenFlight™ API is necessary to automatically modify converted files. “OSGConv.exe” is used as the converter engine, which decouples the requirement to build and maintain multiple file format converters and makes available all 3D file formats currently supported by OSGConv as possible simulator models. Currently supported file formats are shown in Table 2.1. The programming language used for the Tile Model Integrator Tool is Python 2.7.8, and it relies on the tkinter library as well as other modules included in a standard Python install. Tkinter is the standard Python

interface to the Tk GUI toolkit and is included with the standard Windows and Mac OS Python installations ^[1].

Table 2.1 – 3D file formats supported by the converter engine

3D file format	Read	Write
3DC point cloud reader	Y	
3DS Auto Studio reader/writer	Y	Y
AC3D Database reader	Y	
BEX Bentley LandXML extract file reader/writer	Y	Y
BSP file reader	Y	
Design Workshop Database reader	Y	
FBX reader/writer	Y	Y
GEO reader/writer	Y	Y
Lightwave Object reader	Y	
Quake MD2 reader	Y	
Valve/Source Engine MDL reader	Y	
Wavefront OBJ reader	Y	
FLT reader/writer	Y	Y
OSG reader/writer	Y	Y
present3D	Y	
XML reader/writer		
POV reader/writer	Y	Y
RAW raw triangle file reader/writer	Y	Y
STL reader	Y	
DirectX reader	Y	

2.7 Integrator Tool User Interface

The integrator tool is organized by panels into three areas of functionality:

- a) tile category operations – add new categories, change color records;
- b) import model panel – specify model to import into library; and
- c) model attribute data entry panel – data entry for header attributes.

Each panel is available only when the user satisfies the conditions necessary for operation. For example, there is no way to import a model without previously selecting a tile category. Importing a model and then cancelling any required operation prevents access to the model attribute data entry panel.

The integrator tool reads the tile model library configuration file “alltiles.txt”. This file defines the organization of models into categories and contains a color value for each category expressed as an R, G, B triplet of integer numbers. Using the tool, users can add new tile categories, modify existing category color parameters using a graphical color chooser, and import new OBJ models into selected tile categories.

Upon choosing a category, the user interface updates to reflect the current selection. Choosing “import” initiates a series of workflow steps, all of which may be cancelled at any time. When the user satisfies model import requirements, the process of model importing begins. The user must choose a file to import; file choice is filtered by file location, number of associated files, and file type. The next step is defining model header attributes using a graphical user interface. This step is managed through the use of color-coded feedback and reinforced by status messages informing the user about the state of their work.

Upon successful data entry, the user can then choose to save their work, which triggers all associated file management activities. First, the selected model is copied from the source model location into the model library to the proper model folder location based on model category selection. Then this copy is converted to OpenFlight™, the

TMT model source file format. The model data entered by the user (and validated by the tool) is then injected into the converted OpenFlight™ model file header. The library configuration file “alltiles.txt” is updated with the imported model data and the model is integrated into the model dimension configuration file. At this point the user can exit the integrator tool and open the TMT, which now contains the imported model. A brief processing step is performed on the imported model when it is first used in the TMT, and then the user may begin to create a simulation environment using the tile model library that now includes the imported model. At this stage the imported model has been fully integrated into the tile model library and may be used the same as any pre-installed tile model.

Currently the tool does not support removal of models or tile categories. This is due to historic use and re-use of model resources. Allowing removal of a tile can cause projects to fail catastrophically due to missing tiles when they are regenerated. The TMT is able to open configurations that contain missing tile model references. However, saving the configuration with missing tile models creates an invalid layout, which will then always fail to open. There is no way to recover the layout file once this situation occurs. NADS has seen the original library grow to several hundred models over the course of 16 years; it is unlikely that a simulation laboratory would outgrow hard drive storage requirements given low storage costs. Therefore, there is no need to manage the number of library models by removing models or tile categories.

While the tool also does not support renaming tile model categories, there are no requirements for specific categories to be present. Thus, a category can be renamed without incurring any penalties. Because the “alltiles.txt” file is a text file, it is possible to change category names by editing the file using a text editor. Furthermore, the integrator tool does not permit duplicate file names in the tile model library configuration file

“alltiles.txt” or in the tile size list file “tilesizes.txt”. Encountering a duplicate tile model name will terminate the current process and return the user to Panel 1 of the tool.

2.8 System Configuration Requirements

The Integrator tool requires a standard MiniSim™ installation. This typically includes a MiniSim™ installed root folder, an OSG folder, and a TMT installation folder, including the “TMT utils” folder. Additional requirements include Python 2.7.8 and OpenFlight™ API python bindings.

On startup, the integrator tool performs system level configuration diagnostics. If the system is configured incorrectly, the tool will present an error message to the user and exit. If any of the required system components are not detected during the initialization phase, the tool will notify the user and exit. The tool queries system environment variables and the “nadsconfig_system.bat” script file to identify TMT, MiniSim™, and OSG file system locations.

2.9 Python Requirements

The tile model integrator tool is written in Python 2.7.8, which is not compatible with Python 3.0 or above. There are no external downloads required for the tool, as all modules used are included as part of the standard Python 2.7.8 installation with the exception of the OpenFlight™ API files.

2.9.1 *Python Modules*

The tool requires the following modules: os, system, stat, re, shutil, subprocess, tkinter, tkFileDialog, tkMessageBox, tkCommonDialog, tkSimpleDialog, tkColorChooser, and the external Presagis module mgapi.

2.9.2 *OpenFlight™ API Requirements*

The required API files are bundled as distributable library files that are the intellectual property and copyright of Presagis. These include

- a) fltdata.dll,

- b) mgapilib.py,
- c) mgapilib.pyc, and
- d) _mgapilib.pyd.

These files are included with the tool software courtesy of a licensing agreement with Presagis and may not be redistributed without permission.

2.10 Integrator Tool User Interface

This section includes screen shots of the tool and describes various user interface features and operation. Figure 2.3 shows the default configuration following successful initialization and after selecting one of the tile model categories. The status area prompts the user to select a category.

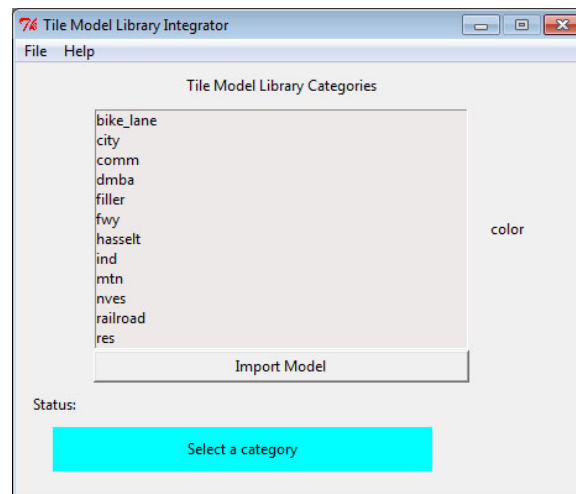


Figure 2.3 – Category operations panel after successful initialization

During category operations, the user is able to select from a list of tile model categories presented in list form. As each category is selected, a color swatch updates to reflect the color record for each selection. The user is able to change the color by clicking on the swatch and choosing a new color from the color picker.

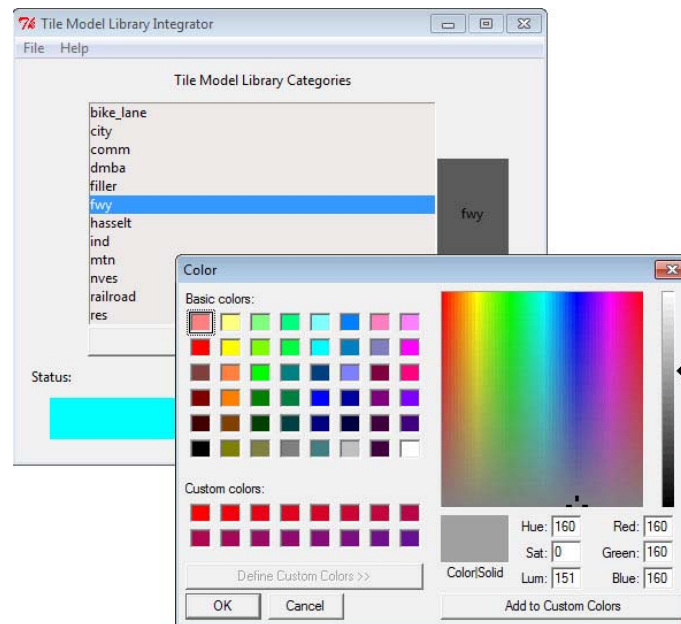


Figure 2.4 – Color picker

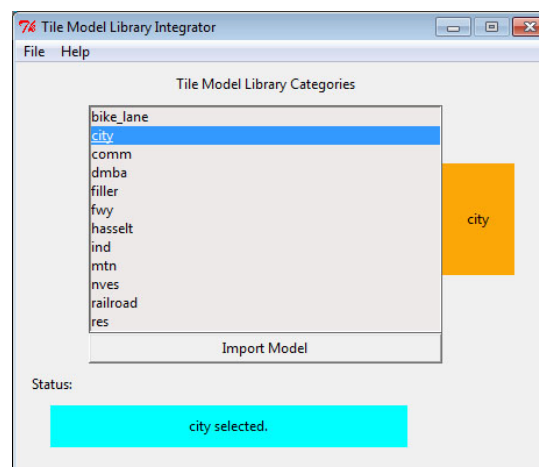


Figure 2.5 – User feedback for selected category

For contrast, the selected category is shown below in native text form (with intermediate missing files indicated by ellipses):

city: 250, 167, 7

2In_city_01

2In_city_01_day

2ln_city_02

...

After selection of a category, the “file >> insert” menu is activated and the user may insert a new category before or after the current selection. A dialog prompts the user to enter a name for the new category and also prompts for a color record associated with the tile category. Canceling either of these operations will terminate the new category process.

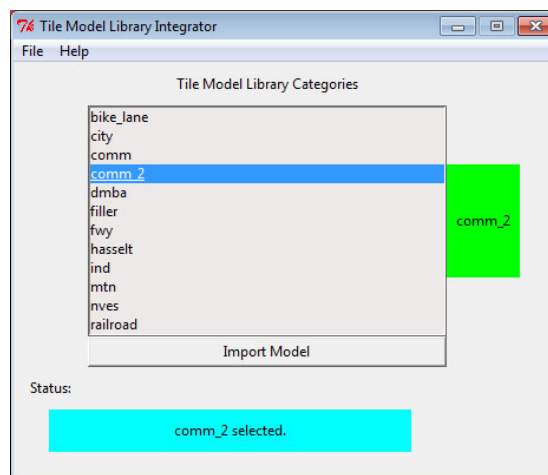


Figure 2.6 – New comm_2 category inserted

Selecting a tile category activates the “Import Model” button. The user is then presented with the “Import Model” panel shown below.

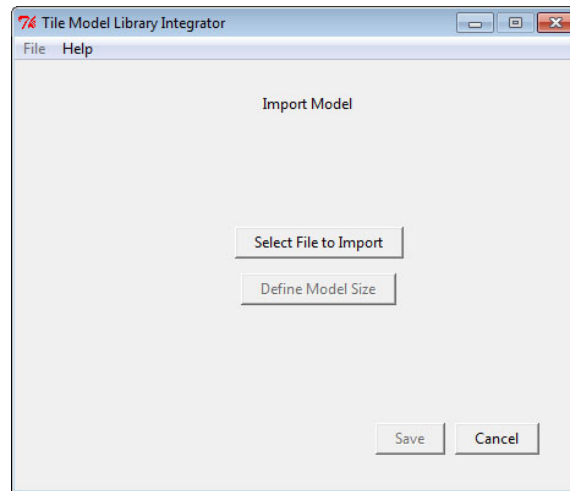


Figure 2.7 – Import model panel

After selecting a file for import, the “Define Model Size” button is activated, and the user is presented with a data entry panel for model header attributes as shown below.

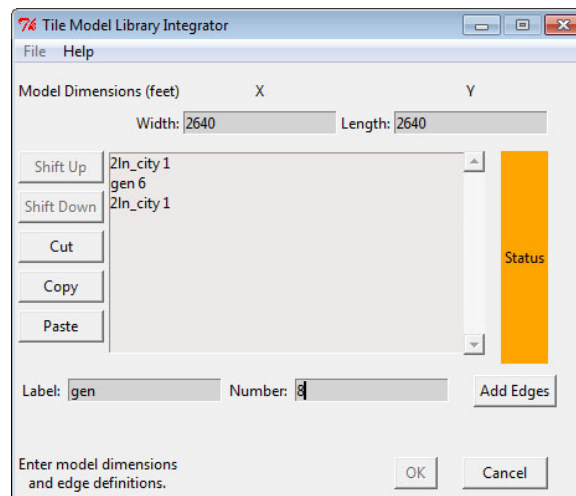


Figure 2.8 – Data entry in progress

The data entry panel is the most complicated tool element. The interface guides the user to enter data in the format needed for correct and accurate model dimension

data. The tile dimension data is used by the TMT to determine tile edge connectivity as environments are created. Tile edges are defined for the model beginning with the lower left corner and travelling counter-clockwise around the model perimeter. The number of tile edges must satisfy the dimensions entered. Edges are defined in terms of tile units.

As the user enters data, it is evaluated to ensure the required number of elements are met or exceeded and the status indicator updates to provide cues to the user as they complete the required records. As data is entered, the user can perform various list operations as shown by the buttons adjacent to the list. This interface provides flexibility in how the user enters the data; data can be entered using the data entry fields or it can be duplicated (through copying and pasting), and, once entered, it can be adjusted using the shift buttons. When the required numbers of edges match the dimensions entered, the status indicators update to reflect that the required number of edges has been achieved. It is left to the user to ensure the edge list reflects the correct order for the model.

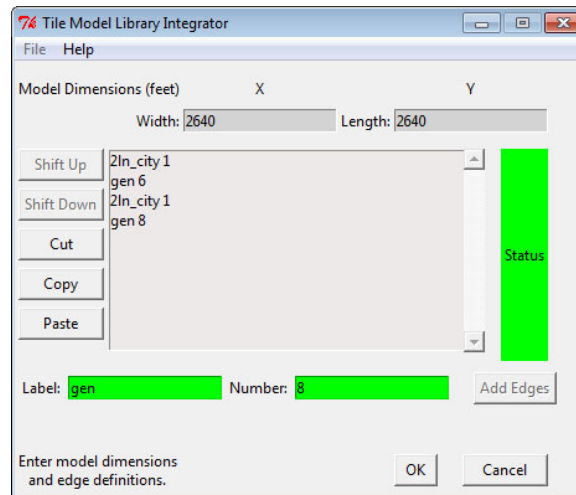


Figure 2.9 – Data entry completed

The “Model Dimensions” fields are in feet because it is most likely that the person who created the model will have this information available or can obtain it. It is not preferable to rely on the model for these dimensions because there will be cases where it is desirable to not use actual dimensions; for example, if the model is designed to enclose another model or configuration, the specified dimensions can purposely be significantly smaller than the actual model dimensions.

After successful edge definition, the user is presented again with the “Import Model” panel, where they can choose to cancel or save. “Cancel” returns the user to the “Tile Category” panel. “Save” initiates the remaining model import processing steps, which occur automatically:

- a) copying the model specified into the tile model library folder structure,
- b) converting the model to OpenFlight™(model.flt),
- c) inserting header attributes into “model.flt”,
- d) editing the tile model library configuration file to include the new model and saving the modified file to disk, and
- e) adding the new model dimensions to the tile dimension configuration file “tileSizes.txt”.

2.11 Tile Model

This section details what a tile is and is not. A tile model is not simply a textured 3D model, no matter how simple or complex the model happens to be. A tile model is a collection of files that includes geometry, texture, meta-data, attributes, and associated data that is located in various files located in specific folders within the tile model library file system.

2.11.1 *Tile Model Files*

This section describes a number of tile model files.

- model.flt - by convention, no spaces in the file name;

- model.icn – used by the TMT;
- model.pet – a text file containing object definitions and attributes for all virtual elements within the model, it includes definitions of roads, roadway data files, intersections, intersection connectivity, objects, and locations in local model coordinates for the preceding elements;
- PATH – files that contain road centerline data in local model coordinates, in which points must be ordered sequentially. This file type is only required if the tile model contains a road that must be drivable;
- CORR – files that contain intersection lane connection data in local model coordinates, in which points must be ordered sequentially. This file type is only required if the tile model contains an intersection;
- model.txt – this file can be used to define tile model header attributes. If not present, the TMT will generate this file using model header data within the “model.flt” file if that file is present. If this file is missing and no header data is in the model, TMT will not recognize the model.

2.12 Associated Model File Set

These associated model files are part of the tile model library and may be referenced by the TMT and tile model:

- allTiles.txt – this text file contains a list of the Tile model library tiles and categories (configuration file for the TMT);
- CD1 – a project component file for the TMT, it is a text file containing a list of the tile model library tiles used in the MOS;
- CD2 – a project component file for the TMT, it is a text file that defines the connective edges between tiles;
- DDS – these are compressed texture files for use on the MiniSim™ and are generated during the build process;

- FLT – necessary for MiniSim™ simulator operation in native FLT form or optimized binary IVE format, these are binary OpenFlight™ files that contain geometry and references to image texture files;
- FTR – necessary for MiniSim™ simulator operation, this is a TMT project component file that contains all the tile references necessary for a terrain configuration including X, Y, Z offset, rotation, and tile category type;
- Intersection.map – a text file that contains terrain data and specifications for unique elevation maps applied inside intersections;
- IVE – these are binary files converted from FLT and optimized for MiniSim™ use during the build process;
- LatProfileList.lat – this text file contains all the lateral specifications for every road type and includes a cross-section profile and a material code index;
- MOS – this binary file is a TMT project file, also referred to as a world or configuration file;
- RGB, RGBA, DDS – necessary for MiniSim™ simulator operation, these are binary image texture files. The RGB and RGBA files may also have associated “.attr” files produced from the modelling environment tools. These files will already be present in the MiniSim™ configuration, but additional (new or modified) files must be copied into the proper location;
- SUP – this binary file is used by the TMT™; and
- SurfaceMaterialSpecifications.xlsx – an Excel™ document that contains surface material codes for road surfaces.

2.12.1 Import Model Requirements

This section describes assumptions regarding the imported model using the integrator tool. Despite the dual nature of this project, the integrator tool does not apply

texture to models during the import process. The texture tool is currently a standalone command line tool.

2.12.2 *Modelling Conventions*

The following modelling conventions exist:

- Models must be oriented on the X-Y plane in a coordinate system where Z increases upward, X increases to the right, and Y increases forward.
- Models must be located with the lower left corner at a local origin of 0,0.
- Models units = feet. Metric objects currently must be converted to their imperial equivalents.
- Models must use dimensions that are normalized to 'tile units'. One tile unit = 660 feet.
- Models must be rectilinear or square in standard tile models.
- Holes are permitted in special purpose models only.
- Models cannot overlap in standard tile models.
- Terrain baseline is 0.5 Z elevation.
- Terrain at tile outer boundary edges is baseline elevation. In some cases, terrain is articulated at the tile boundary (e.g., to introduce ditches to rural and freeway tiles).

2.13 Future Work

The tile model integrator tool provides a graphical user interface and error checking for a complex manual task – integrating new models into the tile model library. In the short term, it is recommended that a survey be conducted to establish what file formats are the most desirable imports, and then modify the converter to support these formats to provide immediate benefits to users. Longer term, this tool should be leveraged as the graphical front end to an SQL type database content management ecosystem that incorporates all the present individual files and folders. This would have

far ranging implications for all levels of the NADS simulator architecture and eliminate redundant information and, more importantly, create a more robust content generation pipeline.

References

1. Tkinter Wikipedia, accessed 02.18.2016.

<https://en.wikipedia.org/wiki/Tkinter>

Appendix A: Sample OBJ File

This section includes a simple example OBJ file, consisting of a textured plane polygon.

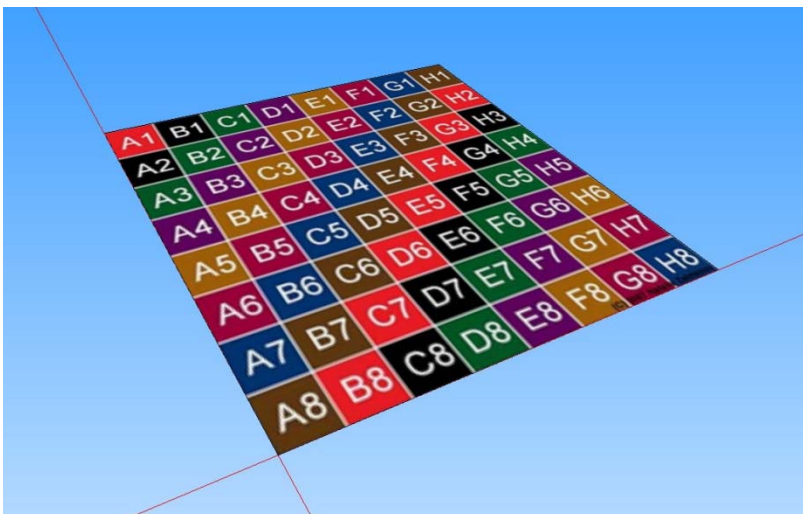


Figure A.1 Sample OBJ file

Sample.obj

```
mtllib plane_100f_1.obj.mtlv 0 0 0
```

```
v 100 0 0
```

```
v 100 100 0
```

```
v 0 100 0
```

```
vn 0 0 1
```

```
usemtl 0vt 0 0
```

```
vt 1 0
```

```
vt 1 1
```

```
vt 0 1
```

```
f 1/1/1 2/2/1 3/3/1 4/4/1
```

#created by NADS_ObjectGo Fri Jun 26 16:08:52 2015

Sample.obj.mtl

newmtl 0

map_Kd E:/Nads/ProjectData/TileTx/UV_checker_256.rgb

Ka 0.722 0.722 0.729

Kd 0.918 0.918 0.918

Ks 0.137 0.137 0.137

Tr 1

d 1

Ns 10

illum 2

Appendix B: OBJ File Format Specification

https://en.wikipedia.org/wiki/Wavefront_.obj_file

<http://www.martinreddy.net/gfx/3d/OBJ.spec>

<http://www.fileformat.info/format/wavefrontobj/egff.htm>