Intersection State Visualization for Realtime Simulations

INTERSECTION STATE VISUALIZATION FOR REALTIME SIMULATIONS

BY

JUSTIN ROTH, B.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2012) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Intersection State Visualization for Realtime Simulations
AUTHOR:	Justin Roth B.Sc., (Software Engineering) McMaster, Hamilton, Ontario
SUPERVISOR:	Dr. Martin v. Mohrenschildt
NUMBER OF PAGES:	xi, 112

To my patient and supportive parents

Abstract

Driving simulators have existed since the beginning of the 20^{th} century. From its roots, it has been a technology used primarily to train drivers, test and prototype new technology, and improve the safety of automobile users. As technology has progressed, so has the quality of the driving simulation, and along side it, the complexity of experiments performed. The McMaster motion simulation system combines the latest software with state of the art psychology techniques, to analyze the driving experience in new and unique ways. To accommodate the wide range of plausible experiments, a robust software system was developed that allows for custom driving scenarios. The software system is comprised of several sub-components including content generation, scenario management, visualization and artificial intelligence. This thesis details the development of a traffic light system and its incorporation into the existing simulation system. A variety of challenges were encountered including real-time constraints, adapting flight software to driving simulation, inter-system communication, and interoperability of multiple APIs. A secondary objective was to document, this thesis records the methodology used to overcome these challenges in an attempt to facilitate future work in this field.

Acknowledgements

Acknowledgments go here.

Notation and abbreviations

AI: Artificial intelligence

API: Application programming interface

DFS: Depth first search

DOF: Degree of freedom

FOV: Field of View

FSM: Finite State Machine

OO: Object oriented

VSG: Vega Scene Graph

Contents

List of Figures

Chapter 1

Introduction

A driving simulator is a machine that simulates a driving environment or conditions for the purpose of training or experimentation [6]. Starting at the beginning of the 20^{th} century, simulators have been used as a tool to understand driver behavior and reduce casualties on the roads. This was particularly true in the 60s where the wide spread adoption of the automobile resulted in a drastic increase in the number of driving related causalities [12, p.1]. Since then, the number of casualties related to driving incidents has decreased, but there is still a need for driving simulation. Cell phones are a relatively new technology that has had a drastic impact on how people drive a vehicle. A study by VirginiaTech Transportation Institute states that "secondary tasks account for 23 percent of all crashes and near-crashes". Clearly, there is still a need for driving simulators and more specifically, driving simulations capable of studying the impact of current technology on people use of automobiles. McMaster's motion simulator is one such tool that attempts to tackle these issues using the latest in simulation software and psychology techniques.

McMaster's motion simulator is designed to perform unique experiments that

combine stimulus presentation and state of the art psychology techniques. One series of experiments focuses on driving simulation and more specifically, how the brain of candidates reacts while performing driving tasks. The objective of this simulator is to perform a wide range of experiments and to minimize setup time. With this in mind, the software side of the system was developed in order to create a realistic driving simulation with the capability to generate new driving environments very rapidly. One aspect of this automation process is the traffic light system. This system visually signals to the user what actions they may perform when approaching an intersection. At the same time, it effects how the AI behaves under the same circumstances. The goal of this thesis is to document the development of a traffic light visualization system that integrates with the rest of the simulation system.

The development of the traffic light visualization system has a unique set of challenges. The system is comprised of 13 different computers, each responsible for a particular aspect of the simulation. The motion platform, the system responsible for generation motion, is a real time system. Interacting with a real time system involves hard deadlines and a variety of constraints. The system described in this thesis must integrate within the exiting framework while not interfering with its performance. Another interesting factor is the software used. Presagis has created a suite of products designed for "out of the box" motion simulation development. Most simulation systems currently in use world wide are flight simulators. For this reason much of the software's design, optimization and architecture were originally intended for use in flight scenarios. This thesis adapts a flight based software package for use in a automobile simulation.

Development of the intersection visualization system was not a traditional development process. The designer, as well as the co-workers on the project, were only partially familiar with the variety of tools available in the Presagis software package. Ideally, a certain amount of time could be committed to fully understanding and learning each system as well as what task they are capable of performing. The Presagis software tools are powerful but come with a step learning curve. Understanding them fully would be very time consuming. To facilitate the design process we focused on using software packages with the most extensive documentation. Then we choose a testing intensive approach. A certain amount of uncertainty was present in the design process and so it was important to rapidly prototype in order to expose any flaws in design. With these concepts in mind, software development began.

The thesis has a standard layout. First, a background section introduces the reader to pertinent information. The background consists of a specification of the OpenFlight database format, a description of rendering and 3D graphics, an introduction to the software used, and a history of driving simulation. Second, the simulation system is introduced. This chapter describes the McMaster motion simulation system as well as the design constraints the system architecture imposes. Constraints are then used to draft an initial system design. The next chapter details the intersection state visualization system's development process. The first chapter discussed the database approach to development while the later details the XML approach. Each chapter contains several proposed models. These models were proposed solutions to the intersection state visualization system. The earlier implementations were not successful but lead to some interesting refinements. The model description consists of

phases: design, implementation, and evaluation. The design phase consists of outlining the program's algorithms and detailing potential implementation strategies. The implementation phase contains the specifics of the implementation as well as issues encountered. The evaluation phase analysis the causes of implementation issues, and makes suggestions for future models. The thesis concludes with a discussion on the lessons learned during development. The focus is on the effectiveness of the program as a whole as well as the potential pitfalls in adapting Presagis's software to a driving simulation.

Chapter 2

Background

2.1 OpenFlight Database

"OpenFlight is the leading visual database standard in the world and has become the defacto standard format in the visual simulation industry." [10]. OpenFlight is a database system created by MultiGen. It is designed "to support both simple and relatively sophisticated realtime software applications" [9, p.2]. Because of its success as realtime database system, Multigen and Openflight, was purchased by CAE and incorporated in Presagis. OpenFlight is the storage medium for rendered databases in Presagis' suite of realtime development systems, Terra Vista, Creator and VegaPrime. In these system, OpenFlight facilitates "variable levels of detail, instancing (both within a file and to external files), replication, animation sequences, boundingboxes for real time culling, shadows, advanced scene lighting features, lights and light strings, transparency, texture mapping, and several other features" [9, p.2]. Due to its effectiveness, Openflight's use is not limited to Presagis software. It is used in a variety of open source realtime system including Open Scene Graph, a 3D

graphics tool kit, and Flight Gear a flight simulator. OpenFlight is the standard of the simulation industry. Its modification and use, was essential in this thesis's traffic indicator system and for this reason, will be examined in greater detail.

2.1.1 Hierarchical Structure

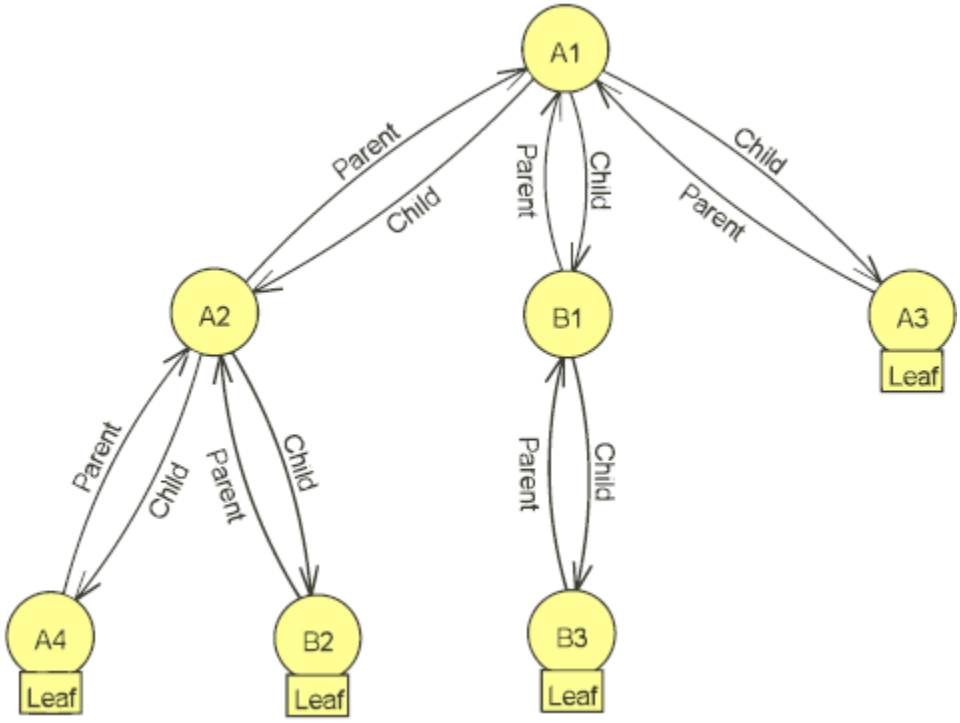


Figure 2.1: A tree illustration the terminology used in trees. The letters A and B describe the type of the nodes while the number describes the n^{th} instance of the respective node type. NodeType = {A, B}, RootNode = A1 and LeafNode = {A4, B2, B3, A3}

OpenFlight databases are optimized for real time efficiency. The database uses a tree structure for global organization and several logical groupings for local organization. This structure increases the speed of searches while groupings customize performance based on use.

A tree structure consists of nodes and connections between them. Each node is connected to at least one other node via a child or parent relation. A parent relation is a connection to a node above it in a hierarchy. Each node may have only on parent. A child relation is a connection between a node below it in the hierarchy. Unlike the parent relation, a node may have any number of connection and therefore may have n children. The arity of the relation is 2, meaning that one nodes connects to another node and only one other node. There is the additional restriction that the connection cannot be with itself.

Several terms are used when describing a tree. The top most node is called the root node and nodes without children, leaf nodes. A node may also have any number of siblings which are nodes that share the same parent. Additionally, node types have restrictions on the types of their parents as well as the types of their children. Each node type has a set that defines which nodes may be its parents and a separate set which describes which nodes may be its children.For example, an object node cannot be the parent of another object node.

The distribution of node types differ between the top and bottom levels of the tree. The top levels use node types to group, organize and prioritize nodes. The nodes situated at the bottom of the hierarchy are used for rendering specific tasks. The leaf nodes are made exclusively of vertexes that describe faces and meshes.

Each node has a type "with attributes specific to its function in the database" [1, p.13]. The attributes store information related to their function and will be discussed in greater detail in section 2.1.6 Node Types. The connection between parent and child have restrictions. These restrictions may be changed using node extension but are generally kept in place so that assumptions may be made. For instance, face

nodes can only have vertex nodes as children. When a face node is read, it can be implied that at least 3 vertices nodes will follow since any face must be made of three points connected by lines.

2.1.2 Low Level Implementation

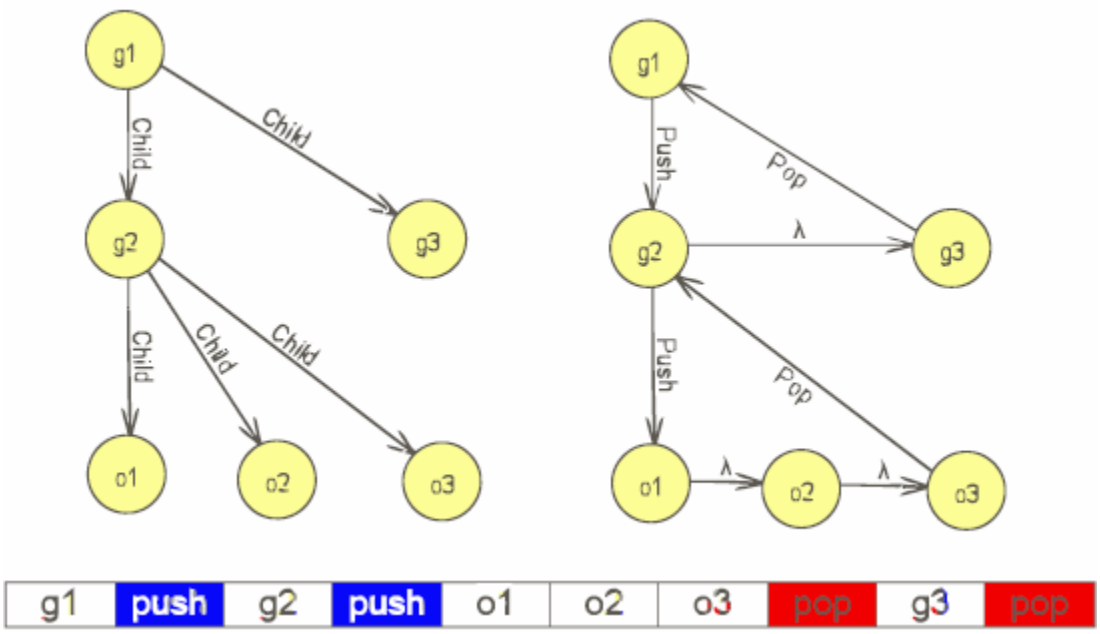


Figure 2.2: Comparison of Database and Memory Structures in OpenFlight. Left (tree model), right (stack model), and bottom (memory implementation.)

OpenFlight file format is optimized for performance. Memory is continuous and linear in structure and therefore, some conversions must be made to remove the branching structure of the OpenFlight hierarchical tree. OpenFlight transforms the tree structure into a linear structure using a depth first search (DFS). When moving down the tree from parent to first child node, a push operation is performed and respectively when moving up the tree a pop operation. Sibling nodes are then listed from left to right when the traversal reaches maximum depth. This structure has a

few implications. One is that there is priority processing of nodes from left to right based on the ordering on that level. The other, is that sibling nodes are not listed continuously in memory unless the nodes are leaf nodes. This behavior is not optimal in all situations and so open flight contains several control nodes (LOD, switch, group) to customize the traversal.

All nodes in an OpenFlight database have a common structure. The first two bytes encode an opcode. An opcode is a unique integer $\mathbb{I}$ that identifies the node type. The second two bytes describe the length of the node information. The node specific data is then encoded with the next bytes specified by the length field. The encoding is specified in the OpenFlight User Manual [3].

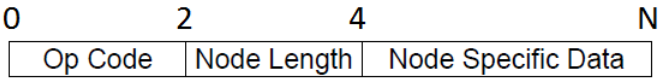


Figure 2.3: Layout of OpenFlight Nodes in Memory. 2 byte opcode describes the node type, 2 byte node length, and node specific data length is N-4. Length includes the opdcode and node length fields and its value is N.

Following the node there are ancillary records. Ancillary encode additional information that is useful for most nodes types yet not necessary enough to warrant its inclusion in all nodes basic encoding. Information includes, long IDs, node names, comments, and matrix transformations. The specifics of these node types will be discussed in 2.1.6 Node Types section.

2.1.3 World Coordinate System and Positioning

OpenFlight was designed for use in flight simulations. The positioning and coordinate system are designed to handle many different units of measure while still ensuring efficient database organization. Each OpenFlight database contains a header file that

specifies how distance is measured. One field specifies the units(meters, kilometers, feet, inches, miles) and another the projection type. The projection type is a mathematical model of the earth's surface. The models differ on their accuracy relative to the database's global position. That is, the models are more accurate at predicting the surface of the earth based on a specific longitude and latitude. The global projections predicts the world's surface and therefore require units of measure that are very large relative to any object that might be place within a database. To avoid forcing the same units as the global projection OpenFlight uses relative positioning. The database defines a world coordinate as the origin. All vertexes are then specified relative to this database position. When world coordinates are needed, then the projection type and database position are retrieved from the header node.

Additionally, global transformation may be applied to database sections using transformation matrices. Each node may have an additional transformation matrix inserted. A transformation matrix is a matrix that when multiplied by a position vector results in a new position. Given a node's transformation T matrix and its position p, the node's new position p' is computed using as $p = T * p'$. For more information on transformation matrices refer to section 2.2.5. In OpenFlight, a transformation matrix applied to a node also applies to the entire subtree. For this reason, position is computed by traversing the entire scene graph.

As described in section 2.1.1 Hierarchical Structure, rendered nodes are situated at the lower levels of the databases. Additionally, node position is effected by its ancestors' transformation matrices. To find a rendered nodes' world position a sequence of transform matrices must be accumulated from the database's root node to the rendered nodes. The accumulated sequence of matrices is called the matrix stack. When

a rendered node's position is required, the matrices on the stack are multiplied and applied to the node's position. Typically a single node's position is not computed but rather all the rendered node's positions. In fact, this process is performed each time a frame is drawn. The list of drawn nodes is known from the culling phase, see section 2.2.3. The scene is traversed, in a depth first manner, along the active branches. As the tree is descended, from root to child, each transformation matrix encountered is pushed on the matrix stack. Using similar criteria, transformations are pop during the ascension. When a vertex node is found and that node must be drawn, then the stack is computed. The computation involves multiplying the product of the matrices on the stack by the position of the vertex node. This process is performed for every vertex belonging to the draw list. In this manner, the location of all drawn nodes are computed using a single scene traversal.

2.1.4 Pallets

Index	Texture
1	Grass.rgb
2	Road.rgb
4	Sidewalk.rgb

Figure 2.4: A sample texture pallet that maps index value to corresponding textures.

OpenFlight uses a pallet system for storage and access of common information such as colors, textures, materials and vertexes. In OpenFlight, a palette can be viewed as a mapping P between an integer and an output type b where $P(a) = b$. Consider the texture pallet for a face node in figure 2.4. The pallet consists of input/output pairs of integer and textures respectively. The face node stores the

integer representation of its texture rather than a local copy. When rendering the face node, the integer is mapped, using the texture palette, to a corresponding texture that is then drawn to the face. In this example, an index value of 1 would result in Grass being drawn on the face node. The pallet system greatly reduces the cost of storing local information but requires some additional resources to store and manage the pallet. This compromise is desirable in most 3d objects systems. The number of faces is usually several scales of magnitude larger than the number of textures and so the cost of managing the pallet system is relatively negligible. It should be noted that the pallet system is not unique to OpenFlight but commonly used in most 3D graphic models such as blender(.blend), Legacy 3D Studio(.3ds) and COLLADA (.dae)

2.1.5 Instancing

Instancing is defined as "the ability to define all or part of a database once, then reference it one or more time while apply various transformations" [3, p.14]. Instancing is best explained with an example. Consider an automobile with four wheels. The wheels are identical in terms of geometry yet differ in their positioning. An efficient database implementation would define the wheel once, use instancing to replicate the node, then apply a transformation to reposition the wheels. This automobile scenario has its advantages and disadvantages when implemented using instancing. Its advantages are that the database size is reduced by eliminating redundant geometry definition for each tire. As well, if the user decides to change the look and feel of the tire then the user can alter the geometry of one node and have it applied to others automatically. The disadvantage is adaptability. If the user wanted to implement a flat tire then altering a single tire would result in all tires being flat. Instancing is a

tool to reduce database size and increase speed at the cost of reduced adaptability.

2.1.6 Node Types

Push/Pop

This section discusses the push and pop node types. As discussed in the hierarchical structure section 2.1.1, push and pops are used to move up and down the tree structure of an OpenFlight database. There exists several different type of push/pop operations. There are generic, surface, extension, and attribute push/pops.

Generic push pops are used to describe the parent/child relationship between nodes in the OpenFlight hierarchal database.

Subface push/pop are used when distinguishing between two overlaping coplanar faces. In OpenFlight, surface refers to the parent/child relationship and NOT to geometrical relationship where one face lines entirely within the other. "Creator and real-time applications use [suface] convention to determine the drawing order of coplanar polygons, when drawing with a z buffer" [3, p.27]. Practically, when a subface and parent face both occupy the same plane the subface is visible and not the parent. If this relationship is not established then Z-fighting may occur and rendering artifact may be present. Z-fighting will be discussed in section A.2, Z-fighting.

Extension push/pops are used when defining new node type. It will be discussed fully in the extension node section.

Header

"The header record is the primary record of the header node and is always the first record in the database file. Attributes within the header record provide important

information about the database file as a whole" [3, p.21]. The header nodes stores all data describing the database as a whole. This include the current OpenFlight standard, projection type, and database corners.

The OpenFlight standard defines which node definition are used. OpenFlight versions differ on their node types, node size, and bit offset of attributes.

Projection type describes the mathematical model used to represent the surface of the earth. The models have different accuracies for different world coordinates. World Geodetic System (WGS 84) is the projection type used in the current thesis's database.

The database is defined as a square that is then transformed into a surface using the projection type. The southwest and northeast latitude and longitude are used to define the database area.

Group

Group nodes are a generic node type used to, organize database structure, aide in the culling process 2.2.3, and perform simple animations. The group node's byte encoding is one of the smallest in the OpenFlight database structure. Additionally, there are very few restrictions on which nodes may be a group node's parent or child. For these reasons, group nodes are used to reorganize the scene graph.

Often databases are reorganized based on some criteria. This criteria often includes spacial location to increase culling performance, or conceptual grouping to improve the readability of the database. The process involves, introducing group nodes into the database, attached nodes based on the grouping criteria and creating a bounding box. In OpenFlight, bounding boxes are volumes of space that encompass

a particular set of geometries, A.3. The group node's bounding volume encompass its entire subtree. Using this volume, the cull process determines whether the group's subtree needs to be drawn.

Group nodes may also be used to create animation. In this process each direct child is activated sequentially in a specified order. Animation will not be discussed in great detail as it is not used in the traffic system.

Object

Object nodes are low-level grouping nodes that contain attributes pertaining to the state of it child geometry [3, p.28]. The attributes determine, transparency, special effects, and how the geometry is rendered. The rendering options include the time of day the object should be rendered in and how shadows are computed. Transparency describes the degree to which an object behind the object record is visible. For example, a window has a high transparency, wax paper an intermediate, and a wall no transparency(opaque). Special effects fields are usually used as marker to signal different interpretation of the node and its children.

Face

The face node is the primary rendering unit of a database and contains attributes describing the visual state of its child vertices [3, p.28]. The face node describes a polygon along with attributes for its rendering. The attributes include a surface material, transparency, flags, light mode, color,material, shader pallet indexes. Together these values determine the rendering of the face.

Color, texture, and material are essential to the rendering process and drastically

effect the realism of the object. For a detail description of the rendering process please refer to the drawing process describded in section 2.2.3.

Transparency describes the degree to which an object behind the face is visible. Flags are used to enable special processing of polygons. Terrain, cut out, and roof line are some examples of flags that change the rendering of face nodes. Light mode determines what lighting model is used to render the polygon.

In Openflight, faces are defined by a sequence of vertexes, a separate node type. The vertices specify a x,y,z coordinate of a point in space. To associate a vertex with a face node the vertices are attached, as direct children, to the face node. The drawing of the face/polygon follows the order of the child vertices from left to right. The specifics are described in figure 2.5.

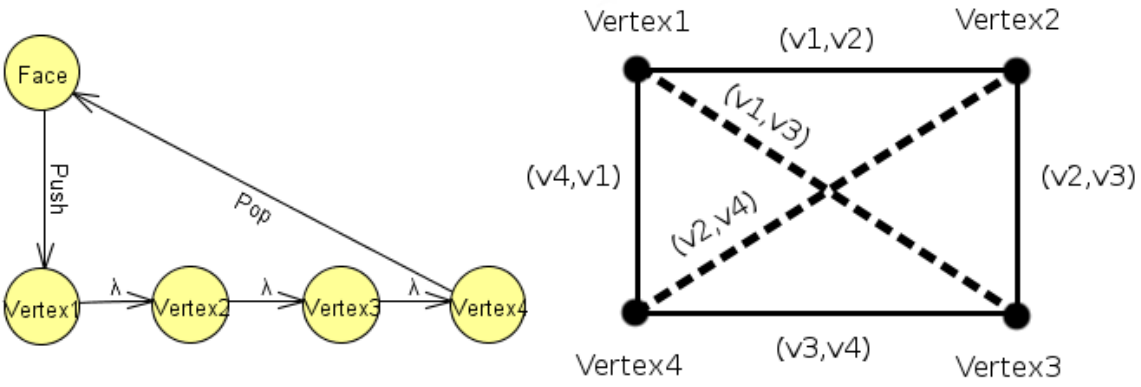


Figure 2.5: A face node comprised of 4 vertices. The left image describes the hierarchal structure of face nodes and its vertices. The right image is the resultant polygon. A line is generated from each vertex pair that are located directly beside each other in the the tree. This also includes the the first and last vertex pair. The set of line for the polygon are $V = \{(V_1, V_2), (V_2, V_3), (V_3, V_4), (V_1, V_4)\}$ and the dotted lines that do not comprise the polygon are $V_{not} = \{(V_1, V_3), (V_2, V_4)\}$

Mesh

"A mesh node defines a set of geometric primitives that share attributes and vertices" [3, p.31]. These shared attributes and vertices are identical to those found in a face node. A mesh can be thought of as a collection of faces with identical attributes that share at least two vertices with another face in the mesh. In other words, a mesh is a set of faces that share and edge with at least one other face in the set. The shared vertices are only defined once but used to generate more than one face. The set of vertices used to generate the mesh are called the vertex pool. The optimization used by meshes reduces the size of databases.

DOF

Degree of freedom node(DOF) "specifies a local coordinate system and the range allowed for translation, rotation, and scale with respect to that coordinate system" [3, p.40]. OpenFlight databases are modeled using a world coordinate as an origin, but in many situations, it is preferable to specify transformations relative to another object or point. This process is achieved using a DOF node. When a DOF node is introduced into a scene graph, the location of insertion becomes a new origin for all subtree nodes. The location is computed using the matrix stack, as described in section 2.1.3.Following DOf insertion, a new coordinate system is established. The local coordinate system redefines the x,y,z axis as well as the origins.

DOF nodes may place restrictions on how nodes may be manipulated at run time. DOF nodes specify the range of valid translations or rotations. Restrictions are useful for modeling the real world since motions naturally occur on some axis but not others. The wheel, figure 5.1, typically rotates about the pitch axis when moving forward and

rotates about yaw when turning. It does not rotation about the roll axis and so a degree of freedom node could be used to restriction motion about that axis.

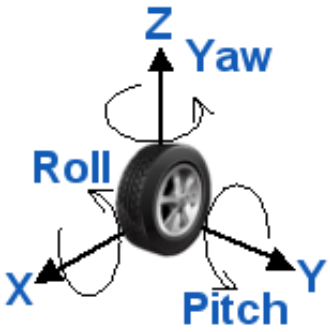


Figure 2.6: DOF node coordinate system with restrictions on certain axises. The DOF nodes stores a new local coordinate system at the center of the wheel. Restrictions may be places on the x,y, or z axis.

External Reference

External references allow one database to reference, or instance, a node in another database (or an entire database) [3, p.40]. An external reference is conceptually similar to instancing as described in section 2.1.5. It differs in that the sub tree is an entire external OpenFlight database. The attributes in an external reference node are used to resolve issue arising from using two databases concurrently. The flag field determines which database's pallets should be used when rendering the external reference. The material, color, texture, and sound pallets may either be taken from the parent's database or external's. The other important field is the path. The path may be relative or absolute. Absolute path involves specifying the entire directory structure to the node including the drive letter. Relative path specifies the path relative to the current database directory location. Care must be taken when using both approaches. Absolute paths have issues when moving OpenFlight databases between

system since the drive/folder structure are typically different between systems. However moving the database file to a different directory within the system does not cause any problems. Relative paths have issues when moving databases within a system. They remain functional when moved between systems. This assumes that the relative directory structure is maintained between the database and the externally referenced file. Another issue to consider is node naming. By default, the name of the node is its path. Consequentially, external reference names cannot be changed within the OpenFlight database and naming cannot be used to distinguish between two external nodes that reference the same file.

Switch

Switch nodes are a set of masks that control the display of the switchs children [3, p.52]. A switch node defines a set of masks. The mask's states are a sequential enumeration of $\mathbb{N}_0$ starting at 0. For each state, a flag is set for each child directly attached to the switch node. The flag, a boolean, determines whether the child is visible for that state. The switch node provides a mechanism for the rapid change of visibility at runtime. A fitting example is a traffic light with three lights and three mask states. The switch node representing the light would have three children sub trees that represent, a green, yellow, and red light.

Level of Detail

Level of detail(LOD) nodes are specialized switch nodes which may have only one child. The LOD record determines whether a child is rendered based on its distance from the observer. The selection process uses the switch in distance and switch out

distance attributes to compute the visibility state. Please refer to table 2.1. Transition range is an optional attribute that may be specified. It is used to blend one LOD node with another.

Range	Visibility
$Distance < SwitchIn$	None
$Distance = SwitchIn$	$ChildPresent$
$SwitchIn < Distance < SwitchOut \wedge$ $Distance \geq SwitchOut$	$Child$ $Child_2$

Table 2.1: Visibilty of child nodes based on distance from observer

Extension

Extension nodes are user created node types introduced into the database system. The process involved in creating a new node type is quite involved. The user may assign any number of attributes to the node. The type of the attribute must be an integer, boolean, float, double, ASCII string, or and XML formated ASCII character string. The user must also define the parent/child restrictions of the node. More specifically, which node type may be this node's parents and which may be its children. For a full discussion of how extension nodes are created please refer to section 4.1.2.

Ancillary

Ancilliary data is additional information that can be attached to existing nodes. Ancillary information decreases the size of the database by providing a framework for adding commonly occurring information. Ancillary information is written sequentially after the primary node but only if it is required. Using this technique avoids the need to assign common attributes, such as transformations, to every node that does not

require them. The following section briefly introduces the various ancillary nodes and their intended purposes.

Comment ancillary records are used to store user created comments. This may include notes on the organization of the database or the reasoning behind certain node attributes.

Long ID are used when a node's name exceeds 8 characters. The long ID is an adjustable size node that stores any name that may be assigned to a node.

Multitexture ancillary records are used when more than one texture is assigned to a polygon. The nodes attributes describe the various textures assigned to it as well as rendering specific information such as mip maps and visual effects.

UV list is used to map textures to a face. It "always follows the vertex list or morph vertex list record and contains texture layer information for the vertices represented in the vertex list record it follows" [3, p.59]. UV mapping is a complicated process in itself and will not be explained in this thesis.

Replicate nodes are used in conjunction with instancing. The replicate node follows an instance node and counts the number of time it has been replicated. It ensures that all replicated/instanced nodes are updated when the source node is altered.

Transformation matrix applies rotations, scaling, and shearing to the node's children. The information is stored in the form of a 4x4 matrix. Please refer to the section 2.2.5 more information.

Bounding volume record is a geometric shape that is used to determine culling and its visibility. The bounding volume is theoretically a minimal volume geometry

that contains all its subtree nodes. Practically the bounding volume may be manipulated by the user and may not meet the above criteria. In OpenFlight, the bounding volume may be in the shape of a box, sphere, cylinder, convex hull or histogram [3, p.64].

2.2 Visualization and 3D Graphics

"The term computer graphics includes almost everything on computers that is not text or sound" [8]. Simply put, computer graphics are images generated by computers. The programs described in this thesis manipulate computer graphics in order to create a virtual environment. The following section introduces the language and concepts used in computer graphics in order to avoid any ambiguity while discussing this thesis.

2.2.1 Object and Scenes

Conceptually, the basic unit of a a virtual environment is an object. Objects are defined as a set of polygons each with visual properties describing its surface. The set of polygons considered together create the object. There are no strict restrictions on which polygons should make up the set, but conceptually the grouping is based on the object it is trying to model. The overall organization of the virtual environment including all objects is called the scene graph. The scene graph is a hierarchical tree consisting of object constructed from other objects or polygons. Typically the polygons are grouped into objects and in turn these nodes may be grouped. An example of object grouping would be the human body with arms, leg, head and torso as described in figure 2.7.

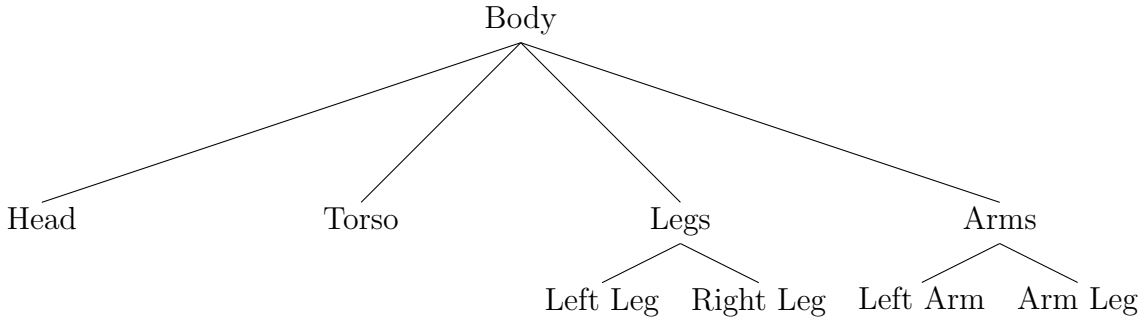


Figure 2.7: This scene describes a simple representation of the human body as an object in a scene graph. Each leaf in the tree consists of a set of polygons, that considered together, describe the object.

In most applications, polygons are convex. Convex, as opposed to concave, polygons facilitate intersection problems, a frequent operation in computer graphics. A convex polygon is a polygon where all it's internal edges are less than 180 degrees. More formally,

> if the are n vertexes that make up a polygon and $v_i, i = 1, 2, ..., n$ are the verteces that define the polygon. Then the edges that define the polygon are $e_i = v_{i+1} - v_i$. Also assume that x_{n+1} is equivalent to x_1. Then a convex polygon is a polygon where $v_i \cdot v_{i+1}$ has the same sign for all i. [5, p.138-148]

2.2.2 Virtual Environment

The application scope of computer graphics is vast. In order to accommodate all situations, a combination of textures, materials, and colors are used. The color's attribute covers the surface of the polygon with a uniform color. Material affects how lighting interacts with a face in order to mimic real world objects. For instance, specular material mimics the reflection of metallic surfaces and ambient lighting simulates the

lighting of object with no direct light source. Textures are images that are mapped onto a polygons surface. This process involves sampling an area of a 2d image and placing that sample on the polygon's surface. Together, color, material, and texture successfully model 3D object.

Several other factors are involved in making a virtual environment. The skybox is a box in which the virtual environment operates. It is used in large volume environments. The box's internal walls are images that represent the landscape view. Within the skybox, the camera's view resembles the view of distant object such as clouds, mountains or celestial bodies. The skybox moves with the camera in order to maintain this illusion. The skybox is an essential element in creating large scale virtual environment.

The next concept is levels of detail (LOD). Levels of detail are several different representations of the same object. The representations differ in the number of polygons used to model them. Based on an object distance from the camera, the LOD system selects which model to use. For example, a house with 3 levels of detail; high, medium, and low. Low resolution models are used for houses when they are more than 2 km away, the high resolution model when closer than 500 meters and medium for the range between 500, and 2 km. LODs reduce the number of polygons in the scene by using simplified models when detail in not necessary.

The final factor to consider when generating a virtual environment is the software system. There are several terms important to the understanding of the software system. Frame rate is a common term used to measure performance in computer graphics. Frame rate is the number of times the frame loop is executed per second. The frame loop determines which images to draw and handles the procedure for

drawing the frame. Culling is the process that determines which images should be drawn. Finally, rendering is the process of converting computer data into images for display.

2.2.3 Frame Loop

The frame loop is the process that creates and draws images to a display. The process can be divided into 3 major phases called application, culling and, drawing. These phases are essential to the rendering process and together form the main loop that runs the virtual environment.

The Application phase performs computations related to the program running the visualization. The programs process data and determines the appropriate response to user input.

The cull phase determines which objects are to be drawn. The process involves three concepts, levels of detail, bounding box,A.3, and viewing volume.

The bounding volume is defined using a point,the camera, and two planes,far and near clip plane. From the camera, rays are projected out that represent the camera's field of view. This creates a pyramid volume that is called the viewing volume. Typically limitations are placed on how close and far images may be seen. Using a near and far plane called clipping planes, a volume is defined where objects are visible to the camera. Relative to the camera, only objects in front of, near, and behind the rear clip plane are visible. Figure 2.8 illustrates the viewing volume concept.

The culling process traverses the scene graph from top to bottom searching for relevant nodes. The nodes of interest are, at the top level, LOD nodes, the mid

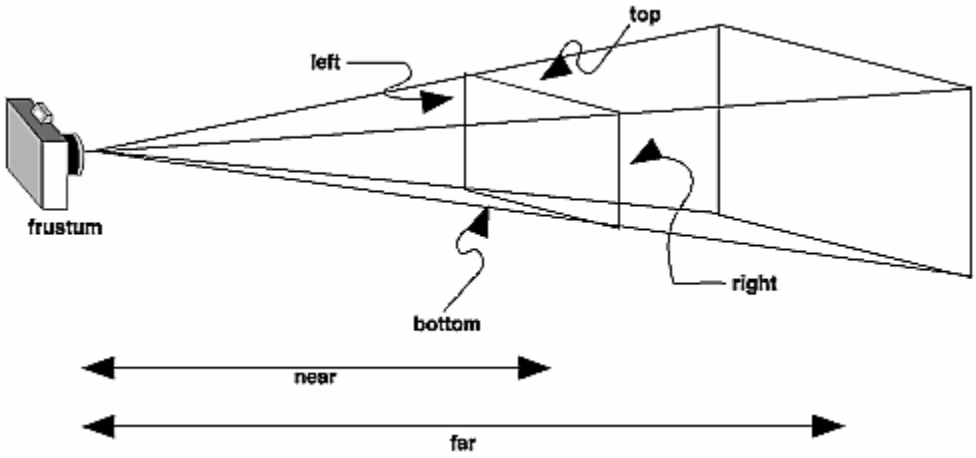


Figure 2.8: Illustration of a bounding volume. This example includes a near and far clipping plane.

levels, bounding boxes, and the bottom, geometries. LOD nodes are typically encountered first during the traversal. For a given LOD node it is determined whether $switch_{in} \leq camera_{distance} \leq switch_{out}$. If it holds than the LOD child is traversed. When an active node is encountered with a bounding box, a computation is performed to determine whether the box intersects the bounding volume. If no intersection is found the traversal does not proceed down that branch. When an intersection is found the traversal continues downward. The process continues until a geometry is found. A geometry being a face, mesh or light point that the application may render. The encountered geometry is added to the draw list, the list of geometries that will be rendered in the draw phase. Using this process, the traversal systematically determines which nodes are to be drawn. Bounding boxes and LOD nodes are not strictly necessary to determine which nodes to draw, however, they drastically reduce the scene traversal time but quickly eliminating irrelevant branches. Care must be taken not to introduce to many LODs and bounding boxes as the gains from eliminating

branches will be offset by the additional computations necessary for each comparison.

The draw phase renders relevant data onto a display. Issues arise when more than one image request the same display location for rendering. The z-buffer is used to resolve this issue. During the draw phase, the list of geometries that must be drawn is processed. This list is known as the draw list and it was computed during the cull phase. The geometries in the draw list are converted into images using the attributes defined by their node type. The final result is a set of pixel each with color and distance to the camera. This information is passed to the z-buffer. The z-buffer consists of a list of pixels with an associated depth value. The depth values are initialized to a very large value. For each inputed pixel, the depth value of that pixel is compared to that of z-buffer. If the inputed pixel's depth is smaller than the z-buffer's depth then the z-buffer's data is overwritten. The final result is a process where images are display in a manner consistent with real world applications. More specifically, when two images are in the same direct line of sight of the observer than the closer image appears on the display.

2.2.4 Real Time Rendering

Visual media is a common means of communication with several subcategories including films, interactive video, and real-time simulations. Although the display of images is common in all three subcategories, there are significant differences in how these technologies are implemented. This section outlines the features of real time simulations and the unique constraints they impose.

The first feature of real-time systems is user input. Videos, unlike interactive media, do not take user input. Therefore, the sequence of images in a video may

be precomputed. When a video is displayed to a user, the images are replayed to the user without complex computations at runtime. Interactive media differs. It involves input from the user and this input may change the sequences of images that are rendered. Interaction therefore forces computation to be performed after the user input. The result being that in order to maintain a consistent frame rate constrains must be placed on the frame computation time. These constrains are called deadlines.

Deadline type is the distinguishing factor between video games and simulations. Video games and simulations both seek to immerse the user in an environment. Creating a dynamic environment that responds to the user's interaction is critical to achieving this immersion. However, the systems differ on whether immersion may be dropped when deadlines are not met. Video games have soft deadlines while simulations hard deadlines. Simulations by definition have a set frame rate. This means that for each frame there is a specific time interval where all computations must be completed. If the computations are incomplete, the images will be drawn yet incomplete. In video games, the frame rate may vary. This allows the application to exceed a frame's time interval in order to complete the drawing stages. This situation results in a decreased frame rate. Real-time systems, unlike games, maintain a consistent frame rate.

Models in real-time systems have less detail as measured by number of polygons. The number of polygons greatly affects the time required to render a scene. To ensure that deadlines are met, level of details(LOD) are implemented. The LOD systems allows for the rapid exchange of models at run-time. The models increase or decrease the number polygons of the object. Using this system the application programmer may adjust his scene to ensure the deadlines are met and no rendering artifacts occur

due to missed deadlines.Refer to section 2.1.6 for a detail description of level of detail implementation.

In real-time system, user input is given high priority. If a user controls an object within the simulated environment, then the object must react within a time interval for immersion to occur. Without this deadline constraint, the illusion of control is lost and the effectiveness of the simulation decreased. These characteristics define a real-time system and are serious design constraints in this thesis's system design.

2.2.5 Transformations Matrices

Transformation matrices are the typical technique used to manipulate objects in interactive systems. The concept of Transformation matrices was introduced in section 2.1.3 and here it will be elaborated on in more detail.

Transformation matrices are the common method used to represent transformation in 3d graphics. For a given euclidean space $\mathbb{P}^n$, the square matrix used to represent transformation in this space is of size $n+1$. $n+1$ matrix is used to allow for the addition of translation into the transformation. The $n+1^{th}$ column or row see figure 2.12 is used to specify translations. It is possible to represent many translations as a n size matrix, however, the mathematics are more complex. The n+1 form results in independent value entries for transformation and scaling that can be easily extracted without computation. The cost of more memory for the n+1 vs n matrix is more than offset by the reduced computation time in a deadline critical environment.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.9: Identity transformation matrix

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_y = \begin{pmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_z = \begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.10: Rotation transformation matrix about the x, y, z axis respectively by an amount θ

2.3 Presagis Software

A large software package was used during the development of the intersection visualization system. The package, designed for the development of motion simulators, was acquired from Presagis Inc. Presagis is a subsidiary of CAE and CAE is the world leader in civil aviation simulation and modeling technology. Logically, Presagis software is tailored for flight simulation. This thesis describes the process of adapting Presagis software to ground level simulation. More specifically, these sections introduce the software packages used in development in order to identify the design constraints they impose and to give context to the development process.

2.3.1 Terra Vista

Terra Vista is a tool used for the generation of large scale terrain. It facilitates the creation of content by automating many of the features common to most databases.

$$\begin{pmatrix} a_x & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.11: Scaling transformation matrix along the x,y,z axis respectively

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_t & y_t & z_t & 1 \end{pmatrix} T' = \begin{pmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.12: Translation transformation matrix. This transformation would move a vector v = $\{x, y, z, 1\}$ to position v' = $\{x + x_t, y + y_t, z + z_t, 1\}$. The two forms of the matrix depend on how the vector is represented as a matrix. If v is a 1x4 matrix than the left matrix is used and the multiplication is $v \cdot T$ and if v is a 4x1 matrix then the multiplication is $T' \cdot v$

It allows the user to specify, levels of detail, world coordinates for the database, vector based culture generation, and correlated output.

Database Initialization

Terra Vista facilitates the creation of databases. In the initial step, the user specifies the world coordinates of the database. Next, the terrain blocks are set up. This process specifies the gaming area of the database. The gaming area is a 2 dimensional area of blocks defining the database. Each block consists of an OpenFlight database. The blocks are then merged together in a master file using external references, 2.1.6. This process facilitates the loading of the database by allowing the application to load the master file without referencing each sub database. The databases are also given specific dimensions and levels of detail.

$$\begin{pmatrix} a_x & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.13: Scaling transformation matrix along the x,y,z axis respectively

Level of Detail Setup

When creating the database in Terra Vista, levels of detail are specified. As explained in section 2.1.6 and 2.2.4, levels of detail are different representations of the same object with a varying number of polygons. During the initialization process, the number of LODs are specified as well as the range of each LOD. By adjusting the range, the number of polygons for each LOD level can be specified. Additionally, the total number of LODs can be increased. Given a fixed polygon budget, this increase results in a smoother visual transition since the LOD's polygon numbers differ by a smaller amount between LODs.

Culture Generation

Terra Vista facilitates the generation of culture using vector data. The entire process involves three components, vector data, attributes and processing paths.

After the database is initialized, vector data is specified. In Terra Vista, vector data may be points, lines, and areas. Points are used to insert OpenFlight models at a location. Typically points represent trees, houses or street lights. Lines are a sequence of interconnected points. Lines are used for the generation of power lines, roads and rivers. Typically, culture is placed directly on the line or parallel to the line. Areas are describes as a sequence of lines where the initial and final point are

equal. They are also used for the generation of lakes, forest or fields. Areas are useful when specifying locations where the ground texture differs from that of the default terrain.

Following vectors creation, attributes are assigned. Any number of attributes may be assign to a given vector. These attributes are then used by processing paths at compile time to generate appropriate culture.

Terra Vista generates culture using a tree structure. Each node consists of a set of attributes that it may match. Leaf nodes specify models or actions. When building the database, the tree is traversed in a depth first manner following only branches that match the attributes assigned to each vector. When a match is found, the leaf nodes actions are performed and culture is generated.

Correlated Output

Terra Vista contains a variety of tools that correlate vector data to other output formats. Those of relevance in this thesis, are the road compiler and ACX generator.

The road compiler, as the name implies, is responsible for the generation of complex roads. Given a vector, the complex road compiler generates roads with custom width, side walks and medians. When two line vectors intersect, the compiler identifies this point as an intersection. The point, along with the attributes of each line, are then used to generate a traffic intersection with appropriate traffic lights and pedestrian crossing lines.

The ACX generator is responsible for the generation of AI implant data. The AI implant is the program responsible for the control of pedestrian and vehicle AI in the simulation. The ACX generator creates road networks and navigational meshes,

which are used determine traveling behavior and intersection data. In turn, this data determines AI behaviors at intersections. The type of intersection generated depends on the number of lines intersecting the vector as well the attributes of these lines.

Terra Vista is responsible for the creation of AI data and OpenFlight databases. The generation process has a considerable amount of flexibility, however, some factors cannot be changed. The result being that several design constraints are imposed by the software system. They will be discussed in section 4.1.1, initial design.

2.3.2 Creator

Creator is a 3D modeling environment specifically designed for use with OpenFlight. It allows for the creation and alteration of OpenFlight models. In this thesis, Creator was used for a variety of tasks including creating/importing models, optimizing database organization and testing custom nodes.

Creating Content

Creator's primary purpose is the generation and modification of OpenFlight models. In this thesis, very few models were created from a blank database. Typically, an existing model is modified to satisfy some purpose in the simulation. The modification process involves changing the orientation of models, removing unnecessary faces and changing textures.

Another source of content is external file formats. Creator has the capability to import other 3D modeling databases. The importing process has a few bugs and so, the resulting database is not entirely transferable. To ensure proper conversion, incorrect materials and textures must be corrected. The process involves adding

appropriate material and correcting the uv mapping of textures.

Optimizing Performance

Many databases are created with a non-optimal structure. For instance, some database consist of only one level of face nodes. To optimize this database, group and LOD nodes are inserted with respective bounding boxes and switch in distances. The introduction of new nodes increases the performance of the culling process as described in section 2.2.3.

Another source of inefficiency is duplicate texture pallets. When there are several external references within one database the common practice is to maintain a texture pallet for each reference. When there exists textures common to several pallets this is redundant. To optimize, Creator creates a large texture pallet that contains all the sub database pallets. Duplicates textures are then manually removed.

Creator can be used to find and optimize sequences of face nodes. When a sequence is drawn, the attributes of the current node are compared to that of the previous. If the attributes differ, then the new attributes are loaded. Loading is a costly operation. This procedure can be optimized in two ways. One, if nodes exist with identical attributes then these nodes should be grouped into meshes. Meshes,2.1.6, are a set of face nodes that share the same value for attributes. Two, if there exists a set of of face nodes with similar attributes, then they should be ordered sequentially to prevent frequent reloading of attributes.

Custom Nodes

The first attempt at creating a traffic system, 4.2, involved using custom nodes to store traffic information. Within creator there exists a tool named the attribute visualizer. This tool allows the user to visualize and edit attributes of a given node. The tool was used frequently during the custom node development process as a means of validating the new node's implementation.

2.3.3 Stage

Stage is responsible for scenario management in the virtual environment. It provides several services including, entity management, external program interfaces and event scripting.

Entity Management

Stage tracks and manages all entities in the virtual environment. Entities are objects and stage tracks their position and a vector of their velocity. Stage allows for the repositioning or alteration of the entities at runtime. As well, it specifies the initial value of their variable when the application starts. Another interesting feature is its ability to change entities at run-time. Any of the previously described processes may be performed at runtime. This includes the addition or removal of other entities.

External Plugging

Stage controls entities in the virtual environment. Stage's role as a management system makes it the ideal medium for the integration of other external applications. Stage is written to allow plug-ins that interface between it and other applications.

There are a few relevant detail to consider during the integration process. One, Stage runs at a set frame rate of 30 frames/second. This means that all plug ins have set deadlines for the reception and processing of information. Two, plug-ins must utilize the networking system. The system in this thesis runs across several different computers each with its own application. Networking is used to communicate between the different applications. Therefore, plug-ins are responsible for receiving and decoding network messages. To ensure that frame rate deadlines are met, strict size constraints are placed on the information being passed between applications.

Scripting

Scripting is the process of automating a series of tasks. Stage supports the use of scripting to trigger events such as the creation, movement or deletion of entities. Scripting is not thoroughly used in this thesis and so it will not be described in great detail.

2.3.4 Vega Prime

Vega Prime is responsible for the rendering of a scene. It takes one or multiple OpenFlight databases and renders them. Additionally, Vega Prime determines the climate conditions of the simulation and the manner in which lighting is calculated. Finally, Vega Prime performs computations that are specific to the rendering process such as collision detection and object transformations.

Rendering

Vega Prime is a program specifically designed for the rendering of OpenFlight databases. It loads, optimizes and renders the scene specified by the OpenFlight database.

Vega Prime renders the database using the traditional rendering model with application, culling and drawing phase, as described in section 2.2.3. Additionally, tools are available to fine tune rendering performance and realism of the final product. This includes lighting models which differ on realism and performance, and LOD scaling to reduce the number of polygons in a scene.

IG Application

Vega Prime is responsible for visualization of the virtual system, and as such, is responsible for processes that directly effect rendering. Vega Prime contains a programming frame work to facilitate the development of applications. The framework, an API, facilitates the rapid transformation of rendered objects. In this thesis, the API was used for, state change of traffic lights, collision detection, custom traversal and optimization of the scene graph.

2.4 History of Driving Simulation

Driving simulators were initially developed to assess the skills and competence of public transit operators in the early 1910s [12, p.1]. The simulation consisted of artificial vehicle equipped with various stimuli and sensors. The stimuli used were devices created to imitate specific components of the driving experience. In the 1960s, driving simulation projects were undertaken in the US and Japan due to a very high

fatality rate amongst vehicle operators [14, p.1] [12, p.1]. At the same time, many automobile companies started developing their own driving simulators [4, p.2-4]. Computers were not in wide spread use and so the simulations consisted of "analogue computers or electrical circuits" [14, p.3]. The driving environment was displayed to the user via a screen and a prerecorded film of a vehicle driving [12, p.1]. As time progressed, so did the technology used to drive the driving simulators. Various studies were performed through the 70's and 80's on a wide variety of topic including: performance variables of older and younger drivers, hazard mitigation, and visual attention [12, p.1]. The theme amongst the driving simulators of the present and past, is that they "place the subject in an artificial environment believed to be a valid substitute for one or more aspects of the actual driving experience" [13]. As technology and society changes, so do the way in which people use their vehicles. For instance, advances in cell phone technology have resulted in greater cell phone use while driving. A study by VirginiaTech Transportation Institute states "secondary tasks account for 23 percent of all crashes and near-crashes". It is import that modern driving simulators be developed that address the impact of an ever changing driving environment.

Chapter 3

Simulation System

3.1 Overview

This thesis describes the process of developing an intersection visualization system for the simulator at McMaster university. System development started when a project was undertaken to upgrade the existing simulator system. The new system haw several new objectives, but the one of concern in this thesis was the development of a realistic driving simulation. Before commencing a discussion on the traffic system, it is worth introducing the simulator system. Its design introduces constraints on the implementation of the traffic signaling system as whole.

Figure 3.1 describes the systems involved in running full simulation.

The McMaster simulator is designed to perform unique experiments that combine stimulous presentation and state of the art psychology techniques. To accommodate the wide range of possible scenarios and experiments a combination of visual, motion, audio, and data gathering systems, are employed.

To gather information on the user response, the EEG, eyetracker, user controls

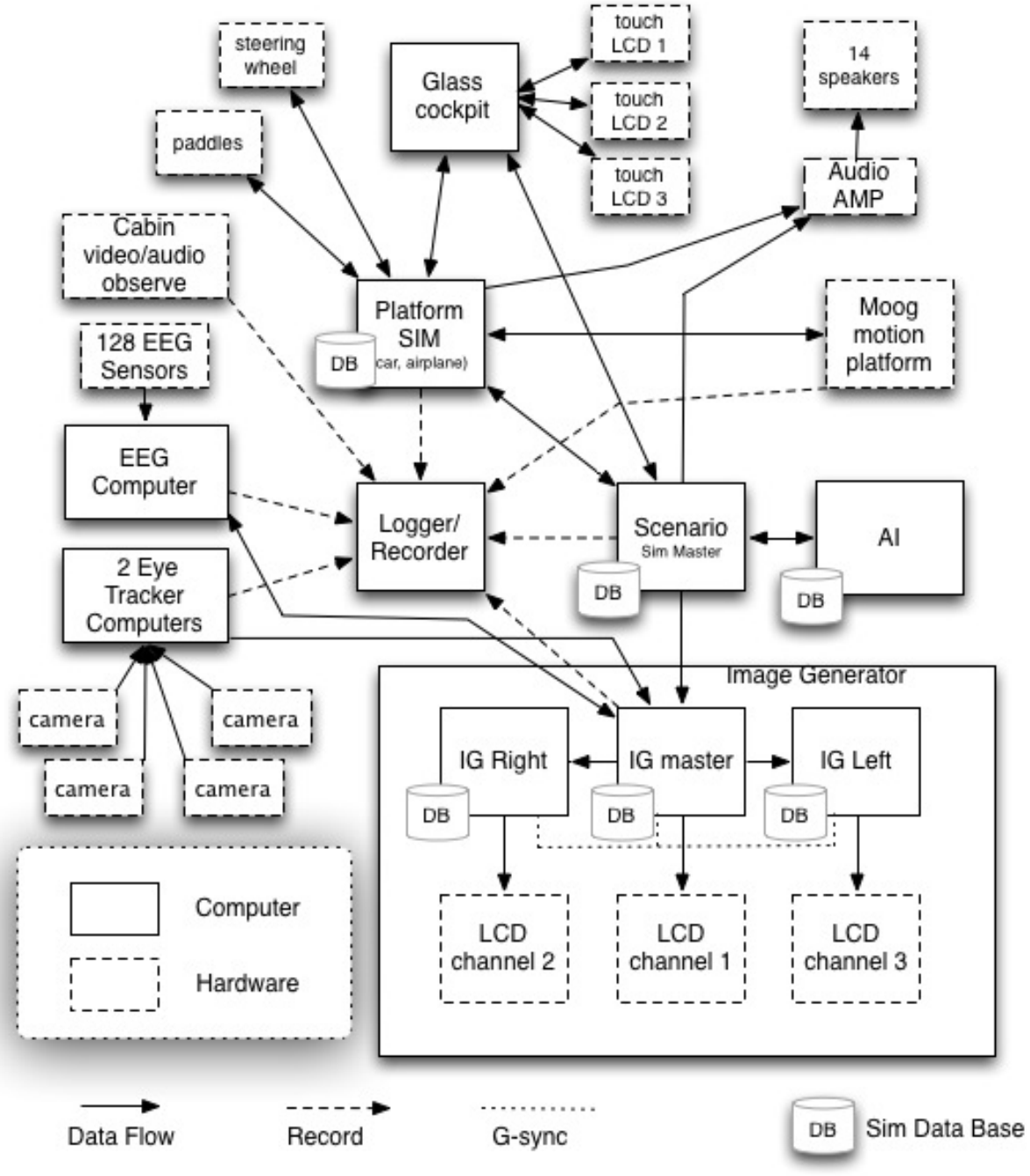


Figure 3.1: This figure describes the components comprising the simulator system as well as their relationship to other components.

and logger are used. The Electroencephalograph (EEG), measures the magnetic fields generated by the brain during experiments in order to isolate patterns of activity. The eyetracker system tracks the motion of the eyes during the course of the experiment. The system is accurate enough that it can determine where the user is looking at a given time point. The user is typically provided with some sort of control in order to interact with his environment. This comes in the form of button boxes, keyboards, steering wheels and touch screens. Finally, all these systems pass their information to the logger system that records the various user input. The information is then analyzed at a further date.

The simulation system itself is comprised of several systems that may or may not be used based on the context of the experiment. The systems are audio, motion and image generation.

The sound system provides auditory stimulus to the user. Audio is typically used to immerse the user in an environment, or as a cue in a experiment. Immersion sounds are sounds appropriate to scenario. As an example, these could be the sounds of the engine while driving or blinker sounds when signaling a turn. Cue sounds are used to communicate to the user that some action will, must or can be performed. For instance, in some experiments a sound is played slightly before a stimulus appears on screen.

The motion platform applies forces to the user. The platform is used for two purposes in the context of this system. One, to analyze a user's perception of external forces or two, to add realism to a simulation. The science of how humans perceive motion cues within motion simulators is not well understood. This system allows for controlled application of forces so that detailed experiments on human perception of

forces can be performed. The motion platform is used to simulate a flight or driving experience. When driving, forces are experienced due to acceleration and deceleration of the vehicle. A more realistic simulation can be created by computing these same forces and applying them to the user. The forces are computed by the physics engine localized in the platform's sim. The forces are then simulated by the movement of the motion platform. The processes of mimicking forces using a motion platform will not be explained in this thesis since it is a complicated subject on to itself.

The image generation(IG) system is another important method for providing stimulus to the user. Again, the IG system is used for experiments or simulations. The IG system is interesting as it is comprised of three separate computer systems. Each IG's display is installed within the simulator. One is centered on the users view while the others are installed at approximately 60 degrees off center. When running a scenario, each IG contains an identical copy of the database. Using view frustum calculations, each screen displays an appropriate camera view into the scenario world. As position of the camera changes so does the position of the camera in each IG. Using these techniques, the user has a sense that he or she is perceiving the scenario world.

Creating a realistic driving simulation is not a trivial process, and therefore several properties were identified that are factors involved in achieving this goal, including, a realistic physical environment, AI controlled entities, motion simulation and a traffic system. This thesis describes the development of the visual component of the traffic system as well as its interaction with the other subsystems involved in the project.

This concludes the introduction of the simulator system. The discussion focuses on user interaction with the simulator as well as the techniques used to record the user's interaction. Next, the discussion will focus on the development process for

scenario development and traffic signal constraints.

3.2 Scenario Development

Scenario development is the process where a virtual environment is created with the purpose of simulating a specific set of conditions. A major goal of the simulator system at McMaster is the development of a realistic driving simulation. This section discusses the development of a driving simulation scenario as well as the components used to create this environment. The components are intelligent characters that follow traffic rules and visual content creation.

Content creation is a process where the virtual environment is populated with elements that fit the scenario. Elements include terrain, culture and entities. Typically this process is performed by a team of artists. In the case of video games, half the development staff are involved with content creation. This is not a realistic scenario given the budget of McMaster's simulator. Instead, development was focused on tools for the rapid development of content. Terra Vista a terrain generation tool was instrumental in this process. As described in section 2.3.1, Terra Vista is GMS system that translates vector data into a database populated with models. A fair amount of time was spent programing Terra Vista to utilize our models, generate appropriate traffic AI data and analysis vector data in a manner that meets our traffic requirements.

Entities are "intelligent characters" that are capable of analyzing their environment and reacting appropriately. The term used to describe intelligent character in the context of computing is artificial intelligence (AI). In an urban driving environment there are typically other vehicles on the road and pedestrians walking on the sidewalk as well as crosswalks. To simulate an urban driving environment, AI was

developed to simulate vehicles and pedestrians. The AI implant from Presagis greatly facilitated the process by providing basic behaviors. Additional behaviors were added to the system by another member of the development team.

The final component used to create the driving environment are traffic lights. Traffic lights are used to signal to the user, the vehicle and pedestrian when they may proceed into an intersection and what actions they may perform. Creating a traffic intersection system by itself is not a complicated task, however, the design of the simulator system requires that the intersection system communicate with the OpenFlight database, the image generators and the AI implant, which complicates the design process significantly. This thesis describes development and integration of the intersection visualization system into McMaster's motion simulation system.

3.3 Intersection Signaling System Development

The final intersection signaling system must meet certain constraints based on the design of the simulator system.

I **The final product must integrate into the current development process.**

The generation of intersection signaling data must work with Terra Vista. More specifically, the intersection setup must be generated by Terra Vista or computed from its outputted AI or Open Flight data.

II **The inter-system communication must consist of small messages that require little computation at run time.**

The image generator, artificial intelligence, and scenario manager are all located on different computers and communicate using a combination of TCP/UDP protocols.

Due to the demanding deadlines for rendering in a real time system, it is important that the signaling system messages do not interfere with network communication via overly large packets or frequent messages.

III **The system must have a high level override**

The simulator system will be used to perform experiments on driving conditions. Many of these experiments will involve custom traffic patterns or sudden changes in light state. It is therefore important that the traffic signaling system be able to quickly change state via a simple message.

IV **The system must be able to model all possible traffic light systems used in the real world**

The simulator may be used to model real world locations. It is therefore important that it be able to model any type of intersection or sequence of states.

V **A user must be able to program their own traffic intersection**

Eventually the system should be user friendly enough that a psychology student without extensive knowledge of programming may be able to program their own custom intersection system.

Given these constraints, the development of traffic system commenced.

3.4 Solution Development Process

The development process used in this thesis was not formal methods. Rather, a custom process was used that focused on testing and a short development cycle. This

decision was made because some elements required for a complete design were not present. The individual software packages used were quite well documented however, the interaction between software modules was not. More specifically, it was unknown how a change in the database, via Terra Vista or Creator, would effect performance in visualization, Vega Prime. Determining these inter-system interactions was key to the design process. The alternative approach was a testing intensive process. By rapidly prototyping, the obvious implementation issues could be exposed. The solution development process is broken down into 3 steps:

1. Design
2. Implementation
3. Evaluation

The system is designed under various constraints, while attempting to provided as many desirable characteristics as possible. Algorithms are developed to compute information and solve encountered problems. Several implementation strategies are evaluation, and eventually, one is chosen. The actual implementation is left to the next section.

The implementation takes the model's design and translates it into an application. It uses tools such as programing languages, outside application, APIs and scripts to complete the physical program. During this process, unexpected behavior may be encountered or assumptions may be invalidated. These failures are noted and will be analyzed in the next step.

The evaluation stage critically analyses the program generated in the implementation. Successful and unsuccessful components, of the implementation, are identified. The unsuccessful component's root causes are identified and used to generate a set

of properties that guide the refinement of the current model. These properties will guide the redesign of the current model in the next development cycle.

When issues are encountered during implementation, the model must be adjusted. Using the properties identified in the evaluation section, the design phase creates a new model of the intersection state visualization system. The design, implement, evaluation cycle is repeated until a complete solution.

The programs specified by each model can be generalized. Each model consists of three steps: information gathering, intersection generation and the run-time application. As each model is developed, new process and algorithms are added in order to reach a complete solution. Each of this components will be integrated into one of these three steps. To improve the readers understanding of changes amongst each model, a diagram is added at the beginning of each model that illustrates each new process as well any changes to the program structure or data flow.

Chapter 4

Database Solution

The OpenFlight database solution went through several iterations before being abandoned. Although not used, the older models did lead to several useful conclusions. This chapter will outline the motivation for using OpenFlight, the three implementations and finally the conclusions drawn from this model.

4.1 Overview

The initial development of the intersection signaling system focused on utilizing the powerful set of tools for OpenFlight databases. Implementation issues arose and a new solution was proposed. This cycle repeated itself several times resulting in three models. The models are, one, the custom intersection node; two, Vega Prime intersection control; and three, finally the external reference ID system. All these implementations utilize the OpenFlight API to some degree. During the development process several limitations of the development tools were discovered. The limitations eventually resulted in the abandonment of the OpenFlight extension tools all together

but still utilizing the OpenFlight API.

The following section details justification for implementing the intersection signaling system in OpenFlight. Following that discussion, the OpenFlight custom node framework is briefly introduced.

4.1.1 Initial Design

Fewer Inputs

By incorporating traffic light information into the OpenFlight database encoding, we eliminate the need to add another file format to create, maintain, and update. This model is consistent with **Constraint I**, integrate into the current development process.

Another possible solution was storing intersection data as a part of the AI implants data. The AI file formate (.acx) is an XML based encoding and it would be suitable candidate for storing intersection data, however the interface for ACX modification had little to no documentation. It was decided that the OpenFlight tool set was more extensive and would be a more appropriate solution. It should be noted, that post development, there was a release of the AI implant that created intersection data within the acx file format. This system is utilized in the second implementation of the intersection signaling system.

OpenFlight Synergy

As described previously, OpenFlight contains a set of tools for the generation and use of custom node types. OpenFlight is known for its efficiency as a real time application database. It can therefore be assumed that incorporating our intersection signaling

system into the OpenFlight system would lead to some increased performance. This is indeed the case. For one, by incorporating the custom node we leverage the traversal system used in OpenFlight. As described in section 2.1.2, traversal is DFS that eliminates branches of the scene graph when they are not rendered. This means that the custom nodes are not read unless they are to be rendered. This will drastically improve performance across large simulations. Two, OpenFlight is the standard database system for all Presagis products, and therefore, the custom nodes should integrate easily into the other products; AI, stage, and VegaPrime.

Development Tools

The OpenFlight API is a set of tools used for the creation and modification of the OpenFlight database system. The tool set is well documented and allows for a great degree of customization. The sample code and two reference manuals facilitated the learning process which, in turn, lead to rapid prototyping. This was ideal for our agile development process.

4.1.2 OpenFlight API

The OpenFlight API contains functions for the manipulation and customization of OpenFlight databases. A subset of the tools allows the user to generate their own extension of the OpenFlight standard. The process is quite involved but the end results is a new node type. The design of nodes can be described as object oriented (OO), however, the properties of the object are quite restrictive relative to other OO designs. There are three steps in the definition process. First, a new or existing node attributes are defined. Second, function are defined related to that node. Third, the

relation, of that node to the other node types, is defined.

Attributes

In OpenFlight, a node is defined by the data it stores. Data is named an attribute and defined using the dataDef keyword. The attributes are defined by data types similar to those of most programing languages.

- Boolean (true or false)
- Integer (a whole number)
- Float (single precision real number)
- Double (double precision real number)
- Unformatted String (variable length ASCII character sequence)
- XML String (variable length ASCII character sequence interpreted as XML format)

[2, p.17]

Functions

The functions used with a node in OpenFlight are limited relative to most programing languages. The two categories of functions are action and drawing

An action function is one that is called when a node's attribute is altered. Pre action function is called just before the node is altered. This is used when a value is needed for the attribute update process. Post action function is called after a node's attribute is altered. This is used to trigger the update of other attributes that are derived from that node's data.

Draw functions are used in conjunction with creator. They determine how Creator renders a custom node. Draw functions are typically used with drawn node types such as face or meshes.

Relationship

The final property of a node is its relationship to other nodes. In OpenFlight, the relationship is limited to the parent and child. Each relationship defines the set of nodes that are valid as a parent or a child.

4.2 Model 1: Custom Traffic Node

The first implementation of an intersection signaling system used a custom OpenFlight node. Each custom node stored information related to a particular intersection. The node's children were used to identify traffic lights belonging to that intersection. The following discusses the intersection traffic information encoding. Next, the node attributes are specified. Finally, a discussion on the failures of this model and the lessons learned that lead to the next implementation.

4.2.1 Design

Encoding

There are many ways to represent an intersection's state. The state themselves are determined by the components used to create the intersection. At the lowest level are light sources that are either on or off. Constructed from these light sources are traffic lights. Finally, there are intersections consisting of multiple traffic lights.

Traffic lights can be described as having states. A simple light consists of red, yellow, and green indicators. In this example, each light may be on or off independent of the other. A complex light contains left turn signals, right turn signals, and at least 2 pedestrian signals. If there are no assumptions on the relationship between lights, then the number of states are the power set of the set of all indicators. This fact will be important for the encoding of intersection state.

The next encoding is at the level of the intersection. A traffic intersection is a grouping of traffic lights that are located near an intersection of roads. Within the traffic simulator, there is a need for a state that describes the intersection as a whole. This need arises from **constraint II & III** which states that the system must have a high level override and that the traffic messages must be small. A state encoding will be used to specify the requested state in the override. As previously discussed, the number of states for an individual traffic light is the power set of that light's indicator. An intersection's possible states is therefore the power set of the light's states that make up that intersection. Consider a four way intersection with each light having three indicators and a pedestrian crossing signal for two directions. Assume that the pedestrian light for each direction has two indicators each. In this scenario, the number of states

$$= 2^{(light\_indicators + pedestrian\_indicators) * number\_of\_light} = 2^{(3+4)*4} = 2^{28}$$

This simple example requires 28 bits to encode the traffic state. The constants of the traffic message are that they must be at 64 bits. Almost half the message space is required for this simple example. Typically in most modern intersections there are at least two lights per direction, additional turning signals and three indicators for

a pedestrian light. It is easy to imagine a scenario where the 64 bit encoding would not suffice. There is a need for some encoding that simplifies the state space of the intersection states.

Several techniques are used to simplify the traffic state description. Lights are grouped, indicators are mapped to switches and assumptions are made about intersection states.

First, lights are grouped together if their states are identical for every intersection state. This situation occurs frequently in modern intersections where more then one light is used to display identical traffic information for a given direction. The grouping of lights is referred to as a light group. Another situation where grouping occurs is between lights on opposite sides of an intersection. For a given 4 way intersection with north, south, east and west traffic lights, the light for north and south will be in the same state and east and west the same state. This pattern is used to group lights. This relationship does not hold for all intersection so this grouping is optional based on the intersection design.

Second, the indicator states are stored at the level of the database as a switch, 2.1.6. The switch defines a global integer state where each indicator is either on or off. The switch integer values are used to represent the state of the light group. Only a small subset of possible states are used in most traffic lights, therefore, the switch states tend to be small in number. Additionally, encoding light group states as a switch, results in a finite sequence of light states. This fact will be important in the implementation of the intersection state.

Third, only one light group may be in a non-red state at one time. By making this assumption, the intersection state sequence can now be described as some permutation

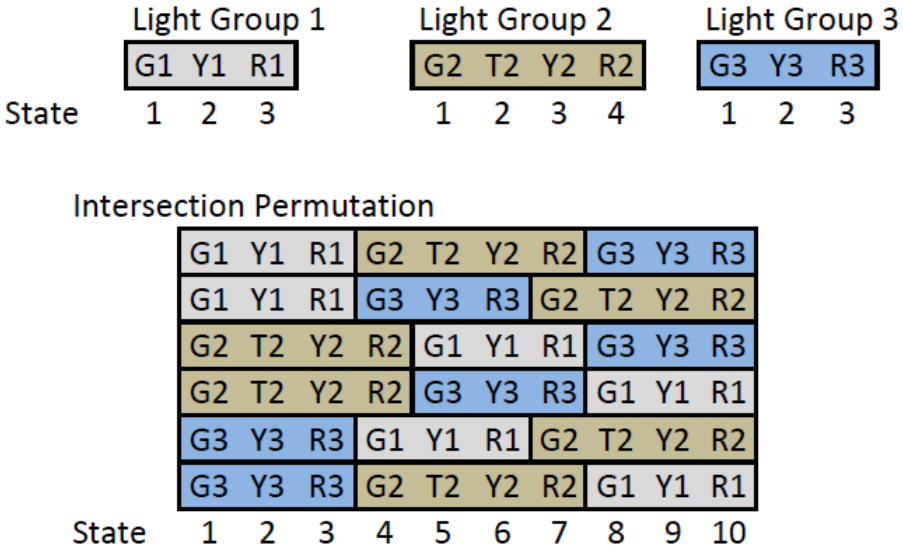


Figure 4.1: A sample of how intersection states are generated from light group states. In this example all possible permutation are display.

of the light state sequence. Figure 4.2.1 displays the permutations generated from a light state encoding.

Using these simplifications, an intersection state machine is constructed. Figure 4.2.1 illustrates the intersection level state machine in detail. The system comprises of a state machine for each light group and one intersection level machine. There exist an intersection state for each light group state machine. Only one light grouping state machine is in a non-red state at a given time. The currently active light group determines which state the intersection level state machine is in. The transitions that occur at the intersection level, depend on the light group state; a self transition for non-red states and a transition to new state when a red state is reached.

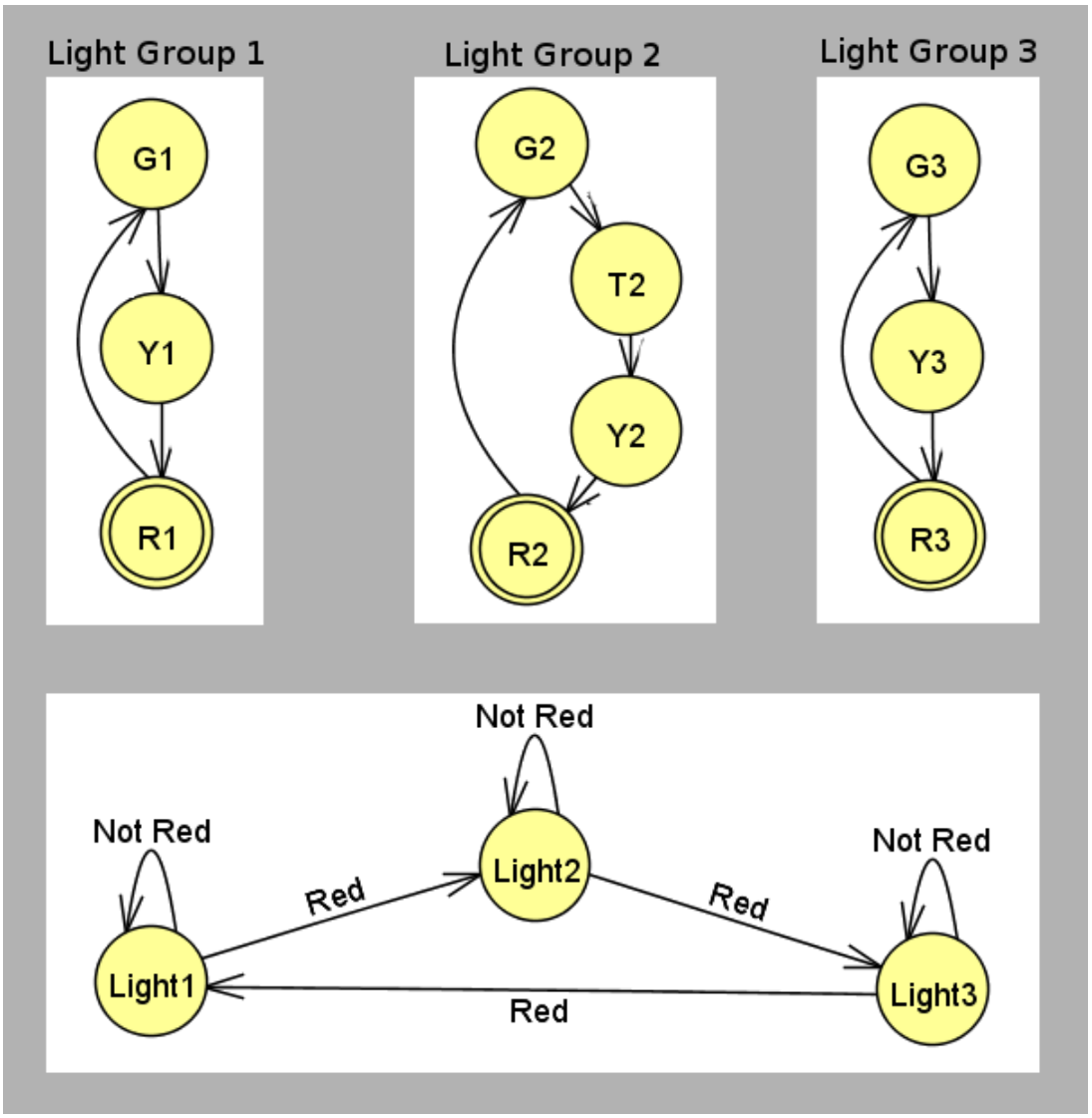


Figure 4.2: Intersection signaling system for an intersection with 3 light groups. The light groups have a variable number of states but always consist of a red_state = {R1,R2,R3} and a non_red_state = {G1,G2,G3,Y1,Y2,Y3,T2}. The bottom state machine transitions are determined by those of the light grouping above. When a red state is reached, the intersection level state machine may transition into a new state and correspondingly so may a new light group.

Run Time Application

Once the intersections are created, the database needs to be visualized. The visualization process is performed by the image generator(IG) using Vega Prime. At the time of this model's implementation, it was assumed that there was a direct correspondence between OpenFlight database structure and VegaPrime's internal representation, that all information within the database could be accessed and modified by VegaPrime API. Under these assumptions, VegaPrime application would access the myTraf node, and using the heuristics previously described, render the correct light states.

4.2.2 Implementation

Node Setup

Custom node types are generated using a set of API tools. The custom node generated within this thesis is named myTraf. Each myTraf node is meant to represent one traffic node. The nodes are programed in a OpenFlight specific programing language .dd. The specifics of the language are not important to the understanding of the node type, and so, to increase readability, the .dd code has benn translated to C++ for the reader's convenience.

```
class myTraf
{
public:
  myTraf();
  bool update_light_group(int index, int value);
  bool update_light_max(int index, int value);

private:
  int light_group[10];
```

```
  int light_max[10];
  int num_states;
  int num_groupes;
}
```

The OpenFlight generation process allows for additional specifications that are not fully covered by the above C++ description. The following are points that are missed by the above C++ header.

- Arrays do not exist within OpenFlight

To implement arrays in OpenFlight and conversion from arrays to a list of variables is performed. An array of length n is specified as n separate variables. That is the light_group[10] becomes light_group0,light_group1,...,light_group9. The same process is used for light_max[10]

- The value of light_group and light_max are between 0 and 11.

The OpenFlight framework allows for limitations to be placed on variable's min and max value. In this implementation, the range of the variables was limited. This minimized the myTraf node size in the OpenFlight database.

- The value of num_states and num_groupes are derived from those of light_group and light_max.

OpenFlight contains functions that are triggered when a value is changed. In the myTraf node alteration of the light_group or light_max triggers the update of num_states and num_groupes.

- Child and Parent relation definition

There were restrictions placed on the child relation. As described in section 2.1.1, OpenFlight nodes are connected to other nodes via a parent or child relation. In the case of the myTraf node, all parent relations are valid but the child relation is only valid between a myTraf and an external reference.

Organization & Interpretation

The myTraf node, as it is implemented, is only a list of integers. It requires organization and a specific interpretation to ensure proper functionality.

The current development process creates the database using Terra Vista. When traffic lights are generated using this process, they are inserted into the database as an external reference. The external references need to be assigned to an intersection. The child/parent relationship is used for this purpose. To assign a light to an intersection, an external reference is made the child of the myTraf node.

Next, the lights must be ordered into groups. The ordering process uses the light group number to imply grouping. Moving from left to right, across the myTraf node's children, the light groups are ordered increasingly. Starting from the first child, num_groups value determines how many of the children will belong to the first group. The second group children are listed to the right of the last child of the first group. The process continues inductively. The range for each grouping can be summarized by the following formula.

$sum(n-1) \leq range(n) \leq sum(n-1) + light\_group(n)$

where

- range(m) = the children that belong to m^{th} light grouping

- light_group(j) = the value stored in the j^th light_grouping field of the myTraf node
- sum(k) = $\sum_0^k light\_group(i)$

A final convention is used for naming light switches. The switches used to define the light states are given a particular naming convention. The naming can then be used to identify switches controlling light state from other switches used by the OpenFlight system.

Together the heuristics and the myTraf data are used to represent a intersection.

Run Time Application

The run time application component was only implemented in a rudimentary manner. An external reference node specifying a light was loaded in Vega Prime. The light's switches was found and used to change the light indicators states. A full implementation was not achieved since issues arose that lead to a redesign of the model.

4.2.3 Evaluation

The custom node model does not satisfy all the constraints presented in section 3.3. These violations arise from assumptions made about the Presagis software that did not hold true. Other issues arose from the difficulty of implementing this model across a large database system. Throughout this discussion, suggestions are made which will form the foundation for the next model, the vega prime intersection implementation.

Terra Vista is used to generate the database and models that populate the virtual environment. It was assumed that Terra Vista could be configured to insert custom

nodes into the database. It was also thought that external references could be attached and organized, as myTraf children, in the manner described in section 4.2.2. Both these assumptions are invalid. The process that generates intersections, within Terra Vista, is a black box. There are attributes that may be used to alter intersection appearance, however there exists no framework for altering the actual procedure used in the generation process. This suggests that insertion of myTraf nodes must be handled by a separate procedure or program. This violates **constraint I** where the traffic system must integrate into the current development process. Later, it was discovered that Terra Vista has no API for any form of programmable elements, and ultimately, any intersection signaling system must be generated outside of Terra Vista.

There exists a corner case where an intersection lies between two databases. Remember that large databases are composed of OpenFlight databases joined together, via external references in a master file, 2.3.1. Scenarios may arise where an intersection lies across multiple databases. More specifically, that the lights within the intersection are spread across two databases. Assigning lights to an intersection, by making them a child, is not possible. The myTraf node may only exist in one database while the set of lights exist across two. The myTraf convention system fails under these conditions. Clearly using child/parent relationship to group lights into an intersection is not sufficient to cover all scenarios. The solution, presented in the next section, stores the intersection and light grouping outside the database.

The myTraf node represents an intersection but requires lights to be assigned to it. At the time of intersection creation, in Terra Vista, it is known which lights belong to which intersection. The generation process is a black box and post database creation,

this information must be recomputed. The next intersection model must compute which external reference belong to which intersection, and more specifically, which light grouping.

In summary, the custom traffic node fails to provide the functionality required. The new model will not use TerraVista to generate intersection information but rather it will recompute which lights belong to which intersection.

4.3 Model 2: Vega Prime Intersection Control

Intersection organizational information must be moved outside of the OpenFlight database. The previous model failed under some scenariosCence. Additionally, the custom node could not be incorporated into the existing development process. The decision was made to move the intersection generation process from the front end, Terra Vista, to the back end, Vega Prime.

4.3.1 Design

Intersections are generated in Terra Vista when two roads intersect at a point. At the time of creation, the intersection's characteristics are known. This includes the intersection center, number of lanes, lane position as well as the lights belonging to the intersection. Unfortunately, there is no programming interface to access this information from Terra Vista. The result is that this information must be recomputed post database generation.

The task is therefore to gather or compute the necessary information to construct intersections. The minimum required information is,

1. Number of states in a light
2. Which lights belong to which intersection
3. Which lights are grouped together

A new model was designed to compute and gather this information. The new system runs as a programmable extension of Vega Prime. The application configuration consists of three steps. First, relevant data is gathered. Second, intersection organization is computed . Third, an application runs the intersection simulations.

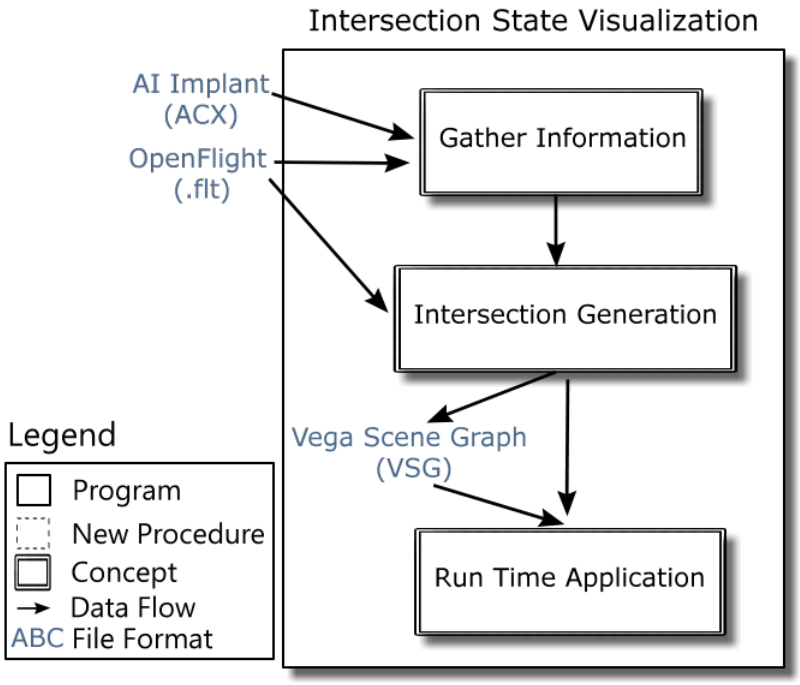


Figure 4.3: The organization of the intersection state visualization application. The system is broken down into programs, conceptual unit, information flow between units and file formats.

Intersection data must be reconstructed using available data. In model 2 there are three potential sources of data: the OpenFlight database, the AI.implant data and the vega scene graph(VSG) representation. Ideally, the VSG data would contain

all the required information for computing intersection info as it is already used to render the database, however, if insufficient information is present within the VSG, then other sources of information will be queried.

The VSG is constructed by the Vega Prime loader. The loader takes, as input, an OpenFlight database and with it creates an internal representation called the vega scene graph(VSG). Three potential pieces of information are retrievable from the VSG: light state, list of lights and light position. For light list and position, it is unclear from the documentation whether this information is retrievable from the VSG. The document does specify that VSG contains switches which operate in a similar manner to OpenFlight switches. The VSG will be used to locate the switches controlling the traffic light states. Based on preliminary research, the states stored in the switch in OpenFlight are maintained when converted to VSG switches. Therefore light states represented as OpenFlight switches in model 1 can be represented as VSG switches in model 2. In the Vega Prime implementation, the number of states in a light will be determined from the OpenFlight database using the Vega Prime loader and the VSG.

If VSG contains insufficient information, then the OpenFlight database and OpenFlight API will be used. With it, a list of all traffic lights can be gathered. Potential lights are identified from the list of external references. This assumption can be made since Terra Vista is used to generate the database and Terra Vista creates lights as external references. The list of external references can be narrowed down to only traffic lights using the external reference names. Additionally, the position of the light can be determined from the external references node. The list of lights and their respective positions are computed from OpenFlight database and stored for later use.

ACX Version1 Another potential source of information is the AI.Implant. At the time of this model's development, the AI implants data (ACX version 5.6) contained limited traffic system information. The information consisted of way points and road networks. A waypoint is a point along a path and a road network is a set of paths. It is clear when visualizing the AI data that some way points are intersection centers, however it is unclear how to determining whether a way point is an intersection center or another point along the path. For this reason, this version of the AI data was not used.

ACX Version2 During the design and implementation of model 2, a new version of the AI.Implant (version 5.7) was released. It contained significant updates, one of which is an intersection system. The new system detailed the center position of intersections, the incoming lanes and the outgoing lanes. This information is used in the Vega Prime intersection implementation.

Intersection Generation

Intersections are recomputed using the gathered information. Given that the techniques described in section 4.3.1 are successful, then the following information is available for computation

1. Light State
2. Light Position
3. List of All Lights
4. Intersection Center
5. Intersection Lane Location

Using this information an algorithm was devised to generate intersections. First, lights are assigned to intersections and second, intersection lights are grouped.

The list of intersections are created from the intersection center. For each intersection center retrieved from the ACX, an intersection is created. Following this step, lights are assigned to intersections. Light's position is compared to that of every intersection. The light is assigned to the intersection that it lies closes to. This procedure continues until all lights are assigned to an intersection.

The lights are then grouped. In some intersections, when lights are opposite each other, then they have identical states for all intersection states. The grouping procedure attempts to identify pairs of lights which meet this criteria, that is, lights that are opposite one another. The lanes that enter the intersection are used divide the intersection into areas. If two lights lie in opposite areas then they are grouped. Figure 4.3.1 illustrates the concept and the implementation section will have more details.

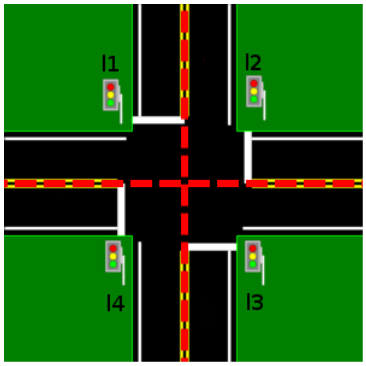


Figure 4.4: The division of an intersection into areas using lanes. Lights in opposite areas are grouped. Grouping1 = {l1,l3} and Grouping2 = {l2,l4}

With the completion of the intersection organisations, the essential information is in place for running an intersection visualization system. The number of states in a

light is retrieved from the VSG. The lights to intersection association is assigned using distance information. Light grouping is computed from light position and intersection lanes. The run-time system may now visualize and control the intersection system.

Run Time Application

The runtime application is responsible for visualizing and changing the state of traffic lights. The application is implemented in c++ using the Vega Prime API. In the previous model, a rudimentary implementation was achieved that could change a single traffic light's state. The new intersection signaling system must organize and store light and intersection data across several databases. This level of complexity requires organization and is achieved using OO objects. The objects have three levels of organization. At the lowest level are lights. Lights are associated with a visual element or model that represents a traffic light. It is responsible for storing its own state, changing it and interacting with the VSG. Intersections consist of several lights. It stores the intersection state and is responsible for storing the subjective mapping between intersection state and individual light states. At the top level is the traffic manager. This singleton stores all the intersections in the database. It can be programmed to trigger state changes and is responsible for any queries related to the state of the traffic system. The details of how these systems are implemented are left to the implementation section.

4.3.2 Implementation

This section details the techniques used to implement the concepts described in the design section. This includes the software modules used to gather information, the

Figure 4.5: A uml description of traffic system. Traffic Manager is composed of Intersections and Intersections are composed of Lights.

algorithm used to group lights and the structure of the run time application.

Information Gathering

The VegaPrime's internal representation is named the vega scene graph (VSG). The Vega Prime loader takes as input an OpenFlight database and with it creates the VSG. The conversion process has several variables to customize the level of optimization in the final version of VSG. The implementation attempted to use the Vega Prime loader to find traffic lights along with their position, identification and states.

The task of finding light switches within the VSG was relatively simple. The traffic lights used to generate intersections in Terra Vista were modified. The traffic light's switch nodes controlling state where assigned a static name. During conversion from OpenFlight to VSG, the naming for switches was kept in place. The VegaPrime API used the static name to search the VSG and rapidly identify switches controlling light states.

Determining light position using the VSG was problematic. The VSG is built for visualization and it discards any information not required for the rendering process. With traffic lights, polygon and texture information are kept but light position is discarded. The polygons are not grouped based on object, but rather by levels of detail or switch node. Determining which polygon belongs to the light, and additionally

whether it can be used to approximate the light position is a challenging problem. To avoid unnecessary computations, the OpenFlight database is used to retrieve exact light position instead of the VSG.

The OpenFlight API is used to search the database for external references with specified names. A list is created with external references containing names corresponding to traffic lights. Next, the program computes the position of the light using the database hierarchy. For more information on how position is calculated in OpenFlight, please refer to section 2.1.3. This information is kept and eventually passed to the run time application.

The ACX data was used to gather intersection and lane position. The procedure used the AI Implant API. The API's use was straight forward and no problems were encountered.

A secondary use of the information gathering system was the instantiation of the run time classes. For each intersection center found, an intersection object was created, and similarly, for each light position found, a light object is created. The two objects are associated together in the next section, intersection generation.

Intersection Generation

The first step in intersection generation was the assignment of lights to intersections. For each light position found a light was created, and for each intersection position, an intersection. The algorithm sorted lights into the closest intersection based on distance. The implementation details were relatively straight forward and do not warrant additional explanation.

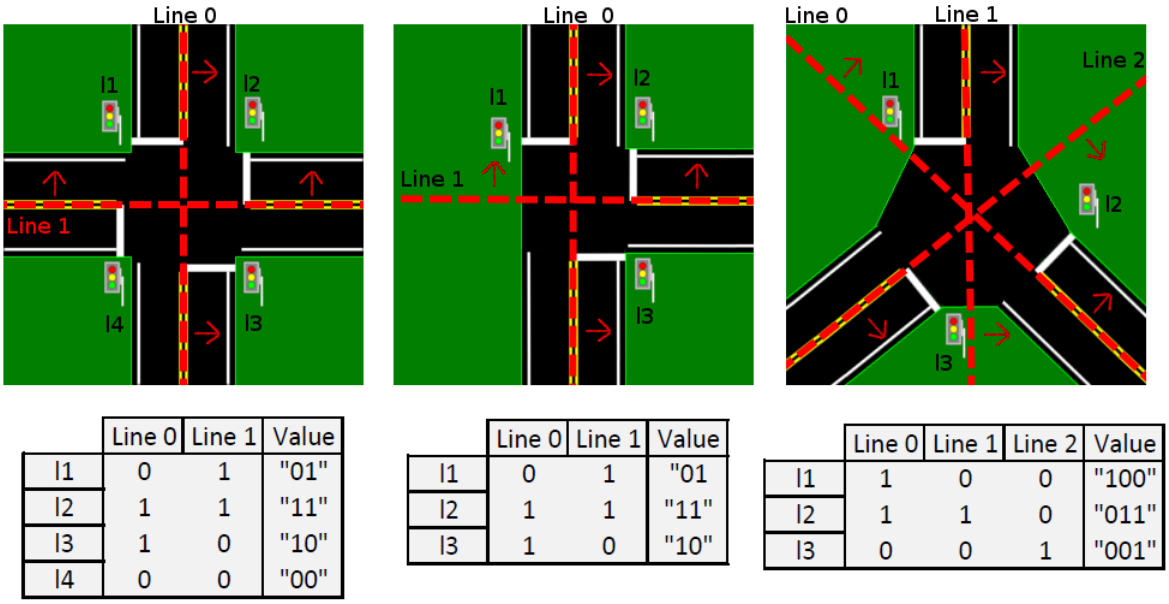
Following the assignment of lights to intersections, light grouping was performed.

The procedure divides intersection into areas, then groups lights based on which area they lie within. The division is created using lines. These lines are generated from two points: the lane positions and intersection center. Often, there are lanes opposite each other in an intersection that correspond to the same line or even lanes that are slightly non parallel. In these situations, having two lines is redundant and so one is removed based on a minimum difference value between their slopes. With intersection divided, the areas lying between lines are assigned numeric values. The values depend on which side of the line the areas fall. First, a left or right destination is arbitrarily assigned for each line. A value of 0 is assigned if the space is on the left and 1 if the space lies to the right. For a given light, it is assigned a value for each lines in the intersection. Each value is taken from the side of the line on which the light lies. Taken together, these value create a binary value. The lights are grouped if they have an identical binary value or if the values are complements. For a sample intersection and light encoding refer to figure 4.3.2.

Run Time Application

The run time application was implemented as C++ classes. The classes where designed to match the conceptual model of an intersection proposed in the design component. Each light controls its own state and intersection perform global state changes on groups of lights. With this division, changes can be made to each class without compromising the functionality of the entire system. The details of the programming are left out of this discussion as they are consistent with standard practices.

As discussed previously, the light switches and light position were found using the VSG and OpenFlight database respectively. Switches, within the VSG, have no



	Line 0	Line 1	Value
I1	0	1	"01"
I2	1	1	"11"
I3	1	0	"10"
I4	0	0	"00"

	Line 0	Line 1	Value
I1	0	1	"01
I2	1	1	"11"
I3	1	0	"10"

	Line 0	Line 1	Line 2	Value
I1	1	0	0	"100"
I2	1	1	0	"011"
I3	0	0	1	"001"

Figure 4.6: Grouping of lights based on intersection division. Intersection are divided into areas based on the lines generated from lanes. Each light is assigned a value based on which side of a line it lies. The values are then used to group lights

name and no position. The problem is that switches need to be associated with a light positions for the light to be sorted into intersections. One proposed solution involved finding the position of a polygon controlled by the switch and using it to imply position. The alternative is to assign an ID to switch node. This decision is not trivial and will be discussed in more detail in the conclusions. The fact is, the next model must be able to map a switch node to a position in the database.

Intersection class performs intersection organization and stores state information. It stores the lights belonging to the intersection and performs the light grouping procedure as described in figure 4.3.2.

The TrafficMgr class is responsible for high level control of the intersection state visualization. It was implemented as a singleton since there should only be one management system for the entire intersection signaling system. The API contains

functions which retrieve information from the OpenFlight, ACX and VSG. These functions trigger the instantiation of intersection and light class and consequentially the intersection organization and light grouping.

4.3.3 Evaluation

The failure of this model was the inability to match light switches, or the polygons representing a light, to a position in the database. The issue arose because of optimizations made by the Vega Prime loader that removed any unnecessary information for the rendering process. In this case the names and position of switch nodes were removed. Additionally, the run time system was incomplete in its implementation. No ID system was in place for lights or intersections bur rather, hard coded values were used instead. These design decisions were chosen in accordance with a short development cycle. We thought it more important to get a basic system in place to determine whether the design of the model had any major flaws rather than focus on a complete implementation. This enabled us to identify issues with the switch node at an early stage. In the end, the next model needs to create an identification system for lights and intersections, as well as resolve the issue of mapping switch nodes to positions.

4.4 Model 3: External Reference Labeling

The external reference labeling model is a solution to the issues encountered in model 2. In the previous model, it was not possible to match individual traffic light models with switch nodes that control their state. It was assumed that the node names,

present in the OpenFlight database, would also be present in the VSG representation. The names would then be used to match traffic light model data to state control data. This solution was not viable because the conversion process from OpenFlight to VSG removes randomly generated node names. This model proposes an alternative mechanism for labeling switch nodes. A custom node loading procedure will label VSG nodes as they are generated from the OpenFlight database. This approach will rely on the "subscriber" framework present in The VegaPrime API. As a secondary objective, this model will provide a uniform identification (ID) system for both lights and intersections. The ID system was only partially implemented in the previous model. This model completes the implementation.

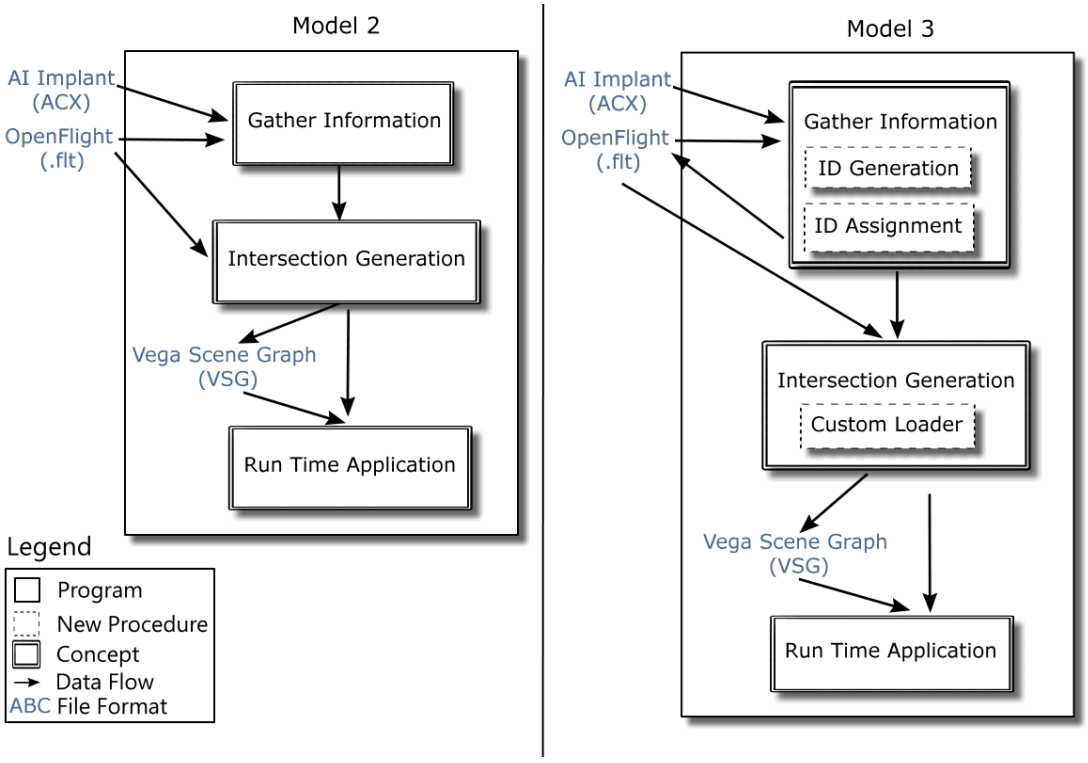


Figure 4.7: An overview of how the program structure has changed between model 2 and model 3.Model 3 introduces three new procedures: ID Generation, ID Assignment, and Custom Loader. The OpenFlight database is now modified.

4.4.1 Design

The Vega Prime loader is the class that loads OpenFlight databases, perform optimizations, and generates the VSG. In model 2, the loader was used to generate the visual environment using the OpenFlight database. In this model, the loader is modified in an attempt to label switch nodes as they are generated. This task uses the Vega Prime concept of subscribers. Subscribing is a process where function 1 wish to be triggered when function 2 performs a specific action called an event. In this case, the event of interest is the generation of a VSG switch node from an OpenFlight VSG node. When this event occurs, the VSG will be assigned an ID. Later this ID will be used to match the VSG switch to the OpenFlight data. In order to implement this functionality, 3 processes are required: one, a convention for generating ID; two, a procedure for assigning ID's to nodes; and three, a custom loader to read and assign IDs.

ID Generation

The ID generation procedure assigns values using property's of the OpenFlight database. The ID is a tuple comprised of a database component and a instance component. When a database is generated in Terra Vista, it is comprised of one master file consisting of a list of external references each referencing a OpenFlight database, 2.3.1. The sub databases are given names of the form flightX_Y.flt where X and Y are positive integers. In the ID generation process the X and Y identify the database component of the ID. The instance ID is assigned based on the database organization. The first traffic light is assigned an ID of 1, the second an ID of 2, and so forth. These two IDs are sufficient to uniquely identify any traffic light generated

using Terra Vista.

OpenFlight ID Assignment

The ID assignment refers to the process where ID are assigned to OpenFlight nodes and VSG switches. This section describes ID assignment for OpenFlight nodes exclusively. In model 1 , it was discovered that Terra Vista could not be used to assign values to custom nodes or attributes. The OpenFlight API is already being used to gather traffic light position and orientation. Model 3 will use the OpenFlight API to assign ID to traffic lights. Nodes that are assigned IDs may not have attributes in which to store the ID. In this case, the custom/extension framework will be used to create an additional field. For more information on custom nodes refer to section 4.1.2.

Custom Loader

The custom loader will be responsible for labeling switch nodes that control traffic light state. The loader will first recognize switch nodes and second, read ID information from the node's attribute. In order to trigger node labeling, the concept of subscribers in Vega Prime are used. Subscribers are classes that wish to receive notifications if events occur in another class. In this application, the event is the loading of an OpenFlight node and the notification triggers the labeling.

Switch nodes must be labeled, however, the database generation process creates lights as references to one light model. In order to store ID information at the level of switch nodes, there would need to be a distinct model for each light. This is clearly not a reasonable solution. As an alternative, labeling is performed at the level of the

external reference. In this way, the switch node may be labeled at generation time by determining parent external references.

Assuming that this procedure may be implemented, the custom ID attributes must be read from within the Vega Prime framework. Reading custom OpenFlight nodes using the Vega Prime loader is not part of the documentation. However, it is known that OpenFlight database nodes are written to memory sequentially, 2.1.2. Additionally, when a node is extended, custom attributes are appended to the existing node structure in memory. These facts will be used to retrieve IDs from external reference nodes.

4.4.2 Implementation

The implementation of the external reference labeling system had two interesting components. It used the Vega Prime and OpenFlight API to complete the intersection generation process. It reorganized some of the computation task from run time to non run time, however, other major problems were encountered.

ID Generation

The ID generation process assigns an ID to external reference nodes. The ID is comprised of two parts: the database and the object. The database ID is retrieved from the database's name. The database names are of the form flightX_Y where they represent two distinct positive integers. The implementation used a shell script to parse the nodes name and pass the X and Y value to the ID generation application at run time. The X and Y value are then combined into a single value using the following formula $ID\_Database = X * 1000 + Y$. The database ID is unique within

the range $1 \leq X, Y \leq 999$. This assumption is valid for our system since one, X,Y increase sequentially from 1 as database block size increases and two, the size of our world is 18X18 blocks and future scenarios are unlikely to increase beyond 200.

The object ID is generated from the database layout. The entire database is traversed. If an external reference is found that refers to a traffic light then it is assigned an object ID. The ID starts at 1 and increases for each identified light. The traversal is detailed in the OpenFlight ID assignment.

OpenFlight ID Assignment

External references require two new attributes to store database and object ID. These attributes were added using the OpenFlight node extension API, 4.1.2. The implementation procedure was similar to that in Model 1, 4.2.2, however, this extension only adds two new integers: database ID and object ID.

The OpenFlight API is used in Model 2 to gather information from the OpenFlight database. API is used to gather a list of traffic lights and determine their position. In model 3. the API is used to assign IDs to external reference nodes. The existing traversal is modified to include a check for external references that point to traffic light models. When these nodes are found, attributes are assigned values based on ID generation previously described.

Custom Loader

The custom loader is responsible for loading OpenFlight nodes and converting them to a VSG representation. The implementation is broken down into two major processes: triggering a procedure when an external reference is loaded and reading the reference's

attributes.

In Vega Prime, each file format is loaded and converted into VSG by a separate loader. The loader "vsNodeLoader_flt" is used to load the OpenFlight database. A subscriber function is used to tie into the loading process. Remember that a subscriber is a function that is triggered when an event occurs in the parent class. The function prototype is

```
void notify(  vsNodeLoader_flt::Event,
        const vsNodeLoader_flt* loader,
        const vsNode* node,
        const mgBead* bead,
        const fltAncillaries* ancil
)
```

The event used in the custom loader is EVENT_GEOMETRY. The event is triggered every time a geometry is created. More specifically, it is trigger when an OpenFlight node is used to create VSG node that can be rendered. External references specify models which contain geometry and therefore this event is triggered when an external reference is loaded. The function parameter "const vsNode* node" points to the VSG node that was generated and "const mgBead* bead" points to the binary structure of the OpenFlight node. In order to identify external reference nodes the binary structure of bead is decoded.

[2, p.44]

```
typedef struct {
  short        opcode; /* 2 */
  char         filler[ 214 ]; /* 214 */
  int database;
  int object;
} decodeStruct;
```

External Reference Record

Data type	Offset	Length	Description
Int	0	2	External Reference Opcode 63
Unsigned Int	2	2	Length - length of the record
Char	4	200	199-char ASCII path; 0 terminates Format of this string is: filename<node name> if <node name> absent, references entire file
Int	204	4	Reserved
Int	208	4	Flags (bits, from left to right) 0 = Color palette override 1 = Material palette override 2 = Texture and texture mapping palette override 3 = Line style palette override 4 = Sound palette override 5 = Light source palette override 6 = Light point palette override 7 = Shader palette override 8-31 = Spare
Int	212	2	View as bounding box 0 = View external reference normally 1 = View external reference as bounding box
Int	214	2	Reserved

Figure 4.8: The structure of an external reference node in memory. Note that the Opcode value for an external reference node is 63 and that the node is 214 bytes long. The extension attributes database and object would then be located at location 216 and 218 respectively.

When an event is triggered, a decodeStruct pointer is assigned to the bead variable. If the opcode value is 63 then the structure corresponds to a external reference node. Similarly, the database and object ID are retrieved from the bead using the decodeStruct.

Final Application

With these procedures implemented, the node matching procedure can be completed. The database ID, object ID and the VSG node pointer "vsNode* node" are stored in an associated list. Later the traffic light position and orientation data are matched using IDs. The VSG node using the vsNode pointer.

4.4.3 Evaluation

Model 3 was successfully implemented. Unlike the other models, there were no issues translating the design into the implementation, however, a major flaw was revealed during testing. Changing the state of a traffic light via a switch node changed all other traffic lights to the same state. For a given intersection with 4 traffic lights represented by 4 switch nodes, when one switch node state is changed to red then all other switches are changed to red. The exact reason for this behavior is still unknown, however, several hypothesis were considered.

The VSG is built specifically for rendering real time databases. It contains several optimizations that improve rendering performance. We believe that one of these optimizations resulted in simplified switch nodes. When rendering several external references that point to the same database, the resulting geometries are an identical set of polygons. We believe the optimization recognizes this pattern and automatically group these external references in a process called instancing,2.1.5. Remember, instancing is "the ability to define all or part of a database once, then reference it one or more times while applying various transformations" [3, p.14]. However, the intersection signaling system requires lights with independent states. A solution must be found that removes instancing.

With future reading of the documentation, it was discovered that the vpObject class object was the "fundamental database unit for rendering". The current application setup had the entire database loaded as a single vpObject. Perhaps the application has to be reorganized such that each object with state must be its own vpObject. With a simple experiment, it was shown that two external references, each loaded as a separate vpObject, could have independent states. The next model will

load each traffic light model as a separate vpObject.

Chapter 5

XML Traffic Description

5.1 Model 4: Independent Intersection Control

5.1.1 Design

This model was design to solve the issue discovered in model 3, changing any light state would result in the same change in all lights. The reasons for this behavior are not well understood, however it is known that traffic lights loaded as separate vpObjects are capable of having distinct states. Model 4 loads traffic lights as vpObjects. The loading system lead to some major structural changes in the program, including the division of run-time and non-run-time. Additionally, a watch dog program was added. The watch dog is a separate program that models the intersection system and attempts to identify invalid state transitions and request intersection state changes.

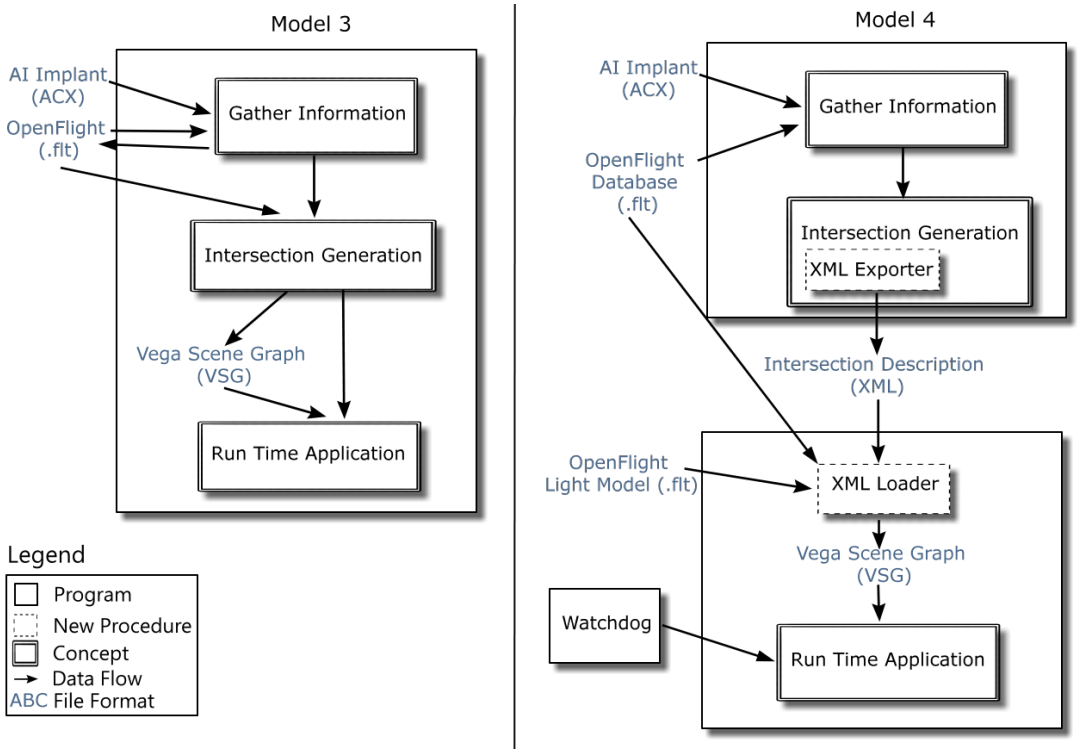


Figure 5.1: An overview of how the program structure has changed between model 3 and model 4. Model 4 introduces three new procedures: XML Exporter, XML Loader, and a Traffic Light Loader. Additionally, the program was divided into run time vs non run time components.

Lights as vpObjects

Two mechanisms were considered to create vpObjects from light switches. The first, involved creating vpObjects when nodes are loaded from the OpenFlight database. This approach involves a custom loader that triggers vpObjects creation when traffic lights are loaded. The implementation details would be similar to those of the previous model's custom loader. The second technique involves loading traffic after the database is loaded. In this process, the OpenFlight database is loaded and subsequently, the traffic lights are added. In this approach, additional information must be gathered in order to specify traffic lights orientation, however, it has the major

advantage of removing the Intersection Generation process's dependence on the VSG. Remember, in the previous models the VSG's switch nodes were matched to traffic light data in oder to create the run-time intersection system. This new implementation computes intersection information and then manipulates the VSG to meet the intersection's design. Both techniques are viable, however, solution two will be used.

Program Restructuring

Model 4 separates the intersection state signaling system into non-run-time(front end) and run-time(back end) programs. This division requires a medium of passing information between the two programs. The decided upon medium is XML. Although there may be more efficient ways of passing information, we believed that XML is a convenient way of organizing our information while providing enough flexibility for future development. Additionally, there exists several APIs that facilitate the reading of XML databases.

Several changes must be made in order to divide the program into front end and back end. More specifically, an export/import procedure must be developed. In addition, the intersection generation procedure must be modified. In previous versions, the intersection generation outputs were class objects that were used by the run-time system. In model 4, the intersection generation process will export to XML. The XML will be read by the XML Loader that will generate intersection visualization objects used by the run-time system.

The XML loader takes as input an XML intersection description and with it, creates the objects needed by the run time system. The XML loader will load the world database, generate run-time objects from XML description and append traffic

lights to the VSG.

XML Encoding

The XML data encompasses all information related to the intersection indicator system. Information referring to intersections is stored at the intersection level and information regarding lights at the light level.

Intersection will store

- ID
- Position
- Traffic Lights

Traffic lights will store

- ID
- Position
- Orientation
- Group
- Model Type

ID for intersection come from ACX, while for lights, they are generated from the database naming using the procedure in section 4.4.1. Position will contain an X,Y,Z coordinate and will be relative to database center. Orientation, is the direction the traffic light is facing. Group, is a number paring lights using a mechanism described in section 4.3.1. Model type, stores the database name used to load the traffic light model.

Watch Dog

The final component in model 4 is a watch dog program. The watch dog is a separate program designed to setup, manage and test the intersection signaling systems. The program will be responsible for communicating and interacting with Stage, VegaPrime and the AI Implant. The system consists of three components: intersection modeling, intersection management and intersection validation. Finite state machine(FSM), will be used to model intersection states. The conceptual design of the intersection is a FSM, so implementing the watch dog as an FSM is a natural conclusion. The FSMs will be defined using the formal 5-tuple definition (Q,Σ, δ, q_0, F), although accept state and start state will not be required in the implementation.

- finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- a start state ($q_0 \in Q$)
- a set of accept states ($F \subseteq Q$)

[7]

In addition to the intersection modeling, the watchdog manages the intersection state signaling system. The watchdog will be responsible for interacting with the run-time system. It will periodically request state changes in order to simulate realistic road conditions. Valid states and transition will be defined using the FSM definition Q and $\delta : Q \times \Sigma \rightarrow Q$.

Intersection state sequences eventually repeat. The watchdog will have built in facilities for programing repeating state transitions. Intersection's transition will be specified using a period, start state, next state, offset and condition. The period is the

time frame before the counter resets to zero. The offset is the point in time in which the transition will occur. This value must be less than the period. The condition is a guard that determines whether the transition is valid. Note, the definition of multiple repeat transition is a scheduling problem. To avoid conflicts, the watchdog will detect scheduling errors arising from simultaneous events transitions.

The final component of the watchdog system is intersection validation. The scheduler already performs validation by preventing simultaneous transitions. Other forms of validation will include property checks such as safety and liveness. Computing these properties from FSMs is a well documented algorithm.

5.1.2 Implementation

Lights as vpObjects

Traffic lights are now added to the VSG after the terrain is loaded. The Vega Prime paging system is used to add objects to the VSG at runtime. The paging system loads or removes objects when system resources become available. The use of this system is relatively straight forward and will not be detailed further. Once a vpObject is created using the traffic light model, the subtree of the object is searched for a switch node representing light state control. The switch node is then associated, via a pointer, with a Light object used in the run-time application.

Program Restructuring

The intersection state visualization system is now composed of two components. The front end non-run-time system and the back end run-time system. The restructuring

requires an intermediate medium for communication, in this case XML, and procedures to both import and export this information.

The export procedure converts Intersection Generation information into XML data. For every intersection and traffic light, a corresponding procedure is used to write XML data to a file. The formating follows the encoding described in section 5.1.2.

The import procedure reads XML information and creates corresponding intersection and traffic lights run-time objects. The XML file processing is performed using tinyXML. TinyXML is a simple and small C++ XML parser. For every XML intersection read, an intersection constructor class is called with the relevant information. For every XML light read, a corresponding light class is constructed. A reference to the light class is stored in an intersection object. Once the entire XML system is read, the run-time system can perform state simulation in the manner described in figure 4.2.1 and 4.2.1.

XML Encoding

The XML encoding translates intersection and light information into an XML representation. The translation process can be seen as a mapping between intersection information and XML representation. The direct translation follows.

Intersections are stored using the tag <intersection>. The information describing the intersections are separate child tags or attributes.

- ID is an attribute of interection <intersection ID = ””>
- Position uses a tag <position>with respective x,y,z components as children <x>,<y>,<z>

Traffic lights are children of intersections and the information they express are child tags and attributes. The light tag is <ID>.

- Traffic light ID is stored as attribute num <ID num = "">
- Position uses a tag <position>with respective x,y,z components as children <x>,<y>,<z>
- Orientation is stored as tag <Heading>
- Group is stored as tag <group>
- Model Type is stored as tag <Name>

A complete XML representation of an intersection can be seen in figure 5.1.2

```
<TRAF_DATA>
  <intersection ID ="565">
  <position>
    <X>156.042007</X>
    <Y>551.028992</Y>
    <Z>0.000000</Z>
  </position>
  <ID num = "514">
    <group>0</group>
    <position>
      <X>169.843842</X>
      <Y>551.361328</Y>
      <Z>0.000000</Z>
    <position>
    <Heading>240.913345</Heading>
    <Name>signal01_01.flt</Name>
  </ID>
</TRAF_DATA>
```

Figure 5.2: A sample XML encoding. This example contains one intersection with ID 565 and one light with ID 514.

Watch Dog

The watch dog is a separate program designed to setup, manage and test the intersection signaling systems. The implementation consists of, FSM definition, state change requests, scheduling and property checks.

The formal definition of a FSM is (Q,Σ, δ, q_0, F). For more information on each component refer to the design component, 5.1.1. State is defined as an integer. When a new state is defined, the given intersection is checked to see whether it contains that same state. If it does, the new state is invalid and if it does not, the new state is add to the intersection possible states Q. A transition function is ($\delta : Q \times \Sigma \rightarrow Q$). In the implementation, the transition function is defined as a list of transition. A transition is represented as a class. The class contains a variable to store its current state. Next, it contains a map. A map is an implementation associative list. The first value(key) defines the states being transitioned to. The second value is a function pointer, Condition. Condition is a pointer to a function that validates the transition. It is a guard that determines whether that transition is currently valid. Finally, a multimap is used to define the transition function. A multimap is an associative list where the key may correspond to more than one value. In other words, the multimap is a non-injective function. The multimap is used so that a given state may have more than one valid transition. Most intersection have state sequencess that repeat. However, there are scenarios with emergency vehicles where light must be able to quickly transition to red or green light. The multi map allows for these scenarios to be modeled by allowing alternative state sequences.

```
// Function pointer for condition required for transition
typedef bool (*Condition)();
```

```
class Transition
{
public:
  int current_state;  // instial state for transition
  map<int, Condition> NEXTSTATE_Condition;
};

multimap<int, Transition*> TransitionFunction; // the
    transition function
```

The scheduling services are responsible for creating periodic state transitions. As described in the design process, the requested transitions are specified using period, start state, next state, offset and condition. This information is stored as a c style struct. When a new periodic event is requested, a computation is performed to determine whether a conflict will arise between the new event and existing events. A conflict is defined as a time where two transitions are valid at the same time.

The final validation step is providing safety or liveness. These features were not implemented since implementation issues arose.

The watchdog was integrated with the rest of the simulation system. In addition to its interaction with the intersection state signaling system, the watch dog is responsible for sending state updates to the AI Implant. The AI Implant has its own internal intersection state representation that it uses for pedestrian and vehicle AI computations. It was assumed that the AI Implant API could be used to modify the AI's intersection state setup to correspond with this system, however, custom intersection states are not possible in its current implementation. This violates **Constraint I** that states that the final product must integrate into the current development process.

5.1.3 Evaluation

Model 4 was successful at resolving the light state issue encountered in the previous model. The separation of the system, into front end and back end, resulted in two beneficial properties. One, is reduced load time. Often the application is run several times in succession in order to test functionality. With the new program division, the data gathering and intersection generation do not have to be repeated. The second beneficial property is facilitated intersection design. The XML traffic description is a reasonable interface for allowing non-technical users to modify the traffic system. **Constraint V** states that a user must be able to program their own traffic intersection. Modification of the XML directly seems like a reasonable mechanism. In future developments, an application could be generated that manipulates the XML description directly for the user.

Some issues arose while integrating the intersection state signaling system with the rest of the simulation system. Model 4' implementation worked correctly with Stage and Vega Prime however there where issues with the AI Implant. The issue was that the AI data could not be modified so that it had identical states as those in the intersection state signaling system. This is clearly an issue. A driving simulation without AI controlled cars is not a very good simulation. The issue of how to synchronize intersection state signaling system and the AI implants will be addressed in Model 5.

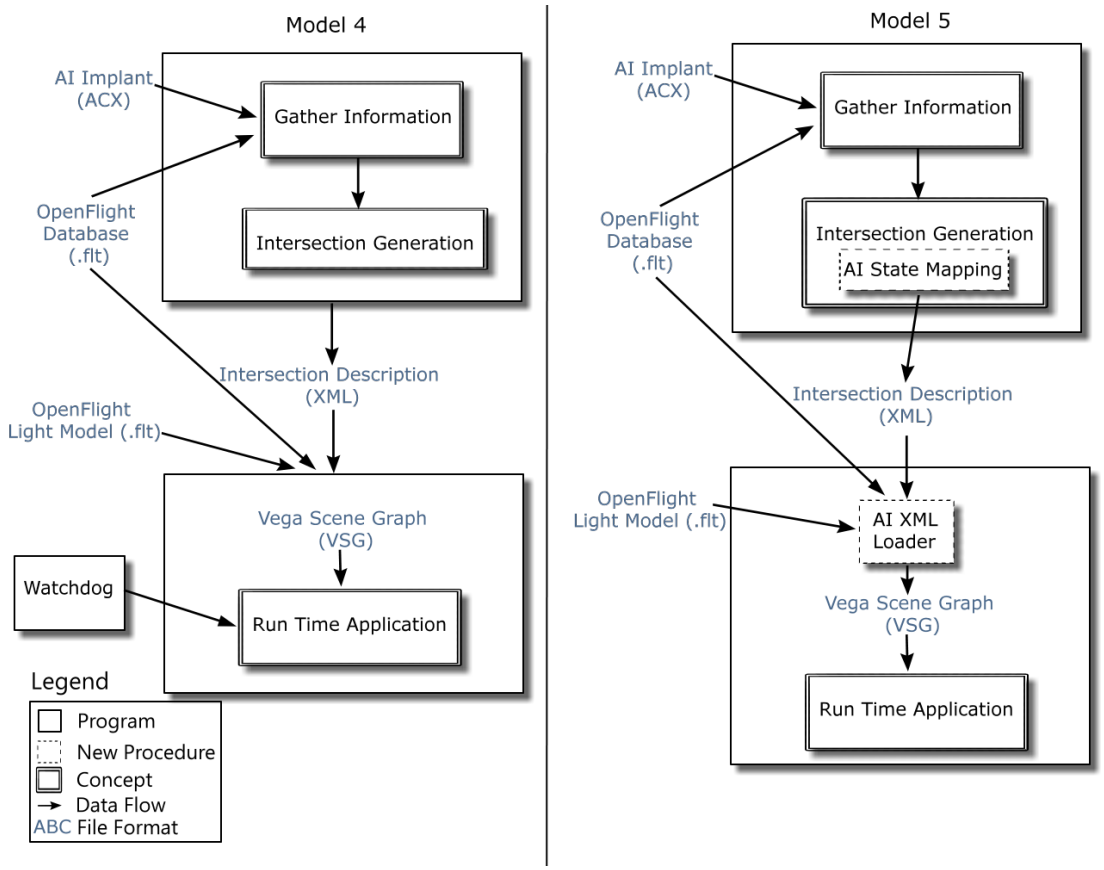


Figure 5.3: An overview of how the program structure has changed between model 4 and model 5. Model 5 introduces two new procedures: AI State Mapping, and AI XML Loader. The watchdog was removed since the intersection state is now determined by the AI.

5.2 Model 5: AI Intersection Control

5.2.1 Design

In Model 4, issues arose while integrating the intersection state visualizer with the rest of the simulation system,3.1. The AI Implant's intersection state representation could not be made to correspond with that of the intersection signaling system. This problem resulted in two violated constraints. **Constraint IV**, the system must be

able to model all possible traffic light systems used in the real world. The AI Implant has two intersection modes. In mode 1, intersection states are a permutation of each light's individual state sequence. The intersection's state is generated in a similar process to those described in model 2's implementation, 4.2.1. In mode 2, lights opposite one another have the same states. These two modes model quite a few real world scenarios, however, the major problem is that the sequence of light states cannot be modified. Every light goes through the same sequence based on its mode and cannot be modified. For instance, in mode 2 the light sequence is red, green, yellow and this sequence cannot be modified. Clearly this sequence does not represent all possible traffic lights and therefore **Constraint IV** is violated. **Constraint V** states that a user must be able to program their own traffic intersection. Again, the AI Implant light states sequence are limited to those of the two modes and cannot be modified. Consequently, all intersection designs will not be possible for the user and **Constraint V** is violated.

The complete solution proposed in the initial system design is not achievable. The source of constraint violates arise from factors beyond the design scope, however, an intersection state signaling system is still needed and must be completed. Model 5 proposes a new solution that will create a functional solution, although limited in the number of intersection systems it may model. The objective is a solution that may be rapidly developed into a complete solution once the AI Implant issues are rectified. Three methods were considered while developing model 5: one, wait for new AI implant solution; two, modify the AI Implant; three, adapt to AI Implant.

The first solution involved waiting for a new AI Implant release. New releases are relatively frequent but the immediate need for a traffic visualization system is still

present. As well, the writer of this thesis may not be present at the time of the AI's release. Modification of the visualization system would be easier if performed by its developer.

The second solution involved using the AI Implant API to modify the AI Implant directly. The solution would involve modifying the AI Implant in order to allow for a more customized intersection representation. The major problem with this approach is time. The AI Implant is a powerful tool that allows for the creating of custom AI. The power of the tool also comes with a steep learning curve. Becoming familiar with the AI API would take a significant amount of time and the objective, intersection state modification, may be unachievable. Additionally, when the AI Implant update is released, the intersection modification is unlikely to be compatible with the new system. In this scenario, the custom solution would be discarded. There is no guarantee of comparability between the two system and the implementation would be time consuming, For these reasons, solution two was not implemented.

The third solution involves adapting the intersection signaling system to AI Implant. In this solution, the steep learning curve associated with learning a new API can be mitigated by mapping classes. In solution three, the intersection state would no longer be programmed by the user, but rather, they would be computed from the AI Implant's internal state representation. A process would be developed that reads the AI Implant data, determines each light's state for a given intersection state, and then exports this information to an XML representation. The solution will be implemented in such a way that cuts down on development time post AI Implant release. The solution focuses on mapping each class in the AI's intersection state, to a particular line of XML code. Using this approach, classes modified in the update

are the ones needing update in the XML mapping process.

The intersection state visualization system will be modified to take state information from the AI Implant. The third solution, adapt to AI Implant, was chosen since it had short development time and for its compatible with eventual AI update. The solution is broken down into 3 steps: AI state decoding, XML representation, run time system modification. The first step is retrieve AI state information from the AI Implant. Second the AI information is exported to an XML representation. Third the XML information is read and used to modify the intersection and light setup in the run time system.

Front End Modification

The AI's state setup will be used to configure the intersection visualization's state. The exact specification will be retrieved from the ACX using the AI Implant API. In the AI, each light is described as a series of indicators. Each indicator consists of a color, a direction and a status. This information will be gathered and then exported to an XML file.

In the state visualizer program, the light IDs are generated using the algorithm described in section 4.4.1. For the sake of consistency, the intersection visualization system will now use the AI's ID system for lights. The AI's cell IDs are unique for a given map and therefore a valid unique identifier.

Intersection Generation will be simplified. In previous versions, Intersection Generation was a process where lights were grouped and intersection states where computed,4.3.2. Now, that information is specified from the AI data. Intersection Generation now consists of matching traffic light data to intersection data. The matching

process consisted of using traffic light position and intersection center to determine which light models belonged to which intersection. In the new implementation, the lane position will be used instead of intersection center, to match traffic light to AI data.

AI State Mapping

The AI's state representation will be mapped to an XML description. The mapping is intended to be adaptable to modifications made to the AI's intersection. In order to do so, each AI intersection class will have its own corresponding structure within the XML file. The AI Implant chooses to describe the traffic light state as a series of indicator. Remember, an indicator is a single light source that is described as either on or off. The information used to represent traffic light state in the AI is

- State Number is an a positive integer used to distinguishes one intersection state from another. $state\_number = \mathbb{Z}_{\geq 0}$
- Color describes the color of the indicator. $color = \{red, yellow, green\}$
- Arrows can be used to specify a direction to a particular indicator $arrow = \{\emptyset, left, right, straight\}$
- Status is used to indicator whether a light is on or off. $status = \{on, off, flashing\}$

This information will be used to configure lights states in the visualization system.

Run Time Modification

When an intersection is created in Terra Vista, there is no correspondence between the AI and the database's light state. More specifically, the switches representing light state in the traffic light model do not correspond with the states generated in

the AI. For synchronization to occur, the VSG switch and the AI's must be equal. In Model 5, the VSG switch will be modified using the VegaPrime API. The intersection description in the XML will be used to create a VSG switch that corresponds to visual representation present in the AI.

5.2.2 Implementation

Front End Modification

The AI state information was retrieved using the AI Implant API. The intersection information is dependent upon the intersection state and each piece of information must be requested individually. Additionally, the individual lights that compose the intersection must be accessed using their ID. The data gathering process is a series of nested loops. The procedure is described in the following pseudo code:

```
for each intersection
  for each light
    for each intersection state
      for each indicator
        get indicator status
        get indicator arrow
        get indicator color
      end
    end
  end
end
```

Once data gathering is complete, a separate procedure parses and writes the information to the XML file. The exact formating will be described in the next section 5.2.2.

For backwards compatibility, the AI state description and the OpenFlight information were kept separate within the XML. As well, the light grouping entry is no longer necessary since it is replaced by the AI encoding. Therefore, a default value of 0 is assigned to the group field.

Lights now have an ID derived from the AI cell ID. To match the OpenFlight data to cell ID, the position of the traffic light is compared to that of the cell midpoint. The closest cell's ID is assigned to the OpenFlight data. In TerraVista, traffic lights are placed next to the lanes to which their state refers. The matching problem consists of finding the closest lane to a traffic light and using its ID.

With these minor details implemented, the mapping of AI information to XML description can begin.

AI State Mapping

The AI information ID, Color, State, Arrows, and Status must be translated to an XML representation. The question is what information must be written to the XML presentation. Light ID is used to identify traffic lights and must be written. State is the underlining factor controlling changes in intersection representation and is therefore written. Status determines whether a light is on or not. Color and Arrow are properties describing an indicator. For a given light, they could potentially be specified once, however, the AI changes the values of arrow and color in some situations. The reason for this change is unknown. In figure 5.2.2, color changes from green to yellow in state 3 and 4 respectively. The value of arrow and color vary across states therefore must be specified for each state.

The mapping used in this implementation is relatively simple. All AI information

is setup as a single field called <Indicator>. Status is written as a element since we believe that the onoff state of an indicator is its most important property. When other properties are found that describe the indicator, they are added as attributes. Therefore, color and arrow attribute of the field <Indicator>. The following is a sample intersection containing a single light.

```
<intersection ID ="16">
  <position>
    <X>300.159546</X>
    <Y>162.303375</Y>
    <Z>0.000000</Z>
  </position>
  <ID num = "20">
    <group>0</group>
    <position>
      <X>295.891327</X>
      <Y>155.285065</Y>
      <Z>0.000000</Z>
    </position>
    <Heading>89.653481</Heading>
    <Name>signal05_01.flt</Name>
  </ID>
  <TraffidMap>
    <InCell ID= "186">
      <State num= "0">
        <Indicator col= "Green" arr= "Left">Off</Indicator>
        <Indicator col= "Red" arr= "None">On</Indicator>
        <Indicator col= "Yellow" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "Right">Off</Indicator>
      </State>
      <State num= "1">
        <Indicator col= "Green" arr= "Left">Off</Indicator>
        <Indicator col= "Red" arr= "None">On</Indicator>
        <Indicator col= "Yellow" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "Right">Off</Indicator>
```

```
      </State>
      <State num= "2">
        <Indicator col= "Green" arr= "Left">On</Indicator>
        <Indicator col= "Red" arr= "None">Off</Indicator>
        <Indicator col= "Yellow" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "None">On</Indicator>
        <Indicator col= "Green" arr= "Right">On</Indicator>
      </State>
      <State num= "3">
        <Indicator col= "Yellow" arr= "Left">On</Indicator>
        <Indicator col= "Red" arr= "None">Off</Indicator>
        <Indicator col= "Yellow" arr= "None">On</Indicator>
        <Indicator col= "Green" arr= "None">Off</Indicator>
        <Indicator col= "Yellow" arr= "Right">On</Indicator>
      </State>
      <State num= "4">
        <Indicator col= "Green" arr= "Left">Off</Indicator>
        <Indicator col= "Red" arr= "None">On</Indicator>
        <Indicator col= "Yellow" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "Right">Off</Indicator>
      </State>
      <State num= "5">
        <Indicator col= "Green" arr= "Left">Off</Indicator>
        <Indicator col= "Red" arr= "None">On</Indicator>
        <Indicator col= "Yellow" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "None">Off</Indicator>
        <Indicator col= "Green" arr= "Right">Off</Indicator>
      </State>
  </TraffidMap>
</intersection>
```

Figure 5.4: Sample encoding for an intersection with one light. This example contains a light with 3 direction and 6 states. The light contains 5 indicators : Green, Yellow, Red, Left Arrow and Right Arrow.

Run Time Modification

The run time system must be modified to read the new XML encoding. The existing frame work established in model 4 allowed the program to read XML data. This same framework is used to parse the new XML encoding. The AI information is then used to reprogram VSG switch node. In order to facilitate the process, a new OpenFlight traffic light model was designed.

The OpenFlight switch node contains the following indicator = { Red, Yellow, Green, LeftRed, LeftYellow, LeftGreen, RightRed, RightYellow, RightGreen, StraightRed, StraightYellow, StraightGreen } . When the OpenFlight light switch is converted to VSG switch, the left to right ordering of the switch's mask will correspond with the ordering presented in indicator. For every intersection state and every light, a new mask is created. The arrow and color value are used to determine to which bit the status value refers. If the status value is on or flashing, the mask values are assigned a value of 1. If no value is assigned then the default value is set to 0. A sample of the translation process is described in the following figure 5.2.2.

	R	Y	G	LR	LY	LG	RR	RY	RG	SR	SY	SG
Mask 0	1	0	0	0	0	0	0	0	0	0	0	0
Mask 1	1	0	0	0	0	0	0	0	0	0	0	0
Mask 2	0	0	1	0	0	1	0	0	1	0	0	0
Mask 3	0	1	0	0	1	0	0	1	0	0	0	0
Mask 4	1	0	0	0	0	0	0	0	0	0	0	0
Mask 5	1	0	0	0	0	0	0	0	0	0	0	0

Figure 5.5: This figure demonstrates the masks generated from the AI XML data. This particular diagram refer to light 186, specified in figure 5.2.2. In the top column, R = red, Y = yellow, G = green, L = left, RR = Right Red, and S = Straight.

With the new masks in place, the intersection visualization system is now in correspondence with the AI Implant.

5.2.3 Evaluation

The AI intersection control model was successfully implemented. There were some issues in understanding the AI Implants intersection representation but these problems were overcome using testing. The current implementation is functional but there are still improvements to be made.

The current XML representation for state information is rather large. Optimization could be performed to reduce the size of the XML file. When comparing several intersection across a large database, it is obvious that there are several repeated patterns in terms of intersection encoding. For instance, several intersections consist of 4 lights with each light containing 3 indicator that are green, yellow, red. This information could be detailed once, then light containing the same encoding could point to that information.

The XML file contains a fair amount of information and should be reorganized to meet the conceptual model of an intersection. For instance, information from OpenFlight database, such as orientation, and information from the AI, such as state, should both be under a <light>label. This would improve the readability of the XML and prevent a certain amount of duplicate information already present in the current format.

The light models used in the visualization are those generated by Terra Vista. When the AI information is generated in Terra Vista, different intersections are created based on the number of lanes in an intersection and the angle between them.

The results are AI intersection with different number of states and often different indicators. In future development, it would be convenient if the intersection state visualizer could select a light model that corresponds with the AI intersection. When an AI intersection contains green and yellow left turn arrows then a light model would be loaded that contains such properties.

Pedestrian crossing signals are an important part of intersection that was not discussed in this thesis. A pedestrian system was partially implemented but issues arose with understanding the AI pedestrian system. The issues were never resolved so the discussion was left out. Future models should incorporate a pedestrian intersection system.

Chapter 6

Conclusion

An intersection state visualization system was successfully designed, implemented and incorporated into the McMaster motion simulator. There were major differences between the initial design and final product. This was not necessarily a bad outcome as the final product included several optimizations not foreseen in the initial design. Some constraints were not satisfied in the final solution. **Constraint IV**, the system must be able to model all possible traffic light system used in the real world. This constraint was violated due to the AI Implant implementation. Besides this failing, the overall driving system is still successful since most real world intersection can be modeled by the AIs two modes. Outside the scope of the application, there were several lessons learned during development.

Several assumptions were made that lead to implementation issues, the first being inter-system compatibility. OpenFlight databases are used and generated by several Presagis applications. Additionally, there exist tools for the modification and extension of the OpenFlight standard. It was assumed that modification to the OpenFlight standard would be compatible across all Presagis products. This was not the case. A

simple fix would be the development of an inter-system compatibility guide. It would detail which features are common between products vs. those with highly significant differences.

In the content creation, Terra Vista is often the first program used to generate large scale databases. It contains tools for the automated generation of roads, intersection and AI information. Once the database is created, this information is discarded. Ideally, an API would be present to modify the tools used in Terra Vista and perhaps even access information used internally in the generation process. In this way, information such traffic light position, orientation and even intersection states would not have to be recomputed. A TerraVista API would have greatly facilitated the development of this thesis.

OpenFlight is an open standard used for real time application. VegaPrime's own representation, VSG, is a similar product that is optimized for rendering performance. Typically, OpenFlight databases are converted to a VSG format and used in VegaPrime applications, however, the exact relationship between OpenFlight and VSG is not well documented. In this thesis, models 2 and 3 encountered issues during implementation due to a lack of understanding of the VSG. A detailed description of the relationship between OpenFlight and VSG would help to avoid these issues in future development.

The Intersection state visualization system is functional but not yet optimal. In future designs, pedestrian lights and custom intersection development should be developed. The current pedestrian traffic states are tied to the state of the vehicle lights. More specifically, the switch that controls the state of vehicle lights also controls that of pedestrians. When the vehicle light is green, the walk signal is green; when the

light is yellow, the pedestrian light is a caution; and when the vehicle light is red then the walk signal is red. In future works, the pedestrian lights should be independent of the vehicle lights. Another ideal feature would be a separate system for the generation of custom intersections. The program would allow the user to create a traffic light system using a user interface. The design could then be assigned to intersections in Terra Vista or the XML representation. This would greatly facilitate the rapid development of traffic simulations and driving experiments.

Appendix A

Rendering Specific Concepts

A.1 Double buffer

Double buffering is a process used to remove rendering artifacts resulting from drawing data directly to the display. In the process, two buffers are used to display and draw images. One buffer displays an image while the other draws images that have not been culled. When the frame expires and a new frame is requested the buffers swap. The buffer that was drawn in the previous frame is displayed and the previous display buffer is written to draw the new frame. This process prevents visual artifacts that occur when drawing images directly to the display.

A.2 Z-fighting

Z-fighting is a rendering artifact that occurs when two rendered geometries have the same value in the z-buffer. The artifact is typically a blend of the two surfaces or is displayed as a rapid flickering between the two. A major source of z-fighting

are coplanar faces. When two faces are coplanar and overlap, then the entire overlap section will have rendering artifacts. This is because the image have the same z-buffer value.

A.3 Bounding Volume

A bounding volume is a geometry that encompasses another object or set of objects. In Openflight, the bounding volume is described as a box, sphere, cylinder, or convex hull. Typically the volume is minimal, meaning that it is the smallest volume bounding that encompass the object fully. With several objects, the bounding volume encompasses the union of the object's volumes.

Bibliography

[1] (2010a). *OpenFlight API User Guide Volume 1*. Presagis, 4.2 edition.

[2] (2010b). *OpenFlight API User Guide Volume 2*. Presagis, 4.2 edition.

[3] (2011). *OpenFlight Scene Description Database Specification*. Presagis USA, 16.4 edition.

[4] Budianto (2009). Refactoring autosim for program comprehension.

[5] Hill, F. S. J. (1994). *The Pleasures of 'Perp Dot' Products in Ch. II.5 in Graphics Gems IV (Ed. P. S. Heckbert)*. San Diego: Academic Press, San Diego.

[6] http://dictionary.reference.com/browse/simulator (2012). Simulator definition.

[7] http://en.wikipedia.org/wiki/Deterministic_finite_automaton (2012). Fsm @ONLINE.

[8] http://www.graphics.cornell.edu/online/tutorial/ (1998). What is computer graphics? @ONLINE.

[9] http://www.presagis.com/files/standards/OpenFlight9.0.pdf (1990). Multigen @ONLINE.

[10] http://www.presagis.com/products_services/standards/openflight/ (1998). Openflight @ONLINE.

[11] Klauer, S., Dingus, T. A., Neale, V. L., and Sudweeks, J. (2006). The impact of driver inattention on near-crash/creash risk.

[12] Strause, S. H. (2005). New, improved, comprehensive, and automated driver's license test and vision screening system.

[13] Wachtel, J. (1995). Brief history of driving simulators.

[14] Yoshimoto, K. and Suetomo, T. (2008). The history of research and development of driving simulator in japan.