MODELING SENSORS AND THREATS IN A THREE-DIMENSIONAL REAL-TIME SIMULATION OF A SEAPORT ENVIRONMENT

By

ALLEN CHRISTIANSEN

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
Department of Electrical Engineering and Computer Science

MAY 2009

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of ALLEN CHRISTIANSEN find it satisfactory and recommend that it be accepted.

Lawrence Holder, Ph.D., Chair

Diane Cook, Ph.D.

Robert Lewis, Ph. D.

ACKNOWLEDGMENT

I would like to acknowledge the help and guidance that my graduate advisor Professor Lawrence Holder gave me throughout the entirety of the project. He helped me to look at problems I was encountering from a different angle, and encouraged me to explore new ideas I had on how to improve the project, regardless of how strange they sounded. I would also like to thank him for having patience with me when I procrastinated making the progress I should have been. I would also like to acknowledge the help of Dr. Cook, and Dr. Lewis in providing me with areas of research I overlooked in order to make the project more complete.

MODELING SENSORS AND THREATS IN A THREE-DIMENSIONAL

REAL-TIME SIMULATION OF A SEAPORT ENVIRONMENT

Abstract

by Allen Christiansen
Washington State University
May 2009

Chair: Larry Holder

Performing on-site testing for security configurations for a seaport can be quite costly both from a financial, and security perspective due to the effects that testing an ineffective configuration could have on both the efficiency and safety of the seaport in question. In order to alleviate these costs a simulation can be used to analyze new security configurations without disrupting normal operations. We postulate that such a simulation can provide enough fidelity that, even with performance limitations, an accurate evaluation of sensor configurations can be provided. We present the development of the particle emitter and sensor system which is used in PortSimulator, a three-dimensional simulation environment which can help security professionals perform cost effective analysis on configurations of passive sensors in a seaport environment. The main focus for our work was to try to develop a system which could provide the most accurate information while dealing with system performance constraints. Our tests for the different systems focus primarily on the aspect of performance. We address how the accuracy of our two primary propagation systems degrades when the number of particles fired at it increases. We also address ratios of sensors and emitters in regards to at what point does the number of sensors or emitters cause the system to perform so poorly that it is no longer usable.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter One

Introduction

When considering security, it is often too costly or dangerous to perform on-site testing of new security measures. However a simulation system can be used to perform this testing. Our goal was to develop a simulation that can provide a high enough fidelity, despite performance limitations, to obtain an accurate evaluation of security configurations namely that of passive sensors. Therefore we introduce PortSimulator [17], a three-dimensional real-time simulation environment of a seaport in order to help analyze the effectiveness of passive sensor layouts in order to help those in charge to perform a more cost effective analysis of their security configurations. Having this simulation run in real-time helps users to understand problem areas of their configurations and what the source of those problems might be. Having the simulation run in three-dimensions helps users to better understand the environment that they are interacting in and provides the users with a more robust simulation.

An important part of that simulation is the modeling of both the threats and sensors in the environment in order to achieve a level of accuracy sufficient enough as to provide the users with a level of confidence in which they feel safe in using this simulation in assessing their security. This pertains to a number of different areas that must be focused on in order to achieve the previously stated result. The first of those areas is to achieve a high accuracy with how the particles emitted from the threats interact with the environment. This includes the physics simulation run upon the particles in a manner that appropriately represents reality. The second area of focus is that of the method in which the emitters release particles into the environment, in order to ensure a real-time simulation while representing reality in a way that is helpful for the users of the simulation. The last main area to focus on is that of modeling the sensors in a manner that is helpful to the user. This includes having the sensors provide appropriate feedback to the user with their status, including logging the data that they collect in any given simulation run.

The environment where the simulation is being developed is that of the Torque Game Engine [TGE]. The TGE provides us with a platform to develop our simulation, and provides us with visualization and collision detection tools. The TGE is primarily a first person shooter [FPS] game engine developed by Garage Games as a modified version of a 3D computer game engine originally developed by Dynamix for the 2001 FPS game called Tribes 2. There have been a few notable titles released on the TGE, such as Blockland, Marble Blast Gold, Minions of Mirth, TubeTwist, Ultimate Duck Hunting, Wildlife Tycoon: Venture Africa, ThinkTanks, and Penny Arcade Adventures (see Figure 1.1) [20].

Figure 1.1: Minions of Mirth, also developed in Torque

In obtaining a license for TGE we were able to develop our simulation as a new environment within the engine, while gaining access to the engine source code to make very specific changes to how things function if need be. The engine is written in C++ and can be compiled on Windows, Macintosh, Linux, Wii, Xbox 360 and iPhone platforms. It runs their own proprietary scripting language, Torque Script, on the surface to handle normal game interactions between objects as well as the creation of environments or levels.

In chapter 2 we will discuss other work that has been done in the field. In chapter 3 we describe the process used in creating a test environment within the TGE. In chapter 4 we present the methods preformed while developing the particles as objects which simulate reality, in an efficient manner. In chapter 5 we discuss the creation of the emitters and how they operate in a manner which effectively and accurately releases the particles into the environment. In chapter 6 we detail the process in which the Sensors were developed. In chapter 7 we discuss the steps that were taken to integrate the emitter, sensor, and particles into the seaport environment. In chapter 8 we go over the experimental results gathered in testing the effectiveness of the systems. Finally, in chapter 9 we discuss the conclusions as well as possible future work that can be done to extend the project.

Chapter Two

Related Work

2.0 Overview

In this chapter we will discuss the related work that was looked at during the development of this system. We will present a summary of the relevant parts of the papers as well as why they were initially looked at and how their methods differ from that of our own.

2.1 Real-Time Sound Synthesis and Propagation for Games

This paper [18] by Raghuvanshi et al. presented a way to simulate real-time sound propagation, in complex virtual scenes. They used a combination of ray tracing and volumetric representation to simulate their propagation.

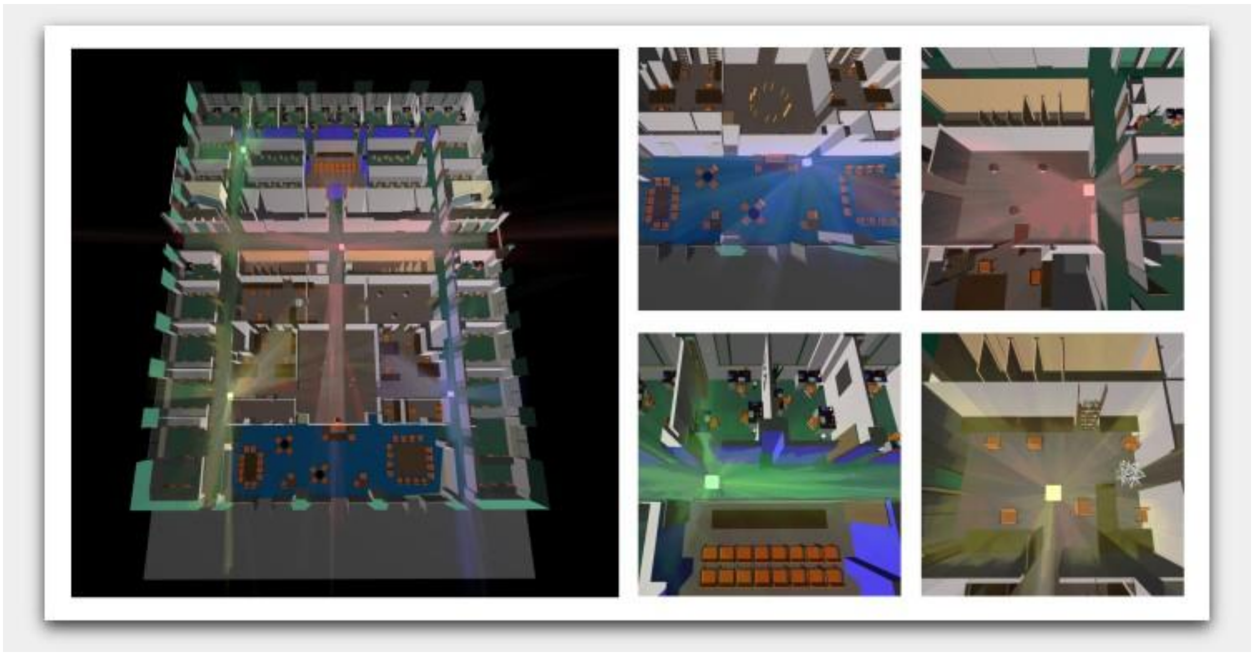


Figure 2.1: A screen shot of some of the environments used for their Sound propagation testing, with the ray frustums visible.

We felt that perhaps their methods of real-time sound propagation could help us in designing our real-time particle propagation. They simulate all the sound waves as a large geometric volume, or frustum in order to allow the scene to remain real-time, since the computational complexity of tracking all the sound waves would be too expensive. They used these four-sided frustums in order to simulate where the sound currently is in the environment (see Figure 2.1). When the frustum interacts with an object in the environment, they clip and divide the frustums to make up for simulating the different sound waves, rather than just maintaining the estimation that the previous larger frustums simulated. This allowed them to get the performance increase with the larger volumes prior to the frustums encountering anything in the environment and helped in their accuracy of modeling sound wave propagation upon colliding with something in the environment. Their approach to addressing the real-time propagation by using a larger object to simulate the whole was very intriguing and led us to investigate their methods more thoroughly. This idea of using a larger object to simulate what was happening in real life is something that we were able to adapt for our simulation, however since we were modeling particles as objects themselves and our particles did not reflect off of surfaces, an exact adaptation of their methodology would not work for our goals.

2.2 A Game Engine Based Simulation of the NIST Urban Search & Rescue Arenas

This paper [21] presented a project done by a group at the University Of Pittsburgh School of Information Sciences. Their goal was to develop a series of simulation environments (see Figure 2.2a for example) of the National Institute of Standards and Technology (NIST) Reference Test Facility for Autonomous Mobile Robots (Urban Search and Rescue) and develop testing scenarios with varying degrees of difficulty. The environments will then be used for testing teleoperation interfaces and the sensors capabilities for robots in the environment, which was to be used in upcoming robot designs. Their goal, like ours, was develop a way to perform inexpensive simulations to dynamically evaluate configurations that were currently in development.

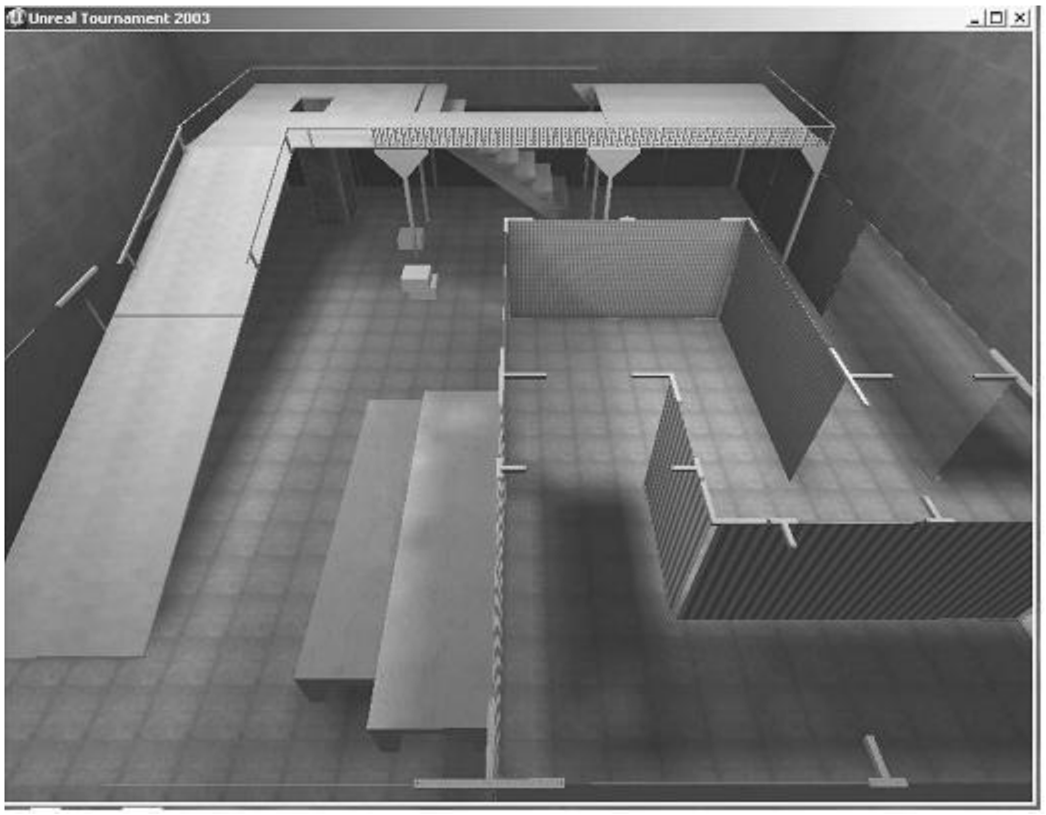


Figure 2.2: An environment developed for the NIST Urban Search & Rescue Arenas

The idea of the cost effective simulations using a three-dimensional environment was very similar to our initial goals and as such we review the tools that they used for development of their simulation environment. The main difference between our two different approaches was that they used the Unreal Engine. This Engine was developed by Epic games, and its rudimentary development kit is provided to those who purchased their Unreal Tournament 2003 game. We looked into using this engine, however not having complete access to the source code of the engine quickly caused us to look in other directions. Another engine that we looked at was the Quake III game engine. This was a game engine produced by id Software in 1999 and released with a game of the same name. It was widely licensed in other games, however in 2005 the source code was released under the GNU General Public License. When we were choosing which engine to use in our development, we considered the Quake III engine, however when someone downloads the source code for the engine they are provided with no documentation or support from the company who produced the engine. These shortcomings caused us to look at another game engine which we could license in order to obtain more documentation and support and ended up with the Torque Game Engine. This engine, like the Quake III engine, provided us with a three-dimensional environment in which to develop our simulation environments, however it was still being actively licensed in the game development industry and provided us with online documentation and support for any bugs or errors we might encounter.

2.3 PortSim - A Port Security Simulation and Visualization Tool

This paper [12] written by Daniel B. Koch presents PortSim, a simulation system used to analyze the impact that various changes to the geographical layout of the port (both in deploying new detection systems, and inspection procedures) will have on both the cost and operational models. While this paper addresses a similar topic as our project does, the way that it presents said topic is quite different. It focuses heavily on how changing port procedures and adding new equipment will affect the efficiency of the port, rather than a simulation attempting to address the effectiveness of the placement of said equipment. The main idea behind this system differed from ours in a major way. First their idea was not to model threats and check to see whether security configurations could detect those threats, they were addressing the cost of performing various operations at a seaport environment, and using a real-time data acquisition tool in order to help analyze the risks of various cargo containers or ships passing through the port. Their system focuses a lot less on the visualization of the port, and resorts to a two-dimensional representation of a port. In contrast, our simulation system will focus heavily on the visualization of a port and provide a user with a representation of what is actually happening in their port at any given time.

2.5 GIS-Based Performance Prediction and Evaluation of Civilian Harbour Protection Systems

This paper [3] written by A. Caiti et al. presents a simulation system to assess the level of underwater security in civilian harbors by looking closely at the geographical characteristics of a region and the effect those characteristics have on passive sensors. This simulation takes a geographical area and runs genetic algorithms on the said area to try and find an optimal sensor configuration. This system is similar to ours in that it is addressing what the optimal sensor configurations for an area at a port environment are, however our project is trying to simulate the environment and the threats directly (emitters, particles) rather than just attempting to run algorithms over a region for maximum coverage. In addition they focused a lot less on the visualization aspects of their simulation system and generated two-dimensional topographical maps that their simulation could process to try and determine the optimal positions of their sensors. In contrast we are providing the user with the ability to test out not only any configuration they want, by manually setting up the positions of each sensor, and threat themselves, but they can see just how effective the configurations that they are setting up will actually be when they run the simulation.

2.6 Summary

In this section we discussed some of the other projects which were looked at when we were working on developing this project. We looked at what each of the other projects goals where and what they were doing to address those goals. We then looked at how those goals differed from what we planned on doing to see if their methods could help to benefit our project. By looking at these other projects we were able to learn a number of things. First, we learned that we could use a larger particle to represent a group of particles, thus reducing the load we have on our system. Secondly, we learned what some of the game engines available to use could provide and what some of the pros or cons of each might be in order to help select one that more accurately addresses our needs. Finally, we learned that while there are a few other systems that also address seaport security, in terms of system configurations, ours seemed to be the only one that was focusing on making the system a three-dimensional real-time simulation of the entire port.

Chapter Three

Modeling a Test Environment

3.0 Overview

In order to perform testing of the sensor, particle, and emitter systems while the actual seaport environment was still under development a few suitable test environments needed to be created. In this chapter we will discuss the process of creating such environments, as well as how we modeled materials in said environments so that particles could interact properly.

3.1 Building an Environment

The first step in creating these environments was that we had to decide what was needed for the tests that we would be performing. The first series of tests need nothing but the sensors and emitters themselves, so having a blank template (see Figure 3.1a) would work fine. However, for other tests that needed to test particle interactions with other objects, we need more. Therefore in order to create those objects and buildings needed in the testing, a few different modeling tools were used. First, all of the objects used in testing were developed with Maya, and then exported to the torque propriety dts format with plug-ins provided to the Torque community. The second tool used was that of Constructor, a free tool that Garage Games provides to develop simple buildings for their TGE.

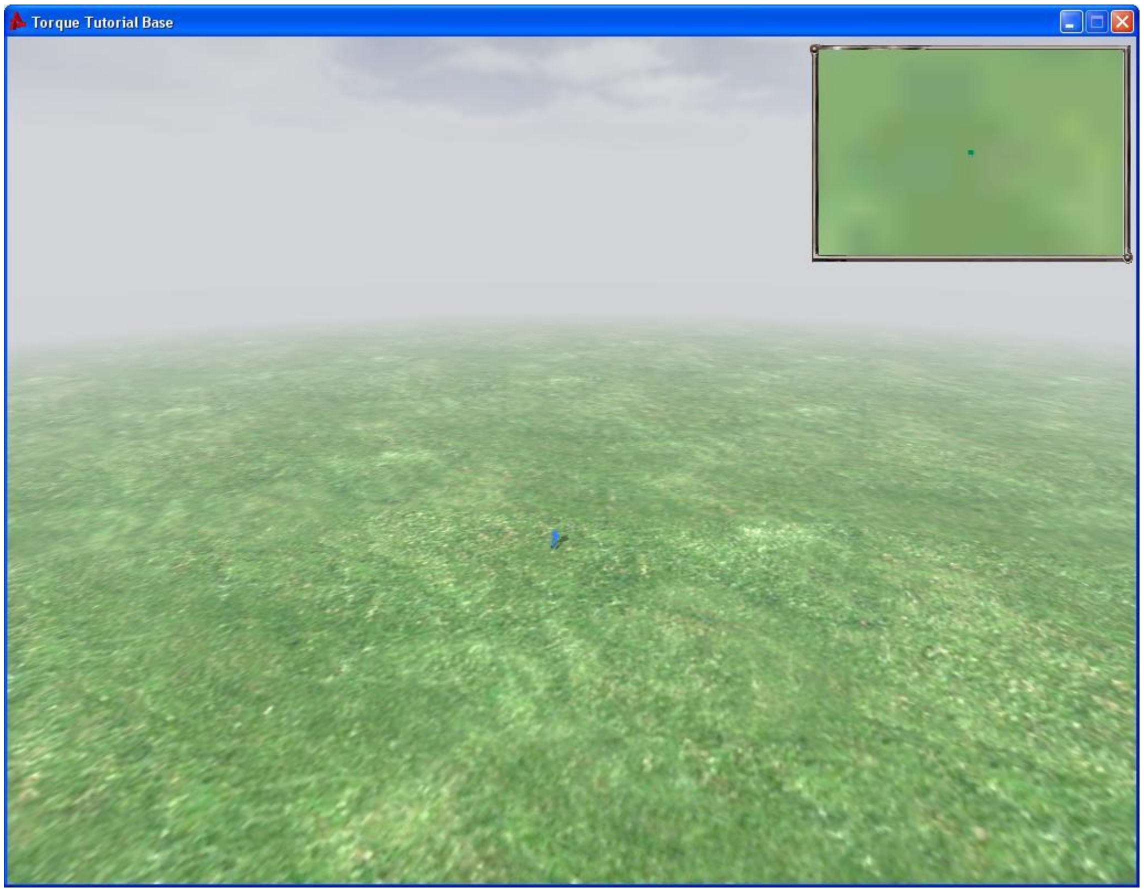


Figure 3.1: A screen shot of our empty test bed.

Constructor allows users to create buildings with a fair amount of ease, but the trade off is that unless you want to spend a lot of time modeling with this tool, the results will be fairly blocky with a lot of squares and rectangles. Using these tools, a number of other scenarios were created for testing of the emitter and sensors. The first of these was a blank template with a number of obstacles set up in various positions in order to test all of the reflection, refraction and collisions that were to be used by particles. The second environment set up was a little more complicated. Using the buildings, the objects created, and the TGE built-in terrain tools, a small street environment was established in which to perform more sophisticated tests. This consisted of a street, with buildings/houses on all sides, and a pile of shipping containers on one end of the street. The environment can be seen in Figure 3.1.

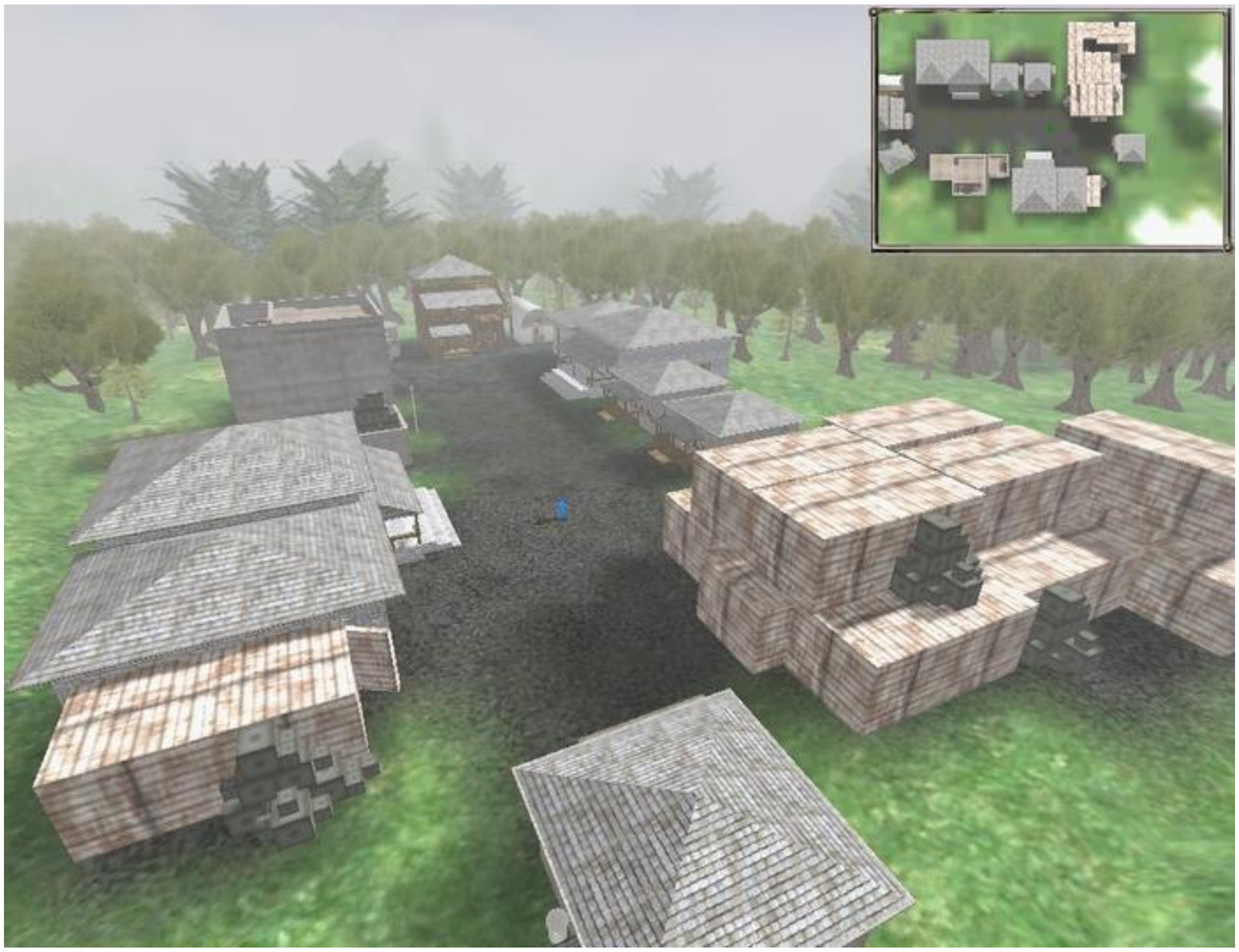


Figure 3.2: A screen shot our street test environment

In order to achieve more collisions handled at a faster pace, we decided to integrate the PhysX physics engine into the TGE. This would allow us to essentially multithread all of the collision detection and allow us to handle a lot more particles at one time. After finding a nice resource [19] on the Garage Games forums and getting the SDK which can be found at http://developer.nvidia.com/object/physx_downloads.html, we were able to incorporate the PhysX engine with not a lot of effort but several significant engine changes. However the big change was that now all the objects in our environment that we wanted to interact with had to be wrapped up as PhysX objects. This had a few major problems associated with it, the first being the arduous task of converting all of the objects, buildings, etc. in the missions that we had done in order for them to be compatible. The second thing that presented a major problem was that the buildings were now PhysX objects which led to some weird behavior. The main issue was that when the player used the character to traverse the environment rather than just the camera, everything that they encountered would fly off in some long arcing path as if it had just been kicked by some giant or fired out of a cannon, this was the case with all of the PhysX objects in the scene, which as you can imagine presented a major problem if the user decided to walk around as the character and attempted to enter a building. We found that this was due to the low mass and density that these PhysX objects had in relation to the user's character. The solution to this problem was to manually adjust the mass property for the objects in our scene. This allowed the character to interact with the environment without the risk of kicking a building out of the environment. Another issue with having the objects as just PhysX objects was that objects could not be placed within each other when you are setting up the environment or when the mission started off. The PhysX engine would attempt to rectify this as quickly as possible causing the objects to fly off in opposite directions. As discussed in chapter 5 a solution to this problem was to enable collision groups between objects in which we wanted a collision to occur.

3.2 Working with Materials

In order for the particles to act appropriately we needed them to interact in certain ways based on the materials (concrete, steel, wood, etc.) they were colliding with in our environments. As such we needed a way to model these materials. We knew that if we wanted the particles to act in a certain way when they encountered a material we need a way to identify the material in question. For instance we needed objects like steel or lead, to stop a different number of particles than say wood, or dirt. In order to accomplish this we decided to use key words in the names of objects, such as "steelCargoContainer1". With this tag the engine would then know that the material for the object was steel. We were able to then have the particles react to the said material. So in the collision handling for the particles we simply had to select what they were doing based on the tag in the name of the object. For example if the particle were to collide with an object with the name “steelCargoContainer1” and in the engine steel had been set up as a 20% chance to absorb a particle, the engine would look at the name and check for any material tags within that name. In this case it would see the “steel” tag and then choose a random number between zero and one hundred. If that number is below 20 then the particle’s velocity is set to zero. As the systems progressed in their development we started to model the particles in our system as gamma particles. We then had to take a closer look on how these particles interacted with our materials in the environment. We talk more thoroughly about this in chapter 4.

3.3 Summary

In this section we explained how in order to test the systems developed for the propagation and detection of particles a few simple test beds were introduced. These test beds varied from being plain empty test fields to that of a more complicated street scene to make sure the particle system could still operate at high speeds, with a high degree of accuracy in a more congested area. Also we talked about how the materials for different objects were handled and what was done in order to get these materials to interact with the particles properly.

Chapter Four

Modeling Particles

4.0 Overview

In this chapter we discuss the steps that were taken in order to accurately model particles as they interact with the environment. This chapter addresses the different types of particles that were tested, as well as what was done with those particles to obtain the desired characteristics.

4.1 Goals

In modeling these particles there were a series of goals that we had in order to ensure that the particles behaved appropriately. The first was particles needed to be able to provide us, the modelers, an option to perform custom collisions, thereby allowing us to determine the course of action taken by the particle as it collides with the different objects that it encounters while traversing the environment. These custom collision reactions were initially that the particle should be able to reflect off an object, refract through an object, or pass through the said object unobstructed. The second goal was that the particles should have the capability to be invisible to the user. When the simulation is running the user should not see these particles as they traverse the environment. As the development of the simulation continued the goals changed. As we continued to develop the system we narrowed the focus of what type of particles we were planning on modeling, and we settled on gamma radiation particles. With this narrowing of what we were modeling we were able to take advantage of specific properties of gamma radiation in order to simplify the simulation as described in Section 4.2.

4.2 Modeling Gamma Radiation

When we decided to look into modeling gamma radiation there were several things that we had to come to understand about how they interact with their environment. First is how gamma particles are created. Gamma particles, or rays as they are commonly referred to, are produced when a radioactive atomic nuclei disintegrates or when certain subatomic particles decay [7]. Gamma rays are found at the low end of the electromagnetic spectrum, with regards to its wavelength (generally shorter than a few tenths of an angstrom), and the high end in terms of its energy (generally greater than tens of thousands of electron volts [eV]). Next, we looked at some of the characteristics of the gamma particle. Gamma particles have no mass or electric charge, and they travel at the speed of light in free space[13]. The energy that gamma particles carry is also known as radiative flux, and is more accurately described as the amount of energy moving at some distance from a source per unit area per second in the form of photons [6]. As gamma particles travel they interact with matter in a number of different ways. However since a gamma particle has no mass or charge it has a high capability of penetration and is very difficult to stop [13]. The process of having flux losing its intensity is referred to as attenuation. The three main ways of attenuating gamma rays are through the Photoelectric Effect, Compton Scattering, and Pair production. The Photoelectric effect (Figure 4.1) occurs when a gamma ray intersects with an orbital electron of some atom that is from the material in which the gamma ray is passing through and it transfers all its energy to the electron. This causes the gamma ray to cease to exist.

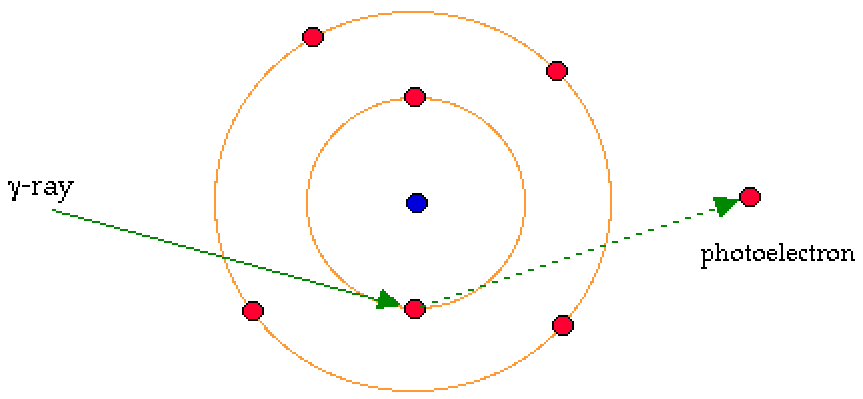


Figure 4.1: A diagram of the photoelectric effect [13]

Compton Scattering (Figure 4.2) occurs when a gamma ray collides with an orbital electron of an atom of the material through which it is passing, much like the photoelectric effect. However in Compton Scattering not all of the energy is transferred into the electron and the gamma particle continues on at some trajectory [13].

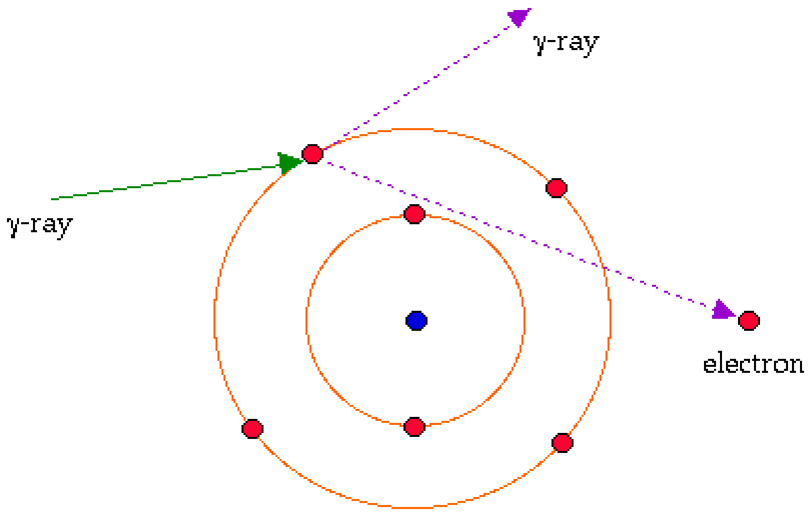


Figure 4.2: A diagram of the Compton Effect [13]

The third primary form of matter interaction is that of Pair Production (Figure 4.3). This method occurs when a high energy gamma ray passes close enough to a heavy nucleus, the gamma ray completely disappears, and a electron and positron are formed. In order for this reaction to occur the original gamma ray must have at least 1.2 MeV [5].

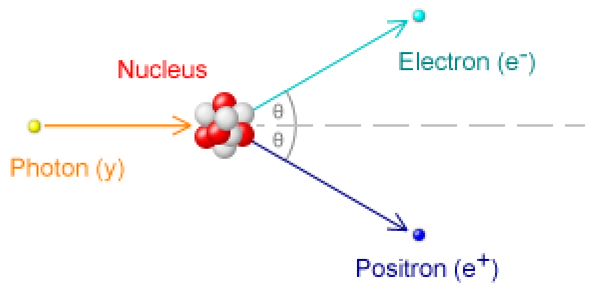


Figure 4.3: A diagram of Pair Production
(http://en.wikipedia.org/wiki/Pair_production)

There are a number of properties that affect the change of intensity when a gamma ray is attenuating with some material, including the atomic number, density, and thickness of the material the ray is passing through and the energy of the gamma ray. As the atomic number of the material increases it provides the gamma ray with a larger target with which to collide and thus has a greater chance for interactions. In contrast, with low atomic numbers the atoms are smaller and the chance of having a gamma ray collide with the atom is reduced. The density of an object works the same way, when the atoms of a material are closer together the chance that a gamma ray will collide with an atom are greater. The thickness of the material also acts in a similar fashion in that when there are more atoms to collide with, the chances of that collision happening increases. In contrast to the previous three factors when the energy of a gamma ray increases the chance of attenuation is decreased. Using these three factors we can come up with $I_x = I_0 * e^{-\mu x}$, a basic expression to describe the resulting radiation intensity (I_x) as a function of the initial intensity (I_0), the Linear Attenuation Coefficient (μ), and the thickness of the material (x). This equation is shown as a graph of final Intensity over thickness in Figure 4.4 [13].

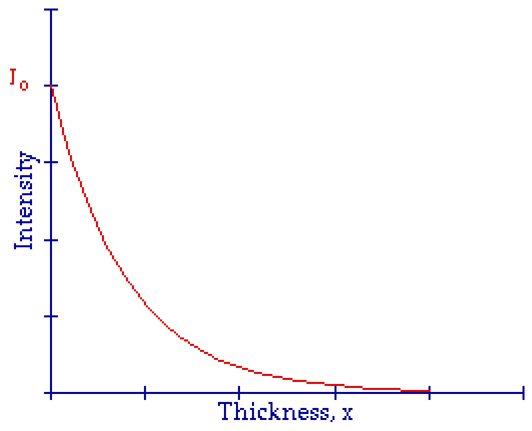


Figure 4.4: A graph of $I_x = I_0*e^{-\mu x}$ which is a mathematical model for the attenuation of gamma rays.

The Linear Attenuation Coefficient can be described as a characteristic of some material with regards to its gamma ray absorption capabilities based on the energy levels of the gamma ray. For some objects like air or water this coefficient is low, where as items like lead have a high coefficient [13].

According to George Chabot [1] gamma radiation has minimal reflection. In fact, the gamma particle never reflects but causes an electron to shoot off in another direction. Usually the gamma particle is absorbed by the material or passes through it at a reduced energy. Therefore we no longer had to consider any reflections or refractions when tracking our particles in our system. However we would need to have different absorption rates for each material (each object would only be able to be represented as a single material though) in our system. A shortcoming of using gamma radiation for modeling our particle is that cesium-137 (a common substance found in dirty bombs according to Jonathan Medalia [15]) at 2 Curies and a yield of about 0.01% will yield roughly 950 photons per second towards a point 25 game units (one game unit is roughly equal to one meter) away from it [4]. In contrast our simulation at its peak was able to produce only 500 particles per second before a significant performance drop, as shown in Chapter 8.

4.3 Built-in Particle System

The first method that was explored was that of the built-in particle system within the TGE. This system is primarily used for gases or types of lighting effects (fire, etc.), but we felt that we could adapt this system to meet our goals. One such example of what the particle emitter would be used for would be for smoke coming off of a fire (see Figure 4.1). The developer can set the density of the smoke by adjusting the size and opacity of the particles being fired, as well as alter if and how wind and gravity might be affecting the smoke.

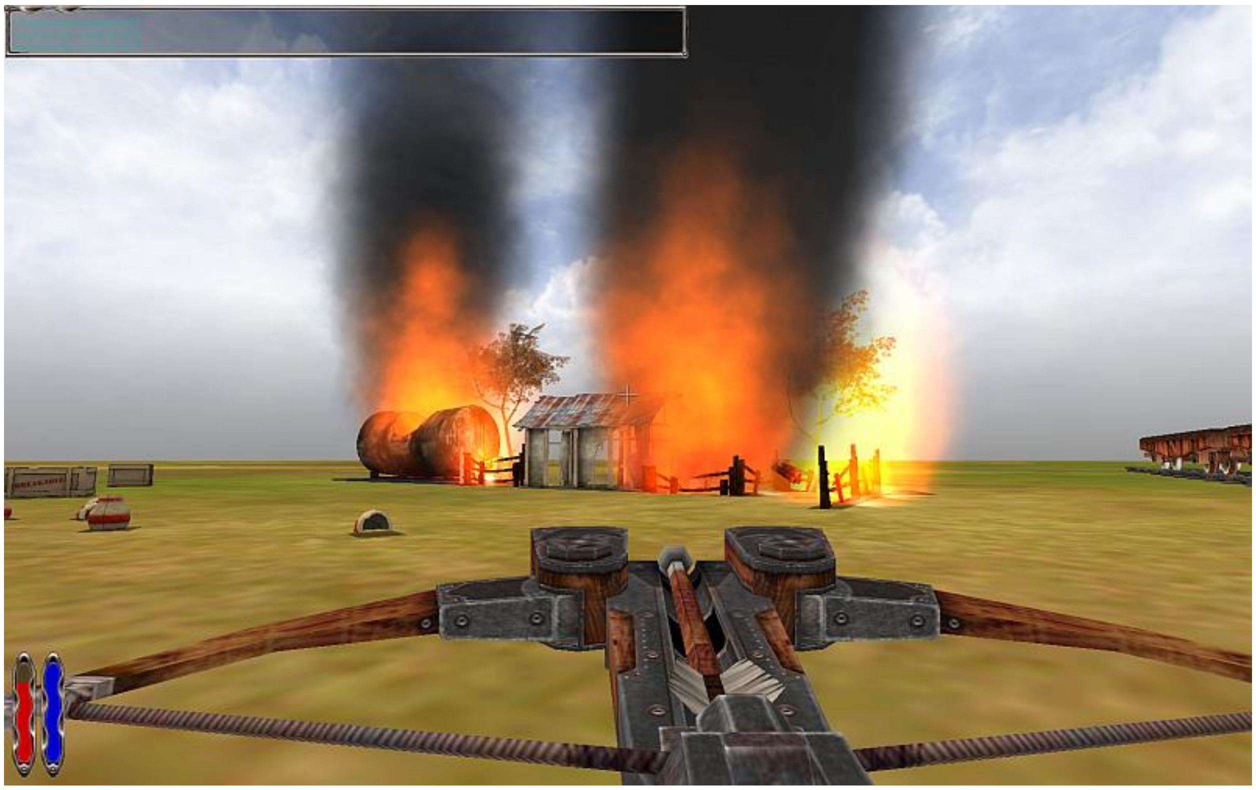


Figure 4.5: An example of the built-in particle system being used for fire, and the smoke from the fire in one of Torque's demos provided with the game.

After looking over the various parameters [14] used in creating the particles, it was quickly discovered that this particle system would not meet our goals, out of the box. The primary offender was that once the particles were released from an emitter, the user had no control over what they did. We also discovered that others have had similar problems with the default particles, albeit for different reasons. As a solution to some of these problems a TGE developer created a resource [9] on the Garage Games forums describing some of the engine changes that he had made in order to allow the user to take direct control over what the particles are doing as they interact with the environment. Incorporating this resource into our working engine, we were able to use the built-in particles to now accomplish what we were unable to do previously, that is that we were able to dictate exactly what the particles were to do when they encountered certain types of objects. However, as we continued to narrow down what we wanted from the particles and how they were emitted, we discovered that the built-in particles would not work for what we wanted to accomplish.

4.4 Projectile Particles

The second type of particle that we used was that of a projectile. This was essentially an in-game object traveling at a given speed and acceleration in a given direction. This type of object is most often used for weapons in a FPS developed in TGE, for example the bullet from a gun (see Figure 4.3a). These objects have a variety of properties that can be set such as how and if gravity affects them, how long they will be around, and what will happen when they are created or hit an object (muzzle flash, explode, etc.) [14].

Figure 4.6: An example of Torque projectiles being fired in “Lore: Aftermath” a game made with Torque.

The biggest problem that was encountered with these projectile particles at the start was that they would quickly destroy themselves upon encountering another object in the environment. This was solved by added a flag to the projectile object in the engine code that can be flagged from the script which tells the engine to ignore those self destructive calls. The second problem is that there is no way to set projectiles as invisible. This also was fairly easy to overcome in that a few more changes to the engine code and another flag allows the engine to just ignore the rendering of certain objects based on that flag being set or not. This flag can be set by a simple function available in the script. Another problem that we were encountering was that if we set the speed on the particles too high, the engine would miss the collisions that the particle should be having. This was due to the game cycle only acting upon a certain time interval. If the velocity placed an object, which was previously in front of another object, to now be behind that object, then the collision would be ignored. This prohibited us from being able to have our particles travel at the incredibly fast speeds of real radiation particles. This method was working well, but we wanted to see if there was something we could do or change to how the engine handled our collisions in order to increase the number of particles that we were able to have in the scene at a given time.

4.5 PhysX Actors

In response to the desire to try out other ways to keep track of the particles, we decided to try incorporating the PhysX physics engine into the TGE so that the new PhysX engine could handle all of the collisions with the particles. The PhysX engine developed by Ageia is designed as a real-time physics engine which works in conjunction with a Physics Processing Unit [PPU] in order for the game to offload physics calculations from the CPU, allowing it to perform other tasks instead [16]. If no PPU is present the CPU treats all the calculations needed to be done by the engine as a separate thread. The wrapped up PhysX actors behaved in a similar fashion as that of the projectiles without the timer for their lifetime, which could be handled by a built-in suicide function handled by a scheduled timer. The biggest problem we encountered with the PhysX engine objects was that the PhysX engine handles all of the collisions automatically based on the properties (size, mass, density, etc.) of the objects colliding with each other. Now this is fine if you want physics based collisions but since we wanted to be able to pass through objects, and refract through objects, the standard collisions would not be enough to handle all the things that we wanted to do with our particles such as reflecting off of objects, refracting or passing directly through objects, and being stopped (absorbed) by objects.

Therefore we searched for a way to fix this problem, and came up with what is known as collision groups [16]. These groups allow a programmer to use a callback to be called for the collisions between objects of different groups based on some values set up prior to the collision. This allowed us to handle the collisions for the particles and all of the objects they encountered in these call backs, with the limitation that there were only 32 groups that we could set up. So we defined one group for terrain and other objects we do not want to interact with, one group for the particles, one group for the sensors, and 29 groups to use as different types of materials that our particles could interact with. It looked very promising and the collisions were being handled nicely, exactly like wanted. However we quickly realized that as we increased the number of particles we had in the environment at one time, the performance dropped significantly. The issue was that when the PhysX engine had to just make callbacks for when a low number of particles encounter the sensors, the number of callbacks was still low enough that we did not have to suffer that much from the performance hit. But when we had to make a callback for everything in the environment that the particles hit, and the number of particles increased, we were essentially handling the majority of the collisions in the environment with the call backs. As a result of having to deal with the callbacks in addition to the other collision costs, the performance hit was too great (see experimental results in chapter 8 for more specifics).

4.6 Increasing Performance through Abstraction

In order to increase the performance of the system there are a number of things which can be done. One approach is to abstract away some of the particles. By this we mean that you could use a single particle as a group of particles. This would allow the simulation to represent a larger number of particles traveling through the system while retaining the performance capabilities of the previous system. For example each particle object could have a value known as energy, and every time the particle encounters an obstacle that would have stopped it, prior to the abstraction, it would have its energy reduced. Then the particle would only stop when the energy on that particle was reduced to zero. Even if you have the particles represent as few as two or three particles, the number of particles represented in the system would be very close to the goal of 950 photons per second (as presented in section 4.1) with little performance hit on the overall system.

4.7 Summary

In this chapter we discussed how the evolution of our goals helped to guide us in working on these different particle systems in order to achieve a better simulation. We addressed the task of using the built-in particle emitter to model particles and the steps that were used to get these particles working properly. We then addressed the processes of modeling the particles as both projectiles and PhysX objects. Based off our development of each of the respective particle systems, it was clear that the projectile system was a better option. First of all, the built in particle emitter system was unable to perform all the goals for the particles that we developed during the development of the system and was subsequently discarded. The projectile particle system provides a much cleaner interface for future developers to work with. All of the code that needs to be changed for the behavior of particles (with regards to material interactions) is all in one neat function that gets called by the engine when those two particles collide. This provides an easily readable and maintainable system. In contrast the PhysX system uses callbacks and other libraries not natively present in the TGE, and what is happening when particles collide is not readily apparent to the developers, causing them to have to manually trace the calls to the callbacks in order to properly understand what is happening. Also when the user of the system lacks a PPU or multi-core processor the addition of the PhysX libraries will only hamper performance of the system. In the end with these considerations and the results garnered from the tests in Chapter 8, it is clear that the projectile particle system is the better system.

Chapter Five

Modeling Emitters

5.0 Overview

In this chapter we discuss the steps that were taken in order to create a particle emitter which was able to fire particles in an efficient manner. This covers the three different methods of emitters that were used, the first being that of the TGE built-in emitter, the second being a projectile emitter, and lastly a PhysX object emitter. For these emitters a simple shape was generated with Maya and can be seen in Figure 5.0a.

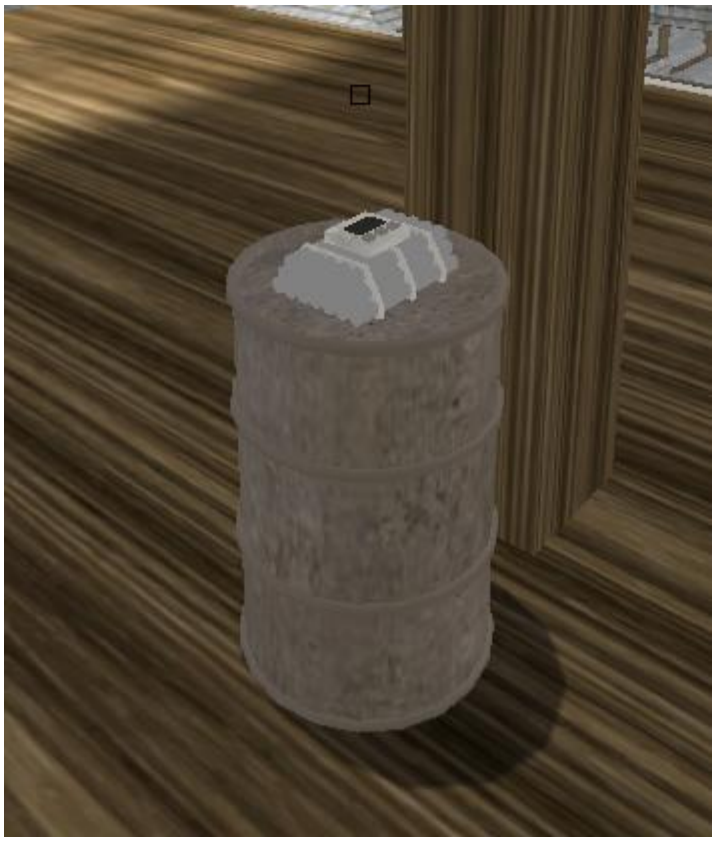


Figure 5.1: The model used for the emitters.

5.1 Goals

Over the course of the project the goals for what was required of the emitters has changed significantly as we refined the type of particles we were actually going to track. At the start the goals for the emitter were simply for the emitter to emit particles as fast as possible in every direction, preferably in a random manner while remaining somewhat uniform in its distribution. As time progressed we decided we wanted the particles to be released as a pulse or wave, and have groups of particles released at one time in every direction. This change was a result of our desire to mimic what occurs when particles are release from a point source. They are not emitted in a single direction one particle at a time, but as a wave of particles in all directions. And finally as we narrowed our view of what type of particles we wanted to represent (gamma radiation), as discussed in chapter 4, we realized it was pointless to fire particles in a direction away from the sensors. So our goals changed again to just aim the particles at a point (a part on the sensor) and fire them as fast as possible towards that point.

5.2 Built-in Particle Emitters

TGE provides an object known as a particle emitter which can be set up to fire particles in a given arc (determined by two degrees since it is a three dimensional object), at a specified interval, and with a specified speed. The primary use for the particle emitters is for smoke and gas. After looking at how the particle emitters worked [14] it looked quite promising. The emitter allowed us to fire particles in random directions at a specified interval and speed which is just what we wanted. In this task the particle emitter acted beautifully, but as the goals of the emitters changed to that for the particles to be released as waves of particles we determined that the built in emitters were no longer a viable option to rely on.

5.3 Projectile Emitter

The next emitter setup that we looked at was that of a projectile emitter. The idea behind this emitter was to use the built-in projectile objects and fire a large number of them from one point in different directions, thus creating a wave or pulse of particles as seen in Figure 5.3a.

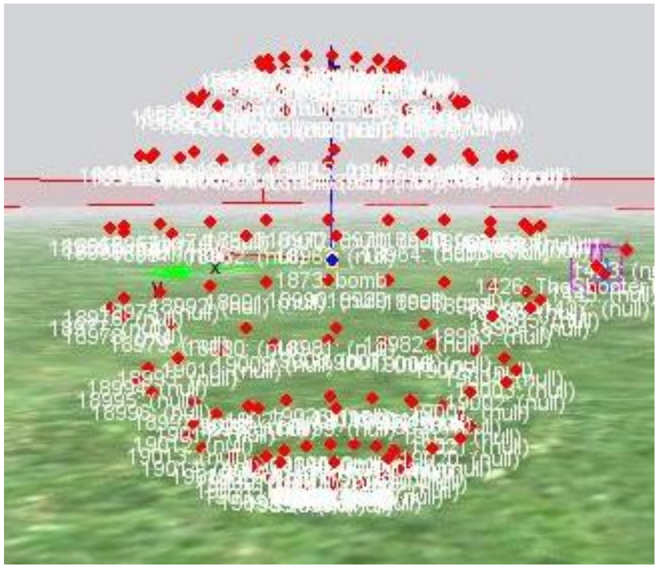


Figure 5.2: A screen shot of the particles pulsing away from an emitter.

The main problem with this approach was that in order to have the coverage, in all directions, that we wanted to get, we had to have hundreds of these particles all packed in tight together. With this increase in particles the performance took a big hit. We then tried to use a smaller density of particles and shift the direction a little bit every time the emitter fired to cover the unaccounted for areas. The problem with this oscillation was that it essentially just gave us more coverage but at a cost of increasing the interval in which particles reached that area. This dramatically slowed the effective rate at which we were firing particles. It seemed that no matter what method we used we were not going to be able to support the speed and number of particles that we wanted to achieve. We knew we would have to be using our slower larger particles, compared to that of real-life particles, as a rough estimation of what is happening, but we were hoping for better results nonetheless. Eventually we decided to change the idea to just target the particles at a specific point. We decided on this approach due to the fact that gamma radiation does not reflect or refract off/through objects. This allowed us to ignore particles not being fired at the target, and focus on firing the particles as fast as the TGE could handle it, thus achieving much better performance.

5.4 PhysX Emitter

In an attempt get better performance with our particle emissions, we also tried to use the PhysX engine. In order to get an emitter for our PhysX engine, all of our objects had to be PhysX Actors, an object in the engine code, so we could not use the projectile emitter method anymore. So we had the emitter spawn new objects at its position and then apply a force in the desired direction to that object. This worked out well with a few exceptions. The first problem we encountered was that the objects spawning on the same location would instantly be forced out of the emitter and fired off in a random direction. In order to overcome that problem we had to turn off the collisions of the object for a short period of time right at its inception. That way for the time that it was on top of the emitter and the other newly spawned particles, it would not collide with anything. This was causing problems in that the period of time in which the collisions were turned off was hard to judge, because if the emitter was right next to a wall there was a good chance the particles would be through that wall by the time their collisions were turned back on. This problem was fixed however when we started using collision groups (as discussed in chapter 4) since the particles would no longer collide with each other or the emitter so it was no longer necessary to turn off their collisions. This method also suffered from similar problems as the projectile method in that they were leaving big gaps in the areas between where the particles were being fired, and as we attempted to alleviate those problems we suffered the same drop in performance. However, much like the projectile emitter, when we switched to firing all of the particles at a target point, we were able to fire the particles at a much higher speed, and as such the emitters were not degrading performance.

5.5 Summary

In this chapter we address the creation of different types of particle emitters and how their development progressed. We first talked about how we adapted the built-in particle emitters to shoot particles quickly in every direction, however this system was soon discarded due to it not meeting all of our goals for how emitters should work. Then we proceeded to talk about how we developed both the projectile and PhysX systems. These two systems were both developed in a manner to achieve all of our goals, and from an emitter perspective they are both able to act at a high level of efficiency. Both can be simply placed into an environment and will emit particles in a given direction at a given speed. The major difference between the two emission systems did not begin to reveal itself until we preformed the testing discussed in Chapter 8 at which point the projectile system was proven to be significantly more accurate. Over the course of this chapter we also address how changes to the goals of how the particles were fired affected the approaches we were using and their impact on performance.

Chapter Six

Modeling Sensors

6.0 Overview

In this chapter we will address the problems we encountered while modeling the sensors and the solutions that were used to solve them. While the requirements for both the emitters and particles changed significantly throughout the different iterations of methodology, the sensors have remained somewhat consistent in their lifetime. However the main things that they have had to address are: ensuring that all of the particles that hit the sensor are recorded, letting the user know which sensors are being hit, the frequency at which they are being hit, and the total number of particles to hit the target. While working with these sensors a mock shape was created with Maya and can be seen in Figure 6.0a.

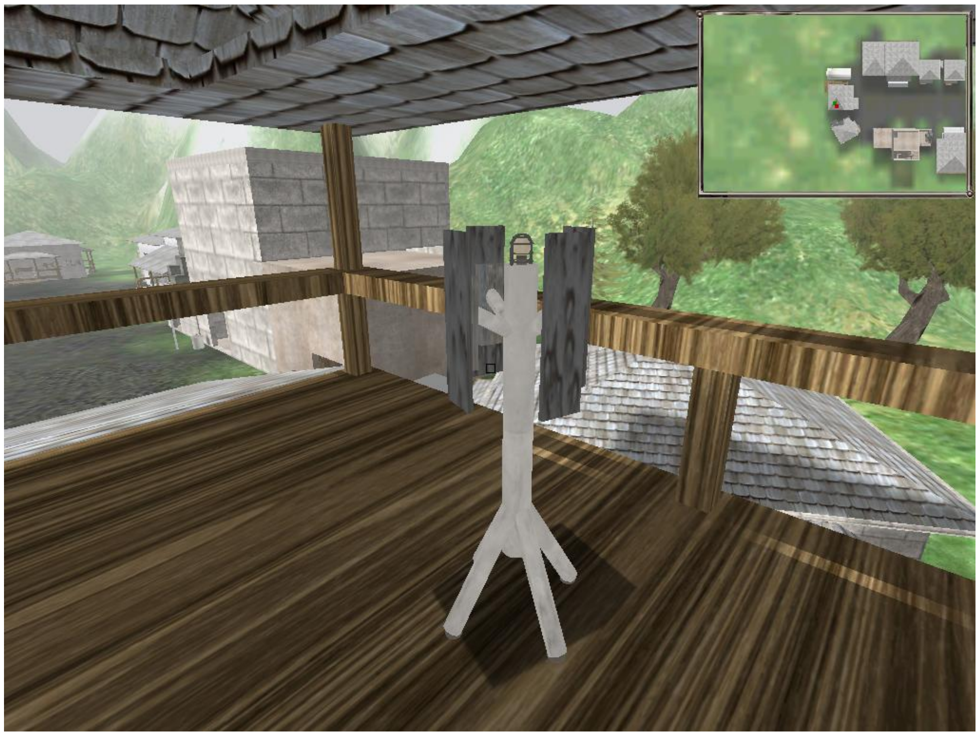


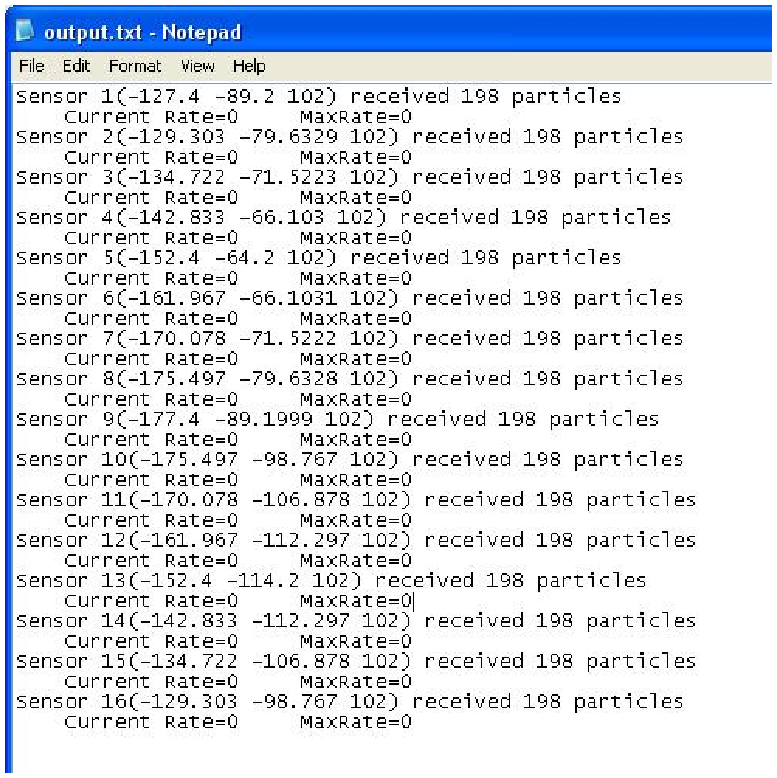
Figure 6.1: A model used to represent the sensor for our simulation.

6.1 Ensuring Collisions are recorded

For the most part a passive sensor's job is easy. It has to sit at some location and wait for some object, in this case particles, to run in to it. Then it needs to react accordingly; in our case it needs to record that the particle hit it. Throughout the lifetime of the project this job has not changed and for the most part it has done its job efficiently and effectively. The only real problem that we encountered was that with the PhysX engine there is a cap on the number of simultaneous collisions that it can handle. This lead to some terribly low performance results when it came to testing the accuracy of the PhysX system (as shown in chapter 8).

6.2 Reporting Status

From the start we wanted the sensors to report to the user, not whenever they get hit by a stray particle, but when they are receiving particles at a set rate. When this threshold is broken the user should be informed in a manner that effectively allows the user to tell which sensors are at the threshold and their locations. To address these problems we first gave all of the sensors a light on the top that would light up when that given sensor broke the threshold. This was okay if the user was near the sensor, but if they did not have a clear view of the sensor, there would be no way for the user to know if that sensor was ever hit by any particles. The second approach was to make an intelligent mini-map (see Figures 3.1, 3.2, and 6.1) that would show a little dot on the mini-map where the sensor was placed, and this dot would change to a different color when the threshold was broken. This allowed the user to see which sensors were being triggered as well as their position both in the environment as well as in relation to each other. However this method was found to be only effective when the user was actively looking at the mini-map, and if the mini-map did not cover all the sensors there would be parts of the map that could not be seen by the user. Therefore we decided to log all of the data that these sensors collected and periodically output it to a file as shown in figure 6.2. This way we have a detailed listing of each



```
Sensor 1(-127.4 -89.2 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 2(-129.303 -79.6329 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 3(-134.722 -71.5223 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 4(-142.833 -66.103 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 5(-152.4 -64.2 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 6(-161.967 -66.1031 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 7(-170.078 -71.5222 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 8(-175.497 -79.6328 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 9(-177.4 -89.1999 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 10(-175.497 -98.767 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 11(-170.078 -106.878 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 12(-161.967 -112.297 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 13(-152.4 -114.2 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 14(-142.833 -112.297 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 15(-134.722 -106.878 102) received 198 particles
    Current Rate=0     MaxRate=0
Sensor 16(-129.303 -98.767 102) received 198 particles
    Current Rate=0     MaxRate=0
```

Figure 6.2: An example of a system log

sensor, its location, the total number particles that it received, the maximum rate at which it was receiving particles, and how many times the sensor broke the threshold. So with the combination of the lights, mini-map, and logging we felt that these provided the user with enough data to see what is happening in a scene for a run of the simulation.

6.3 Summary

In this chapter we talk about the steps performed in order to get the sensors working properly in the simulation environments. We also learned of the major short coming (only being able to keep track of 4 simultaneous collisions) with the PhysX system which caused the accuracy tests performed in Chapter 8 to point to the projectile system as superior. We also talked about how we had to address the problem of the user not knowing what was happing in the system, since the particles were invisible and the user could not tell if any particles were actually hitting the sensor. This problem was addressed by including a light on each sensor, a detailed mini-map, and a system log.

Chapter Seven

Incorporating Particles and Sensors into the Port

7.0 Overview

In this chapter we discuss the steps that were taken to incorporate the sensor/emitter system into the port simulation that is being developed separately. It goes over some of the challenges that we encountered and what went smoothly.

7.1 Incorporation

Now that the sensor and emitter system was pretty much done being developed the task to incorporate it into the rest of the project was upon us. The first thing that was done was to clean up all of the code and scripting so that the sensors and emitters could be placed into the environment as easily as possible with very little effort. In order to do this we modularized all of the objects so that all that was necessary to get them working was that you needed to call placeSensor(pos) or placeEmitter(pos) where pos is the position you want to place them, these functions set everything up themselves, and once the emitter is aware that there is at least one sensor in the area it will automatically start to send out the particles. You could even take the objects passed back from these functions and then mount them within other objects, e.g., mount emitters in cargo containers or on a ship (see Figure 7.1a).

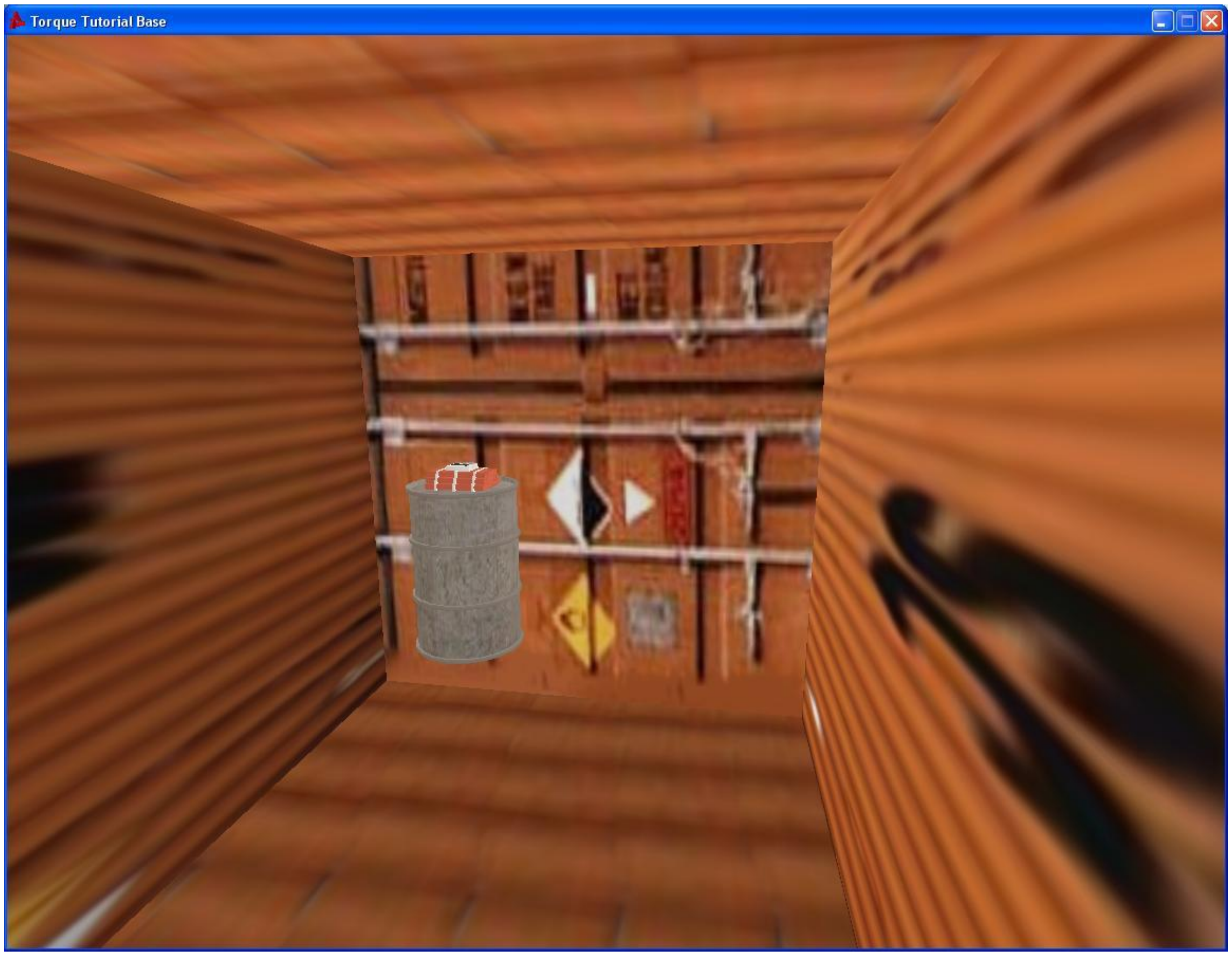


Figure 7.1: An emitter mounted inside of a cargo container.

With these functions you can effectively add both sensors and emitters to a scene and have the particle emissions and interactions work automatically. This allows the other developers using the system to be able to set up the sensors in the positions that they want without the need for any outside help in the development. The main thing that was noticed when the first set of emitter and sensor were placed into the environment (see Figure 7.2) was that the performance of the scene degraded significantly.

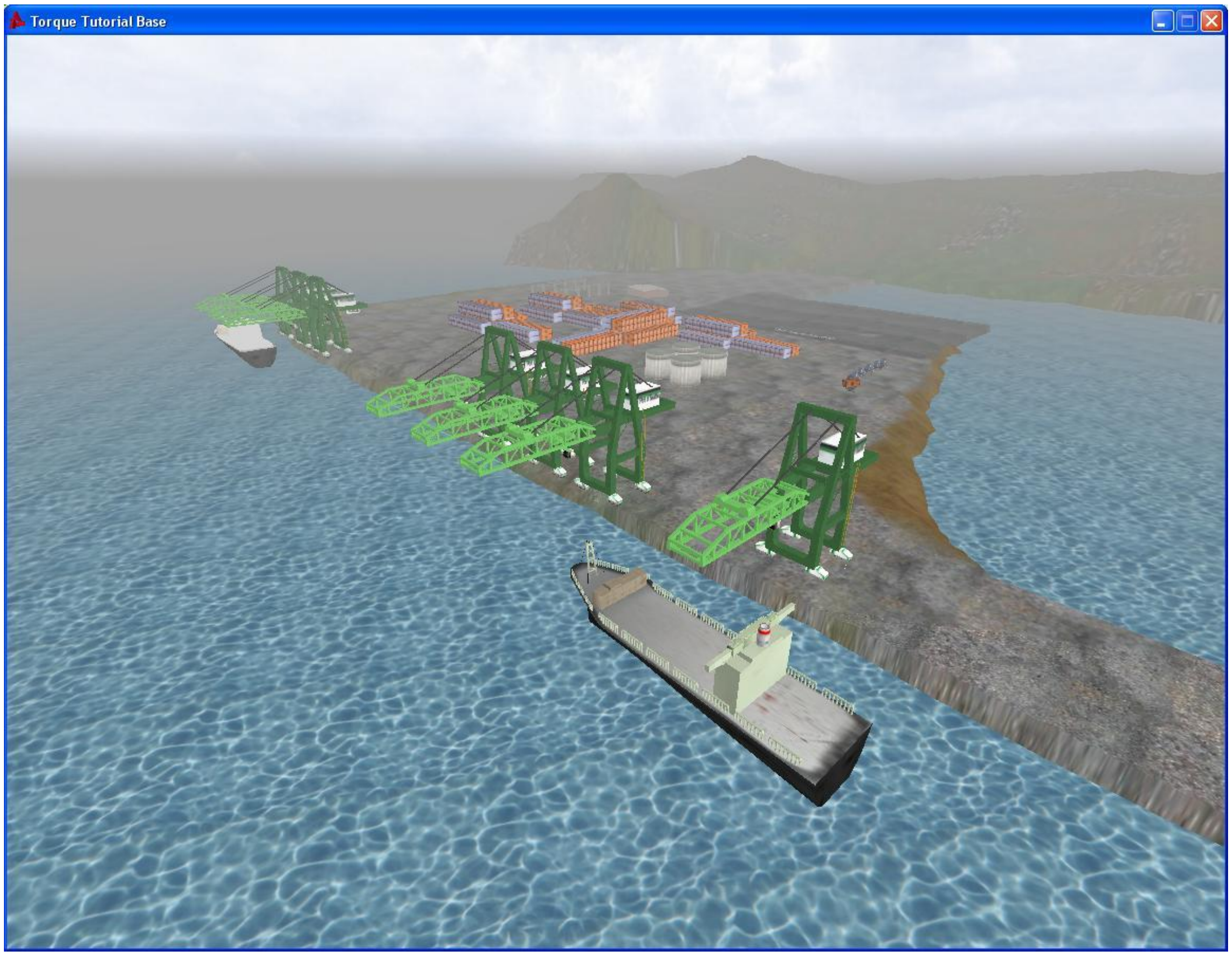


Figure 7.2: A screen shot of our simulation seaport

This was due to the fact that both systems were being developed as to be as lifelike as possible without taking a performance hit, which put them both near the edge of a poor performance area. However after comparing the performance (as a measure of frames per second) with the particles running and them not running in the scene, the difference was almost negligible. So in response to this we had to work on reducing the vertex count of many of the objects in the scene. With these simple reductions we were able to alleviate the problems that we were having with the poor performance.

7.2 Summary

In this chapter we presented the steps that were taken in order to incorporate the particle system into the seaport environment which was developed in conjunction with this project. We talked about some of the changes that were made to make the particle propagation and detection system more modular for the benefit of other developers using the system. And we also addressed some of the problems that were affecting the performance of the simulation when we initially added the particle/sensor/emitters to the environment.

Chapter Eight

Experimentation Results

8.0 Overview

In this chapter we go over the experiments done on the particle emission systems, namely the Projectile system and the PhysX object system, in order to compare them with regards to accuracy vs. emission rates. The goal of these experiments was to determine which of the two systems would be the best representation of reality for the use in the simulation. For these experiments particles were sent at varying time intervals directly from the emitter to the sensor, and each minute the simulation would report the number of particles that hit the sensor and the total number of particles fired. After 5 minutes these sums were then added together, averaged, and recorded. These time intervals were varied for each run of a test (starting at every .2 seconds and incrementally reducing that number to .025 seconds), and after all of the runs are completed the results are analyzed. Initially these steps were repeated a total of four times and their combined results were averaged, however after performing the tests outlined in sections 8.1-8.4 we concluded that since there was no variation in the accuracy of the simulation, one run would then suffice.

8.1 No Obstruction testing Accuracy vs. Rate

In these tests the goal was to simply make sure that if there were no obstructions between the emitter and the sensor, all of the particles would reach their destination. So using an empty test bed for this experiment a sensor and emitter were placed 20 game units apart. This test was preformed for both the projectile system and the PhysX object system. The data that was recovered from the first test as presented in Figure 8.1 was very illuminating. First of all we discovered the PhysX system becomes incredibly inaccurate as the time interval between launching particles decreases, whereas the accuracy for the projectile system remained high. This caused us to believe that there was a limit on the number of collisions that the PhysX system would be able to handle at a time. And after doing some additional testing, by sending

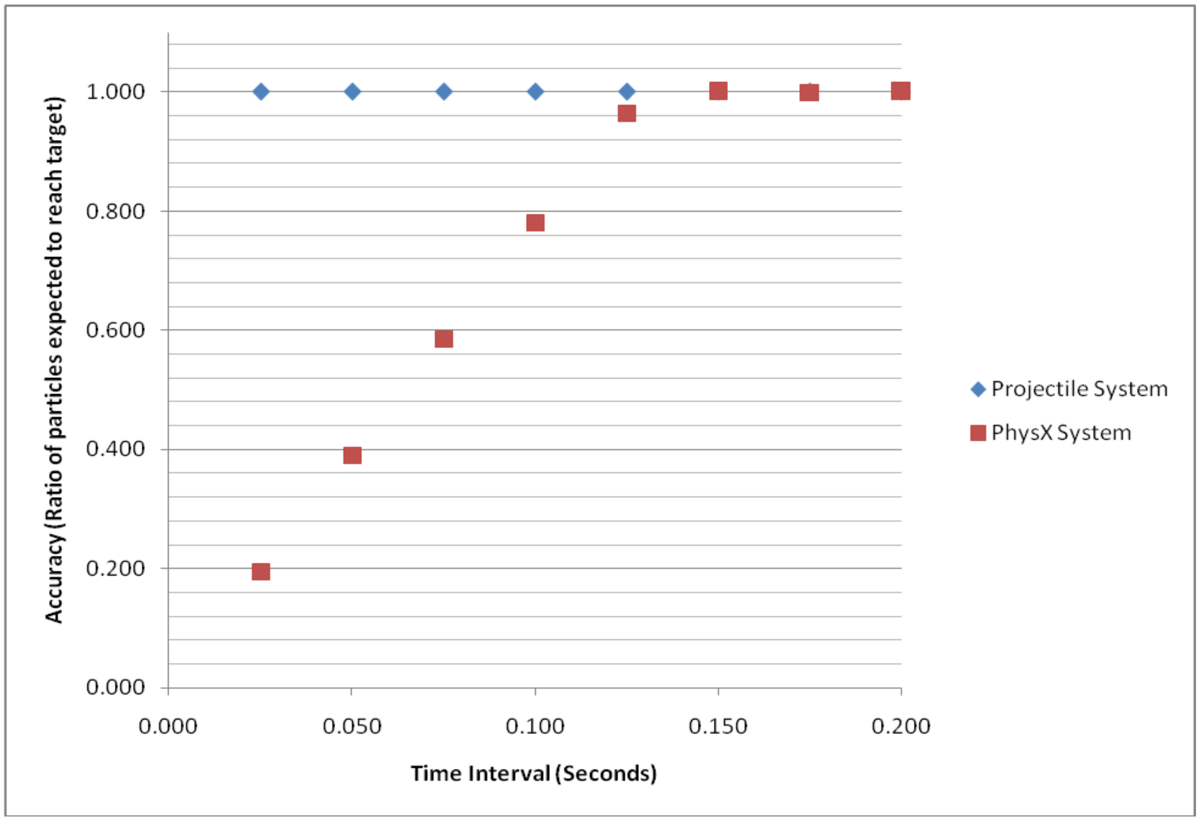


Figure 8.1: Results for the no obstruction test. A 1.0 in accuracy means that all of the particles fired at the sensor were recorded.

increasingly larger groups of particles at the sensor, we discovered that the PhysX system can only handle four simultaneous collisions. As the time interval between the firing of the particles was reduced, this limitation would make itself evident. So in performing this experiment we learned two important things. First of all that the projectile system was working correctly in that all of the particles that were being fired were safely making it to the sensor given that there was nothing to stop their progression. Secondly we learned that the PhysX system had a major handicap that would probably end up preventing us from using it all together.

8.2 50% Obstruction testing Accuracy vs. Rate

In this test an empty test bed was used, placing a sensor and emitter 20 game units apart from each other with an obstruction that would prevent statistically 50% of the particles that hit it from passing through it. Then, as before, a test was run recording the average number of particles hitting the sensor and the total number of particles which were fired. Results from both the projectile system and the PhysX system were recorded. Instead of basing the ratio off of all the particles that were fired, we based it on half the particles, because with the obstruction we expect the total particles that should be able to reach the sensor would be half the total, as can be seen in Figure 8.2.

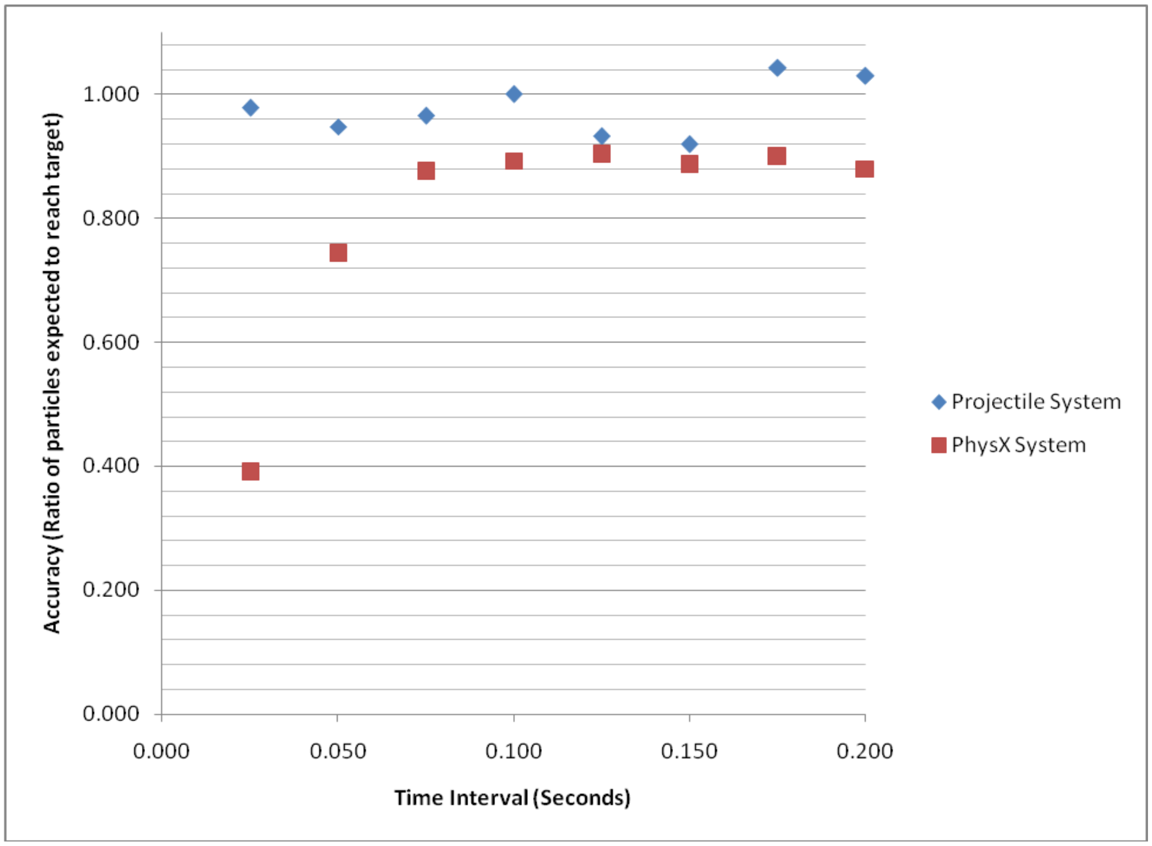


Figure 8.2: Results for the 50% obstruction. An accuracy of 1.0 means that half the particles hit the sensor, as expected.

8.3 Two Sources, One Sensor testing Accuracy vs. Rate

In this test an empty test bed was used, placing one sensor and two emitters with the two emitters placed 40 units apart with the sensor directly between them (20 units from each emitter). The test was run recording the average number of particles hitting the sensor and the total number of particles which were fired. Results from both the projectile system and the PhysX system were recorded and can be seen in Figure 8.3. The poor performance in the accuracy of the PhysX system can be attributed to the limitation it has with the number of simultaneous collisions. When the PhysX system only has to handle the collisions of a single emitter, it can keep up with the load. However even with the long time intervals, as we increase the number of emitters, the load is beyond the system's capabilities. As the time interval decreases, the number of simultaneous collisions increases, and the PhysX system's accuracy suffers even more.

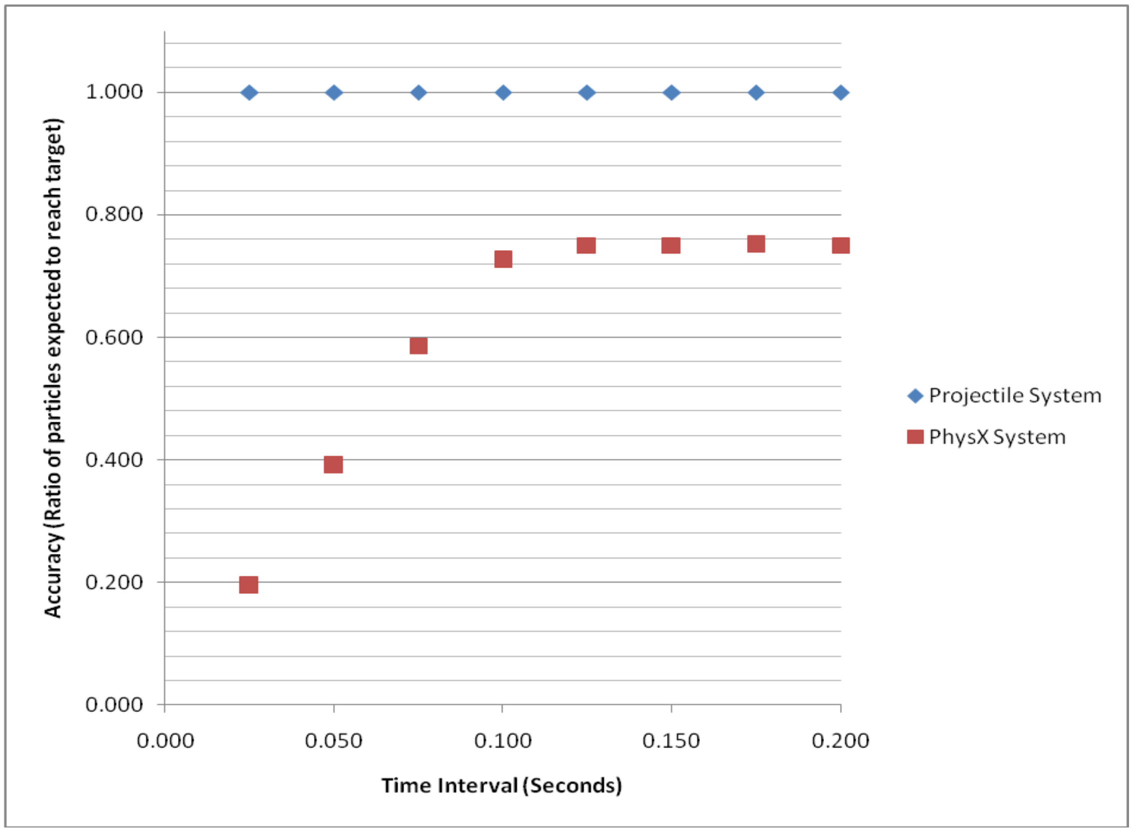


Figure 8.3: Results for the two Sources, one sensor test. A 1.0 in Accuracy means that all of the particles fired at the sensor were recorded.

8.4 One Source, Two Sensors testing Accuracy vs. Rate

In this test an empty test bed was used, placing two sensors and one emitter with the two sensors placed 40 units apart with the emitter directly between them (20 units from each sensor). The test was run recording the average number of particles hitting the sensor and the total number of particles which were fired. Results from both the projectile system and the PhysX system were recorded and can be seen in Figure 8.4. The time interval is the time between firing a particle at each sensor so the emission rates are for each sensor/emitter pair. The poor performance in the accuracy of the PhysX system can be attributed to the limitation it has with the number of simultaneous collisions. As the time interval decreases the number of simultaneous collisions increases and the PhysX system's accuracy suffers.

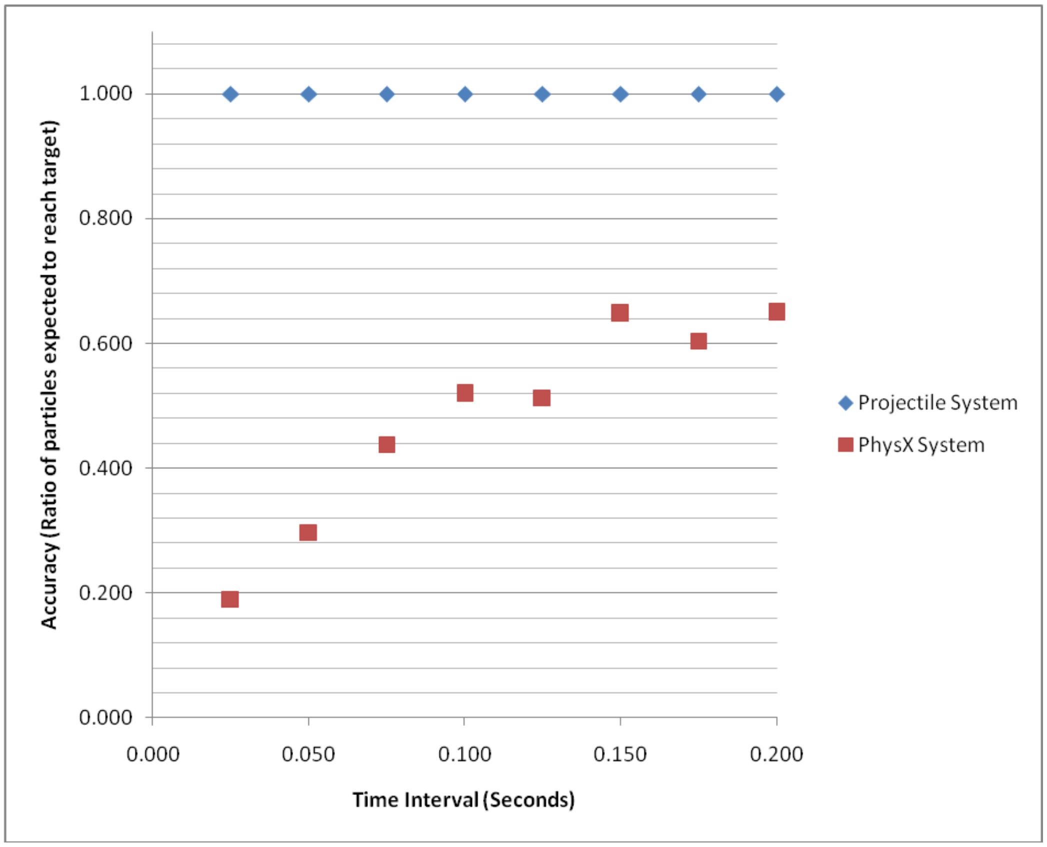


Figure 8.4: Results for the one Source, two Sensors test. A 1.0 in Accuracy means that all of the particles fired at the sensor were recorded

8.5 N Sensors, 1 Emitter testing performance load

In this test we placed a single emitter in a part of the port environment and iteratively added sensors in a circle, with a radius of 25 units around the emitter (see Figure 8.5).

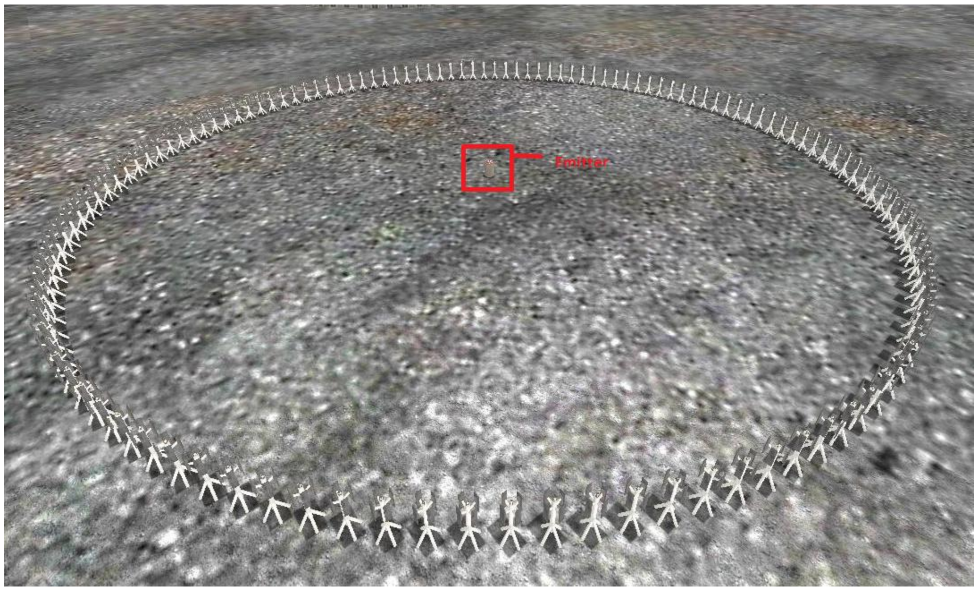


Figure 8.5: A screen shot for the N Sensors, 1 Emitter test

The test was run with the particles being emitted from the emitter towards each sensor at a rate of 10 per second. Then for each run the average frames per second was recorded. After 3 minutes the emitters would stop emitting particles. These runs were done with a varying number of sensors ranging from 0 at 25 frames per second to 256 sensors at less than 1 frame per second. The results from the test can be seen in Figure 8.6. In each test the effects of the load on the system could be seen while the emitters were actively firing particles, and once they stopped, the frames per second returned to 25. We can see from Figure 8.6 that as we add sensors to the environment the performance degrades gradually at first but quickly degrades to the point where a real-time simulation becomes impossible. This leads us to believe that we can use 30-40 sensors in a system before seeing any serious impact on performance.

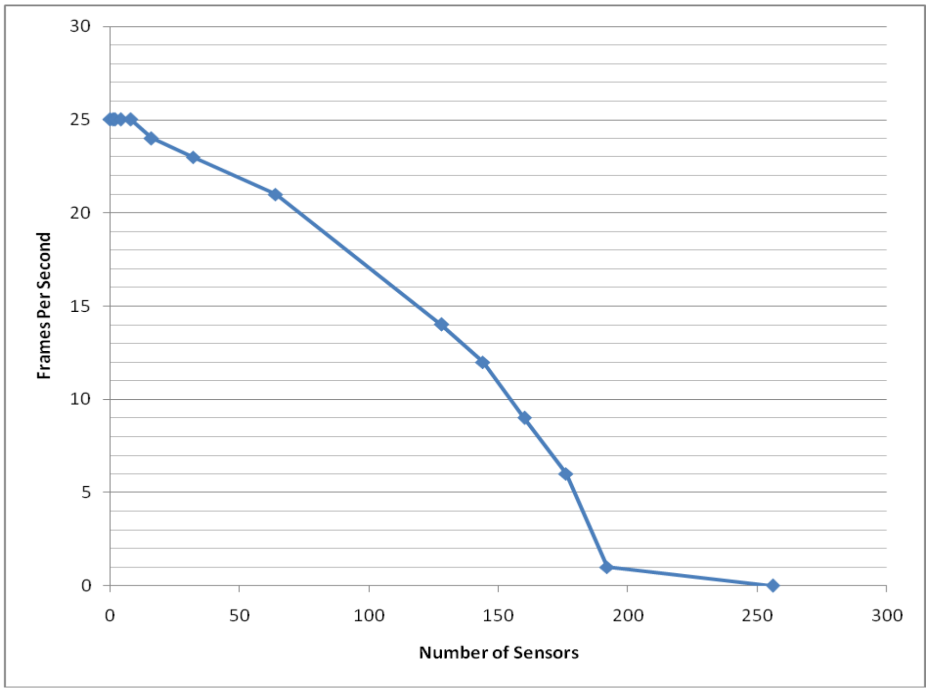


Figure 8.6: Results from testing a single emitter with a varying number of sensors and the effect that has on performance.

8.6 1 Sensor, N Emitters testing performance load

In this test we placed a single sensor in a part of the port environment and iteratively added emitters in a circle, with a radius of 25 units around the emitter (see Figure 8.7).

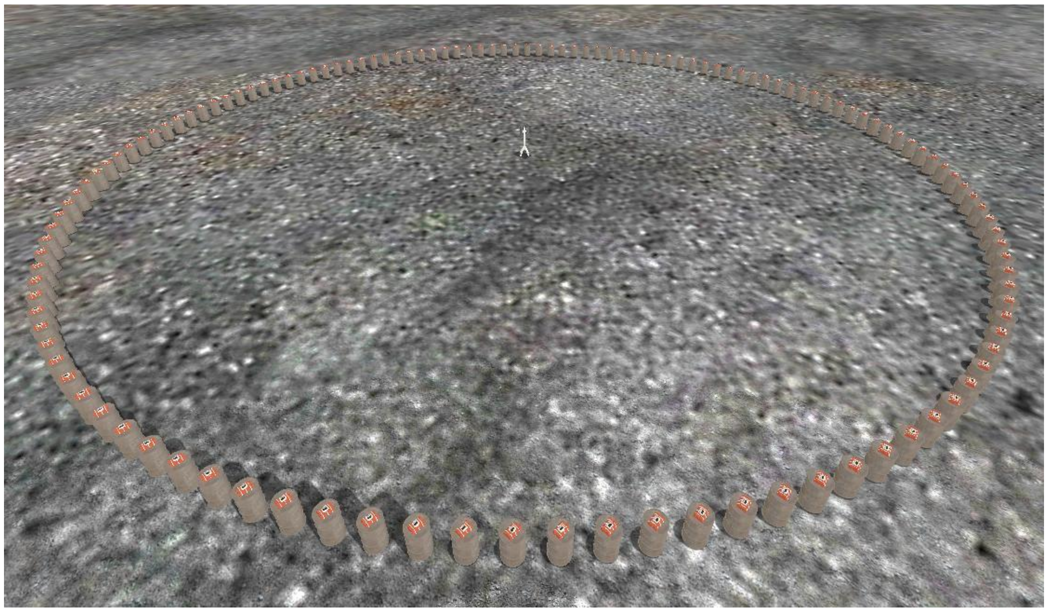


Figure 8.7: A screen shot for the 1 Sensor, N Emitters test.

The test was run with the particles being emitted from the emitters towards the sensor at a rate of 10 per second. Then for each run the average frames per second was recorded. After 3 minutes the emitters would stop emitting particles. These runs were done with a varying number of emitters ranging from 0 at 25 frames per second to 256 sensors at less than 1 frame per second. The results from the test can be seen in Figure 8.8. In each test the effects of the load on the system could be seen while the emitters were actively firing particles, and once they stopped, the frames per second returned to 25. We can see from Figure 8.8 that as we add emitters to the environment the performance degrades gradually at first but quickly degrades to the point where a real-time simulation becomes impossible. This leads us to believe that we can use 30-40 emitters in a system before seeing any serious impact on performance. The fact that these results are so similar to those found in the test performed in Section 8.5 is to be expected. Regardless of whether there is a large number of sensors or a large number of emitters, the number of particles being fired, the number of objects, and the number of collisions in the scene should all remain the same.

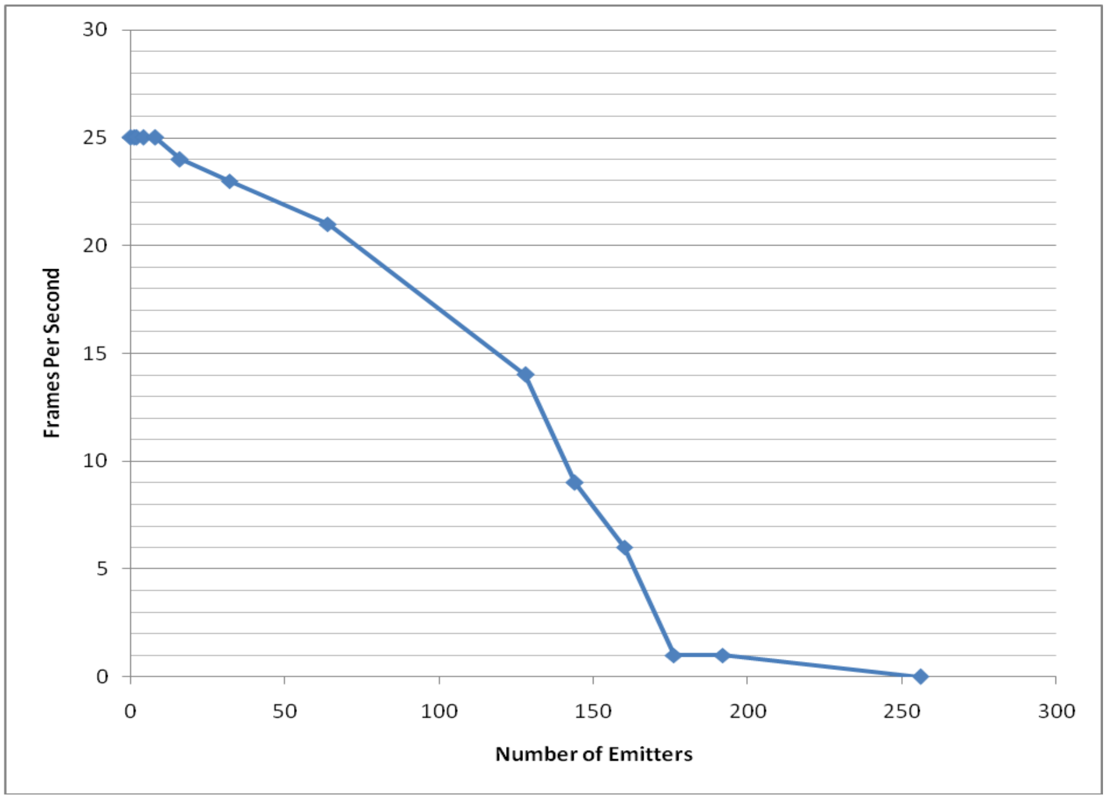


Figure 8.8: Results from testing a sensor with a varying number of emitters and the effect that has on performance.

8.7 Particle load testing

In this test we examined the effects of iteratively increasing the rate at which particles are fired from an emitter on the performance of the seaport environment. A sensor and emitter were placed in the port environment 25 units from each other. The first run had the emitter releasing particles at a rate of 10 per second and for each subsequent test that rate was increased. The results of the test can be seen in Figure 8.9. The interesting thing we learned from this test is that while we do take a performance hit while emitting particles at a rate of 1000 particles per second, based on calculations (as discussed in Chapter 4) we are approaching a good estimation of what is naturally occurring.

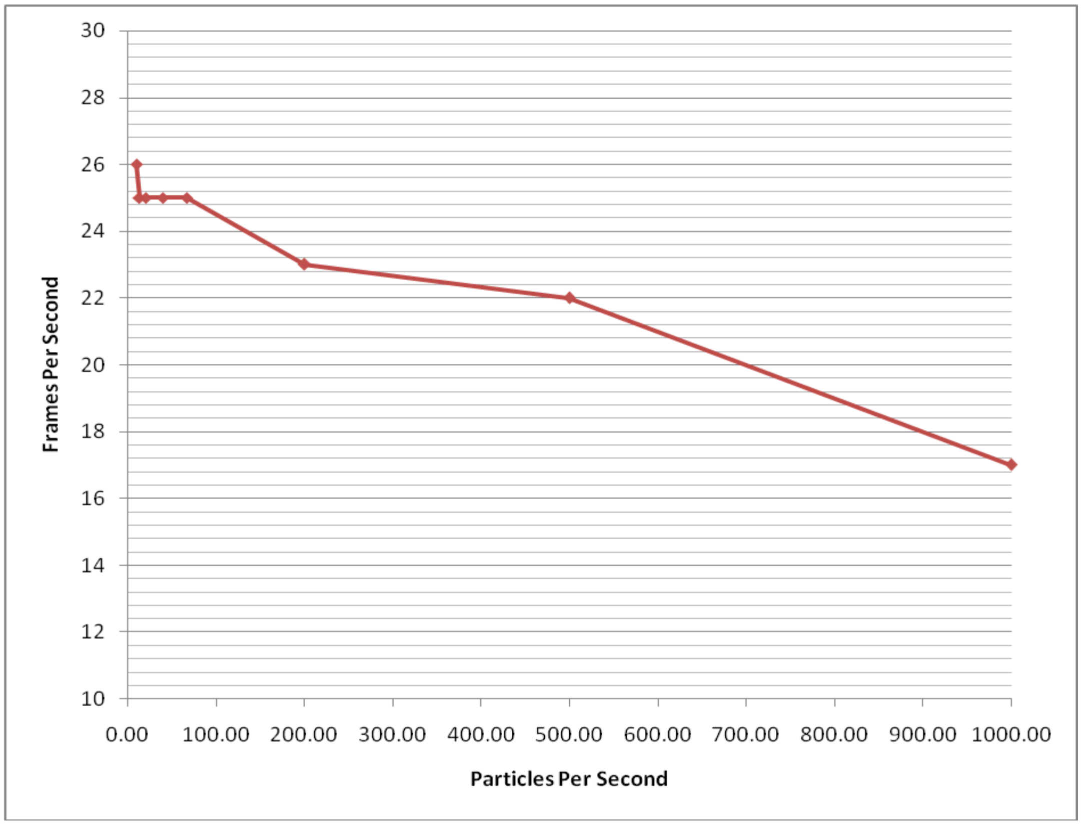


Figure 8.9: Results from the particle load testing

8.8 Obstacle load testing

In this test we fired particles through an increasing number of obstacles in order to check when the collisions generated from the obstacles would start affecting the performance of the system. An emitter was placed in the port environment, starting with no obstacles and increasing to 1000 obstacles. The frames per second of the system was then recorded. We learned that the load for the collisions of the particles was negligible; we did not notice a drop of frame rate at all for the test. As such, we can have many active particles in a system colliding with many objects while suffering little to no impact on performance.

8.9 Testing a port run

In this test an emitter was mounted in a piece of cargo and the cargo was brought in on a ship, offloaded onto a truck, and the truck drove out of the port. Two sensors were placed in the port environment, along the truck route. Obstacles were placed to obscure the path between

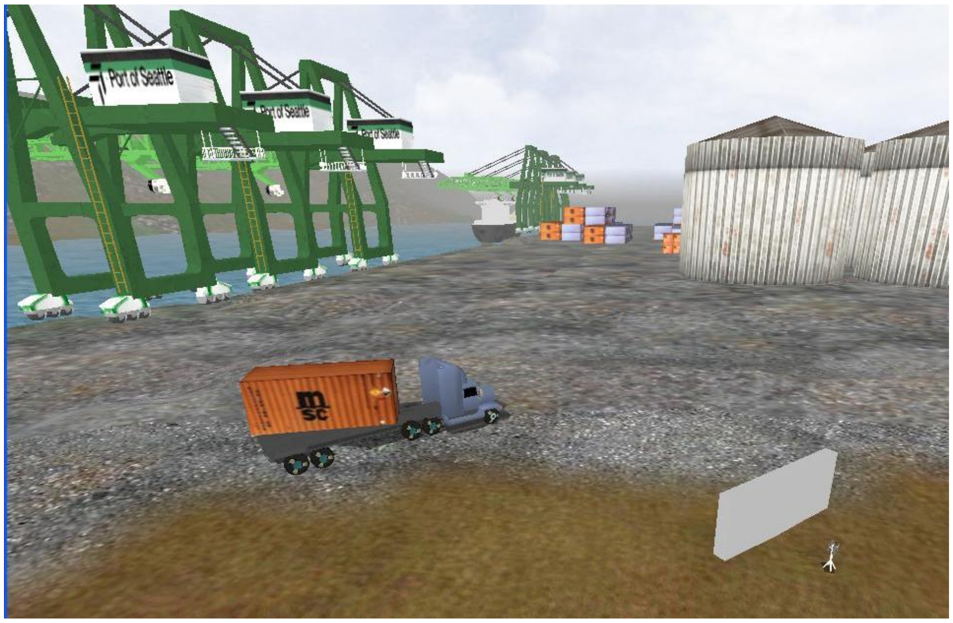


Figure 8.10: A screen shot of the port run test, showing a truck, sensor, and obstacle.

emitter and sensors along parts of the truck's route. This scenario was run four times, twice with the emitter actively firing particles at each of the sensors, and twice with the emitters doing nothing. The average frames per second was recorded for each run. The difference between the runs with the emitters actively firing and those without was at most 1 frame per second. This helped to solidify our belief that the particles' interactions with the environment are a very low load on the system when it comes to performance. This was a great boon in that since the particle system is running at such a low performance impact, it can be placed in more complex custom environments and work to the same effect.

8.10 Summary

In this chapter we discussed several of the tests performed on two of the particle emission systems. We started by comparing their emission rates vs. the accuracy of recording all the particles hitting the sensors. We quickly realized the limitations of the PhysX system and how only allowing four simultaneous collisions prohibited the PhysX system from competing with the projectile system. As such we conclude that the projectile system should be the one used in our simulation. However we continued to perform load testing on the projectile system to see if it would indeed be viable as a particle emission system. We learned that the number of sensors in an environment has to get above 30 before having any serious impact on a system. We also learned that regardless of having N:1 sensors or N:1 emitters, the effect on the environment remains similar. This is important because it shows us that the load is due to the number of active particles in the system, which will be the same whether we have a large number of emitters firing at a single sensor, or a single emitter firing at a large number of sensors. We were also able to see that our system is able to reach the 1000 particles per second milestone, albeit with a significant performance hit, and reach the 500 particles per second emission rate with a small performance hit. This is important in verifying the effectiveness of our solution at modeling what is naturally occurring. We also learned that the collisions that the particles are having with objects within the scene do not seem to have a notable effect on the performance of the system. With the light loads at which the particle emission system is able to run, as well as the speed at which it is able to emit particles, it is clear that the projectile system is a viable particle emission system.

Chapter Nine

Conclusions and Future Work

9.0 Overview

In this chapter we will discuss the conclusions made about doing this project and what we learned from the experiments that were preformed. It will address both the positives and negatives that can be drawn from these conclusions. Also we will mention future work that can be done to continue working on this project.

9.1 Conclusions

While working on this problem something was glaringly obvious even from the very beginning, that no matter how fast we have the particles traveling and no matter how many particles we can squeeze into the scene, we will not be able to come close to the number and speed of particles that would actually be traveling in a real life scenario. Surprisingly in our port environment while performing our particle load testing, we were able to get quite close to the 950 particles per second milestone with a significant performance hit and just over half that (500 particles per second) with a slight performance hit. This is of course not taking into account the surface area of our sensor which, in reality would be receiving a larger number, as we are making our estimation using a single point.

The next major conclusion that we were able to make was the performance difference between the projectile particle system and the PhysX object system. This conclusion was that the projectile particle system was able to outperform the PhysX object system in all of the tests that we were able to perform. The restriction on the number of simultaneous collisions that we encountered with the PhysX system became too much of an obstacle to overcome given the number of collisions in a typical scenario. This result was actually beneficial in that we had to do less work in the long run, since if the PhysX system had performed better, all of the engine changes that had to be done for that system (including wrapping all of the objects in a scene as PhysX actors, and losing some of the basic functionality of the TGE objects as shown by the particle emitter and projectiles) would then have to be done to the rest of the projects that were being developed for this same simulation. This would cause those developers to possibly have to change core sections of their code to work with the new system.

Overall it can be seen that a simulation tool, specifically using the particle and emission systems as described in this paper, can provide users with enough fidelity to accurately evaluate security configurations of passive sensors.

9.2 Future Work

Our work on this project is in no way the final answer to this problem and there are several ways that the project could be altered or improved with future work and experimentation. First, we could update the built-in particle emitter with the idea of just shooting a stream of particles at a single point, as we did with the projectile-based emitter. Since we stopped working with this system fairly early in the development of the project, the idea of reusing it to see if it would perform with comparable or improved results was never put into action. Another approach is to focus on the probability of particles reaching their destination rather than trying to track all of the particles individually. This would allow the user to set up a field of probability around a sensor so that if an emitter enters that area the sensor would then alert the user to its detection, based on some probability. Another aspect that should be looked at is to more closely address the physics of gamma rays as discussed in chapter 4. This information was collected for this project, but due to time constraints not all of those characteristics of material absorption rates and photon energy could be adapted to work with the current systems. Lastly the system developed in the course of this project could be used to simulate different environments including that of a border scenario.

Bibliography

1. Chabot, George. “Relationship Between Radionuclide Gamma Emission and Exposure Rate.” Health and Physics Society. <http://www.hps.org/publicinformation/ate/faqs/gammaandexposure.html>.
2. Christiansen, Allen, Damian Johnson, and Lawrence Holder. “Game-Based Simulation of the Evaluation of Threat Detection in a Seaport Environment” *2008 International Conference of Entertainment Computing.*
3. Caiti, A., Morellato, V., Munafo, A., 2007. “GIS-Based Performance Prediction and Evaluation of Civilian Harbour Protection Systems”. *Oceans 2007 – Europe* (June 2007), 1-6.
4. Decay Radiation Search < http://www.nndc.bnl.gov/nudat2/indx_dec.jsp>.
5. “DOE Fundamentals Handbook, Nuclear Physics and Reactor Theory, Volume 1”. Department of Energy, United States, 1995.
6. "Flux." Wikipedia, The Free Encyclopedia. 12 Apr 2009, 20:32 UTC. 21 Apr 2009 <http://en.wikipedia.org/w/index.php?title=Flux&oldid=283421457>.
7. "gamma ray." Encyclopædia Britannica. 2009. Encyclopædia Britannica Online. 19 Apr. 2009 <http://www.britannica.com/EBchecked/topic/225048/gamma-ray>.
8. Garage Games Developers Chat irc.maxgaming.net, Port 6667, #GarageGames
9. Garage Games Developers Network <http://www.garagegames.com/community/forums/54>.
10. Garney, Ben. “Commander Map”. Torque Game Engine Resource. <http://www.garagegames.com/community/resources/view/5277>.

11. Johnson, Damian. “Port Locale Modeling and Scenario Evaluation in 3D Virtual Environments,” Master’s Thesis, School of Electrical Engineering and Computer Science, Washington State University, May 2009.
12. Koch, D.B., "PortSim-A Port Security Simulation and Visualization Tool," *Security Technology, 2007 41st Annual IEEE International Carnahan Conference on* , vol., no., pp.109-116, 8-11 Oct. 2007.
13. Maher, Kieran. “Basic Physics of Nuclear Medicine”. WikiBooks. <http://en.wikibooks.org/wiki/Basic_Physics_of_Nuclear_Medicine>.
14. Maurina III, Edward F.. “The Game Programmer’s Guide to Torque”. Garage Games Inc. 2006.
15. Medalia, Jonathan. “Terrorist ‘Dirty Bombs’: A Brief Primer.” *CRS Report for Congress*. April 1, 2004.
16. PhysX Documentation <http://developer.nvidia.com/page/documentation.html>.
17. "PhysX." Wikipedia, The Free Encyclopedia. 3 Apr 2009, 04:48 UTC. 4 Apr 2009 <http://en.wikipedia.org/w/index.php?title=PhysX&oldid=281441680>.
18. Raghuvanshi, N., Lauterbach, C., Chandak, A., Manocha, D., and Lin, M. C. 2007. “Real-time sound synthesis and propagation for games”. *Commun. ACM* 50, 7 (Jul. 2007), 66-73. DOI= <http://doi.acm.org/10.1145/1272516.1272541>.
19. Scarvaci, Shannon. “PhysX in TGE Version 0.3” Torque Game Engine Resource. <http://www.garagegames.com/community/resources/view/10258>.
20. "Torque Game Engine." Wikipedia, The Free Encyclopedia. 2 Mar 2009, <http://en.wikipedia.org/w/index.php?title=Torque_Game_Engine&oldid=274548497>.
21. Wang, Jijun, Michael Lewis, and Jeffery Gennari. “A Game Based Simulation of the NIST Urban Search & Rescue Arenas.” *2003 Winter Simulation Conference*.