

Large Scale Traffic Simulation on Graphics Hardware

**A THESIS SUBMITTED TO THE GRADUATE SCHOOL OF  
THE UNIVERSITY OF MINNESOTA**

**BY**

**Vishnu Praveen Pedireddi**

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE IN COMPUTER SCIENCE**

**Dr. Pete Willemsen**

**September 2008**

---

**UNIVERSITY OF MINNESOTA**

This is to certify that I have examined this copy of a Master's Thesis by

**Vishnu Praveen Pedireddi**

and have found that it is complete and satisfactory in all respects,  
and that any and all revision required by the final  
examining committee have been made.

Pete Willemsen

---

Name of Faculty Adviser

---

Signature of Faculty Adviser

17 September 2008

---

Date

Graduate School



## **Acknowledgements**

I am grateful to a number of people who have contributed directly and indirectly to the completion of my thesis work. I have had a great fortune to work with some of the brilliant minds in Department of Computer Science at University of Minnesota Duluth.

I begin by thanking my advisor, Pete Willemsen. I consider mutual trust and cooperation as foundation for any beneficial endeavor. Pete had provided with everything I needed to learn and succeed in graduate studies and beyond. His passion, interest, and patience has kept me driving through inevitable low points in my research and encouraged me keep going one step at a time.

I would like to thank my committee members, Doug Dunham, and Steven Trogdon for their feedback and comments.

I would like to thank Carolyn Crouch to keep my research efforts going in the summer of 2007. The work accomplished in summer was truly critical to the completion of my work.

I would like to thank Lori Lucia and Linda Meek in the Computer Science Department for supporting me throughout my Master's degree in helping me with the necessary paperwork and administrative tasks.

I would like to thank my colleague, Andrew Norgren for his enthusiasm and interest in helping me out through the course of the work.

Lastly, I would like to thank my parents, Pedireddi Sateesh Kumar and Pedireddi Krishna Kumari for their valuable guidance and advice. They continue to be my role models for living life with passion, creativity, and integrity.

## **Dedication**

This thesis is dedicated to my parents, Pedireddi Venkata Sateesh Kumar, and Pedireddi Krishna Kumar. Also to Anupama Atmakur.

## Abstract

The use of virtual environments is interesting if it closely mimics real world scenarios. An interesting virtual environment is characterized by autonomous agents inhabiting the virtual world. A large scale simulation of entities in a virtual environment introduces dynamic activity into a virtual environment, closely resembling an urban area or vehicles in a freeway. However, support of large scale simulation demands high computational resources. The work presented in this thesis addresses this issue and massively expands the scale of traffic simulation.

The approach to massive expansion to the scale of traffic simulation is achieved by augmenting the Central Processing Unit with additional computational resource available from the Graphics Processing Unit or a “GPU” in a desktop. A modern Graphics Processing Units are parallel computational engines with up to 128 vector processors operating in parallel. A GPU framework is designed and implemented to support traffic simulation. A working model in a GPU follows the paradigms of stream computing. The datastructures used in GPU framework are textures units and vertex buffer objects. The shaders contain code that operates on an input stream and writes output to a predetermined initialized texture memory unit. The use of extensions such as Framebuffer Object, Vertex Buffer Object, Transform feedback, Fragment Shader, and Geometry Shader form an integral part in functioning of the GPU subsystem.

The framework built into the GPU subsystem functions in parallel with the CPU traffic simulation system. The data transfer between the CPU and the GPU via the system bus is kept to minimum to reduce latency. The GPU subsystem operates independent to its CPU built framework. The use of GPU subsystem has massively expanded the scale of traffic simulation by at least 100 times from 200 to 20,000 vehicles simulated in real time. This expansion is achieved in real time. The work demonstrates the feasibility of using a Graphics Processing Unit for computation intensive applications in real time for massive scale up in performance of applications.

## Index

Acknowledgements.....	i
Dedication.....	ii
Abstract.....	iii
Index.....	iv
List of Figures.....	vi
<b>Chapter 1</b>	
Introduction.....	1
Problem Statement.....	1
Hardware Architecture.....	4
Background Work.....	4
Project Objectives.....	5
Project Overview.....	5
Thesis Organization.....	6
<b>Chapter 2</b>	
Computer Graphics – Overview.....	7
Graphics Software Hierarchy.....	9
Graphics Pipeline - OpenGL.....	11
OpenGL pipeline - Overview.....	12
Programmability of Graphics Pipeline – Shaders.....	13
Shader Programming Model.....	15
<b>Chapter 3</b>	
Environment Description Framework.....	17
Components of EDF.....	18
Representation of navigable pathways.....	18
Representation of Roads.....	19
Curvilinear Coordinate System.....	19
Queries.....	20
<b>Chapter 4</b>	
GPU Simulation Framework.....	22
Design of GPU Subsystem.....	23
GPU Subsystem.....	24
Implementation of Algorithm.....	27
Execution Model.....	28
Simulation framework.....	28
Route Determination.....	30
Behavior and Dynamics Module.....	30
CPU and GPU control areas.....	31
<b>Chapter 5</b>	
GPU Subsystem Implementation.....	33
Datastructures and Texture Organization.....	33
Data-Dynamic Textures.....	33

Data -Static Textures.....	34
Contents of Textures.....	35
Representation of Roads.....	36
Multisegment Texture.....	38
Implementation of Intersection.....	39
Link Textures.....	40
Vehicle Attributes.....	41
GPU simulation Algorithm.....	42
Injection into GPU framework.....	43
Route Determination.....	44
Behavior and Dynamics Calculation.....	45
CPU-GPU Vehicle Transfer.....	46
Queries.....	48
Premises-Queries.....	49
Implementation.....	49
Quadratic Minimization.....	50
Newton's Method.....	51
<b>Chapter 6</b>	
Results.....	52
Conclusion.....	55
Future Work.....	56
<b>Bibliography.....</b>	<b>57</b>

## List of Figures

Figure 1: Architecture of CPU.....	2
Figure 2: Architecture of GPU.....	3
Figure 3: Wireframe Model of Road.....	7
Figure 4: Image Rendered With High Degree of Realism.....	8
Figure 5: Software Layers in Rendering Computer Graphics.....	9
Figure 6: Fixed and Programmable Parts of Graphics Pipeline.....	11
Figure 7: Interaction between CPU and GPU framework.....	24
Figure 8: Data Flow in GPU Subsystem.....	25
Figure 9: 1-D Texture.....	26
Figure 10: A 2-D Texture.....	26
Figure 11: A 3-D Texture.....	27
Figure 12: Execution Model.....	28
Figure 13: The Organization Simulation Framework.....	29
Figure 14: Data Organization in Textures.....	36
Figure 15: Index Scheme of 2-Dimensional Textures.....	37
Figure 16: Lanes and Intersection Layout.....	40
Figure 17: Multi Pass Simulation Framework.....	42
Figure 18: Time taken for Route Determination Module for a single simulation step.....	52
Figure 19: Time taken for Behavior & Dynamics Module for a single simulation step.....	53
Figure 20: Time taken for Filtering Module for a single simulation step... ..	53
Figure 21: Time taken for a single simulation step.....	54

## **Chapter 1**

### **Introduction**

Rich dynamic and interesting virtual environments are characterized by multiple autonomous agents that populate virtual environments and interact with the residents. The Environment Description Framework (EDF) by Willemsen et al (2006) is a platform that supports real time traffic simulations. The framework provides a substrate to build autonomous vehicular behavior. The EDF encapsulates information about the structure, layout of the navigable pathways and, rules that govern behavior on roads and walkways.

Massive computational resources are required for simulation of large interactive simulated environments. The work presented expands the scale of traffic simulation by using a subsystem that supplies additional computation power and simulating approximate ambient vehicular simulations by using commodity graphics processors (Graphics Processing Units).

### **Problem Statement**

The Environment Description Framework (EDF) is a system implemented to support real time vehicular simulation as described in Willemsen et al (2006). Vehicles navigating in virtual environments utilize the services provided by the framework to govern its locomotion. The framework and its associated vehicular simulations are processed by the Central Processing Unit (CPU). The CPU is the fundamental computational resource. However, the cores of CPU execute instructions in serial. For massive expansion of the scale of vehicular simulation, a CPU is constrained as it remains a serial processor.

Graphics Processing Units or GPU's provide an attractive and inexpensive alternative that supplies massive computational power. GPU's are characterized by high concentration of arithmetic logic units that provide it massive computational horsepower.

The work presented uses the GPU as a computational engine and augments the existing framework to support large scale vehicular simulations. The GPU is used as co-processor to the CPU in simulating real time vehicle driving simulations. The use of OpenGL API together with OpenGL Shading Language is implemented on a framework built in C++ language. The GPU subsystem operates independently of its CPU counterpart with minimal data transfer between the systems.

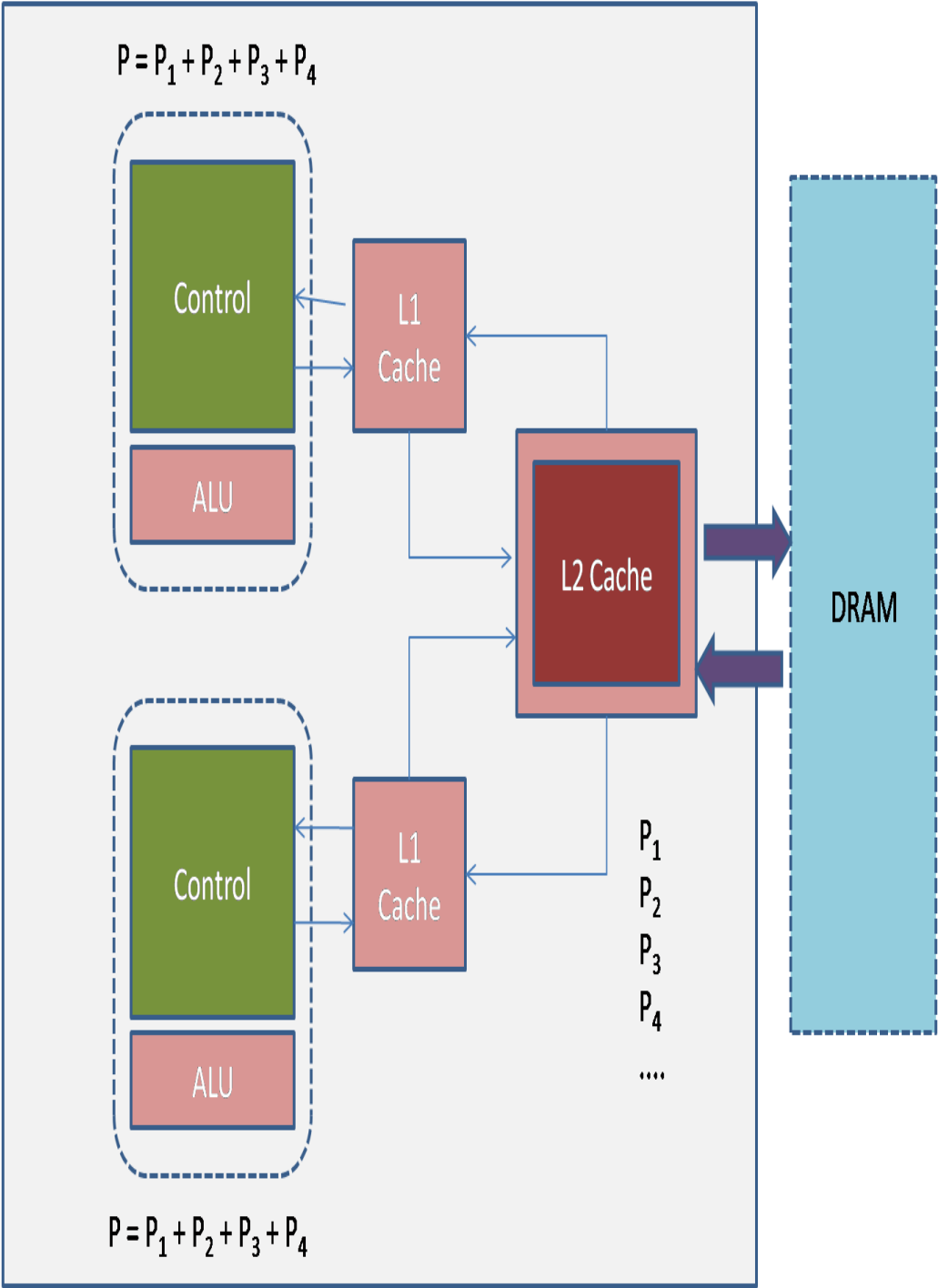


Fig 1: Architecture of CPU.

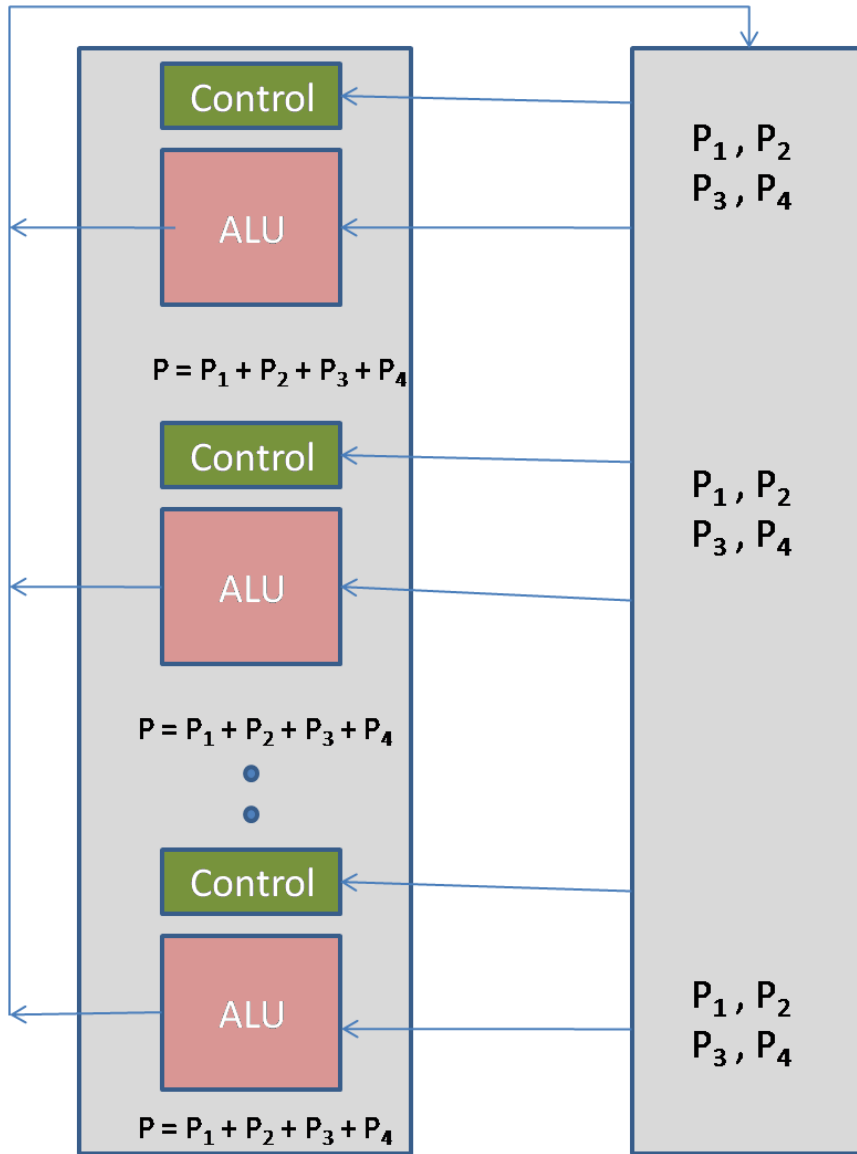


Figure 2: Architecture of GPU.

The GPU is characterized by multiple cores. A high end GPU can be characterized up to 128 parallel processing cores operating in parallel.

## **Hardware Architectures**

The CPU is designed to be a general purpose processor, used for developing general purpose applications. The CPU is designed to offer efficient implementation of coding structures such as conditional and iterative statements. For this purpose, a CPU is designed with high density of transistors devoted for control with a relatively low percentage of transistors for arithmetic computations. It is depicted pictorially in figure 1. The CPU is suited for control intensive programs rather than data-intensive programs.

A GPU is designed for use in rendering of computer graphics. Executing the graphics pipeline in real time implies a need for hardware with number crunching capability. GPU is just meant for this purpose. The GPU is characterized by multiple cores working independently in parallel. Each core of a GPU is composed of high density of transistors devoted for numerical computation than for control logic. This is depicted pictorially in figure 2. This difference is what makes a GPU a powerful number-crunching hardware suitable for high throughput, data-intensive applications.

Use of GPU in the research work presented, has been optimized so the algorithms implemented on the GPU are data-intensive in application. Conditional and iterative loops are minimized, and are replaced at appropriate places by the use of mathematical functions.

## **Background Work**

Willemsen et.al. (2006) presents environment description framework (EDF) that models complex interconnecting network of roads and intersections. A robust and efficient framework provides a base or querying mechanism and provides an intuitive ribbons structure for vehicle simulation in real time. Wang et.al. (2002a) describes a simple and efficient technique to approximate arc-length parameterized spline curves that closely match spline curves typically used to model roads in virtual environments.

Wang et al (2005) describes the details and implementation of autonomous vehicular behavior in a virtual environment using EDF. Wang et al (2004) describes the behavior of vehicles at negotiating intersection in EDF. A detailed implementation of the navigable surfaces that compose EDF is described in Wang et al (2002a). Wang et al (2002b) describes the convergence methods used in the EDF in conversion between Cartesian coordinates and curvilinear coordinates local to a road.

## **Project Objectives**

The present work seeks to redesign the implementation of EDF to fit the parallel architecture of Graphics Processing Units (GPU). It can be summarized as follows:

1. Definition and implementation of GPU framework that operates independent of its CPU counterpart.
2. Implementation of robust and efficient vehicular behavior in real time.
3. Massive expansion of scale of the vehicular simulation.
4. Create an ambient backdrop of vehicular activity simulated by GPU.
5. To provide effective communication channels and interaction mechanisms between CPU and GPU framework.

## **Project Overview**

The work presented in this thesis describes an implementation that utilized the processing capabilities of a GPU to extend the scale of the vehicular simulations. The GPU subsystem designed provides a dynamic backdrop of vehicular simulations against accurate and local computations handled by the CPU framework. The GPU subsystem offloads the CPU of computations, working as a co-processor providing approximate behavior decisions and dynamics calculation. The simulated traffic virtual environment is categorized into two regions.

1. The CPU controlled regions.

The CPU controlled regions performs an accurate traffic simulation in its sphere of influence. The convergence algorithms implemented are accurate as the operation in the framework is performed at double precision floating point numbers.

2. The GPU controlled regions.

The GPU controlled regions operates at single precision floating point number. Consequently the dynamics calculation is approximate than its CPU counterpart. However, the scale of traffic simulation handled by the GPU compensates for the loss of

precision. The GPU is used to create a dynamic backdrop of traffic against more accurate CPU simulations.

As vehicles navigate on the roads, they enter from a CPU controlled region to a GPU controlled region. Appropriate mechanisms have been implemented using OpenGL extensions to support for such a transfer between control regions. The implementation of these features adds extra passes to the whole framework.

Thus, vehicular simulation by CPU and GPU is performed in pre-defined regions. The regions are delineated by the accuracy of simulation desired in a given area against more approximate navigation.

### **Thesis Organization**

The first chapter introduces the problem statement and approach taken to solve it. The GPU simulation framework is based on the programming model and architecture of the Graphics Processing Units themselves. Chapter Two outlines the present generation graphics programming model. An introduction of the graphics pipeline in OpenGL and computer graphics in general is described. With the increasing programmability of graphics processing units, parts of the pipeline programmable together with techniques to use are described.

The third chapter introduces Environment Description Framework (EDF) that is the substrate on which virtual environments are built. Representation of navigable surfaces and curvilinear system of coordinates are introduced. The querying mechanism provided by the EDF is described.

The fourth chapter presents an overview of framework implemented in the GPU. The design and its compatibility with its CPU counterpart are described. The multi pass offline rendering algorithm and the comprising modules are described.

The fifth chapter presents the implementation of the GPU simulation framework. The algorithm, datastructures, and the mechanisms implemented are described. Implementation details of every pass comprising the GPU simulation framework are discussed.

The sixth chapter presents the results of the work, conclusion, and the improvements to the present GPU subsystem.

## Chapter 2

### Computer Graphics – Overview

Computer Graphics is a branch of engineering defined and applied by mixture of knowledge and skills from mathematics, computer science, hardware design & logic and sound programming substance. Graphics achieved in animation movies, games and other 3D applications are not mere two dimensional pictures on screen as in painting but are a product of computational processes of representing virtual objects mathematically and polishing the result to add realism to the scene. These seeming simplistic and entertaining graphics visuals mask a complex framework of rendering engines and logic that weaves itself out of the specialized hardware more commonly known as graphics card or Graphics Processing Unit (GPU) in graphics parlance. The realism to the virtual objects in a virtual scene in research and commercial applications like 3D games is based on two distinct graphic components.

1. Mathematical representation of three dimensional geometric skeletons of virtual objects. At the lowest level every object is represented as a combination of geometric primitives such as triangles, triangle strips and quadrilaterals. Processing of vertices making up the primitives forms an important part of the rendering process.

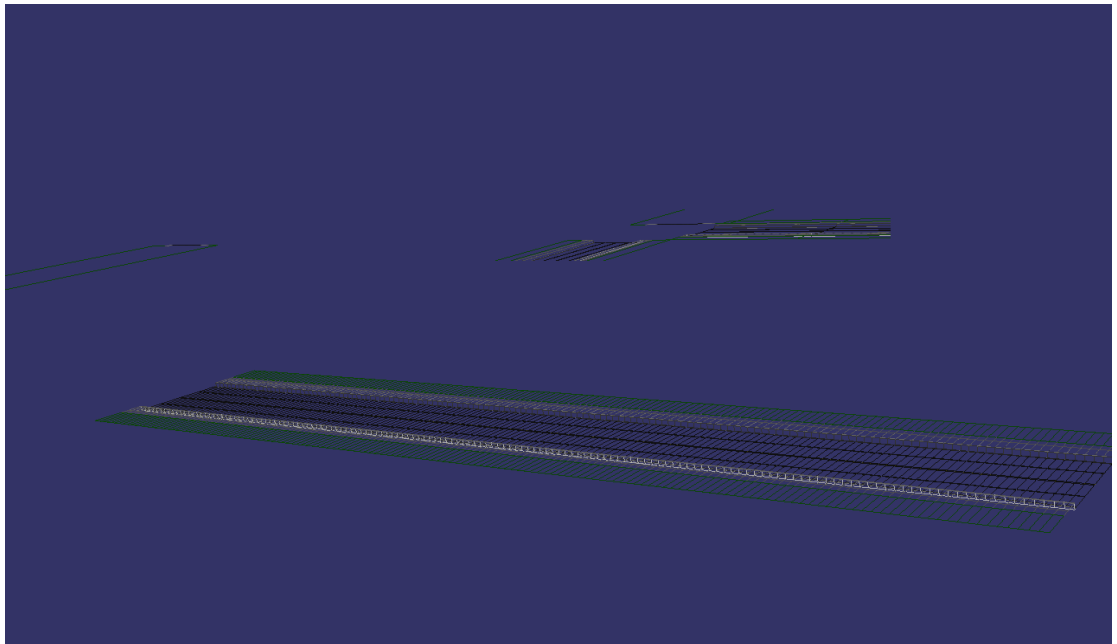


Figure 3: Wireframe Model of Road

2. Visual effects rendered in the scene for added realism.

The three dimensional objects rendered are textured and rendering techniques are applied for real-life like images. The techniques applied include shadow processing, light reflection, light refraction, tessellation and several other characteristics of materials making up virtual object. An example of which is shown in the figure below.



Fig 4: Image rendered with high degree of realism.

Quality of computer graphics generated at highest level of abstraction is thus dependent upon processing of the vertices and its attributes and image processing of the virtual objects that make up the virtual space. Processing of these two major components however is computationally dense due to sheer number of vertices of the primitives processed and number of distinct individual image units that need to be processed for display of a single frame. Handling such dense computations in real-time led to the development of Graphics Processing Unit (GPU), a dedicated processor as an aid to the Central Processing Unit in the generation of high quality graphics application.

## Graphics Software Hierarchy

Graphics generation from a developer point in naïve sense is conversion of 3D models into two dimensional images to be displayed on the screen. This creation process being computationally intensive, the current implementations utilize a pipeline-based approach to process the data from the time it is received from the CPU to creation of pixel data for the display device. The software layers that make up this process are pictorially represented in the figure below:

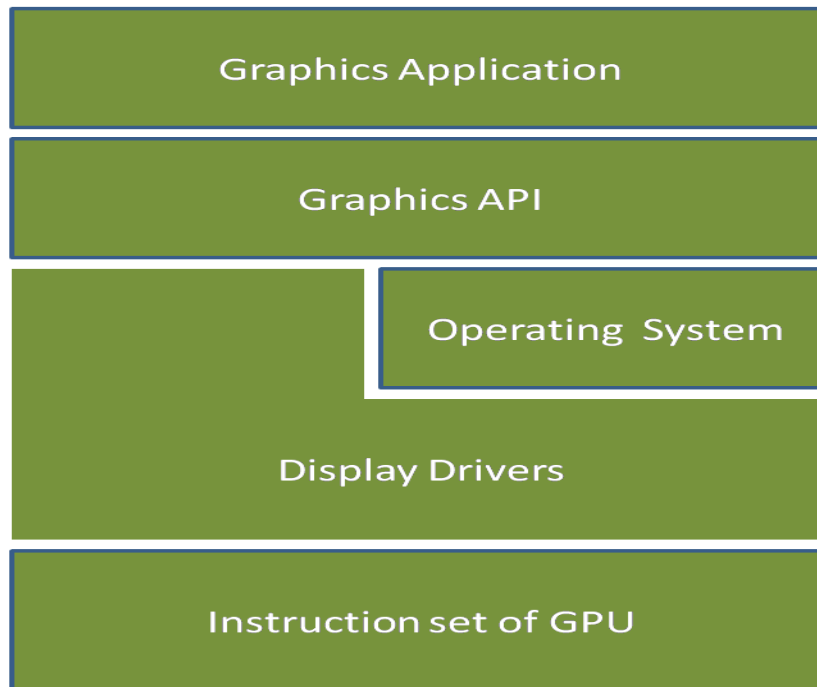


Figure 5: Software layers in rendering of computer graphics.

As shown in the software hierarchy, graphics programming involves use of graphics Application programming interface (API) such as OpenGL or DirectX that takes charge of lower level instructions issued to drivers of the graphics card. The API directly communicates and coordinates with the operating system and graphics card driver for setting up resources that need to be used for the graphics application to function appropriately. The data and instructions are generated by the graphics API are sent down to the drivers that then generate appropriate instruction set for the graphics processing unit(GPU) operating in the computational pipeline. The end result is the set of pixel values representing colors that are to displayed on the display device.

This pixel-data is then stored in the frame buffer and awaits request for data transfer to the display device.

The hierarchy of software in figure 5 is the high level representation of software organization in most common graphics applications. GPU being a specialized hardware with its own framework of parallel processors, a memory hierarch and communication mechanism follows a pipelined architecture to processing of graphics primitives. This pipelined architecture is the heart of any graphics device and is explained in the next section.

## Graphics Pipeline -OpenGL

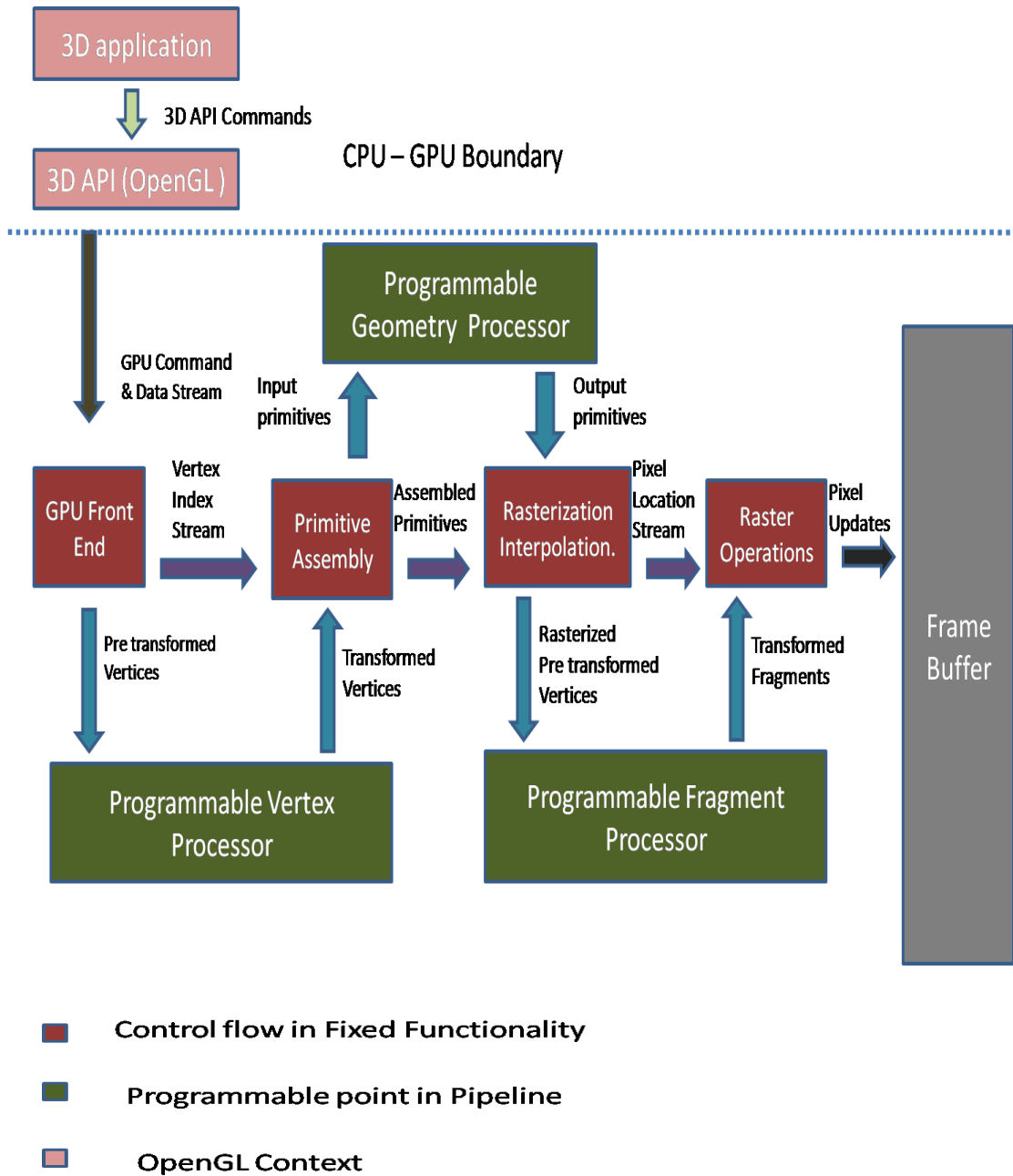


Figure 6: Fixed and Programmable Parts of Graphics Pipeline.

Designing a pipelined architecture for efficient task execution allows an implementation to achieve high throughput in terms of instruction execution. Pipelining is a mechanism of breaking

a large task into chunks of smaller work units. These small work units are then executed by hardware units separately coordinating among themselves to pass the output generated by one as input to the next unit. This technique can be compared to assembling cars on an assembly line.

A “free feature” is defined as any functionality that can be embedded into an implementation without any extra or marginal increase in computational cost. The pipeline architecture allows for inclusion of free features. In software, performance is inversely proportional to algorithm complexity. However, in hardware implementations, performance is either invariant to algorithm complexity or falls off catastrophically due to software fallbacks.

The process of transforming data provided by an application into something that is visible on the display screen is referred to as RENDERING. Utilizing the pipelined architecture as a means for rendering is thus adopted in the design and implementation of several graphical Application Programming Interfaces like OpenGL. In this section and we will be taking a tour of OpenGL and its pipelined architecture. The important characteristics of OpenGL will be explained in this section.

### **OpenGL Pipeline – Overview**

The OpenGL pipeline is pictorially depicted in Figure 6. The overview of the rendering process is summarized below:

1. It would be a massive software engineering task to build a graphics application that interfaces directly with the underlying hardware to create a realistic graphics. Graphics Application programming interfaces(API) are thus built to manage the underlying hardware and system resources and provides easy to use handles for designing and building graphics programs. OpenGL and DirectX are the most commonly used API’s for graphics programming. The flow of control is initiated by the application in the form of 3D API commands.
2. The underlying API generates the appropriate command stream and the data stream as an input for the GPU. The data stream consists of the following information:
  - a) Vertices and its attributes.
  - b) Geometry of the vertices defined (i.e. primitives defined).
  - c) Position and characteristics of light sources.
  - d) The texture coordinates of vertices.

- e) The surface characteristics of the objects composing the three dimensional scene.

The Command stream consists of the following:

- a) The transformation such as scaling, skewing, translation and rotations that need to be applied on the primitives.
  - b) The viewing volume, direction of viewing, and the type of projection (orthographic or perspective) of the three dimensional images and size of viewport determine the 2D image created on the screen.
  - c) The OpenGL extensions used in OpenGL rendering algorithms such as Frame Buffer Objects, Transform Feedback etc.
3. The second stage is the per-vertex operations, including transformations, lighting, primitive assembly, clipping, projection, and viewport mapping.
4. The third stage is Rasterization. This stage produces fragments, which are series of frame buffer addresses and values, from the viewport-mapped primitives as well as bitmaps and pixel rectangles.
5. The fourth stage is the per-fragment operations. Fragments are defined as the smallest units in the rasterized image consisting of quadruples defining the color and transparency of the 2D image basically per-pixels generated after rasterization. Before fragments go to the framebuffer, they may be subjected to a series of conditional tests and modifications, such as blending or z-buffering.

### **Programmability of Graphics Pipeline – Shaders**

OpenGL has presented the application developers a flexible interface but did not allow for the programmability in the definition and implementation of the graphics pipeline. Parameters of the pipeline stages can be altered to select the variation on the processing that occurred in the pipelined stages. However, the sequence of operations on the geometry or on the image data remains the same.

Exposing these processing points to be programmable in the graphics pipeline will allow the application developers to define the processing that occurs. These programmable points in the

graphics pipeline are referred to as SHADERS or KERNELS. Shaders programs serve as plug-ins to the graphics card driver to define the operations for vertex and fragments in the pipeline.

The advantages of using shaders are given as follows:

1. Allows the application developers an access to the stream processors on the graphics processing unit to define custom per-vertex and per-fragment operations.
2. Such programmability allows for implementation of realistic Lighting effects such as reflection, refraction and shadows.

These programmable points in the graphics pipeline are shown in figure 6. The Shaders can be categorized into three variations based on their functionality.

a. Vertex Shader.

The vertex shader defines the per-vertex operation to be performed on each vertex coordinates of the primitives. The vertex processor in the GPU executes the vertex shader and replaces the fixed-functionality OpenGL per-vertex operations. The operation includes the following:

- i. Translation, scaling, rotation transformations are applied to the vertex coordinates.
- ii. The projection and the modelview matrices that define the volume and point of view are applied to the vertex coordinates.
- iii. Normals are transformed to eye coordinates.
- iv. Texture coordinates are to be assigned to the vertex coordinates.
- v. Per-Vertex lighting, Color material computations and color index lighting are computed.

b. Fragment Shader.

The fragment shader defines the per-fragment operations to be performed on the fragments of Rasterized image. The fragment processor executes a fragment shader. The following fixed-functionality operations are affected:

- i. Texture application is performed.

- ii. Color sum and fog is applied to each of the fragments.
- c. Geometry Shader.

The geometry shader defines the per-primitive operations to be performed on the primitives defined in the application. The per-primitive operations are:

- i. Generation of new primitives from the existing primitives defined in the application.
- ii. Changing the geometry of input primitives. For example a triangle can be changed into a set of three dimensional lines.

### **Shader Programming Model**

Graphics processors have evolved from hardware implementations of OpenGL/DirectX graphics pipelines to programmable parallel processors (Rost et al (2004)). Modern GPU have arrays of parallel processors executing the shader programs in parallel. The programming model implemented by modern GPU is a CONCURRENT READ EXCLUSIVE WRITE (CREW) PARALLEL RANDOM-ACCESS MACHINE MODELS execution model. This memory model allows any number of processors to read from any memory location but allows only exclusive, single processor memory writes. A computational kernel or shader is executed on multiple input data streams in parallel and writes the output to single pre-determined locations.

Current GPUs have Multiple-Instruction, Multiple Data (MIMD) vertex processing units and Single Program, Multiple-Data (SPMD) fragment processing units. GPUs execute *batches* of fragment threads in Single Instruction-Multiple Data (SIMD) style of execution (one thread per pixel). The size of single batch of threads varies between hundreds and thousands of concurrent threads depending on the hardware implementation. Branching is supported in fragment kernels. Coherence of branching operations in a batch being executed adversely affects the execution speed of the fragment threads. If all the threads take the same branch, performance remains high. However, if both branches are needed by the batch, both branches are executed by the fragment thread resulting in steady decrease in performance.

The massive parallelization achieved by thousands of threads being executed in parallel in GPU is concealed from the programmer thus allowing the hardware vendor to change underlying hardware architecture and the number of processors without requiring any modifications to

existing programs. The programmer only specifies the data stream and kernel to be executed. The threads are not allowed to perform inter-process communication.

The kernels/shaders from a programmer perspective is an opportunity to utilize the parallel processors of the GPU for general purpose computations by defining appropriate input streams into the GPU. The input streams are defined and initialized by the graphics API such as OpenGL. The key therefore to utilize the parallel processors is to define algorithm at two levels.

1. Algorithms in kernel/shaders applied on input data streams and capturing the output at the end of the pipeline in the frame buffer.
2. Algorithm at an application level in 3D graphics API that
  - a. Sets the context and state of the graphics pipeline.
  - b. Creates data structures encapsulating the input data that kernel should be operational.
  - c. Captures the output stream from the kernel/shader at the end of the pipeline into a data structure usable by the application.

The work presented in this thesis uses this algorithmic structure to use the GPU as co-processor to extend the scale of a model for autonomous vehicle simulation running in real time. The next chapter describes the previous work and vehicular simulation framework.

## Chapter 3

### Environment Description Framework

To develop an immersive virtual environment, robustness and efficiency of the code that creates such an effect is of outmost importance. Engaging virtual environments mean autonomous agents such as pedestrians strolling pathways, autonomous vehicle behavior on roads, leaves swaying on the trees etc. The details of the Environment Description framework are described in Willemsen et al (2006), Wang et al (2004), and Wang et al (2005). The entities in a virtual environment can thus be abstracted in two distinct sets depending on their behavioral characteristics:

1. Static structures like roads, buildings, benches in the park etc.
2. Dynamic entities such as pedestrians, vehicles travelling on roads, waves in the lake etc.

The static structure like roads and buildings define the topography of a virtual environment. Such entities do not exhibit behavior in a timeline of any simulation system. On the other hand, there exist dynamic entities such as people in a park that define their behavior based on the behavior they are in. For example children in a park tend to run and play around whereas students in a class usually sit at a place and listen to the lecture and interact with the instructor. The feature that differentiates static from dynamic entities is *behavior* exhibited by them.

Creation of realistic virtual environment from a developer's perspective is based on efficiency of framework of programming model implemented in the system. Environment Description Framework (EDF) is a model that is one of the building blocks of traffic simulation described in my work. Features of Environment Description Framework are described below:

1. Representation of static and dynamic entities.

Entities in a virtual environment are described mathematically in their most basic representation. Based on the work of Willemsen et al, roads are modeled as cubic splines that can twist and turn in a three dimensional space. The orientation of cubic splines defines the structure of the road on the corridor.

Entities in the virtual environment are furthermore can be represented by using the graphics API such as OpenSceneGraph and OpenGL. Forms of appropriate structures can

be described using graphics API. OpenSceneGraph is used for the present implementation.

## 2. Interaction between the static and dynamic entities.

The behavior exhibited by dynamic entities such as cars and pedestrians is what makes a virtual environment interesting. Such a behavior by dynamic entities is dependent on the robustness and completeness of the interface provided by the static structures such as roads. The constructs modeling static structures such as roads should provide information to a requesting entity such as a car driving on the road about its position and orientation in the three dimensional space. The Environment description framework thus is a interface to the behavior of dynamic entities by responding to its queries. The EDF provides a substrate to build the behavior of dynamic entities populating the simulation environment.

The EDF can be thought of as a collection of logical structures that makes the representation of static & dynamic object and interfaces defined used by dynamic objects. It glues the various low-level aspects of the static entities with the behavior of autonomous agents populating the virtual environment. The present version of EDF supports the simulation of vehicles navigating a network of roads. Each vehicle is characterized by its autonomous behavior which implies that the behavior of large group of vehicles is not stereotyped. The behavior of every vehicle is independently defined.

### **Components of EDF**

The EDF being the substrate for the support of all programmable features can be characterized by the components mentioned below:

#### **Representation of navigable pathways**

The layout and interconnections of pathways in the virtual environment are represented as parametric cubic splines. As described in the paper by Willemsen et al (2006), parameterized cubic splines are the curves of choice for motion guidance due to its simplicity in representation and ease of usability. The parametric cubic splines in the three dimensional space forms the centerline in the representation of roads that turn and twist as defined by the underlying terrain. The navigable pathways defined by the cubic

splines are termed “*ribbons*” for use in the virtual environment. The ribbon network thus defines spatial layout of the roads in the virtual environment.

Ribbons defined in a virtual environment can be of various types depending on its usability. For example roads and intersections can be two very different kind of navigable pathways meant for different purposes based on structure and shape of ribbons. Ribbons may be modeled to resemble different types of roadways such as roundabouts and intersections. However, the two constructs that will be used quite extensively are:

1. Roads
2. Intersections

### **Representation of Roads**

Three dimensional space curves define the central axis of the ribbon in the EDF. Parametric cubic splines are best suited for the purpose of motion control and guidance and are widely used in computer animation and virtual environments as [Willemsen et al 2003]. As the parameter ranges over the interval of definition, the computed position traces a smooth curve in the three dimensional space.

Given a set of points  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$  .....  $(x_n, y_n)$ , the procedure to obtain the equations for the parametric spline curves are defined in Wang et al (2002a). The equations for the curve are given as

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

$$sl(t) = a_{sl} t^3 + b_{sl} t^2 + c_{sl} t + d_{sl}$$

### **Curvilinear Coordinate System**

Navigation in a virtual environment is considerably simplified if appropriate frames of references are created that are intuitive, easy and efficient in use. EDF simplifies the

decision making process for the vehicles navigating the ribbon network by defining a frame of reference local to the roads themselves apart from the global Cartesian frame of reference in the virtual environment. The coordinates in the new frame of reference are called CURVILINEAR COORDINATES. A vehicle in the virtual environment is therefore defined in two coordinate systems:

1. The three dimensional Cartesian coordinates.
2. The Curvilinear coordinates.

There are three coordinates that are described as follows:

- Distance: This coordinate specifies the distance of the vehicle on the ribbon along its axis.
- Offset: This coordinate specifies the lateral distance of the vehicle from the axis of the ribbon.
- Loft: This coordinate specifies the distance of the vehicle in the direction of the normal to the surface of the ribbon.

Each vehicle is characterized with a set of DOL coordinates that define its position relative to the ribbon in which the vehicle is navigating. The curvilinear coordinates are used for tactical decision making purposes. For example, two vehicles travelling with DOL's defined as (100, 0.0, 0.0), and (110, 0.0, 0.0) are separated by a distance of 10 units with respect to the ribbon in which it is defined. The calculation of this information from the vehicles corresponding Cartesian coordinates follows an elaborate mathematical procedure which induces bottleneck (induced due to the iterative loops required for convergence to sufficient iteration) in the real time functioning of the system. However, the flexibility of maintaining the DOL coordinates comes at the cost of implementing efficient conversion procedures to convert the Cartesian coordinates to curvilinear coordinates in the real time implementation.

### **Queries**

Agents populating the virtual environment are interesting if they exhibit independent behaviors in their interaction with surrounding environment. Interactions of autonomous agents can be well defined if the state of the environment is efficiently accessible to guide

its decisions. Support for such a query mechanism forms an indispensable part of any simulated environments. For example a pedestrian crossing a road at an intersection looks and waits for the sign on the opposite direction. Once the walk sign is signaled, the pedestrian takes a decision to cross the road. This simple example can be viewed from EDF's perspective in the following steps.

1. The autonomous agent (Pedestrian) queries the EDF for the state of signal in the virtual environment.
2. Upon receiving the state from the EDF about the signal ("Walk" or "Don't walk"), the pedestrian decides to wait (if the signal flashes "Don't walk" sign) or cross the intersection (if the signal flashes "Walk" sign).

Support of a robust and query mechanism for enquiring the state of the environment is provided by the EDF for the vehicles to navigate autonomously on the ribbon network. Some of the queries supported by the EDF are:

- Querying the Cartesian and curvilinear coordinates of the vehicle.
- Conversion from Cartesian coordinates to the curvilinear coordinates and vice-versa of the vehicles.
- Querying the speed limit of the road.
- Querying for the next ribbon to be navigated.
- Querying for occupancy information of the vehicles.
- Querying the route, the vehicles intend to take.

## Chapter 4

### GPU Simulation Framework

The creation of an ambient backdrop of vehicle activity to the scale of couple of thousands of vehicles simulated in real time implies a heavy load on the CPU. It has been found upon testing that simulation of more than 200 vehicles in real time on the CPU built framework (EDF) leads to drastic reduction in frame rates, to the extent that the user observes a jittering motion on the display screen. One of the primary reasons for jittering is that the CPU being a serial processor, is not fast enough to simulate autonomous behavior of a large number of vehicles in real time. A need for an alternative computational resources is desired which shares the computational burden with the CPU.

The approach presented in this work utilizes graphics hardware as a computational engine for aiding the CPU as a co-processor. The GPU subsystem is therefore designed and built to work in tandem with its CPU counterpart. The main goals in the design and implementation of the GPU subsystem are the following:

1. Create a software module that would work in parallel with its CPU counterpart in simulating the ambient traffic conditions on the road in the virtual environment.
2. Work in tandem cooperating with the CPU independently without any significant CPU intervention.
3. Minimize the flow of data between the CPU and the GPU. Transfer of data between the two systems is extremely slow when compared with the speed of execution in each of the system.
4. Extract only the important attributes and procedures of the CPU simulation to be included into the GPU framework. This would mean a clear separation of data from algorithms. Such a separation is needed for tailoring the simulation to the setting and framework of the GPU.
5. Implementation of CPU and GPU controlled areas in a virtual environment that mark the regions that are simulated by the CPU and the regions simulated by the GPU.
6. Interaction between the CPU and GPU controlled areas without causing delays in the rendering of subsequent frames during the simulation.

7. Implementation of such a subsystem should be supported at real-time (at 60 Hz.)

The design and implementation of above goals is made possible by the use of several extensions to the OpenGL API. The extensions in OpenGL API that are used are the following:

1. OpenGL shading Language.
2. OpenGL framebuffer object.
3. Vertex buffer Object.
4. Transform Feedback.

### **Design of GPU Subsystem**

Conventional software systems are composed of software code written in an intermix of high and low level language compiled to machine code. The flow of control within software code is designed and guided by the programmer. Augmenting computing power to a software framework by using a GPU or graphics card involves setting up of proper interface and efficient control and data flow between subsystems. Utilizing the OpenGL API as a means of tapping this computational resource implies using the client-server programming model of the OpenGL architecture. OpenGL API operates in client-server settings to deliver the 2D raster image on the display device. The GPU is the server that accepts rendering and non-rendering operations from CPU, the client that requests for 2D image to be Rasterized on the display device. Therefore, a server may be connected to multiple clients or multiple CPU systems connected across a network to accept display requests. In the simplest case, the desktop computer to which the graphic card is a part forms the client-server execution model. Thus, the algorithms and implementation of the GPU subsystem is designed to seamlessly become part of the client-server execution model of 2D Rasterization. A figure illustrating a high level view of the system is shown

below.

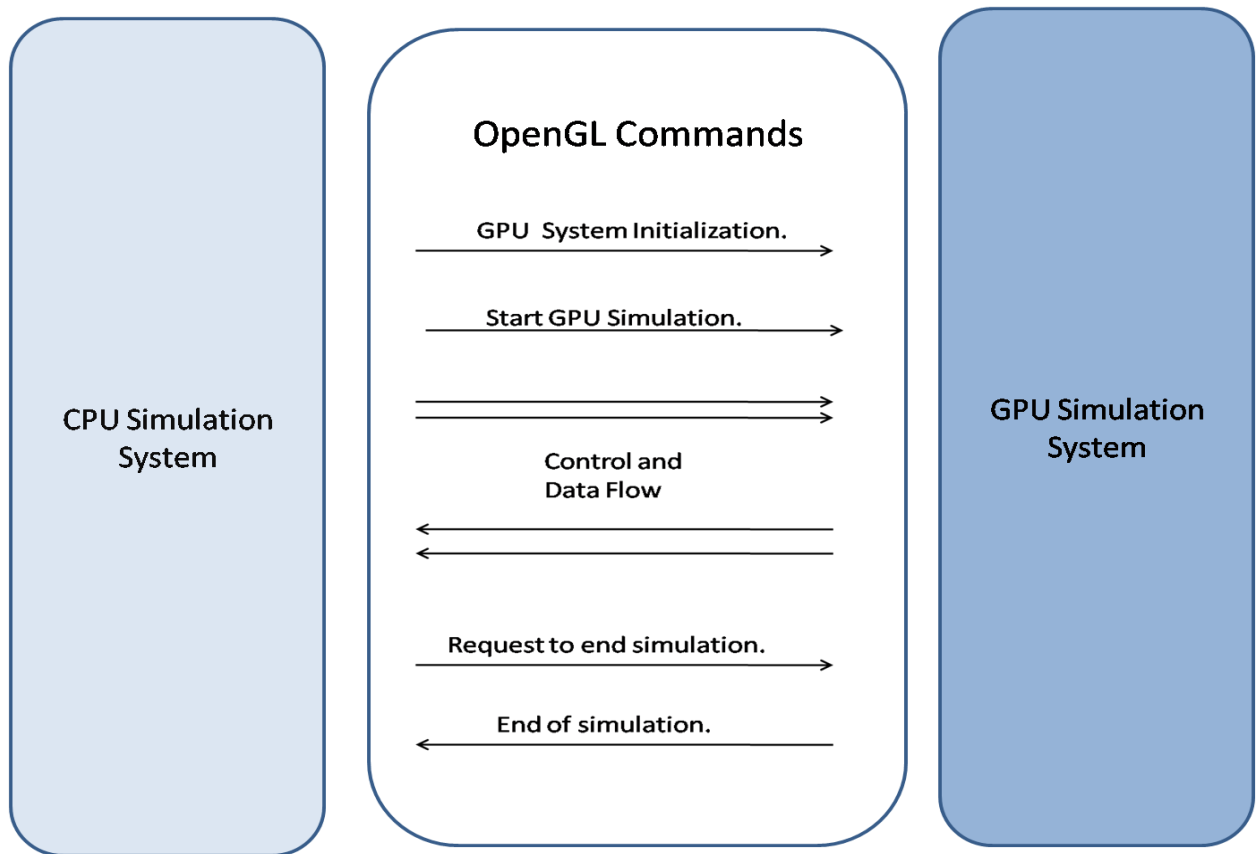


Figure 7: Interaction between CPU and GPU framework.

### GPU Subsystem

Definition and delineation of algorithms from the dataset it works on is the first step in defining a GPU based system. The GPU operates as a vector processor on incoming stream and streams the output on to pre-defined regions in the graphics memory. The GPU working model does not allow for writing to random region of graphics memory. The data cannot be intertwined and mixed with the algorithm itself because it's working model. A black box representation of the subsystem is pictorially drawn

below.

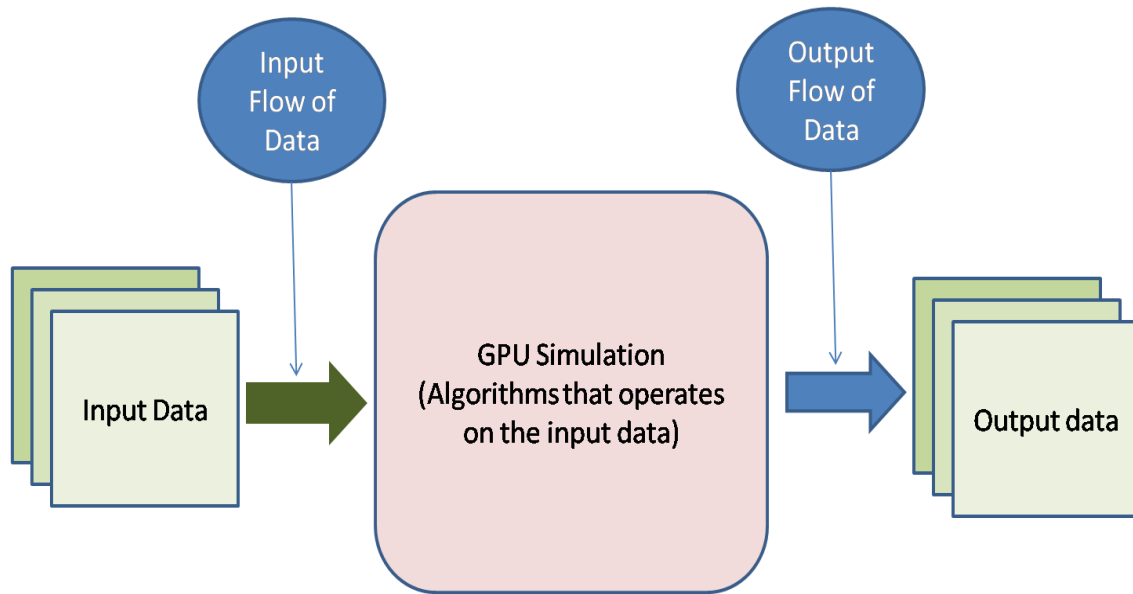


Figure 8: Data Flow in GPU Subsystem

The datasets to the Graphics Processing Unit are not the part of working code. The input and output datasets to a Graphics Processing Unit are well defined regions of memory in the graphics card called as TEXTURES. A texture is defined as a specialized memory unit in a graphics memory capable of storing images that are pasted on to the three dimensional objects in the virtual environment. A texture is originally meant for storage of colored image format in the graphics memory for fast access by the graphics processor during Rasterization. These texture units are potential carriers of general purpose data if the data can be organized in the format in which the textures are defined. Textures in the GPU are basically of three types:

1. 1-D Textures.
2. 2-D Textures.
3. 3-D Textures.

The types of textures mentioned are defined by the way texels are composed in each type of texture. A TEXEL is defined as smallest element a texture can be composed of. A texel stores the color of smallest element the texture is made of. A texel is composed of the following four components:

1. Red component.
2. Green component.
3. Blue component.
4. Alpha component.

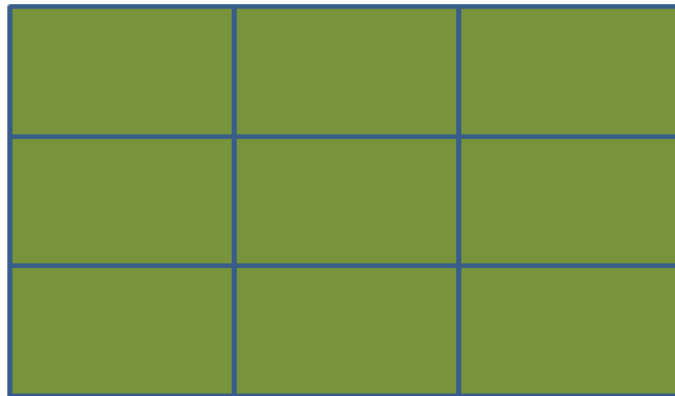
A set of four values make up a texel in a texture. A collection of texels make up a texture. The diagram of 1-D, 2-D and 3-D textures defined in terms of texels is shown below.



## 1-D Texture

Figure 9: 1-D Texture.

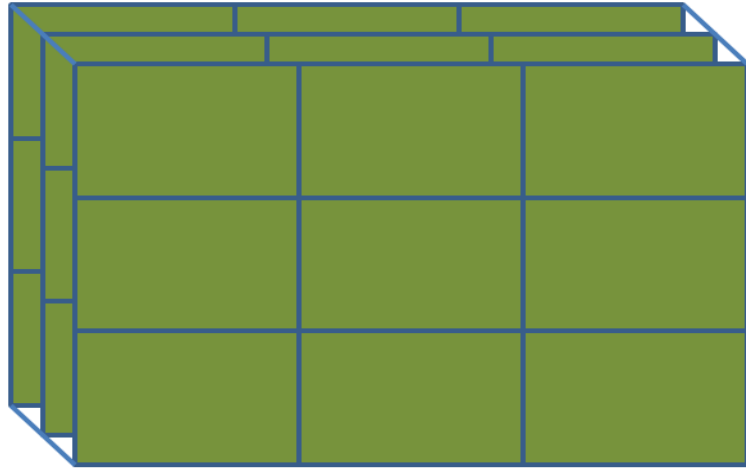
A 1-D Texture contains an arrangement of texels in the texture in one dimension.



## 2-D Texture

Figure 10: A 2-D Texture

A 2-D Texture would contain the arrangement of texels in two dimensions.



## 3-D Texture

Figure 11: A 3-D Texture

A 3-D texture is composed of texels arranged in three dimensions. A 2-D texture can also be thought of as a collection of 2-D textures.

In the GPU simulation framework, 2-D textures have been used as appropriate data structures to hold the datasets comprising the traffic simulation. Use of 2-D textures is much faster than the use of 3-D textures.

### Implementation of Algorithm

The algorithm described to operate on datasets defined as textures forms the part of shader code in the graphics pipeline as described in the previous sections. The shaders are the programmable parts of the graphics pipeline. The goal of the graphics card is to assign color to each pixel on the display screen.

A single complete cycle in a graphics pipeline is defined as one rendering pass or PASS. A single pass therefore populates the framebuffer with the pixel values. In a programmable graphics pipeline, a single pass involves applications of the three shaders in a graphics pipeline.

## Execution Model

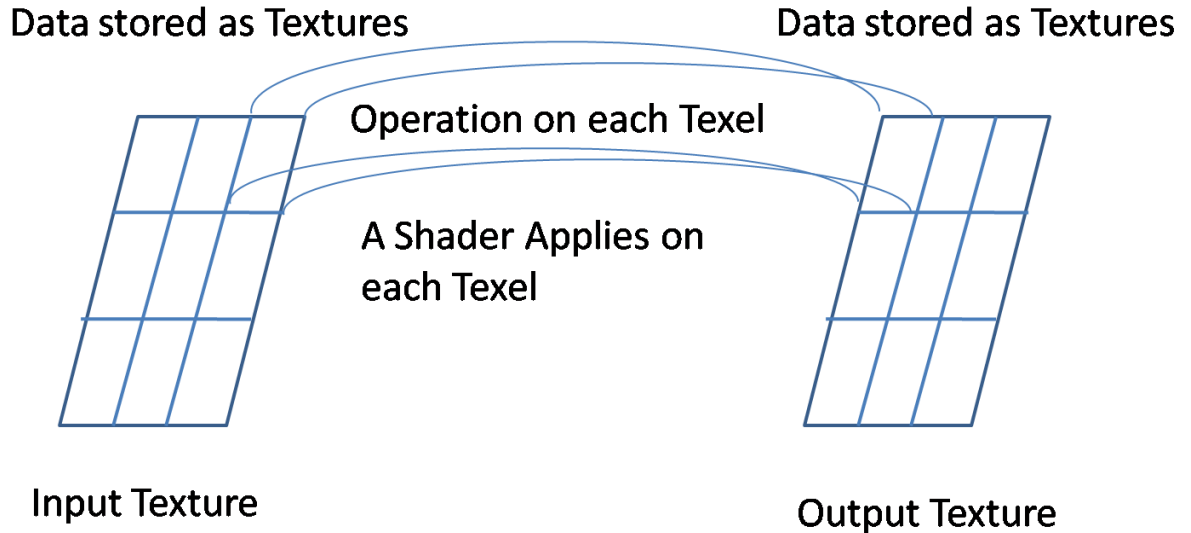


Figure 12: Execution Model

Each texel of texture represents data pertaining to a single vehicle in the GPU framework. For example as shown in the above diagram, the input texture can be defined to hold the Cartesian coordinates of each vehicle. A fragment shader can be defined to do a simple change in position of each vehicle by performing simple arithmetic operations on the four components of the texel. A graphics card is a stream processor and is designed to follow SIMD model of execution. The output of the fragment shader is then streamed to a texture of similar dimensions so that there is one on one mapping from input texture to output texture.

## Simulation Framework

This section sketches an overview of the control and the data flow in the simulation framework without getting into the technical details of the execution. The purpose of the GPU software system is simulating autonomous behavior of vehicles moving on a virtual road. In its lowest level, it means implementation of a mechanism that computes the position of the vehicles after every time segment  $t$ . However, the algorithm for such a functioning is not simple. It involves the use of complex numerical algorithms efficiently coded to run on the GPU. The simulation system is designed to operate as a multi-pass offline rendering system. Unlike the principles of object-

oriented or functional programming methodologies, a fragment shader with multiple branching, conditional and iterative statements results in rapid fall in the performance of the GPU. Organizing the algorithm into multiple shaders based on the frequency of usage of the branching and, conditional and iterative statements for increase in the performance is much warranted. Thus, the simulation framework is organized into three distinct passes. They are shown in the diagram drawn below.

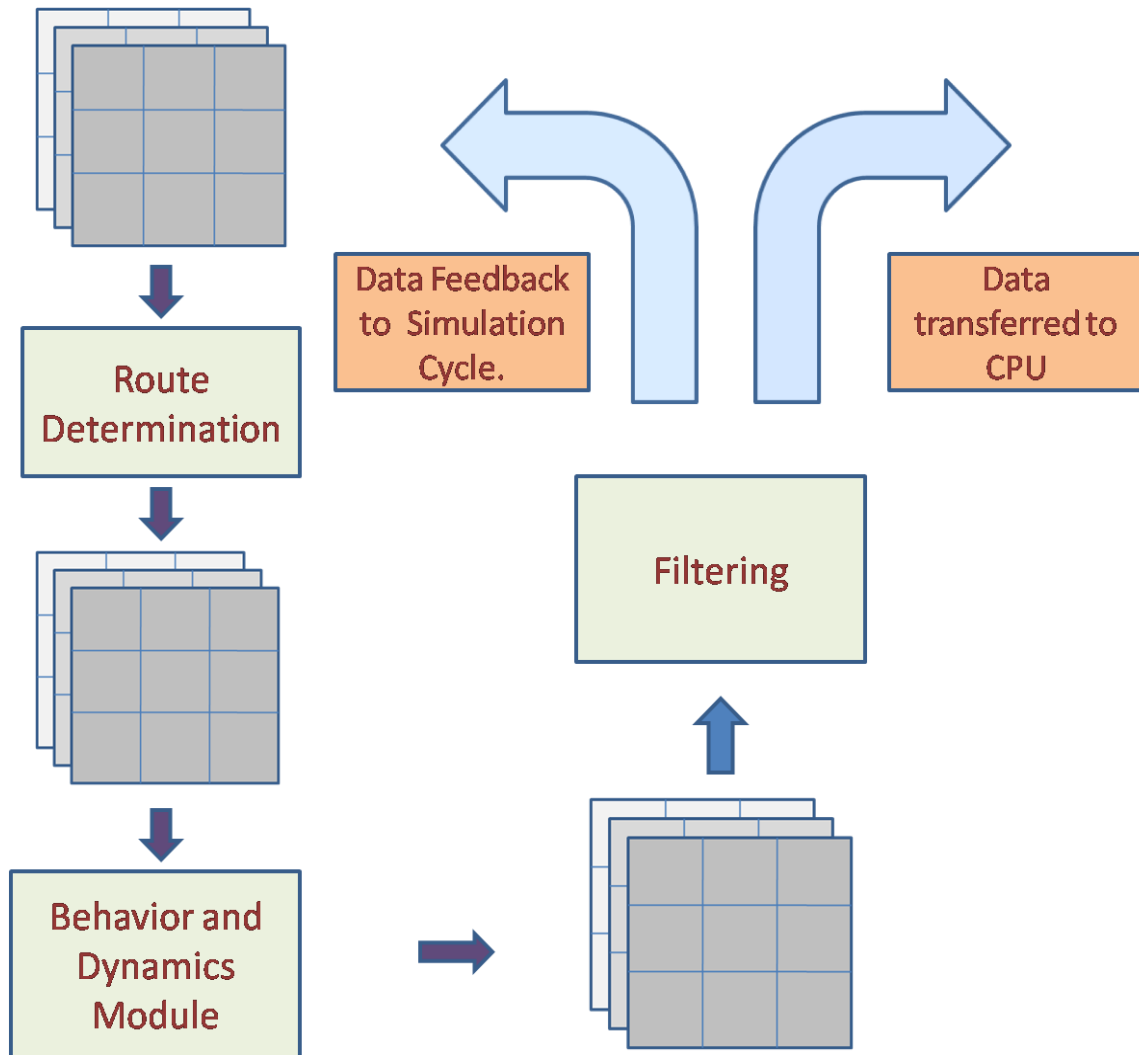


Figure 13: The Organization Simulation Framework.

The simulation framework is implemented in passes with each pass signifying a specific function in simulation. The functionality of the three passes is described as follows:

1. Route Determination.
2. Behavior and Dynamics.
3. Filtering and management of vehicles into CPU and GPU control areas.

### **Route Determination**

Vehicles navigate on the roads defined as parametric three dimensional cubic splines. Vehicles navigate on multiple roads through the course of the simulation. Decision as to the next road to be chosen at an intersection is the goal of this pass. As the vehicle approaches the end of the current road it is navigating on, a decision is randomly made as to which road at the intersection it is to take. A random behavior will suffice the implementation because the goal of the implementation is to generate an ambient behavior that acts as a backdrop to a more precisely calculated and implemented CPU versions. This decision making process is implemented as follows:

1. Check if the vehicle is approaching the end of the road.
  - a. If the decision about the next segment to be chosen at an intersection is already made, do nothing.
  - b. Else choose a segment at random from the available roads at the intersection as the next segment to be navigated.
2. Else if the vehicle is not approaching the end of the road, do nothing.

### **Behavior and Dynamics Module**

A vehicle navigating on a three dimensional road is defined by its behavior. As with the real world, the behavior of a vehicle is defined by following parameters:

1. Magnitude of acceleration or deceleration of the vehicle.
2. Magnitude of the steering angle by the driver.

These two parameters follow from the responsibilities of the driver on the road. A person driving at any time accelerates or decelerates and steers on the road( assuming the car is automatic). These two parameters therefore define the behavior of any driver.

The behavior module tracks the road and determines the magnitude of curvature the vehicle is to be steered to keep it from going off the road. The acceleration of the vehicles is also determined by the vehicle. In this implementation, the acceleration is kept constant throughout the simulation.

The algorithm for the behavior of a driver can be defined as follows.

1. Given the present position of the vehicle, track the road and determine the Cartesian coordinates on the road at distance  $ds$  (look ahead point).
2. Calculate the curvature of the arc drawn from the current position to the look ahead point.
3. The curvature calculated defines the magnitude of steering angle taken by a vehicle.

Based on the curvature and the acceleration, the dynamics module computes the next the position of the vehicle in the virtual space. Simple laws of locomotion are used to simulate the motion of the vehicles.

### **CPU and GPU control areas**

The GPU subsystem works in tandem with its CPU counterpart in defining an approximate behavior to the vehicles running in the backdrop. The CPU simulates accurate behavior of vehicles within a region usually defined to be a sphere. The center of the sphere is the position of the vehicle around which accurate behavior is desired. The transfer of control in simulating the vehicles between the CPU and the GPU framework is the goal for this stage of simulation. This stage filters out the vehicles leaving from the GPU control space into the CPU control sphere. Similarly, vehicles are injected into the GPU controlled areas by the CPU after the execution of the GPU framework as a whole. The algorithm for the filtering stage is defined as follows:

1. For every vehicle defined in the texture, calculate if its Cartesian coordinates are outside the CPU controlled sphere.
  - a. If the Cartesian coordinates are outside the control sphere, do nothing.
  - b. else, filter the coordinates out to the CPU and deactivate the texel corresponding to the vehicle.

The filtering process is implemented in the subsystem by utilizing the geometry shader extension along with transform feedback extension on NVIDIA graphics cards. The filtered attributes of the

vehicles entering into the CPU sphere are transferred to the CPU controlled simulation framework. The transfer of this data across the bus to CPU induces delay proportional to the amount of data to be transferred. This magnitude of the data to be transferred should be kept to a minimum at any time. The whole simulation loop as described in the figure 13 executes once in every single simulation step of the CPU.

## **Chapter 5**

### **GPU Subsystem – Implementation**

The implementation of GPU subsystem has two distinct parts.

1. Definition and initialization of graphics primitives (such as textures) and states of graphics pipeline.
2. Coding of shaders that are injected into the GPU as plug-in to the driver. This shader code defines the operations to be performed on the input data stream.

A very large part in developing the GPU subsystem involves the setting up the format of the data in proper order to be loaded into the 2-D textures. Thus, defining the form and meaning of each texture is significant in understanding the technical implementation. The next section describes the functional categorization of the types of textures and their format.

### **Datastructures and Texture organization**

Textures form the primary datastructures used in the implementation of the GPU simulation framework. Initialization of textures is a onetime process before the start of the simulation. The reading and writing operations to a texture is entirely independent with the CPU simulation framework. The texture data is organized so as to contain the minimal information required for vehicular simulation. Having a big collection of textures implies increased time step for the simulation which is not desired. Based on the use and functionality, the textures in the GPU simulation framework are divided into two types.

1. Data-Dynamic textures.
2. Data-Static textures.

### **Data-Dynamic Textures**

Data-Dynamic textures encapsulate the state of virtual entities that have a dynamic state. The vehicles in vehicular simulation are a good example. The state of a vehicle can be defined to be its position in the three dimensional space, its orientation in the three dimensional space and the like. Shaders are set-up to perform its operation on each texel of the textures. In the GPU

simulation framework, each texel in these texture represents a vehicle. The data-Dynamic textures are as follows:

1. Cartesian coordinates of vehicles in three dimensional space.
2. Orientation of vehicles in three dimensional space.
3. The curvilinear coordinates and the index of multisegment the vehicle is on.
4. The attributes of the vehicles navigating the three dimensional space.

Since, we are working in a setting of stream computing, we would need two pairs of four textures in the format defined above. The state of GPU is set appropriately by using frame buffer objects extensions such that each processing core of the GPU runs a single instance of the shader on each texel of the input texture and writes out the changed state to each of the corresponding textures at the output. Thus, state for each vehicle is updated.

### **Data-Static Textures**

Data-Static textures are the textures that encapsulate the state of the virtual entities that do not have a changing state. In this simple model of vehicular simulation, the roads in the three dimensional space are the static structures. The shape and the orientation of the road do not change with time as the position of the vehicle. In the present implementation of the GPU framework for vehicular simulation, the following entities are encoded as textures that do not change with time.

1. The parametric cubic segments that make up the spine of the three dimensional road.
2. The information about connectivity between the roads and the intersections.

The central spine of the road is defined as parametric cubic spline equations. The Cartesian coordinates and the slope of the road are defined as a function of a parameter  $t$ .

Since each of the cubic equation is defined by four coefficients as shown in the above equation, a single texel is sufficient for holding the four coefficients for each of the following:

1. X-coordinate.
2. Y-coordinate.

3. Z-coordinate.

4. Slope of the road.

A set of four textures are thus defined holding the coefficients of the x, y, z and the slope of each cubic segment. The first texel of each of the four textures would hold the coefficients of the first cubic spline defined in the EDF.

### **Contents of Textures**

Examining the contents of each texture forms an important step in understanding the working and organization of the simulation. The list of implemented textures in the GPU subsystem is defined below:

1. Texture holding the coefficients of cubic equation defining the x-coordinate of a spline segment.

2. Texture holding the coefficients of cubic equation defining the y-coordinate of a spline segment.

3. Texture holding the coefficients of cubic equation defining the z-coordinate of a spline segment.

4. Texture holding the coefficients of cubic equation defining the slope of a spline segment.

5. Texture holding the information about the segments that make up the navigable surface. (Each texel represents a segment that is a group of contiguous cubic splines in the virtual environment)

6. Textures defining the attributes of roads. (Each road or a multisegment is defined to be a group of segments adjacent to each other.)

7. Two textures holding information about multisegments connected to a multisegment at its beginning.

8. Two textures holding information about multisegments connected to a multisegment at its end.

9. A texture holding the curvilinear coordinates and the index of the multisegment it is on. (Conveniently referred to as DOLMS texture)

- 10. A texture holding the Cartesian coordinates of vehicles in the simulation.
- 11. A texture holding the orientation of every of vehicle as quaternion.
- 12. A texture that holds the attribute of every vehicle.

The textures 1 through 8 are the static textures. They represent the nature of the virtual environment. Textures 9 through 12 are the dynamic textures that define the state of vehicles simulated by GPU.

**Representation of Roads**

A spline is defined as parameterized cubic equation. Each of the coordinates is defined in terms of cubic equation as shown as follows:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

$$sl(t) = a_{sl} t^3 + b_{sl} t^2 + c_{sl} t + d_{sl}$$

The equation defines a three dimensional curve along with its orientation as the parameter t increments. Each texel in the textures from 1 through 4 hold the coefficients of the equations described above.

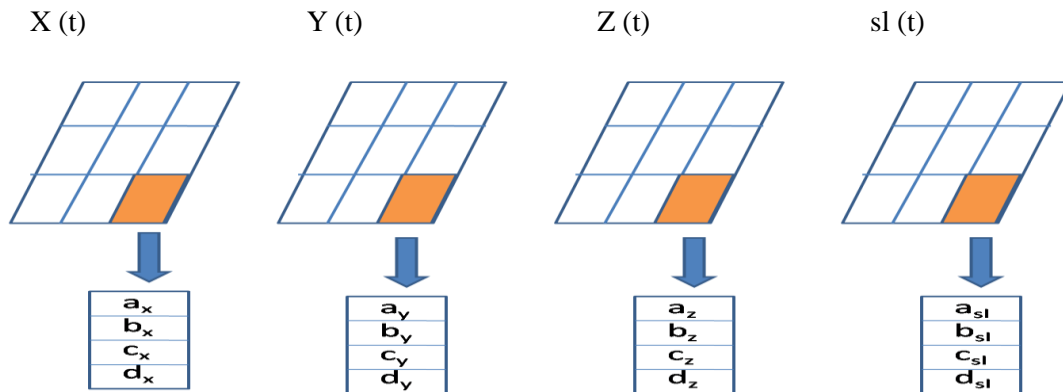


Figure 14: Data Organization in Textures.

The picture illustrates the layout of the textures 1 through 4. The four textures are therefore of equal size and dimensions.

A segment is defined as a group of splines of equal length. In the GPU framework, a segment is defined to be a collection of adjacent texels in the textures from 1 through 4. For example the bottom row of texels shown in the figure above represents a segment.

A single texel in the segment texture (SegmentTex) holds the following information in its components.

1. Start index of the texel that signifies the start of the cubic segment in the segment.
2. The length of segment.
3. The length of each piece of cubic spline that makes up the segment.
4. The distance at which the segment starts within a multisegment.

A two dimensional texture defined the simulation framework are defined to be square textures. A texel in a texture is referenced by pair of two coordinates, that start at lower left corner of the square grid of texels.

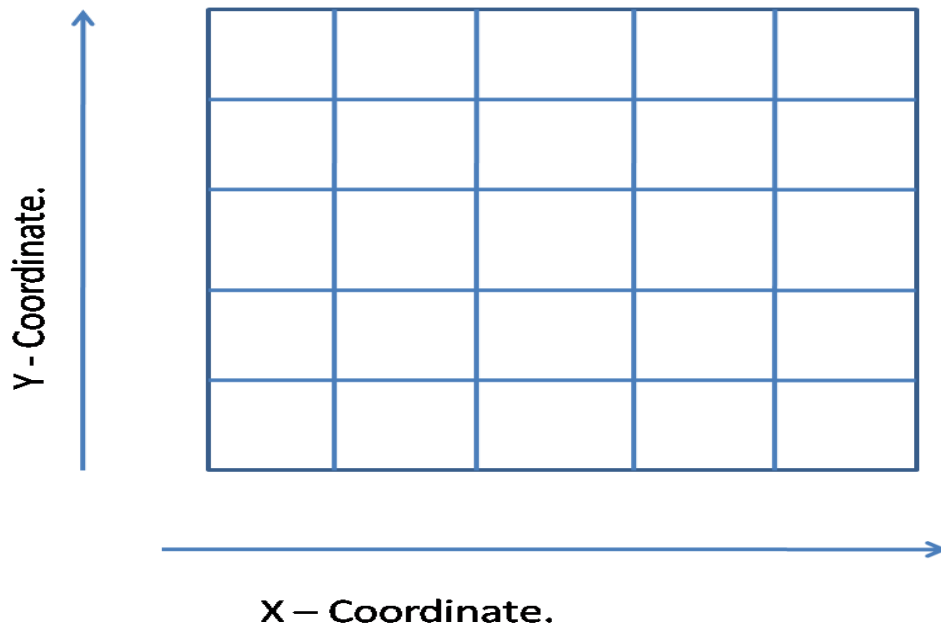


Figure 15: Index Scheme of 2-Dimensional Textures

Each texel can be referenced by a pair of two values. The textures defined in the framework are square in structure. This property lends us a novel way of accessing each texel by a single index number rather than a pair of two values if the dimension of the texture is known. For example, in the figure above, the texture is defined to be a 5 x 5 texture. If the dimension of any one side of the square texture is known ( in this case 5), a number ranging from 0 to 24 (5 x 5) can be used to reference each texel individually. The dereferencing to a coordinate pair from a single value can be done in the following way.

If 'x' represents the index then, the row index (or the y-coordinate) and the dimension of the texture is d. then the coordinates are given by,

Row index =  $\text{floor}(x/d)$ .

Column index =  $x \bmod d$ .

The coordinate pair therefore are, (  $\text{floor}(x/d)$  ,  $x \bmod d$ ).

This dereferencing scheme is extensively used in the shader to infer the coordinates and cubic equation of the multisegment the vehicle is navigating upon.

The segment texture defined in the simulation framework is a binding agent that ties together segments of navigable surfaces defined by three dimensional cubic splines. The segment texture is the static-data structure that remains unaltered throughout the simulation.

### **Multisegment Texture**

A multisegment in the simulation framework represents the structure and attributes of road in simulated environment. The multisegment texture holds the attributes of all the multisegments defined in the EDF. A single texel in the texture represents a multisegment. A road in the EDF is characterized by multiple lanes that channel traffic into streams. Each lane in the road defined in the EDF is associated with an offset that defines its relative position with respect to the central spine of the multisegment. The offset is defined to positive to the right side of the multisegment and defined negative to the left side of the multisegment. A multisegment is composed of several lanes each differentiated with the magnitude of the offset from the central spine. In the GPU implementation of the simulation framework, each lane itself is considered to be a multisegment. Such design decision is made because the task of defining a data structure (texture) that describes

the interconnection among multisegments is made simple and intuitive to implement. This leads to redundancy of texels in the multisegment texture. The benefit of this design however in terms of its ease of use and gain in efficiency justifies it. This design is efficient as there is no additional description to be encapsulated into a separate texture that describes the characteristics of each multisegments housed in each texel of the texture. The following information is stored in each texel of the multisegment texture.

1. The signed offset of the multisegment (representing a lane) from the spine of the multisegment.

The magnitude of the offset defined in the texel is used to conversion functions. It is used extensively in the conversion from curvilinear coordinates to Cartesian coordinates (defined as queryPT in the implementation).

2. Index of the texel in the segment texture that marks the start of the multisegment.

The start index number marks the texel that signifies the segment at which the multisegment starts.

3. Index of the texel in the segment texture that marks the end of the multisegment

The end index number marks the texel that signifies the segment at which the multisegment ends. All the segments defined between the start and the end segment indices are the part of the multisegment.

4. Length of the multi-segment.

The length of the multisegment is stored in the fourth component in the multisegment.

### **Implementation of Intersection**

An intersection in an EDF is an structure defined to interconnect multisegments. An intersection in the EDF resembles the real world intersections. An intersection is made of a collection of “corridor” which is basically a multisegment with additional properties. A corridor connects the lanes in the multisegment at one side to any one of the multisegments defined on the other side. A corridor is its basic meaning and functioning is a multisegment itself. In the GPU simulation framework

## Link Textures

Having defined multisegments or roads for the vehicles to navigate on, defining constructs that describe the interconnectivity between the multisegments is equally plays a major role. In the real world, humans use visual aids to notice the end of a road when approaching an intersection. At the intersection, a driver guides himself to the appropriate lane through the intersection and to the next lane of the connecting road basing the decision on purely based on vision. However, in simulated environments this action is accomplished by setting up appropriate and efficient constructs used for lookup and decision making.

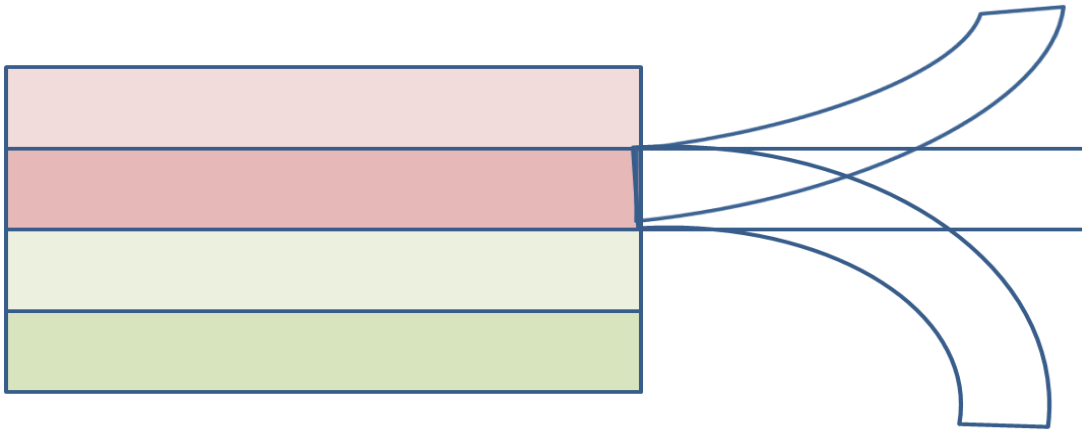


Figure 16: Lanes and Intersection Layout

The above figure illustrates the lanes of the road, each marked with a different color. As an example, the possible multisegments following a lane in a multisegment are shown. Thus, each lane on the roads begins or ends with a set of multisegments as shown.

A simple, fast and efficient way to describe such connectivity among multisegments is desired. A very simple way is to declare textures of similar size to that of multisegment texture and populate it with indices of multisegments that connect from its corresponding texels in the multisegment texture. A one to one mapping between such link texture and the multisegment texture would reveal the multisegments connected to a particular texel. Since there are two ends to a multisegment, two set of textures one for each end will suffice to define connectivity among multisegments.

Each texel is capable for holding up to four values. Therefore, declaring two textures, one for each side of multisegment will hold up to eight connecting multisegments.

## **Vehicle Attributes**

The attributes of the vehicles form the only dynamic aspect of the GPU simulation. Four textures are defined that represent the current state of vehicles.

### 1. Curvilinear coordinates.

This texture holds the curvilinear coordinates of every vehicle in the first three components of every texel. The fourth component of the texel contains the index number of the multisegment it is currently navigating on, with respect the multisegment texture.

### 2. Cartesian coordinates

The Cartesian coordinates of every vehicle corresponding to its curvilinear coordinates is contained in this texture. The three components of the texels are the Cartesian coordinates. The fourth component is assigned a value of one. Information in each texel of the texture is later taken and drawn on the screen.

### 3. Orientation

An orientation of every vehicle can be represented by three 3-D vectors. However, such a representation requires 9 components for every vehicle. A compact form of representing orientation is by defining a quaternion. The four components of the quaternion form entries into each texel in the texture.

### 4. Attributes

Every vehicle is characterized by attributes. The four attributes required by vehicles for the simulation are as follows:

1. Speed of the vehicle.
2. Acceleration of vehicle.
3. Direction of vehicle navigation with respect to the implicit direction of current multisegment.
4. Index of next multisegment to be navigated.

The four textures that represent the state of the vehicles are of similar size and dimension. Attributes of any one vehicle is given by one to one mapping of the texel in the one texture to other three textures.

**GPU Simulation Algorithm**

The algorithm for the implementation is has two distinct phases:

1. OpenGL commands.
2. Computations defined in shader.

Once the initialization phase of setting up textures is completed, appropriate states are set and commands are issued in OpenGL that signals the start of a single simulation step. A single simulation step consists of four distinct passes. Each pass performs a set of computation on the input stream and channels the output stream to set textures. This is depicted in pictorially as follows:

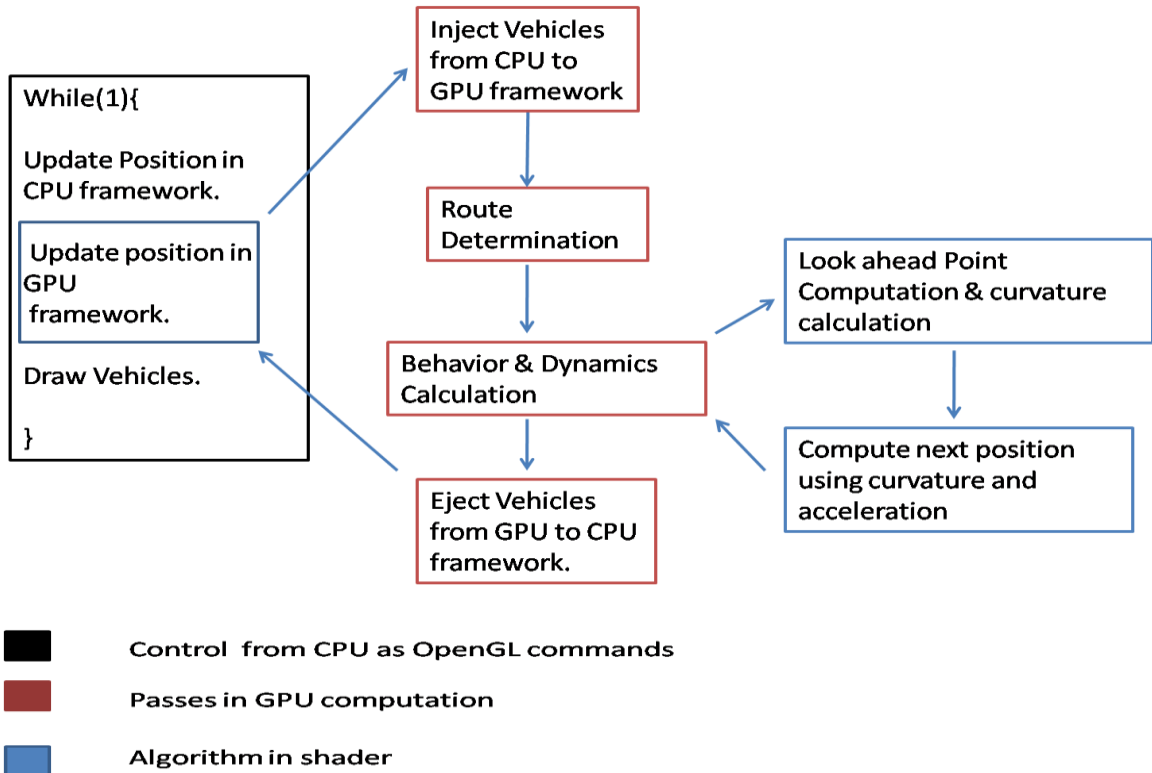


Figure 17: Multi Pass Simulation Framework

The GPU simulation is designed and implemented to work in tandem with its CPU counterpart. The simulation progresses as an endless loop until the user commands the system to quit. The CPU having completed its duties initiates its GPU counterpart. The appropriate states that are set are explained as follows:

1. Bind a total of eight textures (four textures representing the initial state of vehicles and four textures that capture the updated state of the vehicles) to an initialized framebuffer object. (Framebuffer object extension in OpenGL allows a programmer to perform OFFLINE RENDERING. Offline rendering allows a program to render the output to a texture memory.)
2. Modify the viewport to the dimensions of the vehicle attributes texture to facilitate one-to one mapping from input textures to output textures.
3. Invoke the four passes with appropriate openGL states (Each of them explained below).
4. Render the vehicles in the CPU & GPU framework.
5. Until command for termination issued loop back to step 1.

The four passes composing the GPU framework for vehicular simulation is explained as follows:

### **Injection into GPU Framework**

The simulated environment consists of areas controlled by CPU and by the GPU framework. Vehicles are characterized by constant motion, enter and leave GPU controlled area to CPU controlled areas and vice-versa. As vehicles leave CPU simulated areas into GPU simulated area, an entry is made into the set of textures, thus transferring the load due to simulation onto the GPU. This injection of the simulation control of vehicle is supported by existing data structures that maintain indices that are vacant texels in a texture. The algorithm for the implementation is described below:

1. Determine the location of the texel available for injection of vehicle attributes (by accessing the “vacancy stack” and determining if it is non-empty).
2. Modify the viewport to 1x1 dimension (This would allow us to write to a pre-determined texel in the texture.) and place it right over the texel to be modified.
3. Pass the vehicle attributes as uniform variables into the shader.

4. In the fragment shader, update the attributes texture by copying the values from the uniform variable into it.

The attributes of the vehicles are now injected into the data structures and are simulated by GPU subsystem.

### **Route Determination**

As the vehicle approaches the end of the multisegment, a decision to navigate a connecting multisegment is made. Such a decision is the purpose of this pass. This algorithm describing this pass is described below:

- a. Check if the end of current multisegment is at least ' $d$ ' units away.
- b. If yes, do nothing.
- c. Else, Check if next multisegment to be navigated is decided in the previous step.
  1. If yes, do nothing.
  2. Else, do following,
    - a. Access the multisegments connected to the approaching end by accessing the link texture of the current multisegment.
    - b. Randomly select one of the available multisegment and determine which end is connected to the current multisegment at the juncture. (The multisegment can be connected from its begin or its end).
    - c. If the next multisegment begins at the juncture, update the fourth component of the attributes texture with the index of the multisegment chosen (with respect to the multisegment texture).
    - d. Else, update the fourth component of attributes texture with the negative value of the index of the chosen multisegment texture.

This pass updates only the fourth component of the attributes texture.

## **Behavior and Dynamics Calculation**

The goal of this pass is to move the vehicles one step forward on the multisegment. This is accomplished in two phases.

1. Behavioral phase.
2. Dynamics phase.

### **Behavior Phase**

A behavior of a vehicle is defined by two parameters:

1. The acceleration or deceleration of the vehicle.
2. The magnitude of the steering angle or curvature adopted by the vehicle.

The acceleration of the vehicle is kept constant. The steering angle of the vehicle determines the curvature of vehicle's locomotion. The present implementation supports basic a lane tracking mechanism. Vehicles navigate in their respective lanes only. Lane changes and occupancy control is not implemented. The curvature or the steering angle is estimated by looking ahead 't' units on the present lane and using the orientation of the road. The detailed procedure is given in Willemsen et al (2006). The shader program accesses the textures that define x, y, z, and slope of the splines to determine the required parameters.

### **Dynamics Phase**

After determining the curvature, the vehicle is moved by a distance  $\Delta d$ . This distance is governed by the kinematics equation,  $\Delta d = ut + \frac{1}{2} at^2$ . Where,

'u' is the initial velocity.

't' is the time step.

'a' is the acceleration.

The acceleration and orientation guides the motion of the vehicle along the multisegment. The querying mechanism of EDF is implemented in the shader program. The convergence algorithms

such as Newton's convergence and quadratic minimization methods are implemented. These algorithms are similar to its CPU counterpart. However, these algorithms are optimized with minimum number of branching and iterative loops.

Having calculated the new position, the orientation of the vehicle, and its curvilinear coordinates are updated and streamed into the four textures bound to the frame buffer object. The second pass thus updates the following parameters of the vehicle.

1. Cartesian coordinates.
2. Curvilinear coordinates.
3. Orientation of the vehicle.
4. The attributes of the vehicle.

The algorithm for the second pass executed for every vehicle can be summarized as follows:

1. Estimate the curvature by calculating the look ahead point.
2. Using a pre-defined value  $\Delta t$ , compute the distance covered in a single simulation step.
3. Compute the new position of the vehicle using the spline equations.
4. Update the curvilinear coordinates by using an efficient mix of convergence algorithms ( A mix of quadratic minimization and Newton's method).
5. Update the state of the vehicle at the output stream.

### **GPU-CPU Vehicle Transfer**

Over the course of the simulation, vehicles navigate in and out of CPU and GPU controlled regions. A CPU controlled regions is defined by three dimensional spheres in a virtual space. The final pass of the simulation filters out the vehicles entering the CPU simulated region from the GPU framework. The vehicles entering these pre-defined CPU controlled regions are no longer simulated by the GPU framework. The algorithm for this pass is described as follows:

1. Set the state of texture memory representing the Cartesian coordinates of vehicles as a vertex buffer.

2. Invoke the geometry shader on the vertex buffer after the setting the appropriate state and output buffers for the use of transform feedback extension. The geometry shader is configured to accept vertices as inputs and emit a single triangle for every vertex. (The rationale behind emitting a single triangle for every vertex is that, the attributes of the vertices of the triangle contains the state of vehicle selected for transfer into the CPU framework).

3. The geometry shader performs the following operations:

a. Access the Cartesian coordinates of the vehicle disguised in the form of input vertex and check if it lies inside a CPU control region (A CPU control region is defined by a three dimensional point and pre-defined radius).

i.e. the point  $(x_p, y_p, z_p)$  lies inside a sphere with center  $(x_c, y_c, z_c)$  with radius  $r$  if,

$$(x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2 < r^2$$

b. If the above vehicle is determined to lie inside the CPU controlled sphere, then access the three remaining textures that represent the state of the vehicle.

c. Assign the attributes of vehicles as value for the vertices to be drawn and emit the triangle from the shader (Therefore, each triangle emitted represents the attributes of the vehicles filtered into the CPU framework from its GPU counterpart).

d. The following format maps the coordinates of the vertices to its corresponding vehicle attributes.

**First Vertex:**

X – Coordinate: x-coordinate of the vehicle.

Y – Coordinate: y-coordinate of the vehicle.

Z – Coordinate: z-coordinate of the vehicle.

W-Coordinate: A value of 1.

**Second vertex:**

X – Coordinate: Speed of vehicle.

Y – Coordinate: Acceleration of the vehicle.

Z – Coordinate: Direction of the vehicle with respect to the implicit direction of the vehicle.

W-Coordinate: The index of next multisegment to be navigated.

**Third Vertex:**

X – Coordinate: First coordinate of quaternion representing the orientation.

Y – Coordinate: Second coordinate of quaternion representing the orientation.

Z – Coordinate: Third coordinate of quaternion representing the orientation.

W-Coordinate: Fourth coordinate of quaternion representing the orientation.

e. The attributes of emitted vehicles are captured in the GPU memory.

4. The contents of the GPU buffer is copied to a memory location on the main memory. The data copied to the memory location is then accessed to maintain to retrieve the attributes of vehicles.

5. The index of the texel from which the filter was made is maintained in a stack. The stack represents locations in the textures to which a vehicle can populated to be simulated by the GPU framework.

The updated position of every vehicle is rendered on the display. The GPU simulation framework is thus a plug-in to the original simulation.

**Queries**

The support of local curvilinear coordinates expressed in terms of distance from the start of the spline segment, offset from the center spine, and loft on the road makes behavioral decisions of navigating vehicles easy to implement and code. On the other hand, the dynamics calculations are best expressed in Cartesian coordinates. The high frequency conversion between the frames of reference requires the implementation to be efficient and robust in real time. Conventional convergence method that estimates the closest point on the spline segment to a given point in a three dimensional space are limited by the quality of inputs. The quadratic minimization method

is characterized by high convergence rate in the initial iterations but slows down considerably as optimal value is approached. Newton minimization method converges rapidly to an optimal value if the initial estimate is closer to the optimal value. Detailed experiments on convergence rate have been elaborated in Wang et. al. (2002b). Wang et. al. (2002b) describes a method that uses complementary strengths of the two methods to arrive at the optimal value in real time quickly. This method is used in driving simulators based on EDF.

The convergence method is implemented in the GPU subsystem similar to its counterpart implemented in the CPU framework. The implementation of quadratic minimization and Newton's minimization have been optimized and tested for accuracy.

### **Premises-Queries**

A single simulation step updates the position of the vehicles in real time based on the values of its control parameters. The control parameters include acceleration and speed. As described in figure 17, the end of the behavioral step sets the value of curvature to be followed at each step and its acceleration. These values of acceleration and speed are used for navigation of the vehicles its intended direction.

The dynamics step computes the new position of the vehicle based on its acceleration and speed. These computations are performed in Cartesian frame of reference. However, the curvilinear coordinates and the orientation of the vehicles need to be updated for the new position of the vehicles. The change in the orientation of the vehicles is calculated without much computational steps. However, determining the curvilinear coordinates of the updated Cartesian coordinates requires the use of convergence algorithms to determine the spline segment that is closest to the updated position of the vehicle.

This is the premise for the use of convergence algorithms. The nature and implementation of the convergence algorithms is explained in the next section.

### **Implementation**

The implementation of the algorithms for convergence is two folds.

1. The application of quadratic minimization methods to give a good initial estimate to the Newton's method.

2. The application of Newton's convergence algorithm until an optimal solution of acceptable magnitude of error to its optimal value is obtained.

The centerline of a curved road modeled as parametric cubic spline is expressed as,

$(x(s), y(s), z(s))$ , where  $0 \leq s \leq L$  where  $s$  denotes the arc-length and  $L$  is the length of the spline curve.  $x(s)$ ,  $y(s)$  and  $z(s)$  are cubic spline functions.

The equation representing the distance between a point on a spline curve and a point in space,  $P_0(x_0, y_0, z_0)$  is given below:

$$D(s) = (x(s) - x_0)^2 + (y(s) - y_0)^2 + (z(s) - z_0)^2$$

The value  $s^*$  that minimizes  $D(s)$  determines  $p_1 = (x(s^*), y(s^*), z(s^*))$ , the closest point to  $P_0$  on the spline curve. The mapping algorithm to curvilinear coordinates from its corresponding Cartesian coordinates is modeled as an optimization problem, minimizing the value of  $D(s)$  by finding an optimum value of  $s^*$ .

### **Quadratic Minimization**

Quadratic minimization uses quadratic interpolation to minimize  $D(s)$ . If  $s_1$ ,  $s_2$ , and  $s_3$  are given as initial estimates of  $s^*$  that optimizes the value of  $D(s)$ . Minimizing quadratic polynomial that interpolates  $D(s)$  at  $s_1$ ,  $s_2$ , and  $s_3$  is used to approximate the minimum of  $D(s)$ ,  $s_1^*$ . We then pick three values from  $s_1$ ,  $s_2$ ,  $s_3$ , and  $s_1^*$  eliminating the value that is farthest from spline segment. This method is continued until an optimum value is estimated. The algorithmic details of the quadratic minimization is explained in Wang et.al.(2002b). The rate of convergence is initially very high. It has been found after rigorous testing for different shapes and layout of multisegments that four iterations of the algorithm yields good estimate of the final value. The use of lesser number of iterations may lead to a coarse estimate of the final value. This may lead to troublesome scenarios if the multisegments navigable are very long in length. The use of longer multisegments with multiple local minima poses a significant challenge to this algorithm. It has been experimented that four iterations for the algorithm is an ideal number iterations for a good estimate. The computed approximate value of the curvilinear coordinates is tested for consistency. The consistency determines if the multisegment for which the convergence algorithm is applied is one closest to the Cartesian coordinate. If the multisegment is found not to be consistent, next multisegment is checked for consistency.

### **Newton's Method**

The value of  $s^*$  that minimizes  $D(s)$  satisfies  $D'(s^*) = 0$ . Newton's method is then used to find the roots of this equation. The algorithmic implementation of Newton's Method is described in Wang et.al.(2002b). The GPU supports floating point computations and does not support double precision computational capability. The GPU cannot perform computations for numbers less than  $10^{-7}$ . This benchmark is found after extensive testing on the GPU using GLSL shading language. This limits the accuracy of the convergence algorithm applied. The application of Newton's method to obtain a final value to the accuracy of  $10^{-7}$  is practically infeasible as it leads to underflow conditions. Application of any arithmetic operators like multiplication, addition, subtraction and division of precision of numbers less than  $10^{-4}$  is found to cause underflow errors. While some arithmetic operations of this magnitude may lead to a correct answer, the result was found to contain a garbage value for other operations. Due to this constraint the implementation of convergence algorithm halts if the magnitude of the error of the estimate to the optimum value is less than  $10^{-4}$ . The Newton's method is applied for 10 iterations but is stops if the acceptable error is less than  $10^{-4}$ . It is found upon experimental testing that the maximum number of iterations required to achieve acceptable level of accuracy was four. The Newton's method is observed to perform an accurate estimate of the curvilinear coordinates up to the order of  $10^{-4}$ .

## Chapter 6

### Results

The GPU framework is designed and implemented to work independently without any transfer of state information or data for a single simulation step. The number of vehicles navigating the virtual environment is significantly increased from 200 vehicles up to 20,000 vehicles in real time. The graphs shown below represent the time taken by each module in the GPU simulation framework.

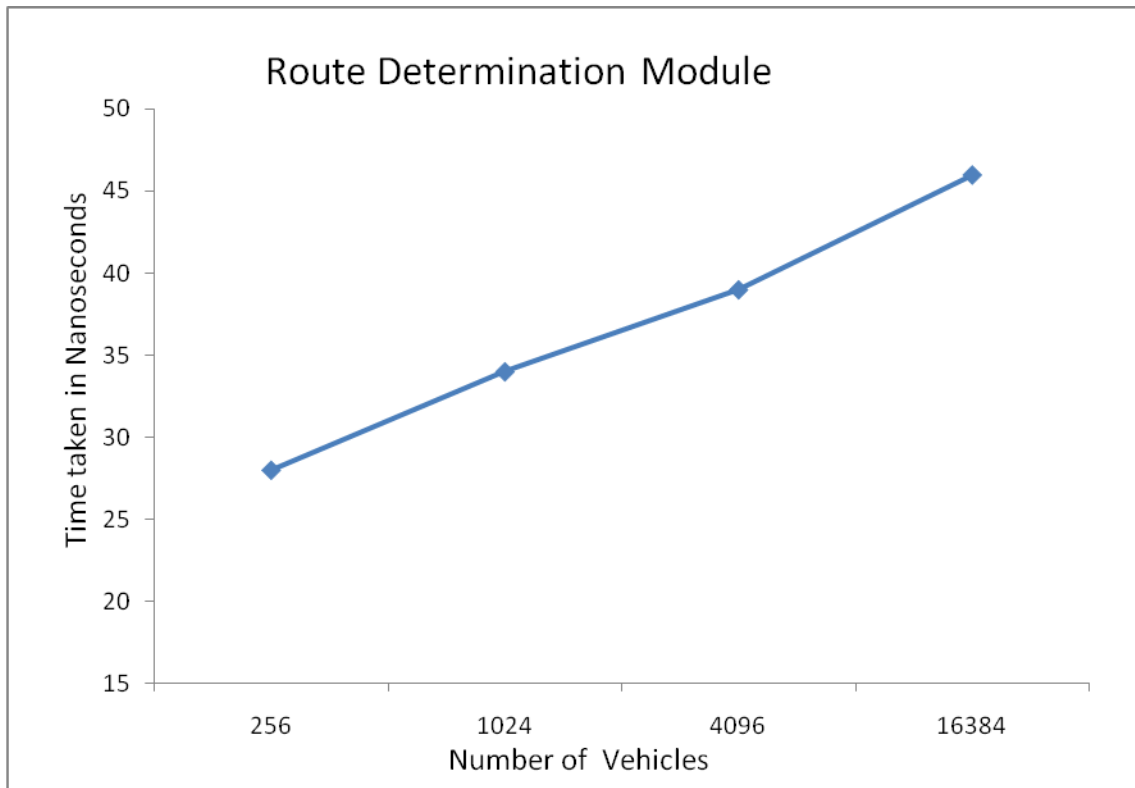


Figure 18: Time taken for Route Determination Module for a single simulation step.

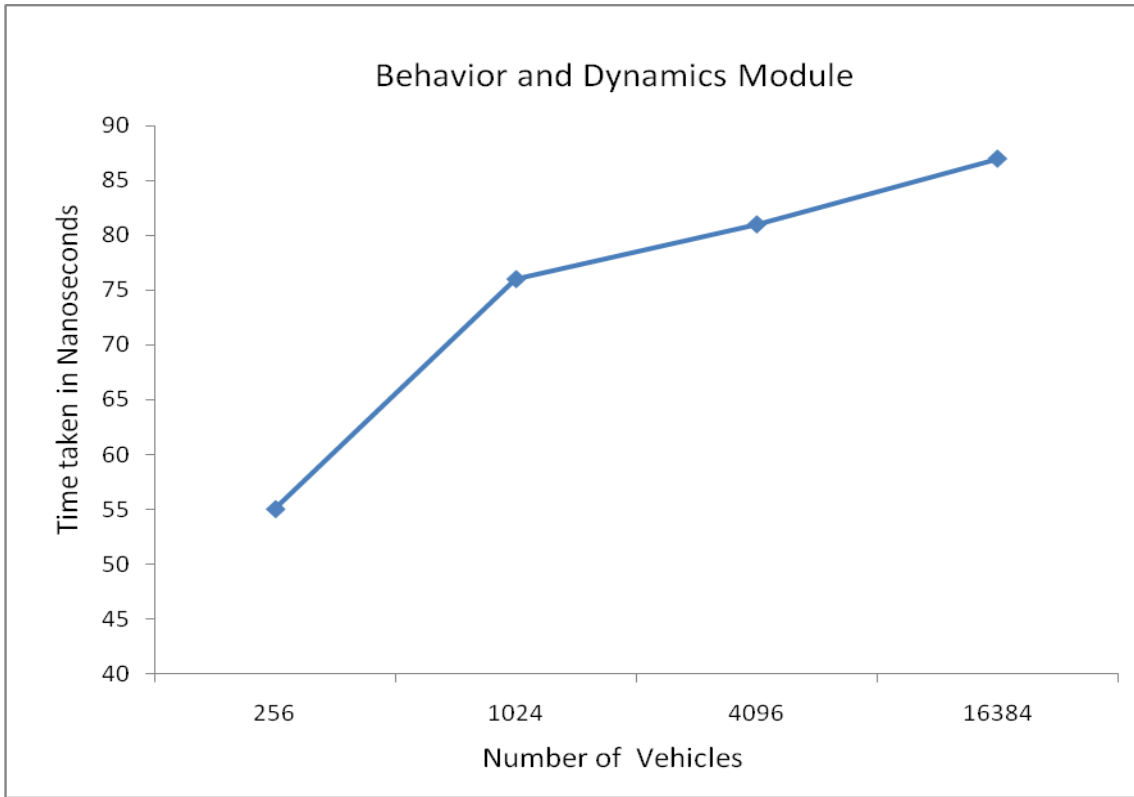


Figure 19: Time taken for Behavior and Dynamics module for a single simulation step.

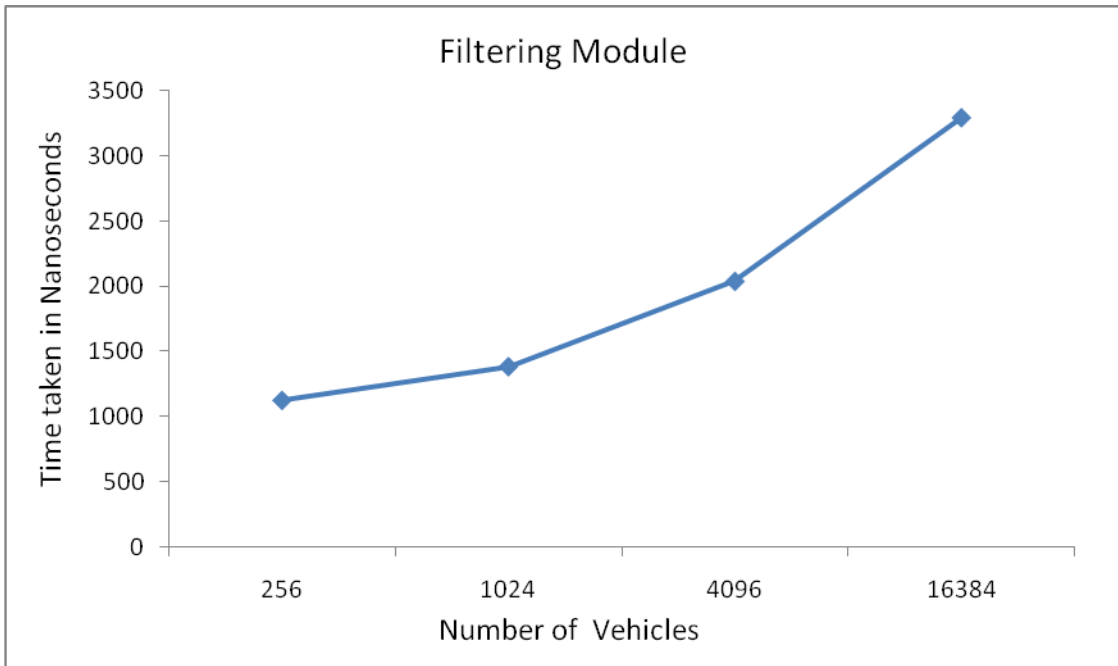


Figure 20: Time taken for Filtering Module for a single simulation step.

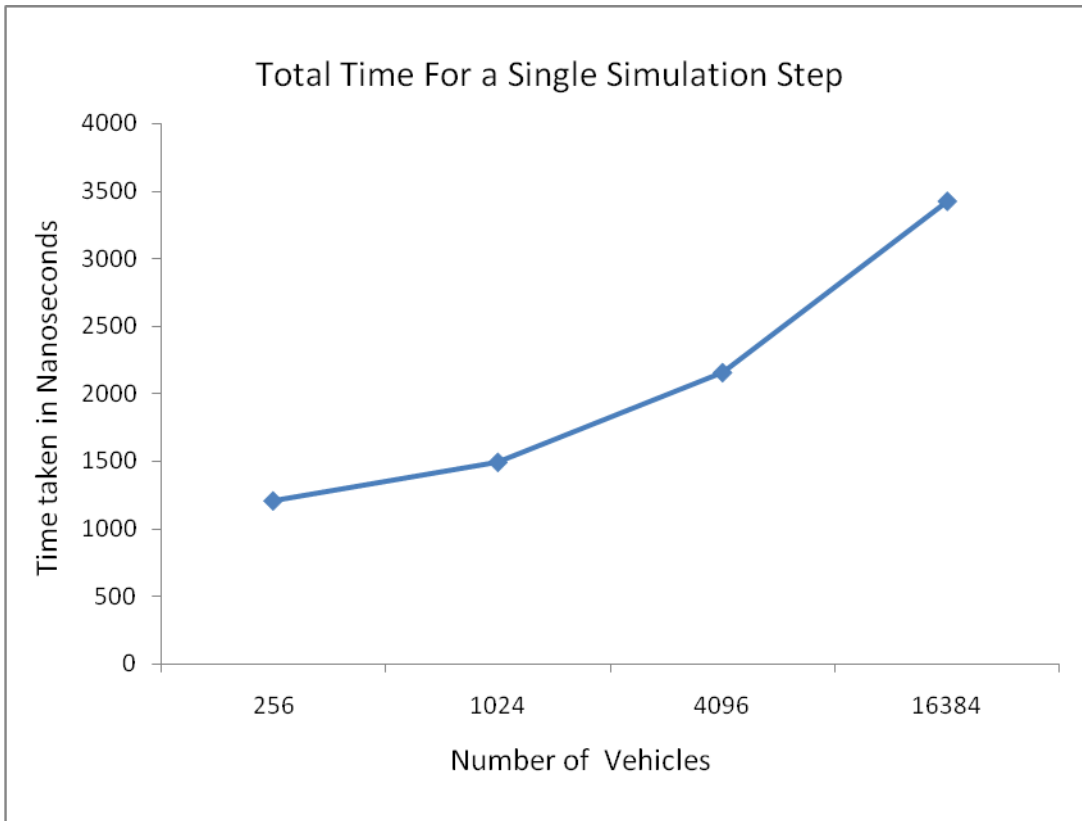


Figure 21: Time taken for a single simulation step.

In the present simulation model, the framework can support up to 16,384 vehicles in real time. The vehicles navigating the roadways negotiate at intersections. The core of the simulation framework, the Newton's minimization and Quadratic minimization techniques are implemented efficiently and tested for robustness.

Another prominent feature of the simulation model is the interaction between the CPU framework and the GPU framework. Mechanisms have been implemented to support the transfer of vehicles between the GPU controlled regions and the CPU controlled regions. This functionality implemented in the final module, is slowest among the three modules because of the data transfer between the CPU and the GPU over the data bus. However, the framework is stable and works in real time.

## **Conclusion**

The use of commodity graphics processing units as an inexpensive alternative for high performance computations is demonstrated by this research work. The graphics processing units are equipped with high ratio arithmetic logic units to control units. This design characterizes the massive computational capability. The graphics processing units are more suited for data-intensive rather than control – intensive applications.

A massive scale up to the order of 50 is achieved without any significant computational load on the CPU. The framework operates in real time at hard frame rates with no significant change in simulation speed.

The EDF structure is redesigned in this implementation. The most important and significant simplification is the flattening of hierarchical organization of navigable roads into a single flat structure packed into the textures. The query mechanisms are also optimized to efficiency in the GPU simulation framework. This optimization can be used to improve the performance of its CPU counterpart.

Redesigning an algorithm to be implemented on Graphics Processing Units is not particularly straightforward. It is an intermix of algorithm and data accesses. In general the following are to be considered when an GPU subsystem is built using OpenGL and OpenGL shading language.

- Is the algorithm data-intensive or control intensive?
- Does it need a frequent communication and control from application running on CPU?
- The level of parallelization intended.

## **Future Work**

The implementation of lane tracking and decision making at intersection is demonstrated in this research work. The fundamental framework required for a vehicular simulation based on EDF is built by using textures as primary data-structures. These data-structures represent the bare-minimal but sufficient infrastructure for vehicular simulation. Any future redesigns of the EDF to suit a given programming model such as an application in CUDA can be designed based on these basic datastructures.

The Vehicles navigating in the virtual space are not however aware of its surrounding vehicles. Such an awareness falls within the realms of collision detection. A pass before the start of the GPU simulation cycle can be introduced to hold information about the occupancy of the roads.

OpenGL architecture is designed to operate in server-client paradigm. Graphics processing units mounted on other desktops can be networked together to provide additional computational resources to the single desktop or workstation. Since the application runs on the client (i.e. The system that is equipped with display and requests the processing of openGL commands from graphics processing units that act as servers), multiple Graphics processing units can augment the current simulation framework.

Also, minimizing the query calls made in the GPU simulation framework is absolutely essential as context switches and method invocations inside the shader program results in notable slow down in the execution of the simulation.

The vehicles are rendered on the display as points. Point sprites can be used to add realism to the scene as opposed to points circling on roads.

## **Bibliography**

Buck, Ian., "GPU Computing: Programming a Massively Parallel Processor," *Code Generation and Optimization, CGO' 07, International Symposium on*, (2007), (April 2007), pp.17-17.

Cremer, James., Kearney, Joseph., and Willemsen, Peter., "A Directable Vehicle Behavior Model for Virtual Driving Environments", Proceedings of 1996 Conference on AI, Simulation, and Planning in High Autonomy Systems,2006.

Rost, Randi J., Kessenich, John M., and Lichtenbelt, Barthold., *OpenGL Shading Language*, Addison-Weasley, 2004.

Shreiner, Dave., Woo, Mason., Neider, Jackie. and, Davis, Tom., *OpenGL Programming Guide*, Addison-Weasley, 2006.

Trancoso, Pedro., and Charalambous, Maria., "Exploring Graphics Processor Performance for General Purpose Applications," *8<sup>th</sup> Euromicro Conference on Digital System Design (DSD'05)*, (2005), pp. 306-313.

Wang Hongling., Kearney, Joseph K., and Atkinson, Kendall, "Robust and Efficient Computation of the Closest Point on a Spline Curve," *Curve and Surface Design*, Lyche, Tom., Mazure, Marie-Laurence., and Schumaker, Larry L. (eds. 2002b), pp.397-405.

Wang Hongling., Kearney, Joseph., and Atkinson, Kendall, "Arc-length Parameterized Spline Curves for Real-Time Simulation," *Curve and Surface Design*, Lyche, Tom., Mazure, Marie-Laurence., and Schumaker, Larry L. (eds. 2002a), pp.387-396.

Wang, Hongling., and Kearney, Joseph K., "A Parametric Model for Oriented, Navigable Surfaces in Virtual Environments," *ACM International conference on Virtual Reality Continuum and its applications*, (2006), pp . 51-57.

Wang, Hongling., Kearney, Joseph K., Cremer, James., and Willemsen, Peter, "Steering Autonomous Driving Agents Through Intersection in Virtual Urban Environments," *International Conference on Modeling and Visualization Methods*, (2004).

Wang, Hongling., Kearney, Joseph K., Cremer, James., and Willemsen, Peter, "Steering Behaviors for Autonomous Vehicles in Virtual Environments," *IEEE Virtual Reality Conference 2005 (VR 2005)*, ( 2005), pp. 155-162.

Willemsen, Pete., Kearney, Joseph K., and Wang, Hongling, "Ribbon Networks for Modeling Paths of Autonomous Agents in Virtual Environments," *IEEE Transactions on Visualization and Computer Graphics*, 12:3(May/June 2006), pp.331-342.